Verifying the Performance of Network Control Algorithms

by

Venkat Arun

B.Tech. Indian Institute of Technology, Guwahati (2017)
M.S. Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2023

Verifying the Performance of Network Control Algorithms

by

Venkat Arun

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

As networked systems become critical infrastructure, their design must reflect their new societal role. Today, we build systems with hundreds of heuristics but often do not understand their inherent and emergent behaviors. This dissertation presents, *performance verification*, a set of tools and techniques to prove performance properties of heuristics running in real-world conditions. It provides an alternative to queuing and control theory, which are typically too optimistic about performance because of their limited capacity to accurately model real-world phenomena. Overly optimistic analysis can lead to heuristic designs that fail in unexpected ways upon deployment. Rigorous proofs on the other hand, can not only inspire confidence in our designs, but also give counter-intuitive insights about their performance.

A key theme in our approach is to model uncertainty in systems using non-random, non-deterministic objects that cover a wide range of possible behaviors under a single abstraction. Such models allow us to analyze complex system behaviors using automated reasoning techniques. We will present automated tools to analyze congestion control and process scheduling algorithms. These tools prove performance properties and find counter-examples where widely deployed heuristics fail. We will also prove that current end-to-end congestion control algorithms that bound delay cannot avoid starvation.

Thesis Supervisor: Hari Balakrishnan
Title: Fujitsu Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Mohammad Alizadeh
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First, I would like to thank my advisors, Hari and Mohammad. I have learned a lot from both of them. In particular, Hari has taught me to focus on practical applications and to understand who benefits from my work. I also hope to learn how to communicate with his clarity and precision. Mohammad, through his advice and his other work, inspired me to seek solid truths even in the complexity of networked systems. He has also shown me that it is fruitful to stick with a research agenda for many years. Most of all, I would like to thank both of them for striking the right balance between letting me pursue my interests, stopping me from making mistakes and encouraging me when I needed it the most.

Next, I would like to thank my collaborators: Aditya Akella, Ahmed Saeed, Akshay Narayan, Anand Sarwate, Anirudh Sivaraman, Anup Agarwal, Ben Mikek, Devdeep Ray, Emily Marx, Frank Cangialosi, Hamsa Balakrishnan, Jehad Aly, Junchen Jiang, Karthik Gopalakrishnan, Mina Tahmasbi Arashloo, Neil Agarwal, Prateesh Goyal, Radhika Mittal, Rahul Bothra, Ravi Netravali, Ruben Martins, Saksham Goel, Scott Shenker, Srinivas Narayana, Srinivasan Seshan, Sudarsanan Rajasekaran, Vikram Nathan, Will Sussman, and Zili Meng. They have helped me in many ways, ranging from brainstorming ideas and adding new perspectives to making research a fun enterprise with friends.

I would like to thank my lab-mates and people of G9 for making me look want to come in to lab every day and still spend time with them outside of work.

Most importantly, I would like to thank my family. They have created, enabled and shaped every aspect of my life. No acknowledgments section can do justice to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The internet has evolved from being a novelty to being critical infrastructure. Yet its design does not reflect this new need. For instance networked systems rely on tens of heuristics to allocate resources to users and tasks. Examples include congestion control, CPU scheduling and load balancing, caching, wireless scheduling and cluster provisioning algorithms. While these heuristics work well most of the time, they exhibit pathological behaviors where their decisions severely degrade performance. As a result, networked systems today deliver unreliable performance, even when all components function as intended and ample resources are available to meet the demand.

Designing robust heuristics whose behavior we fully understand has been challenging. Designers often use queuing and control theory to understand heuristic behavior, a tradition that dates back to Kleinrock [84, 85]. This style of mathematical analysis is overly optimistic because it makes several simplifying assumptions. For instance, it assumes that traffic arrives as a stochastic (often Poisson) process, that packet service times are independent and identically distributed random variables or that overheads such as CPU context-switching and wireless channel setup costs are negligible.

Because of such assumptions, this type of analysis is only useful for obtaining coarse-grained guidance on system design or for explaining system behavior "after-the-fact". Heuristics must undergo several rounds of design iteration guided by empirical experiments before they are fit for deployment. The most widely deployed heuristics, such as congestion control and CPU scheduling, have undergone decades of iteration.

In spite of this, practically relevant performance issues are regularly discovered [95, 64, 14, 123].

It is worth noting that resource allocation heuristics themselves are often simple and can be described in tens of lines of pseudo-code. The complexity stems from the *emergent* behavior of the system.

This dissertation introduces *performance verification* as a viable alternative to queuing and control theory, to better bridge the gap between theory and practice. It incorporates two ideas that enable proofs to take a more active role in designing networked systems. First, it proposes a methodology to capture the complexity of real-world systems in mathematical models without requiring a detailed understanding of all the code and hardware in the system. The key idea is to prove worst-case properties over *non-deterministic* models of the environment. In contrast, queuing and control theoretic analysis of networked systems prove statistical properties over stochastic models.

Second, it uses automated reasoning techniques [35] to prove (or disprove) performance properties of resource allocation heuristics under these models. The complexity of the real-world makes purely hand-written proofs infeasible. Automated tools can help search through combinatorially many possibilities. It is thus able to use methods developed to verify the correctness of computer programs to verify the performance behaviors of networked systems.

We first developed performance verification to better understand the performance of internet congestion control algorithms. Later we realized that the technique is more general and applied it to other domains such as processor scheduling in operating systems and flow synchronization in machine learning clusters.

This dissertation is a first step toward developing performance verification. Chapter 12 outlines a future research agenda that aims to make performance verification more powerful and easy to use. We hope that it will help make networked systems of the future have predictable performance by becoming a standard part of their engineering workflow.

1.1 Traditional analysis is overly optimistic

To see where traditional analysis misses real-world phenomena to produce overly optimistic results, consider the classic Additive Increase, Multiplicative Decrease (AIMD) congestion control algorithm (CCA). Using performance verification, we discovered instances where AIMD's performance is much worse than classical analysis indicated.

1.1.1 Quick primer on congestion control, AIMD and simple network models

CCAs are distributed algorithms that decide how quickly a flow of data can be transmitted over a network. They need to send data fast enough to fully and fairly utilize available network capacity, yet not so fast that packets get delayed or dropped. CCAs typically run on the sender, an use acknowledgments (ACKs) sent by the receiver upon receipt of each packet to determine the sending rate. If the ACKs get excessively delayed or indicates that a packet is lost, the CCA infers that the network is congested and slows down.

The AIMD algorithm works by maintaining a congestion window (`cwnd`), which is a number that represents the maximum number of bytes that can be in flight at any point in time. After sending the first `cwnd` bytes of data, the sender can only send new packets if `cwnd` increases or when previously sent data is either acknowledged or determined to be lost.

It is common practice to characterize network paths using their "bandwidth-delay product (`BDP`)", which is the product of the bandwidth of the bottleneck link in the network path and the round trip time (`RTT`) of the network path in the absence of congestion. Under simplifying assumptions, the network is fully utilized when $\texttt{cwnd} > \texttt{BDP}$. However if $\texttt{cwnd} > \texttt{BDP} + \beta$, where β is the buffer size of the bottleneck link, the bottleneck link drops packets. Thus AIMD seeks to maintain `cwnd` between the `BDP` and $\texttt{BDP} + \beta$. To do so, it increments `cwnd` by one packet once every `RTT` (additive increase). If it detects that a packet was dropped, say because `cwnd` exceeded $\texttt{BDP} + \beta$, it halves the `cwnd` (multiplicative decrease).

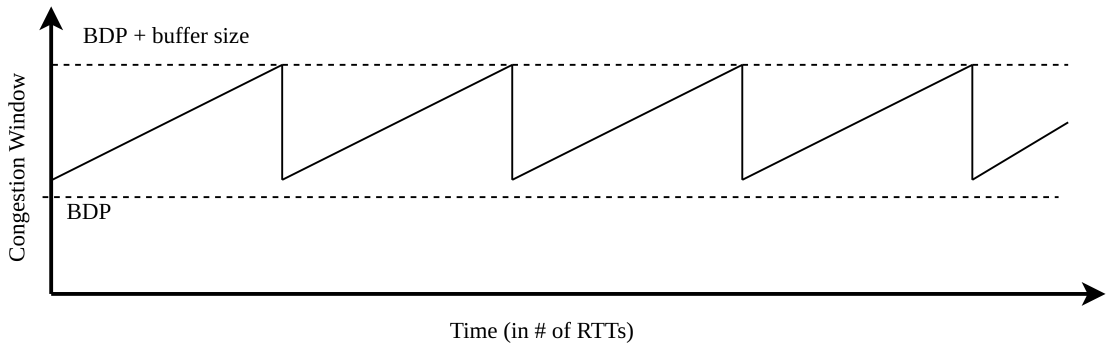


Figure 1-1: Time evolution of the `cwnd` of AIMD on an ideal link

1.1.2 AIMD on more complex networks

Traditional analysis of AIMD, like the one given above, ignores variations in packet `RTT` caused by reasons other than queuing at the bottleneck. Reality is often more complicated. For instance, wireless links like Wi-Fi use "frame aggregation" to improve MAC-layer efficiency by sending packets in bursts. These generate transport-layer ACK bursts. When ACKs arrive in a burst, however, we know that the sender will respond by sending new packets in a burst.[1] These bursts can cause premature packet drops, causing AIMD to reduce its window to be smaller than the `BDP` as shown in figure 1-2. This can cause it to under-utilize the link, defeating the purpose of the Wi-Fi mechanism.

A surprising corner-case

One way to capture this phenomenon is to say that loss happens not when `cwnd` exceeds $\texttt{BDP} + \beta$, but when it exceeds $\texttt{BDP} + \beta - \texttt{max\_burst\_size}$. However, this patch also proves insufficient. For instance, our formal analysis uncovered a problem where AIMD can send a burst of packets when it detects a large number of losses. Ideally, this is the time to be conservative and *avoid* sending a burst! Intuitively, one might expect that this cannot happen because AIMD would decrease its `cwnd` by

[1]Burst transmissions can happen even with a paced sender, because implementations (including the Linux kernel [110]) often pace packets faster than the estimated rate to avoid under-utilization.

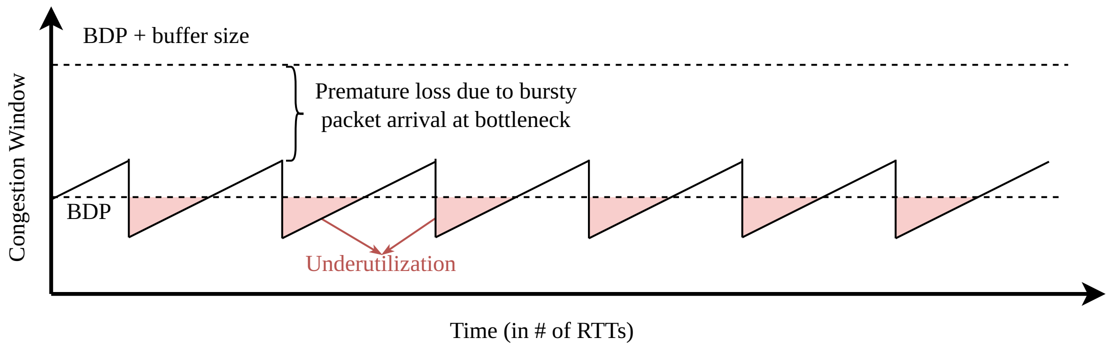


Figure 1-2: Time evolution of `cwnd` of AIMD on a bursty link

half. Our method generated a counterexample that sidesteps this mitigation. In the counterexample, loss happens in two bursts. When the sender detects the second burst of losses, the `cwnd` has already been halved once and will not be halved again since the losses belong to the same window of data [46]. Instead, the loss prompts a burst of packets from the sender, which triggers another loss. Through this mechanism, AIMD can drop its `cwnd` *twice* in quick succession, leading to utilization as low as 50% even when the buffer is as large as 2 BDP. In contrast, simple network models optimistically predict full utilization for any buffer size larger than 1 BDP. Although IETF RFCs have included mitigations for such situations [46], our counter-example sidesteps them. We discuss this counterexample in more detail in Section 6.2.

Prior mathematical methods of congestion control fail to characterize these behaviors because they ignore complex short timescale behaviors (i.e. shorter than an `RTT`). We find that short timescale behaviors strongly influence the performance of a CCA, sometimes degrading utilization by an order of magnitude (see §6.1, §6.2 and, §6.3). Our method of performance verification fills this gap in our understanding of such behaviors. It uncovers surprising CCA behaviors that the users or designers of the CCAs may not have anticipated. In addition, it helps prove bounds on the performance of CCAs, including on steady-state behavior.

1.1.3 Macroscopic effects of microscopic delay jitter

In the above examples, larger bursts lead to lower utilization. While large bursts, sometimes going up to 100 ms, are common on the internet, there are other scenarios where even a small delay jitter can have a large effect. For instance, suppose two AIMD flows are sharing a common bottleneck, but one of them is well-paced while the other sends packets in small bursts. This situation can occur with generic segment offloading (GSO) [1] used by the sender for CPU efficiency, WiFi frame aggregation [59], or delayed ACKs [72, 26]. As the queue gets nearly full, the flow that sends packets in bursts is more likely to lose packets. When this happens, this flow reduces its `cwnd` and the queue stops being full until a while later when again the bursty flow is more likely to lose packets. The figure below illustrates this using an ns3 [3] simulation for the Reno [65] (an instantiation of AIMD) and Cubic [61] congestion control algorithms. We also reproduced similar results in a Mahimahi [112] emulation.

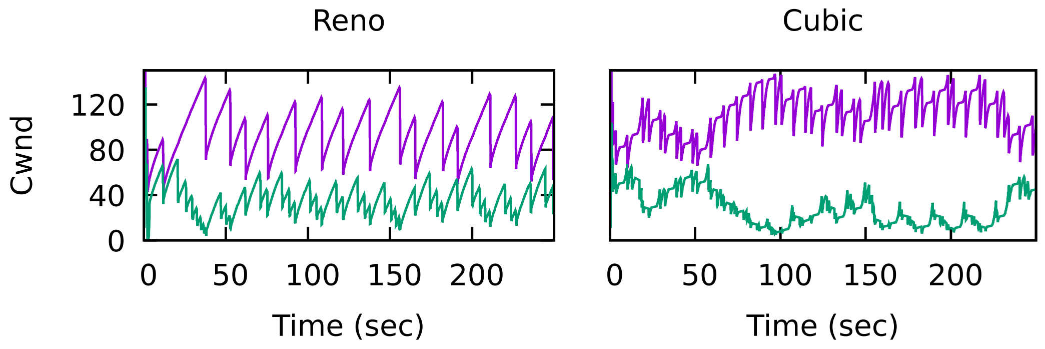


Figure 1-3: Congestion window evolution when two flows are run on a 6 Mbit/s, 60 ms link and 60 packets (1 BDP) of buffer. The lower flow's receiver uses delayed ACKs of up to 4 packets while the other ACKs every packet. The CCAs used are Reno (left) and Cubic (right). The ratio of throughput obtained between the two flows is 2.7× and 3.2× for Reno and Cubic respectively.

1.2 Performance verification overview

This dissertation introduces performance verification as an alternative to queuing and control theory to bridge the gap between theory and practice. We demonstrate that it can capture a multitude of real-world phenomena when applied to different heuristics to obtain results that until now we inaccessible to theory:

- In analyzing CCAs, it captures complex sub-RTT phenomena [12, 11].
- In analyzing classic scheduling algorithms such as work stealing and "shortest remaining processing time first" scheduling, it captures non-idealities such as context-switching costs and blocking tasks [56].
- In analyzing the Linux CPU load balancer, it found new performance bugs because it considers all possible ways in which tasks can be distributed across CPUs [56].

We also used the technique to not only analyze, but systematically *synthesize*, new CCAs that are provably performant by construction [5].

1.2.1 Modeling strategy

The key to capturing real-world complexities in a simple mathematical model is to use non-deterministic and *non-stochastic* models. Such models describe the *set* of ways in which the environment reacts to heuristic decisions. We want our heuristics to perform well under all behaviors in this set. Critically, in contrast to most prior work, it does not specify any probability distribution over that set. We made this choice because the real world involves many complex joint probability distributions that are hard to specify. For instance, some network routers send packets in large groups because that is more efficient. The size of this group depends on many factors, including link rate and the router's queue occupancy. Modeling this requires us to know the joint probability distribution of link rates, propagation delays, group sizes and more. Specifying only a set of behaviors and not the associated distribution greatly simplifies the modeling task. Further, we allow the model to be a *superset* of environment behaviors that we wish our heuristics to be robust against. Allowing the model to be larger can make it simpler to describe.

One can treat a non-deterministic model of the environment as an adversary. For instance, one could have an adversary that delays network packets in any pattern that it likes, subject to minimal constraints. If we can prove that a congestion control algorithm (CCA) works well in this model, then it works well under all conditions the adversary can emulate.

The key challenge, and the part where a system designer's insight is essential, is in determining how to restrict the adversary. The restrictions encode the assumptions we make about the environment. They must be loose enough that most real-world environments satisfy them. Yet they must be tight enough that a heuristic can provide meaningful guarantees, because if the adversary is too powerful (i.e. assumptions are too minimal), no heuristic can work. Chapter 4 discusses how we design an adversary to reason about congestion control. We have also developed a general strategy for designing such adversaries for other heuristics [56].

The model translates a system designer's instincts about the real world to formal mathematics. If done well, rigorous reasoning under the model will yield useful insights for the real world. Two criteria assess the success of the modeling effort:

Is the adversary too powerful?

The adversary should be weak enough that there exists some heuristic that offers worst-case performance guarantees under the adversary without compromising too much on the common/ideal case. For instance, in our work on congestion control, we designed a CCA that offers worst-case guarantees, with the only cost being that delay is increased by a constant additive amount.

In our work we have often found that, in our models, widely deployed heuristics fail to achieve properties that they were presumed to achieve. For instance, although designed to achieve reasonable inter-flow fairness, we proved that current methods to develop delay-bounding CCAs cannot always avoid starvation, an extreme form of unfairness. How do we tell whether this is merely an artifact of the model or whether such cases actually occur in practice? This requires a human expert to examine the adversary behavior that causes the heuristic to fail and determine how often the identified phenomenon might occur in the real world. For widely deployed CCAs, we found that starvation occurs when network delay variations due to real-world factors such as ACK aggregation and end-host scheduling cause different flows to estimate congestion differently. It was easy to reproduce starvation even in simple network scenarios.

If all identified issues have a real-world counterpart, the modeling effort has been successful in ensuring the adversary is not too powerful. Else, the adversary needs to be restricted further.

Is the adversary powerful enough?

This criterion is simpler to satisfy and is the main benefit of using a non-deterministic model over a stochastic one. We simply need to prove that the adversary can emulate each environment behavior we care about. Often we can do this for entire classes of behaviors. For instance, an adversary that can delay packets in any pattern it likes as long as the delay is bounded can capture *any* bounded probability distribution with arbitrary dependencies between packets. Often we can also prove that the model can capture arbitrary combinations of behaviors.

The hard part of this step is in identifying the types of behaviors that occur in the real world. Once identified, ensuring the model can capture them is typically easy.

1.2.2 The properties we want to prove

While stochastic models predict the average or tail (e.g. 99th percentile) statistics of throughput, delay etc., non-deterministic models predict worst-case behavior. These can be expressed as theorems. For instance, we could prove that a CCA moves toward the correct sending rate exponentially quickly no matter what the adversary does. Upon reaching steady state, we prove bounds on the CCA's worst case link utilization and self-inflicted delay.

For many widely deployed heuristics, these properties do not hold. In this case, worst case analysis can reveal weaknesses in these algorithms, giving insight into where improvements are needed.

Worst case predictions about performance are often qualitative, rather than quantitative. This can make them more insightful. For instance, we found instances where widely deployed CCAs can obtain arbitrarily low utilization or arbitrarily large unfairness when subjected to an adversary that can perturb packets by arbitrarily small amounts. These statements are true in the limit that the link rate goes to infinity. For constants that occur commonly in practice, we found that CCAs can be over $10\times$ away from optimal.

1.2.3 How we get computers to help us

To reason about a non-deterministic model, one must quantify over all adversary actions. This is made more complicated by the fact that, for practical reasons, heuristics tend to have mechanisms to handle corner cases (e.g. via "if conditions") which make them hard to reason about analytically. In this thesis, we frequently rely on automated reasoning to explore the combinatorially many adversary actions to prove and disprove properties about heuristics.

For this purpose, we use SMT solvers [35]. An SMT solver takes a first-order logic formula as input. It attempts to find an assignment to the variables of that formula such that the formula evaluates to `true`. If no such assignment exists, it outputs "unsat" (for unsatisfiable). We create a formula that has a satisfying assignment if and only if there exist adversary actions that can cause the heuristic to violate a given property. If the solver finds an assignment, it has "broken" the heuristic under our model. If it outputs "unsat", it has proven that no such adversary action exists.

Humans can use the solver interactively to explore questions about heuristics by creating variants of the original formula. They can modify the property, heuristic or the adversary to see where heuristics do and do not break. From this exploration, they can construct real-world scenarios where heuristics break or prove theorems about the heuristics' performance using lemmas proved automatically by the solver.

1.3 Summary of contributions

Performance verification

This dissertation uses techniques developed to verify the correctness of programs to verify the performance of resource allocation heuristics used in computer systems. Key to this is an approach to use non-deterministic, non-stochastic models of the environment that capture complex real-world behaviors in simple mathematical abstractions.

Congestion control

Over the past decade, numerous CCAs have been developed and deployed on the internet. However, they all experience unexpectedly poor performance on certain network paths, affecting users every day. In response, the research community and industry have developed many innovative methods to improve congestion control. Large teams of engineers in the industry work on this problem.

Congestion control is challenging because real-world network paths exhibit a wide range of complex behaviors like wireless frame aggregation and channel quality variations, policers, end-host scheduling jitter, delayed acknowledgments, and more. A key problem in CCA development is *evaluation*: how can developers, operators, and the networking community gain confidence in any given proposal? It is impossible even for seasoned engineers to contemplate the composition of every "weird" thing that could happen along a path, much less model or simulate these behaviors faithfully.

This thesis develops the method of performance verification to lead to a deeper understanding and use those methods to design better protocols. It makes three contributions:

- CCAC [12]: We discovered a simple mathematical description that captures the effects of these complex real-world phenomena and built a tool, CCAC, that can verify performance properties of CCAs.

Using CCAC we found previously unknown ways in which three widely deployed algorithms—AIMD, BBR and Copa—perform poorly. We were also able to use insights from the tool to modify BBR and prove that the modified version always converges to full utilization and bounded delay in our model. Facebook/Meta uses this version of BBR today for most of their user-facing traffic.

The proofs produced by CCAC are the first to include the complex network behaviors discussed above. Prior work on congestion control has only proved properties on links whose rate is either constant or described by a simple stochastic process.

- Starvation [11]: Encouraged that we could use CCAC to formally prove the existence of an algorithm that could achieve high utilization with bounded delays, we turned our attention to multiple flows sharing a bottleneck link, expecting to formally prove that this modified BBR achieved fairness. Surprisingly, the tool found that not only was it unfair, but that it could be arbitrarily unfair—one flow could obtain all the bandwidth, starving the other flows. We then found that none of the other prior delay-bounding end-to-end protocols, including the original BBR and Copa, could avoid starvation

  Intriguingly, we found a property shared by all known delay-bounding end-to-end CCAs: when run on ideal constant bit-rate links, they all maintain a constant queue size, making small oscillations (if any) about an equilibrium. While this looks like an intuitive and even desirable property, we proved that any CCA with this property will definitely starve on some network path.

  Does starvation occur in the real world? If so, why hasn't it been observed before? It is hard to detect starvation in the wild because no entity has the visibility needed to correctly assign blame between the CCA and a genuine lack of network capacity. This may be why it has not been noticed before. Nevertheless, starvation is easy to reproduce even in simple settings. For example, BBR starves when the flows have different propagation delays and the network has some delay-jitter of any kind. Both are extremely common on the internet.

- Copa [13]: Prior to working on CCAC, we designed a new CCA, Copa [13], which incorporates two key ideas. First, it includes a filter that helps it distinguish queuing delay at the bottleneck from other sources of delay variations. Second, it detects whether it is sharing a link with a non-delay-sensitive CCA that is playing by a different, and incompatible, set of rules and adapts by transitioning to a more competitive mode. Facebook/Meta has been using Copa for live video uploads since 2019 because it provides better performance for their workload [51]. Experience designing Copa showed us that the key challenge in congestion control is that it is not straightforward to infer congestion from packet delay and loss, and that the key confounding factor is the complex behaviors exhibited by real network paths. This motivated our later work on CCAC and starvation in congestion control.

Follow on work

I collaborated on two projects inspired by this thesis [5, 56]. The first [5] develops conceptual, mathematical and automated tools to explore the design space of congestion control algorithms. The goal is to find a CCA that provably achieves given properties under the CCAC network model. It uses program synthesis to make the computer output a design that is provably performant by construction. As part of the program synthesis, it uses the automated CCAC verifier to criticize candidate designs. Our key insight is a theorem that identifies a *complete* set of congestion signals for the CCA to monitor. It proves that if there exists *any* CCA that achieves good performance properties under our model, then there also exists a CCA that sets the rate as a pure function of the congestion signals we identified. Identifying the right congestion signals has historically been a major challenge in CCA design. Tens of candidates have been proposed in the literature [73, 27, 9, 13, 31, 38, 152], leading to ambiguity about the "best" choice. Given a non-deterministic network model, this theorem conclusively settles this question. Note, this project focuses on the case where there is only a single flow in the bottleneck, which is a common scenario in the internet [53, 98]. Overcoming our result on starvation in CCA remains interesting future work.

We initially developed performance verification for understanding congestion control, but later understood its generality and developed a methodology to apply it to other domains. In the second project [56], we undertook four case studies to demonstrate this approach. The first demonstrated our tool's ability to provide new insights into work-stealing schedulers by establishing bounds on their optimality, while considering context-switching costs. Context-switching costs are hard to incorporate in hand-written proofs. In the second case, we examined the Linux CFS load balancer's performance and identified a new bug that could potentially disrupt the work conservation property. Our third case study demonstrated the methodology's applicability to resource allocation problems beyond CPU schedulers by verifying a recent observation [121] that TCP unfairness can cause traffic synchronization. In our final study, we expanded the classical results of the Shortest Remaining Processing Time First algorithm (SRPT) to cases that allow task preemption and blocking.

Chapter 2

Related work

2.1 Congestion control

The algorithms

Congestion control has a long history, starting with the early loss-based TCP Tahoe [89] to the more principled AIMD algorithm in DECbit [34], TCP Reno [73], and NewReno [65]. These were improved upon for several years [45, 18, 148, 17, 102, 46, 124] to improve congestion detection/recovery and speed of convergence, culminating in the development and deployment of Cubic [61] and Compound [139].

Since then, link rates have increased significantly, wireless links (with their time-varying link rates) have become common, and the Internet has become more global with terrestrial paths exhibiting higher round-trip times (RTTs) than before. Faster link rates mean that many flows start and stop quicker, increasing the level of flow churn, but the prevalence of video streaming and large bulk transfers (e.g., file sharing and backups) means that these long flows must co-exist with short ones whose objectives are different (high throughput versus low flow completion time or low interactive delay). At the same time, application providers and users have become far more sensitive to performance, with notions of "quality of experience" for real-time and streaming media [16, 136], and various metrics to measure Web performance being developed [57, 58, 10, 24, 111]. Many companies have invested substantial amounts of money to improve network and application performance. Thus, the performance of congestion control algorithms, which are at the core of the transport protocols used to deliver data on the Internet, is important to understand and improve.

Like TCP Reno and NewReno, the Cubic and Compound algorithms are primarily loss-based, and increase queue occupancy until a loss occurs. When buffers on the path are large compared to the bandwidth-delay product (BDP), they can cause excessive delay, a phenomenon known as “buffer-bloat” [54]. This motivated the development of Active Queue Management (AQM) [47, 113, 118] and delay-based CCAs [27, 143, 9].

Modern congestion control research has evolved in multiple threads. One thread of research has focused on important special cases of network environments or workloads, rather than strive for generality. The past few years have seen new congestion control methods for datacenters [6, 7, 8, 120], cellular networks [145, 152], Web applications [42], video streaming [55, 93, 109], vehicular Wi-Fi [41, 96], and more. The performance of special-purpose congestion control methods is often significantly better than prior general-purpose schemes.

Another thread has been led by Google in developing BBR [31, 33], a hand-crafted CCA refined through real-world testing. A fourth thread of end-to-end congestion control research has argued that the space of congestion control signals and actions is too complicated for human engineering, and that algorithms can produce better actions than humans (e.g., Remy [144, 127], PCC [38, 39, 104], and deep learning-based methods [150, 75, 76, 4]). These approaches define an objective function to guide the process of coming up with the set of online actions (e.g., on every ACK or periodically) that will optimize the specified function. Remy and the deep learning methods perform this optimization offline, producing rules that map observed congestion signals to sender actions. PCC and its variants perform online optimizations.

Despite the rich history and research into congestion control, a robust algorithm that performs well across a diverse array of network paths remains elusive. This dissertation aims to address this gap by developing theoretical tools for systematically evaluating CCA performance across diverse paths.

Analysis of the algorithms

Congestion control has been subject to extensive theoretical analysis using both deterministic and statistical models of the network [34, 103, 129]. For instance, recent work has analyzed the ability of delay and ECN to create unique fixed points that flows can converge to and studied their stability once converged [156, 138]. These papers prove, for instance, that a CCA that converges to a time-invariant delay that does not change with the number of senders cannot be fair. Another body of work examines which CCA properties can be achieved simultaneously [153, 154]. While prior work has analyzed CCAs on paths with non-congestive packet loss, to our knowledge this is the work with a theoretical analysis of paths with non-congestive delay jitter.

A related line of work is on formal verification of the *implementation* of a CCA given a specification of the algorithm using manual [23] and automated [135, 132, 133, 134] techniques. This line of work identifies bugs in the implementation of an algorithm but does not make any statements on the performance of the studied algorithm. For instance, a verification of the implementation can try all patterns of packet losses to see if an AIMD implementation drops its window on loss. It does not, however, answer the query of whether it *should* drop its window on loss.

Our path model introduced in chapter 4 is similar to the ones developed in network calculus [91, 25]. We adapt those ideas to create a model well-suited to analyzing CCAs. We ensure that our model is expressive, while avoiding behaviors that no CCA can handle. Our approach differs from traditional network calculus in a key aspect; for us, the rate at which packets are sent into the network is a function of the network's past behavior. This models the closed-loop control created by CCAs. However, it complicates analysis and precludes the use of standard network calculus techniques. To manage the complexity, we use an SMT solver. Two prior works use network calculus to simulate [82] and theoretically analyze [15] CCAs. Their goal is to make the analysis of a simple *deterministic* network easier. In contrast, we use a non-deterministic path model, giving us many degrees of freedom to expose unexpected CCA behavior.

2.2 Formal verification

Formal verification is a field that seeks to ensure the correctness and reliability of computer hardware and software programs by writing human and/or computer interpretable proofs of their behavior. The foundations of the field were set in the 1960s and 70s by the works of Dijkstra [37], Hoare [63], King [83] and Floyd [43].

Later emphasis shifted to automated proof checking and writing to deal with the complex of real software and hardware. Today we have mature proof assistants [20, 114, 115, 36], SAT and SMT solvers [35, 130, 19] and high-level interfaces to these solvers [90, 117, 122, 92, 137, 90, 122].

Most of this work has focused on questions of *correctness*: "Does the program produce the correct output?" "Does the program terminate?" "Is the distributed guaranteed to make progress?". This thesis applies these tools to questions of heuristic *performance*. The key technical advancement is in posing them as sharply defined yes/no questions that faithfully represent real-world complexities. Previous work has either oversimplified the mathematical analysis or used purely empirical measures of performance that do not capture all circumstances that may occur in practice.

Two threads of research seek to verify performance. However the questions they seek to answer are different than ours. One thread seeks to quantify the amount of resources (CPU cycles, memory bytes etc.) used by a computer program [66, 86, 67]. Another thread seeks to verify whether the resources available in a system are sufficient for a certain mix of tasks, often motivated by proving tight bounds on real-time system performance [97].

Formal verification has been applied to computer networking as well [87, 52, 21, 22], but again in answering questions of correctness such as "Are all endpoints reachable with this routing mechanism?", "Are all packets arriving from the outside guaranteed to first traverse this node?" or "Is this network cycle-free?".

Chapter 3

Overview of our work on congestion control

Congestion control is hard because real-world network paths exhibit a wide range of complex behaviors due to token-bucket filters, rate limiters, traffic shapers, network-layer packet schedulers with various artifacts, link-layer schedulers that vary link rates, physical-layer vagaries, link-layer acknowledgment (ACK) aggregation, higher-layer ACK compression or aggregation, delayed ACKs, and more. CCAs estimate congestion using packet loss and delay. However due to the diversity of paths, it is hard to separate delays and losses that happen due to congestion from those that occur for other reasons.

This thesis develops techniques to study the impact such paths can have on congestion control performance and shows surprising ways in which widely deployed CCAs fail even in simple scenarios.

3.1 Verifying the performance of congestion control

The process of evaluating and gaining confidence with a CCA today involves some combination of simulation [2, 3], prototype implementation with tests on a modest number of emulated [30, 112, 62] and real-world paths [150, 149], and, in some cases, empirical analysis via controlled A/B tests at large content providers. Simulations and small-scale tests are invaluable in the design and refinement stages, but provide little confidence about performance on the trillions of real-world paths.

If one has access to servers at a large content provider, then A/B tests are feasible where a new CCA can be tried on a fraction of the users to compare its performance with another scheme. If the measured results of the new CCA compare well, it increases confidence in its behavior, but still does not guarantee that it will perform well in all scenarios. Moreover, as is likely, the new CCA will not perform better in the A/B tests for all users. The aggregate results of an A/B test may hide significant weaknesses that arise in certain cases. When such cases are identified, understanding the behavior of a CCA requires a massive data analysis, which may be futile because the operator might not have visibility into the network conditions that led to poor performance. We also note that most of the community does not work at a "hyper-scaler" with access to such a live-testing infrastructure, yet has good ideas that deserve serious consideration.

