Toward Formally Verifying Congestion Control Behavior

Venkat Arun*, Mina Tahmasbi Arashloo†, Ahmed Saeed*, Mohammad Alizadeh*, Hari Balakrishnan*
MIT CSAIL* and Cornell University†
Email: ccac@mit.edu Website: https://projects.csail.mit.edu/ccac

ABSTRACT

The diversity of paths on the Internet makes it difficult for designers and operators to confidently deploy new congestion control algorithms (CCAs) without extensive real-world experiments, but such capabilities are not available to most of the networking community. And even when they are available, understanding why a CCA under-performs by trawling through massive amounts of statistical data from network connections is challenging. The history of congestion control is replete with many examples of surprising and unanticipated behaviors unseen in simulation but observed on real-world paths. In this paper, we propose initial steps toward modeling and improving our confidence in a CCA's behavior. We have developed Congestion Control Anxiety Controller (CCAC),[1] a tool that uses formal verification to establish certain properties of CCAs. It is able to prove hypotheses about CCAs or generate counterexamples for invalid hypotheses. With CCAC, a designer can not only gain greater confidence prior to deployment to avoid unpleasant surprises, but can also use the counterexamples to iteratively improve their algorithm. We have modeled additive-increase/multiplicative-decrease (AIMD), Copa, and BBR with CCAC, and describe some surprising results from the exercise.

CCS CONCEPTS

• **Networks** → **Transport protocols**; • **Theory of computation** → *Automated reasoning*;

KEYWORDS

Congestion Control, Formal Verification, Transport Protocols

ACM Reference Format:
Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, Hari Balakrishnan. 2021. Toward Formally Verifying Congestion Control Behavior. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3452296.3472912

1 INTRODUCTION

Innovations in Internet congestion control algorithms (CCAs) are occurring at a rapid pace, spurred by evolving network technologies, a fast-changing application mix, and the rising importance of quality-of-experience for users, who react negatively to poor performance (e.g., by giving applications poor ratings or finding alternatives). Performance matters not only in the mean, but also in the tail statistics. In response, the research community and industry have developed numerous innovative methods to improve congestion control, because CCAs determine when packets are sent and determine transport performance [3, 5, 14, 18, 19, 36, 49, 50, 52, 54].

A key problem in CCA development is *evaluation*: how can developers, operators, and the networking community gain confidence in any given proposal? 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. 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.

The process of evaluating and gaining confidence with a CCA today involves some combination of simulation [1, 2], prototype implementation with tests on a modest number of emulated [13, 26, 39] and real-world paths [53, 54], 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 going 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 "hyperscaler" with access to such a live-testing infrastructure, yet has good ideas that deserve serious consideration.

In this paper, we propose initial steps to mitigate these issues. We have developed the Congestion Control Anxiety Controller (CCAC), pronounced "seek-ack" or "see-cack". CCAC uses formal verification to prove certain properties of CCAs. With CCAC, a user can (1) express a CCA in first-order logic, (2) specify hypotheses about the CCA for the tool to prove, and (3) test the hypothesis in the context of the expressed CCA running in a customizable, built-in path model. The user's ingenuity is useful in expressing the CCA and using CCAC to propose and iterate on useful hypotheses, while CCAC will prove the hypothesis correct or find insightful

[1] Because it helps control anxiety about whether a CCA will be robust in the field. Pronounced "seek-ack" or "see-cack".

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08.
https://doi.org/10.1145/3452296.3472912

counterexamples that disprove the hypothesis. A proof provides confidence that the CCA will perform well under *all* conditions consistent with the path model. On the other hand, counterexamples help in understanding corner cases mishandled by the CCA, and can help the user improve its design. Our work aims to provide an analytical tool that can capture unusual behaviors previously accessible only through empirical methods.

CCAC relies on a built-in path model that captures the complex sub-RTT behaviors observed on real network paths. The model does not attempt to characterize specific behaviors, but is instead constructed to capture the composition of a wide range of possibilities, including token bucket filters and ACK aggregation. Thus, a proof that a CCA satisfies a hypotheses with this model gives us confidence that it will satisfy the hypothesis on a substantial portion of real-world paths.

