ETH zürich

Networked Systems
ETH Zürich — seit 2015

More Bytes Please

Tricking BBR into Sending Faster Than It Should

Master Thesis

Author: Nicola Manuel Uetz

Tutor: Ayush Mishra

Supervisor: Prof. Dr. Laurent Vanbever

February 2026 to August 2026

Abstract

BBR estimates a connection's bandwidth-delay product purely from its own RTT samples, trusting the receiver to acknowledge packets as soon as they arrive, and prior work already shows it systematically favours longer-RTT flows over shorter-RTT ones at a shared bottleneck. In this thesis, we build `delay_acks`, a client-side tool that deliberately delays a client's outgoing ACKs to inflate this RTT sample, and systematically test how far a receiver can exploit this bias across varying bottlenecks, against both BBR and CUBIC contenders, and with realistic traffic on the real Internet. The effect is strongest against other BBR flows at low base RTT with deep buffers, and is far less reliable against CUBIC. It survives contact with realistic workloads too: `delay_acks` measurably shortens file-download completion time and improves YouTube playback progress against a real competing flow at a shared controlled bottleneck. A no-contender experiment then shows where that gain actually comes from: with nobody to take bandwidth from, inflating your own RTT never helps and only costs latency.

Contents

1 Introduction **1**
- 1.1 Contributions . . . 1
- 1.2 Key Findings . . . 2
- 1.3 Thesis Outline . . . 2

2 Related Work **4**
- 2.1 RTT Unfairness in Traditional Congestion Control Algorithms . . . 4
- 2.2 RTT Unfairness in BBR . . . 5
- 2.3 Exploiting RTT Unfairness in BBR . . . 5
- 2.4 BBR vs. CUBIC . . . 6

3 Background & Motivation **7**
- 3.1 The BBR Congestion Control Algorithm . . . 7
  - 3.1.1 Bandwidth and RTT Estimation . . . 7
  - 3.1.2 State Machine: `STARTUP`, `DRAIN`, `PROBE_BW`, `PROBE_RTT` . . . 8
  - 3.1.3 From RTT Sample to Send Rate: The BDP-Inflation Mechanism . . . 8
- 3.2 BBR Version History . . . 9
  - 3.2.1 `PROBE_RTT`: BBRv1 vs. BBRv2/v3 . . . 9
- 3.3 Motivation . . . 10

4 System Design **11**
- 4.1 Goal and Concept . . . 11
- 4.2 The ACK-Delaying Client . . . 12
  - 4.2.1 Release Strategies . . . 14
  - 4.2.2 Logging . . . 14

5 Evaluation **15**
- 5.1 Experimental Setup . . . 15
  - 5.1.1 Emulated Network Topology . . . 15
  - 5.1.2 Emulating Delay, Bandwidth, and Buffers . . . 16
  - 5.1.3 Traffic Capture and Analysis Pipeline . . . 16
  - 5.1.4 Workload Generators . . . 17
- 5.2 Studying RTT Unfairness . . . 17
  - 5.2.1 Effect of Base RTT . . . 18
  - 5.2.2 Effect of Buffer Size . . . 22
  - 5.2.3 Effect of Bandwidth . . . 26
  - 5.2.4 Effect of Varying Bottleneck at Constant Buffer/BDP Ratio . . . 30
  - 5.2.5 Effect of Receive/Send Window Scaling . . . 36

5.3 File Download Completion Time . . . 41
5.3.1 Packet and ACK Release Delay . . . 45
5.3.2 Per-Flow Dynamics . . . 51
5.4 Video Playback . . . 53
5.4.1 Packet and ACK Release Delay . . . 58
5.4.2 Per-Flow Dynamics . . . 64
5.5 No Contender & Unconstrained . . . 66

6 Discussion & Conclusion & Future Work 69
6.1 Discussion . . . 69
6.2 Conclusion . . . 70
6.3 Future Work . . . 71
6.3.1 Why Do the Curves Have This Shape? A Proposed Delay Decomposition . . . 71
6.3.2 Sensitivity to the `PROBE_RTT` Variant . . . 72
6.3.3 A Secondary Benefit for the Client's Own Outgoing Traffic . . . 73
6.3.4 Robustifying `delay_acks` . . . 73

References 74

A Additional Data I
A.1 Buffer/BDP Ratio Sweep – Full Peak Data . . . I
A.2 Window Scale Sweep – Full Peak Data . . . III

Chapter 1

Introduction

Two identical browser tabs open the same YouTube video at the same moment, behind the same bottleneck link. Thirty seconds later, one has advanced to 0:20 into the clip; the other only to 0:13 (Figure 5.30). Neither client touches the server, the network path, or TCP itself. The only difference is that the faster client runs a small program that decides, packet by packet, when to send back the acknowledgements (ACKs) it was going to send anyway.

That program's name is `delay_acks`. A BBR (Bottleneck Bandwidth and Round-trip propagation time) sender estimates its RTT (Round-Trip Time) purely from how long its own packets take to be acknowledged; `delay_acks` sits on the receiving client and chooses when those acknowledgements actually leave the machine. Holding an ACK back a few extra milliseconds inflates the sender's next RTT sample, which BBR takes at face value, uses to estimate a larger BDP (Bandwidth-Delay Product), and rewards with more data in flight than the client was actually owed – the trick behind this thesis's title, "More Bytes Please – Tricking BBR into Sending Faster Than It Should."

This matters beyond a neat demonstration because BBR is not a niche algorithm: by 2024 it carried all of Google.com and YouTube's public-facing TCP traffic [4]. Prior work already shows BBR systematically favours longer-RTT flows over shorter-RTT ones sharing a bottleneck (Section 2.2). Things get worse once you find out that nothing in how BBR estimates the RTT between server and client lets the sender verify the path's true propagation delay or lets the sender know whether an ACK was delayed by the network or deliberately by the receiver. That raises the natural question this thesis pursues: does a design that quietly trusts the receiver's ACK timing hold up against a receiver with no reason to play fair?

This isn't a new question. Ma et al. [10] already demonstrated this "latency-cheating" effect experimentally (Section 2.3), so the core idea is not this thesis's novelty. What their demonstration left open though is whether the idea survives outside the virtual testbed and at one fixed experimental setup. This thesis is the systematic follow-up: testing the idea against varying bottlenecks, contending against BBR or CUBIC flows, and using realistic application traffic on the real Internet, to find out when the trick actually pays off and when it does not.

1.1 Contributions

This thesis makes the following contributions:

- `delay_acks` (Chapter 4): a client-side tool that intercepts a client's own outgoing TCP packets via an NFQUEUE hook and controls exactly when their ACKs leave the machine, with three

release strategies (Section 4.2.1) trading off how far the sender's RTT sample gets inflated against how much extra latency the client's own outgoing traffic absorbs.

- A synthetic throughput-share study (Section 5.2) that maps out, without `delay_acks` itself, how BBR's underlying RTT-unfairness bias depends on base RTT, buffer size, bandwidth, and window scaling, against both BBR and CUBIC contenders.
- Studying `delay_acks`'s effectiveness against realistic workloads – file downloads and YouTube video playback (Sections 5.3 and 5.4) – under a bottleneck we still shape and control, downloading from and streaming from Google's real production servers rather than a synthetic sender, and a no-contender study (Section 5.5) that removes both the rival flow and the emulated bottleneck.

1.2 Key Findings

Within the controlled RTT-unfairness study (Section 5.2), the effect is strongest exactly where the underlying mechanism predicts it should be: against other BBR flows, at low base RTT, with a buffer deep enough relative to the path's bandwidth-delay product to actually sustain the inflated in-flight window BBR's model calls for – up to 10.57× fair share against a single BBR contender at 5 ms base RTT, decaying back toward fair as base RTT grows, though at that same low base RTT adding *more* BBR contenders makes the relative unfairness more pronounced rather than diluting it. Window scaling turns out not to be the real constraint on either side; what governs the outcome throughout is the buffer relative to the BDP, not any absolute buffer size or bandwidth value on its own (Sections 5.2.2 to 5.2.5). That dependency is also why the effect is far less reliable against CUBIC contenders: it always beats not inflating RTT at all, but only clearly outperforms CUBIC's own loss-based growth in shallow-to-moderate buffers, and can fall below a fair share once the buffer or the bandwidth grows large enough that CUBIC catches up regardless of how far `client-2`'s RTT is inflated.

This advantage survives contact with realistic traffic. Over the real Internet, against a single contender, `delay_acks` measurably shortens file-download completion time (Section 5.3) and lets `client-2` advance further into the same YouTube video within the same fixed real-time window (Section 5.4) – the 0:20 vs. 0:13 gap from the opening of this chapter. But the no-contender experiment (Section 5.5) makes clear where that gain actually comes from: with no rival flow to compete against, file download completion time never drops below its own baseline for any of the three release strategies, no matter how far the target RTT is inflated. Inflating your own RTT is a double-edged sword: the gains measured in Sections 5.3 and 5.4 come specifically at a competing flow's expense; with nobody to take bandwidth from, inflating your RTT never buys anything back – it only ever costs latency.

1.3 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 surveys prior work on RTT unfairness, both in classical loss-based congestion control and in BBR itself, and the existing work that first showed this unfairness could be deliberately triggered. Chapter 3 covers the BBR mechanics this thesis relies on: how BBR estimates bandwidth and RTT, its state machine, and the BDP-inflation mechanism the attack exploits, together with the version history relevant to interpreting later results. Chapter 4 presents `delay_acks` itself, including its three release strategies. Chapter 5 contains the full empirical evaluation backing the headline result above, from the controlled synthetic

study through realistic workloads to the no-contender experiment. Chapter 6 discusses the results, their limitations, and directions for future work.

Chapter 2

Related Work

This chapter surveys the prior work this thesis builds on. Section 2.1 covers RTT unfairness in classical, loss-based congestion control such as CUBIC, where the bias runs opposite to BBR's: shorter-RTT flows are the ones favoured. Section 2.2 covers the well-documented opposite bias in BBR itself, which favours longer-RTT flows and is the phenomenon this thesis sets out to exploit. Section 2.3 covers existing work that goes one step further and demonstrates this bias can be deliberately triggered by manipulating ACK timing, including the specific finding this thesis takes as its starting point. Finally, Section 2.4 covers the more general interaction between BBR and CUBIC at a shared bottleneck, relevant because any real deployment of this thesis's attack will typically compete against a mix of both algorithms rather than against other BBR flows alone.

2.1 RTT Unfairness in Traditional Congestion Control Algorithms

There has been significant research into RTT unfairness in traditional, loss-based congestion control, spanning well over ten years. These works reveal a bias that runs in a specific direction, and, importantly for this thesis, in the *opposite* direction to the one BBR exhibits. Since Additive Increase Multiplicative Decrease (AIMD) algorithms update their congestion window once per round-trip time of a packet, flows with a shorter RTT can grow their congestion window generally more frequently compared to flows with a longer RTT, resulting in an advantage at a shared bottleneck in the network. Algorithms like CUBIC try to mitigate this issue by defining their congestion window growth "in terms of the elapsed time since the last loss event" [7], making window growth independent of RTT.

Kozu, Akiyama and Yamaguchi [9] were among the first to quantify how much of the classical bias survives this fix. Rather than relying on the network simulators most earlier fairness studies used, they measured the bandwidth share of two competing CUBIC flows using actual Linux TCP implementations connected through real network elements, across a range of RTT ratios. They found that CUBIC flows with shorter RTT can still increase their sending rate more rapidly than longer-RTT competitors. They trace this to CUBIC's parameter K. They propose shrinking this parameter K as RTT grows, which restores much better fairness in their tests. Their specific fix is not directly relevant to the goals of this thesis, which concerns BBR rather than CUBIC, but their finding confirms the direction of the classical bias.

More recently, Gomez, Kfoury, Crichigno, and Srivastava [6] address both directions of the bias at once, in a network carrying CUBIC and BBR flows at the same time. Instead of modifying either sender's algorithm, they use a P4-programmable switch to passively measure the RTT of every flow crossing an ordinary, non-programmable router and "rebalancing the router's queue according to

the RTT of TCP flows" [6]. This work is only loosely relevant to this thesis as it attempts to fix RTT unfairness from inside the network, whereas this thesis asks what an end host can do to a single BBR sender without any network-side help or interference in various network conditions.

2.2 RTT Unfairness in BBR

The exact opposite phenomenon – an RTT unfairness in favour of longer-RTT flows – remains a heavily researched topic in Internet Congestion Control, one this thesis builds directly on.

Ma, Jiang, Wang, and Li [10] reported that after deploying BBR in a geo-distributed cloud, "long BBR flows are bound to seize bandwidth from short ones" [10]. According to their findings, a longer-RTT flow injects proportionally more excess traffic into the bottleneck queue during BBR's periodic bandwidth-probing phase: the resulting overshoot is sustained for a full round trip, so a longer RTT simply means more absolute bytes queued per probe. To make things worse, the queues don't get drained fully. Their findings highlight that this bias develops in two phases: "a persistent queue forms on the bottleneck due to the excess traffic" as BBR probes for more bandwidth, after which "the long RTT flow probes for longer time than the short RTT flow, and hence floods in more excess traffic, dominating the queue backlog while overwhelming the short RTT flow" [10]. They propose a modified version of BBR called BBQ which limits the probing period to some fixed parameter α which results in both flows contributing to the bottleneck queue in a similar fashion without one flow utilising significantly more. This finding is highly relevant to the goals of this thesis because understanding how this persistent queue forms is fundamental to analysing BBR's sending behaviour and addressing it at the end host.

Hock, Bless and Zitterbart [8] and Scholz et al. [12] independently confirmed the bias Ma et al. [10] reported. The former states that as a result of BBR's model not reflecting "the perspective of an individual sender if multiple flows are traversing the bottleneck [...] each sender overestimates the available bandwidth leading to a too high total $D^{inflight}$." [8] They predict that $2 * BDP \leq \sum_i D_i^{cap} < 2.5 * BDP$ where D_i^{cap} is the inflight cap of flow i [8]. This finding is highly relevant to us because it quantitatively establishes the bottleneck overload mechanics that our thesis aims to exploit.

2.3 Exploiting RTT Unfairness in BBR

On top of the studies made on RTT unfairness in BBR itself covered in Section 2.2, some work has already been made on exploiting the phenomenon.

Ma et al. [10] do not just document BBR's bias towards long RTT flows, they also present one critical consequence. "Because BBR is a sender-based congestion control" [10] that estimates RTT purely from received acknowledgements, they note that "it would be very hard, if not impossible, for the sender to tell if the probed RTT has been manipulated by receivers." [10] They call this "latency-cheating" [10] and demonstrate it experimentally: two BBR flows with a true RTT of 5 ms take turns delaying their own inbound traffic to inflate their apparent RTT to twice that of the other. "Because a flow can always delay its traffic to steal more bandwidth from the other" [10], the two flows drive each other's apparent RTT up exponentially in what the authors call a "race-to-the-bottom" [10]. Similar remarks appear in later work: Zhang [13] notes that "BBR can be easily manipulated by malicious receiver for data transmission acceleration by delaying the acknowledge packets." [13]. This finding is directly relevant to the goals of this thesis because we will be directly building on top of their finding and it proves that we can indeed achieve an increase in throughput by inflating our RTT.

2.4 BBR vs. CUBIC

The interaction between BBR and CUBIC flows remains a complex phenomenon in Internet Congestion Control. These works reveal the following key findings about bandwidth sharing between distinct algorithms.

Hock et al. [8] and Scholz et al. [12], already discussed above for their RTT findings, also report that particularly in shallow buffers, BBR generally claims a disproportionate share of the bottleneck bandwidth from competing CUBIC flows. Claypool, Claypool, Chung, and Li [5] confirm this pattern across a range of buffer sizes and loss rates, and summarise it in the title of their paper: BBR and CUBIC are "sharing but not caring" [5]. BBR's probing for more bandwidth in shallow buffers results in packet loss which CUBIC responds to by reducing its congestion window but BBR does not. This finding is relevant for our thesis as we will naturally be competing against flows with all sorts of congestion control algorithms when trying to trick the network to give us more throughput than our fair share.

