Proceedings of Machine Learning Research vol XX:1–16, 2023

On The Formally Verifying Congestion Control Algorithms Behavior

Behrooz Farkiani B.FARKIANI@WUSTL.EDU
1 Brookings Dr., Washington University in St. Louis, MO, 63130

Abstract

Congestion control algorithms play a vital role in the efficiency of communication over the Internet. However, the primary way to analyze their performance has traditionally been through extensive simulation and/or emulation. This work investigates the use of formal methods to analyze the behavior of AIMD and BBR congestion control algorithms based on the paper ”Toward Formally Verifying Congestion Control Behavior.” The authors found that AIMD can exhibit surprising behavior, where premature loss leads to a very small congestion window even when the network has a noticeably large bottleneck buffer. They also analyzed AIMD and proposed a steady-state bound defined in terms of the congestion window and loss limits. For BBR, they observed that it can achieve very low utilization, which is harmful to its performance.

In this study, I investigated the authors' implementation in detail and documented the behavior of the network and congestion control algorithms. I identified and questioned several aspects of the original implementation and proposed enhancements. I compared the original implementation with my enhancements. My modifications achieved better real-world results while still preserving the surprising behavior of AIMD and the low utilization reported for BBR. Additionally, I found that the proposed steady-state bounds are not valid and can be violated.

Keywords: Congestion Control, BBR, AIMD, SMT, Formal Verification, Z3

1. Introduction

Efficient end-to-end data delivery must (1) prevent a sender from congesting the network and (2) enforce limits on incoming traffic to avoid capacity overflow while improving reliability and availability. The former is embodied in *congestion-control* algorithms and the latter in *flow-control* algorithms. The paper I am discussing here, *Toward Formally Verifying Congestion Control Behavior* (Arun et al. (2021)), focuses on congestion control and proposes a tool named *CCAC* that uses formal-verification tools to study congestion-control algorithms' behavior. Traditional evaluation relies on extensive simulations or live A/B tests, which are time and resource-intensive. This work instead uses formal verification to model both the network path and the sender's control logic as an SMT problem, allowing Z3 to prove performance bounds or generate concrete counter-examples.

The network between sender and receiver is shared among multiple senders; for instance, it can be an interconnection of independent networks such as the Internet. The network has finite capacity and may carry traffic from many senders simultaneously. If the aggregate input rate exceeds the link's capacity, *congestion* occurs, leading to buffer overflow and packet loss. Senders must therefore regulate their sending rate to avoid both network congestion and receiver overload. Although receivers also have finite buffers, CCAC does not model receiver-side flow control. There exist many congestion-control algorithms in the literature (Jiang et al. (2021)); CCAC models AIMD (Gurtov et al. (2012)), BBR (Cardwell et al. (2016)), and Copa (Arun and Balakrishnan (2018)), and we focus here on AIMD and BBR as they are most widely deployed.

CCAC assumes a reliable, in-order byte stream between sender and receiver. The sender partitions data into segments (*packets*) of arbitrary size, each labeled with a unique byte-sequence number. When the receiver obtains all bytes up to sequence number n, it sends a *cumulative ACK* of value $n+1$. If later packets arrive out of order, the receiver can emit *duplicate ACKs*, each carrying the same ACK number. For example:

- Bytes 10-99 arrive correctly, so the receiver ACKs 100.
- Bytes 100-199 are lost, but then bytes 200-299, 300-399, and 400-499 arrive.
- The receiver cannot advance its ACK beyond 100, so it emits three duplicate ACKs with value 100.

Upon receiving three duplicate ACKs, the sender's fast-retransmit mechanism resends the missing segment. Separately, each packet starts a retransmission timer: if its ACK does not arrive before timeout, the sender assumes loss and retransmits.

CCAC combines the path and receiver into a single *path-server* model. Thus the formalization comprises:

1. *Sender equations*, which specify how the sender adjusts its congestion window $cwnd(t)$ in response to network behavior, ACKs, and losses from the path-server.
2. *Path-server equations*, which enforce capacity, buffering, jitter, and waste constraints on incoming traffic.

When traffic flows, it experiences various delays, including *propagation delay* and *queueing delay*. CCAC uses a simplified path-server as shown in Figure 1. It assumes network capacity depends on a bottleneck link with a fixed capacity C (bytes/s). The network also has a round-trip propagation delay R_m (s). Capacity is enforced via a *token bucket* that generates tokens at rate C and stores them in a token-bucket queue. The token queue has a limited capacity K, and tokens generated when this capacity is filled will be wasted. In addition, new tokens can arbitrary be *wasted* if the token-queue length exceeds the packet-queue length, ensuring that waste does not reduce service capacity. This allows jitter in service rate. In addition, the path has a FIFO packet queue. Arrivals first enter this queue and are served as tokens allow. When X bytes of packets arrive:

- In the absence of arrivals, the bucket accumulates tokens up to K
- Admitted packets may be delayed up to D seconds before service, simulating burstiness, ACK aggregation, MAC-layer scheduling, etc.
- Admitted tokens cannot remain unused longer than D, so $K = C \cdot D$ bounds the maximum burst.