3.1.1 Overview of CCAC

To better understand CCA behavior in the presence of ambiguity in congestion signals, we have developed the Congestion Control Anxiety Controller (CCAC), pronounced "seek-ack" or "see-cack". CCAC allows the user to express a CCA in first-order logic, and a hypothesis about performance properties as logical formulae. Then, it uses an automated prover to do the bulk of the work. The theorem prover either proves that the performance property always holds under CCAC's path model, or generates counterexamples invalidating the hypothesis.

The path model. The model abstracts complex network paths by a single *path-server* with a FIFO queue followed by a fixed delay, leveraging ideas from Network Calculus [25, 91]. The path-server introduces variable delay per packet. It can choose when to transmit or drop packets in its queue subject to certain constraints. The constraints seek to emulate a link with a fixed *average* capacity and a limited buffer, but allow for short-term deviation from this average including an arbitrary per-packet delay jitter up to D seconds (§4.2.1). The user can use this model to reason about

CCA specification:
$\bigwedge_t((\ \ loss\_detected_t \land can\_decr_t) \Rightarrow \texttt{cwnd}_t = \frac{1}{2}\texttt{cwnd}_{t-1})$
$\bigwedge_t((\neg loss\_detected_t \land can\_incr_t) \Rightarrow \texttt{cwnd}_t = \texttt{cwnd}_{t-1} + \alpha)$
Query:
$\bigvee_t(loss\_happened_t \land \texttt{cwnd}_t \leq 1\text{BDP})$
buffer size = 2 BDP, max. jitter = 1 RTT

Figure 3-1: Query used to discover instances where AIMD can cause loss prematurely. *can_incr* and *can_decr* prevent AIMD from increasing/decreasing more than once per RTT.

variable capacity links as well (§4.3).

This model follows our performance verification philosophy of using non-deterministic non-random models (see §1.2.1). As a result, it captures a wide range of underlying network-specific behaviors. Non-determinism helps here for three reasons. First, the probability distribution of link delays on the internet is unknown, ever-changing, and depends on one's vantage point. Second, many in-network processes such as ACK aggregation and token-bucket filters are deterministic, not random. Third, we are often interested in the performance on the long tail of low-quality links (paths), whose behaviors would be buried by aggregate metrics. To our knowledge, this paper presents the first analysis of worst-case CCA behavior in such an expressive model.

Input to CCAC. A CCAC user expresses their CCA and properties of interest as first-order-logic formulae. In principle, such formulae can represent any circuit, but CCAC's ability to reason about it depends on how the user expresses the CCA. Thus, the user's ingenuity is essential in expressing the CCA in a way that enables automated reasoning. The user expresses a CCA as a function that maps past ACKs to a `cwnd` and/or rate. Further, they express a property of the CCA as arbitrary constraints on the `cwnd`, utilization, delay, or packet loss, connected by logical operators. Examples of hypotheses (queries) include: "Does `cwnd` ever fall below 90% of `BDP` in 8 RTTs?" and "Is there any case where the queue length starts below 1.5 `BDP`, but increases beyond 1.5 `BDP` within 20 RTTs?"

In §5, we show how we encode the description of the path-server, the CCA, and hypotheses about the CCA as a Satisfiability Modulo Theories (SMT) problem, allowing CCAC to use Z3 [35] as its automated theorem prover.

CCAC's Operation. Typically, the user will formulate queries to capture bad CCA behavior. CCAC searches through all possible behaviors of the path-server, subject to its constraints, to find a network trace where the CCA fails to achieve the property. If a trace exists, CCAC outputs it. Otherwise, CCAC outputs "unsat" informing the user that no such bad behavior exists, which proves the user's hypothesis about the CCA. When the user finds bad behavior using CCAC they can (a) live with it, (b) improve the algorithm so that the bad behavior no longer exists, or (c) restrict the path-server's model to exclude the cases that trigger the bad behavior. In the last case, the user will know what additional constraints on the network are necessary for the CCA to work.

The user repeats this process until they are confident that they understand the bounds on the performance of their CCA. However, CCAC can only make statements over finite time horizons when network parameters such as the link rate and propagation delay are constant. This does not prove the CCA always exhibits good behavior over arbitrarily long time horizons with varying network parameters (e.g., varying link capacity). Nevertheless, the user can prove these general long-term theorems by mathematical induction over lemmas that CCAC *can* prove (see §6.2.2 for an example).

Example. Figure 3-1 shows the CCAC query used to discover the behavior of AIMD discussed earlier. CCAC evaluates the behavior of algorithms over a finite number of time steps. The behavior of the algorithm is defined at every time step; if loss (congestion) is detected and it has been an RTT since the last reduction in `cwnd`, then halve `cwnd`. If no loss is detected for an RTT, then increase `cwnd` by α. The query asks if loss can happen when `cwnd` is less than or equal to 1 BDP. CCAC answered "yes" to this query and produced traces that contained the behaviors we discussed for AIMD earlier, which we elaborate on in §6.2.

Assumptions. Because CCAC focuses on worst-case behavior, a central goal of its design is to exclude excessively antagonistic behavior that no CCA can handle.

Thus, the path model only includes a carefully chosen subset of paths (see §4.2.1). Our model of TCP timeouts is different from the standard for the same reason (see §4.2.2). We represent network state using cumulative functions that preclude the modeling of packet reordering. CCAC's mechanism to detect packet losses emulates endpoints that use an unbounded number selective acknowledgment (SACK) blocks. This corresponds to the QUIC protocol [71], while TCP is limited to a maximum of four SACK blocks [102].

3.2 Starvation in congestion control

The above results apply when the flow we analyze is not sharing its bottleneck queue with any other flow. This scenario is common on the internet due to extensive deployment of traffic shapers and fair queuing [53, 98]. Nevertheless, multiple flows do often share the same bottleneck queue too. We study this next.

CCAs are designed to cooperate and achieve reasonable fairness when multiple flows share the same bottleneck queue. Due to network jitter, we expected some degree of unfairness and hoped to prove bounds on it as a function of the delay jitter. For loss-based CCAs, unfairness was indeed bounded. However loss-based CCAs have significant weaknesses, which motivated the development of delay-bounding CCAs (see §2.1).

What CCAC found for delay-bounding CCAs was surprising. Although designed to be cooperative and achieve reasonable fairness, for every delay-bounding CCA that we studied, CCAC found scenarios where they experience starvation, an extreme form of unfairness. At first, we suspected the path-model we used was too adversarial; that such scenarios would never occur on the internet. However closer inspection revealed that starvation occurs under fairly simple and common scenarios on the internet. Next we tried simple modifications to existing CCAs in an attempt to mitigate scenarios identified by CCAC. None of these worked.

Thus we began suspecting that there may be a fundamental difficulty for delay-bounding CCAs to avoid starvation. In characterizing this difficulty, we identified a common property, *delay-convergence* that is shared by all delay-bounding CCAs we are aware of. On ideal network paths with a constant bottleneck rate and propagation delay, a delay-convergent CCA eventually converges to a small delay range and oscillate within that range. We proved that for all delay-convergent CCAs the exist network scenarios where they experience starvation.

The proof identifies an important consequence of delay-convergence. Because most CCAs attempt to work across many orders of magnitude of rates, they *must* map a large rate range into a small delay range. Thus, even small changes in estimated queuing delay would induce enormous changes in the inferred rate. This observation suggested to us that perhaps bandwidth may not be shared equitably unless delays were perfectly measurable.

However network jitter prevents CCAs from being able to measure the congestive component of delay perfectly. CCA designers use a variety of techniques to attempt to distinguish queuing from non-queuing (non-congestive) variations, including averages (Vegas, FAST, BBR) [27, 143, 31], minimums (LEDBAT, Copa) [13, 125] and maximums (Verus) [152] of RTT, maximums of rate (BBR) [31], and repeating experiments (PCC) [38]. However the problem is inherently difficult due to the wide variety of delay patterns observed in real-world paths. Our work shows that tiny imperfections in estimating congestion can lead to large amounts of unfairness.

It is important to note that starvation is a strong form of unfairness, going well beyond traditional notions of RTT unfairness or even one flow getting a constant factor higher throughput than the other. We prove that there is *no finite* $s \geq 1$ where the faster flow will always get less than s times the throughput of the slower one. To demonstrate that these results are not contrived or hypothetical, we use insights from the proof to show empirical scenarios with BBR, Copa, and PCC where starvation occurs — simple topologies with equal propagation RTTs where the ratio of throughput between two flows is $10:1$.[1]

This result sounds pessimal for delay-bounding CCAs, so the question is whether we are doomed to choose between bounding delays and avoiding starvation. We

[1] This ratio would be higher but for limitations of our link emulator.

discuss how we might be able to achieve both desirable goals by being explicit about non-congestive delays in CCA design, ensuring that the CCA's delay variations in equilibrium are at least half as as large as the non-congestive jitter expected along a path. If that is not the case, then our results prove that starvation is inevitable.

3.2.1 Delay-convergence

Informally, our theorem states that all delay-convergent CCAs will starve. Now we define *delay-convergence*, a property that is common to most (if not all) delay-bounding CCAs developed to date, despite their many operational differences. They all seek to converge to a fixed sending rate and queuing delay, making only small (if any) oscillations about that point. This design pattern offers two benefits. First, stable sending rates provide stable performance for the application. Second, many schemes, either implicitly or explicitly, map sending rates to corresponding (inferred) queuing delays, which makes it easier to reason about their behavior. As a result, many end-to-end delay-bounding CCAs employ this strategy including Vegas [27], FAST [143], Sprout [145], BBR [31], PCC Vivace [39], Copa [13], PCC Proteus [104], and Verus [152].

We define a delay-convergent CCA based on how it behaves when a single flow runs on an *ideal path.* An ideal path has a constant bottleneck link rate C, minimum RTT (round-trip propagation delay) R_m, and a bottleneck queue large enough that overflows never occur. Naturally, such a large enough queue size only exists for CCAs that bound their maximum queue.

Definition 1. *A CCA $\mathcal{A}$ is* ***delay-convergent*** *if two conditions hold when it is run on an ideal path with a given R_m:*

1. *There is a time T after which the RTT experienced is always in a bounded interval $[d_{\min}(C), d_{\max}(C)]$, where C is the bottleneck rate. Let $\delta(C) \triangleq d_{\max}(C) - d_{\min}(C)$.*

2. *Both $d_{\max}(\cdot)$ and $\delta(\cdot)$ are bounded for C not approaching zero, i.e., there exists*

a link rate $\lambda > 0$ *and finite bounds* $d^{\max}$ *and* $\delta^{\max}$ *such that* $d_{\max}(C) < d^{\max}$ *and* $\delta(C) < \delta^{\max}$ *for all* $C > \lambda$.[2]

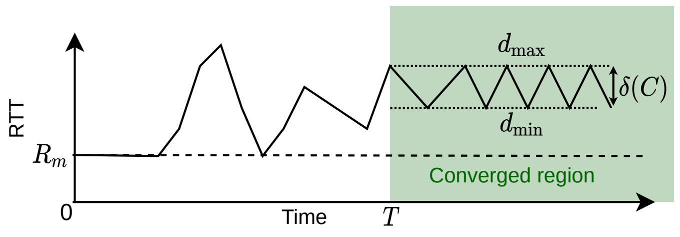


Figure 3-2: Ideal-path behavior of a delay-convergent CCA, $\mathcal{A}$.

Figure 3-2 depicts this definition, showing the time evolution of the RTT for a hypothetical CCA, $\mathcal{A}$. As we will see, CCAs that ensure a smaller $\delta^{\max}$ — and hence are "more convergent" — are more susceptible to starvation. In particular, we prove that starvation can occur if the delay *ambiguity* caused by non-congestive jitter delay is $> 2\delta^{\max}$. For many CCAs, $\delta^{\max}$ is small because they strive to keep delay variations small compared to R_m. Hence even a little ambiguity can cause starvation.

For instance, $\delta(C)$ is 0 for Vegas, FAST, and BBR in `cwnd`-limited mode; $\frac{4\alpha}{C}$ for Copa, where α is the packet size ($\frac{4\alpha}{C} < 0.5$ ms when $C > 96$ Mbit/s and $\alpha = 1500$ bytes); $R_m/4$ for BBR in pacing mode (see §8.2); and $R_m/20$ for PCC Vivace. We show that all these protocols suffer from starvation even in simple two-flow scenarios with small non-congestive jitter.

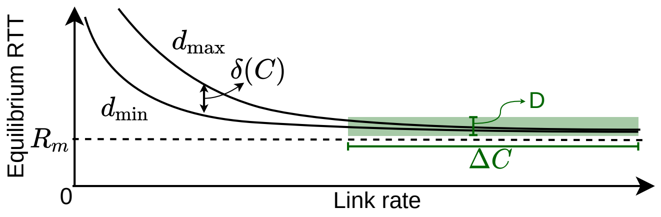


Figure 3-3: The rate-delay graph for a hypothetical delay-convergent algorithm. The C of the ideal path varies while its R_m is fixed.

For a fixed R_m, a delay-convergent CCA maps each link rate C to a delay range in steady state. Figure 3-3 shows how this range may vary as C changes. $d_{\max}(\cdot)$ and

[2]A finite $d^{\max}$ guarantees that a CCA is delay-convergent. We still discuss $\delta^{\max}$ separately because it controls how susceptible a CCA is to starvation.

$d_{\min}(\cdot)$ are typically non-increasing functions of C because CCAs typically increase their rate as the RTT decreases. However, our results do not require that the delay bounds are monotonic functions of C.

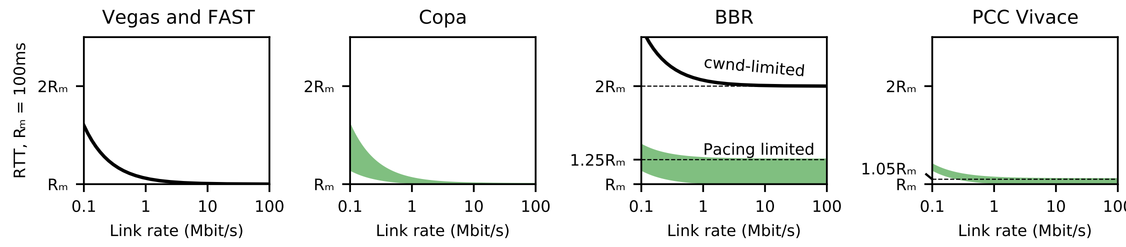


Figure 3-4: How various delay-bounding CCAs map delay to sending rates for various CCAs. The shaded region shows d_{max} and d_{min} for each algorithm. BBR has two modes (shown). The region's width is $\delta(C)$. For Vegas, FAST and BBR (`cwnd` limited), $d_{max} = d_{min}$, hence it is a line, not a region. For them, $\delta(C) = 0$. Section 8 describes how these mappings arise from the dynamics of these algorithms. Delay increases as $C \to 0$ for all CCAs since a transmission delay of $1/C$ is unavoidable.

Figure 3-4 shows the rate-delay graphs for some real-world delay-convergent CCAs. For all these CCAs, $\delta(C)$ is small (or grows smaller with C).

3.2.2 Example of starvation

Chapter 7 formalizes and proves our result. Here, we describe an example that gives insight how non-congestive delay can lead to starvation. Consider a CCA such as Vegas or FAST that seeks to maintain α packets in the queue *per flow* in equilibrium. α is a parameter of the algorithm (*e.g.* $\alpha = 4$ packets). Over an ideal path, once the CCA hits this target, its rate stabilizes and the queue length never changes. Hence $\delta_{max} = 0$, and the RTT is $R_m + \alpha/C$ as shown in Figure 3-4.

The issue is that, to achieve this equilibrium, the CCA must measure the queuing delay with high precision. For example, with $\alpha = 4$ and each packet being 1500 bytes, $\alpha/C = 0.5$ ms for $C = 96$ Mbit/s and 0.05 ms for $C = 960$ Mbit/s. Thus a difference of only 0.45 ms in the estimated queuing delay can cause the CCA to vary its sending rate by 10×! Therefore, if the delay measurement ambiguity exceeds this amount, it can easily cause severe unfairness or starvation.

This problem is not limited to Vegas or FAST or even CCAs with decreasing rate-delay functions as shown in Figure 3-3. Any delay-convergent CCA that seeks to bound delay variation below the level of jitter (delay ambiguity) of the network will suffer from the same problem. At a high level, the fundamental issue is that very different link rates are consistent with similar delay measurements for such a CCA. By applying different non-congestive delays to the paths of different flows, the adversary can cause the flows to perceive very different link rates and drive them to starvation.

3.3 Copa: a new delay-based CCA

Before we worked on formally understanding CCA performance, we designed Copa, a delay-based CCA that emphasized correctly interpreting delay measurements in the presence of non-congestive delay. It incorporates three ideas. First, it shows that a *target rate* equal to $1/(\delta d_q)$, where d_q is the (measured) queuing delay, optimizes a natural function of throughput and delay under a Markovian packet arrival model. Second, it adjusts its congestion window in the direction of this target rate, converging quickly to the correct fair rates even in the face of significant flow churn. These two ideas enable a group of Copa flows to maintain high utilization with low queuing delay. However, when the bottleneck is shared with loss-based congestion-controlled flows that fill up buffers, Copa, like other delay-sensitive schemes, achieves low throughput. To combat this problem, Copa uses a third idea: detect the presence of buffer-fillers by observing the delay evolution, and respond with additive-increase/multiplicative decrease on the δ parameter. Experimental results show that Copa outperforms Cubic (similar throughput, much lower delay, fairer with diverse RTTs), BBR and PCC (significantly fairer, lower delay), and co-exists well with Cubic unlike BBR and PCC. Copa is also robust to non-congestive loss and large bottleneck buffers, and outperforms other schemes on long-RTT paths.

Facebook/Meta uses Copa for live video uploads because Copa improves their end-to-end application metrics [51]. Chapter 11 describes the algorithm in detail.

Chapter 4

Congestion control path model

In section 1.2.1, we discussed our performance verification methodology, which advocates a non-deterministic model of the environment. In this chapter, we delve into our model to analyze congestion control. To make it easier to understand, we initially present a simpler model in section 4.1. This simplified model assumes that all buffers on the network path have infinite capacity. Later, in section 4.2, we expand the model to account for finite buffers. The models in this chapter only deal with a single flow. Chapter 7 extends them to multiple flows.

4.1 A simple model that assumes infinite buffers

Our simple model assumes that buffers on the network path are infinitely large and excludes other factors that may cause packet loss. Thus, every packet is acknowledged without fail. The focus of this model is to determine the delay, specifically the Round Trip Time (RTT), between when a packet is sent and when the sender receives its acknowledgment (Ack). To accomplish this, the model divides the RTT into three components, with particular emphasis on distinguishing between congestive and non-congestive sources of delay variation.

1. *Congestive (bottleneck queuing) delay*, which is the sum of the queuing delay incurred by packets waiting to be sent on the bottleneck link and the transmission time over the bottleneck link;

2. *Minimum packet propagation RTT (or delay)*, denoted R_m, defined as the time it takes for a single packet to traverse the non-bottleneck portions of the path and for the sender to receive an ACK (this includes both the speed-of-light propagation delay of the entire path and the packet transmission delays on each link except the bottleneck);

3. *Non-congestive delay*, which we define as *jitter* delays due to network elements (perhaps also at the bottleneck) that may temporarily hold packets or ACKs but are not by themselves persistent rate bottlenecks.

We model these delays as shown in figure 4-1:

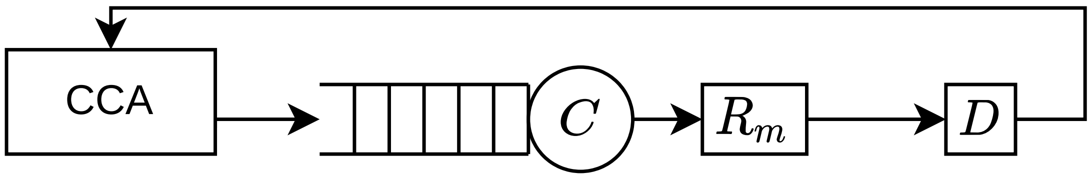


Figure 4-1: The simple network model of the loop between the sender and receiver.

The packets transmitted by the CCA are directed to an infinitely large First-In-First-Out (FIFO) queue, which represents the congestive delay. These packets are dequeued from the queue at a steady rate of C bits per second. Additionally, they experience a minimum packet propagation RTT of R_m.

After leaving the queue, the packets encounter a component that introduces a non-deterministic, bounded delay. This delay captures the non-congestive delay experienced along the network path. Specifically, this component can delay packets by any duration between 0 and D seconds, without altering their order.

After this stage, the packets are acknowledged to the CCA at the sender's end.

Examples of non-congestive delay include packet aggregation (which leads to ACKs arriving in bursts), overhead from end-host or in-network scheduling, and timing variations resulting from hardware offloading. In practical scenarios, non-congestive delay can vary widely. It may range from hundreds of microseconds caused by operating system thread scheduling [106] to tens of milliseconds induced by link-layer technologies like Wi-Fi [59]. Each of these factors introduces unpredictable fluctuations, leading to variations in the RTT. To complicate matters further, paths can include multiple such sources of non-congestive delay.

Is the model powerful enough?

With its ability to non-deterministically introduce bounded delays, it can emulate a wide range of behaviors. Let us consider a concrete example to illustrate this point.

Many wireless devices employ frame aggregation, where packets like acknowledgments (ACKs) are combined into larger bursts for more efficient transmission. To demonstrate, let us assume the following algorithm: frames will not be forwarded from the queue for a specific wireless destination device until it either contains at least 10 KBytes of payload or the oldest payload component has been waiting for over 10 milliseconds. Once this condition is met, all packets in the frame for that destination are promptly forwarded. This algorithm ensures that packets are forwarded at the maximum rate achievable for that particular destination until the queue holds fewer than 10 KBytes.

Our model emulates this behavior as follows. Most of the waiting time in the algorithm is modeled by the FIFO queue. When the queue contains less than 10 KBytes, the packet waits for a maximum of 10 milliseconds. This part of the waiting time is modeled by the non-deterministic delay component as long as $D > 10$ milliseconds.

In addition to emulating individual behaviors like this, we want our model to also be capable of emulating combinations of these behaviors. Instead of proving our simple model can emulate these, we shall directly prove this for our primary model in section 4.2.

The adversary must be restricted enough to ensure that there exists some CCA that works well against it. If the adversary possesses excessive power, CCAs would be unable to make any assumptions about the network, rendering them ineffective. Therefore, we impose a constraint on the non-congestive delay, bounding it by a constant value, D. This constraint also aligns with the natural behavior of network components, which are designed to forward packets rather than to hold on to them indefinitely. Thus, unless a packet becomes trapped in a congested queue, network components will forward it within a finite timeframe (i.e. D seconds).

4.2 The full model that allows for finite buffers

The model presented in this section serves as the foundation for all the analysis conducted in this dissertation. The key improvement over the simpler version (see §4.1), is that it allows for network components with finite buffers while also allowing every behavior allowed by the simpler version. However, it is worth noting that the simpler version still suffices to understand most of the subsequent content in this dissertation. With it, one can even gain an intuitive understanding of the counter-examples uncovered by our automated tool (CCAC) that involve finite buffers.

To create a model that captures a broad range of network behaviors, we ensure it satisfies two properties. First, it can emulate *known* behaviors such as link-layer aggregation, token-bucket filters, and arbitrary per-packet delay up to D seconds (see §4.2.1). Arbitrary delays can be used to emulate scheduling errors, MAC scheduling artifacts, and delay-measurement errors. Second, it *composes*; i.e., for any two path-servers, there exists a path-server, perhaps with different parameters, that can do anything that the two path-servers can do when placed serially. Hence, the path-server can also emulate any sequential composition of the above behaviors.

Our initial attempts at creating a general path model produced models that were "too expressive" and allowed behaviors that no CCA can handle. We discuss some of these behaviors and how our final model avoids them in §4.2.1. We believe we have struck a good balance between expressiveness and restricting unreasonable behavior because CCAC produces network behaviors that are plausible on real networks.

Intuition. The path-server can be described as a generalized token-bucket filter (TBF) as shown in Figure 4-2. First, consider a standard TBF. It has two queues, one for packets and another for tokens. Tokens arrive continuously at a fixed rate of C bytes/second and a packet of size x bytes can be dequeued only if the token

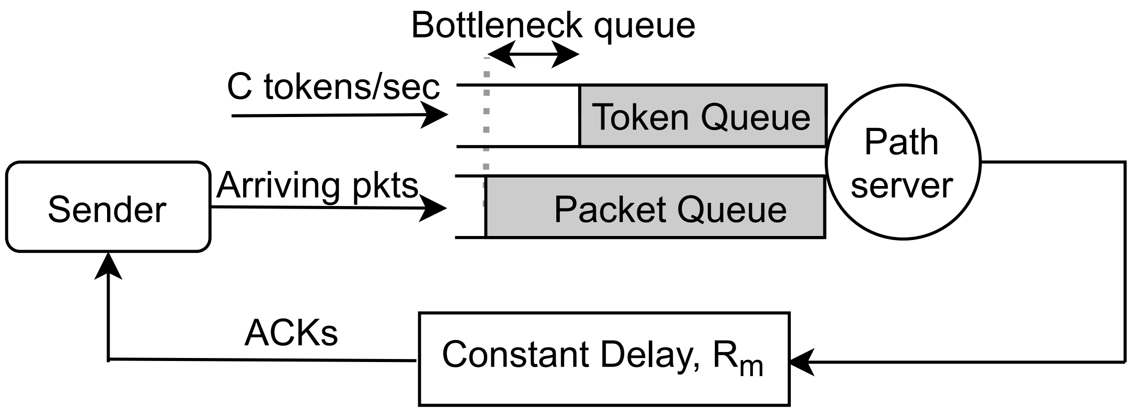


Figure 4-2: CCAC's path model.

C – link rate	R_m – propagation delay
β – buffer size	D – max per-packet jitter
α – MTU size	*dupacks* (threshold) – 3α
T – number of time steps	$\texttt{cwnd}(t)$ – congestion window
$Q(t)$ – packet queue length	$T(t)$ – token queue length
$A(t)$ – cumulative arrivals	$S(t)$ – cumulative service
$W(t)$ – cumulative waste	$L(t)$ – cumulative losses
$L^d(t)$ – cum. losses detected	$\tau_o(t)$ – timeout happened

Table 4.1: Glossary of symbols.

queue has at least x bytes worth of tokens. If no packets arrive for a while, tokens accumulate to a maximum of K bytes. Any tokens that come after the token queue is full are *wasted*. Then, if packets arrive in a burst, the TBF can transmit a burst of up to K bytes at once, temporarily exceeding the link rate C.

Our path-server generalizes the standard TBF. When a token arrives at the token queue, the path-server can non-deterministically choose to either admit it or waste it. Wasting a token is allowed only when there are more tokens than the total number of bytes in the packet queue. Tokens represent transmission opportunities at the bottleneck. Wasted tokens represent transmission opportunities that were wasted because there were packets at the bottleneck.

Like a TBF, when a packet is dequeued, a number of tokens equal to the packet's size are dequeued. However, unlike a TBF, the path-server can *choose to delay sending packets even when tokens are available*, subject to the constraint that once a token is admitted, the token cannot be in the queue for more than D seconds. This represents the fact that transmission opportunities may appear non-periodically. There may be up to D seconds of jitter, which the the path-server can use to emulate various network effects. One corollary of the D-second bound is that the path-server can never accumulate more than $C \cdot D$ tokens.

Recall that the path-server models an entire path, not just the bottleneck. Therefore, the packets in the path-server's queue represent packets enqueued throughout the path, not just at the bottleneck. The bottleneck queue is represented by the difference between the total number of bytes in the packet queue and the number of available tokens. Hence, the server can waste tokens only if this difference is < 0, indicating that the bottleneck is empty.[1] To emulate a bottleneck buffer of β bytes, the server drops packets when the bottleneck queue exceeds β bytes. One can set β to a finite value in units of BDP. One can also set β to ∞ or let CCAC search over all its possible values to find one that satisfies the query.

We note that, when we first developed the model, it was just a set of constraints discussed below. It was devoid of any physical interpretation beyond the rough intuition that the path server should transmit roughly C packets per second, while allowing for some short term jitter. We simply sought a model that satisfied two mathematical properties described in section 4.2.1. The non-deterministic TBF interpretation described above was added after-the-fact.

Formal definition. Let $Q(t)$ and $T(t)$ denote the number of bytes in the packet and token queues, respectively. $A(t)$ denotes the cumulative number of bytes that have arrived at the server till time t. Similarly $S(t)$, $L(t)$, and $W(t)$ are the cumulative number of bytes serviced from the path-server, bytes lost, and tokens wasted, respectively (see Table 4.1). A number of constraints bound the behavior of the network, which we express as constraints on these functions:

- Tokens arrive at rate C bytes/s but at time t, $S(t)$ of them have been used and $W(t)$ of them have been wasted. Hence, $T(t) = Ct - W(t) - S(t)$.

[1]Note that if packets are arriving too slowly, the server is forced to waste tokens to avoid keeping a token for more than D seconds.

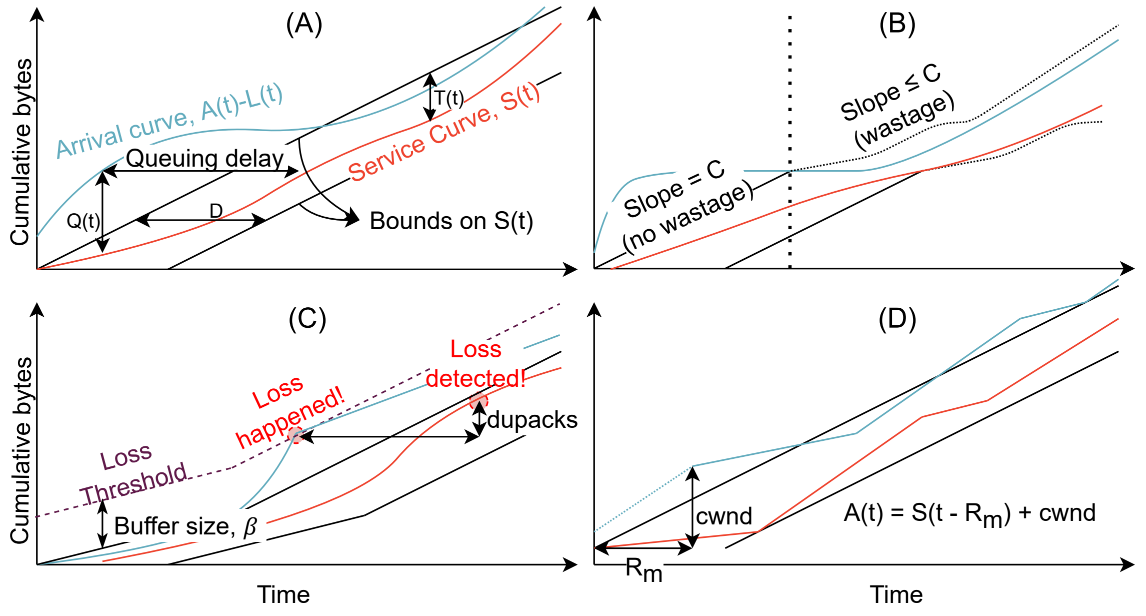


Figure 4-3: Graphical representation of the constraints.

- The queue length is the number of bytes that have arrived but have not been serviced or lost. Hence, $Q(t) = A(t) - L(t) - S(t)$.

- Wastage can only happen when there are more tokens than packets. That is, $T(t) \leq Q(t) \Rightarrow W'(t) = 0$, where $W'(t)$ is the time derivative indicating how much waste occurred.

- Loss is disallowed unless the *bottleneck* queue, $Q(t) - T(t)$, exceeds β bytes. Hence, $Q(t) - T(t) < \beta \Rightarrow L'(t) = 0$. We also have $Q(t) - T(t) \leq \beta$.

- Naturally, $Q(t) \geq 0$ and $T(t) \geq 0$ and the cumulative functions $A(t), S(t), L(t)$, and $W(t)$ are all non-decreasing. Since wastage can happen only when tokens enter the queue, $Ct - W(t)$ should also be non-decreasing.

- Finally, the server would not have admitted tokens if they were going to stay in the queue for more than D seconds. Hence, all tokens that arrived D seconds ago and were not wasted must have been used by now. That is, $S(t) \geq C \cdot (t - D) - W(t - D)$. Together with $T(t) > 0$, we have that $C \cdot (t - D) - W(t - D) \leq S(t) \leq Ct - W(t)$. These are the bounds on the service curve, $S(t)$.

Visualization. To make it easier to present examples of network behavior in our model, we use the graphical representation shown in Figure 4-3. The representation is akin to a TCP's time-sequence graph, where the x-axis represents time and the y-axis represents the cumulative number of bytes arriving at the path-server (i.e., arrival curve) or served from it (i.e., service curve). Figure 4-3(A) illustrates a simple example with the arrival curve in blue, the service curve in red, and the bounds on the service curve in black. Note that $A(t) - L(t)$ is the cumulative number of bytes admitted to the queue. The horizontal gap between the black curves is D. The horizontal gap between the arrival and service curve represents the time spent by a packet in the path-server's queue (which does not include R_m, the propagation delay). The vertical gap represents $Q(t)$. The vertical gap between the service curve and upper black curve equals $T(t)$ and represents the largest burst possible at this time.