Mishra, Tiu, and Leong [11] present a model to compute the Nash equilibrium of a population of flows choosing between CUBIC and BBR, i.e., the distribution at which no individual flow can improve its own throughput by unilaterally switching algorithm. In other words, the more other flows switching to BBR, the smaller the gain of switching to BBR yourself and at some point it might even become better to switch back to CUBIC as they show the Nash equilibrium lying in between both extrema. The fact that it is highly unlikely for all TCP flows to eventually end up being BBR is relevant for us as it confirms that we should look into both BBR and CUBIC contender traffic when doing our simulations.

Chapter 3

Background & Motivation

This chapter lays the technical groundwork the rest of the thesis relies on, and motivates why it is worth writing in the first place. Section 3.1 introduces BBR's core bandwidth- and RTT-estimation model and its four-state state machine, and from there derives the BDP-inflation mechanism this thesis's attack exploits. Section 3.2 covers BBR's version history from BBRv1 through BBRv3, with particular attention to how the `PROBE_RTT` state's congestion-window cut has evolved between versions, a detail needed later to interpret the per-flow dynamics observed in Chapter 5. Finally, Section 3.3 draws on both this chapter and Chapter 2 to motivate the specific question this thesis's design (Chapter 4) and evaluation (Chapter 5) set out to answer.

3.1 The BBR Congestion Control Algorithm

BBR is a congestion control algorithm developed by Google, first published in 2016 [2]. Where classical loss-based algorithms such as CUBIC or Reno infer the network's capacity from packet loss or queuing delay, BBR instead builds an explicit model of the path by continuously estimating the two physical quantities, bottleneck bandwidth (`BtlBw`), and round-trip propagation time (`RTprop`) [2]. The product of the two, called the bandwidth-delay product BDP = $\texttt{BtlBw} \times \texttt{RTprop}$, is the amount of data that exactly fills the path without creating a standing queue at the bottleneck. BBR's stated goal is to operate as close as possible to this point: enough data in flight to keep the bottleneck link fully utilised, but no more than necessary, since any excess simply queues at the bottleneck and adds delay without adding throughput [2]. Section 2.2 already covers how well this goal is met in practice once multiple flows with different RTTs compete for the same bottleneck; the remainder of this section covers the mechanics needed to understand *why* that competition plays out the way it does.

3.1.1 Bandwidth and RTT Estimation

Both `BtlBw` and `RTprop` are estimated purely from the sender's own stream of packet departures and the ACKs that arrive back for them, without any explicit signalling from the network or the receiver [2]. Every ACK a BBR sender receives yields two raw samples: an RTT sample (the time between sending the packet and receiving its ACK back from the receiver) and a delivery-rate sample (the amount of data delivered since that packet was sent, divided by the elapsed time). Neither raw sample is used directly; both are filtered for noise instead. For the estimated RTT, the smallest RTT observed over a trailing window "which is typically tens of seconds to minutes" [2] is taken as the true `RTprop`. `BtlBw`, conversely, is estimated as the largest delivery rate observed over

a trailing window of several round trips, since a measured delivery rate can only ever underestimate the true bottleneck capacity [2].

These two estimates cannot both be measured cleanly at the same time. Whenever the sender has less than one BDP in flight, only `RTprop` is visible, since the pipe is not yet full enough to reveal its capacity; once more than one BDP is in flight, the bottleneck queue grows and delivery rate saturates at `BtlBw`, but the growing queue now inflates every further RTT sample so `RTprop` can no longer be measured cleanly [2]. BBR's four operating states, covered next, exist largely to manage this trade-off: to keep both estimates fresh, the sender has to deliberately alternate between probing for more bandwidth (pushing inflight above the current BDP estimate) and briefly draining the pipe back down (to get an uncontaminated `RTprop` sample).

3.1.2 State Machine: `STARTUP`, `DRAIN`, `PROBE_BW`, `PROBE_RTT`

BBR structures all of its behaviour, from connection startup to steady-state operation, as transitions between four states, each of which fixes a `pacing_gain` (a multiplier applied to the `BtlBw` estimate to set the sender's pacing rate) and a `cwnd_gain` (a multiplier applied to the current BDP estimate to bound how much data may be in flight at once), together with a condition for when to move to the next state [2].

`STARTUP` The state a connection begins in. To discover `BtlBw` quickly across the roughly twelve orders of magnitude of link speeds found on the Internet, `STARTUP` doubles its sending rate every round trip. It exits once the measured delivery rate stops growing by more than 25 % for three consecutive round trips, at which point BBR assumes the bottleneck has been found [2]. This search overshoots and builds up queues in the network.

`DRAIN` Immediately follows `STARTUP` and exists solely to remove the queue `STARTUP` built up. It paces at the reciprocal of `STARTUP`'s gain until the amount of data in flight matches the BDP estimate. After the queues have been drained that way it hands off to `PROBE_BW` [2].

`PROBE_BW` The steady operating state in which a connection spends "about 98 percent" [2] of its time. The sender cycles its `pacing_gain` through a repeating sequence of phases. Most of the time it will be pacing at the current `BtlBw` estimate but periodically probes for more bandwidth, immediately followed by a compensating phase with a gain slightly below `BtlBw` to drain the queue built up by the preceding gain increase [2].

`PROBE_RTT` Entered whenever the `RTprop` estimate hasn't been updated by a new minimum for longer than its filter window. The sender deliberately shrinks its congestion window far below the current BDP estimate for a short interval to empty the bottleneck queue and obtain a clean, uncontaminated RTT sample before returning to either `STARTUP` or `PROBE_BW`, depending on whether the pipe is already estimated to be full [2]. The exact size of this cut and how long the filter window is have changed between BBR versions which will be covered in Section 3.2.1 in detail, since we will also be looking at per-flow dynamics later in this thesis (e.g. Section 5.4.2).

3.1.3 From RTT Sample to Send Rate: The BDP-Inflation Mechanism

The mechanism this thesis exploits follows directly from how `RTprop` feeds into BBR's sending rate. On every send opportunity, BBR computes its current BDP estimate as

$$\widehat{\mathrm{BDP}} = \widehat{\texttt{BtlBw}} \times \widehat{\texttt{RTprop}},$$

which paces outgoing data at `pacing_gain` $\times \widehat{\texttt{BtlBw}}$, and permits at most `cwnd_gain` $\times \widehat{\text{BDP}}$ bytes in flight at any time before it pauses sending to wait for further ACKs [2]. After `STARTUP`, `cwnd_gain` is held at a small, fixed constant specifically so the sender can keep pacing smoothly even when ACKs arrive delayed or grouped together, rather than stalling every time this happens [2].

Crucially, $\widehat{\texttt{RTprop}}$ is not obtained from any ground truth about the path; it is nothing more than the smallest RTT the sender has itself measured from its own packets' departure and acknowledgement timestamps, over the sender's own recent history (Section 3.1.1). The sender has no way to independently verify this value against the path's actual propagation delay, and no mechanism in the protocol tells it that a given ACK was deliberately withheld rather than genuinely delayed by the network. Consequently, whatever value the receiver's ACK timing makes the sender observe as its minimum RTT becomes $\widehat{\texttt{RTprop}}$.

Since $\widehat{\text{BDP}}$ scales linearly with $\widehat{\texttt{RTprop}}$, and the sender's permitted in-flight budget scales linearly with $\widehat{\text{BDP}}$, an RTT sample inflated by any means at all directly and proportionally inflates how much data the sender is willing to keep in flight for that connection, without the underlying bottleneck's actual capacity having changed at all. At a bottleneck which is shared with other flows and a finite buffer, more in-flight data for one flow comes largely at the expense of the buffer's headroom, and hence the throughput share available to the other flows [10, 8]. This is precisely the RTT-unfairness bias Section 2.2 already documents occurring naturally between flows whose RTTs simply happen to differ sharing the same bottleneck. Chapter 4 describes `delay_acks`, the client-side tool this thesis builds to manufacture exactly this kind of inflated RTT deliberately at the receiver, purely by choosing when to release the ACKs it would otherwise have sent immediately without any changes to the sender, the network, or the TCP protocol itself.

3.2 BBR Version History

BBR has gone through three publicly released major versions since 2016. BBRv1 is the version described in the original publication and in Sections 3.1.1 to 3.1.3 above. Google released BBRv2 in 2018, adding Explicit Congestion Notification (ECN) marks and the packet loss rate as additional congestion signals alongside `BtlBw` and `RTprop`, specifically to address the inter-flow and inter-protocol fairness problems that had been observed with BBRv1 in the years after its release [1]. BBRv3 followed in 2023 as a refinement of BBRv2, continuing to target the same fairness and coexistence problems mostly through further tuning rather than a further change in the core estimation model [1]. By 2024, BBRv3 had become the congestion control algorithm Google uses for all Google.com and YouTube public-Internet traffic carried over TCP; QUIC (the UDP-based transport underlying HTTP/3) lagged behind, still limited to a small-percentage A/B experiment against BBRv1 as of the same slide deck [3, 4]. This thesis's attack, in any case, reaches only the TCP case. `delay_acks` (Chapter 4) manipulates ACKs at the TCP header level, a mechanism that does not extend to QUIC, whose acknowledgement frames are carried inside its encrypted payload rather than in a plaintext, packet-level field. Since a modern browser negotiates QUIC by default with Google's servers whenever it is offered, every file-download and video-playback experiment in Sections 5.3 to 5.5 explicitly disables QUIC in the client browser to force the connection onto BBR-over-TCP, the case this thesis actually targets.

3.2.1 `PROBE_RTT`: BBRv1 vs. BBRv2/v3

Of the changes between BBR versions, the one most relevant to this thesis is how aggressively `PROBE_RTT` (Section 3.1.2) cuts the congestion window, since this directly determines how deep

and how frequent the periodic throughput dips flows experience. In BBRv1, a connection enters `PROBE_RTT` if its `RTprop` estimate has not been refreshed by a new minimum for 10 seconds [1], and, once inside, cuts its congestion window to a hard floor of four packets (four times the maximum segment size), maintained "for at least 200 ms and one round trip" [2]. This floor is independent of the path's BDP: a connection with a 150-packet steady-state window is cut down to essentially the same near-empty four packets as one with a 15-packet window, which produces large, disruptive throughput dips regardless of the connection's actual operating point. BBRv2 and BBRv3 both shorten the staleness interval that triggers `PROBE_RTT` to 5 seconds, but soften the cut itself considerably. Instead of a fixed four-packet floor, the congestion window is only reduced to half of the current BDP estimate [1]. Halving the window rather than emptying it still lets the queue drain enough to obtain a clean `RTprop` sample, without causing the same throughput instability BBRv1 caused. This distinction matters directly for Section 5.4.2, which reports six regular throughput dips roughly 5.2–5.4 s apart, during which the competing flow's data in flight falls from a ≈145-packet baseline to 50–66 packets, i.e. to roughly half its baseline rather than down to a small, BDP-independent floor. Both the ≈5 s spacing and the fact that the cut only reaches roughly half the baseline rather than near zero are consistent with the BBRv2/v3 `PROBE_RTT` behaviour described above, not with BBRv1's harsher near-empty-pipe cut, which is consistent with Section 3.2's observation that Google's YouTube servers run BBRv3.

3.3 Motivation

BBR already accounts for a substantial and growing share of Internet traffic, including all of Google.com and YouTube's public-facing TCP traffic [4], so any systematic bias in how it shares bandwidth between competing flows is not a purely academic concern: it shapes real, everyday throughput outcomes at scale, at least for the substantial fraction of the traffic still carried over TCP rather than QUIC (Section 3.2). Section 2.2 already establishes, from prior work, that an RTT bias exists and runs in BBR's case in favour of longer-RTT flows, and Section 2.3 already establishes that at least one prior study has demonstrated, in a controlled two-flow experiment, that a receiver can deliberately inflate its own RTT to claim a disproportionate share of a bottleneck [10]. What remains open, and what this thesis sets out to close, is whether the same idea survives contact with conditions closer to how BBR is actually used in practice: competing against one or several CUBIC and BBR flows at varying bottlenecks, and delivering workloads with realistic traffic patterns (file downloads, video playback) rather than synthetic saturating flows, and, crucially, real-world networks and real BBR deployments rather than a fully controlled testbed alone. Chapter 4 and Chapter 5 address exactly this: a practical implementation of the attack (Chapter 4) and a systematic, end-to-end empirical evaluation of when inflating your RTT does and does not deliver a real advantage (Chapter 5).

Chapter 4

System Design

This chapter describes the design of `delay_acks`, the client-side tool this thesis builds to deliberately trigger BBR's RTT-unfairness bias (Sections 2.2 and 3.1.3) from an ordinary end host. Section 4.1 states the goal this design serves and sketches, at a conceptual level and before any implementation detail, how it achieves it. Section 4.2 then covers the tool's actual architecture, Section 4.2.1 the three concrete release strategies we worked out, and Section 4.2.2 the logging it produces for further analysis.

4.1 Goal and Concept

The goal of this thesis's system is exactly the one Section 3.3 motivates: a client-side tool that, without any cooperation from the sender and without any change to the network path or the TCP protocol itself, deliberately inflates the RTT the sender of the flow observes, in order to leverage BBR's RTT-unfairness bias (Section 2.2) and claim more than a fair share of a bottleneck shared with other flows – an idea already demonstrated in principle by Ma et al.'s "latency-cheating" [10] (Section 2.3), which this thesis turns into a practical, evaluable tool. Figure 4.1 sketches the idea: `C2` deliberately delays the ACKs it sends back to `S2` by some amount, inflating `S2`'s RTT sample for that flow and, per the BDP-inflation mechanism Section 3.1.3 derives, its share of the shared bottleneck's throughput at `C1`'s expense, purely by choosing when its own ACKs leave the client.

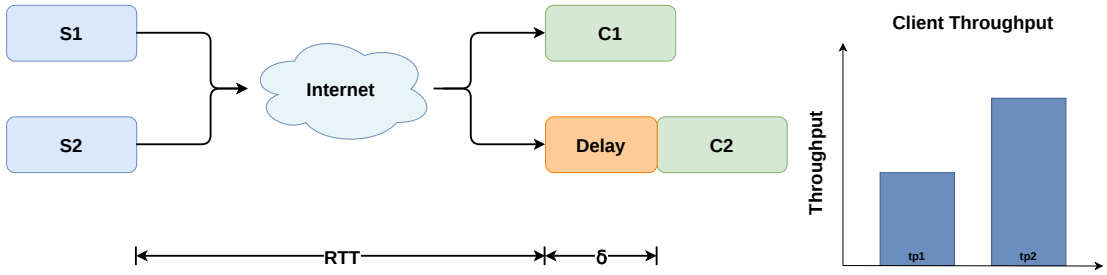


Figure 4.1: Artificially inflating the RTT to leverage BBR's RTT-unfairness bias: `C2` delays its own outgoing ACKs, inflating the RTT `S2` observes for that flow and, in turn, `C2`'s share of the bottleneck's throughput relative to the undelayed `C1`/`S1` flow.

Delaying every packet by a fixed amount indiscriminately, however, would delay the client's own outgoing application data by the same amount, adding latency to the client's own traffic for no

benefit. Figure 4.2 sketches the refinement this thesis's design is actually built around: an outgoing application data packet doesn't need to wait for the whole duration of the δ-RTT delay queue. Instead, as soon as an already-delayed but not-yet-released ACK is available, the payload packet can be sent out immediately, stamped with that ACK number. The pure ACK packet that would otherwise have carried the same number is now redundant and can be dropped. This is the core idea that Section 4.2.1's three strategies each implement with a different trade-off.

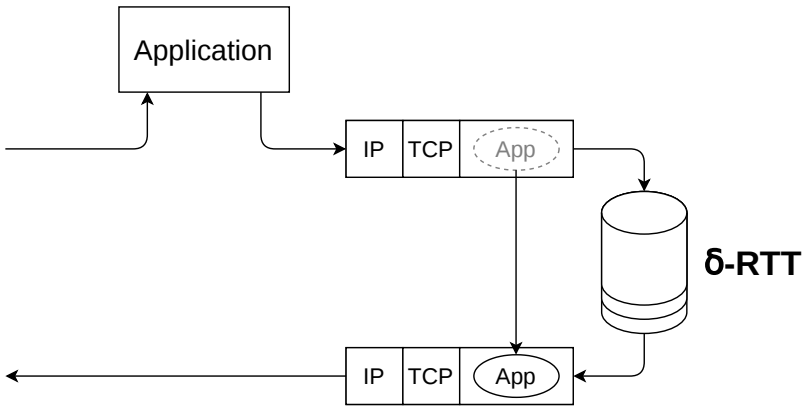


Figure 4.2: An outgoing application data packet is sent as fast as possible rather than having to wait behind the δ-RTT holding queue, carrying the ACK number of an already-delayed, not-yet-released ACK instead. This method avoids adding any latency to the client's own outgoing data while still inflating the sender's RTT sample for the ACKs which are still held back.

