Towards provably performant congestion control

Anup Agarwal, *Carnegie Mellon University;* Venkat Arun, *University of Texas at Austin;* Devdeep Ray, Ruben Martins, and Srinivasan Seshan, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Towards provably performant congestion control

Anup Agarwal[†], *Venkat Arun*[⋆], *Devdeep Ray*[†], *Ruben Martins*[†], *Srinivasan Seshan*[†]
[†]*Carnegie Mellon University,* [⋆]*University of Texas at Austin*

Abstract

We seek to ease the design of congestion control algorithms (CCAs) that *provably* perform well under diverse network scenarios including, cellular links, policers, token bucket filters, operating system jitter, etc. Guaranteeing performance under such conditions is hard as it requires considering combinatorial possibilities of CCA and network interactions. We build a framework that allows us to reason about CCAs. It describes (1) the *necessary* actions that any performant CCA must take, and (2) a provably *sufficient* amount of information for CCAs to consider when deciding their sending rate. Combining this framework with techniques in formal methods, we synthesize CCAs that provably perform across a diverse set of network conditions. Our methodology also led us to discover and prove fundamental impossibility results.

1 Introduction

End-to-end congestion control algorithms (CCAs) for the Internet must operate on varied network paths, each with its unique combination of physical links (e.g., wired, cellular, low-latency data center, satellite [3, 14, 15, 31, 57, 61, 63]) and processing elements (e.g., load balancers, schedulers, NICs, switches, routers). On such paths, CCAs balance often conflicting objectives like utilization, delay, packet losses, convergence time, fairness, and flow-completion time.

A robust general-purpose CCA that performs well across diverse network scenarios has remained elusive. All existing general-purpose CCAs (e.g., Cubic, BBR, PCC, Copa) perform poorly in some practical scenario [6, 7, 17, 20, 60] (§2).

We report on our attempt at an ambitious task: "to design CCAs that can *provably* meet performance objectives on a broad set of network paths". Unsurprisingly, this was hard. Our initial optimism came from recent work, CCAC [7] that uses computational methods to search for network behaviors that break a given CCA. We thought we could simply iterate over CCA design by asking CCAC if it violates a performance property, and if so, fix the CCA and repeat the process.

This was not as straightforward. While CCAC describes a scenario where a given CCA breaks, *it does not tell us how to fix the CCA*, i.e., (Q1) how should it change its sending rate choices, and in doing so (Q2) what signals/statistics might it consider. CCAC also does not tell us (Q3) if the performance property is even achievable by some CCA.

To answer these questions, we examined the CCAs developed over the last few decades. They consider numerous statistics to infer congestion (e.g., derivatives, integrals, exponentially weighted moving averages of loss/delay signals). There is no consensus on what statistics a CCA should maintain. On closer observation, we find that the statistics maintained by CCAs (implicitly) describe latent properties of the path that the CCA is running on. For instance, `ssthresh` in Reno/New Reno [33, 34] estimates a lower bound on the *BDP* of the network; BBR [15] explicitly maintains estimates of bandwidth and propagation delay.

We formalize this intuition by defining *belief set* as the set of paths (described using parameters like link rate, propagation delay, and buffer size) that the CCA believes it could be running on and the possible instantaneous state(s) of each path (e.g., packets in queue/on the wire). Given a model of the network (e.g., CCAC [7]), we give a canonical way to compute the belief set as the exhaustive set of paths and states that can explain the history of observations made by the CCA.

The belief set (or beliefs) gives us a way to model *all* CCAs and formally reason about them. First, we show that it is *necessary* for any performant CCA to shrink the size of the belief set, i.e., reduce uncertainty in the possible paths it could be running on, e.g., probe to check if the link rate could be higher. Second, we formally prove that the belief set is a *sufficient* set of statistics for a performant CCA to consider, i.e., if a CCA can ensure a certain performance property, then a "belief-based CCA" can also ensure it, where a belief-based CCA is one whose sending rate is a pure function of the belief set.

Beliefs help answer the above questions (Q1-3) and enable two key results. First, by combining beliefs with CCAC, we built a tool, CCmatic. It uses program synthesis techniques to systematically solve the *search problem*: "find a CCA in a *search space* that ensures given *performance properties* over all specified *network paths or scenarios*". Sufficiency of beliefs allows us to define an exhaustive and tractable search space, necessity of beliefs allows us to define performance properties, and techniques from CCAC allow us to define the network paths. CCmatic synthesized CCAs that guarantee performance across all paths described by models like CCAC, where existing CCAs struggle to even guarantee 1% utilization [7]. Despite being designed for theoretical "worst-case" links, the synthesized CCAs outperform or match existing CCAs on empirical links that resemble "average-case" networks. By design, the synthesized CCAs are short, modular, human-interpretable, and come with proven performance guarantees.

Second, experimenting with CCmatic, it sometimes reported that no CCA in the search space could meet the performance property, hinting that perhaps our performance property cannot be achieved. Despite the sufficiency of the belief set, this is not a definitive proof because CCmatic only explores a subset of belief-based CCAs due to computational limits. Nevertheless, using the sufficiency and necessity properties of the belief set, we prove a previously unknown fundamental tradeoff between loss and convergence time on shallow buffered networks. Intuitively, the combination of short buffers and jitter creates uncertainty in delay measurements, forcing CCAs to rely on loss-based signals. If CCAs probe for bandwidth aggressively, they converge faster but risk losses. If they probe conservatively, they mitigate losses but converge slowly. We quantify this relationship and synthesize CCAs that achieve different points on the Pareto frontier. Note, while this tradeoff may seem intuitive, formalizing it requires careful treatment (§6.2).

Our contributions are: (1) the belief framework to reason about congestion control (§4), (2) CCmatic, a tool to synthesize CCAs (§5), (3) the CCAs synthesized by CCmatic, proofs about their performance properties, and their empirical evaluation (§6.1, §6.3, §6.4), and (4) the impossibility theorems (§6.2). Note, to provide provable performance guarantees, we make several assumptions which we hope to relax in future work (§2, §8). In particular, we do not formally study fairness between multiple flows.

2 Motivation

We motivate beliefs using an example and outline our goals.

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

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

LEMMA 2.1. *To avoid arbitrarily low utilization, a CCA needs to* shrink *the set of possible paths (the belief set) it could be running on by obtaining an upper bound on C.*

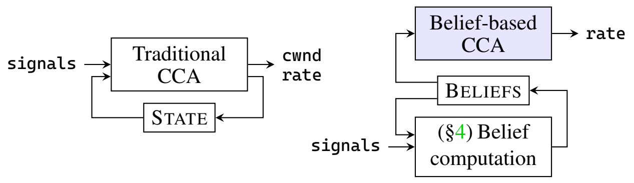


Figure 1: Beliefs standardize the state in congestion control.

In addition to it being necessary to shrink beliefs, we show in §4 that the belief set is the only information a CCA needs in order to decide sending rate. This is because the belief set is the only information a CCA needs to estimate the performance impact of its actions. This vastly simplifies CCA design (Fig. 1).

Traditionally, CCAs decide (Q1) what state (statistics) to maintain from input signals, and (Q2) sending rate. For instance, to answer Q1, CCA designers often consider "what does packet loss tell us about the state of the network?", "when can bandwidth be measured?", "what length of interval should be considered to sample bandwidth?" [15, 16].

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

Goals and non-goals. We want CCAs that *provably* achieve bounds on utilization, delays, losses, and time to converge to variations in link rate, under the following scenarios:

S1. Non-congestive delays or jitter. Delays can occur due to reasons unrelated to congestion [6, 29], e.g., delayed ACKs, ACK aggregation, OS scheduling, delays at the MAC and physical layers. This hinders CCAs from bounding end-to-end delays while ensuring high utilization. Traditional loss based CCAs [31, 33, 34, 52] fill up queues until they experience a loss and cannot bound delays. Delay based CCAs [8, 12, 15, 21, 46, 56, 57] use variation in measured round-trip times (RTTs) to estimate congestion. Jitter can cause these CCAs to mis-estimate congestion and send at a rate as much as 10× away from the correct rate [7, 21, 46, 60]. For instance, jitter can trick BBR [15] and Copa [8] into achieving near-zero utilization [7]. Recent learning-based CCAs (§7) also do not explicitly consider jitter.

S2. Shallow buffers with jitter. Paths with shallow buffers are common on the internet [24, 27, 28], because they prevent buffer-bloat and help manage cost/area/power of routers [4, 23]. On such paths, it is challenging to maintain utilization while avoiding excessive losses. Traditional loss-based CCAs, including Cubic [31], get poor utilization in single flow cases ([55], §6.4). ACK-clocked CCAs, despite pacing, send bursts of packets due to jitter (e.g., ACKs-aggregation [29]) which

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

risks excessive losses [7]. BBRv1 [15] is paced, but uses aggressive probes that incur O(BDP) losses periodically [17]. BBRv2 [27] and BBRv3 [28] incur lower loss on average but still incur O(BDP) losses in some cases (Appendix H).

In this paper, we only focus on the single-flow case with an infinite backlog of data to transmit when designing provably performant CCAs for S1 and S2. Designing provably fair CCAs robust to jitter is a hard problem [6] that we do not address. Our current formal framework gives no guarantees or predictions on the outcome of multi-flow experiments. Nevertheless, we empirically evaluate the fairness of our synthesized CCAs under simple network conditions (Appendix H). We find that some of our CCAs are fair while others are not. We believe that addressing S1 and S2 in the single-flow case is a key step towards addressing the multi-flow case. S1 and S2 are important real-world scenarios that have been the focus of recent changes to BBR [27, 28]. There are several situations where a flow is alone in its bottleneck queue [13] where our insights immediately apply (e.g., cellular networks). Our loss-convergence tradeoff (§6.2) also applies with multiple flows and may guide how buffers should be sized for networks with jitter.

Formally addressing S1 and S2, even in the single-flow case, was challenging. As described above, none of the existing CCAs achieve our goals. Beliefs and automated reasoning allowed us to systematically explore the design space of CCAs, unveiling previously unknown tradeoffs and novel CCA mechanisms. We discovered and quantitatively proved a fundamental tradeoff between loss and convergence time on networks with shallow buffers. We synthesized CCAs that are on this Pareto frontier. Existing CCAs, unaware of the tradeoff, make sub-optimal tradeoffs or only explore a subset of useful points on the Pareto frontier. We also discovered new ways to use sending rate decisions to augment the information obtained from RTTs and ACK rates. This leads to better estimates of the network parameters (bottleneck bandwidth/buffer) and state (queuing or extent of congestion) (§6.2.1). We illustrate here with two examples.

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

Example 2. Say a CCA sent a burst of packets to probe for available bandwidth and observed that the probe did not incur any packet loss. Then we can conclude that either the buffer or the bandwidth is large enough to have accommodated the burst. I.e., both the buffer and bandwidth cannot be small as that would have incurred a loss. We cannot make such a conclusion by relying on ACK rate and RTT measurements alone. Due to jitter, a burst may not lead to an immediate increase in ACK rate ([7], §4) which traditionally would have allowed us to conclude bandwidth is large. Likewise, an inflation in RTTs could be due to jitter, and we cannot assume that the bottleneck buffer is large enough to accommodate the inflation in RTTs.

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

Table 1: Glossary of symbols. The square brackets show the units: bytes [b], rate [r], and time [t]. inst. = instantaneous. Inflight is bytes that are unacknowledged and not inferred as lost.

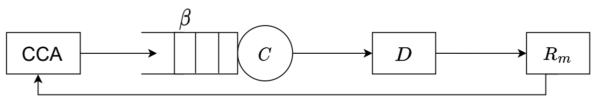


Figure 2: CBR-delay network model.

CCmatic automatically synthesizes CCAs that use such insights to make non-trivial decisions about when/how to probe/drain. E.g., it realized that draining is necessary not only to maintain low delay, but also to restrict losses when probing for bandwidth on paths with jitter and shallow buffers (§6.2.1).

Our work addresses an important and challenging set of scenarios, and establishes a formal methodology for further exploration. In the future, we hope to extend our methodology to formally explore fairness between multiple flows, and robustness to application-limited flows and non-congestive losses (§8).

3 Network models

We use CCAC [7] to succinctly express and efficiently explore the scenarios in §2. CCAC uses a single bottleneck abstraction to summarize the cumulative effects multiple elements on a network path. It uses bounded model checking [19] to provide a trace of CCA execution under various network behaviors.

Our investigations revealed that CCAC expresses behaviors that are perhaps too adversarial for any CCA to handle (§6.1, §6.2). So, we explore two other network models that are *weaker*, i.e., they are less challenging from the CCA's point of view as they capture strictly fewer behaviors. We briefly describe these models and use notation from Table 1. Note, if a CCA works on a stronger model then it also works on a weaker model, and an impossibility result for a weaker model holds for a stronger model.

CBR-delay. This is motivated from [6]. It abstracts the network as a constant bit rate (CBR) box followed by a non-deterministic delay (or jitter) box, and a propagation delay of R_m seconds (shown in Fig. 2). The CBR box has a constant (over time), but arbitrary bottleneck bandwidth of C bytes/second, and a buffer of size β bytes. It expresses the queueing or *congestive delays* (and losses) at the bottleneck queue in a network path. The delay box can add up to D seconds of delay *non-deterministically*. **Note, non-determinism is different from randomness or stochasticity (e.g., uniform random delays).** Non-determinism allows the network to *arbitrarily* inject bursts and provably express the cumulative effect of various sources of jitter [6, 7, 10, 42]. Non-determinism (as opposed to randomness) can express non-congestive delays that may have causal effects or correlations.

CCAC [7]. It includes all behaviors captured by CBR-delay. Additionally, CCAC can non-deterministically accept a burst of packets without building up a queue, effectively hiding congestive delays/losses even when the CCA is sending above the link rate. In contrast, CBR-delay can inject non-congestive delays but not non-deterministically hide congestive delays. This is a crucial and previously unknown distinction that changes the tradeoffs that CCAs must make (§6.1, §6.2).

Ideal link. It cannot add any jitter to packets. It is simply a FIFO queue, with a constant (but arbitrary) bandwidth and propagation delay. Several theoretical analysis [9, 18, 45, 49, 62] have used similar modeling. We study this to compare CCAs designed for the ideal link vs. stronger models.

Variations in link rate over time. Like CCAC, all the models assume a fixed link rate over time. We use CCAC's approach to express variations in link rates. CCAC and CBR-delay express short-term link rate variations using jitter. All the models express long-term link rate variations by arbitrarily choosing initial conditions. E.g., a trace that begins with a high congestion window (*cwnd*) relative to C emulates a scenario where link rate decreased. Alternatively, a large initial queue buildup, and low *cwnd* can emulate a case where the link rate decreased and the CCA backed off, but the queue has not drained. One can stitch such traces together to explore longer executions with potentially multiple link rate variations (§5.2).