The difference between what arrives and what is served forms the *bottleneck queue*. This queue has a finite capacity B; if the backlog exceeds B, overflow occurs and packets are lost. The sender regulates the amount of traffic it send to the network via the congestion window $cwnd(t)$, which caps the *in-flight* data (bytes sent but not yet acknowledged or declared lost). Whenever

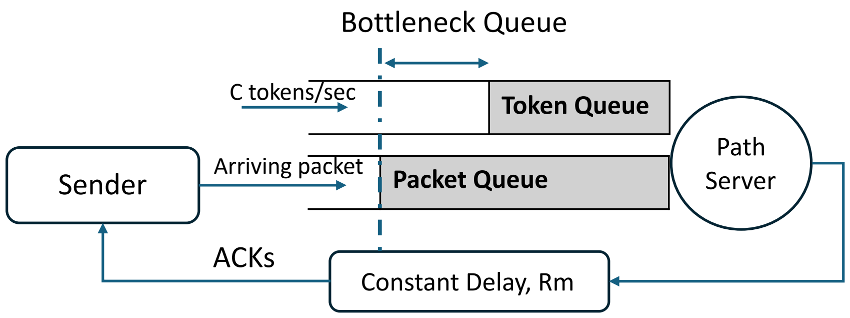

Figure 1: CCAC's Path-server model

$\text{inflight} < cwnd$, the sender may inject additional data up to the window limit. Different congestion control algorithms adjust cwnd in distinct ways; for example, based on duplicate ACKs, timeouts, or measured delivery rates, and CCAC's simplified model captures these behaviors as we describe in later sections.

The paper studies a single sender-receiver flow, simplified versions of congestion-control algorithms, and worst-case network behavior. Its main contributions are:

- Modeling the path-server and congestion-control logic as an SMT instance, and using a set of SMT queries to analyze congestion-control behavior.
- Showing that AIMD can incorrectly reduce its congestion window due to buffer overflow even when buffer is large. This phenomenon is referred to as *surprising behavior* in AIMD.
- Proposing a new steady-state bound for AIMD.
- Demonstrating that BBR's utilization can drop to very low levels.

My analysis and contributions address the following:

- I describe and validate the SMT implementation of the model, filling in details omitted from the original paper.
- I propose four enhancements (*fixes*) to the model and its queries.
- I demonstrate, using the enhanced model, that the proposed steady-state bounds for AIMD do *not* always hold.

The remainder of the paper is organized as follows. Section 2 describes the path-server model as implemented in SMT. Section 2.2 presents the AIMD congestion control algorithm and its implementation, and Section 2.3 describes BBR. Section 3 discusses the proposed enhancements, and Section 4 presents the evaluation results of the original implementation and the enhanced model. Finally, Section 5 concludes the paper.

2. Modeling of Network and Algorithms

This section first describes the network model, and then investigates how AIMD and BBR are implemented. Notations are shown in Table 1.

Symbol	Description
$A(t)$	Cumulative bytes arrived at the path-server by time t.
$S(t)$	Cumulative bytes served by the path-server (i.e., ACK bytes sent toward the sender) by time t.
$L(t)$	Cumulative bytes lost by the path-server by time t.
$W(t)$	Cumulative tokens wasted by the path-server by time t.
$Q(t)$	Packet queue length at time t, $Q(t) = A(t) - L(t) - S(t)$.
$A_f(t)$	Cumulative bytes sent by the sender (per-flow) by time t.
$L_f(t)$	Cumulative bytes lost (per-flow) by time t.
$Ld_f(t)$	Cumulative losses detected by sender (dupacks/timeouts) by time t.
$S_f(t)$	Cumulative bytes served (per-flow) by time t.
$r_f(t)$	Pacing rate (bytes per timestep) at time t.
$\text{Inflight}(t)$	Sender's in-flight bytes at time t, $\text{Inflight}(t) = A_f(t) - Ld_f(t) - S_f(t - R_m)$.
$\text{timeout}(t)$	Boolean: sender timeout at time t.
$T(t)$	Token queue length at time t, $T(t) = C \cdot t - W(t) - S(t)$.
C	Fixed bottleneck link capacity (bytes per timestep).
D	Maximum per-packet jitter (timesteps).
β	Buffer capacity (bytes).
α	Additive-increase increment.
R_m	Round-trip propagation delay (timesteps).
`buf_min`, `buf_max`	Minimum/Maximum buffer thresholds.
dupacks	Duplicate-ACK threshold for loss detection.
$\text{cwnd}_f(t)$	Congestion window (bytes) of flow f at time t.
T	Evaluation duration.

Table 1: Notation used in the CCAC path-server SMT model.

2.1. Network model

The network is modeled as a token-bucket filter with maximum service rate C, up to D timesteps of per-packet delay, and a fixed round-trip propagation delay of R_m.

2.1.1. TOKEN-BUCKET BALANCE

$$T(t) \;=\; C \cdot t \;-\; W(t) \;-\; S(t)$$

2.1.2. NETWORK QUEUE LENGTH

$$Q(t) \;=\; A(t) \;-\; L(t) \;-\; S(t)$$

The difference between Q(t) and T(t), packet queue and token bucket queue, will be held at bottleneck queue. Loss happens when bottleneck queue exceeds bottleneck queue capacity.

2.1.3. MONOTONICITY (FOR $1 \le t < T$):

$$\begin{aligned} A_f(t) &\ge A_f(t-1), & L_f(t) &\ge L_f(t-1), \\ S_f(t) &\ge S_f(t-1), & Ld_f(t) &\ge Ld_f(t-1), \\ W(t) &\ge W(t-1), & A_f(t) - L_f(t) &\ge A_f(t-1) - L_f(t-1). \end{aligned}$$

All cumulative quantities (arrivals, losses, service, and waste) must be non-decreasing.

2.1.4. INITIALIZATION (AT $t = 0$):

$$\mathrm{cwnd}_f(0) > 0, \quad r_f(0) > 0, \quad L_f(0) \geq 0, \quad Ld_f(0) \geq 0, \quad S_f(0) \quad = 0.$$

As we see, the original implementation does not enforce wasted tokens W(t) to be non-negative.