4.2 The ACK-Delaying Client

The client-side tool (`delay_acks`) is the tool that actually manufactures `client-2`'s additional RTT in Sections 5.3 to 5.5. It intercepts every outgoing packet of TCP flows via a `netfilter_queue` (`NFQUEUE`) hook installed with `iptables`, giving it full control over when each packet, ACK or payload, actually leaves the client. The RTT-unfairness study in Section 5.2 does not use the tool at all. Instead, there, every additional RTT is injected via `netem` as described in Section 5.1.2.

Once a packet is diverted into NFQUEUE, it passes through four stages before it leaves the host (Figure 4.3): the NFQUEUE callback `cb()` classifies it and enqueues it, without ever issuing a verdict itself. Every packet, pure ACK or payload, enters a per-flow packet queue. Every ACK number the flow carries is also separately recorded in a per-flow ACK queue, except under `--pure-delay` (Section 4.2.1), which never consults that bookkeeping and skips recording it entirely. A single worker thread polls all flows roughly every 0.5 ms, applies the active release strategy (Section 4.2.1) to decide what to release and with which ACK number, and finally a verdict is issued – `NF_ACCEPT` (possibly with a re-stamped ACK field) or `NF_DROP`. Whenever a packet's ACK field is re-stamped this way, its TCP checksum is recomputed over the modified header before the packet is released. A few cases bypass this pipeline entirely and are not shown in the figure: non-TCP traffic and `SYN` packets are forwarded immediately without queueing, and a `FIN`/`RST` flushes and forwards everything still queued for its flow right away, as whenever a flow is terminated, we don't care about our RTT inflation anymore and sending out the remaining ACKs early won't carry any negative effects.

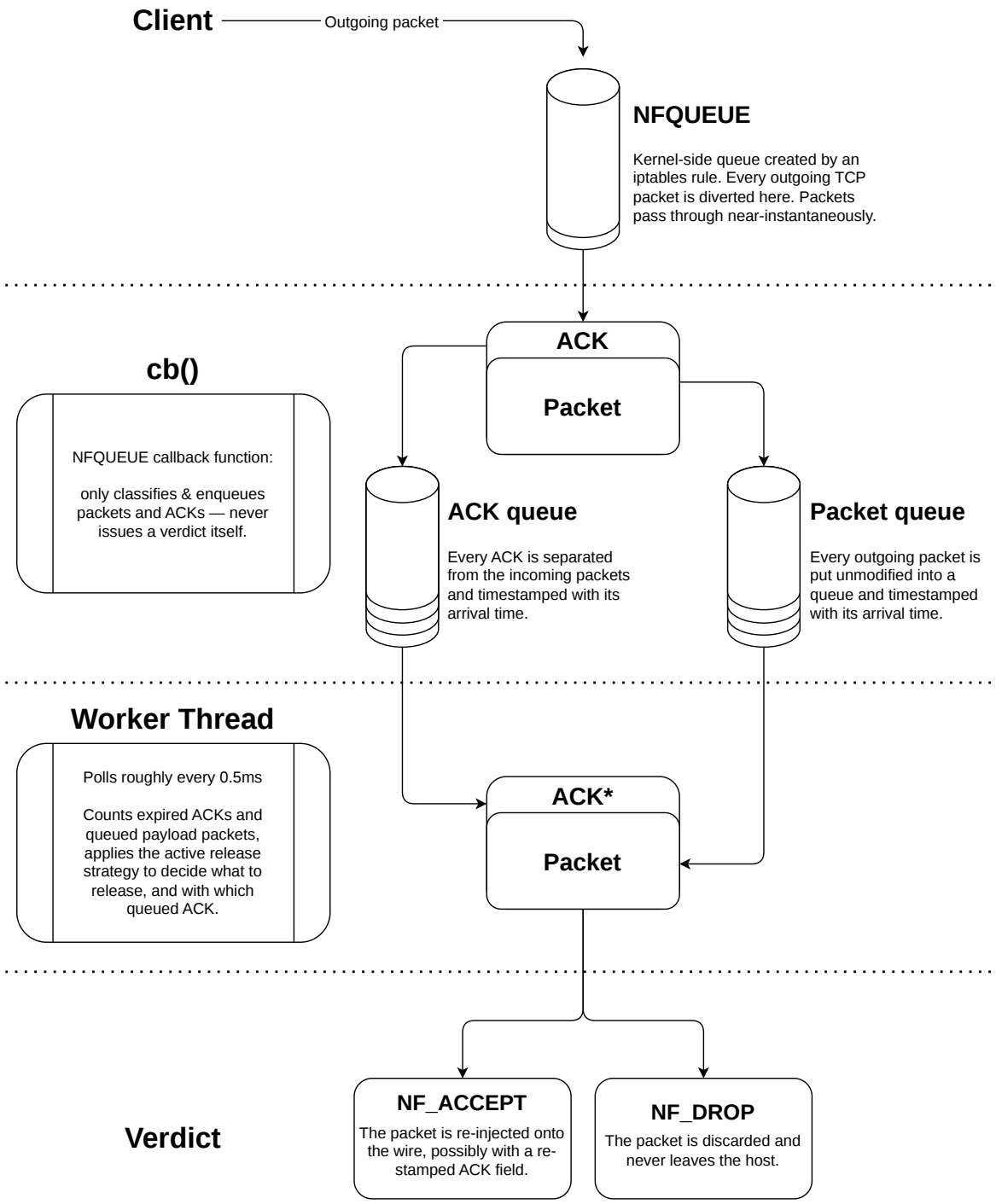


Figure 4.3: Architecture of `delay_acks`: an intercepted outgoing packet moves from the kernel's NFQUEUE, through classification and queueing in `cb()`, to a release-strategy decision in the worker thread, and finally to an `NF_ACCEPT`/`NF_DROP` verdict.

4.2.1 Release Strategies

The tool supports three release strategies, selected with a command-line flag:

Default (ACK-matching) Every arriving ACK is held for a fixed time `DELAY_MS` before it is eligible for release. While delayed ACKs are queued, the tool opportunistically stamps outgoing packets with an already-delayed (but not-yet-released) ACK number of an earlier packet as soon as one becomes available. Unless payload packets are waiting in the queue, in every tick of the worker thread, we try to keep half of the ACKs in our queue for the next tick so newly arrived payload packets can be sent out as fast as possible – releasing every expired ACK immediately would leave nothing in reserve for a payload packet that happens to arrive moments later.

`--pure-delay` Every packet, ACK or payload, is queued and released unmodified exactly `DELAY_MS` after it arrived. No ACK numbers are rewritten. This is the simplest possible strategy: delay everything.

`--eager-ack` A relaxation of the default mode: if a payload packet is waiting to be sent but no expired ACK is available yet, the tool will pair it with an ACK that is still in its hold-back window rather than waiting for the deadline, trading a slightly less inflated apparent RTT for less added latency on the data path.

Each strategy above trades the same two quantities against each other in different proportions: how much of the sender's `RTprop` estimate ends up inflated, versus how much extra latency is added to the client's own outgoing data. Per Section 3.1.3, it is precisely that inflated RTT sample which feeds directly and linearly into the sender's BDP estimate and, in turn, in-flight budget – the entire mechanism this thesis tries to exploit. The default strategy and `--pure-delay` both hold every ACK for the full `DELAY_MS` duration before it becomes eligible for release, so neither one ever gives back any of the inflation it manufactures by revealing to the sender that our actual RTT is much shorter than we pretend it to be. `--eager-ack` is the one exception: by pairing payload with an ACK that has not yet expired, it lets the sender observe that specific packet's RTT closer to the path's true propagation delay, undercutting its own inflation in exchange for lower added latency on the data path – a trade-off Section 5.3 returns to when explaining why this strategy underperforms the other two.

4.2.2 Logging

Independently of which strategy is active, `delay_acks` logs every packet and ACK it processes to two CSV files, written asynchronously by a dedicated logger thread so logging never blocks the worker thread's release decisions. `packets.csv` holds one row per intercepted packet (TCP sequence number, its original and final ACK field, payload length, arrival/release timestamps, and whether it was released or dropped), and `acks.csv` holds one row per distinct ACK number the tool considered releasing (whether it was ultimately released or superseded/dropped, and, if released, which packet it was stamped onto). This is the data Sections 5.3.1 and 5.4.1 analyse directly to measure how long each strategy actually held packets and ACKs in practice.

Chapter 5

Evaluation

This chapter empirically evaluates whether, and under what conditions, deliberately inflating a BBR sender's RTT sample translates into a real advantage. Section 5.1 describes the experimental infrastructure used throughout this chapter: the emulated network topologies, the setup used to control bandwidth, delay, and buffering at the bottleneck, the traffic capture and analysis pipeline, and the workload generators. Section 5.2 studies BBR's RTT-unfairness bias itself in a controlled, synthetic setting, isolating how it depends on base RTT, buffer size, bottleneck bandwidth, and window scaling, before the ACK-delaying client explained in Chapter 4 is introduced. Section 5.3 tests said ACK-delaying client for a first time and measures its effect on file download completion time against a single contender over the real Internet. Section 5.4 repeats this with a more latency-sensitive workload, YouTube video playback, where moment-to-moment delivery rate rather than pure completion time determines the user-perceived Quality of Experience. Finally, Section 5.5 removes the competing flow and bottleneck entirely to isolate whether the exploit's gains come at a contender's expense or from BBR simply granting more bandwidth outright.

5.1 Experimental Setup

5.1.1 Emulated Network Topology

All our experiments run on a single Linux VM (6 threads of an AMD EPYC 8-Core Processor with 32 GiB of RAM) using network namespaces (`netns`) connected by veth pairs, rather than physical machines or a public cloud testbed. The topology we set up depends on the test performed; the two we use are explained below:

- **Emulated server topology** (used in Section 5.2): `client-1` and `client-2` each connect to `router-1`, which connects to `router-2`, which connects to `server`. `router-1`–`router-2` is the shared bottleneck link; every other link only carries propagation delay. Everything, including the server, lives in a namespace on the same machine, so no traffic ever touches a physical NIC or the Internet. Every RTT in this topology is injected via the `netem` qdisc described in Section 5.1.2. The ACK-delaying client introduced in Section 4.2 is not used anywhere in Section 5.2.
- **Real-Internet topology** (used in Sections 5.3 to 5.5): `client-1` and `client-2` again connect through `router-1`–`router-2`, but `router-2` is NATed onto the host's real network interface, so traffic leaving the clients reaches the actual Internet. This lets us keep full control over the clients' RTT and the shared bottleneck while downloading real files or streaming real YouTube videos from Google's infrastructure.

In both topologies, the server is the data sender and `client-1`/`client-2` are the receivers whose delayed ACKs the sender's BBR reacts to (Chapter 4). In the Emulated server topology, the server is our own namespace on this VM's shared kernel (Linux 6.8.0), which provides only the standard mainline `tcp_bbr` module – i.e., BBRv1, not BBRv2/v3 – selected per-connection via `iperf3`'s `-C` flag (Section 5.2). In the Real-Internet topology, the sender is Google's infrastructure, which we do not control or have kernel-level visibility into; Section 3.2 covers what is known about which BBR version Google runs there.

5.1.2 Emulating Delay, Bandwidth, and Buffers

Bottleneck bandwidth, buffering, and propagation delay are all emulated with Linux traffic control (`tc`). On the bottleneck-facing interface of each router we install a three-level qdisc hierarchy:

```
tc qdisc add dev $IFACE root handle 1: htb default 10
tc class add dev $IFACE parent 1: classid 1:10 htb rate $BW
tc qdisc add dev $IFACE parent 1:10 handle 10: netem delay $LINK_DELAY
tc qdisc add dev $IFACE parent 10: handle 20: pfifo limit $BUFF
```

`htb` caps the link at a target bandwidth `$BW`, `netem` adds a fixed one-way propagation delay `$LINK_DELAY`, and a `pfifo` with a limit of `$BUFF` packets acts as the bottleneck queue. This is the buffer whose size we vary relative to the path's bandwidth-delay product (BDP) in Section 5.2.2. All other links (client–router, server–router) only carry a plain `netem delay` qdisc with no rate limit.

Given a target base RTT for `client-1` and an additional RTT that `client-2` should experience on top of it, rather than concentrating the delays on a single hop, the setup scripts distribute the necessary one-way delays across the individual links (bottleneck link, both access links, and the server link or the host link, respectively) so that the measured round-trip time of each client's TCP connection matches the intended value while still trying to simulate real-life Internet setups. In the Emulated server topology this base RTT is the connection's entire round-trip time, since the server is itself one of our own namespaces. In the Real-Internet topology it only covers the path up to our own host's uplink (Section 5.1.1); beyond that point, traffic travels the real Internet to Google's servers, adding further latency that we do not control or measure.

For every experimental setup we calculate the bandwidth-delay product

$$\mathrm{BDP}_{\mathrm{packets}} = \frac{\mathrm{BW} \times \mathrm{RTT}}{8 \times 1500 \text{ bytes}}$$

and log it alongside every run so that buffer sizes can later be expressed as a multiple of the BDP instead of an absolute packet count.

5.1.3 Traffic Capture and Analysis Pipeline

Every run captures a `tcpdump` at every interface of interest (typically at both access links and at the other side of the bottleneck). Captures are parsed in Python with `dpkt` into `polars` data frames, packets are grouped into bidirectional flows, and retransmitted packets are matched and excluded from RTT and delivery-rate calculations (a retransmission makes it ambiguous which departure a given ACK belongs to, so we conservatively drop the pair rather than risk an incorrect measurement). From the surviving packets we reconstruct, per flow and over time: goodput and throughput, packets/bytes in flight, and (where relevant) round-trip time. Throughput *share* between the two clients, our primary metric in Section 5.2, is simply the ratio of `client-2`'s average bits-per-second to `client-1`'s over the run.

5.1.4 Workload Generators

We use three different workload generators depending on the experiment:

- **Synthetic throughput** (Section 5.2): `iperf3` servers and clients, with the congestion control algorithm, BBR or CUBIC, selected per flow.
- **File downloads** (Sections 5.3 and 5.5): a headless Selenium-driven Chrome instance inside each client namespace requests a real file directly by URL (e.g. a file on Google's download servers) and polls for the disappearance of Chrome's `.crdownload` marker to time the download to completion.
- **Video playback** (Section 5.4): a headless Selenium-driven Chrome instance opens a YouTube video, rejects cookies, skips ads, forces a specific playback resolution through the settings menu (re-selecting it if it was already active, to force a fresh buffering start), waits for a synchronisation signal shared with the other client, and starts playback for a fixed duration.

Both browser-driven workloads (**File downloads** and **Video playback**) used in Sections 5.3 to 5.5 launch Chrome with `--disable-quic`, forcing every connection onto TCP rather than QUIC/HTTP3. This is a prerequisite rather than an incidental flag: `delay_acks` intercepts TCP packets only (Section 4.2), so without it a modern Chrome would silently negotiate QUIC with Google's servers by default and the tool would have no effect on traffic at all.

5.2 Studying RTT Unfairness

Before trying to exploit BBR's bias towards long-RTT flows on the real Internet, we first wanted to see how large and how robust the bias towards long-RTT BBR flows actually is in practice, and what it depends on. In every experiment in this section, `client-1` runs either BBR or CUBIC at the testbed's base RTT, and `client-2` runs BBR at the base RTT plus some additional RTT, competing for the same bottleneck. We report the throughput share as ratio $= {}^{\text{throughput}_{c2}}/_{\text{throughput}_{c1}}$. For example, given one contender, a ratio above 1 means the longer-RTT flow (`client-2`) is getting more than its fair share of the bottleneck. We performed each experiment of the various configurations shown in this section five times and took their mean and standard deviation values in order to create the plots. All the experiments were performed with 1, 2, and 5 contenders on the `client-1` namespace. As showing every single plot of all measurements we took would overwhelm this thesis, we decided to only include a subset of the datapoints generated in this report.

