Designing Network Control Algorithms with Performance Guarantees

Anup Agarwal

CMU-CS-25-151

December 2025

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

Thesis Committee:
Srinivasan Seshan, Chair
Vyas Sekar
Justine Sherry
Philip Brighten Godfrey (UIUC)
Venkat Arun (UT Austin)

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

This thesis was supported in part by NSF grants CNS-2403026, CNS-2212102 and CNS-2212390. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Computer Networks, Transport Protocols, Congestion Control, Formal Methods, Automated Reasoning, Performance Verification.

For my parents, Renu Jagdish and Jagdish Prasad Agarwal.

Abstract

Control algorithms are ubiquitous in networked systems—from congestion control and load balancing to scheduling and caching. Despite their performance-critical nature, these algorithms are designed using human intuition and heuristics, and they frequently exhibit poor or unpredictable performance. This thesis envisions a methodology for designing controllers with *formally verified performance guarantees*. We focus on congestion control algorithms (CCAs)—a domain that continues to experience repeated failures despite decades of research.

Two main reasons make congestion control hard. First, CCAs operate in diverse environments with noisy feedback (e.g., cellular links, policers, token-bucket filters, operating system jitter, etc.). Second, the problem itself is computationally hard because CCAs operate under uncertainty—they lack direct visibility into the state of the network or the flows they compete with. To address the first challenge, we adopt the approach proposed by recent work that models networks as non-deterministic, non-stochastic automata. These models can capture a wide range of real-world phenomena and enable computer verification of controller performance on such networks. We seek to design CCAs that pass such verification checks. However, this approach does not scale out of the box.

We find that the key to making it tractable is to formally reason about uncertainty in the state of the network and other flows. This thesis contributes two abstractions—*beliefs and contracts*—that enable such reasoning and reveal new structure in CCAs that simplifies their design and analysis. Beliefs formalize what a CCA can infer about latent network state from its observations. Contracts formalize how flows coordinate with each other to share the network. Since flows cannot directly communicate, they implicitly encode information in observable congestion signals (e.g., delay or loss). Contracts formalize these communications mechanisms. Building on these abstractions, we develop CCmatic, a tool that automatically synthesizes CCAs with verified performance guarantees. Our abstractions and tools allowed us to discover previously unknown tradeoffs and design new CCAs that are on the Pareto-frontier and provably guarantee performance even under challenging network conditions.

We believe that the methodology developed in this thesis generalizes beyond congestion control and we outline steps towards this vision.

Acknowledgments

They say that if you want to go fast, go alone—but if you want to go far, go together. I want to thank all the people who walked alongside me on this long and unforgettable journey.

First and foremost, I want to thank my advisor, Srinivasan Seshan. Thank you, Prof. Srini, for your guidance throughout the years, for giving me the space to make mistakes and learn from them, for being open to every kind of discussion—whether low-level debugging or high-level storytelling—and for the countless lessons wrapped in your wisdom and humor. Most of all, thank you for having more faith in my abilities than I even had myself and for pushing me to pursue increasingly ambitious research projects. This thesis would not have been possible without your support and I could not have asked for a better advisor.

I am deeply grateful to all the other members of my thesis committee: Vyas Sekar, Justine Sherry, Philip Brighten Godfrey, and Venkat Arun. Thank you, Prof. Vyas, for your sharp questions that helped bring more clarity to this work. Thank you, Prof. Justine, for instilling in me the importance of communicating well both in writing and speaking. Thank you, Prof. Brighten, for your thought provoking feedback on various parts of this thesis and also shepherding some of the work in this thesis. Thank you, Prof. Venkat, for teaching me most of what I know about congestion control, for collaborating on all the work in this thesis, and for encouraging me to pursue academic research when we were both undergraduate students.

Over the years, I have had the privilege of working with an amazing set of research collaborators. Thank you for all the spirited technical discussions, patient explanations, and sharing the stress and chaos of academic deadlines. I truly learned something unique from each one of you. Thank you Abdul Kabbani, Ahmad Ghalayini, Anirudh Badam, Antonis Manousis, Daehyeok Kim, Devdeep Ray, Hari Balakrishnan, Hun Namkung, Íñigo Goiri, Işıl Dillig, Jia (Tony) Pan, Jinghan Sun, Manya Ghobadi, Mohammad Alizadeh, Mohit Jain, Prateesh Goyal, Pratyush Kumar, Ranveer Chandra, Ravi Netravali, Ruben Martins, Sanjoli Narang, Sepehr Abdous, Shadi Noghabi, Shivkumar Kalyanaraman, Srinivasan Iyengar, Tommaso Bonato, and Zaoxing (Alan) Liu.

I am fortunate for the wonderful community of colleagues at CMU who continuously shaped and strengthened my work. Thank you for countless rounds of feedback on practice talks, for inspiring me to keep pushing the boundaries of my research, and for sharing technical insights over various research discussions. Thank you to the faculty members: Dimitrios Skarlatos, Mor Harchol-Balter, Nathan Beckmann, Peter Steenkiste, Philip Gibbons, Rashmi Vinayak, and Yuvraj Agarwal. And, thank you to the students: Aditi Kabra, Adithya Abraham Philip, Anuj Kalia, Brian Zhang, Cayden Codel, Christopher Canel, Francisco Pereira, Hugo Sadok, Isabel Suizo, Mallesham Dasari, Margarida Ferreira, Miguel Ferreira, Michael Rudow, Nirav Atre, Pratiksha Thaker, Ranysha Ware, Sagar Bharadwaj, Sara McAllister, Shawn Chen, Timothy Kim, Wan Shen Lim, Yash Savani, Yiran Lei, Yuze (Zoey) He, Zhipeng Zhao, and Zili Meng.

I would also like to thank all the staff members at CMU who handled countless logistics—from booking rooms and coordinating reimbursements to resolving visa issues—allowing me to focus my time and energy on research. Thank you Catherine Copetas, Charlotte Yano, Deborah Cavlovich, Diana Hyde, and Matthew Stewart.

Like many international students, I traveled half way across the world to pursue graduate studies. My time would not have been bearable without colleagues at CMU above and all the friends who made this foreign land feel like home. Thank you for partaking in all the activities I cherished: skiing/ice-skating trips, hiking/camping trips, climbing/gym/tennis sessions, brunches/dinners, coffee runs, gaming/movie sessions, and even car repair sessions. Thank you Abhiram Kothapalli, Ananya Joshi, Andrew Park, Ankush Jain, Arjun Lakshmipathy, Arjun Teh, Asher Trockman, Daiyaan Arfeen, David Kahn, Dorian Chen, Don Dennis, Edward Chen, Eitan Zlatin, Elanor Tang, Francisco Maturana, Giulio Zhou, Ioannis Anagnostides, Isaac Grosof, Jack Kosaian, Jalani Williams, Jason Choi, Jatin Arora, Jovina Vaswani, Juncheng Yang, Justin Raizes, Justin Whitehouse, Long Pham, Lucio Dery, Mara Aragones, Michael Zlatin, Mihir Bala, Mikhail Kodhak, Mughda Deokar, Myra Dotzel, Noah Singer, Pallavi Koppol, Patrick Coppock, Praneeth Kacham, Pratik Fegade, Rohan Aggarwal, Roger Iyengar, Sai Sandeep, Sambhav Satija, Sanjith Ahtlur, Sachin Goyal, Shir Maimon, Shilpa George, Shivram Gowtham, Siddharth Prasad, Siva Somayyajula, Sophia Roshal, Sol Boucher, Suhas Jayaram, Vinay Banakar, and many others.

I would like to thank my partner, Surabhi Gupta, who is effectively an unnamed author on this thesis, staying by my side through the highs and lows of undergraduate and graduate school. I am excited for all the future adventures that await us.

Finally, I would like to thank my family for their unwavering love and support. I want to especially thank my parents, Renu Jagdish and Jagdish Prasad Agarwal, for all their sacrifices in life and laying a strong foundation for me to achieve my dreams and hopes. This thesis is dedicated to them.

If I have missed anyone, it is only my memory falling short, not my appreciation. Thank you to everyone who helped me in ways big and small.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The point of rigor is not to destroy all intuition; instead, it should be used to destroy bad intuition while clarifying and elevating good intuition. It is only with a combination of both rigorous formalism and good intuition that one can tackle complex mathematical problems."

— Terrence Tao

Computer systems rely on control policies for performance-critical decisions, e.g., scheduling, caching, and congestion control. These controllers are often designed in an ad-hoc manner, guided by heuristics rather than principled reasoning. Consequently, we lack clarity on the workload or environments in which they perform well or fail. We routinely discover new failure scenarios and apply ad-hoc patches. Over decades, this cycle has culminated in hundreds of papers promising improved performance but failing to provide lasting, generalizable solutions.

We take a step back and ask: *What would it take to design controllers with provable performance guarantees?* We seek to extend the ideas of *correctness verification* to the realm of *performance verification*. Just like we ensure software is free from deadlocks or race conditions across all thread interleavings, our goal is to design controllers that can *prove* bounds on metrics like latency, throughput, and fairness across all workloads and environments they may encounter.

Within this broad vision, this thesis focuses on bringing such assurances to end-to-end congestion control algorithms (CCAs). We briefly explore adaptive bitrate (ABR) algorithms for video streaming and outline an agenda for generalizing our approach in Chapter 8. CCAs determine the sending rate of network flows and are critical for applications like video streaming, cloud gaming, and distributed AI/ML training. Despite decades of research, a general-purpose CCA that performs robustly across diverse network scenarios remains elusive. All existing general-purpose CCAs (e.g., Cubic, Reno, BBR, PCC, Copa) perform poorly in some practical scenario [19, 20, 32, 41, 120] (Chapter 2).

This thesis builds tools to advance the design and analysis of CCAs, getting us closer to "solving" congestion control. The difficulty in congestion control stems mainly from two key challenges: (1) noise and diversity in network environments, and (2) computational hardness due to uncertainty about system state.

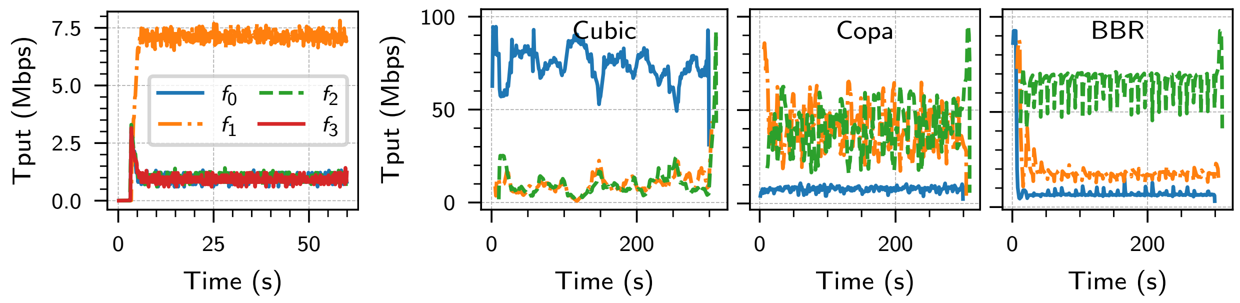


Figure 1.1: Existing CCAs are unable to cope with noise and diversity in networks. **Diversity (Left).** Astraea [84] works fine when link capacity is high (e.g., 100 Mbps), but breaks when it is low (e.g., 10 Mbps). The left figure shows the throughput (Tput) of four Astraea flows on a dumbbell topology with 10 Mbps capacity. One flow takes most of the link capacity. **Noise (Right 3).** Cubic, Copa, and BBR starve flows when there is noise in the network. For Cubic and Copa, starvation occurs when the blue flow experiences ACK aggregation. Whereas for BBR, starvation occurs when flows have different round-trip propagation delays (RTprops), e.g., blue, orange, and green flows have RTprop of 10 ms, 20 ms, and 30 ms respectively.

Challenge 1: Noise and diversity in environments. CCAs must deliver performance across a wide range of network paths, each with unique combinations of physical links (e.g., wired, cellular, satellite [12, 29, 30, 59, 117, 121, 123]) and processing elements (e.g., load balancers, switches, routers). This manifests as (1) wide diversity in network parameters (e.g., link capacities, buffer sizes) and (2) noise (e.g., jitter or non-congestive delays) in network feedback. [1]

Challenge 2: Hardness and uncertainty. Recent formulations show that CCA design is NEXP-hard [116]. This hardness stems from uncertainty, which arises from two sources. First, CCAs are *decentralized*: flows cannot directly communicate and are unaware of others' states. Second, they operate under *partial observability*: flows lack direct visibility into network state (e.g., topology, link capacities, queue occupancies). They only observe the timings of their own transmissions and acknowledgments (ACKs). From these narrow local observations, they must compute statistics or congestion signals (e.g., packet delays or losses) to modulate their rate decisions. If they knew the global state, rate decisions would have been trivial.

In the first challenge, existing designs often model networks using idealized or stochastic queues, which fail to capture real-world complexity. Many designs overlook noise; since CCAs rely on feedback, noise can cause them to mis-estimate congestion and send at rates 10× or more away from optimal. Furthermore, existing approaches often restrict the range of network parameters they consider. Empirical evidence suggests tradeoffs between performance and environment diversity [104]. For instance, Fig. 1.1 illustrates some common scenarios where existing CCAs break: Astraea—a recent reinforcement learning (RL) based CCA—performs well at low link capacities but poorly at high ones (Fig. 1.1, § 6.3); traditional loss-based CCAs perform well only when network buffers are large (not shown in Figure); and recent work [20] showed that all existing CCAs starve on networks with jitter and high link capacities (Fig. 1.1). Our community does not fully understand these tradeoffs or know if they are fundamental or artifacts of individual designs.

[1]This noise comes because network middleboxes often batch packets or delay ACKs. In WiFi routers, nodes negotiate access to shared wireless links. Furthermore, OS scheduling can delay threads. All these facets create variable (non-congestive) delays in packet processing.

To model noise and diversity in real-world networks, we borrow the approach of recent work [19, 20]. It uses non-deterministic automatons to model networks. This approach has two key tenets. First, in this model, the network can add arbitrary (or adversarial) noise (e.g., delays) up to a bounded amount. We design controllers as a function of this bound, allowing them to degrade gracefully as measurement errors increase. We also treat network parameters as symbolic variables, making no restrictions on their ranges to capture diversity. This is similar to design of adversarially robust controllers in robotics. Second, this framework provides a mathematical encoding of the network that allows use of automated reasoning tools (e.g., Z3 [40]) to symbolically explore control trajectories enabling formal verification of controller performance. It is hard for humans to manually reason about all possible interactions between a controller and the network. These tools exhaustively reason about all possible control trajectories to verify that the controller meets desired performance objectives in the worst-case over all modeled environments.

Unfortunately, this formal approach alone does not solve the computational hardness problem and it is intractable to design controllers that meet the pass the verification checks. We initially tried designing controllers by hand, when that failed, we tried throwing compute at the problem, which also failed. We used program synthesis to automatically explore the space of CCAs to design one that provably meets desired objectives on the above network models. The problem was that this framework required us to make a lot of non-trivial design decisions in specifying the CCA design space, e.g., what features should it derive from its observations to guide rate decisions. We found that if we gave too much design freedom to the computer, it failed to terminate, while if we restricted the freedom, we were unsure if any performance limitations were fundamental or a mere artifact of the restrictions.

The key contribution of this thesis is making this formal approach tractable. We find that reasoning about uncertainty is the central design element in congestion control. Once uncertainty is correctly addressed, the rest of network control becomes immensely simple. We build two abstractions, **beliefs** and **contracts**, to formally reason about uncertainty. These abstractions reveal new structure in CCAs that vastly simplifies their design and analysis.

Beliefs address partial observability and **contracts** address decentralization. Since CCAs cannot directly observe the network state, they rely on local observations to infer it. Beliefs formalize these inferences. Similarly, since CCAs cannot directly communicate with each other, they (implicitly) exchange information with each other by manipulating observable congestion signals. For this to work, CCAs "agree" on a common way to interpret these signals. Contracts formalize such communication mechanisms.

Together, the network model and contracts capture the prior assumptions that a CCA makes. The model represents assumptions about the environment (e.g., how packets are delayed or dropped). Contracts represent assumptions about the behavior of other flows and how they affect the network state and, in turn, the local observations of other flows. These assumptions define a relationship between the global system state and the local observations of a flow. Beliefs "invert" this relationship to infer the possible network and flow states consistent with the observations.

Beliefs and contracts guide two key questions any controller must answer: (1) what state to maintain, and (2) what actions to take for a given state. For instance, we show that a CCA must (a) maintain beliefs and continually reduce uncertainty in what it "believes" to be true about the network state, and (b) modulate rate in a way that obeys its contract to ensure compatibility with other flows. Such guidance substantially constrains the design space of controller, making systematic exploration of controller designs computationally tractable. This enabled us to design, CCmatic, our tool that automatically synthesizes formally verified CCAs. In follow up work, we improved upon CCmatic to design Syntra that enables joint synthesis of video streaming and congestion control algorithms that perform robustly across challenging network conditions.

Outside of enabling automation and structured design, these abstractions help us independently reason about CCAs. For instance, we found that many of the known and unknown tradeoffs in congestion control stem from uncertainty. Beliefs and contracts helped discover and quantify these tradeoffs. Unaware of these tradeoffs, existing CCAs often make implicit choices that lead to suboptimal performance along one or more dimensions. We hope that our framework will empower future designs to intentionally navigate tradeoffs, rather than unknowingly suffer from them.

To summarize, this thesis contributes the abstractions of beliefs and contracts that make the approach of performance verification using non-deterministic network modeling tractable. This allowed us to break new ground in congestion control. We uncover new tradeoffs and design novel CCAs that solve long-standing open problems, in support of the following statement.

Thesis statement. It is possible to design controllers that provide formally verified performance guarantees even under challenging network conditions.
The key to making this tractable is formally reasoning about uncertainty in global network state through the abstractions of beliefs and contracts, which bring structure that vastly simplifies controller design and analysis.

1.1 Overview of contributions (Tools and Abstractions)

We provide an overview of the beliefs and contracts abstractions, our synthesis tools CCmatic and Syntra. Then we outline that the results (new CCAs and tradeoffs) that these tools allowed us to produce.

1.1.1 Beliefs: Inferring network state

Beliefs address partial observability by formalizing how a CCA infers unknown global network state from local observations. Existing CCAs derive numerous ad-hoc statistics from their local observations to guide rate decision such as derivatives, integrals, or moving averages of sending rates, ACK rates, packet loss and delay signals. There is no consensus on which statistics to maintain. These statistics implicitly track latent properties of the network path; for instance, ssthresh in Reno estimates a lower bound on the network's bandwidth-delay product, while BBR explicitly estimates bandwidth and propagation delay.

We formalize this intuition by defining the *belief set* as the set of all possible network paths (described by parameters like link rate and buffer size) and instantaneous states (e.g., queue occupancy) that are consistent with the CCA's history of observations.

There are no ad-hoc decisions in computing this set. We show that beliefs can be mechanically derived from the network model we want to design CCAs for. Specifically, the network model describes the relation between network state and CCA's local observations. We can "invert" this relation to derive possible network states given local observations. This inversion automatically answers key questions that have confounded CCA researchers, e.g., "how long/large should capacity probes be" or "how to estimate amount of queue buildup".

Beliefs have two properties that allow us to reason about CCAs and simplify their design.

1. We show that it is *necessary* for any performant CCA to shrink the size of the belief set, i.e., reduce uncertainty in the possible paths it could be running on, e.g., probe to check if the link rate could be higher.

   This helps us in two different ways. First, it makes CCA design scalable by giving us an intermediate "short-term" objective to guide rate decisions. Typically, control objectives are specified over long-term executions (e.g., long term utilization or delay). Beliefs give us an intermediate objective of shrinking them allowing us to quickly evaluate which rate actions are good vs bad.

   Second, beliefs give us a way to prove impossibility results. Since shrinking beliefs is necessary, if we can show that for a certain belief set, the only way to shrink its size is by violating some objective (e.g., incur losses), then we obtain a fundamental tradeoff.

2. We formally prove that the belief set is a *sufficient* set of statistics for a performant CCA to consider, i.e., if a CCA can ensure a certain performance property, then a "belief-based CCA" can also ensure it, where a belief-based CCA is one whose sending rate is a pure function of the belief set.

   This allows us to convert CCA design from synthesizing stateful programs into pure stateless functions allowing us to use program synthesis techniques to automatically design CCAs.

1.1.2 Contracts: Coordinating between flows

Contracts formalize how flows coordinate with each other to address decentralized nature of congestion control. While CCAs can use probes to infer network state as link capacity or queue buildup, there is no way for a flow to unilaterally infer how many flows it may be competing with (§ 5.1). To work around this, CCAs (implicitly) coordinate with each other by encoding fair shares into observable signals. For instance, Reno uses loss rate to coordinate fair shares, where each endpoint transmits at a rate $\propto 1/\sqrt{\texttt{loss rate}}$ [89].

We call such communication mechanisms *contracts*. To our knowledge, all existing CCAs that share bandwidth fairly incorporate such a map in their design, either implicitly (e.g., Reno [89], Vegas [87]), or explicitly (e.g., TFRC [48], Swift [73], Poseidon [112]). CCA contracts differ in the signals they use and their shape (e.g., steeper vs gradual). For example, delay-based CCAs encode fair share as: "$1/\texttt{delay}$" [18, 27, 115], "$1/\texttt{delay}^2$" [73], or "$e^{-\texttt{delay}}$" [20].[2] Contracts effectively parameterize the space of CCAs without fully specifying a CCA. For instance, the entire family of

[2]Throughout this thesis, we use delay to mean a queuing delay estimate (e.g. $\texttt{RTT} - \min \texttt{RTT}$).

TCP-friendly CCAs [21, 48, 58, 63] "follow" Reno's contract (i.e. send at rates $\propto 1/\sqrt{\texttt{loss rate}}$), but they differ in other aspects, such as stability and convergence time properties.

The diversity in existing contracts motivated us to examine their performance implications. In doing so, we discovered a surprising result: contracts—a choice that most CCAs don't even make explicitly—fully determine most steady-state performance metrics in congestion control. And since one design element determines multiple metrics, we find these are at odds with each other (described below in § 1.2.3). These tradeoffs arise because of the decentralized nature of CCAs, if flows could communicate with each other these tradeoffs would disappear.

Furthermore, the linkages between properties of contracts and CCA performance allowed us to identify previously unknown scenarios and (avoidable) design mistakes that cause severe performance degradation or suboptimal performance in a wide range of CCAs (§ 5.1, § 6.3). We find that CCAs whose contracts have *extreme shapes* (e.g., logarithmic, exponential), *shifts* (e.g., $\texttt{rate} = 1/(\texttt{delay} - \texttt{c})$ vs. $\texttt{rate} = 1/\texttt{delay}$), *clamps* (e.g., $\texttt{delay} = \max(\texttt{c}, 1/\texttt{rate})$), or *intercepts* can starve flows. We observe this in BBR [30], ICC [68], and Astraea [84]. Further, CCAs with an explicit contract (e.g., Swift [73]) do not need AIMD (Additive Increase, Multiplicative Decrease) updates to reach fairness. Instead, MIMD (Multiplicative Increase, Multiplicative Decrease) updates allow exponentially fast convergence to both fairness and efficiency (§ 6.4). Finally, we show that having fixed end-to-end thresholds, e.g., AIMD on delay [24, 78, 103], causes starvation on topologies with multiple bottlenecks.

To avoid design mistakes and enable the deliberate selection of Pareto-optimal tradeoffs, we develop blueprints for CCA design and analysis (§ 5.1). These adopt a "contracts-first" approach, where the designer explicitly chooses the CCA's contract, making the resulting trade-offs and pitfalls immediately evident (after a few to no algebraic steps) and ensuring that design choices are both intentional and well-understood. In contrast, even though existing CCAs exhibit contracts, these are not explicitly chosen but are an emergent property of the design leading to unintended performance degradation.

1.1.3 CCmatic: Automatically synthesizing and analyzing CCAs using beliefs

Beliefs made it computationally tractable to apply program synthesis techniques for automatically synthesizing CCAs with formal performance guarantees. We built CCmatic a computational tool that uses program synthesis to systematically solve the *search problem*:

> "Find a CCA in a *search space* that ensures given *performance properties* over all specified *network paths or scenarios*".

The sufficiency of beliefs allows us to define an exhaustive and tractable search space, necessity of beliefs allows us to define performance properties, and non-deterministic environment modeling allowed us to define the network paths.

Using CCmatic, we synthesize CCAs that guarantee performance across paths with jitter and shallow buffers, scenarios where existing CCAs struggle to even guarantee 1% utilization [19]. Despite being designed for theoretical "worst-case" links, the synthesized CCAs outperform or match existing CCAs on empirical links that resemble "average-case" networks. By design, the synthesized CCAs are short, modular, human-interpretable, and come with proven performance guarantees.

Further, experimenting with CCmatic, it sometimes reported that no CCA in the search space could meet the performance property, hinting that perhaps our performance property cannot be achieved. Despite the sufficiency of the belief set, this is not a definitive proof because CCmatic only explores a subset of belief-based CCAs due to computational limits. Nevertheless, using the sufficiency and necessity properties of the belief set, we prove a previously unknown fundamental tradeoff between loss and convergence time on shallow buffered networks that we describe below.

Crucially, it is hard for humans to manually reason through combinatorially many possibilities of interactions between the controller and environment complexities. By building CCmatic, this thesis shows that computers can instead do this reasoning while still producing results that can be interpreted at a human-level unlike control policies generated using machine learning. This thesis builds the key abstraction of beliefs that make such reasoning computationally tractable.

1.1.4 Syntra: Synthesizing joint controllers for video streaming and congestion control

In follow up work, we improved upon CCmatic to build Syntra. While CCmatic's scalability limits us to only explore congestion control, Syntra can jointly synthesize video streaming and congestion control algorithms by making decisions across multiple axes including video quality, frame rate, sending rate, and forward error correction.

Syntra generated controllers beat key state-of-the-art baselines including WebRTC-GCC [35], WebRTC-Vegas [27], and Salsify [50] on a variety of metrics. Syntra improves P95 one-way delay by over 6×, increases median video quality (SSIM) by more than 2 dB, and maintains a higher of frame rate even under challenging conditions such as network jitter. The central idea behind Syntra is to avoid synthesizing a full controller program. Instead, Syntra computes the optimal action for each belief state and then uses imitation learning to construct a human-interpretable controller that reproduces these optimal actions.

Apart from enabling joint controllers, this formulation gave us two interesting insights about network control. First, we found that formulas describing belief computations—while derived from the network model—are exponentially larger than the formulas describing the network model. This is not an artifact of our approach, belief computations are inherently complex. This may explain why congestion control has been so hard: our community has been trying to estimate latents (compute beliefs), a task that is fundamentally hard, using heuristics. Systematically reasoning about uncertainty paved the way for us to finally solve this task.

Second, we found insights about the structure of the optimal network controller. If the control specifications (network model and objectives) can be expressed using formulas in linear real arithmetic, which is true for our specifications, then the optimal controller can also be expressed in linear real arithmetic. This is a striking finding. Our community has built algorithms like Cubic [59] that use non-linear mathematical operators. Such complicated operators are unnecessary unless the specification requires them. This insight played a key role in making Syntra scalable and producing human-interpretable controllers.

1.2 Overview of contributions (Results)

This thesis breaks new ground in congestion control by settling two key open problems: (1) dealing with jitter and shallow buffers in the single-flow setting, and (2) dealing with jitter in the multiple flow setting. Within these, we discover two new sets of tradeoffs (negative results), and two new sets of CCAs (positive results).

1.2.1 Tradeoff: packet loss vs convergence time

On networks with jitter and shallow buffers, there is a tradeoff between amount of packet loss and convergence time: the faster a CCA wants to converge the more loss it must risk. Intuitively, the combination of short buffers and jitter creates uncertainty in delay measurements, forcing CCAs to rely on loss-based signals. If CCAs probe for bandwidth aggressively, they converge faster but risk losses. If they probe conservatively, they mitigate losses but converge slowly. We quantify this relationship in § 4.2.2 in Chapter 4.

While this tradeoff may seem intuitive, formalizing it requires careful treatment (§4.2.2). For instance on idealized networks without jitter, techniques like packet trains [74, 75] can allow CCAs to infer link capacity quickly without compromising convergence time or risking packet loss. However, such techniques rarely work in practice [43].

We argue that in contrast to the idealized models, the non-deterministic models provide a more faithful representation of real network behaviors. The trade-offs exhibited by CCAs on real networks align more closely with those on non-deterministic models, whereas, such tradeoffs disappear under idealized network models. The non-deterministic models allowed us to rigorously quantify these tradeoffs that practitioners have observed empirically, thereby bridging the gap between theoretical reasoning and real-world behavior.

1.2.2 CCmatic CCAs: First class of loss bounding CCAs

While the loss vs convergence tradeoffs define theoretical upper limits on achievable performance, we used CCmatic and beliefs to synthesize CCAs that explore different points on the loss–convergence Pareto-frontier, demonstrating that these limits are tight and attainable in practice.

All existing CCAs incur significant loss when operating under jitter and shallow buffers and lie far from the Pareto-frontier (§ 4.2.4). This motivated us to explore if we can design CCAs that match the Pareto-frontier. CCmatic, our computer-aided synthesis tool, allowed us to automatically design such CCAs.

CCmatic discovered an interesting coordinated pattern of probing and draining that enables a CCA to shrink beliefs (converge towards available bandwidth) while simultaneously bounding the amount of packet loss it risks. Beliefs allow the controller to determine exactly how large in bytes and how long in seconds each bandwidth probe should be—enough to infer bandwidth but not so large as to trigger excessive loss. These probes naturally build up queues, and gathering further information about bandwidth requires additional probing. Here, draining plays a crucial role: after each probe, the controller reduces its sending rate to drain the accumulated queue, where beliefs again tell exactly how much queue may have built up and what rate is required to empty it.

The algorithm consists of a parameter representing the amount of loss it is willing to tolerate. By tuning this parameter, we can obtain different points on the Pareto-frontier. We describe this class of algorithms in § 4.2.2.1. Crucially, decades of CCA research failed to produce such algorithms; this wouldn't have been possible without the formalization of beliefs and the computer-aided synthesis approach provided by CCmatic.

1.2.3 Tradeoff: robustness and fairness vs congestion and generality

Contracts fully determine four key congestion control metrics: (1) *robustness* to noise in congestion signals, (2) *fairness* in a multi-bottleneck network, (3) amount of *congestion* (e.g., delay, loss), and (4) *generality* (e.g., the range of link rates the CCA supports).

We find that robustness and fairness are at odds with congestion and generality, and we cannot have best in all four metrics. Robustness and fairness are better with gradual contracts functions (e.g., Vegas [27, 87] "`1/delay`"), while congestion and generality are better with steeper contracts (e.g., Swift [73] "$\texttt{1/delay}^2$") (§ 6.1).

Additionally, we find that for a fixed contract, tolerating more congestion allows supporting a larger bandwidth range. In that sense congestion and generality are also at odds. Contracts also determine link utilization and total throughput. Our analysis subsumes these under fairness definitions (§ 6.1), where throughput and fairness are known to be at odds [93].

Similar tradeoffs have been explored before. Arun et al. [20] discuss a special case where full generality precludes simultaneous robustness and bounded variation in congestion. We find that their proposed workarounds cause unfairness due to other reasons (§ 6.1, § 6.3). Zhu et al. [124] show that delay-based CCAs cannot simultaneously ensure fixed delays and fairness. That is, if a delay-based CCA maps the same delay value to multiple rates, it does not have a contract, and cannot be fair. The NUM (Network Utility Maximization) literature [71, 85, 87, 106] studies a subset of these tradeoffs for individual CCAs. To our knowledge, we analyze a wider range of tradeoffs and also generalize them to many CCAs.

1.2.4 FRCC: First starvation-free CCA

All existing CCAs starve flows in the presence of jitter and multiple competing flows (see right side of Fig. 1.1, Chapter 2, and [20]). Contracts help us understand why this occurs and build FRCC—the first CCA that avoids starvation despite jitter. Since CCAs coordinate fair rates by encoding them into congestion signals, even small noise in the congestion signals translates to large errors in inferring fair rates, causing flows disagree on the fair rate creating unfairness.

FRCC incorporates two key ideas to work around this issue. First, rather than coordinating fair *rates* through congestion signals, FRCC coordinates flow count or fair link *fractions*. Flows independently estimate the bottleneck link capacity to know if the fraction of link they *currently* consume is more or less than the *target* fraction encoded in the congestion signals. Flows adjust their `cwnd` to consume the same fraction of the link as communicated in the congestion signals. This reduces the bits of information communicated through congestion signals, thereby reducing the impact of noise.

Second, capacity estimation is notoriously hard (see [43] and § 7.4.3), often requiring large probes for high accuracy. Our key idea is that FRCC only needs the capacity estimation to be accurate enough for flows to know if their current link fraction is above or below the target fraction. We precisely compute how large and long our probes need to be to build an accurate enough capacity estimator. This creates the delay variations that recent work showed are necessary to avoid starvation [20] under jitter.

1.3 Dissertation plan

Roadmap. In Chapter 2, we (1) motivate the network scenarios we target (e.g., jitter and shallow buffers) including how we mathematically model them, and (2) describe the performance objectives we aim to optimize, such as latency, utilization, and fairness.

We organize the main content into two parts corresponding to (I) single-flow and (II) multi-flow congestion control. In the first part, Chapter 3 introduces beliefs, which enable reasoning about partial observability. Building on beliefs, Chapter 4 presents CCmatic, our tool for automating CCA synthesis in the single-flow setting, along with the new CCAs and tradeoffs we uncovered using beliefs and CCmatic. We briefly describe Syntra towards the end of Chapter 4 and refer interested readers to [97] for a full description.

The second part turns to multi-flow settings. In Chapter 5, we introduce contracts, which enable reasoning about decentralization or how CCAs coordinate fairness. Then, Chapter 6 uses contracts to identify a variety of tradeoffs and design mistakes that cause poor performance in existing CCAs, and presents blueprints for principled CCA design and analysis. Finally, Chapter 7 applies these insights to design FRCC, the first CCA to resolve the long-standing open problem of starvation in end-to-end congestion control [20].

Organization. We split our description into two parts because of the following two reasons. First, in single-flow congestion control, there is no uncertainty due to decentralization—the controller does not have to reason about other flows. This reduces the computational hardness of the problem from NEXP-Hard to PSPACE-Hard[3], and consequently changes the tradeoffs that CCAs must consider. There are several situations where a flow is alone in its bottleneck queue [28] (e.g., cellular networks). In such cases, the insights and designs developed for single-flow congestion control remain directly relevant and sufficient.

Second, this organization mirrors the progression of our research. We began by developing beliefs to address what state a congestion controller should maintain in single-flow settings.

[3]Remy's [116] formulation changes from Dec-POMDP to POMDP when going from multi-flow to single-flow settings, where Dec-POMDP stands for Decentralized Partially Observable Markov Decision Process [22].

In these cases, the network model specifies the relationship between network state and observations, which we can invert to infer latent state from observable signals. Extending this reasoning to multi-flow scenarios introduced a new layer of uncertainty: to infer state, each flow must also make assumptions about the behavior of others. This challenge led us to introduce contracts, which formalize these assumptions and enable reasoning about coordination among decentralized controllers.

Currently, our abstractions of beliefs and contracts remain somewhat distinct. In our present formalism, contracts are too strong: once a contract is fixed, there is little flexibility left for innovation in CCA design, and belief computation becomes trivial, reducing the need for the full abstraction of beliefs. Moving forward, we aim to refine the abstraction of contracts to make them weaker and more expressive, and to unify them with beliefs to enable automated CCA design even in multi-flow settings. We outline ideas along this thread and other future directions in Chapter 8.

Summary. This thesis contributes beliefs and contracts, which are powerful abstractions to reason about network control. These (1) help discover and intentionally navigate performance tradeoffs, (2) identify and avoid controller design mistakes, (3) enable rapid synthesis of performant, bespoke CCAs that are provably performant by construction, and (4) inspire new approaches to address long-standing open problems in network control—bringing our community closer to truly "solving" congestion control.

1.3.1 Previously published material

Chapter 3 and Chapter 4 revise the previous publications: [8] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Automating network heuristic design and analysis. In HotNets, New York, NY, USA, 2022; and [9] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Towards provably performant congestion control. In NSDI Santa Clara, CA, April 2024.

Chapter 5 and Chapter 6 revise the previous publication: [10] Anup Agarwal, Venkat Arun, and Srinivasan Seshan. Contracts: A unified lens on congestion control robustness, fairness, congestion, and generality. arXiv preprint arXiv:2504.18786, April 2025.

Chapter 7 revises the previous publication: [11] Anup Agarwal, Venkat Arun, and Srinivasan Seshan. FRCC: Towards provably fair and robust congestion control. In NSDI 26, Renton, WA, May 2026 (forthcoming).

§ 4.5 in Chapter 4 and § 8.3.4 in Chapter 8 are adapted from: [97] Jia Pan, Anup Agarwal, Isil Diling, and Venkat Arun. Syntra: Synthesizing cross-layer controllers for low-latency video streaming. In NSDI, Renton, WA, May 2026 (forthcoming).

1.3.2 Artifacts

The code for the tools and algorithms built in this dissertation is available at:

1. Chapter 3 and § 4.5 in Chapter 4: `https://github.com/tonyPan123/syntra-experiment`.

2. Chapter 4: `https://github.com/108anup/ccmatic`.
3. Chapter 5 and Chapter 6 are more theoretical and do not have any significant code artifacts. One can use open-source network emulators (Pantheon [120], mahimahi [94]) and simulators (HTSIM [5]) to reproduce the results in these chapters.
4. Chapter 7: `https://github.com/108anup/frcc`.

Chapter 2

Target network scenarios and models

"The purpose of abstraction
is not to be vague, but to create a new semantic level in which one can be absolutely precise."

— Edsger Dijkstra

2.1 Goals

In this thesis we specifically address the following network scenarios that have been key pain points for past congestion control algorithms. On these scenarios, we desire CCAs that provide guarantees on metrics like throughput/utilization, latency, packet loss, convergence time, and fairness.

2.1.1 S1. Non-congestive delays (or *jitter*) with a single-flow

Delays can occur due to reasons unrelated to congestion [20, 55], e.g., delayed ACKs, ACK aggregation, OS scheduling, delays at the MAC and physical layers. This hinders CCAs from bounding end-to-end delays while ensuring high utilization. Traditional loss based CCAs [58, 59, 63, 107] fill up queues until they experience a loss and cannot bound delays. Delay based CCAs [18, 27, 30, 42, 90, 116, 117] use variation in measured round-trip times (RTTs) to estimate congestion. Jitter can cause these CCAs to mis-estimate congestion and send at a rate as much as 10× away from the correct rate [19, 42, 90, 120]. For instance, jitter can trick BBR [30] and Copa [18] into achieving near-zero utilization [19]. Recent learning-based CCAs (§4.3) also do not explicitly consider jitter.

2.1.2 S2. Shallow buffers with jitter with a single-flow

Paths with shallow buffers are common on the internet [33, 34, 46], because they prevent bufferbloat and help manage cost/area/power of routers [15, 44]. On such paths, it is challenging to maintain utilization while avoiding excessive losses. Traditional loss-based CCAs, including Cubic [59], get poor utilization in single flow cases ([111], §4.2.4). ACK-clocked CCAs, despite

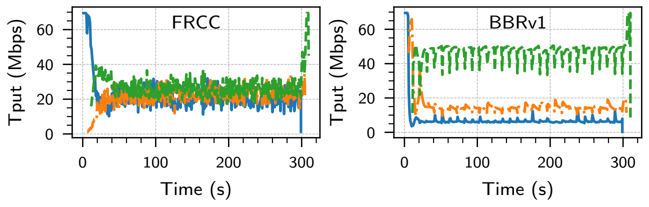


Figure 2.1: Throughput (Tput) of three flows with round-trip propagation delay (RTprop or R) of 10, 20 and 30 ms. BBR starves the (blue) flow with $R = 10$ ms.

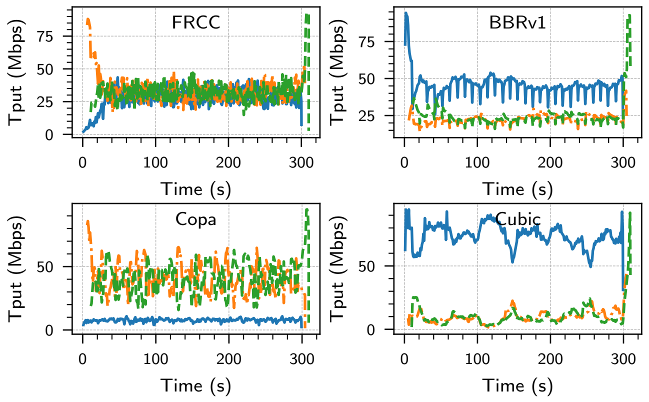


Figure 2.2: Three flows with RTprop = 32 ms. The blue flow experiences 32 ms of ACK aggregation. Copa and Cubic starve flows.

pacing, send bursts of packets due to jitter (e.g., ACKs-aggregation [55]) which risks excessive losses [19]. BBRv1 [30] is paced, but uses aggressive probes that incur O(BDP) losses periodically [32]. BBRv2 [33] and BBRv3 [34] incur lower loss on average but still incur O(BDP) losses in some cases (§ B.6).

2.1.3 S3. Jitter with multiple-flows

All existing CCAs starve flows in the presence of jitter and multiple competing flows. For instance, Fig. 2.1 and Fig. 2.2 show how Cubic [59], BBR [30] and Copa [18], three widely deployed CCAs, experience extreme unfairness between flows as a result of differences in ACK aggregation or round-trip propagation delays (*RTprops*)[1]; phenomena that are common on the Internet. § D.5 shows unfairness in other CCAs.

A recent result, the *starvation theorem* [20], shows that these are not isolated failures. The theorem states that any CCA with small self-induced delay or delay variation can starve in the presence of jitter (from sources like ACK aggregation or OS scheduling delays). To our knowledge, all existing CCAs that are buffer-bloat resistant (i.e., bound delays) cause little to no self-induced delay variations and, therefore, starve. This happens on realistic paths as shown above.

Given this result, we ask whether there exists an end-to-end CCA that can provably bound

[1]BBR's unfairness is different from RTT unfairness in traditional CCAs [59, 63]. For BBR, a small difference in RTprops leads to arbitrarily large unfairness as the link rate goes to infinity [20].

fairness, albeit at the cost of large (enough) self-induced delay and delay variations, while still bounding the maximum delay. This led us to design FRCC (Fair and Robust Congestion Controller), the first such CCA with formal guarantees of fairness and efficiency under the same pessimistic, worst-case model used by [20]. With this, we can be confident that our design is robust in any real networks that can be emulated by the worst-case model, which includes components like token bucket filters and ACK aggregation.

2.2 Non-goals

In this thesis, we do not address several properties that may be desirable for a CCA, including, coexistence with other congestion control algorithms or short/application-limited flows. Similarly, while we study the combination of jitter and shallow buffers in the single flow case, we only consider jitter and not shallow buffers in the multi-flow case.

We justify our scope with three reasons. First, the scenarios that we address are extremely challenging. Despite decades of research, all existing algorithms fail in these settings, often starving flows (e.g., Fig. 2.1, Fig. 2.2). To even begin addressing these issues, we had to invent new abstractions—beliefs and contracts—that make reasoning about such settings tractable. These abstractions revealed previously unknown fundamental tradeoffs (§ 4.2.2, § 6.1), and allowed us to systematically explore the design space, building CCAs on different points on the Pareto-frontier. Existing CCAs, unaware of the tradeoff, make sub-optimal tradeoffs or only explore a subset of useful points on the Pareto frontier.

Second, in general, our community has luxurious desires from CCAs, including coexistence with legacy TCP [18, 57, 98, 113] and sophisticated fairness notions based on latency or application-visible metrics [41, 93, 99, 116]. However, today's CCAs fail to ensure even basic flow-level throughput fairness on simple dumbbell topologies even in homogeneous settings where *all flows are using the same CCA*. We need to solve these issues before pursuing higher ambitions.

Third, our work addresses an important and challenging set of scenarios, and establishes a formal methodology for further exploration. We believe that our tools generalize to scenarios beyond those explicitly considered in this thesis. For instance, in follow up work, we explore co-designing the control loops of adaptive bitrate (ABR) video streaming algorithms and CCAs. This setting introduces an additional challenge of coordinating decisions between the two loops For instance, we need to coordinate the ABR video quality and the CCA sending rate ensuring CCA has enough data to probe for capacity without harming the video application. We elaborate on this effort in Chapter 8.

Similarly, for coexistence, contracts describe how CCAs coordinate with each other by encoding information in congestion signals. At a high level, this suggests that for coexistence, CCAs must align on the semantics of how they encode and decode such information. Building on this intuition, we aimed to prove that contracts are necessary and sufficient for fairness or coexistence between CCAs, i.e., that two CCAs are fair if and only if they have the same contract. Our current formalism of contracts does not yet allow us to prove or disprove this conjecture. We hope to extend this framework in future work (Chapter 8), which may enable reasoning about such coexistence-style settings.

C – bottleneck link rate [r]	$\theta(t)$ – instantaneous inflight [b]
R – round trip prop. delay [t]	$q(t)$ – inst. bottleneck queue [b]
β – bottleneck buffer size [b]	$qdel(t)$ – inst. queueing delay [t]
D – max per-packet jitter [t]	$\texttt{RTT}(t)$ – inst. round trip time [t]
$\texttt{MSS}$ – maximum segment size [b]	$S(t)$ – cumulative service [b]
$\texttt{BDP}$ – $C \cdot R$ [b]	$A(t)$ – cumulative arrivals [b]
β_s – buffer in seconds (β/C) [t]	$L(t)$ – cumulative loss [b]
T – time steps [unitless]	$[.]_L(t)$, $[.]_U(t)$ – inst. lower and upper bounds on parameter or state, e.g., $C_L(t)$, $C_U(t)$
$\lambda(t)$ – inst. sending rate [r]	

Table 2.1: Glossary of symbols. The square brackets show the units: bytes [b], rate [r], and time [t]. inst. = instantaneous. Inflight is bytes that are unacknowledged and not inferred as lost.

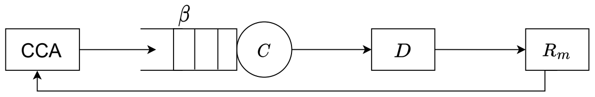


Figure 2.3: CBR-delay network model.

2.3 Network models

Network models mathematically describe how the network processes packets, e.g., how does it serve, delay, or drop packets. They allow us to (1) rigoursly describe what assumptions we make about the network, and (2) use computers to exhaustively reason about controller behavior on the network scenarios we target. This allows us to formally prove that all execution trajectories produced by the controller meet desired performance objectives.

We use CCAC [19] to succinctly express and efficiently explore the scenarios in §2.1. CCAC uses a single bottleneck abstraction to summarize the cumulative effects multiple elements on a network path. It uses bounded model checking [38] to provide a trace of CCA execution under various network behaviors.

CCAC also proposes the CCAC network model. Our investigations revealed that CCAC expresses behaviors that are perhaps too adversarial for any CCA to handle (§4.2.1, §4.2.2). So, we explore two other network models that are *weaker*, i.e., they are less challenging from the CCA's point of view as they capture strictly fewer behaviors. We briefly describe these models and use notation from Table 2.1. Note, if a CCA works on a stronger model then it also works on a weaker model, and an impossibility result for a weaker model holds for a stronger model.

CBR-delay. This is motivated from [20]. It abstracts the network as a constant bit rate (CBR) box followed by a non-deterministic delay (or jitter) box, and a propagation delay of R seconds (shown in Fig. 2.3). The CBR box has a constant (over time), but arbitrary bottleneck bandwidth of C bytes/second, and a buffer of size β bytes. It expresses the queueing or *congestive delays* (and losses) at the bottleneck queue in a network path. The delay box can add up to D seconds of delay *non-deterministically.* **Note, non-determinism is different from randomness or stochastic-**

ity (e.g., uniform random delays). Non-determinism allows the network to *arbitrarily* inject bursts and provably express the cumulative effect of various sources of jitter [19, 20, 25, 80]. Non-determinism (as opposed to randomness) can express non-congestive delays that may have causal effects or correlations.

CCAC [19]. It includes all behaviors captured by CBR-delay. Additionally, CCAC can non-deterministically accept a burst of packets without building up a queue, effectively hiding congestive delays/losses even when the CCA is sending above the link rate. In contrast, CBR-delay can inject non-congestive delays but not non-deterministically hide congestive delays. This is a crucial and previously unknown distinction that changes the tradeoffs that CCAs must make (§4.2.1, §4.2.2).

Ideal link. It cannot add any jitter to packets. It is simply a FIFO queue, with a constant (but arbitrary) bandwidth and propagation delay. Several theoretical analysis [21, 37, 89, 96, 122] have used similar modeling. We study this to compare CCAs designed for the ideal link vs. stronger models.

Variations in link rate over time. Like CCAC, all the models assume a fixed link rate over time. We use CCAC's approach to express variations in link rates. CCAC and CBR-delay express short-term link rate variations using jitter. All the models express long-term link rate variations by arbitrarily choosing initial conditions. E.g., a trace that begins with a high congestion window (`cwnd`) relative to C emulates a scenario where link rate decreased. Alternatively, a large initial queue buildup, and low `cwnd` can emulate a case where the link rate decreased and the CCA backed off, but the queue has not drained. One can stitch such traces together to explore longer executions with potentially multiple link rate variations (§4.1.2).

Formal definition. Mathematically, we view a network model as a relation that relates ⟨`path`, `state`, `CCA_action`⟩ to ⟨`next_state`, `CCA_feedback`⟩. Note, due to non-determinism, the relation may map a CCA action on a given path and state to multiple feasible next states and feedbacks. The network model also defines a set of initial states, and the domains of path, state, action, and feedback.

For example, in the CBR-delay model, a path is described by the parameters: link rate, propagation delay, amount of jitter, and buffer size, e.g., $\langle C, R, D, \beta \rangle$; and state by: bytes in the bottleneck queue (q) and bytes in flight (θ), e.g., $\langle q, \theta \rangle$. The model's relation is defined by constraints, such that feasible solutions to the constraints are the tuples in the relation.

Part I

Single flow

Chapter 3

Beliefs: Inferring network state

This chapter introduces our abstraction of *beliefs*, which enables reasoning under partial observability. Beliefs formalize what a controller can infer about the underlying network state from its local observations. Traditionally, CCAs do not reason about unknown network state in a principled way. Instead, they are designed as reactive control loops that optionally maintain a collection of ad-hoc state variables—delay gradients, loss rates, ACK rates, and more—with no consensus on which quantities should be tracked.

The problem is that these ad-hoc variables are often insufficient to make correct rate decisions. For example, an inflated RTT measurement may be caused by a decrease in link capacity, self-induced queueing, or measurement noise. The optimal control response differs in each case, yet from the RTT alone we cannot disambiguate between these possibilities to take the appropriate action.

Recent work [30, 116] suggests that CCAs should instead attempt to estimate latent network parameters such as link capacity, propagation delay, buffer size, queue buildup, etc. Beliefs formalize this intuition. We argue that the state maintained by a CCA should explicitly capture *uncertainty* in these latent parameters. Concretely, CCAs should maintain beliefs or the *belief set*: the set of all latent parameter combinations consistent with its observations. Fig. 3.1 illustrates such a set. Each point on the blue surface corresponds to a possible parameter vector, e.g., $\langle \text{Capacity} = 100 \text{ Mbps}, \text{RTprop} = 10 \text{ ms}, \text{Buffer} = 10 \text{ KB} \rangle$, that could have produced the controllers observations. Because there is uncertainty in what we can infer, we get a set *set* of multiple possibilities rather than a single point estimate.

We argue that all CCAs should maintain the belief set to reason about partial observability for three reasons. First, we show that beliefs, unlike ad-hoc state variables, contain sufficient information to guide rate decisions. Formally, we prove that if any deterministic CCA can achieve a performance objective using the full history of observations, then a *belief-based* CCA—one that consults only the belief set—can also achieve the same objective.

Second, beliefs provide a canonical way to estimate latent parameters. In our community, there has not been any consensus on how CCAs should estimate quantities such as capacity or propagation delay: how long should ACK rates be measured, how large should probes be,

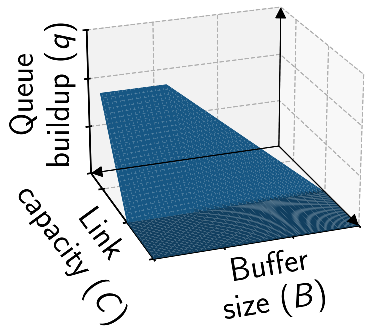


Figure 3.1: An example belief set. The blue surface shows all the possible latent network parameters that are consistent with the controller's local observations. We only show three dimensions for visualization.

etc.? Beliefs provide a principled answer. By simply *inverting* the network model, we can derive the set of latent states consistent with observations. Specifically, the network model describes: given some global network state, what are the possible local observations that the controller can make. Beliefs describe the inverse: given local observations, what are the possible states of the network. This inversion process automatically incorporates how factors like noise impact our measurements and answer how long measurements should be to tolerate noise.

This process resembles Bayesian inference, but rather than requiring a probability distribution over network behaviors—which is notoriously difficult to obtain [19]—beliefs operate over the *set* of possible behaviors, enabling worst-case reasoning. This also gives us a closed-form representation of the belief set that allow us to analytically derive tradeoffs that stem from partial observability.

Finally, beliefs help simplify CCA design and analysis by giving us an intermediate objective to guide rate decisions. Traditionally, control objectives like high utilization and low delay are specified over long executions. Beliefs provide an intermediate objective of "shrinking the belief set". We show that any performant CCA must take actions that *shrink* the belief set, i.e., reduce uncertainty in the latent network parameters. When a new flow begins, the controller knows nothing about its network path. As it sends packets and receives feedback, it learns something new of about the possible network states and the belief set shrinks. We prove that all CCAs, regardless of whether they explicitly maintain beliefs or not, must shrink beliefs over time. Because, if the belief set remains large—e.g., the link capacity could be 1 Mbps or 100 Mbps link—then the CCA cannot simultaneously ensure high utilization and low delay without first reducing uncertainty.

The sufficiency and necessity properties of beliefs allow us to prove fundamental results in congestion control. If for each belief set, we can find rate choices that result in shrinking the belief set while obeying other objectives (e.g., we can probe for bandwidth without incurring too much packet loss or delay), then the rate choice for each belief set gives us the CCA. Whereas, if there are belief sets for which no such rate choice exists, then we have a proof that no CCA can achieve the desired objectives.

We begin by motivating and illustrating beliefs through intuitive examples (§ 3.1), and then develop a formal framework (§ 3.2). We show how beliefs help us reason about controller behavior and bring new structure to controllers that simplifies their design. In the next chapter,

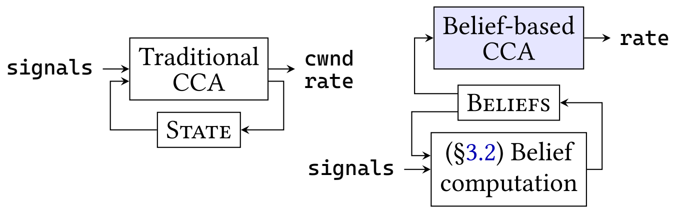


Figure 3.2: Beliefs standardize the state in congestion control.

we will show how the properties of beliefs make automated synthesis tractable, enable the design of novel CCAs, and allow us to discover and prove new tradeoffs in congestion control.

3.1 Beliefs through examples

Belief set example. Consider a hypothetical CCA that knows it is running on a simple link with constant round-trip propagation delay $R = 100$ ms, an infinite buffer $\beta = \infty$, and a constant, but unknown, bandwidth C MBps. Initially, the CCA could be running on *any* path on the Internet, i.e., C could be any non-zero value (e.g., 100 MBps). I.e., the CCA believes $C \in (0, \infty)$ MBps. Say the CCA has been sending at rate $\lambda = 10$ MBps, and observes that all packets are ACKed 100 ms after transmission (i.e., $\texttt{RTT} = R$). Such `RTT`s could be produced by any path with $C \geq 10$ MBps. Now, say the CCA increases λ to 15 MBps. If `RTT`s increase or losses happen, the CCA can conclude that $C \leq 15$ MBps. Combining this with CCA's past observations, we can update the belief set to $C \in [10, 15]$ MBps. Otherwise, if `RTT` remains at 100 ms, then $C \in [15, \infty)$ MBps.

Beliefs help reason about CCA behavior. We can compute beliefs for *any* CCA given its past observations on a network model. The belief set serves as a useful tool to reason about the performance of *any* CCA. For instance, if the CCA above believes that $C \in [15, \infty)$ MBps, then it needs to keep increasing its rate until it obtains an upper bound on C by deliberately causing losses or increasing `RTT`s. Otherwise, without an upper bound, the CCA risks arbitrarily low utilization because the actual link rate could be arbitrarily large, e.g. 1500 MBps. I.e.,

Lemma 3.1.1. *To avoid arbitrarily low utilization, a CCA needs to* shrink *the set of possible paths (the belief set) it could be running on by obtaining an upper bound on C.*

Beliefs bring structure to CCA design. In addition to it being necessary to shrink beliefs, we show in §3.2 that the belief set is the only information a CCA needs in order to decide sending rate. This is because the belief set is the only information a CCA needs to estimate the performance impact of its actions. This vastly simplifies CCA design (Fig. 3.2).

Traditionally, CCAs decide (Q1) what state (statistics) to maintain from input signals, and (Q2) sending rate. For instance, to answer Q1, CCA designers often consider "what does packet loss tell us about the state of the network?", "when can bandwidth be measured?", "what length of interval should be considered to sample bandwidth?" [30, 31].

Belief computation is uniquely determined by a network model and exhaustively derives all possible information about the network's path/state directly from the time series of CCA's sending and acknowledgment sequence numbers.[1] We no longer need to make ad-hoc decisions to answer Q1. This effectively decouples Q1 and Q2 and standardizes the state a CCA has to maintain. With beliefs, the CCA only decides the computation in the shaded box that maps beliefs to sending rate.

Beliefs give new insights on how to infer unknown network state. We also discovered new ways to use sending rate decisions to augment the information obtained from RTTs and ACK rates. This leads to better estimates of the network parameters (bottleneck bandwidth/buffer) and state (queuing or extent of congestion) (§4.2.2.1). We illustrate here with two examples.

Beliefs led us to discover new ways to use sending rate decisions to augment the information obtained from RTTs and ACK rates. This leads to better estimates of the network parameters (bottleneck bandwidth/buffer) and state (queuing or extent of congestion) (§4.2.2.1). We illustrate here with two examples.

Example 1. CCAs use the gap between instantaneous and minimum RTTs to estimate queueing delay. Non-queueing delays can create RTTs larger than the minimum RTT and cause a CCA to erroneously infer that a queue is built up. However, if the CCA has been sending at a low rate, then we know there is no queue buildup even if there is inflation in RTTs. In such cases sending rate choices provide a better estimate about queueing than RTTs alone.