2.1.5. TOTAL-FLOW RELATIONS (SINGLE FLOW):

$$A(t) = A_f(t), \quad L(t) = L_f(t), \quad S(t) = S_f(t).$$

2.1.6. NETWORK CAPACITY AND QUEUING (FOR ALL $0 \leq t < T$):

$$S_f(t) \leq A_f(t) - L_f(t),$$
$$C\,(t - D) - W(t - D) \;\leq\; S(t) \;\leq\; C \cdot t - W(t).$$

with $W(t - D)$ replaced by $W(0)$ if $t < D$. Service rate must be less than or equal to effective input to the network. Service rate is also bounded by generated tokens that are not wasted from above. We need to also ensure that tokens remaining from D second ago that are not wasted must have been server. These two are service bounds.

Wastage constraint:

$$W(t) > W(t-1) \implies A(t) - L(t) \leq C\,t - W(t)$$

The above condition ensures the wastage allows only if effective input $A(t) - L(t)$ is less than serving capacity.

Buffer-minimum:

$$L(t) > L(t-1) \implies A(t) - L(t) \geq C\,(t-1) - W(t-1) + \mathrm{buf\_min}.$$

Loss happens only when $Q(t) - T(t-1) > \beta$, i.e., difference between packet queue and token queue exceeds the bottleneck buffer size. We see authors used $T(t-1)$ instead of $T(t)$ because loss means we inserted data when token was not available, i.e., tokens were not generated at the same pace as we inserted data.

Buffer-maximum:

$$A(t) - L(t) \;\leq\; C\,t - W(t) + \mathrm{buf\_max}.$$

2.1.7. LOSS DETECTION (FOR ALL $0 \leq t < T$ AND dt WITH $t - R_m - dt \geq 0$):

Define

$$\mathrm{detectable}(t, dt) \;:\; A_f(t - R_m - dt) - L_f(t - R_m - dt) + \mathrm{dupacks} \;\leq\; S_f(t - R_m).$$

Loss is detectable when three duplicate acks arrive. Assume $dt = 0$ and the formula above would be

$$\mathrm{detectable}(t, dt) \;:\; A_f(t - R_m) - L_f(t - R_m) + \mathrm{dupacks} \;\leq\; S_f(t - R_m).$$

Suppose $t = 4$, $R_m = 1$, and $\mathit{dupacks} = 3$. At time $t - R_m = 3$, let $A_f(3) = 12$ and $L_f(3) = 2$. Then $A_f(3) - L_f(3) = 10$, the count of in-order bytes delivered. To detect the two lost bytes among those 12, the sender needs $10 + \mathit{dupacks} = 13$ ACKs. Hence, if by time 3 the path-server has served at least 13 bytes (so $S_f(3) \geq 13$), those 13 ACK-bytes arrive at the sender at $t = 4$ because of propagation delay, and the code sets `detectable` to true.

Whatever packets arrive may experience queueing delay before being served. Since the loss has already occurred in the past, CCAC introduces an index dt to scan all candidate loss epochs $t - R_m - dt$ and determine the earliest time at which the sender has received enough duplicate ACKs to detect that loss.

$$\neg\mathrm{timeout}(t) \wedge \mathrm{detectable}(t, dt) \implies Ld_f(t) \geq L_f(t - R_m - dt),$$
$$\neg\mathrm{timeout}(t) \wedge \neg\mathrm{detectable}(t, dt) \implies Ld_f(t) \leq L_f(t - R_m - dt).$$

These constraints ensure that, if no timeout occurs at time t and a loss is detectable after a delay offset dt, then the cumulative detected-loss counter $Ld_f(t)$ must be at least the actual cumulative loss $L_f(t - R_m - dt)$. Conversely, if the loss is not detectable, $Ld_f(t)$ cannot exceed $L_f(t - R_m - dt)$. Since both Ld_f and L_f are cumulative, this encodes a superset of possible real-world behaviors.

Note that the model does not preclude spurious loss detections; it permits $Ld_f(t) > 0$ even when no actual loss has occurred.

Timeout definition: For $t < R_m$: $\mathrm{timeout}(t) = \mathsf{false}$. For $t \geq R_m$:

$$\mathrm{timeout}(t) \;=\; \big(S_f(t - R_m) < A_f(t - 1)\big) \;\wedge\; \big(S_f(t - R_m) = A_f(t - R_m) - L_f(t - R_m)\big).$$

Timeout at time t is triggered when there are still outstanding bytes (i.e. $S_f(t - R_m) < A_f(t - 1)$) and the service delivered at $t - R_m$ exactly equals the effective arrivals up to that point ($A_f(t - R_m) - L_f(t - R_m)$). As the authors note, this *magical* RTO logic does not correspond to real-world timers, and implementing it in this way is strange.

Effects on detected loss:

$$\mathrm{timeout}(t) \implies Ld_f(t) = L_f(t),$$

$$L_{d,f}(t) \;\leq\; L_f(t - R_m).$$

On timeout, the detected-loss counter $Ld_f(t)$ jumps to the current cumulative loss $L_f(t)$. Moreover, it ensures detected losses never exceed the actual losses that occurred at least one RTT earlier.

2.1.8. 8. CWND AND ARRIVAL RATE (FOR $t \geq R_m$):

The congestion window $\mathrm{cwnd}_f(t)$ upper-bounds the amount of new data the sender may inject. In the CCAC model, the formula

$$A_w = S_f(t - R_m) \;+\; Ld_f(t) \;+\; \mathrm{cwnd}_f(t)$$

encodes the usual *window-limited* bound on how many bytes the sender may have cumulatively sent by time t. Concretely, $S_f(t - R_m)$ is the total bytes the sender has *already* seen acknowledged (via ACKs that departed the path-server at $t - R_m$). $Ld_f(t)$ is the total bytes the sender has *detected* as lost up through time t that sender needs to resend, and $\mathrm{cwnd}_f(t)$ is the sender's current congestion window. We know $S_f(t - R_m) \;\leq\; A_f(t - R_m) - L_f(t - R_m)$ and $Ld_f(t) \;\leq\; L_f(t - R_m)$. Adding these two terms and consideing Loss is cumulative, we get $S_f(t - R_m) + Ld_f(t) \;\leq\; A_f(t-1)$. Therefore, $A_w \leq A_f(t-1) + \mathrm{cwnd}_f(t)$ and it shows new data is capped by cwnd. The model picks maximum of $A'_w = \max(A_w, A_f(t-1))$.