5.2.1 Effect of Base RTT

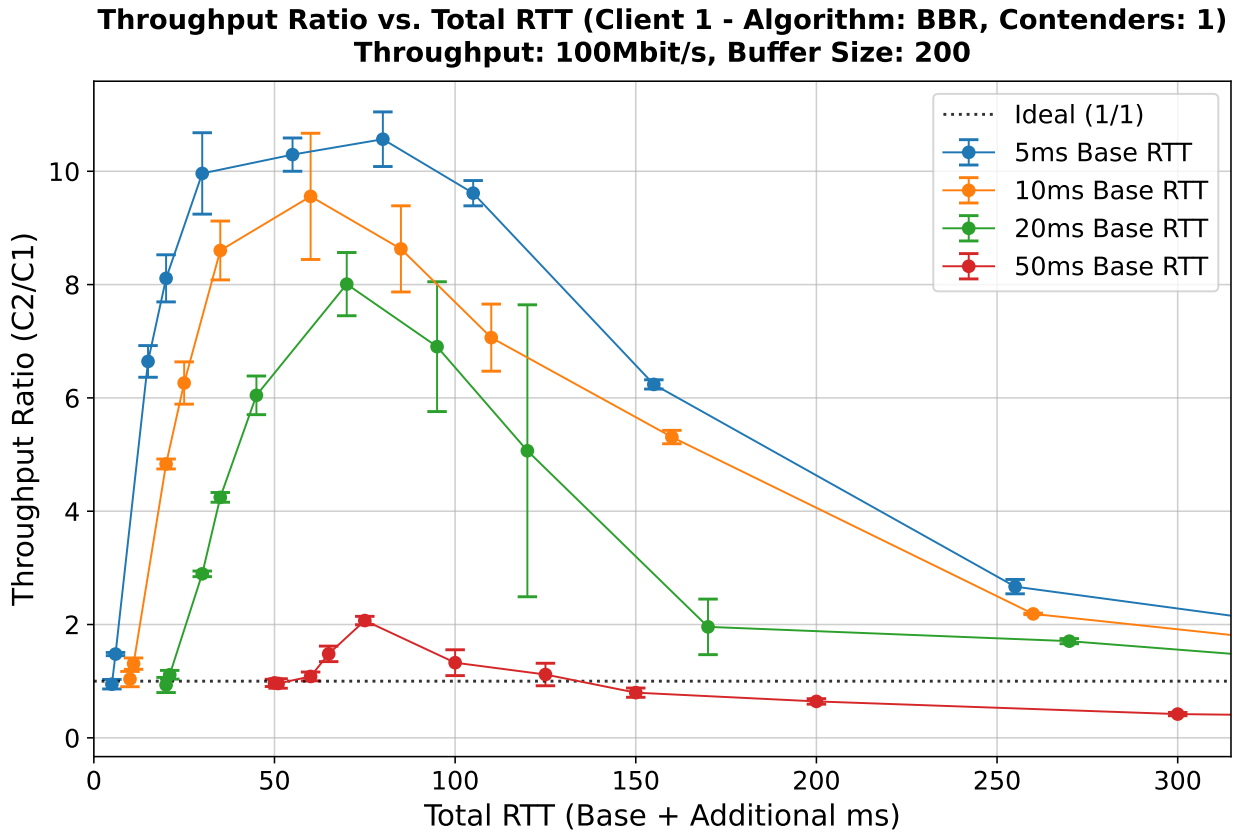


Figure 5.1: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` BBR flow, as a function of total RTT (100 Mbit/s, 200-packet buffer). See Table 5.1 for the corresponding peak-ratio summary.

Table 5.1: Peak throughput-share ratio summary corresponding to Figure 5.1.

Base RTT (ms)	Peak Add. Delay (ms)	Peak Total RTT (ms)	Peak Ratio
5	75	80	10.57
10	50	60	9.56
20	50	70	8.01
50	25	75	2.07

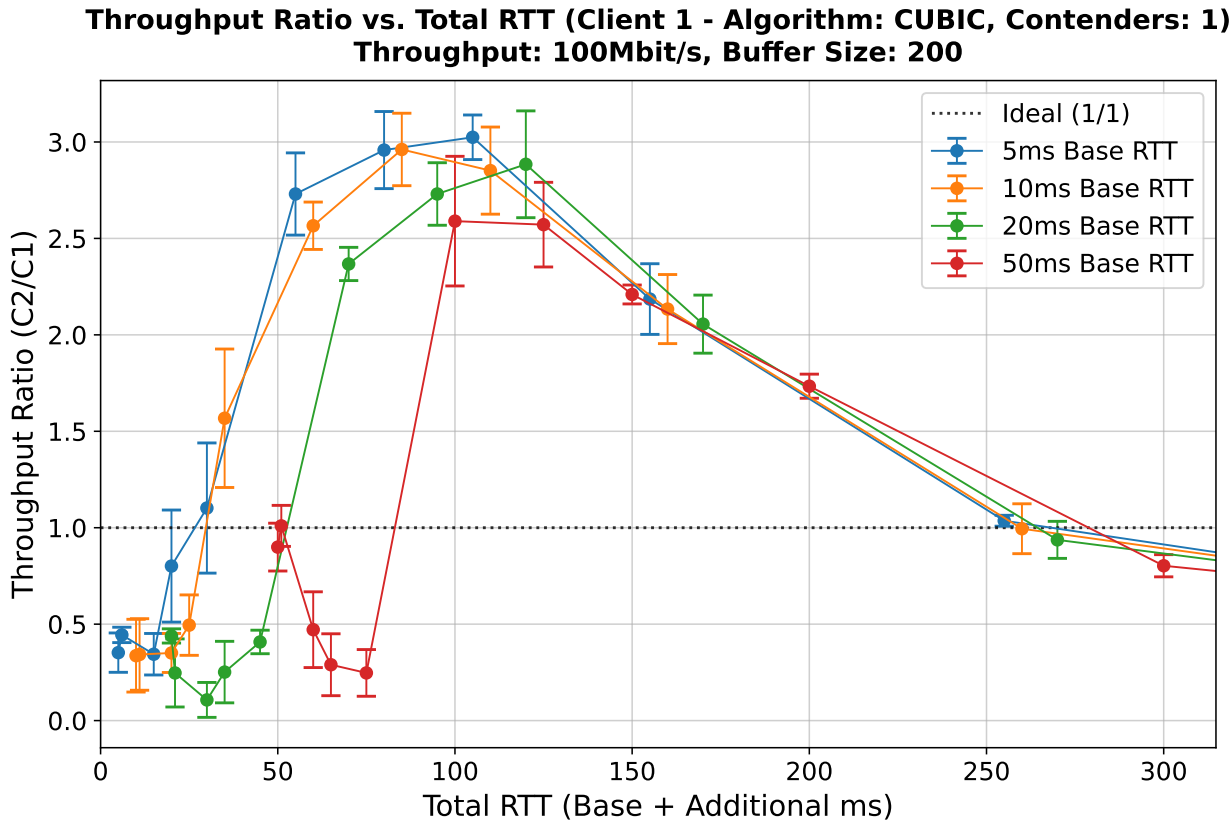


Figure 5.2: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` CUBIC flow, as a function of total RTT (100 Mbit/s, 200-packet buffer). See Table 5.2 for the corresponding peak-ratio summary.

Table 5.2: Peak throughput-share ratio summary corresponding to Figure 5.2.

Base RTT (ms)	Peak Add. Delay (ms)	Peak Total RTT (ms)	Peak Ratio
5	100	105	3.02
10	75	85	2.96
20	100	120	2.88
50	50	100	2.59

Figures 5.1 and 5.2 (numeric summary in Tables 5.1 and 5.2) plot the throughput-share ratio against `client-2`'s *total* RTT (base RTT plus additional RTT) for a 100 Mbit/s bottleneck with a 200-packet buffer with one contender competing on `client-1`. In Figure 5.1 both clients used BBR and in Figure 5.2 the contender on `client-1` used CUBIC. Both plots show the same qualitative shape: the ratio rises sharply as additional RTT for `client-2` is introduced, peaks well above 1 – i.e. the longer-RTT BBR flow takes several times more than its fair share – and then falls off again as the additional RTT keeps growing.

Two things stand out. First, BBR-vs-BBR unfairness is roughly three to five times stronger than BBR-vs-CUBIC unfairness at the same base RTT. Second, for BBR-vs-BBR unfairness, the effect is strongest at small base RTTs and clearly erodes as the base RTT grows. At 5 ms base RTT the BBR peak ratio is over 10×, but at 50 ms it has dropped to about 2×. This tells us that the

"attack surface" for RTT-based unfairness in scenarios where our BBR client contends with other BBR clients is largest on already-low-latency paths. When competing with CUBIC on the other hand, our share peaks at approximately 3× the fair share for all our tested base RTTs while only marginally eroding as base RTT grows (3.02× at 5 ms ⟶ 2.59× at 50 ms).

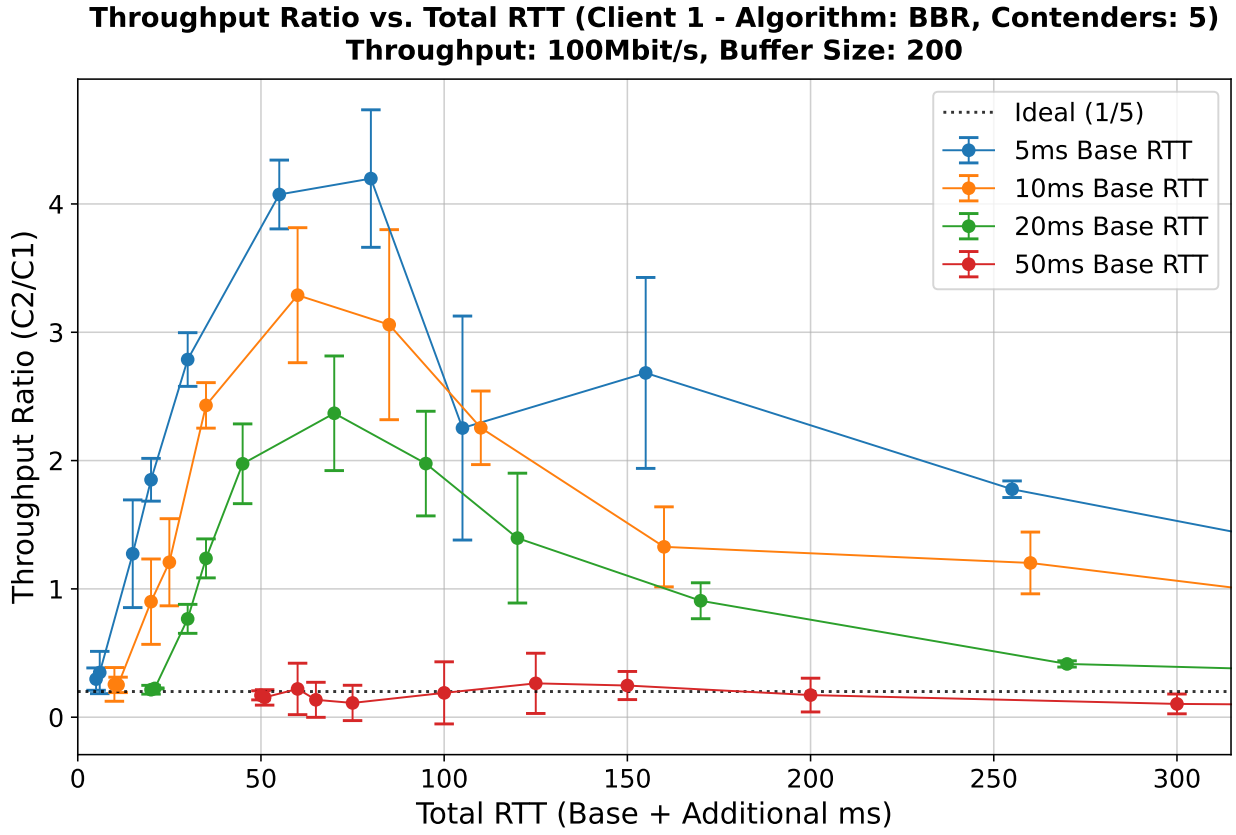


Figure 5.3: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` BBR flows, as a function of total RTT (100 Mbit/s, 200-packet buffer). See Table 5.3 for the corresponding peak-ratio summary.

Table 5.3: Peak throughput-share ratio summary corresponding to Figure 5.3.

Base RTT (ms)	Peak Add. Delay (ms)	Peak Total RTT (ms)	Peak Ratio
5	75	80	4.20
10	50	60	3.29
20	50	70	2.37
50	75	125	0.26

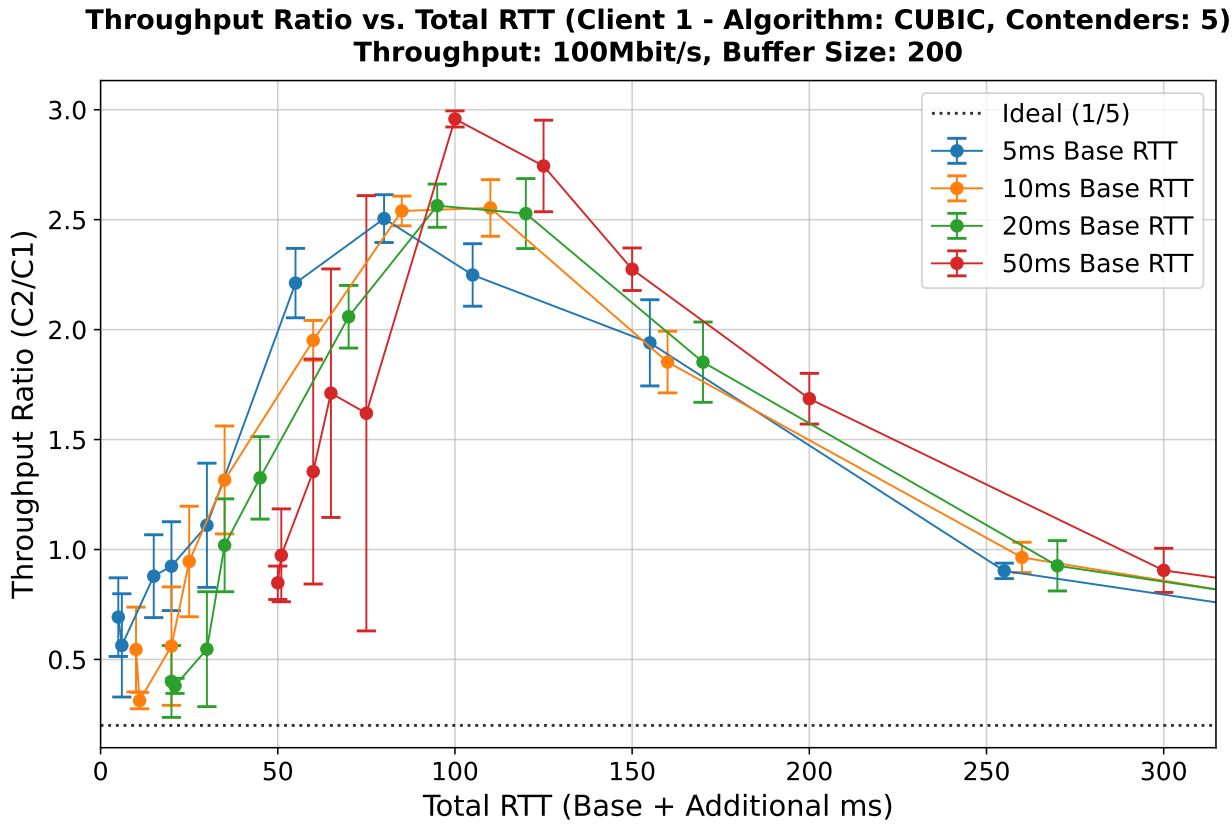


Figure 5.4: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` CUBIC flows, as a function of total RTT (100 Mbit/s, 200-packet buffer). See Table 5.4 for the corresponding peak-ratio summary.

Table 5.4: Peak throughput-share ratio summary corresponding to Figure 5.4.

