Battle for Bandwidth: On The Deployability of New Congestion Control Algorithms

Ranysha Ware

CMU-CS-24-135

August 2024

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

Thesis Committee:
Justine Sherry, Co-Chair
Srinivasan Seshan, Co-Chair
Jim Kurose (University of Massachusetts Amherst)
Theophilus A. Benson

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

This work is supported by the National Science Foundation under award numbers #2212390 and #2007733 as well as a CMU Cylab Seed Grant, a Google Faculty Research Award, and a Facebook Emerging Scholars Fellowship.

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: Computer Networks, Congestion Control, Internet Measurement

To my grandmothers. I miss you.

Abstract

The Internet has become the central source of information and communication in modern society. Congestion control algorithms (CCAs) are critical for the stability of the Internet: ensuring that users can fairly and efficiently share the network. Over the past 30 years, researchers and Internet content providers have proposed and deployed dozens of new CCAs designed to keep up with the growing demands of faster networks, diverse applications, and mobile users. Without tools to understand this growing heterogeneity in CCAs deployed on the Internet, the fairness of the Internet is at stake.

Towards understanding this growing heterogeneity, we develop CCAnalyzer, a tool to determine what CCA a particular web service deploys, outperforming previous classifiers in accuracy and efficiency. With CCAnalyzer, we show that new CCAs, both known and unknown, have widespread deployment on the Internet today, including a recently proposed CCA by Google: BBRv1. Next, we develop the first model of BBRv1, and prove BBRv1 can be very unfair to legacy loss-based CCAs, an alarming finding given the prolific deployment of BBRv1.

Consequently, we argue the need for a better methodology for determining if a new CCA is safe to deploy on the Internet today. We describe how the typical methodology testing for equal-rate fairness (every user gets the same bandwidth) is both an unachievable goal and ultimately, not the right threshold for determining if a new CCA is safe to deploy alongside others. Instead of equal-rate fairness, we propose a new metric we call, *harm*, and argue for a *harm*-based threshold.

Lastly, we present RayGen, a novel framework for evaluating interactions between heterogeneous CCAs. RayGen uses a genetic algorithm to efficiently explore the large state space of possible workloads and network settings when two CCAs compete. With a small budget of experiments, RayGen finds more harmful scenarios than a parameter sweep and random search.

Acknowledgments

Completing a dissertation is *really* hard and hard things are rarely done alone. These are only a few named acknowledgments, but that is not a measure of my gratitude, but rather my page limit and faulty memory. I would like to thank my family, advisors, collaborators, mentors, and friends who have made this work possible. My achievements are a reflection of your unwavering support.

I foremost want to thank my phenomenal co-advisors Justine Sherry and Srini Seshan. In the most cliché way, it is difficult to find the words to express the immense gratitude I have for your advocacy and guidance during the trials, tribulations, and triumphs of completing this work. Thank you for nurturing my ideas and for giving me the confidence to trust in my intuition. Thank you for showing me how to stay curious and never stop asking questions. Thank you for teaching me how to tell stories in my writing and talks. Thank you for encouraging me to push myself beyond what I thought I was capable of, knowing I could do it all along. Thank you to Srini for sharing your wealth of wisdom. And lastly, thank you to Justine for giving me the special privilege of being your first student.

I want to thank those who encouraged me and helped me get to CMU including my undergraduate mentor and advocate, Dr, Stacie Nunes, my mentors and recommendation letter writers from MIT Lincoln Lab, Benjamin A. Miller and Kevin Carter, and my professor during my MS at UMass Amherst, Jim Kurose.

I want to thank those who encouraged me and helped me stay at CMU. Thank you to all of my past and present lab mates in the SNAP lab for giving me the courage to ask and receive help, listening to a dozen practice talks, working on paper drafts, debugging code (thank you Christopher Canel), and laughing at my jokes. Thanks to the co-authors of the work presented in this dissertation for the many hours spent in the trenches of these paper submissions: Matthew Mukerjee, Adithya Abraham Philip, Isabel Suizo, Nicholas Hungria, and Yash Kothari. Thank you to the wonderful students who I had the joy of mentoring: Monica Pardeshi, Megan Yu, Joshua Slaughter, Anne Kohlbrenner, and Rukshani Athapathu. Thank you to the students, faculty, and researchers in the IMC, SIGCOMM, HotNets, and NSDI communities who embraced me and the work in this dissertation. There are too many to enumerate, but I am deeply grateful for the many CMU faculty, staff, and students that I've met along the way who did even the smallest things that kept me going.

I want to thank the friends who are more like family, for giving me a soft place to land, helping me build a life worth living, and keeping me sane in an insane world. Thank you to Katherine Kosaian for infinite positivity and blessings. Thank you to Sara McAllister for sharing my love of board games and making sure I rarely win. Thank you to Melva James, for encouraging me to pursue a PhD, reminding me research is fun, and looking at me over the top of your glasses. Thank you to my day-one Pittsburgh homies Sosa, Craig,

Natasha, Cas, and Danielle for endless laughter, dancing, and compassion.

Lastly, nothing I do is possible without the unconditional love and support I have from my family. A very special thank you to my parents, Mike and Kim, for teaching me many things, but especially that I could always come home and there was nothing I couldn't do. Thank you to my sister, Kim, my life-long best friend and my fiercest supporter. My love for my parents and my sister is greater than anything I could ever write.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I've been thinking a lot about this question of Internet fairness that you posed during your IC talk ... How should we define "fairness"?

Email to Justine Sherry titled 'Is The Internet Fair?' (August 2017)
Ranysha Ware

In 1988, the rapidly growing Internet was on the verge of collapsing. The saviors at the time were congestion control algorithms (CCAs) added to the reliable transport protocol: TCP Tahoe and later TCP Reno. These protocols used the additive increase multiplicative decrease algorithm (AIMD) to manage congestion [60]. AIMD offered much-needed stability to the Internet with provable guarantees: as long as every server used AIMD, everyone could fairly and efficiently share the Internet [30].

However, since the initial deployment of TCP Reno, the Internet has changed drastically. Bandwidth speeds have increased from Kbps to Gpbs. Internet users spend most of their time on tasks hardly imagined in the 1980s including scrolling social media, watching video streaming content, messaging and video conferencing, online shopping, and playing video games [97]. In addition, there are billions of active mobile Internet users. While AIMD and TCP Reno ensured efficiency and fair use of the Internet, as these changes to the Internet emerged, TCP Reno could no longer meet the requirements of faster networks, more diverse applications, and mobile users.

This desire for movement away from the homogenous deployment of "Standard TCP" (TCP Reno) sparked many concerns about the stability of the Internet. A decade after the deployment of TCP Reno, in 1999, Floyd and Fall's seminal paper [39] described concerns for possible congestion collapse:

> *The danger of congestion collapse from undelivered packets is due primarily to the increasing deployment of open-loop applications not using end-to-end congestion control.*

In this paper, Floyd and Fall advocate for enforcement at routers for flows that are not TCP-friendly: "A flow that is not 'TCP-friendly' is one whose long-term arrival rate exceeds that of any conformant TCP in the same circumstances."

In the same year, Legout and Biersac argued that TCP-friendliness required homogenous deployment and was too restrictive; emerging applications were going to continue to use non-TCP-friendly congestion control [71]:

> *Companies start to use non-TCP-friendly congestion control schemes, as they observe better performance for audio and video applications than with TCP-friendly schemes. However the benefit due to non-TCP-friendly schemes is a transitory effect and an increasing use of non-TCP-friendly schemes may lead to a congestion collapse in the Internet.*

Ultimately, the requirement for TCP-friendly congestion control became the gold standard to ensure new CCAs were not igniting a race to the bottom of deploying increasingly aggressive algorithms. It was cemented that equal-rate fairness was the goal: "if a non-TCP connection shares a bottleneck link with TCP connections, traveling over the same network path, then the non-TCP connection should receive the same share of bandwidth (i.e., achieve the same throughput) as a TCP connection" [88]. In RFC 5033 [42], Floyd and Allman describe the requirements for specifying a new congestion control within the IETF:

> *The minimum requirements for approval for widespread deployment in the global Internet include the following guidelines on: (1) assessing the impact on standard congestion control, (3) investigation of the proposed mechanism in a range of environments, (4) protection against congestion collapse, and (8) discussing whether the mechanism allows for incremental deployment.*

Consequently, CCAs proposed for deployment on the Internet included evaluations of their protocol's TCP-friendliness. In 2006, TCP Cubic became the default CCA in Linux, in part because the protocol developers carefully designed Cubic to be TCP-friendly. In a published paper [49], Cubic developers declare: "CUBIC tackles the shortcomings of BICTCP and achieves fairly good Intra-protocol fairness, RTTfairness and TCP-friendliness."

Even though there was widespread deployment of TCP Cubic, which resolved some issues with TCP Reno/NewReno, the Internet continued to change in ways that inspired further innovation. Between 2012 and 2015, Google first proposed and deployed QUIC [70], a replacement for TCP which implements reliable transport at the application layer running

atop UDP. This enabled far easier testing and deployment of new CCAs. QUIC has rapid adoption including open-source implementation in the Linux kernel [96, 82].

Around the same time, there were several innovations in congestion control schemes including machine-learning-based approaches [129, 33] and approaches focused on minimizing delay while maximizing throughput [130, 6, 24]. A notable algorithm proposed around this time in 2016, was Google's algorithm TCP BBR [24]. BBR developers proposed it as a replacement for Cubic, reducing queueing delays imposed by loss-based congestion control, while still achieving maximal utilization [25, 26, 79]. Like BBR, there is also an open-source implementation in the Linux kernel. Anyone could deploy BBR, and several content providers discussed doing just that including Akamai [9], Spotify [28], Dropbox [59], and Verizon CDN [105].

Both QUIC and TCP BBR are examples of the current state of congestion control on the Internet. With implementations by organizations outside of Google and open-source implementations in the Linux kernel, anyone can, and has deployed these protocols. In addition, content providers are privately tuning their transport protocols and networking stacks to squeeze out better and better network performance for their applications [99, 69]; some even deploying unknown congestion control algorithms [133, 83].

Consequently, we noticed two trends. First, there was still agreement to arguments from 1999: new CCAs should at least try to share reasonably with already wide-deployed CCAs to prevent congestion collapse. CCA designers all attempted to show that their new proposed CCA was "fair" and could reasonably share with Cubic [33, 34, 6, 24]. However, the second thing we noticed was widespread disagreement on the methodology for showing that a new CCA does share reasonably. The experiments, metrics, and thresholds used to claim "reasonable deployability" were inconsistent. This state of affairs motivated the three over-arching questions we ask in this dissertation:

What CCAs are widely deployed on the Internet today? While several large content providers have *discussed* deploying new CCAs, and there is some research showing increased deployment of new protocols, known and unknown [133, 83, 46], there is an ever-growing need to understand how the congestion control landscape on the Internet has evolved (and will continue to evolve).

How does BBR interact with loss-based CCAs? In their initial white paper, Cardwell et al. boldly claim BBR[1] is TCP-friendly: "BBR converges toward a fair share of the bottleneck bandwidth whether competing with other BBR flows or with loss-based congestion control" [24]. However, even their presentations showed this blanket statement was not true in networks with small queues [25]. In this dissertation, we investigate these

How should we evaluate inter-CCA interactions to decide if a CCA is deployable?

[1]While Google developed BBRv2 and then BBRv3, when we mention BBR we mean BBRv1. At the time of this work, BBRv1 is still the BBR version deployed in Linux

Despite BBR's TCP-unfriendliness, it was deployed. Should we re-think our definitions of fairness? Can we come up with a consistent methodology for evaluating new algorithms for deployability?

This thesis focuses on answering these questions. We aim to develop methodologies and tools to evaluate if a new CCA is deployable on the Internet today. This dissertation shows the following:

Thesis statement: *Given the growing diversity in novel congestion control algorithms (CCAs) deployed on the Internet today, we argue that the deployability of new CCAs must be evaluated for how they harm widely deployed CCAs in realistic network settings.*

To support this thesis, we first measure the current deployment of congestion control algorithms in Chapter 2, then prove when and why BBR, a new CCA, can be very unfair to widely deployed loss-based CCAs in Chapter 3, and lastly propose a new metric in Chapter 4 and tool for evaluating interactions between heterogeneous congestion control algorithms in Chapter 5.

1.1 Overview of Contributions

We support this thesis statement through our insights and contributions which we describe in this section.

1.1.1 Chapter 2: CCAnalyzer

In our thesis statement, we claim there is a growing diversity in new CCAs based on an open-source implementation of BBR in the Linux kernel and the discussions about deployment at large content providers. To quantify this diversity, we seek to develop a congestion control classifier that addresses limitations of prior work, that the networking community can use to give a comprehensive survey of what CCAs are deployed on the Internet today.

Limitations of Prior Work: Prior work relies on brittle active measurements techniques forcing packet drops and timeouts, attempting to generate `CWND` traces to classify websites [133, 83, 46]. We find these tools have significant limitations including limited coverage of all known CCAs in Linux, inefficiency, and an inability to discover unknown CCAs without considerable effort.

Key Insights & Approach: We develop CCAnalyzer, a congestion control classifier that outperforms prior work both in accuracy and efficiency. To overcome the limitations of prior work, we show that we can create a local bottleneck between a server we want to classify and a receiver we control. Then we can passively observe CCAs in their natural habitat: at the bottleneck queue. Turns out, each CCA has a unique queue occupancy trace. Given queue occupancy traces, we frame the problem of CCA classification as a time series classification problem and use Dynamic Time Warping (DTW) [128] to determine if two traces are from the same CCA.

Takeaways: Using CCAnalyzer, we classify Top 10K websites from Google Chrome's UX Report and find there is indeed heterogeneity within congestion control deployment. CCAnalyzer finds widespread deployment of BBRv1 at massive content providers including Akamai, Cloudfront, Cloudflare, and Cubic at Fastly. CCAnalyzer is even able to discover the deployment of BBRv3 at Google without BBRv3 samples in its training set. BBRv3 deployment is another significant shift in the Internet's congestion control landscape.

Broader Impact: This work was very recently published at SIGCOMM 2024 [127]. While we have yet to see what its broader impact will be, when we shared our results with Google's BBR developers, they were surprised that BBRv1 had such widespread deployment. One nice feature of CCAnalyzer is it treats CCAs as black boxes: to extend to classify new and known CCAs (like BBRv3), we only need to add training samples. CCAnalyzer will be a useful tool to continue to monitor the deployment of versions of BBR as well as other new CCAs. We also hope this work inspires more innovation in congestion control classification.

1.1.2 Chapter 3: BBRv1 Model

With CCAnalyzer, we measure a prolific deployment of BBRv1 beyond Google. As described in RFC 5033 [42], it is a minimal requirement to evaluate a new CCA's interactions with already widely deployed CCAs in a range of environments. This begs the question: How does BBRv1 interact with widely deployed loss-based CCAs?

Limitations of Prior Work: BBR's initial white paper and presentations made blanket statements about BBR's fairness to Cubic based on 1 BBR flow and 1 Cubic flow in a few settings [25, 24]. BBR developer's claims about fairness to TCP Cubic did not meet the bar for "investigation of the proposed mechanism in a range of environments", inspiring further study by networking researchers. Several empirical studies found after BBR's deployment into the Linux kernel, it was actually *extremely unfair* in certain common scenarios. In one example paper, Scholz et al. found: "In fact, independent of the number of BBR and CUBIC flows, BBR flows are always able to claim at least 35% of the total bandwidth." However, missing from conflicting empirical results was a clear explanation.

Key Insights & Approach: We first reproduce prior empirical measurements and observe a single BBR flow consuming 40% of link capacity when competing with as many as 16 Cubic or Reno flows. After a bit of digging, we discovered why: although BBR is a rate-based algorithm, under competition with loss-based congestion control it becomes window-limited by an "in-flight cap", a seemingly Innocuous parameter which happened to be set to an arbitrary value. Therefore, if we could model the in-flight cap, then we could model BBR's interactions with loss-based CCAs. We present a simple model of BBR's throughput fraction when competing with one loss-based flow, and a more complex model that relaxes that assumption.

Results: We develop the first model proving BBR's fixed in-flight cap determines its

bandwidth consumption when competing with loss-based flows. Because BBR does not directly respond to packet loss, instead of backing off due to congestion it actually *increases its sending rate* during its `ProbeBW` state when competing with Cubic or Reno under certain conditions. Our model shows all the variables that impact BBR's link fraction under competition and none are dependent on the number of competing loss-based flows, explaining empirical results about BBR's constant link fraction.

Broader Impact: This work was published at IMC 2019 [126]. Of the work presented in this dissertation, our findings about BBR received the most news coverage [116, 121, 14]. In addition, it is often cited in papers referencing BBR's unfairness, inspiring additional models of BBR's interactions with Cubic [84]. Since our finding, Google has proposed and deployed BBRv2 and BBRv3 which increased the complexity of `ProbeBW` to account for issues with the fixed in-flight cap and not backing off in response to packet losses [22, 23, 27]. Google developers are working to replace the implementation of BBRv1 in the Linux kernel with BBRv3, recently submitting an Internet Draft to the IETF describing the algorithm in detail [11]. Hopefully this time around, they spend more time evaluating BBRv3 before Linux kernel deployment, though recent results suggest they may have yet again failed to deliver on their promises [138] (We revisit some of these BBRv3 findings in Chapter 5).

1.1.3 Chapter 4: Harm

In our thesis statement, we argue for the need to change the metric and threshold used to evaluate interactions between new CCAs and widely deployed CCAs to determine if a new CCA is safe to deploy.

Limitations of Prior Work: Over the past 30 years, the methodology for evaluating the deployability of a new CCA on the Internet, has essentially been a manual and arbitrary process: pick some network settings, some duration of experiments to emulate long-running flows, run competing flows and check how far away the bandwidth allocation is to perfect fair sharing. These evaluations all have the same goal: show that the new CCAs is not *too unfair* to widely deployed CCAs (usually Cubic). The threshold for "too unfair" was typically equal-rate bandwidth sharing which was consistently unattainable with CCA developers making various, inconsistent excuses about why being unfair was ok [33, 6, 35, 24]. We argue that classical definitions of fairness are neither an achievable goal nor the right one. A fairness-based threshold suffers from three key issues: ideal-driven goal-posts, throughput-centricity, and assumption of balance.

Key Insights & Approach: In contrast to fairness and TCP-friendliness, a new metric we call *harm*, meaningfully quantifies the impact of deploying a new CCA α on already widely deployed CCAs β. Harm measures the throughput achieved by β when β is running solo in a network with a certain workload, and when β is under competition with α. We propose a harm-based threshold for the deployability of a new CCA: *If the amount of harm caused*

by flows using a new algorithm α on flows using an algorithm β is within a bound derived from how much harm β flows cause other β flows, we can consider α deployable alongside β. A harm-based threshold is demand-aware, practical, status-quo-biased, and future-proof.

Broader Impact: Perhaps the most impactful work from this thesis is our harm proposal published at HotNets 2019 [125]. It is often cited in discussions about the current state of Internet congestion control and the deployability of new schemes (ex: [19, 134, 18]). Our harm proposal has even made it into the textbook *TCP Congestion Control: A Systems Approach* [91] by the authors of one of the most prolific networking textbooks *Computer networks: A Systems Approach* [92].

This paper won the IRTF's Applied Networking Research Prize and was presented at IETF-109 with a positive reception and lively discussion IETF-109 was moved online due to the COVID-19 pandemic. It was originally supposed to be in Bangkok, Thailand! Interestingly over the past year, the IETF's congestion control working group (CCWG) has been writing a draft to update RFC 5033 [109]. The latest version from July 2024, explicitly states CCA proposals to the IETF must evaluate interactions between "the proposed congestion control algorithm and commonly deployed algorithms." The draft says, "In contexts where differing congestion control algorithms are used, it is important to understand whether the proposed congestion control algorithm could result in more harm than previous standards-track algorithms." Although it is not cited, this is nearly verbatim our proposal in Chapter 4 [125].

1.1.4 Chapter 5: RayGen

We have both measured and modeled interactions between a new CCA and legacy CCAs in our analysis of BBRv1 vs. Cubic/Reno. We see that in certain conditions, CCAs are unfair to one another, and not so much in others. It is critical to find these cases where new CCAs significantly harm widely deployed CCAs. The key question then is: how do we find the conditions under which unfairness may occur?

Limitations of Prior Work: The current methodology for finding these conditions is typically manual: pick a network setting (*e.g.* bottleneck bandwidth, RTT, queue size), pick some number of flows, and measure throughput achieved by CCA α and CCA β when they compete[104, 138, 55, 119, 34, 25] looking for how far away the results are from perfect fair sharing. There are several limitations of this process. First, as we discussed in the previous chapter, equal rate fairness is not an achievable goal, the duration of experiments may impact results and there are wide-variety of realistic network settings to test.

Key Insights & Approach: We propose doing this search for worst-case settings in an automated way to *automatically generate network settings where poor interactions between CCAs may occur.* Toward this goal, we frame finding worst-case outcomes as an optimization problem. We present RayGen a novel framework for evaluating interactions between CCAs. We discuss how we can measure the *relative harm* when flows between CCAs

interact in a testbed. RayGen uses a genetic algorithm to search a large state space of network scenarios and over generations find more and more scenarios with high harm.

Results: Using RayGen we can find high-harm scenarios with a small budget of only 300 experiments compared to random search with the same budget. Surprisingly, we can even find more high-harm scenarios than a parameter sweep of 3500 experiments over the same parameter space.

Broader Impact: We will soon refine and submit Chapter 5 to a conference, and we plan to share RayGen to help the networking community determine if a new CCA is safe to deploy on the Internet. The recent RFC draft 5033 on specifying new congestion control algorithms says: "An evaluation MUST assess the potential to cause starvation" and RayGen can help with this evaluation. RayGen helps CCA developers search a wide state space of possible network settings and workloads to find worst-case high-harm scenarios.

Chapter 2

CCAnalyzer: An Efficent and Nearly-Passive Congestion Control Classifier

> *The ability to understand what CCAs are used on the Internet is useful, and the work clearly demonstrates advantages over prior work. The proposed mechanism is simple, efficient, and accurate.*
>
> Reviewer #1

2.1 Introduction

With the growing diversity in CCA proposals and potential deployments, we have an ever-growing need to understand what CCAs are currently deployed on the Internet today. Assumptions about what CCAs are widely deployed underlie decisions about how to size buffers in routers [44] (proportional to $\frac{1}{\sqrt{n}}$, *if* everyone is deploying NewReno [53]); whether or not routers need multiple queues [17] (to protect low-latency traffic from buffer filling traffic, *if* both classes of CCAs are deployed); and how to test new Internet services to ensure that they do not starve legacy traffic [125, 126, 49] (*if* Reno is no longer widely used, perhaps we do not need to test new CCAs for Reno-friendliness).

This leads to the first question of this thesis: **What congestion control algorithms are widely deployed on the Internet today?** The desire to understand CCA deployment motivated the development of *CCA classifiers* starting with TBIT in 2001 [89, 101, 133, 46, 83]. Most of these tools try to estimate the CCA's congestion window (`cwnd`) by requesting a bulk data transfer from the server and then observing the transfer's reaction to dropping and delaying packet acknowledgments or to modulating the available bandwidth. Unfortunately, state-of-the-art CCA classifiers using these techniques, e.g., Gordon [83] and Inspector Gadget [46], have several limitations preventing them from providing a truly comprehensive picture of CCA deployments. We discuss prior approaches and their limitations in detail in §2.2.

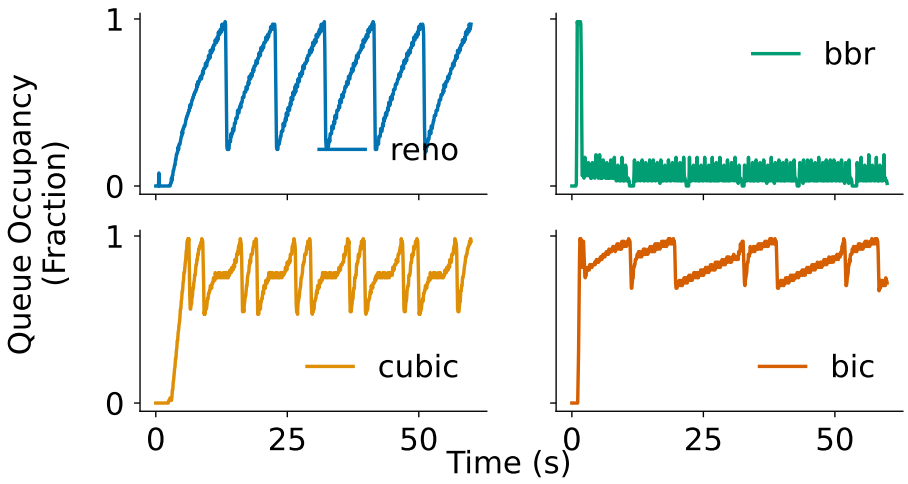


Figure 2.1: Time series of queue occupancy for four CCAs (from top, left to right: New Reno, BBR, Cubic, and BIC). Each CCA has a visually distinct queue occupancy behavior.

We seek to develop a CCA classifier with several desirable properties:

Support for all well-known CCAs: A CCA classifier should be able to identify known CCAs with minimal errors. Supporting identification of the 15 built-in wide area CCAs in Linux[1] is especially desirable.

Efficient and nearly-passive: Network measurements should aim to be as lightweight and minimally burdensome as possible on non-cooperating parties. Heavyweight techniques make it difficult to perform large-scale measurement studies and can lead to measurement tools being 'blocklisted' by services.

Discover new CCAs: Open-set classification is the ability for a classifier to classify that a testing sample is not in the training set [81]. In the current period of significant experimentation in the congestion control space, a CCA classifier should be able to identify if a website is using a known or unknown CCA. Furthermore, to identify truly novel CCAs, the classifier should be able to determine which servers using unknown CCAs all appear to be using the *same* algorithm.

Interpretable results: A CCA classifier should be 'interpretable' [75]. That is, as human experts, we should be able to understand why our algorithm classifies two web servers as using the same CCA. This allows for evaluation and validation of results as well as aiding in the discovery of new CCAs.

In this chapter, we present CCAnalyzer, a new CCA classifier. CCAnalyzer can correctly classify *all* built-in Linux CCAs. It is 40x faster than Gordon, and unlike Inspector

[1]In fairness, we exclude the lp and dctcp algorithms because these algorithms require in-network support which is not available in the wide area. All other prior work also excludes these algorithms.

Gadget, CCAnalyzer can efficiently identify if a group of servers are all using the same unknown algorithm. CCAnalyzer achieves this by taking a radically different approach to classification than prior work. Both Gordon and Inspector Gadget use decision trees hand-crafted or trained on observed `cwnd` values or gradients; they inflate round-trip-times (RTTs) and/or introduce timeouts to precisely measure the `cwnd` at each point in time. In contrast, CCAnalyzer starts from a simple observation: if we visually observe the occupancy of packets in a bottleneck queue over time, even a human expert can identify the connection's CCA. In Figure 2.1, we present the queue occupancy of the bottleneck link from real TCP connections; the familiar Reno 'sawtooth' is visible for Reno while other CCAs have their patterns of rising and falling queue size. Because CCAnalyzer does not interfere with a connection's normal behavior (beyond introducing a low-capacity link to force a bottleneck) we describe the approach as *nearly-passive* and argue that it is minimally intrusive for operators.

Rather than trying to collect `cwnd` traces, CCAnalyzer works by measuring a connection's queue occupancy over time and uses this time series data as input to a classic algorithm for measuring the distance between two time series called Dynamic Time Warping (DTW) [128]. DTW is used in a variety of applications requiring signal comparison, such as voice recognition and shape detection. DTW compares two signals for similarities in shape and magnitude while accounting for distortions such as stretching or noise – this latter accounting is especially valuable since we expect to see such distortions in network traces due to variances in RTT, jitter, random packet loss, *etc.* CCAnalyzer uses a 1-Nearest Neighbor(1NN) classifier with DTW as the distance measure and labeled time-series as the training set. A testing trace is given the label as the closest training sample. CCAnalyzer collects 4 queue occupancy traces for each website, and votes across the labels of those traces to give a website a final label. We describe our methodology in more detail in §2.3.

We find that, in addition to being more *efficient* and *broadly applicable* than prior approaches, CCAnalyzer offers additional advantages. Collecting queue occupancy traces as well as the ability to compare these traces to one another using the 'distance' measure provided by DTW allows us to visualize and validate results. By looking at the website traces and their closest training sample we can see when and why the classification may have been incorrect for identifying possible errors. In addition, using a matrix of all the pairwise distances between a set of traces, we can cluster traces and identify the deployment of new CCAs outside of our training set. We demonstrate these additional advantages in §2.4 and §2.5.

We use CCAnalyzer to conduct a measurement study of Top 10K websites ranked by Google Chrome's UX Report (CrUX) [135] and find the following:

1. Inspector Gadget can only classify 1% of these 10K websites.
2. We find several major CDNs have deployed BBRv1 (Cloudflare, Akamai), while others still use Cubic (Fastly).

Table 2.1: CCA classifier desirable properties

Classifier	Accurate	Unobtrusive	Open-set	Interpretable
Gordon [83]	✗	✗	✓	✗
IG [46]	✗	✗	✗	✗
CCAnalyzer	✓	✓	✓	✓

3. Clustering queue occupancy traces makes our results interpretable and straightforward to validate. It allows us to fix when a website's traces are marked as unknown when they are actually known and using a CCA in the training set.

4. CCAnalyzer was able to discover Google's deployment of BBRv3, even though we do not have a BBRv3 implementation in our testbed and did not train CCAnalyzer on BBRv3 traffic.

5. We see some deployment of other unknowns CCAs.

The rest of this chapter is organized as follows. In §2.2 we discuss prior work in classifying CCAs. In §2.3, we present the CCAnalyzer methodology. In §2.4 we evaluate CCAnalyzer's accuracy, speed, and resource utilization. In §2.5 we provide a brief measurement study focusing on (a) a 2023 update on CCAs used by web servers and (b) the results of clustering unknown CCAs. In §2.6 we conclude and highlight future work.

2.2 Prior Work and Limitations

There have been several attempts at CCA classification over the past two decades beginning with TBIT [133, 101, 89, 83, 46]. Of recent classifiers, we focus on the two state-of-the-art algorithms: Gordon [83] (2019) and Inspector Gadget[46] (2020). Table 2.1 highlights the limitations of these classifiers.

Gordon: Gordon inspired a renaissance in CCA classification algorithms after two decades of relative dormancy. The authors insightfully noted the deployment of numerous novel algorithms (at the time, BBRv1 was beginning to 'take off' [83]) and the need to measure the changing CCA landscape due to the impact of CCAs on a wide range of Internet issues from infrastructure design to network fairness. In addition to developing the Gordon classification tool, the paper also provides the widest measurement study of CCA deployment in the post-BBR era; significantly, the authors noted the surprisingly rapid growth in the deployment of BBRv1, which 17.75% of servers they measured used at the time.

The Gordon classifier works by creating a bottleneck between the web server and the client, introducing various network events including packet losses and changes in bandwidth and delay in the hopes of exactly measuring the cwnd. Generating these cwnd traces comes at a high cost: Gordon requires incremental probing, RTT-by-RTT, starting and restarting connections with a web server many times—requiring up to 800MB of data transferred to successfully perform a classification. In addition, we observe in our own evaluation that more servers reject connections from the Gordon tool [1] than reported in 2019; conversations with one of the Gordon authors lead to the hypothesis that Gordon is being blocked or rate-limited due to these overheads. In 2023, Gordon authors were only able to classify 4% of Alexa Top 10K. As we will show in §2.4.2, CCAnalyzer trace collection transfers 85% fewer bytes, and is 40x faster than Gordon. CCAnalyzer's passivity avoids the pitfalls Gordon has with onerous active cwnd estimation.

After collecting cwnd traces, Gordon, uses a hard-coded decision tree to classify these traces. Because some algorithms are not distinguishable based on the parameters in this decision tree, Gordon cannot tell the difference between Compound TCP/Illinois, Vegas/Veno, and New Reno/Highspeed (HSTCP) and instead groups these into the same category although all of these algorithms are distinct.

Consequently, Gordon requires detailed knowledge about how each CCA works to support a new CCA. For example, it needed a special-cased test to support BBR. While Gordon can mark a cwnd trace as 'unknown', Gordon cannot group web servers as using the same unknown CCA without running several additional hand-crafted tests putting even more additional load on web servers. In addition, we will show in §2.4.1, although Gordon has good accuracy for supported CCAs, its lack of support for many CCAs, requirements for special tests for new CCAs, inefficiency and inability to natively discover new novel CCAs makes it challenging to use with a constantly evolving transport layer.

Inspector Gadget (IG): Published in 2020, IG's authors developed the tool to fingerprint a web server's networking stacks, including its CCA. In their results, they notably found that Cubic was the dominant CCA followed by BBR in North America, but also saw most servers from other regions were still using Reno. Similarly to Gordon, IG also tries to carefully inject network events including timeouts and changes in delay to generate cwnd traces. To generate these traces, IG addresses issues with prior work's cwnd estimations with some optimizations. Rather than classifying raw cwnd traces, IG extracts a vector capturing the cwnd as a series of offsets, using a decision tree classifier on these vectors.

IG's published code [57] includes a user-level TCP stack and modifications to a TLS library to manipulate packets in a HTTPS connection, which we find does not work in practice. We ultimately had to re-implement IG to the best of our ability. As we will show in §2.4.1 we obtain reasonably good accuracy with our re-implementation. We find this technique is more efficient than Gordon. However, we highlight three limitations of IG.

First, we find that IG does not make it straightforward to classify a CCA as unknown

Table 2.2: IG measurement results for 10K websites

Result	Count
BBR	71
Cubic	25
Yeah	10
Highspeed	5
BIC	2
Not large enough object	9533
Trace collection fail	354
Total	10000

or discover new CCAs. Given the decision tree classifier, we can only mark a trace as a known label. Second, it takes considerable effort to re-implement; we find that we need to carefully account for TCP stack optimizations at the sender like F-RTO [67] that impact how a TCP flow will respond to losses that are independent of CCA behavior. These special cases are also challenges in prior work that try to collect `cwnd` traces [133].

Lastly, when we try to use our re-implementation of IG to classify the 10K websites in our measurement study, we find that we can only successfully classify 1% of these websites. The labels for these websites are shown in Table 2.2 if we restrict the training set to the the CCAs in the IG paper. We are able to use the IG web crawler code to try and find large enough objects on the 10K websites we want to classify. From our controlled measurements (§2.4) we find that we need a file size of at least 1.5 MB. However, for 95% of the websites, we could not not find files that large despite crawling up to 500 links per website. Ultimately, we are only able to successfully classify 113 websites. IG `cwnd` estimation technique generally fails in practice when attempting to classify real websites.

We are able to use the IG web crawler code to try and find large enough objects on the 10K websites we want to classify. From our controlled measurements (§2.4) we find that we need a file size of at least 1.5 MB. However, for 95% websites we could not not find files that large despite crawling up to 500 links per website. Ultimately, we are only able to successfully classify 113 websites. The labels for these websites are shown in Table 2.2 if we restrict the training set to the the CCAs in the IG paper.

Furthermore, because of IG and Gordon's significant active manipulation of ACK timings and packet drops, their extensibility to other protocols with encryption (e.g. QUIC) or applications (e.g. video) is severely limited relative to a more passive measurement approach.