Example 2. Say a CCA sent a burst of packets to probe for available bandwidth and observed that the probe did not incur any packet loss. Then we can conclude that either the buffer or the bandwidth is large enough to have accommodated the burst. I.e., both the buffer and bandwidth cannot be small as that would have incurred a loss. We cannot make such a conclusion by relying on ACK rate and RTT measurements alone. Due to jitter, a burst may not lead to an immediate increase in ACK rate ([19], §3.2) which traditionally would have allowed us to conclude bandwidth is large. Likewise, an inflation in RTTs could be due to jitter, and we cannot assume that the bottleneck buffer is large enough to accommodate the inflation in RTTs.

Our tool, CCmatic, automatically synthesizes CCAs that use such insights to make non-trivial decisions about when/how to probe/drain. E.g., it realized that draining is necessary not only to maintain low delay, but also to restrict losses when probing for bandwidth on paths with jitter and shallow buffers (§4.2.2.1).

3.2 Beliefs formally

Definition 3.2.1. *A **belief set** (or beliefs) for a given network model is the set of paths (and their latest states) that could have produced (according to the network model) the historical sequence of CCA's observations. It is the CCA's* belief *about the paths and states of the network it is running on.*

For example, for the CBR-delay model, the belief set is a set of tuples of the form $\langle C, R, D, \beta, q, \theta \rangle$. Such a tuple is in the belief set *if and only if* it can explain (according to the network model), the

[1]RTTs, ACK rates, and losses can be derived from sequence numbers.

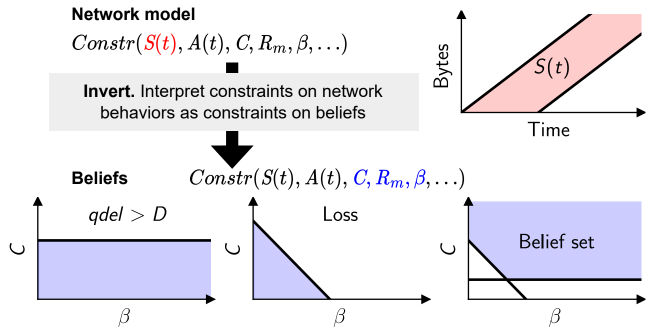

Network model
$Constr(S(t), A(t), C, R_m, \beta, \ldots)$
Invert. Interpret constraints on network behaviors as constraints on beliefs
Bytes
$S(t)$
Time
Beliefs
$Constr(S(t), A(t), C, R_m, \beta, \ldots)$
$qdel > D$
Loss
Belief set
C
β

Figure 3.3: Inverting the network model to compute beliefs. Depending on the CCA's observations, we may get different bounds on the belief set. The bottom three plots illustrate the constraints on the belief set we get depending on whether the CCA observed $qdel > D$, loss, or neither (for CBR-delay).

observations of the CCA thus far. We use the term observations to collectively refer to CCA's actions and feedback from the network.

Computing beliefs. We can "invert" the network model's relation to compute beliefs (Fig. 3.3). Specifically, the constraints of the model describe the feasible ways in which the network's state and feedback can evolve (e.g., how it services (delays) packets, drops packets, and builds queues), given the CCA's sending behavior, the network path, and the network's initial state. I.e., the constraints describe feasible combinations of $\langle S(t), L(t), q(t), \theta(t)\rangle$, $A(t)$, $\langle C, R, D, \beta\rangle$, and $\langle q(0), \theta(0)\rangle$. If we fix the observations, i.e., $\langle A(t), S(t), L(t)\rangle$, then the constraints describe the feasible paths and states that could have produced CCA's observations. This is the belief set. Each constraint of the network model gives us a constraint on the belief set. This is similar to conversion of a partially observable markov decision process (POMDP) into a belief MDP [69]. We perform the inversion §4.1.1 and §4.2.2.

As a quick example, for the CCAC model, in an interval of length T, the network can serve at a rate C and inject a burst of D seconds, i.e., $S(T) - S(0) \leq CT + CD$, where $S(t)$ is cumulative bytes served (delivered) until time t. If we invert this constraint on S to a constraint on C, we get $C \geq \frac{S(T)-S(0)}{T+D}$.

Shrinking beliefs is necessary. CCAs need to shrink the size of the belief set, i.e., infer the possible parameters and states of the network they are running on. We illustrate this using a family of lemmas (Lemma 3.1.1, Lemma 3.2.1) that show the dimensions along which beliefs need to shrink to ensure different performance properties.

Lemma 3.2.1. *To ensure an upper bound on queueing delay (qdel), a CCA needs to shrink the set of possible propagation delays the network could have.*

Consider a CCA that aims to ensure $qdel \leq 10$ ms on an ideal link. Say it has set `cwnd` = 10 MB, and observes that all `RTT`s are 100 ms, yielding an average throughput of `cwnd`/`RTT` = 100 MBps. Such `RTT`s can be explained by $C = 100$ MBps, and any $R \in [0, 100]$ ms, and $qdel = 100 - R$, i.e., $qdel \in [0, 100]$ ms. Now, say the CCA decreases `cwnd` to 5MB. If `RTT`s are still 100ms, i.e., RTTs do not decrease with cwnd, the CCA can conclude that R = RTT = 100 ms and $qdel$ = 0 ms. Otherwise, if RTTs drop to 50 ms, i.e., the average throughput is still 100 MBps, R and $qdel$ still remain in $[0, 100]$ ms.

Until a CCA decreases cwnd to the point that RTTs stop decreasing, it cannot obtain a lower bound on R, consequently it cannot ensure an upper bound on $qdel$.

Beliefs are sufficient. Beliefs allow a CCA to compute possible next state(s) and feedback(s) (and consequently potential future performance) for different rate choices. Due to this, a CCA does not need to look at any information other than the belief set when deciding its actions.

Theorem 3.2.1. *If there exists a* deterministic CCA *that ensures a performance property on a network model, then, there exists a* belief-based *CCA that ensures the performance property on the model.* Where, a deterministic CCA is a CCA whose actions are a function of the entire history of past observations (i.e., its actions and feedback) and a belief-based CCA is a CCA whose actions are a function of the belief set computed using the network model over the history of past observations.

For this theorem, we assume that the performance property is specified as a boolean valued function over a belief set and CCA's action on that set. We show in § A.1 how the properties we use can be expressed in this form.

We use this theorem to (1) synthesize CCAs as function of the belief set, and (2) to prove impossibility results, i.e., if there is no action that can ensure a performance property over all tuples in a valid belief set, then the performance property is not achievable. We use such arguments in §4.2.2.

§ A.1 gives a formal proof by constructing a belief-based CCA using the deterministic CCA. Here we give the intuition. We view congestion control as a 2-player (CCA vs. network) zero-sum game. The network tries to prevent the CCA from achieving its performance property. The CCA chooses its sending rate and the network delivers, delays, or drops packets. The only "rule" is that the network's actions must correspond to some path in the network model.

The belief set exhaustively summarizes the history of the game, making it *memoryless* (similar to the board state in chess). Beliefs serve as a "board" by meeting the two requirements: (R1) we can determine the set of feasible moves for the players from the board allowing us to enforce the game rules (Lemma A.1.1), and (R2) we can update the board by applying the moves (Lemma A.1.2). As a result, future progressions of the game (and any optimal strategies) depend only on the board (beliefs) irrespective of the history that led to the board.

3.3 Chapter summary

Beliefs address *partial observability* in congestion control. They answer (1) what state should CCAs maintain and (2) how CCAs should estimate latent parameters and state of the network. We no longer need to manually reason about "how long to measure throughput to estimate link capacity", or "how much to drain to estimate propagation delays". The answer lies in the network model itself. We showed that we can compute beliefs (estimate uncertainty in latents) by just inverting the network model.

We showed that beliefs are *sufficient*, i.e., they provably contain all information required to guide rate decisions, and that it is *necessary* for any congestion control algorithm—whether or not it explicitly maintains beliefs—to shrink the belief set (or reduce uncertainty in the network state).

In the next chapter, we will show that these properties (1) make automated program synthesis tractable, (2) enable the systematic design of novel CCAs with provable performance guarantees, and (3) uncover new fundamental tradeoffs in congestion control.

Chapter 4

CCmatic: Automatically synthesizing and analyzing CCAs using beliefs

In this chapter, we describe the design of our tool CCmatic that uses program synthesis for automatically synthesizing CCAs (§ 4.1). Program synthesis works similar to reinforcement learning (RL), where we try to iterate on CCA designs based on feedback from a verifier. In RL, the feedback is from a simulation environment that explores controller performance on a finite set of traces. Consequently, we have no guarantees on whether the controller performance generalizes beyond this finite set. In contrast, in program synthesis, the verifier uses bounded model checking [38] to "symbolically" explore all modeled behaviors. This yields proofs that the synthesized controller guarantees performance on all the modeled networks.

However, program synthesis does not scale out of the box and takes weeks to synthesize controllers even for very simple network scenarios [8]. Beliefs (Chapter 3) allowed us to make program synthesis tractable through the following two ways. First, beliefs allow us to convert control synthesis from a "stateful" to a "stateless" synthesis problem. Traditionally, CCAs maintain some state from their observations to decide their rate. Beliefs directly answer what state controllers should maintain and split stateful controllers into two parts: (1) belief computation (belief set or controller state as a function of observations) and (2) rate computation (rate as a function of belief set). Here, belief computation is a fixed piece of computation that is directly derived from the network model (Chapter 3). Consequently, we only need program synthesis to decide the rate computation part that is a much smaller design space to explore than designing the entire controller.

Second, beliefs allows us to speed up synthesis by quickly discarding bad CCAs. Typically, when evaluating CCAs, we need to measure their performance over a long enough time period, e.g., measure average utilization in steady-state. It becomes intractable to symbolically explore long execution traces (e.g., 100s of RTTs) with bounded model checking. With beliefs, we know that it is necessary for any performant CCA to shrink beliefs (Chapter 3). Consequently, instead of checking utilization-style metrics over long-executions (e.g., 100s of RTTs), we can just check if the candidate controllers shrink beliefs over short-executions (e.g., 6 RTTs) and speed up design iteration.

We use beliefs and CCmatic to design new CCAs that deliver performance on networks with

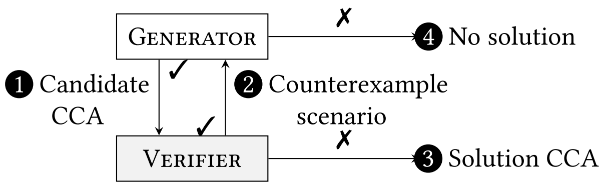


Figure 4.1: CEGIS loop (adapted from [6]).

jitter and shallow buffers where existing CCAs struggle to even ensure 1% utilization. In this process, we discovered and proved a previously unknown tradeoff between loss and convergence time on such networks. Specifically, the more loss a CCA is willing to tolerate, the faster it can converge to the capacity of the link. Intuitively if the algorithm is conservative, it can bound loss but converge slowly. All existing CCAs cause high amount of packet loss in this tradeoff space. They cause $O(\text{capacity})$ losses, i.e., losses that grow with the capacity of the link. You may think that an algorithm link Reno is conservative and should cause low loss. Unfortunately, due to ACK-clocking in Reno, a burst of ACKs causes it to transmit a burst of packets causing high loss. CCmatic allowed us to design the first CCAs that can provably bound loss to $O(1)$ independent of the capacity of the link.

We start by describing how CCmatic formulates controller synthesis as a program synthesis (§ 4.1) problem. We then describe how beliefs help build tractable controller search space (§ 4.1.1) and built short-term synthesis invariants to speed up synthesis (§ 4.1.2). Finally, we detail the new CCAs and tradeoffs that beliefs and CCmatic allowed us to discover (§ 4.2).

4.1 CCmatic design

CCmatic uses Counter-Example Guided Inductive Synthesis (CEGIS) [105] (a program synthesis technique) to synthesize (search for) CCAs given a specification (i.e., a network model and a performance property).

CEGIS iteratively generates a candidate CCA from a search space ❶, and finds a counterexample scenario (network path, initial state, and non-deterministic choices) that breaks the performance property for the candidate CCA ❷ (Fig. 4.1). The search space is pruned using the counterexamples (see below), and eventually the loop terminates if either (1) the verifier cannot find a counterexample (and thus, the CCA achieves the performance property) ❸, or (2) the entire search space has been pruned (no CCA in the search space can achieve the performance property) ❹.

We implement the generator and verifier using the constraint solver Z3 [40], by encoding the search inputs (i.e., search space, network models, and performance properties) into SMT (Satisfiability Modulo Theories) constraints in the theory of linear real arithmetic (LRA) [72]. In the generator, we only search for CCAs that pass unit checks (e.g. do not add bytes with seconds). For each counterexample, we add constraints to prune all CCAs that make the same sending rate choices as the candidate CCA on the counterexample. We explored encodings that prune more CCAs, these did not yield significant reduction in search time, and we omit their details.

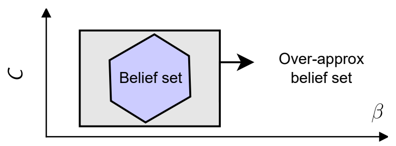


Figure 4.2: Over-approximating beliefs.

For encoding into SMT, we use beliefs to define the search space (§4.1.1) and a transition system abstraction to systematically define performance properties (§4.1.2). For the network models, we adopt the encoding proposed by CCAC [19]. It discretizes time and uses Network Calculus [79] style formulas to constrain how the network serves packets. Note, the synthesis happens offline. One can directly implement the synthesized CCAs in network stacks like the Linux kernel and QUIC [77]. For completeness, we provide details on the implementation of the verifier and generator in § B.1.

4.1.1 Belief-based CCA Template

In CEGIS, the search space is often specified using a template or grammar. The template has placeholders (or holes) that the generator fills to synthesize a concrete CCA. It describes the **inputs** (e.g., loss/delays signals) that the CCA takes and the mathematical/logical **operators** it can use to compute its outputs (i.e., rate and state). Due to Theorem 3.2.1, the templates do not need to describe state computations. They can just take beliefs as inputs and produce rate as the output to fill the shaded box in Fig. 3.2.

Inputs. We face two challenges with beliefs. First, the belief set may be a complicated object in the $\langle \texttt{path}, \texttt{state} \rangle$ space. For easier encoding, we *over-approximate* it using closed-form expressions (see below). Second, our network models do not directly model variations in link rate (§2.3). When the link rate varies, it can go outside the belief set. To address this, we re-compute beliefs using a recent history of observations (§4.1.1.2).

Over-approximating beliefs. We construct closed-form expressions representing bounds on the network parameters and states. For example, we use C_L and C_U to represent lower and upper bounds on the C values in the belief set, and pass these bounds as input to the CCA. By default, we pass upper and lower bounds on C (C_U, C_L), and $qdel$ ($qdel_U$, $qdel_L$) as a proxy for q. We describe how we compute them in §4.1.1.1. We assume the CCA knows R, and design CCAs for $D = R$ (i.e., jitter can be as large as R, e.g., due to WiFi ACK aggregation [55]). This enables efficient synthesis by discretizing time in units of R (as in CCAC [19]). In reality, a CCA does not know R. Nevertheless, we show in § A.2 that if we use a CCA designed for R equal to the minimum RTT seen thus far, we can guarantee performance because the synthesized CCAs are inherently robust to uncertainty in `RTT`s caused by jitter.

While the closed-form expressions may over-approximate beliefs, i.e., include extra paths that cannot produce the CCA's observations (Fig. 4.2), they never remove paths that can produce its observations. We use the verifier's (i.e., CCAC's) assistance to both validate correctness of the closed-form expressions and to derive them (§4.1.1.1).

Over-approximation (as opposed to under-approximation) does not break soundness. The CCA believes it could be running on extra paths, and needs to take actions that do not violate performance on the extra paths. However, over-approximation does break completeness, i.e., Theorem 3.2.1 no longer applies.[1] It may happen that we summarize two different histories using the same approximate beliefs even though the actual belief sets are different. A belief-based CCA is allowed to take different actions on the histories, but a CCA in the template is forced to take the same action. As a result, CCmatic may output "no solution in template" even though a belief-based CCA works. When this happens, we explore weaker network models where we can add additional beliefs (e.g., β_L, q_U) or tighten existing ones (§4.2.2.1). Despite approximations, CCmatic synthesizes novel CCAs (§4.2.1).

Operators. Some CCAs use nonlinear operators like cube root [59] or division [18]. Nonlinearities slow down SMT solvers [118]. Instead, we search for piece-wise linear functions to map belief bounds to sending rates, represented as (nested) if-else statements. The generator synthesizes the conditionals and expressions in these statements as linear combinations of the belief bounds. While CCmatic only searches for CCAs in the template, we are able to generalize CCmatic's insights to arbitrary CCAs, e.g., impossibility results in §4.2.2.

Listing 4.1: Belief-based CCA template

```
cond_i = $\boxed{?}C_U + \boxed{?}C_L + \boxed{?}qdel_U + \boxed{?}qdel_L +$
                    $\boxed{?}$MSS/$R$ + $\boxed{?}R > 0$
expr_j = $\boxed{?}C_U + \boxed{?}C_L + \boxed{?}$MSS/$R$
if (cond_1):  rate = expr_1;
elif (cond_2):  ...
else:  rate = expr_n;
rate = max(rate, MSS/R)  # Ensure +ve rate.
```

Summary. Our templates take the form in Listing 4.1. $\boxed{?}$ denotes holes to be chosen by the generator. Different templates have different number (and nesting) of the "`if`" conditions. To keep the search tractable, we restrict the domain of the holes to be a small finite set, e.g., $\{-3, -5/2, -2..., 3\}$.

4.1.1.1 Computing belief bounds

Queueing delay. Traditionally CCAs estimate queueing delay as $\texttt{RTT} - R$ [18]. However, this does not account for non-congestive delays. In the CCAC and CBR-delay models, $\texttt{RTT}(t) = R + qdel(t) + \texttt{jitter(t)}$, where $0 \le \texttt{jitter(t)} \le D$. Consequently, $qdel(t) \in [qdel_L(t), qdel_U(t)]$ where,

$$\begin{aligned} qdel_U(t) &= \texttt{RTT}(t) - R \\ qdel_L(t) &= \max(0, \texttt{RTT}(t) - R - D) \end{aligned} \tag{4.1.1}$$

[1]Note, Theorem 3.2.1 also does not apply because we only explore a subset of belief-based CCAs using CCmatic.

Link rate. BBR [30] estimates link rate as the measured ACK rate. Jitter can create transient variations in ACK rate and mislead this estimate. Using the verifier (CCAC), we computationally derive the set of link rates that can explain an average ACK rate of r^{ACK} (i.e., the CCA received $r^{ACK}T$ bytes in an interval $[t_1, t_2]$ of length $t_2 - t_1 = T$ seconds). We also derive this set analytically in § A.2.

We query CCAC for feasible values of r^{ACK} after fixing $\lambda(t) = \lambda$ (for all t). We repeat for different values of λ to obtain the bounds: $r^{ACK}T \in [C(T-D), C(T+D)]$ if $\lambda \geq C$, and $r^{ACK}T \in [\lambda(T-D), C(T+D)]$ otherwise. A CCA can be sure that $\lambda \geq C$ in two cases, (1) $\texttt{RTT} > R + D$ over the *entire* interval (or $qdel_L > 0$) (indicating non-zero queueing), or (2) there are losses in the interval (we verify this, see below). By inverting the bounds on r^{ACK}, we get bounds on C (algebraic steps in § A.2):

$$C_L(t) = \max_{0 \leq t_1 \leq t_2 \leq t} \frac{r^{ACK} \cdot T}{T + D} \qquad C_U(t) = \min_{0 \leq t_1 \leq t_2 \leq t} \frac{r^{ACK} \cdot T}{T - D} \tag{4.1.2}$$

Note, C_U is only computed over the intervals where $qdel_L > 0$ or loss > 0. We checked these calculations by asking CCAC if there is a trace and CCA (i.e., CCAC is free to choose $\lambda(t)$) for which $C \notin [C_L, C_U]$. CCAC returned UNSAT, confirming that no traces violate our calculation.

4.1.1.2 Handling stale beliefs

Our network models do not explicitly model variations in link rate. Such variations can make the beliefs **inconsistent (or stale)**, i.e., the network can take actions outside the belief set, leading the CCA to make bad decisions. For example, the link rate may decrease below C_L making the beliefs stale. The CCA may still transmit at rate C_L thinking that $C \geq C_L$, but cause losses due to the reduced link rate.

We "time out" beliefs periodically (e.g., every $10R$), and also when they become **empty (or invalid)**, e.g., $C_U < C_L$. On a timeout, we re-compute beliefs using the history of observations since the last timeout. When beliefs become inconsistent by a large margin, they become invalid quickly. However, if they are slightly inconsistent, they may remain valid. The periodic (speculative) timeouts helps make beliefs consistent in the second case. For example, say C decreases below C_L, i.e., $C = C_L - \epsilon$ for some $\epsilon > 0$. The beliefs become empty when $C_U < C_L$. $C_U = \frac{r^{ACK} \cdot T}{T-D}$, where r^{ACK} could be as large as $\frac{C \cdot (T+D)}{T}$. For $C_U < C_L$, we need $\frac{C(T+D)}{T-D} < C_L = C + \epsilon$. This can take arbitrarily long time for arbitrarily small ϵ.

We retain our performance guarantees as long as *any* of the following holds: (1) the network parameters change infrequently so that they become consistent on the periodic timeouts, (2) parameters change by large margin, so that beliefs become invalid and timeout, or (3) parameters change within the belief set (i.e., remain consistent). We may violate our guarantees if the parameters *frequently* (e.g., every R) change to values just outside the belief set. However, since the beliefs are slightly off (e.g., 5%), our guarantees will also only be slightly off. Note, in general, frequent changes in network parameters is a hard problem for end-to-end congestion control due to feedback delay. Our CCAs react to changing network parameters at similar timescales as existing end-to-end CCAs (§ B.4, § B.6).

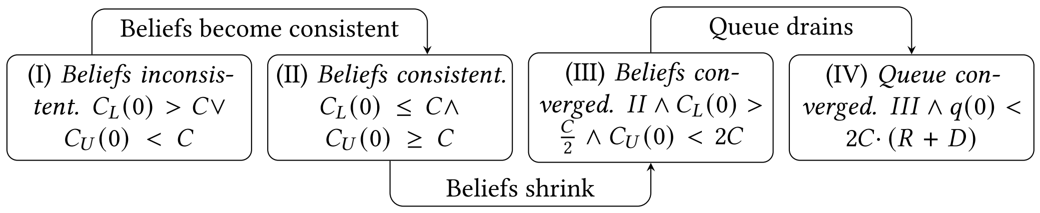

Beliefs become consistent
Queue drains
(I) Beliefs inconsistent. $C_L(0) > C \vee C_U(0) < C$
(II) Beliefs consistent. $C_L(0) \leq C \wedge C_U(0) \geq C$
(III) Beliefs converged. $II \wedge C_L(0) > \frac{C}{2} \wedge C_U(0) < 2C$
(IV) Queue converged. $III \wedge q(0) < 2C \cdot (R + D)$
Beliefs shrink

Figure 4.3: An example transition system. The formulas in the boxes (e.g., "$III \wedge q(0) < 2C \cdot (R + D)$") define the states.

Note, the periodic timeouts can interfere with a CCA's probes to estimate the belief set. We add constraints to prevent unnecessary timeouts. E.g., a CCA might be draining the queue and may not observe any delays/losses. If we recompute beliefs only using the recent history, there may be no upper bound on C, and the CCA would need to re-probe C_U from scratch. To prevent this, we only time out beliefs when the size of belief set is small (e.g., $C_U \leq 1.1C_L$), and we put bounds on how much the belief set can expand, e.g., $C_U(\texttt{now}) \leq 2C_U(\texttt{last\_timeout})$. These restrictions along with the timeout period affect how quickly CCAs tracks changes in link rate. § B.4 (Lemma B.4.2) studies this theoretically and § B.6 (Fig. B.7 and Fig. B.6) studies this empirically.

4.1.2 Transition system based properties

The verifier uses bounded model checking to explore short snapshots of a CCA's execution under the network model. On such snapshots, metrics like long-term average utilization may be violated (e.g., CCA may take time to ramp up sending rate when the link rate increases). We use CCAC's approach to prove lemmas over the snapshots and stitch them using mathematical induction (on time) to prove properties about arbitrarily long executions. To do this stitching, we need to define what "progress" CCAs need to make in a transient period (e.g., CCA ramps up sending rate until it meets a utilization objective and then maintains utilization). CCACs approach to define progress is ad hoc and CCA dependent. This becomes unwieldy as the number of objectives and CCAs increase. To systematically state lemmas, we use a transition system abstraction. We use it to build (1) proofs about the performance of CCAs, and (2) invariants used for synthesis.

Transition system. Fig. 4.3 shows an example transition system. The states are represented as symbolic boolean formulas (e.g., "beliefs are consistent", or $C_L(0) \leq C \leq C_U(0)$). For each state, users specify (1) transitions made in time 0 to T (e.g., "beliefs shrink": $C_L(T) > 1.5C_L(0)$) and (2) objectives during the period (e.g., "delay is at most D seconds": $\wedge_{t \in [0,T]} qdel(t) \leq D$). Note, users only declare "what" properties the CCA should ensure, CCmatic figures out "how".

A belief-based CCA under our network models typically makes the following transitions. Whenever the link rate changes, the beliefs can become inconsistent (stale). Eventually, the beliefs become consistent (due to §4.1.1.2), then the beliefs shrink (as this is necessary from §3.2), and finally, the CCA reaches the steady state (e.g., state IV) where it meets its steady-state objectives. Users can express both steady-state and transient objectives. For instance, high loss may be acceptable during slow start (e.g. in state I-III), but not for subsequent bandwidth probes (e.g., in state IV).

Proofs and encoding. We build a *lemma* for each state, e.g.,

$$\text{State I} \implies (\text{State I objectives} \land \text{State I transitions})$$

and take the conjunction (logical and) of these lemmas. This conjunction serves as the *proof of performance* for the CCA, as it *exhaustively* describes the states that the CCA visits and performance it ensures in each state. It is exhaustive because the disjunction of the transition system states is a tautology, i.e., covers all possible states. § B.4 gives the encoding of the lemmas and shows how they work together in a proof.

The transitions happen eventually, i.e., may occur over multiple steps. Due to this, the encoding of lemmas needs to ensure that transition progress adds up over stitched executions. For instance, we encode "beliefs shrink" as "at least one of $C - C_L$ and $C_U - C$ decreases" *and* "neither increases". The second literal, "neither increases", is required. Without it, both C_L and C_U can increase in one execution, and decrease in the subsequent execution and the progress does not add up. Such a CCA can meet the "at least one" criterion without ever transitioning to state III.

Synthesis invariant. We do not directly use the proof (i.e., the conjunction of lemmas) for synthesis. Instead, we build *under-specified* invariants that are *necessary* for a proof but not sufficient. We under-specify for two reasons. First, the lemmas contain constants that depend on the CCA, which is not known at the time of synthesis (e.g., $\mathbf{2}C$, $\frac{C}{\mathbf{2}}$, in Fig. 4.3). Second, before synthesis, we do not even know if there exists a CCA that can meet the lemmas. Under-specification allows us to synthesize reasonable-looking CCAs, which we then process *post synthesis* (see below).

To under-specify, we drop literals (inequalities) with unknown constants. Notice, that such literals define states III and IV (Fig. 4.3). Due to dropping, we cannot distinguish between states II-IV. We coalesce states II-IV into one, and allow the CCA to take any transition that is valid for states II-IV (effectively taking disjunction of the transitions). We similarly combine state III and IV objectives into `steady_state_obj`, and retain state II objectives as `transient_obj`. The result is Eq. 4.1.3 below. We use it for synthesizing CCAs with different choices of objectives (§4.2.1).

$$\begin{aligned}
&\text{beliefs inconsistent} \implies \\
&\quad (\text{State I objectives} \;\land\; \text{beliefs become consistent}) \\
&\land\, \text{beliefs consistent} \implies (\texttt{transient\_obj} \land (\text{beliefs shrink} \\
&\quad \lor\; \text{large queue drains} \;\lor\; \texttt{steady\_state\_obj}))
\end{aligned} \tag{4.1.3}$$

Users are free to choose the degree of under-specification. They may drop literals (as we do), set loose bounds for the constants (e.g., $C_U(0) < 10C$ for state III), or even synthesize CCAs that meet specific constants.

Post synthesis. The synthesis invariant is not a sufficient proof. Hence, after synthesis, we build proof lemmas for the solution CCAs. To determine the constants in the lemmas, we use binary search to identify the region of values for which the lemmas hold. We describe this process in § B.4. If there is no value of constants for which a particular lemma holds, we tweak the CCA, invariant, and/or the lemma(s). For example, when trying to build proof lemmas for a particular synthesized CCA (`cc_probe_slow` in §4.2.1), we found that the queue needs to be drained before beliefs can converge, i.e., "queue converged" in state IV (Fig. 4.3) needs to happen before "beliefs converged" in state III. So we reorder and redefine the states in the transition system and lemmas to reflect this and build a proof of performance for `cc_probe_slow`.

4.2 Results

We present four types of results. (§4.2.1) CCAs synthesized by CCmatic for various environments and objectives combinations. (§4.2.2) Fundamental tradeoffs inspired by negative outputs of CCmatic. (§4.2.3) Proofs that the synthesized CCAs ensure their performance objectives. (§4.2.4) Empirical evaluation of the synthesized CCAs to validate our mathematical modeling and proofs of performance.

	Environment		Objectives	Template		Trace	# Solutions	Time	# Itr
	Network model (§2.3)	β	Constrain loss (`transient_obj`	# Expr ("if")	Search size (# CCAs)	Length (# R)		(secs)	(CEGIS)
G1	CCAC	Infinite	N/A	2	3×10^4	4	0	16	40
	CCAC	Infinite	N/A	2	3×10^4	5	4 CCA1	48	58
	CCAC	Infinite	N/A	2	3×10^4	11	6 CCA1 CCA3	7328	64
G2	CCAC	Small fixed	No	2	3×10^4	7	6 CCA2 CCA3	4942	49
G3	CCAC	Arbitrary	No	2	3×10^4	11	2 CCA3	7699	52
G4	CCAC	Small fixed	Yes	3	10^8	11	0	5067	79
	CCAC	Arbitrary	Yes	3	10^8	11	0	6851	76
G5	Ideal	Infinite	N/A	2	3×10^4	3	48	26	86
	Ideal	Small fixed	Yes	2	3×10^4	3	194	51	229
	Ideal	Arbitrary	Yes	2	3×10^4	3	23	21	67
G6	CBR-delay	Arbitrary	Yes	2	2×10^6	7	20 CCA4	19331	513
G7	CCAC	Large	No	2	3×10^4	11	6 CCA1 CCA3	9812	79
	CCAC	Large	Yes	2	3×10^4	11	4 CCA1	6884	45
	CCAC	Arbitrary	Yes only if $\beta \geq 3C(R+D)$	4	3×10^{11}	9	6 CCA5	46582	691

Table 4.1: Summary of queries. The "# solutions" column also lists CCAs that are representative of the solutions produced. Some entries in "constrain loss" column are not applicable (N/A) as (congestive) loss is not possible when buffer is infinite.

4.2.1 Synthesis queries

A query describes the search inputs: (1) search space, (2) network model, (3) performance properties. Using CCmatic is an iterative process. One may realize that the performance properties are infeasible, the network model is too adversarial, or the approximations in the template are limiting. As we ran queries, our understanding improved, and we built new queries that better reflected our requirements (Table 4.1).

Objectives. We require CCAs to ensure a lower bound on the utilization, and upper bounds on the amount/frequency of losses and bytes in flight (to bound packet latencies). We add all these objectives to `steady_state_obj` in Eq. 4.1.3. Our primary focus is on exploring asymptotic bounds, e.g., are losses $O(C)$, $O(\log(C))$, etc. So we specify asymptotic bounds with loose constants. For example, we query CCAs that ensure utilization $\geq 50\%$, and inflight $\leq 5 \cdot C \cdot (R + D)$. The loose constants allow under-specifying the synthesis invariant (§4.1.2). Later, when building proofs, we identify the best constants the synthesized CCAs can achieve (§ B.4). Note, the inflight bound has to be at least $CR + CD$. Because, to provide a utilization lower bound, a CCA needs to fill the wire (CR bytes) and build D seconds or CD bytes of queueing (Theorem 2 of [20]).

We classify loss amount into *small* and *large*. Small means that loss is at most a few packets independent of C, or $O(1)$. Otherwise, loss is large, or $\omega(1)$, it increases with C. The verifier (SMT solver) explores traces numerically and there is no direct way to encode $\omega(1)$. As a workaround, we classify more than `3MSS` loss as large. For frequency, we query CCAs that cause at most one small loss every other R, and never incur large losses. By default, we put these constraints in `steady_state_obj`, later we also put them in `transient_obj`, e.g., G4.

Environments. We sought CCAs that can handle paths with arbitrary buffer sizes and adversarial delay jitter as modeled by CCAC and CBR-delay (§3.1, §2.3). As stepping stones, we designed CCAs for restricted buffer sizes, viz. (1) infinite ($\beta = \infty$), (2) small fixed ($\beta = \frac{1}{2}CD$), and (3) large ($\beta \geq 3 \cdot C \cdot (R + D)$). These values formed interesting thresholds during our experimentation. We represent arbitrary buffer as $\beta \geq 0$, i.e., the verifier is free to choose the buffer size.

Templates and solutions. We synthesize *all* CCAs that satisfy a query. When CEGIS finds a solution, we prune it from the search space and let the loop continue until it cannot find more solutions. For each solution, we also prune CCAs that have same coefficients for the belief bounds but different coefficients for the constants (i.e., `MSS`$/R$, R) to avoid enumerating similar CCAs.

When CCmatic does not produce a solution, we increase (1) the program size (number of "`if`" expressions), and (2) the length of the trace that the verifier considers (to give CCAs more time to show the progress required by our invariant). E.g., in G1, we get more solutions as we increase the trace length. For other queries we only show queries with the largest templates and longest traces. We also add new or tighter beliefs to the templates when we explore weaker models, e.g., G6.

Synthesized CCAs. Listing 4.2 shows the synthesized CCAs. We group similar queries together. G1, G2, and G3 explore CCAC with infinite, small and arbitrary buffer respectively. CCmatic synthesized CCA1 (`cc_qdel`), CCA2 (`cc_probe_fast`), and CCA3 (`cc_probe_drain`). CCmatic automatically figured out non-trivial insights about the network. E.g., `cc_qdel` simultaneously guarantees bounds on utilization and inflight. It sends above C_L until $qdel_L > 0$. $qdel_L > 0$ (Eq. 4.1.1) can only happen if queue is non-zero which can only happen if the link is utilized. Likewise, draining whenever $qdel_L > 0$ ensures inflight is bounded.

`cc_probe_fast` and `cc_probe_drain` "probe" when C_L and C_U are far. They send above C_L resulting in either increasing C_L (due to increase in ACK rate), or decreasing C_U (due to increase in $qdel$ or losses). `cc_probe_drain` additionally drains queues by sending below C_L. `cc_probe_fast` does not drain as it was synthesized for a small buffer which trivially bound inflight.

Listing 4.2: Synthesized CCAs (α_R = MSS$/R$)

```
(G1) (CCA1) cc_qdel
if ($qdel_L > 0$):
    rate $= 1/2C_L$
else:
    rate $= 2C_L$

(G2) (CCA2) cc_probe_fast
if ($C_U < 2C_L$):
    rate $= C_L$
else:
    rate $= 2C_L$

(G3) (CCA3) cc_probe_drain
if ($C_U > 2C_L - 2\alpha_R$):
    rate $= 2C_L + \alpha_R$
else:
    rate $= C_L - \alpha_R$

(G6) (CCA4) cc_probe_slow
if ($q_U >$ MSS):
    rate $= C_{L,\lambda} - q_U/R$
else:
    rate $= 2C_{L,\lambda} + \alpha_R$

(G7) (CCA5) cc_probe_qdel
if ($C_U < 3/2C_L$):
    if ($qdel_L > R$):
        rate $= \alpha_R$
    else:
        rate $= C_L$
else:
    if ($C_U > 2C_L$):
        rate $= 2C_L + \alpha_R$
    else:
        rate $= C_L$
```

We empirically evaluated `cc_probe_drain` and found that it incurs periodic large loss (experimental setup described in §4.2.4). This was surprising as we specifically queried for a CCA that avoids large losses in steady state. We realized this happened because of under-specifying the synthesis invariant. Due to the disjunction, "beliefs shrink $\vee \cdots \vee$ `steady_state_obj`", the CCA is allowed to cause large losses if this allows shrinking beliefs. As a result, on the periodic belief timeouts (§4.1.1.2), `cc_probe_drain` caused losses when re-probing to re-estimate beliefs (similar to BBR's 8 cycle probes).

(G4) We updated our query to ensure that when beliefs are consistent, CCA's probes (increasing sending rate) should not incur large losses.[2] I.e., we "AND" the synthesis invariant with the formula: "beliefs consistent $\implies$ (sending rate increases $\implies$ no large loss)". This is equivalent to updating `transient_obj` in Eq. 4.1.3. CCmatic did not produce any solution after this modification.

(G5) To dig deeper, we investigated weaker network models (§2.3). We set $D = 0$ to emulate an ideal link. CCmatic synthesized a CCA that sends at rate "C_L + MSS$/R$", allowing it to probe for bandwidth while risking at most constant losses. However, this does not work with CCAC. Since CCAC can delay packets, this probe may not lead to an immediate increase in ACK rate. For CCAC, a CCA needs to build D seconds of queueing to disambiguate effects of utilization (congestion) from non-congestive delays (see §4.2.2). Sending at C_L + MSS$/R$ takes $\Omega(C_L)$ time to build a queue of D seconds, and the same CCA cannot show progress (shrink beliefs) in a short fixed-length trace.

[2]Large losses may still happen when the link rate decreases. No CCA can avoid this due to feedback delay.

G6 We ran synthesis for CBR-delay using the same beliefs as CCAC. CCmatic could not synthesize any CCA that could avoid large loss. This led us to discover and prove a fundamental tradeoff between loss and convergence time (§4.2.2). The proof led us to a tighter beliefs for CBR delay. Using these, CCmatic synthesized CCA4 (`cc_probe_slow`). `cc_probe_slow` meets the loss-convergence tradeoff implying that it is tight. It risks $O(1)$ packet loss and takes $O(C)$ time to converge. For the synthesis, we added belief bounds on link rate ($C_{L,\lambda}$), buffer size, and bytes in queue (q_U). We explain these bounds and working of `cc_probe_slow` in §4.2.2.1.

G7 There is no loss-convergence tradeoff when buffers are large. We want a CCA that converges fast without incurring large loss. Additionally, on shallow buffers, we want large loss to occur only when needed i.e., for probing (shrinking beliefs). CCAs already synthesized do not fit this bill. CCA1 (`cc_qdel`) avoids large losses when the buffer is large, but causes large losses on short buffers even when it is not shrinking beliefs. CCA3 (`cc_probe_drain`) avoids large loss when it is not shrinking beliefs, but causes large losses when probing even on large buffers. CCmatic synthesized CCA5 (`cc_probe_qdel`). It gets the best of `cc_qdel` and `cc_probe_drain`.

4.2.2 Loss vs. convergence tradeoff

In G4 and G6, CCmatic had failed to produce CCAs that risk at most constant loss on CBR-delay and CCAC. All human designed CCAs that we could think of also failed. On investigating the counterexample traces for the human and machine designed CCAs, we suspected that it is impossible to avoid large loss events. On trying to prove this, we discovered a tradeoff between amount of loss and *convergence time*, i.e., time it takes for a CCA to ramp up its sending rate to the link rate. **This tradeoff applies whenever the link rate increases, and the CCA needs to ramp up (including slow start).**

Theorem 4.2.1. *For an end-to-end deterministic CCA running on a CBR-delay network with parameters $\langle C, R, D, 0 < \beta \leq CD \rangle$, to avoid getting arbitrarily low utilization, the CCA must either (1) cause $\omega(1)$ packet loss, i.e., losses that increases with C, or (2) take $\Omega(C(R + \beta_s))$ time to converge to the link rate. Where, $\beta_s = \beta/C$, i.e., buffer size in seconds.*

In general, for a CCA to ramp from C_0 to C while risking $O(f(C))$ loss, the convergence time is $\Omega(F^{-1}(C)(R + \beta_s))$, where F^{-1} is the inverse of F, and function F is defined as:

$$F(0) = C_0 \quad \text{and,} \quad F(k) = F(k-1) + f(F(k-1))/\beta_s$$

If C is not in the domain of F^{-1}, we evaluate F^{-1} at the smallest value greater than C in the domain of F^{-1}. The function $F(k)$ tracks the maximum rate a CCA can ramp up to in k `RTT`s under the loss allowance f. Correspondingly, $F^{-1}(C)$ gives the minimum number of `RTT`s needed to ramp up to rate C under loss allowance f.

Fig. 4.4 shows the convergence time F^{-1} for different choices of loss allowances f. For example, if the CCA is willing to risk O(C) losses, i.e., $f(C) = C$, then the convergence time is $\Omega(\log(C/C_0))$. If $f(C) =$ `const` (independent of C), then the convergence time is $\Omega(C - C_0)$. For other functions f, the convergence time may not have a nice closed form expression.

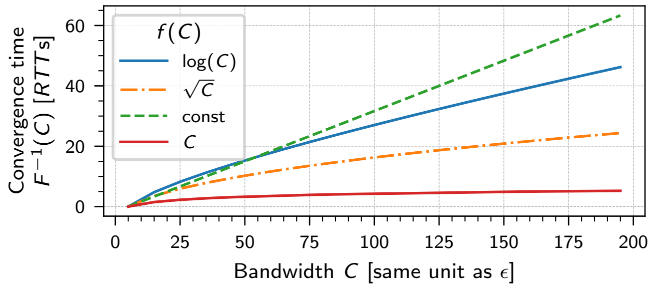


Figure 4.4: Each curve shows $F^{-1}(C)$ corresponding to $f(C)$ in the legend. I.e., under loss allowance f, the time F^{-1}, to converge from some small (positive) rate ϵ to C. Time from C_0 to C is given by "time from ϵ to C" – "time from ϵ to C_0" or $F^{-1}(C) - F^{-1}(C_0)$.

While this tradeoff may seem intuitive, it only holds when there is complicated jitter. On ideal links, a CCA can ramp up in $O(1)$ time while risking $O(1)$ packet losses, using packet trains [74]. The inter-arrival times for a packet train is the inverse of the bottleneck rate (i.e., `MSS`$/C$) and reveals C. Such techniques may even work for links with iid (independent and identically distributed) jitter. However, in practice, links do exhibit complicated jitter patterns and break such bandwidth estimates [43].

Below, we provide intuition and outline the proof for Theorem 4.2.1. We also prove a similar loss-convergence tradeoff for ideal links when packet trains are not allowed and $\beta = 0$. This proof is similar to that of Theorem 4.2.1, we omit it for brevity.

Intuition & counterexample trace. The tradeoff is valid only on shallow buffers (i.e., $\beta \leq CD$). This renders `RTT` measurements meaningless, forcing the CCAs to rely on losses . On larger buffers, faster convergence is attainable by relating how `RTT` varies with varying sending rate. When $\beta \leq CD$, the queueing delays are at most D seconds. The delay box can choose jitter such that `RTT`s are at most $R + D$. Such `RTT`s could be due to queueing delays (resulting from varying sending rates), or due to jitter in the delay box, and there is no way for the CCA to distinguish between the two.

We investigated a CCA that additively increases its sending rate every `RTT` until it sees a loss, to see why it cannot avoid large loss. The verifier gave us a trace where the CCA keeps blindly increasing its sending rate even when it is already above the link rate. Eventually the queue builds up and $O($`BDP`$)$ loss happens. The CCA needs to resort to blind increases because it does not get any feedback until it causes loss (as queueing delay measurements are meaningless). If these blind increases are aggressive, this results in larger losses with faster convergence and vice versa. **`cc_probe_slow`** drains queues along with additive increments to meet the bound in Theorem 4.2.1 (§4.2.2.1).

Proof outline. (Full proof in § B.2) We first show that due to $\beta \leq CD$, a CCA must cause loss to avoid arbitrarily low utilization (**Step 1.**). Then we compute a *tight* lower bound belief ($C_{L,\lambda}$) for C, for CBR-delay (belief computations in §4.1.1.1 were for CCAC) (**Step 2.**). The CCA could be running on any link with $C \geq C_{L,\lambda}$, and it needs to ensure it does not cause loss on any of these links. This allows us to compute the amount of loss a CCA risks any time it probes for bandwidth

(**Step 3.**). If we restrict this risk of loss to a constant independent of C, it gives us a constraint on how quickly the CCA can ramp up, giving us a lower bound on the convergence time (**Step 4.**).

4.2.2.1 `cc_probe_slow` shows Theorem 4.2.1 is tight

We describe beliefs for CBR-delay, how `cc_probe_slow` works, and why `cc_probe_slow` does not work for CCAC.

Bandwidth and buffer beliefs. For CBR-delay, to obtain an upper bound on C, a CCA needs to cause loss or build more than D seconds of queueing (from **Step 1.** of Theorem 4.2.1). We compute the set of paths $\langle C, \beta\rangle$ that can produce CCA's observations until time t^*, such that the CCA has not observed $qdel > D$ or loss until t^*. This means that until $t^* - \texttt{RTT}(t^*)$, the enqueued bytes never exceeded the dequeued bytes by more than D seconds (CD bytes) or buffer size. I.e., $\forall t_1, t_2$, such that, $0 \le t_1 \le t_2 \le t^* - \texttt{RTT}(t^*)$:

$$\int_{t_1}^{t_2} \lambda(s)ds - C\cdot(t_2 - t_1) \le CD \tag{4.2.1a}$$

$$\text{and,} \int_{t_1}^{t_2} \lambda(s)ds - C\cdot(t_2 - t_1) \le \beta \tag{4.2.1b}$$

From (4.2.1a), we define $C_{L,\lambda}$ as:

$$C_{L,\lambda}(t^*) = \max_{0\le t_1\le t_2\le t^*-\texttt{RTT}(t^*)} \frac{\int_{t_1}^{t_2}\lambda(s)ds}{t_2 - t_1 + D} \tag{4.2.2}$$

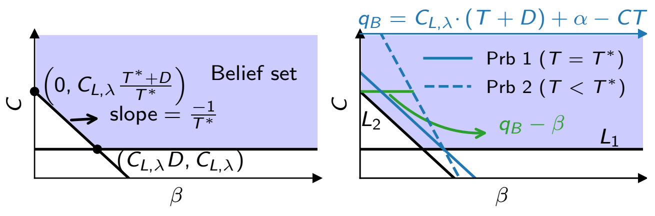
We evaluate (4.2.1b) over the interval $[t_1^*, t_2^*]$ (length $T^* = t_2^* - t_1^*$)[3] corresponding to the tightest bound on $C_{L,\lambda}$:

$$\begin{aligned} & C_{L,\lambda}(t^*)\cdot(t_2^* - t_1^* + D) - C\cdot(t_2^* - t_1^*) \le \beta \\ \text{or,}\quad & C\cdot T^* + \beta \ge C_{L,\lambda}(t^*)\cdot(T^* + D) \end{aligned} \tag{4.2.3}$$

From (4.2.2) and (4.2.3), we get the belief set illustrated in Fig. 4.5: $\{\langle C, \beta\rangle \mid C \ge C_{L,\lambda}(t^*) \wedge C\cdot T^* + \beta \ge C_{L,\lambda}(t^*)\cdot(T^* + D)\}$. $C_{L,\lambda}$ **tells two things, (1) the link rate is at least $C_{L,\lambda}$, and (2) if the link rate is $C_{L,\lambda}$, then the buffer is at least $C_{L,\lambda}D$ bytes.**

In the SMT encoding, we only evaluate Eq. 4.2.2 over intervals of length $T^* = R$. This is because the measurement interval T^* influences CCA's probing behavior (see below and § B.3). The initial conditions (including initial beliefs) are chosen by the verifier (§2.3). As a result, verifier's initial choice of T^* influences the CCAs probing which should be in full control of the CCA. Fixing $T^* = R$ solves this issue.

[3]Other intervals may produce a tighter Eq. 4.2.3 but we over-approximate.



$\left(0, C_{L,\lambda}\frac{T^*+D}{T^*}\right)$
Belief set
slope $= \frac{-1}{T^*}$
$(C_{L,\lambda}D, C_{L,\lambda})$
C
β
$q_B = C_{L,\lambda}\cdot(T+D)+\alpha-CT$
Prb 1 ($T = T^*$)
Prb 2 ($T < T^*$)
$q_B - \beta$
L_2
L_1
C
β

Figure 4.5: Belief set for CBR-delay (left). Queue buildup as a function of C (right). L_1, L_2 lines are the belief constraints.

Queue beliefs. For ease of discussion we define "probe interval" as an interval that leads to increase in $C_{L,\lambda}$ using Eq. 4.2.2. We show in § B.3 that probe intervals needs to be start with a drained queue, otherwise, the CCA risks losing all packets in the queue at the beginning of the probe interval. For the CCA to gauge the queue state, we add q_U as an upper bound on the bottleneck queue bytes. We compute it as: (bytes already in queue) + (enqueued bytes) − (dequeued bytes on smallest C). I.e., $q_U(t + \delta t) = \max(0, q_U(t) + (\lambda(t) - C_{L,\lambda}(t))\delta t)$.

In summary, the template now includes inputs as $C_{L,\lambda}$, $T^* = R$, and q_U. For brevity, we omit details on timeouts for these beliefs.

How `cc_probe_slow` works. We discuss a generalized version called `cc_probe_slow_k`. It drains queues until $q_U \leq \alpha$ (according to Listing 4.2). Then probes by sending $C_{L,\lambda}\cdot(T^* + D) + \alpha$ bytes paced over T^* time. α is a constant and $T^* = \Bbbk R$ is the interval length over which the CCA measures $C_{L,\lambda}$. This probe ensures (1) progress (beliefs shrink), i.e., causes losses, $qdel > D$, or increases $C_{L,\lambda}$; while ensuring (2) loss $\leq 2\alpha$.

Progress. The probe builds queues. On the network with rate C, the queue build up is (enqueued bytes) − (dequeued bytes) $= C_{L,\lambda}\cdot(T^* + D) + \alpha - C\cdot T^*$ ("Prb 1" illustrates this in Fig. 4.5). On the network with rate $C_{L,\lambda}$, the queue build up is $C_{L,\lambda}D+\alpha$ (i.e., more than D seconds). If the CCA does not observe $qdel > D$, then it knows that C is higher than $C_{L,\lambda}$, and the line L_1 (Fig. 4.5) moves up. I.e., $C_{L,\lambda}$ becomes $\frac{C_{L,\lambda}\cdot(T^*+D)+\alpha}{T^*+D} = C_{L,\lambda} + \frac{\alpha}{T^*+D} > C_{L,\lambda}$. Likewise, if the CCA does not observe loss, then L_2 shifts right. If the CCA does observe $qdel > D$ or loss, then it knows that it sent above C and obtains an (implicit) upper bound on C, ensuring a lower bound on utilization.

Loss. Loss due to the probe is ΔL = (bytes already in queue) + (queue buildup) − (buffer) = $q + q_B - \beta$. In Fig. 4.5, the horizontal distance between a point (β, C) in the $\boxed{\text{belief set}}$ and the line Prb 1 shows $q_B - \beta$. This distance, i.e., "$C_{L,\lambda}\cdot(T^* + D) + \alpha - CT^* - \beta$" is at most α (from Eq. 4.2.3). Bytes already in the queue is also at most α as probe only happens when $(q \leq)\ q_U \leq \alpha$. Hence, $\Delta L \leq 2\alpha$, i.e., constant independent of C.

Also notice, if we probe over a shorter interval $T < T^*$, then there are networks (β, C) for which $q_B - \beta$ is larger (Prb 2 in Fig. 4.5) and loss on those networks is $O(C)$. To avoid large loss, the probing interval T needs to be at least as long as the past measurement interval T^* (§ B.3 formally proves this).

Path	Metrics	Probe	Drain$_1$	Drain$_2$
		Duration		
		T^*	$D = R$	$D = R$
	Sent by CCA	$C_{L,\lambda} \cdot (T^* + D) + \alpha$	α	$C_{L,\lambda}R - 2\alpha$
	q_U	$C_{L,\lambda}D + \alpha$	2α	0
P_1	Serviced$_1$	$C_{L,\lambda}T^*$	$C_{L,\lambda}D$	$C_{L,\lambda}R$
	Queue$_1$	$C_{L,\lambda}D$	2α	0
	Loss$_1$	α	0	0
	Util$_1$	100%	100%	100%
P_2	Serviced$_2$	$C_{L,\lambda} \cdot (T^* + D)$	α	$C_{L,\lambda}R - 2\alpha$
	Queue$_1$	0	0	0
	Loss$_2$	α	0	0
	Util$_2$	100%	$\approx 0\%$	$\frac{T^*}{T^*+D} \cdot 100\%$

Table 4.2: Steady-state behavior of `cc_probe_slow` on two extreme paths in the belief set: $P_1 = \langle C = C_{L,\lambda}, \beta = C_{L,\lambda}D \rangle$ and $P_1 = \langle C = C_{L,\lambda}\frac{T^*+D}{T^*}, \beta = 0 \rangle$. "Serviced" shows bytes serviced by the CBR box, and q_U is computed at the end of the probe/drain duration. Note, for drain$_1$, the CCA sends packets due to line 7 of the template (Listing 4.1), even though `rate` $= C_{L,\lambda} - q_U/R < 0$.

`cc_probe_slow` steady-state behavior. In steady state, `cc_probe_slow` follows a "Probe, Drain$_1$, Drain$_2$" cycle. Table 4.2 shows the result of this cycle on two paths in the belief set. These lie on the corners of belief set on the line L_2 (Fig. 4.5). Notice the two paths are very different, but CCA's observations are exactly the same. The loss is the same, and the delay box can ensure `RTT`s are the same (as $qdel \leq D$). In fact all paths on the line joining P_1 and P_2 can produce CCA's observations, and form the steady-state belief set. Utilization is lowest on P_2 and highest on P_1. The bandwidths of P_1 and P_2 differ by a factor of $\frac{T^*}{T^*+D}$. To bound inflight on P_1, CCA's average sending rate cannot be more than $C_{L,\lambda}$, as a result, the average utilization on P_2 cannot be more than $\frac{T^*}{T^*+D}$. The only way to increase utilization is to reduce uncertainty (size of belief set) by increasing T^*. This also increases convergence time (e.g. if the link rate increases when the CCA is in steady state).

If we replace α with $f(C_{L,\lambda})$, where $f(.)$ is the loss allowance (§4.2.2). Then the resulting family of CCAs (parameterized by $f(.)$ and T^*) allows us to tune the tradeoff between loss vs. convergence time vs. utilization. We can even adapt these over time (§ B.6).

Discussion. The probe works as CBR-delay ensures that if a probe did not cause loss in the past then repeating the probe will not cause a loss. This is not true for CCAC due to its *non-deterministic* token bucket filter (TBF). CCAC can arbitrarily decide how many tokens to keep in the bucket. A past probe may not have lost packets as the token bucket was full. However, on repeating the probe, CCAC can choose to keep the token bucket empty, and drop $O($`bucket`$)$ packets. Due to this, we conjecture that either large losses cannot be avoided for CCAC or will require asymptotically longer convergence time (e.g., quadratic instead of linear in `BDP`).

4.2.3 Proofs of performance

Ensuring the under-specified synthesis invariant is not a sufficient proof (§4.1.2). We summarize the lemmas that serve as the proof of performance for CCA1 (`cc_qdel`) and CCA4 (`cc_probe_slow`). These represent CCAs designed for deep and shallow buffers respectively. We give the full list and encoding of lemmas in § B.4. Note, we describe theoretical worst-case bounds. Empirical performance is better (§4.2.4).

`cc_qdel`. On a CCAC link with parameters $\langle C, R, D = R, \beta \geq 3C\cdot (R + D)\rangle$, `cc_qdel` ensures that beliefs become consistent exponentially fast, they converge exponentially fast, and it drains queue at a rate proportional to C.[4] After beliefs are consistent, converged, and the queue is drained, the CCA is in steady state and remains in steady state. Note, this is despite the periodic beliefs timeouts (§4.1.1.2), i.e., the beliefs remain close to each other even after timing out. In steady state (state IV in §4.1.2), `cc_qdel` gets at least 89% utilization, keeps $\texttt{RTT} \leq 4.4(R + D)$ seconds, and loses at most 3 packets in any R duration. Additionally, `cc_qdel` never incurs large loss events when probing for bandwidth as long as the beliefs are consistent, i.e., in states II, III, IV.

`cc_probe_slow`. On the CBR-delay model with parameters $\langle C, R, D, \beta\rangle$, `cc_probe_slow` ensures that beliefs $(C_{L,\lambda}, q_U)$ become consistent exponentially fast, and $C_{L,\lambda}$ converges additively. I.e., when the link rate decreases, `cc_probe_slow` ramps down exponentially fast. When the link rate increases, `cc_probe_slow` ramps up additively. In steady state, `cc_probe_slow` ensures at least 30% utilization, keeps $\texttt{RTT} \leq 1.5(R + D)$. It ensures that it loses at most 2 packets in any R duration whenever beliefs are consistent and the link rate has not decreased. Note, in §4.2.2.1, we showed steady-state utilization $\geq \frac{T^*}{T^*+D}100\% = 50\%$ (for $T^* = R = D$). The proved worst-case utilization is lower because $C_{L,\lambda}$ may reduce on timeouts.

4.2.4 Empirical evaluation

Our goal with empirical evaluation is to validate our mathematical modeling and proofs of performance. Further evaluation is warranted before deployment.

Implementation. We implement CCA1 (`cc_qdel`) and CCA4 (`cc_probe_slow_k`, see §4.2.2.1 for "`_k`") over UDP using [17]. For `cc_probe_slow_k`, we run `cc_qdel` until a large loss event (resembling TCP slow start), and set $\alpha = 5\texttt{MSS}$ instead of $1\texttt{MSS}$ to account for false-negatives in loss detection. As a result, probes may lose $2\alpha = 10\texttt{MSS}$ bytes.

We initially implemented the CCAs in the Linux kernel, but found bugs in kernel's pacing implementation and the `cong_control` API to be insufficient (§ B.5). We compare against Cubic, BBRv1 (Linux kernel v5.4.0), BBRv2 [33], BBRv3 [34], and Copa [17, 18].

Scenarios and metrics. We use iperf to generate traffic and mahimahi [94] to emulate scenarios with jitter and shallow buffers (§3.1) with the aim of validating our performance proofs.

[4]A CCA cannot drain the queue faster than this. Even if it stops sending packets, the queue will only drain by $C\cdot t$ bytes in time t.