Base RTT (ms)	Peak Add. Delay (ms)	Peak Total RTT (ms)	Peak Ratio
5	75	80	2.50
10	100	110	2.55
20	75	95	2.56
50	50	100	2.96

Figures 5.3 and 5.4 (numeric summary in Tables 5.3 and 5.4) repeat the same experiment with *five* contenders on `client-1` instead of one. This means that `client-2`'s "fair" share drops to $1/5 = 0.2$ instead of 1, so the raw peak ratios in the tables above are naturally much smaller than in the 1-contender case. In order to compare these values with the ones from the 1-contender case, we have to divide the values by 0.2. Doing so gives us a clearer picture of how much unfairness `client-2` actually achieves: at 5 ms base RTT, a peak ratio of 4.20 against 5 BBR contenders means `client-2` still claims about 21× its fair share, compared to 10.57× against a single BBR contender – i.e. the relative unfairness is roughly *twice as large* with more contenders present, not smaller. This advantage decays with base RTT just as sharply as in the 1-contender case, though: by 50 ms base RTT the same calculation gives only $0.26/0.2 \approx 1.3\times$, which is essentially fair. Against CUBIC contenders the picture is different again: the relative advantage sits consistently around 12–15×

fair share across every base RTT we tested (e.g. $^{2.50}/_{0.2} = 12.5\times$ at 5 ms up to $^{2.96}/_{0.2} = 14.8\times$ at 50 ms) and, similarly to the 1-contender result, doesn't erode significantly as base RTT grows – on the contrary, it actually increases slightly.

5.2.2 Effect of Buffer Size

For the remaining parameters in this section, our clients' base RTT is fixed while we sweep the parameter of interest, so we plot throughput-share ratio against *additional* RTT directly instead of total RTT (unlike Section 5.2.1, where total RTT was the more meaningful axis since base RTT itself varied between curves).

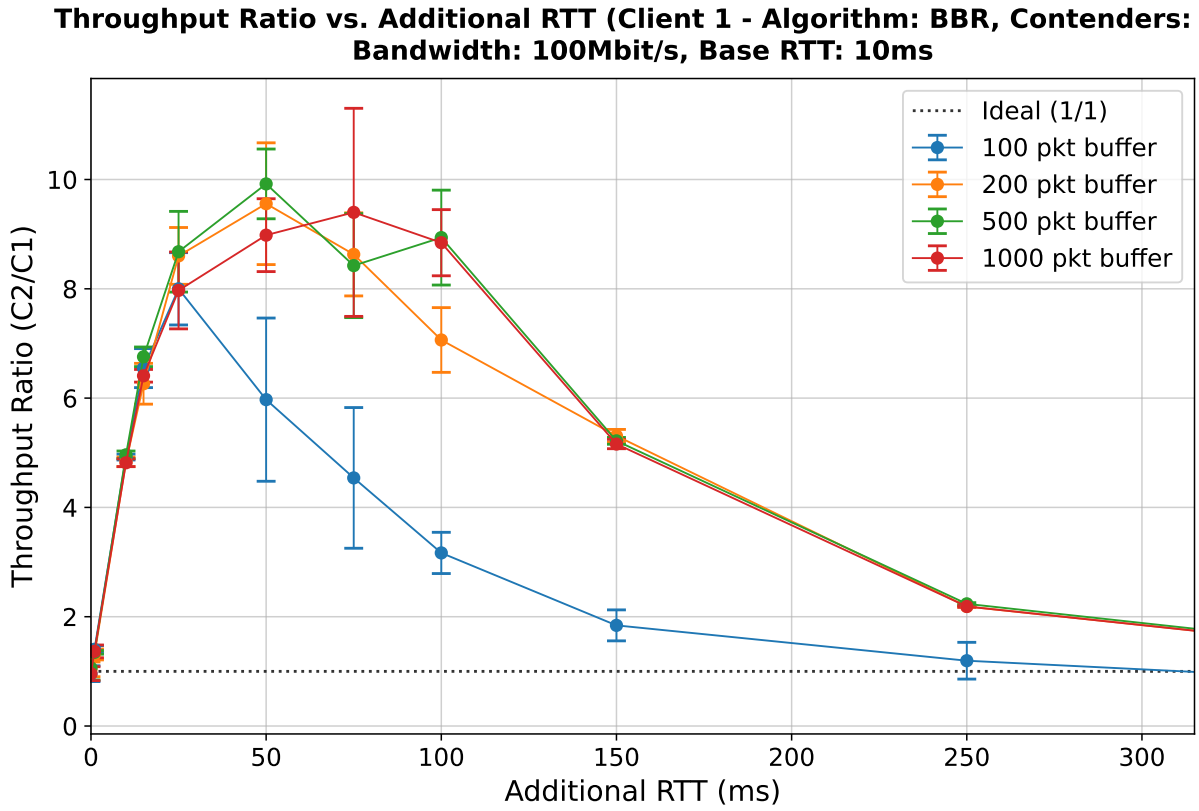


Figure 5.5: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` BBR flow, as a function of additional RTT (100 Mbit/s, 10 ms base RTT). See Table 5.5 for the corresponding peak-ratio summary.

Table 5.5: Peak throughput-share ratio summary corresponding to Figure 5.5.

Buffer Size (pkts)	Peak Add. Delay (ms)	Peak Ratio
100	25	8.00
200	50	9.56
500	50	9.92
1000	75	9.40

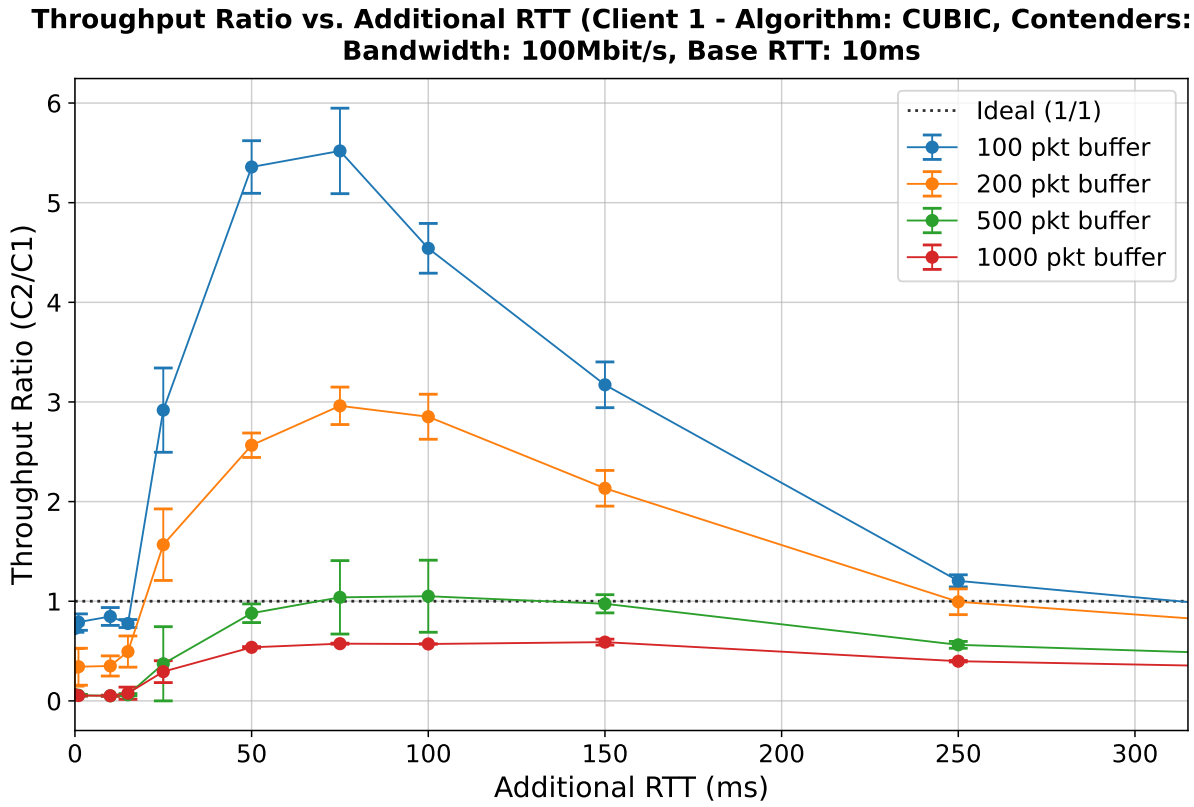


Figure 5.6: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` CUBIC flow, as a function of additional RTT (100 Mbit/s, 10 ms base RTT). See Table 5.6 for the corresponding peak-ratio summary.

Table 5.6: Peak throughput-share ratio summary corresponding to Figure 5.6.

Buffer Size (pkts)	Peak Add. Delay (ms)	Peak Ratio
100	75	5.52
200	75	2.96
500	100	1.05
1000	150	0.59

Figures 5.5 and 5.6 (numeric summary in Tables 5.5 and 5.6) sweep the bottleneck buffer from 100 to 1000 packets at a fixed 100 Mbit/s bottleneck throughput and 10 ms base RTT from the server towards both clients and one contender on `client-1` competing against our BBR flow on `client-2`. For BBR-vs-BBR the peak ratio is fairly insensitive to buffer size, staying between roughly 8× and 10× throughput ratio in favour of `client-2`, but the additional delay at which the peak occurs shifts steadily to the right as buffer grows. This shows that a deeper queue can sustain a larger inflated-RTT estimate for longer before it stops paying off. For CUBIC-vs-BBR the picture is very different: growing the buffer from 100 to 1000 packets collapses the peak ratio from 5.52× down to 0.59× the fair share – i.e. at the largest buffer we tested, `client-2` ends up with *less* than a fair share. This is not the trick backfiring, though: at 1000 packets `client-2` already trails badly even at 0 ms additional delay (roughly 0.06× fair share), so the peak still represents

an almost 10× improvement over doing nothing; it just isn't enough to close the gap all the way to parity. This finding is in line with the findings of Hock et al. [8] discussed in Section 2.4: in shallow buffers, BBR claims a disproportionate share of the bottleneck bandwidth from competing CUBIC flows. A deeper buffer on the other hand lets CUBIC's loss-based growth keep pushing its own window up eventually outpacing BBR's model-based estimate once the buffer is large enough to sustain it.

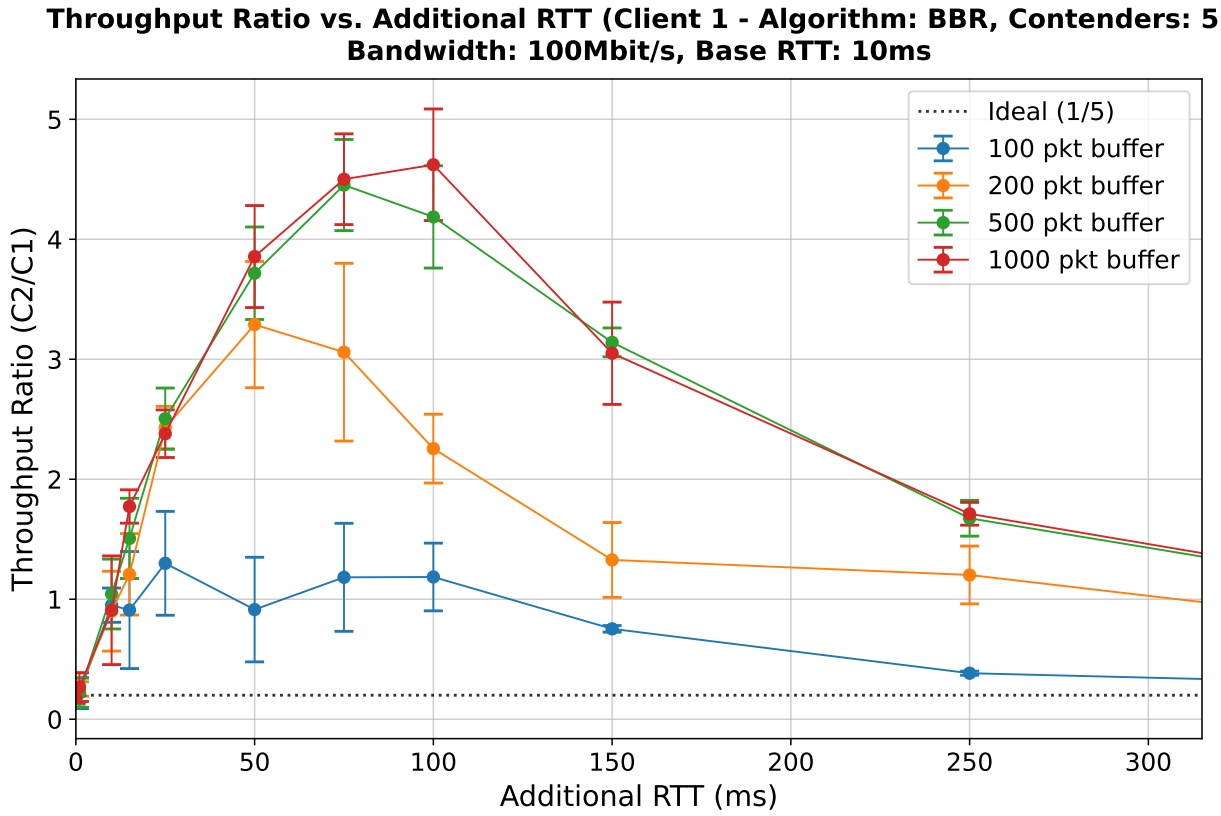


Figure 5.7: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` BBR flows, as a function of additional RTT (100 Mbit/s, 10 ms base RTT). See Table 5.7 for the corresponding peak-ratio summary.

Table 5.7: Peak throughput-share ratio summary corresponding to Figure 5.7.

Buffer Size (pkts)	Peak Add. Delay (ms)	Peak Ratio
100	25	1.30
200	50	3.29
500	75	4.45
1000	100	4.62

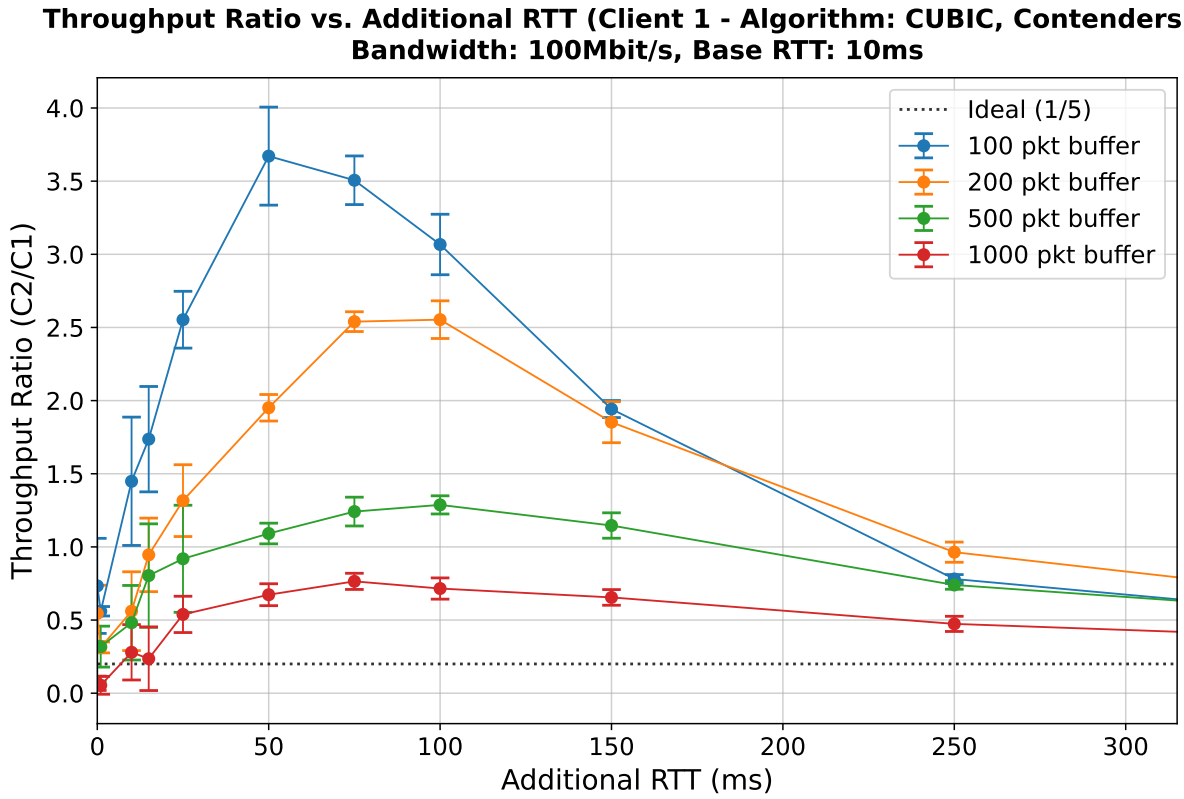


Figure 5.8: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` CUBIC flows, as a function of additional RTT (100 Mbit/s, 10 ms base RTT). See Table 5.8 for the corresponding peak-ratio summary.