The model also uses a pacing limit $A_r = A_f(t-1) + r_f(t)$ which regulates how quickly the sender can send new data. The $r_f(t)$ for AIMD is set to $100C$ to remove any limit in sending rate as AIMD is not pace-based. BBR calculates $r_f(t)$ in a different way that we discuss later. The model then takes

$$A_f(t) \;= \min\{A'_w,\; A_r\},$$

so that the actual cumulative arrival respects both the window-limit A_w (adjusted to be non-decreasing) and the pacing-limit A_r.

2.2. AIMD Congestion-Control Model

A simplified AIMD manages the congestion window as follows: if the network is not congested, the sender can increase the congestion window by α; otherwise it sets $\mathrm{cwnd}_f(t) = \mathrm{cwnd}_f(t-1)/2$. The implementation uses two Boolean variables plus a pointer:

$\mathrm{incr}_f(t)$	: sender **can increase** cwnd_f by α,
$\mathrm{decr}_f(t)$	: sender **can decrease** cwnd_f by half,
$\ell_f(t)$	: sequence number of the *last* loss event detected by t.

Initially $\ell_f(0) = S_f(0)$. In AIMD, if congestion window is reduced once for a loss event, it will not be reduced again for lost packets that were sent before the previous loss event.

2.2.1. CAN CCAC INCREASE $\mathrm{cwnd}_f(t)$ AT $t \geq 1$?

The code sets $\mathrm{incr}_f(t)$ to `true` if any of three rules holds:

$$\mathrm{incr}_f(t) \iff \big(\text{RULE A} \vee \text{RULE B} \vee \text{RULE C}\big),$$

$$\text{RULE A} : \exists\, dt \in \{1,\ldots,t\} : \underbrace{\Big[\bigwedge_{k=1}^{dt} \mathrm{cwnd}_f(t-k) = \mathrm{cwnd}_f(t)\Big]}_{\text{window fixed } dt \text{ steps}}$$
$$\wedge\ \mathrm{cwnd}_f(t-dt-1) \neq \mathrm{cwnd}_f(t-dt)\ \wedge\ S_f(t) - S_f(t-dt) \geq \mathrm{cwnd}_f(t),$$

$$\text{RULE B} : \Big[\bigwedge_{k=1}^{t} \mathrm{cwnd}_f(t-k) = \mathrm{cwnd}_f(t)\Big]\ \wedge\ S_f(t) - S_f(0) \geq \mathrm{cwnd}_f(t),$$

$$\text{RULE C} : S_f(t) - S_f(t-1) \geq \mathrm{cwnd}_f(t).$$

Take $t = 10$, choose $dt = 5$ for RULE A. The condition becomes

$$\big(\mathrm{cwnd}_f(10) = \mathrm{cwnd}_f(9) = \cdots = \mathrm{cwnd}_f(5)\big)$$
$$\wedge\ \big(\mathrm{cwnd}_f(4) \neq \mathrm{cwnd}_f(5)\big)\ \wedge\ \big(S_f(10) - S_f(5) \geq \mathrm{cwnd}_f(10)\big).$$

Thus the window has stayed constant for five steps, was *different* just before that, and the network has delivered at least one full window during those five steps, so an increase is allowed.

With the same $t = 10$, RULE B reads

$$\big(\text{cwnd}_f(10) = \text{cwnd}_f(9) = \cdots = \text{cwnd}_f(0)\big) \;\wedge\; \big(S_f(10) - S_f(0) \geq \text{cwnd}_f(10)\big),$$

i.e. the window has never changed and the network has processed at least one entire window since start-up.

RULE C is the one-step check: if the network delivered one full window between $t-1$ and t, the sender may grow the window.

As the authors note, the model is *more relaxed* than a real-world implementation. Crucially, none of the three rules mentions the round-trip delay R_m; ACKs generated at time t only reach the sender at $t + R_m$, so the model may increase the window sooner than a real-world implementation. This mismatch becomes important in the steady-state proofs discussed later.

2.2.2. DECREASE TRIGGER

The model halves the congestion window only when *new* loss is detected. Formally,

$$\text{decr}_f(t) = \begin{cases} Ld_f(t) > Ld_f(t-1), & t \leq R_m + 1, \\ Ld_f(t) > Ld_f(t-1) \;\wedge\; \ell_f(t-1) \leq S_f\big(t - R_m - 1\big), & t > R_m + 1. \end{cases}$$

At very early times ($t \leq R_m + 1$) the sender cannot yet compare against a full round-trip's worth of ACKs, so *any* increase in the detected-loss counter Ld_f is enough to trigger a decrease.

Once one RTT has passed ($t > R_m + 1$), the model requires two conditions: 1) the sender has just observed additional loss; and 2) the highest sequence number recorded for the *previous* loss event lies *at or before* the highest byte that served one R_m before. Using this equation, sender knows up to which sequence number has been procesed. This guarantees the window is not halved twice for the same loss. In other words, the pointer ℓ_f prevents duplicate multiplicative decreases: the sender will only react to loss of packets that were sent after previous loss event.