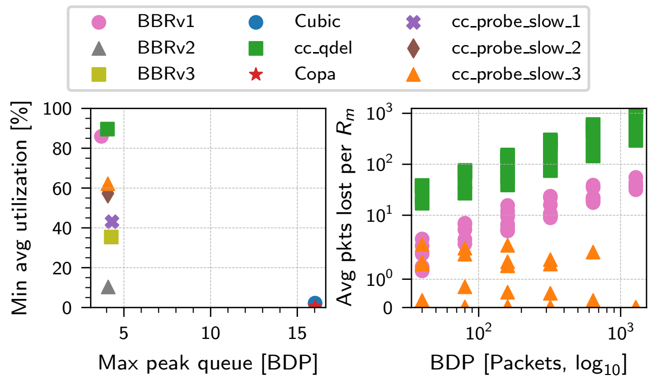


Figure 4.6: Summary of utilization, queue, and loss. Right subplot shows runs with $\beta \leq$ BDP. Additionally, for visibility, we only show three CCAs, and use symlog scale on y-axis.

We measure utilization, delay, loss, and convergence time metrics under a variety of parameter $\langle C, R, D, \beta \rangle$ choices. Each tuple constitutes a different "run". We emulate jitter by injecting up to $D = R$ seconds of *uniformly random* delay, while ensuring FIFO service and constant average link rate (§ B.6).

Results. Fig. 4.6 summarizes utilization, queue (proxy for delay), and losses across runs (e.g., each run gives us an average utilization, and we compute the minimum across runs). § B.6 shows the metrics for each run and also studies convergence time and fairness. The synthesized CCAs are within their proven performance bounds and achieve tradeoffs between loss, convergence, utilization, and delay that prior CCAs cannot achieve.

`cc_probe_slow` meets the proven lower bound on utilization, upper bound on `RTT`s, and incurs at most constant loss (independent of BDP) across buffer sizes, bandwidths and propagation delays. On the same networks, Copa starves as it is not robust to jitter [19]. BBRv1 is able to ensure utilization despite random jitter, but incurs excessive losses (that increase with the `BDP`) on shallow buffers. Cubic gets low utilization when buffers are short ($\beta \ll$ `BDP`) and also bloats queues when buffers are large ($\beta \gg$ `BDP`). `cc_qdel`'s performance is similar to BBRv1, but with provable guarantees on utilization even with worst-case jitter. Unlike BBRv1, the v2 and v3 variants do not incur high losses on average but incur $O($`BDP`$)$ losses to converge exponentially fast when the link rate increases (§ B.6). They also get lower utilization than the v1 variant. `cc_probe_slow_k` gets higher utilization at the cost of higher convergence time (§ B.6) with increasing `k`.

4.3 Related work

Related work not covered in Chapter 2 can be classified into:

Automatic CCA design. Past works have explored online learning [41], reinforcement learning [67, 83, 110], model predictive control (MPC) [60, 64], and (partially observable) markov decision process (POMDP) formulations [116]. CCAs produced by these works are not human-interpretable or are not explicitly designed for adversarial network behaviors. CCmatic CCAs are modular, human-interpretable, and provably robust under adversarial network behaviors.

Reasoning about CCAs. Past works use different network models: (1) deterministic (e.g. fluid model) [37, 122], (2) stochastic [56], and (3) non-deterministic [20]. Some also limit their scope to a small class of CCAs [21, 37, 122]. Deterministic models are easier to reason about but may not accurately reflect real world behaviors. With stochastic models, it is hard to deduce probability distributions that characterize real networks. Non-deterministic models do not require a distribution but may be too adversarial. Beliefs can be computed regardless of the modeling choice, and facilitate reasoning about complicated network models across all possible CCAs. Beliefs and CCmatic add to the emerging toolkit for performance reasoning.

4.4 Chapter summary

Using beliefs from Chapter 3, we built CCmatic to automatically synthesize CCAs for different environment/objective combinations, alleviating humans from figuring out complex details like when/how long to probe/drain. CCmatic also gives insights when objectives are infeasible. Due to formal methods and program synthesis, CCmatic CCAs are human-interpretable and provably performant. We used CCmatic to build novel CCAs and also showed how we can use the properties of beliefs to discover and prove new tradeoffs in congestion control.

In this part, we focused on single-flow settings, in the next part we build contracts to enable reasoning about multi-flow scenarios.

4.5 Syntra: Joint video streaming and congestion control using deductive synthesis

In work related to this thesis, we collaborated with researchers at UT Austin to develop techniques that extend and improve upon CCmatic. In CCmatic, we relied on *inductive* synthesis: we "searched" for a controller that met the specification by filling holes in a template. This template-based approach restricts the search to a subset of belief-based CCAs and sacrifices completeness guarantees. In follow-up work, we found that we can preserve both soundness and completeness by using *deductive* synthesis. Instead of searching over a template, deductive synthesis "derives" the algorithm by algebraically manipulating the input specification using known logical rules and identities.

Directly solving the deductive formulation is computationally intractable, but it yields two key insights that enabled us to build Syntra [97], a tool for automatically designing joint controllers for video streaming and congestion control. Unlike CCmatic, which reasons only about the congestion-control sending rate, Syntra jointly reasons across multiple decision axes including video quality, frame rate, sending rate, and forward error correction.

Syntra performs this reasoning automatically and produces controllers that outperform strong baselines such as WebRTC-GCC [35], WebRTC-Vegas [27], and Salsify [50]. It improves

P95 one-way delay by over 6×, increases median video quality (SSIM) by more than 2 dB, and maintains higher frame rates even under challenging conditions, including jitter.

To achieve this, Syntra addresses key scalability and expressivity limitations of CCmatic. While CCmatic can take weeks to synthesize a controller that is only a few lines of code, Syntra can design 100-line controllers within days. This is critical for expressing optimal policies that make joint decisions across multiple control axes.

Below, we give a brief overview of the deductive formulation, the key insights it revealed, and Syntra's approach to synthesizing joint controllers. We refer the reader to [97] for a complete description of Syntra.

Deductive control synthesis. We model the design of congestion control algorithms as a two-player game between the controller and the network. For each game state (belief set), we compute an action for the controller such that, for all actions that the network may take, the controller "wins" (it either makes progress toward shrinking the belief set or satisfies the desired performance objectives). This design problem can be expressed as a quantified logical formula (Eq. 4.5.1), which can be solved using *quantifier elimination* (QE) [109].

$$\exists\, \texttt{Controller action}\ \forall\, \texttt{Network action}.\ \texttt{Feasible network} \implies \texttt{Controller wins} \tag{4.5.1}$$

Here, `Feasible network` is a boolean valued SMT formula that encodes that the `Network action` is feasible according to the network model under the current belief set. The solution to this formula eliminates the ∀ quantifier to give us a formula describing the optimal controller action as a function of the belief set that causes the controller to win. This formula, mapping the belief set to the optimal control action, is the desired controller. Such formulations have been explored for other problems in the control theory literature [23].

Insight #1. Belief projections are exponentially large and can be computed accurately. While in Chapter 3, we showed that we can compute the complete belief set by inverting the network model, in practice, we often also need to compute *projections* of the belief set along different dimensions. For example, to determine whether the controller is reducing uncertainty in link capacity, we compute the set of capacities in the belief set. Computing such projections is hard and to simplify this, we over-approximated the projections in CCmatic (Fig. 4.2).

The deductive viewpoint gave us an automatic and accurate way to compute the projections. Specifically, the complete belief set is given by:

$$\begin{aligned}\texttt{Belief set} = \{\langle C, R, D, \beta, q(t), \theta(t)\rangle \mid \exists\, q(0..t-1), \theta(0..t-1).\\ \texttt{Network}(C, R, D, \beta, q(0..t), \theta(0..t), A(0..t), S(0..t), L(0..t))\}\end{aligned} \tag{4.5.2}$$

I.e., belief set is the set of latents "$\langle C, R, D, \beta, q(t), \theta(t)\rangle$" for which there exists a history of network states "$q(0..t-1), \theta(0..t-1)$" that can satisfy the network model "`Network`" under the observations of the controller "$A(0..t), S(0..t), L(0..t)$".

Projections of this set along the capacity dimension is given by:

$$\texttt{Belief set}_C = \{C \mid \exists\, R, \beta, q(0..t), \theta(0..t). \\ \texttt{Network}(C, R, D, \beta, q(0..t), \theta(0..t), A(0..t), S(0..t), L(0..t))\} \tag{4.5.3}$$

I.e., the set of capacities for which there exists other latents and history of network states that can satisfy the network model under the controller's observations. We can get a closed form expression for "$\texttt{Belief set}_C$" using quantifier elimination. QE simplifies the input formula by eliminating the quantified variables to obtain an output formula that has the same satisfying assignment for the free variables.

As a very simple example, consider the quantified formula: "$\exists\, x \in \mathbb{R}.\ ax^2 + bx + c = 0$". Here, x is a quantified variable and a, b, c are the free variables. Applying QE to this formula yields the quantifier-free formula "$b^2 - 4ac \geq 0$". Here, both the input and output formula have the same set of satisfying assignments on the free variables a, b, c. Turning back to Eq. 4.5.3. The free variables are "C" and the observations "$A(0..t), S(0..t), L(0..t)$". The satisfying assignments to this formula are the set of capacities "C" such that the network model is satisfied under the controller's observations. QE gives us an equivalent formula that does not include the quantified variables: other latents and history of states.

Applying this procedure led to a surprising result: the (quantifier-free) formulas describing the projected belief sets are exponentially larger than the formulas describing the underlying network model. This is not an artifact of our implementation; belief computations are inherently large. In hindsight, this may explain why congestion control has been so hard: our community has been effectively trying to perform the giant task of computing beliefs in an ad hoc manner. We can solve this once and for all by computing beliefs systematically and automatically.

Intuitively, belief computations have to be large because unlike the network model that maps initial state to next state, the belief computations map the history of states to inferences about latest states. There are exponentially many variations that can happen in the history that each give a different inference about the network state. For example, if we consider the history of losses over three time steps, we may observe that loss did not occur in any of the three steps, loss occurred at one of the three steps, loss occurred at two of the three steps, etc. Each of these combinatorially many possibilities gives us a different constraint in the belief set formulas bloating the size of the belief computation formula.

Insight #2. Optimal controller is in LRA. Although we cannot directly solve the two-player game using quantifier elimination, the formulation provides insights into the structure of the optimal controller allowing us to approximately solve the formula and design novel controllers that beat key state-of-the-art baselines. Specifically, QE has the property that if the input quantified formula is in linear real arithmetic (LRA), then the equivalent quantifier-free formula is also in LRA [102, 109]. Since our game-theoretic formulation is entirely in LRA, this implies that the optimal controller—the solution to the QE problem—must also be in LRA.

This is a striking finding. Our community has developed algorithms such as Cubic, which rely on cube-root like functions. Our result suggests that if the network model and performance objectives can be written without complicated mathematical functions, then the optimal controller also does not require such functions.

Deductive synthesis enabled Syntra. Syntra synthesizes controllers by approximately solving the QE formula. Rather than solving the QE formula symbolically, we solve it numerically for a finite set of concrete game states (belief sets), to obtain optimal control actions for the game states. These $\langle$belief, action$\rangle$ pairs serve as training data for an imitation-learning phase, where we fit a decision tree that "imitates" the optimal decisions. Using a decision tree (rather than a neural network) preserves human-interpretability of the resulting controller. We know that a decision tree is enough because the optimal controller is in LRA. This approach, guided by our insights, allowed us to go beyond CCmatic and design more complicated controllers in a shorter amount of time.

Part II

Multiple flows

Chapter 5

Contracts: Coordinating between flows

In the previous part, we focused on single-flow congestion control, where the only source of uncertainty was the lack of visibility into the state of the network. Beliefs enabled principled reasoning about this partial observability. In this part, we turn to multi-flow congestion control. Here, in addition to partial observability, we need to deal with *decentralization*, i.e., lack of visibility into the state of the other flows.

In this chapter, we introduce contracts, our abstraction that formalizes how CCAs coordinate with each other to reach fairness under decentralization. Traditionally, our community does not reason about uncertainty explicitly and uses Chiu-Jain (Fig. 5.1) plots to reason about CCA fairness. These plots show that additive increments (AI) and multiplicative decrements (MD) converge to fairness and efficiency. We argue that this is a bad way to reason about fairness because it hides the tradeoffs imposed by decentralization and that our community should use contracts instead.

Since CCAs cannot directly observe each other's internal states, they implicitly communicate with each other by encoding information into globally observable congestion signals. Most CCAs do this by encoding their fair share into these signals. For instance, Reno uses the encoding "$\texttt{fair share} \propto 1/\sqrt{\texttt{loss rate}}$". All flows collectively agree to maintain a loss rate and sending rate (or fair share) pair that obeys this relation. All the fair CCAs that we know of honor such an encoding. For instance, Copa/Vegas/FAST use "$\texttt{rate} = 1/\texttt{delay}$" [18, 27, 115], Swift uses "$\texttt{rate} = 1/\texttt{delay}^2$" [73], and Arun et al. in [20] proposed a CCA that uses "$\texttt{rate} = e^{-\texttt{delay}}$" [20].[1] We call such encoding functions as *contracts*.

We show that these encoding functions fully determine the steady-state performance metrics of CCAs including throughput, latency, fairness, generality (e.g., range of link rates that the CCA can support), and robustness (to noise in congestion signals). We use this to reveal fundamental tradeoffs between these performance metrics. We derive the contracts for a variety of existing CCAs showing where they lie in the tradeoff space and identify the corner points in the tradeoff space. Since CCAs follow the contract in steady-state, the model of their steady-state behavior (e.g., fixed-points or limit cycles) reveals the encoding function used by the CCA. This allows us to

[1]Throughout this thesis, we use delay to mean a queuing delay estimate (e.g. $\texttt{RTT} - \min \texttt{RTT}$).

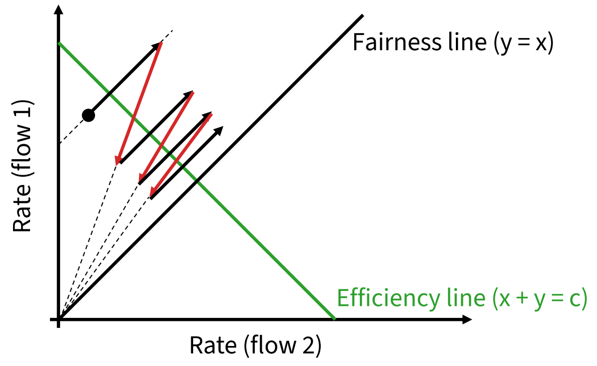


Figure 5.1: Chiu-Jain plots [37] illustrating how Reno (AIMD) converges to fairness and efficiency. The black arrows show how the system state (i.e., the sending rates of two competing flows) evolves under additive increments, while the red arrows show the evolution under multiplicative decrements.

analytically and empirically *reverse engineer* the contracts of a variety of CCAs including not only analytically designed CCAs, but also black-box implementations, such as Tao (Remy) [104, 116], and learned CCAs, such as PCC [41] (online-learning) and Astraea [84] (reinforcement-learning).

Contracts also allow us to identify a variety of design mistakes that result in poor performance. For instance, we find that CCAs whose contracts have *extreme shapes* (e.g., logarithmic, exponential), *shifts* (e.g., $\texttt{rate} = 1/(\texttt{delay} - \texttt{c})$ vs. $\texttt{rate} = 1/\texttt{delay}$), *clamps* (e.g., $\texttt{delay} = \max(\texttt{c}, 1/\texttt{rate})$), or *intercepts* can starve flows. We observe this in BBR [30], ICC [68], and Astraea [84].

To avoid these pitfalls and navigate the tradeoff space in a principled manner, we develop blueprints for CCA design and analysis. We argue that the design (or analysis) of any CCA should begin by choosing (or identifying) its contract. Doing so forces the designer to explicitly confront the inherent tradeoffs and select Pareto-optimal points rather than inadvertently committing to suboptimal choices. In contrast, prior CCA design methodologies such as Network Utility Maximization (NUM) [71, 85, 86, 106], Remy [116], or reinforcement learning (RL) [84] tend to *hide* these tradeoffs, creating the illusion that conflicting performance metrics can be optimized independently when, in reality, they are fundamentally coupled.

We begin by explaining why contracts are necessary to address decentralization, highlighting design mistakes that could have been avoided had we explicitly reasoned about contracts, and providing blueprints for principled CCA design and analysis (§ 5.1). We then formally define contracts and derive the contracts for a variety of CCAs (§ 5.2). In the next chapter (Chapter 6), we show how contracts fully determine the steady-state performance of CCAs and use this insight to derive fundamental performance tradeoffs.

5.1 Contracts through examples

5.1.1 Why do we need contracts?

Since CCAs do not know how many flows they are competing against, they need some form of agreement between flows to determine their fair share. For instance, consider the scenarios in Fig. 5.2. If the green (upper) flow in scenario B exactly emulates the cumulative effect of the three green flows in scenario A, then the red flow cannot tell the difference. Its observations (timestamps of packet transmissions and acknowledgements) are identical. Yet, its fair share is different.

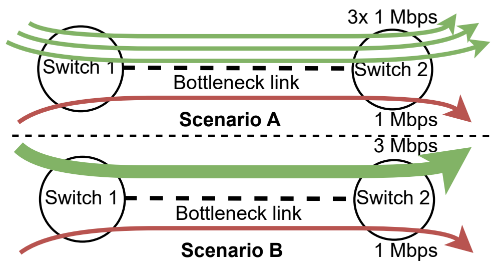


Figure 5.2: To distinguish the scenarios, the red (lower) flow must coordinate with green (upper) flows.

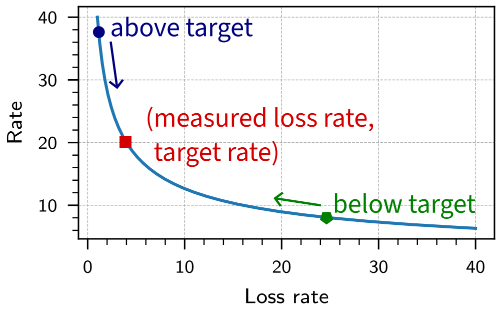


Figure 5.3: Reno uses loss rate to communicate fair shares.

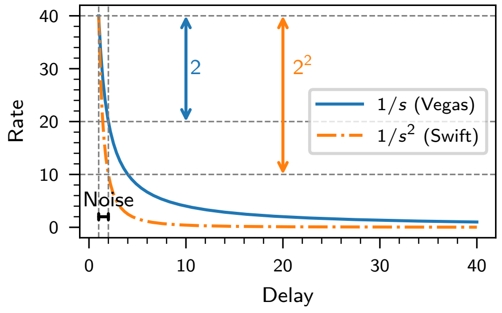


Figure 5.4: Steeper contract implies worse robustness to noise.

Contracts help flows to disambiguate between such scenarios. Since flows cannot directly communicate with each other, all flows "agree" to follow an encoding of fair share into globally observed signals. For example, Reno uses: "$\texttt{rate} \propto 1/\sqrt{\texttt{loss rate}}$" [48, 89]. While this is not explicit in the design, the emergent behavior is that all flows react to losses such that they effectively measure the average packet loss rate and calculate a "target rate" using this formula. Then they increase or decrease their actual sending rate to move towards this target (Fig. 5.3). Returning to scenario B in Fig. 5.2, both the red and green flows measure the same loss rate, and hence calculate the same target rate, but they have different sending rates. Thus, at least one of them will change their rate until they reach a steady-state where everyone's sending rates are equal.

Mathematically, the contract forces a unique equilibrium for flows. Consider a fluid model execution of a CCA on a dumbbell topology, with n flows (f_1, to f_n), flow f_i having an RTprop (round-trip propagation delay) of R_i seconds and a link capacity of C packets/second. The fluid model equations yield $n+1$ independent variables with 1 independent equation: $\sum_i \texttt{rate}_\texttt{i} = C$. Where, $\texttt{rate}_\texttt{i} = \texttt{cwnd}_\texttt{i}/\texttt{RTT}_\texttt{i}$ and $\texttt{RTT}_\texttt{i} = \texttt{delay} + R_i$. The independent variables are $\texttt{cwnd}_\texttt{i}$ and $\texttt{delay}$. The CCA through its contract (e.g., $\texttt{rate}_\texttt{i} = \texttt{cwnd}_\texttt{i}/\texttt{RTT}_\texttt{i} = 1/\texttt{delay}$) yields n additional equations to ensure a unique solution to this system.

To our knowledge, all existing CCAs use a similar method of coordination. They just use different signals and contract shapes. E.g., DCTCP, DCQCN, and MPRDMA use average ECN marking rate [12, 88, 123, 124]; Poseidon uses the maximum per-hop delay [112] collected using INT (In-band Network Telemetry); AIMD on delay uses bytes (or time) between high delay (§ 6.3). Even CCAs generated through machine-learning (e.g., Astraea [84]) implicitly learn to use contracts (§ 5.2.1).

Due to the coupling between rate and congestion, a CCA's choice of contract induces a variety of tradeoffs, e.g., halving the link capacity quadruples the steady-state loss rate for Reno (§ 6.1).

5.1.2 Contracts help identify CCA design mistakes

Contracts share mathematical foundations with NUM (§ 5.2.2) and some of our results may seem like simple extensions of existing results. Despite this, many CCAs (including recent ones) repeatedly commit avoidable mistakes resulting in poor performance. We detail some of these mistakes below with others in § 6.3. To our knowledge, outside of mistake #1, other mistakes (including § 6.3) have not been documented before. Contracts helped us discover these issues, and make it easier to systematically avoid them in future designs.

These mistakes stem from several anti-patterns, including, treating CCA design as a mere reaction to congestion (without analytically understanding their consequences) or blindly using AIMD in hopes of ensuring fairness. The most prominent anti-pattern is myopically optimizing latency without reasoning about fairness. This manifests in various ways, including, setting constant delay targets in hand-designed CCAs (§ 6.3) or in reward functions of RL-based designs (§ 5.2, § 6.3). However, latency is intertwined with other metrics and cannot be optimized unilaterally (§ 6.1, § 6.3). Note that this is different from the tradeoff between latency and throughput, which stems from variations in link capacity as opposed to the coordination mechanisms used by CCAs.

Mistake #1: Not having a contract. CCAs like TIMELY [92] do not have a contract, i.e., there is no unique mapping between steady-state rate and congestion signals to force a unique equilibrium. As shown in [124], TIMELY admits infinitely many solutions to the steady-state equations described above. Each solution corresponds to a different allocation of flow rates (provided they sum to link capacity), with a potentially arbitrarily large ratio of flow rates (arbitrary unfairness).

Mistake #2: Picking extreme contracts. Recent proposals like ICC [68], Astraea [84], and the exponential CCA from [20] optimize for a narrow subset of performance metrics, yielding contracts that are good for some metrics but extremely bad for others. All three CCAs cause extreme unfairness in multi-bottleneck topologies (§ 6.1). ICC and Astraea are also extremely susceptible to network jitter (§ 6.1).

Mistake #3: Conflating AIMD with contracts, and picking sub-optimal dynamics. Swift [73] uses AIMD to update cwnd despite having an explicit contract. AIMD is not necessary for fairness and unnecessarily increases convergence time. With explicit contracts, MIMD updates can reach fairness exponentially fast while maintaining a stable control loop (§ 6.4). For instance, in Fig. 5.3, all flows measure the same loss rate and compute the same target rate. Flows can update their current rate to move towards the target using any increment choice (additive or multiplicative). Fairness is ensured despite MIMD updates because flows stop changing their rate *if and only if* all flow rates are equal to the target rate (and hence to each other).

Likewise, PowerTCP [7] claims that RTT-gradient based CCAs ("current-based" in [7]) are more reactive and precise than RTT/delay/loss/ECN-based CCAs ("voltage-based" in [7]). We argue that reactivity is orthogonal to the choice of congestion signal. We can have exponentially fast convergence to both fairness and efficiency even when using "voltage-based" control (§ 6.4).

Finally, works like Poseidon [112] artificially distinguish AIMD control and "target scaling" (contracts) even though they are mathematically equivalent. AIMD *implicitly* creates a contract (or scaling/mapping between rate and congestion), while "target scaling" is an explicit contract.

5.1.3 "Contract-first" blueprints to avoid mistakes

While there is no exhaustive list of mistakes, by following a contract-first blueprint, many common mistakes can be avoided. We argue in contrast to other CCA design methodologies, our blueprints are more intuitive, actionable, and generalizable.

- Intuitive. The tradeoffs are obvious from the contract choice (after a few to no algebraic steps), forcing the designer to deliberately choose between uncomfortable tradeoffs and picking Pareto-optimal points.
- Actionable. contracts align with how our community designs CCAs: a variety of CCAs start with a target rate equation and build dynamics around it, e.g., Swift [73], TFRC [48].
- Generalizable. Our methodology applies to not only analytically designed CCAs, but also black-box implementations, such as Tao (Remy) [104, 116], and learned CCAs, such as PCC [41] (online-learning) and Astraea [84] (reinforcement-learning or RL).

Other methodologies often start from a utility function: local utilities (e.g., "$\log(\texttt{tput}) - \log(\texttt{delay})$" in Copa [18]), global utilities (e.g., NUM/Remy [116]), or reward functions in RL [84]. These hide tradeoffs giving an illusion of unilaterally optimizing latency or throughput when these are intertwined with other metrics like fairness, robustness, and generality (§ 6.1). Consequently, many designs pick sub-optimal tradeoffs as inadvertent artifacts of other design decisions (§ 5.2, § 6.1, § 6.3).

Design blueprint. To design a new CCA, first, (D1) pick a contract. In § 5.2, we define contracts as a function. Picking a contract involves deciding its (D1.1) input (e.g., delay, loss, ECN, etc.) and output (e.g., rate, cwnd, fraction of link, etc.), (D1.2) shape (e.g., linear, square-root, exponential, etc.), and (D1.3) parameters (e.g., scale, shift, clamps, etc.). § 6.1, § 6.3 and § 6.6 give guidance on how these choices affect steady-state performance. Then, (D2) implement dynamics to follow the contract. § 6.4 gives guidance on how the dynamics impact convergence time/stability along with other design considerations. Note that unless congestion control is solved, our list of considerations is necessarily incomplete.

Analysis blueprint. For analyzing an existing CCA, one should: (A1) compute its contract (§ 5.2.1), (A2) see where the contract lies in the tradeoff space (§ 6.1), (A3) identify any obvious issues due to shifts/clamps/intercepts in the contract (§ 6.3), and (A4) compare dynamics with those in § 6.4.

Fig. 5.2 shows why some form of agreement between flows is necessary to achieve fairness with end-to-end CCAs. While we do not formally prove this, we believe this agreement can always be represented as a contract function, and consequently, the tradeoffs induced by contracts are fundamental. The only way to work around the tradeoffs is to change the input/output in the contract to decouple performance metrics from the contract (§ 6.6). We believe that other efforts for improving steady-state performance are futile and will likely lead to reinventing a CCA already covered in the design/tradeoff space in § 6.1.

5.2 Contracts formally

Definition. The contract of a CCA is a function of the form

$$\texttt{average sending rate} = \texttt{func}(\texttt{aggregate statistic})$$

that "describes" the CCA's steady-state behavior (e.g., at time infinity) when competing with itself on a dumbbell topology. The aggregate statistic (e.g., delay, loss rate, etc.) is derived from the CCA's observations: the time series of sending and acknowledgment sequence numbers, along with any explicit signals (e.g., ECN [49], INT [112]). As a shorthand, we use "$r = \texttt{func}(s)$", or "$r = \texttt{f}(s)$".

Here, "describe" depends on the form of equilibrium or steady-state. Most CCAs exhibit a fixed-point or a limit-cycle equilibrium. In the fixed-point case (e.g., Vegas [87]), the set of fixed-points is same as the set of input/output pairs of `func`. If this set forms a relation rather than a function, the CCA is unfair and does not have a contract (e.g., TIMELY [92]). In the limit-cycle case (e.g., sawtooth behavior in Reno [63], DCTCP [12]), we convert the cycle into a fixed-point by considering the aggregate behavior (e.g., average sending rate) over the cycle. For instance, DCTCP's contract is "$\texttt{avg cwnd} = 1/(\texttt{avg ECN marking rate})^2$", derived in [12, 13].

In general, it is difficult to rigorously define contracts given CCA behaviors can be complex. We discuss possible extensions in Chapter 8. Our current definition captures all the CCAs we analyzed; for each CCA, we can either compute its contract or show that it is unfair and does not have a contract.

In § 5.1, we introduced the contract for a CCA as its encoding of fair shares into observable signals, which enforces a unique equilibrium in the fluid model equations. This "encoding" and the "steady-state behavior" are two equivalent views of a contract: the CCA follows the encoding in steady-state "revealing" the encoding in its steady-state behavior. We adopt the steady-state view in defining contracts as it provides a *constructive* definition, allowing one to derive the contract directly from a given CCA.

5.2.1 Computing contracts

We compute the contract of a CCA by analyzing its steady-state behavior. We run it analytically or empirically on a variety of dumbbell configurations (with different capacities, RTprops, buffer sizes, and flow counts) and collect the set of steady-state observations. Then, we group this set based on an aggregate statistic, such that observations with the same statistic value have the same rate. These equivalence classes define the CCA's contract function. When the same steady-state observations result in different rates, the CCA is unfair and does not have a contract.

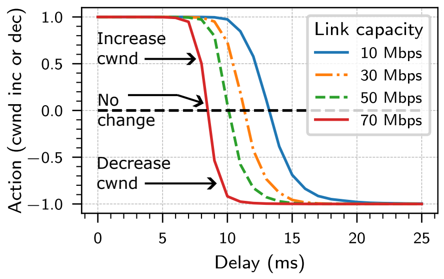


Figure 5.5: Astraea's state to action map (from [84]). Points on the dashed black line are the fixed-points.

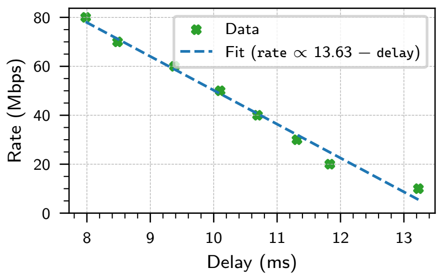


Figure 5.6: Curve fitting analytical fixed-points (from Fig. 5.5) to compute Astraea's contract.

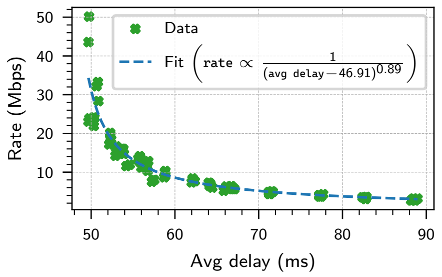


Figure 5.7: Curve fitting empirical fixed-points to compute BBR's contract.

Often, this reduces to program analysis of the CCA's code (see Vegas's example below) or fluid model analysis (see [124] for DCQCN [123]). One rarely has to compute contracts oneself. The literature has already computed contracts for most CCAs; we cite these in the "$\texttt{func}(s)$" column of Table 6.2. This allows us to focus on the performance impact of contract choice, instead of computing contracts. For completeness, we show below examples of analytical and empirical contract derivations.

Note, even for CCAs where fluid modeling is hard, their equilibrium behavior is well understood. For example, BBR [30] encodes fair shares in delay when cwnd-limited [20], and in "growth in delivery rate" when rate-limited [54]. In some cases the aggregate statistic is multi-dimensional and/or the contract is a compound function. For instance, Copa emulates Reno on detecting competing loss-based flows [18]. When the loss rate is zero, Copa follows a delay-based contract versus Reno's contract otherwise. For simplicity, we focus on homogeneous settings where flows use the same CCA and run in a single mode, e.g., we disable mode-switching in Copa. We expect the different modes to follow the tradeoffs according to the contract followed in the mode.

Analytical contract derivation (Vegas, taken from [106]). To update its cwnd, Vegas compares $\texttt{diff} = \texttt{expected\_throughput} - \texttt{actual\_throughput} = \texttt{cwnd}/\texttt{RTprop} - \texttt{cwnd}/\texttt{RTT}$ to $\alpha_{\texttt{rate}}$, where $\alpha_{\texttt{rate}}$ is a configurable parameter. Vegas increases cwnd when $\texttt{diff}$ is larger than $\alpha_{\texttt{rate}}$, and decreases otherwise. Fixed-point steady-state occurs when Vegas has no incentive to change its cwnd, i.e., $\texttt{diff} = \alpha_{\texttt{rate}}$. This equation gives us the contract. We substitute $\texttt{cwnd} = \texttt{rate}\cdot\texttt{RTT}$,

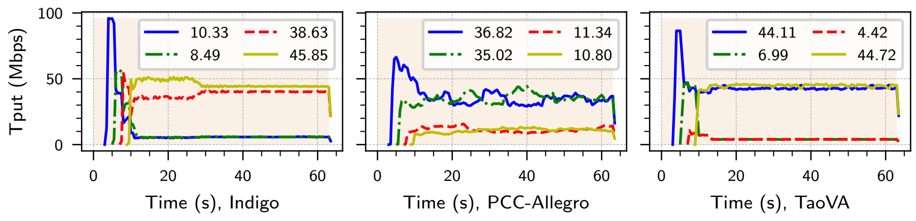


Figure 5.8: Indigo [120], Fillp/FillpSheep (TACK [82]), PCC-Allegro/Vivace/Experimental [41], and TaoVA (Remy) [104, 116] are unfair and have no contract. We only show three for brevity. We show the time series of throughput of 4 flows. The legend shows the time average throughput for the 4 flows.

CCA	Contract
BBR	$(\texttt{avg delay} - 46.91)^{-0.89}$
Vegas	$\texttt{avg delay}^{-1.38}$
Copa	$\texttt{p50 delay}^{-1.08}$
LEDBAT	$102 - \texttt{p50 delay}$
Sprout	$134 - \texttt{avg delay}$
Cubic	$\texttt{loss rate}^{-0.65}$

Table 5.1: Contracts computed for CCAs in Pantheon [120]. We omit constant scaling factors for brevity.

and $\texttt{RTT} = \texttt{RTprop} + \texttt{delay}$, and simplify, to get Vegas's contract: $\texttt{rate} = \alpha_{\texttt{rate}} \cdot \texttt{RTprop}/\texttt{delay}$.

Analytical contract derivation (Astraea, adapted from [84]). We can compute contracts even for black-box CCAs without running them. We show this for the RL-based CCA Astraea [84], which implicitly learns a contract. Fig. 5.5 shows its feedback (state) to action mapping. We obtain this by querying its neural network's action for different feature vectors (states). For a given capacity and delay combination (state), action > 1 increases cwnd, and action < 1 decreases cwnd. Astraea is at a fixed-point when it has no incentive to change cwnd (action = 0). For each link capacity (or fair share), Astraea maintains a unique delay, given by the X-coordinates of points with action = 0. We fit a curve ($\texttt{rate} = a(\texttt{delay} + b)^c + d$) to these points to get its contract (Fig. 5.6).

Empirical contract derivation (CCAs in Pantheon). To demonstrate that (1) contracts are easy to compute, and (2) fair CCAs have a contract, we empirically derive contracts for all CCAs in Pantheon [120] using an automated procedure. We use Pantheon to run each CCA for 60 seconds on a dumbbell topology with link capacities of 24, 48, and 96 Mbps, 40 ms RTprop, 4 BDP buffer, and vary the flow count from 2 to 8 (except Cubic, which uses 1 BDP buffer). These yield samples for throughput and three aggregate statistics: p50 delay (ms), avg delay (ms), and loss rate. We compute contracts by fitting a curve to these points (Fig. 5.7) and pick the statistic "s" and fit that minimizes mean squared error.

The configurations we picked are not special. One should pick the dumbbell configurations that best align with the target deployment of the CCA that one is analyzing. Most CCAs produce relatively continuous steady-state behaviors and interpolating/extrapolating steady-state behavior from a few configurations is often enough for computing contracts.

Of the 17 CCAs in Pantheon, we faced deprecation/dependency issues for QUIC Cubic and

Verus. We run the remainder 15. 2 CCAs (SCReAM and WebRTC) get ≈ 0 utilization (consistent with the Pantheon report [3]). 7 CCAs (Indigo, Fillp / FillpSheep, PCC-Allegro / Vivace / Experimental, TaoVA) are unfair (Fig. 5.8) and hence, do not have a contract. Unfair means different flow rates despite similar statistics/signals, indicating the absence of a unique signal-to-rate mapping (contract). Table 5.1 shows the contracts derived for remainder 6 CCAs.

Our automated procedure is for illustration purposes and is not full-proof. These contracts are for the range of networks we experiment with. The shifts in delays (e.g., 46.91 in BBR and 102 in LEDBAT) depend on the RTprop or buffer sizes, e.g., for BBR the shift is equal to RTprop. One would require more experiments to deduce this empirically. Similarly, the unfair CCAs may be fair and exhibit contracts on a narrower range of scenarios. For instance, we find that PCC-Vivace and PCC-Experimental exhibit the contracts "$\texttt{avg delay}^{-0.4}$" and "$\texttt{p50 delay}^{-0.46}$" respectively, if we only consider data from the 96 Mbps link. Finally, the contracts may be compound functions that take different shapes on different ranges of networks and use other statistics than the 3 we measured. For instance, [96] shows Reno's contract under different operating regimes: loss detections are dominated by timeouts vs duplicate ACKs.

5.2.2 Scope

We only consider strictly decreasing contracts, with closed intervals as their domain and range, ensuring they are continuous and invertible. All contracts that we are aware of are decreasing: the statistic typically measures congestion; an increasing contract suggests increasing the rate with increasing congestion. This further increases congestion, creating a positive feedback loop.

Rate- vs cwnd-based contracts. A contract can be written in terms of rate or cwnd, and using $\texttt{rate} = \texttt{cwnd}/\texttt{RTT}$ to convert between the two forms. Independent of the form, the CCA can be implemented using either rate or cwnd (§ 6.4). Without loss of generality, we consider rate-based contracts.

RTT or RTprop bias. Many CCAs have an RTT or RTprop bias, i.e., flows with different RTTs get different rates, e.g., Reno allocates more rate to flows with lower RTT. This bias shows up as a factor in the contract, e.g., Reno: $\texttt{rate} \propto 1/(\texttt{RTT}\sqrt{\texttt{loss\_rate}})$, Vegas: $\texttt{rate} = \alpha_{\texttt{rate}} \cdot \texttt{RTprop}/\texttt{delay}$. Even though RTT-bias has been a topic of interest in the congestion control literature, contracts show that unlike the tradeoff induced by the shape of contracts, the RTT bias is not fundamental, and that it can be easily removed. So we omit such factors in our presentation. For instance, many implementations of Vegas easily remove the bias by setting $\alpha_{\texttt{rate}} = \alpha_{\texttt{pkts}}/\texttt{RTprop}$ [1]. The same can be done for Reno by removing the RTT term from its contract and implementing the contract using TFRC [48].

Contracts in NUM. The concept of contracts can also be described using NUM [71, 85, 86, 106]. A contract maps the aggregate statistic (congestion measure or price) to rate, same as the demand function (target rate) for a given price [71]. The inverse of a contract represents the link price (e.g., target delay) for a given load (e.g., link's ingress rate) [71]. Consequently, the utility function that the CCA optimizes is (derived in [85]): $U(r) = \int \texttt{func}^{-1}(r)\, dr$ (5.2.1). Here, U is the utility derived from a rate of r. Eq. 5.2.1 only holds for statistics that add up over links (e.g., delay). For other statistics (e.g., max per-hop delay), the utility is different [112].

5.3 Chapter summary

We showed that to coordinate fairness, CCAs implicitly encode fair shares into observable congestion signals. We defined *contracts* as functions that represent this encoding and formalize these coordination mechanisms. Because CCAs must follow this encoding in steady-state, contracts manifest in their steady-state behavior. This allows us to *reverse engineer* the contract of a CCA by modeling its equilibrium behavior (e.g., limit-cycles or fixed-points).

Using this approach, we analytically and empirically derived the contracts for a variety of CCAs showing that all fair CCAs possess such a contract. In the next chapter we will show how these contracts—which are often an emergent artifact as opposed to explicit design choices—fully determine the steady-state performance metrics of CCAs, and use this to derive fundamental tradeoffs between these metrics. We will then show how these insights can be used to design better CCAs, providing blueprints for both CCA design and analysis.

Chapter 6

CCA design and analysis using contracts

In the previous chapter, we introduced contracts and saw that there is a wide diversity in the contracts of existing CCAs: they use different congestion signals (e.g., delay, loss rate, ECN marking rate), different functional shapes (e.g., linear, quadratic, square-root, etc.), and different shifts (e.g., "`rate` = 1/(`delay`- R)" [20, 30] vs. "`rate` = 1/`delay`"). In this chapter, we explore the performance consequences of these differences and uncover a surprising result: contracts—a design choice that most CCAs do not even make explicitly—*fully determine* key steady-state performance metrics in congestion control, including, throughput (on multi-bottleneck topologies), latency, fairness (on multi-bottleneck topologies), generality (range of link rates that the CCA supports) and robustness (tolerance to noise).

Specifically, we find that latency and generality are better when the contract is steeper (e.g., Swift [73] "1/`delay`^{2}"), whereas fairness and robustness are better when the contract is gradual (e.g., Vegas [27, 87] "1/`delay`"). Thus, no single contract can optimize all metrics simultaneously. Our analysis subsumes throughput under fairness definitions, where prior work has shown that throughput and fairness are at odds with each other. Additionally, we find that for a fixed contract, tolerating more congestion allows supporting a larger bandwidth range. In that sense latency and generality are also at odds.

Several of these tradeoffs have appeared in prior work. For instance, Arun et al. [20] discuss a special case where full generality precludes simultaneous robustness and bounded variation in congestion. Similarly, Remy [116] found that designing CCAs for a larger range of networks (generality) results in degraded latency. While in Remy, this was an "empirical" observation, with contracts, its obvious why this tradeoff exists and we can analytically quantify it. The NUM (Network Utility Maximization) literature [71, 85, 87, 106] studies a subset of these tradeoffs for individual CCAs. We believe we are the first to analyze a wider range of tradeoffs and generalize them to many CCAs.

Using the CCA contracts derived in the previous chapter, we construct a "periodic table" that shows where each algorithm lies in the tradeoff space and identifies the corner points of this space. We also show how implicit contract choices in many CCAs cause them to exhibit sub-optimal tradeoffs. Finally, we present canonical dynamics (i.e., `cwnd` updates) that CCAs should use to implement a chosen contract. We show that when CCAs choose a contract explicitly, MIMD `cwnd` updates allow the CCA to converge exponentially fast. In contrast, many CCAs blindly use AIMD updates—often justified by the Chiu-Jain plots—and suffer from unnecessarily slow convergence. For instance, while AIMD updates in Reno implicitly create a contract, AIMD updates are not necessary to create this contract, TFRC [48] uses the same contract but can choose different dynamics.

We start by defining the performance metrics, showing how contracts govern them and deriving the fundamental tradeoffs between the metrics (§ 6.1). We then use these results to substantiate our design and analysis blueprints (§ 6.2, § 6.4) and detail how implicit contract choices can lead to poor performance (§ 6.3). We validate our findings using simulation and emulation, finding a near-perfect match between analysis and measurements (§ 6.5). Finally, we discuss potential ways to work around the tradeoffs (§ 6.6).

In the next chapter (Chapter 7), we apply one of these workarounds to break the coupling between robustness and generality, enabling us to design FRCC: the first CCA that avoids starvation (robustness) on high link capacities (generality), thereby resolving the long-standing open-problem of starvation in end-to-end congestion control [20].

In Chapter 8, we discuss the limitations of contracts and how we may expand their uses. For instance, much of our current focus is on steady-state performance, and we hope to extend reasoning to short-flows and inter-CCA fairness. Further, if the network supports mechanisms like fair queueing, or if fairness is not desired, then contracts are not needed, and the tradeoffs do not apply.

6.1 Metrics and tradeoffs

Metrics. We list the metrics that CCAs typically optimize, and study how contracts affect them.

(1) Link utilization, flow throughput
(2) Amount of congestion (e.g., delay, loss)
(3) Stability, and convergence time/reactivity
(4) Fairness (on general topologies)
(5) Robustness to noise in congestion signals
(6) Generality (range of link rates, flow counts)

Of these metrics, fairness notions (see below) subsume Pareto-optimality; that is, the bottleneck links are fully utilized. They also subsume the tradeoff between total throughput and fairness (see below). Stability and convergence time are concerned with transient behavior as opposed to steady-state equilibrium. While contracts may affect them, we can often independently optimize them by choosing how fast a CCA's sending rate moves toward the target rate (§ 6.4). We are left with the following four metrics: robustness, fairness, congestion, and generality.

Approach. The choice of contract determines these four performance metrics. We show this by expressing each metric explicitly as a function of the contract `func`. Conversely, specifying a desired value for one metric constrains the contract, which in turn constrains the other metrics (i.e., a tradeoff). We derive these tradeoffs quantitatively by characterizing the feasible values

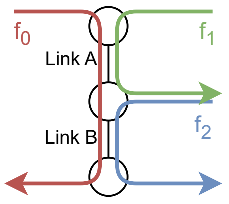


Figure 6.1: Parking lot topology

Fairness notion	α	Global utility	r_0	$r_1 = r_2$	Total rate, $\sum_i r_i$	Example CCA
Max throughput	0	$\sum_{i=0}^{k} r_i$	0	1	2	
Proportional	1	$\prod_{i=0}^{k} r_i$	1/3	2/3	5/3	Vegas
Min potential delay	2	$\sum_{i=0}^{k} -1/r_i$	$\sqrt{2}-1$	$2-\sqrt{2}$	$3-\sqrt{2}$	Reno
Max-min	∞	$\min_i r_i$	1/2	1/2	3/2	Poseidon

Table 6.1: Rates under different fairness notions on the parking lot topology (Fig. 6.1). Both links have capacity $C = 1$. r_i shows the rate of flow f_i. From top to bottom, fairness (rate equality) improves but total throughput (or rate) decreases.

of one metric under a desired constraint on another metric. We keep our metric definitions unit-less to minimize dependence on network parameters (e.g., link capacity, RTprop).

We use running examples of Vegas [27, 87] ($r = 1/s$) and Swift [73] ($r = 1/s^2$), where s is delay. We omit constant factors that ensure that the units are consistent. E.g., the unit of 1 in Vegas's and Swift's contract is bytes, and byte · seconds respectively, so that the function value has the unit bytes/second. Table 6.2 shows where existing CCAs fall in the tradeoff space.

Generality (e.g., range of link rates supported). Practical constraints limit congestion signals to a finite range, $s \in [S_{\min}, S_{\max}]$. Small delays and losses are difficult to measure due to noise, while large delays and losses increase application perceived latency. Since the contract maps congestion signals to rates, this range confines the CCA's operating range of link rates to $[\texttt{func}(S_{\max}), \texttt{func}(S_{\min})]$, where $\texttt{func}(S_{\min}) > \texttt{func}(S_{\max})$ because `func` is decreasing. CCAs must select the constants in `func` to suit the networks they operate on. For instance, if delay spans a 200× range, e.g., `delay` $\in [0.5\text{ ms}, 100\text{ ms}]$, Vegas can support only a 200× range of sending rates (e.g. 1 Mbps to 200 Mbps), whereas Swift can support a $200^2\times$ range (e.g. 1 Mbps to 40 Gbps). In general, steeper contracts enable a broader range of bandwidths for the same domain of the congestion signal.

Robustness to noise. We study the impact of "δs" error in the statistic "s" on the CCA's rate. We quantify the impact by looking at the (worst-case) ratio of rates without and with the noise: Error factor $= \max_s \frac{\texttt{func}(s)}{\texttt{func}(s+\delta s)}$ (6.1.1). A higher error factor means that the CCA is more sensitive to noise and is less robust. From Eq. 6.1.1, the error factors for Vegas ($r = 1/s$) and Swift ($r = 1/s^2$) are "$1 + \delta s/S_{\min}$", and "$(1 + \delta s/S_{\min})^2$" respectively. If δs error perturbs Vegas by 2×, then the same error perturbs Swift by 4×. Steeper contracts yield higher (worse) error factors (Fig. 5.4).

CCAs	`func`(s) = `f`(s)	`fold`	Robustness error for δs noise	Unfairness with k hops	Congestion for n flows	Range of bandwidths
			$\max \frac{f(s)}{f(s+\delta s)}$	$\max \frac{f(s_k)}{f(\mathtt{fold}(s_k,\ldots}$	$\max \frac{f^{-1}(C/N)}{f^{-1}(C)}$	$\frac{f(S_{\min})}{f(S_{\max})}$
			Lower is better	Lower is better	Lower is better	Higher is better
			Want gradual `f`	Want gradual `f`	Want steeper `f`	Want steeper `f`
FAST, Vegas, Copa	$1/s$ [20, 85, 115]	$\sum$	$1 + \delta s/S_{\min}$	k	n	$\frac{S_{\max}}{S_{\min}}$
Swift, DCTCP	$1/s^2$ [12, 13, 73]	$\sum$	$(1+\delta s/S_{\min})^2$	k^2	$\sqrt{n}$	$\left(\frac{S_{\max}}{S_{\min}}\right)^2$
Reno	$1/\sqrt{s}$ [63, 85, 89]	$\sum$	$\sqrt{1 + \delta s/S_{\min}}$	$\sqrt{k}$	n^2	$\sqrt{\frac{S_{\max}}{S_{\min}}}$
Poseidon	e^{-s} [112]	max	$e^{\delta s}$	1	$\log n$	$e^{S_{\max}-S_{\min}}$
α-fair (§ 6.1)	$1/\sqrt[\alpha]{s}$ (Eq. 5.2.1)	$\sum$	$\sqrt[\alpha]{1 + \delta s/S_{\min}}$	$\sqrt[\alpha]{k}$	n^α	$\sqrt[\alpha]{\frac{S_{\max}}{S_{\min}}}$
Exponential	e^{-s/S_0} [20]	$\sum$	$e^{\delta s/S_0}$	$e^{k\cdot S_{\max}/S_0}$	$\log n$	$e^{(S_{\max}-S_{\min})/S_0}$
ICC	$\log(S_0/s)$ [68]	$\sum$	∞	∞	$\left(\frac{S_0}{S_{\min}}\right)^{\frac{n-1}{n}}$	$\frac{\log S_0 - \log S_{\min}}{\log S_0 - \log S_{\max}}$
Astraea	$C_0(1 - s/S_0)$ (§ 5.2)	$\sum$	∞	∞	$\frac{nC_0 - C_{\max}}{n(C_0 - C_{\max})}$	$\frac{S_0 - S_{\min}}{S_0 - S_{\max}}$
AIMD on delay	$1/\sqrt{s}$ (§ 6.3)	(§ 6.3)	$\sqrt{1 + \delta s/S_{\min}}$	∞	n^2	$\sqrt{\frac{S_{\max}}{S_{\min}}}$

Table 6.2: Existing CCAs and the tradeoff space. We organize into 3 sections: good contracts (blue, top), corner points (green, middle), and bad contracts (pink, bottom). For Reno, s is loss rate. For DCTCP, s is ECN marking rate. For Poseidon s is max per-hop delay. For AIMD on delay, s is bytes between high delay. For all other CCAs s is delay. We assume loss and ECN marking rates are small enough to approximate `fold` as $\sum$. For ICC and Astraea, unfairness and robustness error are worst when $s \approx S_0$.

Note, the amount of noise we want to tolerate may depend on the statistic. E.g., delay may have tens of milliseconds of noise [55], while loss rate may have noise of a few percent [76].

Fairness. For general network topologies, there exist multiple notions of fairness (Table 6.1). These are typically parameterized using α-fair utility functions [93] given by: $U_\alpha(r) = \frac{r^{1-\alpha}}{1-\alpha}$, where U represents the utility a flow derives from a rate of r, and $\alpha > 0$. The rate allocation maximizes the sum of utilities across all flows ($\sum_i U_\alpha(r_i)$) (global utility). Larger values of α indicate greater fairness at the cost of total throughput [108]. Typically, $\alpha \geq 1$ is desired.

Not all contracts correspond to an α-fair utility. As a proxy, to compare the fairness of contracts, we study their behavior in the parking lot topology (Fig. 6.1). This topology exposes differences between fairness notions (Table 6.1), and is common in data centers (e.g., when inter- and intra-rack flows compete). In Fig. 6.1, each link has a capacity of C bytes/second and flows experience congestion signals from multiple congested links. The *long* flow (f_0) observes signals from two hops, while the *short* flows (f_1 and f_2) observe signals from only one hop. More generally, for k hops, we consider flows $\langle f_0, f_1, \ldots, f_k \rangle$, where f_0 observes signals from all k hops, and other flows see signals from a single hop.

We quantify unfairness as the (worst-case) throughput ratio of f_k to f_0. We derive unfairness using three steady-state equations. First, at each hop, the sum of the incoming rates is equal to the link capacity: $\forall i \geq 1,\ r_0 + r_i = C$ (6.1.2). Second, each flow's rate (r_i) and statistic (s_i) follow the contract $\forall i \geq 0,\ r_i = \texttt{func}(s_i)$ (6.1.3), Third, f_0 sees an accumulation of signals from all the hops: $s_0 = \texttt{fold}(s_1, s_2, \ldots, s_k)$ (6.1.4). Here, `fold` describes the statistic f_0 sees when other flows see $s_1, s_2, \ldots, s_k$. For delay, $s_0 = \sum_{i=1}^{k} s_i$, as delays add up over hops. For loss rate and ECN marking rate, $s_0 = 1 - \prod_{i=1}^{k}(1 - s_i)$, as survival probability gets multiplied over the hops. Note, when the absolute value of loss rates (or ECN marking rates) is small (e.g., 10^{-2}), then `fold` can be approximated as a sum. For max per-hop delay (e.g., Poseidon [112]), $s_0 = \max(s_1, s_2, \ldots, s_k)$.

From Eq. 6.1.2, we get $r_1 = r_2 = \ldots r_k$. Given, `func` is invertible, we get $s_1 = s_2 = \cdots = s_k$. Then, the worst-case throughput ratio is given by:

$$\max_C \frac{r_k}{r_0} = \frac{\texttt{func}(s_k)}{\texttt{func}(s_0)} = \max_C \frac{\texttt{func}(s_k)}{\texttt{func}(\texttt{fold}(s_k, s_k, \ldots, k \text{ times}))} \tag{6.1.5}$$

Here, s_k solves $\texttt{func}(s_k) + \texttt{func}(\texttt{fold}(s_k, s_k, \ldots, k \text{ times})) = C$ (from Eq. 6.1.2).

Substituting the contracts for Vegas and Swift, and `fold` as $\sum$, we get the throughput ratios of k and k^2 respectively. Steeper contracts imply worse fairness.

Congestion growth. We study how congestion grows with (decreasing) fair share or (increasing) flow count. Consider a dumbbell topology with capacity C and n flows. For each flow to get its fair share of $r_i = C/n$, the statistic needs to satisfy: $\forall i,\ r_i = C/n = \texttt{func}(s_i)$. This implies that $\texttt{func}^{-1}(C/n) = s_n = s_i$. We define worst-case growth (i.e., (s for n flows)/(s for 1 flow)) as: $\texttt{growth}(n) = \max_C \frac{\texttt{func}^{-1}(C/n)}{\texttt{func}^{-1}(C)}$ (6.1.6). For Vegas and Swift, $\texttt{growth}(n)$ is n and $\sqrt{n}$ respectively. Steeper contracts imply slower growth.

Tradeoff summary. Gradual contracts give better robustness and fairness, while steeper contracts give better congestion and generality. In § C.1, we mathematically show these tradeoffs by considering each pair of contending metrics. E.g., if we want error factor $\leq \epsilon_r$ for δs noise, then $\texttt{growth}(n)$ is $\Omega\left(\delta s \frac{\log(n)}{\log(\epsilon_r)}\right)$. These bounds are tight, i.e., we show CCAs that meet these bounds (see below).

The tradeoffs involving fairness depend on `fold`. We consider $\texttt{fold} \in \{\sum, \max, \min\}$. The tradeoff exists for $\texttt{fold} = \sum$, e.g., delay, loss rate, and ECN marking rate (assuming loss and ECN marking rates are small enough). There is no tradeoff for max and min, e.g., max per-hop delay, as this allows independently ensuring max-min fairness. So, for CCAs like Poseidon [112], the only tradeoff is between robustness versus congestion and generality.

There are exactly two corner points in this tradeoff space. If we fix desired robustness, e.g., we want to tolerate δs noise, then the Exponential CCA [20] gives the best generality and congestion growth, but it has poor fairness (Table 6.2). If we fix desired fairness, e.g., proportional fairness ($\alpha = 1$), then Vegas gives the best generality and congestion growth (for $\texttt{fold} = \sum$). In general, if we want α-fairness, then the contract "$\texttt{func}(s) = \frac{1}{\sqrt[\alpha]{s}}$" gives the best generality and congestion growth (for $\texttt{fold} = \sum$). We mathematically show that these are corner points in § C.1.

6.2 (Design blueprint) Guidance on picking contract (D1)

(D1.1) Input/output. Unless one chooses a workaround to the tradeoffs (§ 6.6), the choice of contract input is one of delay, loss rate, or ECN marking rate. This choice depends on buffer sizes, availability of ECN, and constraints on dynamics (§ 6.4). Similarly, outside of the workaround of coordinating fraction of link use (§ 6.6), the output of the contract is rate.

All the workarounds in § 6.6 either require in-network support (e.g., max per-hop-delay signal) or fundamental congestion control research. For instance, [20] shows that to ensure robustness and full generality (arbitrarily large link rates), CCAs must deliberately vary rate and create delay variations. This goes against our desire to have a stable application-level rate. In fact, RL-based CCAs [84] and Remy [116] update cwnd based on moving averages of historical signals. CCAs that are deterministic responses to such histories are incapable of creating deliberate rate/delay variations.

(D1.2) Shape. For the above input/output choices, the tradeoffs exist. One should decide which of the two corner points in the tradeoff space is preferred. The tradeoff point fixes the contract shape (e.g., $\texttt{rate} = 1/s$) and the asymptotics in the tradeoffs. Next, we want to tune the contract for deployment, i.e., picking parameters that control shift or scaling (e.g., a, b, c in $\texttt{rate} = a/(s-b)+c$). The scale (e.g., a) affects the constant factors in the tradeoffs, and shifts/clamps (e.g., b, c) should be avoided.