Wastage is disallowed when $T(t) \leq Q(t)$. Substituting T, Q, and rearranging, we get $Ct - W(t) \leq A(t) - L(t)$. Hence, wastage is possible only when the arrival curve is less than the upper black curve (see Figure 4-3(B)). When wastage happens, the slope of the black curve is smaller than C, indicated by dotted lines in the figure. Figure 4-3(C) shows we can define a loss threshold curve, which is the upper black curve plus β. The arrival curve must remain below this line, and loss is allowed only when the arrival curve touches this line. The sender detects losses when it receives *dupacks* number of acknowledgments of packets *after* the loss event. Figure 4-3(D) shows how a CCA that simply maintains a constant `cwnd` controls $A(t)$ in response to $S(t)$. Here, $A(t) = S(t - R_m) + \texttt{cwnd}$ because the ACKs leaving the server will reach the sender R_m seconds later, which maintains `cwnd` packets in flight.

4.2.1 The set of paths the model captures

The behavior of a real network path can be decomposed into a series of "boxes". Individual boxes represent various phenomena such as bottleneck links and link-layer aggregation. In some cases, it helps to decompose one physical device as multiple boxes. For instance, the sender's network stack can be broken into a component that packetizes and another that paces packets with timing errors.

Individual "boxes". Now we discuss which real network boxes the path-server can emulate. Since it is a generalized TBF, it can naturally implement a regular TBF as shown in Figure 4-4. If the TBF has a link rate of C and a burst size of K, then we can emulate it with a path-server with the same link rate and $D = K/C$. When the sender stops sending, tokens accumulate, allowing the path-server to burst when the sender bursts (i.e., the arrival curve has a large single-step increase). The path-server can burst a maximum of $C \cdot D$ bytes, buffering the rest. Then, the TBF sends data from its buffer at a rate of C.

The path-server can also emulate all behaviors in our simpler model in section 4.1 (see Theorem 5 in Appendix A.4). Thus, it can emulate a wide range of phenomena such as packetization, link-layer aggregation, scheduling errors, MAC-layer jitter, and arbitrary delays ($\leq D$). The second example in Figure 4-4 shows the behavior of a link-layer aggregator. Regardless of the sending rate of the arrival curve, the service curve aggregates packets (which can be caused by link-layer aggregation) and sends them in bursts. Note that the simple model cannot emulate token bucket filters with finite buffers, while the path-server can.

Composition. If network boxes τ_1 and τ_2 can be emulated by path-servers with infinite buffers ($\beta = \infty$) and parameters (C_1, D_1) and (C_2, D_2), then the composition of τ_1 and τ_2 can be emulated by a path-server with parameters $(\min(C_1, C_2), D_1 + D_2)$ (see Theorem 9 in Appendix A.4), where C, D, β are the link rate, jitter parameter, and buffer size, respectively. When buffers are finite, our result is weaker: if $C_1 \leq C_2$ and $\beta_2 \geq C_1 D_1$, then their composition can be emulated by a path-server with parameters $(\min(C_1, C_2), D_1 + D_2, \beta_1)$ (see Theorem 7 in Appendix A.4). We do not have results for when $C_1 > C_2$ and buffers are finite.

Design decisions. The weakening of results with the constraint $\beta_2 \geq C_1 D_1$ is by design. If the path-server is able to emulate a bursty box followed by one with a small buffer, it can emulate a network that drops packets *no matter what* the CCA does. This is because, even if the CCA sent evenly spaced packets, the bursty box can generate bursts of up to CD bytes larger than the buffer size of the small-buffered

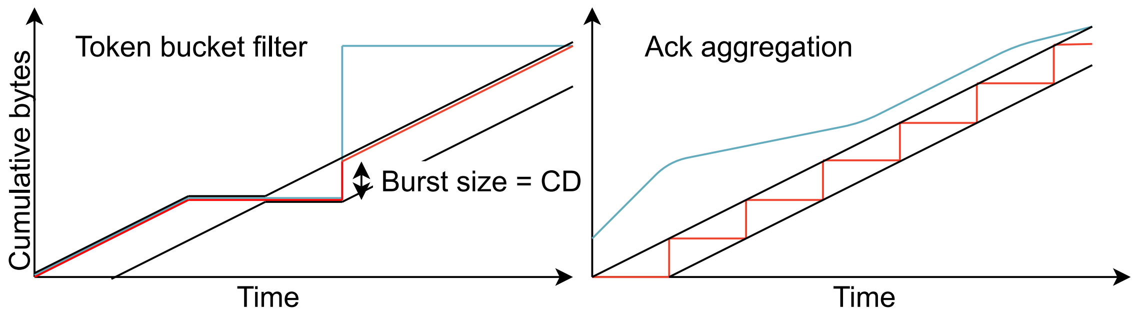


Figure 4-4: Examples of how the path-server emulates a token bucket filter and link-layer aggregation.

box, leading to packet drops that a CCA cannot avoid. While such paths are possible on the Internet, it is not useful to include them in the model because no CCA can help here. Thus, Theorem 7 considers only the case when the buffer size of τ_2 is big enough to absorb bursts created by τ_1 ($\beta_2 \geq C_1 D_1$). Indeed, we prove that when $C_1 \leq C_2$ and $\beta_2 \geq C_1 D_1$, the second box can never lose packets (see Theorem 6 in Appendix A.4). Operationally, we achieved this by defining the condition for loss on $Q(t) - T(t)$ and not on $Q(t)$, even though the latter might seem more natural (and is what we used in earlier iterations of our model).

Another design decision was to limit the *time* a token can spend in the queue to D seconds. It may seem more natural to limit the number of tokens in the queue to $K = CD \approx BDP$ bytes instead (this is what we used in earlier iterations of the model). Here, there is nothing forcing the path-server to use tokens. If the sender sends fewer than K bytes (say, because its initial `cwnd` $< K$), the server can hold on to the packets indefinitely, causing the sender to timeout. Requiring the initial window to be equal to the BDP is unreasonable, since determining the BDP is precisely the CCA's job! Exclude this behavior by using D instead of K is a clean resolution.

4.2.2 Expressing CCAs

A CCA controls the sender's transmission rate based on observed network behavior. In CCAC, a CCA determines the cumulative arrivals, $A(t)$, by determining the congestion window, `cwnd`(t), and pacing rate, $r(t)$. At time t, the CCA can observe the service curve up to time $t - R_m$, since feedback is delayed by R_m. The CCA can also observe $\tau_o(t)$ and $L^d(t)$, which are functions built into CCAC. $\tau_o(t)$ indicates whether a timeout happened at time t. $L^d(t)$ captures the cumulative number of losses detected. Losses are detected through duplicate ACKs and timeouts. As a convenience, the logic to calculate queuing delay based on A and S is also built into CCAC. The user needs to write constraints that determine $A(t)$ as a function of the observables. Figure 3-1 shows how to implement AIMD in CCAC.

Timeouts. If we implement timeouts according to RFC 6289 [124], the path-server can cause timeouts by simply emulating a smooth network in the beginning to keep `rttvar` (variation in RTT) low. Then, it can suddenly increase delay by D seconds to cause a timeout. While this scenario is possible in real networks, CCAC would produce excessively antagonistic worst-case behavior in this case. Instead, we trigger a timeout only when all in-flight packets have been lost. The sender would certainly timeout when this happens, and this mechanism avoids antagonistic timeouts.

4.3 Discussion of modeling choices

Variable link rates. We use two approaches to capture variable link rates. The jitter allowance, D, captures short-term variations. Long-term variations, like changes in the rate of a wireless channel, require C to be variable. However, C is constant in our model. To model a variable C, the user uses CCAC to prove lemmas about a CCA's behavior over a fixed link rate for some duration T'. Then, one can use mathematical induction on these lemmas to prove that as the fixed rate changes, the CCA will move toward a correct set of of `cwnd` and rate values for that link rate. One can pick T' to be the smallest (CCA-dependent) value such that the lemmas hold. Sections 6.2 and 6.3 show examples of this approach for AIMD and Copa.

Had we allowed the path-server to vary C with time, the path-server would have been able to emulate *any* network (i.e., it can pick any $S(t) \leq A(t)$). No CCA can function on a network where the capacity can vary arbitrarily, rendering CCAC useless since no interesting theorems about the CCA can then be proved with it (since they will not be true).

Choice of D. The path-server can capture jitter up to D seconds. There are two ways of setting D. First, if we know the path, we can calculate what D would be sufficient to model the individual components. Then, our composition theorems state that the net D is the sum of the Ds on the path. Alternately, we believe setting D to be one RTT is appropriate for congestion control because that is the timescale at which end-to-end CCAs can react to changes in the network. CCAs must hedge against fluctuations in the rate at smaller timescales. For longer timescales, they can simply adapt their rate to follow the network. Note that for any two path-servers with the same link rate and buffer size, with jitter parameters $D_1 > D_2$, the first path-server can emulate a superset of the paths emulated by the second path-server.

Non-congestive loss. $L(t)$ captures loss due to congestion. It is possible to model non-congestive loss as well by defining another function $L_{\text{nc}}(t)$ that is constrained as $0 \leq L_{\text{nc}}(t) \leq \eta \cdot (A(t) - L(t))$, where η is the maximum non-congestive loss rate.

Chapter 5

Formal analysis using an SMT solver

In this section, we show how CCAC uses SMT solvers. An SMT formula is a first-order logic formula over predicates. In CCAC, we only use predicates that are Boolean variables or linear arithmetic inequalities, as they are more efficient for automated analysis. Each linear predicate takes the form $\sum_i b_i v_i \geq c$ where b_i and c are real or integer constants and v_i are the real variables on the formula.[1]

An SMT solver attempts to find a satisfying assignment to the variables of the formula. If no such assignment exists, it outputs "unsat" (for unsatisfiable). CCAC uses the SMT solver, Z3 [35], to search through the space of all possible network traces generated from the interactions between the path model and the CCA. In this section, we describe the key ideas needed to express the path model in SMT constraints.

Representation. Our model for the network and CCAs include several functions over continuous time (i.e., functions of the form $f(t)$ where t is a real number, like the service and loss curves). To encode those functions in SMT constraints, we could use a single variable U_f to represent a function f. U_f would be an *uninterpreted function* with a single real input and a single real output. However, using a mixture of uninterpreted functions and linear arithmetic constraints proved to be intractable for our purposes. Thus, we represent those functions as a sequence of real variables. For example, we express the service curve, S, as $S_0, \ldots, S_T$, denoting S's values at

[1]Linear equations with real variables are easier for an SMT solver to handle than integer variables because they use linear programming as a subroutine. We use real variables everywhere, except for representing the state in BBR's state machine as an integer.

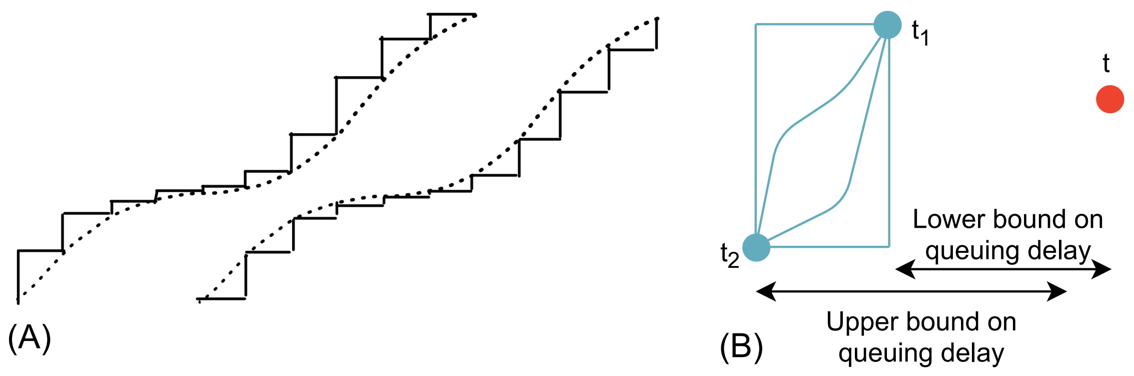


Figure 5-1: (A) While the bounds on $S(t)$ in the continuous model look like the dotted lines, we over-approximate that region using the solid lines as bounds. (B) In the discrete model, the queuing delay at time t can be any value between the upper and lower bounds.

times $t \in \{0, \cdots, T\}$.[2] Constraints can also be discretized. For instance, to express $\forall t, Q(t) \geq 0$, we add $Q_0 \geq 0 \wedge \cdots \wedge Q_T \geq 0$ to the formula.[3]

For computational efficiency, T must be small, leading to a coarse discretization with only 1 to 3 time steps per R_m.[4] Thus, the discretized constraints and functions can become a poor approximation to the continuous ones. To nevertheless get meaningful results, we adopt the following strategy.

Superset property. When formulating the constraints over discrete time, we ensure that they allow a superset of behaviors possible with the original constraints, so that any network trace that conforms to the continuous model is reproducible in the discrete SMT formulation. Hence, any bounds proved in the discrete model will be true in the continuous model as well.

Writing constraints that respect the superset property is often simple. In many cases, since the discrete model only constrains functions at discrete time steps (i.e., $t = 0, 1, .., T$), it admits more behaviors than the continuous model. For example, $S(t)$ is constrained with $Ct - W(t)$ as upper bound and $C(t - D) - W(t - D)$ as lower bound, shown as dotted lines in Figure 5-1(A). When discretized, the bounds on S become step functions that contain their continuous counterpart.

[2]We index to T rather than $T - 1$ since that includes T time steps.

[3]For the rest of the paper, we use $X(t)$ to represent functions over continuous time and X_t for the SMT discretization.

[4]Congestion control has a natural time scale of 1 RTT since that is the feedback delay.

Sometimes ensuring the superset property is more complicated. One example is computing the queuing delay used by the CCA. Recall that delay(t) can be defined as the horizontal distance between S and $A - L$ at $S(t)$. However, in the SMT formulation, S, A, and L are only defined at discrete time steps. Thus, as shown in Figure 5-1(B), a horizontal line drawn from $S(t)$ (red dot) can cross $A - L$ anywhere between t_1 and t_2 (the blue dots). As such, the SMT formulation allows delay_t, the discrete counterpart of delay at time t, to take any value between $t - t_1$ and $t - t_2$. To express this in SMT constraints for each $\Delta t \in \{0, \cdots, t\}$, we add the constraints $S_t > A_{t-\Delta t} - L_{t-\Delta t} \rightarrow \mathit{delay}_t \leq \Delta t$ and $S_t \leq A_{t-\Delta t} - L_{t-\Delta t} \rightarrow \mathit{delay}_t \geq \Delta t$.[5] Appendix A.3 discusses how we constrain L^d_t to maintain the superset property.

Due to the superset property, a trace that satisfies the discrete SMT formulation may not necessarily exist in the continuous model. As such, a bound proved using CCAC may be looser than necessary, or a corner case caught by CCAC may not be reproducible in the original model. The saving grace is that a human can always look at the trace generated by CCAC to ensure that it makes sense, as we have done in our case studies. In our experience, traces generated by CCAC always had a correspondence with real networks, even if it occasionally exploited the relaxation due to discretization.

Initial state. Unless specified by the user, we leave the initial state unconstrained. In particular, when CCAC explores the space of all possible network traces, it has full freedom to pick any initial values for variables like the queue size, wastage, and the number of lost packets. We will show in our case studies how to use this feature to prove properties over an infinite time horizon.

5.1 Expressing CCAs

A CCA controls the sender's transmission rate based on observed network behavior. In CCAC, a CCA determines the cumulative arrivals, $A(t)$, by determining the congestion window, `cwnd`(t), and pacing rate, $r(t)$. At time t, the CCA can observe the

[5]We omit handling the corner case when $S_t = A_{t-\Delta t} - L_{t-\Delta t}$ for clarity.

service curve up to time $t - R_m$, since feedback is delayed by R_m. The CCA can also observe $\tau_o(t)$ and $L^d(t)$, which are functions built into CCAC. $\tau_o(t)$ indicates whether a timeout happened at time t. $L^d(t)$ captures the cumulative number of losses detected. Losses are detected through duplicate ACKs and timeouts. As a convenience, the logic to calculate queuing delay based on A and S is also built into CCAC. The user needs to write constraints that determine $A(t)$ as a function of the observables. Figure 3-1 shows how to implement AIMD in CCAC.

Timeouts. If we implement timeouts according to RFC 6289 [124], the path-server can cause timeouts by simply emulating a smooth network in the beginning to keep `rttvar` (variation in RTT) low. Then, it can suddenly increase delay by D seconds to cause a timeout. While this scenario is possible in real networks, CCAC would produce excessively antagonistic worst-case behavior in this case. Instead, we trigger a timeout only when all in-flight packets have been lost. The sender would certainly timeout when this happens, and this mechanism avoids antagonistic timeouts.

SMT interface. Recall that each CCA can introduce its own set of variables and constraints to set $\texttt{cwnd}(t)$ and/or $r(t)$. In our SMT formulation, we have variables $\texttt{cwnd}_0, \ldots, \texttt{cwnd}_T$ and $r_0, \ldots, r_T$ as discretized versions of $cwnd(t)$ and $r(t)$, respectively. The user must discretize other CCA-specific variables and constraints and express the CCA as first-order-logic formulae. Specifically, for a CCA that uses a congestion window, we include constraints of the form $\texttt{cwnd}_t = f(\bar{\texttt{cwnd}}, \bar{Q}, \bar{E})$, where $\bar{\texttt{cwnd}}$ is the vector of all previous window values, Q is the vector of all state variables maintained by the algorithm, and E is the vector of all the information derived from the network such as loss and delay. While Z3 supports nonlinear constraints, we manage to avoid them, improving efficiency. For example, a common technique is to express a nonlinear function using linear constraints via a lookup table. For instance, we used it in Copa to multiply queuing delay and `cwnd`.

Asking Queries about CCAs.

Queries (hypotheses) about CCAs in CCAC must be expressed as first-order-logic formulae. For instance, to ask whether the network utilization can drop below a threshold u, we can add the formula $S_{T-1} - S_0 < uT$ to our SMT formulation and ask CCAC whether or not it is satisfiable. If it is, the solver will output an assignment to all the variables (i.e., S_t, A_t, L_t, and W_t) along with the CCA's variable that will cause the utilization to drop below u. If there is no such assignment, CCAC will have proved that the CCA will always achieve utilization of at least u over a period of T time steps.

By default, CCAC is designed to be free to choose many parameters. For instance, it can pick a network with a BDP that is small relative to the MTU (α). This may not be interesting to the user, so they can add an additional constraint such as $\alpha \leq \frac{1}{5}CR_m$. Hence, CCAC allows the user to explore many counterexamples depending on their interest. As another example, we do not need CCAC to tell us that AIMD reduces *cwnd* in response to non-congestive loss, as this is well-understood. Hence, we simply disabled non-congestive loss to focus on loss caused by buffer overflow. For the queries we asked in this paper, this leads to more interesting and unexpected counterexamples.

5.2 Parameters and linearity

The model has parameters like the link rate, C, and the propagation delay, R_m. In addition, each CCA may introduce its own parameters. We would like to prove properties for *any* choice of these parameters. Ideally, we would leave all of them as variables that are picked by the solver. However, that is not always possible and, for some parameters, we must resort to other techniques. To better understand how to pick parameter values, we will start by explaining parameter units.

There are two units in our framework: time and bytes. Without loss of generality, we can pick them such that $C = 1$ and $R_m = 1$. Hence, our formulation quantifies over all C and R_m "for free". Having C be a constant helps because many constraints involve a product of C with a variable. If C were a variable, picked by the solver, multiplication with C would make the constraint non-linear. The same benefit holds for R_m. In addition, some model constraints relate values of functions across time steps that are R_m or D apart. For instance, the sender can use S_{t-R_m} as the number of ACKs received so far to set `cwnd` at time t. Thus, both R_m and D need to be integers. R_m is a small integer that controls the number of time steps per RTT. As such, it controls the granularity of discretization. The user needs to pick D and in so doing they can sweep over different values of D/R_m. D/R_m is the value of D measured in units of propagation delay.

Parameters that do not appear in a product with another variable can be left as variables whose value will be picked by the SMT solver. Examples of such parameters are the buffer size, β, and the additive increase constant, α, in algorithms like AIMD and Copa. Note that when the solver picks α, it is implicitly picking the number of packets in a BDP, CR_m/α.

Chapter 6

CCAC case studies

We demonstrate the power of CCAC through three case studies in the next three sections: BBR [31], AIMD [34], and Copa [13]. CCAC's model of these algorithms is simplified and does not correspond exactly to code. That said, we make three observations. First, CCAC resembles some implementations of CCAs that also react only a small number of times per RTT due to CPU limitations. These have been empirically demonstrated to have similar behavior [108]. This is because the fundamental timescale of operation for a CCA is one RTT. Second, CCAC captures complexities such as duplicate ACKs and timeouts. Third, congestion control is far from being a solved problem and CCAs are not yet fully understood at an abstract level. Thus, CCAC is still able to uncover surprising behaviors. Further, for simplicity, our analyses of BBR and Copa assume that the sender has a correct estimate of R_m.

In each case, we formulate queries that probe the studied algorithm for "bad behavior". CCAC produces counterexamples that allow us to discover unexpected behavior in all three algorithms that significantly impair their performance. We also use CCAC to prove bounds on the worst-case performance of AIMD and Copa.

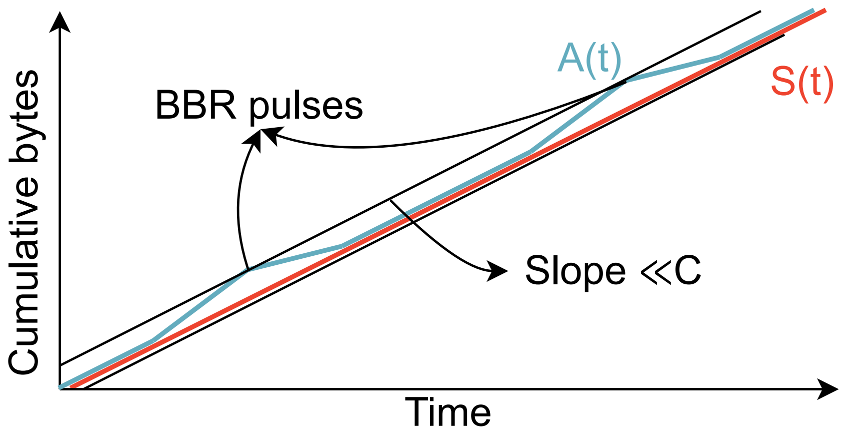


Figure 6-1: Network behavior generated by CCAC that prevents BBR from discovering bandwidth.

6.1 Case study 1: BBR

BBR [31] is a complicated rate-based algorithm that relies on a number of "if" conditions. However, the core idea of BBR is simple; the sender calculates the BDP as the current rate multiplied by the minimum RTT, i.e., as the (total number of bytes ACKed in the last RTT) * (min RTT) / (RTT). The sender sets its BDP estimate to the maximum value calculated over the last 10 RTTs. BBR sets its `cwnd` to twice this estimate and its pacing rate to (BDP estimate) / RTT. It has an 8-RTT cycle. In the first RTT of the cycle, it attempts to probe available bandwidth in the network through "pulsing". Thus, it increases the pacing to a value larger than the value calculated using the above formula. In the second RTT, it significantly decreases the pacing rate to clear the queue generated in the previous RTT. Then, it maintains the calculated rate for the remaining 6 RTTs.

We implement this core idea in CCAC, encoding it in SMT constraints. Then, we ask queries of the form "Can BBR achieve less than x% utilization?", for different values of x. To do so, we add the constraint $S_T - S_0 \leq xCT$, which instructs the solver to find instances where the total number of bytes served, $(S_T - S_0)$, is a fraction, x, of the maximum, CT. We also add periodic boundary conditions to ensure that the trace output can be repeated. In the absence of these, the solver can produce a trace where BBR gets low utilization because its *initial* `cwnd` is low and does not ramp up in 20 RTTs; we are looking for low utilization in *steady-state*. Concretely, we add

$Q_0 = Q_T \wedge L_0 - L_0^d = L_T - L_T^d \wedge Q_0 - T_0 = Q_T - T_T$ to the constraints. While the execution time depends on the query, queries up to 20 time steps finish within a few minutes on a standard laptop.

CCAC generates examples of poor utilization, even for arbitrarily small values of x. Our next step is to analyze these low-utilization examples. Figure 6-1 shows a schematic of the examples produced. When BBR increases the pacing rate to probe for network bandwidth, the pulse is small (i.e., the increase in pacing rate is small), and the network does not serve it at a higher bandwidth. Hence, BBR's probe fails (i.e., the BDP estimate remains small).

BBR's design incorporates a feature that we think helps it avoid this behavior in many networks; BBR's BDP estimate is the *maximum* calculated over the last 10 RTTs. Thus, it usually over-estimates the quantity because most networks have some delay jitter. This overestimation implies BBR will cause queue build-ups on jittery paths. This finding is consistent with empirical observations that BBR often maintains 1 RTT of queuing in practice [131, 141]. However, *if the network is clean with a smooth service*, the problem CCAC identifies can manifest itself.

One approach to solve this problem is to *intentionally overestimate* the pacing rate. For instance, the sender can pace BBR flows at twice the prescribed rate. In fact, recently, Facebook made this change to their BBR implementation in mvfst [70], the version of QUIC [71] they use in production. We implemented that version of the algorithm in CCAC. We found that CCAC no longer finds any cases where the algorithm gets <100% utilization in the steady state when the buffer is infinite.[1] We believe that intentionally over-estimating the pacing rate can increase delay on average, while the worst-case delay remains the same as before.

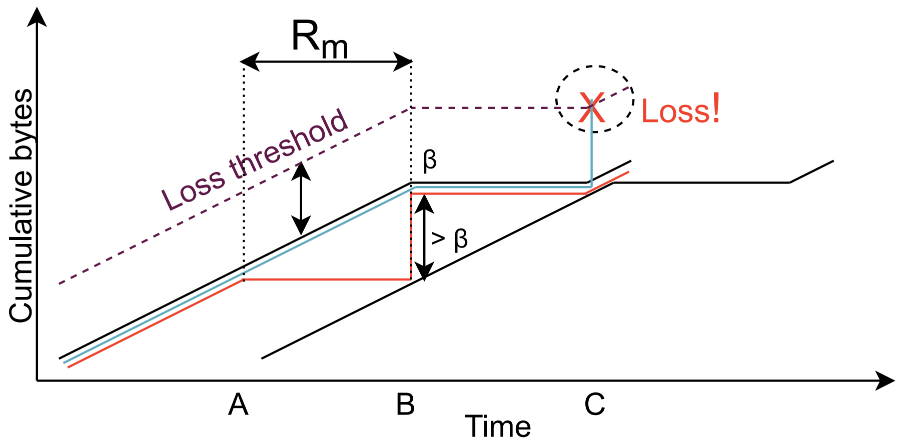


Figure 6-2: An example where ACK aggregation causes loss even when the congestion window is small.

6.2 Case study 2: AIMD

In this section, we first describe the surprising AIMD behavior we discovered using CCAC. Then, we show how we can prove bounds on AIMD's behavior that are valid over an infinite time horizon. Our implementation of AIMD is ACK-clocked and unpaced. The algorithm sends packets when L^d, S, or `cwnd` increase. It seeks to maintain `cwnd` bytes in flight. A packet is "in flight" when it has neither been ACKed nor marked as lost: $\mathit{inflight} = A(t) - L^d(t) - S(t - R_m)$. When $\mathit{inflight}$ drops below `cwnd`, the sender sends a packet. Our AIMD implementation handles duplicate ACKs and timeouts. It increases its `cwnd` only when it gets enough ACKs and does not react more than once to the same loss event. Due to the discretization of time in CCAC, however, AIMD reacts only once per time step. The query is of the form shown in Figure 3-1.

6.2.1 The surprise

We study how jitter can cause AIMD to incorrectly reduce its `cwnd` due to buffer overflow even when the buffer is large. Thus, our query (Figure 3-1) aims to find scenarios where AIMD can observe packet loss when `cwnd` is small. Specifically, we add the following constraint: $\bigvee_t (L_t > L_{t-1} \wedge \mathit{cwnd}_t \leq \mathit{thresh})$ (in Figure 3-

[1]Note that this is not a formal proof that the modified version will always have high utilization. We show how these proofs can be constructed for AIMD and Copa in the next two sections, leaving the formal proof of BBR's properties to future work.

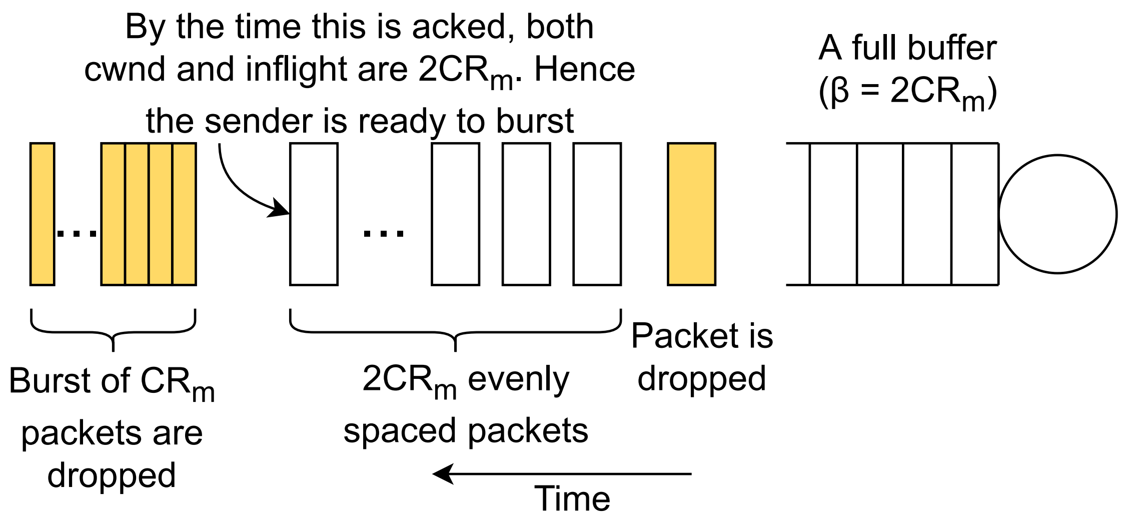

By the time this is acked, both cwnd and inflight are 2CR$_m$. Hence the sender is ready to burst
A full buffer (β = 2CR$_m$)
Burst of CR$_m$ packets are dropped
2CR$_m$ evenly spaced packets
Packet is dropped
Time

Figure 6-3: At the end of this sequence of packets, 1) the sender has reduced its `cwnd` from $4CR_m$ to $2CR_m$ due to loss, 2) the server has dropped CR_m packets 3) *inflight* = `cwnd` = $2CR_m$ making the sender ready for another burst. Packets go from left to right.

1, *loss_happened* is simply $L_t > L_{t-1}$). CCAC uncovered two ways in which this situation can occur. We discuss these in turn.

Loss due to ACK bursts. We start with a well-understood behavior that CCAC uncovered. Consider an unpaced and ACK-clocked CCA. If ACKs arrive in a burst, the CCA will send packets in a burst. A burst of ACKs can cause the algorithm to send a burst of packets, overwhelming the buffer and causing packet drops.

CCAC generated an example of this behavior, shown in Figure 6-2. Suppose $D = R_m$ and `cwnd` $= \beta = CR_m$. For ease of understanding, assume that `cwnd` is roughly constant, say because it is large compared to the additive constant. Initially, the path-server maintains zero queue and hence the sender sends at a rate `cwnd`$/R_m = C$. At time A in the figure, the path-server decides to stop transmitting to emulate an ACK aggregator that withholds ACKs, only to send them in a burst later at time B. The ACK aggregator can pause packets for at most D seconds. When it sends a burst at time B, the ACKs reach the sender R_m time steps later, at time C. These ACKs causes the sender to send a large burst of size $CD = CR_m = \beta$ bytes, overwhelming the buffer and causing packet loss. Note that this can happen even when `cwnd` $< C \cdot R + \beta$, which is the threshold at which fluid models predict loss will happen. In general, this phenomenon can cause packet drops when `cwnd` $> \beta$ and $\beta < CD$.

Loss due to loss-bursts. We now discuss a finding that took us by surprise, again with an unpaced and ACK-clocked AIMD CCA. CCAC found that a burst can also happen if $L^d(t)$ increases suddenly (recall that $\mathit{inflight} = A(t) - L^d(t) - S(t - R_m)$). But this discovery is surprising because `cwnd` is halved when losses are detected! Thus, we would expect `cwnd` $-$ *inflight* to not increase.

CCAC found that this safeguard can fail, by finding a situation where losses occur in two steps. The first loss halves `cwnd`. Then, packets are ACKed until *inflight* $=$ `cwnd`. Now, the sender detects a burst of losses and does not halve its `cwnd` again because it is part of the same loss event [46] (`cwnd` decreases only if the packet that was lost was sent *after* the last `cwnd` decrease).

A concrete example of this behavior arises when $D = R_m$ and $\beta = 2CR_m$ (we use CCAC to prove bounds for other values of β and D later). First, the path-server inflates the propagation delay to $R_m + D$. This allows `cwnd` to increase to $C(R_m + D) + \beta = 4CR_m$ without loss. When the `cwnd` exceeds this quantity, one packet gets dropped and the sender reduces its `cwnd` to $2CR_m$, while still having $4CR_m$ packets in flight. After the loss, the path-server services the next $2CR_m$ packets evenly. However, the sender does not transmit any packets in response because the number of packets in flight is still larger than `cwnd`. At the end of this process, *inflight* $=$ `cwnd`.

Now, the path-server does not service any packets for the next R_m time steps. Then, it drops CR_m packets and services the remaining CR_m packets in a burst. Thus, rather than receiving $2CR_m$ ACKs, the sender receives only CR_m ACKs, indicating that an additional CR_m packets were lost. However, this burst in loss does not trigger another `cwnd` decrease since the sender recently decreased `cwnd`. Now, $L^d(t)$ increases by CR_m and $S(t - R_m)$ also increases by CR_m because of the last burst. This empties the in-flight packets and causes the sender to burst $CR_m + CR_m = 2CR_m$ packets at once. The combined burst is twice as large as what the path-server can burst at once. This burst is enough to overwhelm the buffer again (recall, $\beta = 2CR_m$), causing another packet drop. *This* drop occurs for a packet that was sent when `cwnd` was already $2CR_m$. Hence, the sender will reduce its `cwnd` again to CR_m. We discuss this example in more detail in Appendix A.2.