We have used CCAC to better understand classical AIMD, Copa, and BBR. Our findings include:

- Sub-RTT variability and bursts in packet or ACK delivery cause loss patterns and suboptimal performance in surprising ways for even the well-studied AIMD scheme. For example, CCAC found that it is possible to cause AIMD to transmit packets in a burst right after it detects loss. This burst can cause another loss, leading to severe under-utilization. CCAC also proved bounds on when such bad behavior can occur.
- For Copa, CCAC proved that when there is only one jittery network element on the path, utilization can drop to 50%, but no lower. With a *sequence* of jittery network elements, however, CCAC found that the worst-case throughput is near-zero!
- For BBR, CCAC found a network behavior that prevents BBR's bandwidth probe from discovering available bandwidth even when it is available. Analyzing the example by hand, we propose a fix that prevents CCAC from being able to force low utilization. Facebook has independently adopted this fix in their implementation of BBR in mvfst [27], their implementation of QUIC [28].

In this paper, we focus on the worst-case behavior of a single end-to-end congestion-controlled flow operating over complex network paths, without considering in-network feedback control. We do not consider inter-flow fairness propositions. We discuss these limitations and future work in Section 9.

2 MOTIVATION

To appreciate what a formal verification tool like CCAC can do to improve our understanding of CCAs, consider classical AIMD. Standard textbook analysis assumes that loss occurs only when the congestion window (*cwnd*) exceeds the bandwidth-delay product (BDP) plus the buffer size. In practice, 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.[2] These bursts can cause packet drops, under-utilizing the link, defeating the Wi-Fi mechanism.

CCAC not only models these issues, but it also identifies subtle behaviors. For instance, CCAC 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 this cannot happen because AIMD would decrease its *cwnd* by half. CCAC generated a counterexample that sidesteps this mitigation. In the counterexample, loss happens in two steps. When the second burst of losses are detected, the *cwnd* has already been halved once and will not be halved again since the losses belong to the same window of data [22]. Instead, the loss prompts a burst of packets from the sender, which triggers another loss. These bursts are bad because they can cause under-utilization. For example, 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. Although IETF RFCs have included mitigations for similar situations, CCAC sidesteps those as well. We discuss this counterexample in more detail in Section 7.

Prior mathematical methods of congestion control fail to characterize these behaviors because they ignore complex sub-RTT dynamics. We find that sub-RTT dynamics strongly influence the performance of a CCA, sometimes degrading utilization by an order of magnitude (see §6, §7 and, §8). CCAC fills this gap in our understanding of such behaviors. It uncovers surprising CCA behavior that the user or designers of the CCA may not have anticipated. In addition, it can help prove bounds on the performance of CCAs, including on steady-state behavior.

To provide the machinery to reason about sub-RTT jitter and packet/ACK dynamics, we need a framework that can capture a wide range of network behavior and a systematic way to express CCAs and hypotheses about them. Such a framework should be:

(1) *expressive*, capturing a wide range of networks, algorithms, and network stacks;
(2) *interpretable*, providing formal *proofs* to support its findings or counterexamples understandable by a human expert; and
(3) *iterative*, allowing the user to ask a wide range of queries about the behavior of a CCA while iterating on the algorithm and tuning/customizing network behavior.

Achieving these objectives, particularly proofs, requires the ability to examine an astronomically large number of network scenarios. Fortunately, the formal verification community has been developing automated tools for examining such large spaces for satisfiability queries. These tools have been applied for verifying the correctness of software and hardware. Automated theorem provers either prove that the software has the desired behavior or give a counterexample of an input that breaks the software [16]. In this paper, we introduce a *performance-as-correctness framework* that applies these ideas to congestion control.

3 OVERVIEW

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 [11, 33]. The path-server

[2] Burst transmissions can happen even with a paced sender, because implementations (including the Linux kernel [38]) often pace packets faster than the estimated rate to avoid under-utilization.

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

Figure 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.**

introduces a 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). The user can use this model to reason about variable capacity links as well (§4.4).

The power of this model is that, despite its simplicity, it captures a wide of range of underlying network-specific behaviors. Importantly, we do not specify any probability distribution over the path's behavior. Thus, the path-server is non-deterministic, but not random, restricting CCAC to worst-case analyses. This decision was a conscious one, made 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 [16] 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 §7.2 for an example).