(D1.3) Shift/clamps. Shifting the contract right (positive b) changes the fairness properties (§ 6.3). Shifting left (negative b) restricts the range of link capacities by introducing a Y-intercept. Shifting down (negative c) introduces an X-intercept which severely degrades robustness and

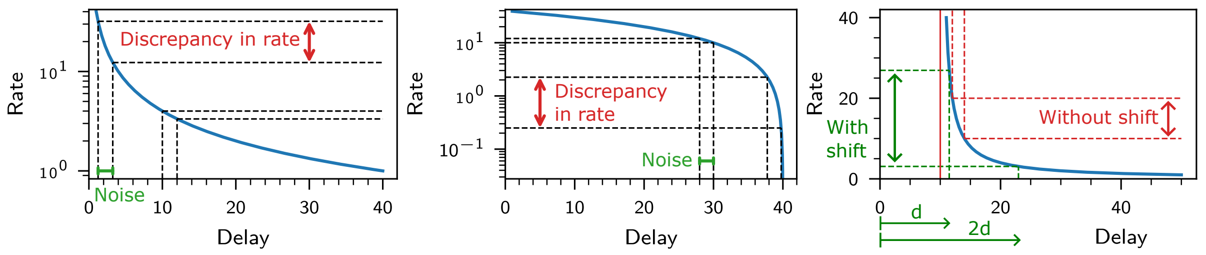


Figure 6.2: (Left) Vegas's contract (`rate` = 1/`delay`). The same noise creates larger discrepancies with increasing link capacities. **(Middle)** Astraea's contract (`rate` $= S_0 -$ `delay`) has an X-intercept. Discrepancies increase as delay approaches the intercept. **(Right)** Shifting the contract changes the fixed-point delays and rates on the parking lot topology. Solid red line shows Y-Axis without any shift in contract. The dotted lines show the rates of two flows at steady-state (red shows without shift and green shows with shift). Their delays are 2× apart (i.e., d and $2d$). We omit units (e.g., Mbps or ms) here as the shape/shift/intercept matter not the scale.

fairness (§ 6.3). Shifting up (positive c) only makes sense if there is a known lower bound on the fair share of a flow (e.g., fair share ≥ 512 Kbps). Clamping (e.g., $s = \min(a, 1/\texttt{rate})$) has similar consequences.

(D1.3) Scale. We discuss the scaling choice for the two corner contracts. We use C_0 and S_0 respectively to represent scaling of `rate` and s, e.g., `rate` $= C_0 e^{-s/S_0}$. One can set these parameters to meet a desired value for one of the four metrics. The tradeoffs decide the other metrics.

The general form of the exponential contract is `rate` $= C_0 e^{-s/S_0}$. C_0 is the maximum rate at which the CCA can ever send. If the link capacity is higher, the CCA would under-utilize the link. S_0 controls the robustness error (throughput ratio) for δs error in $s = C_{\max} e^{-s/S_0} / C_{\max} e^{-(s+\delta s)/S_0} = e^{\delta s/S_0}$. Larger S_0 gives better robustness but worse congestion. These parameters also decide the range of delays ($S_{\min}$, $S_{\max}$) produced by the CCA for a deployment's range of fair shares ($C_{\min}$, $C_{\max}$).

The general form of the α-fair contract is `rate` $= C_0 {S_0}^{\alpha}/s^{\alpha}$, or `rate` $= C_0 S_0/s$ for $\alpha = 1$. The contract passes through $\langle S_0, C_0 \rangle$: S_0 is the delay maintained when the fair share is C_0. The deployment's range of fair shares ($C_{\min}$, $C_{\max}$) implicitly decides the range of congestion ($S_{\min}$, $S_{\max}$), and robustness to noise. For small capacities ($C_{\min} \rightarrow 0$), $S_{\max}$ is ∞. For large capacities ($C_{\max} \rightarrow \infty$), $S_{\min}$ is 0 and the error factor is ∞ (as $\delta s/S_{\min} \rightarrow \infty$). When $C_{\max}$ is known (e.g., a data center), and one desires to maintain a minimum delay of $S_{\min}$ (to ensure utilization despite variations [84], or bound error factor by "$1 + \delta s/S_{\min}$"), an example contract choice is `rate` $= C_{\max} S_{\min}/$`delay`.

The exponential contract ensures a finite error factor at the cost of not supporting arbitrarily large link capacities. The α-fair contract supports arbitrarily large capacities but cannot bound the error factor. This is the same as the fundamental tradeoff in [20].

6.3 Learnings from contracts

Avoid extreme shape and intercepts. Astraea (linear contract) and ICC (logarithmic contract) exhibit poor robustness and fairness due to extreme shapes and X-intercepts. The issue is that all the low rates (e.g., 0.1 to 1 Mbps) map to delays near the X-intercept (e.g., 40 ms) (Fig. 6.2 middle). Consequently, small delay jitter creates large discrepancies in inferring fair shares. Similarly, on parking lot (Fig. 6.1), if the hop delays are close to the intercept delay, the long flow (f_0) observes their sum, which exceeds the intercept delay. Attempting to reduce this "excessive delay", f_0 reduces its rate to zero and starves.

Fig. 6.4 demonstrates these issues empirically. Ambient emulation noise causes Astraea to starve flows when the fair share is low (4 flows on a dumbbell topology with 10 Mbps capacity, 30 ms RTprop, and infinite buffer). ICC starves the long flow (f_0) on a parking lot topology with 3 flows (2 hops) and the same network parameters. Importantly, contract analysis reveals these performance issues without running the CCAs or understanding their internal working.

Note, unlike [20], the starvation here is not fundamental and is easily avoided by removing the X-intercept (e.g., `rate` = 1/`delay`). This increases the delay at low fair shares, however, this is unavoidable as transmission delays anyway grow as $1/C$ with decreasing capacity. In contrast, the issue in [20] is fundamental and is caused by the asymptote on the Y-axis, where all the high fair shares (e.g., $>$ 100 Mbps) map to near-zero delays (Fig. 6.2 left). This asymptote is hard to remove while supporting arbitrarily large rates with a monotonically decreasing contract.

Avoid shifting contracts. CCAs like Swift [73] define target delay as: `target_delay` = $b + 1/$`rate`. In such formulations, it may seem convenient to shift the contract (set a positive b) to maintain a minimum delay for high utilization. However, shifting changes the steady-state fixed-point causing undesired unfairness (Fig. 6.2 right). For instance, in the parking lot topology with 2 hops, the short and long flows see delays of d and $2d$ respectively. These delays are such that the corresponding contract rates add up to the link capacity. Shifting the contract rightward changes these delays in a way that increases discrepancies in the rates. We show this empirically for BBR in § 6.5. To avoid this issue, one should tune the scaling (§ 6.1) to maintain a minimum delay instead.

With active queue management (AQM), e.g., RED-based ECN marking [47, 49], we can increase queue buildup (delay) independent of the contract. We can increase the RED parameters $K_{\min}, K_{\max}$ to increase delay without changing the steady-state ECN marking probabilities (and rate allocations).

Avoid fixed thresholds (e.g., delay targets) at end-hosts. We illustrate how AIMD on delay (i.e., MD when delay crosses a fixed threshold and AI otherwise) causes starvation. AIMD on delay appears in 1RMA [103] and the CCAs (SMaRTT [24] and STrack [78]) proposed for standardization in the Ultra Ethernet Consortium [4]. It was also proposed by [20] to work around their impossibility result.

On a dumbbell topology, AIMD on delay works fine. It induces a Reno-like contract: "$r = 1/\sqrt{s}$", where s is the high-delay probability (inverse of bytes between high-delay events)

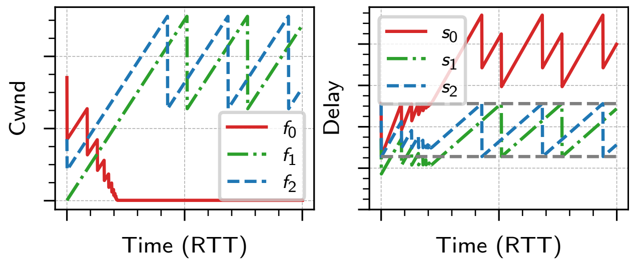


Figure 6.3: AIMD on delay starves the long flow (f_0) on parking lot topology.

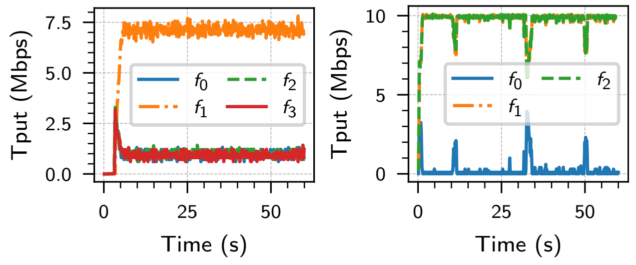


Figure 6.4: **(Left)** Astraea is brittle under ambient emulation noise. **(Right)** ICC starves on parking lot.

instead of loss probability. However, it creates starvation with multiple bottlenecks. All hosts attempt to maintain delay around the same (fixed) target value, but when flows observe delays from different combinations of hops, they cannot simultaneously maintain the same delay. Fig. 6.3 shows this on a 2-hop parking lot. The short flows (f_1 and f_2) oscillate delay between the delay threshold and half the threshold (shown in gray). The long flow (f_0) observes their sum, which always exceeds the threshold. Consequently, f_0 never increments cwnd and starves.

We can resolve this by either (1) not using a fixed threshold, e.g., Swift scales the (target delay) threshold with rate, which transforms the contract to use delay instead of high-delay probability, allowing different flows to maintain different delays, or (2) moving the threshold to links instead of end-hosts, e.g., packet drops (Reno) and ECN marks (DCTCP) occur when delay crosses a threshold at the links.

This issue may arise for any CCA that uses fixed end-to-end thresholds to decide when to increase or decrease their rate, e.g., BBRv3 uses `BBRLossThresh` = 2% [32].

Minor changes in CCA change the contract and consequently steady-state performance. MPRDMA [88], DCTCP [12], and Reno [63] all perform AIMD on binary feedback. In fact, [5] uses MPRDMA as an approximation of DCTCP. However, the minor differences in their design result in different contracts: Reno: "$r = 1/\sqrt{s}$", DCTCP: "$r = 1/s^2$", MPRDMA: "$r = 1/s$".

Similarly, we found an algebraic mistake in the Linux implementation of TCP Vegas that changes its contract from "$r = 1/\texttt{delay}$" to "$r = 1/\texttt{delay}^2$" when RTprop is small. We have confirmed this with the maintainers. This bug has existed for 17 years due to a refactoring commit [2]. Such bugs may be caught immediately if CCA implementations explicitly delineate contracts.

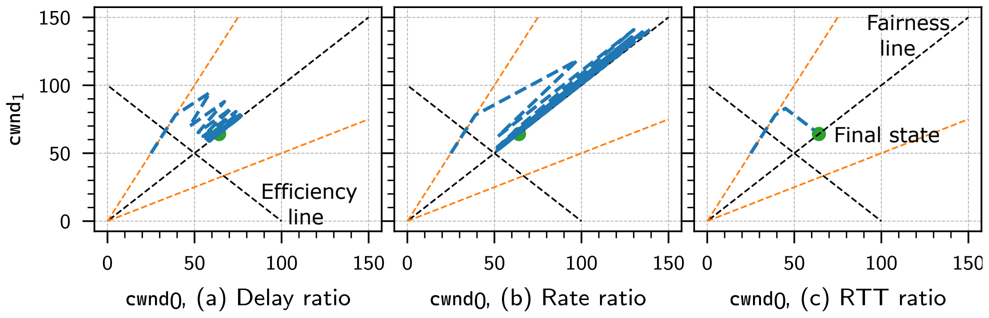


Figure 6.5: MIMD using RTT ratio is more stable than rate or delay ratio. Dynamics stay above the efficiency line showing no throughput loss, contrary to PowerTCP's claim on voltage-based CCAs [7].

6.4 (Design blueprint) Canonical CCA dynamics (D2)

We discuss how to best implement a CCA (cwnd or rate updates) to follow a contract. We can implement updates using either rate or cwnd, regardless of whether the contract is based on rate or cwnd. We discuss cwnd-based updates to follow a rate-based contract. Other combinations are similar.

While the contract fixes steady-state performance, the updates determine dynamics (stability and convergence time). Existing CCAs correspond to various ways of implementing cwnd updates to follow a contract. Vegas uses AIAD, Swift uses AIMD, Posiedon uses MIMD (using rate ratio), FAST uses (a different kind of) MIMD (using RTT ratio), and Copa uses AIAD with increasing gains when cwnd updates occur in the same direction.

We argue that TCP FAST's method is the best way to implement a contract. AIMD/AIAD are sub-optimal in convergence time. Copa uses convex—instead of concave (e.g., ETC [61])—changes to cwnd, which creates overshoots and instability. Ideally, we want to dampen changes to cwnd when it is already close to convergence [61]. Poseidon's MIMD creates instability depending on network parameters (see below). FAST's update is stable and converges exponentially fast to both efficiency and fairness. We illustrate (Fig. 6.5) the issue with Poseidon's MIMD and how FAST's MIMD fixes the issue in the context of delay, and later discuss other congestion signals.

MIMD using rate or delay ratio (C.f. Poseidon). We can interpret a contract (e.g., $\texttt{rate} = 1/\texttt{delay}^2$), in two ways: (1) *target rate* given *current delay* ($\texttt{target\_rate} = 1/\texttt{current\_delay}^2$), or (2) "*target delay* as a function of *current rate*" ($\texttt{target\_delay} = 1/\sqrt{\texttt{current\_rate}}$). Where, $\texttt{current\_rate} = \texttt{cwnd}/\texttt{RTT}$, and $\texttt{current\_delay} = \texttt{RTT} - \min \texttt{RTT}$. These yield two natural MIMD updates to implement a contract:

$$\texttt{target\_cwnd} \leftarrow \texttt{cwnd}\frac{\texttt{target\_rate}}{\texttt{current\_rate}} \text{ or } \texttt{cwnd}\frac{\texttt{target\_delay}}{\texttt{current\_delay}}$$

Where cwnd moves towards the target with optional clamps bounding the magnitude of change: "$\texttt{next\_cwnd} \leftarrow (1-\alpha)\cdot \texttt{cwnd} + \alpha\cdot \texttt{target\_cwnd}$" and "$\texttt{cwnd} \leftarrow \texttt{clamp}(\texttt{next\_cwnd}, \beta_{\min}\texttt{cwnd}, \beta_{\max}\texttt{cwnd})$". Outside of "$\texttt{rate} = 1/\texttt{delay}$", rate and delay ratios are different yielding different dynamics. For each α, clamp choice, both updates are only stable for specific network parameters (e.g., capacity, flow count).

MIMD using RTT ratio (C.f. FAST). To ensure stability, we want to consider how rate and delay affect or are affected by changes in cwnd. The following update achieves stability without requiring any averaging (α), clamps, or assumptions on network parameters:

$$\texttt{cwnd} \leftarrow \texttt{cwnd}\frac{\texttt{target\_RTT}}{\texttt{current\_RTT}} = \texttt{cwnd}\frac{\min\texttt{RTT} + \texttt{target\_delay}}{\min\texttt{RTT} + \texttt{current\_delay}}$$

Intuitively, cwnds map to "packets in queue + packets in pipe". We only want to move the "packets in queue" part from the current delay to the target delay. This `cwnd` update isolates the term responsible for queueing delay and only scales that term using the delay ratio. We can also interpret the update as: $\texttt{cwnd} \leftarrow \texttt{rate} * \texttt{target\_RTT} = \texttt{cwnd}/\texttt{RTT} * (\min\texttt{RTT} + \texttt{target\_delay})$.

For other signals, e.g., ECN or loss, the best cwnd update may differ. The current and target RTT in the update depends on the relation between RTT and aggregate statistic. For instance, for RED-based ECN, given `target_ECN_rate`, the `target_RTT` is "$\min\texttt{RTT} + (1/C)\cdot(K_{min} + (K_{max} - K_{min}) * \texttt{target\_ECN\_rate})$". We derive this by inverting the mapping from queue size (and hence queueing delay) to ECN marking probability. This works directly when ground truth link capacity "C" is known (e.g., a data center), otherwise either the capacity needs to be estimated or one needs to use delay or rate ratio with a value of α tuned for the range of network parameters one wants to support.

TFRC [48] explores cwnd update choices for loss-based contracts. The challenge is that while a constant cwnd creates constant delay and ECN rate, a constant cwnd may not create a constant loss rate (at least when loss rate is less than one loss per window). Thus, we may need cwnd variations to maintain a persistent loss rate even when target and current loss rates are the same.

Other design considerations. A complete the CCA needs other decisions including: (O1) how to aggregate multiple statistic samples, (O2) how long to measure the samples, (O3) time between cwnd updates, and (O4) how to compute any other estimates (e.g., bandwidth or RTprop estimate). For instance, Copa computes standing RTT by taking the "minimum" (O1) over RTT samples in the last "half srtt" (standard smoothed RTT) (O2). It updates cwnd "every ACK" (O3) and uses minimum RTT over last 10 seconds to estimate RTprop (O4). Alternatively, one can estimate RTprop using BBR's RTT probes. Implementations may also require other features like using rate instead of cwnd when the BDP is less than 1 packet [73]. Such details are orthogonal to contracts.

6.5 Empirical validation

We empirically validate the trends in metrics and tradeoffs predicted by contracts. For visual clarity in plots, we only show a handful of contracts/CCAs. This section complements the empirical results in § 5.2, § 6.1, and § 6.3; where we already showed performance issues and contracts for a large set of CCAs including Sprout, PCC, Indigo, ICC, Astraea, AIMD on delay (1RMA, SMaRTT, STrack, etc.).

Methodology. We use packet-level simulation and emulation. Simulations allow us to control noise and isolate one source of tradeoff at a time where emulation always has ambient noise, e.g.,

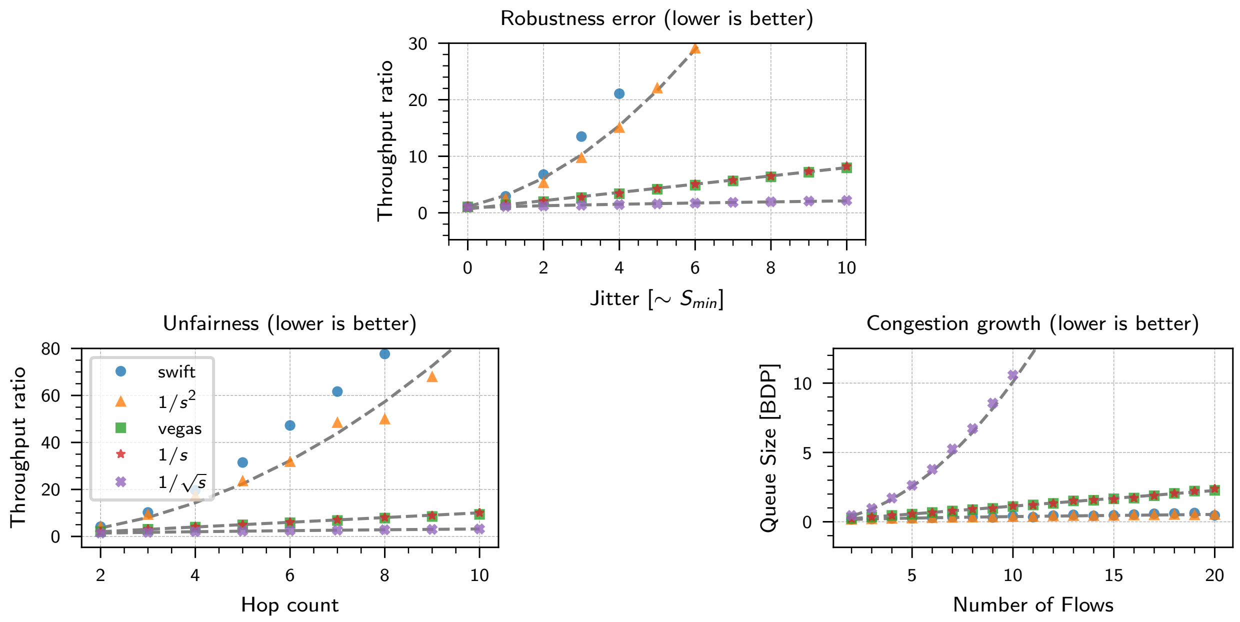


Figure 6.6: Contracts-based performance estimates match packet-level simulation. The markers show empirical data and gray lines show performance estimated by contracts. Fig. C.1 in § C.2 shows log-log scale. Note: markers may be hard to see because they overlap.

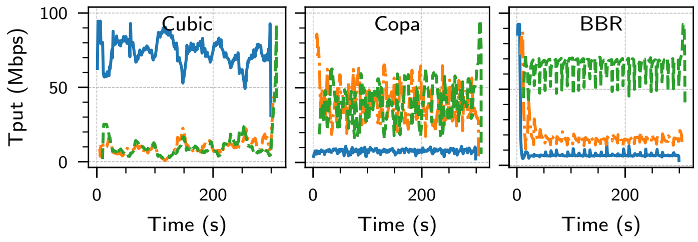
due to OS scheduling jitter. We use htsim [5] for simulation (previously used in MPTCP [100], NDP [62], EQDS [95]), and mahimahi [94], Pantheon [120], and mininet [39] for emulation.

In simulation only, for two reasons, we give oracular knowledge of RTprop to all CCAs. First, CCAs like Swift target data center deployments where RTprop may be known. For consistency, we provide RTprop to all CCAs. Second, CCAs like Vegas do not explicitly drain queues resulting in misestimating RTprop and poor performance. We remove this source of poor performance as this is not fundamental unlike the tradeoffs imposed by contracts. For instance, Copa and BBR explicitly drain queues to estimate RTprop accurately (at least in the absence of noise).

Note that we are validating negative results (i.e., tradeoffs). Simplifications only make the validation stronger. If tradeoffs exist with oracular knowledge of RTprop, then performance is only worse without it, e.g., under-estimation causes under-utilization, and over-estimation increases congestion.

Simulation CCAs. We implement and test Swift [73] and Vegas [27]. For reference, we also show 3 canonical (§ 6.4) contract implementations: $1/\sqrt{s}$, $1/s$, and $1/s^2$, where $s = \texttt{delay} = \texttt{RTT} - \texttt{RTprop}$. These avoid RTT-bias unlike vanilla Vegas/Swift, and use MIMD instead of AIAD/AIMD. In the canonical CCAs, we update cwnd every 2 RTTs and aggregate delay as the minimum over delay samples since the last cwnd update. To isolate the impact of contract shape, we also tune the scale parameters symmetrically. All CCAs have the same $S_{\text{min}} = 1.2\ \mu s = 0.1$ RTprop and $C_{\text{max}} = 100$ Gbps = link capacity. S_{max} is then decided by C_{min} and the contract shape.

Simulation scenarios. We set link capacity = 100 Gbps, RTprop = 12 μs, packet size = 4 KB. These are default parameters in htsim for data-center deployments. In § 6.1 we defined the metrics



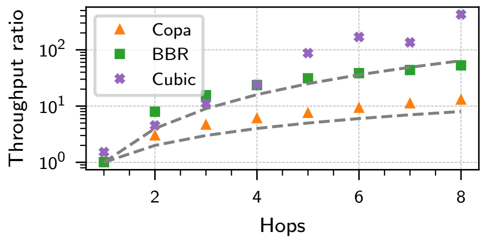


Figure 6.7: Robustness error (Left 3). Cubic starves the orange/green flows that do not witness jitter, while Copa starves the blue flow that witnesses jitter. BBR starves the blue flow with the smallest (10 ms) RTprop. **Unfairness (Right).** Grey lines show $y = x$ and $y = x^2$. Copa matches the unfairness predicted by its contract. BBR and Cubic are worse due to shift and RTT-bias.

in to be unitless and the tradeoffs we showed exist for all choices of network parameters (link capacity, RTprop, etc.). Consequently, the specific parameter values are of little importance and we could have used any other values as well. We set the buffer size to be infinite to remove any effects of packet losses, since we use delay-based CCAs. To measure robustness error, we use a dumbbell topology with 2 flows where one of them witnesses noise. We inject controlled error by adding a hop that persistently delays packets by "δs" μs, and vary δs. We do not include this in the RTprop provided to the CCAs. We inject noise this way to show trends. In emulation, we show the impact of realistic noise. To measure unfairness and congestion growth, we instantiate parking lot (with varying hops) and dumbbell (with varying flow count) topologies respectively. For the CCAs we test, generality is just the inverse of congestion growth (Table 6.2), so we do not show generality.

Simulation results. Empirical performance matches that estimated by contracts (Fig. 6.6). Swift's RTT-bias causes slightly worse fairness and robustness than the equivalent "$1/s^2$" canonical CCA which removes the bias. Vanilla Vegas has RTprop-bias (instead of RTT-bias). Since the RTprop is the same for all flows, the steady-state performance of Vegas is the same as the canonical "$1/s$" CCA.

Emulation CCAs, scenarios, and results. We run Cubic [59], BBR [30] (Linux kernel v5.15.0) and Copa [17, 18]. The empirical contract derivations (§ 5.2) already showed generality and congestion growth, e.g., increase in delay or loss rate with decreasing fair share (increasing flow count). In Fig. 6.7, we show robustness and fairness. We run flows for 5 mins on dumbbell and parking lot topologies with capacity = 100 Mbps, buffer = 1 BDP, and describe RTprop and flow count below. Emulation does not scale to data-center link speeds (unlike htsim). Here, our parameter choices align with Internet deployments. Again, the tradeoffs are independent of the absolute parameter values and we get qualitatively similar results with other values.

Note that introducing noise in signal may not create an equivalent amount of error in the statistic used by the CCA. Hence, we do not see a persistent trend in robustness error with varying noise. To validate that robustness is an issue, we show that the CCAs incur large throughput ratios (starvation) with small delay jitter. We inject jitter in two ways: (J1) slightly different RTprops (3 flows with RTprop of 10, 20, and 30 ms), and (J2) ACK-aggregation (3 flows with RTprop of 32 ms but 1 flow additionally witnesses 32 ms of ACK aggregation). We emulate ACK-aggregation in the same way as Pantheon [120]. In Fig. 6.7 (Left 3), Cubic and Copa show starvation with J2 and BBR shows starvation with J1. Note, BBR's unfairness in J1 is different from RTT-unfairness in traditional CCAs [59, 63]. For BBR, a small difference in RTprops leads to large unfairness that increases with the link rate [20].

Fig. 6.7 (Right) shows unfairness on parking lot with 5 ms RTprop. Copa matches the trend estimated by the contract. With BBR, the shape (derivative) of the contract is same as Copa. However the shift and RTT-bias in BBR's contract causes worse unfairness. For Cubic, the throughput ratio should be at least $\texttt{hops}^{4/3}$ (contract is $\texttt{rate} = \texttt{loss rate}^{-0.75}$ [81]). Reality is worse due to RTT-bias.

6.6 Working around the tradeoffs

As discussed in § 5.1, we believe the only way to work around the tradeoffs is to pick the input/output of the contract in a way that decouples physical quantities (e.g., rate or delay) from the contract. We show this for the four metrics.

Note that compound contract functions that take different shapes on different link rates or switch the shape on the fly do not alleviate the tradeoffs. In the worst-case, all the scenarios may occur simultaneously, e.g., multiple flows per hop on a parking lot topology with noise.

- **Fairness.** As mentioned in § 6.1, statistics that accumulate using max or min, like max per-hop delay, decouple multi-bottleneck fairness, trivially ensuring max-min fairness. However, such accumulation often relies on in-network support [112].
- **Congestion and robustness.** The congestion growth metric describes growth in the statistic and not congestion. Decoupling statistic and congestion allows independently bounding congestion. For instance, [124] shows use of a PI controller to obtain different ECN marking probabilities for the same queue buildup (i.e., different statistic for the same congestion). Likewise, explicit communication in packet headers (using enough precision) may eliminate noise to meet robustness [70].
- **Generality.** The domain of the statistic limits generality. Existing CCAs encode fair shares using a "unary" encoding. As proposed in [20], we can improve encoding efficiency using a "binary" encoding that communicates fair shares over time—similar to deriving multi-bit feedback from single ECN bit [55]. Another workaround is coordinate the "fraction of link use" (i.e., a quantity between 0 and 1) instead of "absolute fair shares" (i.e., an arbitrarily large number). This reduces the range of output values that a contract needs to support. BBR's rate-limited mode does this [54], but BBR often operates in cwnd-limited mode [113], without fully leveraging this workaround. Our new CCA, FRCC (described in Chapter 7), uses exactly this workaround to improve generality.

6.7 Chapter summary

We showed that contracts determine key performance metrics, resulting in tradeoffs. We identify pitfalls to avoid when designing CCAs. We hope that with our work, contracts will be a conscientious design choice rather than an afterthought. Contracts should be a direct consequence of desired steady-state performance, and rate updates should be a consequence of desired reactivity/convergence time.

In the next chapter (Chapter 7), we will use contracts to solve the open problem of starvation in congestion control, where all existing CCAs starve some flows when multiple flows compete on networks with jitter.

Chapter 7

FRCC: First starvation-free CCA

This chapter presents FRCC, the first congestion control algorithm that provably avoid starvation on networks with jitter and multiple competing flows. As we showed in Fig. 2.2 in Chapter 2, all existing CCAs starve flows in such scenarios. The *starvation theorem* from [20] shows that these are not one-off failures. The theorem states that any CCA with small self-induced delay or delay variation can starve in the presence of jitter (from sources like ACK aggregation or OS scheduling delays). To our knowledge, all existing CCAs that are buffer-bloat resistant (i.e., bound delays) cause little to no self-induced delay variations and, therefore, starve.

Given this result, we ask whether there exists an end-to-end CCA that can provably bound fairness, albeit at the cost of large (enough) self-induced delay and delay variations, while still bounding the maximum delay. We design FRCC (Fair and Robust Congestion Controller), the first such CCA with formal guarantees of fairness and efficiency under the same pessimistic, worst-case model used by [20]. With this, we can be confident that our design is robust in any real networks that can be emulated by the worst-case model, which includes components like token bucket filters and ACK aggregation (Chapter 2).

Contracts from Chapter 5 enabled us to better understand the root cause of starvation and discover key insights to overcome starvation. Since CCAs encode rate in congestion signals, noise in signals creates discrepancies in the fair rates inferred by different flows. Unfortunately, the contracts of all existing CCAs have an "*asymptote*", where a large range of fair rates are encoded in an narrow interval of congestion signals (§ 7.1). For instance, the contract for Copa "`rate` = 1/`delay`" has an asymptote at zero delay. Due to this, even an epsilon amount of noise (e.g., micro-seconds of jitter) can cause arbitrarily large errors in inferring fair rates, causing some of the flows to starve.

To work around this, FRCC decouples the coordination of fair rates into two parts. Existing CCAs coordinate fair "rates". Rate conveys two pieces of information: (1) capacity of the bottleneck link and (2) number of flows competing at the bottleneck. From ideas in beliefs (Chapter 3), we know that flows can vary their rate to independently probe for link capacity. We only need coordination across flows to infer the flow count. Capitalizing on this insight, FRCC only encodes flow count (or fair link fractions = 1/flow count) into congestion signals. This removes the asymptote in the contract allowing us to bound the error caused by noise.

We formally analyze FRCC's performance properties proving that it indeed avoids starvation (bounds fairness) on networks with jitter and multiple competing flows. Our proofs model the execution of FRCC as a dynamical system, describing both its transient and steady-state performance (§ 7.5). We cover a range of scenarios from ideal links (without jitter), links with jitter, and even flows vastly different RTprops and multiple-bottleneck links. To exhaustively reason about network behaviors, we use a number of analysis tools, including the Z3 SMT (satisfiability modulo theory) solver [40], and numerical methods [36]. Our formal approach also uncovered previously undocumented behaviors in `cwnd`-based CCAs that occur due to ACK-clocking (§ 7.4.3.2) and are unrelated to jitter. These are interesting in their own right independent of FRCC.

We implement FRCC in the Linux kernel and empirically evaluate it. FRCC's performance closely matches our theoretical analysis for all the challenging scenarios listed above and it consistently achieves fairness and efficiency, even when state-of-the-art CCAs exhibit starvation (e.g., Fig. 2.1, Fig. 2.2). We have open sourced our implementation, automated proofs, and empirical setup at `https://github.com/108anup/frcc` detailed in § D.1.

Note, our goal was to investigate if it is possible to design a CCA that operates at the limits of existing theoretical tradeoffs. FRCC's current design does not consider important practical issues such as coexistence with other CCAs, shallow-buffers, etc. (§ 7.1). Further, we had to make several assumptions (listed in § D.3) to make our proofs tractable. We conjecture that our results are true even without these assumptions as supported by our empirical evaluation (§ 7.6).

7.1 Background and motivation

Starvation theorem statement [20]. On networks with jitter, no end-to-end CCA can simultaneously achieve three desirable properties: (P1) avoid starvation, i.e., ensure a finite bound on the ratio of flow throughputs (s-fairness in [20]), (P2) scale to arbitrarily large link capacities (f-efficiency in [20]), and (P3) incur only small self-induced delay or delay variation (delay convergence in [20]).

This result applies only to end-to-end schemes, not to those relying on in-network support (e.g., XCP [70]) or active queue management (e.g. fair queuing). We focus on end-to-end CCAs because in-network mechanisms have been historically hard to deploy: explicit schemes require header changes (via IP or TCP options), which risk packet drops at legacy routers and incompatibility with encryption [55].

Goals. Our motivation is to address the practical starvation observed in Fig. 2.1 and Fig. 2.2, both of which arise in common scenarios. For example, two users in a household may have different RTprops when downloading from separate CDNs (e.g., YouTube and Amazon CloudFront, both using BBR), causing one flow to starve. Likewise, when one user connects via Ethernet and another via WiFi, the resulting differences in jitter lead to starvation under Cubic, Reno, or Copa. In the context of the starvation theorem, rectifying these issues requires achieving both (P1) bounded fairness and (P2) scaling to large capacities (e.g., 100 Mbps in Fig. 2.1 and Fig. 2.2). So we must give up (P3).

Consequently, our main goal was to determine if an end-to-end CCA can provably avoid starvation (achieve bounded fairness) and scale to arbitrarily large link capacities in the presence of jitter by accepting the tradeoff of creating large self-induced delays and delay variations.

Here, large is a relative term, and the absolute value of delay and delay variations can still be low. Specifically, these only need to be (1) larger than the amount of jitter in the network, and (2) be *independent of the link capacity*. CCAs, like Vegas/Copa/BBR, may maintain large delays or delay variations, but these vanish with increasing link capacity (§ 7.6), and their unfairness increases with link capacity, eventually leading to starvation. This also means that we can still bound the maximum delay and delay variation we create to avoid buffer-bloat, unlike traditional CCAs like Reno and Cubic.

This represents a challenging theoretical problem. To date, no delay-bounding end-to-end CCA has been proven to avoid starvation, particularly when faced with arbitrary jitter patterns as modeled by the CBR-delay network model (Fig. 7.11 and [20]). While these jitter patterns may seem pessimistic or rare, existing CCAs break under simple scenarios (Fig. 2.1, Fig. 2.2). We believe that designing FRCC for the pessimistic model, even though it may not capture all real-world scenarios, will make it robust to a wide range of network behaviors.

Beyond our primary objectives of bounded fairness, high utilization (scaling to arbitrarily large bandwidths), and bounded maximum delay (and delay variations), our secondary goal was to design a practical CCA. We partially achieve this goal. FRCC converges exponentially fast to both efficient and fair rate allocations (its convergence time is $O(N_G \log \mathtt{BDP})$ RTTs), where N_G is the number of flows.

Non-goals. As discussed in Chapter 2, in building FRCC, we do not address other properties that may be desirable for a CCA, including, coexistence with other CCAs, operation under shallow buffers, handling losses (including detection, recovery, and reaction), or short/application-limited flows.

Our analysis assumes buffers are sufficiently large to avoid loss. Here, sufficiently large is relative to the amount of jitter and not RTprop. Specifically, FRCC maintains a steady-state queueing delay of $O(N_G \mathcal{D})$ (§ 7.5). Where, $\mathcal{D}$ is the amount of jitter we want FRCC to tolerate. If RTprop = 100 ms, $N_G = 4$ flows, and $\mathcal{D} = 10$ ms, FRCC needs $4 \times 10 = 40$ ms of buffering. In contrast, CCAs like Reno/Cubic/BBR need $O(\text{RTprop})$ or 100 ms of buffering. In Chapter 4, we showed that buffering smaller than jitter is also an extremely challenging scenario involving an additional tradeoff between amount of packet loss and convergence time and we leave the combination of jitter, shallow buffers, and multiple flows to future work (Chapter 8).

Why does the starvation occur? Contracts (covered in Chapter 5) explain why starvation occurs. Specifically, we showed how since flows cannot directly talk to each other, they effectively coordinate by encoding fair rates into observable congestion signals. The issue with this coordination is that rate gets coupled with noise in congestion signals. The contract maps a small range of congestion signals to a large range of rates. As a result, even small noise in congestion signals creates large errors in inferring fair rates, causing different flows to disagree on what is the fair rate (Fig. 7.1). This disagreement increases with increasing link capacity, and

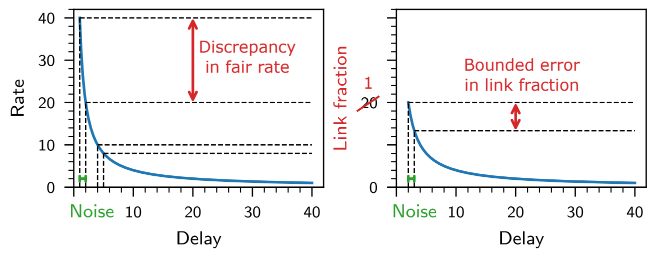


Figure 7.1: [Left] Noise creates discrepancies in inferring fair rates. The discrepancy increases with increasing link capacity. **[Right]** Link fraction gets rid of the asymptote on the Y-axis, ensuring bounded error in inferring fair link fraction for bounded jitter.

due to the asymptote on Y-axis, creates arbitrarily large ratios of flow throughputs (starvation) for arbitrarily large capacity.

7.2 Overview

7.2.1 Key insights to overcome starvation

Why do strawman solutions fail and key learnings. From Fig. 5.2 in contracts (Chapter 5), to coordinate fairness, we need some form of contract (e.g., rate-vs-delay mapping). And, the asymptote in such contracts is the main culprit causing starvation (Fig. 7.1). The problem is that there is no rate-vs-delay curve that (1) is asymptote-free, (2) scales to arbitrarily large rates, and (3) is monotonically decreasing. Monotonic decrease is needed as increasing rate with delay creates a positive feedback loop.

This explains why decades of research have failed to resolve starvation: no amount of noise filtering or parameter tuning can eliminate the asymptote. At high link capacities, even microsecond-level jitter causes large divergence in the inferred fair rates. Adjusting parameters only shifts the asymptote rather than removing it. For example, Vegas uses the contract $\texttt{rate} = \alpha_{\texttt{Vegas}}/\texttt{delay}$. Increasing $\alpha_{\texttt{Vegas}}$ allows Vegas to tolerate up to 10 ms of jitter on a 100 Mbps link by maintaining a 10 ms queueing delay. However, this tuning dramatically increases delay at lower capacities—e.g., 1 s of queueing at 1 Mbps—and the asymptote resurfaces at higher capacities (e.g., 1 Gbps). Similarly, changing the contract shape (e.g., $\texttt{rate} = 1/\texttt{delay}^2$ [73]) does not eliminate the asymptote, and contract proposals such as "$\texttt{rate} = C_{max}e^{-\texttt{delay}}$" [20] only eliminate the asymptote by give up scaling to arbitrary bandwidths.

First key insight. Instead of relying on a contract that coordinates the fair *rate* across flows, we split the challenge for determining fair rate into two parts. Our first insight is that inter-flow coordination should focus on the *fair link fraction*, i.e., the fraction of the bottleneck link that flows should occupy in steady-state. For example, we use the contract: "$\texttt{target\_link\_fraction} = \min(1, \mathcal{D}/\texttt{delay})$" (shown in Fig. 7.1). This eliminates the asymptote in the contract since the link fraction is naturally bounded by 1. Consequently, bounded noise in delay measurements creates bounded errors in inferring the fair link fraction. Here, $\mathcal{D}$ is a tunable bound on the amount of network jitter that we want to tolerate.

Each flow updates its `cwnd` to ensure that the fraction of the link it consumes is equal to the globally observable target fraction, so that in steady-state, all flows occupy the same fraction of the link (fairness). Flows increase their `cwnd` if their current fraction is less than the target and decrease otherwise.

Second key insight. The above requires that flows estimate the fraction of the link they currently consume. One way to achieve this is by estimating their throughput and the bottleneck link capacity, i.e., $\texttt{current\_fraction} = \texttt{throughput}/\texttt{capacity}$. To estimate capacity, flows can "probe" the link by temporarily increasing their `cwnd`. However, estimating link capacity is a surprisingly difficult problem [43, 74, 75], with accurate estimates requiring large probes (§ 7.4.3). Our second key insight is that our capacity estimates only need to be accurate enough for flows to determine if their current fraction is above or below the target so that they can change their `cwnd` in the correct direction.

We precisely compute how large capacity probes need to be, allowing us to make our probes, and hence, delay variation smaller than otherwise needed (§ 7.4.3). Specifically, our probes increase the `cwnd` by:

$$E = \frac{\texttt{throughput}}{\texttt{target\_fraction}} \cdot \gamma \mathcal{D} \tag{7.2.1}$$

where $\gamma > 1$ is a constant. This probe increases queuing delay by E/C seconds, i.e., all packets packets have to wait behind (the transmission delay of) the increased inflight. The starvation theorem does not apply to this design due to the resulting delay variation.

This probe is large enough for us to compare the target and current fraction. The intuition is that when the target and current fractions are close (their ratio is close to 1), the flow's $\texttt{throughput} \approx \texttt{target\_fraction} * C$. As a result, the probe size (Eq. 7.2.1) is close to $C\gamma\mathcal{D}$, and creates $\gamma\mathcal{D}$ delay. Jitter can only impact the expected delay by $\pm\mathcal{D}$. If the measured delay is more than $\gamma\mathcal{D} + \mathcal{D}$, then the probe must have created more than $\gamma\mathcal{D}$ delay, which can only happen if the flow's throughput is larger than the target fraction of the link. Conversely, if the excess delay is less than $\gamma\mathcal{D} - \mathcal{D}$, flow's throughput is smaller. When all flows' excess delays are close to $\gamma\mathcal{D}$, they consume a fraction of the link that is close to the target fraction (fairness).

Summary. We use delays larger than jitter to (accurately enough) coordinate fair link fractions, and delay variations larger than jitter to (accurately enough) estimate link capacity (and current link fraction). These allow us to ensure that the link fractions of all flows (and correspondingly the throughputs of the flows) are close to each other (fairness).

7.2.2 Leveraging the insights

We combine these insights to build a practical congestion control loop. For reference, Alg. 1 in § 7.2.3 provides FRCC's pseudocode. For convenience in implementation and analysis, we perform a change of variables:

$$\texttt{current\_flow\_count} = 1/\texttt{current\_link\_fraction}$$

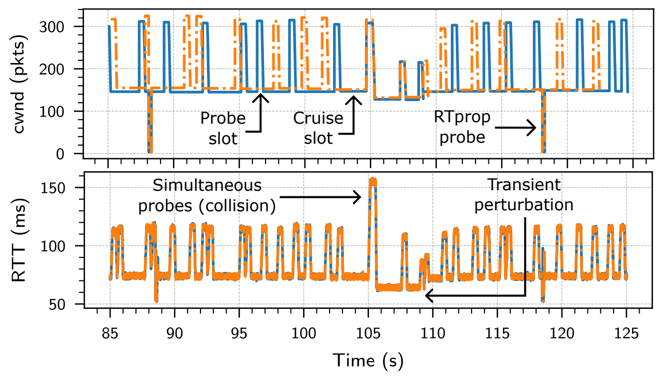


Figure 7.2: Timeseries of FRCC's `cwnd` and RTT.

$$\texttt{target\_flow\_count} = 1/\texttt{target\_link\_fraction}$$

FRCC tracks estimates for *four independent quantities*: (1) RTT ($\widehat{\texttt{rRTT}}$), (2) RTprop ($\widehat{R}$), (2) throughput ($\widehat{\texttt{rtput}}$), and (4) link capacity ($\widehat{C}$), where $\widehat{\texttt{rRTT}} - \widehat{R}$ gives `delay`. These give us the current ($\widehat{N_C}$) and target ($\widehat{N_T}$) flow counts as:

$$\widehat{N_C} = \frac{\widehat{C}}{\widehat{\texttt{rtput}}} \quad \text{and} \quad \widehat{N_T} = \texttt{Contract}_{\texttt{Func}}(\widehat{\texttt{rRTT}} - \widehat{R}) \tag{7.2.2}$$

Where $\texttt{Contract}_{\texttt{Func}}$ is our contract function (§ 7.4.1).

Since we cannot simultaneously measure all four quantities [30], we use a structure similar to BBR [30] to obtain estimates over time (Fig. 7.2). We divide time into rounds, which are further divided into slots. There are two types of slots: (1) probe and (2) cruise. Each flow independently selects exactly one slot in the round as its probe slot. Flows make this choice uniformly randomly and freshly for each round.

In the probe slot, FRCC temporarily increases `cwnd` to estimate link capacity ($\widehat{C}$). In the cruise slot, FRCC retains its `cwnd` to measure throughput ($\widehat{\texttt{rtput}}$) and RTT ($\widehat{\texttt{rRTT}}$). In addition to the slots, every $T_R = 30$ seconds, FRCC flows launch a synchronized RTprop probe to empty queues and estimate RTprop ($\widehat{R}$), similar to BBR [30]. To ensure synchronized draining, we assume clock synchronization (through the Network Time Protocol) to keep implementation simple, though we could use BBR's implicit synchronization as well (§ 7.4.6). Finally, at the end of the probe slot, FRCC updates `cwnd` to move the current and target flow counts closer to each other.

Note, we use a `cwnd` instead of rate to keep a closed-loop system and bound queues. We pace packets at the rate of $2 \cdot \texttt{cwnd}/\widehat{R}$ to avoid transmitting bursts while ensuring we do not get rate-limited. This is a common-case optimization and does not affect performance under worst-case jitter.

7.2.3 FRCC Pseudocode

For simplicity we do not show (1) RTprop probes (§ 7.4.6), and (2) pacing the probe's gain and drain over an RTT (instead the pseudocode just shows abrupt changes in `cwnd`) (§ 7.4.3), and (3) pacing rate.

Algorithm 1: FRCC pseudocode.

```
Function OnACK(Seq ack, RTT rtt)
    updateEstimates(ack, rtt)
    if probing & shouldInitProbeEnd() then  ▷ See § 7.4
        cwnd ⟵ prev_cwnd  ▷ Drain probe's excess
    end
    if slotEnded(ack, rtt) then  ▷ See § 7.4
        if probing then
            probing ⟵ False
            updateCwnd()  ▷ See Alg. 2
        end
        if shouldProbe() then  ▷ See § 7.4
            startProbe()
        end
        startSlot()  ▷ Reset $\widehat{\texttt{sRTT}}$ (if not probing)
    end
    if roundEnded() then  ▷ $kN_{max}$ slots elapsed
        ▷ Reset $\widehat{\texttt{rRTT}}$, $\widehat{\texttt{rtput}}$
    end
end
Function updateEstimates(Seq ack, RTT rtt)
    $\widehat{R} \leftarrow \min(\widehat{R}, rtt)$
    $\widehat{\texttt{rRTT}} \leftarrow \min(\widehat{\texttt{rRTT}}, rtt)$
    if probing & partOfProbe(ack) then  ▷ See § 7.4
        $\widehat{\Delta d} = \min(\widehat{\Delta d}, rtt - \widehat{\texttt{sRTT}})$
    else if not probing then
        $\widehat{\texttt{sRTT}} \leftarrow \min(\widehat{\texttt{sRTT}}, rtt)$
        $\widehat{\texttt{rtput}} \leftarrow \max(\widehat{\texttt{rtput}}, \texttt{cwnd}/rtt)$
    end
end
Function startProbe()
    probing ⟵ True
    $\widehat{\Delta d} \leftarrow \infty$
    prev_cwnd ⟵ cwnd
    $\widehat{N_T} \leftarrow (\widehat{\texttt{rRTT}} - \widehat{R})/\theta$
    $E \leftarrow \max(\gamma \widehat{N_T} \widehat{\texttt{rtput}} \mathcal{D}, 1)$
    cwnd ⟵ cwnd + $E$
end
```

7.3 Design requirements

Notation. We use hats to denote estimates (e.g., $\widehat{N_{C_i}}$, $\widehat{N_{T_i}}$) and no annotation to denote true values (e.g., N_{C_i}, N_T), and subscript of i for flow f_i. We drop the subscript when clear from context. You can find our glossary in Table D.1 and Table D.2.

Desired bounds. Flows update their `cwnd` based on their local estimates, $\widehat{N_{T_i}}$ and $\widehat{N_{C_i}}$. To ensure throughput ratios are bounded, we need to ensure: **(D1)** the target flow counts of all flows are within a multiplicative bound of each other, and **(D2)** each flow is able to infer when its true current flow count (N_{C_i}) (inverse link fraction) is multiplicatively far from its local target ($\widehat{N_{T_i}}$). On combining these, the true current flow count (inverse link fraction) N_{C_i} is close to local target ($\widehat{N_{T_i}}$), which is close to the single true global target (N_T). So the true link fractions (N_{C_i}) (and thus throughputs) are close to each other.

Meeting D1/D2. From Eq. 7.2.2, $\widehat{C}$ and $\widehat{\texttt{rtput}}$ affects error in $\widehat{N_C}$ and the shape of our $\texttt{Contract}_{\texttt{Func}}$ affects how errors in delay ($\widehat{\texttt{rRTT}}-\widehat{R}$) propagate to $\widehat{N_T}$ (D1). Finally, errors in both $\widehat{N_C}$ and $\widehat{N_T}$ affect (D2). It is challenging to directly reason about necessary error bounds on the four individual estimates to meet (D1/D2). We build $\widehat{C}$ using our second key insight (§ 7.2.1) and we build other estimates $\widehat{\texttt{rRTT}}$, $\widehat{R}$, and $\widehat{\texttt{rtput}}$ to be the best achievable under our error model below. We then show that our estimates are indeed sufficient to meet (D1/D2). We do this by encoding all the error bounds in the Z3 SMT solver [40] and show that FRCC converges to bound fairness (§ 7.5.3).

Error model. We need to consider three sources of errors: **(E1)** network jitter, **(E2)** RTT (and throughput) variations due to a combination of multiple hops (§ 7.4.3), difference in feedback delay (RTprop) of flows (§ 7.4.3), and interleaving of packets across flows (§ 7.4.5), and **(E3)** self-induced delay variations due to our capacity probes.

Tolerating E1/E2. (E2) errors are unrelated to network jitter and even occur on smooth, non-time varying links. In two instances, we were able to model these and mitigate their impact (§ 7.4.3, § 7.4.5). Outside of these two instances, we treat (E2) same as (E1). We design FRCC to tolerate a bounded (configurable) amount of cumulative errors due to (E1) and (E2). As described in Chapter 2, we formally model these errors as non-deterministic (or arbitrary) similar to prior work [19, 20], as opposed to stochastic (or random). Arbitrary delays (as opposed to stochastic delays) can express causal effects and correlations, allowing us to be provably robust to a number of sources of jitter in real networks [19].

Tolerating E3. (E3) affect us in two ways. First, when a flow probes, it temporarily inflates RTT and reduces throughput for other flows, potentially skewing their estimates of $\widehat{\texttt{rRTT}}$, $\widehat{R}$, and $\widehat{\texttt{rtput}}$. Fortunately, these are one-sided errors: RTTs only increase and throughputs only decrease. We filter them by computing $\widehat{\texttt{rRTT}}$ and $\widehat{R}$ as the min over RTT samples, and $\widehat{\texttt{rtput}}$ as the max over throughput samples collected in cruise slots. This bounds additive error in $\widehat{\texttt{rRTT}}$ (best achievable under (E1/E2)). For $\widehat{\texttt{rtput}}$, we measure throughput over packet-timed RTT

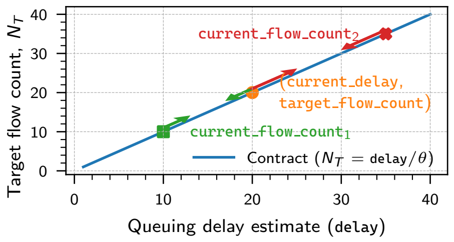


Figure 7.3: Contract used by FRCC. FRCC updates `cwnd` so that the target and estimated flow counts move towards each other.

intervals (i.e., the interval starts with sending a packet and ends with receiving its ACK) which bounds multiplicative error (also the best achievable as shown in § A.2). We show in § 7.4.5 that measuring throughput over other (even potentially longer) intervals is either sub-optimal or provides no additional benefit.

Second, if two flows probe for capacity together, their induced delays add up and they misestimate capacity ($\widehat{C}$). Our round and slot design is specifically to address the second issue. Probing in exactly one slot of the round reduces the likelihood of simultaneous probes from multiple flows (§ 7.4.4) and helps retain fairness despite probe collisions.

7.4 Design

We begin by describing FRCC's contract which governs its steady-state performance (§ 7.4.1). Then we describe how we perform `cwnd` updates to reach this steady-state (§ 7.4.2). The remaining subsections detail the design of our estimates and slots (§ 7.4.3, § 7.4.4, § 7.4.5, § 7.4.6). In these, tolerating the error bounds directly guides our design of estimates: $\widehat{N_T}$ (§ 7.4.1), $\widehat{C}$ (and $\widehat{N_C}$) (§ 7.4.3 for (E1/E2), and § 7.4.4 for (E3)), $\widehat{\texttt{rtput}}$ (§ 7.4.5), and $\widehat{R}$ (§ 7.4.6). We already covered $\widehat{\texttt{rRTT}}$ in § 7.3.

7.4.1 FRCC's Contract

We use the contract (Fig. 7.3):

$$\texttt{Flow count} = \texttt{Contract}_{\texttt{Func}}(\texttt{delay}) = \texttt{delay}/\theta \tag{7.4.1}$$

We interpret it in two ways: (1) "**target flow count** as a function of **current delay**" (N_T = `delay`/θ), and (2) "**target delay** as a function of **current flow count**" (`delay` = $\theta \cdot N_C$). The constant $\theta > 0$ denotes the seconds of delay we maintain per flow in steady-state, and amount of noise we tolerate: ΔR of noise in `delay` creates $\Delta R/\theta$ of additive error in $\widehat{N_T}$. Increasing θ increases error tolerance at the expense of delay. In steady-state, without jitter, $N_T = N_C = N_G$, where N_G is the ground truth number of competing flows (§ 7.5.2). I.e., all link fractions (and throughputs) are equal and there is θN_G of queuing delay.

Past CCAs have employed different contract shapes, e.g., linear (Copa/Vegas/FAST, `delay` = 1/`rate` [18, 27, 115]), square-root (Swift, `delay` = $1/\sqrt{\texttt{rate}}$ [73]), square (Reno [63, 89], `loss_rate` = $1/\texttt{rate}^2$). As we showed in Chapter 6, the shape affects three metrics: (M1) delay growth with flow count, (M2) tolerance to jitter, and (M3) fairness with multiple bottlenecks (e.g., Reno achieves minimum potential delay fairness, while Copa/Vegas/FAST achieves proportional fairness [106]). Since a single choice affects all these metrics, they are at odds with each other (Chapter 6). Note, in Chapter 6, we showed the tradeoff along four metrics. FRCC precisely decouples the fourth metric—"generality" (range of link rates)—from the tradeoff space enabling fairness and robustness at high link capacities.

While our contract bounds additive error in $\widehat{N_T}$, we only needed to bound multiplicative errors (for D1 in § 7.3). The logarithmic contract, $\texttt{delay} = \theta \log(N_C)$, gives the optimal tradeoff between delay (M1) and noise tolerance (M2) [20]. Delay grows logarithmically with flow count and multiplicative error in $\widehat{N_T}$ is bounded ($\widehat{N_T} = e^{(\texttt{delay} \pm \Delta R)/\theta} = e^{\pm \Delta R/\theta} \cdot N_T$). However, this creates unfairness with multiple bottlenecks (M3) (§ 7.5.5), so we use do not use this contract.

7.4.2 Congestion window update

At the end of the probing slot, flows update their `cwnd` to move the current and target flow counts closer to each other (Fig. 7.3). When current exceeds target ($\widehat{N_C} > \widehat{N_T}$), FRCC increases `cwnd`. This decreases the current flow count (increases link fraction), and increases the target (due to the increased delay). Likewise, when current is below target ($\widehat{N_C} < \widehat{N_T}$), FRCC decreases `cwnd`.

The `cwnd` updates are proportional to the gap between $\widehat{N_C}$ and $\widehat{N_T}$. This enables: (1) exponentially fast convergence to both efficiency (link utilization) and fairness in $O(\log \texttt{BDP})$ rounds, and (2) stability, as proportional updates dampen as FRCC gets closer to steady-state, similar to [61, 115].

Algorithm 2: FRCC `cwnd` update.

```
1  Function updateCwnd()
2      $\widehat{N_T} \leftarrow (\widehat{\texttt{rRTT}} - \widehat{R})/\theta$   ▹ $\texttt{cwnd} \leftarrow \texttt{cwnd}\,\delta_H$ if $\widehat{N_T} = 0$
3      $\widehat{C} \leftarrow E/\widehat{\Delta d}$   ▹ $\widehat{C} \leftarrow \infty$ if $\widehat{\Delta d} \leq 0$ (See § 7.4.3)
4      $\widehat{N_C} \leftarrow \max(\widehat{C}/\widehat{\texttt{rtput}}, 1)$   ▹ As $\texttt{rtput} \leq C$
5      $\widehat{N_C} \leftarrow \min(\max(\widehat{N_C}, \widehat{N_T}/\delta_L), \widehat{N_T}\delta_H)$   ▹ Clamps
6      $\texttt{target\_cwnd} \leftarrow \texttt{cwnd} \cdot \widehat{N_C}/\widehat{N_T}$   ▹ Alternate cwnd update
7      $\texttt{target\_cwnd} \leftarrow \texttt{cwnd} \cdot (\widehat{R} + \theta\widehat{N_C})/(\widehat{R} + \theta\widehat{N_T})$
8      $\texttt{cwnd} \leftarrow (1 - \alpha) \cdot \texttt{cwnd} + \alpha \cdot \texttt{target\_cwnd}$   ▹ $\alpha = 1$
9      $\texttt{cwnd} \leftarrow \textbf{ceil}(\texttt{cwnd})$   ▹ Ensures at least 1 packet inflight
10 end
```

Alg. 2 shows two `cwnd` updates: line 7 (main) and line 6 (alternate). The main update is more stable and we use it in all our analysis and implementation. We only use the alternate update in our proof for the jittery link in § 7.5.3 as Z3 times out for the main update (**Assumption 6.**).

The main update focuses on moving only the "queuing delay" portion of the RTT towards our target delay from $\theta\widehat{N_T}$ to $\theta\widehat{N_C}$. Intuitively, `cwnd` maps to "packets in queue + packets in pipe". The main update isolates and scales the term for "packets in queue", i.e., it is same as: $\texttt{target\_cwnd} = \texttt{rate} * \texttt{target\_RTT} = \texttt{cwnd}/\widehat{\texttt{rRTT}} * (\widehat{R} + \theta\widehat{N_C})$. In contrast, the alternate update is

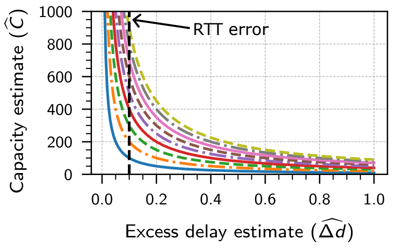


Figure 7.4: Lines show increasing E. A small range of excess delays map to a large range of potential link capacities. For $C = 600$, only the largest three probes create more than ΔR of excess delay.

more aggressive (less stable) and scales the entire `cwnd`. Specifically, the alternate update requires $\theta N_G > R$ for stability, i.e., the queuing portion of the `cwnd` is more dominant than the pipe portion.