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

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

4 Belief framework

DEFINITION 4.1. *A **belief set** (or beliefs) for a given network model is the set of paths (and their latest states) that could have produced (according to the network model) the historical sequence of CCA's observations. It is the CCA's* belief *about the paths and states of the network it is running on.*

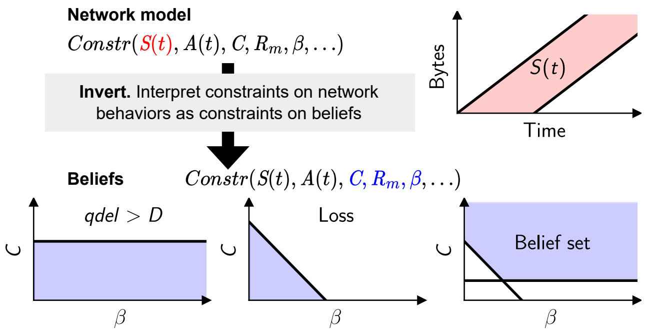

Network model
$Constr(S(t), A(t), C, R_m, \beta, \dots)$
Invert. Interpret constraints on network behaviors as constraints on beliefs
Bytes
$S(t)$
Time
Beliefs
$Constr(S(t), A(t), C, R_m, \beta, \dots)$
$qdel > D$
Loss
Belief set
C
β

Figure 3: Inverting the network model to compute beliefs. Depending on the CCA's observations, we may get different bounds on the belief set. The bottom three plots illustrate the constraints on the belief set we get depending on whether the CCA observed $qdel > D$, loss, or neither (for CBR-delay).

For example, for the CBR-delay model, the belief set is a set of tuples of the form $\langle C, R_m, D, \beta, q, \theta \rangle$. Such a tuple is in the belief set *if and only if* it can explain (according to the network model), the observations of the CCA thus far. We use the term observations to collectively refer to CCA's actions and feedback from the network.

Computing beliefs. We can "invert" the network model's relation to compute beliefs (Fig. 3). Specifically, the constraints of the model describe the feasible ways in which the network's state and feedback can evolve (e.g., how it services (delays) packets, drops packets, and builds queues), given the CCA's sending behavior, the network path, and the network's initial state. I.e., the constraints describe feasible combinations of $\langle S(t), L(t), q(t), \theta(t) \rangle$, $A(t)$, $\langle C, R_m, D, \beta \rangle$, and $\langle q(0), \theta(0) \rangle$. If we fix the observations, i.e., $\langle A(t), S(t), L(t) \rangle$, then the constraints describe the feasible paths and states that could have produced CCA's observations. This is the belief set. Each constraint of the network model gives us a constraint on the belief set. This is similar to conversion of a partially observable markov decision process (POMDP) into a belief MDP [37]. We perform the inversion §5.1 and §6.2.

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

Shrinking beliefs is necessary. CCAs need to shrink the size of the belief set, i.e., infer the possible parameters and states of the network they are running on. We illustrate this using a family of lemmas (Lemma 2.1, Lemma 4.1) that show the dimensions along which beliefs need to shrink to ensure different performance properties.

LEMMA 4.1. *To ensure an upper bound on queueing delay (qdel), a CCA needs to shrink the set of possible propagation* *delays the network could have.*

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

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

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

THEOREM 4.1. *If there exists a* deterministic CCA *that ensures a performance property on a network model, then, there exists a* belief-based *CCA that ensures the performance property on the model.* Where, a deterministic CCA is a CCA whose actions are a function of the entire history of past observations (i.e., its actions and feedback) and a belief-based CCA is a CCA whose actions are a function of the belief set computed using the network model over the history of past observations.

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

We use this theorem to (1) synthesize CCAs as function of the belief set, and (2) to prove impossibility results, i.e., if there is no action that can ensure a performance property over all tuples in a valid belief set, then the performance property is not achievable. We use such arguments in §6.2.

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

The belief set exhaustively summarizes the history of the game, making it *memoryless* (similar to the board state in chess). Beliefs serve as a "board" by meeting the two requirements: (R1) we can determine the set of feasible moves for the players from the board allowing us to enforce the game rules (Lemma A.1), and (R2) we can update the board by applying the moves (Lemma A.2). As a result, future progressions of the game (and any optimal strategies) depend only on the board (beliefs) irrespective of the history that led to the board.

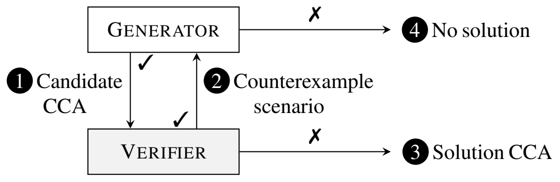


Figure 4: CEGIS loop (adapted from [1]).

5 CCmatic: Synthesizing CCAs

We use Counter-Example Guided Inductive Synthesis (CEGIS) [51] (a program synthesis technique) to synthesize (search for) CCAs given a specification (i.e., a network model and a performance property).

CEGIS iteratively generates a candidate CCA from a search space ❶, and finds a counterexample scenario (network path, initial state, and non-deterministic choices) that breaks the performance property for the candidate CCA ❷ (Fig. 4). The search space is pruned using the counterexamples (see below), and eventually the loop terminates if either (1) the verifier cannot find a counterexample (and thus, the CCA achieves the performance property) ❸, or (2) the entire search space has been pruned (no CCA in the search space can achieve the performance property) ❹.

We implement the generator and verifier using the constraint solver Z3 [47], by encoding the search inputs (i.e., search space, network models, and performance properties) into SMT (Satisfiability Modulo Theories) constraints in the theory of linear real arithmetic (LRA) [38]. In the generator, we only search for CCAs that pass unit checks (e.g. do not add bytes with seconds). For each counterexample, we add constraints to prune all CCAs that make the same sending rate choices as the candidate CCA on the counterexample. We explored encodings that prune more CCAs, these did not yield significant reduction in search time, and we omit their details.

For encoding into SMT, we use beliefs to define the search space (§5.1) and a transition system abstraction to systematically define performance properties (§5.2). For the network models, we adopt the encoding proposed by CCAC [7]. It discretizes time and uses Network Calculus [41] style formulas to constrain how the network serves packets. Note, the synthesis happens offline. One can directly implement the synthesized CCAs in network stacks like the Linux kernel and QUIC [40]. For completeness, we provide details on the implementation of the verifier and generator in Appendix C.

5.1 Belief-based CCA Template

In CEGIS, the search space is often specified using a template or grammar. The template has placeholders (or holes) that the generator fills to synthesize a concrete CCA. It describes the **inputs** (e.g., loss/delays signals) that the CCA takes and the mathematical/logical **operators** it can use to compute its outputs (i.e., rate and state). Due to Theorem 4.1, the templates do not need to describe state computations. They can just take beliefs as inputs and produce rate as the output

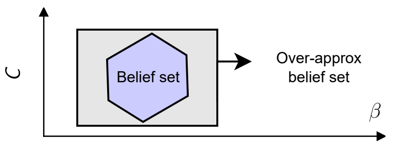


Figure 5: Over-approximating beliefs.

to fill the shaded box in Fig. 1.

Inputs. We face two challenges with beliefs. First, the belief set may be a complicated object in the ⟨`path`,`state`⟩ space. For easier encoding, we *over-approximate* it using closed-form expressions (see below). Second, our network models do not directly model variations in link rate (§3). When the link rate varies, it can go outside the belief set. To address this, we re-compute beliefs using a recent history of observations (§5.1.2).

Over-approximating beliefs. We construct closed-form expressions representing bounds on the network parameters and states. For example, we use C_L and C_U to represent lower and upper bounds on the C values in the belief set, and pass these bounds as input to the CCA. By default, we pass upper and lower bounds on C (C_U, C_L), and $qdel$ ($qdel_U$, $qdel_L$) as a proxy for q. We describe how we compute them in §5.1.1. We assume the CCA knows R_m, and design CCAs for $D = R_m$ (i.e., jitter can be as large as R_m, e.g., due to WiFi ACK aggregation [29]). This enables efficient synthesis by discretizing time in units of R_m (as in CCAC [7]). In reality, a CCA does not know R_m. Nevertheless, we show in Appendix B that if we use a CCA designed for R_m equal to the minimum RTT seen thus far, we can guarantee performance because the synthesized CCAs are inherently robust to uncertainty in RTTs caused by jitter.

While the closed-form expressions may over-approximate beliefs, i.e., include extra paths that cannot produce the CCA's observations (Fig. 5), they never remove paths that can produce its observations. We use the verifier's (i.e., CCAC's) assistance to both validate correctness of the closed-form expressions and to derive them (§5.1.1).

Over-approximation (as opposed to under-approximation) does not break soundness. The CCA believes it could be running on extra paths, and needs to take actions that do not violate performance on the extra paths. However, over-approximation does break completeness, i.e., Theorem 4.1 no longer applies.[2] It may happen that we summarize two different histories using the same approximate beliefs even though the actual belief sets are different. A belief-based CCA is allowed to take different actions on the histories, but a CCA in the template is forced to take the same action. As a result, CCmatic may output "no solution in template" even though a belief-based CCA works. When this happens, we explore weaker network models where we can add additional beliefs (e.g., β_L, q_U) or tighten existing ones (§6.2.1). Despite approximations, CCmatic synthesizes novel CCAs (§6.1).

Operators. Some CCAs use nonlinear operators like cube root [31] or division [8]. Non-linearities slow down SMT solvers [58]. Instead, we search for piece-wise linear functions to map belief bounds to sending rates, represented as (nested) if-else statements. The generator synthesizes the conditionals and expressions in these statements as linear combinations of the belief bounds. While CCmatic only searches for CCAs in the template, we are able to generalize CCmatic's insights to arbitrary CCAs, e.g., impossibility results in §6.2.

[2]Note, Theorem 4.1 also does not apply because we only explore a subset of belief-based CCAs using CCmatic.

Listing 1: Belief-based CCA template

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

Summary. Our templates take the form in Listing 1. $\boxed{?}$ denotes holes to be chosen by the generator. Different templates have different number (and nesting) of the "`if`" conditions. To keep the search tractable, we restrict the domain of the holes to be a small finite set, e.g., $\{-3, -5/2, -2..., 3\}$.

5.1.1 Computing belief bounds

Queueing delay. Traditionally CCAs estimate queueing delay as $RTT - R_m$ [8]. However, this does not account for non-congestive delays. In the CCAC and CBR-delay models, $RTT(t) = R_m + qdel(t) + \texttt{jitter(t)}$, where $0 \le \texttt{jitter(t)} \le D$. Consequently, $qdel(t) \in [qdel_L(t), qdel_U(t)]$ where,

$$qdel_U(t) = RTT(t) - R_m$$
$$qdel_L(t) = \max(0, RTT(t) - R_m - D) \quad (5.1)$$

Link rate. BBR [15] estimates link rate as the measured ACK rate. Jitter can create transient variations in ACK rate and mislead this estimate. Using the verifier (CCAC), we computationally derive the set of link rates that can explain an average ACK rate of r (i.e., the CCA received rT bytes in an interval $[t_1, t_2]$ of length $t_2 - t_1 = T$ seconds). We also derive this set analytically in Appendix B.

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

$$C_L(t) = \max_{0 \le t_1 \le t_2 \le t} \frac{r \cdot T}{T+D} \quad C_U(t) = \min_{0 \le t_1 \le t_2 \le t} \frac{r \cdot T}{T-D} \quad (5.2)$$

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

5.1.2 Handling stale beliefs

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

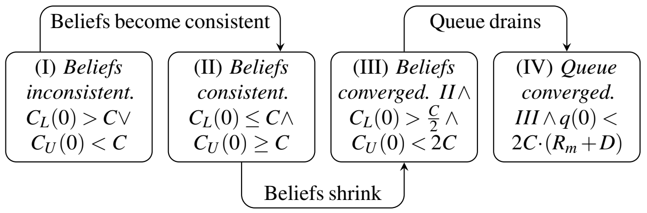
We "time out" beliefs periodically (e.g., every $10R_m$), and also when they become **empty (or invalid)**, e.g., $C_U < C_L$. On a timeout, we re-compute beliefs using the history of observations since the last timeout. When beliefs become inconsistent by a large margin, they become invalid quickly.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

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

Note, the periodic timeouts can interfere with a CCA's probes to estimate the belief set. We add constraints to prevent unnecessary timeouts. E.g., a CCA might be draining the queue and may not observe any delays/losses. If we recompute beliefs only using the recent history, there may be no upper bound on C, and the CCA would need to re-probe C_U from scratch. To prevent this, we only time out beliefs when the size of belief set is small (e.g., $C_U \leq 1.1C_L$), and we put bounds on how much the belief set can expand, e.g., $C_U(\texttt{now}) \leq 2C_U(\texttt{last\_timeout})$. These restrictions along with the timeout period affect how quickly CCAs tracks changes in link rate. Appendix F (Lemma F.2) studies this theoretically and Appendix H (Fig. 19 and Fig. 18) studies this empirically.

5.2 Transition system based properties

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



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

Figure 6: An example transition system. The formulas in the boxes (e.g., "$III \land q(0) < 2C \cdot (R_m + D)$") define the states.

Transition system. Fig. 6 shows an example transition system. The states are represented as symbolic boolean formulas (e.g., "beliefs are consistent", or $C_L(0) \leq C \leq C_U(0)$). For each state, users specify (1) transitions made in time 0 to T (e.g., "beliefs shrink": $C_L(T) > 1.5C_L(0)$) and (2) objectives during the period (e.g., "delay is at most D seconds": $\land_{t \in [0,T]} qdel(t) \leq D$). Note, users only declare "what" properties the CCA should ensure, CCmatic figures out "how".

A belief-based CCA under our network models typically makes the following transitions. Whenever the link rate changes, the beliefs can become inconsistent (stale). Eventually, the beliefs become consistent (due to §5.1.2), then the beliefs shrink (as this is necessary from §4), and finally, the CCA reaches the steady state (e.g., state IV) where it meets its steady-state objectives. Users can express both steady-state and transient objectives. For instance, high loss may be acceptable during slow start (e.g. in state I-III), but not for subsequent bandwidth probes (e.g., in state IV).

Proofs and encoding. We build a *lemma* for each state (e.g., State I $\Longrightarrow$ (State I objectives $\land$ State I transitions)) and take the conjunction (logical and) of these lemmas. This conjunction serves as the *proof of performance* for the CCA, as it *exhaustively* describes the states that the CCA visits and performance it ensures in each state. It is exhaustive because the disjunction of the transition system states is a tautology, i.e., covers all possible states. Appendix F gives the encoding of the lemmas and shows how they work together in a proof.

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

Synthesis invariant. We do not directly use the proof (i.e., the conjunction of lemmas) for synthesis. Instead, we build *under-specified* invariants that are *necessary* for a proof but not sufficient. We under-specify for two reasons. First, the lemmas contain constants that depend on the CCA, which is not known at the time of synthesis (e.g., $\mathbf{2}C$, $\frac{C}{\mathbf{2}}$, in Fig. 6). Second, before synthesis, we do not even know if there exists a CCA that can meet the lemmas. Under-specification allows us to synthesize reasonable-looking CCAs, which we then process *post synthesis* (see below).