Example. Figure 1 shows the CCAC query used to discover the behavior of AIMD discussed in the previous section. 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 in §2, which we elaborate on in §7.

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). Our model of TCP timeouts is different from the standard for the same reason (see §4.3). 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 [28], while TCP is limited to a maximum of four SACK blocks [34].

4 THE PATH MODEL

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 ACK aggregation, token-bucket filters, and arbitrary per-packet delay up to D seconds (see §4.2). 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. 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.

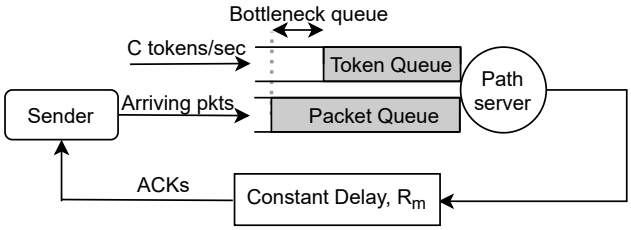

Figure 2: CCAC's path model.

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	$cwnd(t)$ – 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 1: Glossary of symbols.

4.1 Model Specification

Intuition. The path-server can be described as a generalized token-bucket filter (TBF) as shown in Figure 2. First, consider a regular 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 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 this scheme. When a token arrives at the token queue, the path-server can non-deterministically choose to either admit it or waste it. Wasting is allowed only when there are more tokens than the total number of bytes in the packet queue. 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 mechanism gives the path-server D seconds of slack to emulate various network effects. In addition, the D-second bound constrains the maximum number of tokens to $C \cdot D$.

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.[3] To emulate a bottleneck buffer of β bytes, the server drops packets when the bottleneck queue exceeds β bytes. The user can set β to a finite

[3]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.

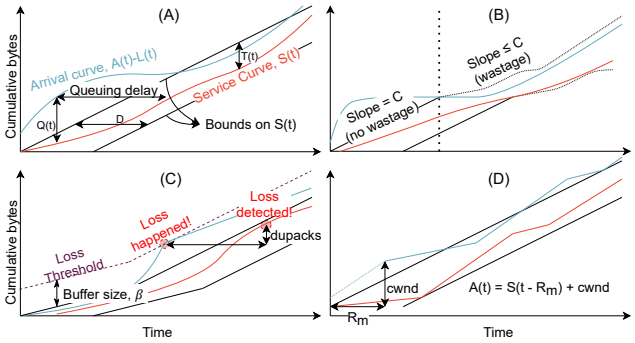

Figure 3: Graphical representation of the constraints.

value in units of BDP. The user can also set β to ∞ or let CCAC search over all its possible values to find one that satisfies the query.

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 1). A number of constraints bound the behavior of the network, and therefore, 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, $T(t) = Ct - W(t) - S(t)$.
- 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 introduce a standard graphical representation shown in Figure 3. The representation is akin to 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 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 3(B)). When wastage happens, the slope of the black curve is smaller than C, indicated by dotted lines in the figure. Figure 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 3(D) shows how a CCA that simply maintains a constant *cwnd* controls $A(t)$ in response to $S(t)$. Here, $A(t) = S(t - R_m) + cwnd$ because the ACKs leaving the server will reach the sender R_m seconds later, which maintains *cwnd* packets in flight.

4.2 The Set of Paths CCAC 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 ACK 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. 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 a constant-bit rate (CBR) link with rate C followed by an arbitrary delay box (see Theorem 3 in Appendix D). 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. Thus, the CBR + delay box can emulate a wide range of phenomena such as packetization, ACK aggregation, scheduling errors, MAC-layer jitter, and arbitrary delays ($\leq D$). The second example in Figure 4 shows the behavior of an ACK aggregator. Regardless of the sending rate of the arrival curve, the service curve aggregates packets (which is the effect of ACK aggregation) and sends them in bursts. Note that the CBR + delay box cannot emulate token bucket filters, 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 7 in Appendix D), 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 5 Appendix D). We do not yet have results for when $C_1 > C_2$ and buffers are finite.

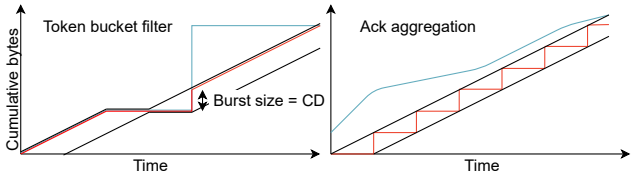

Figure 4: Examples of how the path-server emulates a token bucket filter and ACK aggregation.