2.2.3. CWND AND LAST-LOSS UPDATE ($t \geq 1$)

After knowing when to increase or decrease congestion window, the rest is easy. The model adds the following conditions to regulate congestion window and loss.

$$\neg\text{timeout}(t) \wedge \neg\text{decr}_f(t) \implies \ell_f(t) = \ell_f(t-1),$$

$$\text{timeout}(t) \implies \big[\text{cwnd}_f(t) = \alpha \;\wedge\; \ell_f(t) = A_f(t) - L_f(t) + \text{dupacks}\big],$$

$$\neg\text{timeout}(t) \wedge \neg\text{decr}_f(t) \wedge \text{incr}_f(t-1) \implies \text{cwnd}_f(t) = \text{cwnd}_f(t-1) + \alpha,$$

$$\neg\text{timeout}(t) \wedge \neg\text{decr}_f(t) \wedge \neg\text{incr}_f(t-1) \implies \text{cwnd}_f(t) = \text{cwnd}_f(t-1).$$

$$\neg\text{timeout}(t) \wedge \text{decr}_f(t) \implies \big[\text{cwnd}_f(t) = \tfrac{1}{2}\,\text{cwnd}_f(t-1) \;\wedge\; \ell_f(t) = A_f(t) - L_f(t) + \text{dupacks}\big].$$

2.3. BBR Congestion-Control Model

Although BBR is a complex algorithm, the paper attempts to model how BBR regulates its congestion window in an abstract way. AIMD is a *loss-based* congestion-control algorithm: it changes the congestion window in response to losses and acknowledgements. BBR, on the other hand, tries to estimate the network's bottleneck bandwidth and round-trip propagation delay and then regulate its sending rate so that buffer overflow, and hence, loss never occurs.

Classic loss-based algorithms probe for capacity until the in-flight data reaches BDP $+ \beta$ and realise they have gone too far only after a loss event. BBR instead maintains an explicit model of the path. For each ACK it measures the round-trip time (RTT), updates its model, and uses a pacing rate to regulate how fast it injects packets. The key relations are:

- **Bottleneck bandwidth (BtlBw).** Once per RTT the sender forms a delivery-rate sample

$$\text{rate} = \frac{\text{bytes\_delivered}}{\text{elapsed\_time}},$$

  and keeps BtlBw as the *maximum* of the last ten such samples.

- **Propagation delay (RTprop).** RTprop is the *minimum* RTT observed in a ten-second sliding window.

- **BDP and congestion window.** BDP = BtlBw × RTprop. The congestion window is

$$\mathrm{cwnd} = \text{cwnd\_gain} \times \text{BDP}, \qquad \text{with cwnd\_gain} = 2.0,$$

  and the sender transmits only while inflight $< \mathrm{cwnd}$.

- **Pacing rate.** Packets are paced at

$$rate = \text{pacing\_gain}(t) \times \text{BtlBw},$$

  where pacing_gain(t) runs through an eight-RTT cycle $\{1.25,\ 0.75,\ 1, 1, 1, 1, 1, 1\}$. RTT 0 probes for extra bandwidth (gain 1.25); RTT 1 drains any queue it created (gain 0.75); the next six RTTs cruise at gain 1.0.

In CCAC the round-trip propagation delay R_m is treated as a fixed, given constant, the sender in the model does *not* attempt to re-measure it. The CCAC models BBR with a **4-RTT cycle** (rather than the full 8-RTT cycle of the real algorithm) and with a short, 4-sample filter for the bottleneck bandwidth. A per-flow phase counter $s_f(t) \in \{0, 1, 2, 3\}$ indicates which step of the cycle is in force when deciding the pacing rate $r_f(t)$.

2.3.1. CONGESTION-WINDOW CALCULATION

For every timestep $t \geq 2R_m$ the sender forms rate samples from the cumulative service curve S_f:

$$r_f(t - dt) \;=\; \frac{S_f\big(t - dt - R_m\big) \;-\; S_f\big(t - dt - 2R_m\big)}{R_m}, \qquad dt \in \{0, 1, 2, 3\}.$$

The bottleneck-bandwidth estimate is

$$\text{BtlBw}_f(t) = \max_{dt=0}^{3} r_f(t - dt),$$

and hence

$$\text{BDP}_f(t) = \text{BtlBw}_f(t)\, R_m, \qquad \mathrm{cwnd}_f(t) = 2\, \text{BDP}_f(t).$$

2.3.2. PACING RULE (4-RTT CYCLE)

Let The model uses a *pacing-gain* table table as follows to calculate $s_f(t)$.

$$g_f(t) = \begin{cases} 1.25, & s_f(t) = 0, \\ 0.80, & s_f(t) = 1, \\ 1.00, & s_f(t) = 2 \text{ or } 3, \end{cases} \qquad r_f(t) = g_f(t) \times \text{BtlBw}_f(t).$$

Thus RTT 0 of each four-RTT cycle probes up with gain 1.25, RTT 1 drains with gain 0.80, and RTTs 2 and 3 cruise at line rate.

3. Model Enhancement

After describing the model, there are some enhancements that can be seen as *fixes* to the original model. The first point is the lack of enforcing wasted tokens to be greater or equal than zero. Technically, if wasted tokens are negative, it means the model violates the fixed upper bound service rate of C. Although this enforcement is already described in the paper (see Appendix, Section D), as we see in Section 2.1.4, the model does not enforce $W(0) \geq 0$. Also, as we see in the next section, in the output of the original model in response to the *surprising query* of AIMD, W is negative. Therefore, we add the following constraint in Section 2.1.4:

$$W(0) \ \geq \ 0.$$

The second enhancement is in Section 2.1.7. As we see, with the current implementation it is possible for the sender to detect a loss when there is no actual loss in the network. This *spurious* detection leads to an unnecessary decrease in the congestion window. Therefore, I added the following theorem in Section 2.1.7:

$$Ld_f(t) > 0 \implies L_f(t - R_m) > 0.$$

As we saw in AIMD (Section 2.2.1), the authors used a relaxed implementation to calculate when the model can increase the congestion window. I attempted to tighten the implementation by replacing $S_f(t)$ with $S_f(t - R_m)$, i.e., accounting for the propagation delay in that condition. However, with this change broke the surprising query and it returned unsat.