To under-specify, we drop literals (inequalities) with unknown constants. Notice, that such literals define states III and IV (Fig. 6). Due to dropping, we cannot distinguish between states II-IV. We coalesce states II-IV into one, and allow the CCA to take any transition that is valid for states II-IV (effectively taking disjunction of the transitions). We similarly combine state III and IV objectives into `steady_state_obj`, and retain state II objectives as `transient_obj`. The result is Eq. 5.3 below. We use it for synthesizing CCAs with different choices of objectives (§6.1).

$$\begin{aligned}&\text{beliefs inconsistent} \implies\\&\quad(\text{State I objectives} \land \text{beliefs become consistent})\\&\land \text{beliefs consistent} \implies (\texttt{transient\_obj} \land (\text{beliefs shrink}\\&\quad\lor \text{large queue drains} \lor \texttt{steady\_state\_obj})) \qquad (5.3)\end{aligned}$$

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

Post synthesis. The synthesis invariant is not a sufficient proof. Hence, after synthesis, we build proof lemmas for the solution CCAs. To determine the constants in the lemmas, we use binary search to identify the region of values for which the lemmas hold. We describe this process in Appendix F. If there is no value of constants for which a particular lemma holds, we tweak the CCA, invariant, and/or the lemma(s). For example, when trying to build proof lemmas for a particular synthesized CCA (`cc_probe_slow` in §6.1), we found that the queue needs to be drained before beliefs can converge, i.e., "queue converged" in state IV (Fig. 6) needs to happen before "beliefs converged" in state III. So we reorder and redefine the states in the transition system and lemmas to reflect this and build a proof of performance for `cc_probe_slow`.

6 Results

We present four types of results. (§6.1) CCAs synthesized by CCmatic for various environments and objectives combinations. (§6.2) Fundamental tradeoffs inspired by negative outputs of CCmatic. (§6.3) Proofs that the synthesized CCAs ensure their performance objectives. (§6.4) Empirical evaluation of the synthesized CCAs to validate our mathematical modeling and proofs of performance.

6.1 Synthesis queries

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

Objectives. We require CCAs to ensure a lower bound on the utilization, and upper bounds on the amount/frequency of losses and bytes in flight (to bound packet latencies). We add all these objectives to `steady_state_obj` in Eq. 5.3. Our primary focus is on exploring asymptotic bounds, e.g., are losses $O(C)$, $O(\log(C))$, etc. So we specify asymptotic bounds with loose constants. For example, we query CCAs that ensure utilization $\geq 50\%$, and inflight $\leq 5 \cdot C \cdot (R_m + D)$. The loose constants allow under-specifying the synthesis invariant (§5.2). Later, when building proofs, we identify the best constants the synthesized CCAs can achieve (Appendix F). Note, the inflight bound has to be at least $CR_m + CD$. Because, to provide a utilization lower bound, a CCA needs to fill the wire (CR_m bytes) and build D seconds or CD bytes of queueing (Theorem 2 of [6]).

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

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

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

When CCmatic does not produce a solution, we increase (1) the program size (number of "`if`" expressions), and (2) the

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

Table 2: Summary of queries. The "# solutions" column also lists CCAs that are representative of the solutions produced. Some entries in "constrain loss" column are not applicable (N/A) as (congestive) loss is not possible when buffer is infinite.

Listing 2: Synthesized CCAs ($\alpha_R = MSS/R_m$)

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

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

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

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

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

length of the trace that the verifier considers (to give CCAs more time to show the progress required by our invariant). E.g., in G1, we get more solutions as we increase the trace length. For other queries we only show queries with the largest templates and longest traces. We also add new or tighter beliefs to the templates when we explore weaker models, e.g., G6.

Synthesized CCAs. Listing 2 shows the synthesized CCAs. We group similar queries together. G1, G2, and G3 explore CCAC with infinite, small and arbitrary buffer respectively. CCmatic synthesized CCA1 (`cc_qdel`), CCA2 (`cc_probe_fast`), and CCA3 (`cc_probe_drain`). CCmatic automatically figured out non-trivial insights about the network. E.g., `cc_qdel` simultaneously guarantees bounds on utilization and inflight. It sends above C_L until $qdel_L > 0$. $qdel_L > 0$ (Eq. 5.1) can only happen if queue is non-zero which can only happen if the link is utilized. Likewise, draining whenever $qdel_L > 0$ ensures inflight is bounded.

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

We empirically evaluated `cc_probe_drain` and found that it incurs periodic large loss (experimental setup described in §6.4). This was surprising as we specifically queried for a CCA that avoids large losses in steady state. We realized this happened because of under-specifying the synthesis invariant. Due to the disjunction, "beliefs shrink $\vee\cdots\vee$ `steady_state_obj`", the CCA is allowed to cause large losses if this allows shrinking beliefs. As a result, on the periodic belief timeouts (§5.1.2), `cc_probe_drain` caused losses when re-probing to re-estimate beliefs (similar to BBR's 8 cycle probes).

G4 We updated our query to ensure that when beliefs are consistent, CCA's probes (increasing sending rate) should not incur large losses.[3] I.e., we "AND" the synthesis invariant with the formula: "beliefs consistent $\implies$ (sending rate increases $\implies$ no large loss)". This is equivalent to updating `transient_obj` in Eq. 5.3. CCmatic did not produce any solution after this modification.

G5 To dig deeper, we investigated weaker network models (§3). We set $D = 0$ to emulate an ideal link. CCmatic synthesized a CCA that sends at rate "$C_L + MSS/R_m$", allowing it to probe for bandwidth while risking at most constant losses. However, this does not work with CCAC. Since CCAC can delay packets, this probe may not lead to an immediate increase in ACK rate. For CCAC, a CCA needs to build D seconds of queueing to disambiguate effects of utilization (congestion) from non-congestive delays (see §6.2). Sending at $C_L + MSS/R_m$ takes $\Omega(C_L)$ time to build a queue of D seconds, and the same CCA cannot show progress (shrink

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

beliefs) in a short fixed-length trace.

G6 We ran synthesis for CBR-delay using the same beliefs as CCAC. CCmatic could not synthesize any CCA that could avoid large loss. This led us to discover and prove a fundamental tradeoff between loss and convergence time (§6.2). The proof led us to a tighter beliefs for CBR delay. Using these, CCmatic synthesized CCA4 (`cc_probe_slow`). `cc_probe_slow` meets the loss-convergence tradeoff implying that it is tight. It risks $O(1)$ packet loss and takes $O(C)$ time to converge. For the synthesis, we added belief bounds on link rate ($C_{L,\lambda}$), buffer size, and bytes in queue (q_U). We explain these bounds and working of `cc_probe_slow` in §6.2.1.

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

6.2 Loss vs. convergence tradeoff

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

THEOREM 6.1. *For an end-to-end deterministic CCA running on a CBR-delay network with parameters $\langle C, R_m, D, 0 < \beta \leq CD \rangle$, to avoid getting arbitrarily low utilization, the CCA must either (1) cause $\omega(1)$ packet loss, i.e., losses that increases with C, or (2) take $\Omega(C(R_m + \beta_s))$ time to converge to the link rate. Where, $\beta_s = \beta/C$, i.e., buffer size in seconds.*

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

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

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

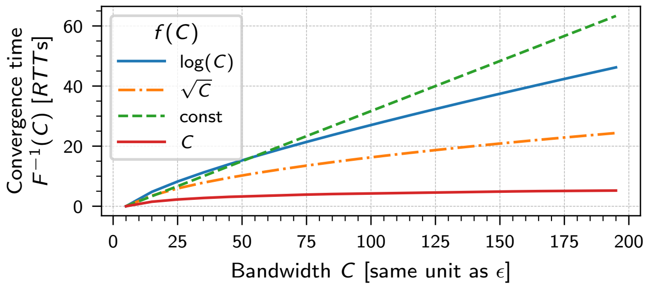


Figure 7: Each curve shows $F^{-1}(C)$ corresponding to $f(C)$ in the legend. I.e., under loss allowance f, the time F^{-1}, to converge from some small (positive) rate ε to C. Time from C_0 to C is given by "time from ε to C" − "time from ε to C_0" or $F^{-1}(C) - F^{-1}(C_0)$.

Fig. 7 shows the convergence time F^{-1} for different choices of loss allowances f. For example, if the CCA is willing to risk O(C) losses, i.e., $f(C) = C$, then the convergence time is $\Omega(\log(C/C_0))$. If $f(C) =$ `const` (independent of C), then the convergence time is $\Omega(C - C_0)$. For other functions f, the convergence time may not have a nice closed form expression.

While this tradeoff may seem intuitive, it only holds when there is complicated jitter. On ideal links, a CCA can ramp up in $O(1)$ time while risking $O(1)$ packet losses, using packet trains [39]. The inter-arrival times for a packet train is the inverse of the bottleneck rate (i.e., MSS/C) and reveals C. Such techniques may even work for links with iid (independent and identically distributed) jitter. However, in practice, links do exhibit complicated jitter patterns and break such bandwidth estimates [22].

Below, we provide intuition and outline the proof for Theorem 6.1. We also prove a similar loss-convergence tradeoff for ideal links when packet trains are not allowed and $\beta = 0$. This proof is similar to that of Theorem 6.1, we omit it for brevity.

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

We investigated a CCA that additively increases its sending rate every RTT until it sees a loss, to see why it cannot avoid large loss. The verifier gave us a trace where the CCA keeps blindly increasing its sending rate even when it is already above the link rate. Eventually the queue builds up and $O(BDP)$ loss happens. The CCA needs to resort to blind increases because it does not get any feedback until it causes loss (as queueing delay measurements are meaningless). If these blind increases are aggressive, this results in larger losses with faster convergence and vice versa. `cc_probe_slow` drains queues along with additive increments to meet the bound in Theorem 6.1 (§6.2.1).

Proof outline. (Full proof in Appendix D) We first show that due to $\beta \leq CD$, a CCA must cause loss to avoid arbitrarily low utilization (**Step 1.**). Then we compute a *tight* lower bound belief ($C_{L,\lambda}$) for C, for CBR-delay (belief computations in §5.1.1 were for CCAC) (**Step 2.**). The CCA could be running on any link with $C \geq C_{L,\lambda}$, and it needs to ensure it does not cause loss on any of these links. This allows us to compute the amount of loss a CCA risks any time it probes for bandwidth (**Step 3.**). If we restrict this risk of loss to a constant independent of C, it gives us a constraint on how quickly the CCA can ramp up, giving us a lower bound on the convergence time (**Step 4.**).

6.2.1 `cc_probe_slow` shows Theorem 6.1 is tight

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

Bandwidth and buffer beliefs. For CBR-delay, to obtain an upper bound on C, a CCA needs to cause loss or build more than D seconds of queueing (from **Step 1.** of Theorem 6.1). We compute the set of paths $\langle C, \beta \rangle$ that can produce CCA's observations until time t^*, such that the CCA has not observed $qdel > D$ or loss until t^*. This means that until $t^* - RTT(t^*)$, the enqueued bytes never exceeded the dequeued bytes by more than D seconds (CD bytes) or buffer size. I.e., $\forall t_1, t_2$, such that, $0 \leq t_1 \leq t_2 \leq t^* - RTT(t^*)$:

$$\int_{t_1}^{t_2} \lambda(s)ds - C\cdot(t_2 - t_1) \leq CD \tag{6.1a}$$

$$\text{and, } \int_{t_1}^{t_2} \lambda(s)ds - C\cdot(t_2 - t_1) \leq \beta \tag{6.1b}$$

From (6.1a), we define $C_{L,\lambda}$ as:

$$C_{L,\lambda}(t^*) = \max_{0 \leq t_1 \leq t_2 \leq t^* - RTT(t^*)} \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} \tag{6.2}$$

We evaluate (6.1b) over the interval $[t_1^*, t_2^*]$ (length $T^* = t_2^* - t_1^*$)[4] corresponding to the tightest bound on $C_{L,\lambda}$:

$$C_{L,\lambda}(t^*) \cdot (t_2^* - t_1^* + D) - C \cdot (t_2^* - t_1^*) \leq \beta$$

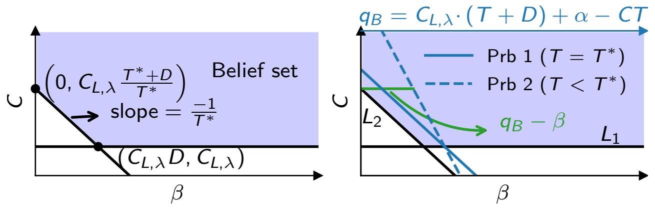
$$\text{or,} \quad C \cdot T^* + \beta \geq C_{L,\lambda}(t^*) \cdot (T^* + D) \tag{6.3}$$

From (6.2) and (6.3), we get the belief set illustrated in Fig. 8: $\{\langle C, \beta \rangle \mid C \geq C_{L,\lambda}(t^*) \wedge C \cdot T^* + \beta \geq C_{L,\lambda}(t^*) \cdot (T^* + D)\}$. $C_{L,\lambda}$ **tells two things, (1) the link rate is at least $C_{L,\lambda}$, and (2) if the link rate is $C_{L,\lambda}$, then the buffer is at least $C_{L,\lambda}D$ bytes.**

In the SMT encoding, we only evaluate Eq. 6.2 over intervals of length $T^* = R_m$. This is because the measurement interval T^* influences CCA's probing behavior (see below and Appendix E). The initial conditions (including initial beliefs) are chosen by the verifier (§3). As a result, verifier's initial choice of T^* influences the CCAs probing which should be in full control of the CCA. Fixing $T^* = R_m$ solves this issue.

Queue beliefs. For ease of discussion we define "probe interval" as an interval that leads to increase in $C_{L,\lambda}$ using Eq. 6.2. We show in Appendix E that probe intervals needs to be start

[4] Other intervals may produce a tighter Eq. 6.3 but we over-approximate.



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

Figure 8: Belief set for CBR-delay (left). Queue buildup as a function of C (right). L_1, L_2 lines are the belief constraints.

with a drained queue, otherwise, the CCA risks losing all packets in the queue at the beginning of the probe interval. For the CCA to gauge the queue state, we add q_U as an upper bound on the bottleneck queue bytes. We compute it as: (bytes already in queue) + (enqueued bytes) − (dequeued bytes on smallest C). I.e., $q_U(t + \delta t) = \max(0, q_U(t) + (\lambda(t) - C_{L,\lambda}(t))\delta t)$.

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

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

Progress. The probe builds queues. On the network with rate C, the queue build up is (enqueued bytes) − (dequeued bytes) $= C_{L,\lambda} \cdot (T^* + D) + \alpha - C \cdot T^*$ ("Prb 1" illustrates this in Fig. 8). On the network with rate $C_{L,\lambda}$, the queue build up is $C_{L,\lambda}D + \alpha$ (i.e., more than D seconds). If the CCA does not observe $qdel > D$, then it knows that C is higher than $C_{L,\lambda}$, and the line L_1 (Fig. 8) moves up. I.e., $C_{L,\lambda}$ becomes $\frac{C_{L,\lambda} \cdot (T^* + D) + \alpha}{T^* + D} = C_{L,\lambda} + \frac{\alpha}{T^* + D} > C_{L,\lambda}$. Likewise, if the CCA does not observe loss, then L_2 shifts right. If the CCA does observe $qdel > D$ or loss, then it knows that it sent above C and obtains an (implicit) upper bound on C, ensuring a lower bound on utilization.

Loss. Loss due to the probe is ΔL = (bytes already in queue) + (queue buildup) − (buffer) $= q + q_B - \beta$. In Fig. 8, the horizontal distance between a point (β, C) in the $\boxed{\text{belief set}}$ and the line Prb 1 shows $q_B - \beta$. This distance, i.e., "$C_{L,\lambda} \cdot (T^* + D) + \alpha - CT^* - \beta$" is at most α (from Eq. 6.3). Bytes already in the queue is also at most α as probe only happens when $(q \leq)$ $q_U \leq \alpha$. Hence, $\Delta L \leq 2\alpha$, i.e., constant independent of C.