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 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 5 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 4 in Appendix D). 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 (again, 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.

4.3 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 1 shows how to implement AIMD in CCAC.

Timeouts. If we implement timeouts according to RFC 6289 [40], 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.

4.4 Discussion

Variable link rates. CCAC uses 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 the CCAC model. To model a variable C, the user uses CCAC to prove lemmas about a CCA's behavior over a fixed link rate for a longer timescale T'. Then, the user can use mathematical induction on these lemmas to manually 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. The user can pick T' to be the smallest (CCA-dependent) value such that the lemmas hold. Sections 7 and 8 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 straightforward 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.

5 FORMAL ANALYSIS USING SMT SOLVERS

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.[4]

[4]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.

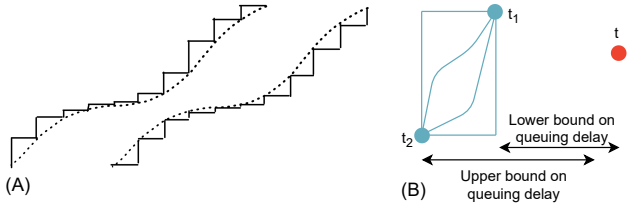

Figure 5: (A) While the bounds on $S(t)$ in the continuous model look like the dotted lines, we over-approximate that region using the solid lines as bounds. (B) In the discrete model, the queuing delay at time t can be any value between the upper and lower bounds.

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 Z3 SMT solver [16] 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.

5.1 SMT Formulation

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 times $t \in \{0, \cdots, T\}$.[5] 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.[6]

For computational efficiency, T must be small, leading to a coarse discretization with only 1 to 3 time steps per R_m.[7] 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

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

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

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

with $Ct - W(t)$ as upper bound and $C(t-D) - W(t-D)$ as lower bound, shown as dotted lines in Figure 5(A). When discretized, the bounds on S become step functions that contain their continuous counterpart.

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(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 $S_t > A_{t-\Delta t} - L_{t-\Delta t} \rightarrow delay_t \leq \Delta t$ and $S_t \leq A_{t-\Delta t} - L_{t-\Delta t} \rightarrow delay_t \geq \Delta t$.[8] Appendix C discusses how we constrain L_t^d 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.

CCAs. Recall from Section 4 that each CCA can introduce its own set of variables and constraints to set $cwnd(t)$ and/or $r(t)$. In our SMT formulation, we have variables $cwnd_0, \ldots, cwnd_T$ 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 $cwnd_t = f(\bar{cwnd}, \bar{Q}, \bar{E})$, where $\bar{cwnd}$ 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*.

5.2 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.

5.3 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.

5.4 Evaluation via Case Studies

We demonstrate the power of CCAC through three case studies in the next three sections: BBR [14], AIMD [15], and Copa [5]. 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 [37]. 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.

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

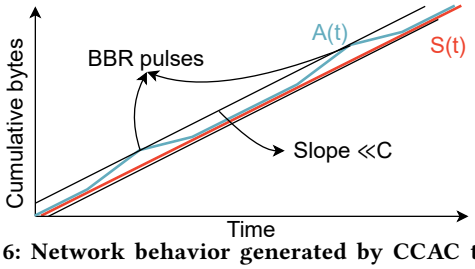

Figure 6: Network behavior generated by CCAC that prevents BBR from discovering bandwidth.

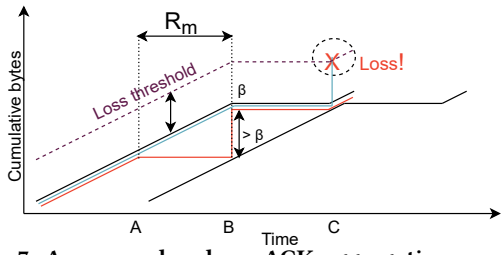

Figure 7: An example where ACK aggregation causes loss even when the congestion window is small.

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.

6 CASE STUDY 1: BBR

BBR [14] 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 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 [42, 47]. 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 [27], the version of QUIC [28] 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.[9] We believe that intentionally overestimating the pacing rate can increase delay on average, while the worst-case delay remains the same as before.