Figure 6-3 depicts the above discussion. It shows the spacing (in time) between packets that arrive at the server just as its buffer is about to exceed capacity for the first time (when $\texttt{cwnd} = 4CR_m$). The packets are spaced this way because the path-server sent ACKs in that pattern R_m time steps earlier. Note that when the path-server services a packet, the effect is seen R_m time steps later in the sender's packet transmissions.

An important question to consider here is: If the minimum cwnd in this scenario is CR_m, doesn't the sender always achieve full utilization? No, because jitter in delay can inflate the RTT. Hence, when $\texttt{cwnd} = CR_m$, utilization can be as low as $\texttt{cwnd}/(R_m + D)$, which is just 50% of C in this example. This phenomenon can happen repeatedly, causing consistently low utilization.

Mitigation 1: limit transmissions per ACK. At first glance, RFC6582 [46] handles this case. It says "the implementation is encouraged to take measures to avoid a possible burst of data, in case the amount of data outstanding in the network is much less than the new congestion window allows. A simple mechanism is to limit the number of data packets that can be sent in response to a single acknowledgment." Note, however, that our burst due to loss is followed by a burst of actual ACKs. Thus, the problem occurs despite this mitigation.

Mitigation 2: Pacing. Premature losses can occur when pacing is implemented with slack. For instance, the Linux kernel used a pacing of $2\texttt{cwnd}/\text{smoothed\_rtt}$. The factor of 2 does not prevent premature losses, because bursts can still occur. Recently, Google produced a patch reducing it to 1.2 because they noticed a performance improvement [110], which perhaps happened for the reasons discussed above. We used CCAC to confirm that losses can indeed occur when $\texttt{cwnd} < CR_m + \beta$ when the sender is paced in this way.

The example above was when $\beta = 2CR_m$. Naturally, beyond a certain buffer size the network cannot orchestrate a burst large enough to overwhelm the buffer prematurely (when $\texttt{cwnd} = \beta < CR_m + \beta$). However, it is difficult for even experienced engineers to determine this threshold, especially when multiple phenomena interact.

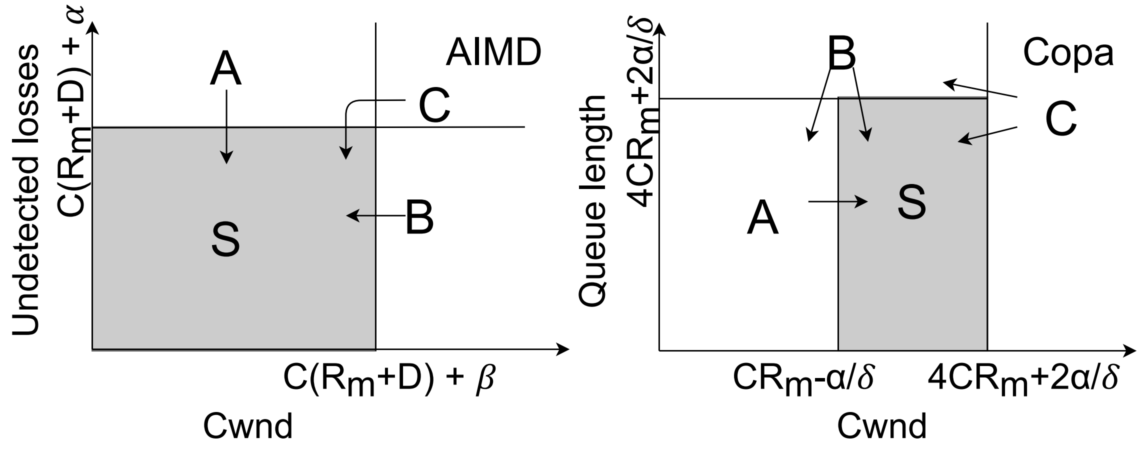


Figure 6-4: State diagram we use to prove AIMD and Copa's steady-state behavior.
In the next subsection, we show how CCAC can help us discover and prove this threshold (Theorem 2).

6.2.2 AIMD's steady-state analysis

CCAC only searches through traces that are a finite number of time steps long. Nevertheless, we can stitch together statements proved over finite time to prove theorems about arbitrarily large time horizons. We focus on "steady state behavior" and exclude transients that occur when network parameters such as the link rate or propagation delay change. To do so, we first leave the initial conditions unconstrained; if the network parameters were different before $t = 0$, they could leave the network in any state and we continue our analysis from there. Then, we assume C and R_m are constant and prove that the CCA moves toward the steady state.

A *steady state* for a given CCA is a set of network states such that as long as the network parameters remain unchanged (1) if the network enters it, it will never leave it and (2) it will always enter the steady state, regardless of initial conditions. In our case studies, a steady state is defined by bounds on `cwnd`, queue length, and number of undetected losses. The steady state for a CCA may not be unique and only needs to be as "tight" as needed by the theorems we wish to prove about them. Note that our approach to steady-state analysis allows us to reason about variable link rates as well, since we prove that the `cwnd` always moves in the "right" direction. Hence, as the link rate varies, the `cwnd` will always track it.

The user's intuition and experience are essential to arrive at the steady state. They can pose different queries to CCAC to validate their guesses. For AIMD, we guess that the steady state is defined by an upper bound on `cwnd` and the maximum number of undetected losses, $L_t - L^d_t$. We guess $max\_cwnd = C(R_m + D) + \beta + \alpha$ and $max\_undetected = C(R_m + D) + \alpha$. With these guesses, we prove the following theorem:

Theorem 1. *For AIMD, if $CR_m > 5\alpha$,[2] the steady state is defined by* `cwnd` $< C(R_m + D) + \beta$ *and* $L(t) - L^d(t) < C(R_m + D) + \alpha$. *Under the CCAC path model, AIMD will eventually enter this steady state from any initial state. Further, once entered, AIMD will never leave the steady state.*

To prove this theorem, we divide the state space of AIMD as shown in Figure 6-4. Then, we prove the following lemmas, which show that both `cwnd` and undetected losses always move in the right direction (shown with arrows in the figure). The proof uses these lemmas:

1. If $\texttt{cwnd}_0 > max\_cwnd \wedge L_0 - L^d_0 \leq max\_undetected \wedge CR_m > 4\alpha$ then $\texttt{cwnd}_T < \texttt{cwnd}_0 - \alpha$

2. If $L_0 - L^d_0 > max\_undetected \wedge CR_m > 5\alpha$ then at least one of the following holds

   (a) $L_T - L^d_T \leq L_0 - L^d_0 - C$ (i.e., undetected losses decrease by at least C) and $\texttt{cwnd}_T \leq max\_cwnd$

   (b) $\texttt{cwnd}_0 > max\_cwnd \wedge \texttt{cwnd}_T < cwnd_0 - \alpha$

3. Once AIMD has reached steady state, it will remain there. That is, if $L_0 - L^d_0 \leq max\_undetected \wedge \texttt{cwnd}_0 \leq max\_cwnd \wedge CR_m > 3\alpha$ then $\bigwedge_t L_t - L^d_t \leq max\_undetected \wedge \texttt{cwnd}_t \leq max\_cwnd$.

[2] We constrain the BDP to be more than 5α, because small BDPs elicit a different type of worst-case behavior which we don't study for the sake of brevity. The threshold was determined by repeatedly querying CCAC and it happened be a small integer.

Lemma (2) implies that if the number of undetected packets and `cwnd` are both larger than the threshold, then first `cwnd` will fall below the threshold. At this point, the number of undetected losses will fall until it is also below the threshold. Combined with (1) and (3), the lemmas prove the theorem.

To prove a statement using CCAC, we add its negation as a constraint and confirm that CCAC returns “unsatisfiable”. Each proof works for a specific value of β (specified in number of BDPs). We sweep over several values between 0.1 to 4 BDP and prove the theorem for each. Having established the steady state, we prove bounds on premature drops. Using insights from experimenting with CCAC, we formulate the following theorem:

Theorem 2. *If $\beta <= C(R_m + D)$, loss can happen if and only if* `cwnd` $\geq \beta - \alpha$. *For other values of β, the condition is* `cwnd` $\geq C(R_m - 1) + \beta - \alpha$.

The latter threshold in the theorem agrees with the fluid model except for the $-C$ term. This term comes from discretization, because the discretized path-server can burst $C \cdot 1$ bytes more than the continuous version (see Figure 5-1(A)). The finer our discretization, the smaller this difference. Recall from Section 5.2 that units of time are arbitrary, and the absolute value of R_m only controls the granularity of discretization. Higher values lead to larger SMT formulae, requiring CCAC to take longer to solve. The quantity of interest is actually R_m/D. Hence, we prove the result for $R = 2; D = 1, 2, 3$ and $R = 3; D = 1, 2, 3$ while sweeping over $\beta \in (0.1, 4CR_m]$ and conjecture that the theorem is true in general.

6.3 Case study 3: Copa

Copa [13] is a delay-based algorithm like Vegas [27] and Fast [143] that we designed and is described in section §11. It incorporates two new ideas. First, while Vegas computes queuing delay as (RTT - minimum RTT), Copa uses (Standing RTT - minimum RTT). Standing RTT is the minimum RTT over a short period of time, typically the last RTT. Copa increases its rate when the estimated queuing delay is low, and decreases its rate otherwise. Thanks to the use of Standing RTT, it decreases its rate only when there is *persistent* queue buildup. Second, Copa has a mode-switching algorithm that helps it detect if it is sharing the bottleneck with cross traffic that uses a buffer-filling CCA (e.g., Cubic). If so, it switches to a more aggressive mode, similar to AIMD or Cubic, to compete with such traffic.

6.3.1 Worst-case utilization

Our goal is to understand the value of using Standing RTT and whether it guarantees high utilization. We implement Copa in CCAC without mode-switching and ask it a series of queries of the form "Can Copa achieve less than $x\%$ utilization?", for different values of x. This is the same query that we used for BBR and includes the same periodicity constraint. We find that CCAC generates examples of poor utilization, even for small values of x. This section describes how we used CCAC to understand why Copa might perform poorly.

The intuition behind Copa's Standing RTT idea is that when Copa is sending at less than link rate, the queuing delay would be zero at least once every RTT. Thus, (Standing RTT - min RTT) would be zero, allowing Copa to increase its `cwnd`. Since it would not be prudent to expect a real measurement to be exactly zero, Copa also increases its rate if it believes the queue is *nearly* empty; that is, it has fewer than $1/\delta$ packets, where δ is a constant parameter of the algorithm (e.g. Facebook's implementation of Copa used $\delta = 1/25$ [50]).

One would expect Copa to always be able to maintain high utilization. However, the counterexample generated by CCAC tells a different story. Figure 6-5 shows that Copa maintained a persistent queue of up to CD packets, or $\approx CR_m \gg 1/\delta$ packets (recall that the queue length is the horizontal distance between $A(t)$ and $S(t)$). This caused the sender to decrease `cwnd`. So why was our[3] understanding incongruous with this counterexample?

When utilization is low, we expect the arrival curve to almost meet the service curve frequently, representing an emptying of the queue. However, in Figure 6-5,

[3]Two of the authors of this paper were the designers of Copa.

the arrival and service curves don't come close, meaning that a standing queue is maintained persistently in the network. This behavior causes Copa to overestimate delay despite the Standing RTT mechanism, degrading throughput. Copa decreases its rate unless the standing queuing delay it measures is less than $1/(\delta r)$, where r is its current rate. Intuitively, in the worst case, the network can maintain a standing queue of D, which means it can fool Copa into reducing its rate down to a negligible rate of $1/(\delta D)$.

We now try to identify a path where such behavior can occur. We start by trying to identify a single network box that can produce this behavior. To do so, we change the model to allow waste only when $Q(t) = 0$, while in the original model it is allowed when $T(t) \geq Q(t)$. The modified model retains its ability to emulate many network boxes but it no longer composes; it cannot emulate a cascade of these boxes (see §4.2.1). When we ran the same query with this non-composing model, we found that CCAC no longer generates examples of Copa achieving very low utilization. This is because, if the CCA is sending packets at a rate lower than the link rate, the non-composing model will have to waste tokens so that they don't expire. Waste is allowed only when $Q(t) = 0$. This forces the path-server to empty the queue, which Copa detects and increases its rate.

What is the difference between the composing and non-composing model? How can multiple boxes maintain a standing queue, even when a single box cannot? The answer is that a standing queue can arise when the different boxes empty their queues at different points in time. For example, consider a Wi-Fi device, W, that has to share the medium with other devices. When W gains medium access, it sends at a high instantaneous rate, while on average it has a lower rate of C. Suppose W is followed by another box with little jitter and comparable or lower average throughput as illustrated by the dotted line in Figure 6-5.[4] Here, the first box has arrival curve $A(t)$ and service curve $S_1(t)$ and the second one has arrival curve $A_2(t) = S_1(t)$[5] and service curve $S(t)$. As evident in the figure, the arrival curve and the service curve

[4]We added the dotted line by hand; the rest of the figure was generated by CCAC.

[5]To emulate delay d we can set $A_2(t) = S_1(t - d)$. This does not materially change the analysis since d can be included in R_m.

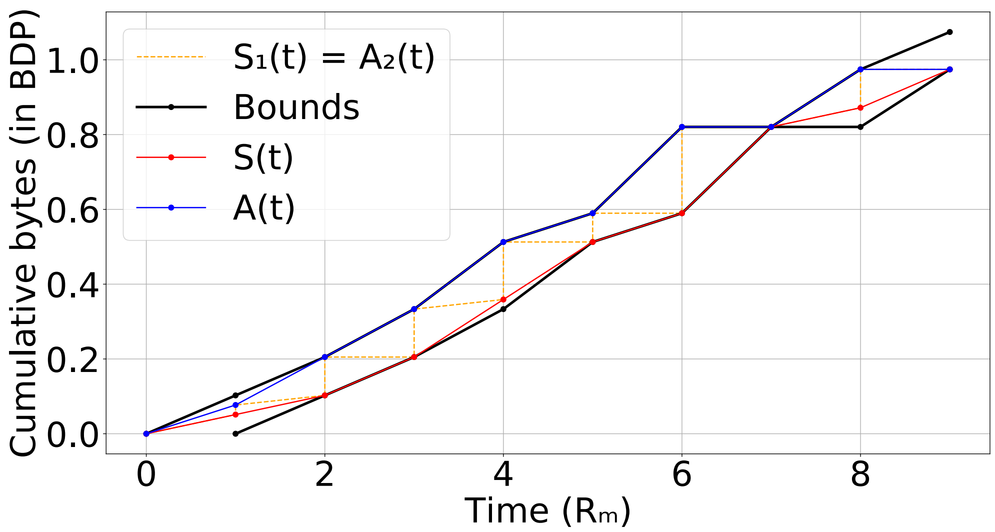


Figure 6-5: A trace generated by CCAC that causes Copa to severely under-utilize network capacity. Here, $C = 1BDP/R_m$ and Copa is severely under-utilizing the link. A network can have a standing delay when it has more than a single box, even if the individual boxes do not maintain a standing queue. $S_2(t)$ and $A_1(t)$ denote S and A of the first and second boxes respectively.

never meet after $t = 0$, producing the behavior seen in the counterexample generated by CCAC.

As designers of Copa, we had not considered the fact that multiple boxes may be able to create persistently high delays despite a lack of persistent queues in any individual box, and discovered it only when working with CCAC. We also found that Copa performs well not only when the condition for waste is $Q(t) = 0$, but also if it is $Q(t) \leq \frac{\alpha}{2\delta}$, which confirms that Copa's δ works as designed. The same holds for proofs in the next section.

6.3.2 Copa's steady-state analysis

We analyze Copa's steady state behavior using a similar approach to §6.2.2. Here, we analyze the special case where $D = R_m$ and the buffer is infinite. We guess that in steady state Copa maintains a queue length smaller than $4CR + 2/\delta$ and `cwnd` between $CR - 1/\delta$ and $4CR + 2/\delta$ (see Figure 6-4). We use CCAC to show that (a) Copa will eventually enter the set of states S and (b) once entered, it never leaves S.

For instance, to prove A→S in Figure 6-4, we ask CCAC to find an example where the initial `cwnd` and queue length are in A and at time T "bad things" happen (i.e., $A_T - S_T > 4CR_m + 2\alpha/\delta \vee (\texttt{cwnd}_T < \texttt{cwnd}_0 + \alpha/\delta \wedge \texttt{cwnd}_T < CR_m - 1/\delta))$. If CCAC finds no such instance, it proves the converse of the above. That is, if Copa is in A, it moves toward S. We similarly prove assertions from B and C, and finally prove that if Copa starts from state S, it remains there. *This proves that S is Copa's steady state.*[6]

Utilization. To determine the minimum utilization for Copa, we first constrain the initial conditions to be within the steady state and ask CCAC to give examples where $S_T - S_0 < xCT$. We conduct a binary search to find that $x = 0.5$ is the minimum value where CCAC can find an example. *This proves that Copa always achieves at least 50% utilization in steady state.*

But Copa always ensures its `cwnd` $\geq C \cdot R_m - 1/\delta$, so why is the bound only 50%? Why is not nearly 100%, as the fluid model would predict? The reason is the same as for AIMD. The server can inflate delay by $D = R_m$, which slows down ACKs and, hence, packet transmissions. This happens even though this restricted path model cannot maintain a large standing queue. Transient queues are enough to hurt utilization. We confirm that this bound is tight by asking CCAC to produce an example with 50% utilization with *periodic* boundary conditions (i.e., initial queue length, `cwnd` etc. are equal to the final values). The periodicity ensures that we can continue the pattern indefinitely, ensuring that the behavior is not transient.

Delay. Copa was designed to maintain a maximum queuing delay of $3\alpha/\delta$ bytes, so why does our analysis show a much higher value? Examples generated by CCAC show that this is a feature, not a bug, in Copa. When the network is jittery, Copa will (and should) increase its `cwnd` even if the *maximum* queue length is large, since it looks at the standing-RTT, not the latest RTT like Vegas does. Vegas would decrease its `cwnd` to nearly zero, adversely degrading its throughput.

Does this mean Copa always maintains a small *minimum* RTT, even though the

[6]Copa has some uninteresting corner-cases when α/δ is large relative to a CR_m (BDP), so we constrain it to be $< CR_m/5$ in all analysis.

maximum may be large? Unfortunately not. CCAC found examples with large minimum RTTs as well. This happens if the network is jittery to begin with, causing Copa to increase `cwnd` and then becomes smooth. In the period that Copa reduces its `cwnd`, the queue length will be large. Utilization can be $<$100% for the same reason; Initially the network is smooth, causing Copa to maintain a `cwnd` $\approx$ 1BDP, but becomes jittery later, causing low utilization.

Chapter 7

Statement and proof of the starvation theorem

The previous section's case studies focused on individual flows. However, as noted in section 3.2, our usage of CCAC for studying competition between two or more flows led us to instances of "starvation" - a severe form of unfairness - in all delay-bounding CCAs. In this section, we state and prove a key result: for every delay-convergent CCA (refer to §3.2), a scenario exists in our model (see §4) where one of the two competing flows experiences starvation.

7.1 Extension of our network model

First, we extend the models presented in chapter 4 to capture multiple flows. We only extend the simpler model since its behaviors are a subset of the full model. Thus any impossibility result we prove with it, must also hold for any extension of the simple model.

The new model shown in figure 7-1 has all flows using the same bottleneck queue, emulating that the flows share a bottleneck in a real network. However they each can experience different non-congestive delays because their non-deterministic delay components (denoted as "D") are separate. This emulates that the rest of their paths might be separate.

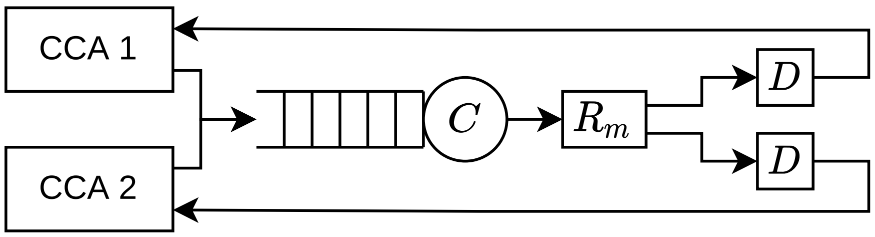


Figure 7-1: The simple network model for multiple flows

7.2 Definitions

Next, we define a few terms. We assume, without loss of generality, that all flows in the network start at time ≥ 0. We define the *throughput* of a flow at time t to be the number of bytes acknowledged between time 0 to t divided by t. We are interested in starvation, an extreme form of unfairness. We propose the following definition.

Definition 2. *Consider two flows f_1 and f_2 starting from arbitrary initial conditions (e.g., one of the flows could have run for a long time and the other just starting). The network is* **s-*fair*** *if there always exists a finite time t such that for all time beyond t, the ratio of the throughput achieved by the faster flow to the slower one is smaller than s.*

Definition 3. ***Starvation*** *is said to occur if the network is not s-fair for any finite s.*

This definition allows for massive unfairness without causing starvation. For example, if in steady state the two flows achieve a throughput ratio of a million to one, we would still say that there is no starvation. Our analysis asks if *any* finite ratio is achievable and proves that it is impossible for delay-convergent CCAs.

To prove that starvation occurs, we need to show that there *exist* starting states and network behavior after which the flows never improve their bandwidth allocation *no matter how long they run.* This is surprising because we expect our CCAs to eventually converge to a good allocation no matter the initial allocation. Flows start with unequal allocations, for instance, when one starts after the other. We also show empirically that for Copa, BBR and PCC it is quite easy to cause starvation even when the flows start at the same time.

One might think that this starvation claim requires CCAs to run at high efficiency; for example, perhaps a focus on 100% utilization leads to aggressive behavior causing starvation. We find that our result applies to CCAs that always utilize at least some constant fraction of the link capacity, which can be quite small (say 1%), and far from extreme efficiency. We only need to eliminate "silly" CCAs such as "set cwnd = 10 always", which are inefficient and impractical, but avoid starvation.

Definition 4. *A CCA is* **f*-efficient*** *if, when run on an ideal path with bottleneck link rate C and minimum RTT R_m, it eventually gets a throughput of at least fC; i.e., for any t, there exists a $t' > t$ such that the number of delivered bytes in the period 0 to t' is at least fCt'.*

We define f-efficiency in this way because we need to make statements about *all* delay-convergent CCAs, even absurd ones. For instance one can contrive a CCA for which the limit of the throughput as $t \to \infty$ does not exist. Practical CCAs are usually better behaved. We believe this definition adequately captures steady-state throughput since throughput must be at least fC infinitely many times.

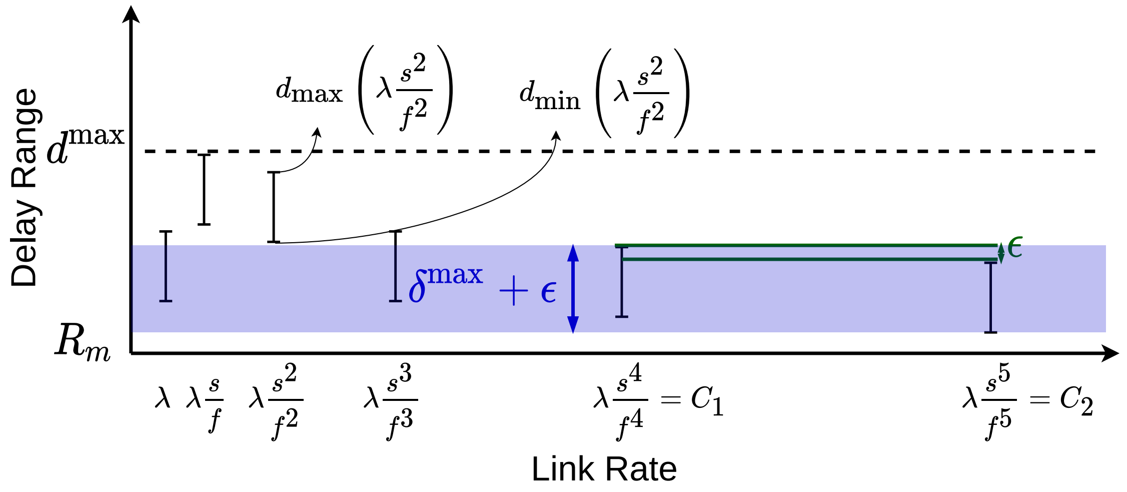
7.3 Starvation theorem

The following theorem states our result about the inevitability of starvation for delay-convergent CCAs. It assumes the flow is never application limited; variable load only makes the congestion control problem harder.

Theorem 3. *For any deterministic, f-efficient, delay-convergent CCA $\mathcal{A}$, any propagation delay R_m, any throughput ratio $s \geq 1$, and any $D > 2\delta^{\max}$, there exists a network scenario with two flows (specified via an initial state and two per-flow trajectories of non-congestive delays), such that one flow gets a throughput x_1 and the other flow gets a throughput $x_2 \geq s \cdot x_1$.*

The outline for the proof is as follows. Appendix A.5 fills in the mathematical details that we omit here.

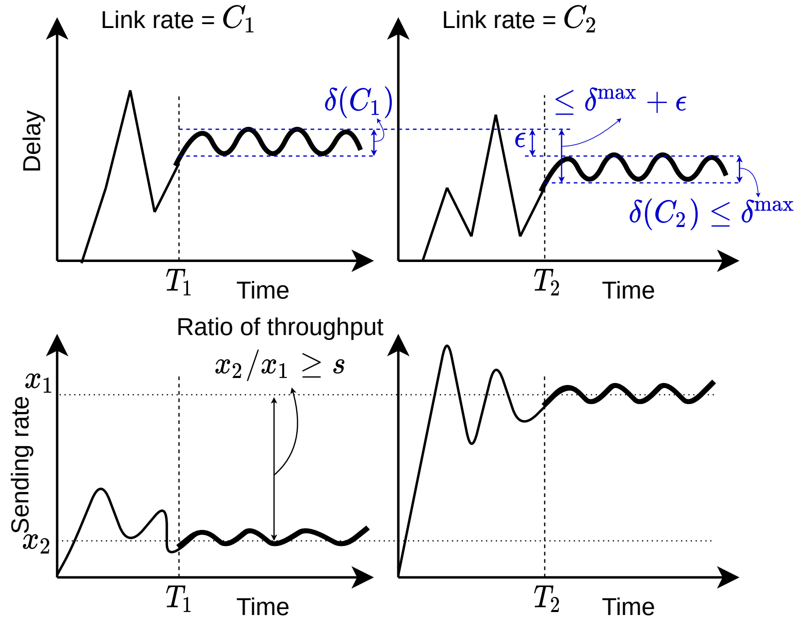
Step 1. Recall that the bound $d^{\max}$ has a corresponding λ; $\delta(C) < \delta^{\max}$ and $d^{\max}(C) < d^{\max}$ for all $C > \lambda$. We identify two bottleneck link rates $C_1, C_2 \geq \lambda$, such that C_2 is much larger than C_1 (at least a factor s/f larger) but the CCA $\mathcal{A}$, when run independently on links with these two rates, converges to delays in two ranges that are close to each other. Here, "close" means that the delay ranges achieved at rates C_1 and C_2 lie within an interval of size $\delta_{max} + \epsilon$. We will prove that for any $\epsilon > 0$, we can always find such a C_1 and C_2 for a delay-convergent CCA (we will pick ϵ later).



Delay Range
$d^{\max}$
R_m
$d_{\max}\left(\lambda\frac{s^2}{f^2}\right)$
$d_{\min}\left(\lambda\frac{s^2}{f^2}\right)$
$\delta^{\max} + \epsilon$
ϵ
λ
$\lambda\frac{s}{f}$
$\lambda\frac{s^2}{f^2}$
$\lambda\frac{s^3}{f^3}$
$\lambda\frac{s^4}{f^4} = C_1$
$\lambda\frac{s^5}{f^5} = C_2$
Link Rate

Our claim follows from a pigeonhole principle argument illustrated above. Recall that the delays for any link rate $> \lambda$ must fall in the interval $[R_m, d^{max}]$. Now there are only a finite number of non-overlapping intervals of size ϵ that can fit in $[R_m, d^{max}]$ (at most $\lceil (d^{max} - R_m)/\epsilon \rceil$ of them). But consider, for example, the infinite sequence of link rates $(\lambda_0, \lambda_1, \ldots)$, defined as $\lambda_i = \lambda \cdot (s/f)^i$. We have: $\lambda_i \geq \lambda$ for all i and $\lambda_j \geq (s/f) \cdot \lambda_i$ for all $j \geq i$. Since this is an infinite sequence of link rates, we can find a pair $\lambda_j \geq (s/f) \cdot \lambda_i$ such that $d_{max}(\lambda_i)$ and $d_{max}(\lambda_j)$ fall within the same interval of size ϵ, i.e. $|d_{\max}(\lambda_1) - d_{\max}(\lambda_2)| < \epsilon$. Let $C_1 = \lambda_i$ and $C_2 = \lambda_j$. The claim follows because the delay range for any link rate has size at most δ^{max}.

Step 2. Consider the trajectories of sending rate and delay when running a flow using CCA $\mathcal{A}$ independently on ideal paths with link rates C_1 and C_2. Recall that an ideal path has zero non-congestive delay. The figure below shows an example of a hypothetical delay-convergent algorithm that converges after times T_1 and T_2 in the two cases. The CCA converges to similar but distinct delay ranges on the two links. However, it converges to very different sending rates in the two cases. Let x_1 and x_2 denote the long-term throughput achieved on links of capacity C_1 and C_2 respectively. Clearly $x_1 \leq C_1$, and since CCA $\mathcal{A}$ is f-efficient, $x_2 \geq fC_2$. It follows that $x_2 \geq fC_2 \geq f \cdot (s/f)C_1 = s \cdot C_1 \geq s \cdot x_1$, where we have used the fact that $C_2 \geq (s/f)C_1$.



Step 3. To recap, so far we have identified two link rates C_1, C_2 such that the CCA converges to delays in a similar range but its sending rates are far apart on these links. In the final step, we construct a 2-flow scenario on a *shared* link with rate $C_1 + C_2$ and propagation delay R_m, such that the two flows follow exactly the same trajectories we found in Step 2. Therefore, in this scenario, the two flows converge to throughputs x_1 and x_2 that satisfy our starvation criteria: $x_2 \geq s \cdot x_1$.

Our starting observation is that for a deterministic CCA,[1] the sending rate at

[1]While CCAs such as BBR and PCC employ randomness, it does not materially affect the result (see §8).

any time t is a function of the delays observed up to time t and the initial state of the algorithm. Therefore, if we can set the non-congestive delays in the 2-flow scenario such that each flow observes a *total* delay that is identical to one of the delay trajectories from Step 2, then the two flows' sending rates will follow the rate trajectories from Step 2 as well. We will refer to controlling the non-congestive delay on a flow's path to achieve a specific delay trajectory as *emulating* that delay trajectory. The question is can we emulate the delay trajectories from Step 2 by adding up to D seconds of non-congestive delay to each flow's path in the 2-flow scenario?

To complete the construction of the 2-flow scenario, we must specify the initial state of the two flows' CCAs, the initial state of the shared link's queue, and the trajectories of the non-congestive delay for the two flows at all times. Let $d_1(t)$ and $d_2(t)$ be the delay trajectories and $r_1(t)$ and $r_2(t)$ be the rate trajectories achieved for links C_1 and C_2 respectively in Step 2. Assume that the flows converged to their eventual delay ranges at times T_1 and T_2, as shown in the figure above. Define $\bar{d}_1(t) = d_1(t + T_1), \bar{r}_1(t) = r_1(t + T_1), \bar{d}_2(t) = d_2(t + T_2), \bar{r}_2(t) = r_2(t + T_2)$ as time-shifted versions of the delay and rate trajectories such that the time origin is set to the time of convergence. These trajectories correspond to the bold segments shown in the figure.

We initialize the internal state of the two flows to the states of the corresponding flow in Step 2 at times T_1 and T_2. Our goal now is to emulate the delays $\bar{d}_1(t)$ and $\bar{d}_2(t)$ for all $t \geq 0$ by choosing the non-congestive delay for the two flows appropriately. Let $\delta_1^\star(t)$ and $\delta_2^\star(t)$ be the non-congestive delays for flows 1 and 2 respectively, and let $d^\star(t)$ be the sum of the propagation delay R_m and queuing delay in the 2-flow scenario.[2] Note that $d^\star(t)$ is common to both flows. To achieve emulation, we need to ensure that: $d^\star(t) + \delta_1^\star(t) = \bar{d}_1(t)$ and $d^\star(t) + \delta_2^\star(t) = \bar{d}_2(t)$ for all $t \geq 0$.

We control $\delta_1^\star(t)$ and $\delta_2^\star(t)$, but what is $d^\star(t)$? To get a handle on $d^\star(t)$, let's assume for the moment that our delay emulation is successful and the two flows send precisely at the rates $r_1(t)$ and $r_2(t)$. Then, we can compute the queuing delay in

[2]We use a superscript $\star$ for all signals in the 2-flow scenario.

the 2-flow scenario exactly: it is simply the delay of a queue with net arrival rate of $r_1(t) + r_2(t)$ and net drain rate of $C_1 + C_2$. In the proof (see App. A.5 for details), we use this observation to show that $d^\star(t)$ is given by:

$$d^\star(t) = \underbrace{\frac{C_1\bar{d}_1(t) + C_2\bar{d}_1(t)}{C_1 + C_2}}_{\text{Time-varying}} - \underbrace{(\delta^{\max} + \epsilon)}_{\text{Constant}}.$$

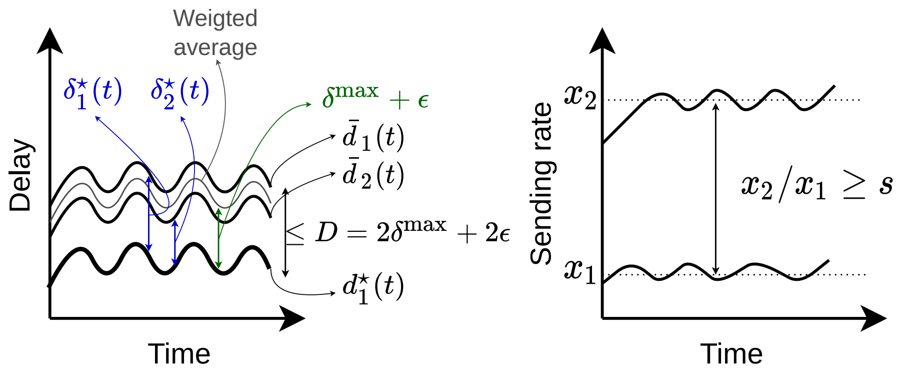
$d^\star(t)$ has two components: (i) a time-varying part that is a weighted average of the delay trajectories $\bar{d}_1(t)$ and $\bar{d}_2(t)$ from Step 2; (ii) a constant part that depends on the initial queuing delay chosen in the 2-flow scenario. This initial delay is $d^\star(0)$, and we are free to set it to any value $\geq R_m$.