Also notice, if we probe over a shorter interval $T < T^*$, then there are networks (β, C) for which $q_B - \beta$ is larger (Prb 2 in Fig. 8) and loss on those networks is $O(C)$. To avoid large loss, the probing interval T needs to be at least as long as the past measurement interval T^* (Appendix E formally proves this).

`cc_probe_slow` steady-state behavior. In steady state, `cc_probe_slow` follows a "Probe, Drain$_1$, Drain$_2$" cycle. Table 3 shows the result of this cycle on two paths in the belief set. These lie on the corners of belief set on the line L_2 (Fig. 8).

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

Table 3: Steady-state behavior of `cc_probe_slow` on two extreme paths in the belief set: $P_1 = \langle C = C_{L,\lambda}, \beta = C_{L,\lambda}D\rangle$ and $P_1 = \langle C = C_{L,\lambda}\frac{T^*+D}{T^*}, \beta = 0\rangle$. "Serviced" shows bytes serviced by the CBR box, and q_U is computed at the end of the probe/drain duration. Note, for drain$_1$, the CCA sends packets due to line 7 of the template (Listing 1), even though `rate` $= C_{L,\lambda} - q_U/R_m < 0$.

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

If we replace α with $f(C_{L,\lambda})$, where $f(.)$ is the loss allowance (§6.2). Then the resulting family of CCAs (parameterized by $f(.)$ and T^*) allows us to tune the tradeoff between loss vs. convergence time vs. utilization. We can even adapt these over time (Appendix H).

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

6.3 Proofs of performance

Ensuring the under-specified synthesis invariant is not a sufficient proof (§5.2). We summarize the lemmas that serve as the proof of performance for CCA1 (`cc_qdel`) and CCA4 (`cc_probe_slow`). These represent CCAs designed for deep and shallow buffers respectively. We give the full list and encoding of lemmas in Appendix F. Note, we describe theoretical worst-case bounds. Empirical performance is better (§6.4).

`cc_qdel`. On a CCAC link with parameters $\langle C, R_m, D = R_m, \beta \ge 3C\cdot(R_m+D)\rangle$, `cc_qdel` ensures that beliefs become consistent exponentially fast, they converge exponentially fast, and it drains queue at a rate proportional to C.[5] After beliefs are consistent, converged, and the queue is drained, the CCA is in steady state and remains in steady state. Note, this is despite the periodic beliefs timeouts (§5.1.2), i.e., the beliefs remain close to each other even after timing out. In steady state (state IV in §5.2), `cc_qdel` gets at least 89% utilization, keeps $RTT \le 4.4(R_m+D)$ seconds, and loses at most 3 packets in any R_m duration. Additionally, `cc_qdel` never incurs large loss events when probing for bandwidth as long as the beliefs are consistent, i.e., in states II, III, IV.

`cc_probe_slow`. On the CBR-delay model with parameters $\langle C, R_m, D, \beta\rangle$, `cc_probe_slow` ensures that beliefs ($C_{L,\lambda}$, q_U) become consistent exponentially fast, and $C_{L,\lambda}$ converges additively. I.e., when the link rate decreases, `cc_probe_slow` ramps down exponentially fast. When the link rate increases, `cc_probe_slow` ramps up additively. In steady state, `cc_probe_slow` ensures at least 30% utilization, keeps $RTT \le 1.5(R_m + D)$. It ensures that it loses at most 2 packets in any R_m duration whenever beliefs are consistent and the link rate has not decreased. Note, in §6.2.1, we showed steady-state utilization $\ge \frac{T^*}{T^*+D}100\% = 50\%$ (for $T^* = R_m = D$). The proved worst-case utilization is lower because $C_{L,\lambda}$ may reduce on timeouts.

6.4 Empirical evaluation

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

Implementation. We implement CCA1 (`cc_qdel`) and CCA4 (`cc_probe_slow_k`, see §6.2.1 for "`_k`") over UDP using [54]. For `cc_probe_slow_k`, we run `cc_qdel` until a large loss event (resembling TCP slow start), and set $\alpha = 5MSS$ instead of $1MSS$ to account for false-negatives in loss detection. As a result, probes may lose $2\alpha = 10MSS$ bytes.

We initially implemented the CCAs in the Linux kernel, but found bugs in kernel's pacing implementation and the `cong_control` API to be insufficient (Appendix G). We compare against Cubic, BBRv1 (Linux kernel v5.4.0), BBRv2 [27], BBRv3 [28], and Copa [8, 54].

Scenarios and metrics. We use iperf to generate traffic and mahimahi [48] to emulate scenarios with jitter and shallow buffers (§2) with the aim of validating our performance proofs. We measure utilization, delay, loss, and convergence time metrics under a variety of parameter $\langle C, R_m, D, \beta\rangle$ choices. Each tuple constitutes a different "run". We emulate jitter by injecting

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

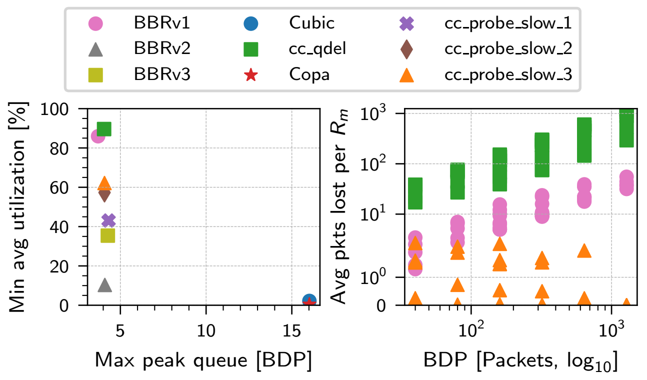


Figure 9: Summary of utilization, queue, and loss. Right subplot shows runs with $\beta \leq BDP$. Additionally, for visibility, we only show three CCAs, and use symlog scale on y-axis.

up to $D = R_m$ seconds of *uniformly random* delay, while ensuring FIFO service and constant average link rate (Appendix H).

Results. Fig. 9 summarizes utilization, queue (proxy for delay), and losses across runs (e.g., each run gives us an average utilization, and we compute the minimum across runs). Appendix H shows the metrics for each run and also studies convergence time and fairness. The synthesized CCAs are within their proven performance bounds and achieve tradeoffs between loss, convergence, utilization, and delay that prior CCAs cannot achieve.

`cc_probe_slow` meets the proven lower bound on utilization, upper bound on *RTT*s, and incurs at most constant loss (independent of BDP) across buffer sizes, bandwidths and propagation delays. On the same networks, Copa starves as it is not robust to jitter [7]. BBRv1 is able to ensure utilization despite random jitter, but incurs excessive losses (that increase with the *BDP*) on shallow buffers. Cubic gets low utilization when buffers are short ($\beta \ll BDP$) and also bloats queues when buffers are large ($\beta \gg BDP$). `cc_qdel`'s performance is similar to BBRv1, but with provable guarantees on utilization even with worst-case jitter. Unlike BBRv1, the v2 and v3 variants do not incur high losses on average but incur $O(BDP)$ losses to converge exponentially fast when the link rate increases (Appendix H). They also get lower utilization than the v1 variant. `cc_probe_slow_k` gets higher utilization at the cost of higher convergence time (Appendix H) with increasing `k`.

7 Related work

Our work extends [2], which introduced the concept of CCA synthesis using CEGIS. We propose the belief framework to make such synthesis far more practical and powerful. Related work not covered in §2 can be classified into:

Automatic CCA design. Past works have explored online learning [20], reinforcement learning [36, 43, 53], model predictive control (MPC) [32, 35], and (partially observable) markov decision process (POMDP) formulations [56]. CCAs produced by these works are not human-interpretable or are not explicitly designed for adversarial network behaviors. CCmatic CCAs are modular, human-interpretable, and provably robust under adversarial network behaviors.

Reasoning about CCAs. Past works use different network models: (1) deterministic (e.g. fluid model) [18, 62], (2) stochastic [30], and (3) non-deterministic [6]. Some also limit their scope to a small class of CCAs [9, 18, 62]. Deterministic models are easier to reason about but may not accurately reflect real world behaviors. With stochastic models, it is hard to deduce probability distributions that characterize real networks. Non-deterministic models do not require a distribution but may be too adversarial. Beliefs can be computed regardless of the modeling choice, and facilitate reasoning about complicated network models across all possible CCAs. Beliefs and CCmatic add to the emerging toolkit for performance reasoning.

8 Discussion, limitations, and future work

We built the belief framework allowing us to both (1) build novel CCAs and (2) prove tradeoffs between objectives. Using this, we built CCmatic to automatically synthesize CCAs for different environment/objective combinations, alleviating humans from figuring out complex details like when/how long to probe/drain. CCmatic also gives insights when objectives are infeasible. Due to formal methods and program synthesis, CCmatic CCAs are human-interpretable and provably performant.

While our work makes significant progress, it has several limitations that, if addressed, would bring us closer to "solving congestion control". First, we focus on single-flow scenarios. Designing provably fair CCAs will likely require a contract between CCAs allowing them to disambiguate effects of jitter from the actions of other flows. For instance, (1) flows could agree on a mapping between delay (or delay variation) and their sending rates [6], or (2) flows could agree on how much they can increase or decrease their sending rate in a single *RTT* ([18], Appendix H). Second, the CCAs synthesized for adversarial noise perform reasonably on ideal links. This may not hold as we explore other scenarios/objectives, and we may require extensions for average-case analysis. Third, we only explore a subset of belief-based CCAs due to computational limits. We hope to use techniques like robust adversarial reinforcement learning [50] and minimax [44] to improve design space exploration, given our 2-player game formulation in Theorem 4.1.

We believe our methodology (i.e., inverting environment models to build beliefs) is applicable to other domains where environment models exist. E.g., adaptive bitrate (ABR) algorithms and scheduling. ABR shares similar environments as CCAs, and recent work has built models for scheduling [5, 26].

Acknowledgments

We would like to thank anonymous reviewers and our shepherd Philip Brighten Godfrey for feedback that helped improve our paper. This work was supported in part by NSF grants CNS-2212102, and CNS-2212390.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 270–288.

[2] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. "Automating Network Heuristic Design and Analysis". In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. HotNets '22. Austin, Texas: Association for Computing Machinery, 2022, pp. 8–16.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: Association for Computing Machinery, 2010, pp. 63–74.

[4] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. "Sizing Router Buffers". In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 281–292.

[5] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. "Formal Methods for Network Performance Analysis". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 645–661.

[6] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. "Starvation in End-to-End Congestion Control". In: *Proceedings of the 2022 ACM SIGCOMM 2022 Conference*. SIGCOMM '22. Amsterdam, Netherlands: Association for Computing Machinery, 2022.

[7] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. "Toward Formally Verifying Congestion Control Behavior". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1–16.

[8] Venkat Arun and Hari Balakrishnan. "Copa: Practical Delay-Based Congestion Control for the Internet". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 329–342.

[9] D. Bansal and H. Balakrishnan. "Binomial congestion control algorithms". In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*. Vol. 2. 2001, 631–640 vol.2.

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

[11] Patricia Bouyer, Stéphane Le Roux, Youssouf Oualhadj, Mickael Randour, and Pierre Vandenhove. "Games where you can play optimally with arena-independent finite memory". In: *Logical Methods in Computer Science* 18 (2022).

[12] L.S. Brakmo and L.L. Peterson. "TCP Vegas: end to end congestion avoidance on a global Internet". In: *IEEE Journal on Selected Areas in Communications* 13.8 (1995), pp. 1465–1480.

[13] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. "How I Learned to Stop Worrying About CCA Contention". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. HotNets '23. Cambridge, MA, USA: Association for Computing Machinery, 2023, pp. 229–237.

[14] Carlo Caini and Rosario Firrincieli. "TCP Hybla: A TCP Enhancement for Heterogeneous Networks". In: *Int. J. Satell. Commun. Netw.* 22.5 (Sept. 2004), pp. 547–566.

[15] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20–53.

[16] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, and Van Jacobson. *BBR Congestion Control Work at Google IETF 101 Update*. [Online; accessed 4. Mar. 2024]. Slides 4, 17. URL: https://datatracker.ietf.org/meeting/101/materials/slides-101-iccrg-an-update-on-bbr-work-at-google-00.

[17] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Ian Swett, and Van Jacobson. *BBR Congestion Control*. [Online; accessed 26. Jun. 2023]. Mar. 2022. URL: https://datatracker.ietf.org/doc/html/draft-cardwell-iccrg-bbr-congestion-control-02.

[18] Dah-Ming Chiu and Raj Jain. "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks". In: *Computer Networks and ISDN Systems* 17.1 (1989), pp. 1–14.

[19] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design* 19.1 (July 2001), pp. 7–34.

[20] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. "PCC: Re-architecting congestion control for consistent high performance". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 395–408.

[21] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. "PCC Vivace: Online-Learning Congestion Control". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 343–356.

[22] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. "What do packet dispersion techniques measure?" In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 2. IEEE. 2001, pp. 905–914.

[23] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden. "Routers with Very Small Buffers". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–11.

[24] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. "An Internet-Wide Analysis of Traffic Policing". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 468–482.

[25] Hugo Gimbert and Wiesław Zielonka. "Games Where You Can Play Optimally without Any Memory". In: *CONCUR 2005 - Concurrency Theory*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 428–442.

[26] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. *Quantitative Verification of Scheduling Heuristics*. 2023. arXiv: 2301.04205 [cs.LO].

[27] Google. *BBRv2*. [Online; accessed 1. Aug. 2023]. URL: https://github.com/google/bbr/tree/v2alpha.

[28] Google. *BBRv3*. [Online; accessed 14. Sep. 2023]. URL: https://github.com/google/bbr/tree/v3.

[29] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. "ABC: A Simple Explicit Congestion Controller for Wireless Networks". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 353–372.

[30] Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. *Optimal Congestion Control for Time-varying Wireless Links*. 2022. arXiv: 2202.04321 [cs.NI].

[31] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-Friendly High-Speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74.

[32] Cunwu Han, Dehui Sun, Lei Liu, and Song Bi. "A new robust model predictive congestion control". In: *Proceeding of the 11th World Congress on Intelligent Control and Automation*. IEEE. 2014, pp. 4189–4193.

[33] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. *RFC 6582: The NewReno Modification to TCP's Fast Recovery Algorithm*. [Online; accessed 15. Aug. 2023]. Apr. 2012. URL: https://datatracker.ietf.org/doc/html/rfc6582.

[34] Janey C. Hoe. "Improving the Start-up Behavior of a Congestion Control Scheme for TCP". In: *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '96. Palo Alto, California, USA: Association for Computing Machinery, 1996, pp. 270–280.

[35] Amjad J Humaid, Hamid M Hasan, and Firas A Raheem. "Development of model predictive controller for congestion control problem". In: *feedback* 2 (2014), p. 3.

[36] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. "A deep reinforcement learning perspective on internet congestion control". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3050–3059.

[37] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial intelligence* 101.1-2 (1998), pp. 99–134.

[38] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. "Learning SMT(LRA) Constraints Using SMT Solvers". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. IJCAI'18. Stockholm, Sweden: AAAI Press, July 2018, pp. 2333–2340.

[39] K. Lai and M. Baker. "Measuring bandwidth". In: *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*. Vol. 1. 1999, 235–245 vol.1.

[40] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik

Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196.