Other classifiers: The literature prior to Gordon and IG includes other influential classifiers such as TBIT [89] and CAAI [133], however, all of these approaches are superseded

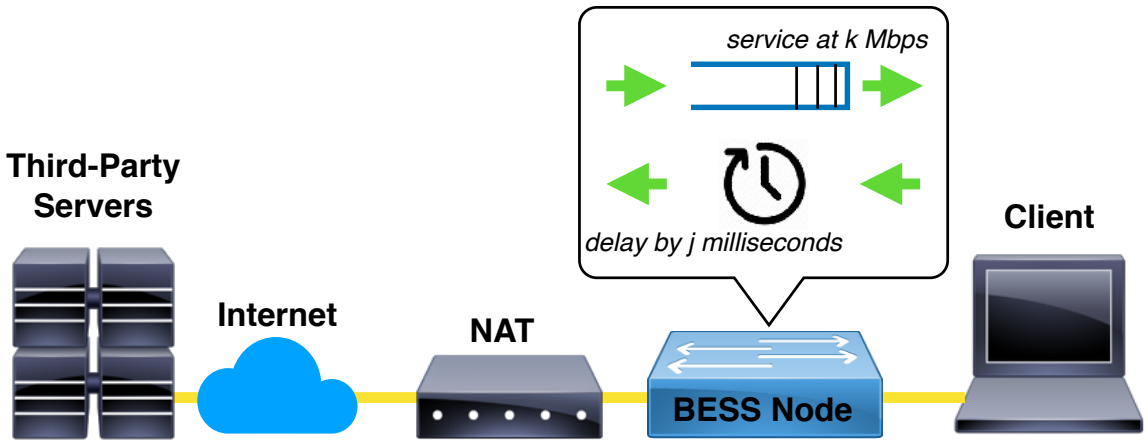


Figure 2.2: Testbed to issue requests to third-party servers and identify their CCAs.

in both accuracy and coverage by Gordon and IG, therefore we focus our comparisons on these to prior approaches only. Other techniques that attempt to classify the CCA of a flow as it crosses a router (rather than classifying a server) such as DeePCCI [101] and DragonFly [29], are solving an orthogonal problem that is out of scope for this work. Given the limitations of prior work our goal is the following: **We want to design a new CCA classifier with higher coverage of known CCAs, better efficiency, better passivity, and open set: able to discover new CCAs without considerable effort.** In the following sections we discuss how CCAnalyzer achieves these goals.

2.3 Methodology

We propose a new algorithm CCAnalyzer for identifying CCAs in an efficient and nearly-passive way. CCAnalyzer takes a radically different approach to prior cwnd estimation techniques by relying on bottleneck queue occupancy traces. In this section, we describe how we can frame the CCA classification problem as a time series classification problem and how this enables CCAnalyzer to achieve the goals outlined in previous sections.

2.3.1 Observing Queue Occupancy

A key issue with prior techniques is that they require brittle and resource-intensive flow manipulation to *estimate* the cwnd, which is not directly observable, and then perform classification. Our key insight is that we need not try to force network events *e.g.*timeouts to force a CCA to behave in some expected way, but rather we can observe CCAs in their natural habitat: at the bottleneck queue.

In order to observe the bottleneck queue occupancy when downloading data from a server, we insert our own switch with a deliberately slowed egress link between the server and the client using a testbed as shown in Fig. 2.2. Because the switch processes incoming packets at a speed much slower than upstream links, it becomes the connection

bottleneck. The switch uses a queue of a chosen size and we configure it to record when packets are enqueued, dequeued, or dropped. We implement this switch using the BESS software switch [13], and the client issues pipelined HTTP requests to third party servers using h2load [48] to utilize the available bandwidth.

On page 1, Fig. 2.1 shows *real* example bottleneck queue occupancy traces collected from our testbed. A human observer can clearly see the classic 'TCP sawtooth' of Reno, x^3 curves of Cubic and even periodic bandwidth probes of BBR in these traces. CCAs will cycle through their behavior: increasing their sending rates to use the available bandwidth and react to losses (depending on their design) that occur naturally if they fill the bottleneck queue. We posit that if the patterns observed by two different flows in the bottleneck queue are equivalent, then the CCAs are equivalent.

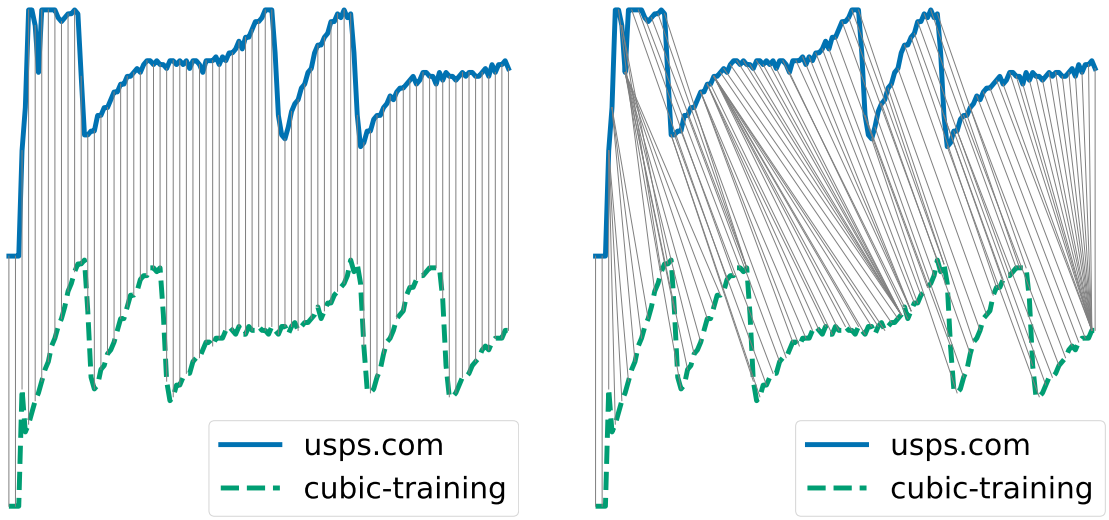
CCAnalyzer's simple inference from queue occupancy traces achieves the goals outlined in the previous section. CCAnalyzer has higher coverage of known CCAs and is more general than prior work. We can support classifying a CCA, if we can collect queue occupancy traces for that CCA. CCAs may be loss-based, may be latency sensitive, or have other characteristics and CCAnalyzer can still classify them without needing any special tests.

CCAnalyzer is nearly-passive: it does not need to force timeouts, radically modulate bandwidth, implement numerous serial connections, *etc.*. Although CCAnalyzer does normalize round-trip times and bottleneck bandwidth, to the server under test it appears as a normal TCP connection with no anomalous behaviors.

Lastly, CCAnalyzer is also open-set. Because we can compare queue occupancy traces, we can determine if a trace does not match anything in the training set. Further, we can cluster like traces and detect if multiple servers are deploying the same CCA that is not in the training set. No prior tool can *automatically* cluster servers using like, novel CCAs and we believe that this trait of CCAnalyzer is crucial to measuring and modeling a continuously-evolving Internet. While some prior work also creates a local bottleneck (*e.g.* Gordon [83]), or may try to *estimate* queue occupancy for a particular flow crossing a router (*e.g.* DragonFly [29]), our work is the first to *directly measure* bottleneck queue occupancy by creating a local bottleneck and recording every time a packet is enqueued, dequeued, and dropped from that bottleneck queue to use this trace to classify CCAs.

2.3.2 A Time Series Classification Approach

CCAnalyzer compares two queue occupancy traces to each other using a well-known algorithm called Dynamic Time Warping (DTW) [128], which takes in two time series traces and returns a 'distance' measurement quantifying how similar the two traces are. DTW is traditionally used in pattern matching tasks like automatic speech recognition and speaker identification; just as a speaker will have a signature pitch and cadence, congestion control algorithms each have a unique typical queue occupancy and rate of change. These types of problems are known as 'time series classification' problems, and despite 40 years of



(a) Euclidean distance = 2.47

(b) DTW distance = 0.76

Figure 2.3: Queue occupancy distance calculation for a sample from usps.com to a Cubic training sample. DTW allows a flexible one-to-many mapping between similar points, while euclidean is a one-to-one mapping to points at the exact same time.

research since the invention of DTW, it remains a widely used general-purpose algorithm for this class of challenges [7].

To understand DTW, we first consider a naïve approach to compare two traces using Euclidean distance (ED). Consider two queue occupancy traces, $X = (x_1 \ldots x_n)$ and $Y = (y_1 \ldots y_m)$, where x_i is the queue occupancy at time i in trace X, where X is n time steps long. A simple approach to measuring the difference between the two traces is to to calculate the Euclidean distance (ED) (assuming X and Y are the same length $n = m$):

$$ED(A, B) = \sqrt{\sum_{i=0}^{n} (X[i] - Y[i])^2}$$

Fig. 2.3 shows why this one-to-one mapping approach fails for most network traces. In Fig. 2.3a, we compute the ED between a trace collected from `usps.com` to a Cubic training sample, while in Fig. 2.3b we take the same traces and compute the DTW distance. Traces can dilate and contract relative to time on the real Internet. For example: a host may stall during the trace, sending a packet a few ms later than expected; an in-network queue may fill up with background traffic, temporarily increasing the RTT; a long-running flow in the background may end, suddenly reducing the RTT. These effects can cause two traces from the same CCA to appear stretched and squeezed relative to one another.

Unlike ED, DTW allows a one-to-many mapping between X and Y: a given index from each trace can map to one or more indices in the other trace. DTW finds the best point-to-point mapping between two traces to minimize the sum of distances between all their points with two constraints: 1) The first and last indices must be mapped to one another and 2) the mappings must be monotonically increasing. Fig. 2.3b shows how this results in DTW measuring a smaller distance than ED for same-CCA traces. DTW finds the optimal "warp path", the one-to-many mapping between points in an NxM matrix where N and M are the lengths of two time series, that minimizes the overall distance between the time series. Exact mappings would be a diagonal line, but DTW accounts for phase shifts with horizontal and diagonal lines which show when many points in one trace are mapping to the same point in the other trace. For DTW the time series need not be the same length, although in this work we truncate all the traces to be the same size.

More formally, let $DTW(i, j)$ be the optimal distance between the first i and j elements in time series X and Y. Then, the value of $DTW(i, j)$ is defined recursively as follows:

$$DTW(i,j) = distance(x_i, y_i) + \min \begin{cases} DTW(i, j-1) & \text{repeat } x_i \\ DTW(i-1, j) & \text{repeat } y_i \\ DTW(i-1, j-1) & \text{repeat neither} \end{cases}$$

where $distance(x_i, y_i)$ may be defined in different ways including the squared difference which we use in Fig. 2.3; in the rest of this work we find the absolute difference works better for our use case $|x_i - y_i|$. There are many more well-studied aspects and applications of DTW [128, 63, 66, 100, 94, 7], but we do not require their discussion here to understand CCAnalyzer.

CCAnalyzer uses a one-nearest-neighbor classifier with DTW as the distance measure (1NN-DTW), a commonly used time series classification methodology [7]. Given a website to classify, CCAnalyzer computes the DTW distances between the queue occupancy traces of all training samples and the queue occupancy trace of the website. The website is given the label of the closest training sample.

Given this approach, DTW allows us to classify if a time series matches one within the training set, but how will we determine if a CCA is not in the training set and should be classified as unknown? We explore using a well-known extension to our 1NN classifier called TNN where T is a distance threshold [81]. If the DTW distance between a website trace and its closest label is higher than T, then the trace is marked as unknown.

2.3.3 Parameter Tuning

CCAnalyzer observes a TCP connection's natural behavior as its `cwnd` rises and falls, probing for bandwidth. However, classifying CCAs based on this natural behavior requires that we observe TCP connections in sufficient conditions that they *act distinguishably* from one another. To be specific:

We must choose bottleneck bandwidth, RTT, and queue size such that same-family CCAs exhibit different behavior (§2.3.3.2): This is most important for Reno-family CCAs (Westwood, Highspeed, YeAH, etc.) which are all variants of each other. Many are designed to simply 'act like Reno' in low BDP environments and only exhibit their unique growth and backoff behaviors at higher BDP environments.

We must observe connections for long enough that each CCA goes through several 'cycles' of operation (§2.3.3.3): DTW matches similar traces to each other, but minor perturbations in the network environment (arrival/departure of background flows, external packet loss) can make traces appear dissimilarly. Having multiple iterations of the CCA's characteristic behavior allows DTW to self-correct for brief abberations as the characteristic connection behavior re-emerges after a few RTTs.

We need to identify when a trace is too far from its nearest neighbors in the training set (§2.3.3.4): We would expect servers using novel CCAs to produce a DTW distance which is 'far' from any training sample: but how far is far enough to declare that a server is indeed using a new algorithm?

Note that the above issues all somewhat depend on the set of CCAs that the system is meant to classify. We take an empirical approach to setting appropriate parameters to correctly distinguish CCAs which we describe in the following sections. However, it is not unlikely that if the CCA landscape were to evolve dramatically with the deployment of many new CCAs and the phasing out of many old ones, that we would need to re-tune these parameters for CCAnalyzer to remain effective in the future.

2.3.3.1 Experimental Setup

The CCAnalyzer testbed is installed on Cloudlab servers in Wisconsin, USA [36] (see Fig 2.2). To generate ground truth data for evaluation, we collect traces to servers installed on Amazon Web Services (AWS) datacenters in Virginia and Microsoft Azure's 'East' US datacenter. We use the AWS-Virginia dataset as our *training* data for CCAnalyzer and our Azure-East datasets for *testing*. When measured using `iperf`[58], the total available bandwidth between the CCAnalyzer testbed client and the AWS machines is 500Mbps and between the testbed client and Azure machines is 920Mbps.

Each server is configured as follows:

- Training Set (**AWS-Virginia**): Ubuntu 22.04.2, Linux kernel version 5.19. RTT to testbed 22ms. 3 samples per CCA.
- Testing Set (**Azure-East**): Ubuntu 20.04.6, Linux kernel version 5.15. RTT to testbed 24ms. 5 samples per CCA.

Training and Testing for CCAnalyzer: Using AWS-Virginia we generate training sam-

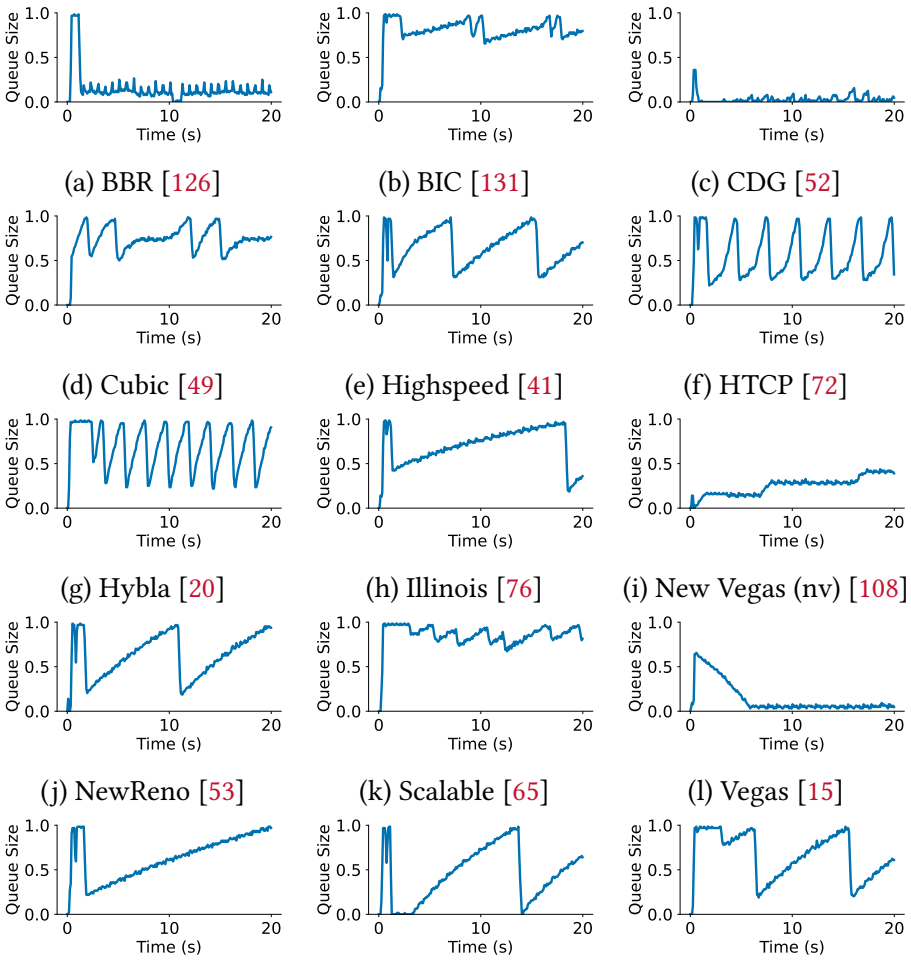


Figure 2.4: Example CCAnalyzer CCA training sample traces from AWS-Virginia (5bw-85rtt-64q setting)

ples for 15 CCAs available in Linux.[2] We run `iperf` flows between a transmitting host located in AWS Virginia and a receiving host in our testbed for 120s (as we will discuss in §2.4.2 we need not use all 120s for accurate training and only need 20s). To generate testing data, we set up an Apache web server on Azure-East with a 100 MB file. We use `wget` to download the file to the receiving host in our testbed for 60s.

We highlight some example training samples for the 5bw-85rtt-64q setting in Fig. 2.4 to give some sense for what the traces look like for 20s for each CCA. These traces each capture some of the cyclical behavior of each CCA which helps CCAnalyzer to be accurate

[2]Note: We only include BBRv1 in our testing and training sets. In §2.5.2 we will show we can also classify websites using BBRv3.

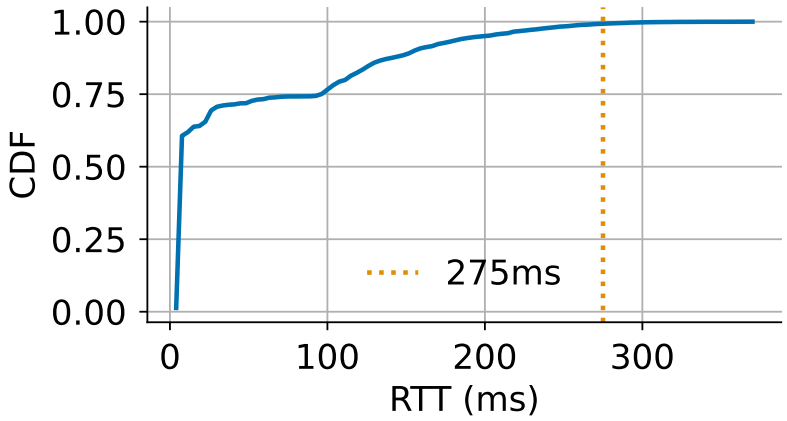


Figure 2.5: Distribution of ping time to 10K websites. Most websites are within a distance of 275ms.

and interpretable.

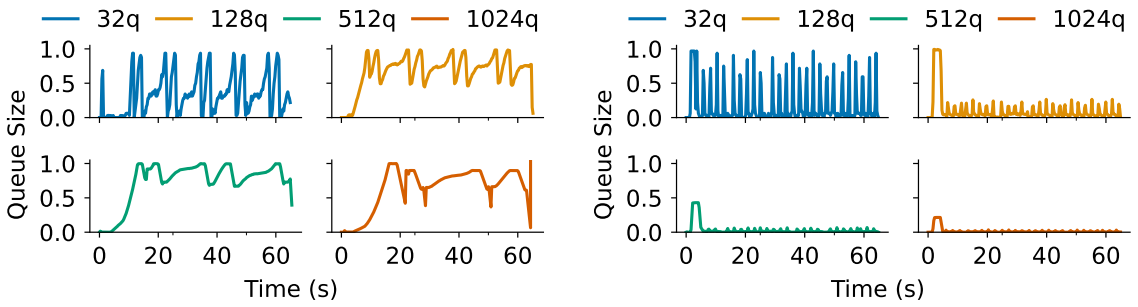
2.3.3.2 Network Configuration

Many CCAs, especially Reno-family CCAs, are designed to behave similarly in low-BDP environments. Therefore, we need to identify network settings in which these CCAs exhibit their distinguishing behavior. Our testbed enables us to emulate different network conditions by varying the bottleneck bandwidth, round-trip time, and the bottleneck queue size. Our main goal is to find a minimal set of network settings that we can confidently use to classify all 15 CCAs in Linux and identify unknown CCAs. We need to capture just enough cycles of CCA probing behavior that makes these algorithms distinguishable.

Bandwidth setting: We choose to use small bandwidth ranges because we want to ensure that our queue is the bottleneck for the connection; if queueing were to build up elsewhere in the network we would not observe useful behavior in the queue occupancy traces. We test setting the bandwidth to 5Mbps, 10Mbps, and 15Mbps.

RTT setting: We enforce an RTT in our testbed by adding additional delay to packets sent to the web server. Therefore, the RTT we choose for our network settings cannot be so small that the majority of websites will be too far away. In addition, if we set the RTT to be too large, then it can take the CCA a long time to fill the queue resulting in traces without enough cycles of CCA probing behavior to distinguish different algorithms. Fig. 2.5 shows the distribution from the 10K websites we will attempt to classify in §2.5. We test setting the RTT to 85ms, 130ms, and 275ms.

Queue size: Given the bandwidth and RTT of a setting, we need to chose a queue size that captures the right number of cycles of CCA probing to highlight distinguishable



(a) Cubic traces

(b) BBR traces

Figure 2.6: 5bw-275rtt: Example BBR and Cubic queue occupanchy traces from Azure-East for varying queue sizes (pkts).

behavior. We find a queue size of 1BDP works well.[3] Fig. 2.6a and Fig. 2.6b show how queue occupancy traces change depending on the queue size for a 5Mbps and 275ms RTT network setting (128 packets is ~1 BPD in this setting). With queue sizes too large, queue occupancy traces degrade. In case of Cubic, it takes too long to fill the queue so the trace does not have enough cycles of Cubic probing behavior. In the case of BBR, it uses very little of the queue when the queue it is too large.

We run 1NN-DTW on our test dataset from Azure-East, classifying each 60s trace as its 'nearest' training trace. Fig. 2.7 shows the accuracy of CCAnalyzer for 9 network settings for each testing set. Some settings work slightly better than others but overall accuracy across these settings is 96% (649 correct out of 675 samples). Misclassifications include Illinois samples misclassified as Westwood, both Reno variants. Similarly, BBR samples get misclassified as Vegas, both low-latency CCAs. The most accurate 4 settings are when the bandwidth is 5mbps or 10mbps, and when the RTT is 85ms or 130ms.

Our ultimate design relies on voting across multiple settings in order to 'boost' our accuracy to 100%, but we want each voter to be as confident as possible: hence we restrict our measurements in CCAnalyzer to the four most accurate settings. In addition, we want to minimize the load on the web servers by finding a small number of network settings that can produce distinct traces.

Fig. 2.8a shows an example of why 1NN-DTW works well, with a BBR testing sample and its closest training sample which are nearly identical. More illuminating is how closely the testing sample relates to the *incorrect* CCAs. Fig. 2.8b shows all the distances between a BBR sample and all the training samples in the 10bw-130rtt setting. All the BBR training samples are closest to this testing sample, but other similar CCAs that are also not loss-based, such as CDG and Vegas, are the next closest. These algorithms all have relatively low magnitude in their queue occupancy compared to, *e.g.*, Reno and Cubic variants. This

[3]BESS requires the queue size to be a power of 2 so the actual queue size is set to be a power of 2 closest to 1BDP.

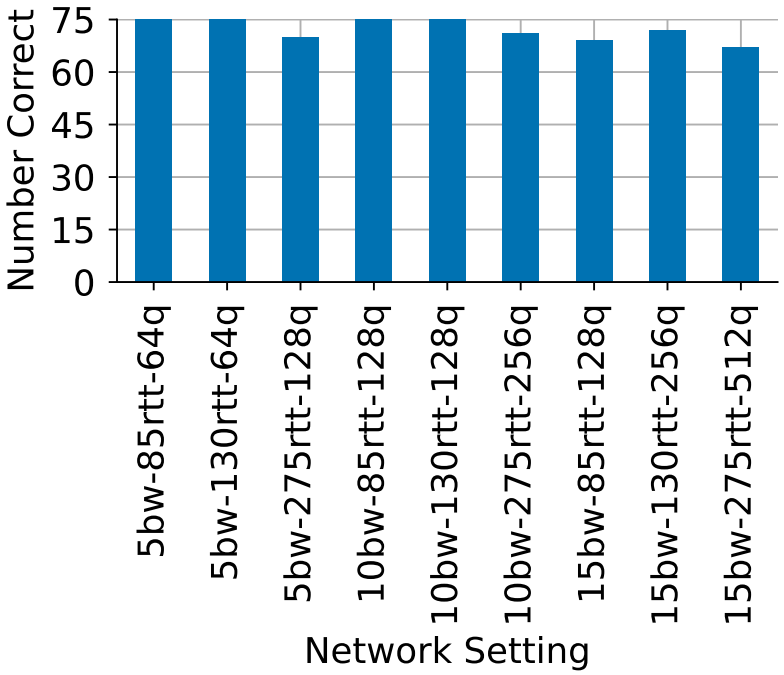


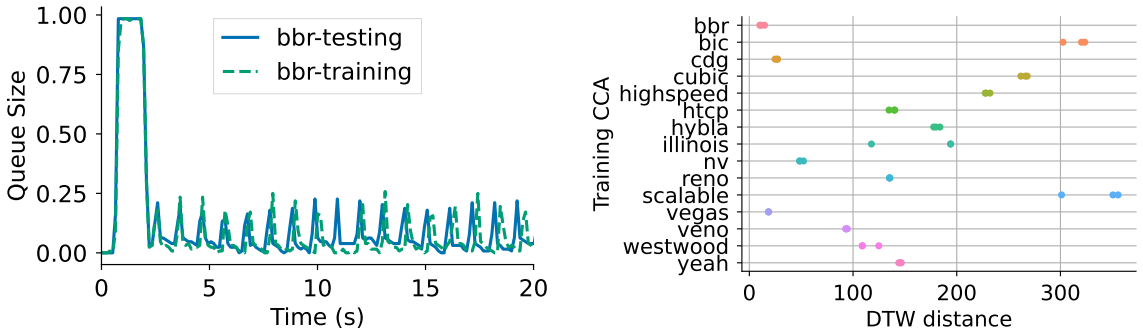
Figure 2.7: Accuracy mapping each testing sample to closest training sample per network setting.

also highlights the interpretability of our results as the traces are visually distinct, with clear similarities between testing samples and their closest training sample and the DTW distance *quantifies* the similarity. Furthermore, unlike both Gordon and IG, CCAnalyzer *does not need a special test* to classify BBR or other algorithms that are not loss-based.

Given the accuracy we have for these 4 settings, we complete the rest of our analysis and measurement study using these settings. These work well and achieve our goals but these are not the *only settings* that will have high accuracy. There are many settings that could accurately distinguish CCAs using 1NN-DTW. We discuss further network setting options and their accuracy in §2.5.3.

2.3.3.3 Trace Length/Duration

One of our key goals with CCAnalyzer is to reduce the overhead of probing relative to prior approaches. At the same time, we need to observe CCAs over a sufficient period of time such that they iterate through multiple 'cycles' of their bandwidth probing mechanisms. Consequently, we aim to identify the minimum duration we should measure a network trace while still ensuring strong accuracy. Fig 2.9 shows the accuracy from classifying flows individually (without voting) with durations ranging from 10 to 50s. We see a modest dip in accuracy when we drop as low as 10s. However, for traces from 20s-50s, we see relatively indistinguishable accuracy. Hence, we can use traces as short as 20s with



(a) 10bw-130rtt setting: A BBR testing trace correctly labelled.

(b) 10bw-130rtt: BBR trace correctly labelled. It is close to other low-latency CCAs, Vegas and CDG.

Figure 2.8: 10bw-130rtt: BBR trace correctly labelled. It is close to other low-latency CCAs, Vegas and CDG.

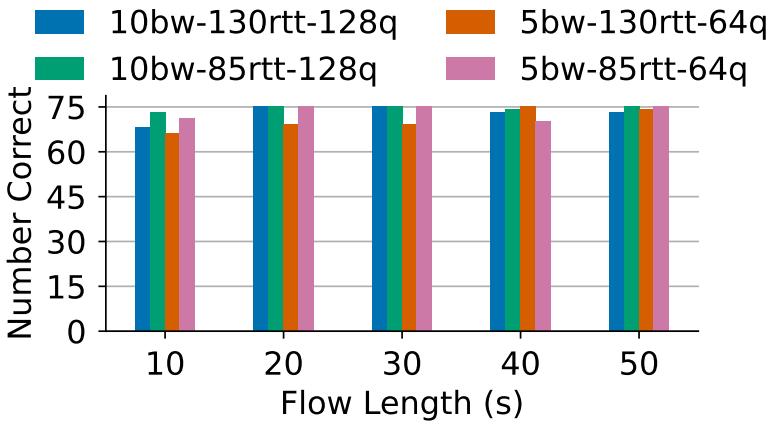


Figure 2.9: Results from classification for truncated traces in accurate settings. Near perfect accuracy is reached with as little as 20s flows.

minimal impact to classification accuracy and hence use this duration as our minimum trace length.

2.3.3.4 Classifying Unknowns

Our final parameter tuning step enables us to identify *unknown* or *novel* CCAs. This is referred to as solving an 'open-set' classification problem (a problem in which some of the data to be classified may not match any of the labels in the training set) rather than a 'closed-set' problem. In prior work, only Gordon [83] provides an open-set algorithm – all other algorithms in the literature, including Inspector Gadget, are closed-set, meaning that they will always erroneously identify novel CCAs as some other existing algorithm in

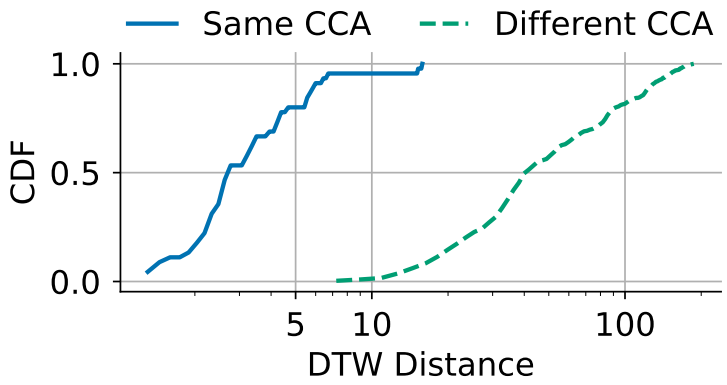


Figure 2.10: Distribution of distances between training samples for 5bw-85rtt-64q accurate settings. The seperation between the same CCA distribution and different CCA distribution suggests we can set a distance threshold to mark CCAs as unknown.

the training set.

CCAnalyzer's mechanism for identifying novel CCAs requires identifying some DTW distance threshold T such that if the nearest training sample to a trace is more than T distance away according to DTW, we should mark it as unknown. The algorithm for classifying with such a threshold, called TNN [81], is otherwise identical to the 1NN algorithm we described previously. Figure 2.10 provides intuition as to why such a threshold is useful. Here, we plot a CDF of all DTW distances between pairs of traces in our training data in which the pairs use the *same* CCA or in which they represent *different* CCAs. The distribution of distances between samples with the same CCA is tight – between roughly 1 and 15 – where pairs of different CCAs generally have a much higher DTW distance between them. The key is to choose the threshold T smartly: if we set T too high, we will mark true unknowns with a known CCA (*a false known*) and if we set T too low we will mark things that should have been labeled as a known CCA as unknown (*a false unknown*). Between the two classes of errors, we slightly prefer false unknowns because we believe that the vast majority of servers on the Internet do indeed use well-known CCAs. Consequently, we choose a low T that will have some false unknowns. In §2.5 we explore how we can further reduce false unknowns through clustering.

Our challenge in setting T is that we lack a way to rigorously evaluate our choice of T, since we lack ground-truth knowledge about the deployment of novel CCAs on the Internet, or even at what frequency novel CCAs are used. We can, however, emulate the deployment of novel CCAs to guide our search for a good value of T.

We use our *existing* training data (AWS-Virgina) and run classification on a *new* testing set (we use a server hosted in the AWS-Ohio region) to simulate unknowns. To classify a testing sample, we remove that testing sample's CCA from the training set. For example, when we want to classify a Reno testing sample, we remove all Reno training samples from

Table 2.3: Distance thresholds per setting.

Setting	Quantile	Distance Threshold
10bw-130rtt-128q	0.90	4.41
10bw-85rtt-128q	0.94	6.45
5bw-130rtt-64q	0.90	9.73
5bw-85rtt-64q	0.95	6.68

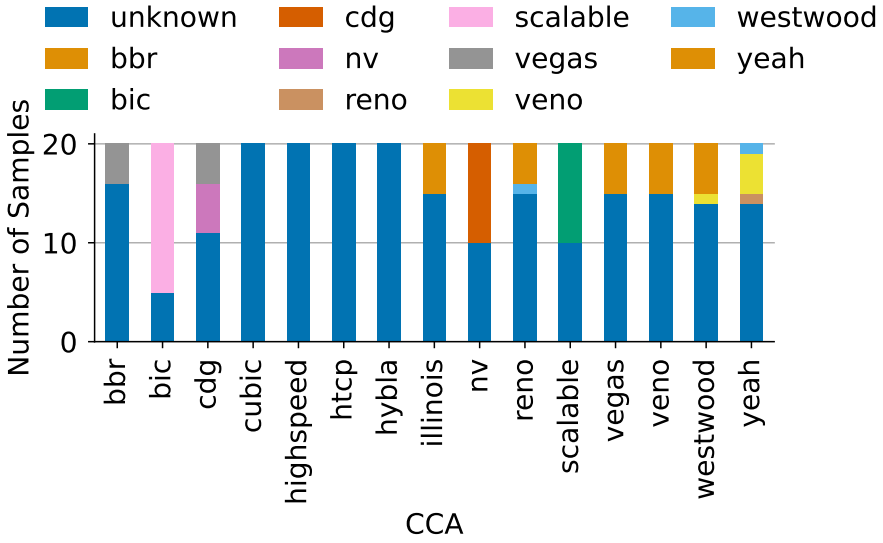


Figure 2.11: False positives when removing the training samples with the correct label from the testing set and seeing if we can correctly classify as unknown using a distance thresholds in Table 2.3 per CCA. After voting only CDG, BIC, and Scalable are misclassified as known labels.

the training set, and see if the Reno testing sample will be correctly classified as unknown, or if it will be erroneously given a known label. We repeat this process for all 15 CCAs, and vary T to balance false knowns and false unknowns. Table 2.3 shows the results of these experiments with our choice of T for each setting. For example, in the 5bw-85rtt setting, we choose the value that is the 95th percentile of the "Same CCA" distribution in Fig. 2.10.

To evaluate how well these values of T work, we repeat this process with the Azure-East testing set. Figure 2.11 shows how each CCA is classified when we remove that CCA's training samples; ideally the CCA should be classified as unknown. Once we apply our voting scheme across all four settings (voting description in §2.3.4), only CDG, BIC, and Scalable are misclassified with known labels – and are mislabeled with similar CCAs (CDG

is mapped to another low-latency CCA; BIC and Scalable are mapped to each other).

Now that we have a mechanism to classify unknowns, a new question arises: how do we tell which services are all using the *same* unknown? The short answer is that we can cluster unknown traces using pairwise DTW distance measures – groups of traces with small distances between them are likely to represent the same novel CCA. We return to this clustering procedure in §2.5.1.

2.3.4 CCAnalyzer End-to-End

In order to classify servers, CCAnalyzer is configured with a ground truth set of labeled queue occupancy traces for 15 CCAs for 4 network settings. Using TNN-DTW and the testbed in Fig. 2.2, CCAnalyzer does the following to classify a server:

1. Collect a queue occupancy trace for 20s (§2.3.3.3) for 4 network settings where the bandwidth is 5 or 10mbps and RTT is 85 or 130ms (§2.3.3.2).
2. Compute the DTW distance between each queue occupancy trace and all the training traces in the same network setting.
3. Each queue occupancy trace is given the label of the CCA that has the closest DTW distance.
4. If the distance is bigger than a distance threshold shown in Table 2.3 (§2.3.3.4) the trace is marked as unknown.
5. To assign the final label, for a website there is a vote between the 4 traces for the website. The final label for the website is the majority label across the 4 traces. If there is a tie between a known label and marking it as unknown, the CCA is marked as the known label. Lastly, if there is a tie between multiple CCAs, the final label is from the trace with the minimal distance to its closes training sample.

Finally, we use Agglomerative Clustering [86] to group unknown traces based on their DTW distances to each other. We use the distance threshold and manual inspection of these clusters to detect and identify proprietary, new, or unknown algorithms. CCAnalyzer is the only classifier which clusters unknown CCAs in any automated fashion. We explore the accuracy and efficiency of this approach in the next section.