7 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: $inflight = A(t) - L^d(t) - S(t - R_m)$. When *inflight* drops below *cwnd*, the sender sends a packet. Our AIMD implementation handles duplicate ACKs and timeouts (§5.4). 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 1.

7.1 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 1) aims to find scenarios where AIMD can observe packet loss when *cwnd* is small. Specifically, we add the following constraint: $\bigvee_t (L_t > L_{t-1} \wedge cwnd_t \leq thresh)$ (in Figure 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

[9] 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.

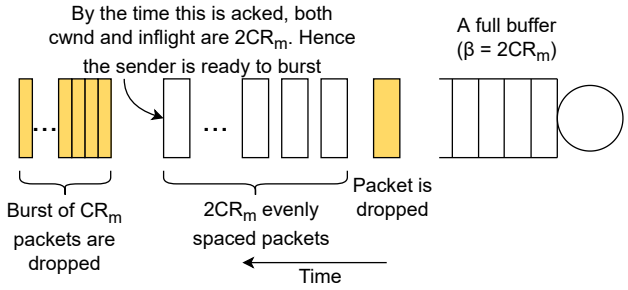

Figure 8: At the end of this sequence of packets, 1) the sender has reduced its *cwnd* from $4CR_m$ to $2CR_m$ due to loss, 2) the server has dropped CR_m packets 3) $inflight = cwnd = 2CR_m$ making the sender ready for another burst. Packets go from left to right.

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 7. 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 $inflight = A(t) - L^d(t) - S(t - R_m)$). 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 [22] (*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.

Figure 8 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 $cwnd = 4CR_m$). 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 $cwnd = CR_m$, utilization can be as low as $cwnd/(R_m + D)$, 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 [22] 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 $2cwnd/\text{smoothed\_rtt}$. 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 [38], which perhaps happened for the reasons discussed above. We used CCAC to confirm that losses can indeed occur when $cwnd < CR_m + \beta$ 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 $cwnd = \beta < CR_m + \beta$). However, it is difficult for even experienced engineers to determine this threshold, especially when multiple phenomena interact. In the next subsection, we show how CCAC can help us discover and prove this threshold (Theorem 2).

7.2 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

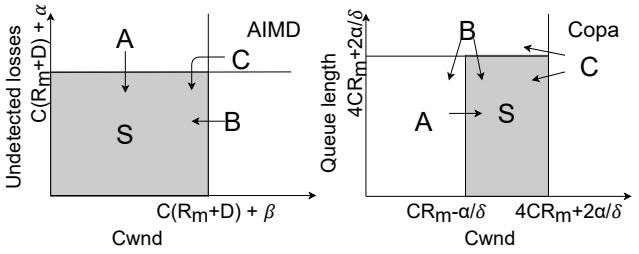

Figure 9: State diagram we use to prove AIMD and Copa's steady-state behavior.

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, $L_t - L_t^d$. 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 $CR_m > 5\alpha$,[10] 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 9. 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 $cwnd_0 > max\_cwnd \wedge L_0 - L_0^d \leq max\_undetected \wedge CR_m > 4\alpha$ then $cwnd_T < cwnd_0 - \alpha$
(2) If $L_0 - L_0^d > max\_undetected \wedge CR_m > 5\alpha$ then at least one of the following holds
   (a) $L_T - L_T^d \leq L_0 - L_0^d - C$ (i.e., undetected losses decrease by at least C) and $cwnd_T \leq max\_cwnd$
   (b) $cwnd_0 > max\_cwnd \wedge cwnd_T < cwnd_0 - \alpha$
(3) Once AIMD has reached steady state, it will remain there. That is, if $L_0 - L_0^d \leq max\_undetected \wedge cwnd_0 \leq max\_cwnd \wedge CR_m > 3\alpha$ then $\bigwedge_t L_t - L_t^d \leq max\_undetected \wedge cwnd_t \leq max\_cwnd$.

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(A)). The finer our discretization, the smaller this difference. Recall from Section 5.3 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.

8 CASE STUDY 3: COPA

Copa [5] is a delay-based algorithm like Vegas [12] and Fast [48] with 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.

8.1 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

[10] 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.