[41] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[42] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Vol. 2050. Springer Science & Business Media, 2001.

[43] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. “QTCP: Adaptive congestion control with reinforcement learning”. In: *IEEE Transactions on Network Science and Engineering* 6.3 (2018), pp. 445–458.

[44] G.F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Pearson Addison-Wesley, 2009.

[45] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. “The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm”. In: *SIGCOMM Comput. Commun. Rev.* 27.3 (July 1997), pp. 67–82.

[46] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. “PCC proteus: Scavenger transport and beyond”. In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 615–631.

[47] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[48] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. “Mahimahi: Accurate Record-and-Replay for HTTP”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 417–429.

[49] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. “Modeling TCP Throughput: A Simple Model and Its Empirical Validation”. In: *Proceedings of the ACM SIGCOMM ’98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 303–314.

[50] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. “Robust Adversarial Reinforcement Learning”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, Aug. 2017, pp. 2817–2826.

[51] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. “Sketching Concurrent Data Structures”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 136–148.

[52] K. Tan, J. Song, Q. Zhang, and M. Sridharan. “A Compound TCP Approach for High-Speed and Long Distance Networks”. In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–12.

[53] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. “Reinforcement Learning for Datacenter Congestion Control”. In: *SIGMETRICS Perform. Eval. Rev.* 49.2 (Jan. 2022), pp. 43–46.

[54] venkatarun95. *genericCC*. [Online; accessed 9. Jun. 2023]. URL: https://github.com/venkatarun95/genericCC.

[55] Curtis Villamizar and Cheng Song. “High performance TCP in ANSNET”. In: *ACM SIGCOMM Computer Communication Review* 24.5 (1994), pp. 45–60.

[56] Keith Winstein and Hari Balakrishnan. “TCP Ex Machina: Computer-Generated Congestion Control”. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM ’13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 123–134.

[57] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. “Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks”. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 459–471.

[58] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. “Boosting SMT solver performance on mixed-bitwise-arithmetic expressions”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 651–664.

[59] Lisong Xu, K. Harfoush, and Injong Rhee. “Binary increase congestion control (BIC) for fast long-distance

networks". In: *IEEE INFOCOM 2004*. Vol. 4. 2004, 2514–2524 vol.4.

[60] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. "Pantheon: the training ground for Internet congestion-control research". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 731–743.

[61] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. "Adaptive Congestion Control for Unpredictable Cellular Networks". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 509–522.

[62] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. "Axiomatizing Congestion Control". In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.2 (2019), 33:1–33:33.

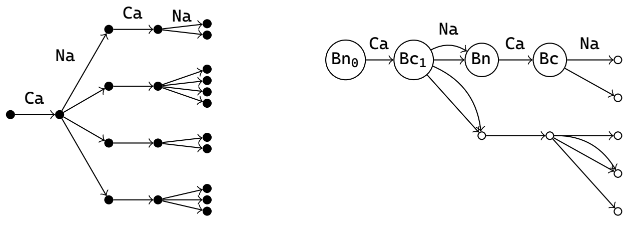
[63] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. "Congestion Control for Large-Scale RDMA Deployments". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 523–536.

A Beliefs are sufficient

LEMMA A.1. *The set of feasible actions that the network is allowed to take in the future can be determined by the belief set.*

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

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



(a) Game tree $\mathcal{T}$. **(b)** Condensed graph $\mathcal{G}$.

Figure 10: Constructing condensed graph from game tree.

LEMMA A.2. *Belief set for a future time can be computed by the belief set at the current time and the CCA's observations between the current and future times.*

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

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

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

Preliminaries. An execution (or trace) is a sequence of CCA actions ($\mathtt{Ca_i}$) and network actions ($\mathtt{Na_i}$), e.g., $\langle \mathtt{Ca_1}, \mathtt{Na_1}, \mathtt{Ca_2}, \mathtt{Na_2}, \ldots \rangle$. We say that an execution is valid if it conforms to the rules of the game, i.e., the sequence of network actions correspond to some path in the network model.

We annotate each step of the execution by the beliefs computed over the entire history until that step, e.g., the belief set is $\mathtt{Bc_i}$ after $\mathtt{Ca_i}$, and $\mathtt{Bn_i}$ after $\mathtt{Na_i}$. At the start of all traces, the belief set is $\mathtt{Bn_0}$. The annotation looks like: $\langle |_{\mathtt{Bn_0}} \mathtt{Ca_1} |_{\mathtt{Bc_1}} \mathtt{Na_1} |_{\mathtt{Bn_1}} \mathtt{Ca_2} |_{\mathtt{Bc_2}} \mathtt{Na_2} |_{\mathtt{Bn_2}} \ldots \rangle$.

The performance property is defined as a function $\mathcal{P}_f$ that maps $\langle$Belief, CCA action$\rangle$ (e.g., $\langle \mathtt{Bn}, \mathtt{Ca} \rangle$) to `True` or `False`. While in §5.2 and §6.3, we do not directly state the performance properties this way, we show that they can be defined this way at the end of the proof.

Construction. The high level summary of the construction is that $\mathcal{A}_\mathcal{B}$ can arbitrarily pick one CCA action (sending rate choice) whenever $\mathcal{A}_\mathcal{D}$ takes different actions on the same belief set. As a result, $\mathcal{A}_\mathcal{B}$ is a pure function of belief set (it does not take different actions on the same belief). First, we will show that $\mathcal{A}_\mathcal{B}$ function is well-defined i.e., it assigns a rate choice to each belief $\texttt{Bn}$ that it can witness in an execution. We will show this is because $\mathcal{A}_\mathcal{B}$ only ever reaches a belief set $\texttt{Bn}$ if $\mathcal{A}_\mathcal{D}$ reached it. We will use Lemma A.2 and Lemma A.1 to show this. Second, we will show that $\mathcal{A}_\mathcal{B}$ satisfies the performance property because it only takes an action $\texttt{Ca}$ on $\texttt{Bn}$ if $\mathcal{A}_\mathcal{D}$ took it and so $\mathcal{P}_f(\langle \texttt{Bn}, \texttt{Ca} \rangle)$ evaluates to true for $\mathcal{A}_\mathcal{B}$ as it evaluated true for $\mathcal{A}_\mathcal{D}$.

We consider the set of all executions of $\mathcal{A}_\mathcal{D}$ on the given network model. We visualize them in the form of a (directed) game tree $\mathcal{T}$ (Fig. 10a). The nodes in $\mathcal{T}$ are placeholders. A unique node includes the entire history of the CCA and network actions until the node. If it is the turn of the CCA to play, the node has an outgoing edge labelled with the CCA action $\texttt{Ca}$. If it is the network's turn to play, the node has outgoing edges corresponding to all feasible network actions, each labelled with a network action $\texttt{Na}$. We label each node in the tree with the belief set computed over the history of actions until the node. We get two kinds of labels, (1) beliefs after CCA action ($\texttt{Bc}$), and (2) beliefs after network action ($\texttt{Bn}$).

From the tree, we construct a condensed graph $\mathcal{G}$ (Fig. 10b) as follows. Within each label type, we merge the nodes that have the same belief value (i.e., if two nodes have labels $\texttt{Bn}^I$, $\texttt{Bn}^{II}$, with $\texttt{Bn}^I = \texttt{Bn}^{II}$, we coalesce the nodes). Likewise for $\texttt{Bc}^I = \texttt{Bc}^{II}$. Note, we do not merge nodes with different label types even if they have the same belief value. Then for each node with a $\texttt{Bn}$ label type (i.e., CCA's turn to play), we just keep one outgoing edge (arbitrarily) resembling the CCA action that $\mathcal{A}_\mathcal{B}$ takes on that belief value.

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

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

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

Case 1. We know $\texttt{Bn}_\texttt{i} \neq \texttt{Bn}_0$ because $\texttt{Bn}_0$ is in $\mathcal{G}$ (it is the root node of $\mathcal{T}$, and we do not remove any nodes when constructing $\mathcal{G}$). As a result, $\texttt{e}$ looks like $\langle \cdots |_{\texttt{Bn}_\texttt{i-1}} \texttt{Ca}_\texttt{i} |_{\texttt{Bc}_\texttt{i}} \texttt{Na}_\texttt{i} |_{\texttt{Bn}_\texttt{i}} \rangle$, where, $\texttt{Bn}_\texttt{i-1}$ is in $\mathcal{G}$.

Since $\mathcal{A}_\mathcal{B}$ produced $\texttt{e}$, $\mathcal{A}_\mathcal{B}$ took action $\texttt{Ca}_\texttt{i}$ on the belief $\texttt{Bn}_\texttt{i-1}$, i.e., $\texttt{Ca}_\texttt{i}$ is the outgoing edge for node $\texttt{Bn}_\texttt{i-1}$ in $\mathcal{G}$. $\mathcal{A}_\mathcal{B}$ only takes this action, if $\mathcal{A}_\mathcal{D}$ took this action on a node with label $\texttt{Bn}_\texttt{i-1}$. We argue that in $\mathcal{T}$, this action leads to a node with label $\texttt{Bc}_\texttt{i}$. This comes from Lemma A.2, $\mathcal{A}_\mathcal{D}$ could have arrived from any history at the belief $\texttt{Bn}_\texttt{i-1}$, but taking the action $\texttt{Ca}_\texttt{i}$ on $\texttt{Bn}_\texttt{i-1}$ updates the belief to $\texttt{Bc}_\texttt{i}$ no matter the history. Thus, from our construction, $\texttt{Bc}_\texttt{i}$ also exists in $\mathcal{G}$ and is the destination node on the edge $\texttt{Ca}_\texttt{i}$ (as we do not remove nodes in the construction and the destination node labels are same in $\mathcal{G}$ and $\mathcal{T}$ for each edge).

Now, we argue that in $\mathcal{T}$, $\texttt{Na}_\texttt{i}$ is an outgoing edge for the node with label $\texttt{Bc}_\texttt{i}$. This is because according to $\texttt{e}$, $\texttt{Na}_\texttt{i}$ is a feasible network action on the belief $\texttt{Bc}_\texttt{i}$, as a result $\texttt{Na}_\texttt{i}$ is a feasible action on the node with label $\texttt{Bc}_\texttt{i}$ independent of history in $\mathcal{T}$ that led to this node (from Lemma A.1). Since the game tree describes *all* executions of $\mathcal{A}_\mathcal{D}$, each node with label type $\texttt{Bc}$ has an edge for each feasible network action, so $\mathcal{T}$ has an edge with action $\texttt{Na}_\texttt{i}$ on the node with label $\texttt{Bc}_\texttt{i}$. Again from Lemma A.2, $\texttt{Na}_\texttt{i}$ takes belief from $\texttt{Bc}_\texttt{i}$ to $\texttt{Bn}_\texttt{i}$ in $\mathcal{T}$.

Since $\texttt{Bc}_\texttt{i}$, $\texttt{Na}_\texttt{i}$ and $\texttt{Bn}_\texttt{i}$ exist in $\mathcal{T}$, they also exist in $\mathcal{G}$. And in $\mathcal{G}$, $\texttt{Na}_\texttt{i}$ points from $\texttt{Bc}_\texttt{i}$ to $\texttt{Bn}_\texttt{i}$, and $\texttt{Ca}_\texttt{i}$ points from $\texttt{Bn}_\texttt{i-1}$ to $\texttt{Bc}_\texttt{i}$. This is because we never remove any edges corresponding to network actions, nor do we remove any nodes, and the destination nodes have same labels in $\mathcal{T}$ and $\mathcal{G}$. Since $\texttt{Bn}_\texttt{i}$ exists in $\mathcal{G}$, we arrive at a contradiction.

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

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

□

Performance properties in §5.2 and §6.3 can be expressed as a boolean valued function of belief set and CCA's action on that belief set. At a high level, our performance properties dictate that the CCA (1) ensures some bounds on metrics like loss, delays, and utilization, and (2) makes progress. We show how both these can be expressed as a function of belief and CCA action.

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

`False` on the ⟨belief,CCA action⟩ pair, otherwise the property evaluates to `True`.

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

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

Also, note that we cannot express our performance properties solely as a function of the belief set. For example, the belief set does not tell us if the CCA will cause loss on an action, and we need to know the CCA's action on a belief set to evaluate loss. This is different from chess where the winning condition is just a function of the board state. Thus, Theorem 4.1 is non-obvious [11, 25].

B Computing beliefs

B.1 Propagation delay and jitter

We assume that the CCA knows R_m and D. Due to discretization of time in the SMT encoding, all quantities with units of time are integer multiples of the discretization interval. The time for synthesis and verification scales with the number of intervals considered (§6.1). To cover as many *RTT*s in as few intervals as possible, we set $R_m = D = 1$ interval. As a result the constant value 1 reveals R_m and D in the templates.

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

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

Listing 3: CCAC constraints on S

$$S(t) \leq T_A(t) = S_U(t) \quad \text{(B.1)}$$
$$S(t) \leq A(t) - L(t) \quad \text{(B.2)}$$
$$S(t) \geq T_A(t-D) = S_L(t) \quad \text{where,} \quad \text{(B.3)}$$
$$T_A(t) = C\cdot t - W(t) \quad \text{(B.4)}$$
$$q(t) \geq 0 \implies W'(t) = 0 \quad \text{(B.5)}$$
$$q(t) = A(t) - L(t) - T_A(t) \quad \text{(B.6)}$$

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

B.2 Bandwidth

We show the analytical derivation of the link rate belief bounds (C_L and C_U) for the CCAC model. Before that we briefly give relevant background on CCAC (for detailed background please see [7]).

Background on CCAC. CCAC models the network as a generalized token bucket filter. It puts constraints (Listing 3) on how the network serves packets (i.e., $S(t)$, service curve, or cumulative bytes serviced by time t), based on how packets arrive into the network (i.e., $A(t)$, cumulative bytes arrived by time t).

Tokens arrive at rate C bytes/secs, i.e., $C\cdot t$ tokens arrive by time t. On a token arrival, the network decides whether to admit it or waste it (Eq. B.4). The network uses admitted tokens to send packets, so the packets sent are upper bounded by the number of admitted tokens (Eq. B.1). Also, the network cannot send more bytes than arrived but not lost (Eq. B.2).

To emulate non-congestive jitter, the network can choose to delay sending packets even when tokens are available. To ensure jitter is bounded, all tokens must be used within D seconds, and the network should only admit a token if it knows it will be used within D seconds.[6] This puts a lower bound on service (Eq. B.3).

Additionally, to prevent the network from wasting all the tokens, tokens cannot be wasted when there are packets waiting for tokens, i.e., there are packets that have arrived (but not lost) and do not have corresponding admitted tokens (Eq. B.5 and Eq. B.6). These packets are put into a queue, and they build congestive delays. Other packets that have corresponding tokens but have not been delivered as considered as facing non-congestive delays.