2.4 Evaluation

In §2.4.1 and §2.4.2, we measure the accuracy and efficiency of CCAnalyzer and compare its performance with Gordon and IG. We were unable to obtain an executable version from the authors of IG, and ultimately had to re-implement it using the same techniques described in the paper (we describe them in §2.2) to the best of our ability. We find that CCAnalyzer is able to achieve 100% accuracy using its voting scheme for *all* 15 built-in CCA algorithms in Linux. During trace collection, on average, CCAnalyzer transmits 85% fewer bytes of the data that Gordon needs to classify a website, and completes 40x faster in terms of wall-clock time. CCAnalyzer achieves the same accuracy and coverage as IG and better efficiency than Gordon (§2.4.2), with the flexibility of open-set classification

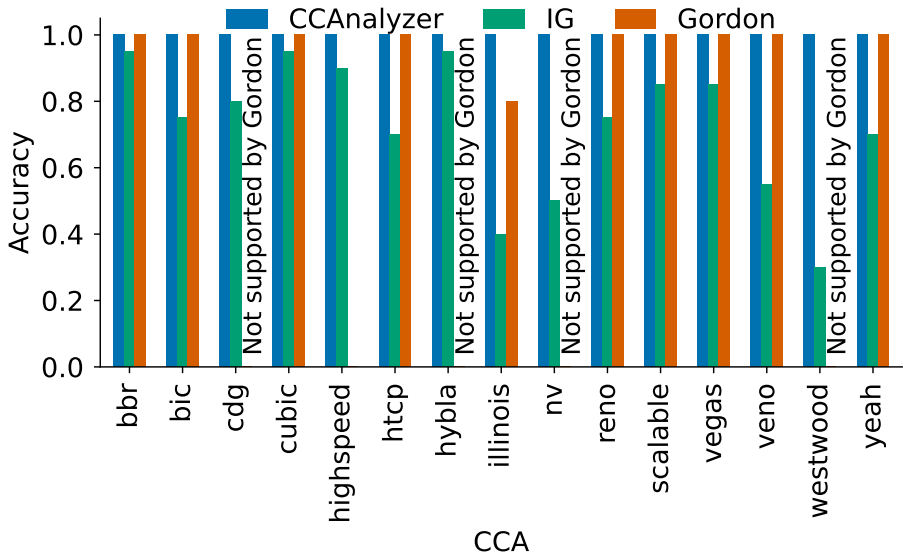


Figure 2.12: Comparison between CCAnalyzer, IG and Gordon classifying the same servers.

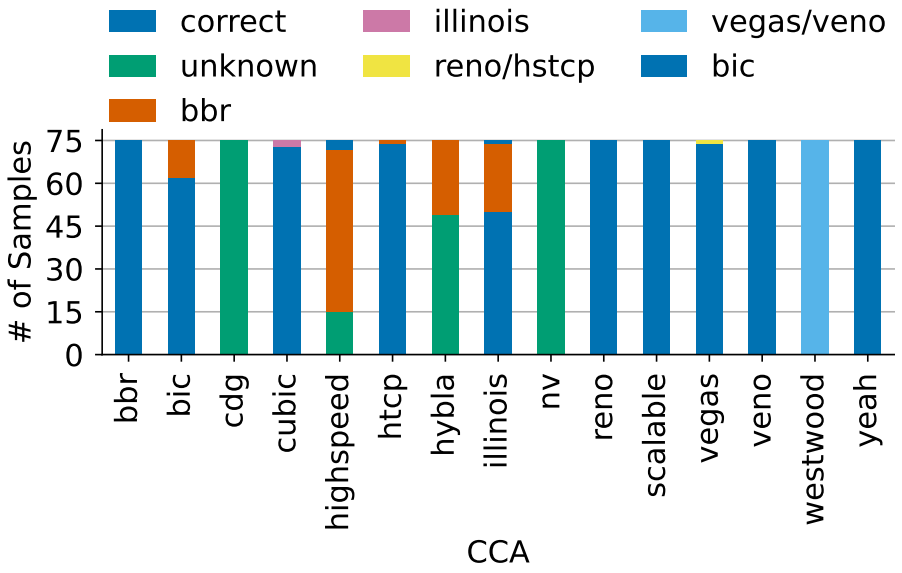
(§2.5.2) and more interpretable results (§2.4.1).

2.4.1 Accuracy

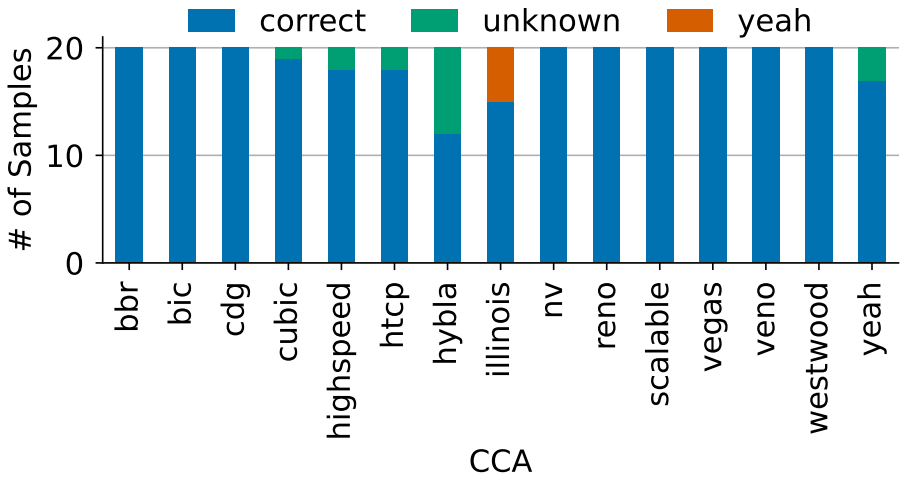
Experimental setup: We evaluate Gordon and IG using the same Azure-East web server we use to evaluate CCAnalyzer (§2.3.3.1). We point the Gordon client and IG client, installed on a server in the CloudLab Utah testbed, to download the same 100MB file from the Apache web server. We classify each CCA 5 times for Gordon and CCAnalyzer using their hand-crafted decision tree (as is done in the Gordon paper). We classify each CCA 20 times for IG (as is done in the IG paper) using traces collected from the same AWS-Virginia web server as the training set.

Both Gordon and CCAnalyzer use a voting scheme to determine their final result. In the case of CCAnalyzer, we generate measurements in four bandwidth/RTT/queue-size settings, measure DTW distances to our training data for each sample, and then vote across these four settings (§2.3.4). In the case of Gordon, they run 15 trials and take a vote across these 15 trials. To repeat classifying each CCA 5 times, CCAnalyzer classifies 20 queue occupancy samples per CCA and Gordon classifies 75 `cwnd` trace samples per CCA.

In Fig. 2.12 we show the number of correct classifications for IG, Gordon and CCAnalyzer. For both Gordon and CCAnalyzer we report the results after applying their voting schemes. CCAnalyzer achieves 100% accuracy across all CCAs. The results for Gordon are more complex: CDG, Hybla, and New Vegas (nv) are not supported by Gordon and so we mark these as unsupported. Further, the published code does not support Westwood so we also mark that as unsupported. For the algorithms that Gordon does support, it misclassified all Highspeed samples, and is mostly accurate for the other CCAs



(a) Gordon



(b) CCAnalyzer

Figure 2.13: Individual votes for each CCA trace for Gordon and IG. Note that CDG, NV, and Hybla are correctly marked as unknown for Gordon because they are not in their training set.

We illustrate the accuracy of these individual votes in Fig. 2.13a for Gordon and in Fig. 2.13b for CCAnalyzer. Stacked bars show how many 'votes' went to each CCA. For CCAnalyzer, its individual votes are accurate with the exception of marking known CCAs as unknown (we do favor false unknowns vs. false knowns §2.3.3.4) and mislabelled Illinois

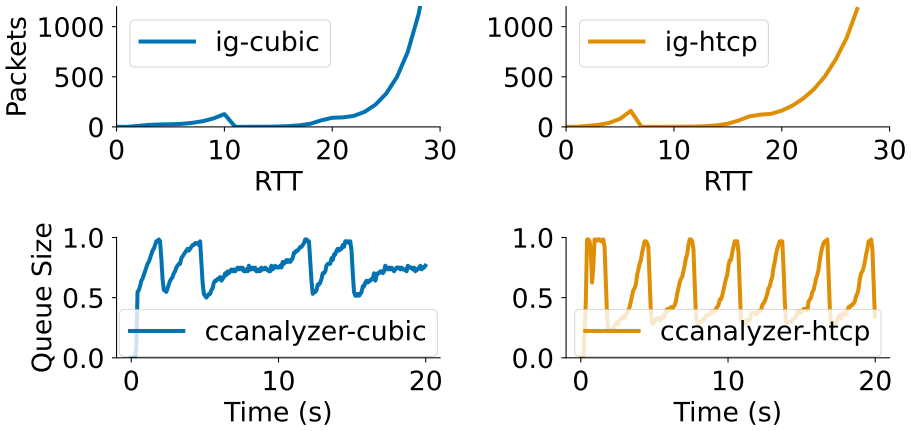
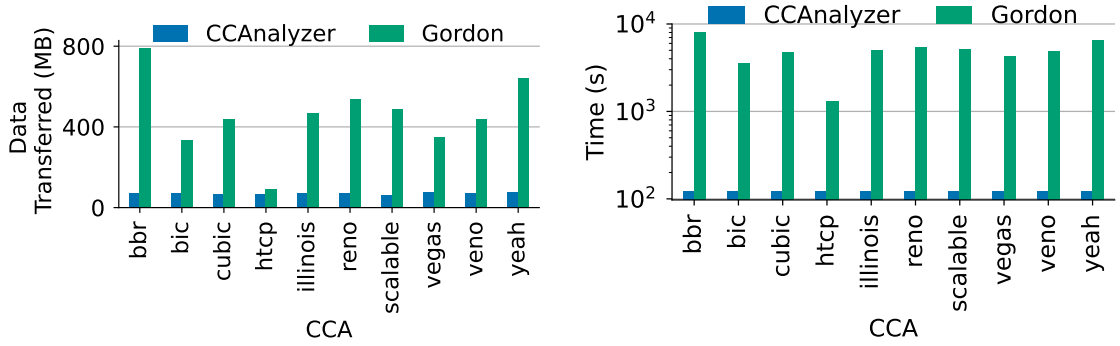


Figure 2.14: Traces from CCAnalyzer and IG.

samples as YeAH (both are variants of Reno). For Gordon, the errors are more varied, with several loss-based protocols (BIC, Highspeed, and Illinois), identified unexpectedly as BBR. While the results in the Gordon evaluation include correct classifications for Westwood, the publicly released code [47] for Gordon does not classify traces as Westwood, and therefore does not support this algorithm. Notably, Gordon does correctly classify 3 algorithms it does not support (CDG, Hybla, and NV) as unknown, demonstrating its ability to classify some CCAs not in its known set as unknown.

To validate that our IG implementation is faithful, we attempt to replicate the results in [46] by using Azure servers to generate both testing and training samples. Under this setting, IG achieves 100% accuracy, likely due to over-fitting. IG's overall accuracy dips to 73% if we include all 15 CCAs in the training and testing set and use AWS training samples to classify Azure testing samples (like we do to evaluate CCAnalyzer); these are the results shown in Fig. 2.12. When restricting this set to just the 12 CCAs that IG classifies in their paper, the accuracy is 74%.

Interpretability: There are many competing definitions for what makes a classifier "interpretable" [75]. Human experts want to be able to understand our classifier's output: why are these two traces labeled as the same CCA? Is this classification likely correct? This is one of the key advantages of CCAnalyzer over the prior work: capturing the inherent cyclical nature of CCAs, makes them more distinguishable. So much so, that not only can a classifier find these distinguishable patterns, but so too can a human observer. In Fig. 2.14, we compare cwnd traces from IG to queue occupancy traces from CCAnalyzer. While the top graphs show traces for IG for Cubic and HTCP are nearly identical, the traces for the same CCAs from CCAnalyzer are easily distinguishable. Note cwnd is not *directly measurable*, and techniques that *estimate* cwnd by forcing timeouts have a shorter set of observations. This makes it more difficult to interpret these cwnd estimations rather than queue occupancy measurements. CCAnalyzer is able to achieve better accuracy than



(a) Data transfer comparison

(b) Time comparison

Figure 2.15: Efficiency comparison between CCAnalyzer and Gordon classifying the same web server. Gordon's cwnd estimator depends on the CCA so both bytes transferred and time is CCA dependent.

prior work, with the important additional benefit of interpretable results.

2.4.2 Efficiency

We have two measures of efficiency: total bytes transferred and wall-clock time. Using our testbed experiments, we measured that for CCAs supported by Gordon, CCAnalyzer requires on average 15% fewer bytes to perform classification and completes probing 40x faster in terms of wall-clock time. IG is more efficient than Gordon and CCAnalyzer. For all of these classifiers, classification is inexpensive and done offline after collecting traces, so here we only consider the efficiency of collecting the traces before classification. We collect pcaps for all experiments and measure the average amount of bytes transferred between the web server and the client for classifying each CCA. In addition, we measure the time from the first packet sent from the client to the last received from the web server.

Bytes transferred. Fig. 2.15a compares the number of bytes transferred between CCAnalyzer and Gordon. Because Gordon waits to measure the reaction to packet loss, the time and amount of data transferred to classify a webpage is heavily dependent on the CCA. Because BBR does not respond to individual packet losses, it transmits more data during the measurement and requires a special test to classify. In contrast, CCAnalyzer's classification is not as dependent on the CCA, aside from CDG which doesn't always manage to maintain full throughput, data transferred is independent of the underlying algorithm. The mean number of bytes transferred for CCAnalyzer over the 13 CCAs supported by Gordon is 68MB (total for collecting 4 traces) while for Gordon it is 456 MB (with a large std dev. of 186) since it heavily depends on the CCA. IG only collects 1 trace for up to 50 RTTs without any repetitions or restarts and at most transfers 2MB. However, since IG only emulates a single timeout, this efficiency comes at the cost of failing to capture the cyclical nature of CCA behavior, leading to worse accuracy than CCAnalyzer

and poor interpretability of generated traces (see Fig. 2.14).

Time comparison. Fig. 2.15b compares the amount of time it takes to collect traces for CCAnalyzer and Gordon. CCAnalyzer only needs 20s per trace. Including setup, CCAnalyzer overhead takes only a maximum of 30s per measurement and is not dependent on the CCA. Since we collect 4 traces for each CCA the total amount of time for trace collection for CCAnalyzer is about 2 minutes. In contrast, Gordon's runtime heavily depends on the CCA with a max of 130 minutes and a minimum of 2.6 minutes to complete all of its 15 trials. IG takes at most 90s to collect traces.

2.5 CCA Measurement Study of Top 10K Websites

Table 2.4: Classification results for websites by CDN websites. The values after the slashes are after a clustering step on traces within each CDN.

CDN	Akamai	Cloudflare	Cloudfront	Fastly	Google	Other CDN	No CDN	Total
BBR	470/491	1233/1595	530/545	21/25	29	28/32	116/122	2427/2839
BIC	0	0	0	1/13	0	2/0	3/0	6/13
CDG	3/4	6/7	9/10	3	1	1	8/9	31/35
CUBIC	4	5/6	7/10	25/130	2	53/92	89/116	185/360
Highspeed	0	0	0	0	0	0	3/5	3/5
HTCP	0	0	0	0	0	0	2	2
NV	0	0	3/2	0/1	2	1	5	11
Reno	0	1	0	0	0	0	4/3	5/4
Vegas	0	0	0	0	0	0	1/0	1/0
Westwood	0	0	0	0	0	0	3	3
Yeah	0	0	0	0	1	0	0	1
Unknown	115/91	824/460	74/56	174/52	230	72/31	146/115	1635/1035
All Invalid	189	394	78	26	37	54	127	905
RTT > 85ms	36	55	10	3	18	41	1205	1368
Unresponsive	233	989	121	30	66	226	1752	3417
Total	1050	3507	832	283	386	478	3464	10000

We conduct a measurement study using our comprehensive tool and our testbed in Fig. 2.2. We have two goals here. The primary goal is to demonstrate the effectiveness and robustness of CCAnalyzer in classifying known CCAs and detecting novel CCAs. We show how we can detect a new CCA, BBRv3, with minimal effort. The second goal is to take steps towards answering important questions about the current state of CCA deployment on the Internet today, for example: Is Cubic still the most dominant CCA? How has the

deployment of BBR evolved? Is Reno deprecated?

2.5.1 Methodology

The Google Chrome UX Report (CrUX) releases rank ordered lists of top websites, which is more accurate than alternatives [103, 98]. We pull the websites from the Top 10K bucket from the February 2023 dataset, which accounts for 70% of all Chrome page loads [135, 98]. The websites in the CrUX dataset are identified by *origin*, not domain. For example, this list includes `www.google.com`, `scholar.google.com`, `maps.google.com`, and so on as separate websites, so we try to classify each of these separately. While we believe that this measurement study covers a large fraction of popular websites, and we draw some important conclusions, we do not claim to be a comprehensive Internet measurement study. We leave a larger measurement study for future work (which is considerably more feasible with CCAnalyzer than prior work).

Both Gordon and Inspector Gadget had to search websites for a webpage large enough to download to generate `cwnd` traces. We found we could only classify 1% of the 10K websites with IG primarily because we could not find large enough files (§2.2). Similarly, we need a web transfer between the client and server for at least 20s. To achieve this goal without requiring large files, we use the `h2load` [48] tool to send multiple parallel HTTP requests to the websites we want to classify to download enough data from the webpage to utilize the available bandwidth (5Mbps, 10Mbps). We use the `findcdn` [38] tool to identify if a website is hosted by a CDN. Occasionally, this tool returns more than one CDN for a given website. In those cases, we use the first result returned by this tool.

Unresponsive and invalid traces: Table 2.4 shows a summary of how many websites we were able to successfully classify and their classifications. 34% of these 10K websites are "unresponsive" because they did not respond to pings or the homepage did not respond with a 200 `OK` response to `h2load`. 13% had RTTs that were larger than 85ms. In addition, we measure the bandwidth utilization for each trace. We set a bandwidth threshold of 80% because for all our training samples the CCA is able to use at least 90% of the available bandwidth; a threshold of 80% gives some headroom. A trace is marked invalid if it does not meet the bandwidth threshold. A website is marked as "All Invalid" if all of the traces collected for that website do not meet the bandwidth threshold. 9% of the websites have traces that are all invalid.

Validation and clustering within CDNs: We initially classify each web server using the methodology described in §2.3.3.1, and report those numbers in Table 2.4 (the numbers before slashes). We notice about 1600+ websites are marked as unknown which means most of the traces for these websites were not close enough to their closest training sample. Recall in §2.3.3.4, when we determined the distance threshold, we set a small T based on experiments to an AWS server. We favored *false unknowns* vs. *false knowns*. Because of the likely possibility of more noise in our measurements to third-party servers, the distance

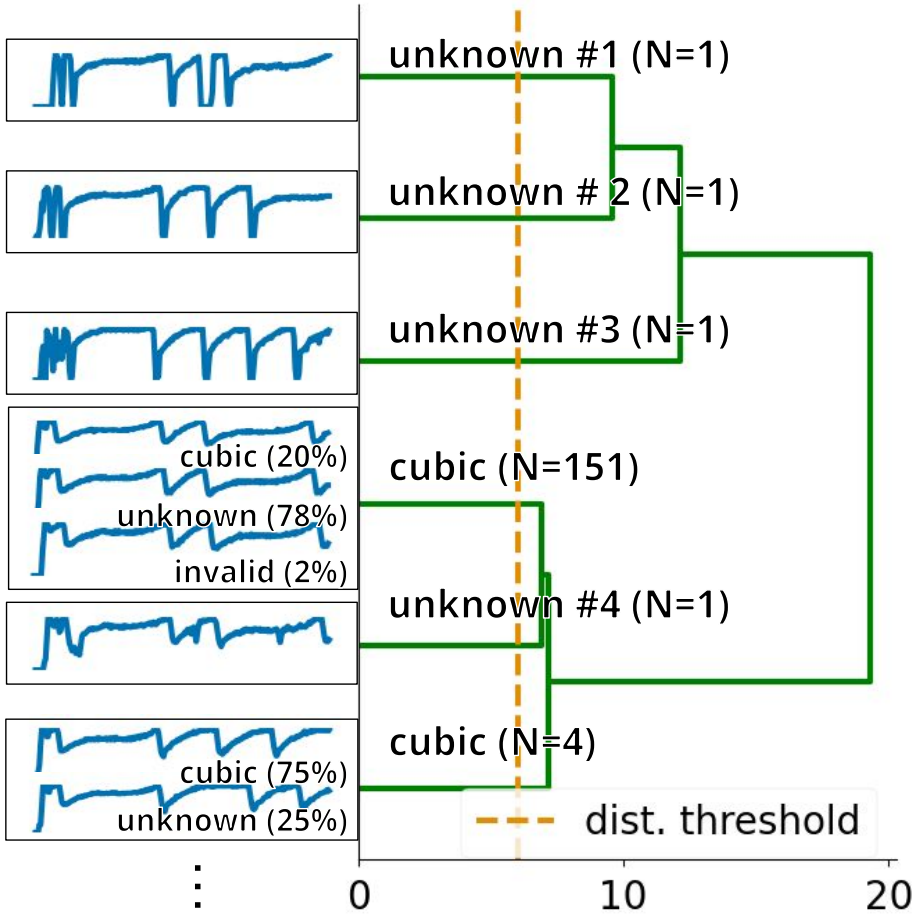


Figure 2.16: Example of a portion of a dendrogram from hierarchical clustering Fastly websites in 5bw-85rtt-128q setting. The vertical line is the distance threshold.

threshold may be too conservative. To further reduce *false unknowns*, we do an additional clustering step where we may re-classify websites. The values after the slashes are the counts per CDN, per CCA if there were changes after this additional clustering step.

In this additional step, we cluster all the CCAs that are in the same CDN using agglomerative clustering [86]. Agglomerative clustering works by putting each sample initially in its own cluster, and then merging samples into the same cluster based on the distances between the samples. We use the "average" metric based on the DTW distances between all samples, which links samples to minimize the average of the distances of each observation of the two sets. Once we compute the links between all the samples, samples can be put into clusters based on a distance threshold; we use the distance thresholds described in Table 2.3. If a resulting cluster contains 5% or more labeled traces, we re-classify unknowns or invalid traces as that label. For the cases where we re-classify, while the traces initially

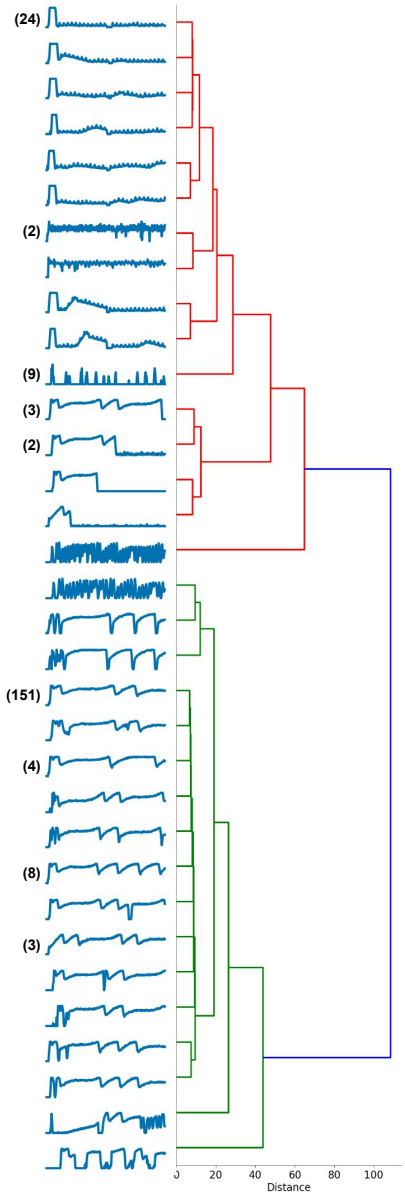


Figure 2.17: The full dendrogram of results from clustering across all the Fastly websites (5bw-85rtt- 128q setting) using the distance threshold. Each leaf shows one testing sample from each of the resulting clusters. Clusters with numbers indicate how many samples are in that cluster.

labelled unknown are not close enough to other *training samples*, they are close enough to other *testing samples* with a known label. We believe in this case, these are *false unknowns* and should be given a known label. After re-classifying traces, we re-do the voting across

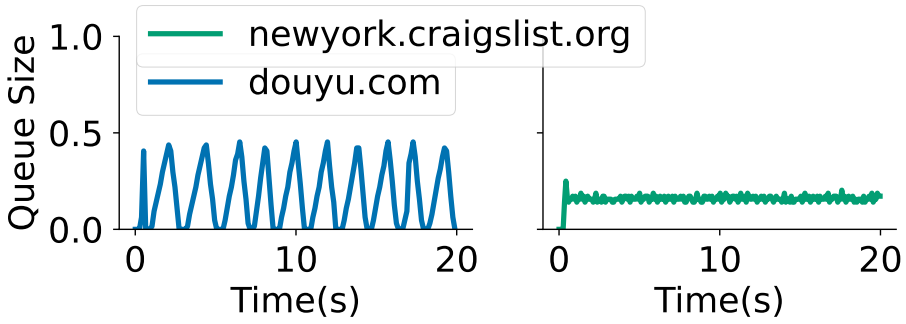


Figure 2.18: Example of unknown traces from websites not hosted by a CDN.

the 4 settings (§2.3.4) and give websites potentially new labels (we do not re-label 'All Invalid' websites).

We see two notable changes after this clustering of all samples from the same CDN. First, the majority of the unknown Fastly websites (105) are re-classified as Cubic. Second, there is a similar shift with Cloudflare websites: 362 websites are re-classified as BBR. In Fig. 2.16, we visually show the partial output of the clustering of Fastly results. The yellow vertical dotted line shows the clustering distance threshold used. The labels on the green lines indicate the final label given to all traces in each cluster and the number of traces in the cluster. The boxes on the left show some example traces in each cluster, along with their initial label. For example, in the cluster labeled 'cubic (N=151)', 20% of these traces are Cubic traces so this cluster is labelled Cubic. It is encouraging that the these traces are highly similar and are clearly Cubic traces based on manual inspection. Similarly, the process does a good job keeping the 'unknown' label for unusual traces.

There are two benefits of this clustering step. First, we can validate the results of our classification. Looking at the dendrograms of the output we can visualize how close samples are to each other and can see at what distance threshold similar samples are clustered together (highlighting the interpretability of CCAnalyzer results). Fig. 2.17 shows an example of a full dendrogram for Fastly websites in the 5bw-85rtt-128q setting. We expect like traces to end up close together, while dissimilar traces to be far apart. In this example, we see clusters of BBR and Cubic samples. Second, which we demonstrate in Fig. 2.16, it can help classify false negative unknowns as actually known CCAs.

2.5.2 Clustering Unknowns

After the initial classification as well as the clustering within CDNs and validation, we now have websites that are still classified as unknown. We take the traces from all of these websites, across CDNs, and run agglomerative clustering on all of them. We manually view the dendrogram of these results and look for web servers that are likely using the same unknown CCA.

BBRv3: We notice that the majority of websites originating from Google CDN are classi-

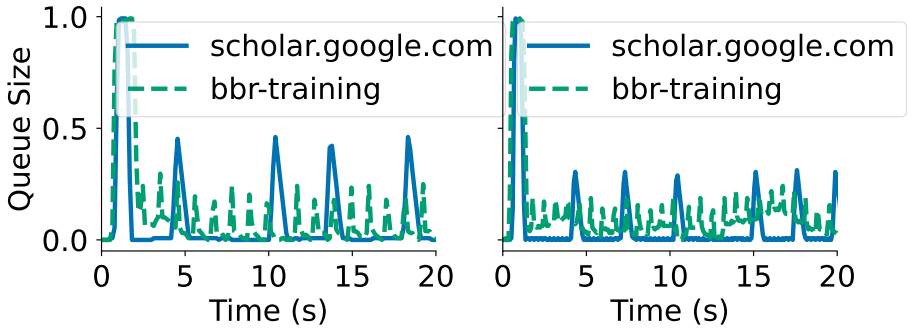


Figure 2.19: Example trace from a Google website that we believe is BBRv3 from 2 settings: 10bw-130rtt-128q (left), 10bw-85rtt-128q (right)

fied as unknown when we expected to see BBRv1. We see these traces are closest to BBRv1, but the periods of bandwidth probing and RTT probing are more spaced out. Fig. 2.19 shows an example queue occupancy traces for 2 settings for `scholar.google.com`. We conclude these sites are using BBRv3 and confirm with Google's BBR team [2]. According to presentations from Google, BBRv3 was deployed on Google servers by Summer 2023 when our measurement study was conducted in Fall 2023. Based on clustering, we label 102 Google CDN websites originally classified as unknown instead as BBRv3. This example also highlights the ease in discovering new CCAs with CCAnalyzer.

Other Unknowns: Our ability to cluster traces using DTW distances makes discovering new CCAs a simple and straightforward process of reviewing queue occupancy traces, dendrograms and DTW distances. We see the potential for further reverse engineering of these CCAs using recent work [37]. While some unknowns can be re-classified as existing designs as part of the clustering process, we still see other behaviors that remain classified as unknown, like clusters unknown #1-#3 in Fig. 2.16. In our broader study of websites, we find several websites using unknown CCAs which we highlight in Fig. 2.18. Note that further study is needed to determine if these are truly novel CCA designs or a known CCA in an unexpected or pathological state.

2.5.3 Addressing Limitations

We find our RTT limit (85ms), bandwidth utilization threshold (80% of 5Mbps and 10Mbps), and `h2load` settings, limit the coverage we have for the websites we could measure in this study. As noted earlier, 13% had too high RTTs. 9% had too low bandwidth, and 34% of the servers did not respond to `h2load`. This is not a fundamental limitation of 1NN-DTW and can be mitigated in several ways. In this section, we discuss how we could increase coverage, to potentially classify the 'unresponsive' websites and gather more valid traces in this study.

An obvious mitigation for the lack of response to `h2load` is to issue HTTP/1.1 requests if HTTP/2 requests fail. In addition, we could avoid using `h2load` by requesting a large

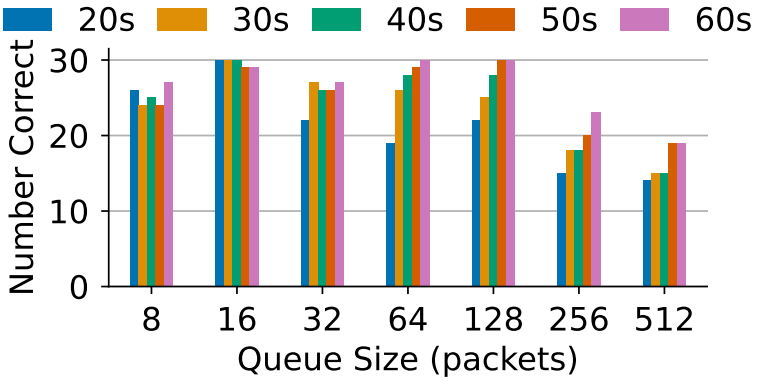
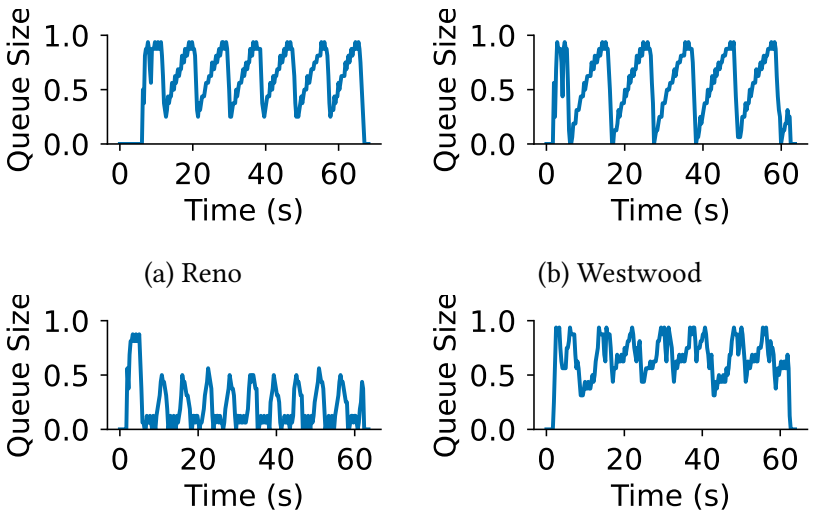


Figure 2.20: 0.3bw-275rtt: Accuracy for different queue sizes and trace lengths for 0.3Mbps and 275ms RTT setting using training and testing samples from Cloudlab server. A queue size of ~1BDP (16 pkts) works well.

file from the website using `wget` as we did in our evaluation. To reduce the number of servers with too high RTTs or too low bandwidth, we consider the smallest bottleneck bandwidth and largest RTT that we can use to generate clear queue occupancy traces. We consider an RTT constraint of 275ms based on Fig. 2.5. To determine a low bandwidth, we observe the queue occupancy plots as we decreased bandwidth by 0.1 Mpbs at a time. We did this until we visually observed a distinctive change in the shape. We find 0.3 Mbps produces distinguishable traces as shown in Fig. 2.21.

Given this bandwidth and RTT constraint, the core question is: what queue sizes will produce distinguishable traces? To answer this question, we generate 5 queue occupancy traces for varying queue sizes from within our testbed using `iperf` with a server that is a Cloudlab machine. We split these traces into 3 training and 2 testing samples for each of the 15 CCAs, using Cloudlab traces to classify other Cloudlab traces to determine which queue sizes result in high accuracy. Fig. 2.20 shows the accuracy for the 0.3 Mbps and 275ms RTT network settings with varying queue sizes and varying trace lengths. A queue size of about ~1BDP (16 packets) has an accuracy of 100% for traces as short as 20s, similar to the accuracy we see in §2.3.3.2. Fig. 2.21 shows the queue occupancy traces using a 16 packet queue.

CCAnalyzer can work with a wide range of network settings to support a large majority of web servers. We only need network settings that produce distinct queuing behavior for different CCAs for 1NN-DTW to work. While we only explore a few settings in this work, we also show that there other settings, especially ones with lower expectations from servers, that could be used in practice. Future work could use alternative network settings to classify websites that we did not in this study, improving coverage further without sacrificing accuracy.



(a) Reno (b) Westwood

(c) BBR (d) Cubic

Figure 2.21: 0.3bw-275rtt-16q: Example training samples traces from Cloudlab

2.5.4 Takeaways

Widespread deployment of BBRv1: While we are not able to classify all websites, we do find that the majority of those we can classify are classified as BBRv1. While we cannot conclude that there is an increasing deployment of BBRv1, we do see widespread deployment at "Hypergiants"[45] like Akamai which was previously known to have deployed a different CCA [83]. We so see some large CDNs (*e.g.* Fastly) still use Cubic.

Discovery of new CCAs: Using CCAnalyzer, we can automatically discover new CCAs, as we show in this work with our discovery of BBRv3. We do not need specialized or hand-crafted tests to classify new CCAs (like Gordon and Inspector Gadget) nor do we need to know details of how the algorithm works (like Gordon). We can add support for new CCAs, like BBRv3, simply by collecting queue occupancy traces and adding them to our training set.

2.6 Conclusion

CCAnalyzer takes a significant step forward in CCA classification. While only relying on collecting bottleneck queue occupancy traces, CCAnalyzer achieves accuracy that is equal to or better than state-of-the-art classifiers. In addition, CCAnalyzer is efficient, unobtrusive, has interpretable results, and supports open-set classification. We use CCAnalyzer to analyze the CCAs of 5000+ websites. CCAnalyzer's DTW-based distance measure allows it to not only detect unknown CCAs, but also cluster them into groups of similar unknowns, simplifying the detection and classification of new CCA variants as they appear on the Internet. Unlike prior work, CCAnalyzer's approach has the potential to classify

the rising popularity of user-space protocols (*e.g.* QUIC) and other popular applications (*e.g.* video streaming), a promising direction for future work.