For delay emulation to succeed, we must be able to satisfy $d^\star(t) + \delta_i^\star(t) = \bar{d}_i(t)$ for some $\delta_i^\star(t) \in [0, D]$ for all $t \geq 0$ and $i \in \{1, 2\}$. This can be done if and only if $d^\star(t)$ satisfies two properties:

1. $d^\star(t) \leq \min\{\bar{d}_1(t), \bar{d}_2(t)\}$, i.e. $d^\star(t)$ must be a lower bound on $\bar{d}_1(t)$ and $\bar{d}_2(t)$. This guarantees that the non-congestive delay is non-negative.

2. $\max\{\bar{d}_1(t), \bar{d}_2(t)\} < d^\star(t) + D$, i.e. $d^\star(t) + D$ must be an upper bound on $\bar{d}_1(t)$ and $\bar{d}_2(t)$. This guarantees that the non-congestive delay is $\leq D$.

The last step of the proof shows that we can choose the initial delay $d^\star(0)$ such that $d^\star(\cdot)$ satisfies both properties. We defer the details to Appendix A.5, but the reason this works is that the delay values for $\bar{d}_1(t)$ and $\bar{d}_2(t)$ are never too far from each other. Recall that Step 1 ensured that the delays lie within an interval of size $\delta_{max} + \epsilon$. In this step, we were free to pick any $\epsilon > 0$. To make our proof work in the appendix, we will pick $\epsilon = (D - 2\delta^{\max})/2$. Since we are given a $D > 2\delta^{\max}$ by the "invoker" of this theorem, $\epsilon > 0$. The figure below shows how $d^\star(\cdot)$ is situated relative to $\bar{d}_1(\cdot)$ and $\bar{d}_2(\cdot)$ for our running example.


Weigted average
$\delta_1^\star(t)$
$\delta_2^\star(t)$
$\delta^{\max} + \epsilon$
$\bar{d}_1(t)$
$\bar{d}_2(t)$
$\leq D = 2\delta^{\max} + 2\epsilon$
$d_1^\star(t)$
Delay
Time
Sending rate
x_2
x_1
$x_2/x_1 \geq s$
Time

Chapter 8

Starvation in the real-world

Extreme unfairness tantamount to starvation can occur when multiple flows share a bottleneck link but the rest of the path they traverse is different. This section shows that starvation is not merely theoretical, but can be observed in real-world delay-convergent CCAs even in simple settings. The scenarios we show are inspired by our proof.

We discuss several CCAs here, explaining why certain CCAs are delay-convergent and giving their oscillation range (i.e., $d_{\min}(C)$ and $d_{\max}(C)$). Then we describe the trace produced by Theorem 3 to cause one of the flows to starve and discuss how it can arise in realistic network path. The only non-delay-convergent CCAs we are aware of are loss based. We discuss them as well.

We have a simple criterion for deciding whether a network scenario is realistic. First, it should possible for a composition of real network elements to produce the behavior. Second, the behavior should not be a sequence of coincidences. That is, the probability of the behavior happening should not decrease with the duration of that behavior; if flows were to be on such a network, one of the flows must be very likely to starve. As we we will see, the paths we come up with are common on the Internet.

8.1 Vegas, FAST, and Copa

These CCAs all have the same equilibrium, though their dynamics differ. They all try to maintain a constant number (α) of packets in the queue. Even with a single flow, these algorithms can send at a rate that is arbitrarily smaller than the link rate. This happens when they overestimate their queuing delay and slow down. They can overestimate their queuing delay on paths with non-congestive delays or if they underestimate their minimum RTT, R_m.

Copa attempts to mitigate these problems by computing queuing delay as *standing RTT - min RTT*, instead of *latest RTT - min RTT* where *standing RTT* is the minimum RTT observed over a short period of time (*min RTT* is the minimum over a long period). Unfortunately, this method is not robust to persistent non-congestive delay. CCAC [12] found a way to use multiple network elements to fool this statistic even when there is no persistent non-congestive delay.

The following simple scenario drives Copa to starvation. Run a Copa flow on a 120 Mbit/s link with R_m of 60 ms. Send one packet with a 59 ms RTT to cause it to under-estimate its minimum RTT. Here Copa achieved a throughput of 8 Mbit/s, which was caused by a 1 ms error in measuring the delay of one packet. We did this experiment with the Mahimahi emulator [112].

This single-flow phenomenon also occurs with two or more flows. e.g., when the delay jitter happens only for one flow. We repeat the above experiment with two flows where only one flow gets the 59 ms packet. In this case, one flow gets 8.8 Mbit/s while the other gets 95 Mbit/s. Vegas and FAST can also be compromised in similar ways.

8.2 BBR

BBR has two modes of operation. The first mode is the one described in the original paper [31]. Here, BBR's sending rate is limited by its pacing rate, which is set to `(pacing gain)·(bandwidth estimate)`. The `pacing gain` is typically 1, allowing

BBR to send at its estimated bandwidth. Every 8 RTTs, `pacing gain` is increased to 1.25 to probe and see if more bandwidth is available. After this, `pacing gain` is reduced to 0.75 to drain any queue created during the gain. The `bandwidth estimate` is the maximum bandwidth measured over the last 10 RTTs, where bandwidth is measured by dividing the number of acknowledged bytes over 1 RTT intervals.

If more bandwidth were available during the probe phase (or at any other time), `bandwidth estimate` would have increased. The way BBR seeks to achieve fairness is by having different flows probe at different random times, as described in a fairness analysis document [140]. In the pacing mode, $d_{\min}(C) = R_m$ and $d_{\max}(C) = 1.25R_m$. If $D > \delta^{\max} = 0.25R_m$, our network model can prevent one of the flows from recognizing that additional bandwidth is available during the probe phase, causing it to send at a rate that is arbitrarily small compared to its fair share. This situation is identical to the one described in the CCAC paper [12] and happens in the presence of a network element, such as a cellular base station, whose bandwidth allocation lags behind the flow's demand. Our proof constructs exactly this behavior. By contrast, if D is smaller, BBR can be broken in the `cwnd`-limited mode that is described below.

Because the bandwidth estimate uses a max filter, BBR tends to over-estimate the link rate when ACKs do not arrive smoothly, since there will be some RTT during which we get greater than average rate. As a result, the queue can grow indefinitely. To prevent such buffer-bloat, BBR incorporates uses `cwnd` to cap the number of in-flight packets. Hence when BBR has overestimated the bandwidth, it is in the `cwnd`-limited mode [64, 142]. In this mode, `cwnd` controls the behavior and dynamics of the pacing rate and its increase/decrease during bandwidth probing are not material to the sender's transmissions.

Starvation in `cwnd`-limited mode Here, `cwnd` is set to $2\cdot$`(bandwidth estimate)`$\cdot$`(`R_m `estimate)` $+\ \alpha$. The α term is called quanta in the BBR document [32], and is intended to "allow enough quanta in flight on the sending and receiving hosts to reach high throughput even in environments using offload mechanisms". This term was removed in a later version, but another additive term `extra_acked` was added in its stead [33]. We believe the α performs a critical function in addition to the intended one; it enables fairness in `cwnd`-limited mode by forcing a unique fixed point.

We now calculate the equilibrium point for BBR on an ideal path. At equilibrium, its bandwidth estimate equals the ACK arrival rate, which equals the sending rate, cwnd/RTT. Hence we have

$$\begin{aligned}\text{cwnd} &= 2R_m \cdot (\texttt{bandwidth\_estimate}) + \alpha \\ &= 2R_m \cdot \text{cwnd}/\text{RTT} + \alpha\end{aligned}$$

Thus, $\texttt{equilibrium\_sending\_rate} = \frac{\text{cwnd}}{\text{RTT}} = \frac{\alpha}{\text{RTT}-2R_m}$ (Figure 3-4). At equilibrium, RTT $> 2R_m$, achieving full utilization.

When there is only one flow, the sending rate is C because the link is fully utilized. This gives cwnd $= CR_m + \alpha$. We can repeat this calculation for multiple flows using the additional constraint that ACKs for the i^{th} flow arrive at the rate of $C\frac{\texttt{cwnd}_i}{\sum_{j=1}^{n}\texttt{cwnd}_j}$. Then we get $\texttt{cwnd}_i = 2CR_m/n + n\alpha$. At this equilibrium, the queuing delay is $2R_m + n\alpha/C$.

This behavior is similar to Vegas, FAST, and Copa where at equilibrium the queuing delay was $R_m + n\alpha/C$. The only difference is that BBR maintains an extra R_m of delay. But this is an important difference; unless BBR overestimates the congestive delay by R_m, it maintains non-zero queuing delay and achieves full utilization. In contrast, even a single Vegas/FAST/Copa flow can under-utilize the link if they mis-estimate the RTT by α/C. However, *fairness* is still achieved for BBR by the $n\alpha/C$ term. If we remove the $+\alpha$ term and recalculate the equilibrium, we find that any value of $\texttt{cwnd}_1$ and $\texttt{cwnd}_2$ can be a fixed point as long as $\texttt{cwnd}_1 + \texttt{cwnd}_2 = 2R_mC$, even if $\texttt{cwnd}_1 = 0$ and $\texttt{cwnd}_2 = 2R_mC$! If one BBR flow is running and has occupied the entire link, when a new flow comes, it will not achieve its fair share. While the $+\alpha$ term fixes the problem, $n\alpha/C$ is a rather small value of delay to measure and becomes smaller as C grows. Hence the same precision is required as in Vegas, FAST and Copa. The analysis suggests that when flows with different RTTs compete, the smaller RTT starves, as has been observed empirically before [64].

Empirical evaluation BBR is a complex protocol, spanning 900 lines of code. To confirm that our simplified theoretical model for BBR is useful, we conducted some experiments. We used Mahimahi [112] to run two BBR flows (implemented in the Linux kernel v5.13.0) with R_m of 40ms and 80ms over a common bottleneck link of 120 Mbit/s for 60 seconds. Since there were two BBR flows, their interaction and natural OS jitter was enough to push them into `cwnd`-limited mode. In this situation, one flow got an average of 8.3 Mbit/s and the other got 107 Mbit/s, an order-of-magnitude difference in sending rates.

Delay-convergence in BBR Strictly speaking, the `cwnd`-limited mode does not meet our definition for delay-convergent CCAs with $\delta(C) = 0$ because (1) some jitter is necessary for this mode to be active while our definition is over ideal links and (2) BBR periodically stops transmitting to probe for minimum RTT. During a probe, RTT falls to R_m, so $\delta(C)$ is $R_m \neq 0$. These are, however, mere technicalities and our starvation proof still works. First, instead of running in an ideal link, we need to run on a link with some jitter. Second, our proof works even if the CCA has oracular knowledge of R_m; alternatively, we can stop emulation when the CCA is probing for RTT and the rest of the argument holds, since BBR will ignore the data collected during probe. Further, BBR employs randomness in when it probes for bandwidth while our theorem only applies to deterministic CCAs. However, this does not make a difference as demonstrated by the empirical experiment above.

8.3 PCC Vivace

The PCC Vivace paper [39] showed that on an ideal link it converges to a fair throughput allocation that fully utilizes the link and maintains zero queuing delay. It regularly increases and decreases its rate to check if that will increase its utility function. Based on the largest constants given in the paper this will cause the queuing delay to oscillate at most between R_m and $1.05R_m$. These form $d_{\min}(C)$ and $d_{\max}(C)$ with $\delta^{\max} = R_m/20$. PCC's rate-delay curve is shown in Figure 3-4. Like BBR, PCC also

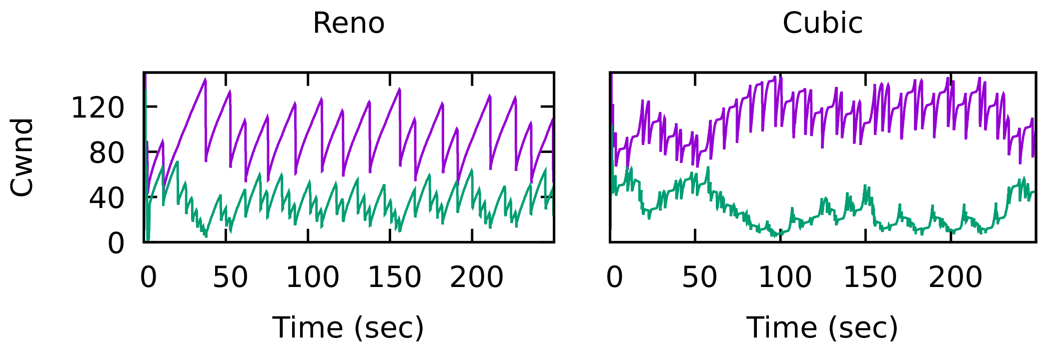


Figure 8-1: Congestion window evolution when two flows are run on a 6 Mbit/s, 60 ms link and 60 packets (1 BDP) of buffer. The lower flow's receiver uses delayed ACKs of up to 4 packets while the other ACKs every packet. The CCAs used are Reno (left) and Cubic (right). The ratio of throughput obtained between the two flows is 2.7× and 3.2× for Reno and Cubic respectively.

employs randomness, but it this does not make a difference to the result. (In fact, we conjecture that Theorem 3 is true for randomized CCAs too.)

To empirically test if PCC experiences starvation, we ran two PCC Vivace flows in a Mahimahi emulator with 60 ms propagation delay and 120 Mbit/s bandwidth. For one of the flows, ACKs are received only at integer multiples of 60 ms, preventing finer delay measurement. This flow only achieved 9.9 Mbit/s while the other flow got 99.4 Mbit/s. We used Vivace's kernel module for the experiments [74]. [1]

8.4 Loss-based CCAs

CCAs like NewReno [65] or Cubic [61] are not delay-convergent. We study their fairness in two ways. First, we extended CCAC to handle multiple flows (see Appendix A.1) and used it to discover bad behavior when there is non-congestive delay jitter. Second, we modify our model to allow it to preferentially drop packets for one flow.

Let us take delay jitter first. Suppose two flows share a bottleneck, but one of

[1]We needed a relatively large jitter of 60 ms because Mahimahi is a noisy emulator. Linux *user-space* scheduling adds several ms of jitter to *both flows*. We need the flows to have different jitter. A cleaner, less-variable network emulation environment will produce a configuration with smaller non-congestive jitter for starvation to occur.

them is well-paced while the other sends packets in bursts. This situation can occur with generic segment offloading (GSO) [1] used by the sender for CPU efficiency, ACK aggregation (say due to WiFi [59]), or delayed ACKs [72, 26]. As the queue gets nearly full, the flow that sends packets in bursts is more likely to lose packets. When this happens, this flow reduces its `cwnd` and the queue stops being full until a while later when again the bursty flow is more likely to lose packets. Packet bursts can reduce the utilization even when there is only a single flow on the link, but only when the bursts are large [12]. However when two or more flows are present, even a small burst can cause unfairness. This is illustrated using an ns3 [3] simulation in Figure 8-1 and was alluded to earlier (see §1.1). We also reproduced similar results in emulation in Mahimahi.

One flow using delayed ACKs of 4 packets can cause it to get 1/3 the throughput of the other flow. Nevertheless this is not starvation since the unfairness is bounded. In loss-based AIMD, when the faster flow reduces its `cwnd` (which it eventually must when it occupies nearly all of the link), it gives the slower flow time to ramp up before it starts to decrease its `cwnd` again. In Cubic, the faster flow will eventually overshoot the entire bandwidth-delay product (BDP) as governed by the cubic function. The slower flow can only increase its `cwnd` and experience losses two or three times before this happens. Hence the unfairness is bounded. We used CCAC to prove that there is no trace of length 10 RTTs where starvation is unbounded for two AIMD flows. Proving this result for any trace length is future work.

PCC Allegro [38] While loss-based AIMD is not delay-convergent and is therefore immune to small delay jitter, it converges to a loss rate as a function of the BDP [103]: $\text{cwnd} \propto \frac{1}{\sqrt{\text{loss rate}}}$. If we add to the network model the ability to arbitrarily drop a small fraction of packets, we can get the CCA to under-utilize the link when the BDP is sufficiently large. This is well known and works even when only a single flow is present.

PCC's behavior is more interesting. To get around this problem, the loss-based PCC variant, PCC Allegro, has a loss threshold that it can tolerate. As long as the packet loss rate is lower than this threshold, it will fully utilize the link. In our framework, this is analogous to BBR in `cwnd` mode always maintaining R_m seconds of queuing; as long as error in delay measurement is smaller than R_m, BBR fully utilizes the link. Just as BBR is an improvement to the Vegas family, PCC is a loss-resilient improvement to the Reno family. However, just like BBR, PCC can also experience starvation when one of the flows (but not the other) experiences even small amounts of congestion signal; for BBR it is propagation delay, for PCC it is random loss. The reason is analogous; the space of loss rates is smaller than the space of sending rates.

As empirical validation, we ran two PCC flows for 60 seconds on a 120 Mbit/s Mahimahi link with 40 ms RTT and 1 BDP buffer. One of the flows experienced a random loss rate of 2% and got only 10.3 Mbit/s while the other, which experienced no random loss, got 99.1 Mbit/s. PCC is supposed to be resilient to up to 5% loss. Indeed, when we ran two flows with 2% loss, they shared the link fairly and efficiently. The same held true with one flow with 2% loss. Like BBR, PCC breaks only when two flows are present and experiences unequal congestion signal (here, loss). We do not believe it is possible to circumvent this problem with algorithms that map loss rates (or delays) to sending rates.

Chapter 9

Implications of the starvation result

The main lesson from this paper is that to avoid starvation delay-convergent CCAs must explicitly model and design for non-congestive delays. This affects the design space of CCAs in three key ways:

1. If D is the bound on network jitter, the CCA must maintain at-least D seconds of delay to be f-efficient (§9.1).
2. To avoid starvation, it is not enough to maintain a queue that is larger D, but the *variation* in delay must be larger as well (see §9.2).
3. If we have an upper bound on the link rate, then we can achieve all three objectives without large variance in queuing delay. The delay bound achieved is a function of D and our maximum tolerable unfairness (§9.3).

The rest of this section expands on these ideas, and concludes by proving an impossibility result for delay-bounded (non-delay-convergent) CCAs.

9.1 Is an f-efficient, delay-convergent CCA achievable?

Can a CCA simultaneously achieve f-efficiency and delay-convergence if we can tolerate starvation? The answer is not obvious because current schemes like BBR, Copa, Vegas, and FAST do not, as shown in the recent CCAC paper [12]. That paper found counterexamples consistent with the delay-jitter network model of this paper showing scenarios where f-efficiency is not achieved for any $f > 0$.

We are not aware of any existing CCAs that are both f-efficient and delay-convergent. That said, we have not analyzed every algorithm in depth. Perhaps the best hope is offered by BBR. The CCAC paper showed that if BBR were modified to have a higher pacing rate, CCAC could no longer find any example where BBR under-utilizes the link. We believe this happens because the higher pacing rate forces BBR to operate in the `cwnd`-limited mode, since the behavior is identical to when BBR over-estimates the pacing rate (see §8.4). In this mode, $d_{\max}(C) \geq 2R_m$, which is large.

Note, however, that CCAC did not *prove* that BBR is f-efficient. It merely ruled out the existence of under-utilization over short ($\leq$ 10 RTTs) sequences of network behavior. It also assumed a sender with oracular knowledge of R_m.

We conjecture, however, that it is possible to design an f-efficient, delay-convergent CCA if we ignored starvation. Perhaps the modified BBR is such an algorithm. Any such CCA must maintain a larger delay than the network jitter, or risk under-utilization. The following lemma formalizes this.

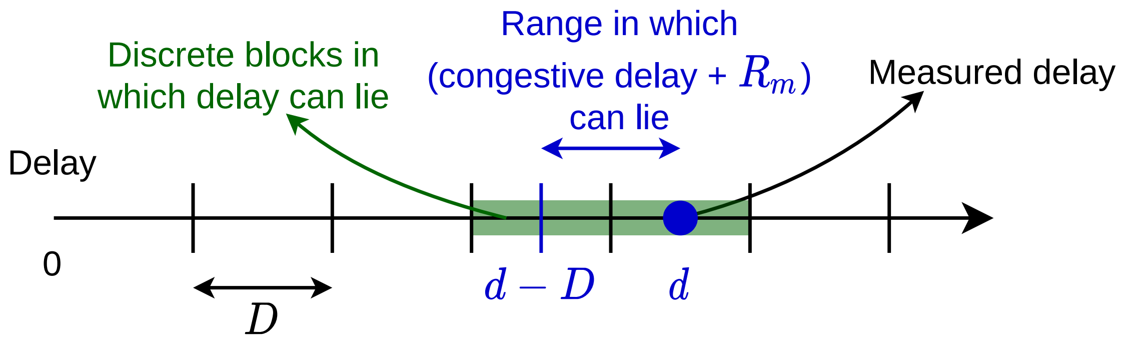
Theorem 1. *Any deterministic CCA for which there exists a link rate C and minimum RTT R_m such that $d_{\max}(C) \leq D$ can experience arbitrarily low utilization in our network model with parameter D.*

Proof. The idea is similar to our proof for Theorem 3. Let the delay experienced by the CCA on an ideal path of rate C and propagation delay R_m be $d(t)$. Let its sending rate be $r(t)$. We will construct a network with propagation delay R_m and $C' \gg C$ such that the delay experienced by the CCA is exactly $d(t)$. As a result, the CCA will transmit at exactly $r(t)$ since the CCA is deterministic. This is possible because we will choose C' to be large enough that the queuing delay, $q(t) \leq d(t)$. Now $d(t) \leq d_{\max}(C) \leq D$ where the first inequality follows from the definition of $d_{\max}(\cdot)$. Hence $0 \leq d(t) - q(t) \leq D$, which is the condition we need for emulation. Since the actual link rate, C', can be arbitrarily larger than the rate $\approx C$ at which the CCA sends, starvation occurs. □

9.2 Larger oscillations may avoid starvation

Theorem 3 shows that a CCA whose ideal-path delay variation, $\delta^{\max}$, is smaller than one-half of the non-congestive delay in the network, D, cannot simultaneously be f-efficient, bound delays, and avoid starvation. Hence the only way to achieve all three properties is to design a CCA that has large delay variation on ideal paths (i.e., at equilibrium).

Why might large delay variations avoid starvation? The reason why CCAs with small variations starve is that there isn't enough space to assign all achievable rates to distinct-enough delay ranges. This is because once a CCA has converged to a small delay range, it keeps receiving the same signal over and over. It cannot distinguish delay variations due to congestion from those due to non-congestive jitter.


Discrete blocks in which delay can lie
Range in which (congestive delay + R_m) can lie
Measured delay
Delay
0
D
$d - D$
d

An imprecise but useful mental model is to think of measurement ambiguity as discretizing measurement. When we measure an RTT of d, we know that up to D of it may be non-congestive. The material portion of d for our purposes is in the range $[\max(0, d - D), d]$. Let us divide the RTT d into discrete blocks of size D. This range tells us that the correct (`congestive delay +` R_m) must lie in one of the two blocks highlighted in the picture above.

A delay-convergent algorithm with $\delta^{\max} \leq D$ gets the same set of blocks over and over again. But one with larger variation can get different blocks/bits of information each time. This forms an infinitely large stream of bits in which to encode the correct sending rate. Different bit streams can now be assigned to different sending rates. This helps sidestep the pigeonhole argument, which forms the bedrock of our impossibility proof.

For instance, sending rate can be encoded in the *frequency* of oscillation of delay, rather than its absolute value. Given enough samples, it may be possible to measure frequency with arbitrary precision, avoiding starvation. Loss-based algorithms like AIMD do this; AIMD's sending rate is determined by the frequency at which packets drop. While our analysis does not include packet losses, it is interesting to note that AIMD has large oscillations relative to D. Hence smaller delay jitter does cause starvation (§8.4). If the oscillations (and hence the buffer size) were smaller than D, it is indeed possible to starve AIMD; one flow always sends packets in bursts that are larger than the buffer, experiencing drops, while the other flow grows its `cwnd` to be arbitrarily larger than the first flow. We conjecture that AIMD on delay is an interesting design space for researchers to seek starvation-free CCAs.

9.3 Avoiding starvation in a bounded rate range

A CCA that seeks to converge to a small range of delays must map sending rates that are far away to delays that are more than D apart, or risk starvation. While this is not possible for an infinitely large range of rates, it becomes possible if we know that the correct sending rate will come from a bounded range. In practice, we may know such bounds *a priori* from a knowledge of network parameters (e.g., access link rate at the sender) or by profiling applications.

Rather than worry about perfect flow-level fairness, which might not be an interesting goal in practice [28], we seek to be s-fair; i.e., bound unfairness to a maximum specified throughput ratio of $s > 1$. For a given D, and maximum tolerable delay $R^{\max}$, we define a figure-of-merit for a rate-delay curve as the ratio of the maximum rate it supports to the minimum: $\frac{\mu_+}{\mu_-}$. Rates in today's Internet can span several orders of magnitude, from < 100 KBit/s to ≈ 10 Gbit/s. Hence having $\frac{\mu_+}{\mu_-} \geq 10^3$ and perhaps $\approx 10^5$ is desirable.

For Vegas, FAST, and Copa, the rate-delay function is $\mu(d) = \alpha/(d - R_m)$, where μ is the sending rate [13]. The function for BBR's `cwnd`-limited mode is $\mu(d) = \alpha/(d - 2R_m)$. The arguments in this section are similar for both these functions, so we analyze the Vegas/FAST/Copa function here.

To achieve s-fairness for all $\mu \in [\mu_-, \mu_+]$, we want the difference in the delays seen between μ and $s\mu$ to exceed D. This gap will ensure that rates that are more than s away from each other are mapped to distinguishable delays. The difference in delays for our rate-delay function is:

$$\begin{aligned}&\left(R_m + \frac{\alpha}{\mu}\right) - \left(R_m + \frac{\alpha}{s\mu}\right) > D\\ \Rightarrow &\mu < \frac{\alpha}{D}\left(1 - \frac{1}{s}\right).\end{aligned}$$

This gives us μ_+. μ_- is the rate corresponding to $d = R^{\max}$, so $\mu_- = \alpha/(R^{\max} - 2R_m)$. Hence, for the Vegas family,

$$\frac{\mu_+}{\mu_-} = \frac{R^{\max} - R_m}{D}\left(1 - \frac{1}{s}\right) \approx \frac{R^{\max}}{D}. \tag{9.1}$$

We can, however, do much better with

$$\mu(d) = \mu_- s^{\frac{R^{\max} - d}{D}} \tag{9.2}$$

When $d = R^{\max}$, $\mu = \mu_-$ as desired. μ_+ occurs when $d = R_m + D$. This is the minimum RTT required to ensure full utilization (Lemma 1). Hence, $\mu_+ = \mu_- s^{(R^{\max} - (R_m + D))/D}$. Thus,

$$\frac{\mu_+}{\mu_-} = s^{\frac{R^{\max} - R_m - D}{D}} = O\left(s^{\frac{R^{\max}}{D}}\right).$$

This range is exponentially larger than the Vegas family and can span several orders of magnitude of link rates, as desired. For instance, for $D = 10$, $s = 2$ and $R^{\max} = 100$

ms we can support a range of $2^{10} \approx 10^3$. With $s = 4$, that increases to $2^{20} \approx 10^6$.

A real-world CCA may make the following modifications to this rate-delay curve. (1) This function never increases its rate beyond μ_+. This is simply solved by using a Vegas-like function for $d < D$ that goes to infinity. Such a CCA will scale to arbitrarily large link rates, but risk starvation when rate exceeds μ_+. (2) If the CCAs have a method to estimate R_m, they can set $R^{\max}$ as a function of R_m: e.g. $R^{\max} = R_m + 100$ ms. Note that both Equation 9.2 and the Vegas family can send at rates lower than μ_-, but will increase delay beyond $R^{\max}$. For rates $< \mu_-$, delays increase more slowly for the Vegas family than for Equation 9.2.

Algorithm 1 A delay-convergent CCA that uses $\mu_- s^{\frac{R^{\max}-(d-R_m)}{D}}$. The following is run every R_m. Here, d is the latest measured RTT, μ is the current sending rate and $a, 0 < b < 1, \mu_-, R^{\max}$ are parameters of the algorithm.

```
if $\mu < \mu_- s^{\frac{R^{\max}-(d-R_m)}{D}}$ then
    $\mu \leftarrow \mu + a$
else
    $\mu \leftarrow b\mu$
end if
```

An Algorithm Algorithm 1 shows a CCA that uses Equation 9.2. This algorithm is incomplete on several fronts. It does not feature a mechanism to discover R_m or handle short buffers by slowing down in the presence of loss. It increases its rate additively, and does not feature the faster increase times of modern algorithms. Further it does not have a `cwnd` cap to be resilient to sudden drops in link capacity [18]. We show this merely to illustrate an idea, not propose a deployable CCA.

To verify this algorithm, we used CCAC to produce traces where the algorithm is either inefficient or more than s-unfair. CCAC was unable to produce such traces, giving us some confidence that they key ideas work. CCAC helped us fine-tune some details of the algorithm such as a) use AIMD instead of the AIAD uses by Vegas and Copa because the fairness properties of AIMD are critical in the presence of measurement ambiguity and b) change the rate by the same amount on every RTT irrespective of the number of ACKs received. Note that this does not constitute a proof, since CCAC only searched over finite traces. We have not yet performed the steady-state analysis method described in the CCAC paper.

Estimating R_m is a common challenge for delay-convergent CCAs, and may require fundamental new insights from the community. Estimating R_m is hard because it requires all flows to coordinate and empty the queue at the same time. Copa's mechanism works well in the absence of delay ambiguity, but not otherwise. BBR's mechanism works when there is a single flow, but its RTT probe does not always succeed in coordinating across multiple flows.

9.4 An absolute upper bound

We have also proved that *no* deterministic CCA can be simultaneously f-efficient, delay-bounding (but not delay-convergent, i.e., the delay bounds can grow with the bottleneck rate), and starvation-free. This theorem uses a stronger network model than those in Section 4; here the adversary can also vary the link rate arbitrarily. We call this the “strong” model. Since there are no bounds on the link rates, this adversary is very powerful. Perhaps too powerful, for it can create unrealistic networks. Thus we still believe that it may be possible to achieve all three properties on *practical* networks. Nevertheless, it serves as a useful upper bound on what is possible. The proof technique is interesting in that we have found it instructive to study the network paths it constructs. It often constructs paths similar to ones constructed by Theorem 3, i.e., consistent with our simpler network model.

Theorem 4. *Any deterministic, f-efficient, delay-bounding CCA will starve in the strong model for any value of the propagation delay R_m.*

The proof is given in Appendix A.6. This theorem does not need to control the initial conditions or require both CCAs to be the same.

Chapter 10

Limitations of our congestion control analysis method

We believe that this thesis sets the initial steps towards a more comprehensive and formal understanding of the behavior of CCAs, raising new challenges and opportunities. Thus far, we have focused on end-to-end CCAs and not yet analyzed receiver-driven protocols, MPTCP [146], schemes using in-network signals such as ECN [44], INT [81], XCP/RCP [79, 40], and ABC [59]. We discuss a possible way to handle these in Appendix A.1.

Because our modeling focuses on worst-case behavior, a central goal of its design is to exclude excessively antagonistic behavior that no CCA can handle. Thus, the path model only includes a carefully chosen subset of paths (see §4.2.1). Our model of TCP timeouts is different from the standard for the same reason (see §4.2.2). We represent network state using cumulative functions that preclude the modeling of packet reordering. CCAC's mechanism to detect packet losses emulates endpoints that use an unbounded number selective acknowledgment (SACK) blocks. This corresponds to the QUIC protocol [71], while TCP is limited to a maximum of four SACK blocks [102].

Our model does not have a composition theorem when buffers are finite and the first box is faster than the second one (see §4.2.1). Due to discretization, the bounds CCAC produces may not be tight (see §5). CCAC focuses on worst-case analysis;

this was a conscious design choice because average-case analysis requires a probability distribution, which is often unknown and can miss important tail cases (§1.2.1).

CCAC's CCA implementations are simplified (see §6). Future work can provide a higher-level interface to writing CCAs, which makes implementing more complex CCAs easier. CCAC does not have an automated method to map a network trace to actual network elements that could produce the trace, though in our experience we were always able to find such elements. There is work in verifying the *implementation* of CCAs [135, 132, 133, 134, 23] which is complementary with CCAC's verification of the *algorithm*. End-to-end verification of both the algorithm and its implementation is also interesting future work. Currently proving statements about infinite time horizons and variable link rates (see §4.3) is semi-automated, and requires a manual component as illustrated in our case studies.

Our starvation theorem shows that delay-convergent CCAs cannot be efficient and delay-bounding. While we conjecture that starvation-freedom, efficiency and bounded delay are simultaneously achievable if we give up on delay-convergence and deliberately oscillate self-inflicted delay, we do not conclusively settle this question. More broadly, our methods have uncovered weaknesses in existing CCAs, but a CCA that is free of weaknesses still remains elusive. Our ongoing work on systematically designing heuristics in general, and CCAs in particular, may find such a CCA (see §12.1).