Analytical derivation. CCAC visualizes its constraints using graphs like Fig. 11. From such graphs, we can determine the minimum and maximum average ACK rate (r) that a CCA

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

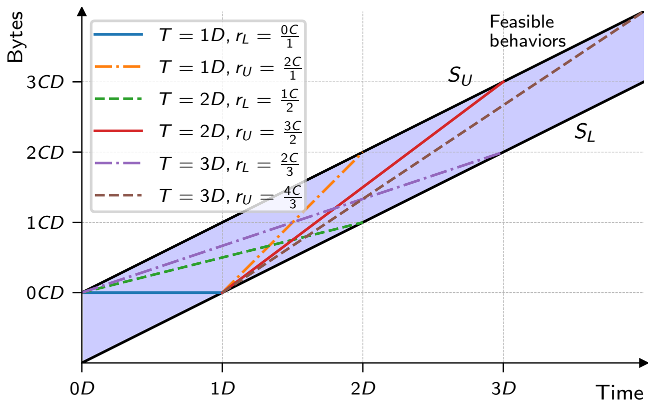


Figure 11: Computing ACK rate range based on C. ACK rate is the slope of the lines. Over different time intervals (T), the lines show feasible service curves with the maximum (r_U) and minimum (r_L) slopes. Service curves need to be non-decreasing and lie in the shaded region.

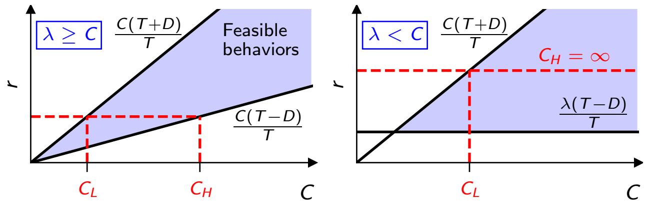


Figure 12: Inverting bounds on observations to get bounds on network parameters. For observed ACK rate, draw horizontal line (red dashed line) corresponding to the observed ACK rate r, the points intersecting with the feasible region give the range of link rates $[C_L, C_U]$ that could have produced r.

can get for different λ and C choices. Specifically, consider the case when $\lambda \geq C$. In this case, tokens cannot be wasted ($W' = 0$). As a result S_L and S_U have slope C, and the network has to ensure that at all times S is between the S_L and S_U lines. We look at different feasible choices of S, and find the maximum and minimum slope of S (ACK rate) over time intervals of different lengths. Note, we are interested in the "average" ACK rate over the intervals, so we look at the slope of the line joining the start and end points of the interval.

Fig. 11 shows different feasible S curves along with minimum and maximum slopes in different intervals. For example, over an interval of length D, $r \in [0, C]$. Similarly, $r \in [C/2, 3C/2]$ for $2D$ long intervals, $r \in [2C/3, 4C/3]$ for $3D$ long intervals. For kD long intervals, $r \in [\frac{k-1}{k}C, \frac{k+1}{k}C]$. Following a similar exercise, we find that for intervals with length T, $r \in [\frac{C(T-D)}{T}, \frac{C(T+D)}{T}]$. I.e., S follows an ideal link with rate C with an additional burst or delay of CD bytes.

We repeat this for the case when $\lambda < C$. Here, we consider two extreme cases: (C1) S_L, S_U have slope λ, and (C2) S_L, S_U have slope C. The first case corresponds to the network wasting tokens because packets arrive slower than tokens, and the second corresponds to a large queue build up at time $t = 0$ preventing the network from wasting tokens. When we inspect the graphs (not shown, similar to Fig. 11), we find that over intervals of length T, $r \in [\frac{\lambda(T-D)}{T}, \frac{\lambda(T+D)}{T}]$ in case C1, and $r \in [\frac{C(T-D)}{T}, \frac{C(T+D)}{T}]$ in case C2. When we look at other cases between C1 and C2, i.e., S_L and S_U take slopes in the range $[\lambda, C]$, we get more feasible values of r. Taking the union of all the r ranges, we find that in any T long interval, whenever $\lambda < C$, $r \in [\frac{\lambda(T-D)}{T}, \frac{C(T+D)}{T}]$.

We invert the bounds on r to get bounds on C. Fig. 12 illustrates this. Specifically, when $\lambda > C$, we have, $\forall T$:

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

On rearranging, we get:

$$\frac{r \cdot T}{T+D} \leq C \leq \frac{r \cdot T}{T-D} \tag{B.7}$$

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

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

On rearranging, we get:

$$\frac{r \cdot T}{T+D} \leq C \tag{B.8}$$

From Eq. B.7 and Eq. B.8, we get:

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

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

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

C Synthesis details

For completeness, we describe the workings of the generator and verifier in CEGIS (§5). For ease of understanding, we interpret the search (or program synthesis) problem as a $\exists\forall$ formula [1]. For instance, "does there *exist* a belief-based CCA, such that *forall* traces captured by the network model, the CCA ensures the desired performance properties". More formally, the formula is:

$$\exists \boxed{?} \forall \texttt{trace} \; (\texttt{CCA} \wedge \texttt{Network}) \implies \texttt{Performance} \tag{C.1}$$

$\boxed{?}$ and `trace` represent vector of variables. $\boxed{?}$ are the holes in the CCA template (Listing 1). Assigning value to the holes produces a concrete CCA. A `trace` (timeseries) describes the execution of the CCA under the network model. It is specified using the dimensions of the network model's relation (§3), e.g., path, feedback, CCA actions. In our case, the `trace` includes variables like C, β, R_m, D, $rate(t)$, $A(t)$, $S(t)$, $L(t)$, $T_A(t)$, $W(t)$ (Table 1, Listing 3).

The `CCA`, `Network`, and `Performance` are boolean valued SMT formulas (in the theory of linear real arithmetic (LRA) [38]) over the $\boxed{?}$ and `trace` variables. `CCA` ensures that the congestion control decisions are made according to the

CCA. It encodes that the $rate(t)$ variables are assigned using the values of $\boxed{?}$ and `trace` variables according to the CCA template. We produce this encoding by symbolically executing the CCA template. `Network` encodes what assignments to the `trace` variables result in executions that are deemed feasible according to the network model. These look like Listing 3. We also encode belief computations (§5.1.1) and timeouts (§5.1.2) in `Network`, these constraints merely populate the belief bounds and don't affect the actions that the network takes. `Performance` encodes the desired objectives using the transition system. We use the synthesis invariants (Eq. 5.3) to specify `Performance`.

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

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

$$\boxed{?} = \texttt{value of } \boxed{?} \wedge \texttt{CCA} \wedge \texttt{Network} \wedge \neg\texttt{Performance} \quad \text{(C.2)}$$

This is a quantifier-free formula (i.e., no $\exists$ or $\forall$ quantifiers). The verifier uses an SMT solver (e.g., Z3 [47]) to solve this formula. This is a formula on the `trace` variables (as the $\boxed{?}$ variables have been substituted by fixed values). The assignment to the `trace` variables describes a trace where the given CCA violates the performance properties on the network model. If the formula is unsatisfiable, then there is no counterexample that breaks the CCA.

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

$$\bigwedge_{\texttt{trace} \in \mathbb{X}} (\texttt{CCA} \wedge \texttt{Network}) \implies \texttt{Performance} \quad \text{(C.3)}$$

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

Note, in this formulation, only `CCA` depends on the holes, the `Network` and `Performance` only depend on the `trace` variables. On each `trace` in $\mathbb{X}$, `Network` evaluates to `True` and `Performance` evaluates to `False` (as the `trace` was generated by the verifier by satisfying "$\cdots \wedge$ `Network` $\wedge$ `Performance`"). As a result, Eq. C.3 simplifies to:

$$\bigwedge_{\texttt{trace} \in \mathbb{X}} \neg\texttt{CCA} \quad \text{(C.4)}$$

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

D Loss vs. convergence tradeoffs

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

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

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

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

Step 1. Since $\beta_s \le D$, the CCA must cause loss to avoid arbitrarily low utilization. This immediately follows from Theorem 2 of [6], i.e., to avoid arbitrarily low utilization, the CCA must cause more than D queueing delay. If the buffer size is $\le D$ seconds, the CCA will have to cause loss. For completeness, we repeat the proof in Appendix D.1.

Step 2. Until the time that loss happens, we compute the set of paths (link rates) that the CCA could be running on given its observations. If the CCA has not seen a loss event by time t^*, then the CCA never over-flowed the buffer until time $t^* - RTT(t^*)$. I.e., in any time interval before $t^* - RTT(t^*)$[7], the net bytes enqueued and the net bytes dequeued differ by at most β. I.e., $\forall t_1, t_2$, such that, $0 \le t_1 \le t_2 \le t^* - RTT(t^*)$:

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

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

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

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

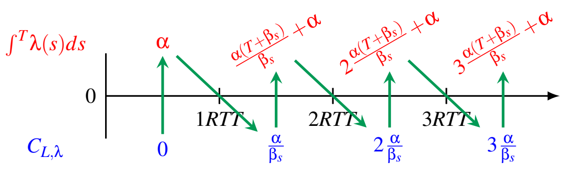

$\int^T \lambda(s)ds$
α
$\frac{\alpha(T+\beta_s)}{\beta_s}+\alpha$
$2\frac{\alpha(T+\beta_s)}{\beta_s}+\alpha$
$3\frac{\alpha(T+\beta_s)}{\beta_s}+\alpha$
0
1RTT
2RTT
3RTT
$C_{L,\lambda}$
0
$\frac{\alpha}{\beta_s}$
$2\frac{\alpha}{\beta_s}$
$3\frac{\alpha}{\beta_s}$

Figure 13: $C_{L,\lambda}$ puts constraints on $\int^T \lambda(s)ds$, and $\int^T \lambda(s)ds$ puts constraints on $C_{L,\lambda}$. Over the intervals, the red and blue values show the maximum value of $\int^T \lambda(s)ds$ and $C_{L,\lambda}$ respectively. The green arrows show the constraint dependencies. I.e., $C_{L,\lambda}$ in $[kRTT,(k+1)RTT)$ puts constraints on $\int^T \lambda(s)ds$ in $[kRTT,(k+1)RTT)$ and $\int^T \lambda(s)ds$ in $[kRTT,(k+1)RTT)$ puts constraints on $C_{L,\lambda}$ in $[(k+1)RTT,(k+2)RTT)$.

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

$$C_{L,\lambda}(t^*) = \max_{0 \le t_1 \le t_2 \le t^* - RTT(t^*)} \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + \beta_s} \tag{D.1}$$

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

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

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

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

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

Loss happens only if buffer overflows, i.e., in some interval $[t_1^*, t_2^*]$ $\int_{t_1^*}^{t_2^*} \lambda(s)ds > C \cdot (t_2^* - t_1^*) + C \cdot \beta_s$. We also know that CCA needs to ensure $\Delta L(t_1^*, t_2^*) \le \alpha$. From this, loss happens only if $C_{L,\lambda}(t_2^*) \ge C - \alpha/\beta_s$. See Appendix D.1 for details.

We compute time for $C_{L,\lambda}$ to increase to $C - \alpha/\beta_s$. Initially $C_{L,\lambda}(0) = 0$. Since $C_{L,\lambda}$ can only be computed after the first RTT (from the definition of $C_{L,\lambda}$), $C_{L,\lambda}$ is 0 over the entire interval $[0, RTT)$. This puts constraints on sending rate choices in $[0, RTT)$. Specifically, substituting $C_{L,\lambda} = 0$ in $\Delta L \le \alpha$, we get, $\int_{t_1}^{t_2} \lambda(s)ds \le \alpha$ for all intervals $[t_1, t_2]$ inside $[0, RTT)$. Thus, if we compute $C_{L,\lambda}(t)$ at any time $t \in [RTT, 2RTT)$, we get $C_{L,\lambda}(t) \le \frac{\alpha}{\beta_s}$ (because on intervals $0 \le t_1 \le t_2 < 2RTT - RTT$, i.e., $t_2 < RTT$, the numerator in Eq. D.1, "$\int_{t_1}^{t_2} \lambda(s)ds$", is at most α).

We can similarly obtain that in interval $[kRTT, (k+1)RTT)$, $C_{L,\lambda}$ is at most $k \cdot (\alpha/\beta_s)$ (illustrated in Fig. 13). For $C_{L,\lambda}$ to ramp up to $C - \alpha/\beta_s$, we need $k \ge C \cdot (\beta_s/\alpha) - 1$. Hence, the convergence time is at least k RTTs, or "$C \cdot (\beta_s/\alpha) - 1$ RTTs". Before loss, each RTT can be as large as $R_m + \beta_s$ (either due to queueing delay or jitter), making the convergence time $= (C \cdot (\beta_s/\alpha) - 1) \cdot (R_m + D) = \Omega(C(R_m + \beta_s))$.

□

Generalizing Theorem 6.1. If the CCA wants to ramp up from C_0 to C while ensuring its risks at most $f(C)$ loss, then the convergence time is $\Omega(F^{-1}(C)(R_m + \beta_s))$, where F^{-1} is the inverse of F and F is defined by the recursion:

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

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

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

D.1 Proof details

We fill in the details skipped in the proof.

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

Proof. We prove by contradiction, i.e., if the CCA does not cause $RTT > R_m + D$ or loss, then we can construct an execution where the CCA gets arbitrarily low utilization. Say the CCA produces an infinite execution with $RTT \le R_m + D$ without ever causing loss on a CBR-delay link (say link I) with bandwidth C_I and β_s seconds of buffering. This exact RTT sequence can be produced by another CBR-delay link (say link II) with rate $C_{II} \gg C_I$ (see construction below). Since the CCA is deterministic, and it gets same sequence of RTTs as feedback on both links, it will make the same sending rate choices on both the links.[8] However, the CCA's average sending rate is $\approx C_I$ (as it does not build large queues on link I given that its RTTs are at most $R_m + D$). Such sending rate gets arbitrarily low utilization on link II for arbitrarily large C_{II}.

Construction. Link II can produce same RTT sequence as link I, by choosing $\texttt{jitter}_{II}(t) = RTT_I(t) - R_m - qdel_{II}(t)$ (we use subscripts I and II to refer to quantities on link I and II respectively). With this choice $RTT_{II}(t) = R_m + qdel_{II}(t) +$

[8]Note RTTs capture all the information that a CCA can obtain from feedback. Metrics like loss, ACK-rate, etc. can be derived from packet send events and RTT sequence [6].

$\texttt{jitter}_{II}(t) = RTT_I(t)$. We just need to show that this is a feasible choice for $\texttt{jitter}_{II}$, i.e., $0 \le \texttt{jitter}_{II}(t) \le D$.