In the BBR implementation (Section 2.3), although the CCAC model is far more abstract than the real BBR algorithm, I adjusted three parameters to bring it closer to the reference design: the pacing-gain cycle length was increased to 8 RTTs (from 4), the "drain"-phase gain was set to 0.75 (instead of 0.8), and the sliding window for computing the bottleneck bandwidth was extended to 10 RTTs (instead of 4).

All of these enhancements (and the behavior-query tweaks described in the next section) can be toggled via `c.enhancement=True/False`. I had also planned to experiment with multi-flow scenarios and add sender-side flow control, but found the CCAC codebase not yet robust enough to accommodate those extensions.

4. Evaluations

This section describes the experiements I did with AIMD and BBR implementations. As I describe the original experiements the authors did and what I did, I also discuss how I enhanced queries to extend the experiements.

4.1. AIMD Evaluations

Probably the most important contribution of the paper is uncovering the *surprising* behaviour of AIMD. In classical textbook analysis of loss-based congestion control, loss is expected only when Inflight $> \mathrm{BDP} + \beta$. With a single flow and parameters

$$\beta = 2, \qquad C = 1, \qquad R_m = 1, \qquad D = 1,$$

we have $\mathrm{BDP} = C{\cdot}R_m = 1$, so the classical bound is Inflight > 3. Under CCAC's more expressive path model the queue may inflate by an extra D, giving the tighter bound Inflight $> C{\cdot}(R_m + D) + \beta = 4$.

Either way, after a loss we would normally expect the congestion window to settle above 1.5 (half of 3). The authors therefore posed the question: *Can CCAC force AIMD's window all the way down to* ≈ 1 *even when* $\beta = 2\,\mathrm{BDP}$*?* They encoded the following SMT query and asked Z3 for a witness:

$$\begin{aligned}\exists\, t \in \{2, \ldots, T-2\}.\ & cwnd_f(t) \le 2 \\ & \wedge Ld_f(t{+}1) - Ld_f(t) \ge 1 \\ & \wedge S(t{+}1 - R_m) - S(t - R_m) \ge C + 1 \\ & \wedge A(t{+}1) \ge A(t) + 2 \\ & \wedge L(t{+}1) > L(t) \\ & \wedge \bigwedge_{u=0}^{T-1} \neg\, \mathrm{timeout}_f(u)\end{aligned}$$

The conjuncts mean:

- $cwnd_f(t) \le 2$: the congestion window is already ≤ 2 packets;
- Ld_f grows by ≥ 1: the sender detects ≥ 1 new loss;
- at least two BDP of ACKs arrives in $[t - R_m,\ t{+}1 - R_m]$;
- the sender therefore injects ≥ 2 new bytes in $[t, t{+}1]$;
- the network actually drops more bytes during that step;
- and *no* timeout occurs anywhere in the trace.

If Z3 finds such a t (it does), the witness shows a sequence where AIMD halves twice in rapid succession, driving the window below 1.5 BDP, confirming the counter-intuitive behaviour.

The results of running the original query are summarised in Table 2. Please note in the CCAC, α and dupack was free variables. The solver finds a satisfying assignment with the first interesting point at $t = 5$. At $t = 2$ the sender has injected four data bytes, pushing the inflight above the safe limit and causing the network to drop $L(t) - L(t{-}1) = 1.0063$ bytes. The duplicate-ACK threshold is not met until $t = 5$; at that instant Ld_f rises, the loss is declared, and AIMD halves its congestion window to 2 bytes. A moment later, at $t = 6$, the sender finishes detecting the remainder of the lost burst, but since those packets were transmitted *before* the previous halving it is not allowed to halve again. At the same time two bytes of ACKs arrives ($S(t+1-R_m) - S(t-R_m) = 2$), and the sender sends two bytes on the wire (one retransmission plus one fresh packet). The burst occurs *at* $t = 6$, and it overflows the queue a second time, producing a new loss that is detected two RTTs later ($t = 8$). At that point the window drops from its slightly-grown value of about 2.1 to ≈ 1.05 bytes, confirming that AIMD can indeed drive cwnd well below the classical $1.5\,\mathrm{BDP}$ bound even with $\beta = 2\,\mathrm{BDP}$.

Time	W	S	A	L	LD_f	$cwnd_f$	r_f
0	-0.2875	0.0000	0.0063	0.0000	0.0000	3.9000	100.0000
1	-0.2875	1.2875	4.0000	1.0063	0.0000	4.0000	100.0000
2	-0.2875	1.2938	5.2875	1.0063	0.0000	4.0000	100.0000
3	-0.2875	2.3938	5.2938	1.0063	0.0000	4.0000	100.0000
4	-0.2875	3.2875	6.3938	1.0063	0.0000	4.0000	100.0000
5	-0.2875	5.2875	6.3938	1.0063	0.0063	2.0000	100.0000
6	-0.2875	6.2438	8.3938	1.0188	1.0063	2.1000	100.0000
7	-0.2875	7.2875	9.3500	1.0250	1.0063	2.1000	100.0000
8	-0.2875	7.2875	9.3563	1.0250	1.0188	1.0500	100.0000

Table 2: The surprising behavior reported by the authors. $\alpha = 0.1, \beta = 2, C = D = R_m = 1$

There are some caveats to these results. First of all, the results do not match the paper's timing of events. The paper explicitly specified that the burst happens at $t = 7$, but we see the burst happen at $t = 6$. Then, as we observe, the value of W is negative, which, as described in the previous section, is not reasonable. We also observe an increase in congestion window from $t = 6$ to $t = 7$, which I think is not justified. Maybe it is because of the relaxed checking conditions described in Section 2.2.1.