Clamps on `cwnd`. The clamps (line 4 and line 5) help manage the impact of noise in estimates in three ways. First, while without jitter, FRCC converges to a fixed-point, with (worst-case) jitter, FRCC converges to a region around the fixed-point (§ 7.5.3). The clamps (among other parameters, and the amount of jitter) affect the size of the region. Tighter clamps imply a smaller region but slower convergence. Second, our capacity estimate is more accurate for some flows than others (§ 7.4.3.1). With worst-case errors, some flows may move away from the steady-state region. The clamps curb these bad movements. The onus is on the flows with accurate estimates to bring the system to convergence. Lastly, clamps help curb perturbations when FRCC misestimates capacity due to probe collisions.

7.4.3 Link capacity estimate

Prior work has explored various capacity estimation techniques, including packet-pairs/packet-trains [74, 75] and max ACK rate [30, 57]. All these methods have known issues, e.g., instead of estimating capacity, max ACK rate and packet-trains may only measure available bandwidth and asymptotic dispersion rate [43] when other flows consume a part of the link.

The three sources of errors (E1/E2/E3) make capacity estimation hard. We elaborate why and describe our solutions. Specifically, for (E1) we ensure that our capacity probes are large enough (§ 7.4.3.1), for (E2) we measure persistent change in delay created by the probes as opposed to transient change (§ 7.4.3.2) and for (E3) we have our slot/round design (§ 7.4.4).

7.4.3.1 Probe size

Problem (Fig. 7.4). Consider the simple estimator we used in § 7.2.1. It increases `cwnd` by E packets, and measures the excess delay created by the probe ($\widehat{\Delta d} = \Delta RTT$) to estimate capacity as: $\widehat{C} = E/\widehat{\Delta d}$. When there is no noise, the estimator is correct (as $\widehat{\Delta d} = \Delta d = E/C$). When there is noise, i.e., $\widehat{\Delta d} = \Delta d \pm \Delta R$, this estimator has a similar issue as we saw in Fig. 7.1. It maps a small range of $\widehat{\Delta d}$ values to a large range of $\widehat{C}$ values (Fig. 7.4). Small noise in $\widehat{\Delta d}$ ($= \Delta d \pm \Delta R$) creates large errors in $\widehat{C}$. The same issue happens for an estimator based on the increase in throughput (or ACK rate) due to probe.

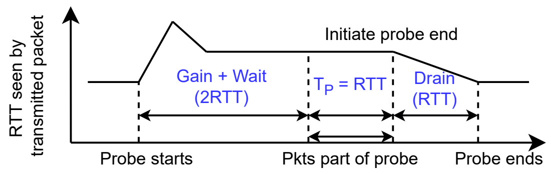


Figure 7.5: Life-cycle of a probe slot.

Unless $E > C\Delta R$, the excess delay is less than jitter, and the CCA cannot disambiguate between queueing delay and noise. C is the quantity we are trying to estimate, so we cannot set such a probe size a priori, to bound the error in $\widehat{C}$ and consequently $\widehat{N_C}$. Though it may be possible to binary search for such a probe size on the fly.

Solution. We only need to decipher if our current flow count is above or below target. For this purpose $E = \widehat{N_T}\widehat{\texttt{rtput}} \cdot \gamma\mathcal{D}$ is enough. Under this probe size, if we propagate the impact of errors from $\widehat{\Delta d}$ to $\widehat{N_C}$, we get (derivation in § D.2.1):

$$\frac{\widehat{N_T}}{\widehat{N_C}} = \frac{\widehat{\texttt{rtput}}}{\texttt{rtput}}\frac{\widehat{N_T}}{N_C} \pm \frac{\Delta R}{\gamma\mathcal{D}} = \gamma_R\frac{\widehat{N_T}}{N_C} \pm \frac{1}{\gamma} \tag{7.4.2}$$

Here, `rtput` and N_C are the true throughputs and current flow count (inverse fraction) of the flow, γ_R is the multiplicative error in throughput, and $\gamma\mathcal{D}$ controls the tradeoff between the seconds of delay variation we create, and the accuracy of our estimate. We get the second equality whenever $\Delta R \leq \mathcal{D}$. Thus, the total error in $\widehat{N_T}/\widehat{N_C}$ is bounded and the flows update `cwnd` in the correct direction when $\widehat{N_T}$ and N_C are far (D2).

Note, the additive error in $\widehat{N_T}/\widehat{N_C}$ causes some flows to have more accurate estimates than others. This asymmetry occurs in two ways. First, the estimate is more accurate for flows whose $\widehat{N_T}$ and N_C are far apart ($\widehat{N_T}/N_C$ is too large or too small). Second, when $\widehat{N_T}/N_C \in (0, 1)$, i.e., additive error has greater impact than when $\widehat{N_T}/N_C \in (1, \infty)$. In essence, flows getting more than their fair share have better estimates. While not all flows may move towards the steady-state region in a given round, flows that are too fast or too far from the fair share are more likely to move in the correct direction and update the global N_T to help bring the system to convergence.

Note, bounding multiplicative error in $\widehat{N_T}/\widehat{N_C}$ relative to $\widehat{N_T}/N_C$, to avoid asymmetries, is same bounding error in $\widehat{N_C}$ which requires $O(C)$ probe size as shown above.

7.4.3.2 Probe timing

With different RTprops, or multiple hops (either bottleneck or non-bottleneck), probes incur transient variations in RTT (and throughput) even without jitter. Bottleneck hops are those that have other competing FRCC flows (Fig. 7.15), while non-bottleneck hops do not have any competing flows (Fig. 7.6). With different RTprops and multiple *bottleneck* hops there is also *persistent* bias in excess delay, i.e., the *ground truth* $\Delta d \neq E/C$. We explain the cause and impact of persistent bias in § 7.5.5. Here, we discuss the transient variations. Both the variations and

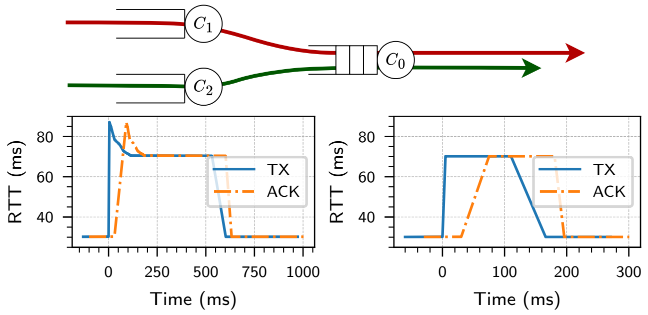


Figure 7.6: Packet trains may not measure the true link capacity. TX shows RTT of a packet vs. its transmission time, while ACK shows the RTT of a packet vs. its ACK time. **[Top]** C_1 and C_2 are non-bottleneck hops, C_0 is the bottleneck hop. Behavior in left plot occurs whenever $C_1, C_2 \geq C_0/2$. **[Left]** with extra hop (e.g., C_1, C_2), **[Right]** with only the bottleneck hop.

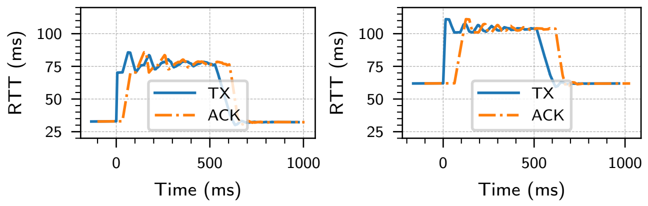


Figure 7.7: Oscillations occur during probes when flows have different RTprops. The excess delay without oscillations ≠ the transmission delay of excess packets. Oscillations dampen over time. Left shows probe of short RTprop flow and right shows long.

biases are interesting in their own right, they even occur in packet-level simulation, and to our knowledge have not been documented before.

Multiple hops. Fig. 7.6 shows that the immediate excess delay created by the probe is more than the transmission delay of the probe. Initially, the sending rate of the probe is higher than the capacity of the hops. As a result, transient packets queue up at the non-bottleneck hop, before also queuing at the bottleneck hop where the other flow is competing. The extra delay subsides when the packets become ACK-clocked in the next RTT.

Different RTprops. Fig. 7.7 shows oscillations in RTT caused by differences in feedback delay of flows. We provide intuition for the case when the longer RTprop flow probes (the other case is similar). At the start of the probe, the long flow's sending rate is high (increased delay). This reduces the ACK rate (and sending rate) of the short flow. The burst packets take a while to trigger new transmissions (in response to their ACKs) as the probing flow's RTprop is long (decreased delay). Each time the burst gets paced by the bottleneck and reduces in rate and eventually dies out.

Design of probe. In lieu of these transient variations, we make two choices (illustrated in Fig. 7.5): (1) we increase and decrease our `cwnd` over an RTT instead of abruptly (this reduces the magnitude of oscillations (Fig. 7.8), and (2) we wait for two RTTs before starting to measure excess delay. This eliminates the variations due to multiple hops, but the feedback delay based

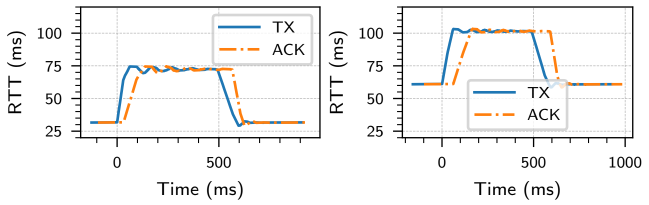


Figure 7.8: Paced probes reduce oscillation magnitudes. Left shows probe of short RTprop flow, and right shows long.

variations may not fully vanish.

We account any oscillations that do not vanish under (E1). This is okay because these oscillations are bounded independent of our design parameters, i.e., (P1) they largely depend on the differences in RTprops, and (P2) do not asymptotically grow with our probe sizes. As a result, the relative impact of these vanishes as we increase our delay/delay variations. We found it hard to mathematically prove (P1/P2) and empirically validated them (not shown).

We measure excess delay as the minimum RTT over the "packets part of probe" in Fig. 7.5 minus the minimum RTT in the slot before the probe.

7.4.4 Duration and number of slots

Duration of slots. Since slots allow us to coordinate probes, all slots, including probe and cruise, need to have the same duration, i.e., 4RTTs (§ 7.4.3.2). We conservatively set the slot size as the 4× the maximum RTT observed in the slot (for both probe and cruise slots). Note, slot durations may differ across flows as RTTs may differ due to noise or differences in RTprops. Empirically, FRCC works despite this (§ 7.6). Our formal proofs assume that slot durations are equal (**Assumption 2.**). We experimented with a design that hard-codes a large enough slot duration, but deprecated this as FRCC empirically worked without it.

Number of slots. When probes overlap, flows may underestimate capacity (and N_C) and incorrectly decrease `cwnd`. Underestimation occurs because excess delays add up ($\widehat{\Delta d}$ is larger so $\widehat{C} = E/\widehat{\Delta d}$ is smaller).

More slots decrease the probability of such overlaps at the cost of increasing convergence time. We dynamically set the number of slots to be linear in flow count as $k\widehat{N_T}$, with the intuition that $\widehat{N_T} = N_G$ in steady-state. In transient state, $\widehat{N_T}$ may not equal N_G. To ensure enough slots, we set the minimum number of slots to $K_{\min} = 6$. When there are too many probes for the number of slots, the natural increase in delay will increase $\widehat{N_T}$ and the number of slots. For practical reasons, our implementation also clamps slot count above ($K_{\max} = 20$). C.f. BBR [30] uses 8 slots. We investigate the impact of the upper clamp empirically (§ D.5).

Note that with slot count linear in N_G, while the probability of *some* probes colliding is high due to the birthday paradox, the probability of *a particular flow* colliding is low (§ D.2.2). Thus, each flow has infinitely many rounds where its probes do not collide, and it makes progress

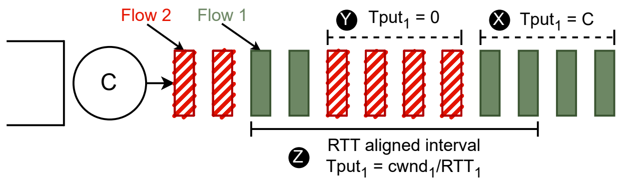


Figure 7.9: Measuring $\widehat{\texttt{rtput}}$ over non-RTT-aligned intervals can yield a measurement of 0 or C. Measuring over RTT-aligned intervals better matches throughput.

towards fairness. Additionally, flows that collide yield bandwidth to other flows, but each flow is equally likely to collide, so on average, the flow throughputs are still equal (§ 7.5.4, § 7.6).

7.4.5 Measuring throughput over packet-timed RTTs

We argue that for `cwnd`-based CCAs (1) $\widehat{\texttt{rtput}}$ should be measured over intervals aligned with packet-timed RTTs (i.e., the interval starts with sending a packet and ends with receiving its ACK), and (2) that measuring over multiple aligned intervals (to get longer intervals) provides no additional benefit.

As shown in Fig. 7.9, the throughput measured over non-RTT-aligned intervals can be anywhere between 0 and C (the link capacity). This occurs due to packet interleaving across flows and exists even on ideal links without jitter. Specifically, in interval X, only packets of flow 1 are served, resulting in throughput equal to C, while in interval Y, only packets of flow 2 are served, yielding zero throughput for flow 1.

In contrast, in the RTT-aligned interval Z, the packets ACKed, between sending a packet and receiving its ACK, are exactly those that were in flight when the packet was sent, which equals the flow's `cwnd`. This gives a throughput estimate of `cwnd`/RTT, where RTT is the round-trip time seen by the last ACK in the interval. While this RTT value may contain noise due to jitter, such noise can persist indefinitely under our network model, causing the same throughput measurement to repeat. Consequently, throughput measurements over multiple RTT-aligned intervals offer no improvement in the worst-case.

Note, this does not contradict Chapter 4, which showed that longer measurement intervals yield better estimates, as this is true for rate-based control. Specifically, with rate-based control, packets transmitted (and ACKed) within an RTT may not equal `cwnd` (when the `cwnd` cap is not hit, or if there is no `cwnd`).

Note, pacing may only partially address interleaving. Jitter before the bottleneck can still reorder packets across flows. Further, with `cwnd`-based control, pacing yields no improvement over RTT-aligned estimates anyway.

7.4.6 RTprop estimate

We can only measure RTprop when queues are drained [30]; however, FRCC maintains θN_G seconds of delay in steady-state. To drain this queue, we launch RTprop probes every $T_R = 30$ seconds, similar to BBR [30]. During these probes, FRCC reduces `cwnd` to 4 packets for one "max

RTT", a conservative upper bound on the RTT of competing flows. This ensures that the probe lasts long enough for all flows, with potentially different RTprops, to simultaneously observe the drain.

We set max RTT to "R_{max}+ max slot delay", where max slot delay is the maximum of RTT $-\widehat{R}$ samples in the latest slot. We use R_{max} = 100 ms. This is similar to BBR's RTT probe duration of (hard-coded) 200 ms [30].

Note, this only works if the RTprop probes occur together. We assume that senders' clocks are synchronized and launch the probe every 30th second on the clock. This keeps our implementation simple. Today, NTP (network time protocol) makes it relatively easy to get clock synchronization with < 10 millisecond-level accuracy. Note, this accuracy depends on RTprop between NTP servers and senders, and not on the RTprop between senders or between senders and receivers.

Note that we could instead use BBR's mechanism of launching RTT probes when the $\widehat{R}$ expires (i.e., 10 seconds elapsed since $\widehat{R}$ last changed) to implicitly synchronize the RTT probes. In our empirical evaluation, BBR's RTT probe always synchronize regardless of injected jitter, though we do not have a formal proof for this.

7.5 Analysis and proofs

We describe highlights of our analyses, with details in § D.3. We use three network models corresponding to our error models (§ 7.3): (1) jittery link (E1/E2/E3) (Fig. 7.11), (2) ideal link (E2/E3), and (3) fluid model (E3) only.

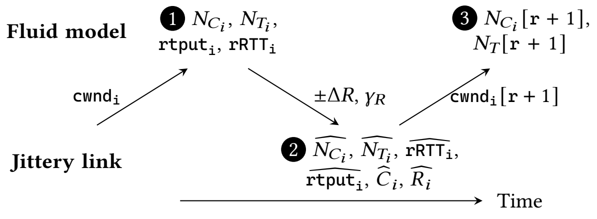
We analyze FRCC under four different settings, each providing unique insights into the FRCC's dynamics and performance guarantees. Our split of these four settings is governed by the tractability of the analyses tools under non-linearity (different RTprops and multiple bottlenecks), stochasticity (probe collisions) and non-determinism (jitter).

§ 7.5.2 Ideal link, N_G flows, equal RTprop, no probe collisions.
§ 7.5.3 Jittery link, **2 flows**, equal RTprop, no probe collisions.
§ 7.5.4 Ideal link, **2 flows**, equal RTprop, **with collisions**.
§ 7.5.5 Fluid model, different RTProps & multiple bottlenecks.

Based on our analysis of these settings, we conjecture that FRCC guarantees bounded fairness with high probability (over the algorithm's randomness in choosing the probe slot) on well-behaved network topologies (i.e., full rank routing matrix [115]), including those with jitter and multiple hops.

7.5.1 Approach

Our analyses proceed by deriving a state transition function that describes how the system state evolves after each round of FRCC's `cwnd` updates. We use a three step process to derive this function (Fig. 7.10):



Fluid model
❶ N_{C_i}, N_{T_i}, $\texttt{rtput}_\texttt{i}$, $\texttt{rRTT}_\texttt{i}$
❸ $N_{C_i}[\texttt{r}+1]$, $N_T[\texttt{r}+1]$
$\texttt{cwnd}_\texttt{i}$
$\pm\Delta R$, γ_R
$\texttt{cwnd}_\texttt{i}[\texttt{r}+1]$
Jittery link
❷ $\widehat{N_{C_i}}$, $\widehat{N_{T_i}}$, $\widehat{\texttt{rRTT}_\texttt{i}}$, $\widehat{\texttt{rtput}_\texttt{i}}$, $\widehat{C_i}$, $\widehat{R_i}$
Time

Figure 7.10: Process for deriving state transition function.

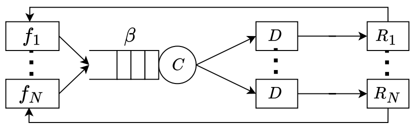


Figure 7.11: Jittery link (CBR-delay model from [20]). N_G flows, f_1 to f_{N_G}, on a dumbbell topology, with capacity C pkts/sec, infinite buffer, and RTprop of R_i for flow f_i. The jitter boxes can delay packets by D seconds non-deterministically (or arbitrarily) to model (E1/E2) errors in § 7.4. **Ideal link** uses $D = 0$.

Notation. We represent state in terms of the current and target flow counts. We use the suffix $[\texttt{r}+1]$ to denote state in next round, $\langle N_{C_i}[\texttt{r}+1], N_T[\texttt{r}+1]\rangle$, and no suffix for current round.

❶ Fluid reference execution. We want to make statements about the gap between flow throughputs and error between our estimates and true values. However, due to (E1/E2), even when all `cwnd`s, link capacities, flow count are fixed (non-time varying), there are variations in RTT and throughput. So there is no single true value of RTT, throughput, or link fraction. To deal with this, we define a "fluid reference execution", to define true values. Our proofs describe how these **reference** values (or state) evolve over time. The **real** state, RTT, and throughput hover around the reference values (§ 7.5.3).

We define the hypothetical fluid reference execution as one that runs parallel to the real one but without (E1/E2). It shares the same network parameters as the real execution (e.g., C, N_G, R_i, etc.). Given some initial $\texttt{cwnd}_\texttt{i}$, we define the reference (true) values for **rRTT**, **rtput**, (and consequently, N_{C_i}, N_T), as the RTT and throughput in the fluid execution. These are solutions to the two fluid model equations: (1) $\texttt{rtput}_\texttt{i} = \texttt{cwnd}_\texttt{i}/\texttt{rRTT}_\texttt{i}$ and (2) $\sum_i \texttt{rtput}_\texttt{i} = C$. These yield a unique solution to $\texttt{rRTT}_\texttt{i}$ and $\texttt{rtput}_\texttt{i}$ [115], and consequently N_{C_i} $(= C/\texttt{rtput}_\texttt{i})$ and N_T $(= (\texttt{rRTT}_\texttt{i} - R_i)/\theta)$. Note, we use the same reasoning to define the excess delay created by a probe of size E as $\Delta d = E/C$.

❷ Bounds on estimates. We respectively define ΔR and γ_R[1] as additive error in a RTT measurement and multiplicative error in a throughput measurement in the real execution relative to the reference $\texttt{rtput}_\texttt{i}$ and $\texttt{rRTT}_\texttt{i}$. These represent cumulative errors due to (E1/E2). Using these, we derive bounds on FRCC's estimates ($\widehat{\texttt{rRTT}_\texttt{i}}$, $\widehat{\texttt{rtput}_\texttt{i}}$, $\widehat{N_{C_i}}$, $\widehat{N_{T_i}}$) relative to the reference state (similar to Eq. 7.4.2).

[1]While ΔR is a pure network parameter. γ_R depends on the RTT we maintain. Larger θ decreases γ_R (γ_R is effectively $\frac{RTT}{RTT\pm\Delta R}$, from $\frac{\widehat{\texttt{rtput}_\texttt{i}}}{\texttt{rtput}_\texttt{i}}$).

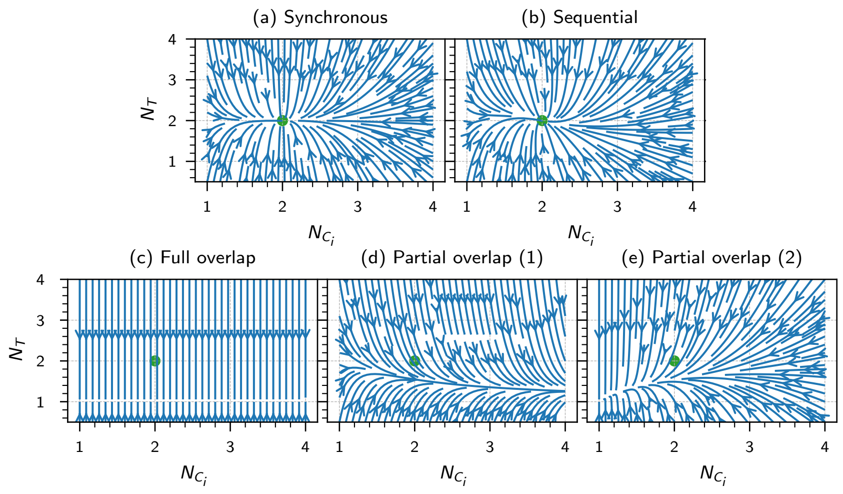


Figure 7.12: Evolution of $\langle N_{C_i}, N_T \rangle$ over rounds on ideal link for $N_G = 2$ flows. In all the overlaps, N_T moves towards 1. With partial overlap, N_{C_i} increases (link fraction decreases) for the flow that witnesses combined excess delay (d) and decreases for the other flow (e).

❸ **Next state(s).** We plug-in the bounds into FRCC's `cwnd` update rules (§ 7.4.2). This yields the (possible) $\texttt{cwnd}_\texttt{i}[\texttt{r} + 1]$ in the next round and consequently the next reference state $\langle N_{C_i}[\texttt{r}{+}1], N_T[\texttt{r}{+}1] \rangle$, as a function of the current state, design (γ, θ, etc.) and network parameters (ΔR, R_i, C, etc.).

Our analyses study the evolution of this function, assuming non-time varying link capacity and flow count. When these change, we restart the evolution from a new initial state, similar to CCmatic (Chapter 4). We show convergence for all initial states.

Assumptions. Since we model FRCC's execution at the granularity of rounds, we make several simplifying assumptions about the slot-level execution. For instance, we assume that rounds and slots of flows align (they have the same durations and start/end times). We assume that flows update `cwnd`s synchronously (i.e., all flows first make estimates, then update their `cwnd`), in reality flows may update `cwnd` one-after-another (sequentially) (we relax this assumption in § 7.5.4). We detail all these assumptions and more in § D.3.2.

Apart from describing FRCC's dynamics and performance, the primary purpose of our analysis is to demonstrate that FRCC correctly handles jitter—a worst-case (or adversarial) element controlled by the network. Since, aspects like the randomness we use to decide probing slots which affects probe collisions are not in control of the adversary, we believe it is okay to rely on numerical and empirical methods for validating these components like any other CCA evaluation.

These assumptions make our analysis tractable. Empirically, FRCC works even though these assumptions do not hold (§ 7.6).

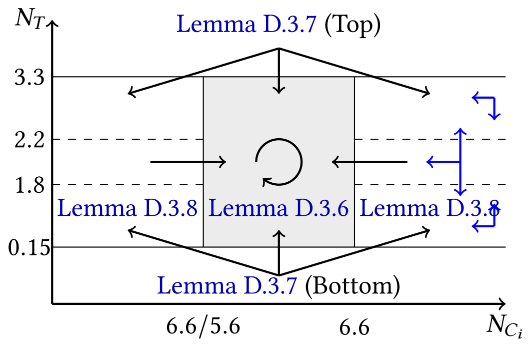


Figure 7.13: Evolution of $\langle N_{C_i}, N_T \rangle$ over rounds on jittery link.

7.5.2 Ideal link (Theorem 7.5.1)

THEOREM 7.5.1. *On an ideal link (Fig. 7.11) with equal RTprops, assuming no probe collisions occur, under assumptions in § D.3.2, FRCC ensures fairness, i.e., every flow's steady-state sending rate is C/N_G, and RTT is $R + \theta N_G$.*

Proof sketch. The proof (§ D.3) mathematically describes for an arbitrary number of flows what we see in Fig. 7.12 (a) for the behavior of two flows. N_{C_i} and N_T move towards N_G, and in steady-state, $\forall i.\, N_{C_i} = N_T = N_G$. This implies (1) 100% utilization (since `delay` $= \theta N_T = \theta N_G > 0$) and (2) fairness (equal N_{C_i} imply equal throughputs, since $N_{C_i} = C/\texttt{rtput}_\texttt{i}$ (definition of N_C (§ 7.5.1)).

Our proof shows that with equal RTprops, there are no (E2) errors and the ideal link is equivalent to the fluid model (Lemma D.3.1). Thus, our estimates are error-free ($\widehat{N_{C_i}} = N_{C_i}, \widehat{N_{T_i}} = N_{T_i}$). We substitute these into the `cwnd` update to get the next state, and state transition function (Eq. D.3.6, Eq. D.3.7).

We show that under the transition function, the gap between N_T and N_G, i.e., $|N_T[\mathbf{r}] - N_G|$ decreases multiplicatively (or exponentially fast) each round. Likewise, the gap between $N_{C_i}[\mathbf{r}]$ and N_G, decreases multiplicatively.

For the ideal link, our proof works for all design parameter choices as long as they are non-zero and positive. Since there are no errors in estimates, we do not need clamps on the `cwnd` (i.e., $\delta_H = \infty$, $\delta_L = \infty$). Our probes just need to create non-zero delay ($\gamma > 0$, $\mathcal{D} > 0$), and we need to create non-zero seconds of queueing per flow ($\theta > 0$).

7.5.3 Jittery link (Theorem 7.5.2)

THEOREM 7.5.2. *On a jittery link (Fig. 7.11), with $N_G = 2$ flows, both having equal RTprops, assuming no probe collisions, and under assumptions in § D.3.2, FRCC bounds unfairness (avoids starvation). Specifically, it ensures that the ratio of steady-state flow throughputs (in the corresponding fluid reference execution), i.e.,* `rtput`*, is at most* $6.6\times$.

Proof. The evolution of FRCC on jittery link is similar to ideal link. The key difference is the state transition function gives a set of possible next states instead of a unique next state and

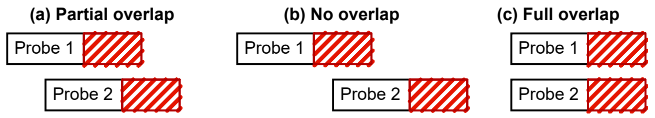


Figure 7.14: Probe overlap scenarios. Rectangles represent probe slots during which `cwnd` is high. The red hatched portions show intervals where flows measure excess delay. (a) Flow 1 measures combined delay while flow 2 does not. (b) Neither flow measures combined delay. (c) Both flows measure combined delay.

consequently FRCC converges to region instead of a fixed-point. The lemmas (Lemma D.3.6, Lemma D.3.7, Lemma D.3.8, proved in § D.3.4), summarized in Fig. 7.13, show that for all initial states, all trajectories (despite jitter) converge to the shaded region, and then stay in that region. As a result in steady-state, $N_{C_i} \in [6.6, 6.6/5.6]$. This implies that the reference throughputs ($\texttt{rtput}_\texttt{i}$) are at most off by 6.6× (as $N_{C_i} = C/\texttt{rtput}_\texttt{i}$).

Since jitter can emulate different "effective" RTprops , the real throughputs can be slightly off this guarantee: if jitter doubles the RTT for one flow, but not the other, its throughput halves, and the throughput ratio could go from 6.6× to 13.2×.

Note, we were only tractably able to write and check the proof of Theorem 7.5.2 for $N_G = 2$ flows since the number of state variables, and hence the solving time for Z3, grows with N_G.

Parameters. There is a one-to-one relation between the design/network parameters and the performance bounds (e.g., 6.6× in the theorem). Given design/network parameters, we find the performance bounds using binary search (§ D.3.4). We iterated this a few times to find a reasonable combination of parameters and performance bounds. We set the design parameters as: $\gamma = 4$, $\alpha = 1$, $\delta_H = 1.3$, $\delta_L = 1.25$. In terms of the amount of jitter, our design works as long as: delay is larger than jitter ($\theta > \Delta R/2$), delay variation is larger than jitter ($\mathcal{D} > \Delta R$), and RTT is large enough to bound throughput variations due to RTT variations ($\gamma_R < 2$).

Additionally, we also need $\theta * N_G > R$ for stability, as we use the alternate `cwnd` update (§ 7.4.2), i.e., less stable update, for jittery link proof. This condition is also needed for ideal link if we use the alternate update.

7.5.4 Impact of probe collisions (overlaps)

Types of probe collisions. When the probes of two flows collide, either one, neither, or both of them may underestimate capacity, depending on exactly how their probes overlap (Fig. 7.14). A probe underestimates capacity when it measures the combined excess delay from multiple probes. Since we take the minimum of RTT samples in the probe's measurement interval (T_P), we measure the combined delay only if *the entirety* of T_P overlaps with another flow's probe slot.

Transition function(s). We update the ideal link transition function to include the impact of probe collisions by updating $\widehat{\Delta d}$. Specifically, if two flows have probe sizes E_1 and E_2, the total excess delay is $(E_1 + E_2)/C$. The flows could see either $\widehat{\Delta d_i} = E_i/C$, or the combined delay. We substitute both these possibilities for both the flows, to get different transition functions.

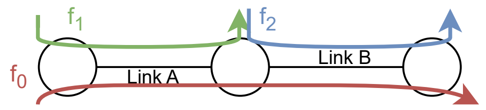


Figure 7.15: Parking lot topology. Flow f_0 gets observations from both hops, f_1, f_2 only see a single hop.

State trajectories. Fig. 7.12 shows the state trajectories of the system under different alignment of probe slots. We get 5 different functions for the possible probe slot alignments. The first two cases are when the probes do not collide. Two flows may measure the same state and update their `cwnd`s together (synchronous), or one flow may first update the `cwnd`, causing the second flow's round RTT and throughput estimates ($\widehat{\texttt{rRTT}}$ and $\widehat{\texttt{rtput}}$) to change before its `cwnd` update (sequential).[2] In both cases, the system state converges to the fair and efficient fixed-point ($N_T = N_{C_i} = N_G = 2$), shown by the green circle.

With probe collisions, both flows may measure the combined excess delay (full overlap), or only one of them may measure the combined excess delay (partial overlap). In full overlap, both flows update their `cwnd` by the same ratio, and this does not change their link fractions (no motion along the N_{C_i} axis). With partial overlap, the flow that underestimates capacity yields bandwidth to the other flow. Fig. 7.12 (d) shows the perspective of the flow that yields bandwidth and (e) shows the flow that receives bandwidth.

Note, for two flows, the state of one flow is enough to represent the state of the system because $N_{C_2} = \frac{N_{C_1}}{N_{C_1}-1}$. (d) and (e) are images of each other under this transformation. This is because link fractions sum to 1, $\sum_{\texttt{i}} 1/N_{C_i} = 1$. Since the fixed point in (e) is $\langle 1, 1\rangle$, the fixed point in (d) is $\langle \infty, 1\rangle$.

FRCC is fair despite collisions. The phase portraits show system evolution if the same transition function is taken in every round. In reality, the system state will evolve under a superposition of these transition functions each taken with some probability. (a), (b) move the system towards fairness and (c) does not change fairness (no change to link fractions). (d) and (e) do cause one flow to yield bandwidth to another, but the system is equally likely to take any of these transitions. This is because, both flows have equal slot and measurement durations, and we resample probing slot in each round. Hence they are equally likely to measure combined excess delay. This explains why FRCC is fair despite probe collisions as the effects of (d, e) cancel out and (a, b, c) do not cause unfairness.

Note, collisions do cause N_T to decrease creating instability and unfairness at short time scales (§ D.5).

7.5.5 Fluid model analysis (different RTprops and multiple bottlenecks)

With different RTprops or multiple bottlenecks, the excess delay (Δd) is no longer equal to the transmission delay of the probe, i.e., $\Delta d \neq E/C$. This creates a bias in our capacity estimate, and

[2] If the first flow increases its `cwnd`, the updates remain synchronous because the second flow's $\widehat{\texttt{rtput}}$ and $\widehat{\texttt{rRTT}}$ do not change. The increased `cwnd` increases RTT and decreases throughput of the second flow, these measurements get filtered out by the max/min filters.

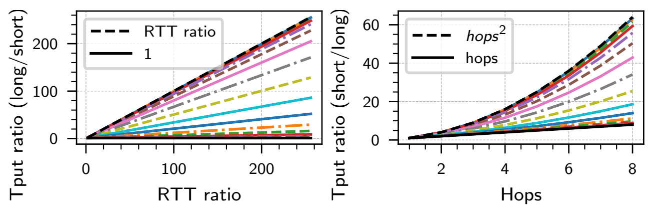


Figure 7.16: Throughput ratios at the fixed point with different RTprops (**Left**), and parking lot topology (**Right**). The different lines correspond to different choices of probe size ($\gamma\mathcal{D}$) relative to the RTprop of the short flow (with $\gamma\mathcal{D}/R$ ranging from 2^{-7} to 2^{8}). Large probe size yields better (lower) throughput ratios.

consequently rate allocation. Instead of equal throughputs, FRCC's throughput ratio may be as bad as the RTT ratio (not RTprop ratio). With parking lot topology, instead of proportional fairness [93] (i.e., a throughput ratio of `hops`), FRCC's throughput ratio may be as large as `hops`^{2}.

Different RTprops. When the short RTT flow probes, the excess delay is more than expected. Conversely, when the long flow probes, the excess delay is less than expected. We give intuition for the short flow probe. During the probe, short flow's rate increases and long's decreases by the same amount. However, since the long flow has larger RTprop, the packets in its pipe reduce by a larger amount. These go into the queue, increasing the delay beyond the probe's transmission delay.

We compute the exact value of the biased excess delay using the fluid model (§ D.4). This yields bias in $\widehat{C}$, and consequently bias in $\widehat{N_C}$. We include these biases in the transition function and solve for the fixed-point (i.e., next state = current state). We use numerical methods to approximate the fixed-point as we do not get closed-form expressions for the transition function in this analysis (§ D.4). Fig. 7.16 shows the throughput ratios at the fixed-point. The ratio depends on the size of our probes (i.e., $\gamma\mathcal{D}$) relative to the short flow's RTprop. With large probes, the throughput ratio approaches 1, while with small probes, it approaches the RTT ratio between flows.

Multi-bottleneck parking lot topology. Fig. 7.15 illustrates the parking lot topology where different flows witness congestion signals from different number of hops. As a result, if f_1 estimates the queuing delay to be `delay`, then f_0 witnesses a delay of `hops` · `delay`. Since CCAs coordinate fairness through congestion signals (Chapter 5 and Chapter 6), this creates bias in *all* end-to-end CCAs, shifting the rate allocation away from max-min fairness (equal rates for all flows) [93, 112]. For instance, Copa/Vegas/FAST achieve proportional fairness [93, 115], where flows using more bottleneck resources receive proportionally less bandwidth. While FRCC would achieve proportional fairness with just the contract bias, the bias in FRCC's capacity estimate results in throughput ratios larger than proportional fairness.

Similar to the different RTprop case, we compute this bias and FRCC's fixed-point (§ D.4). Fig. 7.16 shows the resulting throughput ratios, with FRCC's throughput ratio falling between `hops` and `hops`^{2} depending on the probe size.

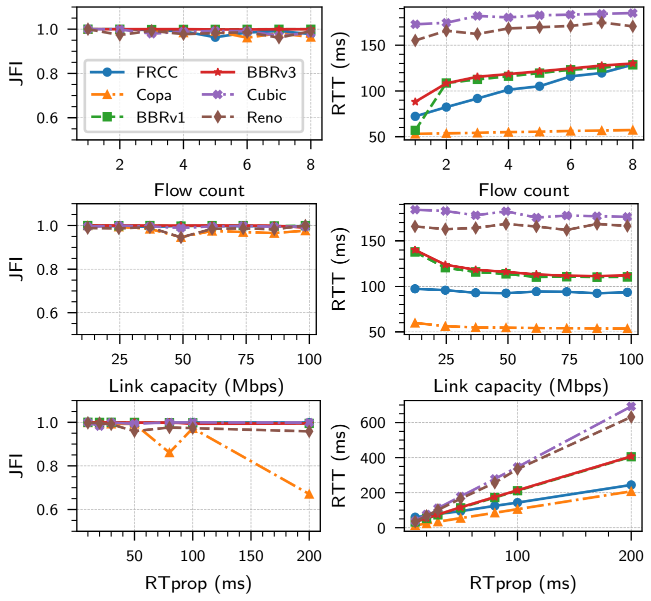


Figure 7.17: Parameter sweeps with ideal link. We omit RTT for Cubic, it bloats buffers building very high delays and skews the plot.

7.6 Empirical evaluation

We evaluate FRCC with four goals in mind: (1) illustrate the performance properties of FRCC, while (2) validating our proofs, (3) measure performance relative to existing CCAs, and (4) explore when FRCC breaks.

Implementation & Methodology. We implement FRCC in the Linux kernel as a pluggable kernel module. We set FRCC's design parameters as guided by our jittery link proof (§ 7.5.3), we set: $\gamma = 4$, $\alpha = 1$, $\delta_H = 1.3$, $\delta_L = 1.25$, $\mathcal{D} = \theta = 10$ ms, $T_R = 30$ seconds, and $T_P = 1$RTT. Our whole goal was to scale to arbitrarily large link capacities. Our proof shows that these parameter choices work for all link capacities and RTprops, as long as the buffers are large enough (§ 7.1, § 7.5.3). Note, even though we set $\mathcal{D} = \theta = 10$ ms, this is for worst-case jitter, our design handles much more than 10 ms of jitter in experiments.

We compare against Cubic [59], Reno [63], BBRv1 [30] (Linux kernel v5.15.0), BBRv3 [34], and Copa [17, 18]. We use mahimahi [94] and mininet [39] to emulate and test a variety of network scenarios. We perform various network parameter sweeps. By default, we use a dumbbell topology with $C = 48$ Mbps, $R = 50$ ms, $N_G = 3$ flows, and buffer size $= \infty$ (we set buffer $= 3$`BDP` for Reno/Cubic, otherwise they set `cwnd` $= \infty$).

We use `tcpdump` at the senders to measure the throughput and RTT. We study the link utilization, flow throughput, fairness (Jain's fairness index (JFI) [65], and throughput ratio $\left(\max_{i,j} \frac{\texttt{throughput}_i}{\texttt{throughput}_j}\right)$), RTTs (latency), and convergence time. We run experiments for 5 mins, and measure these metrics in steady-state (i.e., minute 3 onward).

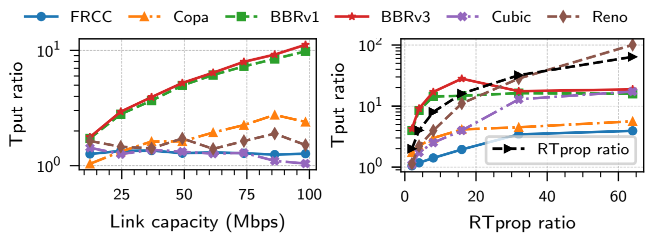


Figure 7.18: Jitter (different RTprops). [**Left**] Three flows with RTprop of 10, 20, and 30 ms, with varying link capacity. [**Right**] Two flows with varying RTprop ratio. FRCC's throughput ratio is between 1 and RTprop ratio even when RTprops are off by 60×.

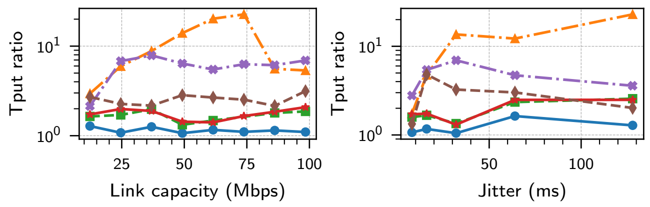


Figure 7.19: Jitter (ACK aggregation). Three flows with $R = 32$ ms. Flow f_0's path has 32 ms of ACK aggregation. [**Left**] Varying link capacity. [**Right**] Varying amount of ACK aggregation on flow f_0's path.

Parameter sweeps without jitter. We sweep flow count (1 to 8 flows), link capacity (12 to 96 Mbps), and RTprop (10 to 200 ms). This is a wide sweep that is at the limits of what mahimahi emulation can support. Fig. 7.17 shows that across all sweeps—flow count, capacity, and RTprop—all CCAs achieve fairness (JFI ≈ 1), except Copa, which struggles at high RTprops. All flows maintain non-zero queueing in steady-state yielding a link utilization of roughly 100% (not shown).

The RTT plots agree with the expected RTTs of the CCAs. With increasing flow count, FRCC's RTT increases linearly by $\theta = 10$ ms per flow (RTT = $R + \theta N_G$). BBR's RTT also increases linearly. It operates in `cwnd`-limited mode (when $N_G > 1$) and maintains $\alpha_{\texttt{BBR}}$ packets (not seconds) per flow ($\texttt{cwnd}_\texttt{i} = 2\widehat{C}_i\widehat{R}_i + \alpha_{\texttt{BBR}}$, RTT = $2R + N_G\alpha_{\texttt{BBR}}/C$ [20]). Copa's RTT also increases linearly maintaining $1/\delta_{\texttt{Copa}}$ packets of queueing per flow (RTT = $R + N_G/(\delta_{\texttt{Copa}}C)$ [18, 20]). Reno/Cubic fill up buffers and have an RTT of $4R$ independent of the flow count (RTT = $R + \texttt{buffer}/C$, $\texttt{buffer} = 3\texttt{BDP}$)

Looking at the capacity sweep, FRCC's RTT remains constant regardless of link capacity. Similarly, Cubic and Reno maintain constant RTT of $4R$. In contrast, BBR's RTT decreases with increasing capacity as the seconds of queueing (transmission delay) of its $\alpha_{\texttt{BBR}}$ packets in queue diminishes. Copa follows the same pattern, with its RTT decreasing as capacity increases. This vanishing queueing delay (or delay variation) with increasing link capacity is precisely what leads to starvation under jitter. FRCC avoids this pitfall by maintaining delay and delay variation that remain larger than network jitter regardless of link capacity.

Finally, the RTT vs RTprop plot is also consistent with the expected RTTs. The slope of RTT vs RTprop is 4 for Reno/Cubic, 2 for BBR, and 1 for FRCC and Copa.

The absolute RTTs (and RTT ranking across CCAs) varies depending on the design parameters, flow count, link capacity (transmission delay), RTprop, and buffer size. Trends in RTTs matter more than the absolute values.

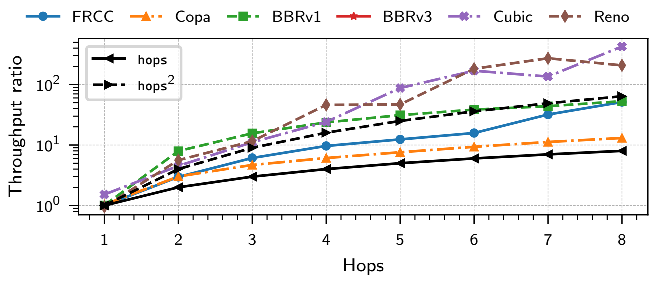


Figure 7.20: Parking lot topology. FRCC's empirical throughput ratio is between `hops` and `hops`^{2}. We do not show BBRv3 as we have separate VMs for BBRv3 and mininet (for emulating parking lot).

Sweeps with jitter. We study two manifestations of jitter: (1) different RTprops and (2) ACK aggregation. These are settings where FRCC shines. Fig. 7.18 [Left] shows flows with slightly different RTprops (10, 20, and 30 ms), FRCC/Cubic/Reno yield close to equal throughputs (ratio $\approx$ 1). BBR causes unfairness which increases with link capacity and ends up starving the shorter RTprop flow (e.g., Fig. 2.1). Copa also causes increasing unfairness with increasing link capacity.

To emulate ACK aggregation, we implement an aggregator parameterized with a burst size in milliseconds and a burst rate. The aggregator collects all ACKs in the burst size period and sends them at the burst rate. We set the burst rate to ensure that the aggregator does not become the bottleneck. We sweep the link capacity in Fig. 7.19 [Left] and burst size in Fig. 7.19 [Right].

Copa and Cubic/Reno cause unfairness which increases with link capacity, eventually starving flows. Copa starves the flow with aggregation while Reno/Cubic starve the other flows (Fig. 2.2). FRCC maintains roughly equal throughputs for flows even when the jitter is 128 ms—10$\times$ what FRCC was tuned to tolerate. BBR exhibits slight unfairness.

Vastly different RTprops and multiple bottlenecks. Fig. 7.18 [Right] shows throughput ratios of two flows with varying RTprop ratio. The throughput ratios for all CCAs increase with RTprop ratio. FRCC's throughput ratio is lower (more fair) than other CCAs even when the RTprops are off by 64$\times$. FRCC's throughput ratio lies between 1 and RTT ratio and is consistent with our analysis (§ 7.5.5). Note, RTprop ratio > RTT ratio.

Fig. 7.20 shows results with the parking lot topology with C = 100 Mbps and RTprop = 10 ms for all flows. FRCC's throughput ratio lies between `hops` and `hops`^{2}, consistent with our analysis (§ 7.5.5). The throughput ratio for Copa is $\approx$ `hops` (proportional fairness), while that for BBR/Cubic is worse than `hops`^{2}.

Convergence time. Fig. 7.21 shows 8 flows competing on a 96 Mbps link with 50 ms RTprop. Each flow enters 90 seconds after the previous one and runs for $8 \times 90 = 720$ seconds. Practically, FRCC's convergence time is slow and takes tens of seconds to converge. FRCC takes $O(\log(\texttt{BDP}))$ rounds to converge. When there are four flows, a slot is 4 RTTs or 360 ms (RTT $= R + \theta N_G = 50 + 4 * 10$), and a round is $kN_G = 8$ slots or 3 seconds. C.f. a round for BBR is $8 * R = 0.4$ seconds.

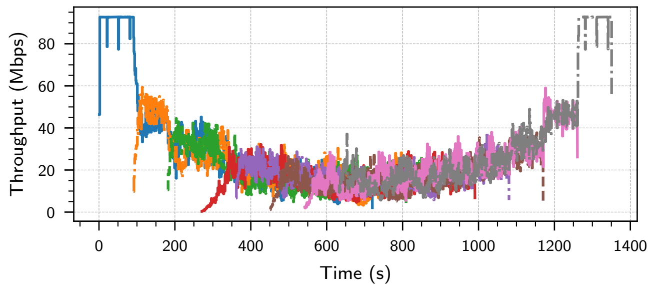


Figure 7.21: FRCC converges to fair rates as flows enter and leave. Each line shows the throughput of a different flow.

7.7 Chapter summary

We presented FRCC, the first CCA that provably achieves fairness despite network jitter, using our key insight of of decoupling fair rate coordination. Our formal analysis demonstrates that FRCC converges exponentially fast to efficient and fair rate allocations with bounded throughput ratios even under worst-case jitter patterns. Our empirical evaluation validates these theoretical guarantees, showing that FRCC maintains fairness in scenarios where existing CCAs starve.

Limitations and future work. While FRCC represents significant progress, three key limitations may need to be addressed for a practical deployment: (L1) operation with shallow buffers, (L2) coexistence with other CCAs, and (L3) convergence time. Of these, we believe the convergence time is the most critical. For instance, even widely deployed CCAs like Reno, Cubic, and BBR struggle with shallow buffers [32] and coexisting with each other [113]. We provide ideas on addressing (L3) here and (L1/L2) in Chapter 8.

While FRCC's asymptotic convergence time of $\log(N_G \mathtt{BDP})$ RTTs is competitive, its practical convergence is longer due to (1) a long, multi-RTT probing slot, and (2) $\mathcal{O}(N_G)$ slots in a round. Further research on capacity estimation could reduce the probing duration. Alternatively, an initial deployment could trade theoretical robustness in favor of convergence time. For instance, the long probing slot helps deal with transient RTT variations with different RTprops and multiple bottlenecks (Figures 7.6, 7.7, and 7.8). A shorter slot duration and fewer slots per round would help improve convergence at the cost of modest unfairness in those specific cases, while still solving the core starvation problem in Figures 2.1 and 2.2.

Takeaways. FRCC may influence the CCA landscape in at least four ways. First, FRCC could be deployed with a pragmatic tuning (as described above) or after further research addresses its limitations. Second, parts of FRCC (e.g., link fraction coordination, link capacity estimation, transient/persistent RTT variations) may inform other CCA designs. Third, further research may conclude that fairness is a much harder problem where end-to-end methods like FRCC come at an unbearable cost, bolstering the case for deployment of in-network solutions or informed over-provisioning of the Internet.

Finally, our design would not have been possible without formal reasoning and worst-case modeling of the network. We began by stating assumptions about the network and desired performance guarantees, then worked backwards to derive a CCA that meets these requirements. This (1) revealed high-level key insights and (2) guided our low-level design decisions. We believe this is a promising methodology in general for developing robust network control algorithms with predictable performance in challenging environments.

Chapter 8

Conclusion

"We have given you the telescopes, it's your turn to discover the planets."

— Anup Agarwal

We envision a future where beliefs and contracts serve not only as tools for designing and analyzing CCAs, but also as a common language for educating and communicating ideas in congestion control. These abstractions vastly simplify the design and analysis of CCAs: they made automated synthesis tractable, revealed a variety of previously unknown tradeoffs, and enabled new CCAs that solve long-standing open problems in congestion control. Using these tools, this thesis "solves" congestion control for a broad range of challenging networks with diverse parameters and various sources of noise. More precisely, we solve congestion control for networks that can be modeled by the CBR-delay and CCAC abstractions in Chapter 2. Future work need not revisit these scenarios outside of the cases we list in § 8.1. Our key results include:

(1) The first class of CCAs that can provably bound loss independent of the capacity of the link on networks with shallow buffers and jitter. Existing CCAs incur significant losses under these networks.
(2) The first CCA that can provably avoid starvation (bound fairness) on networks with multiple competing flows and jitter. Existing CCAs starve one or more flows on such networks.
(3) Fundamental tradeoff between "amount of packet loss" vs "time to converge to the link capacity" on networks with jitter and shallow buffers.
(4) Fundamental tradeoff between "latency" and "generality (e.g., range of link rates)" vs "fairness" and "robustness to jitter".

These are not the only results (planets) that our tools (telescopes) can discover. We believe that by using beliefs and contracts, our community can continue to uncover further results and solve congestion control once and for all. Whenever designing, analyzing, or even reviewing a new CCA, every congestion control practitioner, researcher, or student should ask: "What is the contract for this CCA?" and "What beliefs does this CCA compute?" Answering these questions will deepen understanding of how that CCA really works: what assumptions it makes about the network, what tradeoffs does it make, and which network conditions might it break in.

8.1 What remains to solve congestion control?

We believe that the following two are the main problems that remain to solve end-to-end sender-based Internet congestion control.

Problem 1: Shallow buffers + Jitter + Multiple flows. In the first part of this thesis, we showed how to design algorithms that provably guarantee performance on networks with jitter and shallow buffers. In the second part, we developed FRCC, which guarantees performance on networks with jitter and multiple competing flows. However, it remains unclear how to design CCAs that provide such guarantees when all three conditions—jitter, shallow buffers, and multi-flow competition—occur simultaneously.

Problem 2: Convergence time. Both FRCC and the loss-bounding CCAs synthesized through CCmatic give up convergence time. In the case of FRCC, it remains unclear whether this slow-down is a fundamental limitation or merely an artifact of our design. We conjecture that a deeper fundamental tradeoff involving convergence time may exist. For the loss-bounding CCAs, we formally proved a convergence-loss trade-off, but this result represents a worst-case bound. We believe there is room to improve convergence time in at least the common case for both the CCmatic CCAs and for FRCC.

Promising directions. We outline four promising directions to address these problems. First, to handle shallow buffers, in FRCC, a contract based on packet loss [48, 89] rather than delay could be explored. However, directly adopting Reno's loss-rate encoding is insufficient, as its sawtooth behavior causes under-utilization when buffers are smaller than the BDP. Second, our current contract encodes information in the *magnitude* of delay. Future work could explore encoding the fair rate in the *frequency* or *pattern* of delay variations. A prior proposal for AIMD on delay [20] follows this direction, but we showed in Chapter 6 that this specific design can lead to starvation on multi-bottleneck paths even without jitter. This can help improve the convergence time by removing the need to conduct and coordinate capacity probes in FRCC. Third, in § 7.7, we showed that FRCC can be tuned to give up robustness in the worst-case to improve convergence time in the common case.

Finally, the loss convergence tradeoff in Chapter 4 applies whenever the link capacity increases and the CCA needs to converge to the new capacity. Since the CCA does not know if the capacity could have increased, the CCA needs to continuously re-verify its belief about the capacity. This re-verification imposes the loss-convergence tradeoff in the common case. We can split the the re-verification and converging to new capacity into two phases. In phase 1, we can choose to only slowly increase rate to check if capacity could have increased, so we bound loss in the common case. In phase 2, once we have confirmed that the capacity has indeed increased, we can converge fast and risk large losses. The hope is that the second phase triggers rarely so that we bound losses on average.

Congestion control in data centers. While we believe the above two are the main remaining problems in Internet congestion control, data center environments present a distinct set of problems and opportunities. On the problem side, data center networks are simpler in some respects: links are predominantly wired and therefore experience far less noise, and the range of link capacities to support is bounded and known in advance. However, these environments introduce new challenges. For instance, we desire better performance than Internet congestion controllers on workloads like large-scale *incast* or short/bursty traffic that completes in less than one RTT. We also desire minimal to no packet losses because loss recovery logic adds significant overhead to hardware-offloaded transport stacks predominant in data centers.

At the same time, data center settings offer unique opportunities for advancing congestion control. They enable mechanisms that are infeasible at Internet scale, such as (1) collecting richer information about the network using *in-band network telemetry* (INT), (2) using multi-paths exploiting path diversity, (3) conducting proactive congestion control using receiver-based and credit-based control.

We hope to extend beliefs, contracts, and automated synthesis to systematically explore the design space of data center congestion control. For example, even in their current form, they can guide what telemetry signals should one collect from network switches to guide congestion control decisions (§ 6.6). Similarly, it is hard for humans to manually reason about interactions between multiple decision axes like balancing load between multiple paths and sending rate on each path. Formal methods can aid humans to do this multi-axis reasoning. We describe one such example in § 8.3.4.

8.2 Key learnings

We offer two key lessons and two hot takes for future work on network control.

Lesson 1: Design for ϵ-noise and all network parameters. New control algorithms (1) should be designed to tolerate at least an ϵ amount of noise in their measurements, and (2) should be tested at the boundary points of network parameters—very high and very low link capacities, large and small flow counts, extreme propagation delays, and shallow/deep buffers.

We give this guidance based on what we observed in FRCC. At high bandwidths, even an ϵ-level timing noise—such as minor timestamping differences or operating system jitter—can cause flows to starve. This happens because, to signal information to other flows, most existing CCAs maintain queueing in units of packets, while their delay estimates are measured in seconds. At high link capacities, these mismatched units amplify small timing errors into large rate disparities, breaking fairness. By deliberately designing for ϵ-noise and validating at parameter extremes, such hidden mismatches and corner-case failures can be detected and avoided early in the design process.

Lesson 2: Methodology for controller design. Every decade, there is some change in network environments that leads to an influx of new work on congestion control. We hope that by using the following methodology, our community can streamline the design process for new CCAs. The design of a new controller should begin by explicitly stating a model of the network—specifying the assumptions the controller makes about how the network behaves—and an objective that the controller seeks to optimize. The algorithm should then simply be a function of this model and objective.

Of course, defining the algorithm as a function of the network model and objective alone does not make design tractable. This is precisely where our abstractions of beliefs and contracts come in. One of the central questions in controller design is: what state should the controller maintain, and how should it compute it? Our key argument is that the maintained state should represent uncertainty in the state of the network and that of the other flows. With beliefs, we can compute this uncertainty mechanically—without ad hoc design choices—by inverting the assumptions encoded in the network model and the contract. Together, the network model and contract describes how the global network state gives rise to the controller's observations; beliefs simply invert this mapping to infer what global states are consistent with those observations.

Once uncertainty (partial observability and decentralization) are properly addressed, the rest of control design becomes immensely simpler. To do this, for each belief state, we just need to pick a rate choice that aligns with (1) the contract, (2) the objective and (3) shrinks the belief set if there is too much uncertainty. This reasoning is effectively a 2-player (controller vs network) complete information game that can be solved computationally or manually as we show in [97].

Note, in this methodology, there is an open question of how to pick the contract. This is only relevant when designing controller for multi-flow settings. This choice should happen as the first step after selecting network model and objective, since this is required to compute beliefs. § 6.2 gives guidance on how to pick a contract. However, our current formulation of contracts is too strong, i.e., once we pick a contract, the remaining design decisions are trivial. We hope to relax this in future work (§ 8.3).

Hot Take 1: There is no such thing as model-free controller design. In our community, there has been some debate in model-based vs model-free design. We argue that there is no such thing as model-free design. All controllers need to make some assumptions about the network, and these form the network model.