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

We know $RTT_I(t) \le R_m + D$, or $RTT_I(t) - R_m \le D$,

$$\begin{aligned} \text{thus } \texttt{jitter}_{II}(t) &= RTT_I(t) - R_m - qdel_{II}(t) \\ &\le D - qdel_{II}(t) \le D \\ &(\text{as } qdel_{II}(t) \ge 0 \text{ by definition of } qdel) \end{aligned}$$

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

Since $C_{II} \gg C_I$, $qdel_{II}(t) \le qdel_I(t)$

(increasing C decreases congestive queueing delays)

$$\begin{aligned} \text{As a result, } RTT_I(t) &= R_m + qdel_I(t) + \texttt{jitter}_I(t) \\ &\ge R_m + qdel_{II}(t) + \texttt{jitter}_I(t) \end{aligned}$$

Or, $RTT_I(t) - R_m - qdel_{II}(t) \ge \texttt{jitter}_I(t) \ge 0$

Thus, $\texttt{jitter}_{II}(t) = RTT_I(t) - R_m - qdel_{II}(t) \ge 0$

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

$$\int_{t_1^*}^{t_2^*} \lambda(s)ds > C \cdot (t_2^* - t_1^*) + C \cdot \beta_s \tag{D.2}$$

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

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

$$\text{or, } \int_{t_1^*}^{t_2^*} \lambda(s)ds \le C_{L,\lambda}(t_2^*) \cdot (t_2^* - t_1^* + \beta_s) + \alpha \tag{D.3}$$

We can only meet both these constraints (i.e., Eq. D.2 and Eq. D.3) when:

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

E Synthesizing `cc_probe_slow`

We describe properties of probe intervals used in §6.2.1 to inform belief computation and encoding. Recall, probe intervals are intervals that lead to increase in $C_{L,\lambda}$ through the equation:

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

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

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

$$\Delta L = \int^{T^f} \lambda(s)ds - \left(C \cdot T^f + \beta\right) \tag{E.1}$$

We make the following substitutions in Eq. E.1:

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

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

(3) From the past probe, we know $\int^{T^p} \lambda(s)ds \le C \cdot T^p + \beta$ (as loss did not happen, Eq. 6.3), and $\int^{T^p} \lambda(s)ds = C_{L,\lambda}^p \cdot (T^p + D)$ (from the definition of $C_{L,\lambda}$). From these two inequalities, we get $C_{L,\lambda}^p \cdot (T^p + D) \le C \cdot T^p + \beta$.
We substitute $C \cdot T^p + \beta = C_{L,\lambda}^p \cdot (T^p + D)$, or $C_{L,\lambda}^p = \frac{C \cdot T^p + \beta}{T^p + D}$, to evaluate how much loss would happen on this network.

On making the substitutions, and algebraic simplifications, we get:

$$\Delta L = \frac{(CD - \beta) \cdot (T^p - T^f)}{T^p + D} + \varepsilon \cdot (T^f + D) \tag{E.2}$$

For $T^f < T^p$, ΔL can be an increasing function of C as long as $\beta < CD$, e.g., $\beta = C\beta_s$ for $\beta_s < D$. Hence, loss is not bounded by a constant independent of C.

A probe interval must start with drained queue. Say $[t_1, t_2]$ is a probing interval, i.e., it leads to an increase in $C_{L,\lambda}$, and it does not start with a drained bottleneck queue; then we will show that the probe interval risks losing all the packets in the bottleneck queue. As a result, to ensure losses are bounded, a CCA needs to ensure that the bottleneck queue is bounded at the beginning of the probe interval.

Say the $[t_1, t_2]$ probe interval causes an increase in $C_{L,\lambda}$ at time t, i.e., $0 \le t_1 \le t_2 \le t - RTT(t)$, and,

$$C_{L,\lambda}(t) = \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} > C_{L,\lambda}(t - \varepsilon) \tag{E.3}$$

for some small $\varepsilon > 0$. If this is not true, then $[t_1, t_2]$ is not a probing interval. By definition, $C_{L,\lambda}$ can only increase over time, so $C_{L,\lambda}(t - \varepsilon) \ge C_{L,\lambda}(t_2)$. Using this and Eq. E.3, we get, $\frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} > C_{L,\lambda}(t_2)$, or,

$$\int_{t_1}^{t_2} \lambda(s)ds - C_{L,\lambda}(t_2) \cdot (t_2 - t_1 + D) > 0 \tag{E.4}$$

The risk of loss during the probe is: ΔL = (bytes already in queue) + (enqueued bytes) − (dequeued bytes+buffer). We evaluate this equation on the path $\langle C = C_{L,\lambda}(t_2), \beta = C_{L,\lambda}(t_2)D \rangle$. It can produce observations till time t_2 as it is in the belief set (derived in §6.2.1), assuming until t_2, the CCA has not witnessed $RTT > R_m + D$, or losses. Plugging this path into ΔL, we get:

$$\Delta L = q(t_1) + \int_{t_1}^{t_2} \lambda(s)ds - \left(C_{L,\lambda}(t_2) \cdot (t_2 - t_1) + C_{L,\lambda}(t_2) \cdot D\right) \tag{E.5}$$

From Eq. E.5 and Eq. E.4, we get $\Delta L > q(t_1)$.

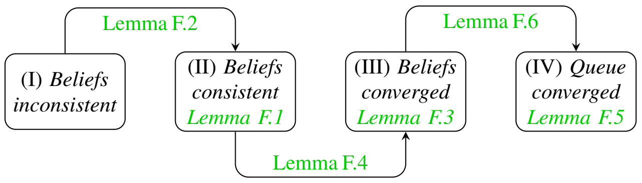

Lemma F.2
Lemma F.6
(I) Beliefs inconsistent
(II) Beliefs consistent Lemma F.1
(III) Beliefs converged Lemma F.3
(IV) Queue converged Lemma F.5
Lemma F.4

Figure 14: State transition lifetime of `cc_qdel`. Note, a link rate change can cause a transition between any two states (these "all to all" transitions are not shown).

F Proofs of performance

The synthesized CCAs ensure the synthesis invariant, but that is not sufficient to meet the performance objectives due to under-specification (§5.2). We build proofs, consisting of lemmas, that describe the states the CCA visits, transitions it makes, and the objectives it ensures. In the interest of space, we only discuss lemmas for `cc_qdel`. Lemmas for `cc_probe_slow` are similar but use $C_{L,\lambda}$, and q_U belief bounds instead of C_L and C_U.

Using the verifier, we checked that the lemmas are true for `cc_qdel` running on the CCAC network model with parameters $\langle C, R_m, D = R_m, \beta \geq 3C\cdot(R_m + D)\rangle$. We give a summary of how the lemmas work together, describe how we built them, and then discuss each lemma.

Summary. Fig. 14 describes how the lemmas are related to each other. The lemmas are true over any $10R_m$ seconds trace of the CCA's execution. We stitch together lemmas to reason about performance over arbitrarily long time horizons.

Whenever the link rate changes significantly, C_L and C_U beliefs may become inconsistent. The CCA ensures that these beliefs become consistent exponentially fast (Lemma F.2), they converge to a small range exponentially fast (Lemma F.4), and finally, the CCA reaches steady state (Lemma F.6), i.e., the bottleneck queue reduces additively at rate proportional to C.[9] The lemmas ensure that progress always happens in the same direction. Specifically, the beliefs cannot become inconsistent on their own, i.e., without link rate varying (Lemma F.1), the beliefs cannot diverge post convergence (Lemma F.3), the bottleneck queue cannot become unbounded after it converges (Lemma F.5). In other words, once the CCA reaches steady state (IV), it stays there. In the steady state (IV), `cc_qdel` gets at least 89% utilization, keeps *RTT*s to less than $4.4(R_m+D)$ seconds, and loses at most 3 packets in any R_m duration (Lemma F.7). Additionally, Lemma F.8 ensures that `cc_qdel` never incurs large loss events when probing for bandwidth as long as the beliefs are consistent, i.e., in states II, III, IV.

Building lemmas using binary search. The lemmas include non-trivial constants, we obtain all these by using binary search. For instance, to compute utilization in steady state, we ask the verifier if `cc_qdel` violates Lemma F.5 for different choices of utilization. When we ask this query for utilization = 100% the verifier gives a counterexample showing that `cc_qdel` can get lower than 100% utilization. However, when we repeat for 50%, the verifier says UNSAT, implying on (worst-case) all executions that start in steady state, `cc_qdel` gets at least 50% utilization. We repeat and find that `cc_qdel` ensures 89% utilization in steady state.

[9]A CCA cannot drain the queue faster than this. Even if the CCA stops sending packets, and the bytes are serviced at rate C, then the queue will only drain by $C\cdot t$ bytes in time t.

LEMMA F.1. *Initial beliefs are consistent $\implies$ final beliefs are consistent. Where, beliefs are consistent $\equiv C_L(t) \leq C \leq C_U(t)$, and we evaluate initial beliefs at $t = 0$ and final beliefs at $t = T = 10R_m$.*

This lemma verifies that our belief computations are correct. I.e., any bandwidth C that can produce the observations in the trace from $t=0$ to $t=T$, and also produce observations before $t=0$ (i.e., C is in the initial belief set or "initial beliefs were consistent") should be in the final belief set (or "final beliefs should be consistent"). This lemma is true for any CCA.

LEMMA F.2. *($\neg$ Initial beliefs are consistent) $\implies$ (final beliefs move towards consistency $\vee$ final beliefs are consistent $\vee$ steady state objectives). Where,*

Final beliefs move towards consistency $\equiv$

$(C_L$ *inconsistent* $\implies$ *C beliefs decrease*$)\wedge$

$(C_U$ *inconsistent* $\implies$ *C beliefs increase*$)$

C_L *inconsist.* $\equiv C_L(0) > C,\quad C_U$ *inconsist.* $\equiv C_U(0) < C$

C beliefs dec. $\equiv$ *At least one decreases* $\wedge$ *None increases*

At least one decreases $\equiv$

$C_L(T)\cdot 1.1 < C_L(0) \vee C_U(T)\cdot 1.1 < C_U(0)$

None increases $\equiv C_L(T) \leq C_L(0) \wedge C_U(T) \leq C_U(0)$

C beliefs inc. is defined symmetric to C beliefs dec.

Steady state objectives $\equiv$ (*Utilization lower bounded*

Inflight upper bounded $\wedge$ *No large loss events*)

$$\equiv \frac{S(T)-S(0)}{C(T-D)} \geq 83\% \wedge$$

$$\forall t.\theta(t) \leq 4.4C\cdot(R_m+D) \wedge \forall t.L(t) - L(t-1) \leq 3MSS$$

In each trace, the beliefs move towards consistency by a multiplicative factor, i.e., *exponentially fast*. The $1.1\times$ factor is a worst-case movement over all possible $10R_m$ long executions. In traces where C_L, C_U are far from C, beliefs move quicker. Beliefs start moving slower (but at least $1.1\times$) only when they are very close to C. So in practice the amount that the beliefs move varies over time and the beliefs become consistent much quicker than if they only improved by only $1.1\times$ every $10R_m$ seconds.

To encode "final beliefs move towards consistency", we require that (i) at least one of the C beliefs move in the correct direction *and* (ii) neither of them move in the wrong direction. The second clause (i.e., ii) is needed to ensure that the progress made by the CCA in consecutive traces adds up. Without this, it can happen that C_L increases and C_U decreases in the first trace, then C_U increases and C_L decreases, and so on. In each trace at least one of C_L or C_U is moving in the right direction (satisfying clause i), but their progress is not adding up.

LEMMA F.3. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged). Where, beliefs are converged* $\equiv C_L(t) \geq \frac{27C}{40} \wedge C_U(t) \leq 3C$.

When the beliefs are in the range $[\frac{27C}{40}, 3C]$, they do not go outside this range unless the link rate varies. This is despite periodic belief timeouts (§5.1.2). The beliefs are only timed out when they are well within this range, so that they stay within the range even after the timeout if the link rate hasn't changed.

Recall, we found the constants 27/40 and 3 by binary search, i.e., the tightest range for which Lemma F.3 is true.

LEMMA F.4. *(Initial beliefs are consistent $\wedge$ $\neg$ initial beliefs are converged) $\implies$ (final beliefs are consistent $\wedge$ (final beliefs shrink $\vee$ final beliefs are converged $\vee$ steady state objectives)). Where,*

$$\textit{Final beliefs shrink} \equiv \textit{At least one improves } \wedge \textit{None degrade}$$
$$\textit{At least 1 imp.} \equiv C_L(T) > 1.7 C_L(0) \vee C_U(T) < 1.7 C_U(0)$$
$$\textit{None degrade} \equiv C_L(T) \geq C_L(0) \wedge C_U(T) \leq C_U(0)$$

The beliefs shrink by a multiplicative factor i.e., *exponentially fast*. Similar to Lemma F.2, we need the "at least one improves" *and* "none degrades" pattern to ensure that the progress made by the CCA adds up.

LEMMA F.5. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ initial bottleneck queue is bounded) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged $\wedge$ final bottleneck queue is bounded). Where, bottleneck queue is bounded* $\equiv q(t) \leq 3.3C \cdot (R_m + D)$.

LEMMA F.6. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ $\neg$ initial bottleneck queue is bounded) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged $\wedge$ (final bottleneck queue is bounded $\vee$ bottleneck queue reduces)). Where, bottleneck queue reduces* $\equiv q(T) < q(0) - CR_m/2$.

To drain the queue, the CCA takes time that is linear in the queue size (i.e., it decreases queue *additively*, proportional to the *BDP* in each trace). Note, no CCA can multiplicatively reduce the number of bytes in the bottleneck queue. Even if the CCA stops sending packets, and the bytes are serviced at rate C, then the queue will only drain by $C \cdot t$ bytes in time t, this is independent of the number of bytes in the queue (i.e., not a multiple of the queue size).

LEMMA F.7. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ initial bottleneck queue is bounded) $\implies$ steady state objectives.*

LEMMA F.8. *(Initial beliefs are consistent) $\implies$ (rate increases $\implies$ no large loss events). Where, rate increases* $\equiv rate(T) > rate(0)$.

Once the C beliefs are consistent, when `cc_qdel` is probing (or ramping up, i.e., "rate increases"), it never incurs any large loss events losses. Large losses may happen if the link rate decreases (there is no way to avoid this). Note, this is only true on networks with large buffers, i.e., $\beta \geq 3C \cdot (R_m + D)$.

G Implementation issues in the Linux kernel

We originally implemented the synthesized CCAs in the Linux kernel. We uncovered bugs in the kernel's pacing implementation and also found the `cong_ctrl` API to be insufficient to implement the synthesized CCAs.

Pacing bug. To implement pacing, the Linux kernel pre-computes the "time to send the next packet" as the inverse of the pacing rate, i.e., inter-send-time. If the CCA's sending rate changes before the pre-computed time, then the kernel implements the wrong sending rate until the time to send the next packet. This leads to a discrepancy in the number of packets actually sent vs. the number of packets that the CCA wanted to send in a time interval.

In general, it is hard to implement pacing correctly. Pacing constrains two things, (1) a lower bound on the inter-send-time between packets and (2) number of packets sent in a time interval. Practically, a pacing implementation cannot meet both these constraints. Due to delays in CPU scheduling and interrupt handling, an instruction's execution may be delayed. These delays can cause a pacing implementation to miss a sending opportunity. When this happens, pacing implementation can either temporarily increase the sending rate to correct for the delayed opportunity (there by violating the lower bound on inter-send-time) or process the delayed sending opportunity as is (there by not sending any bytes corresponding to the delay and violating the constraint on total packets sent).

In fact, the pacing implementation of the genericCC [54] (the framework that we used to implement the synthesized CCAs in userspace over UDP) was also incorrect. We modified it so that it faithfully honors the lower bound on inter-send-time to ensure constant loss. Doing so creates a minor discrepancy between the packets sent over an interval vs. packets the CCA expected to send. Specifically, we maintain `last_sent_time` and compute `inter_send_time` whenever the CCA changes its sending rate. We launch a thread that polls (busy waits) to check if the sender is allowed to send (i.e., `current_time` is $\geq$ `last_sent_time` + `inter_send_time`).

The busy waiting can be avoided by setting two interrupts: (1) sleeping for time = `inter_send_time`, and (2) whenever ACKs are received. Before either of these two interrupts hit, no packets are sent or received and so there is no reason for CCA to change its rate, so it is okay to sleep until these interrupts as the `inter_send_time` does not become stale until these interrupts. Note, setting interrupts increases the delays caused in scheduling threads. This can increase the discrepancy between the actual packets sent vs. the packets the CCA expected to send. The pacing implementation may choose to transiently increase the sending rate to correct for sending opportunities missed due to scheduling delays.