Chapter 11

Copa: A new delay-based congestion control algorithm

Before we worked on formally understanding CCA performance, we designed Copa, a delay-based CCA that emphasized correctly interpreting delay measurements in the presence of non-congestive delay. Facebook uses Copa for live video uploads because Copa improves its end-to-end application metrics [51].

Copa incorporates four ideas: first, a target rate to aim for, which is inversely proportional to the measured queuing delay; second, a window update rule that depends moves the sender toward the target rate; third, a way to filter our non-congestive delay by using the minimum RTT over a short time window; and fourth, a TCP-competitive strategy to compete well with buffer-filling flows.

Approach: Inspired by work on Network Utility Maximization (NUM) [80] and by machine-generated algorithms, we start with an objective function to optimize. The objective function we use combines a flow's average throughput, λ, and packet delay (minus propagation delay), d: $U = \log\ \lambda - \delta \log\ d$. The goal is for each sender to maximize its U. Here, δ determines how much to weigh delay compared to throughput; a larger δ signifies that lower packet delays are preferable.

We show that under certain simplified (but reasonable) modeling assumptions of packet arrivals, the steady-state sending rate (in packets per second) that maximizes

U is

$$\lambda = \frac{1}{\delta \cdot d_q}, \tag{11.1}$$

where d_q is the mean per-packet queuing delay (in seconds), and $1/\delta$ is in units of MTU-sized packets. When every sender transmits at this rate, a unique, socially-acceptable Nash equilibrium is attained.

We use this rate as the *target rate* for a Copa sender. The sender estimates the queuing delay using its RTT observations, and moves quickly toward hovering near this target rate. This mechanism also induces a property that the queue is regularly almost flushed, which helps all endpoints get a correct estimate of the minimum RTT. Finally, to compete well with buffer-filling competing flows, Copa mimics an AIMD window-update rule when it observes that the bottleneck queues rarely empty.

11.1 The algorithm

Copa uses a congestion window, `cwnd`, which upper-bounds the number of in-flight packets. On every ACK, the sender estimates the current rate $\lambda = \texttt{cwnd}/\text{RTTstanding}$, where RTTstanding is the smallest RTT observed over a recent time-window, τ. We use $\tau = \text{srtt}/2$, where srtt is the current value of the standard smoothed RTT estimate. RTTstanding is the RTT corresponding to a "standing" queue, since it's the minimum observed in a recent time window.

The sender calculates the target rate using Eq. (11.1), estimating the queuing delay as

$$d_q = \text{RTTstanding} - \text{RTTmin}, \tag{11.2}$$

where RTTmin is the smallest RTT observed over a long period of time (larger of 10 s or 20$\times$ `RTT`). We find a minimum over a time-period to handle route-changes that might alter the minimum RTT of the path.

If the current rate exceeds the target, the sender reduces `cwnd`; otherwise, it increases `cwnd`. To avoid packet bursts, the sender paces packets at a rate of $2 \cdot \texttt{cwnd}/\text{RTTstanding}$ packets per second. Pacing also makes packet arrivals at the bottleneck queue appear Poisson as the number of flows increases, a useful property that increases the accuracy of our model to derive the target rate (§11.3). The pacing rate is *double* `cwnd`/RTTstanding to accommodate imperfections in pacing; if it were exactly `cwnd`/RTTstanding, then the sender may send slower than desired.

The reason for using the smallest RTT in the recent $\tau = \text{srtt}/2$ duration, rather than the latest RTT sample, is for robustness in the face of ACK compression [155] and network jitter, which increase the RTT and can confuse the sender into believing that a longer RTT is due to queuing on the forward data path. ACK compression can be caused by queuing on the reverse path and by wireless links.

The Copa sender runs the following steps on each ACK arrival:

1. Update the queuing delay d_q using Eq. (11.2) and srtt using the standard TCP exponentially weighted moving average estimator.

2. Set $\lambda_t = 1/(\delta \cdot d_q)$ according to Eq. (11.1).

3. If $\lambda = \texttt{cwnd}/\text{RTTstanding} \le \lambda_t$, then $\texttt{cwnd} = \texttt{cwnd} + v/(\delta \cdot \texttt{cwnd})$, where v is a "velocity parameter" (defined in the next step). Otherwise, $\texttt{cwnd} = \texttt{cwnd} - v/(\delta \cdot \texttt{cwnd})$. Over 1 RTT, the change in `cwnd` is thus $\approx v/\delta$ packets.

4. The velocity parameter, v, speeds-up convergence. It is initialized to 1. Once per window, the sender compares the current `cwnd` to the `cwnd` value at the time that the latest acknowledged packet was sent (i.e., `cwnd` at the start of the current window). If the current `cwnd` is larger, then set *direction* to "up"; if it is smaller, then set *direction* to "down". Now, if *direction* is the same as in the previous window, then double v. If not, then reset v to 1. However, start doubling v only after the direction has remained the same for three RTTs. Since direction may remain the same for 2.5 RTTs in steady state as shown in figure 11-1, doing otherwise can cause v to be > 1 even during steady state. In steady state, we want $v = 1$.

When a flow starts, Copa performs slow-start where `cwnd` doubles once per RTT until λ exceeds λ_t. While the velocity parameter also allows an exponential increase, the constants are smaller. Having an explicit slow-start phase allows Copa to have a larger initial `cwnd`, like many deployed TCP implementations. Further we limit v to ensure that `cwnd` can never more than double once per RTT.

11.1.1 Competing with buffer-filling schemes

We now modify Copa to compete well with buffer-filling algorithms such as Cubic and NewReno while maintaining its good properties. The problem is that Copa seeks to maintain low queuing delays; without modification, it will lose to buffer-filling schemes.

We propose *two distinct modes* of operation for Copa:

1. The *default mode* where $\delta = 0.5$, and
2. A *competitive mode* where δ is adjusted dynamically to match the aggressiveness of typical buffer-filling schemes.

Copa switches between these modes depending on whether or not it detects a competing long-running buffer-filling scheme. The detector exploits a key Copa property that the queue is empty at least once every $5 \cdot$ `RTT` when only Copa flows with similar RTTs share the bottleneck (Section 11.2). With even one concurrent long-running buffer-filling flow, the queue will not empty at this periodicity. Hence if the sender sees a "nearly empty" queue in the last 5 RTTs, it remains in the default mode; otherwise, it switches to competitive mode. We estimate "nearly empty" as any queuing delay lower than 10% of the rate oscillations in the last four RTTs; i.e., $d_q < 0.1(\text{RTTmax} - \text{RTTmin})$ where RTTmax is measured over the past four RTTs and RTTmin is our long-term minimum as defined before. Using RTTmax allows Copa to calibrate its notion of "nearly empty" to the amount of short-term `RTT` variance in the current network.

In competitive mode the sender varies $1/\delta$ according to whatever buffer-filling algorithm one wishes to emulate (e.g., NewReno, Cubic, etc.). In our implementation we perform AIMD on $1/\delta$ based on packet success or loss, but this scheme could respond to other congestion signals. In competitive mode, $\delta \leq 0.5$. When Copa switches from competitive mode to default mode, it resets δ to 0.5.

The queue may be nearly empty even in the presence of a competing buffer-filling flow (e.g., because of a recent packet loss). If that happens, Copa will switch to default mode. Eventually, the buffer will fill again, making Copa switch to competitive mode.

Note that if some Copa flows are operating in competitive mode but no buffer-filling flows are present, perhaps because the decision was erroneous or because the competing flows left the network, Copa flows once again begin to periodically empty the queue. The mode-selection method will detect this condition and switch to default mode.

11.1.2 Application-layer benefits

Many applications benefit from accurate information about path throughput and delay. For example, recently there has been a surge of interest in video streaming, where one of the primary challenges is in estimating the correct bit rate to use. A low estimate hurts video quality while a high estimate risks experiencing a stall in playback. Most algorithms tend to under-estimate rates because stalls hurt the quality of experience more. That, in turn, means they are unable to effectively obtain the true usable path rate.

We showed how every measurement of the queuing delay provides a new estimate of the target rate. Hence, to understand what throughput and delay can be expected from a path, an endpoint only needs to transmit a few packets. The expected performance can be calculated from the measured RTT and queuing delay. These packets can be small, containing only the header and no data, which reduces the bandwidth consumed by probes. Applications can use this information in many ways.

Copa offers a way for applications to obtain rate information. Senders can use the techniques we have developed to measure "expected throughput" – i.e., the rate that a Copa sender will use – by sending only a few small packets, and take an informed decision regarding what quality of content to transfer. As shown in §11.4.1, Copa's rate estimates are accurate and senders are able to jump directly to the correct rate.

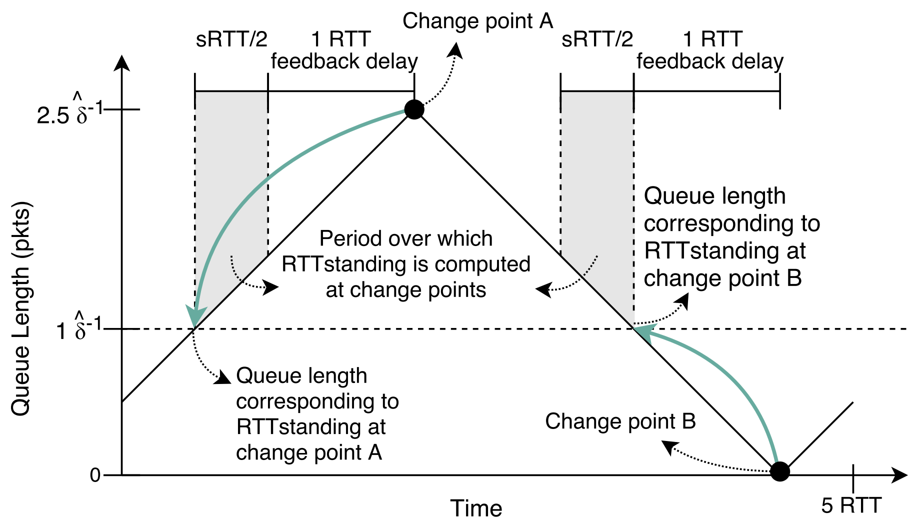

sRTT/2
1 RTT feedback delay
Change point A
sRTT/2
1 RTT feedback delay
2.5 $\hat{\delta}^{-1}$
Queue Length (pkts)
1 $\hat{\delta}^{-1}$
0
Period over which RTTstanding is computed at change points
Queue length corresponding to RTTstanding at change point B
Queue length corresponding to RTTstanding at change point A
Change point B
Time
5 RTT

Figure 11-1: One Copa cycle: Evolution of queue length with time. Copa switches direction at change points A and B when the standing queue length estimated by RTTstanding crosses the threshold of $\hat{\delta}^{-1}$. RTTstanding is the smallest RTT in the last srtt/2 window of ACKs packets (shaded region). Feedback on current actions is delayed by 1 RTT in the network. The slope of the line is $\pm\hat{\delta}$ packets per RTT.

Quick estimation of a transport protocol's expected transmission rate is also useful for selecting good paths or endpoints. For instance, peer-to-peer networks can regularly send tiny packets without payload to monitor the throughput and delay available on the link to a peer. The monitoring is inexpensive, but can enable more informed decisions. Content Distribution Networks using Copa for data delivery can use this too, by routing requests to the appropriate servers. For instance, they can route to minimize the flow-completion time estimated as $2 \cdot \text{RTT} + l/\lambda$, where l is the flow length and λ is the rate estimate.

11.2 Dynamics of Copa

Figures 11-1 (schematic view) and 11-2 (emulated link) show the evolution of Copa's `cwnd` with time. In steady state, each Copa flow makes small oscillations about the target rate, which also is the equilibrium rate (Section 11.3). By "equilibrium", we mean the situation when every sender is sending at its target rate. When the propagation delays for flows sharing a bottleneck are similar and comparable to (or

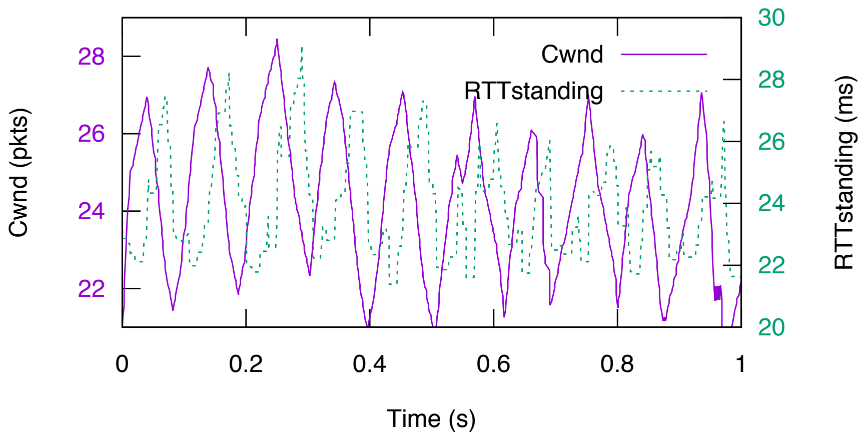


Figure 11-2: Congestion window and RTT as a function of time for a Copa flow running on a 12 Mbit/s Mahimahi [112] emulated link. As predicted, the period of oscillation is $\approx 5\,\texttt{RTT}$ and amplitude is ≈ 5 packets. The emulator's scheduling policies cause irregularities in the RTT measurement, but Copa is immune to such irregularities because the `cwnd` evolution depends only on comparing RTTstanding to a threshold.

larger than) the queuing delay, the small oscillations synchronize to cause the queue length at the bottleneck to oscillate between having 0 and $2.5/\hat{\delta}$ packets every five RTTs. Here, $\hat{\delta} = (\sum_i 1/\delta_i)^{-1}$. The equilibrium queue length is $(0{+}2.5)\hat{\delta}^{-1}/2 = 1.25/\hat{\delta}$ packets. When each $\delta = 0.5$ (the default value), $1/\hat{\delta} = 2n$, where n is the number of flows.

We prove the above assertions about the steady state using a window analysis for a simplified deterministic (D/D/1) bottleneck queue model. In Section 11.3 we discuss Markovian (M/M/1 and M/D/1) queues. We assume that the link rate, μ, is constant (or changes slowly compared to the RTT), and that (for simplicity) the feedback delay is constant, RTTmin $\approx$ `RTT`. This means that the queue length inferred from an ACK at time t is $q(t) = w(t - \text{RTTmin}) - BDP$, where $w(t)$ is congestion window at time t and BDP is the bandwidth-delay product. Under the constant-delay assumption, the sending rate is `cwnd`/`RTT` = `cwnd`/RTTmin.

First consider just one Copa sender. We show that Copa remains in steady state oscillations shown in Figure 11-1, once it starts those oscillations. In steady state, $v = 1$ (v starts to double only after `cwnd` changes in the same direction for at least

3 RTTs. In steady state, direction changes once every 2.5 `RTT`. Hence $v = 1$ in steady state.). When the flow reaches “change point A”, its RTTstanding estimate corresponds to minimum in the $\frac{1}{2}$srtt window of latest ACKs. Latest ACKs correspond to packets sent 1 RTT ago. At equilibrium, when the target rate, $\lambda_t = 1/(\delta d_q)$, equals the actual rate, `cwnd`$/RTT$, there are $1/\delta$ packets in the queue. When the queue length crosses this threshold of $1/\delta$ packets, the target rate becomes smaller than the current rate. Hence the sender begins to decrease `cwnd`. By the time the flow reaches “change point B”, the queue length has dropped to 0 packets, since `cwnd` decreases by $1/\delta$ packets per RTT, and it takes 1 RTT for the sender to know that queue length has dropped below target. At “change point B”, the rate begins to increase again, continuing the cycle. The resulting mean queue length of the cycle, $1.25/\delta$, is a little higher than $1/\delta$ because RTTstanding takes an extra srtt/2 to reach the threshold at “change point A”.

When N senders each with a different δ_i share the bottleneck link, they synchronize with respect to the common delay signal. When they all have the same propagation delay, their target rates cross their actual rates at the same time, irrespective of their δ_i. Hence they increase/decrease their `cwnd` together, behaving as one sender with $\delta = \hat{\delta} = (\sum_i 1/\delta_i)^{-1}$.

11.3 Justification of the Copa target rate

This section explains the rationale for the target rate used in Copa. We model packet arrivals at a bottleneck not as deterministic arrivals as in the previous section, but as Poisson arrivals. This is a simplifying assumption, but one that is more realistic than deterministic arrivals when there are multiple flows. The key property of random packet arrivals (such as with a Poisson distribution) is that queues build up even when the bottleneck link is not fully utilized.

In general traffic may be burstier than predicted by Poisson arrivals [119] because flows and packet transmissions can be correlated with each other. In this case, Copa over-estimates network load and responds by implicitly valuing delay more. This behavior is reasonable as increased risk of higher delay is being met by more caution. Ultimately, our validation of the Copa algorithm is through experiments, but the modeling assumption provides a sound basis for setting a good target rate.

11.3.1 Objective function and Nash equilibrium

Consider the objective function for sender (flow) i combining both throughput and delay:

$$U_i = \log\ \lambda_i - \delta_i\ \log\ d_s, \tag{11.3}$$

where $d_s = d_q + 1/\mu$ is the "switch delay" (total minus propagation delay). The use of switch delay is for technical ease; it is nearly equal to the queuing delay.

Suppose each sender attempts to maximize its own objective function. In this model, the system will be at a Nash equilibrium when no sender can increase its objective function by unilaterally changing its rate. The Nash equilibrium is the n-tuple of sending rates $(\lambda_1, \ldots, \lambda_n)$ satisfying

$$U_i(\lambda_1, \ldots, \lambda_i, \ldots, \lambda_n) > U_i(\lambda_1, \ldots, \lambda_{i-1}, x, \lambda_{i+1}, \ldots, \lambda_n) \tag{11.4}$$

for all senders i and any non-negative x.

We assume a first-order approximation of Markovian packet arrivals. The service process of the bottleneck may be random (due to cross traffic, or time-varying link rates), or deterministic (fixed-rate links, no cross traffic). As a reasonable first-order model of the random service process at the bottleneck link, we assume a Markovian service distribution and use that model to develop the Copa update rule. Assuming a deterministic service process gives similar results, offset by a factor of 2. In principle, senders could send their data not at a certain mean rate but in Markovian fashion, which would make our modeling assumption match practice. In practice, we don't, because: (1) there is natural jitter in transmissions from endpoints anyway, (2) deliberate jitter unnecessarily increases delay when there are a small number of senders and, (3) Copa's behavior is not sensitive to the assumption.

We prove the following proposition about the existence of a Nash equilibrium for Markovian packet transmissions. We then use the properties of this equilibrium to derive the Copa target rate of Eq. (11.1). The reason we are interested in the equilibrium property is that the rate-update rule is intended to optimize each sender's utility independently; we derive it directly from this theoretical rate at the Nash equilibrium. It is important to note that this model is being used not because it is precise, but because it is a simple and tractable approximation of reality. Our goal is to derive a principled target rate that arises as a stable point of the model, and use that to guide the rate update rule.

Theorem 2. *Consider a network with n flows, with flow i sending packets with rate λ_i such that the arrival at the bottleneck queue is Markovian. Then, if flow i has the objective function defined by Eq. (11.3), and the bottleneck is an M/M/1 queue, a unique Nash equilibrium exists. Further, at this equilibrium, for every sender i,*

$$\lambda_i = \frac{\mu}{\delta_i(\hat{\delta}^{-1}+1)} \tag{11.5}$$

where $\hat{\delta} = \left(\sum 1/\delta_i\right)^{-1}$.

Proof. Denote the total arrival rate in the queue, $\sum_j \lambda_j$, by λ. For an M/M/1 queue, the sum of the average wait time in the queue and the link is $\frac{1}{\mu-\lambda}$. Substituting this expression into Eq. (11.3) and separating out the λ_i term, we get

$$U_i = \log\ \lambda_i + \delta_i\ \log(\mu - \lambda_i - \sum_{j\neq i}\lambda_j). \tag{11.6}$$

Setting the partial derivative $\frac{\partial U_i}{\partial \lambda_i}$ to 0 for each i yields

$$\delta_i\lambda_i + \sum_j \lambda_j = \mu$$

The second derivative, $-1/\lambda_i^2 - \delta_i/(\mu-\lambda)^2$, is negative.

Hence Eq. (11.4) is satisfied if, and only if, $\forall i$, $\frac{\partial U_i}{\partial \lambda_i} = 0$. We obtain the following set of n equations, one for each sender i:

$$\lambda_i(1+\delta_i) + \sum_{j \neq i} \lambda_j = \mu.$$

The unique solution to this family of linear equations is

$$\lambda_i = \frac{\mu}{\delta_i(\hat{\delta}^{-1}+1)},$$

which is the desired equilibrium rate of sender i. □

When the service process is assumed to be deterministic, we can model the network as an M/D/1 queue. The expected wait time in the queue is $1/(2(\mu-\lambda)) - \mu/2 \approx 1/2(\mu-\lambda)$. An analysis similar to above gives the equilibrium rate of sender i to be $\lambda_i = 2\mu/(\delta_i(2\hat{\delta}^{-1}+1))$, which is the same as the M/M/1 case when each δ_i is halved. Since there is less uncertainty, senders can achieve higher rates for the same delay.

11.3.2 The Copa update rule follows from the equilibrium rate

At equilibrium, the inter-send time between packets is

$$\tau_i = \frac{1}{\lambda_i} = \frac{\delta_i(\hat{\delta}^{-1}+1)}{\mu}.$$

Each sender does not, however, need to know how many other senders there are, nor what their δ_i preferences are, thanks to the aggregate behavior of Markovian arrivals. The term inside the parentheses in the equation above is a proxy for the "effective" number of other senders, or equivalently the network load, and can be calculated differently.

As noted earlier, the average switch delay for an M/M/1 queue is $d_s = \frac{1}{\mu-\lambda}$. Substituting Eq, (11.8) for λ in this equation, we find that, at equilibrium,

$$\tau_i = \delta_i \cdot d_s = \delta_i(d_q + 1/\mu), \tag{11.7}$$

where d_s is the switch delay (as defined earlier) and d_q is the average queuing delay in the network.

This calculation is the basis and inspiration for the target rate. The *does not* model the dynamics of Copa, where sender rates change with time. The purpose of this analysis is to determine a good target rate for senders to aim for. Nevertheless, using steady state formulae for expected queue delay is acceptable since the rates change slowly in steady state.

11.3.3 Properties of the equilibrium

We now make some remarks about this equilibrium. First, by adding Eq. (11.5) over all i, we find that the resulting aggregate rate of all senders is

$$\lambda = \sum \lambda_j = \mu/(1+\hat{\delta}) \tag{11.8}$$

This also means that the equilibrium queuing delay is $1 + 1/\hat{\delta}$. If $\delta_i = 0.5$, the number of enqueued packets with n flows is $2n + 1$.

Second, it is interesting to interpret Eqs. (11.5) and (11.8) in the important special case when the δ_is are all the same δ. Then, $\lambda_i = \mu/(\delta + n)$, which is equivalent to dividing the capacity between n senders and δ (which may be non-integral) "pseudo-senders". δ is the "gap" from fully loading the bottleneck link to allow the average packet delay to not blow up to ∞. The portion of capacity allocated to "pseudo-senders" is unused and determines the average queue length which the senders can adjust by choosing any $\delta \in (0, \infty)$. The aggregate rate in this case is $n \cdot \lambda_i = \frac{n\mu}{\delta+n}$. When δ_is are unequal, bandwidth is allocated in inverse proportion to δ_i. The Copa rate update rules are such that a sender with constant parameter δ is equivalent to k senders with a constant parameter $k\delta$ in steady state.

Third, we recommend a default value of $\delta_i = 0.5$. While we want low delay, we also want high throughput; i.e., we want the largest δ that also achieves high throughput. A value of 1 causes one packet in the queue on average at equilibrium (i.e., when the sender transmits at the target equilibrium rate). While acceptable in theory, jitter causes packets to be imperfectly paced in practice, causing frequently empty queues and wasted transmission slots when a only single flow occupies a bottleneck, a common occurrence in our experience. Hence we choose $\delta = 1/2$, providing headroom for packet pacing. Note that, as per the above equation modeled on an M/M/1 queue, the link would be severely underutilized when there are a small number (≤ 5) of senders. But with very few senders, arrivals at the queue aren't Poisson and stochastic variations don't cause the queue length to rise. Hence link utilization is nearly 100% before queues grow as demonstrated in §11.4.1.

Fourth, the definition of the equilibrium point is consistent with our update rule in the sense that every sender's transmission rate equals their target rate if (and only if) the system is at the Nash equilibrium. This analysis presents a mechanism to determine the behavior of a cooperating sender: every sender observes a common delay d_s and calculates a common δd_s (if all senders have the same δ) or its $\delta_i d_s$. Those transmitting faster than the reciprocal of this value must reduce their rate and those transmitting slower must increase it. If every sender behaves thus, they will all benefit.

11.4 Evaluation

To evaluate Copa and compare it with other congestion-control protocols, we use a user-space implementation and ns-2 simulations. We run the user-space implementation over both emulated and real links.

Implementations:. We compare the performance of our user-space implementation of Copa with Linux kernel implementations of TCP Cubic, Vegas, Reno, and BBR [31], and user-space implementations of Remy, PCC [38], PCC-Vivace [39], Sprout [145], and Verus [152]. For Remy, we developed a user-space implementation and verified that its results matched the Remy simulator. There are many available RemyCCs. When we found a RemyCC that was appropriate for that network, we reported its results. We use Linux qdiscs and Mahimahi [112] to create emulated links. Our PCC results are for the default loss-based objective function. Pantheon [150], an

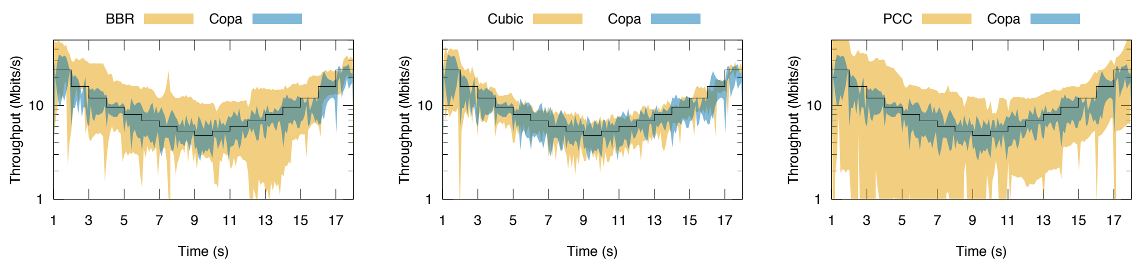


Figure 11-3: Mean ± std. deviation of rates of 10 flows as they enter and leave an emulated network once a second. The black line indicates the ideal fair allocation. Graphs for BBR, Cubic, and PCC are shown alongside Copa in each figure for comparison. Copa and Cubic flows follow the ideal allocation closely.

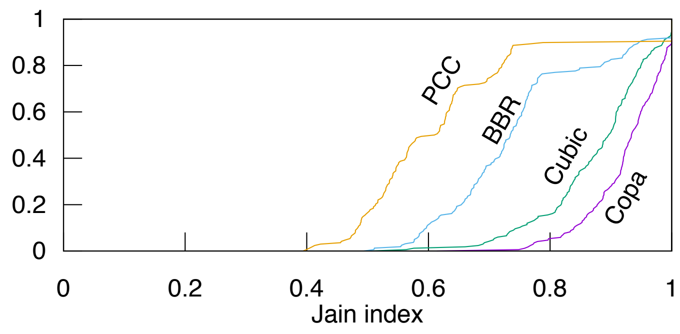


Figure 11-4: A CDF of the Jain indices (higher the better) obtained at various time slots for the dynamic behavior experiment (§11.4.1). Copa achieves the highest median Jain fairness index of 0.93 while Cubic, BBR and PCC achieve median indices of 0.90, 0.73 and 0.60 respectively.

independent test-bed for congestion control, uses the delay-based objective function for PCC.

ns-2 simulations:. We compare Copa with Cubic [61], NewReno [65], and Vegas [27], which are end-to-end protocols, and with Cubic-over-CoDel [113] and DCTCP [6], which use in-network mechanisms.

11.4.1 Dynamic behavior over emulated links

To understand how Copa behaves as flows arrive and leave, we set up a 46 MBit/s link with 20 ms RTT and 1 BDP buffer using Mahimahi. One flow arrives every second for the first ten seconds, and one leaves every second for the next ten seconds. The mean ± standard deviation of the bandwidths obtained by the flows at each time slot are shown in Figure 11-3. A CDF of the Jain fairness index in various time slots is shown in Figure 11-4.

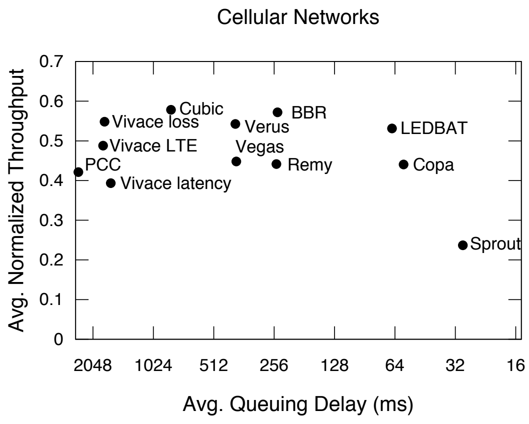
Copa obtains the highest median Jain fairness index, followed closely by Cubic. They both track the ideal rate allocation closely. BBR and PCC respond much more slowly to changing network conditions and fail to properly allocate bandwidth. In experiments where the network changed more slowly, they eventually succeeded in converging to the fair allocation, but this took tens of seconds.

environments. Copa's mode switcher correctly functioned most of the time, detecting that no buffer-filling algorithms were active in this period. There were some erroneous switches to competitive mode for a few RTTs. This happens because when flows arrive or depart, they disturb Copa's steady-state operation. Hence it is possible that for a few RTTs the queue is never empty and Copa flows can switch from default to competitive mode. In this experiment, there were a few RTTs during which several flows switched to competitive mode, and their δ decreased. However, queues empty every five RTTs in this mode as well if no competing buffer-filling flow is present. This property enabled Copa to correctly revert to default mode after a few RTTs.

11.4.2 Real-world evaluation

To understand how Copa performs over wide-area internet paths with real cross traffic and packet schedulers, we submitted our user-space implementation of Copa to Pantheon [150], a system developed to evaluate congestion control schemes. During our evaluation period, Pantheon had nodes in six countries. Each experiment creates flows on a particular day using each congestion control scheme between a node and an AWS server nearest it, and measures the throughput and delay. We separate the set of experiments into two categories, depending on how the node connects to the internet (Ethernet or cellular).

To obtain an aggregate view of performance across the dozens of runs, we plot the average normalized throughput and average queuing delay. Throughput is normalized relative to the run that obtained the highest throughput among all runs in an experiment to obtain a number between 0 and 1. Pantheon reports the one-way



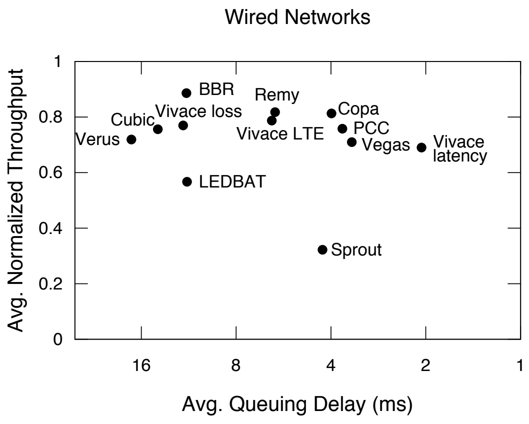


Figure 11-5: Real-world experiments on Pantheon paths: Average normalized throughput vs. queuing delay achieved by various congestion control algorithms under two different types of internet connections. Each type is averaged over several runs over 6 internet paths. Note the very different axis ranges in the two graphs. The x-axis is flipped and in log scale. Copa achieves consistently low queuing delay and high throughput in both types of networks. Note that schemes such as Sprout, Verus, and Vivace LTE are designed specifically for cellular networks. Other schemes that do well in one type of network don't do well on the other type. On wired Ethernet paths, Copa's delays are 10× lower than BBR and Cubic, with only a modest mean throughput reduction.

delay for every packet in publicly-accessible logs calculated with NTP-synchronized clocks at the two end hosts. To avoid being confounded by the systematic additive delay inherent in NTP, we report the queuing delay, calculated as the difference between the delay and the minimum delay seen for that flow. Each experiment lasts 30 seconds. Half of the experiments have one flow lasting 30 s. The other half have three flows starting at 0, 10, and 20 seconds from the start of the experiment. Note: we only consider experiments where the highest throughput achieved by any flow is < 120 Mbit/s, as our user-space program cannot measure delay at granularity finer than one Linux jiffy ($100\mu s$) currently; this corresponds to a target rate of 120 Mbit/s for Copa.