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 $\delta = 1/25$ [23]).

One would expect Copa to always be able to maintain high utilization. However, the counterexample generated by CCAC tells a different story. Figure 10 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[11] 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 10, 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 queueing 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). 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 10.[12] Here, the first box has arrival curve $A(t)$ and service curve $S_1(t)$ and the second one has arrival curve

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

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

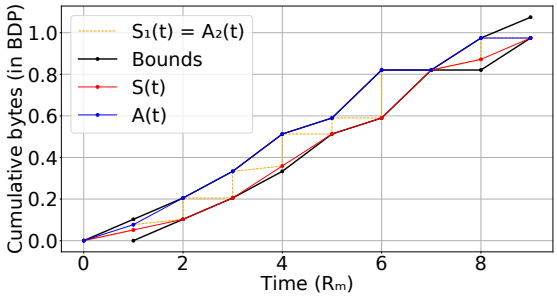

Figure 10: A trace generated by CCAC that causes Copa to severely underutilize network capacity. Here, $C = 1BDP/R_m$ and Copa is severely under-utilizing the link. A network can have a standing delay when it has more than a single box, even if the individual boxes do not maintain a standing queue. $S_2(t)$ and $A_1(t)$ denote S and A of the first and second boxes respectively.

$A_2(t) = S_1(t)$[13] and service curve $S(t)$. As evident in the figure, the arrival curve and the service curve 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.

8.2 Copa's Steady-State Analysis

We analyze Copa's steady state behavior using a similar approach to §7.2. 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 9). 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 9, 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., $A_T - S_T > 4CR_m + 2\alpha/\delta \vee (cwnd_T < cwnd_0 + \alpha/\delta \wedge cwnd_T < CR_m - 1/\delta)$). 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.*[14]

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.*

[13] 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.

[14] 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.

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 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.

9 LIMITATIONS AND FUTURE WORK

We believe that CCAC sets the initial steps towards a more comprehensive and formal understanding of the behavior of CCAs, raising new challenges and opportunities. For example, we have not yet analyzed how multiple flows compete on the same bottleneck, because doing so with our approach requires non-linear constraints in the SMT formulation. Z3 was unable to solve our formulation of these problems. When we restrict our model to linear arithmetic, Z3 was always able to answer queries for $T < 20$. This is a limitation of the solver and/or SMT formulation, not the model. Further, CCAC does not support receiver-driven protocols, MPTCP [51], schemes using in-network signals such as ECN [21], INT [30], XCP/RCP [20, 29], and ABC [25]. We discuss a possible way to handle these in Appendix B. The assumptions made by CCAC are listed in Section 3.

We do not have a composition theorem when buffers are finite and the first box is faster than the second one (§4.2). Due to discretization, the bounds CCAC produces may not be tight (§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 (§3).

Our CCA implementations are simplified (§5.4). 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 [10, 43–46] 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 §4.4) is semi-automated, and requires a manual component as illustrated in our case studies.

10 RELATED WORK

A long line of work in congestion control has relied on theoretical models [15, 33, 35, 41]. Our path model is similar to the ones developed in network calculus [11, 33]. We adopt 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 $A(t)$ is a function of $S(t - R_m)$. This closes the loop between the server and the packet arrivals and allows us to model CCAs. This 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 [31] and theoretically analyze [6] CCAs. Their goal is to make the analysis of a simple *deterministic* network easier. However, CCAC uses a non-deterministic path model, giving it many degrees of freedom to expose unexpected CCA behavior.

A related line of work is on formal verification of the *implementation* of a CCA given a specification of the algorithm using manual [10] and automated [43–46] 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.

Formal verification has been used in recent years in other areas of computer networking as well [8, 9, 24, 32].

11 CONCLUSION

CCAC is an automated tool with a built-in path model that proves correctness properties about CCA behavior or discovers counterexamples.It introduces a *performance-as-correctness* framework that we believe should motivate a new direction of research using formal verification methods to prove performance properties of network algorithms and protocols. We demonstrated its efficacy by analyzing AIMD, Copa, and BBR. We view this work as setting the initiat steps toward mathematically modelling congestion control in a way that captures the complex behaviors on real network paths. Theorems and bounds proved in this model can offer deeper insights into what is possible with end-to-end congestion control.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We thank Behnaz Arzani (our shepherd) and the SIGCOMM reviewers for their thoughtful comments. This work was funded in part by NSF grants CNS-1407470, CNS-2006827, CNS-1563826, CNS-1526791, and CNS-1751009, a Cisco Research Center Award, and a Microsoft Faculty Fellowship.