At the beginning of this chapter we asked: **What congestion control algorithms are widely deployed on the Internet today?** Of the 10K websites we attempted to classify, 35% were from Cloudflare, the most common CDN in our dataset. Of these, 45% (1595 websites) were classified as BBRv1. In addition, we saw deployment of BBRv1 at other large CDNs Cloudfront and Akamai, and the deployment of Cubic at Fastly. We found a number of unknown CCAs. With these results, we confirm that this heterogeneity in the Internet today as we declare in our thesis statement. Given this deployment of BBRv1 and Cubic, in the next chapter we discuss how BBRv1 interacts with loss-based congestion control.

Chapter 3

Modeling BBR's Interactions with Loss-Based Congestion Control

Very insightful paper - and a thorough analysis of the problem. It is interesting that such a major aspect of BBR's behavior – how much of the bottleneck queue it occupies – hinges on an arbitrary choice.

Reviewer #5

3.1 Introduction

In 2016, Google published a new algorithm for congestion control called BBR [25, 24]. Now deployed as the default CCA for Google services including YouTube, which commands 11% [77] of US Internet traffic, BBR consequently impacts a large fraction of Internet connections today. In our measurement study in Chapter 2, we also saw rapid and prolific deployment of BBR beyond just Google. While BBR is available in the Linux kernel, Cubic is still the default. This leads to the second question in this dissertation: **How does BBR interact with loss-based CCAs?**

We are not the first to investigate BBR's properties when competing with traditional loss-based CCAs. Experimental studies have noticed two key phenomena. First, in shallow-buffered networks, BBR's bandwidth probing phase causes buffer overflows and bursty loss for competing flows; these bursts can lead to Cubic and Reno nearly starving for bandwidth. This phenomena was first explored in [55] and BBRv2 was expected to patch the problem [22]. Currently BBRv2 is deprecated and has been replaced by BBRv3 which is now deployed on all Google sites [27].

In residential capacity links (*e.g.* 10-100Mbps) with deep buffers, studies [124, 34, 25, 104, 119] have generated conflicting reports on how BBR shares bandwidth with competing Cubic and Reno flows. We [124] and others [34, 104] observed a single BBR flow consuming a fixed 35-40% of link capacity when competing with as many as 16 Cubic flows. These

findings contradict the implication of early presentations on BBR [25] which illustrated scenarios where BBR was *generous* to competing Cubic flows. In short, the state of affairs is confusing, with no clear indication as to *why* any of the empirically observed behaviors might emerge.

The contribution of this chapter is to model BBR's behavior when it competes with traditional, loss-based congestion control algorithms in residential, deep-buffered networks (studies [68] suggest that residential routers typically have buffer depths 10-30× a bandwidth-delay product for a 100ms RTT). The key insight behind our model is that, while BBR is a rate-based algorithm when running alone, BBR degrades to window-based transmission when it competes with other flows. BBR's window is set to a maximum 'in-flight cap' which BBR computes as $2 \times RTT_{est} \times Btlbw_{est}$, for RTT_{est} and $Btlbw_{est}$, BBR's estimates of the baseline RTT and its share of bandwidth.

While the original BBR publication presented the in-flight cap as merely a safety mechanism – included to allow BBR to handle delayed ACKs [24] – this mechanism, unexpectedly, is the key factor controlling BBR's share of link capacity under competition. Our model focuses on how BBR estimates its in-flight cap under different network conditions; by computing what we expect BBR's in-flight cap to be, we can predict BBR's share of link capacity for long-lived flows. The size of the in-flight cap is influenced by several parameters: the link capacity and latency, the size of the bottleneck queue, and the number of concurrent BBR flows. But, notably absent, the number of competing loss-based (Cubic or Reno) flows *does not play a factor* in computing this in-flight cap. Hence, BBR's sending rate is not influenced by the number of competing traditional flows; this is the reason behind reports that BBR is 'unfair' to Cubic and Reno in multi-flow settings [124, 34].

In what follows, we discuss our testbed in §3.2 and early measurements of BBR's 'fairness' or 'friendliness' in §3.3. We then provide a primer on the BBR algorithm in §3.4. We then develop our analysis of BBR in §3.5 along with an explanation of BBR's convergence to 40% of link capacity. We connect our results to related work in §3.6 and and conclude in §3.7.

3.2 Testbed

Throughout this chapter, we show experiments generated in the testbed illustrated in Fig. 3.1. Each experiment involves three servers: a server/sender, a BESS [50] software switch, and a client/receiver. All servers are running Linux 4.13 (using internal TCP pacing), have Intel E5-2660V3 processors, and have dual-port Intel X520 10Gb NICs. Senders and receivers use iPerf 3 [58] to generate/receive traffic. Within BESS, traffic is serviced at a configurable rate below the link capacity to introduce queueing. The queue size is set to ratios relative to BDP; since the BESS queue module only supports powers-of-two sizes we rounded to the nearest power-of-two. To configure delay, we hold all ACKs for a configurable amount of time. Unless otherwise noted, we set bandwidth to 10 Mbps and RTT to 40ms, following Google's parameters in IETF presentations [25, 26].

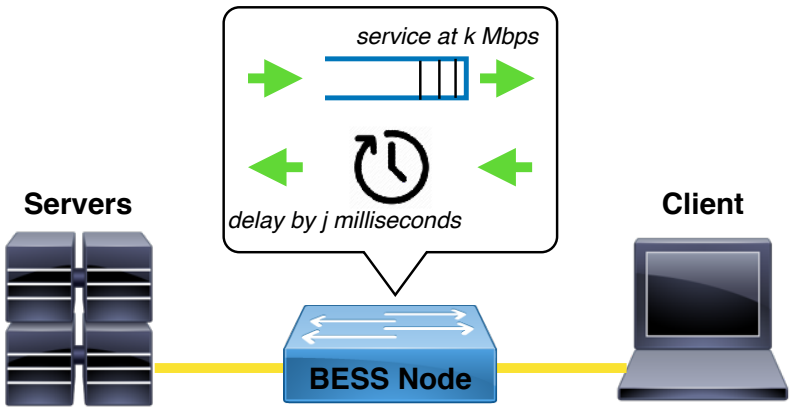


Figure 3.1: Testbed for congestion experiments which introduces queueing at a controlled bottleneck.

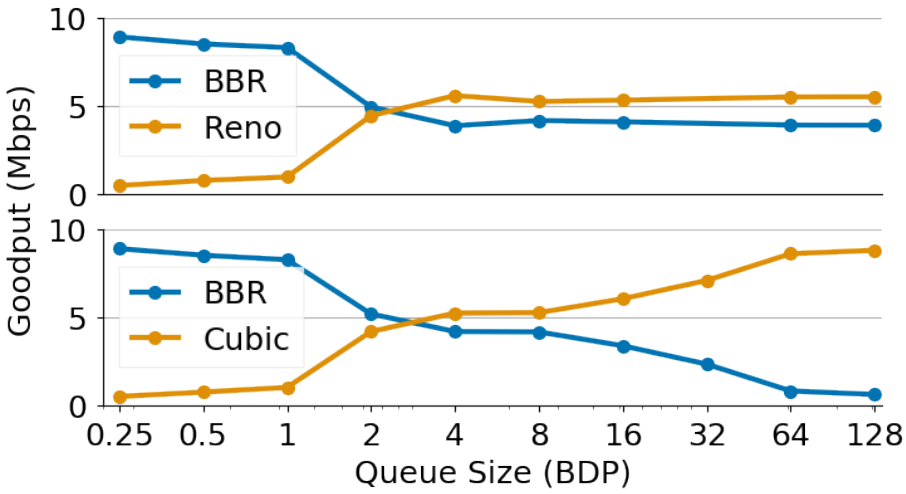


Figure 3.2: Average goodput for two competing flows over 4 min in a 40ms×10Mbps network with varying queue sizes.

3.3 BBR In Competition

A natural concern when deploying a new CCA on the Internet is how the new CCA will interact with other deployed algorithms. Will the new CCA be 'fair' to existing CCAs, or starve them?

An early BBR presentation [25] provided a glimpse into these questions. A graph in the presentation measures 1 BBR flow vs. 1 Cubic flow over 4 minutes, and illustrates a correlation between the size of the bottleneck queue and BBR's bandwidth consumption. We set out to replicate Google's experiments and easily did so – shown in Fig. 3.2 – as done in other studies [104]. The implication of these graphs is that BBR is generous to

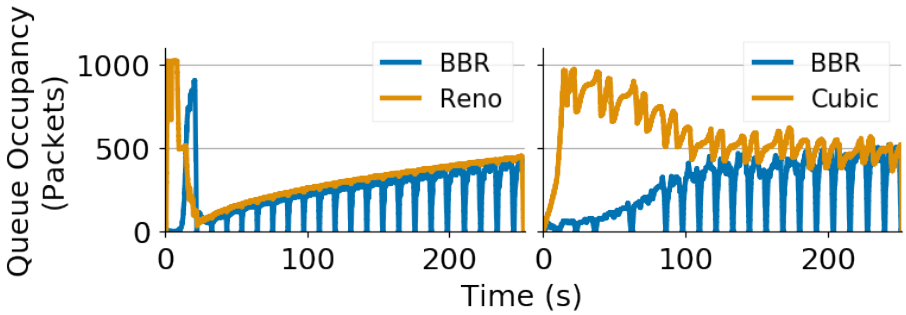
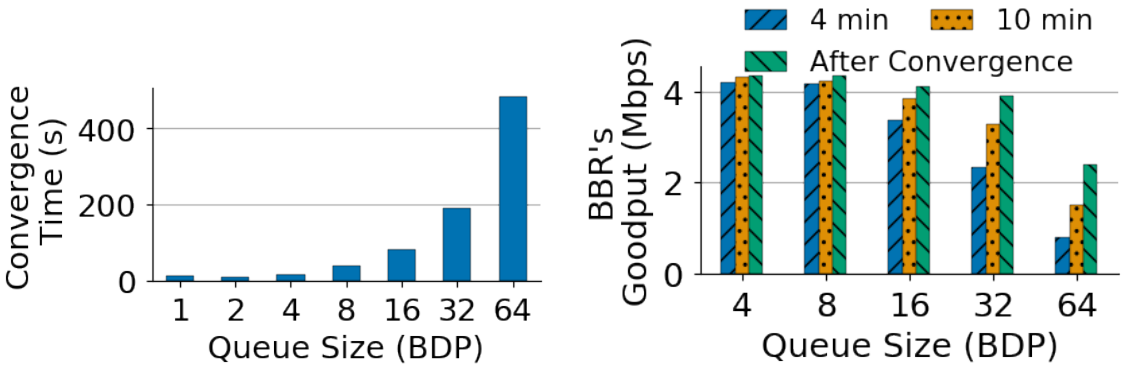


Figure 3.3: BBR and Cubic or Reno's queue when competing for 4 minutes over a network with a 64 BDP (1024 packet) queue.



(a) Convergence time for 1 BBR flow and 1 Cubic flow over varying queue sizes

(b) Goodput for 1 BBR flow and 1 Cubic flow over varying measurement intervals.

Figure 3.4: BBR vs Cubic in a 40ms × 10Mbps network

existing CCAs in typical buffer bloated networks, especially to Cubic.

Subsequent studies in our group and others questioned both the results – what fraction of the link BBR consumed – as well as the implication of generosity [124, 34, 104]. Some data [124] showed that BBR converged to different rates – around 40% of the link capacity for queue sizes up to 32×BDP, matching the Reno graph, but not matching the Cubic graph. We show in Figs. 3.3 and 3.4 that this incongruity is merely the result of differing experimental conditions and the amount of time it takes for BBR to converge to its steady-state share of link capacity. Where BBR quickly matches Reno's queue occupancy – and therefore consumption of the link capacity – BBR takes longer to scale up when competing with Cubic (Fig. 3.3). As a consequence, the 'average goodput' one computes is dependent on how long one measures the competition between BBR and Cubic (Fig. 3.4a). Furthermore, to reach convergence can take on the order of minutes in very deep buffered networks (Fig. 3.4b).

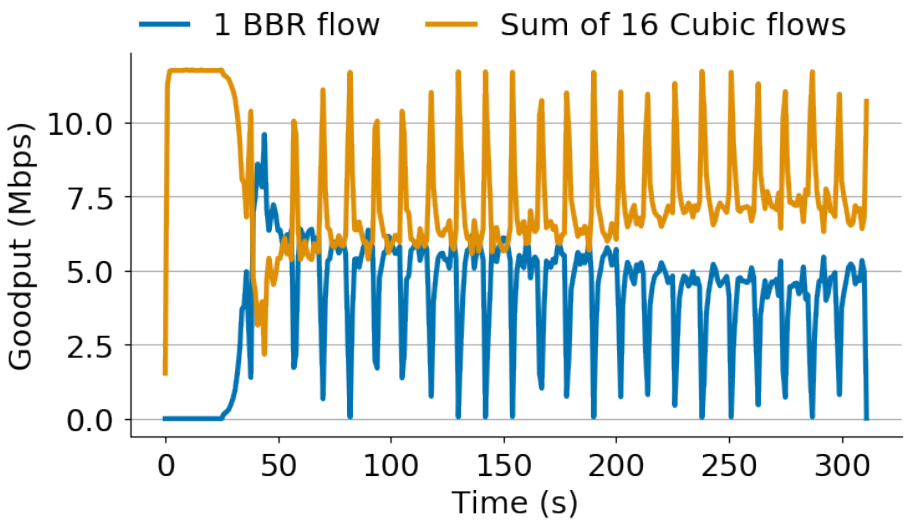


Figure 3.5: BBR's goodput over time competing with 16 Cubic flows in a 40ms×10Mbps network with a 32 BDP queue.

Another set of experiments [124, 34] suggest that BBR may consume far more than its 'fair' share of link capacity. Fig. 3.5 shows goodput over time of BBR vs 16 Cubic flows in the same 40ms × 10Mbps scenario. BBR consumes an outsized share of bandwidth, leaving just over half to be shared by the sixteen other connections.

Unfortunately, relying only on these empirical studies leave us like the blind men and the elephant, each relying on only pieces of the overall picture to understand BBR's characteristics. To get to the bottom of *why* BBR behaves in the way it does, and to *predict* how BBR might behave in unobserved scenarios, we turn to modeling in the rest of this chapter.

3.4 BBR Primer

BBR is designed to be a rate-based algorithm. BBR maintains two key variables: $Btlbw_{est}$ BBR's estimate of the available throughput for it to transmit over the network, and RTT_{est} BBR's estimate of the baseline round-trip time. BBR paces packets at $Btlbw_{est}$ rate. Assuming that BBR is transmitting over a single link with no queueing (and a sender which ACKs instantaneously), BBR should expect to never have more than $Btlbw_{est} \times RTT_{est}$ unacked packets outstanding.

As a failsafe and to keep the pipe full in networks that delay or aggregate ACKs, BBR implementations impose a 'in-flight cap' – it will never allow itself to have *more* than $2 \times Btlbw_{est} \times RTT_{est}$ unacknowledged packets outstanding [24, 26]. As we will show, this cap turns out to be the central parameter controlling BBR's link utilization in competition with Cubic and Reno.

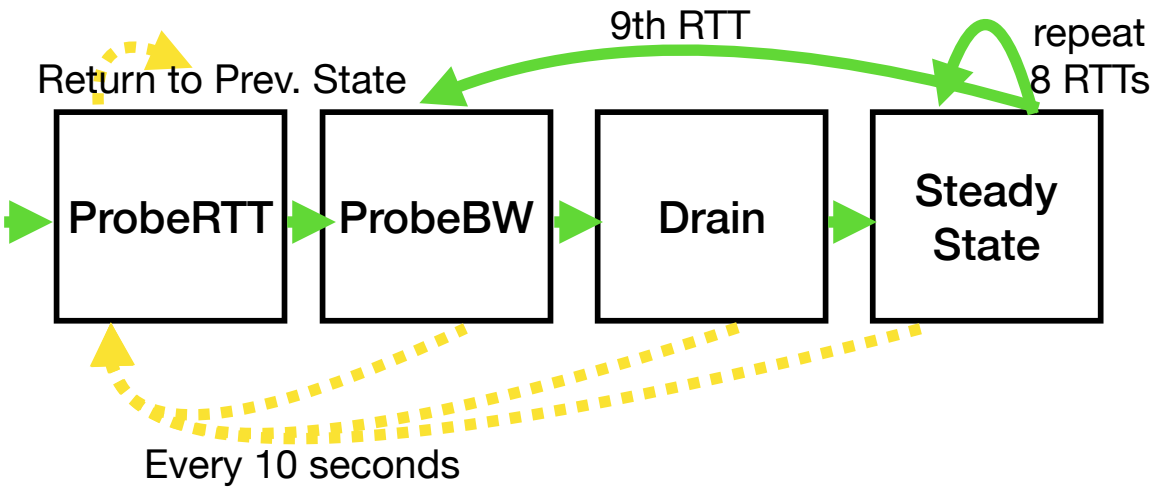


Figure 3.6: BBR's steady-state operation.

To estimate $Btlbw_{est}$ and RTT_{est}, BBR cycles (post-startup) through a simple state machine illustrated in Fig. 3.6.[1]

Estimating the rate. BBR sends at a fixed rate BW_{est}. BBR sets its initial rate using its own version of 'slow start'; henceforth BBR 'probes for bandwidth' (ProbeBW in Fig. 3.6) one out of every 8 RTTs. During this stage, BBR inflates the rate to $1.25 * Btlbw_{est}$ and observes the achieved throughput during that interval. BBR then lowers its rate to $(0.75 * Btlbw_{est})$ to drain any excess packets out of queues. BBR's $Btlbw_{est}$ is then the max observed packet delivery rate over the last 8 RTTs. It then sends at the newly-recalculated $Btlbw_{est}$ for the next 6 RTTs before probing again.

Estimating the RTT. BBR also keeps track of the smallest observed RTT. If BBR goes 10 seconds without observing a smaller RTT, it enters ProbeRTT. During ProbeRTT, BBR caps the amount of data it has in-flight to only 4 packets and measures the RTT for those packets for at least 200ms and one packet-timed round-trip. [2] BBR drops its sending rate to try to ensure none of its own packets are occupying queues in the network: in Fig. 3.5 one can observe BBR dropping its rate to almost zero on ten-second intervals. After ProbeRTT, BBR returns to the state it was in previously.

[1]Our state machine figure differs from the 'standard' BBR figure [21] by focusing on only steady-state operation rather than startup, and separating apart the three sub-phases of ProbeBW.

[2]A "packet-timed round-trip" means that a data packet is sent and then the sender waits for that packet or some late packet to be acknowledged

3.5 Analysis and Modeling

We model BBR's post-convergence share of link capacity when competing with loss-based CCAs in three phases.

(1) Simple Model of In-flight Cap: We first model a simple scenario to understand how BBR's in-flight cap controls BBR's sending rate. In this scenario, the queue is highly bloated, baseline RTTs are negligible, and there are only two flows (one BBR, one loss-based) competing.

(2) Extended Model of In-flight Cap: After demonstrating that BBR's in-flight cap controls its sending rate, we develop a more robust model, covering scenarios with multiple BBR flows, finite queue capacities, and non-negligible RTTs.

(3) Model of Probing Time: BBR's in-flight cap is only 4 packets during ProbeRTT, hence BBR spends time without transmitting data every ten seconds. To predict BBR's sending rate overall, we must reduce the rate predicted by the in-flight cap proportionally to the amount of time BBR spends in ProbeRTT.

3.5.1 Assumptions and Parameters

Table 3.1 lists the parameters in our model. We use these parameters to compute p, Cubic/Reno's share of the link capacity at convergence, and $1 - p$, BBR's share of link capacity at convergence. Our model is based on the following assumptions:

(1) Flows have infinite data to send; their sending rates are determined by their CCA, which is either BBR, Cubic, or Reno.

(2) All flows experience the same congestion-free RTT and the available link capacity is fixed.

(3) All packets are a fixed size.

(4) The network is congested and the queue is typically full; a flow's share of throughput hence equals its share of the queue.

(5) All loss-based CCA's are synchronized [107]. All BBR flows are synchronized [24]. All flows begin at the same time.

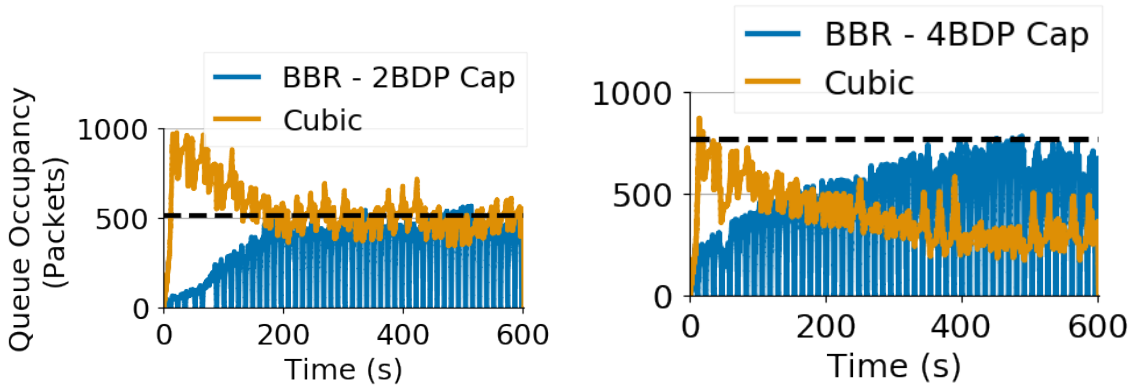
3.5.2 Simple Model: BBR's ProbeBW State

The first insight of our model is that BBR is controlled by its in-flight cap: in BBR's ProbeBW phase, BBR aggressively pushes out loss-based competitors until it reaches its in-flight cap.

Model: Why this happens follows from the BBR algorithm and loss-based CCAs' reaction to packet losses. Assume a link capacity c, where BBR and the loss-based CCAs, in aggregate, are consuming all of the available capacity. By probing for 125% of its current share of bandwidth, BBR pushes extra data into the network (offered load $> c$) leading

Parameter	Description
N	Number of BBR flows sharing bottleneck
q	Bottleneck queue capacity (packets)
c	Bottleneck link capacity (packets per second)
l	RTT when there is no congestion (seconds)
X	Queue capacity as multiple of BDP: $q = Xcl$
d	Flow completion times after convergence (seconds)

Table 3.1: Description of BBR model parameters



(a) 2 BDP in-flight cap

(b) 4 BDP in-flight cap

Figure 3.7: BBR vs Cubic in a 10Mbps×40ms testbed with a 32 BDP queue. Black dashed line is the model (3.5).

to loss for all senders. Loss-based algorithms back off, dropping their window sizes and corresponding sending rate. BBR does not react to losses and instead *increases* its sending rate, since it successfully sent more data during bandwidth probing than it did in prior cycles. The loss-based CCA returns to ramping up its sending rate, and together the combined throughput of the two becomes slightly *higher* than the link capacity and the two flows begin to fill the bottleneck buffer. This process continues until BBR hits an in-flight cap; we expect that in the absence of a cap it would consume the entire link capacity.

Validation: We modified BBR in our testbed to run with a in-flight cap of $4 \times BDP$. In Fig. 3.7b we show one run with our elevated in-flight cap along with a run with the standard cap in a testbed in a 40 ms × 10 Mbps network with a 32 BDP packet queue. BBR increases its share of the link capacity; we show in the next subsection that this increased share matches our prediction of a window-limited sender with a window the size of the in-flight cap.

3.5.3 Simple Model: BBR's In-flight Cap

To understand the impact of the in-flight cap on BBR's performance, we build a model making two simplifying assumptions (we relax these assumptions later): (1) There is only 1 BBR flow competing with any number loss-based CCAs, and (2) The queue capacity is much greater than the BDP ($q \gg cl$).

Model: Recall that the in-flight cap is calculated as:

$$\text{inflight}_{cap} = 2 \times RTT_{est} \times Btlbw_{est} \tag{3.1}$$

With a queue capacity of q we can assume that, at any given point of competition p from loss-based flows, BBR will consume the remaining bandwidth:

$$Btlbw_{est} = (1 - p)c. \tag{3.2}$$

About every 10 seconds, BBR enters ProbeRTT to measure the baseline RTT, draining any packets BBR has in the queue.

When there is no competing traffic, 1 BBR flow can successfully measure the baseline RTT l during ProbeRTT. When there is competing traffic from loss-based CCAs, there will be $p \times q$ data in the queue. Assuming a negligible baseline RTT ($q \gg cl$) — as bufferbloat increases, queuing delay becomes the dominant factor in latency — we have:

$$RTT_{est} = \frac{pq}{c}. \tag{3.3}$$

Plugging (3.2) and (3.3) into (3.1) and reducing gives:

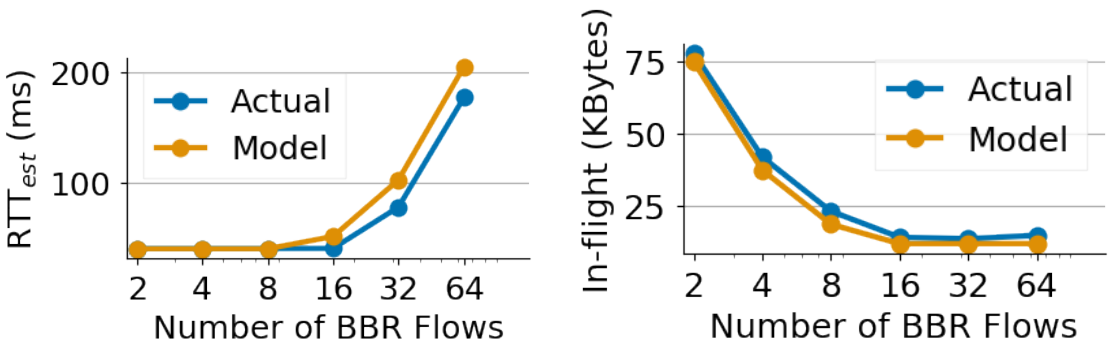
$$\text{inflight}_{cap} = 2(p - p^2)q. \tag{3.4}$$

We know from the previous subsection that BBR will increase its rate until it is limited by the in-flight cap. To compute this, we set inflight_{cap} equal to the amount of data BBR has in-flight and solve for p:

$$\begin{aligned} 2 \times (p - p^2)q &= (1 - p)q \\ p &= \frac{1}{2} \end{aligned} \tag{3.5}$$

We can now see that while 1 BBR flow increases its sending rate during ProbeBW, once it intersects the in-flight cap it will not be able to consume more than 50% of the available capacity.

Validation: This simple model for the in-flight cap in a deep-buffered network says if the BDP cap is 2, then BBR should occupy about half the queue after convergence. Similarly, if the BDP cap is 4, then BBR should occupy at most 75% of the queue after convergence.



(a) RTT$_{est}$. (b) Inflight$_{cap}$.

Figure 3.8: Comparisons between model and observation for RTT$_{est}$ and in-flight$_{cap}$ at 40ms × 15Mbps and 64 BDP queue.

Fig. 3.7 shows BBR converging at each of these points in a deep-buffered network with a 32 BDP queue.

Note: This simple model demonstrates why BBR retains the same share of link capacity regardless of the number of competing Cubic or Reno flows. ProbeBW is aggressive enough to force one or many loss-based flows to back off; the bandwidth cap is set simply by the queue occupancy of the competing loss-based flows – but not *how many* loss-based flows there are. The calculations behind ProbeBW and the in-flight cap lack any signal to infer number of competing loss-based flows and adapt to achieve equal shares/fairness.

3.5.4 Extended Model: In-flight Cap

Our simple model assumes a buffer-bloated network and only one BBR flow. In this section, we show how BBR's in-flight cap changes given the size of the queue (bloated or not) and with an increasing number of BBR flows.

Multiple BBR Flows Alone: To understand multiple BBR flows competing with loss-based flows, we first need to understand multiple BBR flows competing in the *absence of other traffic*. After convergence, each BBR flow has a *slightly* overestimated $Btlbw_{est}$ near their fair share: $\frac{1}{N} \times c + \epsilon$. The additional ϵ is – similar to our discussion in §3.5.2 – due to the aggression of ProbeBW. Here, BBR flows compete against each other; BBR uses a max() operation to compute $BtlBw_{\text{est}}$ over multiple samples of sending rates resulting in, usually, a slight *overestimate* of its fair share. While we ignore this ϵ in our modeling, its existence forces the aggregate of BBR flows to send at a rate slightly higher than c, filling queues until each flow reaches its bandwidth cap and becomes window-limited and subsequently ACK-clocked.

However, the cap may also be elevated due to the presence of multiple competing flows. During ProbeRTT, each flow will limit inflight to 4 packets, so that they can drain all of

their packets from the queue and measure the baseline RTT. For N BBR flows, this means in aggregate they will have $4N$ packets inflight. However, if $4N$ packets is greater than the BDP, the queue will not drain during ProbeRTT so RTT_{est} includes some queueing delay:

$$RTT_{est} = \max(l, \frac{4N - cl}{c} + l) \tag{3.6}$$

Thus, the the in-flight cap when N BBR flows compete is dependent on the BDP. Further, if the queue is smaller than $4N - cl$ when $4N > cl$, then the BBR flows will consume the entire queue and hence 100% link capacity.

Validation: Fig. 3.8a shows the measured median RTT estimate across a varying number of BBR flows versus (3.6). The estimate increases linearly, similar to our prediction. Here, the BDP is only 75KB, so the queue will not completely drain during ProbeRTT when there are 13 or more BBR flows. Fig. 3.8b shows how this corresponds to the inflight cap. If the BDP were larger, the flows would have been able to measure the correct $RTTest$.

Multiple BBR Flows vs Loss-Based Flows: We now return to multiple BBR flows vs loss-based flows. As we saw when BBR flows were only competing with each other, if the BDP is not large enough to accommodate $4N$ packets during ProbeRTT, BBR's RTT estimate will be too large. If we assume $4N$ additional packets are in the queue during ProbeRTT, then,

$$RTT_{est} = \frac{pq + 4N}{c} + l. \tag{3.7}$$

Here, we also include l, no longer assuming it is negligible compared to queueing delay. Plugging (3.7) and (3.2) into (3.1), in aggregate all N BBR flows will have:

$$\text{inflight}_{cap} = 2(1-p)c\left(\frac{pq + 4N}{c} + l\right). \tag{3.8}$$

To compute the BBR flows' aggregate fraction of the link, we set inflight_{cap} equal to the amount of data BBR flows have in-flight and solve for p:

$$\begin{aligned} 2(1-p)c\left(\frac{pq + 4N}{c} + l\right) &= (1-p)q + (1-p)cl \\ p &= \frac{1}{2} - \frac{1}{2X} - \frac{4N}{q} \end{aligned} \tag{3.9}$$

If p were a negative number, this would mean BBR's inflight cap exceeded the total capacity (BDP + the queue size) and hence BBR's share of the link would be 100%.

In the next section, we complete our extended model by computing the amount of time BBR operates at its in-flight cap.

3.5.5 Extended Model: ProbeRTT Duration

During ProbeRTT, BBR stops sending data while it waits for its in-flight data to fall to 4 packets. You can see this behavior impacting goodput in Fig. 3.5. If the queue is large and also full when BBR goes into ProbeRTT, this results in long intervals where BBR is not sending any data. [3] This results in BBR on average consuming a lower fraction of link capacity than if it were sending constantly at a rate proportional to its inflight cap.

Model: If the total duration of time the flows are competing (after convergence) is d, the fraction of the link BBR flows will use when competing with loss-based CCAs is:

$$BBR_{frac} = (1-p) \times \left(\frac{d - \text{Probe}_{time}}{d} \right), \tag{3.10}$$

where p is computed using (3.9). During Probe_{time} throughput is nearly zero.

We compute Probe_{time} by computing the length of time spent in ProbeRTT state, and multiply by how many times BBR will go into ProbeRTT state. Assuming the queue is full before BBR enters ProbeRTT state, BBR will have to wait for the queue to drain before its data in-flight falls to 4 packets. Once it reaches this in-flight cap, BBR also waits an additional 200ms and a packet-timed round trip before exiting ProbeRTT. Assuming synchronized flows and the queue is typically full, BBR flows should rarely measure a smaller RTT outside of ProbeRTT state so it should enter ProbeRTT about every 10 seconds. Altogether, this means probe time increases linearly with queue size:

$$\text{Probe}_{time} = \left(\frac{q}{c} + .2 + l \right) \times \frac{d}{10} \tag{3.11}$$

Validating Probe_{rtt}: First, we measure the probe time from experiments with competing BBR flows in for a 40ms×15 Mbps network for experiments run for 400 seconds after convergence (d=400) for Cubic We compare this to our prediction computed from (3.11). Fig. 3.9 compares (3.11) to measured probe time—the model fits the observations well. Most commonly the predicted probe time for experiments with Cubic is 1-3 seconds larger than the expectation and is at most about 8 seconds too large.

Validating the Extended Model: We measure the average throughput for BBR competing against Cubic or Reno after convergence (d = 400 for Cubic, d = 200 for Reno). We use (3.10) to compute BBR's expected fraction of the link versus our measurements. Our expectations closely follow empirical results in most cases, validating our model. Fig. 3.10 compares (3.10) to the BBR flows aggregate fraction of the link when competing with Reno or Cubic. The median error competing against Cubic 5%, and against Reno 8%.

[3] In fact, BBR authors have even noted that this is a significant limitation on BBR's performance, and in BBRv2 design change ProbeRTT so that it reduces BBR's inflight cap to 50% of it's BDP_{est} instead of 4 packets [22].

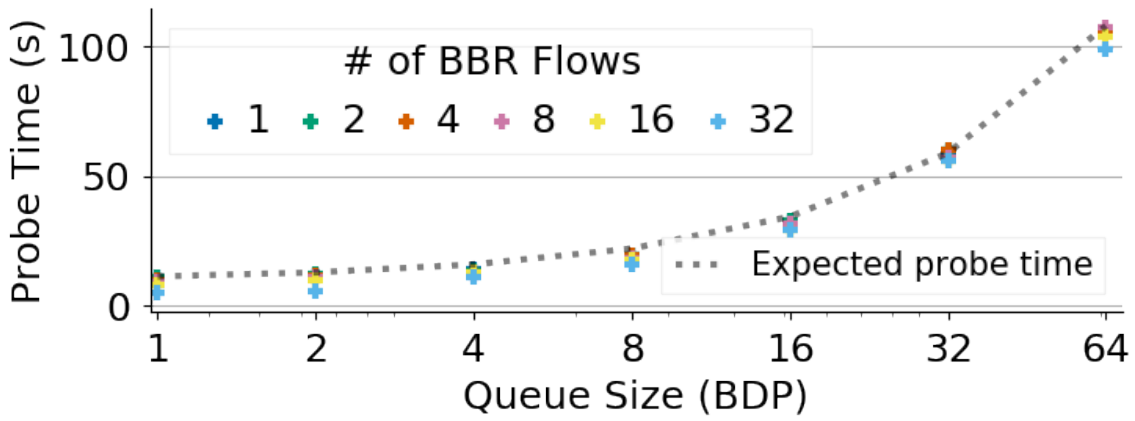
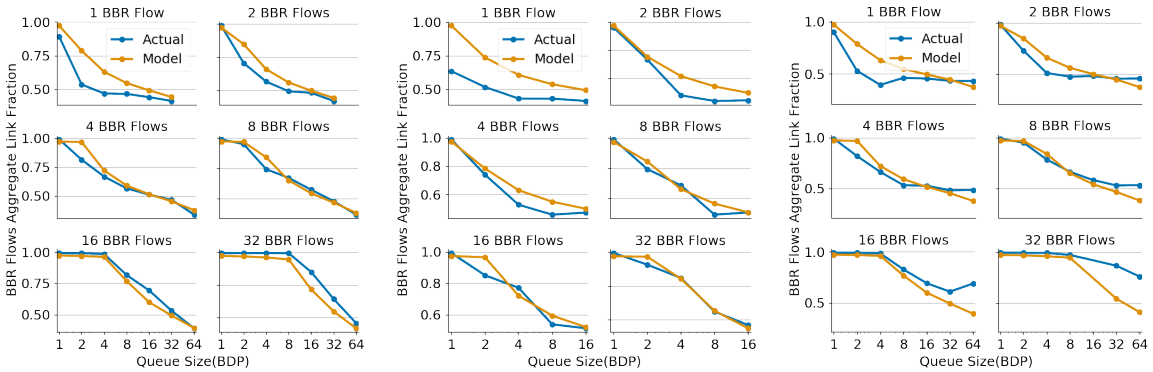


Figure 3.9: Probe$_{time}$ model for 40ms × 10 Mbps link vs. measured probe time for BBR flows competing with 1 Cubic flow in varying queue sizes.