Copa's performance is consistent across different types of networks. It achieves significantly lower queuing delays than most other schemes, with only a small throughput reduction. Copa's low delay, loss insensitivity, RTT fairness, resistance to buffer-bloat, and fast convergence enable resilience in a wide variety of network settings. Vivace LTE and Vivace latency achieved excessive delays in a link between AWS S$\tilde{a}$o Paulo

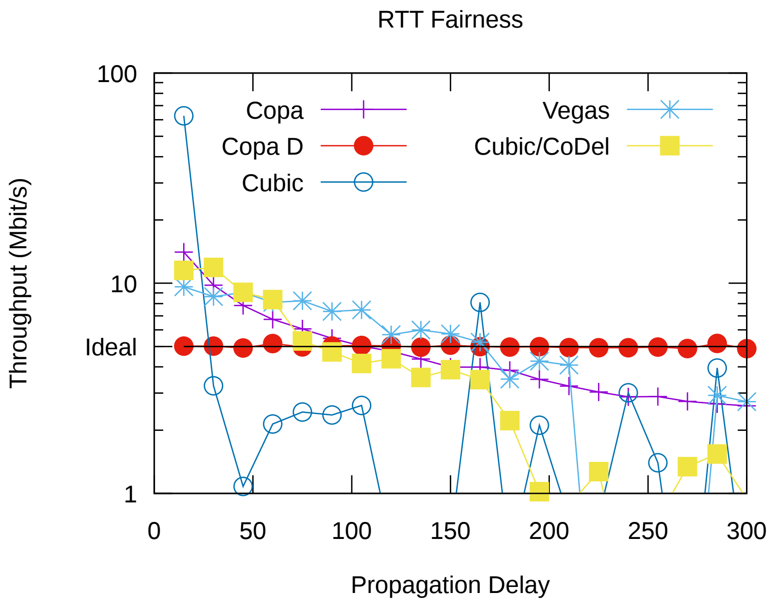


Figure 11-6: RTT-fairness of various schemes. Throughput of 20 long-running flows sharing a 100 Mbit/s simulated bottleneck link versus their respective propagation delays. "Copa D" is Copa in the default mode without mode switching. "Copa" is the full algorithm. "Ideal" shows the fair allocation, which "Copa D" matches. Notice the log scale on the y-axis. Schemes other than Copa allocate little bandwidth to flows with large RTTs.

and a node in Columbia, sometimes over 10 seconds. When all runs with > 2000 ms are removed for Vivace latency and LTE, they obtain average queuing delays of 156 ms and 240 ms respectively, still significantly higher than Copa. The Remy method used was trained for a $100\times$ range of link rates. PCC uses its delay-based objective function.

11.4.3 RTT-fairness

Flows sharing the same bottleneck link often have different propagation delays. Ideally, they should get identical throughput, but many algorithms exhibit significant RTT unfairness, disadvantaging flows with larger RTTs. To evaluate the RTT fairness of various algorithms, we set up 20 long-running flows in ns-2 with propagation delays evenly spaced between 15 ms and 300 ms. The link has a bandwidth of 100 Mbit/s and 1 BDP of buffer (calculated with 300 ms delay). The experiment runs for

100 seconds. We plot the throughput obtained by each of the flows in Figure 11-6.

Copa's property that the queue is nearly empty once in every five RTTs is violated when such a diversity of propagation delays is present. Hence Copa's mode switching algorithm erroneously shifts to competitive mode, causing Copa with mode switching (labeled "Copa" in the figure) to inherit AIMD's RTT unfriendliness. However, because the AIMD is on $1/\delta$ while the underlying delay-sensitive algorithm robustly grabs or relinquishes bandwidth to make the allocation proportional to $1/\delta$, Copa's RTT-unfriendliness is much milder than in the other schemes.

We also run Copa after turning off the mode-switching and running it in the default mode ($\delta = 0.5$), denoted as "Copa D" in the figure. Because the senders share a common queuing delay and a common target rate, under identical conditions, they will make identical decisions to increase/decrease their rate, but with a time shift. This approach removes any RTT bias, as shown by "Copa D".

In principle, Cubic has a window evolution that is RTT-independent, but in practice it exhibits significant RTT-unfairness because low-RTT Cubic senders are slow to relinquish bandwidth. The presence of the CoDel AQM improves the situation, but significant unfairness remains. Vegas is unfair because several flows have incorrect base RTT estimates as the queue rarely drains. Schemes other than Copa allocate nearly no bandwidth to long RTT flows (note the log scale), a problem that Copa solves.

11.4.4 Robustness to packet loss

To meet the expectations of loss-based congestion control schemes, lower layers of modern networks attempt to hide packet losses by implementing extensive reliability mechanisms. These often lead to excessively high and variable link-layer delays, as in many cellular networks. Loss is also sometimes blamed for the poor performance of congestion control schemes across trans-continental links (we have confirmed this with measurements, e.g., between AWS in Europe and non-AWS nodes in the US). Ideally, a 5% non-congestive packet loss rate should decrease the throughput by 5%, not by 5×. Since TCP requires smaller loss rates for larger window sizes, loss resilience

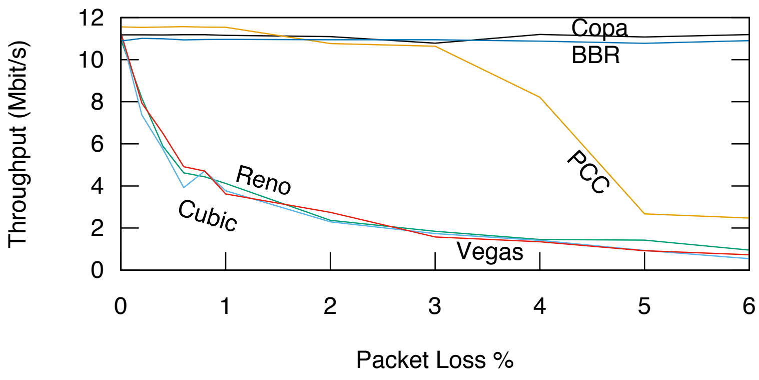


Figure 11-7: Performance of various schemes in the presence of stochastic packet loss over a 12 Mbit/s emulated link with a 50 ms RTT.

becomes more important as network bandwidth rises.

Copa in default mode does not use loss as a congestion signal and lost packets only impact Copa to the extent that they occupy wasted transmission slots in the congestion window. In the presence of high packet loss, Copa's mode switcher would switch to default mode as any competing traditional TCPs will back off. Hence Copa should be largely insensitive to stochastic loss, while still performing sound congestion control.

To test this hypothesis, we set up an emulated link with a rate of 12 Mbit/s bandwidth and an RTT of 50 ms. We vary the stochastic packet loss rate and plot the throughput obtained by various algorithms. Each flow runs for 60 seconds.

Figure 11-7 shows the results. Copa and BBR remain insensitive to loss throughout the range, validating our hypothesis. As predicted [103], NewReno, Cubic, and Vegas decline in throughput with increasing loss rate. PCC ignores loss rates up to $\approx 5\%$, and so maintains throughput until then, before falling off sharply as determined by its sigmoid loss function.

11.4.5 Simulated datacenter network

To test how widely beneficial the ideas in Copa might be, we consider datacenter networks, which have radically different properties than wide-area networks. Many congestion-control algorithms for datacenters exploit the fact that one entity owns

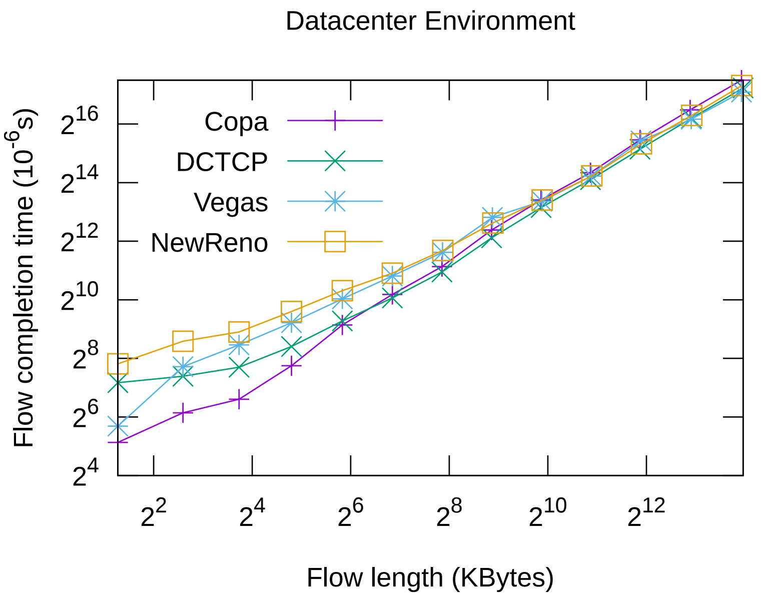


Figure 11-8: Flow completion times achieved by various schemes in a simulated datacenter environment. Note the log scale.

and controls the entire network, which makes it easier to incorporate in-network support [6, 8, 107, 120, 60].

Exploiting the datacenter's controlled environment, we make three small changes to the algorithm: (1) the propagation delay is externally provided, (2) since it is not necessary to compete with TCPs, we disable the mode switching and always operate at the default mode with $\delta = 0.5$ and, (3) since network jitter is absent, we use the latest `RTT` instead of RTTstanding, which also enables faster convergence. For computing v, the congestion window is considered to change in a given direction only if $> 2/3$ of ACKs cause motion in that direction.

We simulate 32 senders connected to a 40 Gbit/s bottleneck link via 10 Gbit/s links. The routers have 600 KBytes of buffer and each flow has a propagation delay of 12 μs. We use an on-off workload with flow lengths drawn from a web-search workload in the datacenter [6]. Off times are exponentially distributed with mean 200 ms. We compare Copa to DCTCP, Vegas, and NewReno.

The average flow completion times (FCT) are plotted against the length of the flow in Figure 11-8, with the y-axis shown on a log-scale. Because of its tendency to

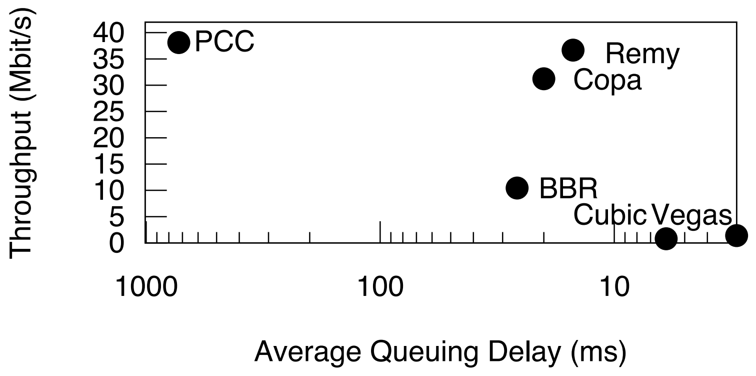


Figure 11-9: Throughput vs. delay plot for an emulated satellite link. Notice that algorithms that are not very loss sensitive (including PCC, which ignores small loss rates) all do well on throughput, but the delay-sensitive ones get substantially lower delay as well. Note the log-scale.

maintain short queues and robustly converge to equilibrium, Copa offers significant reduction in flow-completion time (FCT) for short flows. The FCT of Copa is up to a factor of 4 better for small flows under 64 KBytes compared to DCTCP. For longer flows, the benefits are modest, and in many cases other schemes perform a little better in the datacenter setting. This result suggests that Copa is a good solution for datacenter network workloads involving many short flows.

We also implemented TIMELY [106], but it did not perform well in this setting (over 7 times worse than Copa on average), possibly because TIMELY is targeted at getting high throughput and low delay for long flows. TIMELY requires several parameters to be set; we communicated with the developers and used their recommended parameters, but the difference between our workload and their RDMA experiments could explain the discrepancies; because we are not certain, we do not report those results in the graph.

11.4.6 Emulated satellite links

We evaluate Copa on an emulated satellite link using measurements from the WINDS satellite system [116], replicating an experiment from the PCC paper [38]. The link has a 42 Mbit/s capacity, 800 ms RTT, 1 BDP of buffer and 0.74% stochastic loss rate, on which we run one flow for 100 seconds. This link is challenging because it has a high bandwidth-delay product and some stochastic loss.

Figure 11-9 shows the total throughput v. delay plot for BBR, PCC, Remy, Cubic, Vegas, and Copa. Here we use a RemyCC trained for a RTT range of 30-280 ms for 2 senders with exponential on-off traffic of 1 second, each over a link speed of 33 Mbit/s, which was the best performer among the ones available in the Remy repository.

PCC obtained high throughput, but at the cost of high delay as it tends to fill the buffer. BBR ignores loss, but still underutilized the link as its rate oscillated between 0 and over 42 Mbit/s due to the high BDP, with these oscillations also causing high delays. Copa is insensitive to loss and scales to large BDPs due to its exponential rate update. Both Cubic and Vegas are sensitive to loss and hence lose throughput.

11.4.7 Co-existence with buffer-filling schemes

A major concern is whether current TCP algorithms will simply overwhelm the delay-sensitivity embedded in Copa. We ask: (1) how does Copa affects existing TCP flows?, and (2) do Copa flows get their fair share of bandwidth when competing with TCP (i.e., how well does mode-switching work)?

We experiment on several emulated networks. We randomly sample throughput between 1 and 50 MBit/s, RTT between 2 and 100 ms, buffer size between 0.5 and 5 BDP, and ran 1-4 Cubic senders and 1-4 senders of the congestion control algorithm being evaluated. The flows are run concurrently for 10 seconds. We report the average of the ratio of the throughput achieved by each flow to its ideal fair share for both the algorithm being tested and Cubic. To set a baseline for variations within Cubic, we also report numbers for Cubic, treating one set of Cubic flows as "different" from another.

Figure 11-10 shows the results. Even when competing with other Cubic flows, Cubic is unable to fully utilize the network. Copa takes this unused capacity to achieve greater throughput without hurting Cubic flows. In fact, Cubic flows competing with Copa get a higher throughput than when competing with other Cubic flows (by a statistically insignificant margin). Currently Copa in competitive mode performs

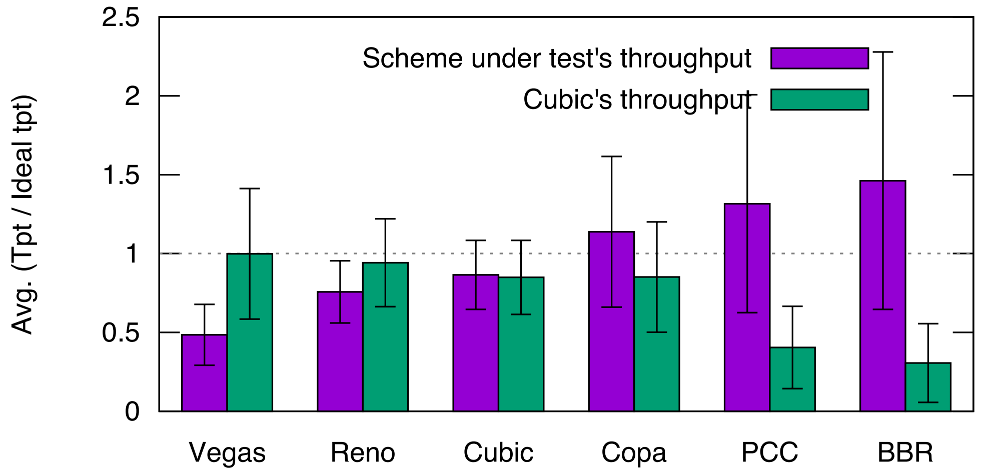


Figure 11-10: Different schemes co-existing on the same emulated bottleneck as Cubic. We plot the mean and standard deviation of the ratio of each flow's throughput to the ideal fair rate. Ideally the ratio should be 1. The mean is over several runs of randomly sampled networks. The left and right bars show the value for the scheme being tested and Cubic respectively. Copa is much fairer than BBR and PCC to Cubic. It also uses bandwidth that Cubic does not utilize to get higher throughput without hurting Cubic.

AIMD on $1/\delta$. Modifying this to more closely match Cubic's behavior will help reduce the standard deviation.

PCC gets a much higher share of throughput because its loss-based objective function ignores losses until about 5% and optimizes throughput. BBR gets higher throughput while significantly hurting competing Cubic flows.

Chapter 12

Future work

This section lays out a future research agenda that aims to make performance verification more powerful and easy to use. To that end, it proposes three thrusts.

12.1 Beyond verification: automatically synthesizing provably performant heuristics

Performance verification allows system designers to convert their intuitions into a mathematically precise wish list. They can then check their designs against this wish list. This dissertation demonstrates that the answer is often no, even for widely deployed heuristics. The natural question to ask here is "what next?". Future work develop methods to analyze, but also automatically design heuristics that are provably performant by construction.

Automated heuristic design has seen renewed interest thanks to advancements in deep learning methods [100, 101, 99, 147, 88, 149]. Attempts have been made to apply reinforcement learning to CCA design as well [150, 75, 76, 4, 144, 127]. However, the impact on practical applications has been minimal due to discrepancies between training environments and real-world scenarios. Our thesis is that non-deterministic models can help by enhancing the robustness of learned heuristics to real-world complexities.

There are two key challenges in automatically synthesizing heuristics.

12.1.1 Challenge: what signals do we monitor?

Identifying the right signals for heuristics to monitor is hard. For each popular heuristic type, a diverse array of signals has been developed over the years, leading to ambiguity about the "optimal" choice. For example, we discussed earlier that CCAs summarize delay measurements using averages, minimums, and maximums of RTT; maximums of rate; and repeating experiments. Likewise, different versions of the Linux kernel CPU scheduler and load balancer use different sets of signals [95, 49]. Decades of evolution have failed to identify the "correct" answer. Similarly, adaptive bit rate algorithms for video streaming employ various summaries of playback buffer size, queue length, and historical rate [69, 77, 151, 128].

As described in section 1.3, our ongoing research has shed light on this question. We developed a systematic method to derive a *sufficient* set of signals, given the (non-deterministic) model and objectives. The signals define the set of all actions the adversary can take in the future that are compatible with past observations given the model. We prove that if any heuristic can satisfy the objectives under our model, we can construct a heuristic whose actions are solely dependent on these signals. This insight significantly shrinks the heuristic search space, allowing us to utilize automated program synthesis methods to create novel and provably performant CCAs.

12.1.2 Challenge: how do we synthesize heuristics automatically?

Our ongoing work uses a popular program synthesis technique called Counter-Example Guided Inductive Synthesis (CEGIS). This has enabled us to automate some of the human drudgery of determining exactly how quickly to probe for more bandwidth and how we should decrease our rate to drain excess queues. However, since CEGIS does not scale to very large programs, and requires all programs and constraints to be expressed in an SMT-solver friendly manner, its use still involves considerable human labor.

Better techniques may be possible. Our observation about a sufficient set of signals allows us to frame congestion control as a fully observable two player game. This has two advantages. First, it allows us to synthesize CCAs one "move" at a time. For example, in CCA design, for every value that the set of signals can take, we can run a separate search procedure that outputs just a single number: the rate at which the CCA should transmit packets. In contrast, our previous approach has to design the entire function that maps signals to rate in a single step. Second, we can use new advances in the use of neural networks in playing games like Go and Chess [126] to solve heuristic design problems. The additional scalability may even allow us to synthesize heuristics more complex than congestion control.

12.2 Verifying the end-to-end performance of systems

This dissertation verified individual heuristics, exploiting the fact that many can be described in tens of lines of pseudo-code. However even when individual heuristics are doing the right things, the interaction of multiple heuristics can cause performance issues. For instance, meta-stable failures [68, 29] occur when heuristics in the system enter a bad feedback loop that causes poor performance that is independent of supplied load. This can cause an entire datacenter or groups of datacenters to suffer an outage. At a smaller scale, application load balancers can interact with network routing algorithms to cause persistent oscillations [94]. CCAs and algorithms for adapting video bit rate can interact poorly and benefit from coordinated design [48, 78]. Interactions between video frame retransmissions, CPU scheduling and video rate control can cause unacceptably high tail latency in real-time gaming applications [105].

A productive line of future work would develop new techniques that can scale to analyzing multiple heuristics. There are two approaches to increase the scale:

1. **Assume-guarantee reasoning:** A standard technique to verify the correct-

ness of large programs using automated reasoning methods is to break it down into sub-programs. A human guesses what *assumptions* and *guarantees* that each sub-program can make, and uses a computer to prove these guarantees. They then compose these lemmas together to prove a property about the entire system. The research question is to determine if the same techniques apply to verifying performance, treating the different heuristics as separate sub-programs. The key challenge is to guess what lemmas to prove about the heuristics. New frameworks need to be developed to help determine this.

2. **Non-rigorous fuzzing:** Fuzzing is an alternative to formal analyses that sacrifices rigor in favor of scale and reasoning about real code. Fuzzing can also leverage machine learning to guide it on where to look for performance properties. Our primary thesis is that worst case analysis can give valuable insight about system performance. Rigor, while desirable, is not necessary for that thesis to work.

12.3 Making performance verification easier to use

To maximize the impact of performance verification by making it a commonly used tool, ease of use is critical. There are two directions that could help achieve that goal:

1. Develop a set of high-level abstractions that make it easy to specify models of the environments and heuristics. The key challenge is to compile this high-level description to *efficient* SMT encoding. This dissertation used hand-engineered constraints because higher level languages would produce constraints that are much harder for SMT solvers.

2. Relaxing rigor can allow us to use fuzzing or machine learning based methods. These are easier to use since they can take arbitrary programs as models for the environments and heuristics. The challenge is in the lack of rigor. We need to develop methods that are good at finding *most* practically relevant bad behaviors, even though they do not exhaustively search through all of them.

3. Since performance verification is a new area of exploration, it would help to develop educational materials explaining the process.

Chapter 13

Conclusion

This dissertation argues that performance verification is a new analytical tool to understand the performance properties of real-world networked systems. It provides an alternative to queuing and control theory, which are typically too optimistic about performance of their because limited capacity to accurately model real-world phenomena. Overly optimistic analysis can lead to heuristic designs that fail in unexpected ways upon deployment.

Using performance verifications, we have shown three types of results in congestion control. First, we discovered previously deficiencies in widely deployed CCAs. Next, we developed new CCAs that are provably robust. Lastly, we proved an impossibility result that all CCAs that follow a widely used design pattern suffer from starvation. In subsequent work [56], we have shown that performance verification generalize to other heuristics beyond congestion control.

We hope that performance verification will become an integral part of the workflow in designing heuristics. This could remove one source of unreliability in future networked systems: heuristics making poor decisions that causes failures in spite of the hardware and software doing exactly what their designers intended.

Appendix A

Appendix

A.1 CCAC extensions

In-network support. Some CCAs [79, 40, 59, 6] use in-network support. To model them in CCAC, it does not suffice to simply run the in-network algorithms treating the path-server as a network server that marks or sets information in packet headers. This is because the queue on the path-server represents all enqueued packets on the path, not just those at the bottleneck link. The non-composing model we used for analyzing Copa in §6.3 offers a solution, because it only models one network "box". Thus we can sandwich the non-composing path-server between two composing path-servers to emulate a link with many boxes where the non-composing path-server represents the bottleneck. Thus, an algorithm where only the bottleneck link is involved in providing feedback can be implemented on the non-composing path-server.

Receiver-driven CCA. To emulate such an algorithm where control decisions are made by the receiver and enforced at the sender, one can use two path-servers, one from the sender to the receiver and one from the receiver to the sender. Each side also needs a propagation delay (analogous to R_m), perhaps identical in each direction. Differences in propagation delay smaller than D can be captured by the non-deterministic path-servers.

Multiple flows. To study starvation, we extended CCAC [12] to handle multiple flows. This turned out to be fairly straightforward. CCAC tracks, among other things, the number of bytes that have arrived at its bottleneck $A(t)$, and the number of bytes that have been served from it $S(t)$. To extend it to multiple flows, we have to maintain separate functions, one for each flow. So we have $\sum_i A_i(t) = A(t)$ and $\sum_i S_i(t) = S(t)$.

At time t, when a total of $S(t)$ bytes have been served, we need to determine how many bytes have been served *per-flow*. Ideally, we'd like to emulate a FIFO queue. Let t' be the time at which $A(t') = S(t)$. Then we'd want $S_i(t) = A_i(t)$, because that is when those packets of that flow must have entered the queue. Doing this directly requires us to intersect two lines, the equation for which involves the multiplication of two SMT variables. As a general rule, SMT solvers tend to not be very good at solving for non-linear constraints. We found that this rule applies to CCAC as well.

To get around this, we used the same relaxation method proposed in the CCAC paper. CCAC discretizes time relatively coarsely. It then ensures that the set of behaviors admitted in the discrete model is a super-set of the behaviors admitted in the continuous model. This way, any theorems proved in the discrete model also hold in the continuous model. However the discrete model may contain behaviors that the continuous model does not. Hence one must be careful to not make the discrete set *too* large.

We found the following approach to strike a good balance between ease of SMT modelling and not deviating too far from the continuous model. We merely ensured that if the queuing delay at time t is $\mathrm{d}t$ per CCAC's definition, then $S_i(t) > A_i(t-\mathrm{d}t)$.

A.2 AIMD counterexample in detail

We discuss further the example in Section §6.2 where CCAC found a way to exploit AIMD to cause a burst of $2CD$ bytes. Figure A-1 shows a trace of the various quantities in the path model that induces such a burst at time $7R_m$.

The path-server begins by inflating the RTT to be $R_m + D$, which causes `cwnd` to increase to $C(R_m + D) + \beta = 4CR_m$ without encountering loss. This corresponds to $S(t)$ touching the lower bound in Figure A-1. When `cwnd` exceeds $4CR_m$, one packet

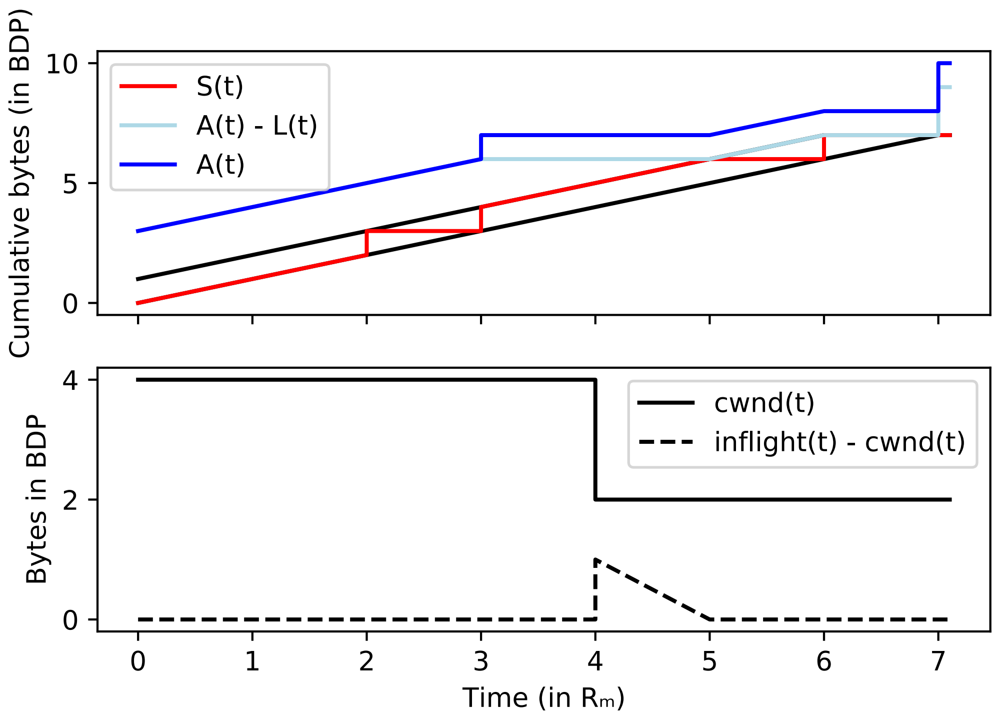


Figure A-1: A trace of how a burst of $2CD$ bytes can be orchestrated by combining the two mechanisms in §6.2

is dropped, at time $1R_m$. This packet was sent when the sender received an ACK $1R_m$ earlier at time 0. ACKs are sent smoothly by the path-server from time 0 to time $2R_m$. Hence the next $2CD$ bytes (from time $1R_m$ to $3R_m$ from the sender after the dropped packet will arrive smoothly as well. When the drop is detected at time $4R_m$, `cwnd` halves to $2CR_m$. At this point, $3CR_m$ bytes are in flight. At time R_m later (at time $5R_m$), only $2CR_m$ bytes are in flight, making the sender ready to burst again. When packets after time $5R_m$ are ACKed at time $7R_m$, the sender detects the CR_m lost packets and bursts that many. At time $6R_m$, the path-server combines this burst with another CR_m of ACKs, causing a total burst of $2CR_m$ at time $7R_m$, which overwhelms the buffer.

A.3 More SMT details

Recall that $L(t)$ denotes the total number of bytes lost by the network. The CCA cannot directly observe this. It only observes $L^d(t)$, which denotes the cumulative number of bytes that it *detected* as lost. In the continuous model, $L^d(t)$ is a time-shifted version of L where the time-shift depends on the gap between A and S, which itself is time-varying. As discussed in §5, when a quantity depends on the gap between two lines, discretization complicates the constraints. if a loss happened at time t (i.e. $L'(t) > 0$), the sender can detect it at time $t + \Delta t + R$ if $A(t) - L(t) + \mathit{dupacks} \leq S(t + \Delta t)$. Here, Δt is the time-varying component of the time-shift. This constraint ensures that the sender has received ACKs for at least *dupacks* number of bytes that were sent *after* the loss happened. Here, R is the propagation delay between when the server serves the packets (i.e. $S(t)$) to when the ACK reaches the sender.

Consider two points on the discrete $L(t)$ curve, L_t and L_{t+1}, where $L_{t+1} > L_t$ indicating that a loss occurred. The CCA can only detect this loss event after some delay. Now consider the corresponding points on the $L^d(t)$ curve. First, $L^d(t)$ can capture that loss event using one or more points, because of time variance in the gap between A and S. Second, any discrete point on the loss detection curve $L^d_{t+\Delta t}$ has to be bounded by $L_t \leq L^d_{t+\Delta t} \leq L_{t+1}$. In order to capture this relationship between the loss curve and loss detection curve in SMT, we introduce a variable $\mathit{detectable}_{t,\Delta t}$, which equals 1 if a loss event that happened at $t - \Delta t$ is detectable at time t, and it's zero otherwise. CCAs often need to know when losses are detected. L^d_t represents the cumulative number of losses *detected* up to time t. Many CCAs detect loss on the receipt of a set threshold of duplicate acknowledgements. This threshold is represented by an SMT variable *dupacks* that the solver is free to choose. Loss can be detected at time t only if that loss happened at time $\leq t - R - \Delta t$ and $S(t - R) \geq A(t - R - \Delta t) - L(t - R - \Delta t) + \mathit{dupacks}$. Hence we formally define $\mathit{detectable}_{t,\Delta t} = S_{t-R} \geq A_{t-R-\Delta t} - L_{t-R-\Delta t} + \mathit{dupacks}$ and add the constraints $\mathit{detectable}_{t,\Delta t} \rightarrow L^d_t \geq L_{t-R-\Delta t}$ and $\neg\mathit{detectable}_{t,\Delta t} \rightarrow L^d_t \leq L_{t-R-\Delta t}$ for each $\Delta t \in \{0, \cdots, t\}$.

A.4 Proofs

Here we prove some of the theorems referenced in the paper. For some of these theorems, we have written computer-checked proofs in Lean [36], a proof assistant similar to Coq [20], which can be found at `https://projects.csail.mit.edu/ccac`. For these theorems, we only give the theorem statement and the proof intuition in English here and leave the details to the Lean proof.

Theorem 5. *A path-server with parameters (C, D, β) can emulate a constant bit rate (CBR) server with link rate C and buffer size β followed by a delay box. The delay box can non-deterministically delay every byte by an arbitrary amount as long as it does not reorder bytes and no byte stays in the delay box for longer than D seconds.*

Proof. We need to show that, given a function $f(x)$ from byte ID (i.e. sequence number) to how long it stays in the delay box, we can produce a corresponding $W(t)$ that is compatible with the arrival, service and loss curves of the CBR+delay box (refer Table 4.1 for notation). The $W(t)$ in this case is simple: waste whenever allowed. That is, a token only enters the token queue if $T(t) < Q(t)$, because of which every token is paired on arrival with a byte; this is the byte it will be dequeued with. With this choice, we notice that a byte gets paired paired with a token at the same time that it would have gotten dequeued from the corresponding CBR server, since tokens arrive at C bytes/second. Note, $Q(t)$ can be greater than $T(t)$ if bytes arrive faster than tokens, which corresponds to those bytes being queued in the CBR server's buffer. When $Q(t) - T(t) > \beta$, both the path-server and the CBR server will drop packets.

Once a byte has been paired with a token, it has D seconds to get dequeued. The path-server can now choose when to dequeue each byte to match $f(x)$ which is always possible since $f(x) \leq D, \forall x$ and it does not cause reordering of bytes. This proves that our choice of $W(t)$ is compatible with the constraints of the generalized token bucket filter □

Composition theorems. To show that path-servers can compose, we show how to create a path-sever τ_s that can emulate all behaviors that are possible when path servers τ_1 and τ_2 are connected. In all these theorems, we assume the initial conditions are such that $A(t) = S(t) = W(t) = L(t) = 0$ when $t \leq 0$. This simplifies the proofs without loosing generality since the path-server can "evolve" to whatever state is needed.

Notation. Each path-server has its own $A(t)$, $S(t)$ etc. We use the dotted-notation to show this. E.g. τ_1's service curve is $\tau_1.A(t)$ and τ_s's waste curve is $\tau_s.W(t)$. Refer Table 4.1 for a glossary of symbols used. Further, the upper and lower bounds on $S(t)$ are denoted as $u(t)$ and $l(t)$ respectively. Hence $\tau_1.u(t) = \tau_1.C * t - \tau_1.W(t)$ and $\tau_1.l(t) = \tau_1.u(t-D) = \tau_1.C * (t - \tau_1.D) - \tau_1.W(t-D)$.

When two path-servers τ_1 and τ_2 are connected in series, the service curve of τ_1 equals the arrival curve of τ_2. We denote this as $\tau_1.S(t) = \tau_2.A(t)$. We wish to prove that a path-server with jitter parameter $\tau_1.D + \tau_2.D$ can emulate a superset of the things the composed version can emulate. To do so, given traces of any two path-servers τ_1 and τ_2 (a trace is a collection of all the functions and parameters such as C and $A(t)$), we need to produce a trace for a third that has exactly the same behavior as the composition. That is, $\tau_s.A(t) = \tau_1.A(t)$ and $\tau_s.S(t) = \tau_2.S(t)$. We split the proof of composition of the model into two parts. One where $\tau_1.C \leq \tau_2.C$ and another where $\tau_1.C \geq \tau_2.C$.

The following proofs will use the principle of mathematical induction on time, and hence treat time as an integer. However, unlike in Section §5, here a time-step can be arbitrarily small. Thus, for all practical purposes, time is continuous.