Table 5.8: Peak throughput-share ratio summary corresponding to Figure 5.8.

Buffer Size (pkts)	Peak Add. Delay (ms)	Peak Ratio
100	50	3.67
200	100	2.55
500	100	1.29
1000	75	0.76

Figures 5.7 and 5.8 (numeric summary in Tables 5.7 and 5.8) repeat the same sweep against five contenders on `client-1` resulting in a fair share of $1/5 = 0.2$. Therefore we again divide the peak ratios by 0.2 for our discussion to have a quantitative value of unfairness. Against BBR contenders, the relative advantage now grows monotonically with buffer size, from $1.30/0.2 = 6.5\times$ at 100 packets up to $4.62/0.2 = 23.1\times$ at 1000 packets, whereas with a single BBR contender the peak ratio had already flattened out by 200 packets and stayed roughly flat from there. The peaks also shift monotonically to the right from 25 ms additional delay at a buffer size of 100 packets to reaching the peak at 100 ms of additional delay for a buffer size of 1000 packets. Against CUBIC contenders, buffer size still hurts `client-2`'s relative advantage as it did with one contender, but the effect is far less severe: at 1000 packets, a single CUBIC contender already pushed `client-2` below a fair share (0.59× the fair share), while against five CUBIC contenders at the same buffer size `client-2` still claims about $0.76/0.2 = 3.8\times$ its fair share.

5.2.3 Effect of Bandwidth

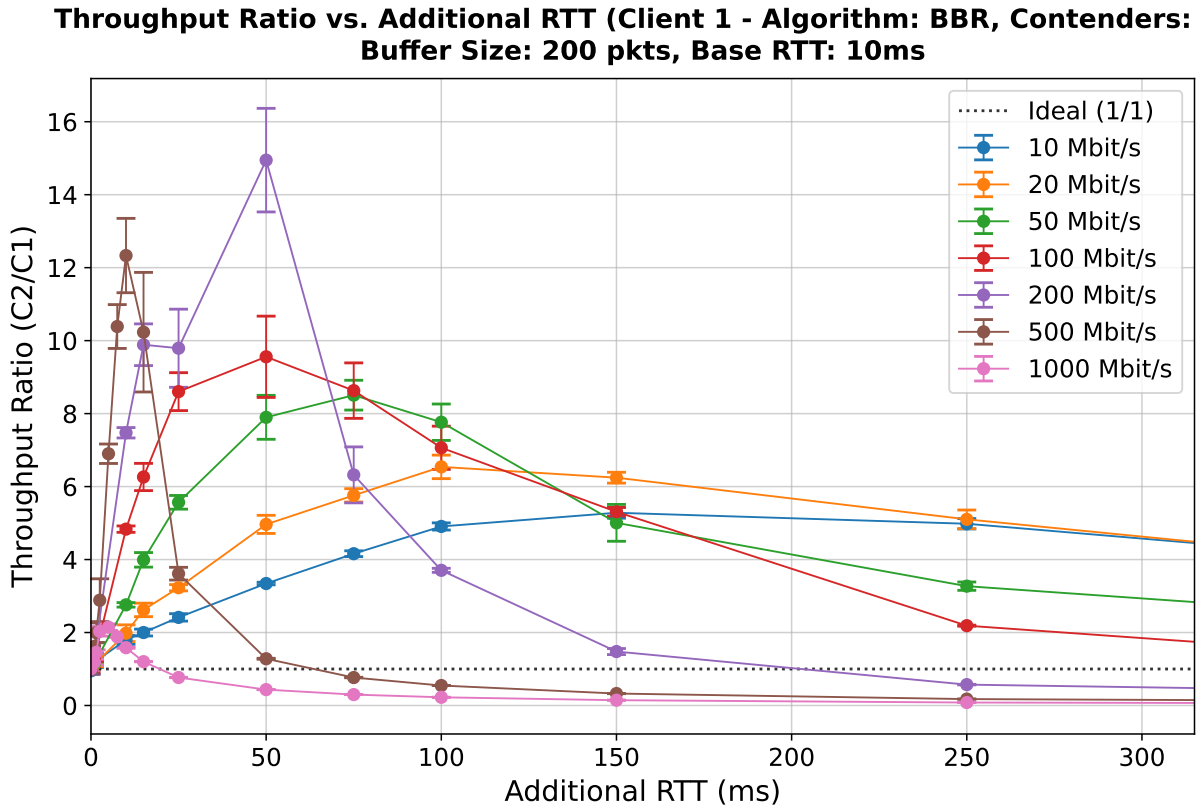


Figure 5.9: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` BBR flow, as a function of additional RTT (200-packet buffer, 10 ms base RTT). See Table 5.9 for the corresponding peak-ratio summary.

Table 5.9: Peak throughput-share ratio summary corresponding to Figure 5.9.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	150	5.28
20	100	6.54
50	75	8.51
100	50	9.56
200	50	14.95
500	10	12.33
1000	5	2.15

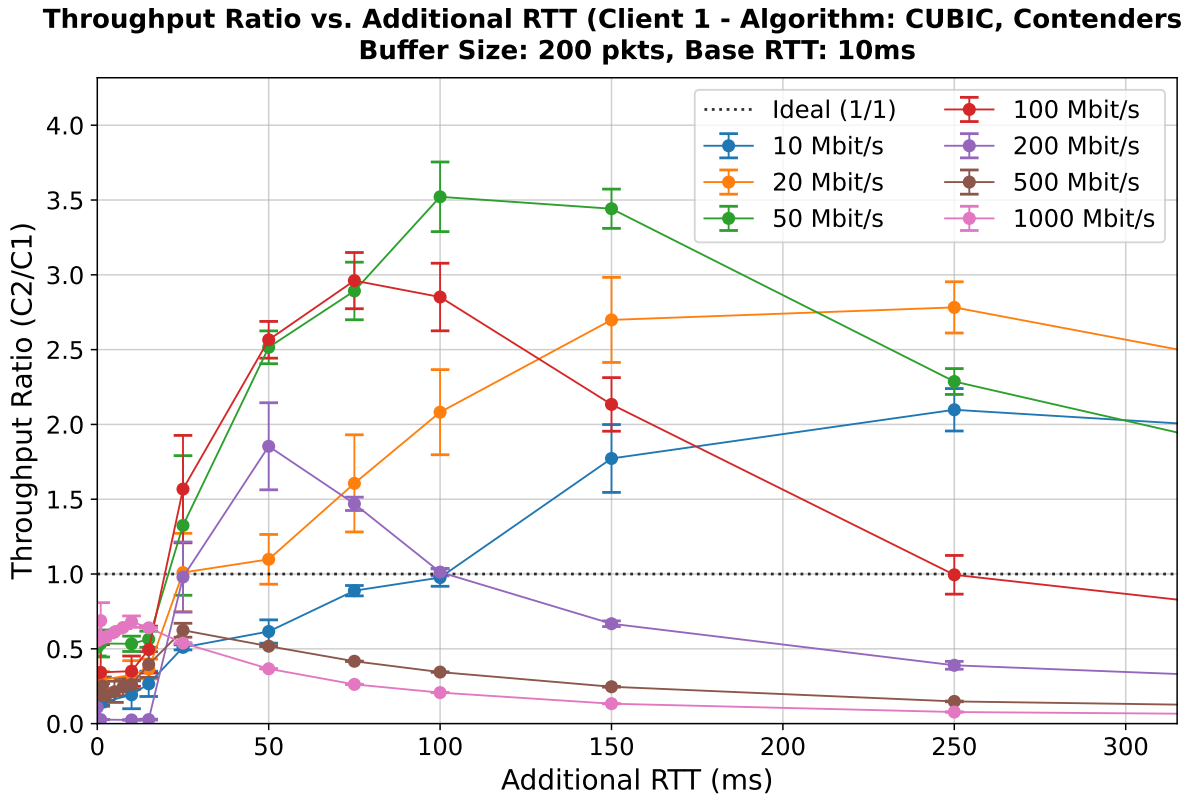


Figure 5.10: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` CUBIC flow, as a function of additional RTT (200-packet buffer, 10 ms base RTT). See Table 5.10 for the corresponding peak-ratio summary.

Table 5.10: Peak throughput-share ratio summary corresponding to Figure 5.10.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	250	2.10
20	250	2.78
50	100	3.52
100	75	2.96
200	50	1.85
500	25	0.62
1000	1	0.69

Figures 5.9 and 5.10 (numeric summary in Tables 5.9 and 5.10) show our experiments where we varied the bottleneck bandwidth from 10 to 1000 Mbit/s at a fixed 200-packet buffer and 10 ms base RTT with one contender on `client-1`. Unlike the base RTT and buffer sweeps, the peak ratio here is not monotonic in either the BBR-vs-BBR or the CUBIC-vs-BBR regime. For BBR-vs-BBR it climbs from 5.28× at 10 Mbit/s up to a maximum of 14.95× around 200 Mbit/s, then collapses sharply to 2.15× at 1000 Mbit/s. This is the same buffer-relative-to-BDP mechanism we already saw in Section 5.2.2, just approached from the bandwidth axis instead. The fixed 200-packet buffer we hold constant here is generous relative to the BDP at low bandwidths, but at 500–1000 Mbit/s the same 200 packets are a comparatively tiny queue, and once the buffer can no longer sustain

the inflated in-flight window BBR's model calls for, the effect breaks down. The additional delay needed to reach the peak shrinks correspondingly, from 150 ms at 10 Mbit/s down to just 5 ms at 1000 Mbit/s, since a fixed packet-count buffer represents an ever-smaller amount of time-in-flight as bandwidth grows. For CUBIC-vs-BBR a similar effect appears, just starting from a much lower ceiling. The peak ratio rises only mildly (2.10× to 3.52×) before falling below a fair share entirely at 500 and 1000 Mbit/s (0.62× and 0.69×). As with the deep-buffer case above, this is not the trick backfiring: at 0 ms additional delay `client-2` already sits at roughly 0.21× and 0.55× fair share at 500 and 1000 Mbit/s respectively, so the peak still nearly triples the 500 Mbit/s baseline and clearly improves on the 1000 Mbit/s one, just not enough to reach parity. This means that at high bandwidth relative to a fixed buffer, `client-2` can no longer out-compete even a single CUBIC contender no matter how much its RTT is increased.

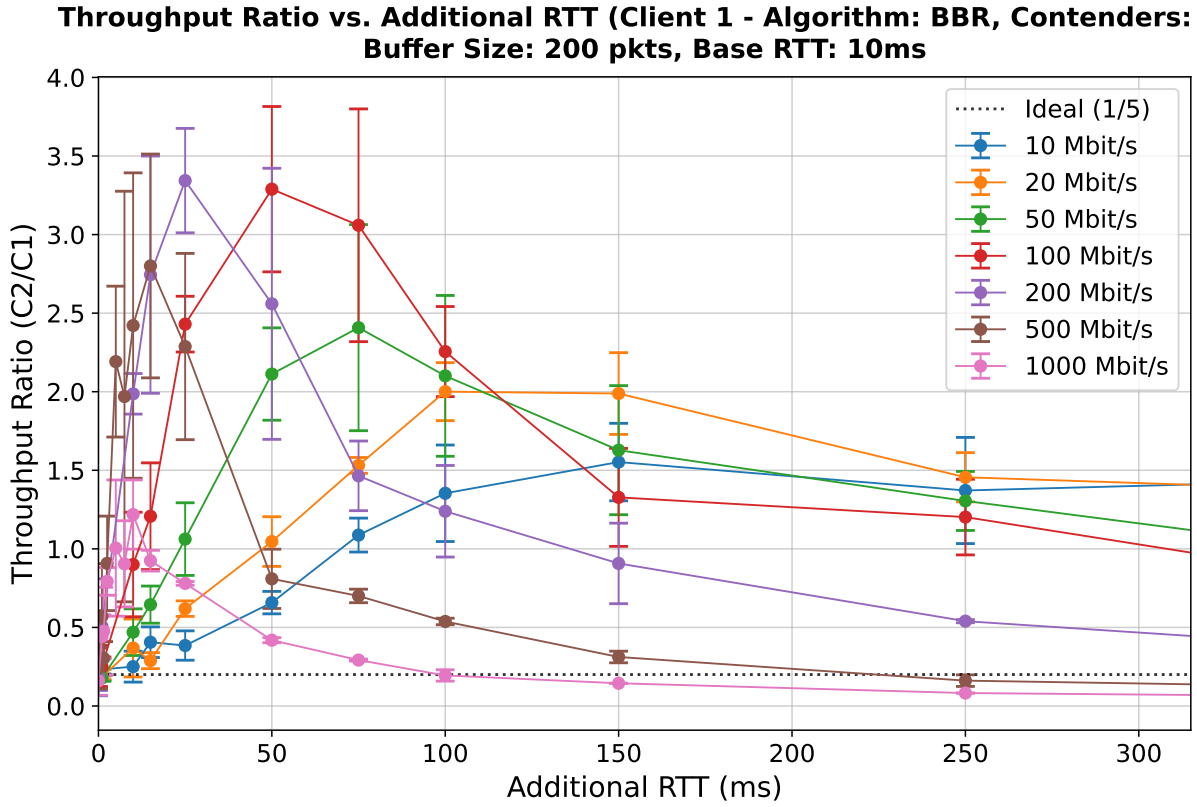


Figure 5.11: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` BBR flows, as a function of additional RTT (200-packet buffer, 10 ms base RTT). See Table 5.11 for the corresponding peak-ratio summary.

Table 5.11: Peak throughput-share ratio summary corresponding to Figure 5.11.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	150	1.55
20	100	2.00
50	75	2.41
100	50	3.29
200	25	3.34
500	15	2.80
1000	10	1.22

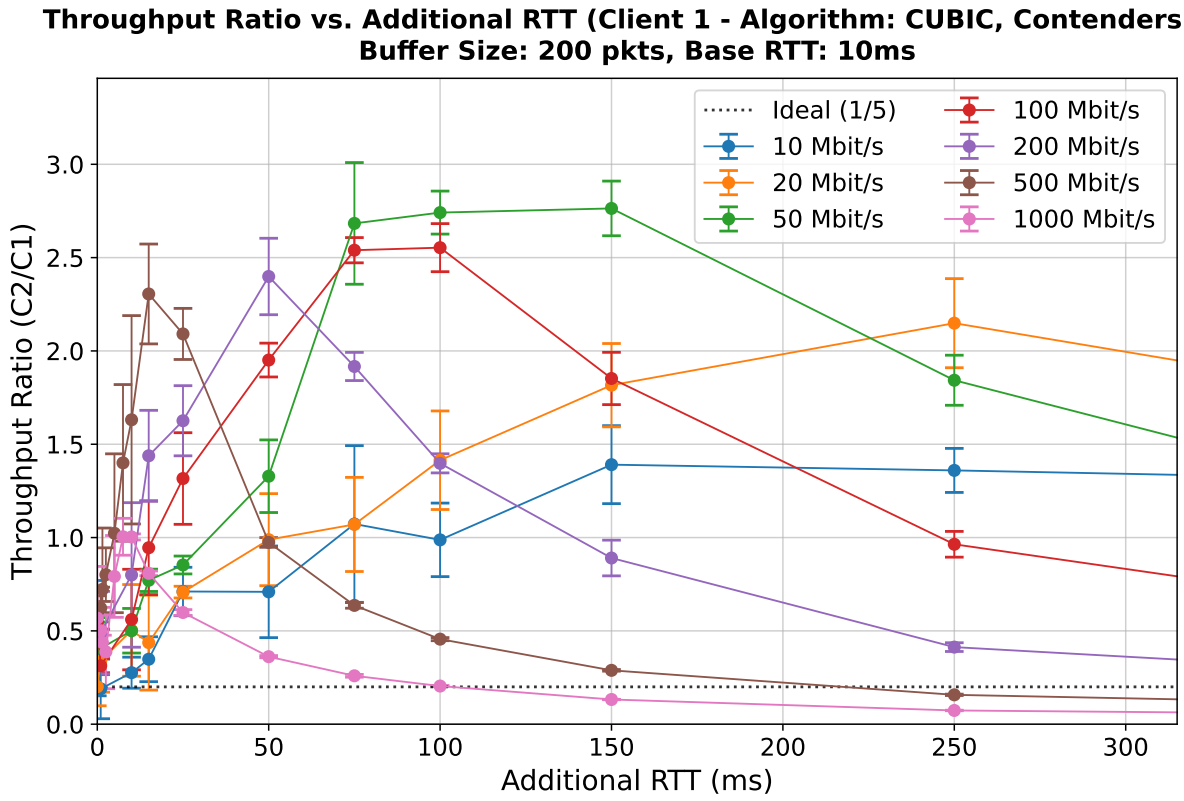


Figure 5.12: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` CUBIC flows, as a function of additional RTT (200-packet buffer, 10 ms base RTT). See Table 5.12 for the corresponding peak-ratio summary.