(a) 40ms × 10 Mbps, vs 1 Cubic Flow

(b) 30ms × 50 Mbps, vs 1 Cubic Flow

(c) 40ms × 10 Mbps, vs 1 Reno Flow

Figure 3.10: Model compared to observed aggregate fraction of the link.

For Cubic, the model fits the observations best with large queue sizes and large numbers of flows. In this case, our assumptions that the queue is typically full, and 4N BBR packets will be in the queue during ProbeRTT, inflating RTT_{est}, are more likely to be true. However, Reno reveals an opposite trend: the model does *worse* as the queue becomes larger. We suspect this is due to Reno's slower (relative to Cubic) additive increase failing to take advantage of the available capacity and hence leaving a larger share of throughput for BBR.

3.6 Related Work

The first independent study of BBR was presented by Hock et al. [55]. Their analysis of BBR identifies the important property that multiple BBR flows operate at their in-flight cap

in buffer-bloated networks. Further, they present experiments for 1 BBR flow and 1 Cubic flow, noting that in large buffers, they oscillate around equally sharing the bottleneck. They also observe that when 2 BBR flows compete with 2 Cubic flows in a shallow-buffered network, BBR flows will starve the Cubic flows. Several additional empirical studies have reproduced and extended these results [104, 34, 119]. Scholz et al. [104] run tests for up to 10 BBR flows competing with up to 10 Cubic flows in a large buffer and conclude that, "independent of the number of BBR and Cubic flows, BBR flows are always able to claim at least 35% of the total bandwidth." Dong et al. [34] also note that as 1 BBR flow competes with an ever increasing number of Cubic flows, BBR's fraction of the bandwidth remains the same.

Each of these studies touches on important aspects of BBR's behavior, but we are the first to model BBR's behavior in these scenarios rather than to simply observe it. Through our model, we are able to explain the missing parts of seemingly conflicting conclusions drawn in prior work.

Google is actively developing BBRv2 and very recently released a Linux kernel implementation of BBRv2 [22, 23, 12]. Early presentations [23] imply that it primarily resolves the fairness issues discussed by Hock et al [55], but does not touch on the fixed proportion of link capacity as discussed in this chapter.

3.7 Conclusion

In this chapter, we have shown that BBR's inflight cap – a 'safety mechanism' added to handle delayed and aggregated ACKs – is in reality central to BBR's behavior on the Internet. When BBR flows compete with other traffic (BBR, Cubic, or Reno), BBR becomes window-limited and ACK-clocked, sending packets at a rate entirely determined by its inflight cap.

When competing with loss-based TCPs such as Cubic and Reno, BBR's cap can be computed using the bottleneck buffer size, the number of concurrent BBR flows, and the baseline network RTT. However, the number of competing loss-based flows *are not* a factor in computing this cap. Hence, BBR does not reduce its sending rate even as more loss-based flows arrive on the network. This is the cause of reports arguing that BBR is 'unfair' to legacy TCPs.

While we were able to use modeling in this chapter, this is not a general methodology for studying the interactions between CCAs. In the next two chapters we revisit how CCAs are evaluated for deployability on the Internet today and describe a new metric, methodology, and tool for this evaluation. While we were able to use modeling in this chapter, we discuss why this is not a general methodology and how we can improve empirical m

Chapter 4

Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms

> *It's a timely and thoughtful paper, there is an explosion of CCAs to handle the variety of workloads that have different connectivity requirements. Often however there is no clear means to decide whether to adopt one or not, and this paper can instigate a lively and timely debate.*
>
> Reviewer #5

4.1 Introduction

In Chapter 2 we measured the deployment of new CCAs both known and unknown. Then in Chapter 3 we both measured and modelled unfairness between recently deployed CCA BBR and legacy loss-based CCAs. This state of affairs leads to the last question in this dissertation: **How should we evaluate inter-CCA interactions to decide if a CCA is deployable?** Without a standard *deployment threshold*, we are left without foundation to argue whether a service provider's new algorithm is or is not overly-aggressive.

A deployment threshold concerns inter-CCA phenomena, not intra-CCA phenomena. Rather than analyzing the outcomes between a collection of flows, all using some CCA α, we need to analyze what happens when a new CCA α is deployed on a network with flows using some legacy CCA β. Is α's impact on the status quo is acceptable?

Our community has traditionally analyzed inter-CCA competition in two ways, which we refer to as 'fairness' and 'mimicry.' While both approaches are insightful, we argue that neither is a sound basis for a deployment threshold.

A throughput allocation is fair if it maximizes every users utility function given limited link capacity [110]. A end-host CCA, typically defines users as flows, aiming to maximize utility per-flow by ensuring that every flow sharing the same bottleneck link gets equal bandwidth. For example, CCA designers try to argue their CCA α is deployable if it is fair to Cubic (β), the default CCA in Linux [130, 33, 35, 6, 26, 22, 27]. However, a fairness-based deployment threshold suffers from three key issues:

(1) *Ideal-Driven Goalposting*: A fairness-based threshold asserts a new CCA α should equally share the bottelneck link with currently deployed CCA β. In practice, this goal is too idealistic to achieve in practice. The end result is that ideal-driven goalposts are simply ignored as impractically high requirements. For example, CCA designers have argued that it is acceptable to be unfair to Cubic because Cubic is not even fair to itself [6].

(2) *Throughput-Centricity*: A fairness-based threshold focuses on how a new CCA α impacts a competitor flow using CCA β by focusing on β's achieved throughput. However, this ignores other important figures of merit for good performance, such as latency, flow completion time, or loss rate.

(2) *Assumption of Balance*: Inter-CCA interactions often have some bias, but a fairness metric cannot tell whether the outcome is biased *for* or *against* the status quo. It makes a difference in terms a deployability whether a new CCA α *takes* a larger share of bandwidth than a legacy CCA β or *leaves* a larger share for β to consume: the former might elicit complaints from legacy users of β, where the latter would not. Jain's Fairness Index [62] assigns an equivalent score to both scenarios.

Mimicry is a separate approach where new algorithms replicate properties of TCP-Reno (*e.g.*, driving throughput as a function of the loss rate [88]) in order to be 'friendly' to legacy, Reno-derived TCPs. The issue with mimicry is that it binds new algorithms to repeating the often undesirable idiosyncrasies of Reno, stifling improvement and evolution [71]. We discuss the drawbacks of fairness and mimicry as the basis of a deployment threshold further in §4.2.

We advocate instead for a deployment threshold based on *harm*. Harm allows us to speak in quantifiable, measurable terms about the impact of deploying a new CCA to the Internet. One can use measurements or models to determine that, in the presence of a competing flow using CCA α, a flow using a CCA β suffers, *e.g.* a 50% reduction in throughput or a 10% increase in latency. We refer to this degradation as the harm.

Perhaps the most crucial aspect of harm is recognizing that we are not designing a clean-slate Internet. We believe that we need to shift our focus from if pairs of CCAs 'fairly' share and instead focus on how a new CCA impacts the *status quo* (whether or not the new algorithm damages the performance of existing traffic). We argue that our deployment threshold should be based on the amount of harm *already caused* by deployed algorithms.

If the amount of harm caused by flows using a new algorithm α on flows using an algorithm β is within a bound derived from how much harm β flows cause other β flows, we can consider α deployable alongside β.

Turning this insight into a concrete threshold is challenging; we present three approaches in §4.4. Nonetheless, we believe that a harm-based threshold is the right way forward. A harm-based threshold avoids ideal-driven goalposts like fairness by settling for outcomes that are unfair, but no worse than the status quo. A harm based-threshold does not suffer from the limits of throughput centricity. We can speak of throughput-harm as well as latency-harm, FCT-harm, or loss-harm. Lastly, a harm based-threshold prescribes no mechanism or behavior to replicate, which allows for a broader range of outcomes than mimicry.

In what follows, we discuss the limitations of fairness and mimicry in §4.2. We then introduce harm in §4.3, how to quantify it, and our intuition as to why a harm-based threshold is the right path forward for the Internet. In §4.4 we propose three possible harm-based thresholds. Finally, in §4.6, we leave open questions for the community and conclude.

4.2 Fairness and Mimicry

Our goal in this chapter is to identify a *deployment threshold*: a bound on the behavior of a new CCA when competing with legacy CCAs. If a new CCA meets the conditions of the threshold, we ought to consider it deployable. Furthermore, if a new CCA *does not* meet the conditions, it should be considered unacceptable for deployment. In this section, we discuss the limitations of prior approaches to evaluating new CCAs and their interactions with other algorithms. We argue that both fairness(§4.2.1) and mimicry based approaches (§4.2.2) are unsuitable for a deployment threshold. Through our discussion, we derive a set of desiderata for a deployment threshold, which we list in Table 4.1.

4.2.1 Limitations of Fairness

Fairness measures are the typical tool used for determining if a new CCA is deployable on the Internet [35, 6]. A fairness-based threshold, asserts if a CCA α is fair to a legacy CCA β, then the algorithm is deployable. Fairness is typically measured by looking at the throughput ratio between competing CCAs or by computing Jain's Fairness Index (JFI) [62], which returns a number between 1 (perfectly meeting the expected fair allocation) and 0 (the closer to 0, the more 'unfair').

Typically, fairness is measured assuming infinitely backlogged flows: each flow wants to use an equal fraction of the bottleneck link. In this case, we expect the throughput ratio and Jain's Fairness Index to be 1. In reality, not all flows are infinitely backlogged and not all flows can fully utilize their equal share. In that case, equal rate fairness has a well-known shortcoming: it does not account for the *demand* of each flow. Demand is the amount of resources a flow uses when operating in absence of contention. Consider

Demand-Aware	Like max-min fairness, takes into account the fact that some flows have different demands than others.
Multi-metric	Addresses throughput, latency, flow completion time, or any other performance metric.
Practical	Practical, rather than ideal-driven (unlike fairness); it should be feasible for new CCAs to meet this threshold.
Status-Quo Biased	Does not suffer from the assumption of balance. Specifically, we should worry about the impact of a new CCA on the currently deployed CCAs, and should not focus on how deployed CCAs harm a new CCA.
Future-Proof	Useful on a future Internet where none of today's current CCAs are deployed; does not restrict the development of new CCAs based on the idiosyncrasies of currently deployed CCAs.

Table 4.1: Desiderata for a deployment threshold, derived from insights and shortcomings of fairness and mimicry.

a new CCA α competing against a TCP NewReno flow on a 10Gbps link. We know that the NewReno algorithm will fail to take advantage of the full link capacity due to its slow additive increase and aggressive reaction to loss [41].

Intuitively, it would seem that a new flow using α should be able to take advantage of the remainder of link capacity, but equal rate fairness disallows such an outcome. Including demand is important for a deployment threshold: a new CCA should not be penalized as 'unfair' to a legacy CCA when the legacy CCA, on its own, is incapable of claiming its equal share of the network. Hence, we list 'Demand-Aware' as our first item in Table 4.1. Unlike equal rate fairness, max-min fairness, allows for flows to increase its rate if it would not decrease the rate of any other flows [110]. Thus, when we refer to fairness throughout this chapter, we refer to max-min fairness.

Although we argue against a fairness-based deployment threshold, fairness measures have many practical uses in the design of CCAs and scheduling systems. In this section, we do not attack fairness as a valued measure for systems in general. In particular, we believe throughput fairness is sometimes a desirable property, especially for intra-CCA interactions. Nonetheless, we object to using fairness measures as a threshold for determining CCA deployability for the reasons we discuss as follows.

4.2.1.1 Throughput-Centricity

Fairness strategies focus on sharing a resource like 'dividing a pie'. This is appropriate for performance metrics like throughput since there is a maximum link capacity which must be divided by all competing flows. However, modern CCA designers consider many performance metrics which do not always easily map to a network resource which should be divided or shared, *e.g.*, latency, loss rate, flow completion time.

Consider a future Internet where the majority of Internet services use a TCP algorithm called TINYBUFFER which, like algorithms BBR and Vegas, has very little queue occupancy and therefore low latency and loss. Hence, TINYBUFFER provides very good user experience for video conferencing and voice calls. A new company wishes to introduce a new algorithm α, which is derived from Cubic and therefore fills buffers. Is this an acceptable deployment? Videochat users of TINYBUFFER would likely say no, since α flows competing with TINYBUFFER would increase latency and loss, harming their video calls.

Unfortunately, we cannot say that the behavior of α relative to TINYBUFFER is 'unfair' – buffer occupancy is not a resource we want to divide equally. Instead, it is a value we simply want minimized which is not captured by the concept of fairness. For this reason, we say that fairness is not 'multi-metric', our second requirement in Table 4.1.

4.2.1.2 Ideal-Driven Goal Posts

A second problem with fairness is that, even when we focus on throughput alone, it is simply very difficult to achieve. For example, Cubic and Reno, both known to have a 'short flow penalty' where short flows do not achieve their fair share of throughput before completing [73, 85]. BBR is also unfair to connections with shorter RTTs, allocating them lower throughput than competing connections with long RTTs [25].[1] If algorithm designers cannot achieve perfect fairness in the intra-CCA context, why would it make sense to expect algorithm designers to achieve perfect fairness in the more challenging inter-CCA context? We list our third requirement in Table 4.1 as being 'practical.'

Many readers at this point may find themselves thinking, 'Of course we don't expect new algorithms to be *perfectly* fair to existing ones!' But, even if we do not expect perfect fairness, the community still leaves algorithm designers with no real guideline for acceptability based on fairness. This often results in CCA designers making the argument that it is acceptable for their algorithm to be somewhat unfair to legacy CCAs. [6, 33, 35]. Nonetheless, we lack a practical threshold and clear consensus on how far from perfectly fair sharing a new algorithm may be permitted to deviate.

4.2.1.3 The Assumption of Balance

We call our third and final objection to fairness the Assumption of Balance, meaning that it values the performance outcome of *both* the new CCA and the existing, deployed CCA.

To illustrate our objection, we look to Figure 4.1. We imagine a future Internet where most senders use some algorithm β; two senders are transmitting very large files over a 100Mbps link. Sender B is using β, but sender A is using some brand new algorithm α. Both senders' demands are 100Mbps – both desire to use as much capacity as they can. However, sender B achieves 90Mbps throughput and sender A only achieves 10Mbps. Is this fair? No. In both the above scenario and a scenario where the allocations are swapped

[1] Reno's throughput allocation has the opposite bias.

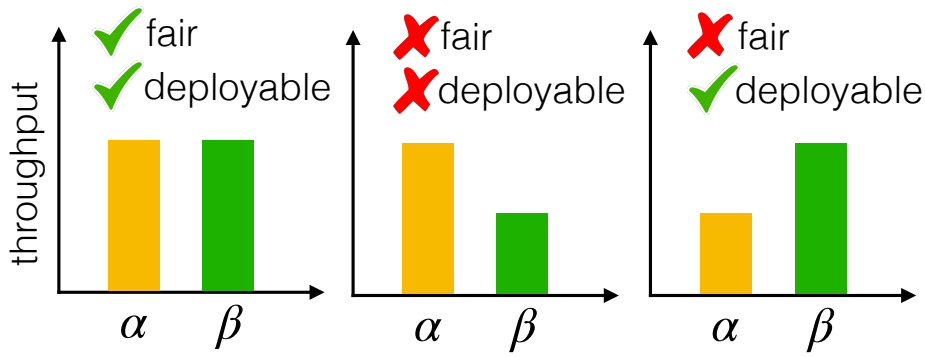


Figure 4.1: Fairness and the assumption of balance.

(B receives 10Mbps, A receives 90Mbps) have the same JFI – 0.61. But, it should be perfectly acceptable to deploy α *if* α *is the one receiving unfair treatment* – no users other than user A, who chose to use α, would be impacted negatively.

It is highly unlikely that many new services will set out to deliberately deploy algorithms that penalize the performance of their own flows – but given the difficulty of achieving perfect fairness (§4.2.1.2) it is likely that this outcome may happen in some scenarios.[2] A very deployable, friendly algorithm would err on the side of harming their own connections where a more aggressive algorithm would err on the side of harming those of others – and a good measure of deployability should be able to distinguish between the two. Thus, our fourth requirement in Table 4.1 is that the new threshold be 'status-quo biased.'

4.2.2 Limitations of Mimicry

A mimicry-based threshold asserts if a CCA α mimics the behavior of a legacy CCA β, then the algorithm is deployable. Two mimicry based approaches are:

TCP-Friendly Rate Control (TFRC) [88]: a CCA using TCP-Friendly Rate Control transmits at a rate $\leq \frac{MSS}{RTT*\sqrt{p}}$ for p the link loss rate; this formula describes TCP Reno's average sending rate over time [87, 80].

RTT-biased Allocation [40]: a CCA obeying RTT-biased Allocation grants more throughput to connections with low RTTs than to those with higher RTTs; this behavior is a property of TCP Reno.

Mimicry introduces an elegant solution to the challenge of ideal-driven goal posts: it should be acceptable to deploy a new CCA which introduces the same side-effects – fair or unfair – as the already deployed algorithm. A mimicry-based approach is always practical

[2]Consider Google's BBR, which, when one BBR flow competes with one Reno flow will only consume 40% of link capacity – less than its fair share [25, 126]. Furthermore, 'background' CCA algorithms, such as TCP-Nice [122] and LEDBAT [106] deliberately only consume leftover bandwidth that is underutilized by other connections.

because the existence of the original, deployed algorithm demonstrates at least one way to achieve these performance outcomes (although as TFRC illustrates, a CCA may use a different algorithm than the original to achieve this outcome).

However, mimicry will not serve as a good threshold because it binds new algorithms to replicating the often undesirable idiosyncrasies of the deployed CCA, and hence stifles improvements and evolution. For example, TFRC limits new CCAs from achieving high throughput. Indeed, an animus for most of the new CCAs which are replacing Reno on today's Internet (*e.g.* Cubic [49], Compound [112]) was to supersede the $\leq \frac{MSS}{RTT*\sqrt{p}}$ rate, as this very limit is what prevents Reno from taking advantage of high-capacity links.

Similarly, RTT-biased Allocation is not an ideal outcome that was proposed from first principles: it is simply the throughput allocation that Reno achieves. Given this, perhaps it should be acceptable, on an Internet dominated by RTT-biased algorithms, to deploy yet another RTT-biased algorithm – but RTT-bias should not be enshrined as the *goal itself.*[3]

In a future Internet where no one any longer deploys Reno variants, a mimicry-based threshold would lack grounding; even worse, it could prevent us from reaching a future Internet with improved fairness, lower latency, *etc.* due to our replicating the inherent limitations of existing CCAs. Thus, our final requirement in Table 4.1 is that our threshold be 'future-proof.'

4.3 Harm

We now present harm, and argue that a harm-based threshold would meet all of our desiderata. In the next section (§4.4), we present a few possible harm-based thresholds.

4.3.1 Calculating Harm

We imagine a TCP connection where Alice is video conferencing with her friend Bob. When running alone, the connection achieves 15Mbps of throughput, packets arrive with 40ms latency, and jitter is 10ms. When Alice's roommate Charlie starts a large file transfer, Alice's video conference connection drops to 10Mbps of throughput, latency increases to 50ms, and jitter increases to 15ms. Since all of these performance metrics became worse due to to Charlie's connection, we say that Charlie's connection caused *harm* to Alice's connection.

We can measure harm by placing it on a $[0, 1]$ scale (much like Jain's Fairness Index [62]) where 1 is maximally harmful, and 0 is harmless. Let x = demand (solo performance); let y = performance after introduction of a competitor connection. For metrics where 'more is better' (like throughput and QoE) harm is $\frac{x-y}{x}$. For metrics where 'less is better' (like

[3]Some will disagree with us, arguing that longer flows consume more network resources and therefore *should* receive lower throughput [40] – and call this 'RTT Fairness.' This could be a good reason to continue with RTT-biased allocation. Our point is that, if we really prefer Max-Min fairness, it would be bad to continue with RTT-biased allocation simply because we have required ourselves to mimic Reno's behavior.

latency or flow completion time) harm is $\frac{y-x}{y}$. On this scale, Charlie caused 0.33 throughput harm, 0.2 latency harm, and 0.33 loss harm.

The amount of harm done by one TCP connection to another depends not only on the algorithm(s) in use, but also the *network* and *workload*. For example, a connection with a very bursty traffic pattern will induce higher worst-case latency for competitor connections than a connection with very steady, paced traffic (a change in workload). Throughput harm between TCP Cubic and BBR varies depending on whether the network is shallow or deeply buffered [25] (a change in network). Note that we also include background traffic in our model of the 'network'.

Harm is multi-metric. It can be computed for latency, throughput, jitter, or any other quantifiable measure.

Harm is demand-aware. Harm is computed from a baseline of a TCP connection running on its own; new algorithms are not penalized when deployed algorithms perform poorly due to their own limitations. A flow which only ever consumes 10% of link capacity has no throughput harm done to it when another flow arrives and consumes the remaining 90%. A flow using a CCA which occupies all of the buffer space has no latency harm done to it when another buffer filling algorithm competes with it.

Harm is status-quo biased. Harm does not suffer from the assumption of balance because it does not involve the performance of connections using the new algorithm at all. Harm only measures the new connection's impact on existing flows.

In the next section (§4.3.2), we provide the intuition behind how to derive a threshold from harm, and why such a threshold would be practical and future-proof. Then (§4.4) we discuss several proposals for a threshold based on harm.

4.3.2 A Harm-Based Threshold

Simply measuring harm does not tell us whether or not the harm introduced by a new algorithm is acceptable. We take inspiration here from TCP mimicry (§4.2.2): the behaviors of deployed algorithms today should be our guidelines for what is acceptable. However, we need to relax mimicry to allow innovation and improvement. We suggest that, *if the harm done by a new CCA α to an widely deployed CCA β is comparable or less than the harm done when β competes against β, we should consider it acceptable to deploy.* We say that the harm of the *alpha* upon *beta* is *bounded* by the harm already done by *alpha* to itself. We leave up to discussion how 'widely deployed' a β must be in order to merit protection under our deployment threshold.

For example, a new algorithm α is developed for a future Internet where the predominant algorithm is called K-LATENCY. Every flow in K-LATENCY maintains a constant queue occupancy of exactly k packets – thus, latency increases linearly with the number of competing flows. A harm based threshold, grounded in how K-LATENCY connections interact in the intra-CCA scenario, would deem α acceptable with respect to latency so long as it never buffers more than k packets per flow.

The difference between bounded harm and mimicry can be subtle. In the above example, mimicry would demand that α always buffer exactly k packets per flow, just as K-LATENCY does (restricting improvement). On the other hand, bounded harm allows α to buffer *up to* k packets while competing with K-LATENCY (allowing for improvement, if feasible). Bounded harm allows for a broader range of outcomes than mimicry. Further the bound for α is undefined when not competing with K-LATENCY – α flows may behave differently when competing with other α flows or competing with some third CCA γ. For example, 'modal' algorithms like Copa [6], exhibit very diverse behaviors within one CCA, adjusting behavior online as different cross-traffic is detected. But, an algorithm need not be explicitly modal to have bounded-harm outcomes that are acceptable across multiple CCAs.

On an Internet with many competing algorithms $\beta, \gamma, \phi \ldots$ (as today's Internet) one might ask why we are bounding harm from a new α to the harm that each algorithm does to itself. We suggest that α do harm to β flows that is bounded by the harm caused to β by other β flows, and that α do harm to γ flows that is bounded by the harm caused to γ by other γ flows. Why not bound α's behavior in the harm done by β to γ and vice versa?

One reason we reject this approach is that on today's Internet, there are many well-known pessimal cross-CCA outcomes (*e.g.*, BBR's starving Cubic on high capacity links [56]). A CCA designer for a new CCA α could use the existence of any single pessimal scenario to justify continuing that behavior with α. By bounding harm to a CCA by the harm *any* other CCA, we settle for the absolute lowest common denominator in performance outcomes.

Furthermore, bounding the inter-CCA harm (caused by a new α on an existing β) by the intra-CCA harm (caused by β to β) carries forward the design trade-offs made by CCA developers. CCA designers tune their algorithms for the outcomes they want under intra-CCA competition. The designers of Reno and Cubic allow the loss rate to increase with the number of competing Reno/Cubic flows [80]. The designers of BBR aim to keep buffers empty, but do allow the queue occupancy to increase when multiple BBR flow compete [54]. In this way, the designers implicitly encode their tolerance to performance degradation for each metric. As a consequence, bounding inter-CCA harm by the intra-CCA harm means that new CCAs will respect the implicitly expressed preferences of typical traffic.

In practice, we already see CCA designers try to make the argument that their algorithm is deployable because it is not any more aggressive towards the status-quo (Cubic) than it is to itself [6, 22, 35, 33]. Unforunately, they try to make this argument using fairness, which suffers from the limitations dicussed in §4.2.1. We believe explicitly measuring harm would give CCA designers a more clear language to argue for deployability. The networking community need only agree on concrete harm-based threshold to provide a guideline for how much harm is allowable.

Unlike fairness, a harm-based threshold is practical. The original algorithm is an

existence proof that the demanded threshold is feasible and hence the goal posts *cannot* be set too high.

Unlike mimicry, a harm-based threshold is future-proof. A harm-based threshold does not require replicating the behavior of a deployed algorithm – only matching or improving upon its outcomes while competing with that algorithm. Furthermore, as CCAs die out in popularity or new ones arise, a harm-based threshold will shift to requiring similar harm to the new algorithms rather than the old ones.

The reader may wonder how an algorithm can be both status-quo biased and future proof. Consider the bi-modal Copa algorithm [6]. Unlike a mimcry approach, Copa only seeks to match the throughput of loss-based algorithms when it detects its competition is loss-based. When Copa is alone, it behaves like a delay-based algorithm, minimizing excess queueing. So, in a world, where legacy algorithms are nearly phased out, Copa will be able to behave differently than legacy loss-based algorithms. Further, if Copa were to become widely deployed, subsequent algorithms would then measure harm against Copa's delayed-based behavior.

Deriving a concise, usable harm-based threshold is challenging. So far, we have only described how to measure harm and why a harm-based threshold is overcomes the limitations of fairness and mimicry. However, we have not yet defined a concrete deployment threshold based on harm. We believe a concrete threshold should consider how harm plays out between existing algorithms. In the next section, we propose several harm-based thresholds – and invite the community to scrutinize and improve upon them.

4.4 Concrete Thresholds

We now discuss several options for a concrete threshold, driven by our intuition from §4.3. We define the following variables and functions to help discuss each:

Let a TCP connection ('flow') $f = (a, w)$ for $a \in A$, the set of all congestion control algorithms, and $w \in W$, the set of all connection workloads (short flows, video streams, *etc.*).

Let M be the set of all performance metrics (throughput, latency, QoE, jitter, *etc.*).

Let N be the set of all network paths (with varying throughput, latency, loss rate, queue capacity, and background traffic).

Let $\textsc{harm}(f_i, f_j, n \in N, m \in M)$ be the harm done to f_i by f_j according to metric m in network n as defined in §4.3.

Note, we assume to compare the harm across workloads, you must compare them using harm functions with the same metric. Because of our status-quo bias, the harm function should be determined by the workload and CCA β. For example, if we assume β is adaptive bitrate video using BBR, the harm function might use some metric for quality

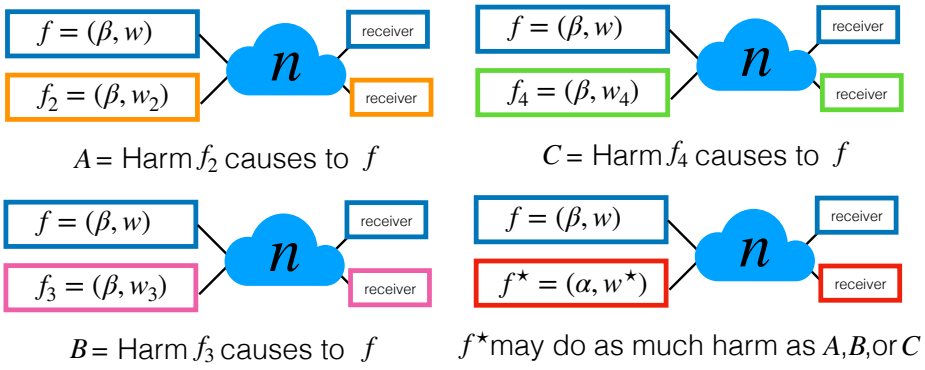


Figure 4.2: Under Worst Case Bounded Harm, a new α may do as much harm to a β flow f as *any* other β flow in the same network.

of experience like rebuffering rate.

4.4.1 Worst Case Bounded Harm

We illustrate our first potential threshold in Figure 4.2. We imagine a network with numerous applications and servers all of which use a legacy CCA β. We want to deploy a new application with workload $w^\star$ using CCA α. Is this acceptable?

We can start by considering whether it is acceptable to deploy $(\alpha, w^\star)$ alongside a specific flow $f = (\beta, w)$. Worst case bounded harm looks to the *worst case harm* f might receive from *any* of the other services, with their workloads $w_1, w_2, \ldots$. If $(\alpha, w^\star)$ does not more harm than this worst case, we would consider it acceptable.

Definition: A TCP connection $f^\star = (\alpha, w^\star)$ for α a new algorithm and $w^\star$ a specific traffic workload, respects *worst case harm* with respect to metric m for an algorithm β iff

$$\forall f = (\beta, w \in W), \forall n \in N :$$
$$\textsc{harm}(f, f^\star, n, m) \leq \max_{w^\dagger \in W} (\textsc{harm}(f, (\beta, w^\dagger), n, m))$$

We similarly say that the algorithm α itself has *worst-case harm equivalence* with respect to m for β if all connections $f_n = (\alpha, w_n \in W)$ all respect worst-case harm in m.

Suitability as a deployment threshold: Worst-case bound-ed harm, as a threshold, is too loose: it can allow the outcomes of pathological scenarios to become common through the deployment of a new CCA. Consider a CCA β which is widely deployed and has perfect performance under competition: an ideal fair-sharing allocation, no additional latency, jitter or loss due to new flows, *etc.*, with only one exception. A pessimal workload, $\hat{w}$ can cause any other flow to suffer starvation. However, $\hat{w}$ is extremely rare – so rare in practice that β generally works well. Nonetheless, a malicious protocol designer can take advantage of that – observing that there exists *any* workload that leads a flow f to starvation can justify that *all* of the flows using the new CCA cause starvation for f. This is the same logic – avoiding falling to the lowest common denominator – we used to argue

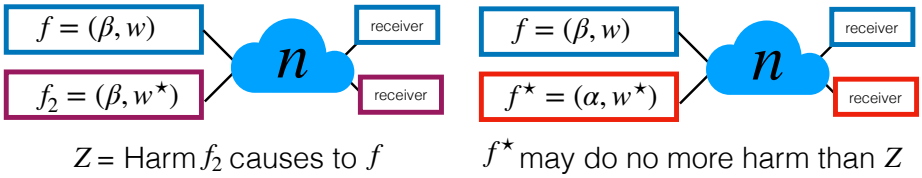


Figure 4.3: Under Equivalent Bounded Harm, a new α with workload $w^\star$ may do as much harm to a β flow as a β flow with workload $w^\star$ as well.

against bounding harm across arbitrary CCA pairs (the max harm in β vs some γ, §4.3.2), and it is the same logic that leads us to reject bounding harm across different networks (the max harm across all $n \in N$).

4.4.2 Equivalent Bounded Harm

To overcome the least common denominator challenge from Worst-Case Bounded Harm, we consider an approach where we pin the workloads being compared. We illustrate the Equivalent Bounded Harm in Figure 4.3. We start by focusing on a pair of workloads w and $w^\star$ in a legacy network where all services use β. We want to switch the service with workload $w^\star$ to use α. With Equivalent Bounded Harm, α would be acceptable iff $(\alpha, w^\star)$ does no more harm to (β, w) than $(\beta, w^\star)$ would.

Definition: A TCP connection $f^\star = (\alpha, w^\star)$ for α a new algorithm and $w^\star$ an specific traffic workload, has *equivalent harm* with respect to metric m for an algorithm β iff

$$\forall f = (\beta, w \in W), \forall n \in N :$$
$$\textsc{harm}(f, f^\star, n, m) \leq \textsc{harm}(f, (\beta, w^\star), n, m)$$

We similarly say that the algorithm α itself has *equivalent bounded harm* with respect to m for β if all connections $f_n = (\alpha, w_n \in W)$ are harmless in m.

Suitability as a deployment threshold: Equivalent bound-ed harm is too strict to serve as a threshold. Consider a CCA BIGFLOW where large flows competing with short flows lead to unfair outcomes. Large flows achieve 75% of available bandwidth capacity and short flows achieve only 25% of available bandwidth capacity. Requiring harm equivalence would entail that short flows using any new algorithm α would only ever be able to achieve up to 25% of the available link capacity when competing with BIGFLOW. Equivalent bounded harm hence falls too close to the trap of mimicry in constraining improvement.

4.4.3 Symmetric Bounded Harm

Our third proposal, shown in Figure 4.4, sits between the too-strict harm-equivalence and the too-permissive worst-case bounded harm. Symmetric Bounded Harm considers pairs of workloads $w, w^\star$ like harm-equivalence. For an existing CCA β with a flow f running workload w, the flow can receive as much harm from $(\alpha, w^\star)$ as *either* f would experience from $(\beta, w^\star)$ *or* as f would *inflict* on $(\beta, w^\star)$.

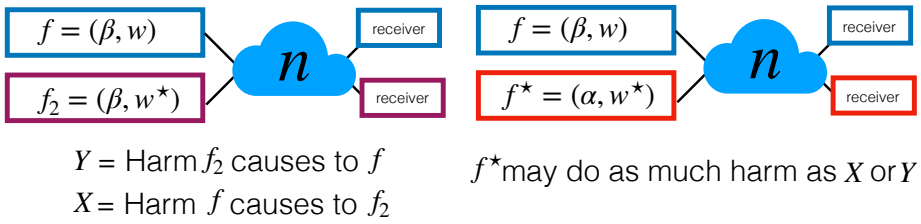


Figure 4.4: Under Symmetric Bounded Harm, a new α with workload $w^\star$ may do as much harm to a β flow f as f does to $(\beta, w^\star)$ or as $(\beta, w^\star)$ does to f.

Returning to BIGFLOW, a large flow f_1 may only receive 25% throughput-harm from a small flow f_2, but since it also inflicts 75% throughput-harm on f_2, a small flow using a new CCA α can inflict up to 75% throughput-harm on f_1.

Definition: A TCP connection $f^\star = (\alpha, w^\star)$ for α a new algorithm and $w^\star$ an specific traffic workload, respects *symmetric-bounded harm* with respect to metric m for an algorithm β iff

$$\begin{aligned}\forall f =&(\beta, w \in W), \forall n \in N : \\ &\textsc{harm}(f, f^\star, n, m) \\ &\leq \max(\textsc{harm}(f, (\beta, w^\star), n, m), \textsc{harm}((\beta, w^\star), f, n, m))\end{aligned}$$

We similarly say that the algorithm α itself has *symmetric-bounded harm equivalence* with respect to m for β if all connections $f_n = (\alpha, w_n \in W)$ all respect symmetric-bounded harm in m.

Suitability as a deployment threshold: Symmetric bound-ed harm resonates with a sense of justice: 'do unto other flows as you would have other flows do to you.' It is not too restrictive, like harm equivalence, but it is not vulnerable to the expansion of harm as we saw in worst-case harm. For these reasons, we prefer symmetric-bounded harm as a potential threshold to the prior two harm-based threshold. Nonetheless, we believe that further work is needed to refine an ideal harm-based threshold.

4.5 Open Questions

Defining a harm-based threshold is only a first step in setting the bar for deploying a new CCA on the Internet. Given a harm-based threshold, we can eventually develop a modern evaluation methodology for CCA deployability. This leaves many open questions and directions for future work:

- Is there an better threshold that improves on symmetric bounded harm?
- Given that Internet outcomes always have some distribution of results, is there 'leeway' in harm? Should we worry about average or worst-case results?
- How widely deployed must a legacy CCA be in order to merit protection by our threshold?
- What are the right workloads for deployability testing?

- If we have a threshold, should it be enforced? If so, how?