A.4.1 Case 1: Second path-server is faster

Before we prove the main theorem, we prove that when $\tau_1.C \leq \tau_2.C$ and $\tau_2.\beta \geq \tau_1.C\tau_1.D$, τ_2 can never lose packets no matter how bytes arrive or what non-deterministic choices each makes. This makes intuitive sense, since $\tau_2.\beta$ is bigger than the largest burst τ_1 can cause.

Theorem 6. *For every pair of traces τ_1, τ_2 that are placed in series (i.e. $\tau_2.A(t) = \tau_1.S(t)$), where $\tau_1.C \leq \tau_2.C$ and $\tau_2.\beta \geq \tau_1.C \cdot \tau_1.D$, the following holds: 1)* $\tau_2.L(t) = 0$

and 2) $\tau_1.l(t) \leq \tau_2.u(t)$

Proof. We only give the outline of the proof here since we wrote a computer-checked proof in Lean which are available at `https://projects.csail.mit.edu/ccac`.

The proof uses induction on time, where we prove both assertions in the theorem statement simultaneously. The intuitive argument is that if $\tau_1.l(t-1) \leq \tau_2.u(t-1)$ then $\tau_1.S(t) \leq \tau_1.l(t)$ cannot be more than $\tau_2.u(t) + \tau_1.C\tau_1.D \leq \tau_2.u(t) + \beta$. Thus τ_2's condition for loss can never be met. In proving this, we use the fact that the upper and lower bounds (i.e. $u(t)$ and $l(t)$) cannot increase faster than C bytes per timestep since $W(t)$ is a non-decreasing function.

Then we prove $\tau_1.l(t) \leq \tau_2.u(t)$ using the fact that $\tau_1.C \leq \tau_2.C$ and τ_2 is not allowed to waste when $\tau_1.S(t) = \tau_2.A(t)$ is greater than $\tau_2.u(t)$. This finishes the induction step. □

Theorem 7. *For every pair of traces* τ_1, τ_2 *where* $\tau_1.C \leq \tau_2.C$, $\tau_2.\beta \geq \tau_1.C \cdot \tau_1.D$ *and* $\tau_2.A(t) = \tau_1.S(t)$, *there exists a trace* τ_s *such that*

1. $\tau_s.C = \tau_1.C$
2. $\tau_s.D = \tau_1.D + \tau_2.D$
3. $\tau_s.\beta = \tau_1.\beta$
4. $\tau_s.A(t) = \tau_1.A(t)$
5. $\tau_s.S(t) = \tau_2.S(t)$
6. $\tau_s.L(t) = \tau_1.L(t)$

Proof. Again we only give the outline of the proof here since we wrote computer-checked proofs in Lean which are available at `https://projects.csail.mit.edu/ccac`. To produce τ_s, we need to pick a $\tau_s.W(t)$ that is compatible with τ_s's arrival, service and loss curves (i.e. satisfies all the constraints listed in section §4). Here, simply setting $\tau_s.W(t) = \tau_1.W(t)$ does the job.

Note, Theorem 6 implies $\tau_2.L(t) = 0$ and $\tau_1.l(t) \leq \tau_2.u(t)$. Showing that $\tau_s.S(t) \leq \tau_s.u(t)$ is straightforward since $\tau_s.S(t) = \tau_2.S(t) \leq \tau_1.S(t) \leq \tau_1.u(t) = \tau_s.u(t)$.

Proving $\tau_s.S(t) \geq \tau_s.l(t)$ requires induction on t, but is relatively straightforward. Finally, since $\tau_s.u(t) = \tau_1.u(t)$, their loss thresholds are identical. Hence τ_s can waste tokens and lose packets whenever τ_1 can. $\square$

A.4.2 Case 2: First path-server is faster

We only prove this theorem when buffers are infinitely large and hence there is no loss.

Theorem 8. *For every pair of traces τ_1, τ_2 where $\tau_1.C \geq \tau_2.C$, $\tau_2.A(t) = \tau_1.S(t)$, $\tau_1.\beta = \tau_2.\beta = \infty$ and $\tau_1.L(t) = \tau_2.L(t) = 0$, there exists a trace τ_s such that*

1. $\tau_s.C = \tau_2.C$
2. $\tau_s.D = \tau_1.D + \tau_2.D$
3. $\tau_s.A(t) = \tau_1.A(t)$
4. $\tau_s.S(t) = \tau_2.S(t)$
5. $\tau_s.\beta = \infty$
6. $\tau_s.L(t) = 0$

Proof. To show that such a τ_s exists, we need to construct a $\tau_s.W(t)$, since all other functions are already defined in terms of τ_1 and τ_2. Then we prove that it satisfies the constraints, namely 1) when $\tau_s.W(t)$ increases, waste is allowed and 2) τ_s's bounds on $S(t)$ contain the full range of τ_2's bounds. We construct it as follows.

We start with $\tau_2.W(0) = -\tau_2.C \cdot \tau_2.D$. We construct $\tau_s.W$ using the following algorithm. The algorithm has two states. It starts in state 1 in timestep 0 with $\tau_s.W(0) = 0$. Suppose we have decided the state and $\tau_s.W$ for time t, we decide these values for $t+1$ as follows.

1. State 1 [tracking]: If $\tau_1.l(t+1) \geq \tau_2.u(t+1)$, transition to state 2 in timestep $t+1$ and set $\tau_s.W(t+1) \leftarrow \tau_s.W(t)$. Else remain in state 1 and set $\tau_s.W(t+1) \leftarrow$ $max(\tau_s.W(t), \tau_1.W(t+1) - \Delta C * t$ where $\Delta C = \tau_1.C - \tau_s.C \geq 0$

2. State 2 [no-tracking]: If $\tau_s.u(t) + \tau_2.C \geq \tau_1.u(t+1)$, transition to state 1 in timestep $t+1$ and set $\tau_s.W(t+1) \leftarrow max(\tau_s.W(t), \tau_1.W(t+1) - \Delta C * t)$. Else remain in state 2 and set $\tau_s.W(t+1) \leftarrow \tau_s.W(t)$

Note, when we set $\tau_s.W(t+1) \leftarrow \tau_1.W(t+1) - \Delta C * t$, we are setting $\tau_s.W(t+1)$ such that $\tau_s.u(t+1) = \tau_1.u(t+1)$. $\tau_s.W$ is non-decreasing by construction.

We need to show that τ_s is allowed to waste whenever the algorithm above causes $\tau_s.W(t)$ to increase. The following claim establishes this

Claim 1: If $\tau_s.W(t) < \tau_s.W(t+1)$ then, $\tau_1.A(t+1) \leq \tau_s.u(t+1)$

Intuitively, $\tau_s.W(t)$ increases only when $\tau_s.u(t)$ is tracking $\tau_1.u(t)$, which only happens when $\tau_1.u(t)$'s slope is $< C$. Thus τ_1 must be wasting and hence $\tau_1.A(t) \leq \tau_1.u(t) = \tau_s.u(t)$. We now give the detailed argument.

The algorithm only changes $\tau_s.W$ when a) we remain in state 1 or b) when we transition to state 1.

Let's analyze a) first, where $\tau_s.W$ is updated in state 1. Here $\tau_s.W(t) = max(\tau_s.W(t-1), \tau_1.W(t) - \Delta C * t) \geq \tau_1.W(t) - \Delta C * t$ and $\tau_s.W(t+1) = \tau_1.W(t+1) - \Delta C * (t+1)$. Hence $\tau_s.W(t) < \tau_s.W(t+1) \Rightarrow \tau_1.W(t) < \tau_1.W(t+1) - \Delta C \leq \tau_1.W(t+1)$. Hence $\tau_1.W$ increases. But this implies that $\tau_1.A(t+1) \leq \tau_1.u(t+1)$ (recall, $\forall t, \tau_1.L(t) = 0$). But, $\tau_s.u(t+1) = \tau_s.C*(t+1) - \tau_s.W(t+1) = \tau_1.u(t+1)$. Hence $\tau_1.A(t+1) \leq \tau_s.u(t+1)$ which is what we wanted to show.

In case b) we first show $\tau_s.u(t) < \tau_1.u(t)$. We know that $\tau_s.u(t) \leq \tau_s.u(t-1) + \tau_2.C$ because $\tau_s.W$ is non-decreasing. This has to be $< \tau_1.u(t)$. If not, and we were in state 2 at $t-1$, we would have transitioned to state 1 for timestep t. If we were in state 1 at $t-1$ and transition to state 2 at t then we did not change $\tau_s.W(t+1)$, hence there is nothing to prove.

Now, it suffices to show that $\tau_1.W$ increased (i.e. $\tau_1.W(t+1) > \tau_1.W(t)$), since then $\tau_1.A(t+1) \leq \tau_1.u(t+1) = \tau_s.u(t+1)$. If $\tau_1.W$ did not increase, then $\tau_1.u(t+1) = \tau_1.u(t) + \tau_1.C \geq \tau_1.u(t) + \tau_2.C > \tau_s.u(t) + \tau_2.C$, which contradicts the condition for transitioning to state 1. Hence $\tau_1.W$ must have increased.

Next we need to argue that τ_s's bounds contains τ_2's bounds so that $\tau_s.S(t)$ can track $\tau_2.S(t)$. Note that when we are in state 1 (tracking), the bounds for τ_1 and τ_2 overlap and $\tau_s.u(t)$ tracks $\tau_1.u(t)$.

Since $\tau_s.D = \tau_1.D + \tau_2.D$, τ_s's bounds contain the bounds for both τ_1 and τ_2. When we are in state 2 (tracking), $\tau_s.W$ does not increase. Therefore $\tau_s.l(t)$ and $\tau_s.u(t)$ increases at the same rate as $\tau_2.l(t)$ and $\tau_2.u(t)$ respectively since $\tau_s.C = \tau_2.C$. Hence if τ_s's bounds contains τ_2's bounds in the beginning, it will continue to contain them.

Thus we have shown that the $\tau_s.W(t)$ generated by the algorithm satisfies the constraints.

□

Finally, we prove the composition theorem when buffers are infinite and there is no loss using the theorems above.

Theorem 9. *For every pair of traces τ_1, τ_2 that are placed in series (i.e. $\tau_2.A(t) = \tau_1.S(t)$), $\tau_1.\beta = \tau_2.\beta = \infty$, there exists a trace τ_s such that*

1. $\tau_s.C = \min(\tau_1.C, \tau_2.C)$
2. $\tau_s.D = \tau_1.D + \tau_2.D$
3. $\tau_s.A(t) = \tau_1.A(t)$
4. $\tau_s.S(t) = \tau_2.S(t)$
5. $\tau_s.L(t) = 0$

Proof. This follows immediately from Theorems 7 and 8 if we set the buffer size to be infinity in theorem 7. □

A.5 Proof of starvation of delay-convergent algorithms

We will now fill in the details of the proof sketch discussed in section 7. The theorem is restated here for convenience:

Theorem 3 For any deterministic, f-efficient, delay-convergent CCA $\mathcal{A}$, any propagation delay R_m, any throughput ratio $s \geq 1$, and any $D > 2\delta^{\max}$, there exists a network scenario with two flows (specified via an initial state and two per-flow trajectories of non-congestive delays), such that one flow gets a throughput x_1 and the other flow gets a throughput $x_2 \geq s \cdot x_1$.

Proof. Steps 1 and 2 of the proof in Section 7 are complete. That section omitted details from step 3, which we now fill.

Recall that in step 2, we created two *different* ideal links where the flows running by themselves will achieve throughputs that are more than a factor s different. The key is that the delay the flows experience both lie within a range of size $\delta^{\max} + \epsilon$. Now we will run both these flows on the *same* FIFO queue as in our model. We will pick the starting states for the two flows to be the same as the state after they have converged, that is their state at times T_1 and T_2 respectively. Next we pick the initial queue length and vary the non-deterministic per-flow delay such that the delay they experience is the same as they experienced in the one-flow case. Since the CCA is deterministic, their sending rates will be identical. It remains for us to show we can indeed recreate the same delay.

There are two cases based on how $\min(d_{\min}(C_1), d_{\min}(C_2))$ compares with $\delta^{\max} + \epsilon$.

Case 1: $\min(d_{\min}(C_1), d_{\min}(C_2)) > \delta^{\max} + \epsilon$ In this case we will run both flows on a common link with propagation delay R_m and bottleneck link rate $C_1 + C_2$. Let us calculate the queuing delay at a link with capacity $C_1 + C_2$ when packets from the two flows arrive at rates $\bar{r}_1(t)$ and $\bar{r}_2(t)$. The derivative of the delay experienced by the individual flow $i \in \{1, 2\}$, $\bar{d}'_i(t) = \frac{\mathrm{d}\bar{d}_i(t)}{\mathrm{d}t} = (\bar{r}_i(t) - C_i)/C_i$ because $\bar{d}_i(t) \geq d_{\min}(C_i) > 0$. Let $d^\star(t)$ be the delay experienced in the combined two-flow network. If $d^\star(t) > 0$, we have:

$$\frac{\mathrm{d}d^{\star}(t)}{\mathrm{d}t} = \frac{\bar{r}_1(t) + \bar{r}_2(t) - (C_1 + C_2)}{C_1 + C_2} \tag{A.1}$$

$$= \bar{d}'_1(t)\frac{C_1}{C_1 + C_2} + \bar{d}'_2(t)\frac{C_2}{C_1 + C_2} \tag{A.2}$$

Hence $\frac{\mathrm{d}d^{\star}(t)}{\mathrm{d}t}$ is a weighted average of $\bar{d}'_1(t)$ and $\bar{d}'_2(t)$ with weights C_1 and C_2 respectively. Since we are allowed to set the initial conditions, we set the initial queue length:

$$d^{\star}(0) = \bar{d}_1(0)\frac{C_1}{C_1 + C_2} + \bar{d}_2(0)\frac{C_2}{C_2 + C_2} - \delta^{\max} - \epsilon$$

Since we assumed $d_1(t), d_2(t) > \delta^{\max} + \epsilon$ in our case analysis, subtracting $\delta^{\max} + \epsilon$ still leaves us with $d^{\star}(0) > 0$ and equation A.1 applies. This continues to hold for all t, as $d^{\star}$ continues to follow A.1 by induction over t.[1] Hence:

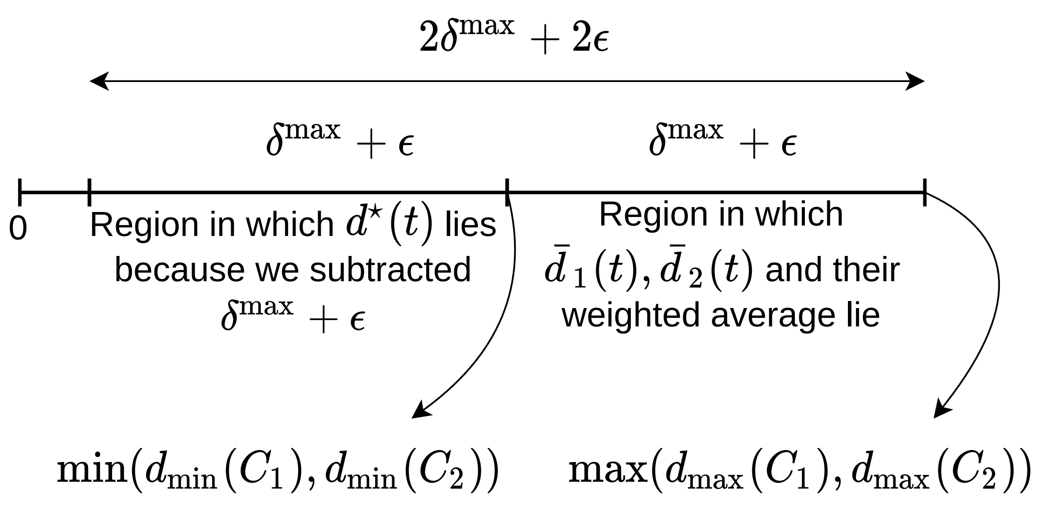
$$d^{\star}(t) = \bar{d}_1(t)\frac{C_1}{C_1 + C_2} + \bar{d}_2(t)\frac{C_2}{C_1 + C_2} - \delta^{\max} - \epsilon \tag{A.3}$$

Recall that to emulate delay, we needed:

$$0 \leq \bar{d}_i(t) - d^{\star}(t) \leq D = 2\delta^{\max} + 2\epsilon$$

for $i \in \{1, 2\}$. Note that both $\bar{d}_1$ and $d_2(t)$ are bound by a common region of size $\delta^{\max} + \epsilon$ and therefore so is their weighted average. The $-(\delta^{\max} + \epsilon)$ term in equation A.3 brings $d^{\star}(t)$ below the minimum value of both $\bar{d}_i(t)$. Hence $0 < \bar{d}_i(t) - d^{\star}(t)$. Further, $\bar{d}_i(t) - d^{\star}(t)$ $d^{\star}(t) \leq \bar{d}_i(t)$. The reason is illustrated in the diagram below:

[1] t is real and hence cannot support induction. However if we discretize time into infinitesimally small pieces, we can apply induction at each step.


$2\delta^{\max} + 2\epsilon$
$\delta^{\max} + \epsilon$
$\delta^{\max} + \epsilon$
0
Region in which $d^\star(t)$ lies because we subtracted $\delta^{\max} + \epsilon$
Region in which $\bar{d}_1(t), \bar{d}_2(t)$ and their weighted average lie
$\min(d_{\min}(C_1), d_{\min}(C_2))$
$\max(d_{\max}(C_1), d_{\max}(C_2))$

Thus we can emulate the two-flow network to make each flow think they are in ideal links of widely different capacities. Hence starvation will ensue.

Case 2: $\min(d_{\min}(C_1), d_{\min}(C_2)) \le \delta^{\max} + \epsilon$ This is the easy case. Since both the delays lie within an interval of size $\delta^{\max}+\epsilon$, we have that $\bar{d}_1(t), \bar{d}_2(t) \le 2\delta^{\max}+2\epsilon = D$. This means that if the queuing delay in the two-flow bottleneck were always 0, the non-deterministic delay element alone can emulate both the delays. Hence we simply pick a link rate that is large enough that $d^\star(t) \le \bar{d}_i(t)$ for $i \in \{1, 2\}$.

Note, in this case we can prove something stronger than starvation; the CCA isn't even f-efficient in our network model (though it is f-efficient in the ideal model). We can have a very large link rate, emulate $\bar{d}_1(t)$ entirely using non-congestive delay and induce the CCA to transmit at rate $\le C_1$. Since the link rate can be arbitrarily large, this causes arbitrarily bad underutilization. $\square$

A.6 Proof of the absolute upper bound

We re-state and prove the theorem discussed in Section 9.4.

THEOREM 4 Any deterministic, f-efficient, delay-bounding CCA will starve in the strong model for any value of the propagation delay R_m.

Proof. We will construct a sequence of single-flow network traces that will eventually let us construct a two-flow trace that causes starvation. We pick an arbitrary rate λ. Then, for our first trace, we run the CCA on an ideal link with rate λ and propagation delay R_m. Let the delay and sending rate in this case be $d_1(t)$ and $r_1(t)$ respectively. Let $D = \max_{t \in [0,\infty)} d_1(t)$ be the maximum delay experienced.[2]

Note that by varying the link rate, the adversary can create *any* queuing delay pattern it likes. This is because it can delay every packet by any amount it likes. Since it is a FIFO queue, it cannot reorder packets. Hence it cannot preferentially send packets of one flow over the other; both flows experience the same delay at the queue. Thus the theorem statement is not vacuously true.

We construct the next single-flow behavior by causing the queuing delay to be $d_2(t) = \max(0, d_1(t) - D)$. If the ratio of throughputs between the first and second case is more than s or less than $1/s$ infinitely many times (according to the theorem statement), we are done. We can run the two flows on the same FIFO queue where the link causes a queuing delay of $d_1(t) - D$. Then the non-deterministic delay element adds D seconds of delay to one flow's packets and 0 seconds of delay to another flow's packets. Since the flows see exactly the same delays as they say in the single-flow case, $d_1(t)$ and $d_2(t)$, they behave exactly the same way. Hence they achieve throughputs that are more than a factor s apart.

If not, we construct a third trace where the queuing delay is $\max(0, d_2(t) - D)$. If the throughputs of the second and third trace differ by a ratio of more than s, again we are done. Else we continue on. In at most $n = \lceil Q/D \rceil$ such steps, we would have either succeeded in causing starvation or reached $d_n(t) = 0$.

We claim that because the CCA is f-efficient, when $d_n(t) = 0$, the throughput should increase to infinity. That is, for all times t and rates λ', there exists a time $t' > t$ such that the total number of bytes transmitted is greater than $t'\lambda$. This is because, for a sufficiently large link rate, $d_n(t) = 0$. At this link rate, if the throughput achieved by the CCA is finite, the CCA can under-utilize by an arbitrary amount as the link rate increases. Here, by "finite" we mean that there exists a λ' such that for

[2]Strictly speaking, we should use the least upper bound, since the maximum may not exist.

all times t, the number of bytes transmitted till t is less than $\lambda' t$. This violates our f-efficiency definition.

Now if the throughput for the n^{th} trace reaches infinity, at some point in between the ratio of must have been greater than s. $\square$

Bibliography

[1] Linux networking documentation/segmentation offloads, 2021.

[2] The ns-2 simulator. `https://isi.edu/nsnam/ns/`, Accessed 2021.

[3] The ns-3 simulator. `https://nsnam.org/`, Accessed 2021.

[4] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *ACM SIGCOMM*, pages 632–647, 2020.

[5] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Automating network heuristic design and analysis. In *ACM SIGCOMM HotNets*, 2022.

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, pages 63–74, 2010.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *NSDI*, 2012.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[9] Thomas E Anderson, Andy Collins, Arvind Krishnamurthy, and John Zahorjan. PCP: Efficient Endpoint Congestion Control. In *NSDI*, 2006.

[10] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of Response Latency on User Behavior in Web Search. In *SIGIR*, 2014.

[11] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *ACM SIGCOMM*, 2022.

[12] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *ACM SIGCOMM*, 2021.

[13] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *USENIX NSDI*, 2018.

[14] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. Caching with delayed hits. 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 495–513, 2020.

[15] François Baccelli and Dohy Hong. Tcp is max-plus linear and what it tells us on its throughput. In *ACM SIGCOMM CCR*, pages 219–230, 2000.

[16] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *SIGCOMM*, 2013.

[17] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *INFOCOM*, 2001.

[18] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms. In *SIGCOMM*, 2001.

[19] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[20] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.

[21] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, pages 155–168, 2017.

[22] Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. An assertion language for debugging sdn applications. In *ACM HotSDN*, pages 91–96, 2014.

[23] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *ACM SIGCOMM*, pages 265–276, 2005.

[24] Enrico Bocchi, Luca De Cicco, and Dario Rossi. Measuring the Quality of Experience of Web Users. In *Proceedings of the 2016 Workshop on QoE-based Analysis and Management of Data Communication Networks*, 2016.

[25] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation.* John Wiley & Sons, 2018.

[26] R Braden. Requirements for internet hosts – communication layers. *IETF*, 1989. RFC 1122, Section 4.2.3.2.

[27] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *ACM SIGCOMM*, pages 24–35, 1994.

[28] Bob Briscoe. Flow Rate Fairness: Dismantling a Religion. *ACM SIGCOMM Computer Communication Review*, 37(2):63–74, 2007.

[29] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 221–227, 2021.

[30] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX ATC*, pages 213–218, 2013.

[31] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. In *ACM Queue*, pages 58–66, 2016.

[32] Neal Cardwell, Yuchung Cheng, S. Hassas Yeganeth, Ian Swett, and Van Jacobson. Bbr congestion control, version 1. IETF Internet Draft, 2017. Section 4.2.3.2.

[33] Neal Cardwell, Yuchung Cheng, S. Hassas Yeganeth, Ian Swett, and Van Jacobson. Bbr congestion control, version 2. IETF Internet Draft, 2021. Section 4.6.4.2.

[34] D-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. volume 17, pages 1–14, 1989.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[36] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[37] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[38] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *USENIX NSDI 2015*, pages 395–408, 2015.

[39] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *USENIX NSDI*, pages 343–356, 2018.

[40] Nandita Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly.* Citeseer, 2008.

[41] Jakob Eriksson, Hari Balakrishnan, and Samuel Madden. Cabernet: Vehicular Content Delivery using WiFi. In *MobiCom*, 2008.

[42] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

[43] Robert W Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993.

[44] S. Floyd. TCP and Explicit Congestion Notification. *SIGCOMM CCR*, 24(5), October 1994.

[45] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.

[46] S. Floyd, T. Henderson, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm, 2004. RFC 6582, IETF.

[47] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking*, 1(4), August 1993.

[48] 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, 2018.

[49] The Linux Foundation. Linux Load Balancer `sched/fair.c`. `https://elixir.bootlin.com/linux/v5.5/source/kernel/sched/fair.c`, 2020.

[50] Nitin Garg. Evaluating copa congestion control for improved video performance. `https://engineering.fb.com/2019/11/17/video-engineering/copa/`, 2019.

[51] Nitin Garg. Evaluating copa congestion control for improved video performance `https://engineering.fb.com/2019/11/17/video-engineering/copa/`. In *Facebook Engineering Blog*, 2019.

[52] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, pages 300–313, 2016.

[53] Daniel Genin and Jolene Splett. Where in the internet is congestion? *arXiv preprint arXiv:1307.3696*, 2013.

[54] Jim Gettys. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing*, 15(3):96–96, 2011.

[55] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate limiting youtube video streaming. In *USENIX ATC*, 2012.

[56] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. Quantitative verification of scheduling heuristics. In *In Submission*, 2023.

[57] Google. Perceptual Speed Index. `https://developers.google.com/web/tools/lighthouse/audits/speed-index`, December 2017.

[58] Google. Time to Interactive (TTI). `https://github.com/WPO-Foundation/webpagetest/blob/master/docs/Metrics/TimeToInteractive.md`, January 12, 2018.

[59] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Abc: A simple explicit congestion controller for wireless networks. In *USENIX NSDI*, pages 353–372, 2020.

[60] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI*, pages 1–14, 2015.

[61] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.

[62] Stephen Hemminger, Fabio Ludovici, and Hagen Pfeifer Paul. The linux netem network emulator. `https://www.linux.org/docs/man8/tc-netem.html`, 2011.

[63] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

[65] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996.

[66] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 357–370, 2011.

[67] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*, pages 781–786. Springer, 2012.

[68] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.

[69] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *SIGCOMM*, 2014.

[70] Facebook Inc. Mvfst: Facebook's quic implementation, commit e04fcaac. `https://github.com/facebookincubator/mvfst/commit/e04fcaacc1633c1bae78c61aac1f5f8a5784f657`, Accessed July 2021.

[71] Jana Iyengar and Martin Thomson. Quic: A udp-based multiplexed and secure transport. 2018.

[72] Jana Iyengar and Martin Thomson. Quic: A udp-based multiplexed and secure transport. *IETF*, 2021. RFC 9000.

[73] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[74] Nathan Jay, Tomer Gilad, Nogah Frankel, Tong Meng, Brighten Godfrey, Michael Schapira, Jae Won Chung, Vikram Siwach, and Jamal Hadi Salim. A pcc-vivace kernel module for congestion control, 2018.

[75] 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.

[76] Nathan Jay, Noga H Rotman, P Godfrey, Michael Schapira, and Aviv Tamar. Internet congestion control via deep reinforcement learning. *arXiv preprint arXiv:1810.03259*, 2018.

[77] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.

[78] Pantea Karimi. *Bridging the Gap Between Real-time Video and Backlogged Traffic Congestion Control*. PhD thesis, Massachusetts Institute of Technology, 2023.

[79] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *ACM SIGCOMM*, pages 89–102, 2002.

[80] F. P. Kelly, A.K. Maulloo, and D.K.H. Tan. Rate Control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[81] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. In-band network telemetry (int). `https://p4.org/assets/INT-current-spec.pdf`, 2016.

[82] Hwangnam Kim and Jennifer C Hou. Network calculus based simulation for tcp congestion control: Theorems, implementation and evaluation. In *IEEE INFOCOM*, volume 4, pages 2844–2855. IEEE, 2004.

[83] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[84] Leonard Kleinrock. Queuing systems, volume i: Theory, 1975.

[85] Leonard Kleinrock. Queueing system. *Volume II, J.*, 1976.

[86] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. Liquid resource types. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.

[87] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *ACM SIGCOMM*, pages 26–37, 2011.

[88] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[89] James Kurose and Keith Ross. Computer networks: A top down approach featuring the internet, 2010.

[90] Leslie Lamport. Introduction to tla. 1994.

[91] 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.

[92] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[93] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali Begen, and David Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE JSAC*, 32(4):719–733, 2014.

[94] Bingzhe Liu, Ali Kheradmand, Matthew Caesar, and P Brighten Godfrey. Towards verified self-driving infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 96–102, 2020.

[95] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[96] Ratul Mahajan, Jitendra Padhye, Sharad Agarwal, and Brian Zill. High Performance Vehicular Connectivity with Opportunistic Erasure Coding. In *USENIX ATC*, 2012.

[97] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[98] Jacob B Malone, Aviv Nevo, Jonathan W Williams, et al. The tragedy of the last mile: Congestion externalities in broadband networks. *No. June in NET Institute Working Paper*, (16-20), 2016.

[99] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems*, 32, 2019.

[100] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[101] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.

[102] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. Tcp selective acknowledgment options, 2012. RFC 1996, IETF.

[103] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. volume 27, pages 67–82. ACM, 1997.

[104] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *ACM SIGCOMM*, pages 615–631, 2020.

[105] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling high quality real-time communications with adaptive frame-rate. In *USENIX NSDI*, 2023.

[106] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[107] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 188–201. ACM, 2016.

[108] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *ACM SIGCOMM*, pages 30–43, 2018.

[109] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.

[110] Yuchung Cheng Neal Cardwell. The linux kernel, commit 43e122b0. `https://github.com/torvalds/linux/commit/43e122b014c955a33220fabbd09c4b5e4f422c3c`, 2015.

[111] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *NSDI*, 2018.

[112] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, pages 417–429, 2015.

[113] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[114] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[115] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[116] Hiroyasu Obata, Kazuya Tamehiro, and Kenji Ishida. Experimental evaluation of TCP-STAR for satellite Internet over WINDS. In *International Symposium on Autonomous Decentralized Systems (ISADS)*, 2011.

[117] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[118] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th international conference on high performance switching and routing (HPSR)*, pages 148–155. IEEE, 2013.

[119] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking (ToN)*, 3(3):226–244, 1995.

[120] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[121] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. *ACM HotNets 2022*, 2022.

[122] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[123] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, et al. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. 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 735–749, 2020.

[124] Matt Sargent, Jerry Chu, Dr Vern Paxson, and Mark Allman. Computing tcp's retransmission timer. Technical report, 2011. RFC 6289.

[125] Sea Shalunov, Greg Hazel, Janardhan Iyengar, Mirja Kuehlewind, et al. Low extra delay background transport (ledbat). In *RFC 6817*, 2012.

[126] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[127] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An Experimental Study of the Learnability of Congestion Control. In *SIGCOMM*, 2014.

[128] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM transactions on networking*, 28(4):1698–1711, 2020.

[129] R. Srikant. *The Mathematics of Internet Congestion Control*. Springer Science & Business Media, 2004.

[130] Aaron Stump, Clark W Barrett, and David L Dill. Cvc: A cooperating validity checker. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, pages 500–504. Springer, 2002.

[131] Bo Su, Xianliang Jiang, Guang Jin, and Haiming Chen. Rethinking the rate estimation of bbr congestion control. *Electronics Letters*, 56(5):239–241, 2020.

[132] Wei Sun, Lisong Xu, and Sebastian Elbaum. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In *ACM SIGSOFT*, pages 79–89, 2017.

[133] Wei Sun, Lisong Xu, and Sebastian Elbaum. Limitations of emulating realistic network environments for correctness testing of internet applications. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[134] Wei Sun, Lisong Xu, and Sebastian Elbaum. Scalably testing congestion control algorithms of real-world tcp implementations. In *IEEE International Conference on Communications (ICC)*, pages 1–7, 2018.

[135] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. Model-agnostic and efficient exploration of numerical state space of real-world tcp congestion control implementations. In *USENIX NSDI*, pages 719–734, 2019.

[136] Streaming Video Alliance: Measurement/Quality of Experience. `https://www.streamingvideoalliance.org/technical-work/working-groups/measurement-quality-of-experience/`.

[137] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[138] Mohit P Tahiliani, Vishal Misra, and KK Ramakrishnan. A principled look at the utility of feedback in congestion control. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1–5, 2019.

[139] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[140] The Google BBR Team. Bbr bandwidth based convergence. `https://github.com/google/bbr`, commit 87d8587c50, Documentation/bbr_bandwidth_based_convergence.pdf, 2018.

[141] Yue Wang, Kanglian Zhao, Wenfeng Li, Juan Fraire, Zhili Sun, and Yuan Fang. Performance evaluation of quic with bbr in satellite internet. In *IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*, pages 195–199. IEEE, 2018.

[142] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond jain's fairness index: Setting the bar for the deployment of congestion control algorithms. In *ACM SIGCOMM HotNets*, pages 17–24, 2019.

[143] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking*, 14(6):1246–1259, 2006.

[144] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *ACM SIGCOMM 2013*, 2013.

[145] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *USENIX NSDI*, pages 459–471, 2013.

[146] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *USENIX NSDI*, volume 11, pages 8–8, 2011.

[147] Zhengxu Xia, Yajie Zhou, Francis Y Yan, and Junchen Jiang. Genet: automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, 2022.

[148] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *INFOCOM*, 2004.

[149] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *USENIX NSDI*, pages 495–511, 2020.

[150] 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 *USENIX ATC*, pages 731–743, 2018.

[151] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[152] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM*, 2015.

[153] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. An axiomatic approach to congestion control. In *HotNets*, pages 115–121, 2017.

[154] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. Axiomatizing congestion control. *ACM POMACS*, 3(2):1–33, 2019.

[155] Lixia Zhang, Scott Shenker, and David D Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. 1991.

[156] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *CoNEXT*, pages 313–327, 2016.