All CCAs implicitly or explicitly make some or the other assumptions about the network. Even algorithms like PCC [41, 42, 90], which claim to "learn" the network online, make assumptions about the network. For instance, PCC concretely sets "how long each micro-experiment should last". Ideally, the answer to this question depends on the network model, and setting a concrete threshold is effectively an assumption about the network. Similarly, PCC assumes that the results of micro-experiments are repeatable, i.e., if a certain rate experiment improved utility in the experiment, then employing it will also improve utility. However, the network state may change due to exogenous (link capacity or flow count might change) or endogenous reasons (the rate change changes the state of queueing). Assumptions on how these factors change are also part of the network model. Making these assumptions explicit clarifies what environments the algorithm is designed for and prevents misinterpretation of its behavior. There is a question of what are the minimal assumptions any controller must make and then learn the remainder things offline. When we answer this question, then the minimal assumptions would be the network model.

Hot Take 2: Stop blindly using reinforcement learning or computer-generated controllers for multi-flow settings. Our community needs to fundamentally rethink how we use computers to generate controllers in multi-flow settings. In particular, we need to focus on how can we get computers to *explicitly explore the design space of contracts*.

In Chapter 6, we showed that reinforcement learning (RL) based approaches effectively learn a narrow contract, e.g., one that works only under a limited set of link rates. While we demonstrated this concretely for Astraea [84], we believe this limitation extends to all existing computer-generated congestion control algorithms. Most automated approaches—whether learning-based or synthesis-based—share the same structure: they specify a template for the algorithm (e.g., features and neural network architecture), a network model or simulation environment, and an objective to optimize. However, none of these components explicitly constrain or incentivize the computer to select a good contract or to explore the space of possible contracts.

As a result, the algorithms they produce almost inevitably rediscover existing canonical contract-based designs or, worse, produce controllers with narrow contracts. In either case, we do not advance the state of the art. Therefore, if we do not rethink how we encode these inputs—particularly how to represent and guide the discovery of contracts—then there is no value in applying RL or any other automated synthesis technique to multi-flow congestion control.

This take aligns with findings from multi-agent RL in other domains. Specifically, without explicit incentives, RL agents rarely learn to cooperate and need reward functions that deliberately incentivize cooperation [66]. For instance, recent work builds domain agnostic metrics like "social influence" that can drive agents towards cooperation [66]. Such ideas are a promising for addressing the limitations of RL-based congestion control algorithms in multi-flow environments.

8.3 Future work

8.3.1 Extending contracts for (1) inter-CCA fairness/coexistence and (2) automatic synthesis of CCAs for multi-flow settings

Contracts emerged from our efforts to extend CCmatic beyond single-flow scenarios. In the single-flow setting, we could infer the state of the network solely from assumptions about the network encoded in the network model. However, in the multi-flow setting, each flow must also reason about the state of *other* flows. This requires making assumptions about their behavior. The simplest assumption—that all flows follow the same congestion control algorithm—does not directly help as, we are still in the process of designing that very algorithm. Thus, we introduced contracts as a principled way to express the assumptions that each flow makes about the other flows in the network.

Current limitations. In defining contracts, we faced a tradeoff between expressivity (breadth of statements we can make) and tractability (mathematically backing the statements). We erred on the side of tractability to mathematically derive tradeoffs in § C.1. Correspondingly, our current formalism is a bit too strong.

For instance, once we fix a contract, there is little room left in completing the CCA design. In § 6.4, we showed how one should implement CCA dynamics to follow a contract. Ideally, a weaker formalism of contracts could enable automated tools to take more charge of the CCA design process. This could allow tools like CCmatic to jointly explore designing a contract and a CCA, possibly enabling progress on the open-problem of designing CCAs that can simultaneously handle jitter, shallow buffers, and multiple flows § 8.1.

Similarly, in our current definition, we compute the contract of a CCA using scenarios where the CCA is competing with itself. As a result, we are unable to reason about inter-CCA fairness. For the same reason, we could not prove/disprove that contracts are necessary or sufficient for fairness (i.e., two CCAs can reach fairness if and only if they have the same contract). Resolving these issues could further guide CCA design. For instance, to meet TCP-friendliness, BBRv3 [32] leaves "headroom" for loss-based CCAs. If contracts are necessary for fairness, then it is better to explicitly follow Reno's contract on detecting competing Reno flows rather than relying on "leaving headroom" which may or may not cause BBRv3 to follow Reno's contract.

Challenges and potential approaches. Addressing these limitations is non-trivial. For instance, two CCAs may achieve fairness even when they have different contracts. Copa employs a delay-based contract when competing with itself but switches to emulating Reno when it detects Reno flows. Even CCAs like Vegas—which employ a delay-based contract all the time—may compete fairly with Reno depending on the network conditions [85]. For instance, RED-based packet drops [47] create a mapping between queuing delay and loss rates. If the loss-rate-based and delay-based fair shares match, then Vegas and Reno may compete fairly.

Conversely, two CCAs may be unfair to each other even when they share the same contract. For instance, TFRC [48] and Reno [63] have the same contract. However, we can construct an adversarial variant of TFRC that is fair to TFRC flows but unfair to Reno flows. This adversarial CCA can detect when it is competing with Reno flows by detecting their sawtooth pattern, and then deliberately blast packets to consume more than fair share.

These examples highlight that representing contracts using simple mathematical functions may not capture all the nuances of multi-flow coordination. A promising direction is to borrow from the formal methods literature, where contract-like abstractions have been used to reason about distributed algorithms. For instance, [14] models contracts as a *set* of traces described using $\omega-$regular grammars, while we defined contracts as a *function*.

Another promising approach is to leverage domain-agnostic notions of coordination. For instance, recent work [66] in multi-agent reinforcement learning uses "social-influence" as a metric to incentivize cooperation between agents. Such metrics may offer a way to parametrize and computationally explore the space of coordination mechanisms between CCAs.

8.3.2 Benefits to and from network measurement

Measurements of workloads and environments can guide the use of beliefs and contracts. Also, beliefs and contracts can help improve our measurement tools.

Improvements from measurements. We uncovered a variety of new tradeoffs. Measurement of workloads and network environments can guide which metrics to prioritize in the tradeoff space. For instance, assessing fairness requires understanding how often flows experience multi-hop congestion and the typical number of flows or hops involved. Recent work [28] suggests that contention may be rare on the Internet, implying fairness may be less critical in some contexts. For robustness, it would be useful to quantify the size and frequency of non-congestive delays and losses. On the workload front, we want to understand which network-level metrics correlate with application-level metrics. For instance, [101] shows that unfairness is better for AI collectives.

Improvements to measurement. Beyond designing controllers, beliefs and contracts can also help improve our community's measurement tools. Beliefs can help understand (1) what is the distribution of parameters like link capacity, buffer sizes, propagation delay, etc. on the Internet, and (2) what mathematical network model(s) best reflect real network paths. Recall, beliefs can be computed *independent* of the CCA (Chapter 3). We only need the timeseries of sending and ACK sequence numbers to compute beliefs. Such traces can be passively collected at scale for a variety of vantage points, deployments, and CCAs. The beliefs for these traces directly tell us the parameters of the network under different network models. Similarly, to learn what model(s) does the Internet conform to, we can start by building a collection of network models (e.g., ideal link, CBR-delay model, CCAC model from Chapter 2 and more). Then, we can check which of these models is satisfied by the collected traces. Multiple models may be satisfied by the traces, and we can pick the model with the fewest behaviors that still captures reality as a target network model to design controllers for. This allows us to make the strongest (correct) assumptions about the network.

Likewise, contracts can improve reverse-engineering and classification of CCAs in the wild [45, 53, 91, 114]. Like past work, we can setup a controlled bottleneck and measure the steady-state behavior of congestion in the bottleneck for a variety of bottleneck link rates. This data gives us the contract for the CCA (similar to our empirical procedure in Chapter 5). The contract can then help understand which bucket of steady-state performance tradeoffs it falls into. Note, knowing the contract of the CCA is almost as good as knowing the full CCA itself, because, the contract is what determines the steady-state performance. One can perform additional measurements to also determine the CCA dynamics to fully characterize the CCA.

8.3.3 Formal methods to reason about system performance

While this thesis demonstrated how automated reasoning tools can aid the *design* of congestion control algorithms, we believe they can play an equally crucial role in the *analysis* of controllers. Further, automated reasoning can also help reason about system performance in other contexts beyond congestion control, such as resource allocation and scheduling. Below, we outline two directions along these lines.

Beyond design: automated analysis. Automated reasoning can go beyond synthesis to formally analyze existing controllers. In this thesis, we formulated the design problem as an *exists-forall (∃∀)* query: "Does there exist a congestion control algorithm (in a given search space) such that, for all network traces permitted by a given model, a specified performance property holds?" Similar reasoning frameworks can be used to answer complementary analytical questions. For example, we can ask: "Does there exist a network model or set of assumptions such that, for all traces under this model, a given controller satisfies the desired performance property?"—thereby uncovering the implicit assumptions under which the controller operates. Likewise, we can query: "Does there exist a workload or environment under which algorithm A outperforms algorithm B?". Such queries can allow operators to decide what policies to use in different deployments.

Formal models of environments. Any systematic approach to designing control policies or heuristics must begin with a formal model of the environment in which the policy operates. These models, along with automated reasoning tools allow one to prove/disprove theorems about system performance under the model. To ease application to other domains, we list some principled ways to construct such models. First, we can build models that fit empirically collected data. Second, we can construct analytical models of individual system components and then derive composite models that describe how these components interact. Finally, a third and perhaps new way to build models is by *inferring* them from existing control algorithms: we can identify the implicit assumptions that different controllers make about their environments, and take a "majority vote" across these assumptions to recover a minimal set of assumptions that any reasonable policy must make. This is because a model of the environment is effectively equivalent to the set of assumptions made about it.

8.3.4 Extending to other domains

While this thesis demonstrated the benefits of beliefs, contracts, and program synthesis in the context of congestion control, we believe that these tools are far more general.

For example, in follow up work (§ 4.5 in Chapter 4), we considered designing cross-layer controllers for low-latency interactive and live video streaming. Here, the controller must reason not only about the sending rate but also about multiple other decision axes—such as the video bitrate, whether to skip a frame, and how much forward error correction (FEC) to apply. Manually reasoning about all these interacting dimensions is difficult, which is why most prior work optimizes only one or two of them in isolation.

Here, directly using CCmatic does not scale to reason about these multiple axes of decisions. To address this, we built a new tool Syntra [97]. The key idea is to replace full program synthesis with *decision synthesis*: rather than generating the entire controller program symbolically, Syntra computes the optimal action for each belief set (or state) by formulating control synthesis as a two player (controller vs network) game. It then collects these belief-action pairs, and applies imitation learning to derive a compact, interpretable policy that reflects these optimal decisions.

We showed that Syntra's controllers outperform state-of-the-art baselines on video quality, latency, and frame rate both because (1) we can reason about noise and diversity in networks, and because (2) we can reason about four dimensions of decision making. We get benefits on both common case networks and challenging networks.

Beyond congestion control and video streaming, we believe our methodology can apply to more different domains like scheduling and load balancing. For instance, recent work [16, 52] used worst-case analysis to identify workloads where existing production load balancers and schedulers break. This worst-case analysis uses the same non-deterministic environment modelling philosophy that we use and thus, we believe our program synthesis approach can generalize to this setting and design policies that address these workloads.

Appendix A

Belief framework proofs and details

A.1 Beliefs are sufficient

Lemma A.1.1. *The set of feasible actions that the network is allowed to take in the future can be determined by the belief set.*

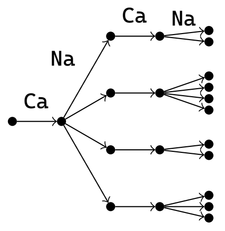
Proof. Recall, the only rule in our game is that the sequence of network's actions should be allowed by some path in the network model. As a result, a network action is feasible if and only if the sequence of past actions combined with the network action is allowed by some path in the network model. We will show that the network can compute the set of all feasible actions using the belief set.

We argue that the set of feasible actions is the set of actions that are allowed by some ⟨path, state⟩ tuple in the belief set. First, if an action $\mathcal{A}$ is allowed on path $\mathcal{P}$, and state $\mathcal{S}$ from the belief set. Then all the past network actions are allowed by path $\mathcal{P}$ (as $\mathcal{P}$ is in the belief set, and a path is in the belief set if all past actions can be explained by it). So the sequence of actions, including past and $\mathcal{A}$, is consistent with the path $\mathcal{P}$, hence $\mathcal{A}$ is a feasible action. Second, if an action $\mathcal{A}$ is not allowed on any ⟨path, state⟩ tuple in the belief set, then it cannot be a feasible action as the sequence of past actions and $\mathcal{A}$ cannot be explained by any single path in the network model.

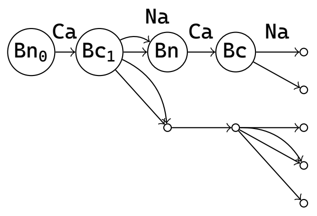
□

Lemma A.1.2. *Belief set for a future time can be computed by the belief set at the current time and the CCA's observations between the current and future times.*

Proof. We compute the future belief set as follows. Enumerate all tuples in the current belief set. Filter out the tuples that cannot produce the trace of CCA's observations between the current and future times. Filtering can be done by replaying the CCA's sending rate choices on the tuple, and checking if the tuple has feasible network actions that produce the exact observations of the CCA. The set of remaining tuples is same as the future belief set (as if it were computed using the entire history of CCA's observations). This is because a tuple is in the remainder set iff it can produce both (1) all CCA's observations till now (as it is in current belief set) and from now to the future time (as it was not filtered out). □



(a) Game tree $\mathcal{T}$.



(b) Condensed graph $\mathcal{G}$.

Figure A.1: Constructing condensed graph from game tree.

THEOREM. *If there exists a deterministic CCA that ensures a given performance property, on a given network model, then, there exists a* belief-based *CCA that ensures the performance property on the network model. Where, the beliefs are derived from the given network model, and the performance property is defined as a boolean valued function of the belief set and the action taken by the CCA on that belief set.*

Proof. We construct a belief-based CCA ($\mathcal{A}_{\mathcal{B}}$) by inspecting the executions of the given deterministic CCA ($\mathcal{A}_{\mathcal{D}}$), and show that $\mathcal{A}_{\mathcal{B}}$ ensures the given performance property ($\mathcal{P}_f$).

Preliminaries. An execution (or trace) is a sequence of CCA actions ($\texttt{Ca}_\texttt{i}$) and network actions ($\texttt{Na}_\texttt{i}$), e.g., $\langle \texttt{Ca}_1, \texttt{Na}_1, \texttt{Ca}_2, \texttt{Na}_2, \dots \rangle$. We say that an execution is valid if it conforms to the rules of the game, i.e., the sequence of network actions correspond to some path in the network model.

We annotate each step of the execution by the beliefs computed over the entire history until that step, e.g., the belief set is $\texttt{Bc}_\texttt{i}$ after $\texttt{Ca}_\texttt{i}$, and $\texttt{Bn}_\texttt{i}$ after $\texttt{Na}_\texttt{i}$. At the start of all traces, the belief set is $\texttt{Bn}_0$. The annotation looks like: $\langle |_{\texttt{Bn}_0} \texttt{Ca}_1 |_{\texttt{Bc}_1} \texttt{Na}_1 |_{\texttt{Bn}_1} \texttt{Ca}_2 |_{\texttt{Bc}_2} \texttt{Na}_2 |_{\texttt{Bn}_2} \dots \rangle$.

The performance property is defined as a function $\mathcal{P}_f$ that maps $\langle$Belief, CCA action$\rangle$ (e.g., $\langle \texttt{Bn}, \texttt{Ca} \rangle$) to `True` or `False`. While in §4.1.2 and §4.2.3, we do not directly state the performance properties this way, we show that they can be defined this way at the end of the proof.

Construction. The high level summary of the construction is that $\mathcal{A}_{\mathcal{B}}$ can arbitrarily pick one CCA action (sending rate choice) whenever $\mathcal{A}_{\mathcal{D}}$ takes different actions on the same belief set. As a result, $\mathcal{A}_{\mathcal{B}}$ is a pure function of belief set (it does not take different actions on the same belief). First, we will show that $\mathcal{A}_{\mathcal{B}}$ function is well-defined i.e., it assigns a rate choice to each belief `Bn` that it can witness in an execution. We will show this is because $\mathcal{A}_{\mathcal{B}}$ only ever reaches a belief set `Bn` if $\mathcal{A}_{\mathcal{D}}$ reached it. We will use Lemma A.1.2 and Lemma A.1.1 to show this. Second, we will show that $\mathcal{A}_{\mathcal{B}}$ satisfies the performance property because it only takes an action `Ca` on `Bn` if $\mathcal{A}_{\mathcal{D}}$ took it and so $\mathcal{P}_f(\langle \texttt{Bn}, \texttt{Ca} \rangle)$ evaluates to true for $\mathcal{A}_{\mathcal{B}}$ as it evaluated true for $\mathcal{A}_{\mathcal{D}}$.

We consider the set of all executions of $\mathcal{A}_{\mathcal{D}}$ on the given network model. We visualize them in the form of a (directed) game tree $\mathcal{T}$ (Fig. A.1a). The nodes in $\mathcal{T}$ are placeholders. A unique node includes the entire history of the CCA and network actions until the node. If it is the turn of the CCA to play, the node has an outgoing edge labelled with the CCA action `Ca`. If it is the network's turn to play, the node has outgoing edges corresponding to all feasible network actions, each labelled with a network action $\mathtt{Na}$. We label each node in the tree with the belief set computed over the history of actions until the node. We get two kinds of labels, (1) beliefs after CCA action ($\mathtt{Bc}$), and (2) beliefs after network action ($\mathtt{Bn}$).

From the tree, we construct a condensed graph $\mathcal{G}$ (Fig. A.1b) as follows. Within each label type, we merge the nodes that have the same belief value (i.e., if two nodes have labels $\mathtt{Bn}^{I}$, $\mathtt{Bn}^{II}$, with $\mathtt{Bn}^{I} = \mathtt{Bn}^{II}$, we coalesce the nodes). Likewise for $\mathtt{Bc}^{I} = \mathtt{Bc}^{II}$. Note, we do not merge nodes with different label types even if they have the same belief value. Then for each node with a $\mathtt{Bn}$ label type (i.e., CCA's turn to play), we just keep one outgoing edge (arbitrarily) resembling the CCA action that $\mathcal{A}_{\mathcal{B}}$ takes on that belief value.

Note, in our construction of $\mathcal{G}$, we never remove edges corresponding to network actions, and all edges (both CCA and network actions) have the same destination node label in $\mathcal{T}$ and $\mathcal{G}$.

$\mathcal{A}_{\mathcal{B}}$ is the desired belief-based CCA. With the above construction, we will argue that (1) $\mathcal{A}_{\mathcal{B}}$ is well-defined, i.e., it assigns an action (outgoing edge) to each belief set that it can reach (ever witness in any execution), and (2) all actions of $\mathcal{A}_{\mathcal{B}}$ satisfy the performance property.

$\mathcal{A}_{\mathcal{B}}$ is well-defined. $\mathcal{A}_{\mathcal{B}}$ assigns a rate choice (CCA action) to each belief set that it can witness under the network model. We prove by contradiction. Say there is a valid execution $\mathtt{e}$ produced by $\mathcal{A}_{\mathcal{B}}$ that reaches a belief set $\mathtt{Bn_i}$ for which $\mathcal{A}_{\mathcal{B}}$ never assigns an action, and $\mathtt{Bn_i}$ is the first such belief in the execution. $\mathcal{A}_{\mathcal{B}}$ does not define an action for $\mathtt{Bn_i}$ in two cases, (1) $\mathtt{Bn_i}$ is not in $\mathcal{G}$, and (2) $\mathtt{Bn_i}$ is in $\mathcal{G}$ but does not have an outgoing edge.

Case 1. We know $\mathtt{Bn_i} \neq \mathtt{Bn_0}$ because $\mathtt{Bn_0}$ is in $\mathcal{G}$ (it is the root node of $\mathcal{T}$, and we do not remove any nodes when constructing $\mathcal{G}$). As a result, $\mathtt{e}$ looks like $\langle \cdots |_{\mathtt{Bn_{i-1}}} \mathtt{Ca_i} |_{\mathtt{Bc_i}} \mathtt{Na_i} |_{\mathtt{Bn_i}} \rangle$, where, $\mathtt{Bn_{i-1}}$ is in $\mathcal{G}$.

Since $\mathcal{A}_{\mathcal{B}}$ produced $\mathtt{e}$, $\mathcal{A}_{\mathcal{B}}$ took action $\mathtt{Ca_i}$ on the belief $\mathtt{Bn_{i-1}}$, i.e., $\mathtt{Ca_i}$ is the outgoing edge for node $\mathtt{Bn_{i-1}}$ in $\mathcal{G}$. $\mathcal{A}_{\mathcal{B}}$ only takes this action, if $\mathcal{A}_{\mathcal{D}}$ took this action on a node with label $\mathtt{Bn_{i-1}}$. We argue that in $\mathcal{T}$, this action leads to a node with label $\mathtt{Bc_i}$. This comes from Lemma A.1.2, $\mathcal{A}_{\mathcal{D}}$ could have arrived from any history at the belief $\mathtt{Bn_{i-1}}$, but taking the action $\mathtt{Ca_i}$ on $\mathtt{Bn_{i-1}}$ updates the belief to $\mathtt{Bc_i}$ no matter the history. Thus, from our construction, $\mathtt{Bc_i}$ also exists in $\mathcal{G}$ and is the destination node on the edge $\mathtt{Ca_i}$ (as we do not remove nodes in the construction and the destination node labels are same in $\mathcal{G}$ and $\mathcal{T}$ for each edge).

Now, we argue that in $\mathcal{T}$, $\mathtt{Na_i}$ is an outgoing edge for the node with label $\mathtt{Bc_i}$. This is because according to $\mathtt{e}$, $\mathtt{Na_i}$ is a feasible network action on the belief $\mathtt{Bc_i}$, as a result $\mathtt{Na_i}$ is a feasible action on the node with label $\mathtt{Bc_i}$ independent of history in $\mathcal{T}$ that led to this node (from Lemma A.1.1). Since the game tree describes *all* executions of $\mathcal{A}_{\mathcal{D}}$, each node with label type $\mathtt{Bc}$ has an edge for each feasible network action, so $\mathcal{T}$ has an edge with action $\mathtt{Na_i}$ on the node with label $\mathtt{Bc_i}$. Again from Lemma A.1.2, $\mathtt{Na_i}$ takes belief from $\mathtt{Bc_i}$ to $\mathtt{Bn_i}$ in $\mathcal{T}$.

Since $\mathtt{Bc_i}$, $\mathtt{Na_i}$ and $\mathtt{Bn_i}$ exist in $\mathcal{T}$, they also exist in $\mathcal{G}$. And in $\mathcal{G}$, $\mathtt{Na_i}$ points from $\mathtt{Bc_i}$ to $\mathtt{Bn_i}$, and $\mathtt{Ca_i}$ points from $\mathtt{Bn_{i-1}}$ to $\mathtt{Bc_i}$. This is because we never remove any edges corresponding to network actions, nor do we remove any nodes, and the destination nodes have same labels in $\mathcal{T}$ and $\mathcal{G}$. Since $\mathtt{Bn_i}$ exists in $\mathcal{G}$, we arrive at a contradiction.

Case 2. $\texttt{Bn}_\texttt{i}$ exists in $\mathcal{G}$ only if it exists in $\mathcal{T}$. All nodes with label type $\texttt{Bn}$ have a CCA action in $\mathcal{T}$. So $\texttt{Bn}_\texttt{i}$ has an action in $\mathcal{T}$. In constructing $\mathcal{G}$, we never remove all edges for a node with label type $\texttt{Bn}$, so we will keep at least one outgoing edge (action) for $\texttt{Bn}_\texttt{i}$. This is a contradiction.

$\mathcal{A}_\mathcal{B}$ satisfies $\mathcal{P}_f$. Recall, $\mathcal{P}_f$ is defined as a function of $\langle \texttt{Bn}, \texttt{Ca} \rangle$. From our construction, $\mathcal{A}_\mathcal{B}$ takes an action $\texttt{Ca}$ on $\texttt{Bn}$ only if $\mathcal{G}$ has edge $\texttt{Ca}$ on $\texttt{Bn}$. This happens only if $\mathcal{T}$ has such and edge which happens only if $\mathcal{A}_\mathcal{D}$ takes $\texttt{Ca}$ on $\texttt{Bn}$. Since, $\mathcal{A}_\mathcal{D}$ satisfies the performance property, $\mathcal{P}_f(\langle \texttt{Bn}, \texttt{Ca} \rangle) = \texttt{True}$. Hence, all of $\mathcal{A}_\mathcal{B}$'s actions satisfy the performance property.

□

Performance properties in §4.1.2 and §4.2.3 can be expressed as a boolean valued function of belief set and CCA's action on that belief set. At a high level, our performance properties dictate that the CCA (1) ensures some bounds on metrics like loss, delays, and utilization, and (2) makes progress. We show how both these can be expressed as a function of belief and CCA action.

We can compute delays, losses, and utilization for each tuple in the belief. The tuple tells us the starting queue, link rate, and buffer; and the CCA action tells us the sending rate. We can use this to calculate if the sending rate choice inflates delays or causes losses (e.g., using $\frac{dq}{dt} = (\lambda(t) - C)^+$, where $(x)^+ = \max(0, x)$). Likewise, we can compute if the link is utilized during the action using starting queue, link rate and the sending rate choice. If the CCA violates the delay, loss, utilization bounds on any tuple then the property evaluates to $\texttt{False}$ on the $\langle$belief, CCA action$\rangle$ pair, otherwise the property evaluates to $\texttt{True}$.

For progress, there are two types, (1) shrinking beliefs, and (2) stabilizing queue. For the first, we can check if an action can lead to shrinking beliefs by emulating all valid network actions. The CCA makes progress only if the beliefs shrink no matter the network action. For the second, the queue state is present in the belief tuple, and we can compute how the rate choice and link rate in the tuple will change the queue; and determine whether the queue will stabilize.

Note, our proof does not hold for arbitrary safety properties defined as boolean valued function over the entire execution. For instance, an arbitrary safety property might require that every time a CCA sends at rate 10 Mbps, in the next RTT it must send at 20 Mbps. Since we change the specific decisions that $\mathcal{A}_\mathcal{B}$ makes compared to $\mathcal{A}_\mathcal{D}$, such a safety property can be violated.

Also, note that we cannot express our performance properties solely as a function of the belief set. For example, the belief set does not tell us if the CCA will cause loss on an action, and we need to know the CCA's action on a belief set to evaluate loss. This is different from chess where the winning condition is just a function of the board state. Thus, Theorem 3.2.1 is non-obvious [26, 51].

A.2 Computing beliefs

A.2.1 Propagation delay and jitter

We assume that the CCA knows R and D. Due to discretization of time in the SMT encoding, all quantities with units of time are integer multiples of the discretization interval. The time for synthesis and verification scales with the number of intervals considered (§4.2.1). To cover as many `RTT`s in as few intervals as possible, we set $R = D = 1$ interval. As a result the constant value 1 reveals R and D in the templates.

This is a non-issue as jitter inherently creates uncertainty in `RTT`s. CBR-delay with parameters $\langle R, D\rangle$ can emulate parameters $\langle R', D'\rangle$ as long as $R' \geq R$ and $R' + D' \leq R + D$. Now, say the actual network parameters are $\langle R^a, D^a\rangle$. From the first `RTT` measurement we set $R = \texttt{RTT} \in [R^a, R^a + D^a]$ (as there are no other flows, so no queuing). At this point, we run the CCA designed for $\langle R, D = R\rangle$ and get the performance guarantees for $\langle R, D\rangle$ as long as $D^a \leq R + D - R^a$. If some future $\texttt{RTT}'$ is $< R$, we update $R = \texttt{RTT}'$ and repeat our argument, this time with better performance guarantees as R is closer to R^a. E.g., if we guarantee $q \leq R$ for $\langle R, D\rangle$, then on the actual network we guarantee $q \leq R \leq R^a + D^a$ and our guarantee improves as $R^a - R$ decreases. Same reasoning holds for CCAC.

In summary, we run the CCA designed for $\langle R = min\texttt{RTT}, D = min\texttt{RTT}\rangle$ and get the guarantee we promised for the network $\langle min\texttt{RTT}, min\texttt{RTT}\rangle$, when we are actually running on the network $\langle R^a, D^a\rangle$. We get these guarantees as long as $D^a \leq 2min\texttt{RTT}$, which is true if the real network can be captured by the CBR-delay model with parameters $D^a \leq R^a$.

A.2.2 Bandwidth

We show the analytical derivation of the link rate belief bounds (C_L and C_U) for the CCAC model. Before that we briefly give relevant background on CCAC (for detailed background please see [19]).

Background on CCAC. CCAC models the network as a generalized token bucket filter. It puts constraints (Listing A.1) on how the network serves packets (i.e., $S(t)$, service curve, or cumulative bytes serviced by time t), based on how packets arrive into the network (i.e., $A(t)$, cumulative bytes arrived by time t).

Tokens arrive at rate C bytes/secs, i.e., $C \cdot t$ tokens arrive by time t. On a token arrival, the network decides whether to admit it or waste it (Eq. A.2.4). The network uses admitted tokens to send packets, so the packets sent are upper bounded by the number of admitted tokens (Eq. A.2.1). Also, the network cannot send more bytes than arrived but not lost (Eq. A.2.2).

To emulate non-congestive jitter, the network can choose to delay sending packets even when tokens are available. To ensure jitter is bounded, all tokens must be used within D seconds, and the network should only admit a token if it knows it will be used within D seconds.[1] This puts a lower bound on service (Eq. A.2.3).

[1]Due to non-determinism, the network can look into the future to make these decisions.

Listing A.1: CCAC constraints on S

$$S(t) \leq T_A(t) = S_U(t) \tag{A.2.1}$$
$$S(t) \leq A(t) - L(t) \tag{A.2.2}$$
$$S(t) \geq T_A(t-D) = S_L(t) \quad \text{where,} \tag{A.2.3}$$
$$T_A(t) = C \cdot t - W(t) \tag{A.2.4}$$
$$q(t) \geq 0 \implies W'(t) = 0 \tag{A.2.5}$$
$$q(t) = A(t) - L(t) - T_A(t) \tag{A.2.6}$$

$T_A(t)$ – cumulative tokens (in units of bytes) admitted
$q(t)$ – inst. bytes in the bottleneck queue
$W(t)$ – cumulative tokens (in units of bytes) wasted
$L(t)$ – cumulative bytes lost

Additionally, to prevent the network from wasting all the tokens, tokens cannot be wasted when there are packets waiting for tokens, i.e., there are packets that have arrived (but not lost) and do not have corresponding admitted tokens (Eq. A.2.5 and Eq. A.2.6). These packets are put into a queue, and they build congestive delays. Other packets that have corresponding tokens but have not been delivered as considered as facing non-congestive delays.

Analytical derivation. CCAC visualizes its constraints using graphs like Fig. A.2. From such graphs, we can determine the minimum and maximum average ACK rate (r^{ACK}) that a CCA can get for different λ and C choices. Specifically, consider the case when $\lambda \geq C$. In this case, tokens cannot be wasted ($W' = 0$). As a result S_L and S_U have slope C, and the network has to ensure that at all times S is between the S_L and S_U lines. We look at different feasible choices of S, and find the maximum and minimum slope of S (ACK rate) over time intervals of different lengths. Note, we are interested in the "average" ACK rate over the intervals, so we look at the slope of the line joining the start and end points of the interval.

Fig. A.2 shows different feasible S curves along with minimum and maximum slopes in different intervals. For example, over an interval of length D, $r^{ACK} \in [0, C]$. Similarly, $r^{ACK} \in [C/2, 3C/2]$ for $2D$ long intervals, $r^{ACK} \in [2C/3, 4C/3]$ for $3D$ long intervals. For kD long intervals, $r^{ACK} \in [\frac{k-1}{k}C, \frac{k+1}{k}C]$. Following a similar exercise, we find that for intervals with length T, $r^{ACK} \in [\frac{C(T-D)}{T}, \frac{C(T+D)}{T}]$. I.e., S follows an ideal link with rate C with an additional burst or delay of CD bytes.

We repeat this for the case when $\lambda < C$. Here, we consider two extreme cases: (C1) S_L, S_U have slope λ, and (C2) S_L, S_U have slope C. The first case corresponds to the network wasting tokens because packets arrive slower than tokens, and the second corresponds to a large queue build up at time $t = 0$ preventing the network from wasting tokens. When we inspect the graphs (not shown, similar to Fig. A.2), we find that over intervals of length T, $r^{ACK} \in [\frac{\lambda(T-D)}{T}, \frac{\lambda(T+D)}{T}]$ in case

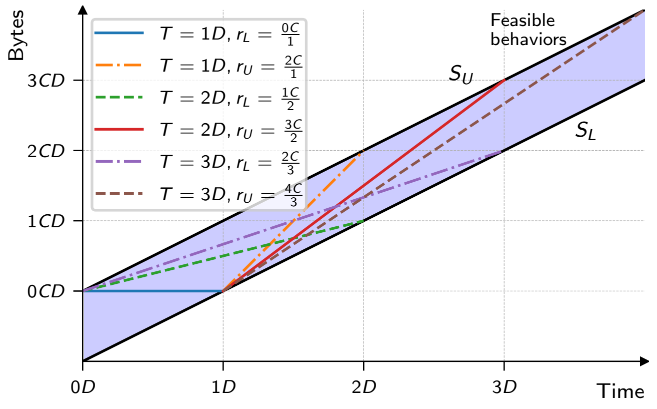


Figure A.2: Computing ACK rate range based on C. ACK rate is the slope of the lines. Over different time intervals (T), the lines show feasible service curves with the maximum (r_U^{ACK}) and minimum (r_L^{ACK}) slopes. Service curves need to be non-decreasing and lie in the shaded region.

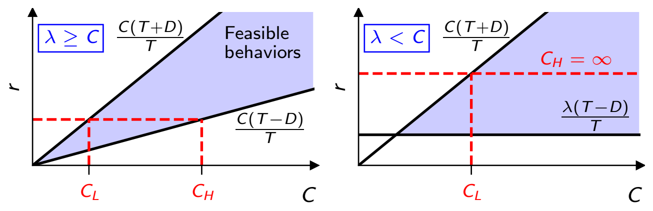


Figure A.3: Inverting bounds on observations to get bounds on network parameters. For observed ACK rate, draw horizontal line (red dashed line) corresponding to the observed ACK rate r^{ACK}, the points intersecting with the feasible region give the range of link rates $[C_L, C_U]$ that could have produced r^{ACK}.

C1, and $r^{ACK} \in [\frac{C(T-D)}{T}, \frac{C(T+D)}{T}]$ in case C2. When we look at other cases between C1 and C2, i.e., S_L and S_U take slopes in the range $[\lambda, C]$, we get more feasible values of r^{ACK}. Taking the union of all the r^{ACK} ranges, we find that in any T long interval, whenever $\lambda < C$, $r^{ACK} \in [\frac{\lambda(T-D)}{T}, \frac{C(T+D)}{T}]$.

We invert the bounds on r^{ACK} to get bounds on C. Fig. A.3 illustrates this. Specifically, when $\lambda > C$, we have, $\forall T$:

$$\frac{C \cdot (T-D)}{T} \leq r^{ACK} \leq \frac{C \cdot (T+D)}{T}$$

On rearranging, we get:

$$\frac{r^{ACK} \cdot T}{T+D} \leq C \leq \frac{r^{ACK} \cdot T}{T-D} \tag{A.2.7}$$

Likewise, when $\lambda < C$, we have, $\forall T$:

$$\frac{\lambda \cdot (T-D)}{T} \leq r^{ACK} \leq \frac{C \cdot (T+D)}{T}$$

On rearranging, we get:

$$\frac{r^{ACK} \cdot T}{T+D} \leq C \tag{A.2.8}$$

From Eq. A.2.7 and Eq. A.2.8, we get:

$$C \geq \max_T \frac{r^{ACK} \cdot T}{T+D} = C_U \quad \text{and} \quad C \leq \min_T \frac{r^{ACK} \cdot T}{T-D} = C_L$$

Where C_U can only be computed over intervals where $\lambda > C$.

The CCA can compute C_L and C_U as defined. r^{ACK} can be measured directly by the CCA. The CCA can also infer if $\lambda(t) > C$ at all time steps t in an interval if $(qdel_L(t) > 0) \vee (L'(t) > 0)$ for all time steps t in the interval. We checked that this condition holds by querying CCAC if $C > C_U$ can happen when we compute C_U over intervals where $(qdel_L(t) > 0) \vee (L'(t) > 0)$ is true. CCAC returned UNSAT implying that C has to be $\leq C_U$.

Appendix B

CCmatic synthesis, tradeoffs, proofs, and evaluation details

B.1 Synthesis details

For completeness, we describe the workings of the generator and verifier in CEGIS (§4.1). For ease of understanding, we interpret the search (or program synthesis) problem as a ∃∀ formula [6]. For instance, "does there *exist* a belief-based CCA, such that *forall* traces captured by the network model, the CCA ensures the desired performance properties". More formally, the formula is:

$$\exists \boxed{?} \forall \texttt{trace} \ (\texttt{CCA} \land \texttt{Network}) \implies \texttt{Performance} \tag{B.1.1}$$

$\boxed{?}$ and **`trace`** represent vector of variables. $\boxed{?}$ are the holes in the CCA template (Listing 4.1). Assigning value to the holes produces a concrete CCA. A **`trace`** (timeseries) describes the execution of the CCA under the network model. It is specified using the dimensions of the network model's relation (§2.3), e.g., path, feedback, CCA actions. In our case, the **`trace`** includes variables like C, β, R, D, $\texttt{rate}(t)$, $A(t)$, $S(t)$, $L(t)$, $T_A(t)$, $W(t)$ (Table 2.1, Listing A.1).

The `CCA`, `Network`, and `Performance` are boolean valued SMT formulas (in the theory of linear real arithmetic (LRA) [72]) over the $\boxed{?}$ and **`trace`** variables. `CCA` ensures that the congestion control decisions are made according to the CCA. It encodes that the $\texttt{rate}(t)$ variables are assigned using the values of $\boxed{?}$ and **`trace`** variables according to the CCA template. We produce this encoding by symbolically executing the CCA template. `Network` encodes what assignments to the **`trace`** variables result in executions that are deemed feasible according to the network model. These look like Listing A.1. We also encode belief computations (§4.1.1.1) and timeouts (§4.1.1.2) in `Network`, these constraints merely populate the belief bounds and don't affect the actions that the network takes. `Performance` encodes the desired objectives using the transition system. We use the synthesis invariants (Eq. 4.1.3) to specify `Performance`.

The formula can be read as "does there exist an assignment to the holes in the template such that for all traces, if the packets are sent according to the CCA and the service/delay/loss decisions are made according to the network model, then the trace satisfies the desired performance properties".

Verifier operation. For a given CCA (i.e., value of $\boxed{?}$), the verifier produces a counterexample `trace` by solving the formula:

$$\boxed{?} = \texttt{value of } \boxed{?} \;\wedge \texttt{CCA} \wedge \texttt{Network} \wedge \neg\texttt{Performance} \tag{B.1.2}$$

This is a quantifier-free formula (i.e., no $\exists$ or $\forall$ quantifiers). The verifier uses an SMT solver (e.g., Z3 [40]) to solve this formula. This is a formula on the `trace` variables (as the $\boxed{?}$ variables have been substituted by fixed values). The assignment to the `trace` variables describes a trace where the given CCA violates the performance properties on the network model. If the formula is unsatisfiable, then there is no counterexample that breaks the CCA.

Generator operation. Given a set of counterexample traces (say $\mathbb{X}$), the generator solves the following formula to propose a new candidate CCA:

$$\bigwedge_{\texttt{trace}\in\mathbb{X}} (\texttt{CCA} \wedge \texttt{Network}) \implies \texttt{Performance} \tag{B.1.3}$$

This is a formula on the $\boxed{?}$ variables (as the `trace` variables have been substituted by concrete counterexamples from the set $\mathbb{X}$). The assignments to the $\boxed{?}$ variables that satisfy the formula are those on which either the `trace` is no longer feasible according to the CCA/network model, or it satisfies the performance properties.

Note, in this formulation, only `CCA` depends on the holes, the `Network` and `Performance` only depend on the `trace` variables. On each `trace` in $\mathbb{X}$, `Network` evaluates to `True` and `Performance` evaluates to `False` (as the `trace` was generated by the verifier by satisfying "$\cdots \wedge$ `Network` $\wedge$ `Performance`"). As a result, Eq. B.1.3 simplifies to:

$$\bigwedge_{\texttt{trace}\in\mathbb{X}} \neg\texttt{CCA} \tag{B.1.4}$$

Effectively, the new candidate should not produce the exact same trace of rate choices as made by the prior candidate CCAs. So all the CCAs that have the same buggy control flow exploited by the counterexample trace are pruned from the search space.

B.2 Loss vs. convergence tradeoffs

Tradeoff theorem and proof. For ease of understanding, we show the proof for a CCA trying to ramp up its rate from 0 to the link rate C, while trying to avoid large loss events. Later we generalize this to CCA trying to ramp up from arbitrary C_0 to C, while trying to risk at most $O(f(C))$ losses for some function $f(.)$.

Theorem. *For an end-to-end deterministic CCA running on a CBR-delay network with parameters $\langle C, R, D, 0 < \beta \leq CD\rangle$, to avoid getting arbitrarily low utilization, the CCA must either (1) cause $\omega(1)$ packet loss, i.e., losses that increases with C, or (2) take $\Omega(C(R + \beta/C))$ time to converge to the link rate.*

Proof. We will first show that under the parameters of the proof, i.e., $\beta \le CD$, the CCA must cause loss to avoid arbitrarily utilizing the link (**Step 1.**). Then we compute a tight lower bound belief for C, under the CBR-delay link (our prior belief computations were for CCAC) (**Step 2.**). This allows us to compute the amount of loss the CCA risks any time it tries to probe for bandwidth (**Step 3.**). If we restrict this risk of loss to a constant independent of C, it gives us a constraint on how quickly the CCA can ramp up, giving us a lower bound on the convergence time (**Step 4.**).

For the proof, we assume the CCA knows R, D, and β_s. β_s is the seconds of queueing that the buffer can tolerate (i.e., β/C). Knowledge of these quantities only makes the proof stronger. If CCAs need to respect the tradeoff with the knowledge, then they also need to respect it without the knowledge. Note, `cc_probe_slow` meets the theorem bounds without knowing β_s.

Step 1. Since $\beta_s \le D$, the CCA must cause loss to avoid arbitrarily low utilization. This immediately follows from Theorem 2 of [20], i.e., to avoid arbitrarily low utilization, the CCA must cause more than D queueing delay. If the buffer size is $\le D$ seconds, the CCA will have to cause loss. For completeness, we repeat the proof in Appendix B.2.1.

Step 2. Until the time that loss happens, we compute the set of paths (link rates) that the CCA could be running on given its observations. If the CCA has not seen a loss event by time t^*, then the CCA never over-flowed the buffer until time $t^* - \mathtt{RTT}(t^*)$. I.e., in any time interval before $t^* - \mathtt{RTT}(t^*)$[1], the net bytes enqueued and the net bytes dequeued differ by at most β. I.e., $\forall t_1, t_2$, such that, $0 \le t_1 \le t_2 \le t^* - \mathtt{RTT}(t^*)$:

$$\int_{t_1}^{t_2} \lambda(s)ds - C\cdot (t_2 - t_1) \le \beta$$

Substituting $\beta = C\beta_s$ and rearranging, we get, $\forall t_1, t_2.\ 0 \le t_1 \le t_2 \le t^* - \mathtt{RTT}(t^*)$:

$$C \ge \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + \beta_s}$$

Based on this we define $C_{L,\lambda}$ as:

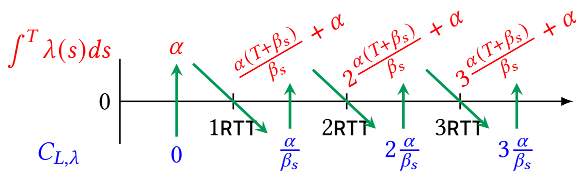
$$C_{L,\lambda}(t^*) = \max_{0 \le t_1 \le t_2 \le t^* - \mathtt{RTT}(t^*)} \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + \beta_s} \tag{B.2.1}$$

The set of paths that can produce the trace up to t^* is $\{\langle C = C^*, \beta = C^*\beta_s\rangle | C^* \ge C_{L,\lambda}(t)\}$. The CCA could be running on any of these paths, and it needs to ensure its performance properties no matter which of these paths it is running on.

Step 3. On the path $\langle C^*, C^*\beta_s\rangle$, loss happens whenever the CCA over-flows the buffer, i.e., if the enqueued and dequeued bytes differ by more than the buffer size in some time interval. I.e., loss happens if for some interval $[t_1, t_2]$,

$$\int_{t_1}^{t_2} \lambda(s)ds > C^* \cdot (t_2 - t_1) + C^* \cdot \beta_s$$

[1]Note, the CCA only knows what happened in the network one $\mathtt{RTT}$ ago. At time t, the CCA has no information about what may have happened in the network during the time interval $(t - \mathtt{RTT}(t), t]$.



$\int^T \lambda(s)ds$
α
$\frac{\alpha(T+\beta_s)}{\beta_s} + \alpha$
$2\frac{\alpha(T+\beta_s)}{\beta_s} + \alpha$
$3\frac{\alpha(T+\beta_s)}{\beta_s} + \alpha$
0
1RTT
2RTT
3RTT
$C_{L,\lambda}$
0
$\frac{\alpha}{\beta_s}$
$2\frac{\alpha}{\beta_s}$
$3\frac{\alpha}{\beta_s}$

Figure B.1: $C_{L,\lambda}$ puts constraints on $\int^T \lambda(s)ds$, and $\int^T \lambda(s)ds$ puts constraints on $C_{L,\lambda}$. Over the intervals, the red and blue values show the maximum value of $\int^T \lambda(s)ds$ and $C_{L,\lambda}$ respectively. The green arrows show the constraint dependencies. I.e., $C_{L,\lambda}$ in $[k\texttt{RTT}, (k+1)\texttt{RTT})$ puts constraints on $\int^T \lambda(s)ds$ in $[k\texttt{RTT}, (k+1)\texttt{RTT})$ and $\int^T \lambda(s)ds$ in $[k\texttt{RTT}, (k+1)\texttt{RTT})$ puts constraints on $C_{L,\lambda}$ in $[(k+1)\texttt{RTT}, (k+2)\texttt{RTT})$.

Specifically, on the path $\langle C_{L,\lambda}(t), C_{L,\lambda}(t)\beta_s\rangle$, i.e., the smallest link rate that can justify CCA's observations, the amount of loss is: $\Delta L(t_0, t) = \int_{t_0}^{t} \lambda(s)ds - \left(C_{L,\lambda}(t)\cdot (t - t_0) + C_{L,\lambda}(t)\cdot \beta_s\right)$

Step 4. Convergence can only happen after the CCA causes loss (until loss the CCA could be running on arbitrarily large link rate from **Step 1.**). It needs to do this while ensuring that the amount of loss it risks is bounded, i.e., for all intervals $0 \le t_1 \le t_2$, $\Delta L(t_1, t_2) \le \alpha$ for some constant α independent of C. We compute a lower bound on the time to loss under this constraint.

Loss happens only if buffer overflows, i.e., in some interval $[t_1^*, t_2^*]$ $\int_{t_1^*}^{t_2^*} \lambda(s)ds > C\cdot (t_2^* - t_1^*) + C\cdot \beta_s$. We also know that CCA needs to ensure $\Delta L(t_1^*, t_2^*) \le \alpha$. From this, loss happens only if $C_{L,\lambda}(t_2^*) \ge C - \alpha/\beta_s$. See Appendix B.2.1 for details.

We compute time for $C_{L,\lambda}$ to increase to $C - \alpha/\beta_s$. Initially $C_{L,\lambda}(0) = 0$. Since $C_{L,\lambda}$ can only be computed after the first RTT (from the definition of $C_{L,\lambda}$), $C_{L,\lambda}$ is 0 over the entire interval $[0, \texttt{RTT})$. This puts constraints on sending rate choices in $[0, \texttt{RTT})$. Specifically, substituting $C_{L,\lambda} = 0$ in $\Delta L \le \alpha$, we get, $\int_{t_1}^{t_2} \lambda(s)ds \le \alpha$ for all intervals $[t_1, t_2]$ inside $[0, \texttt{RTT})$. Thus, if we compute $C_{L,\lambda}(t)$ at any time $t \in [\texttt{RTT}, 2\texttt{RTT})$, we get $C_{L,\lambda}(t) \le \frac{\alpha}{\beta_s}$ (because on intervals $0 \le t_1 \le t_2 < 2\texttt{RTT} - \texttt{RTT}$, i.e., $t_2 < \texttt{RTT}$, the numerator in Eq. B.2.1, "$\int_{t_1}^{t_2} \lambda(s)ds$", is at most α).

We can similarly obtain that in interval $[k\texttt{RTT}, (k+1)\texttt{RTT})$, $C_{L,\lambda}$ is at most $k\cdot (\alpha/\beta_s)$ (illustrated in Fig. B.1). For $C_{L,\lambda}$ to ramp up to $C - \alpha/\beta_s$, we need $k \ge C\cdot (\beta_s/\alpha) - 1$. Hence, the convergence time is at least k RTTs, or "$C\cdot (\beta_s/\alpha) - 1$ RTTs". Before loss, each RTT can be as large as $R + \beta_s$ (either due to queueing delay or jitter), making the convergence time $= (C\cdot (\beta_s/\alpha) - 1)\cdot (R + D) = \Omega(C(R + \beta_s))$.

$\square$

Generalizing Theorem 4.2.1. If the CCA wants to ramp up from C_0 to C while ensuring its risks at most $f(C)$ loss, then the convergence time is $\Omega(F^{-1}(C)(R + \beta_s))$, where F^{-1} is the inverse of F and F is defined by the recursion:

$$F(0) = C_0 \quad \text{and,} \quad F(k) = F(k-1) + f(F(k-1))/\beta_s$$

If C is not in the domain of F^{-1}, we evaluate F^{-1} at the smallest value greater than C in the domain of F^{-1}.

We obtain this result by replacing $C_{L,\lambda}(0) = C_0$ in **Step 4.**, and ensuring that risk of loss $\Delta L \leq f(C_{L,\lambda})$. The function $F(k)$ tracks the maximum possible value of $C_{L,\lambda}$ at any time in the interval $[k\texttt{RTT}, (k+1)\texttt{RTT})$.

B.2.1 Proof details

We fill in the details skipped in the proof.

Step 1. We will prove below that the CCA must cause $\texttt{RTT} > R + D$ or loss to avoid arbitrarily low utilization. Since (1) the buffer is not big enough to build D seconds of delay (because $\beta_s < D$), and (2) the delay box in CBR-delay can avoid adding any jitter, the end-to-end delays can always be $\leq R + \beta_s \leq R + D$, forcing the CCA to cause losses (as it can no longer cause $\texttt{RTT} > R + D$).

Proof. We prove by contradiction, i.e., if the CCA does not cause $\texttt{RTT} > R + D$ or loss, then we can construct an execution where the CCA gets arbitrarily low utilization. Say the CCA produces an infinite execution with $\texttt{RTT} \leq R + D$ without ever causing loss on a CBR-delay link (say link I) with bandwidth C_I and β_s seconds of buffering. This exact RTT sequence can be produced by another CBR-delay link (say link II) with rate $C_{II} \gg C_I$ (see construction below). Since the CCA is deterministic, and it gets same sequence of RTTs as feedback on both links, it will make the same sending rate choices on both the links.[2] However, the CCA's average sending rate is $\approx C_I$ (as it does not build large queues on link I given that its RTTs are at most $R + D$). Such sending rate gets arbitrarily low utilization on link II for arbitrarily large C_{II}.

Construction. Link II can produce same RTT sequence as link I, by choosing $\texttt{jitter}_{II}(t) = \texttt{RTT}_\texttt{I}(t) - R - qdel_{II}(t)$ (we use subscripts I and II to refer to quantities on link I and II respectively). With this choice $\texttt{RTT}_\texttt{II}(t) = R + qdel_{II}(t) + \texttt{jitter}_{II}(t) = \texttt{RTT}_\texttt{I}(t)$. We just need to show that this is a feasible choice for $\texttt{jitter}_{II}$, i.e., $0 \leq \texttt{jitter}_{II}(t) \leq D$.

$\texttt{jitter}_{II}(t) \leq D$.

$$\begin{aligned}
&\text{We know } \texttt{RTT}_\texttt{I}(t) \leq R + D, \text{ or } \texttt{RTT}_\texttt{I}(t) - R \leq D,\\
&\text{thus } \texttt{jitter}_{II}(t) = \texttt{RTT}_\texttt{I}(t) - R - qdel_{II}(t)\\
&\qquad\qquad\qquad \leq D - qdel_{II}(t) \leq D\\
&\qquad\qquad\qquad (\text{as } qdel_{II}(t) \geq 0 \text{ by definition of } qdel)
\end{aligned}$$

$\texttt{jitter}_{II}(t) \geq 0$.

$$\begin{aligned}
&\text{Since } C_{II} \gg C_I,\ qdel_{II}(t) \leq qdel_I(t)\\
&\quad (\text{increasing } C \text{ decreases congestive queueing delays})\\
&\text{As a result, } \texttt{RTT}_\texttt{I}(t) = R + qdel_I(t) + \texttt{jitter}_I(t)\\
&\qquad\qquad\qquad\quad \geq R + qdel_{II}(t) + \texttt{jitter}_I(t)\\
&\text{Or, } \texttt{RTT}_\texttt{I}(t) - R - qdel_{II}(t) \geq \texttt{jitter}_I(t) \geq 0\\
&\text{Thus, } \texttt{jitter}_{II}(t) = \texttt{RTT}_\texttt{I}(t) - R - qdel_{II}(t) \geq 0
\end{aligned}$$

[2]Note RTTs capture all the information that a CCA can obtain from feedback. Metrics like loss, ACK-rate, etc. can be derived from packet send events and RTT sequence [20].

Step 4. Loss only happens when $C_{L,\lambda} \geq C - \alpha/\beta_s$. For loss to happen, the buffer needs to overflow, i.e., in some interval $[t_1^*, t_2^*]$

$$\int_{t_1^*}^{t_2^*} \lambda(s)ds > C\cdot (t_2^* - t_1^*) + C\cdot \beta_s \tag{B.2.2}$$

And, we also know the CCA need to ensure its risk of loss is bounded, i.e., $\Delta L(t_1^*, t_2^*) \leq \alpha$, i.e.,

$$\int_{t_1^*}^{t_2^*} \lambda(s)ds - \left(C_{L,\lambda}(t_2^*)\cdot (t_2^* - t_1^*) + C_{L,\lambda}(t_2^*)\cdot \beta_s\right) \leq \alpha$$

$$\text{or, } \int_{t_1^*}^{t_2^*} \lambda(s)ds \leq C_{L,\lambda}(t_2^*)\cdot \left(t_2^* - t_1^* + \beta_s\right) + \alpha \tag{B.2.3}$$

We can only meet both these constraints (i.e., Eq. B.2.2 and Eq. B.2.3) when:

$$\begin{aligned} & C_{L,\lambda}(t_2^*)\cdot \left(t_2^* - t_1^* + \beta_s\right) + \alpha > C\cdot (t_2^* - t_1^*) + C\cdot \beta_s \\ \text{or, } & C_{L,\lambda}(t_2^*) > C - \alpha/(t_2^* - t_1^* + \beta_s) \\ \text{or, } & C_{L,\lambda}(t_2^*) > C - \alpha/\beta_s \end{aligned}$$

B.3 Synthesizing `cc_probe_slow`

We describe properties of probe intervals used in §4.2.2.1 to inform belief computation and encoding. Recall, probe intervals are intervals that lead to increase in $C_{L,\lambda}$ through the equation:

$$C_{L,\lambda}(t^*) = \max_{0 \leq t_1 \leq t_2 \leq t^* - \mathsf{RTT}(t^*)} \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D}$$

Measurement intervals influence probing intervals, or future probe intervals cannot be shorter than past probe intervals. Say in the past $C_{L,\lambda}^p$ was computed over an interval of length T^p. In the future, $C_{L,\lambda}$ increases to $C_{L,\lambda}^f = C_{L,\lambda}^p + \epsilon$, and was computed over an interval $T^f < T^p$. We will show that this action risks losing more than constant loss, i.e., loss can increase with C.

The amount of loss the future probe risks is (from difference in net enqueued bytes, dequeued bytes, and buffer size):

$$\Delta L = \int^{T^f} \lambda(s)ds - \left(C\cdot T^f + \beta\right) \tag{B.3.1}$$

We make the following substitutions in Eq. B.3.1:

(1) $\int^{T^f} \lambda(s)ds = C_{L,\lambda}^f \cdot (T^f + D)$ (from the definition of $C_{L,\lambda}$).

(2) $C_{L,\lambda}^f = C_{L,\lambda}^p + \epsilon$

(3) From the past probe, we know $\int^{T^p} \lambda(s)ds \leq C \cdot T^p + \beta$ (as loss did not happen, Eq. 4.2.3), and $\int^{T^p} \lambda(s)ds = C^p_{L,\lambda} \cdot (T^p + D)$ (from the definition of $C_{L,\lambda}$). From these two inequalities, we get $C^p_{L,\lambda} \cdot (T^p + D) \leq C \cdot T^p + \beta$.

We substitute $C \cdot T^p + \beta = C^p_{L,\lambda} \cdot (T^p + D)$, or $C^p_{L,\lambda} = \frac{C \cdot T^p + \beta}{T^p + D}$, to evaluate how much loss would happen on this network.

On making the substitutions, and algebraic simplifications, we get:

$$\Delta L = \frac{(CD - \beta) \cdot (T^p - T^f)}{T^p + D} + \epsilon \cdot (T^f + D) \tag{B.3.2}$$

For $T^f < T^p$, ΔL can be an increasing function of C as long as $\beta < CD$, e.g., $\beta = C\beta_s$ for $\beta_s < D$. Hence, loss is not bounded by a constant independent of C.

A probe interval must start with drained queue. Say $[t_1, t_2]$ is a probing interval, i.e., it leads to an increase in $C_{L,\lambda}$, and it does not start with a drained bottleneck queue; then we will show that the probe interval risks losing all the packets in the bottleneck queue. As a result, to ensure losses are bounded, a CCA needs to ensure that the bottleneck queue is bounded at the beginning of the probe interval.