4.6 Discussion and Conclusion

In this chapter we argue for the networking community to adopt a *deployment threshold* which provides a firm definition for when a new CCA is allowed to be deployed on the Internet, and when it is not. We believe that the right way forward is by analyzing harm, a way to quantifiably measure the outcomes of introducing a new CCA to the Internet.

We argue that the harm caused by a new CCA α should be bounded by the status quo: do no more harm to flows from a CCA β than β already inflicts upon itself. In this way, algorithms can improve upon outcomes under contention (do *no more harm* than) but are not required to meet overly idealistic goals. A challenge remains in turning this insight into a concise and practical formula for a threshold.

In the following chapter we continue to discuss how we can use harm to create a tool to evaluate the deployability of new congestion control algorithms.

Chapter 5

RayGen: Evaluating Heterogeneous Congestion Control Algorithm Interactions

Google is most interested in BBR, but I do think a somewhat generic multi-algorithm comparison will have a better chance than "test the daylights out of BBR", mostly because we are already testing BBR far harder than possible at any university.

GOOGLE EMPLOYEE

5.1 Introduction

In Chapter 2, we measure widespread deployment of BBRv1 at large content providers including Cloudflare, Cloudfront, and Akamai, while others still use Cubic like Fastly. Yet, we prove in Chapter 3 that in common scenarios, BBRv1 is unfair to Cubic, sometimes leading to starvation. Consequently, in the previous chapter we asked: **How should we evaluate inter-CCA interactions to decide if a CCA is deployable?**. While we discuss in Chapter 4 theoretical terms how we can set a deployment threshold using harm rather than traditional notions of equal-rate fair sharing, we do not discuss how to, in practice, do an evaluation of inter-CCA interactions. In this work, we aim to provide a comprehensive methodology for evaluating inter-CCA interactions.

With the growing heterogeneity of CCAs on the Internet, it is more important than ever to make sure CCAs share reasonably. Even the IETF's working group on congestion control is working on an updated draft RFC 5033 (latest version from July 2024 [109]) on "Specifying Congestion Control Algorithms". This draft specifically says that newly proposed CCAs are *required* to "include a statement in the abstract describing environments where the protocol is not recommended for deployment and *MUST* evaluate the interaction between the proposed algorithm and already widely deployed CCAs:

> *In contexts where differing congestion control algorithms are used, it is important to understand whether the proposed congestion control algorithm could result in more harm than previous standards-track algorithms (e.g., [RFC5681], [RFC9002], [RFC9438]) to flows sharing a common bottleneck. The measure of harm is not restricted to unequal capacity, but ought also to consider metrics such as the introduced latency, or an increase in packet loss. An evaluation MUST assess the potential to cause starvation, including assurance that a loss of all feedback (e.g., detected by expiry of a retransmission time out) results in backoff.*

While inter-CCA fairness evaluation is critical, it is also difficult, as we saw with the development and deployment of BBR at Google. First proposed and deployed in the Linux kernel in 2016, BBRv1 was purported to fix issues with loss-based CCAs. Since its deployment, us and others consistently found issues when BBRv1 flows interact both with other BBRv1 flows [126] and with loss-based CCAs despite widespread deployment of BBRv1 at major content providers [83, 127]. These undesirable fairness outcomes (as well as other issues with BBRv1 like excessive retransmissions) were significant enough for Google to develop BBRv2 and later BBRv3 to fix these issues. In 2023, all Google sites now use BBRv3 with plans to push it into the Linux kernel, replacing BBRv1 [27]. Even still, recent work has shown BBRv3 is *still unfair to Cubic* [138] (We will revisit these findings in §5.3). This example highlights both the importance of evaluating fairness for CCA developers and the difficulty of that evaluation.

Notably our evaluation of the interactions between BBR and loss-based congestion control was an arduous, arbitary, and manual process. During our empirical measurement, we did what many do for this sort of evaluations (examples: [104, 138, 55, 119, 34, 69, 93, 82]): choose some duration of flows to emulate "long-running flows", choose some network settings, and measure if there is fair sharing amongst the flows such that each flow gets the same throughput, perfectly dividing the available bandwidth amongst the flows. Rinse and repeat this process until we find some behavior we do not expect or is not desirable; then figure out what is wrong with the CCA. In our case, we were able to determine from these measurements that BBR's fraction of the link would always be approximately half of the link. In order to gain a stronger sense for when these poor interactions would occur, we attempted to model BBRv1, something even BBR developers did not think was possible and were miraculously able to do so. This did not come without considerable effort and a bit of luck.

Modeling interactions between any pairing of CCAs is not a practical approach for evaluating CCA interactions. CCAs are complex and models often need to make many simplifying assumptions that can impact the findings and the severity of the issues. For example, our model of BBR interactions with loss-based congestion control was later improved by removing some assumptions by Ayush et al. [84]. Similarly, researchers have

made progress in using formal methods to prove CCAs meet performance goals [5, 3], but these approaches leave proving conjectures about interactions between flows of the same CCA as future work because of the difficulty of these proofs; using these methods for flows from different CCAs seems even more difficult.

In practice, when CCA developers and researchers want to evaluate a new CCA, they turn to empirical measurements in emulated networks with the ultimate goal of finding where the CCA performs poorly, especially where it may cause starvation of already widely deployed CCAs. To alleviate the arduous manual process of finding worst-case scenarios, we present RayGen, a tool that CCA protocol designers and researchers can use to evaluate interactions between CCAs, and automatically find settings with worst-case outcomes.

RayGen addresses several challenges in these evaluations of the interactions between heterogeneous CCAs. We frame finding worst-case outcomes as an optimization problem. RayGen uses a genetic algorithm to search a large state space of network settings (*e.g.* bottleneck bandwdith, RTT, queue size) and workloads (*e.g.* number of competing flows). Expanding on the arguments presented in Chapter 4 for using harm rather than JFI as the metric for inter-CCA evaluations, for a given scenario RayGen measures *relative harm* when flows between different CCAs interaction in a testbed. The GA searches for the scenarios with the highest harm by starting from an initial population of random scenarios, and each generation using the relative harm results for each scenario to produce children for the next generation, narrowing in on finding more and more harmful scenarios.

With only a small budget of 300 experiments, RayGen is able to find higher harm scenarios than a random search with the same budget. Surprisingly, in some cases RayGen can even find higher harm scenarios than a parameter sweep of 3500 experiments over the same state space. In addition, to finding scenarios with high harm, we also do not want to get stuck in local minima, finding only a few high harm settings. We show that not only can RayGen find worst-case scenarios, it also can find a diverse set of them.

The rest of this chapter is organized as follows First, we motivate using *relative harm* as the metric for poor outcomes rather than JFI in §5.3.1. Second, we discuss how the duration of experiments can impact outcomes and develop an algorithm to determine when CCA interactions have "converged" in §5.3.3. We then present RayGen in §5.4, discussing the design and evaluation. Finally, we discuss related work in §5.6 and conclude in §5.7.

5.2 Challenges in Evaluating CCA Interactions

There are several challenges in evaluating inter-CCA interactions. In CCA performance evaluations there are always 4 questions we need to ask and answer. We outline these as challenges we need to address in this work.

What metric to evaluate? Definitions of "fairness" are consistently under debate with arguments that achieving equal flow-rate fairness for inter-CCA interactions is neither an achievable nor realistic goal [125, 136, 16, 19]. Recent publications with experiments

showing interactions between recently proposed CCAs and Cubic, all report slightly different metrics: We see JFI [138], throughput ratio [34, 55, 49], throughput share of Cubic [126, 104], ratio of throughput to ideal fair share [6], and just made up metrics like *fness* in in Kunze et al. [69]. Ultimately, all of these metrics are trying to quantify the same thing: is there or there is or is not equal rate-fair sharing, and how far away from that goal are the outcomes. Is the new CCA too unfair to Cubic? As we argue in Chapter 4, harm is the right metric to quantify this. We further motivate using *relative harm* as our metric in §5.3.

How long should we run experiments? CCA performance evaluations have to happen over some workload. Typically, this workload assumes some "infinitely backlogged flows" for some duration. While most flows on the Internet are short (and fast) [139, 64, 10, 136], we typically see work also trying to make some general statements about "'long-running flow" (ex: [27]). In recent CCA proposals and evaluations, duration of flows to measure inter-CCA interactions range from 10-60 seconds [6, 119] to 2-6 minutes [126, 93, 104, 138, 55]; there is no agreed upon rule-of-thumb. Recall in Chapter 3 we see that when BBR and Cubic compete it may take a long time for flows to "converge" to stable performance in large buffers, and that the outcome of BBR and Cubic interactions depends on the duration of the experiments and if the flows are "converged" or not. However, in that work we manually reviewed traces to decide when flows converged. In this work, we show how definitions of convergence for intra-CCA interactions do not apply to inter-CCA interactions and develop an algorithm to determine if flows have converged in an automated way. We run experiments for 3 minutes and use our convergence algorithm to find the point when we are reasonably certain the throughput has stabilized and compute harm after this point. If the flows do not converge we compute harm after the first minute. We describe this algorithm and rationale in §5.3.3.

What network settings to test? For CCA developers that do attempt this evaluation, they test in just a few "common" case scenarios. However, the outcomes are incredibly dependent on the scenario as we saw in Chapter 3. Further, there are many possible "realistic" scenarios given the diversity in network quality around the world. For example, the range of access link bandwidth varies by three orders of magnitude across regions of the world [118, 31]; depending on access medium there might be high or low background loss; ISPs may have different settings for buffer capacity, drop policies, or rate shaping – and each of these parameters can impact performance outcomes. Testing a few scenarios is not necessarily going to illuminate the bad scenarios CCA developers care about remedying. Therefore, developers are faced with a challenge in identifying under what scenarios their CCA plays nicely with other players and under what scenarios it might fail – but a combinatorial nightmare of literally millions or billions of possible scenarios to test and evaluate.

How can we find worst-case scenarios? In order to find the settings that may result in poor performance, CCA developers need to be able to find worst-case scenarios. Ultimately, this leads us to the core question of this work: how can we find worst-case scenarios with poor CCA interactions so CCA developers and researchers can find where these algorithms fail and possibly fix these issues? Towards answering this question we frame this as an optimization problem: Given the function we want to optimize (harm) we want to find the worst-case settings. Existing frameworks (including CCAC [5], CCmatic [3], CCFuzz [95], and Mahak [90]) that could look for worst-case scenarios fall short of achieving this goal, primarily because they do not consider interactions between flows and leave this as future work with unclear extensions to consider inter-CCA interactions.

In contrast, we develop RayGen, a genetic algorithm with the goal of solving this problem: Given the harm function (input with network setting and output is the harm) what are the settings with the highest harm when two different CCAs compete? While there are many options for solving this optimization problem we are drawn to using a genetic algorithm for its many advantages over alternatives in solving this particular problem. It produces a population of solutions, rather than just one. It does not require any knowledge about the function optimizing other than the inputs and outputs. For example, the function does not need to be differentiable. We describe RayGen in §5.4.

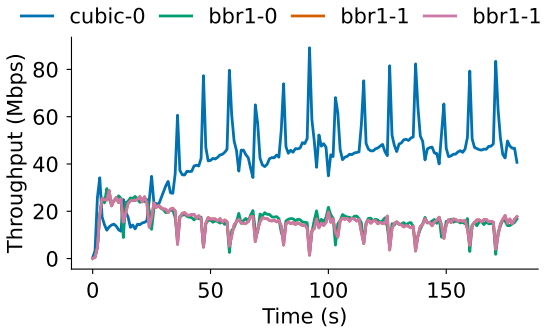
5.3 Motivating A Harm Metric

In this section, we discuss our motivation for using harm as our metric to optimize with our genetic algorithm. In addition, we discuss how the duration of experiments can impact the harm metric, the pitfalls of using harm and how we can overcome those pitfalls.

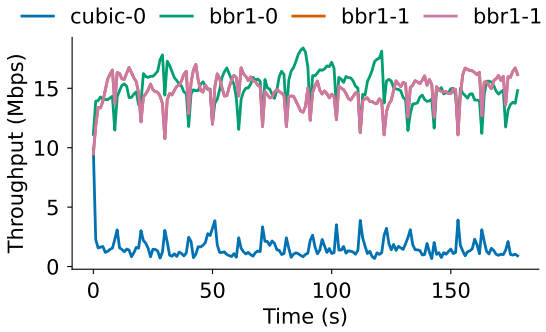
5.3.1 JFI vs. Harm

When thinking about the metric we want to use to evaluate "fairness" we have to think about what is it that we actually want to optimize. Typically, when evaluating CCA interactions we see CCA developers and researchers optimize for equal-rate fair sharing using JFI [61]. Despite our compelling argument against using JFI [125], we still see award-winning research using JFI exactly in the way we discourage [138]. Unfortunately, our description of harm in Chapter 4 is theoretical. We do not say how we can use harm in practice to evaluate CCA interactions and ultimately decide what magnitude of harm is unacceptable. In this section, we discuss why JFI falls short for our goals in inter-CCA evaluations. We are not going to reiterate the arguments made in Chapter 4, but here we do show with empirical results that harm is a better metric, how we can use harm to compare performance outcomes across network settings, and we revisit results in prior work through the lens of harm rather than JFI.

Recall our overall goal: to find scenarios where CCAs have bad interactions. This is our primary evidence for an indictment of JFI: JFI conflates scenarios that we care about distinguishing when we say we want to find “bad interactions.” This conflation is



(a) 100bw-320rtt-4096q: -0.2 relative harm and 0.8 JFI



(b) 50bw-20rtt-128q: 0.2 relative harm and 0.8 JFI

Figure 5.1: 3 BBR flows vs. 1 Cubic flow comparison of JFI and relative harm. While both of these scenarios have the same JFI they do not have the same harm. JFI conflates which CCA is more aggressive while harm does not.

especially bad when trying to compare JFI values to say that one JFI is "worse" or "better" than another, as well as what a higher or lower JFI actually means. JFI evaluates the fairness of resource allocation among flows using the following formula:

$$JFI = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$$

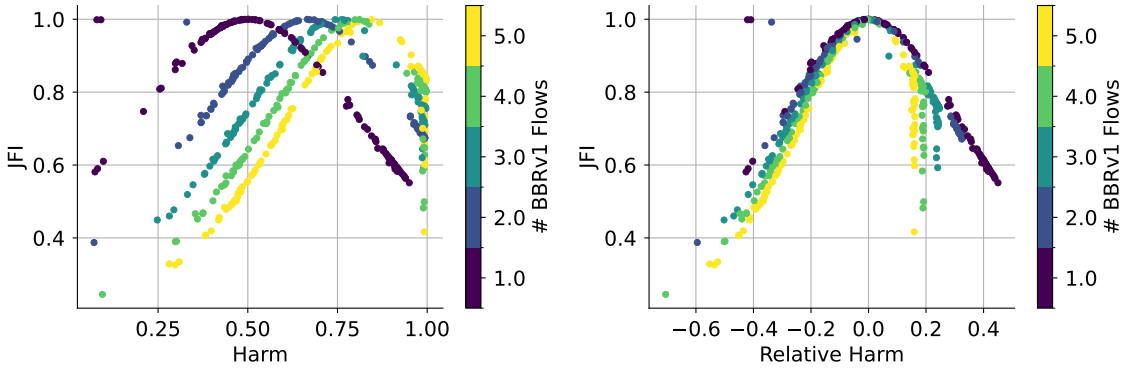
where n is the number of flows and x_i is the throughput for the ith flow. The value of JFI ranges from $\frac{1}{n}$ (worst case, when one flow gets all the bandwidth) to 1 (best case, when all flows get an equal share of the bandwidth).

In Figure 5.1 we highlight an example of two different scenarios we want to distinguish. In both examples shown in Figure 5.1, we have 1 Cubic flow competing with 3 BBRv1 flows in two different settings. Figure 5.1a shows an example network setting where the Cubic flow is being "unfair" to the BBR flows. In contrast, Figure 5.1b shows an example setting where the BBR flows are being "unfair" to the Cubic flow. However, *both of these scenarios have the same JFI of 0.8!*

This is because JFI can only say how far away the outcome is to a perfect fair-sharing bandwidth allocation, and in these cases, that distance is approximately equivalent. As we argue in Chapter 4, we do not need (nor expect) perfect fair-sharing; we actually need to show that a new CCA will share reasonably with an already widely deployed CCA. JFI cannot discern this; it cannot tell the difference between Cubic flows being unfair to BBR flows and vice versa.

Further, and most critical for our goal of finding the "worst" scenarios, JFI cannot meaningfully quantify unfairness to compare the severity of "unfairness" across network settings and scenarios. Comparing JFI values to say that one scenario is more "unfair"

than another is nonsensical, particularly when the scenarios are different (*e.g.* a different number of flows). The difference between how "bad" 0.85 JFI is to 0.80 JFI is unclear and comparing JFI numbers in this way is not meaningful because the scenarios could be completely different in a way that we care about distinguishing like the examples in Figure 5.1.



(a) Absolute harm

(b) Relative harm

Figure 5.2: Comparison of absolute and relative harm for 1 Cubic flow vs. 1-5 BBR flows for varying network settings

Unlike JFI, harm can quantify the reduction in throughput a particular set of flows from one CCA (β) experiences while it is under competition with flows from a different CCA (α). Recall harm computes the fractional difference between a CCA when it competes "alone" β_{solo} versus when it is under competition $\beta_{compete}$ with another CCA for some metric m:

$$Harm = \frac{m(\beta_{solo}) - m(\beta_{compete})}{m(\beta_{solo})}$$

Recall that in Chapter 4 harm is defined in context of both a workload (the number and duration of flows) and the network scenario (bandwidth, RTT, etc.). However, in Chapter 4, we do not discuss how to actually compare harm across scenarios. Turns out, in order to compare across scenarios, we must normalize by the number of competing flows. Consider the case where when a Cubic flow solo achieves 30 Mbps throughput. Under competition with a BBR flow, it now achieves 15 Mbps throughput. The harm here is 0.5, which is actually perfectly fair in this scenario given the number of flows. Consider another example where now there are 2 BBR flows and 1 Cubic flow and the Cubic flow now achieves 10 Mbps. Again, this is perfectly fair given the number of flows. The denominator for computing harm is the same, Cubic's solo throughput, but now the reduction under competition is 0.67. If we were to compare just these harm values it would appear that 0.67 is larger than 0.5 so BBR is more unfair to Cubic in the scenario with more flows which is untrue.

Figure 5.2a shows how the number of flows can impact these harm values. This plot shows the absolute harm in relation to JFI for a variety of network settings with 1 Cubic flow competing with 1-5 BBRv1 flows. As we add flows, absolute harm numbers are going to get worse even though unfairness is not necessarily worse. Absolute harm alone, does not give a sense for how bad the result is; it needs to be put in relation to something. We don't want our "worst case settings" be solely dictated by the number of flows which would happen if we used absolute harm.

Therefore, instead of comparing these absolute harm, we instead use relative harm as our metric to compare how poor interactions may be across scenarios of varying numbers of flows. We compute relative harm by computing expected harm as the harm if there was perfect fair sharing with the number of flows and subtracting expected absolute harm from observed absolute harm. With n_β flows and n_α flows we compute the expected harm as the expected performance for the β flows if there was equal sharing[1]:

$$RelativeHarm = \frac{m(\beta_{solo}) - m(\beta_{compete})}{m(\beta_{solo})} - (1 - \frac{N_\beta}{N_\beta + N_\alpha})$$

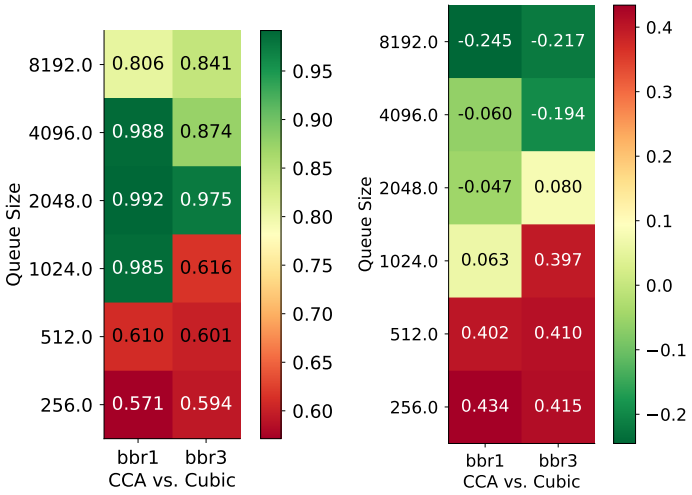
A value of 0 relative harm means there is no harm and there was roughly equal fair sharing; anything above 1 means there is some harm done to β by α flows. A negative value means there is no harm done to β but rather β flows are being unfair to α flows. Absolute harm allows us to distinguish these scenarios and to meaningfully quantify the magnitude of the unfairness[2] across scenarios. Figure 5.2 contrasts absolute harm and relative harm. Figure 5.2b shows that relative harm centers values with no harm and perfect fairness (high JFI) at 0. Thus, when we are searching for scenarios with the worst case relative harm, we are actually searching for scenarios with the worst interactions, independent of the number of competing flows. In the rest of the chapter, when we refer to harm we mean relative harm.

5.3.2 A motivating example

To further motivate using relative harm as our metric over JFI, we revisit a very recent work evaluating the fairness of BBRv3 to Cubic in relation to the fairness of BBRv1 to Cubic. We will show this evaluation is perfectly suited for our relative harm metric rather than JFI. Further, using JFI can lead to the wrong conclusions about how BBRv3's interactions with BBRv1 differ. In addition, this example will lead to our next question which is: how long do we run experiments?

[1]Another possibility is to compute the expected harm as the harm β does to itself, as suggested by our proposal for harm-based threshold in Chapter 3. However, this does double the number of experiments that we need to run for each network scenario. Exploring this alternative definition for relative harm is a promising direction for future work.

[2]We will use the terms 'harm' and 'unfairness' interchangeably.



(a) JFI (b) Harm

Figure 5.3: 100bw-100ms: 1 BBRv1 or BBRv3 flow vs. 1 Cubic for varying queue sizes

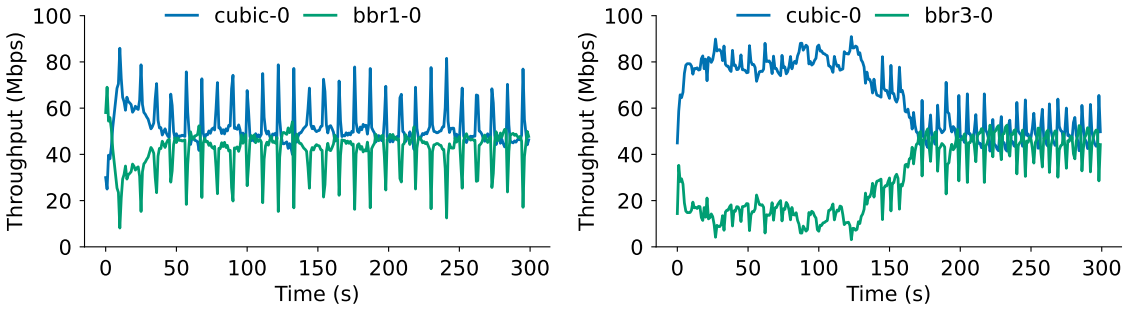
In Zeynali et al. [138], researchers evaluate how the performance of BBRv3 compares to the performance of BBRv1 and BBRv2. The authors run Cubic flows competing with BBRv1, BBRv2, and BBRv3 flows and compare the JFI values to draw conclusions about how the fairness to Cubic changes with each version of BBR. The authors declare in a takeaway from these results: *"BBRv3 does not equitably share bandwidth with loss-based CCAs such as Cubic, and its behavior in some instances is even worse than that of BBRv1".*

We seek to reproduce these results. We run 1 BBRv1 or 1 BBRv3 flow competing with 1 Cubic flow using the same settings as describe in Zeynali et al.: experiments run for 300 seconds, 100 Mbps bandwidth, 100 ms RTT, and a varying bottleneck queue size. Just as the authors visualize their results using a matrix plot, we too visualize our results in this way and roughly achieve the same JFI values. From our shown in Figure 5.3a it appears that BBRv3 is more unfair to Cubic than BBRv1 in the settings with 4096 and 2048 packet size queues. This is misleading so we contrast those JFI values with what is relative harm computed with the same data.

We highlight the throughput traces for the 4096 queue size settings in Figure 5.4, where the JFI when Cubic competes with BBRv1 has a JFI of 0.988 while the JFI when Cubic competes with BBRv3 is lower, 0.874. However, looking at the throughput allocations of the two flows in this figure, we can see that in the case with a higher JFI, Cubic is doing well and is getting slightly higher bandwidth than BBRv1, but they are approximately equally sharing. Conversely, which in the case with a slightly lower JFI, Cubic is still doing well, so well that for the first 150s it is being unfair to BBR! That is why the JFI is lower. This example again highlights that *we absolutely should not compare JFI values to make conclusions about when an inter-CCA interaction is more "fair" or "unfair" than another.*

In contrast, we show what the relative harm values are for the same scenarios in Figure 5.3b and that we can draw relevant conclusions. We see in the 4096 queue setting, it appears that BBRv3 is less harmful to Cubic than BBRv1 because the relative harm is smaller, the correct observation from the traces shown in Figure 5.4. The harm is a small negative value in both cases which already indicates that neither BBRv1 nor BBRv3 is being unfair to Cubic and further, the harm is less for BBRv3 (-0.194) because Cubic is being unfair to BBRv3 in this case. Relative harm allows us to draw the correct conclusions.

There is one more argument that could be made for using JFI. One could say that you could draw the correct conclusions from comparing JFI values if you pair them with the bandwidth allocation between flows (as is shown in figures in Zeynali et al.). However, this defeats the purpose of using JFI in the first place. If we need to pair JFI with the performance of individual flows, it means JFI does not give us all the information we need to decide if the outcome is truly "bad" and the degree of that badness. Relative harm does not have this issue.



(a) Cubic vs. BBRv1 (JFI=0.988, harm=-0.060) (b) Cubic vs. BBRv3 (JFI=0.874, harm=-0.194)

Figure 5.4: 100bw-100ms-4096q: 1 BBRv1 or BBRv3 flow vs. 1 Cubic for 4096 packet queue (~2 BDP). The JFI in these scenarios indicates that BBRv1 is "more fair" to Cubic than BBRv3 which is clearly not true from the traces.

5.3.3 Duration of Experiments & Convergence

In the previous section, in Figure 5.4b we see an example, where inter-CCA dynamics changed over time and if we had computed harm at a later point that would have changed the results. This ultimately leads to a common question when running evaluations of CCA performance: how long should we run experiments for? In the recent CCA proposals and evaluations, duration of flows to measure inter-CCA interactions range from 10-60 seconds [6, 119] to 2-6 minutes[126, 93, 104, 138], so there is no agreed upon rule-of-thumb. Recall in our work in Chapter 4 when different CCAs compete, it may take a long time for flows to "converge" to a stable bandwidth allocation in large buffers and that the outcomes of BBR and Cubic interactions is depends on the duration of the experiments and if the flows are converged or not. However, in this work we do not describe how convergence

is determined and we do this by manually observing the traces and deciding when we believe the flows have "converged". In this work, we ask, can we automatically find the point of convergence? Does that help us decide how long to run experiments?

So what do we mean by "convergence" and how do we determine if flows have converged during inter-CCA competition? We see the term "convergence" and "converging to a fair rate" in recent CCA proposals, both wide area [6, 34] and in data centers [123] as it is one of the important features of a good CCA (as defined by Chiu and Jain [30]). These proposals check for convergence by checking if the flows all end up near the value of equal fair share.

In PCC Vivace [34], authors defines convergence time as time it takes for all the flows to be within 25% of equal fair share:

> *The convergence time is calculated as the time from the second flows entry to the earliest time after which it maintains a sending rate within ±25% of its ideal fair share for at least 5s.*

For the example in Figure 5.4b, this would work since the CCAs each end up converging to a point of roughly equal fair share at about 175 seconds. Where this definition breaks down for inter-CCA dynamics is we do not know a priori that the flows will converge to equal fair share. In fact, in this work we are precisely interested in finding the cases where the flows *do not equally share* like examples in Figure 5.1b where there is not fair sharing the behavior of the CCAs is consistent across the run.

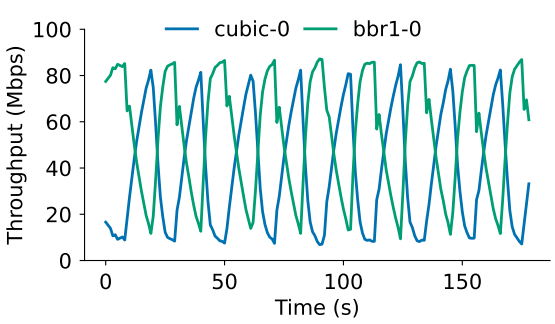
Maybe we might now know that the flows will converge to their equal fair share, but can we expect they will converge to some value? Will the flows be within some percentage of another value? One possible intuitive definition for convergence time is to find time step t such that is for all time after t the throughput does not vary by more than X%. An example of this definition of convergence can be found in Axiomatizing Congestion Control [137], where authors make mathematical definitions for various CCA metrics including convergence time:

> *We say that a congestion-control protocol P is α-convergent, for $\alpha \in [0, 1]$, if there is a configuration of window sizes $(x_1^*, \ldots, x_n^*) \in [0, M]^n$ and time step T such that for any $t > T$ and sender $i \in N$, $\alpha x_i^* \leq x_i^{(t)} \leq (2 - \alpha)x_i^*$ (e.g., $\alpha = 0.9$ means that from some point onwards the window sizes are within 10% from a fixed point.)*

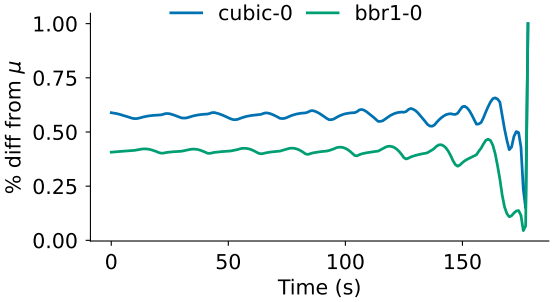
Let's consider a common scenario where both PCC Vivace and Axiomatizing Congestion Control convergence definitions break down. We often see this kind of behavior in our experiments when heterogeneous CCAs interact: large oscillations. In addition, in a comprehensive study of various CCAs and their interactions, authors note that convergence

time to a fair share for many CCAs is long, and often under heterogeneous competition flow behavior is not stable [119]. Figure 5.5a shows an example from our experiments where we see these large oscillations. Notably, while these oscillations are large, they are consistent in their magnitude and frequency, so we do want to say these flows have converged.

In Figure 5.5b shows the output for each flow in Figure 5.5a computing the following: at each time step T the point is the difference (α) between the throughput at time T and the average throughput for all $t > T$. Here we see that BBR1 and Cubic hover around 40% and 60% from their fixed points (average throughput) respectively. We find that this definition suffers from the same pitfalls of PCC Vivace, where it assumes convergence towards a fixed point. In addition, tuning the parameters of α is nontrivial and various CCAs have different kinds of oscillations, so such parameters do not generalize well across multiple CCAs. If we set an α value as large as 60% then flows that have clearly not converged will be considered converged. For example in Figure 5.6 the flows do not converge, but the difference from the average varies a similar amount as Figure 5.5 where it does.



(a) Trace

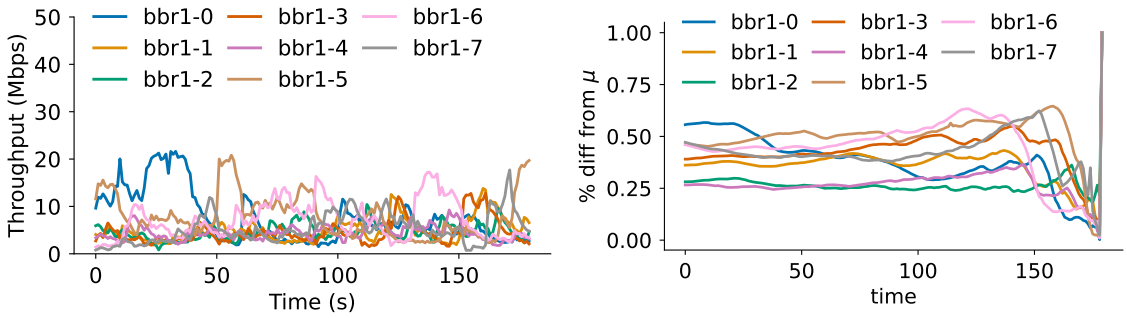


(b) Percent difference of throughput for flows over time.

Figure 5.5: 100bw-10rtt-256q: 1 Cubic flow competing with 1 BBR flow where there are large, but consistent oscillations. Other methods for determining convergence does not work with this example.

We care about convergence because we want the values we compute for harm to be representative about the performance for the CCAs. Ultimately, notions of convergence for intra-CCA fairness does not work well for inter-CCA interactions for two reasons, 1) these definitions focus on convergence to some ϵ where ϵ is known a priori and 2) because it specifically does not want the sorts of large oscillations we see with inter-CCA interactions. We cannot expect CCAs to converge to their fair share, so we cannot check for convergence to some value we know a priori to running an experiment. Given the possibility of oscillations the definition: throughput does not vary more than X%, *does not work*.

Given there is no prior work that defines an algorithm to determine if and when flows



(a) Trace

(b) Percent difference from average

Figure 5.6: 50bw-20rtt-32q: 8 Cubic flow competing where there is no convergence.

under competition have converged, we next seek to develop such an algorithm.

5.3.3.1 Algorithm

Our algorithm takes an experiment with N flows and returns T, where T is a timestamp after convergence has occurred. Thus, the algorithm allows us to conclude with reasonable certainty that all flows have converged after this time T. It is important to note that our algorithm computes the point of *post-convergence* and does not determine the exact point when a flow has converged, rather, it identifies a point where we know convergence has already occurred. It is possible that actual convergence happens before the point we choose, but that is ok because under either circumstance our computation for throughput and subsequently harm would be approximately the same which is our overarching goal. Determining the exact point of convergence or convergence speed is a not our goal.

We have two requirements for our algorithm. First, we do not want to assume that the flows will arrive to a fixed value that we know a priori (like equal-rate fair sharing). Second, we want our algorithm to be able to allow for large oscillations in throughput as long as those oscillations are consistent and stable over time. Consequently, we say a flow has converged at the point where the CCA behavior has reached a consistent state where the standard deviation and the mean has stabilized. More concretely we want to find that the oscillations reach a consistent pattern. In order to find the distinct points where the CCA behavior has changes, we rely on *change point detection* to find the points where the statistical properties of a flow has changed. We describe our algorithm in Algorithm 1.

The core part of post-convergence point selection lies in the *recursive_breakpoints* function. This function takes in a singular flow's throughput over time trace to identify a change in the flow. It accomplishes this by recursively applying a change point detection algorithm (Ruptures-Dynp) on a rolling window of the trace's standard deviation. Ruptures-Dynp is a dynamic-programming based method that identifies a break point, the point where the mean of the signal changes, by minimizing the sum of squared errors when approximating the signal by a piece-wise constant signal [117]. In practice, this is recursive

calls to a function in the `ruptures` Python package asking it to find a single breakpoint in the trace between the point we think convergence has occurred and the end of the trace.