Table 5.12: Peak throughput-share ratio summary corresponding to Figure 5.12.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	150	1.39
20	250	2.15
50	150	2.76
100	100	2.55
200	50	2.40
500	15	2.31
1000	7.5	1.00

Figures 5.11 and 5.12 (numeric summary in Tables 5.11 and 5.12) repeat the sweep against five contenders, so we again divide the ratios by the 0.2 fair share for comparison. BBR-vs-BBR again peaks in the same 100–200 Mbit/s region ($^{3.29}/_{0.2} = 16.4\times$ and $^{3.34}/_{0.2} = 16.7\times$) before collapsing at high bandwidth ($^{1.22}/_{0.2} = 6.1\times$ at 1000 Mbit/s). Competing against five contenders does not shift *where* the sweet spot sits, only how large the advantage is which is consistent with what we already saw for base RTT and buffer size. The CUBIC case, however, reproduces the same qualitative flip we saw in Section 5.2.2. Where a single CUBIC contender pushed `client-2` below a fair share at 500 and 1000 Mbit/s, against five CUBIC contenders at the same bandwidths `client-2` still claims $^{2.31}/_{0.2} = 11.5\times$ and $^{1.00}/_{0.2} = 5.0\times$ its fair share respectively. A fixed 200-packet buffer stops being able to sustain our advantage against any *individual* rival once bandwidth outgrows it, but as long as several such rivals are queueing behind the same buffer, `client-2`'s BBR-driven aggression still wins out overall. This buffer-relative-to-bandwidth interaction is exactly what Section 5.2.4 investigates directly, by holding the buffer proportional to the BDP instead of fixing it in absolute packets.

5.2.4 Effect of Varying Bottleneck at Constant Buffer/BDP Ratio

The bandwidth sweep in Section 5.2.3 held the buffer fixed in absolute packets, so part of what we saw there was really a buffer-size effect in disguise: 200 packets is a generous buffer relative to the BDP at 10 Mbit/s and a tiny one at 1000 Mbit/s. Here we instead size the buffer to a fixed multiple of the BDP at every bandwidth (1×, 2×, or 5×) and sweep bandwidth again, to see whether doing so removes bandwidth as a confounding factor.

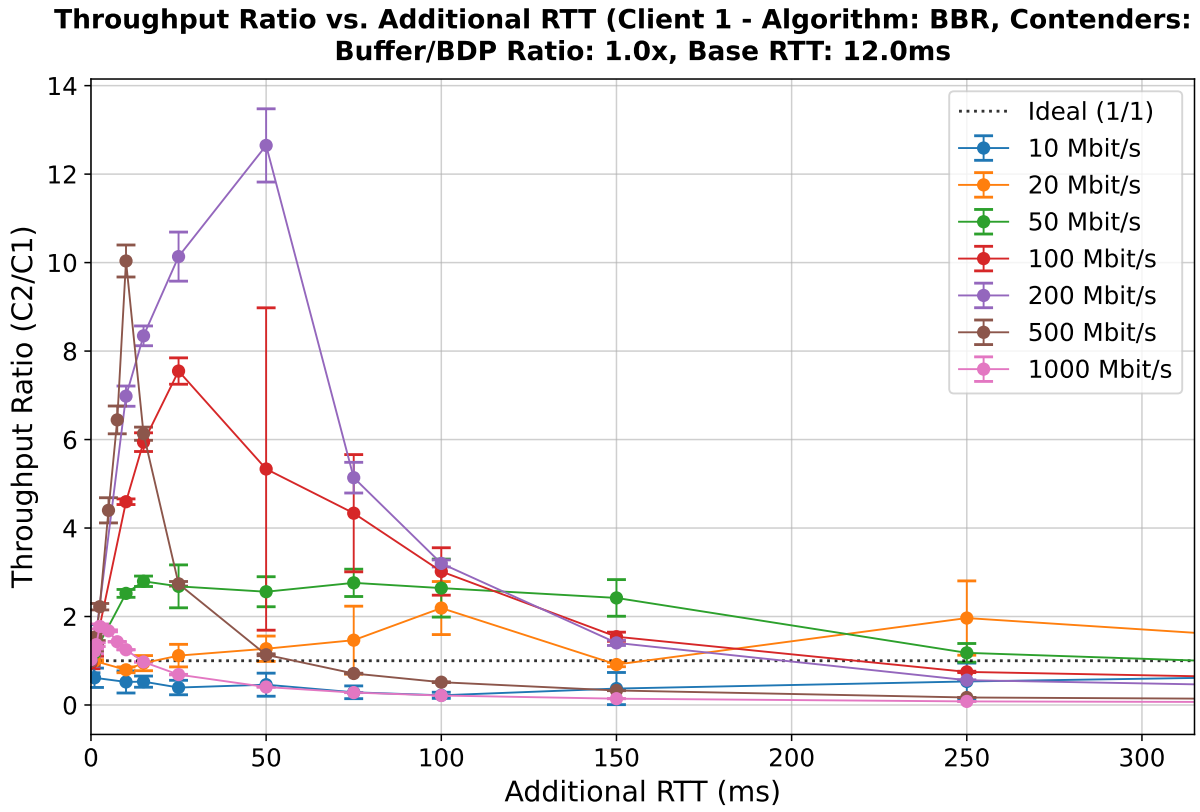


Figure 5.13: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` BBR flow, as a function of additional RTT, buffer fixed at 1.0× the BDP at every bandwidth (12 ms base RTT). See Table 5.13 for the corresponding peak-ratio summary.

Table 5.13: Peak throughput-share ratio summary corresponding to Figure 5.13.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	0	1.27
20	100	2.19
50	15	2.80
100	25	7.55
200	50	12.65
500	10	10.04
1000	2.5	1.78

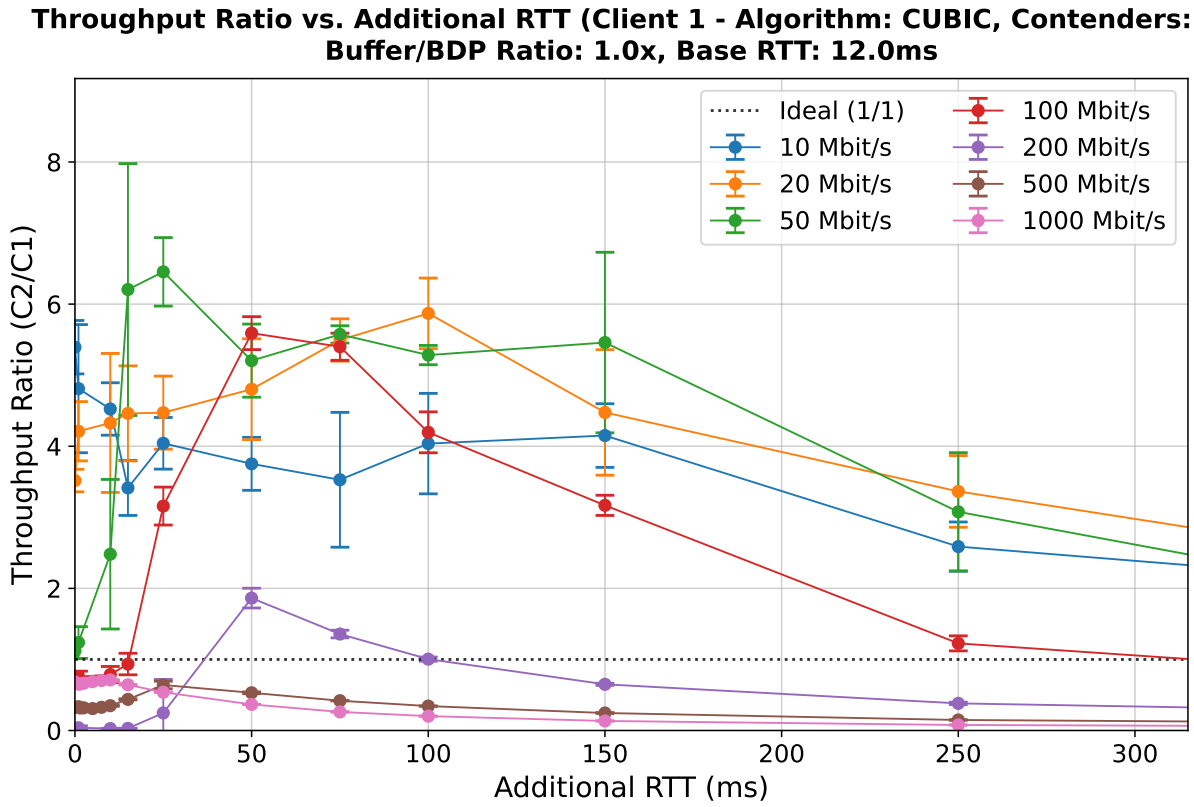


Figure 5.14: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` CUBIC flow, as a function of additional RTT, buffer fixed at 1.0× the BDP at every bandwidth (12 ms base RTT). See Table 5.14 for the corresponding peak-ratio summary.

Table 5.14: Peak throughput-share ratio summary corresponding to Figure 5.14.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	0	5.39
20	100	5.87
50	25	6.45
100	50	5.59
200	50	1.86
500	25	0.64
1000	10	0.71

Figures 5.13 and 5.14 (numeric summary in Tables 5.13 and 5.14) plot the throughput-share ratio of our BBR contender on `client-2` competing against one contender on `client-1` at a bottleneck with the buffer sized to exactly 1× the BDP at each bandwidth tested. If a constant buffer-to-BDP ratio removed bandwidth as a confounding factor entirely, we would expect these peaks to look roughly the same across bandwidths, but they do not. For BBR-vs-BBR the peak ratio still traces the same non-monotonic hump we saw in Section 5.2.3, climbing from 1.27× at 10 Mbit/s to a maximum of 12.65× at 200 Mbit/s before falling back to 1.78× at 1000 Mbit/s. For CUBIC-vs-BBR the peak is highest at low-to-moderate bandwidth (up to 6.45× at 50 Mbit/s), drops back under 2× by 200 Mbit/s (1.86×), and falls below a fair share entirely from 500 Mbit/s

onward even when we were inflating our RTT (0.64× and 0.71× at 500 and 1000 Mbit/s).

Repeating this sweep at 2× and 5× BDP buffer size instead (not plotted here for brevity, but tabulated in full in Section A.1) shows that the *target ratio* itself matters a great deal at low bandwidth but is almost irrelevant at high bandwidth. For BBR-vs-BBR at 10 Mbit/s, going from a 1× to a 5× buffer-to-BDP ratio very nearly triples the peak ratio (1.27× → 2.15× → 3.71×), but at 500 Mbit/s the peak ratio is essentially unchanged regardless of the ratio (10.04× → 9.76× → 9.85×), and at 1000 Mbit/s it is if anything slightly worse with the largest buffer to BDP ratio (1.78× → 1.76× → 1.25×). For CUBIC-vs-BBR a larger buffer-to-BDP ratio erodes the peak ratio almost monotonically across nearly the whole bandwidth range. From 20 Mbit/s upward the peak shrinks every time we grow the buffer-to-BDP ratio from 5.87× → 4.74× → 4.45× when increasing the buffer-to-BDP ratio from 1× → 2× → 5× at 20 Mbit/s, and far more sharply at 100 Mbit/s, from 5.59× at 1× buffer-to-BDP ratio down to 0.96× at 5× buffer-to-BDP ratio. Only at the very lowest bandwidth we tested (10 Mbit/s) does this not hold: the peak stays roughly flat regardless of buffer-to-BDP ratio (5.39×, 4.19×, 4.32×), rather than growing the way it does for BBR-vs-BBR. The two regimes do converge on one point, though: at 1000 Mbit/s the peak ratio for CUBIC-vs-BBR is identical to two decimal places (0.71×) no matter which buffer-to-BDP ratio we used. BBR-vs-BBR is only roughly buffer-invariant at that bandwidth by comparison (1.78× → 1.76× → 1.25×, see above), but in both regimes the buffer has clearly stopped being the limiting factor, just for different underlying reasons: for BBR-vs-BBR the buffer stops mattering because the effect itself has already collapsed at that bandwidth (Section 5.2.3); for CUBIC-vs-BBR it stops mattering because `client-2` has already lost the competition regardless of how the buffer is sized. In short, holding the buffer proportional to the BDP does not make bandwidth irrelevant to how exploitable a connection is. It mainly matters in the low-bandwidth regime, where the buffer stays small in absolute packet terms no matter which multiple of the BDP we target.

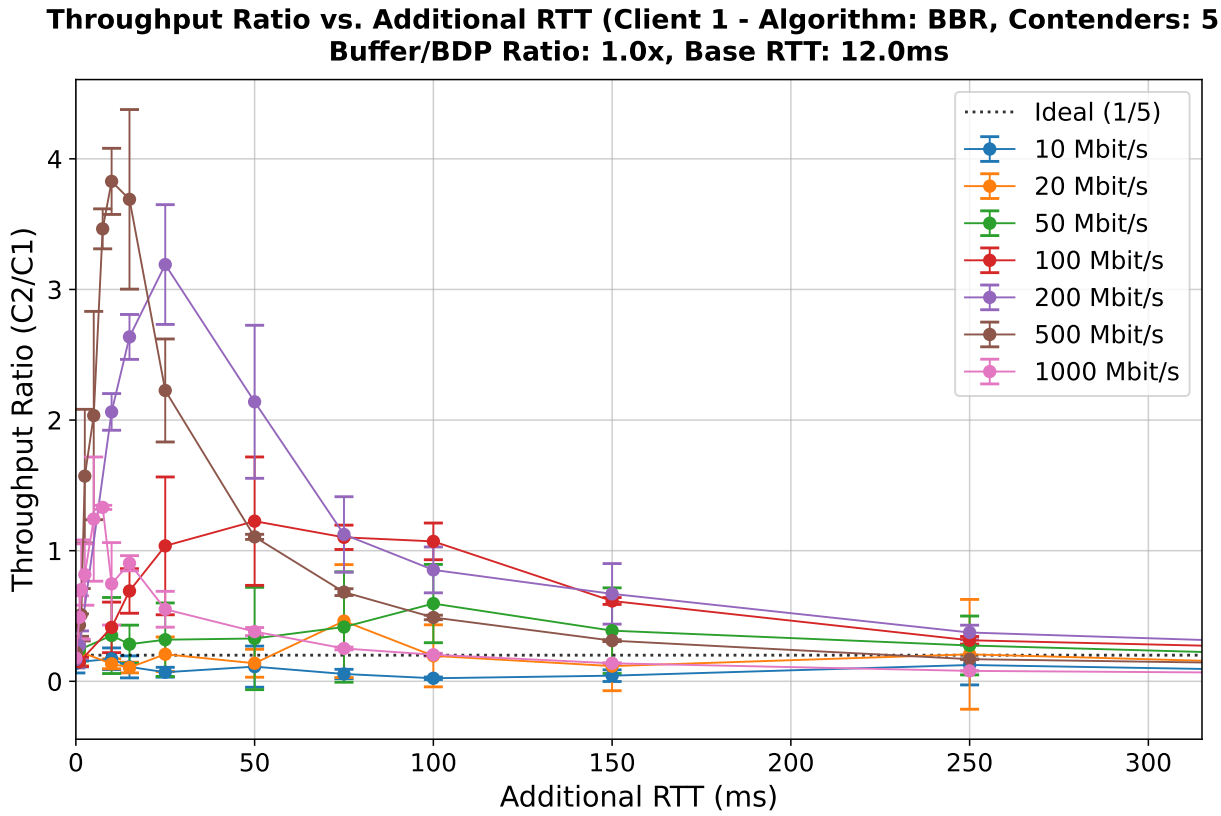


Figure 5.15: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` BBR flows, as a function of additional RTT, buffer fixed at 1.0× the BDP at every bandwidth (12 ms base RTT). See Table 5.15 for the corresponding peak-ratio summary.