Say the $[t_1, t_2]$ probe interval causes an increase in $C_{L,\lambda}$ at time t, i.e., $0 \leq t_1 \leq t_2 \leq t - \mathsf{RTT}(t)$, and,

$$C_{L,\lambda}(t) = \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} > C_{L,\lambda}(t - \epsilon) \tag{B.3.3}$$

for some small $\epsilon > 0$. If this is not true, then $[t_1, t_2]$ is not a probing interval. By definition, $C_{L,\lambda}$ can only increase over time, so $C_{L,\lambda}(t - \epsilon) \geq C_{L,\lambda}(t_2)$. Using this and Eq. B.3.3, we get, $\frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} > C_{L,\lambda}(t_2)$, or,

$$\int_{t_1}^{t_2} \lambda(s)ds - C_{L,\lambda}(t_2) \cdot (t_2 - t_1 + D) > 0 \tag{B.3.4}$$

The risk of loss during the probe is: ΔL = (bytes already in queue) + (enqueued bytes) − (dequeued bytes+buffer). We evaluate this equation on the path $\langle C = C_{L,\lambda}(t_2), \beta = C_{L,\lambda}(t_2)D \rangle$. It can produce observations till time t_2 as it is in the belief set (derived in §4.2.2.1), assuming until t_2, the CCA has not witnessed $\mathsf{RTT} > R + D$, or losses. Plugging this path into ΔL, we get:

$$\Delta L = q(t_1) + \int_{t_1}^{t_2} \lambda(s)ds - \big(C_{L,\lambda}(t_2) \cdot (t_2 - t_1) + C_{L,\lambda}(t_2) \cdot D\big) \tag{B.3.5}$$

From Eq. B.3.5 and Eq. B.3.4, we get $\Delta L > q(t_1)$.

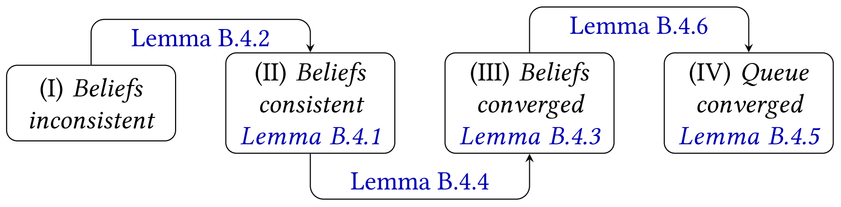

Lemma B.4.2
Lemma B.4.6
(I) *Beliefs inconsistent*
(II) *Beliefs consistent* *Lemma B.4.1*
(III) *Beliefs converged* *Lemma B.4.3*
(IV) *Queue converged* *Lemma B.4.5*
Lemma B.4.4

Figure B.2: State transition lifetime of `cc_qdel`. Note, a link rate change can cause a transition between any two states (these "all to all" transitions are not shown).

B.4 Proofs of performance

The synthesized CCAs ensure the synthesis invariant, but that is not sufficient to meet the performance objectives due to under-specification (§4.1.2). We build proofs, consisting of lemmas, that describe the states the CCA visits, transitions it makes, and the objectives it ensures. In the interest of space, we only discuss lemmas for `cc_qdel`. Lemmas for `cc_probe_slow` are similar but use $C_{L,\lambda}$, and q_U belief bounds instead of C_L and C_U.

Using the verifier, we checked that the lemmas are true for `cc_qdel` running on the CCAC network model with parameters $\langle C, R, D = R, \beta \geq 3C \cdot (R + D) \rangle$. We give a summary of how the lemmas work together, describe how we built them, and then discuss each lemma.

Summary. Fig. B.2 describes how the lemmas are related to each other. The lemmas are true over any $10R$ seconds trace of the CCA's execution. We stitch together lemmas to reason about performance over arbitrarily long time horizons.

Whenever the link rate changes significantly, C_L and C_U beliefs may become inconsistent. The CCA ensures that these beliefs become consistent exponentially fast (Lemma B.4.2), they converge to a small range exponentially fast (Lemma B.4.4), and finally, the CCA reaches steady state (Lemma B.4.6), i.e., the bottleneck queue reduces additively at rate proportional to C.[3] The lemmas ensure that progress always happens in the same direction. Specifically, the beliefs cannot become inconsistent on their own, i.e., without link rate varying (Lemma B.4.1), the beliefs cannot diverge post convergence (Lemma B.4.3), the bottleneck queue cannot become unbounded after it converges (Lemma B.4.5). In other words, once the CCA reaches steady state (IV), it stays there. In the steady state (IV), `cc_qdel` gets at least 89% utilization, keeps `RTTs` to less than $4.4(R + D)$ seconds, and loses at most 3 packets in any R duration (Lemma B.4.7). Additionally, Lemma B.4.8 ensures that `cc_qdel` never incurs large loss events when probing for bandwidth as long as the beliefs are consistent, i.e., in states II, III, IV.

Building lemmas using binary search. The lemmas include non-trivial constants, we obtain all these by using binary search. For instance, to compute utilization in steady state, we ask the verifier if `cc_qdel` violates Lemma B.4.5 for different choices of utilization. When we ask this query for utilization = 100% the verifier gives a counterexample showing that `cc_qdel` can

[3]A CCA cannot drain the queue faster than this. Even if the CCA stops sending packets, and the bytes are serviced at rate C, then the queue will only drain by $C \cdot t$ bytes in time t.

get lower than 100% utilization. However, when we repeat for 50%, the verifier says UNSAT, implying on (worst-case) all executions that start in steady state, `cc_qdel` gets at least 50% utilization. We repeat and find that `cc_qdel` ensures 89% utilization in steady state.

Lemma B.4.1. *Initial beliefs are consistent $\implies$ final beliefs are consistent. Where, beliefs are consistent $\equiv C_L(t) \leq C \leq C_U(t)$, and we evaluate initial beliefs at $t = 0$ and final beliefs at $t = T = 10R$.*

This lemma verifies that our belief computations are correct. I.e., any bandwidth C that can produce the observations in the trace from $t = 0$ to $t = T$, and also produce observations before $t = 0$ (i.e., C is in the initial belief set or "initial beliefs were consistent") should be in the final belief set (or "final beliefs should be consistent"). This lemma is true for any CCA.

Lemma B.4.2. *($\neg$ Initial beliefs are consistent) $\implies$ (final beliefs move towards consistency $\vee$ final beliefs are consistent $\vee$ steady state objectives). Where,*

$$\begin{aligned}
&\textit{Final beliefs move towards consistency} \equiv \\
&\qquad (C_L \textit{ inconsistent} \implies C \textit{ beliefs decrease}) \wedge \\
&\qquad (C_U \textit{ inconsistent} \implies C \textit{ beliefs increase}) \\
&C_L \textit{ inconsist.} \equiv C_L(0) > C, \quad C_U \textit{ inconsist.} \equiv C_U(0) < C \\
&C \textit{ beliefs dec.} \equiv \textit{At least one decreases} \wedge \textit{None increases} \\
&\textit{At least one decreases} \equiv \\
&\qquad C_L(T) \cdot 1.1 < C_L(0) \vee C_U(T) \cdot 1.1 < C_U(0) \\
&\textit{None increases} \equiv C_L(T) \leq C_L(0) \wedge C_U(T) \leq C_U(0) \\
&C \textit{ beliefs inc. is defined symmetric to } C \textit{ beliefs dec.} \\
&\textit{Steady state objectives} \equiv (\textit{Utilization lower bounded} \\
&\qquad \textit{Inflight upper bounded} \wedge \textit{No large loss events}) \\
&\qquad \equiv \frac{S(T) - S(0)}{C(T - D)} \geq 83\% \wedge \\
&\qquad \forall t.\, \theta(t) \leq 4.4C \cdot (R + D) \wedge \forall t.\, L(t) - L(t-1) \leq 3\mathit{MSS}
\end{aligned}$$

In each trace, the beliefs move towards consistency by a multiplicative factor, i.e., *exponentially fast*. The 1.1× factor is a worst-case movement over all possible $10R$ long executions. In traces where C_L, C_U are far from C, beliefs move quicker. Beliefs start moving slower (but at least 1.1×) only when they are very close to C. So in practice the amount that the beliefs move varies over time and the beliefs become consistent much quicker than if they only improved by only 1.1× every $10R$ seconds.

To encode "final beliefs move towards consistency", we require that (i) at least one of the C beliefs move in the correct direction *and* (ii) neither of them move in the wrong direction. The second clause (i.e., ii) is needed to ensure that the progress made by the CCA in consecutive traces adds up. Without this, it can happen that C_L increases and C_U decreases in the first trace, then C_U increases and C_L decreases, and so on. In each trace at least one of C_L or C_U is moving in the right direction (satisfying clause i), but their progress is not adding up.

Lemma B.4.3. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged). Where, beliefs are converged $\equiv C_L(t) \geq \frac{27C}{40} \wedge C_U(t) \leq 3C$.*

When the beliefs are in the range $[\frac{27C}{40}, 3C]$, they do not go outside this range unless the link rate varies. This is despite periodic belief timeouts (§4.1.1.2). The beliefs are only timed out when they are well within this range, so that they stay within the range even after the timeout if the link rate hasn't changed.

Recall, we found the constants 27/40 and 3 by binary search, i.e., the tightest range for which Lemma B.4.3 is true.

Lemma B.4.4. *(Initial beliefs are consistent $\wedge \neg$ initial beliefs are converged) $\implies$ (final beliefs are consistent $\wedge$ (final beliefs shrink $\vee$ final beliefs are converged $\vee$ steady state objectives)). Where,*

$$\textit{Final beliefs shrink} \equiv \textit{At least one improves } \wedge \textit{None degrade}$$
$$\textit{At least 1 imp.} \equiv C_L(T) > 1.7C_L(0) \vee C_U(T) < 1.7C_U(0)$$
$$\textit{None degrade} \equiv C_L(T) \geq C_L(0) \wedge C_U(T) \leq C_U(0)$$

The beliefs shrink by a multiplicative factor i.e., *exponentially fast.* Similar to Lemma B.4.2, we need the "at least one improves" *and* "none degrades" pattern to ensure that the progress made by the CCA adds up.

Lemma B.4.5. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ initial bottleneck queue is bounded) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged $\wedge$ final bottleneck queue is bounded). Where, bottleneck queue is bounded $\equiv q(t) \leq 3.3C \cdot (R + D)$.*

Lemma B.4.6. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge \neg$ initial bottleneck queue is bounded) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged $\wedge$ (final bottleneck queue is bounded $\vee$ bottleneck queue reduces)). Where, bottleneck queue reduces $\equiv q(T) < q(0) - CR/2$.*

To drain the queue, the CCA takes time that is linear in the queue size (i.e., it decreases queue *additively*, proportional to the `BDP` in each trace). Note, no CCA can multiplicatively reduce the number of bytes in the bottleneck queue. Even if the CCA stops sending packets, and the bytes are serviced at rate C, then the queue will only drain by $C \cdot t$ bytes in time t, this is independent of the number of bytes in the queue (i.e., not a multiple of the queue size).

Lemma B.4.7. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ initial bottleneck queue is bounded) $\implies$ steady state objectives.*

Lemma B.4.8. *(Initial beliefs are consistent) $\implies$ (rate increases $\implies$ no large loss events). Where, rate increases $\equiv$ `rate`$(T) >$ `rate`(0).*

Once the C beliefs are consistent, when `cc_qdel` is probing (or ramping up, i.e., "rate increases"), it never incurs any large loss events losses. Large losses may happen if the link rate decreases (there is no way to avoid this). Note, this is only true on networks with large buffers, i.e., $\beta \geq 3C \cdot (R + D)$.

B.5 Implementation issues in the Linux kernel

We originally implemented the synthesized CCAs in the Linux kernel. We uncovered bugs in the kernel's pacing implementation and also found the `cong_ctrl` API to be insufficient to implement the synthesized CCAs.

Pacing bug. To implement pacing, the Linux kernel pre-computes the "time to send the next packet" as the inverse of the pacing rate, i.e., inter-send-time. If the CCA's sending rate changes before the pre-computed time, then the kernel implements the wrong sending rate until the time to send the next packet. This leads to a discrepancy in the number of packets actually sent vs. the number of packets that the CCA wanted to send in a time interval.

In general, it is hard to implement pacing correctly. Pacing constrains two things, (1) a lower bound on the inter-send-time between packets and (2) number of packets sent in a time interval. Practically, a pacing implementation cannot meet both these constraints. Due to delays in CPU scheduling and interrupt handling, an instruction's execution may be delayed. These delays can cause a pacing implementation to miss a sending opportunity. When this happens, pacing implementation can either temporarily increase the sending rate to correct for the delayed opportunity (there by violating the lower bound on inter-send-time) or process the delayed sending opportunity as is (there by not sending any bytes corresponding to the delay and violating the constraint on total packets sent).

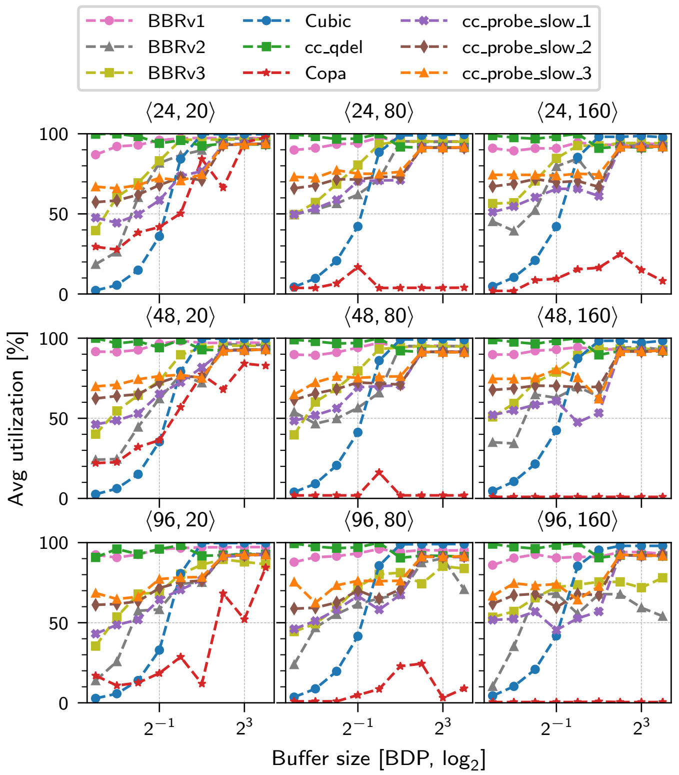
In fact, the pacing implementation of the genericCC [17] (the framework that we used to implement the synthesized CCAs in userspace over UDP) was also incorrect. We modified it so that it faithfully honors the lower bound on inter-send-time to ensure constant loss. Doing so creates a minor discrepancy between the packets sent over an interval vs. packets the CCA expected to send. Specifically, we maintain `last_sent_time` and compute `inter_send_time` whenever the CCA changes its sending rate. We launch a thread that polls (busy waits) to check if the sender is allowed to send (i.e., `current_time` is $\geq$ `last_sent_time` + `inter_send_time`).

The busy waiting can be avoided by setting two interrupts: (1) sleeping for time = `inter_send_time`, and (2) whenever ACKs are received. Before either of these two interrupts hit, no packets are sent or received and so there is no reason for CCA to change its rate, so it is okay to sleep until these interrupts as the `inter_send_time` does not become stale until these interrupts. Note, setting interrupts increases the delays caused in scheduling threads. This can increase the discrepancy between the actual packets sent vs. the packets the CCA expected to send. The pacing implementation may choose to transiently increase the sending rate to correct for sending opportunities missed due to scheduling delays.

API. The Linux kernel also does not provide a direct API to change the sending rate after packets are sent. The kernel only gives a callback on ACKs. Since ACKs can be delayed due to non-congestive delays, the CCA ends up setting a potentially stale sending rate for $O(D)$ time. This can be worked around by setting up timer interrupts or instrumenting a callback on packet send events. The genericCC API provides callbacks on both packet send and receive events.

B.6 Supplementary empirical evaluation

We empirically evaluate the synthesized CCAs across a variety of $\langle C, R, D, \beta \rangle$ parameters. Specifically, we explore combinations of $C \in \{24, 48, 96\}$ Mbps, $R \in \{20, 40, 80\}$ ms, $D = R$, and $\beta \in \{1/16, 1/8, 1/4, 1/2, 1, 2, 4, 8, 16\}$ `BDP`. Each tuple is a 60 seconds long "run". We discard the first 20 seconds of each run (to study steady-state behavior) and compute metrics over the remainder.



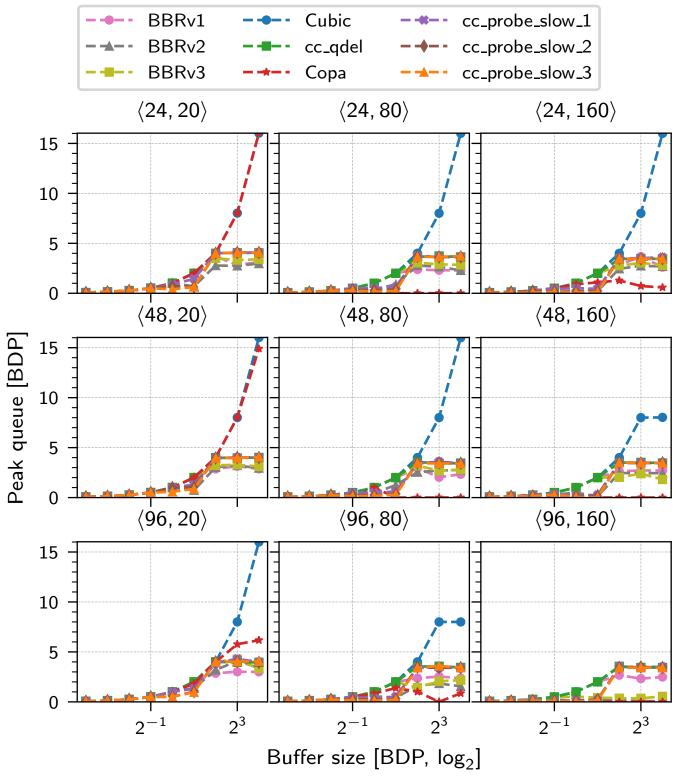


Figure B.3: Average utilization and maximum queue use with varying bandwidth, propagation delay and buffer sizes for a link with random ACK aggregation. The tuple at the top of each subplot shows $\langle C \text{ Mbps}, R \text{ ms}\rangle$.

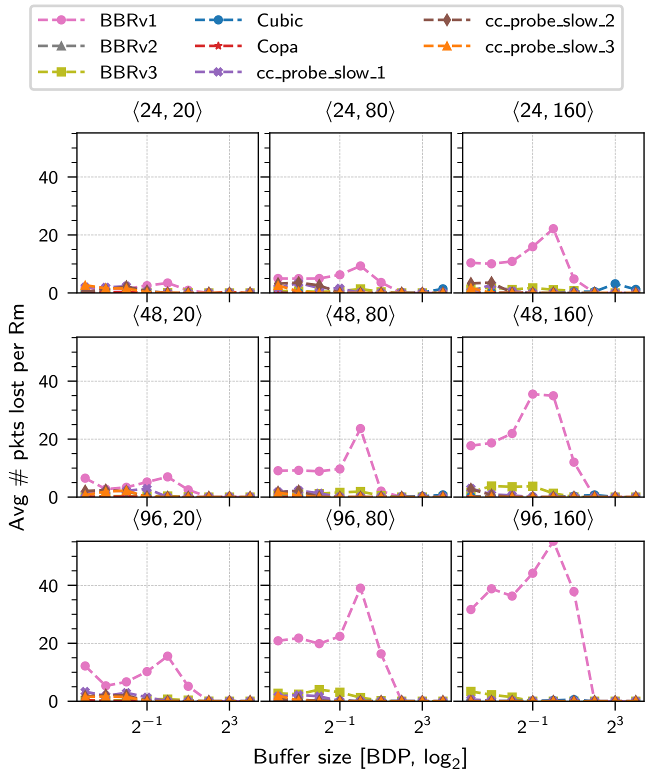


Figure B.4: Packet loss with varying bandwidth, propagation delay and buffer sizes for a link with random ACK aggregation. The tuple at the top of each subplot shows $\langle C \text{ Mbps}, R \text{ ms}\rangle$. We compute number of packets lost in every R long interval and take average over all the intervals. We omit `cc_qdel`, it causes $O($`BDP`$)$ losses with a higher constant factor than BBRv1 (Fig. 4.6), and ends up skewing the graphs.

To emulate jitter, we inject up to R seconds of *random* ACK aggregation with a fixed average link rate. We sample `agg_delay` $\in [0, D = R)$ uniformly randomly. We serve a batch of $C\cdot$ `agg_delay` bytes after waiting for `agg_delay` seconds, ensuring an average bandwidth of C. We sample `agg_delay` after every batch.

Fig. B.3 and Fig. B.4 show the utilization, delay, and loss metrics for each run. The synthesized CCAs are withing their proof bounds in each run. In some of the runs, `cc_probe_slow`'s utilization is less than 50% (expected steady-state utilization in §4.2.2.1), this is because in those runs, it has not reached steady-state at the end of the run.

Note, BBRv2 and BBRv3 are not robust against adversarial jitter. On short buffers, they do not cause loss in steady-state. This is needed to ensure a lower bound on utilization on CBR-delay (**Step 1.** of Theorem 4.2.1), because otherwise their observations could be explained by an arbitrary large link rate. We believe this is why they get lower utilization when buffer is small in Fig. B.3.

Convergence time. We study convergence time in Fig. B.6 and Fig. B.7. We show a run with $R = 80$ ms. For increasing link rate, we double the link rate every 20 seconds starting at 24 Mbps. For decreasing link rate, we halve the link rate every 20 seconds starting at 96

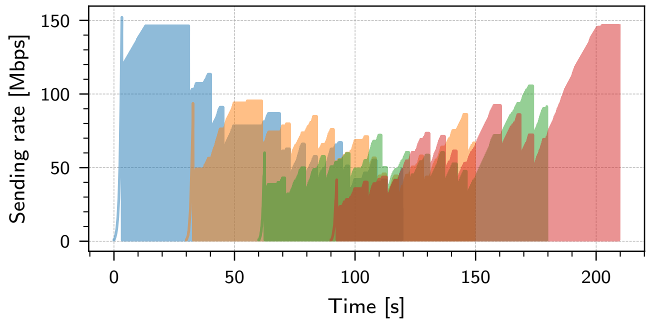


Figure B.5: With multiple flows, `cc_probe_slow` is able to fairly share the available bandwidth. Every 30 seconds, we start a new flow for a total of 4 flows shown by the different colors. We use translucency to show the sending rates when the graphs overlap.

Mbps. In both cases we set the buffer size as the BDP when link (wire) rate is 96 Mbps. I.e., $\beta = \texttt{BDP} = 96$ Mbps $\cdot$ 80 ms $\approx$ 624 packets (each packet sends 1538 bytes on the wire in our setup).

`cc_probe_slow` meets its convergence time bounds. It converges additively when link rate increases and exponentially fast when link rate decreases (due to belief timeouts).

BBRv2 and BBRv3 have low loss on average, but incur large loss events, i.e., $O(\texttt{BDP})$ whenever link rate increases, and they start probing exponentially fast. This happens at the 20 second and 40 second marks when the link rate increases (not shown).

To synthesize a CCA that follows the tradeoff choice made by BBRv2 and BBRv3, we would need to remove the under-specification in our synthesis invariant. We want to be able to cause large losses when we are in state II (i.e., the link rate changed, and we want to converge), but do not want to cause large losses in state IV (when the link rate has not changed, but we just want to check if it may have increased after a belief timeout). Because of under-specification we cannot distinguish between these scenarios during synthesis. We leave this for future work.

We can manually design such a CCA using the version of `cc_probe_slow` parametrized by $f(.)$ and T^* (§4.2.2.1). We can adapt the loss allowance $f(.)$ depending on if the CCA is in steady-state (causing periodic small losses), or if the CCA needs to converge (is not causing losses). This can also be done in a fashion similar to BIC [119] and Cubic [59], i.e., loss allowance adapts depending on how much the link rate changed. We leave this exploration for future work.

Fairness. Currently, our formal theoretical framework does not provide any guarantees in multi-flow scenarios, nor does it provide any predictions for the outcomes of multi-flow experiments. Nevertheless, we explore the fairness properties of the synthesized CCAs empirically.

Fig. B.5 shows a run with $C = 96$ Mbps, $R = 80$ ms, $\beta = 1/2$ BDP on an ideal link. We run 4 flows that share the same bottleneck. Each flow is started 30 seconds after the previous flow and lasts for $4 \times 30 = 120$ seconds. Results are qualitatively similar for other parameter and network model combinations. For each run (i.e., $\langle C, R, D, \beta \rangle$ combination mentioned at the beginning of the section), we compute the Jain's fairness index (JFI) for the average rates of the 4 flows between time 100 to 110 seconds (i.e., when all 4 flows are running). When $\beta \leq 2\texttt{BDP}$, the JFI for `cc_probe_slow` across the parameter combinations is 0.94 ± 0.06 (i.e., mean $\pm$ stddev) without

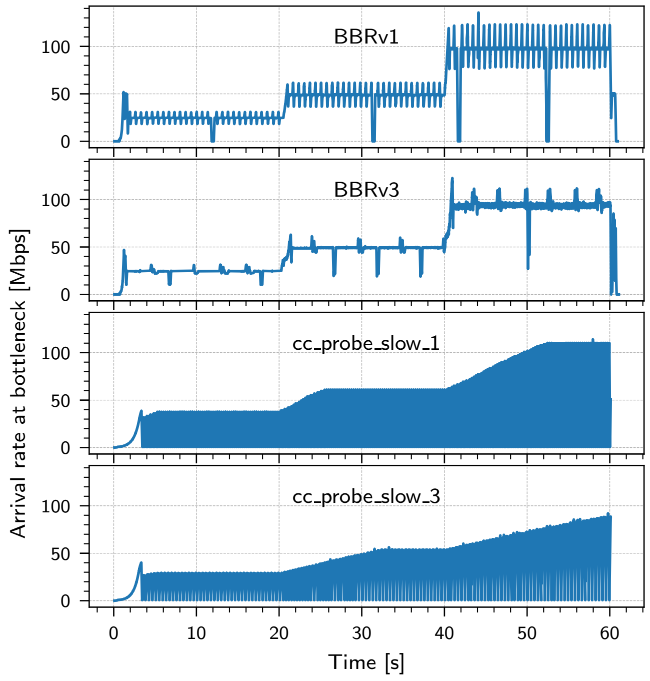


Figure B.6: Convergence time for increasing C. `cc_probe_slow_k` converges additively when the link rate increases. Increasing `k` increases utilization by a multiplicative factor (Fig. B.3) at the cost of increasing convergence time by a multiplicative factor.

jitter and 0.86 ± 0.11 with jitter. When $\beta > 2\texttt{BDP}$, `cc_probe_slow` effectively runs `cc_qdel` as there is no large loss event (§4.2.4). `cc_qdel` is unfair and gets JFI of 0.58 ± 0.24 without jitter and 0.43 ± 0.18 with jitter.

Even though `cc_probe_slow` was designed for single-flow scenarios, it is able to converge to a fair share of the available bandwidth. We believe this is because, to track changes in link rate, `cc_probe_slow` increases its effective rate ($C_{L,\lambda}$) additively and reduces its effective rate ($C_{L,\lambda}$) multiplicatively allowing it to reach a fair allocation similar to AIMD [37].

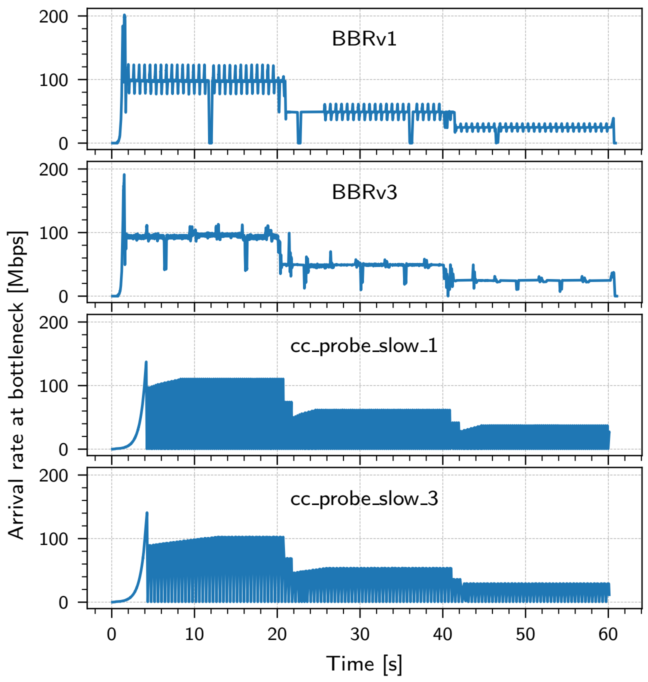


Figure B.7: Convergence time for decreasing C. `cc_probe_slow_k` converges exponentially fast when the link rate decreases.

Appendix C

Contracts tradeoff derivations

C.1 Tradeoffs

We show that metrics (M1) robustness and (M2) fairness are at odds with (M3) congestion and (M4) generality. M1 & M2 require the contract to be gradual, while M3 & M4 require the contract to be steeper. We derive these tradeoffs quantitatively. We assume the contract function `func` has domain $[S_{\min}, S_{\max}]$ and range $[C_{\min}, C_{\max}]$, i.e., $C_{\max} = \texttt{func}(S_{\min})$, and $C_{\min} = \texttt{func}(S_{\max})$.

Our strategy to derive the tradeoffs is as follows, choice of a metric puts constraints on the contract, and this in-turn puts constraints on other metrics. So we fix one metric and see what constraints it puts on other metrics.

The tradeoffs involving fairness depend on `fold`. We consider $\texttt{fold} \in \{\sum, \max, \min\}$. The tradeoffs exists for $\sum$. We do not get tradeoffs for max and min, i.e., we can achieve max-min fairness independent of requirements on robustness/generality. We also show derivation steps for arbitrary `fold`, if future work wants to consider other statistics that accumulate differently across hops.

C.1.1 M1 vs. M3: robustness vs. congestion growth

Say we want the robustness error factor to be at most $\epsilon_r > 1$, then we compute a lower bound on $\texttt{growth}(n)$. From the definition of robustness error factor (Eq. 6.1.1):

$$\forall s. \qquad \frac{\texttt{func}(s)}{\texttt{func}(s + \delta s)} \leq \epsilon_r$$

Picking $s = S_{\min}$,

$$\implies \qquad \frac{\texttt{func}(S_{\min})}{\texttt{func}(S_{\min} + \delta s)} \leq \epsilon_r$$

$$\begin{aligned} \implies \qquad C_{\max} = \texttt{func}(S_{\min}) &\leq \epsilon_r * \texttt{func}(S_{\min} + \delta s) \\ &\leq \epsilon_r * \epsilon_r * \texttt{func}(S_{\min} + 2 * \delta s) \\ &\leq \epsilon_r^k * \texttt{func}(S_{\min} + k * \delta s) \end{aligned}$$

$$
\begin{aligned}
&\implies & \frac{C_{\max}}{\epsilon_r^k} &\leq \texttt{func}(S_{\min} + k * \delta s) && \text{(C.1.1)}\\
&\implies & \texttt{func}^{-1}\left(\frac{C_{\max}}{\epsilon_r^k}\right) &\geq S_{\min} + k * \delta s && \text{(C.1.2)}\\
&\implies & \frac{\texttt{func}^{-1}\left(\frac{C_{\max}}{\epsilon_r^k}\right)}{\texttt{func}^{-1}(C_{\max})} &\geq \frac{S_{\min} + k * \delta s}{S_{\min}} && \\
&\implies & \texttt{growth}(\epsilon_r^k) \geq \frac{\texttt{func}^{-1}\left(\frac{C_{\max}}{\epsilon_r^k}\right)}{\texttt{func}^{-1}(C_{\max})} &\geq \frac{S_{\min} + k * \delta s}{S_{\min}} && \text{(C.1.3)}\\
&\implies & \texttt{growth}(n) \geq \frac{\texttt{func}^{-1}\left(\frac{C_{\max}}{n}\right)}{\texttt{func}^{-1}(C_{\max})} &\geq \frac{S_{\min} + \frac{\log(n)}{\log(\epsilon_r)} * \delta s}{S_{\min}} && \text{(C.1.4)}
\end{aligned}
$$

Note, we get Eq. C.1.2 by taking $\texttt{func}^{-1}$ on both sides, the inequality flips as $\texttt{func}$ and $\texttt{func}^{-1}$ are monotonically decreasing. We get the left inequality in Eq. C.1.3 from the definition of **growth** (Eq. 6.1.6). Eq. C.1.4 shows that lower error factor implies higher signal growth. The inequalities become equalities when (by substituting $S_{\min} + k * \delta s$ by s in Eq. C.1.1):

$$\texttt{func}(s) = C_{\max}\epsilon_r^{\frac{s - S_{\min}}{\delta s}} \tag{C.1.5}$$

This is same as the exponential contract from [20].

C.1.2 M1 vs. M4: robustness vs generality

We compute an upper bound on $\frac{C_{\max}}{C_{\min}}$ given we want the error factor to be at most $\epsilon_r > 1$. We start from Eq. C.1.1, and substitute $S_{\min} + k * \delta s$ by $S_{\max}$:

$$
\begin{aligned}
\frac{C_{\max}}{\epsilon_r^k} &\leq \texttt{func}(S_{\min} + k * \delta s)\\
\implies C_{\max}\epsilon_r^{-\frac{S_{\max} - S_{\min}}{\delta s}} &\leq \texttt{func}(S_{\max}) = C_{\min}\\
\implies \qquad \frac{C_{\max}}{C_{\min}} &\leq \epsilon_r^{\frac{S_{\max} - S_{\min}}{\delta s}}
\end{aligned}
$$

Lower the robustness error, lower is the range of bandwidths we can support. The inequality becomes equality for the exponential contract.

C.1.3 M2 vs. M3: fairness vs. congestion growth

We derive a lower bound on $\texttt{growth}(n)$ given that we want the throughput ratio in parking lot for k hops to be $\texttt{ratio}^\star(k)$. For the parking lot ratio to be $\texttt{ratio}^\star(k)$, we need:

$$\forall k. \quad \max_s \frac{\texttt{func}(s)}{\texttt{func}(\texttt{fold}(s, s, \ldots s))} = \texttt{ratio}^\star(k)$$

Say, the maximum of LHS is achieved when the link capacity in the parking lot is $C = C^\star(k)$. $\texttt{ratio}^\star(.)$ and $C^\star(.)$ are functions of k. For brevity, we drop the k, and refer to them as $r^\star$ and $C^\star$ respectively. The statements are true for all positive k.

Instead of directly computing a constraint on $\texttt{growth}(n)$, we first derive a constraint on $\texttt{growth}_{C^\star}(n)$. This eventually constrains $\texttt{growth}(n)$. Where, $\texttt{growth}_C(n)$ is defined as $\frac{\texttt{func}^{-1}(C/n)}{\texttt{func}^{-1}(C)}$. I.e., $\texttt{growth}(n) = \max_C \texttt{growth}_C(n)$ (from Eq. 6.1.6). Note, $\texttt{growth}_C(1) = 1$ from this definition. We will use this later.

Consider the execution of the contract (CCA) on a parking lot topology with k hops and link capacity $C^\star$. In steady-state, we have:

$$r_0 + r_k = C^\star \quad \text{same as Eq. 6.1.2, and,}$$
$$\frac{r_0}{r_k} = r^\star \quad \text{from the definition of } r^\star \text{ above.}$$

On solving these, we get:

$$r_0 = C^\star/(r^\star + 1) \quad \text{and,} \quad r_k = C^\star r^\star/(r^\star + 1)$$

Say the aggregate statistic seen by flow f_i in the parking lot is $s_i = \texttt{func}^{-1}(r_i)$. Then $\texttt{growth}_{C^\star}(C^\star/r_i) = \frac{\texttt{func}^{-1}(C^\star/(C^\star/r_i))}{\texttt{func}^{-1}(C^\star)} = \frac{s_i}{\texttt{func}^{-1}(C^\star)}$. We define $s^\star = \texttt{func}^{-1}(C^\star)$, and substitute i by 0. On rearranging, we get:

$$s_0 = \texttt{func}^{-1}(C^\star) \cdot \texttt{growth}_{C^\star}(C^\star/r_0) = s^\star \cdot \texttt{growth}_{C^\star}(r^\star + 1)$$
$$\text{Likewise, } s_k = s^\star \cdot \texttt{growth}_{C^\star}(1 + 1/r^\star)$$

In steady-state:

$$s_0 = \texttt{fold}(s_k, s_k, \ldots, k \text{ times}) \quad \text{same as Eq. 6.1.4, on substituting } s_0 \text{ and } s_k \text{ just computed, we get}$$
$$\implies s^\star \cdot \texttt{growth}_{C^\star}(r^\star + 1) = \texttt{fold}(s^\star \cdot \texttt{growth}_{C^\star}(1 + 1/r^\star), \ldots, k \text{ times})$$

If multiplication distributes over accumulation, this is true for $\texttt{fold} \in \{\sum, \max, \min\}$, then:

$$\texttt{growth}_{C^\star}(r^\star + 1) = \texttt{fold}(\texttt{growth}_{C^\star}(1 + 1/r^\star), \ldots, k \text{ times})$$

We consider under different choices of $\texttt{fold}$, what constraint $\texttt{ratio}^\star$ puts on $\texttt{growth}_{C^\star}$. For $\texttt{fold}$ as max or min. We can vacuously meet this constraint for $\texttt{ratio}^\star = 1$ (corresponding to max-min fairness), and there is no tradeoff. For $\texttt{fold} = \sum$, the constraint becomes:

$$\texttt{growth}_{C^\star}(r^\star + 1) = k \cdot \texttt{growth}_{C^\star}(1 + 1/r^\star)$$

Let $n = r^\star + 1 = \texttt{ratio}^\star(k) + 1$, i.e., $k = \texttt{ratio}^{\star -1}(n - 1)$, where $\texttt{ratio}^{\star -1}$ is the inverse of $\texttt{ratio}^\star$ then:

$$\texttt{growth}_{C^\star}(n) = \texttt{ratio}^{\star -1}(n - 1) \cdot \texttt{growth}_{C^\star}(1 + 1/(n - 1))$$

Since, $\texttt{func}$ is decreasing, $\texttt{growth}_C$ is increasing for all C. So $\texttt{growth}_{C^\star}(1 + 1/(n-1)) \geq \texttt{growth}_{C^\star}(1) = 1$. So, we get:

$$\texttt{growth}_{C^\star}(n) \geq \texttt{ratio}^{\star-1}(n-1)$$

If we want better fairness, we need $\texttt{ratio}^\star$ to be slow growing, then $\texttt{ratio}^{\star-1}$ is fast-growing, and so $\texttt{growth}_{C^\star}$ is fast-growing, and so congestion growth $\texttt{growth} \geq \texttt{growth}_{C^\star}$ needs to be fast-growing.

For proportional fairness, $\texttt{ratio}^\star(k) = k$, and so $\texttt{growth}(n) = O(n)$, this is met by Vegas. For α-fairness, $\texttt{ratio}^\star(k) = \sqrt[\alpha]{k}$, and so $\texttt{growth}(n) = O(n^\alpha)$, this is met by the contract $\texttt{func}(s) = \frac{1}{r^\alpha}$.

C.1.4 M2 vs. M4: fairness vs. generality

We compute an upper bound on $\frac{C_{\max}}{C_{\min}}$ given we want the throughput ratio in parking lot for k hops to be at most $\texttt{ratio}^\star(k)$. For the throughput ratio to be at most $\texttt{ratio}^\star(k)$, we need:

$$\forall s, k. \quad \frac{\texttt{func}(s)}{\texttt{func}(\texttt{fold}(s, s, \ldots s))} \leq \texttt{ratio}^\star(k)$$

The derivation beyond this depends on $\texttt{fold}$. For $\texttt{fold} \in \{\max, \min\}$, the above constraint is vacuously true and there is not tradeoff. For $\texttt{fold} = \sum$, we need:

$$\forall s, k. \texttt{func}(s) \leq \texttt{ratio}^\star(k) \cdot \texttt{func}(k * s)$$

Picking $s = S_{\min}$, and $k = \frac{S_{\max}}{S_{\min}}$, we get:

$$\texttt{func}(S_{\min}) \leq \texttt{ratio}^\star\left(\frac{S_{\max}}{S_{\min}}\right) \cdot \texttt{func}(S_{\max})$$
$$\implies \frac{C_{\max}}{C_{\min}} \leq \texttt{ratio}^\star\left(\frac{S_{\max}}{S_{\min}}\right)$$

Better fairness implies smaller ratio $\texttt{ratio}^\star(.)$, implies smaller the range of supported bandwidths.

If we want fairness to be at least as good as proportional fairness, we need $\texttt{ratio}^\star(k) \leq k$, or:

$$\forall s, k. \quad \texttt{func}(s) \leq \texttt{ratio}^\star(k) \cdot \texttt{func}(k * s) \leq k \cdot \texttt{func}(k * s)$$

Substituting $k = S_{\min}/s$, we get:

$$\texttt{func}(s) \leq \frac{S_{\min} C_{\max}}{s}$$

Equality occurs when $\texttt{func}(s) = \frac{S_{\min} C_{\max}}{s}$, which is the same contract as Vegas.

Likewise, for α-fairness, $\texttt{ratio}^\star(k) = \sqrt[\alpha]{k}$, and so $\frac{C_{\max}}{C_{\min}} \leq \sqrt[\alpha]{\frac{S_{\max}}{S_{\min}}}$. The equality occurs for the contract: $\texttt{func}(s) = \frac{1}{r^\alpha}$.

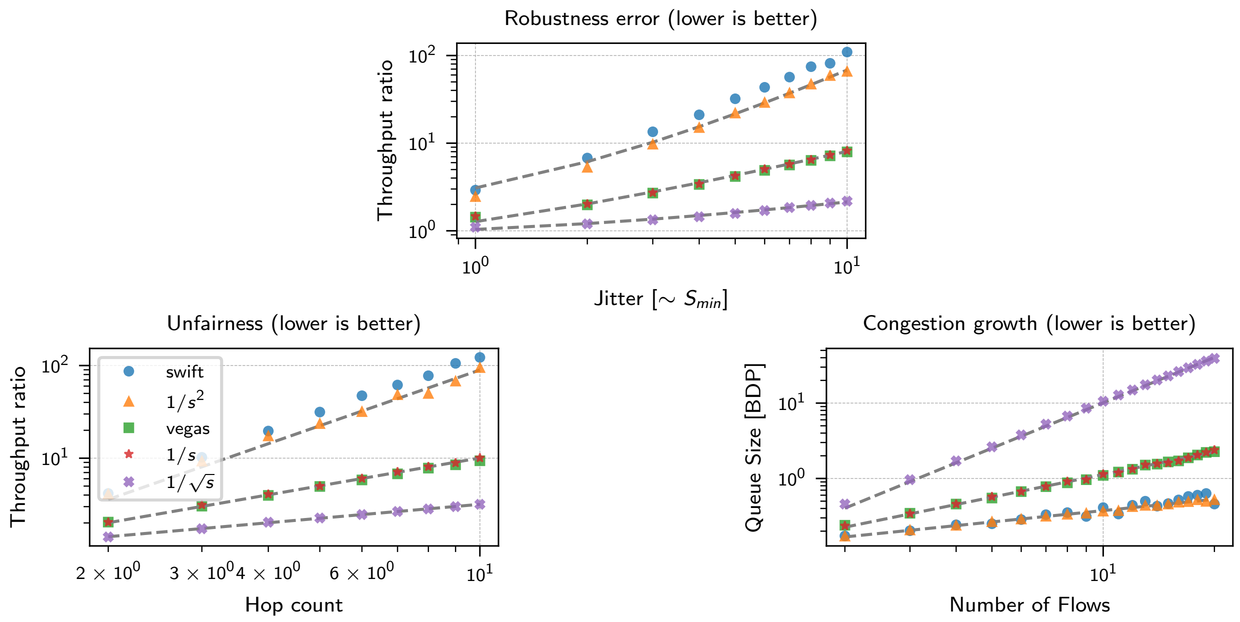


Figure C.1: Contracts-based performance metric estimates match packet-level simulation. The markers show empirical data and gray lines show performance estimated by the contracts.

C.2 Validation

Fig. C.1 shows the simulation results on a log-log scale plot for visual clarity.

Appendix D

FRCC design and analysis details

D.1 Artifacts

Abstract. We release (1) FRCC's Linux kernel module implementation, (2) code for theoretical analysis and proofs, and (3) scripts to run and plot empirical experiments.

Scope. We hope our artifact allows others to verify our theoretical and empirical claims as well as build upon our work.

For the theoretical analysis and proofs, we include code to reproduce Figures 7.12, 7.13, 7.16 and Theorems 7.5.1, 7.5.2.

For the empirical experiments, we include code to reproduce Figures 2.1, 2.2, 7.17, 7.18, 7.19, 7.21. We do not include code to reproduce experiments with BBRv3 and the parking lot topology (i.e., Figure 7.20) as we have separate VMs for these experiments.

Contents. Our artifact is organized across three repositories:

(1) `https://github.com/108anup/frcc/tree/main`. This is the main repository that includes the other two repositories as submodules. It contains documentation and the code for reproducing FRCC's theoretical analysis and proofs.
(2) `https://github.com/108anup/frcc_kernel/tree/main`. This contains FRCC's Linux kernel module implementation.
(3) `https://github.com/108anup/cc_bench/tree/main`. This contains the scripts for running and plotting empirical experiments.

The main repository (`https://github.com/108anup/frcc/tree/main`) contains a detailed `README.md` documenting dependencies, setup, known issues, and commands for reproducing all the experiments.

Hosting. At the time of writing, the main repository (`https://github.com/108anup/frcc/tree/main`) is at commit `4cedada281fb2d7df04f6dad476a30e44be6fd19` on the `main` branch. This commit includes the kernel and benchmarking repositories as submodules.

Requirements. The artifact does not require specialized hardware. Since our artifact modifies the kernel through the kernel module, we recommend running experiments in a virtual machine or on a dedicated host. All evaluations in the paper were performed on an x86_64 server-class machine with 64 cores and 256 GB RAM, allowing multiple concurrent experiments. Users with fewer resources can scale down the degree of concurrency (see `README.md` for details).

D.2 Design details

D.2.1 Probe size derivation

Under our probe size ($E = \gamma \widehat{N_T} \widehat{\texttt{rtput}} \mathcal{D}$), we derive the impact of ΔR noise in $\widehat{\Delta d}$ on our $\widehat{C}$ estimate and consequently its impact on deciphering which of N_C or $\widehat{N_T}$ is bigger.

We used this derivation in inverse (from bottom to top) to derive the probe size in the first place.

$$\widehat{C} = \frac{E}{\widehat{\Delta d}} = \frac{E}{\Delta d \pm \Delta R} = \frac{CE}{E \pm C\Delta R}$$

We get the last equality by substituting $\Delta d = E/C$. Substituting our probe size,

$$\widehat{C} = \frac{C\gamma \widehat{N_T} \widehat{\texttt{rtput}} \mathcal{D}}{\gamma \widehat{N_T} \widehat{\texttt{rtput}} \mathcal{D} \pm C\Delta R}$$

Substituting $\widehat{C}$ into $\widehat{N_C}$ computation,

$$\widehat{N_C} = \frac{\widehat{C}}{\widehat{\texttt{rtput}}} = \frac{C\gamma \widehat{N_T} \mathcal{D}}{\gamma \widehat{N_T} \widehat{\texttt{rtput}} \mathcal{D} \pm C\Delta R}$$

Substituting $\widehat{N_C}$ in,

$$\frac{\widehat{N_T}}{\widehat{N_C}} = \frac{\gamma \widehat{N_T} \widehat{\texttt{rtput}} \mathcal{D} \pm C\Delta R}{C\gamma \mathcal{D}} = \frac{\widehat{N_T} \widehat{\texttt{rtput}}}{C} \pm \frac{\Delta R}{\gamma \mathcal{D}}$$

Substituting $C = N_C \cdot \texttt{rtput}$ (definition of N_C (§ D.4)),

$$\frac{\widehat{N_T}}{\widehat{N_C}} = \frac{\widehat{\texttt{rtput}}}{\texttt{rtput}} \frac{\widehat{N_T}}{N_C} \pm \frac{\Delta R}{\gamma \mathcal{D}}$$

Note, we define the "true" quantities Δd, N_C, `rtput` based on the fluid model and define ΔR as error in observations in jittery link relative to fluid model (§ 7.5.1, § D.3.1). This is why it is okay to use the substitutions $\Delta d = E/C$ and $C = N_C \cdot \texttt{rtput}$.

D.2.2 Probability of probe collisions

If there are kN slots and N flows probing. The probability of a particular flow *not* colliding is:

$$\left(\frac{kN-1}{kN}\right)^{N-1} \to e^{-1/k} \text{ as } N \to \infty$$

Hence, the probability of a particular flow colliding is bounded by $1 - e^{-1/k}$.

D.3 Proof details

D.3.1 Preliminaries

Fluid reference execution. Say flow f_i has congestion window $\texttt{cwnd}_\texttt{i}$, round-trip propagation delay R_i. We look at the execution of these choices on a fluid model of the network with link capacity C and infinite buffer.

Under this execution, below we define "reference" state variables: (1) "current flow count" (from the point of view flow i) (N_{C_i}), and (2) "target flow count" (N_T). Recall, the current flow count is inverse of the fraction of link the flow consumes, and target flow count is based on the contract's target delay function, i.e., $N_T = \texttt{Contract}_\texttt{Func}(\texttt{delay}) = \texttt{delay}/\theta$.

$$N_T \text{ solves } \sum_i \frac{\texttt{cwnd}_\texttt{i}}{\texttt{rRTT}_\texttt{i}} = \sum_i \frac{\texttt{cwnd}_\texttt{i}}{N_T\theta + R_i} = C \tag{D.3.1}$$

$$N_{C_i} = \frac{\sum_j \texttt{rtput}_\texttt{j}}{\texttt{rtput}_\texttt{i}} = \frac{C}{\texttt{rtput}_\texttt{i}} \text{, with, } \texttt{rtput}_\texttt{j} = \frac{\texttt{cwnd}_\texttt{j}}{\texttt{rRTT}_\texttt{j}} \tag{D.3.2}$$

$$\text{For } R_i = R,\ N_T = \frac{\sum_i \texttt{cwnd}_\texttt{i} - CR}{C\theta} \tag{D.3.3}$$

We also define "$\texttt{rRTT}_\texttt{i}$" ($= N_T\theta + R_i$) and "$\texttt{rtput}_\texttt{i}$" ($= \texttt{cwnd}_\texttt{i}/\texttt{rRTT}_\texttt{i}$) as the "reference" RTT and throughput values.

Summary of notation. We track four variables: $\texttt{rRTT}_\texttt{i}$, $\texttt{rtput}_\texttt{i}$, N_{C_i}, N_T (Table D.2).

For each of these variables there are: (1) values in the reference execution, and (2) estimates maintained by the FRCC. Further, all these quantities change as rounds progress. We use the following notation to differentiate between these uses.

- Symbols without any annotation or with $[\texttt{r}]$ suffix (e.g., $\texttt{rRTT}_\texttt{i}$ or $\texttt{rRTT}_\texttt{i}[\texttt{r}]$) denote the state variables in round $\texttt{r}$ of the reference execution. We also use symbols without $[\texttt{r}]$ suffix to refer to state after a probe (in addition to at boundaries of rounds).
- Symbols with hats (e.g., $\widehat{\texttt{rRTT}_\texttt{i}}[\texttt{r}]$) denote the estimates or state variables maintained by FRCC.
- As a shorthand we drop the $[\texttt{r}]$ suffix when clear from context, e.g., we use $\widehat{N_{C_i}}[\texttt{r}_\texttt{F}] = \widehat{N_{C_i}}[\texttt{r+1}]$ and $\widehat{N_{C_i}}$ to denote state in the next round and current round respectively.

D.3.2 Assumptions

Since our analyses are at the level of rounds, we make several assumptions about the slot-level behavior.

> "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise." — Edsger Dijkstra

Assumption 1. RTprop probe works. We assume that our RTprop probing mechanism works; i.e., flows can simultaneously drain queues and get an RTT measurement without queueing delays. This may still have errors due to (E1/E2). Note, there is no error due to capacity probes (E3) because FRCC stops any ongoing capacity probes when its time for an RTprop probe.

Assumption 2. Slots align. Slots of flows are the same duration. This is true for ideal link with equal RTprops (§ D.3.3). On jittery link, slot durations may differ due to differences in the RTTs of the flows. Further, with different propagation delays also slot durations differ. Despite this our analyses yields interesting and empirically valid insights (§ 7.5.3, § 7.5.5, § 7.6).

Assumption 3. Rounds align. Between two `cwnd` updates of a flow, all other flows have a `cwnd` update. For the purpose of analysis, we define rounds at the boundaries of such intervals where each flow probes and updates `cwnd` exactly once. In reality, a flow may probe twice between the probes of another flow, e.g., a flow may probe at the end of one round and at the start of the next round, without other flows updating their `cwnd`s in between. Note, when slot durations are the same, there can be at most two `cwnd` updates of a flow between other flow's `cwnd` updates, as a full round needs to elapse between the first and third `cwnd` update of a flow, during which all other flows will probe at least once.

Assumption 4. Synchronous updates. (We relax this for § 7.5.4) All `cwnd` updates occur synchronously. I.e., in a round, all flows first update their estimates and then use these to update their `cwnd`. In reality, the `cwnd` updates may happen sequentially, i.e., one flow probes and updates their `cwnd`, resulting a new reference state, then another flow estimates the new state, and then updates its `cwnd`.

Assumption 5. At least one non-colliding cruise before probe. Before every capacity probe, there is at least one cruise measurement that does not collide with a probe of some other flow. This ensures we get an estimate of $\widehat{\texttt{rRTT}_\texttt{i}}$ and $\widehat{\texttt{rtput}_\texttt{i}}$ that is not biased by the probe of other flows (i.e., no impact of (E3)). Note, our design ensures that there is at least one cruise slot before a probe. Further, over a round there will be many cruise slots that do not collide with probe (from the pigeon-hole principle, there are kN_G in steady-state and only N_G flows or probe slots). But all the measurements in the cruise slot before a flow's probe may collide with other probes, we assume that this does not happen.

Assumption 6. Use alternate `cwnd` update. Only for Theorem 7.5.2, we use the alternate update (line 6) instead of the main update (line 7).

Assumption 7. Known RTprop and one-sided error in RTT, throughput. Only for Theorem 7.5.2, we assume that there is no error due to (E1/E2) in RTprop, i.e., $\forall i, \widehat{R_i} = R_i$. We also assume that the error due to (E1/E2) in RTT and throughput measurements is one-sided, i.e., RTTs only increase due to noise and throughputs only decrease. This does not hold in reality. Fig. 7.8 shows that RTTs oscillate and thus can be both more or less than the fluid reference. Likewise, if one flow's throughput drops because of RTT inflation then the other flows' throughput increases with a closed-loop system.

We believe these assumptions are not needed for Theorem 7.5.2. We made them historically and hope to remove them. We designed our estimates so that errors vanish with increasing delay and delay variations, and the polarity of the errors has little consequence. For instance, for the capacity estimate, the excess delay is a difference of two potentially erroneous RTT measurements, and we model both positive and negative errors in excess delay. Removing this assumption may change the guidance on design parameter choices. However, empirically, the current design parameters work fine (§ 7.6).

D.3.3 Ideal link

D.3.3.1 Bounds on estimates

Lemma D.3.1. *For an ideal link, with equal propagation delays, when the link is utilized, and there are no* `cwnd` *changes, there are no RTT variations due to (E2), i.e., effects of packet interleaving or differences in feedback delay.*

Proof. When the link is utilized, every $1/C$ seconds (i.e., a transmission delay), a packet is dequeued by the bottleneck link, due to the ACK-clock, this dequeue triggers an enqueue exactly R seconds later (as all $R_i = R$). As a result every $1/C$ seconds a packet is enqueued and dequeued, and there is no variation in queuing delay or RTT. Note, this is not true when RTprops are different, which is why we see the oscillations in Fig. 7.7. $\square$

It suffices for us to look at the execution when the link is already utilized, as FRCC ensures that the bottleneck link is utilized exponentially fast. This is because there is at least one flow in the network, and FRCC increases `cwnd` until, the minimum RTT is $1 * \theta$ more than R. As a result, there is always a queuing delay, and link is utilized. So we can use Lemma D.3.1 in deriving the state transition function. This derivation then works for states that start with a utilized link.

Due to Lemma D.3.1, since all RTTs are the same, they are equal to $\texttt{rRTT}_\texttt{i}$, and so our RTT estimates are correct: $\widehat{\texttt{rRTT}_\texttt{i}} = \texttt{rRTT}_\texttt{i}$, $\widehat{\Delta d_i} = \Delta d_i$, and $\widehat{R_i} = R_i = R$ (equal RTprops, and draining occurs synchronously (**Assumption 1.**)).

Further, since we measure throughput over packet-timed RTT the $\widehat{\texttt{rtput}}$ is `cwnd`/RTT which is same as the `rtput` =`cwnd`/`rRTT` on the fluid reference (as all RTTs are equal to `rRTT` due to Lemma D.3.1).

Note, we used the fact that updates are synchronous in computing these bounds, i.e., the estimates are not impacted by other flows' `cwnd` updates within the round **Assumption 4.** (and consequently **Assumption 3.** and **Assumption 2.**). We also assumed that probes do not collide and that there is a cruise slot before the probe that did not collide with any probe (**Assumption 5.**).

As a result, $\widehat{N_{T_i}} = N_T$, $\widehat{C_i} = C$, $\widehat{N_{C_i}} = N_{C_i}$, i.e., there is no error in estimation compared to reference execution.

D.3.3.2 Next state, and state transition function

Now, we derive how the state variables evolve over rounds. We re-write the `cwnd` update equations (§ 7.4.2) given that the estimates are the same as the reference state, and then consider the reference state after all flows have updated their `cwnd`. Note, in reality, flows may update their `cwnd` one by one (but we assume updates are synchronous **Assumption 4.**).

Reference state after a round is (Eq. D.3.3):

$$N_T[\mathtt{r_F}] = \frac{\sum_i \mathtt{cwnd_i}[\mathtt{r_F}] - CR}{C\theta} \tag{D.3.4}$$

$$N_{C_i}[\mathtt{r_F}] = \frac{\sum_i \mathtt{cwnd_i}[\mathtt{r_F}] - CR}{C\theta} \tag{D.3.5}$$

Substituting next `cwnd`s, (i.e., $\mathtt{cwnd_i}[\mathtt{r_F}] = (1-\alpha)\cdot \mathtt{cwnd_i}[\mathtt{r}] + \alpha \cdot \mathtt{target\_cwnd_i}$, where $\mathtt{target\_cwnd_i} = \mathtt{cwnd_i} \cdot N_{C_i}/N_T$), i.e., the alternate `cwnd` update (line 6) and simplifying:

$$N_{C_i}[\mathtt{r_F}] = \frac{N_{C_i}((1-\alpha)N_T + \alpha N_G)}{(1-\alpha)N_T + \alpha N_{C_i}} \tag{D.3.6}$$

$$N_T[\mathtt{r_F}] = (1-\alpha)N_T + \alpha N_G + \frac{\alpha R}{\theta}\left(\frac{N_G}{N_T} - 1\right) \tag{D.3.7}$$

Note, for the proof on ideal link, we do not need the clamps (δ_H, δ_L). So we just use: $\mathtt{target\_cwnd_i} = \mathtt{cwnd_i} \cdot N_{C_i}/N_T$. The clamps are needed in the jittery link execution to bound the impact of potentially wrong `cwnd` updates when estimates have errors. Further, in the simplification, we also substituted $\sum_{i=1}^{N_G} \frac{1}{N_{C_i}} = 1$ (follows from the definition of N_{C_i} in Eq. D.3.2). We get the N_G from $\sum_{i=1}^{N_G} 1$.