Algorithm 1 Convergence

Input:
f - flow's throughput trace
window_size - size of rolling window for std. deviation
abs_thresh - threshold for absolute difference between left and right means of std. deviation
pct_thresh - threshold for percent difference between left and right means of std. deviation
Output:
t - time after convergence has occurred

procedure RecursiveBreakpoint
  Generate rolling windows of *window_size*
  Calculate standard deviation of each window to generate trace s
  $abs\_diff = inf$
  $pct\_diff = inf$
  $breakpoint = 0$
  while $abs\_diff > abs\_thresh$ or $pct\_diff > pct\_thresh$ **do**
    $s = s[breakpoint :]$
    $breakpoint$ = call *ruptures* on s to find breakpoint
    if can no longer segment s **then**
      return inf
    end if
    $left\_mean = mean(s[: breakpoint])$
    $right\_mean = mean(s[breakpoint :])$
    $abs\_diff = abs(left\_mean - right\_mean)$
    $pct\_diff = abs(left\_mean - right\_mean)/right\_mean$
  end while
  return $breakpoint$
end procedure

To determine convergence for each experiment, we take the PCAP output of an experiment with competing flows and compute throughput over time over windows of 1 second. Then we run Algorithm 1 on each f flow independently and find convergence time f_t. If our algorithm says that any of the flows in an experiment did not converge, we say the experiment did not converge. Alternatively, if all the flows do converge, then we say the point in which we will say convergence has occurred is the latest time t of all flows f. An example of how the convergence algorithm works over iterations is highlighted in Figure 5.8. This is illustrations of each of the iterations of the algorithm for finding the

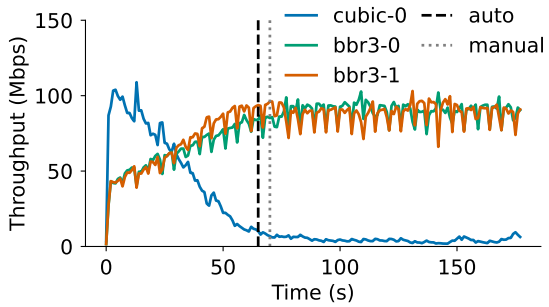
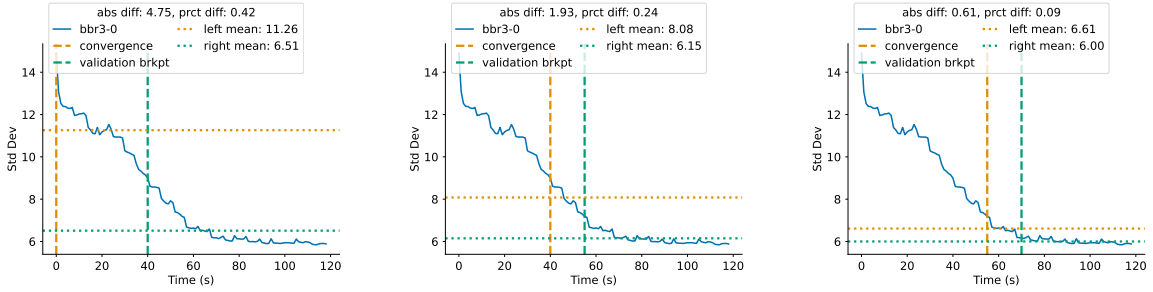


Figure 5.7: An example trace of 1 Cubic flow competing with 2 BBRv3 which we manually label as converging at 65 seconds where Algorithm 1 labels the point of post-convergence as 70 seconds. In either case, the throughput after these points is nearly identical



(a) First iteration (b) Second iteration (c) Third iteration

Figure 5.8: Convergence algorithm output over iterations before deciding the flow has converged for the bbr3-0 flow in Figure 5.7.

point of convergence for the flow labelled bbr3-0 in Figure 5.7.

First, the algorithm computes the standard deviation over time over a rolling window. In the first iteration the algorithm starts by setting the convergence time to time 0. Then it calls `ruptures` to find a change point between our current possible convergence time (time 0) and the end of the standard deviation trace. This change point is labelled the “validation breakpoint” in Figure 5.8. The algorithm then computes the “left mean" of the standard deviation of the trace from where the convergence point is to the validation breakpoint, and the “right mean" of the standard deviation trace from the validation breakpoint to the end of the trace. In the first iteration example in Figure 5.8a, the validation breakpoint is at 40 seconds, and the left mean from 0s-40s of the standard deviation is 11.26 while the right mean from 40s-120s is 6.51. To determine if the convergence point is the point our algorithm decides convergence occurs, we compare the absolute difference between the left mean (4.75) and percent difference (0.42) to thresholds for both. If at least one of thresholds is met, the algorithm says the flow has converged at the convergence point. In this case neither threshold is met, so the algorithm now goes to the next iteration.

In the next iteration the convergence point is where the previous validation breakpoint was. As shown in Figure 5.8b this is at 40s. The algorithm then repeats finding the next change point between this new possible convergence point which is 55s, compute the left mean and right mean and again checks if they meet the thresholds. In this example, it again does not meet the threshold, so the algorithm moves onto the third iteration checking if the convergence point is now at the previous validation breakpoint which is 55s. As shown in Figure 5.8c this new possible convergence point and validation points end up meeting the thresholds and the algorithm returns that the convergence time for this flow is at 70s.

5.3.3.2 Parameter Selection

As shown in Algorithm 1 there are several parameters that we need to tune for the, there are several parameters which we need to choose including: the rolling window size for the standard deviation trace and the thresholds for the absolute and percent difference between the standard deviation left mean and right mean. In order to choose our parameter thresholds, we hand-labeled 100 experiments for the convergence point and calculated the average throughput after these traces. We then used our algorithm to label traces using various *abs_thresh* and *pct_thresh* thresholds. Then, out of the settings that resulted in over 90% agreement on which flows converged, we picked the settings that showed the smallest mean difference in percentage from the average threshold selected by our manual labeling. These settings ended up being an *abs_thresh* and *pct_thresh* of 0.2 and 0.2 respectively. To pick the *window_size*, we simply fixed the *abs_thresh* and *pct_thresh* and varied the window size from 30-100 increasing in increments of 10 and picked the result that had the smallest mean difference from the average threshold selected by our manual labeling. This resulted in an optimal window size of 60.

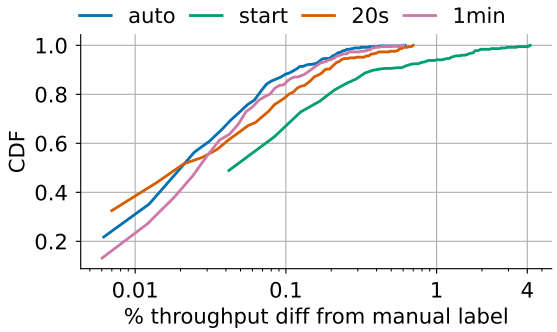


Figure 5.9: Percent difference between manual labeling and auto label from Algorithm 1, 20s, 1min into experiment (on a logx scale)

When comparing the average throughput from the auto labeled vs. the manually labeled experiments, we find that our method is able to get the lowest percent difference in throughput from the manual labeled results. Fig 5.7 shows an example of our labeling

results. We can see in Fig 5.9 that our auto labeling results in the tightest fit, with over 80% having less than a 10% difference in average throughput from our manual labels. In addition, we see that the different in throughput between our manual labels and just ignoring the first 1 minute of the trace is also very close. Consequently, throughput this chapter, we run our experiments for 3 minutes and run the convergence algorithm to decide over what interval to compute relative harm. If our algorithm says the flows converged we compute relative harm from the point of convergence until the end of the trace, ignoring the time before our convergence point. If the algorithm says the flows do not converge, we compute harm from time 60s to the end of the trace, ignoring the first 1 minute of the experiment.

Now that we have spent considerable effort developing a convergence algorithm, it begs the question if it is even useful. More specifically we ask: does the duration of experiments and convergence actually matter when computing harm for inter-CCA interactions? We run experiments for 300 seconds for 1-5 Cubic flows competing with 1-5 BBRv1 or BBRv3 flows over 100 Mbps link and 100 ms RTT, the same network settings in Zeynali et al, once again reproducing this work. We compute the difference between computing relative harm over the entire trace to computing it after the point of convergence for the experiments Algorithm 1 says converged.

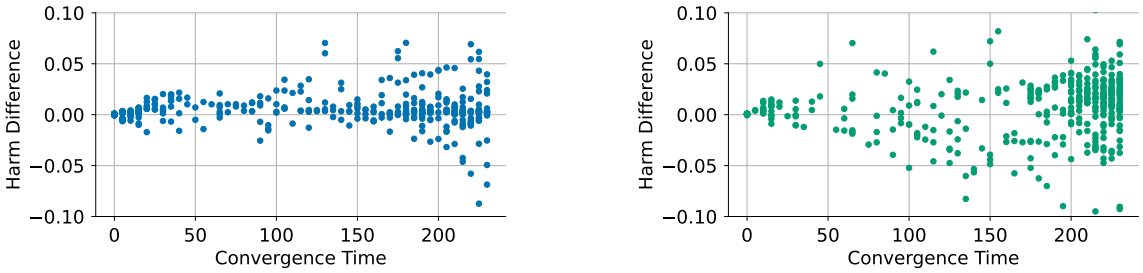
As shown in Figure 5.10 there is a difference. Especially for the experiments that have later convergence times, the difference between the non-convergence and convergence harm values can vary significantly. The percent difference in harm is at its worst when the algorithm identifies later convergence times For example, if the convergence time is at 200 seconds, the harm difference varies by up to 10%.

This further demonstrates the importance of considering the duration of experiments and what you are trying to measure and conclude given the duration. Trying to draw sweeping generalizations like "BBRv3 is more unfair to Cubic than BBRv1" is deceptive and highly depends on the duration of experiments. If we are going to make statements like this, they need to be qualified by the duration of experiment. There is typically no discussion about how the duration of the experiments impacts the results and subsequently conclusions. With our convergence algorithm, we can at least declare that the throughput over time for flows during an experiment has stabilized and consequently can make statements about long-term CCA interactions.

Until now, there has been no agreement on the duration of experiments for exploring inter-CCA interactions; as we mentioned at the beginning of this section, the times vary widely. In evaluations of CCA performance, CCA developers and researchers pick arbitrary times and try to draw conclusions about CCA behavior given these arbitrary times declaring what they care about is "infinitely backlogged flows". While we are not necessarily saying that you must compute convergence time and need to run experiments until the point of convergence (in this work, we do not even do that and run all experiments for 3 minutes), we are offering a tool to determine if the behavior of flows under competition has reached a

stable point so when we compute our metric (in our case, harm) it is meaningful, measuring what we want it to measure.

It is possible of course that CCA developers do not care about long-term behavior and rather care about flows of a specific duration (as most flows are short). In that case, results must be qualified with duration of the flows. We cannot make sweeping generalizations about inter-CCA dynamics without considering convergence, if we care about that. Now we have an algorithm to help us decide how long we may want to run experiments if we care about making statements about long-term behavior.



(a) BBR1: Harm difference

(b) BBR3: Harm difference

Figure 5.10: Reproducing results from Zeynali et al. [138] and running convergence algorithm. For the experiments that do converge late in the CCA, the difference in harm varies. There is a difference between taking the harm over the whole traces vs. after convergence especially for the CCAs with later convergence (Though we are not finding the exact moment of convergence.)

5.4 RayGen

In this section we describe RayGen, our approach for automatically finding worst-case outcomes for CCA interactions. While alternative approaches like simulated annealing are also able to find optimal values of unknown functions, we found that genetic algorithms succeed for three reasons. First, GAs are less susceptible to getting stuck in local minima because they return a random population of results rather than converging to a single value. Second, crossover among populations can be limited to chromosomes of similar network settings, allowing us to achieve a broader range of high harm scenarios across network settings. This is particularly applicable in cases where high harm scenarios may bias towards a single network setting like bandwidth. In this case, we can employ stratification of initialization to force our algorithm to come up with high harm settings across different bandwidths, rather than only coming up with settings with a single bandwidth. Lastly, GAs do not require knowledge of the underlying function (or for the function to be differentiable), we need only the inputs (network settings and workload) and outputs (relative harm). A GA can find worst-case high harm scenarios and a set of them rather than just one.

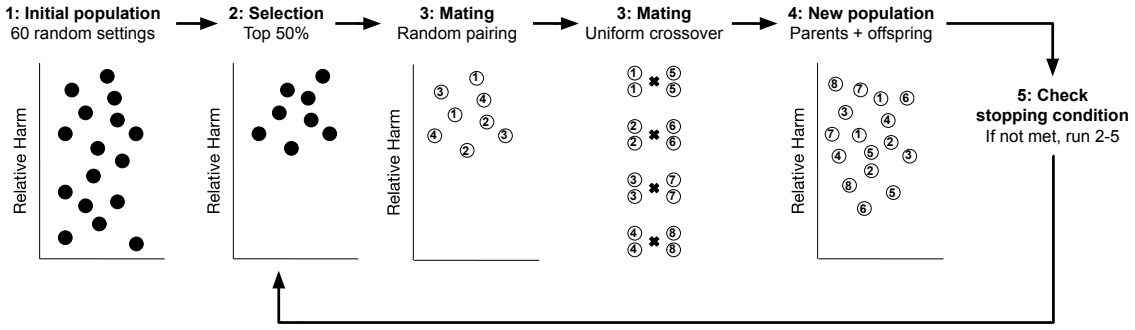


Figure 5.11: Illustration of RayGen's genetic algorithm. After creating an initial random population, the top 50% are chosen for mating in random pairs. Two children are created from each set of parents using uniform crossover. The parents and their children all form a new population for the next generation. If running another generation would exceed the budget of experiments then the GA is terminated.

While GAs work well in a solving a variety of problems, this also comes with the drawback of the sheer volume of parametric choices. In making these choices we choose to keep things as simple as possible. There are many complicated variations of each element described below, but when we considered the options we always make a simple and straightforward approach. We primarily draw inspiration from Haupt and Haupt's popular book and chapter on implementing continuous genetic algorithms [51].

Initialization: A GA is initialized with an initial random population of chromosomes. Our chromosomes consist of 5 genes: bottleneck bandwidth $\in [25, 400]$ Mbps, RTT $\in [10, 320]$ ms, queue size $\in [0.25, 5]$ BDP, as well as the number of competing α and β flows $\in [1, 5]$[3]. This is essentially a vector with 5 elements. Once we have an initial population, we need to compute the fitness function which in our case entails running 2 experiments in our testbed. (The same testbed we use throughout this thesis. Figure 3.1). We need to run an experiment with these network settings with the number of β flows alone and a second experiment with the number of competing β and α flows to compute relative harm.

Selection: Once we have an initial population and the harm values for each chromosome, we have to decide which chromosomes will mate. There are many possible selection algorithms, another parametric choice. RayGen uses "natural selection", selecting the top 50% of chromosomes (where the fitness is the relative harm, the higher harm, the more fit the chromosome) for mating. Because our population size is 60, this will be 30 chromosomes selected for mating. Chromosomes are randomly shuffled into pairs.

[3]The step size of our ranges is log scale because CCA behavior varies by factors of change. For example, there is going to be a significant difference in outcomes between doubling the bandwidth rather than increasing it by 1.

Mating: Once we select the chromosomes we want to mate we need to make two offspring for each pair of parents by performing crossover. [4] There are many possible crossover algorithms, another parametric choice. RayGen uses "uniform crossover". For each gene, we flip a coin to decide if we want to do crossover for that gene. For example, we may decide we are going to crossover the bandwidth gene b between parent 1 and parent 2, b_1 and b_2. Because the variables in our genes are continuous we use a standard blending method to combine these genes to make two offspring by picking a random value $\beta \in [0, 1]$:

$$b_{\text{child1}} = b_1 - \beta(b_1 - b_2)$$
$$b_{\text{child2}} = b_2 + \beta(b_1 - b_2)$$

Intuitively, this will introduce new genes into the population by choosing a random value between the two parents. Each gene has equal likelihood of being selected for crossover. We never blend across genes, bandwidth genes will only every cross with other bandwidth genes, the same for RTT and so on. In the extremes it is possible to crossover every gene or to crossover none of them. In the case there is no crossover the children are copies of the parents. There are ways to prevent this possibility, but the most basic GA implementation does not force unique children. Altering crossover in various ways could improve it, but we choose to keep things simple here.

Mutation: Once we have two children, the next step is to perform mutation. Mutation happens with some probability, another parametric choice. We choose to perform mutation with a probability of 30%. Given our population of 30 children, from mating, this would in expectation mutate 9 children. We do a weighted coin flip to determine if we are going to mutate genes within a child. Weight is 40% so in expectation we are mutating 2 genes. We err on the side of a large mutation probability. The way we are blending the chromosomes during mating will require the gene of a child to be between the value of the parents. While this is good for narrowing down to a specific solution, it could cause RayGen to get stuck in local minima if we are not introducing enough new possible chromosomes in each generation. RayGen uses a random mutation algorithm, taking the mutated gene and assigning it a random value.

Replacement: Once we have children, we need to decide what chromosomes will "die off" and which will remain to form the new population for the next generation. RayGen combines the parents which were selected for mating (the Top 50% from the current population) and the offspring from these children to form the new population. For the children, we compute the relative harm for these possibly new settings.

Stopping condition: RayGen stops when running another generation would exceed the experiment budget which we set to 300 experiments. We will show that with just this

[4]Technically there is an additional parameter: the probability of crossover. In our case we ignore this and set the probability to 1.

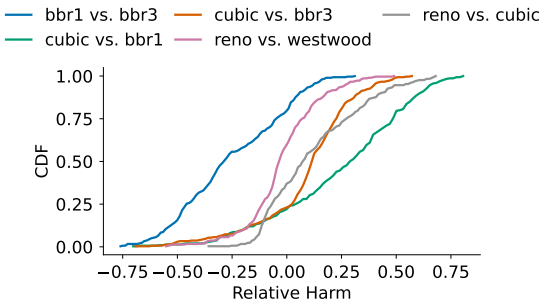


Figure 5.12: Distribution of relative harm for 300 random experiments for various CCA pairs. We use Reno vs. Westwood and BBRv1 vs. BBRv3 in our evaluation.

small budget of experiments, RayGen is able to not only find worst-case harmful scenarios, but also a *diverse set of scenarios.*

5.5 Evaluation

To determine a good set of scenarios to test RayGen, we run experiments for 300 random scenarios for various pairs of CCAs. We show the CDF of relative harm for each of these pairs in Figure 5.12. Each of these lines refers to β vs. α. The distribution of random experiments gives some sense for the difficulty of the problem of finding worst case scenarios. Considering a pair like Cubic vs. BBRv1 will be too easy for an evaluation because you can throw a stone and find a harmful scenario. For our evaluation we focus on Reno vs. Westwood because the majority of the settings found from a random search have low harm, but there is a longer tail than the other pairs, suggesting there are some harmful scenarios that may be hard to find.

Our metric for success for RayGen is to find both harmful scenarios *and* a diverse set of them with a small budget of experiments. We compare RayGen to a random search as well as to an “exhaustive search” where we perform a parameter sweep over the same range of values we allow the GA to search. The range of values is continuous, so we cannot do an actual exhaustive search over the entire parameter space and instead do a “parameter sweep" as an approximation. For the parameter sweep we visit exponential spaced scenarios for a total of 3500 experiments.

To show RayGen can find high harm scenarios, we run RayGen and compare the top 100 scenarios found by RayGen to those found by the parameter sweep and random search. We show that RayGen is better than random search at finding harmful scenarios in Figure 5.13 with the distribution of the Top 100 harm values found by RayGen compared to parameter sweep and random search.

To show RayGen finds a diverse set of high harm scenarios, we take the top 20 harm scenario’s chromosomes and compute the euclidean distance between them. We max-min normalize each feature of the vector, so the ranges of the values does not impact the distance. A small euclidean distance means the scenarios are very similar. We do not want

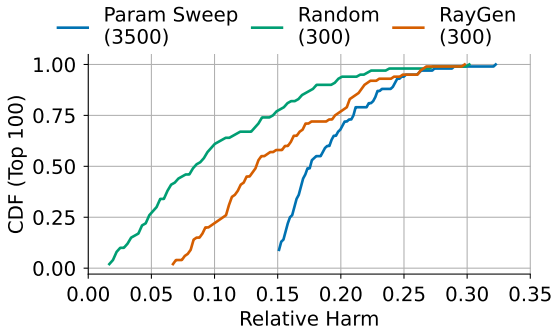


Figure 5.13: Comparison of the top 100 relative harm values found by RayGen to the top values found from a random search and parameter sweep. The GA finds more harmful scenarios than random search.

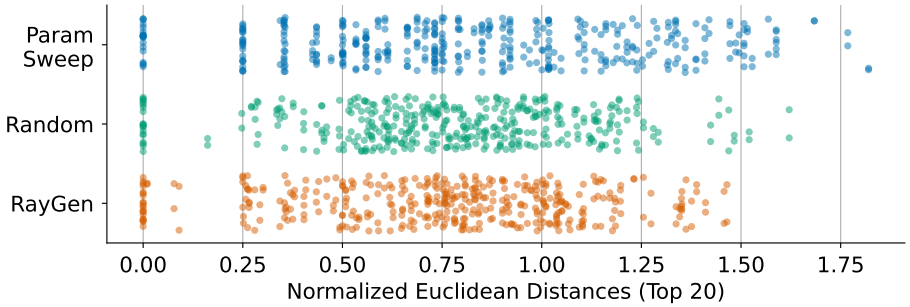


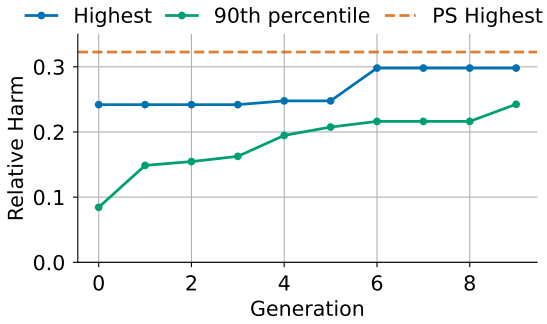
Figure 5.14: Comparison of the euclidean distances between the top 20 harmful scenario's vectors. If the distance is far then the vectors are far away and are different settings, while if the distance is small the settings are very similar.

to just find very similar scenarios. In Figure 5.14 we see that there is as much variety in the high harm scenarios found by RayGen as there are in random search.

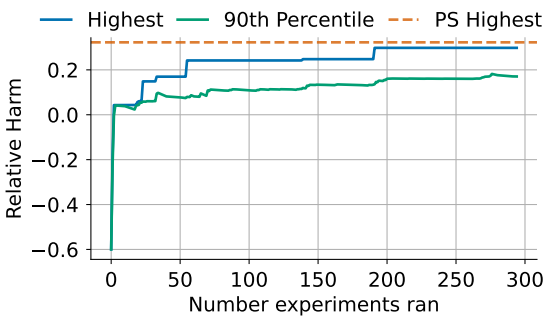
5.5.1 Impact of experiment budget

RayGen is able to find high harm scenarios with only a budget of only 300 experiments. In Figure 5.15 we highlight how the quality of the solutions found by the GA improves over generations and as it runs more experiments. We compare the highest harm value as well as the 90th percentile found by RayGen after each generation in Figure 5.15a, to the highest harm value found by the parameter sweep. After 6 generations and running only 200 experiments, RayGen is able to find harm values almost as high as the parameter sweep which was 3500 experiments. With RayGen, CCA developers can find these worst-case scenarios without having to a run many experiments.

Does increasing the budget of experiments improve RayGen's results? To see the impact of the experiment budget on the results, we run RayGen with a budget of 500 experiments. Figure 5.16 shows that with an increased budget we can find scenarios with

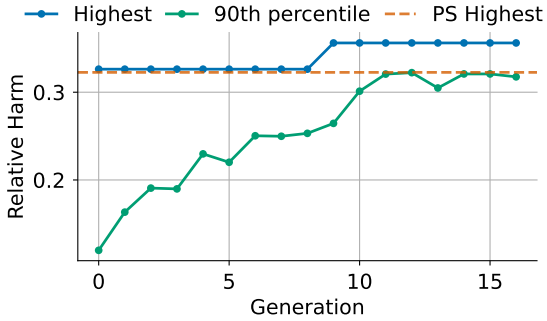


(a) Solution improvement per generation

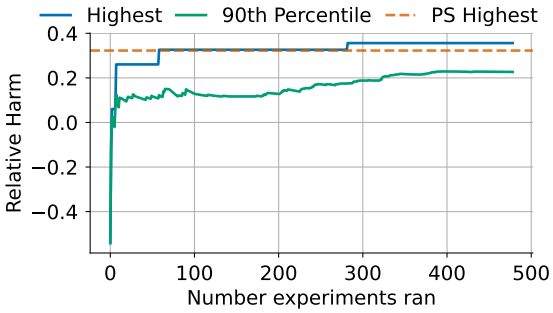


(b) Solution improvement per experiment run

Figure 5.15: Each generation what the highest harm value RayGen has found so far improves as well as the 90th percentile. We see the same trend for the highest harm found in relation to how many experiments RayGen has run so far up to a maximum budget of 300 experiments. We compare these values to the highest harm values found by RayGen the highest harm value found by the parameter sweep.



(a) Solution improvement per generation



(b) Solution improvement per experiment run

Figure 5.16: Each generation what the highest harm value RayGen has found so far improves as well as the 90th percentile for a budget of 500 experiments. With a larger budget we find scenarios with even worse harm than those found by the parameter sweep.

even worse harm than those found by the parameter sweep.

5.5.2 Impact of repeats

We repeat running RayGen two more times to see if we get similar results when re-running the GA. Figure 5.17 compares all three runs of the RayGen to the parameter sweep and random search. While they vary slightly in the quality of top solutions, in all cases the results are better than random search. In addition, we still get the same improvements in the quality of harmful scenarios over generations for the repeated runs of RayGen (Figure 5.18) as well as a variety of those scenarios (Figure 5.19).

The top scenarios found by each run of RayGen are different. This could be considered

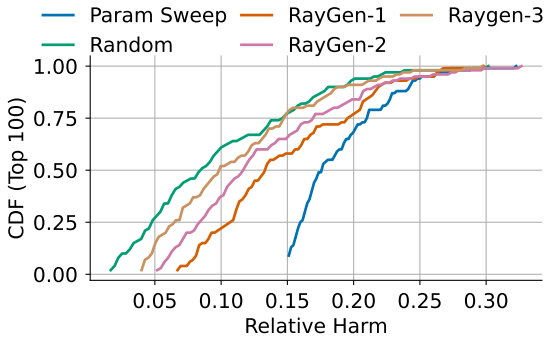
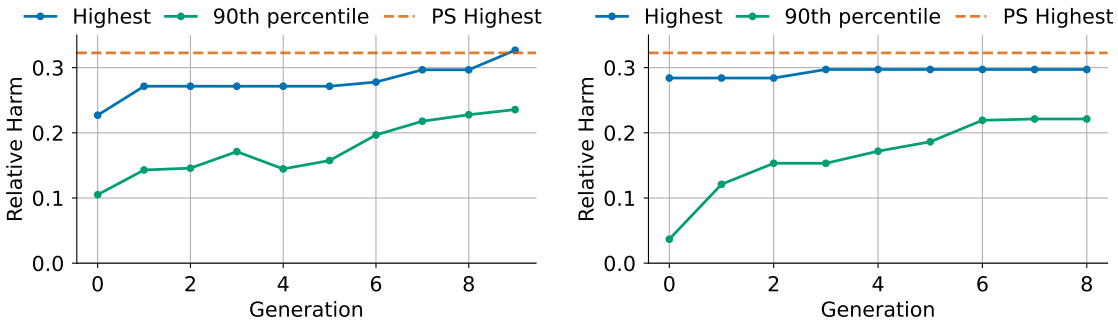


Figure 5.17: Comparison of the top 100 relative harm values found by RayGen to the top values found from a random search and parameter sweep for 3 repetitions. For all repetitions, RayGen finds more harmful scenarios than random search.



(a) Solution improvement per generation (run 2) (b) Solution improvement per generation (run 3)

Figure 5.18: Each generation what the highest harm value RayGen has found so far improves as well as the 90th percentile for repeated runs. We see similar results as Figure 5.15a.

a flaw of RayGen but in fact is an advantage. This indicates that there are local maxima in the harm function and with each run of RayGen it is able to find those settings. If RayGen converged to the exact same worst-case settings every run, this would indicate that there is not enough randomness being introduced over generations. In practice, CCA developers could use RayGen in an iterative process, run RayGen find high harm scenarios, investigate those and possibly fix them, and then run RayGen again and so on.

5.6 Related Work

There are two notable ways prior work evaluate CCA performance: 1) models and proofs that use models of CCAs and networks to prove CCA behavior and 2) empirical methods that take real implementations of CCAs and explore the state space of possible network scenarios.

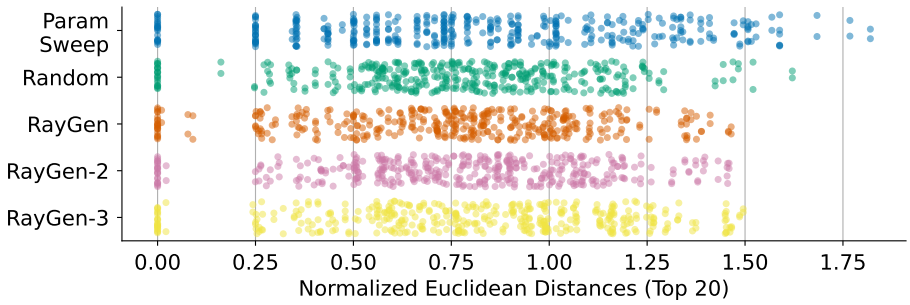


Figure 5.19: Comparison of the euclidean distances between the top 20 harmful scenario's vectors for repeated runs. We see similar results across runs.

Models and proofs: Formal verification techniques [5, 3], generating models of CCAs, and writing proofs about CCA behavior based on those models [126, 137, 115] are all examples of operating on models of CCAs and networks, seeking to prove that certain CCA behavior will (or will not) happen.

For example, Arun et al. develop CCAC [5], using network path and CCA models, to prove certain properties of a CCA using an SMT formulation. The authors note that extending this work to the interaction between multiple flows introduces "non-linear constraints" that the SMT solver is unable to solve. This work does not scale well for multi-flow competition for long timescale, so the authors declare that as future work. The authors later extended CCAC's model to support two flows and developed a proof about how jitter can lead to starvation during intra-CCA interactions [4]. In addition, Agarwal et. al develop CCmatic a framework for designing CCAs that provably meet performance guarantees [3]. The authors also notes the difficulty of designing provably fair CCAs and also declare this future work.

Similarly, there have been other approaches to modeling CCAs and writing proofs about CCA behavior including our BBR modeling work. As we described in §4.1, modeling interactions in the way we did BBRv1 and Cubic, is not always going to be possible or practical. Another example is Zarchy et al. [137] who seek to prove if CCA designs like AIMD and Cubic can meet certain requirements at once like being efficient, fair and TCP-friendly. The main limitation of this work is that it only works for window-based algorithms and is not easily extendable to CCAs like BBR. Another example is Thaker et al. who prove that certain utility functions with trade-offs between throughput and delay are fundamentally incompatible. These utility functions in this work encode application bandwidth and latency requirements rather than specific CCA designs.

Overall, these mathematical techniques are only as good as their models. If the CCA model and network model do not account for multiple flows or do not account for all the kinds of CCAs we want to test, the proofs will not translate well to the realistic network settings and real CCA implementations with bad outcomes we are searching for with

RayGen.

Empirical methods: An alternative to operating on models of CCAs is to operate on implementations of CCAs, testing a variety of network settings using empirical methods. While some of these techniques that present a platform for *manual testing* like Pantheon[5] or Prudentia [93], other tools, like RayGen try to go a step beyond this and automatically search the state space of possible scenarios.

One thread of this work uses software testing techniques, specifically fuzzing. Fuzzing is a software testing technique which can automatically generate test cases. Fuzz testers then run test cases on the target program and then observe the behavior, perturb the inputs based on some heuristic and run the target program again with those new inputs repeating this process until some condition is met [74]. Fuzzing is typically used for finding bugs or vulnerabilities in code where the metric for success is high code coverage. Zou et al. use fuzzing in this way to find bugs in TCP stack code [140]. We are not looking for specific bugs in TCP code.

Two similar works to our problem, ACT [111] and CCFuzz [95], use fuzzing to test CCA implementations using network simulator NS3[6]. The goal of ACT is to find a mapping between network parameters (*e.g.* link bandwidth, loss rate, application rate) and CCA states (*e.g.* `cwnd`, `ssthresh`, `srtt`) to find if a CCA can reach certain bad states depending on the network setting. ACT uses feedback-guided random testing that takes inspiration from genetic algorithms to explore as much of the CCA state space as possible. Unlike any of the prior work, ACT specifically claims: "ACT can be used to test not only a single CCAI flow, but also the interaction among multiple different/same CCAI flows" however there is no evaluation that this is true. CCFuzz uses a genetic algorithm to generate bandwidth traces that capture a certain link variation or cross-traffic variation. The goal is to find traces that are successful in triggering certain undesirable behavior from a CCA. The main challenge with ACT and CCFuzz work is guiding the search of large state spaces in their single CCA settings. In multi-flow scenarios this state space will only explode, making the search even more difficult and possibly inefficient and infeasible.

The most closely related work with nearly the same goals, is Mahak, which uses active learning to automate assessment of CCA performance over a parameter space of network settings [90]. It aims to As the authors describe Mahak as a proof-of-concept, at the time of this writing they have not made their code public (despite claiming so in their paper) and they do not have evaluations for either inter-CCA or intra-CCA interactions. It is unclear if it as simple as adding parameters for the different flows and their CCAs to the parameter space in the same way we have for RayGen. It is certainly possible to try and

[5]While the Pantheon authors note in their publication [132] that adding support for competing flows is future work, I was able to extend it to do so

[6]ACT uses NC3 enabled with DCE [113] to execute actual Linux kernel implementations, while CCFuzz operates on CCA implementations in NS3.

use Mahak to evaluate intra-CCA interactions though the authors have not currently made that possible.

Ultimately the issue with all the prior work is none of these frameworks where designed with evaluating and proving inter-CCA fairness as a first-order goal. In contrast, RayGen was designed with the specific purpose of efficiently evaluating inter-CCA interactions, unlike prior work where these evaluations would be an extension of what is currently proposed in these works. Further, we find that designing CCAs with intra-CCA performance guarantees is already a challenge not tackled by this recent line of work, but it is highly likely extensions to complex inter-CCA interactions will be even more challenging.

We can develop new CCAs and even prove they meet performance goals [5, 3], but those performance goals cannot only focus on CCAs in isolation, but must also include some goals for interactions between flows.

In conclusion, while there are many innovations in designing and evaluating new congestion control algorithms, in all of this work evaluating inter-CCA fairness is either ignored completely (CCFuzz, Mahak) or declared future work in these frameworks (CCAC and CCmatic). To the best of our knowledge, RayGen is the first framework designed specifically with the goal of evaluating interactions between flows efficiently.

5.7 Conclusion

We present RayGen a genetic algorithm for finding harmful inter-CCA interactions and show that it performs well with a small budget of experiments compared to a parameter sweep and a random search. RayGen takes away the arbitrary and manual process of finding worse-case scenarios by using harm as a metric, determining when flows converge to compute harm, and using a GA to automatically find settings with high harm. This work presents a comprehensive methodology that CCA developers and researchers can use to find harmful interactions between CCAs under competition to ease in the task of finding worst-case scenarios to possibly fix or as a recent draft RFC requires, declare as unsafe environments for deployment [109]. In future work, we hope to improve RayGen by exploring other options for its many parameters and configurations. In addition, we want to further demonstrate the efficacy of RayGen by conducting case studies, investigating how new CCAs (*e.g.* BBRv3) interacts with widely deployed CCAs.

Chapter 6

Conclusion

> *It's a classically challenging problem to get these algorithms to play nice with each other, and therefore we worry that there is a lot more unfairness going on on the Internet than we know about.*
>
> *Google's Network Congestion Algorithm Isn't Fair, Researchers Say [14]*
> Ranysha Ware

In this dissertation, we address the limitations of the current methodology for determining if a CCA is reasonable to deploy on the Internet today. Recall our thesis statement:

Thesis statement: *Given the growing diversity in novel congestion control algorithms (CCAs) deployed on the Internet today, we argue that the deployability of new CCAs must be evaluated for how they harm widely deployed CCAs in realistic network settings.*

To support this thesis statement, we present a new CCA classifier, CCAnalyzer, in Chapter 2 that collects bottleneck queue occupancy traces and uses time series classification to determine if a website uses a known or unknown CCA. With CCAnalyzer, we measure the deployment of BBR variations, Cubic, and some unknown CCAs, supporting our statement that there is a growing diversity of CCAs deployed on the Internet. Next in Chapter 3 we can prove BBRv1's fraction of the link when under-competition with Cubic flows is determined by an"in-flight cap." Surprisingly, our model showed the "in-flight cap" did not depend on the number of competing loss-based flows, resulting in significant unfairness. Next, in Chapter 4 we proposed a new metric called *harm* and that the bar for deployability should be how much harm a new CCA does to already widely deployed CCAs. Finally, in Chapter 5 we present RayGen a tool that uses a genetic algorithm to find worst-case high-harm scenarios.