REFERENCES

[1] Accessed 2021. The ns-2 simulator. https://isi.edu/nsnam/ns/.
[2] Accessed 2021. The ns-3 simulator. https://nsnam.org/.
[3] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. Classic meets modern: A pragmatic learning-based congestion control for the Internet. In *ACM SIGCOMM*. 632–647.
[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM*. 63–74.
[5] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical delay-based congestion control for the internet. In *USENIX NSDI*. 329–342.
[6] François Baccelli and Dohy Hong. 2000. TCP is max-plus linear and what it tells us on its throughput. In *ACM SIGCOMM CCR*. 219–230.
[7] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1*. Ph.D. Dissertation. Inria.
[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *ACM SIGCOMM*. 155–168.
[9] Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. 2014. An assertion language for debugging SDN applications. In *ACM HotSDN*. 91–96.
[10] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *ACM SIGCOMM*. 265–276.
[11] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. 2018. *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons.
[12] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *ACM SIGCOMM*. 24–35.
[13] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. 2013. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX ATC*. 213–218.
[14] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. In *ACM Queue*. 58–66.
[15] D-M. Chiu and R. Jain. 1989. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems* 17, 1–14.
[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[17] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.
[18] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *USENIX NSDI 2015*. 395–408.
[19] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC vivace: Online-learning congestion control. In *USENIX NSDI*. 343–356.
[20] Nandita Dukkipati. 2008. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer.
[21] S. Floyd. 1994. TCP and Explicit Congestion Notification. *SIGCOMM CCR* 24, 5 (Oct. 1994).
[22] S. Floyd, T. Henderson, A. Gurtov, and Y. Nishida. 2004. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, IETF.
[23] Nitin Garg. 2019. Evaluating COPA congestion control for improved video performance. https://engineering.fb.com/2019/11/17/video-engineering/copa/.
[24] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*. 300–313.
[25] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. 2020. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In *USENIX NSDI*. 353–372.
[26] Stephen Hemminger, Fabio Ludovici, and Hagen Pfeifer Paul. 2011. The Linux netem network emulator. https://www.linux.org/docs/man8/tc-netem.html.
[27] Facebook Inc. Accessed July 2021. MVFST: Facebook's QUIC implementation, commit e04fcaac. https://github.com/facebookincubator/mvfst/commit/e04fcaacc1633c1bae78c61aac1f5f8a5784f657.
[28] Jana Iyengar and Martin Thomson. 2018. QUIC: A UDP-based multiplexed and secure transport. *Internet Engineering Task Force, Internet-Draft*.
[29] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion control for high bandwidth-delay product networks. In *ACM SIGCOMM*. 89–102.
[30] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. 2016. In-band Network Telemetry (INT). https://p4.org/assets/INT-current-spec.pdf.
[31] Hwangnam Kim and Jennifer C Hou. 2004. Network calculus based simulation for tcp congestion control: Theorems, implementation and evaluation. In *IEEE INFOCOM*, Vol. 4. IEEE, 2844–2855.
[32] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. 2011. Finding protocol manipulation attacks. In *ACM SIGCOMM*. 26–37.
[33] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network calculus: a theory of deterministic queuing systems for the internet*. Vol. 2050. Springer Science & Business Media.
[34] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 2012. TCP Selective Acknowledgment Options. RFC 1996, IETF.
[35] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM CCR* 27, 3, 67–82.
[36] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. 2020. PCC proteus: Scavenger transport and beyond. In *ACM SIGCOMM*. 615–631.
[37] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *ACM SIGCOMM*. 30–43.
[38] Yuchung Cheng Neal Cardwell. 2015. The Linux Kernel, commit 43e122b0. https://github.com/torvalds/linux/commit/43e122b014c955a33220fabbd09c4b5e4f422c3c.
[39] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*. 417–429.
[40] Matt Sargent, Jerry Chu, Dr Vern Paxson, and Mark Allman. 2011. *Computing TCP's retransmission timer*. Technical Report. RFC 6289.
[41] R. Srikant. 2004. *The Mathematics of Internet Congestion Control*. Springer Science & Business Media.
[42] Bo Su, Xianliang Jiang, Guang Jin, and Haiming Chen. 2020. Rethinking the rate estimation of BBR congestion control. *Electronics Letters* 56, 5 (2020), 239–241.
[43] Wei Sun, Lisong Xu, and Sebastian Elbaum. 2017. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In *ACM SIGSOFT*. 79–89.
[44] Wei Sun, Lisong Xu, and Sebastian Elbaum. 2018. Limitations of emulating realistic network environments for correctness testing of internet applications. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.
[45] Wei Sun, Lisong Xu, and Sebastian Elbaum. 2018. Scalably testing congestion control algorithms of real-world TCP implementations. In *IEEE International Conference on Communications (ICC)*. 1–7.
[46] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. 2019. Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations. In *USENIX NSDI*. 719–734.
[47] Yue Wang, Kanglian Zhao, Wenfeng Li, Juan Fraire, Zhili Sun, and Yuan Fang. 2018. Performance evaluation of QUIC with BBR in satellite internet. In *IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*. IEEE, 195–199.
[48] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. 2006. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking* 14, 6 (2006), 1246–1259.
[49] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. In *ACM SIGCOMM 2013*.
[50] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *USENIX NSDI*. 459–471.
[51] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP.. In *USENIX NSDI*, Vol. 11. 8–8.
[52] Yaxiong Xie, Fan Yi, and Kyle Jamieson. 2020. PBE-CC: Congestion Control via Endpoint-Centric, Physical-Layer Bandwidth Measurements. In *ACM SIGCOMM*. 451–464.
[53] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: a randomized experiment in video streaming. In *USENIX NSDI*. 495–511.
[54] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *USENIX ATC*. 731–743.