Therefore, I enhanced the network model as described above. I applied the constraint $W(0) \geq 0$ and added $Ld_f(t) > 0 \implies L_f(t - R_m) > 0$ to the model. I also enforced that the duplicate ACK threshold is always 3α, to remove unnecessary small or large values and match reality. Finally, I explicitly added $\exists\, t_1 \in \{t{+}1, \ldots, T\} : \mathrm{cwnd}_f(t_1) < 2$ to the query. The results are shown in Table 3.

Time	W	S	A	L	LD_f	$cwnd_f$	r_f
0	0.0000	0.0000	0.0125	0.0000	0.0000	3.8250	100.0000
1	0.0000	0.3000	3.8250	1.1125	0.0000	3.8250	100.0000
2	0.0000	1.0000	4.1250	1.1125	0.0000	3.8250	100.0000
3	0.0000	3.0000	4.1250	1.1125	0.0125	1.9125	100.0000
4	0.0000	3.0125	6.1250	1.1250	1.1125	2.0125	100.0000
5	0.0000	5.0000	6.1375	1.1250	1.1125	2.0125	100.0000
6	0.0000	5.0000	8.1250	1.1250	1.1125	2.0125	100.0000
7	0.0000	6.9875	8.1250	1.1250	1.1125	2.0125	100.0000
8	0.0000	7.9938	9.1188	1.1250	1.1250	1.0063	100.0000

Table 3: The results of the enahanced model, $\alpha = 0.1, \beta = 2, C = D = R_m = 1$

The changes are easy to notice. First, we still observe the surprising behavior, but this time the query is satisfied at $t = 3$, the burst occurs at $t = 4$, and at $t = 8$ the congestion window becomes less than 1.5. Second, we observe that no negative wasted tokens are reported. Therefore, the surprising behavior still holds.

Another finding of the paper is the steady-state bounds the authors *guessed* and then posed to the solver as an SMT query. A *steady state* here means (i) the system always reaches it, regardless of the initial conditions, and (ii) once reached, the system never leaves it. The authors expressed their conjectured bounds as

$$\begin{aligned}
&(L_f(0) - Ld_f(0) \; < \; C\,(R_m + D) + \alpha) \\
&\;\wedge\; (\mathrm{cwnd}_f(0) \; < \; C\,(R_m + D) + \beta + \alpha) \\
&\;\wedge\; (3\alpha \; < \; C\,R_m) \\
&\Longrightarrow \\
&(L_f(T) - Ld_f(T) \; < \; C\,(R_m + D) + \alpha) \\
&\;\wedge\; (\mathrm{cwnd}_f(T) \; < \; C\,(R_m + D) + \beta + \alpha).
\end{aligned}$$

To validate the conjecture, they added the *negation* of the right-hand side to the SMT model and asked the solver whether the formula is satisfiable. Because the solver returned unsat, the authors concluded that the bounds hold at time T.

However, steady-state bounds should hold *for all future timesteps*, not just at $t = T$. I therefore tightened the query so that it fails if the system ever leaves the bounds at any time $t \in \{2, \ldots, T-1\}$:

$$\begin{aligned}
&(L_f(0) - Ld_f(0) \; < \; C\,(R_m + D) + \alpha) \\
&\;\wedge\; (\mathrm{cwnd}_f(0) \; < \; C\,(R_m + D) + \beta + \alpha) \\
&\;\wedge\; (3\alpha \; < \; C\,R_m) \\
&\;\wedge\; \Big(\bigvee_{t=2}^{T-1} \Big(\mathrm{cwnd}_f(t) \; > \; C\,(R_m + D) + \beta + \alpha\Big)\Big).
\end{aligned}$$

The results are shown in Table 4. As illustrated, the bound for the congestion window is $1(2) + 1 + 0.305 = 3.305$. Although at $t = T = 9$ the system satisfies the steady-state bounds, the system violates these bounds between $t = 1$ and $t = 3$. In addition, two timeout event happens at 6 and 9. Therefore, it seems that the steady state does not hold.

I further investigated this issue by improving the query and testing it with the enhanced model. In particular, I added constraints to ensure that no timeout happens, the duplicate acknowledgements are exactly 3α, and that the steady-state bounds are satisfied at both $t = 0$ and $t = 1$. I reran the experiment with the same parameters, and the model returned *unsat*.

Does this mean the steady-state bound holds? To further verify, I performed another experiment with the parameters shown in Table 5. As shown, the model returned *sat*, and the system exits the boundary condition at $t = 8$. Therefore, the steady state bounds are not valid.

4.2. BBR Evaluations

For evaluating BBR, the authors asked a simple question: is it possible for the utilization to fall below 10%? They formulated it as follows:

$$S(T) - S(0) < 0.1C \cdot T$$

Time	W	S	A	L	LD_f	$cwnd_f$	r_f
0	-1.0000	0.0000	2.0278	0.0278	0.0000	3.0278	100.0000
1	-1.0000	1.9722	3.3333	1.3333	0.0000	3.3333	100.0000
2	-1.0000	2.7222	5.3056	1.3333	0.0000	3.3333	100.0000
3	-1.0000	3.0000	6.0556	1.3611	0.0000	3.3333	100.0000
4	-0.6944	4.6667	6.0556	1.3611	1.3333	1.6667	100.0000
5	-0.6944	5.6944	7.9722	2.2778	1.3333	1.9722	100.0000
6	0.0000	5.7222	8.2778	2.2778	2.2778	0.3056	100.0000
7	0.9722	6.0000	8.3056	2.2778	2.2778	0.3056	100.0000
8	1.6944	6.3056	8.5833	2.2778	2.2778	0.3056	100.0000
9	1.6944	6.3056	8.8889	2.2778	2.2778	0.3056	100.0000