Consider a new CCA developer in 2030 wanting to develop and deploy a new CCA into the Internet to fix issues with BBRv10. They can use CCAnalyzer, an efficient tool, to

determine the one or two most popular, widely deployed CCAs. Then they can use RayGen to find scenarios where their new CCA performs poorly during interactions with these already widely deployed CCAs. Our perspective in this dissertation is that the methodology for evaluating if a CCA is reasonable to deploy on the Internet today is fundamentally flawed. The battle for bandwidth remains, but we are hopeful for a future where evaluating CCA for deployability is easier thanks to this work.

6.1 Looking Backwards

Since the publication of our first paper in this dissertation, I am frequently asked hypothetical and philosophical questions: If we evaluated harm when we were thinking about making TCP Cubic the default algorithm in Linux, would that have changed things? Should we have deployed it? What about when Google deployed BBRv1? The Internet hasn't collapsed, does any of this talk of deployability even matter? In this section, I discuss my answers to these questions based on our findings.

If we evaluated harm when considering replacing TCP Reno with TCP Cubic, should we have deployed it? I speculate that TCP Cubic likely still would have eventually replaced TCP Reno as the default CCA in Linux. Cubic (and its predecessor TCP BIC) addressed TCP Reno's issues in higher bandwidth networks, which was of critical importance at the time. Arguments for deployment were less about achieving perfect fairness with Reno, and more about not being too harmful to Reno in settings where it worked well [49], similar to the deployment threshold we propose in Chapter 4.

BBR is unfair to Cubic, should we have deployed it? BBRv1 should not have been pushed into the mainline Linux kernel without more *public and published* extensive testing.[1] Our results and model described in Chapter 3, proved significant unfairness *after* it was deployed. So why was it deployed? I believe BBR developers had the best intentions: share this huge step forward in congestion control with the world. On a global scale, we could reduce queueing delays without sacrificing utilization, reshaping the Internet in a time when there are increasingly delay-sensitive applications. Due to Google's sheer size and influence, BBR variants now likely serve a massive fraction of Internet traffic with deployment on YouTube. In addition, with CCAnalyzer, which we describe in Chapter 2, we measure 28% of a Top 10K list of websites we measure using BBRv1 including "Hypergiants" [45] Akamai and Cloudflare.

Why hasn't everything collapsed? Does fairness even matter at all? 7 years since its inception, despite poor interactions with TCP Cubic, BBRv1 is still available to deploy

[1]There is an argument that an open-source implementation in the Linux kernel makes testing easier, but there should have been a similar approach to the development of BBRv2/BBRv3. There are published open-source implementations [12, 114] that the network community can use for testing *before* adding it to the mainline Linux kernel.

in the Linux kernel.[2] However, we did see the backlash of deploying a CCA first and asking questions later [116, 14, 121]; but ultimately, BBR is an important case study in the importance of deployability evaluations. These evaluations led to the changes to subsequent BBR versions and will likely continue to lead to more changes, which our proposals in Chapter 4 and Chapter 5 can facilitate.

Based on this state of heterogeneous deployment of congestion control algorithms, I argue that we must clarify and streamline our methodology for determining whether a CCA is deployable to enable the public evaluation missing from BBRv1's development. Before my harm proposal, CCA developers argued that it was ok to be unfair just not "too unfair". Our proposal for measuring harm instead of traditional notions of unfairness in Chapter 4, codified already existing but inconsistent methodologies for congestion control deployability. In addition, harm can measure additional metrics we care about like latency rather than just bandwidth. In Chapter 5, we take this harm proposal one step further, designing a tool, RayGen, that applies this methodology. CCA developers (and researchers) can use RayGen to identify worst-case high-harm scenarios, much like the ones we found with BBRv1, to make finding and fixing potential issues easier. Evaluation of CCA deployability by comparing how a new CCA will interact with widely deployed CCAs is not going away any time soon. While I do not necessarily declare what the exact right "threshold" for declaring a CCA is deployable, the work presented in this dissertation shows how we can improve that evaluation.

6.2 Looking Forward

This dissertation makes significant progress toward understanding and measuring interactions between heterogeneous CCAs. However, there is still much work to do and in this section, we discuss important future directions and their challenges.

6.2.1 Addressing Limitations

There are several limitations of our work that we can address with future work.

Workloads beyond "long-running" flows: The measurements in this dissertation were all centered around "long-running TCP flows." However, most of the flows on the Internet are short [10, 64, 139] and at least 65% of Internet traffic today is video streaming [102], which can stream at high-quality at low bandwidths [93]. Our work must be extended to consider these prevalent workloads.

Consequently, RayGen needs to include workloads with different kinds of applications, flow durations, and bitrates. It should also include other network parameters like loss rate and jitter. These extensions present three challenges. The first challenge is we need to encode the workloads so we can perform meaningful crossover. The second challenge is adding additional parameters increases the size of the search space. It is unclear if the current design of the GA will still work well in a larger parameter space. The third

[2]BBR developers are working to replace this with BBRv3 but it is unclear when that will happen.

challenge is knowing what are "realistic" values for parameters like loss rate. Cloudflare recently publicly published all of their data from speed tests [32, 31] to Measurement Lab [118] which, unlike other speed tests, includes data on latency, jitter, and packet loss! We could also explore this data to determine more realistic network conditions.

We also want to extend CCAnalyzer to QUIC flows and video streams. Classifying video streams is challenging for two reasons. First, there are complex interactions between adaptive bitrate streaming and congestion control; we want to classify just the CCA when the sending rate may be determined more by the adaptive bitrate algorithm than the CCA. Second, CCAnalyzer requires training samples, and it is unclear if we can generate training samples that will work across different adaptive bitrate algorithms.

6.2.2 Future Opportunities

There are opportunities for future work that uses the methodologies and tools presented in this dissertation.

Is BBRv3 a safer replacement to BBRv1? Perhaps we can expand the model in Chapter 3 to include BBRv3 interactions with Cubic. However, we made many simplifying assumptions and BBRv3 is even more complex than BBRv1. Using RayGen, we can find worst-case scenarios when BBRv3 interacts with Cubic. RayGen currently computes harm relative to if the same number of flows had perfect fair sharing. For this evaluation of BBRv3, we could instead compute harm relative to the harm that BBRv1 causes to Cubic in the same scenario, to quantify the difference and specifically look for cases where the harm BBRv3 does to Cubic is worse than the harm BBRv1 does to Cubic. In addition, now with the deployment of BBRv3, there will be interactions between BBRv1 and BBRv3 that we can also investigate with RayGen.

What are consequences of heterogeneity? How prevalent is CCA contention? In Chapter 2, we measure heterogeneity in the Internet's transport layer and consequently wonder about the implications of that heterogeneity. Does the changing congestion control deployments actually impact congestion on the Internet and is it even possible to measure that? Some aruge that congestion control fairness is not as important as we may believe [120] and implementations of isolation [18, 134] can address unfairness issues without requiring different CCAs to have to work well together.

We do not know how persistent congestion is on the Internet today, however, we do know that most of the bytes on the Internet are from long, fast flows [10, 64, 139] and that in these cases, the throughput allocation will be determined by the interactions of CCAs. Dhamdhere et. al. found that between March 2016 and December 2017, there was no evidence of widespread interdomain congestion from their measurements. However, this was *before* the widespread deployment of BBR. CCAnalyzer can measure the evaluation of congestion control deployments, and it would be interesting to couple this with measurements of interdomain links and congestion.

6.3 Impact

When we set out to do this work, a famous computer scientist in computer networking was kind enough to give us some feedback on an early draft of a grant proposal. Their comments, in retrospect, sum up the difficulty and impact of our work. Here are a few quotes:

> *The goal "fairness" turns into a rathole very quickly, and swallows new researchers every couple of years.*

> *Fingerprinting CC types from transit traffic is really hard, especially if you are downstream from the bottleneck (the bottleneck shapes the traffic and masks the CC). Furthermore it is needless."*

> *Claiming that you might derive models is a bit too much hubris. People have been working on that for years.*

Our work on congestion control deployability has impacted the conversation in the networking community about the importance of measuring the impact of how new CCAs will interact with legacy CCAs. Fairness *is important*, and thanks to our work, the conversation is about the *harm* new CCAs do to legacy CCAs. We highlight two examples.

Our first example is related to the development of BBRv3. BBR developers recently shared an Internet draft with the IETF describing BBRv3 in detail. It describes the specific changes that fix the previous `ProbeBW` state to ensure better coexistence with Reno/CUBIC [11]:

> *Choosing the time scale for probing bandwidth is tied to the question of how to coexist with legacy Reno/CUBIC flows since probing for bandwidth runs a significant risk of causing packet loss, and causing packet loss can significantly limit the throughput of such legacy Reno/CUBIC flows.*
>
> *BBR has an explicit strategy for coexistence with Reno/CUBIC: to try to behave in a manner so that Reno/CUBIC flows coexisting with BBR can continue to work well in the primary contexts where they do today*

As our model shows in Chapter 3, BBRv1 caused significant packet loss by increasing its rate while probing for bandwidth. BBRv3 (and BBRv2) tries to fix this issue. The goal for BBRv3 is not to share perfectly fairly with Reno/CUBIC, but rather not to seriously harm loss-based flows in the context in which they perform well.

Our second example, which we mentioned in previous chapters, is a recent draft update to RFC 5033: Specifying Congestion Control Algorithms from the IETF's congestion control working group [109] which specifically mentions harm:

> *In contexts where differing congestion control algorithms are used, it is important to understand whether the proposed congestion control algorithm could result in more harm than previous standards-track algorithms (e.g., [RFC5681], [RFC9002], [RFC9438]) to flows sharing a common bottleneck. The measure of harm is not restricted to unequal capacity, but ought also to consider metrics such as the introduced latency, or an increase in packet loss. An evaluation MUST assess the potential to cause starvation, including assurance that a loss of all feedback (e.g., detected by expiry of a retransmission time out) results in backoff.*

Despite the doubts and challenges, in this dissertation we achieved our goal of better understanding congestion control heterogeneity on the Internet, and how the networking community should evaluate new CCAs for deployability.

Bibliography

[1] Private communication with Ayush Mishra.

[2] Private communication with Neal Cardwell.

[3] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Towards provably performant congestion control. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 951–978, Santa Clara, CA. USENIX Association, April 2024. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 177–192, Amsterdam, Netherlands. Association for Computing Machinery, 2022. ISBN: 9781450394208. DOI: 10.1145/3544216.3544223. URL: https://doi.org/10.1145/3544216.3544223.

[5] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward Formally Verifying Congestion Control Behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 1âĂŞ16, Virtual Event, USA. Association for Computing Machinery, 2021. ISBN: 9781450383837. DOI: 10.1145/3452296.3472912. URL: https://doi.org/10.1145/3452296.3472912.

[6] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, Renton, WA. USENIX Association, 2018. ISBN: 978-1-931971-43-0. URL: https://www.usenix.org/conference/nsdi18/presentation/arun.

[7] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*, 31(3):606–660, 2017.

[8] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. YeAH-TCP: yet another highspeed TCP. In.

[9] Alex Balford. Akamai improves global delivery performance, 2019. URL: https://web.archive.org/web/20191219225447/https://blogs.akamai.com/2019/12/akamai-improves-global-delivery-performance.html.

[10] Simon Bauer, Benedikt Jaeger, Fabian Helfert, Philippe Barias, and Georg Carle. On the evolution of internet flow characteristics. In *Proceedings of the 2021 Applied Networking Research Workshop*, ANRW '21, pages 29–35, Virtual Event, USA. Association for Computing Machinery, 2021. ISBN: 9781450386180. DOI: 10.1145/3472305.3472321. URL: https://doi.org/10.1145/3472305.3472321.

[11] BBR Congestion Control, 2024. URL: https://datatracker.ietf.org/doc/draft-cardwell-ccwg-bbr/.

[12] BBRv2 alpha Linux code. https://github.com/google/bbr/blob/v2alpha, 2019.

[13] BESS: A Software Switch. https://github.com/NetSys/bess.

[14] Karl Bode. Google's Network Congestion Algorithm Isn't Fair, Researchers Say, 2019. URL: https://www.vice.com/en/article/xwepkw/googles-network-congestion-algorithm-isnt-fair-researchers-say.

[15] Lawrence S Brakmo, Sean W O'malley, and Larry L Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[16] Bob Briscoe. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review*, 37(2):63–74, 2007.

[17] Bob Briscoe, Koen De Schepper, Olivier Tilmans, Mirja Kühlewind, Joakim Misund, Olga Albisser, and Asad Sajjad Ahmed. Implementing theâĂŹPrague RequirementsâĂŹ for Low Latency Low Loss Scalable Throughput (L4S). *Netdev 0x13*, 2019.

[18] Lloyd Brown, Albert Gran Alcoz, Frank Cangialosi, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Eric Friedman, Ethan Katz-Bassett, Arvind Krishnamurthy, Michael Schapira, and Scott Shenker. Principles for Internet Congestion Management. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pages 166–180, Sydney, NSW, Australia. Association for Computing Machinery, 2024. ISBN: 9798400706141. DOI: 10.1145/3651890.3672247. URL: https://doi.org/10.1145/3651890.3672247.

[19] Lloyd Brown, Ganesh Ananthanarayanan, Ethan Katz-Bassett, Arvind Krishnamurthy, Sylvia Ratnasamy, Michael Schapira, and Scott Shenker. On the Future of Congestion Control for the Public Internet. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 30–37, Virtual Event, USA. Association for Computing Machinery, 2020. ISBN: 9781450381451. DOI: 10.1145/3422604.3425939. URL: https://doi.org/10.1145/3422604.3425939.

[20] Carlo Caini and Rosario Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International journal of satellite communications and networking*, 22(5):547–566, 2004.

[21] N. Cardwell, Y. Chen, S. Hassas Yeganeh, and V. Jacobsen. BBR Congestion Control. IETF Draft draft-cardwell-iccrg-bbr-congestion-control-00, 2017.

[22] N. Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Ian Swett, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, Matt Mathis, and Van Jacobson. BBRv2: A Model-Based Congestion Control. In *Presentation in ICCRG at IETF 104th meeting*, 2019.

[23] N. Cardwell, Yuchung Cheng, Soheil Hassas Yeganehand Priyaranjan Jha, Yousuk Seung, Ian Swett, Victor Vasiliev, Bin Wu, Matt Mathis, and Van Jacobson. BBRv2: A Model-Based Congestion Control IETF 105 Update. In *Presentation at IETF105*, 2019.

[24] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *Commun. ACM*, 60(2):58–66, January 2017. ISSN: 0001-0782. DOI: 10.1145/3009824. URL: http://doi.acm.org/10.1145/3009824.

[25] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control. In *Presentation in ICCRG at IETF 97th meeting*, 2016. URL: https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-02.pdf.

[26] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control: An update. In *Presentation in ICCRG at 98th meeting*, 2017.

[27] Neal Cardwell, Yuchung Cheng, Kevin Yang, David Morley, Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Van Jacobson, Ian Swett, Bin Wu, and Victor Vasiliev. BBRv3: Algorithm Bug Fixes and Public Internet Deployment. https://datatracker.ietf.org/meeting/117/materials/slides-117-ccwg-bbrv3-algorithm-bug-fixes-and-public-internet-deployment-00, 2023.

[28] Erik Carlsson and Eirini Kakogianni. Smoother Streaming with BBR, 2018. URL: https://engineering.atspotify.com/2018/08/31/smoother-streaming-with-bbr/.

[29] Dean Carmel and Isaac Keslassy. Dragonfly: In-Flight CCA Identification. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2023. DOI: 10.23919/IFIPNetworking57963.2023.10186432.

[30] Dah-Ming Chiu and Raj Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989. ISSN: 0169-7552. DOI: 10.1016/0169-7552(89)90019-6. URL: http://dx.doi.org/10.1016/0169-7552(89)90019-6.

[31] Cloudflare. Cloudflare Radar, 2024. URL: https://radar.cloudflare.com/.

[32] Cloudflare Spped Test, 2024. URL: https://speed.cloudflare.com/.

[33] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 395–408, Oakland, CA. USENIX Association, 2015. ISBN: 978-1-931971-218. URL: http://dl.acm.org/citation.cfm?id=2789770.2789798.

[34] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, Renton, WA. USENIX Association, 2018. ISBN: 978-1-931971-43-0. URL: https://www.usenix.org/conference/nsdi18/presentation/dong.

[35] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, Renton, WA. USENIX Association, 2018. ISBN: 978-1-931971-43-0. URL: https://www.usenix.org/conference/nsdi18/presentation/dong.

[36] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[37] Margarida Ferreira, Akshay Narayan, Inês Lynce, Ruben Martins, and Justine Sherry. Counterfeiting Congestion Control Algorithms. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, 132âĂŞ139, Virtual Event, United Kingdom. Association for Computing Machinery, 2021. ISBN: 9781450390873. DOI: 10.1145/3484266.3487381. URL: https://doi.org/10.1145/3484266.3487381.

[38] findcdn. https://github.com/cisagov/findcdn.

[39] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999. DOI: 10.1109/90.793002.

[40] Sally Floyd. Connections with Multiple Congested Gateways in Packet-switched Networks Part 1: One-way Traffic. *SIGCOMM Comput. Commun. Rev.*, 21(5):30–47, October 1991. ISSN: 0146-4833. DOI: 10.1145/122431.122434. URL: http://doi.acm.org/10.1145/122431.122434.

[41] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, December 2003. DOI: 10.17487/RFC3649. URL: https://www.rfc-editor.org/info/rfc3649.

[42] Sally Floyd and Mark Allman. Specifying New Congestion Control Algorithms. RFC 5033, August 2007. DOI: 10.17487/RFC5033. URL: https://www.rfc-editor.org/info/rfc5033.

[43] Cheng Peng Fu and Soung C Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[44] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. In volume 9 of number 11, 40:40–40:54, New York, NY, USA. ACM, November 2011. DOI: 10.1145/2063166.2071893. URL: http://doi.acm.org/10.1145/2063166.2071893.

[45] Petros Gigis, Matt Calder, Lefteris Manassakis, George Nomikos, Vasileios Kotronis, Xenofontas Dimitropoulos, Ethan Katz-Bassett, and Georgios Smaragdakis. Seven years in the life of Hypergiants' off-nets. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 516–533, Virtual Event, USA. Association for Computing Machinery, 2021. ISBN: 9781450383837. URL: https://doi.org/10.1145/3452296.3472928.

[46] Sishuai Gong, Usama Naseer, and Theophilus A Benson. Inspector Gadget: A Framework for Inferring TCP Congestion Control Algorithms and Protocol Configurations. In *Network Traffic Measurement and Analysis Conference*, 2020.

[47] Gordon. https://github.com/NUS-SNL/Gordon/blob/master/Scripts/tcpClassify.py.

[48] h2load. https://nghttp2.org/documentation/h2load.1.html.

[49] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[50] 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, 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[51] Randy L Haupt and Sue Ellen Haupt. *Practical genetic algorithms*. John Wiley & Sons, 2004.

[52] David A Hayes and Grenville Armitage. Revisiting TCP congestion control using delay gradients. In *International Conference on Research in Networking*, pages 328–341. Springer, 2011.

[53] Tom Henderson and Sally Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, April 1999. DOI: 10.17487/RFC2582. URL: https://www.rfc-editor.org/info/rfc2582.

[54] M. Hock, R. Bless, and M. Zitterbart. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[55] Mario Hock, Roland Bless, and Martina Zitterbart. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2017.

[56] Geoff Huston. BBR TCP. http://www.potaroo.net/ispcol/2017-05/bbr.html, May 2017.

[57] Inspector Gadget. https://github.com/Brown-NSG/inspector-gadget.

[58] iperf3. https://software.es.net/iperf/, 2018.

[59] Alexey Ivanov. Evaluating BBRv2 on the Dropbox Edge Network, 2020. arXiv: 2008.07699. URL: https://arxiv.org/abs/2008.07699.

[60] V. Jacobson. Congestion Avoidance and Control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988. ISSN: 0146-4833. DOI: 10.1145/52325.52356. URL: http://doi.acm.org/10.1145/52325.52356.

[61] Raj Jain, Dah-Ming Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. DEC TR-301, 1984.

[62] Raj Jain, Dah-Ming Chiu, and William R Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. DEC Research Report TR-301, 1984.

[63] Young-Seon Jeong, Myong K Jeong, and Olufemi A Omitaomu. Weighted dynamic time warping for time series classification. *Pattern Recognition*, 44(9):2231–2240, 2011.

[64] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Borylo. Flow Length and Size Distributions in Campus Internet Traffic. arXiv 1809.03486v2, September 2018.

[65] Tom Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.

[66] Eamonn J Keogh and Michael J Pazzani. Scaling up dynamic time warping for datamining applications. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 285–289. ACM, 2000.

[67] Markku Kojo, Pasi Sarolahti, Kazunori Yamamoto, and Max Hata. Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP. RFC 5682, September 2009. DOI: 10.17487/RFC5682. URL: https://www.rfc-editor.org/info/rfc5682.

[68] Christian Kreibich, Nicholas Weaver, Boris Nechaev, and Vern Paxson. iNetalyzr: Illuminating the Edge Network. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 246–259, Melbourne, Australia. ACM, 2010. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879173. URL: http://doi.acm.org/10.1145/1879141.1879173.

[69] Ike Kunze, Jan RÃijth, and Oliver Hohlfeld. Congestion Control in the WildâĂŤInvestigating Content Provider Fairness. *IEEE Transactions on Network and Service Management*, 17(2):1224–1238, 2020. DOI: 10.1109/TNSM.2019.2962607.

[70] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[71] A. Legout and E.W. Biersack. Beyond TCP-Friendliness: A New Paradigm for End-to-End Congestion Control. Technical report, 1999.

[72] Douglas Leith and Robert Shorten. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDnet*, volume 2004. Citeseer, 2004.

[73] Qingxi Li, Mo Dong, and P. Brighten Godfrey. Halfback: Running Short Flows Quickly and Safely. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, 22:1–22:13, Heidelberg, Germany. ACM, 2015. ISBN: 978-1-4503-3412-9. DOI: 10.1145/2716281.2836107. URL: http://doi.acm.org/10.1145/2716281.2836107.

[74] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018. DOI: 10.1109/TR.2018.2834476.

[75] Zachary C. Lipton. The Mythos of Model Interpretability. *Queue*, 16(3):30:31–30:57, June 2018. ISSN: 1542-7730. DOI: 10.1145/3236386.3241340. URL: http://doi.acm.org/10.1145/3236386.3241340.

[76] Shao Liu, Tamer Başar, and Ravi Srikant. TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, 55–es, 2006.

[77] Rob Marvin. Netflix and YouTube Make Up Over a Quarter of Global Internet Traffic. *PC Magazine*, October 2018.

[78] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297. ACM, 2001.

[79] Matt Mathis and Jamshid Mahdavi. Deprecating the TCP macroscopic model. *SIGCOMM Comput. Commun. Rev.*, 49(5):63–68, 2019. ISSN: 0146-4833. DOI: 10.1145/3371934.3371956. URL: https://doi.org/10.1145/3371934.3371956.

[80] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997. ISSN: 0146-4833. DOI: 10.1145/263932.264023. URL: http://doi.acm.org/10.1145/263932.264023.

[81] Pedro R Mendes Júnior, Roberto M De Souza, Rafael de O Werneck, Bernardo V Stein, Daniel V Pazinato, Waldir R de Almeida, Otávio AB Penatti, Ricardo da S Torres, and Anderson Rocha. Nearest neighbors distance ratio open-set classifier. *Machine Learning*, 106(3):359–386, 2017.

[82] Ayush Mishra and Ben Leong. Containing the Cambrian Explosion in QUIC Congestion Control. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, IMC '23, pages 526–539, Montreal QC, Canada. Association for Computing Machinery, 2023. ISBN: 9798400703829. DOI: 10.1145/3618257.3624811. URL: https://doi.org/10.1145/3618257.3624811.

[83] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The Great Internet TCP Congestion Control Census. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), 2019. DOI: 10.1145/3366693. URL: https://doi.org/10.1145/3366693.

[84] Ayush Mishra, Wee Han Tiu, and Ben Leong. Are We Heading towards a BBR-Dominant Internet? In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, pages 538–550, Nice, France. Association for Computing Machinery, 2022. ISBN: 9781450392594. DOI: 10.1145/3517745.3561429. URL: https://doi.org/10.1145/3517745.3561429.

[85] Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Recursively Cautious Congestion Control. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 373–385, Seattle, WA. USENIX Association, 2014.

[86] Fionn Murtagh and Pierre Legendre. WardâĂŹs hierarchical agglomerative clustering method: which algorithms implement WardâĂŹs criterion? *Journal of classification*, 31(3):274–295, 2014.

[87] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and Its Empirical Validation. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, Vancouver, British Columbia, Canada, 1998. ISBN: 1-58113-003-1.

[88] Jitendra Padhye, Jim Kurose, Don Towsley, and Rajeev Koodli. A Model Based TCP-friendly Rate Control Protocol. In *Proceedings of NOSSDAV '99*. Citeseer, 1999.

[89] Jitendra Pahdye and Sally Floyd. On Inferring TCP Behavior. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, 287âĂŞ298, San Diego, California, USA. Association for Computing Machinery, 2001. ISBN: 1581134118. DOI: 10.1145/383059.383083. URL: https://doi.org/10.1145/383059.383083.

[90] Parsa Pazhooheshy, Soheil Abbasloo, and Yashar Ganjali. Harnessing ML For Network Protocol Assessment: A Congestion Control Use Case. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, pages 213–219, Cambridge, MA, USA. Association for Computing Machinery, 2023. ISBN: 9798400704154. DOI: 10.1145/3626111.3628182. URL: https://doi.org/10.1145/3626111.3628182.

[91] Larry Peterson, Lawrence Brakmo, and Bruce Davie. *TCP Congestion Control: A Systems Approach*. Systems Approach, 2022. URL: https://tcpcc.systemsapproach.org/design.html#evaluation-criteria.

[92] Larry L Peterson and Bruce S Davie. *Computer networks: a systems approach*. Morgan Kaufmann, 2007.

[93] Adithya Abraham Philip, Rukshani Athapathu, Ranysha Ware, Fabian Francis Mkocheko, 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, pages 506–520, Sydney, NSW, Australia. Association for Computing Machinery, 2024. ISBN: 9798400706141. DOI: 10.1145/3651890.3672229. URL: https://doi.org/10.1145/3651890.3672229.

[94] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12*, 2012. DOI: 10.1145/2339530.2339576. URL: http://dx.doi.org/10.1145/2339530.2339576.

[95] Devdeep Ray and Srinivasan Seshan. CC-Fuzz: Genetic Algorithm-Based Fuzzing for Stress Testing Congestion Control Algorithms. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, 31âĂŞ37, Austin, Texas. Association for Computing Machinery, 2022. ISBN: 9781450398992. DOI: 10.1145/3563766.3564088. URL: https://doi.org/10.1145/3563766.3564088.

[96] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. A First Look at QUIC in the Wild. In *Passive and Active Measurement*, pages 255–268. Springer International Publishing, 2018. ISBN: 978-3-319-76481-8.

[97] Kimberly Ruth, Aurore Fass, Jonathan Azose, Mark Pearson, Emma Thomas, Caitlin Sadowski, and Zakir Durumeric. A World Wide View of Browsing the World Wide Web. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, 317âĂŞ336, Nice, France. Association for Computing Machinery, 2022. ISBN: 9781450392594. DOI: 10.1145/3517745.3561418. URL: https://doi.org/10.1145/3517745.3561418.

[98] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. Toppling Top Lists: Evaluating the Accuracy of Popular Website Lists. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, 374âĂŞ387, Nice, France. Association for Computing Machinery, 2022. ISBN: 9781450392594. DOI: 10.1145/3517745.3561444. URL: https://doi.org/10.1145/3517745.3561444.

[99] Jan RÃijth, Ike Kunze, and Oliver Hohlfeld. An Empirical View on Content Provider Fairness. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 177–184, 2019. DOI: 10.23919/TMA.2019.8784684.

[100] Stan Salvador and Philip Chan. Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intell. Data Anal.*, 11(5):561–580, October 2007. ISSN: 1088-467X. URL: http://dl.acm.org/citation.cfm?id=1367985.1367993.

[101] Constantin Sander, Jan Rüth, Oliver Hohlfeld, and Klaus Wehrle. DeePCCI: Deep Learning-Based Passive Congestion Control Identification. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, NetAI'19, 37âĂŞ43, Beijing, China. Association for Computing Machinery, 2019. ISBN: 9781450368728. DOI: 10.1145/3341216.3342211. URL: https://doi.org/10.1145/3341216.3342211.

[102] Sandvine. Sandvine's 2023 global internet phenomena report shows 24% jump in video traffic, with Netflix volume overtaking YouTube, 2023. URL: https://www.prnewswire.com/news-releases/sandvines-2023-global-internet-phenomena-report-shows-24-jump-in-video-traffic-with-netflix-volume-overtaking-youtube-301723445.html.

[103] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, 478âĂŞ493, Boston, MA, USA. Association for Computing Machinery, 2018. ISBN: 9781450356190. DOI: 10.1145/3278532.3278574. URL: https://doi.org/10.1145/3278532.3278574.

[104] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. Towards a Deeper Understanding of TCP BBR Congestion Control. In *IFIP Networking 2018*, Zurich, Switzerland, May 2018.

[105] Anant Shah. BBR Evaluation at a Large CDN, 2019. URL: https://eng.verizondigitalmedia.com/2019/09/27/bbr-explore/.

[106] Stanislav Shalunov, Greg Hazel, Jana Iyengar, and Mirja KÃijhlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817, December 2012. DOI: 10.17487/RFC6817. URL: https://www.rfc-editor.org/info/rfc6817.

[107] Scott Shenker, Lixia Zhang, and David D Clark. Some observations on the dynamics of a congestion control algorithm. *ACM SIGCOMM Computer Communication Review*, 20(5):30–39, 1990.

[108] Joel Sing and Ben Soh. TCP New Vegas: Improving the performance of TCP Vegas over high latency links. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 73–82. IEEE, 2005.

[109] Specifying New Congestion Control Algorithms, 2024. URL: https://datatracker.ietf.org/doc/draft-ietf-ccwg-rfc5033bis/07/.

[110] Rayadurgam Srikant. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.

[111] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 719–734, Boston, MA. USENIX Association, February 2019. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/sun.

[112] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proceedings-IEEE INFOCOM*, 2006.

[113] Hajime Tazaki, Frédéric Urbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In *The 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Santa Barbara, United States, December 2013. URL: https://inria.hal.science/hal-00880870.

[114] TCP BBRv3 Release. https://github.com/google/bbr/tree/v3, 2023.

[115] Pratiksha Thaker, Matei Zaharia, and Tatsunori Hashimoto. Don't Hate the Player, Hate the Game: Safety and Utility in Multi-Agent Congestion Control. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, 140âĂŞ146, Virtual Event, United Kingdom. Association for Computing Machinery, 2021. ISBN: 9781450390873. DOI: 10.1145/3484266.3487392. URL: https://doi.org/10.1145/3484266.3487392.

[116] James Titcomb. Google algorithm 'hogs' internet traffic, researchers show, 2019. URL: https://www.telegraph.co.uk/technology/2019/10/27/google-algorithm-hogs-internet-traffic-researchers-show/.

[117] Charles Truong, Laurent Oudre, and Nicolas Vayatis. Selective review of offline change point detection methods. *Signal Processing*, 167:107299, 2020. ISSN: 0165-1684. DOI: https://doi.org/10.1016/j.sigpro.2019.107299. URL: https://www.sciencedirect.com/science/article/pii/S0165168419303494.

[118] David Tuber. Measuring network quality to better understand the end-user experience, 2023. URL: https://blog.cloudflare.com/aim-database-for-internet-quality.

[119] Belma Turkovic, Fernando A Kuipers, and Steve Uhlig. Fifty Shades of Congestion Control: A Performance and Interactions Evaluation. *arXiv preprint arXiv:1903.03852*, 2019.

[120] Belma Turkovic, Fernando A. Kuipers, and Steve Uhlig. Interactions between Congestion Control Algorithms. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 161–168, 2019. DOI: 10.23919/TMA.2019.8784674.

[121] Un algoritmo di Google "monopolizza" il traffico web, 2019. URL: https://www.wired.it/internet/web/2019/10/28/google-bbr-traffico-web/.

[122] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A mechanism for background transfers. *ACM SIGOPS Operating Systems Review*, 36(SI):329–343, 2002.

[123] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274, Boston, MA. USENIX Association, April 2023. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/wang-weitao.

[124] R. Ware, M. K. Mukerjee, J. Sherry, and S. Seshan. The Battle for Bandwidth: Fairness and Heterogeneous Congestion Control. In *Poster at NSDI 2018*, 2018.

[125] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, 17âĂŞ24, Princeton, NJ, USA. Association for Computing Machinery, 2019. ISBN: 9781450370202. DOI: 10.1145/3365609.3365855. URL: https://doi.org/10.1145/3365609.3365855.

[126] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling BBR's Interactions with Loss-Based Congestion Control. In *Proceedings of the Internet Measurement Conference*, IMC '19, pages 137–143, Amsterdam, Netherlands. ACM, 2019. ISBN: 978-1-4503-6948-0. DOI: 10.1145/3355369.3355604. URL: http://doi.acm.org/10.1145/3355369.3355604.

[127] 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, pages 181–196, Sydney, NSW, Australia. Association for Computing Machinery, 2024. ISBN: 9798400706141. DOI: 10.1145/3651890.3672255. URL: https://doi.org/10.1145/3651890.3672255.

[128] Contributors to Wikimedia projects, 2023. URL: https://en.wikipedia.org/wiki/Dynamic_time_warping.

[129] Keith Winstein and Hari Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 123–134, Hong Kong, China. ACM, 2013. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486020. URL: http://doi.acm.org/10.1145/2486001.2486020.

[130] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 459–472, Lombard, IL. USENIX Association, 2013. URL: http://dl.acm.org/citation.cfm?id=2482626.2482670.

[131] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524. IEEE, 2004.

[132] Francis Y Yan, Jestin Ma, Greg Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. *Measurement at http://pantheon. stanford. edu/result/1622*, 2018.

[133] Peng Yang, Juan Shao, Wen Luo, Lisong Xu, Jitender Deogun, and Ying Lu. TCP congestion avoidance algorithm identification. *IEEE/ACM Transactions On Networking*, 22(4):1311–1324, 2013.

[134] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 219–232, Amsterdam, Netherlands. Association for Computing Machinery, 2022. ISBN: 9781450394208. DOI: 10.1145/3544216.3544240. URL: https://doi.org/10.1145/3544216.3544240.

[135] Zakir Durumeric. https://github.com/zakird/crux-top-lists.

[136] Adrian Zapletal and Fernando Kuipers. Slowdown as a Metric for Congestion Control Fairness. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, pages 205–212, Cambridge, MA, USA. Association for Computing Machinery, 2023. ISBN: 9798400704154. DOI: 10.1145/3626111.3628185. URL: https://doi.org/10.1145/3626111.3628185.

[137] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. An Axiomatic Approach to Congestion Control. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 115–121, 2017. DOI: 10.1145/3152434.3152445. URL: https://doi.org/10.1145/3152434.3152445.

[138] Danesh Zeynali, Emilia N. Weyulu, Seifeddine Fathalli, Balakrishnan Chandrasekaran, and Anja Feldmann. Promises and Potential of BBRv3. In *Passive and Active Measurement: 25th International Conference, PAM 2024, Virtual Event, March 11âĂŞ13, 2024, Proceedings, Part II*, pages 249–272, Berlin, Heidelberg. Springer-Verlag, 2024. ISBN: 978-3-031-56251-8. DOI: 10.1007/978-3-031-56252-5_12. URL: https://doi.org/10.1007/978-3-031-56252-5_12.

[139] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the Characteristics and Origins of Internet Flow Rates. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 309–322, Pittsburgh, Pennsylvania, USA. ACM, 2002. ISBN: 1-58113-570-X. DOI: 10.1145/633025.633055. URL: http://doi.acm.org/10.1145/633025.633055.

[140] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502. USENIX Association, July 2021. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/zou.