Appendices are supporting material that have not been peer-reviewed.

A AIMD COUNTEREXAMPLE IN DETAIL

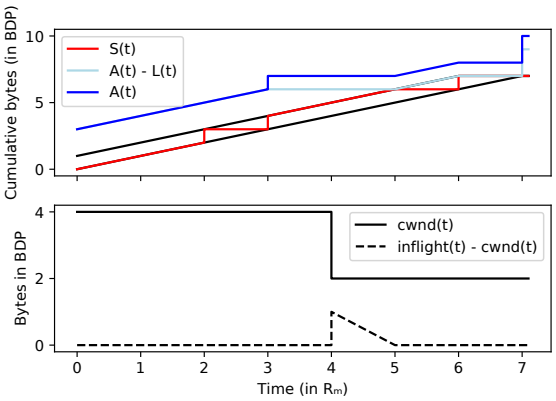

Figure 11: A trace of how a burst of $2CD$ bytes can be orchestrated by combining the two mechanisms in §7

We discuss further the example in Section §7 where CCAC found a way to exploit AIMD to cause a burst of $2CD$ bytes. Figure 11 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 11. When *cwnd* exceeds $4CR_m$, one packet 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.

B EXTENDING CCAC

In-network support. Some CCAs [4, 20, 25, 29] 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 §8 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.

C 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 $A(t) - L(t) + dupacks \leq S(t + \Delta t)$. 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 $detectable_{t,\Delta t}$, 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* upto 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 $S(t - R) \geq A(t - R - \Delta t) - L(t - R - \Delta t) + dupacks$. Hence we formally define $detectable_{t,\Delta t} = S_{t-R} \geq A_{t-R-\Delta t} - L_{t-R-\Delta t} + dupacks$ and add the constraints $detectable_{t,\Delta t} \rightarrow L^d_t \geq L_{t-R-\Delta t}$ and $\neg detectable_{t,\Delta t} \rightarrow L^d_t \leq L_{t-R-\Delta t}$ for each $\Delta t \in \{0, \cdots, t\}$.

D 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 [17], a proof assistant similar to Coq [7], 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 3. *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 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 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.

D.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 4. *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 5. *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 §4.1). Here, simply setting $\tau_s.W(t) = \tau_1.W(t)$ does the job.

Note, Theorem 4 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. □

D.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 6. *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 7. *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 5 and 6 if we set the buffer size to be infinity in theorem 5. □