Table 4: The orignial model results for steady-state with $\alpha = 0.3055, \beta = 1, C = D = R_m = 1$

Time	W	S	A	L	LD_f	$cwnd_f$	r_f
0	0.0000	0.0000	0.0278	0.0000	0.0000	5.0278	100.0000
1	0.0000	0.0000	6.1389	3.1389	0.0000	5.0278	100.0000
2	0.0000	1.1389	6.1389	3.1389	0.0000	5.0278	100.0000
3	0.0000	2.0000	6.1389	3.1389	0.0000	5.0278	100.0000
4	0.1389	3.0000	6.1667	3.1389	0.0000	5.0278	100.0000
5	1.1111	3.8611	7.0278	3.1389	0.0000	5.0278	100.0000
6	1.1111	3.8889	8.0278	3.1389	0.0000	5.0278	100.0000
7	1.2500	5.0278	8.8889	3.1389	0.0000	5.0278	100.0000
8	1.2500	5.7500	9.2222	3.1389	0.0000	5.3333	100.0000
9	1.2500	6.7500	10.8333	3.1389	3.1389	2.6667	100.0000

Table 5: Results of the enhanced model with $\alpha = 0.3056, \beta = 2, C = D = 1, R_m = 2$

and the model returned `sat`, as shown in Table 6.

Time	W	S	A	L	LD_f	$cwnd_f$	r_f
0	-0.0064	0.0000	0.0064	0.0000	0.0000	0.1197	0.0748
1	0.9107	0.0479	0.0543	0.0000	0.0000	0.1197	0.0479
2	1.8692	0.0893	0.1021	0.0000	0.0000	0.0957	0.0479
3	2.8094	0.1308	0.1500	0.0000	0.0000	0.0957	0.0479
4	3.7496	0.1906	0.2098	0.0000	0.0000	0.0957	0.0598
5	4.7423	0.2504	0.2577	0.0000	0.0000	0.1197	0.0479
6	5.6825	0.2974	0.3175	0.0000	0.0000	0.1197	0.0598
7	6.6226	0.3325	0.3774	0.0000	0.0000	0.1197	0.0598
8	7.5479	0.3859	0.4521	0.0000	0.0000	0.1197	0.0748
9	8.5000	0.4936	0.5000	0.0000	0.0000	0.1197	0.0479

Table 6: BBR low utilization results $C = D = R_m = 1$

As shown, problems similar to those in AIMD exist, such as $W(0)$ being negative. To address this, I applied my proposed enhancement, which includes changes to the BBR implementation as described above, and reran the experiments. The results are shown in Table 7. As we can see, the system served only 0.99 bytes over 10 seconds, which corresponds to less than 10% utilization. Therefore, the authors' conclusion that BBR can experience very low utilization is correct.

Time	W	S	A	L	LD_f	$cwnd_f$	r_f
0	0.0000	0.0000	0.0000	0.0000	0.0000	0.2348	0.1174
1	0.8532	0.0294	0.1468	0.0000	0.0000	0.2348	0.1468
2	1.8532	0.1468	0.1468	0.0000	0.0000	0.0587	0.0220
3	2.7358	0.2621	0.2642	0.0000	0.0000	0.2348	0.1174
4	3.5136	0.3753	0.3816	0.0000	0.0000	0.2348	0.1174
5	4.4067	0.4864	0.4990	0.0000	0.0000	0.2348	0.1174
6	5.3836	0.5933	0.6164	0.0000	0.0000	0.2348	0.1174
7	6.1928	0.6982	0.7338	0.0000	0.0000	0.2348	0.1174
8	7.1488	0.8072	0.8512	0.0000	0.0000	0.2348	0.1174
9	8.0021	0.9979	0.9979	0.0000	0.0000	0.2348	0.1468

Table 7: BBR results for the enhanced model $C = D = R_m = 1, \beta = 2$

5. Conclusion

This paper investigated the formal analysis of congestion control algorithm behavior. I examined the implementation of the network and algorithms in detail, and reviewed their weaknesses. I then proposed enhancements to improve the modeling. After evaluating the models and algorithms, I found that, for AIMD, the surprising behavior, where premature loss can lead to a very low congestion window despite the presence of a large bottleneck buffer, is correct. However, the steady-state bounds reported in the original paper are not valid. For BBR, the reported behavior that BBR can achieve very low utilization is also confirmed.

The paper studied simplified versions of the algorithms and network, and many conditions remain to be modeled. These include network behavior under random loss or intermittent connections, multi-flow interactions, and more detailed implementations of algorithms. Future work can incorporate these aspects to create a more comprehensive model.

References

Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based Congestion Control for the Internet. In *Proceedings of the 2018 Applied Networking Research Workshop*, ANRW '18, page 19, New York, NY, USA, July 2018. Association for Computing Machinery. ISBN 978-1-4503-5585-8. doi: 10.1145/3232755.3232783. URL https://doi.org/10.1145/3232755.3232783.

Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 1–16, New York, NY, USA, August 2021. Association for Computing Machinery. ISBN 978-1-4503-8383-7. doi: 10.1145/3452296.3472912. URL https://dl.acm.org/doi/10.1145/3452296.3472912.

Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, October 2016. ISSN 1542-7730. doi: 10.1145/3012426.3022184. URL https://dl.acm.org/doi/10.1145/3012426.3022184.

Anei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. Request for Comments RFC 6582, Internet Engineering Task Force, April 2012. URL https://datatracker.ietf.org/doc/rfc6582. Num Pages: 16.

Huiling Jiang, Qing Li, Yong Jiang, GengBiao Shen, Richard Sinnott, Chen Tian, and Mingwei Xu. When machine learning meets congestion control: A survey and comparison. *Computer Networks*, 192:108033, 2021. ISSN 1389-1286.