API. The Linux kernel also does not provide a direct API to change the sending rate after packets are sent. The kernel only gives a callback on ACKs. Since ACKs can be delayed due to non-congestive delays, the CCA ends up setting a potentially stale sending rate for $O(D)$ time. This can be worked around by setting up timer interrupts or instrumenting a callback on packet send events. The genericCC API provides callbacks on both packet send and receive events.

H Supplementary empirical evaluation

We empirically evaluate the synthesized CCAs across a variety of $\langle C, R_m, D, \beta \rangle$ parameters. Specifically, we explore combinations of $C \in \{24, 48, 96\}$ Mbps, $R_m \in \{20, 40, 80\}$ ms, $D = R_m$, and $\beta \in \{1/16, 1/8, 1/4, 1/2, 1, 2, 4, 8, 16\}$ *BDP*. Each tuple is a 60 seconds long "run". We discard the first 20 seconds of each run (to study steady-state behavior) and compute metrics over the remainder.

To emulate jitter, we inject up to R_m seconds of *random* ACK aggregation with a fixed average link rate. We sample `agg_delay` $\in [0, D = R_m)$ uniformly randomly. We serve a batch of $C \cdot$ `agg_delay` bytes after waiting for `agg_delay` seconds, ensuring an average bandwidth of C. We sample `agg_delay` after every batch.

Fig. 15 and Fig. 16 show the utilization, delay, and loss metrics for each run. The synthesized CCAs are withing their proof bounds in each run. In some of the runs, `cc_probe_slow`'s utilization is less than 50% (expected steady-state utilization in §6.2.1), this is because in those runs, it has not reached steady-state at the end of the run.

Note, BBRv2 and BBRv3 are not robust against adversarial jitter. On short buffers, they do not cause loss in steady-state. This is needed to ensure a lower bound on utilization on CBR-delay (**Step 1.** of Theorem 6.1), because otherwise their observations could be explained by an arbitrary large link rate. We believe this is why they get lower utilization when buffer is small in Fig. 15.

Convergence time. We study convergence time in Fig. 18 and Fig. 19. We show a run with $R_m = 80$ ms. For increasing link rate, we double the link rate every 20 seconds starting at 24 Mbps. For decreasing link rate, we halve the link rate every 20 seconds starting at 96 Mbps. In both cases we set the buffer size as the *BDP* when link (wire) rate is 96 Mbps. I.e., $\beta = BDP = 96$ Mbps $\cdot 80$ ms ≈ 624 packets (each packet sends 1538 bytes on the wire in our setup).

`cc_probe_slow` meets its convergence time bounds. It converges additively when link rate increases and exponentially fast when link rate decreases (due to belief timeouts).

BBRv2 and BBRv3 have low loss on average, but incur large loss events, i.e., $O(BDP)$ whenever link rate increases, and they start probing exponentially fast. This happens at the 20 second and 40 second marks when the link rate increases (not shown).

To synthesize a CCA that follows the tradeoff choice made by BBRv2 and BBRv3, we would need to remove the under-specification in our synthesis invariant. We want to be able to cause large losses when we are in state II (i.e., the link rate changed, and we want to converge), but do not want to cause large losses in state IV (when the link rate has not changed, but we just want to check if it may have increased after a belief timeout). Because of under-specification we cannot distinguish between these scenarios during synthesis. We leave this for future work.

We can manually design such a CCA using the version of `cc_probe_slow` parametrized by $f(.)$ and T^* (§6.2.1). We can adapt the loss allowance $f(.)$ depending on if the CCA is in steady-state (causing periodic small losses), or if the CCA needs to converge (is not causing losses). This can also be done in a fashion similar to BIC [59] and Cubic [31], i.e., loss allowance adapts depending on how much the link rate changed. We leave this exploration for future work.

Fairness. Currently, our formal theoretical framework does not provide any guarantees in multi-flow scenarios, nor does it provide any predictions for the outcomes of multi-flow experiments. Nevertheless, we explore the fairness properties of the synthesized CCAs empirically.

Fig. 17 shows a run with $C = 96$ Mbps, $R_m = 80$ ms, $\beta = 1/2$ BDP on an ideal link. We run 4 flows that share the same bottleneck. Each flow is started 30 seconds after the previous flow and lasts for $4 \times 30 = 120$ seconds. Results are qualitatively similar for other parameter and network model combinations. For each run (i.e., $\langle C, R_m, D, \beta \rangle$ combination mentioned at the beginning of the section), we compute the Jain's fairness index (JFI) for the average rates of the 4 flows between time 100 to 110 seconds (i.e., when all 4 flows are running). When $\beta \le 2BDP$, the JFI for `cc_probe_slow` across the parameter combinations is 0.94 ± 0.06 (i.e., mean $\pm$ stddev) without jitter and 0.86 ± 0.11 with jitter. When $\beta > 2BDP$, `cc_probe_slow` effectively runs `cc_qdel` as there is no large loss event (§6.4). `cc_qdel` is unfair and gets JFI of 0.58 ± 0.24 without jitter and 0.43 ± 0.18 with jitter.

Even though `cc_probe_slow` was designed for single-flow scenarios, it is able to converge to a fair share of the available bandwidth. We believe this is because, to track changes in link rate, `cc_probe_slow` increases its effective rate ($C_{L,\lambda}$) additively and reduces its effective rate ($C_{L,\lambda}$) multiplicatively allowing it to reach a fair allocation similar to AIMD [18].

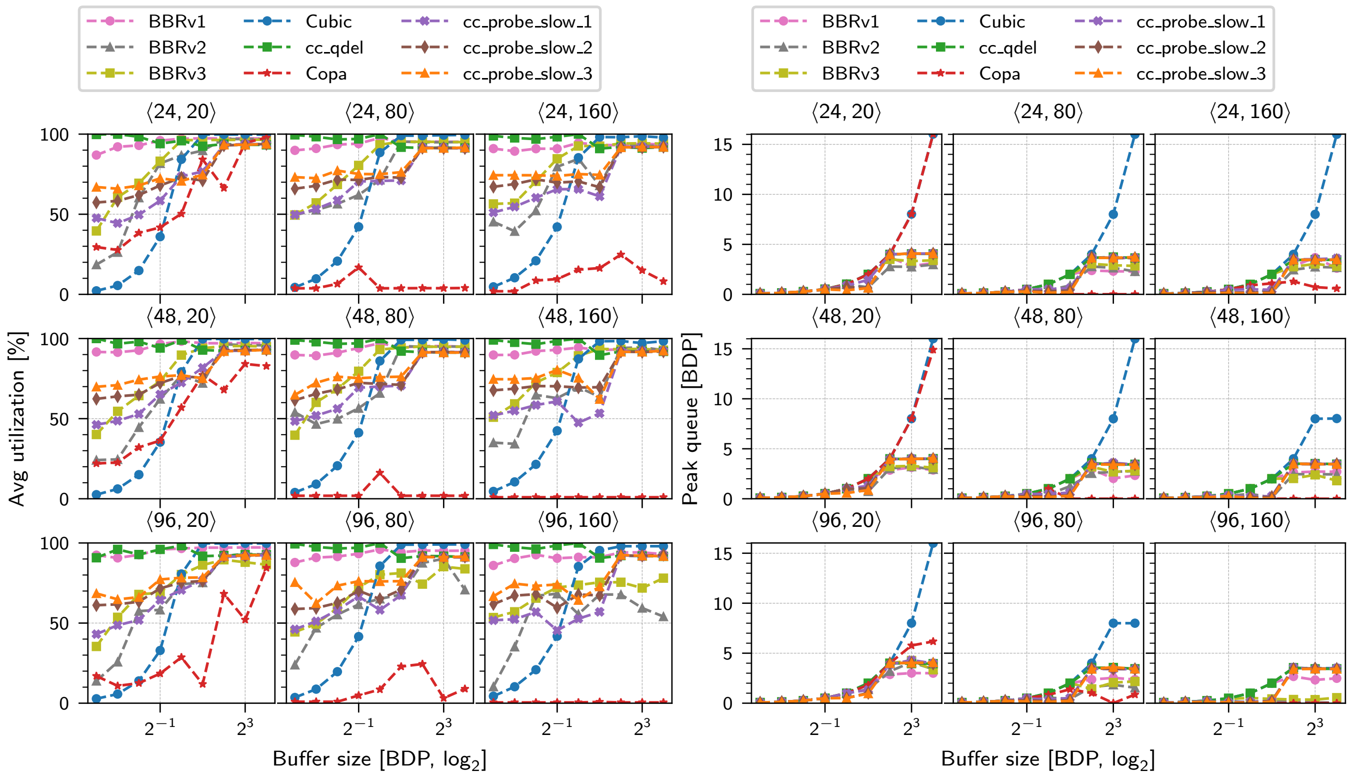


Figure 15: Average utilization and maximum queue use with varying bandwidth, propagation delay and buffer sizes for a link with random ACK aggregation. The tuple at the top of each subplot shows $\langle C \text{ Mbps}, R_m \text{ ms} \rangle$.

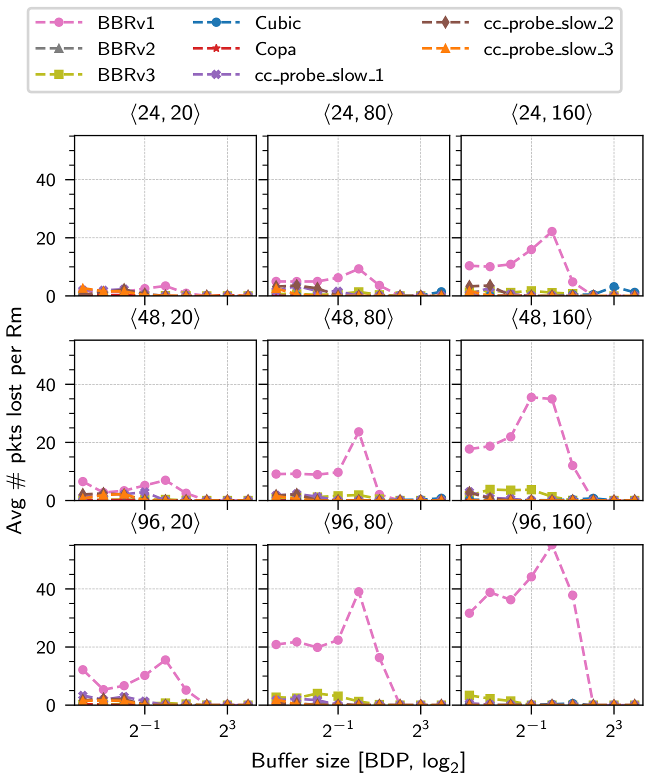


Figure 16: Packet loss with varying bandwidth, propagation delay and buffer sizes for a link with random ACK aggregation. The tuple at the top of each subplot shows $\langle C \text{ Mbps}, R_m \text{ ms}\rangle$. We compute number of packets lost in every R_m long interval and take average over all the intervals. We omit `cc_qdel`, it causes $O(BDP)$ losses with a higher constant factor than BBRv1 (Fig. 9), and ends up skewing the graphs.

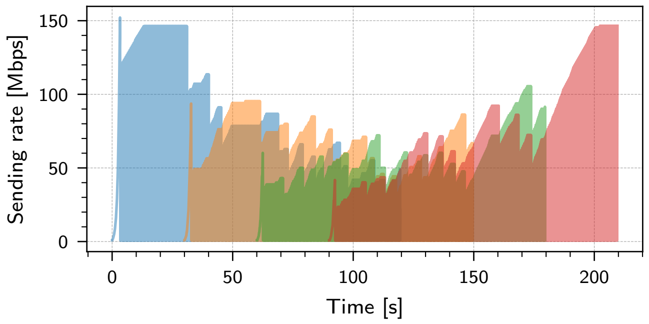


Figure 17: With multiple flows, `cc_probe_slow` is able to fairly share the available bandwidth. Every 30 seconds, we start a new flow for a total of 4 flows shown by the different colors. We use translucency to show the sending rates when the graphs overlap.

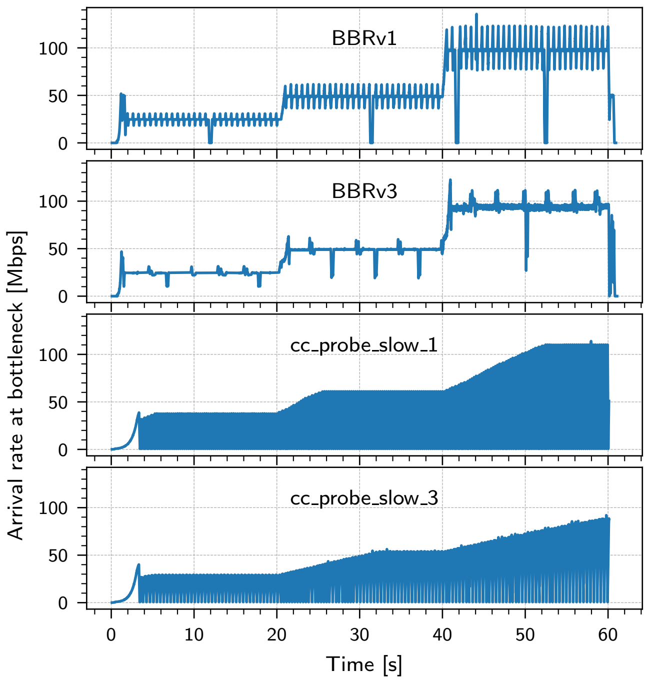


Figure 18: Convergence time for increasing C. `cc_probe_slow_k` converges additively when the link rate increases. Increasing `k` increases utilization by a multiplicative factor (Fig. 15) at the cost of increasing convergence time by a multiplicative factor.

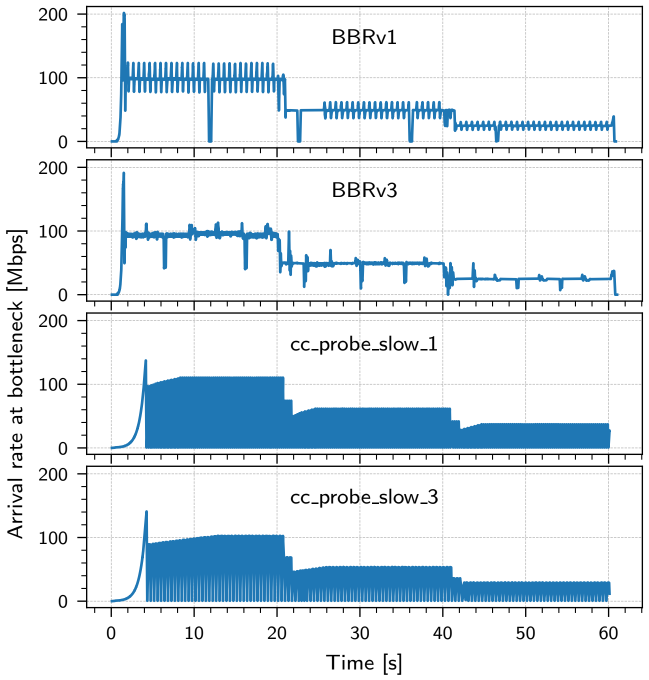


Figure 19: Convergence time for decreasing C. `cc_probe_slow_k` converges exponentially fast when the link rate decreases.