D.3.3.3 State trajectories (alternate `cwnd` update line 7)

We now follow these state update equations to see what happens at round large enough (time large enough). This is visualized in Fig. 7.12.

Lemma D.3.2. *$N_T \longrightarrow N_G$ exponentially fast (as long as $N_T\theta > R$).*

Proof. Rewriting Eq. D.3.7, we get:

$$\frac{N_T[\mathtt{r}+1] - N_G}{N_T[\mathtt{r}] - N_G} = (1-\alpha) + \alpha\frac{-R}{\theta N_T[\mathtt{r}]}$$

Based on this equation, if the magnitude of RHS is less than 1 (say $0 < |\text{RHS}| \leq \beta < 1$), then the gap between N_T and N_G decays exponentially fast. Here, $|x|$ is the absolute value (magnitude) of x.

$$\left|\frac{N_T[\mathtt{r}+1] - N_G}{N_T[\mathtt{r}] - N_G}\right| = \left|(1-\alpha) + \alpha\frac{-R}{\theta N_T[\mathtt{r}]}\right| \leq \beta$$

Substituting $|N_T[\mathtt{r}] - N_G|$ by $x[\mathtt{r}]$,

$$\begin{aligned}
&x[\mathtt{r}+1] \le x[\mathtt{r}]\beta \\
\implies &x[\mathtt{r}+k] \le x[\mathtt{r}]\beta^k \\
\implies &x \to 0, \text{ or } N_T \to N_G \text{ exponentially fast, as } k \to \infty
\end{aligned}$$

The RHS is average of 1 and a negative number, so it is always less than 1. We just need to show that it is greater than -1. If $N_T\theta > R$, then RHS is > -1. If we set $\theta = 2R$ (we know R from **Assumption 1.**), then we just need $N_T > 1/2$.

Recall, N_T comes from queueing delay in the reference execution. Also recall that $\widehat{N_{C_i}} = N_{C_i} \ge 1$ ($N_{C_i} \ge 1$ from definition of N_{C_i} in Eq. D.3.2). Whenever $N_T < 1$, the `cwnd` for *all* flows increases as `target_cwnd` $=$ `cwnd`$\frac{N_{C_i}}{N_T}$ $=$ `cwnd`$\frac{(\ge 1)}{<1}$ $>$ `cwnd`. As a result, the queueing delay and in-turun N_T increase. Once $N_T > 1/2$, it will not decrease below half, as the distance between N_T and N_G always decreases (from the proof above) and N_G is always ≥ 1. □

LEMMA D.3.3. *Given $N_T = N_G$, $N_{C_i} \to N_G$ exponentially fast.*

Proof. Substituting $N_T = N_G$, and $x[\mathtt{r}] = \frac{N_G}{N_{C_i}[\mathtt{r}]}$ in Eq. D.3.6, we get:

$$\begin{aligned}
&\frac{N_G}{x[\mathtt{r}+1]} = \frac{N_G}{x[\mathtt{r}]} \frac{N_G}{(1-\alpha)N_G + \alpha\frac{N_G}{x[\mathtt{r}]}} \\
\implies &x[\mathtt{r}+1] = (1-\alpha)x[\mathtt{r}] + \alpha \\
\implies &x[\mathtt{r}+k] = (1-\alpha)^k x[\mathtt{r}] + (1-(1-\alpha)^k) \\
\implies &x \to 1, \text{ or } N_{C_i} \to N_G \text{ exponentially fast, as } k \to \infty
\end{aligned}$$

□

D.3.3.4 State trajectories (main `cwnd` update line 6)

While `target_cwnd`$_\mathtt{i}$ $=$ `cwnd`$_\mathtt{i}\cdot$ $\widehat{N_{C_i}}/\widehat{N_{T_i}}$ (line 7) was the alternate target `cwnd`, our main target `cwnd` is: `target_cwnd`$_\mathtt{i}$ $=$ `cwnd`$_\mathtt{i}\cdot$ $(R_i + \widehat{N_{C_i}}\theta)/(R_i + \widehat{N_{T_i}}\theta)$.

This update is more stable/robust than the original one. Specifically, this does not require $N_T\theta > R$.

Under the main `target_cwnd` choice , the state variables evolve as (by substituting the new `cwnd` in Eq. D.3.4 and Eq. D.3.5):

$$N_T[\mathtt{r}+1] = (1-\alpha)N_T + \alpha N_G \tag{D.3.8}$$

$$N_{C_i}[\mathtt{r}+1] = N_{C_i}\frac{R + \theta\left((1-\alpha)N_T + \alpha N_G\right)}{R + \theta\left((1-\alpha)N_T + \alpha N_{C_i}\right)} \tag{D.3.9}$$

LEMMA D.3.4. *$N_T \to N_G$ exponentially fast.*

Proof. From N_T state update equation (Eq. D.3.8):

$$N_T[\mathtt{r}+k] = (1-\alpha)^k N_T[\mathtt{r}] + (1-(1-\alpha)^k)N_G$$
$$\implies N_T \longrightarrow N_G \text{ exponentially fast, as } k \longrightarrow \infty$$

□

LEMMA D.3.5. *Given $N_T = N_G$, $N_{C_i} \longrightarrow N_G$ exponentially fast.*

Proof. In Eq. D.3.9, substituting $N_T = N_G$, and re-writing:

$$\frac{N_{C_i}[\mathtt{r}+1] - N_G}{N_{C_i} - N_G} = \frac{R + \theta(1-\alpha)N_G}{R + \theta\left((1-\alpha)N_G + \alpha N_{C_i}\right)}$$

Since $N_{C_i} \geq 1$ (from the definition of N_{C_i}, i.e., Eq. D.3.2), the RHS is < 1, also RHS > 0 (as all quantities are non-negative). Say $0 <$ RHS $\leq \beta < 1$, then:

$$\implies \frac{N_{C_i}[\mathtt{r}+1] - N_G}{N_{C_i} - N_G} \leq \beta < 1$$
$$\implies N_{C_i}[\mathtt{r}+k] - N_G \leq (N_{C_i}[\mathtt{r}] - N_G)\beta^k$$
$$\implies N_{C_i} \longrightarrow N_G \text{ exponentially fast, as } k \longrightarrow \infty \text{ (as } \beta^k \longrightarrow 0)$$

□

Note, since $\alpha = 1$, without clamps, FRCC actually converges N_T in 1 round with the main update and N_{C_i} in 1 round with the alternate update.

D.3.4 Jittery link

D.3.4.1 Bounds on estimates

We compute bounds on the estimates made by FRCC, assuming each RTT can increase by up to $\Delta R/2$ and throughput can decrease by γ_R due to (E1/E2) (**Assumption 7.**). As a result, the error in a RTT difference measurement is $\pm\Delta R$ (since $(R_1 \pm \Delta R/2) - (R_2 \pm \Delta R/2) = (R_1 - R_2) \pm \Delta R$).

In these estimates we do not incorporate impact of (E3). We assume there is a cruise measurement before probe that does not overlap with other probes (**Assumption 5.**), hence (E3) errors for cruise slot estimates ($\widehat{\mathtt{rtput}}$, $\widehat{\mathtt{rRTT}}$, $\widehat{R}$) get filtered out by max/min. The theorem statement also assumes probes do not collide so (E3) does not affect capacity estimate.

1. $\widehat{R} = R$. We have assumed that $\widehat{R}$ is correct (no error) (**Assumption 1.**, **Assumption 7.**).
2. $\widehat{\mathtt{rRTT_i}} \in [\mathtt{rRTT_i}, \mathtt{rRTT_i} + \Delta R/2]$.
3. $\widehat{\mathtt{rtput_i}} \in [\mathtt{rtput_i}/\gamma_R, \mathtt{rtput_i}]$. This implies that $\frac{C}{\widehat{\mathtt{rtput_i}}} \in [N_{C_i}, N_{C_i}\gamma_R]$ as $C/\mathtt{rtput_i} = N_{C_i}$. We will use this later.
4. $\widehat{\Delta d_i} \in [E_i/(C+\Delta R), E_i/(C-\Delta R)]$. (For $\widehat{C}_i$ in $\widehat{N_{C_i}}$).

5. $\widehat{N_{T_i}} = (\widehat{\texttt{rRTT}_\texttt{i}} - R_i)/\theta \in [N_T, N_T + \Delta R/2\theta]$
6. Error in current flow count estimate:

$$\widehat{N_{C_i}} = \frac{\widehat{C_i}}{\widehat{\texttt{rtput}_\texttt{i}}} = \frac{E_i}{\widehat{\Delta d_i}}\frac{1}{\widehat{\texttt{rtput}_\texttt{i}}} = \frac{\gamma\widehat{\texttt{rtput}_\texttt{i}}\widehat{N_{T_i}}\mathcal{D}}{E_i/C \pm \Delta R}\frac{1}{\widehat{\texttt{rtput}_\texttt{i}}}$$
$$= \frac{\gamma\widehat{N_{T_i}}\mathcal{D}}{\gamma\widehat{\texttt{rtput}_\texttt{i}}\widehat{N_{T_i}}\mathcal{D}/C \pm \Delta R}$$

When $\mathcal{D} \geq \Delta R$,

$$= \frac{\gamma\widehat{N_{T_i}}}{\gamma\widehat{\texttt{rtput}_\texttt{i}}\widehat{N_{T_i}}/C \pm 1}$$

Substituting $C/\widehat{\texttt{rtput}_\texttt{i}}$ derived in step 3:

$$\widehat{N_{C_i}} \in \left[N_{C_i}\frac{\gamma\widehat{N_{T_i}}}{\gamma\gamma_R\widehat{N_{T_i}} + N_{C_i}}, N_{C_i}\frac{\gamma\widehat{N_{T_i}}}{\gamma\widehat{N_{T_i}} - N_{C_i}}\right] \tag{D.3.10}$$

Note, we used the fact that updates are synchronous in computing these bounds, i.e., the estimates are not impacted by other flows' `cwnd` updates within the round **Assumption 4.** (and consequently **Assumption 3.** and **Assumption 2.**).

Note, the upper bound in Eq. D.3.10 may be very big (e.g., when N_{C_i} is close to $\gamma\widehat{N_{T_i}}$). The clamps on $\widehat{N_{C_i}}$ allow us to manage the impact of errors during such situations.

D.3.4.2 Next state, state transition function

We re-write the `cwnd` update equations (using the alternate update line 1, using **Assumption 6.**) given the errors in estimates, and then consider the stabilized state after all flows have updated their `cwnd` (**Assumption 4.**). In reality, not all flows may update their `cwnd` in a round, and the flows may not make measurements and updates in lock-step.

Consider the next reference state (Eq. D.3.3). We plug the values of $\texttt{cwnd}_\texttt{i}[\texttt{r}_\texttt{F}]$ from line 1 in Eq. D.3.4 and Eq. D.3.5, and simplifying, we get the following state update equations. Note, these equations are in terms of $\widehat{N_{C_i}}$, this term is after the clamps have been applied.

$$N_{C_i}[\texttt{r}_\texttt{F}] = \frac{\sum_j(\prod_{k\neq j} N_{C_k}\widehat{N_{T_k}})((1-\alpha)\widehat{N_{T_j}} + \alpha\widehat{N_{C_j}})}{(\prod_{k\neq i} N_{C_k}\widehat{N_{T_k}})((1-\alpha)\widehat{N_{T_i}} + \alpha\widehat{N_{C_i}})} \tag{D.3.11}$$

$$N_T[\texttt{r}_\texttt{F}] = \frac{R+\theta N_T}{\theta}\left((1-\alpha)\sum_j \frac{1}{N_{C_j}} + \alpha\sum_j \frac{\widehat{N_{C_j}}}{N_{C_j}}\frac{1}{\widehat{N_{T_j}}}\right) - \frac{R}{\theta}$$
$$= \frac{R+\theta N_T}{\theta}\left((1-\alpha) + \alpha\sum_j \frac{\widehat{N_{C_j}}}{N_{C_j}}\frac{1}{\widehat{N_{T_j}}}\right) - \frac{R}{\theta} \tag{D.3.12}$$

We verified our algebraic steps using sympy [36]. To get Eq. D.3.12, we used $\sum_{i=1}^{N_G} \frac{1}{N_{C_i}} = 1$ (from the definition of N_{C_i}, i.e., Eq. D.3.2). Also observe that if we substitute $\widehat{N_{T_i}} = N_T$, and $\widehat{N_{C_i}} = N_{C_i}$, we get the ideal link equations (Eq. D.3.6 and Eq. D.3.7).

For two flows, these equations simplify to:

$$N_{C_1}[\mathtt{r_F}] = 1 + \frac{N_{C_1}\widehat{N_{T_1}}((1-\alpha)\widehat{N_{T_2}} + \alpha\widehat{N_{C_2}})}{N_{C_2}\widehat{N_{T_2}}((1-\alpha)\widehat{N_{T_1}} + \alpha\widehat{N_{C_1}})} \tag{D.3.13}$$

$$N_T[\mathtt{r_F}] = \frac{R + \theta N_T}{\theta}\left((1-\alpha) + \alpha\left(\frac{\widehat{N_{C_1}}}{N_{C_1}\widehat{N_{T_1}}} + \frac{\widehat{N_{C_2}}}{N_{C_2}\widehat{N_{T_2}}}\right)\right) - \frac{R}{\theta} \tag{D.3.14}$$

These equations describe the reference state in the next round as a function of the reference state in the current round and estimates. In the previous section (§ D.3.4.1), we had derived bounds on estimates relative to the reference state. Using these, we obtain an expressions showing bounds on the reference state in the next round $\langle N_{C_i}[\mathtt{r_F}], N_T[\mathtt{r_F}]\rangle$ in terms of the reference state in the current round $\langle N_{C_i}, N_T\rangle$, and design/network parameters, eliminating all estimates (e.g., $\widehat{N_{C_i}}$, $\widehat{C}_i$, $\widehat{\mathtt{rtput_i}}$, $\widehat{\mathtt{rRTT_i}}$, etc.). We do not show these expressions for brevity.

D.3.4.3 State trajectories. Lemmas in Theorem 7.5.2

We encode the transition function into Z3 to prove lemmas about state evolution over rounds. We ask Z3 to find a counterexample to our lemma. A counterexample is an assignment to the variables, i.e., current state ($\langle N_{C_i}, N_T\rangle$), design parameters ($\mathcal{D}$, θ) and network parameters (C, R, D, γ_R), for which our lemma evaluates to `False`. If Z3 outputs `UNSAT`, i.e., there is no counterexample, then our lemma is `True`. We also used Z3 to come up with the lemmas to begin with. This is an iterative trial and error process where we ideate lemmas, get potential counterexamples and update the lemmas. We describe this process more in § D.3.4.4.

We list the lemmas part of Theorem 7.5.2. We prove these lemmas for the design parameters: $\gamma = 4$, $\alpha = 1$, $\delta_H = 1.3$, $\delta_L = 1.25$. Our design works whenever the network parameters satisfy: $\mathcal{D} \geq \Delta R$, $\theta * N_G > R$, $\theta > \Delta R/2$, $\gamma_R \leq 2$. Note, $\theta * N_G > R$ was also needed when analyzing the less stable `cwnd` update in § 7.5.2. This was not needed for the more stable `cwnd` update. We believe this will also not be needed for the jittery link for the more stable `cwnd` update.

For readability, we use the following substitutions in our lemmas:

$$N_{C_i}^{\mathtt{I}} = N_{C_i}[\mathtt{r}] \qquad N_{C_i}^{\mathtt{F}} = N_{C_i}[\mathtt{r}+1]$$

$$N_{C_i}^{\mathtt{LB}} = \frac{N_T^{\mathtt{UB}}}{N_T^{\mathtt{UB}} - 1} = 6.6/5.6 \qquad N_{C_i}^{\mathtt{UB}} = 6.6$$

$$N_T^{\mathtt{I}} = N_T[\mathtt{r}] \qquad N_T^{\mathtt{F}} = N_T[\mathtt{r}+1]$$

$$N_T^{\mathtt{LB}} = 0.15 \qquad N_T^{\mathtt{UB}} = 3.3$$

$$N_T^{\mathtt{LB-BAND}} = 1.8 \qquad N_T^{\mathtt{UB-BAND}} = 2.2$$

$$\sigma = 1.01$$

We obtained these non-trivial constants using binary search. We find the most extreme values at which the lemmas continue to hold. E.g., if we decrease $N_{C_i}^{\mathtt{UB}}$ from 6.6 to 6.5, Lemma D.3.6 breaks. Note, these are worst-case performance bounds, i.e., over worst-case jitter patterns. Empirical performance of FRCC is better § 7.6. Further, the lemmas also only describe worst-case progress, e.g., N_T may only show minor progress ($\sigma = 1.01$) when it is already close to converging, but it shows major progress when it is far from converging because of asymmetry in our capacity estimate (§ 7.4.3.1).

Note, these lemmas are for a two flow system ($N_G = 2$) with flowsf_i and f_j. Since $N_{C_j} = \frac{N_{C_i}}{N_{C_i}-1}$ (from the definition of N_{C_i} in Eq. D.3.2, along with $N_G = 2$), N_{C_i} represents the state of both the flows, and we state the lemmas only in terms of N_{C_i} without losing generality. The bounds $N_{C_i}^{\mathtt{LB}}$ and $N_{C_i}^{\mathtt{UB}}$ also satisfy $N_{C_i}^{\mathtt{UB}} = \frac{N_{C_i}^{\mathtt{LB}}}{N_{C_i}^{\mathtt{LB}}-1}$. As a result, $N_{C_i} \in [N_{C_i}^{\mathtt{LB}}, N_{C_i}^{\mathtt{UB}}]$ if and only if $N_{C_j} \in [N_{C_i}^{\mathtt{LB}}, N_{C_i}^{\mathtt{UB}}]$.

The lemmas also assume that $N_T > 0$ always. When $N_T < N_T^{\mathtt{LB}} = 0.15$, Lemma D.3.7 shows that N_T increases multiplicatively, so as long as we start with $N_T > 0$, $N_T > 0$ will keep remaining `True`. $N_T > 0$ is true whenever the bottleneck queue occupancy is non-zero (this also ensures that the bottleneck link is utilized). It is okay to assume $N_T > 0$ because FRCC increases `cwnd` whenever $\widehat{N_{T_i}} < 1$ (because there is always at least one flow in the system). Since $\widehat{N_{T_i}} \leq N_T + \Delta R/2\theta$ (§ D.3.4.1), as long as $\Delta R/2\theta < 1$ (which is what we are designing for), $\widehat{N_{T_i}}$ will always be < 1, whenever $N_T \leq 0$, so FRCC will keep increasing `cwnd` (multiplicatively), so that $N_T > 0$ happens exponentially fast.

Lemma D.3.6 (N_T and N_{C_i} bounded). *If initial N_T and N_{C_i} are converged to a bounded range, then they remain converged. I.e.,*

$$N_{C_i}^{I} \in [N_{C_i}^{LB}, N_{C_i}^{UB}] \implies N_{C_i}^{F} \in [N_{C_i}^{LB}, N_{C_i}^{UB}] \textit{ and,}$$
$$N_{T}^{I} \in [N_{T}^{LB}, N_{T}^{UB}] \implies N_{T}^{F} \in [N_{T}^{LB}, N_{T}^{UB}]$$

Lemma D.3.7 (N_T converges). *If initial N_T is outside its bounded range, it converges towards the range exponentially fast without overshooting (e.g., if it is above the range it does not reduce to a value below the range), I.e.,*

$$N_T^I > N_T^{UB} \implies \left(N_T^F \leq N_T^I/\sigma \vee N_T^F \leq N_T^{UB}\right) \wedge N_T^F \geq N_T^{LB} \textit{ and,}$$
$$N_T^I < N_T^{LB} \implies \left(N_T^F \geq N_T^I\sigma \vee N_T^F \geq N_T^{LB}\right) \wedge N_T^F \leq N_T^{UB}$$

Lemma D.3.8 (N_{C_i} converges). *If either the initial N_T or initial N_{C_i} is not converged, then either both converge or two things happen: (1) at least one of them (one that is not already converged) improves (exponentially fast without overshooting) while (2) neither moves away from the bounded range. I.e.,*

$$\neg(N_{C_i}^{I} \in [N_{C_i}^{LB}, N_{C_i}^{UB}] \wedge N_T^I \in [N_T^{LB}, N_T^{UB}])$$
$$\implies (N_{C_i}^{F} \in [N_{C_i}^{LB}, N_{C_i}^{UB}] \wedge N_T^F \in [N_T^{LB}, N_T^{UB}])$$

$$\begin{aligned}
&\vee ((\textit{bad } N_{C_i} \textit{ improves} \vee \textit{bad } N_T \textit{ improves})\\
&\quad \wedge N_{C_i} \textit{ does not degrade}\\
&\quad \wedge N_T \textit{ does not degrade})
\end{aligned}$$

*Where "bad N_{C_i} improves" means that N_{C_i} is outside its converged range, ~~implies~~ **and** it moves exponentially fast towards the range without overshooting, i.e.,*

$$N^{I}_{C_i} > N^{UB}_{C_i} \wedge N^{F}_{C_i} \geq N^{LB}_{C_i} \wedge (N^{F}_{C_i} \leq N^{I}_{C_i}/\sigma \vee N^{F}_{C_i} \leq N^{UB}_{C_i}) \text{ or,}$$
$$N^{I}_{C_j} > N^{UB}_{C_i} \wedge N^{F}_{C_j} \geq N^{LB}_{C_i} \wedge (N^{F}_{C_j} \leq N^{I}_{C_j}/\sigma \vee N^{F}_{C_j} \leq N^{UB}_{C_i})$$

*"bad N_T improves" means that N_T is outside the $[N_T^{LB-BAND}, N_T^{UB-BAND}]$ band, ~~implies~~ **and** it moves towards it exponentially fast without overshooting:*

$$\begin{aligned}
&N^{I}_{T} > N^{UB-BAND}_{T} \wedge N^{F}_{T} \geq N^{LB-BAND}_{T} \wedge\\
&\qquad (N^{F}_{T} \leq N^{I}_{T}/\sigma \vee N^{F}_{T} \leq N^{UB-BAND}_{T}) \text{ or,}\\
&N^{I}_{T} < N^{LB-BAND}_{T} \wedge N^{F}_{T} \leq N^{UB-BAND}_{T} \wedge\\
&\qquad (N^{F}_{T} \geq N^{I}_{T} * \sigma \vee N^{F}_{T} \geq N^{LB-BAND}_{T})
\end{aligned}$$

"N_{C_i} does not degrade" means that if N_{C_i} is not converged then it does not move away from its converged range, i.e.,

$$(N^{I}_{C_i} < N^{LB}_{C_i} \implies N^{F}_{C_i} \geq N^{I}_{C_i}) \wedge (N^{I}_{C_i} > N^{UB}_{C_i} \implies N^{F}_{C_i} \leq N^{I}_{C_i})$$

Similarly, "N_T does not degrade" means:

$$(N^{I}_{T} < N^{LB}_{T} \implies N^{F}_{T} \geq N^{I}_{T}) \wedge (N^{I}_{T} > N^{UB}_{T} \implies N^{F}_{T} \leq N^{I}_{T})$$

In "bad N_{C_i} improves", you will notice that both the conjunctions use the upper bound. Instead of writing the second conjunction in terms of the lower bound, we write it in terms of N_{C_j} which refers to the other flow. If N_{C_i} is below its lower bound then N_{C_j} has to be above the upper bound. This follows from $N_{C_j} = \frac{N_{C_i}}{N_{C_i}-1}$ (from the definition of N_{C_i} in Eq. D.3.2, along with $N_G = 2$), and $N^{\mathsf{UB}}_{C_i} = \frac{N^{\mathsf{LB}}_{C_i}}{N^{\mathsf{LB}}_{C_i}-1}$ (from the constant values). We wrote it this way because the relative change in N_{C_i} is higher when $N_{C_i} \geq 1 \geq N_{C_j}$, (i.e., the flow that is consuming lesser fraction of the link has higher relative change in their fraction after the round updates).

We need the "neither degrades" part to ensure that the N_{C_i} and N_T do not oscillate in a way where one of them improves and the other degrades and vice versa without making any progress towards convergence. Similarly, we also need the "one which is not converged improves", or "bad" in "bad N_T improves". If we replace "bad N_T improves", with just "N_T improves", we would get "N_T above the $[N_T^{\mathsf{LB-BAND}}, N_T^{\mathsf{UB-BAND}}]$ band implies it decreases and N_T below the band implies it increases". This formula is true when N_T is inside the band, and as a result satisfies the post condition of the lemma without forcing N_{C_i} to ever converge towards its range.

Summary. Going back to Fig. 7.13, if $N_T^{\mathtt{I}}$ is outside its converged range (top or bottom) then it eventually converges from Lemma D.3.7. I.e., the system is in the left, center, or right boxes in Fig. 7.13. If state is in the center box, we are done (i.e., system is in steady-state, it stays that way from Lemma D.3.6 and we deliver the steady-state performance bounds). Left and right boxes are symmetric (one of N_{C_i} or N_{C_j} lies in the right box if the system is in left or right boxes). Without loss of generality, say N_{C_i} is in the right box. Then from Lemma D.3.8, if N_{C_i} does not improve, then N_T must move towards the narrow band if it is outside the band, whenever it is inside the band, N_{C_i} must improve. N_T can exit the narrow band but not leave its converged range, and the process repeats. N_{C_i} can never move away from the converged range. The result of these movements is that eventually N_{C_i} must converge. These movements are shown using blue arrows in Fig. 7.13.

All the movements are multiplicative without any overshoots, thus asymptotically the system converges to steady-state in logarithmic steps in the total amount of movement. Each step is a round that lasts for $O(N_G)$ slots, with each slot being 4RTTs, and the total movement is bounded by the steady-state $\texttt{cwnd} = C/N_G * (R + \theta N_G) = \texttt{BDP}/N_G + C\theta N_G$ (this is $\texttt{cwnd}$ FRCC maintains in steady-state on ideal link). Giving a total convergence time of $O(N_G \log(\texttt{BDP}))$ RTTs.

Note, N_T may enter and exit the band multiple times, these are all hidden in the big-O notation.

D.3.4.4 How we come up with the lemmas

Coming up with the lemmas is an iterative process. We start by identifying a region in the state space such that if initial state is in the region, then the final state is in the region. This gives us the shaded region. A rectangular region worked for us. Note, the proof is not tight, the real region could be a different shape. Then we ideate lemmas to describe motions the state makes then it is outside the shaded region. Z3 may prove that the lemma is correct, or give us a counterexample showing how the state moves instead of what our lemma described. Based on this feedback, we update the lemma and iterate.

D.4 Fluid model analysis (different RTprops and multiple bottlenecks)

Different RTprops. We compute value of biased excess delay from the fluid model as follows. Say in the current state, the RTTs (not RTprops) of the two flows are R and ρR (i.e., RTT ratio of ρ), and throughputs are fC and $(1-f)C$. Consequently, the $\texttt{cwnd}$s of the flows are: $\texttt{cwnd}_1 = R \cdot fC, \texttt{cwnd}_2 = R \cdot (1-f)C$ (using $\texttt{throughput} = \texttt{cwnd}/\texttt{RTT}$). When flow f_1 probes, the delay increase seen by the probe Δd is given by:

$$\frac{\texttt{cwnd}_1 + E}{R + \Delta d} + \frac{\texttt{cwnd}_2}{\rho R + \Delta d} = C \tag{D.4.1}$$

I.e., the RTTs of both flows increase by Δd and the sum of the throughputs during the probe are equal to C. We substitute the $\texttt{cwnd}$s and solve for Δd as a function of ρ, R, E $(\gamma\mathcal{D})$, f, and C. This expression is complicated and we do not show it. But $\Delta d \neq E/C$. Note, if we substitute $\rho = 1$

in Eq. D.4.1, then $\Delta d = E/C$. If $\rho > 1$ (short RTprop flow probes), $\Delta d > E/C$, while $\Delta d < E/C$, when $\rho < 1$ (long flow probes).

Since Δd is a complicated expression we not substitute it all the way through to get an updated transition function, instead we solve for the fixed-point indirectly. Specifically, the system is in steady-state when no flow has incentive to change its `cwnd`, i.e., $\widehat{N_{T_i}} = \widehat{N_{C_i}}$. Since flows measure the same queueing delay, $\widehat{N_{T_1}} = \widehat{N_{T_2}}$. Thus, steady-state happens when $\widehat{N_{C_1}} = \widehat{N_{C_2}}$. We substitute the biased Δd in $\widehat{N_C}$ $(= \widehat{C}/\widehat{\texttt{rtput}} = E/(\widehat{\Delta d}\widehat{\texttt{rtput}}))$. We get $\widehat{N_{C_i}}$ as a function of the system state and design/network parameters. Solving $\widehat{N_{C_1}} = \widehat{N_{C_2}}$, gives us the fixed-point state as a function of design/network parameters. Note, in the fluid model, hats ($\widehat{N_{C_i}}$) and non-hat (N_{C_i}) variables are the same.

Solving $\widehat{N_{C_1}} = \widehat{N_{C_2}}$ requires finding roots of a high degree polynomial, we use numerical methods (secant method in sympy [36]) to approximate the root.

Multi-bottleneck parking lot topology. The setup and computation of the fixed-point is similar to the different RTprop case. We write fluid model equations for each hop in the parking lot. This time, the excess delay, Δd, when the long flow (f_0 in Fig. 7.15) probes is given by:

$$\frac{\texttt{cwnd}_0 + E}{R_0 + \Delta d} + \frac{\texttt{cwnd}_1}{R_1 + \Delta d/\texttt{hops}} = C \tag{D.4.2}$$

The main difference is that if before the probe the RTTs of flow f_0 (long flow) and f_1 (short flow) were R_0 and R_1, then after the probe the RTTs become $R_0 + \Delta d$ and $R_1 + \Delta d/\texttt{hops}$ respectively. I.e., the increase in the RTT of the short flow is $1/\texttt{hops}$ times less than the increase in the RTT of the long flow. This is because the RTTs of all the short flow (f_1 to f_{hops}) increases and f_0 witnesses the sum of the delays over all hops. The delay at each hops is the same (due to symmetry fluid equations for each hop). So the total increase in RTTs of all the short flows matches the increase in RTT of the long flow.

When solving for the fixed-point, $\widehat{N_{T_i}} = \widehat{N_{C_i}}$. Due to the same reason as above, this time, $\widehat{N_{T_0}} = \texttt{hops}\widehat{N_{T_1}}$. So we solve for $\widehat{N_{C_0}} = \texttt{hops}\widehat{N_{C_1}}$ to determine the fixed-point.

D.5 Supplementary empirical evaluation

Fig. D.2 and Fig. D.3 complement Fig. 2.1 and Fig. 2.2 to show that BBRv3 and Reno also cause starvation in the scenarios with different propagation delays and jitter.

Impact of upper clamp with higher flow counts. Fig. D.1 shows FRCC with more flows. The upper clamp on slot count is $K_{\max} = 20$, and the dynamic slot count is $k\widehat{N_T}$, with $k = 2$. With up to 10 flows the upper clamp is not hit and JFI is close to 1. Beyond that there are less than kN_G slots and collisions become more likely creating some unfairness.

Note, since FRCC's practical convergence speed is slow, to ensure flows reach steady-state, we ran this experiment with RTprop of 10 ms instead of our default of 50 ms (§ 7.6).

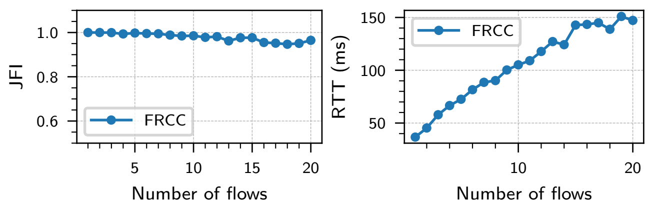


Figure D.1: FRCC with more flows.

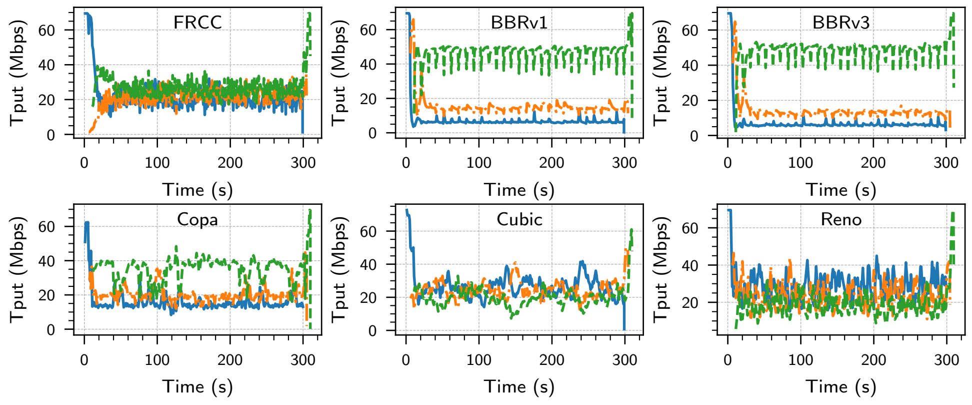


Figure D.2: Flows with round-trip propagation delay (Rtprop or R) of 10, 20 and 30 ms. BBR starves the (blue) flow with lower Rtprop.

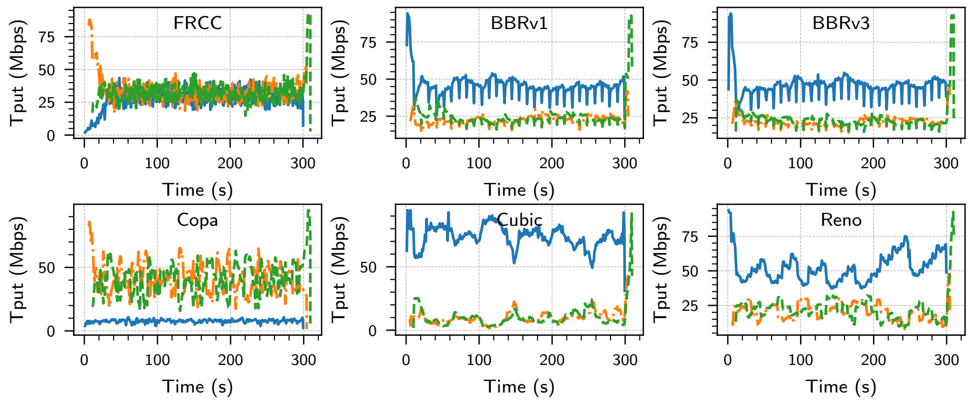


Figure D.3: Three flows with $R = 32$ ms. The blue flow experiences 32 ms of ACK aggregation. Cubic and Copa starve flows.

Symbol	**Unit**	**Domain**	**Description**
Network parameters			
C	r	$\mathbb{R}_{>0}$	Link capacity
`BDP`	p	$\mathbb{R}_{>0}$	Bandwidth delay product
R_i, R	s	$\mathbb{R}_{>0}$	Round-trip propagation delay
D_i, D	s		Maximum network jitter per-packet
N_G	u	$\mathbb{Z}_{\geq 1}$	Number of flows (ground truth)
ΔR	s		Max (additive) error in RTT difference
γ_R	u	$\mathbb{R}_{\geq 1}$	Max (multiplicative) error in Tput
Design parameters			
R_{max}	s	$\mathbb{R}_{>0}$	Assumption about maximum R
$\mathcal{D}$	s	$\mathbb{R}_{>0}$	Amount of ΔR we want to tolerate
θ	s	$\mathbb{R}_{>0}$	Delay we maintain per flow
γ	u	$\mathbb{R}_{>1}$	Gain multiplier for probing
α	u	$(0, 1]$	`cwnd` update weight
δ_L, δ_H	u	$\mathbb{R}_{>1}$	Clamps for `cwnd` update
k	u	$\mathbb{R}_{>1}$	$k\widehat{N_T}$ = Number of slots in a round
T_P	s	$\mathbb{R}_{>0}$	Probe duration (= 1 RTT)
T_R	s	$\mathbb{R}_{>0}$	Time between RTprop probes (= 30 s)

Table D.1: Glossary of parameters. For the units: r = rate (i.e., packets/seconds), s = seconds, p = packets, u = unit-less. By default, the domain is $\mathbb{R}_{\geq 0}$. Subscript i denotes quantity seen/maintained by flow i. We drop the subscripts when clear from context.

D.6 Glossary

Table D.1 and Table D.2 parameter and state variable definitions.

Symbol	Unit	Domain	Description
$\texttt{cwnd}_\texttt{i}$	p	$\mathbb{R}_{\geq 1}$	Congestion window size
$\widehat{C}_i$	r		Link capacity estimate
$\widehat{R}_i$	s		R estimate
$\texttt{rRTT}_\texttt{i}$, $\widehat{\texttt{rRTT}_\texttt{i}}$	s		RTT in a round
$\texttt{rtput}_\texttt{i}$, $\widehat{\texttt{rtput}_\texttt{i}}$	r		Throughput in a round
N_T, N_{T_i}, $\widehat{N_{T_i}}$	u		Target flow count
N_{C_i}, $\widehat{N_{C_i}}$	u	$N_{C_i} \geq 1$, $\widehat{N_{C_i}} \geq 0$	Current flow count (from the perspective of flow i)
E_i	p	$\mathbb{R}_{\geq 1}$	Excess packets sent by probe
Δd_i, $\widehat{\Delta d_i}$	s	$\Delta d_i \geq 0$, $\widehat{\Delta d_i} \in \mathbb{R}$	Excess delay due to probe

Table D.2: Glossary of state variables. We use hats ($\widehat{\texttt{rRTT}_\texttt{i}}$) to denote estimates made by FRCC, and no annotation ($\texttt{rRTT}_\texttt{i}$) to denote the value of the state variable in the fluid reference execution. We use a suffix of $\texttt{[r]}$ to denote the round number (e.g., $\texttt{rRTT}_\texttt{i}\texttt{[r]}$, $\widehat{\texttt{rRTT}_\texttt{i}}\texttt{[r]}$). See Table D.1 for convention on units, domain, and subscript i.

Bibliography

[1] [tcp]: Add tcp vegas congestion control module. · torvalds/linux@b87d856, 6 2005. URL https://github.com/torvalds/linux/commit/b87d8561d8667d221b728ccdcb18eb95b16a687b. [Online; accessed 2025-06-02]. 61

[2] tcp: tcp_vegas cong avoid fix · torvalds/linux@8d3a564, 12 2008. URL https://github.com/torvalds/linux/commit/8d3a564da34e5844aca4f991b73f8ca512246b23. [Online; accessed 2025-06-04]. 71

[3] Pantheon emulation result for scream and webrtc. 12 mbps bottleneck with 30 ms rtprop and 1 bdp buffer, 4 2018. URL https://pantheon.stanford.edu/result/2422/. [Online; accessed 2025-05-26]. 61

[4] Working Groups - Ultra Ethernet Consortium, December 2023. URL https://ultraethernet.org/working-groups. [Online; accessed 18. Nov. 2024]. 70

[5] csg-htsim, November 2024. URL https://github.com/Broadcom/csg-htsim. [Online; accessed 18. Nov. 2024]. 12, 71, 74

[6] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96145-3. xiii, 30, 125

[7] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/addanki. xiv, 56, 72

[8] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Automating network heuristic design and analysis. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 8–16, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398992. doi: 10.1145/3563766.3564085. URL https://doi.org/10.1145/3563766.3564085. 11, 29

[9] 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, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL

https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup. 11

[10] Anup Agarwal, Venkat Arun, and Srinivasan Seshan. Contracts: A unified lens on congestion control robustness, fairness, congestion, and generality, 2025. URL https://arxiv.org/abs/2504.18786. 11

[11] Anup Agarwal, Venkat Arun, and Srinivasan Seshan. Frcc: Towards provably fair and robust congestion control. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, Renton, WA, May 2026. USENIX Association. 11

[12] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302012. doi: 10.1145/1851182.1851192. URL https://doi.org/10.1145/1851182.1851192. 2, 55, 58, 66, 71

[13] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 73–84, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308144. doi: 10.1145/1993744.1993753. URL https://doi.org/10.1145/1993744.1993753. 58, 66

[14] Ashwani Anand, Anne-Kathrin Schmuck, and Satya Prakash Nayak. Contract-based distributed logical controller synthesis. In *Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705229. doi: 10.1145/3641513.3650123. URL https://doi.org/10.1145/3641513.3650123. 112

[15] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, page 281–292, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138628. doi: 10.1145/1015467.1015499. URL https://doi.org/10.1145/1015467.1015499. 13

[16] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 645–661, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/tahmasbi. 115

[17] Venkat Arun. venkatarun95/genericcc, 2025. URL https://github.com/venkatarun95/genericCC. [Online; accessed 2025-04-17]. 44, 75, 101, 135

[18] 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, 4 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/arun. 5, 13, 14, 15, 32, 44, 53, 57, 59, 75, 87, 101, 102

[19] 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, pages 1–16, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837. doi: 10.1145/3452296.3472912. URL https://doi.org/10.1145/3452296.3472912. 1, 3, 6, 13, 14, 16, 17, 22, 24, 31, 45, 86, 121

[20] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the 2022 ACM SIGCOMM 2022 Conference*, SIGCOMM '22, Amsterdam, Netherlands, 2022. Association for Computing Machinery. xv, 1, 2, 3, 5, 9, 10, 13, 14, 15, 16, 17, 37, 46, 53, 56, 59, 63, 64, 66, 68, 69, 70, 76, 79, 80, 81, 82, 86, 88, 95, 102, 108, 127, 129, 142

[21] D. Bansal and H. Balakrishnan. Binomial congestion control algorithms. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 2, pages 631–640 vol.2, 2001. doi: 10.1109/INFCOM.2001.916251. 6, 17, 46

[22] Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002. 10

[23] V Blondel, CT Abdallah, and GL Heileman. Complexity issues and decision methods in control systems. *J. Symbolic Computation*, 11:1–12, 1995. 47

[24] Tommaso Bonato, Abdul Kabbani, Daniele De Sensi, Rong Pan, Yanfang Le, Costin Raiciu, Mark Handley, Timo Schneider, Nils Blach, Ahmad Ghalayini, Daniel Alves, Michael Papamichael, Adrian Caulfield, and Torsten Hoefler. Fastflow: Flexible adaptive congestion control for high-performance datacenters, 2024. URL https://arxiv.org/abs/2404.01630. 6, 70

[25] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation.* John Wiley & Sons, 2018. 17

[26] Patricia Bouyer, Stéphane Le Roux, Youssouf Oualhadj, Mickael Randour, and Pierre Vandenhove. Games where you can play optimally with arena-independent finite memory. *Logical Methods in Computer Science*, 18, 2022. 120

[27] L.S. Brakmo and L.L. Peterson. Tcp vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995. doi: 10.1109/49.464716. 5, 7, 9, 13, 46, 53, 63, 65, 74, 87

[28] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. How i learned to stop worrying about cca contention. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 229–237, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704154. doi: 10.1145/3626111.3628204. URL https://doi.org/10.1145/3626111.3628204. 10, 113

[29] Carlo Caini and Rosario Firrincieli. Tcp hybla: A tcp enhancement for heterogeneous networks. *Int. J. Satell. Commun. Netw.*, 22(5):547–566, September 2004. ISSN 1542-0973. doi: 10.1002/sat.799. URL https://doi.org/10.1002/sat.799. 2

[30] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van

Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October: 20 – 53, 2016. URL `http://queue.acm.org/detail.cfm?id=3022184`. 2, 6, 13, 14, 21, 23, 33, 54, 59, 63, 75, 84, 89, 92, 93, 94, 101

[31] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, and Van Jacobson. Bbr congestion control work at google ietf 101 update, 2018. URL `https://datatracker.ietf.org/meeting/101/materials/slides-101-iccrg-an-update-on-bbr-work-at-google-00`. [Online; accessed 4. Mar. 2024]. Slides 4, 17. 23

[32] Neal Cardwell, Ian Swett, and Joseph Beshay. BBR Congestion Control. Internet-Draft draft-ietf-ccwg-bbr-01, Internet Engineering Task Force, October 2024. URL `https://datatracker.ietf.org/doc/draft-ietf-ccwg-bbr/01/`. Work in Progress. 1, 14, 71, 104, 112

[33] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Ian Swett, and Van Jacobson. google/bbr at v2alpha, 2025. URL `https://github.com/google/bbr/tree/v2alpha`. [Online; accessed 2025-04-17]. 13, 14, 44

[34] Neal Cardwell, Ian Swett, and Joseph Beshay. google/bbr at v3, 2025. URL `https://github.com/google/bbr/tree/v3`. [Online; accessed 2025-04-17]. 13, 14, 44, 101

[35] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016. 7, 46

[36] Ondrej Certik. Sympy, 2025. URL `https://www.sympy.org/en/index.html`. [Online; accessed 2025-04-25]. 80, 156, 160

[37] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1): 1–14, 1989. ISSN 0169-7552. doi: https://doi.org/10.1016/0169-7552(89)90019-6. URL `https://www.sciencedirect.com/science/article/pii/0169755289900196`. xiv, 17, 46, 54, 139

[38] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, July 2001. ISSN 1572-8102. doi: 10.1023/A:1011276507260. 16, 29

[39] Mininet Project Contributors. Mininet: An instant virtual network on your laptop (or other pc) - mininet, 2025. URL `https://mininet.org/`. [Online; accessed 2025-04-17]. 74, 101

[40] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. 3, 30, 80, 86, 126

[41] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015. xiv, 1, 15, 45, 54, 57, 60, 110

[42] 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, 2018. 13, 110

[43] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. What do packet dispersion techniques measure? In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 905–914. IEEE, 2001. 8, 10, 40, 83, 89

[44] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden. Routers with very small buffers. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–11, 2006. doi: 10.1109/INFOCOM.2006.240. 13

[45] Margarida Ferreira, Ranysha Ware, Yash Kothari, Inês Lynce, Ruben Martins, Akshay Narayan, and Justine Sherry. Reverse-engineering congestion control algorithm behavior. In *Proceedings of the 2024 ACM on Internet Measurement Conference*, IMC '24, page 401–414, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705922. doi: 10.1145/3646547.3688443. URL https://doi.org/10.1145/3646547.3688443. 113

[46] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 468–482, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936. doi: 10.1145/2934872.2934873. URL https://doi.org/10.1145/2934872.2934873. 13

[47] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993. doi: 10.1109/90.251892. 70, 112

[48] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, page 43–56, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132239. doi: 10.1145/347059.347397. URL https://doi.org/10.1145/347059.347397. 5, 6, 55, 57, 61, 64, 73, 108, 112

[49] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001. URL https://www.rfc-editor.org/info/rfc3168. [Online; accessed 21. Nov. 2024]. 58, 70

[50] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-Latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/fouladi. 7, 46

[51] Hugo Gimbert and Wiesław Zielonka. *Games Where You Can Play Optimally without Any Memory*, page 428–442. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 3540283099. URL https://doi.org/10.1007/11539452_33. 120

[52] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. Quantitative verification of scheduling heuristics, 2023. 115

[53] 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. 113

[54] Google. bbr/Documentation/startup/gain/analysis/bbr_drain_gain.pdf at master · google/bbr, November 2024. URL `https://github.com/google/bbr/blob/master/Documentation/bbr_bandwidth_based_convergence.pdf`. [Online; accessed 14. Nov. 2024]. 59, 76

[55] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL `https://www.usenix.org/conference/nsdi20/presentation/goyal`. 13, 14, 31, 67, 76, 80

[56] Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. Optimal congestion control for time-varying wireless links, 2022. 46

[57] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: a building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 158–176, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544221. URL `https://doi.org/10.1145/3544216.3544221`. 15, 89

[58] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, April 2012. URL `https://www.rfc-editor.org/info/rfc6582`. [Online; accessed 20. Nov. 2024]. 6, 13

[59] 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 `https://doi.org/10.1145/1400097.1400105`. 2, 7, 13, 14, 32, 75, 76, 101, 138

[60] Cunwu Han, Dehui Sun, Lei Liu, and Song Bi. A new robust model predictive congestion control. In *Proceeding of the 11th World Congress on Intelligent Control and Automation*, pages 4189–4193. IEEE, 2014. 45

[61] Feixue Han, Qing Li, Peng Zhang, Gareth Tyson, Yong Jiang, Mingwei Xu, Yulong Lan, and ZhiCheng Li. ETC: An elastic transmission control using End-to-End available bandwidth perception. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 265–284, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-41-0. URL `https://www.usenix.org/conference/atc24/presentation/han`. 72, 88

[62] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi:

10.1145/3098822.3098825. URL https://doi.org/10.1145/3098822.3098825. 74

[63] Janey C. Hoe. Improving the start-up behavior of a congestion control scheme for tcp. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, page 270–280, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917901. doi: 10.1145/248156.248180. URL https://doi.org/10.1145/248156.248180. 6, 13, 14, 58, 66, 71, 76, 87, 101, 112

[64] Amjad J Humaid, Hamid M Hasan, and Firas A Raheem. Development of model predictive controller for congestion control problem. *feedback*, 2:3, 2014. 45

[65] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21(1), 1984. 101

[66] Natasha Jaques, Angeliki Lazaridou, Edward Hughes, Caglar Gulcehre, Pedro Ortega, Dj Strouse, Joel Z. Leibo, and Nando De Freitas. Social influence as intrinsic motivation for multi-agent deep reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3040–3049. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/jaques19a.html. 111, 112

[67] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019. 45

[68] Wanchun Jiang, Haoyang Li, Jia Wu, Kai Wang, Fengyuan Ren, and Jianxin Wang. Introspective congestion control for consistent high performance. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 428–445, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696084. URL https://doi.org/10.1145/3689031.3696084. 6, 54, 56, 66

[69] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998. 25

[70] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, 8 2002. ISSN 0146-4833. doi: 10.1145/964725.633035. URL https://doi.org/10.1145/964725.633035. 76, 80

[71] Frank Kelly. Fairness and stability of end-to-end congestion control*. *European Journal of Control*, 9(2):159–176, 2003. ISSN 0947-3580. doi: https://doi.org/10.3166/ejc.9.159-176. URL https://www.sciencedirect.com/science/article/pii/S0947358003702738. 9, 54, 61, 63

[72] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. Learning smt(lra) constraints using smt solvers. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 2333–2340. AAAI Press, 7 2018. ISBN 9780999241127. doi: 10.24963/ijcai.2018/323. URL https://doi.org/10.24963/ijcai.2018/323. 30, 125

[73] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam

Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3406591. URL `https://doi.org/10.1145/3387514.3406591`. 5, 6, 9, 53, 56, 57, 63, 65, 66, 70, 73, 74, 82, 87

[74] K. Lai and M. Baker. Measuring bandwidth. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 235–245 vol.1, 1999. doi: 10.1109/INFCOM.1999.749288. 8, 40, 83, 89

[75] Kevin Lai and Mary Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, page 283–294, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132239. doi: 10.1145/347059.347557. URL `https://doi.org/10.1145/347059.347557`. 8, 83, 89

[76] T.V. Lakshman and U. Madhow. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5 (3):336–350, 1997. doi: 10.1109/90.611099. 67

[77] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098842. URL `https://doi.org/10.1145/3098822.3098842`. 31

[78] Yanfang Le, Rong Pan, Peter Newman, Jeremias Blendin, Abdul Kabbani, Vipin Jain, Raghava Sivaramu, and Francis Matus. Strack: A reliable multipath transport for ai/ml clusters, 2024. URL `https://arxiv.org/abs/2407.15266`. 6, 70

[79] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer, 2001. 31

[80] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001. 17

[81] Rodolfo I. Ledesma Goyzueta and Yu Chen. A deterministic loss model based analysis of cubic. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 944–949, 2013. doi: 10.1109/ICCNC.2013.6504217. 76

[82] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer*

Communication, SIGCOMM '20, page 15–30, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405850. URL https://doi.org/10.1145/3387514.3405850. xiv, 60

[83] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2018. 45

[84] Xudong Liao, Han Tian, Chaoliang Zeng, Xinchen Wan, and Kai Chen. Astraea: Towards fair and efficient learning-based congestion control. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 99–114, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3650069. URL https://doi.org/10.1145/3627703.3650069. xiii, xiv, 2, 6, 54, 55, 56, 57, 59, 60, 68, 69, 111

[85] S.H. Low. A duality model of tcp and queue management algorithms. *IEEE/ACM Transactions on Networking*, 11(4):525–536, 2003. doi: 10.1109/TNET.2003.815297. 9, 54, 61, 63, 66, 112

[86] S.H. Low. *Analytical Methods for Network Congestion Control*. Synthesis Lectures on Learning, Networks, and Algorithms. Springer International Publishing, 2022. ISBN 9783031792755. URL https://books.google.com/books?id=tYFyEAAAQBAJ. 54, 61

[87] Steven H. Low, Larry L. Peterson, and Limin Wang. Understanding tcp vegas: a duality model. *J. ACM*, 49(2):207–235, March 2002. ISSN 0004-5411. doi: 10.1145/506147.506152. URL https://doi.org/10.1145/506147.506152. 5, 9, 58, 63, 65

[88] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path transport for RDMA in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/lu. 55, 71

[89] 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 https://doi.org/10.1145/263932.264023. 5, 17, 55, 66, 87, 108

[90] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 615–631, 2020. 13, 110

[91] Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an eye on congestion control in the wild with nebby. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 136–150, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672223. URL https://doi.org/10.1145/3651890.3672223. 113

[92] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely:

Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787510. URL https://doi.org/10.1145/2785956.2787510. 56, 58

[93] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking*, 8(5):556–567, 2000. doi: 10.1109/90.879343. 9, 15, 67, 100

[94] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/netravali. 12, 44, 74, 101

[95] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/olteanu. 74

[96] 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, page 303–314, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130031. doi: 10.1145/285237.285291. URL https://doi.org/10.1145/285237.285291. 17, 61

[97] Jia Pan, Anup Agarwal, Isil Diling, and Venkat Arun. Syntra: Synthesizing cross-layer controllers for low-latency video streaming. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, Renton, WA, May 2026. USENIX Association. 10, 11, 46, 47, 110, 114

[98] Adithya Abraham Philip, Ranysha Ware, Rukshani Athapathu, Justine Sherry, and Vyas Sekar. Revisiting tcp congestion control throughput models & fairness properties at scale. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, pages 96–103, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391290. doi: 10.1145/3487552.3487834. URL https://doi.org/10.1145/3487552.3487834. 15

[99] 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, page 506–520, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672229. URL https://doi.org/10.1145/3651890.3672229. 15

[100] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 266–277,

New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307970. doi: 10.1145/2018436.2018467. URL https://doi.org/10.1145/2018436.2018467. 74

[101] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 235–242, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398992. doi: 10.1145/3563766.3564115. URL https://doi.org/10.1145/3563766.3564115. 113

[102] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998. 48

[103] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 708–721, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405897. URL https://doi.org/10.1145/3387514.3405897. 6, 70

[104] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 479–490, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328364. doi: 10.1145/2619239.2626324. URL https://doi.org/10.1145/2619239.2626324. xiv, 2, 54, 57, 60

[105] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. *SIGPLAN Not.*, 43(6):136–148, June 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375599. URL https://doi.org/10.1145/1379022.1375599. 30

[106] Rayadurgam Srikant and Tamer Başar. *The mathematics of Internet congestion control*. Springer, 2004. 9, 54, 59, 61, 63, 88

[107] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12, 2006. doi: 10.1109/INFOCOM.2006.188. 13

[108] Ao Tang, Jiantao Wang, and S.H. Low. Counter-intuitive throughput behaviors in networks under end-to-end control. *IEEE/ACM Transactions on Networking*, 14(2):355–368, 2006. doi: 10.1109/TNET.2006.872552. 67

[109] Alfred Tarski. A decision method for elementary algebra and geometry. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84, Vienna, 1998. Springer Vienna. ISBN 978-3-7091-9459-1. 47, 48

[110] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. Reinforcement learning for datacenter congestion control. *SIGMETRICS Perform. Eval. Rev.*, 49(2):43–46, January 2022. ISSN 0163-5999. doi: 10.1145/3512798.3512815. URL https://doi.org/10.1145/3512798.3512815. 45

[111] Curtis Villamizar and Cheng Song. High performance tcp in ansnet. *ACM SIGCOMM Computer Communication Review*, 24(5):45–60, 1994. 13

[112] 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, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/wang-weitao. 5, 55, 57, 58, 62, 66, 67, 68, 76, 100

[113] 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, page 137–143, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369480. doi: 10.1145/3355369.3355604. URL https://doi.org/10.1145/3355369.3355604. 15, 76, 104

[114] Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria, Yash Kothari, Justine Sherry, and Srinivasan Seshan. Ccanalyzer: An efficient and nearly-passive congestion control classifier. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 181–196, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672255. URL https://doi.org/10.1145/3651890.3672255. 113

[115] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. Fast tcp: Motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14(6): 1246–1259, 2006. doi: 10.1109/TNET.2006.886335. 5, 53, 66, 87, 88, 94, 95, 100

[116] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 123–134, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320566. doi: 10.1145/2486001.2486020. URL https://doi.org/10.1145/2486001.2486020. xiv, 2, 10, 13, 15, 21, 45, 54, 57, 60, 63, 68

[117] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, Lombard, IL, April 2013. USENIX Association. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein. 2, 13

[118] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 651–664. ACM, 2021. doi: 10.1145/3453483.3454068. URL https://doi.org/10.1145/3453483.3454068. 32

[119] Lisong Xu, K. Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524 vol.4, 2004. doi: 10.1109/INFCOM.2004.1354672. 138

[120] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, 7 2018. USENIX Association. ISBN 978-1-939133-01-4. URL `https://www.usenix.org/conference/atc18/presentation/yan-francis`. xiv, xix, 1, 12, 13, 60, 74, 75

[121] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 509–522, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787498. URL `https://doi.org/10.1145/2785956.2787498`. 2

[122] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. Axiomatizing congestion control. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2):33:1–33:33, 2019. doi: 10.1145/3341617.3326148. URL `https://doi.org/10.1145/3341617.3326148`. 17, 46

[123] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787484. URL `https://doi.org/10.1145/2785956.2787484`. 2, 55, 59

[124] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 313–327, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342926. doi: 10.1145/2999572.2999593. URL `https://doi.org/10.1145/2999572.2999593`. 9, 55, 56, 59, 76