Table 5.15: Peak throughput-share ratio summary corresponding to Figure 5.15.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	10	0.17
20	75	0.46
50	100	0.60
100	50	1.23
200	25	3.19
500	10	3.83
1000	7.5	1.33

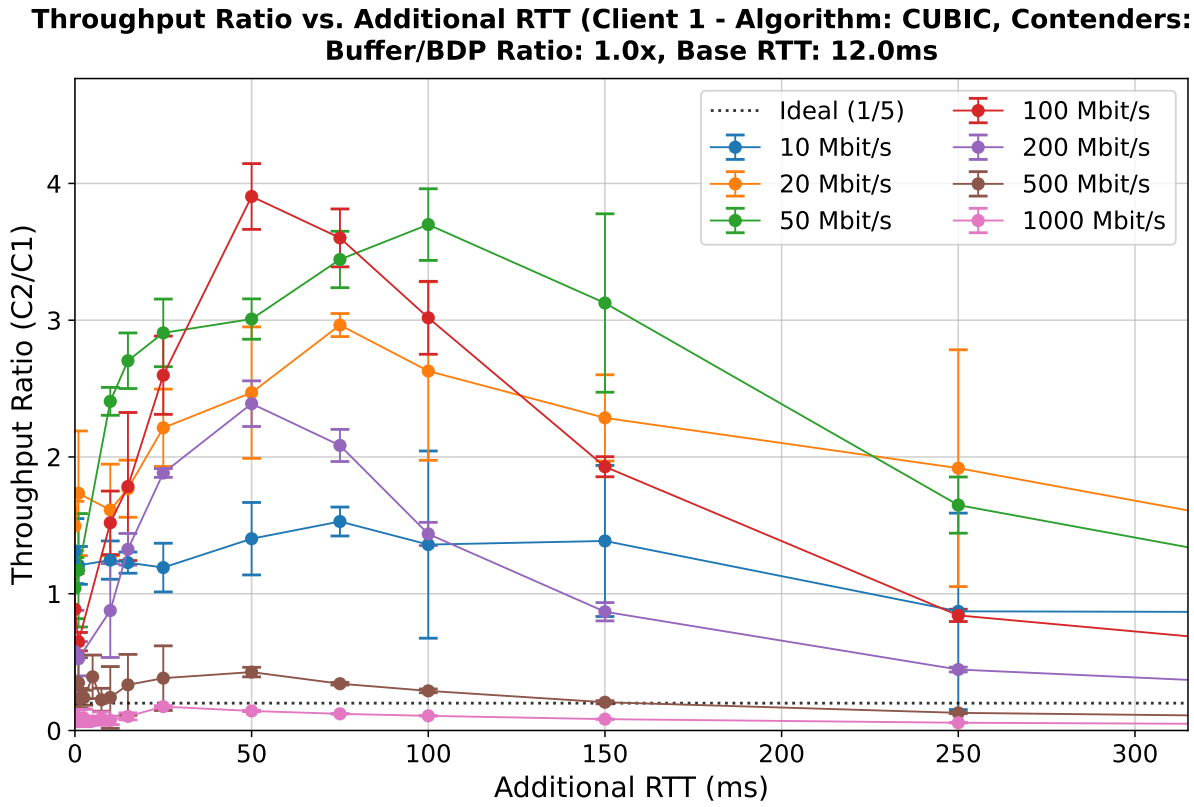


Figure 5.16: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` CUBIC flows, as a function of additional RTT, buffer fixed at 1.0× the BDP at every bandwidth (12 ms base RTT). See Table 5.16 for the corresponding peak-ratio summary.

Table 5.16: Peak throughput-share ratio summary corresponding to Figure 5.16.

Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio
10	75	1.53
20	75	2.96
50	100	3.70
100	50	3.90
200	50	2.39
500	50	0.43
1000	25	0.17

Figures 5.15 and 5.16 (numeric summary in Tables 5.15 and 5.16) repeat the 1× buffer-to-BDP ratio sweep against five contenders. The same qualitative shapes survive. BBR-vs-BBR still peaks in the 200–500 Mbit/s range and CUBIC-vs-BBR still peaks at low-to-moderate bandwidth before collapsing. But dividing by the 0.2 fair share turns up one new edge case: at 10 Mbit/s, BBR-vs-BBR's peak ratio of 0.17 is actually *below* 0.2, i.e. $0.17/0.2 \approx 0.85\times$ – with only a bare 1.0×–buffer-to-BDP ratio to go around and five short-RTT rivals already splitting it, there is no additional delay at which `client-2` manages even a fair share, let alone an advantage. Everywhere else the picture matches what the one-contender experiment already suggested: CUBIC-vs-BBR's sweet spot shifts to somewhat higher bandwidth with more contenders present (peaking around

100 Mbit/s at $^{3.90}/_{0.2} = 19.5\times$ fair share, rather than at 50 Mbit/s as with one contender), and both regimes collapse sharply by 1000 Mbit/s regardless of contender count, though only CUBIC-vs-BBR actually falls below a fair share there (0.71× with one contender, $^{0.17}/_{0.2} = 0.85\times$ with five) – BBR-vs-BBR still holds a modest lead at that bandwidth (1.78× with one contender, $^{1.33}/_{0.2} = 6.65\times$ with five).

The 2×/5× comparison (Section A.1) tells largely the same one-sided story for CUBIC-vs-BBR here as it did with one contender: growing the buffer target erodes the peak almost everywhere. At 100 Mbit/s it falls from 19.5× (at 1× buffer-to-BDP ratio) to $^{2.57}/_{0.2} = 12.9\times$ (at 2× buffer-to-BDP ratio) to $^{1.37}/_{0.2} = 6.9\times$ (at 5× buffer-to-BDP ratio) fair share, and at 500 Mbit/s from 2.15× down to 1.3× down to 0.95×. The largest buffer target is enough to push `client-2` just below a fair share at 500 Mbit/s with five contenders. The one bandwidth where five contenders behave differently from one is again 10 Mbit/s. For such a low bottleneck bandwidth, a bigger buffer target actually *helps* `client-2`, from 7.65× up to 9.15× up to 11.15×, mirroring the low-bandwidth trend we saw for BBR-vs-BBR rather than CUBIC-vs-BBR's usual pattern. And at 1000 Mbit/s the buffer target is once again irrelevant, just as with one contender – `client-2` sits at ≈ 0.85× fair share regardless of whether the buffer is 1×, 2×, or 5× the BDP.

5.2.5 Effect of Receive/Send Window Scaling

So far every parameter we swept was a property of the network path. Here we instead scale `client-2`'s own `net.ipv4.tcp_rmem`/`tcp_wmem` (its advertised receive/send window) from 0.5× to 5× the Linux default, to check whether `client-2`'s own flow-control window, rather than BBR's congestion window, was ever the limiting factor – the server's own window is left untouched throughout, for the same reason we cannot control it on the real Internet either. The default (1×) we scale relative to is Linux's own `tcp_rmem = "4096 131072 6291456"` and `tcp_wmem = "4096 16384 4194304"` (min/default/max, in bytes); 0.5×, 2×, and 5× scale all three values in each triplet by the same factor.

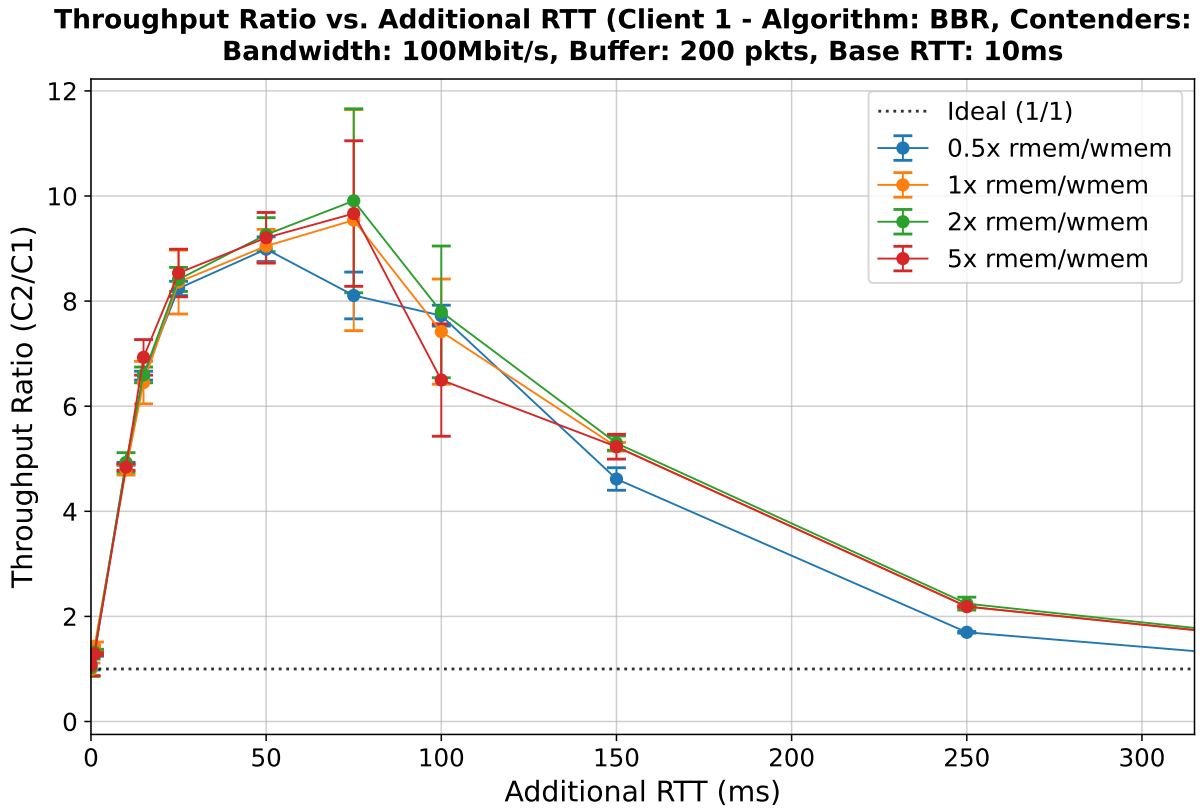


Figure 5.17: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` BBR flow, as a function of additional RTT, for different `client-2` window scales (100 Mbit/s, 200-packet buffer, 10 ms base RTT). See Table 5.17 for the corresponding peak-ratio summary.

Table 5.17: Peak throughput-share ratio summary corresponding to Figure 5.17.

Window Scale	Peak Add. Delay (ms)	Peak Ratio
0.5×	50	8.99
1×	75	9.54
2×	75	9.91
5×	75	9.67

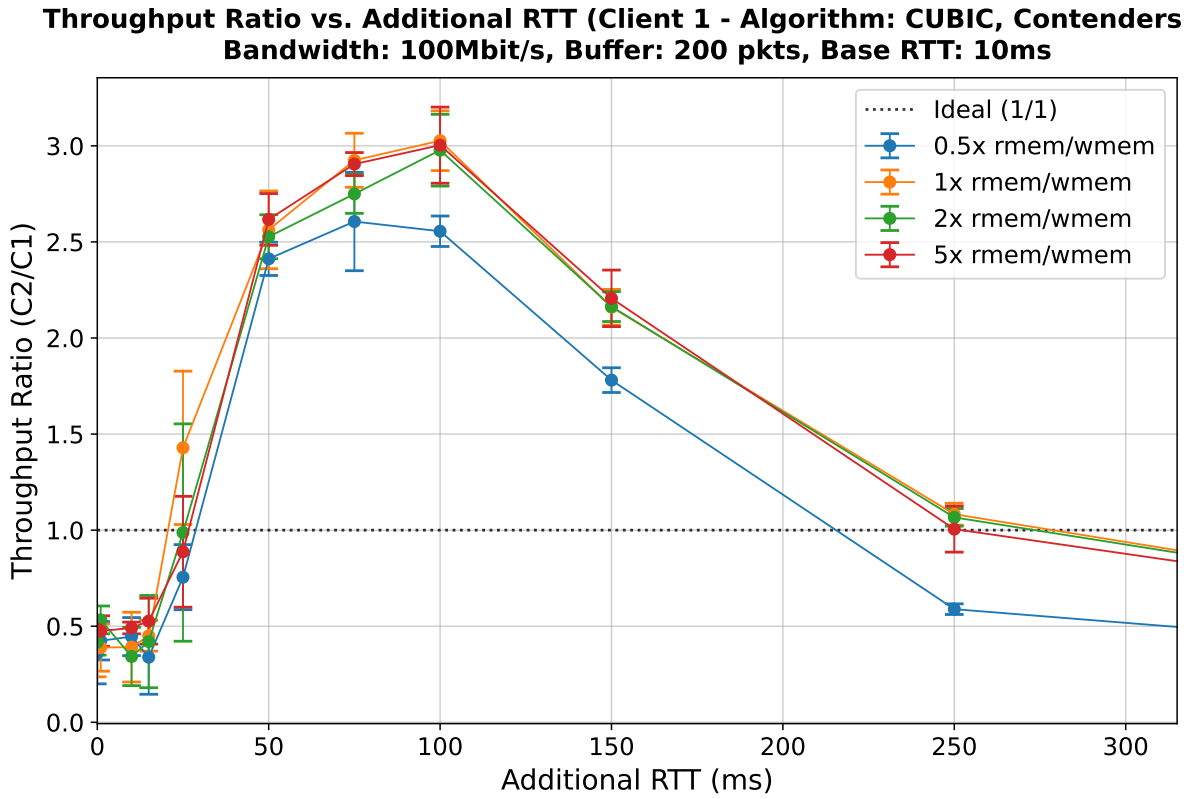


Figure 5.18: Throughput share of `client-2` (BBR, longer RTT) vs. a *single* competing `client-1` CUBIC flow, as a function of additional RTT, for different `client-2` window scales (100 Mbit/s, 200-packet buffer, 10 ms base RTT). See Table 5.18 for the corresponding peak-ratio summary.

Table 5.18: Peak throughput-share ratio summary corresponding to Figure 5.18.

Window Scale	Peak Add. Delay (ms)	Peak Ratio
0.5×	75	2.61
1×	100	3.03
2×	100	2.98
5×	100	3.00

Figures 5.17 and 5.18 (numeric summary in Tables 5.17 and 5.18) show the result of a single BBR flow on `client-2` competing at a bottleneck of 100 Mbit/s with a 200-packet buffer and 10 ms base RTT against a single BBR or CUBIC contender on `client-1` at varying window scales. Enlarging the window beyond the default (2×, 5×) never produces a reliable improvement in throughput ratio over 1× for either algorithm – BBR-vs-BBR moves between 9.54× and 9.91× and CUBIC-vs-BBR between 2.98× and 3.03×, differences well within run-to-run noise. *Shrinking* the window to 0.5×, however, does cost `client-2` something in both cases, just not by much here: 8.99× for BBR-vs-BBR (a 5.8% drop from 1×) and 2.61× for CUBIC-vs-BBR (a 13.9% drop). So when competing against a single contender, the receive and send windows are clearly not the binding constraint. BBR's congestion window governs how much `client-2` can claim, and only an undersized flow-control window starts to get in its way.

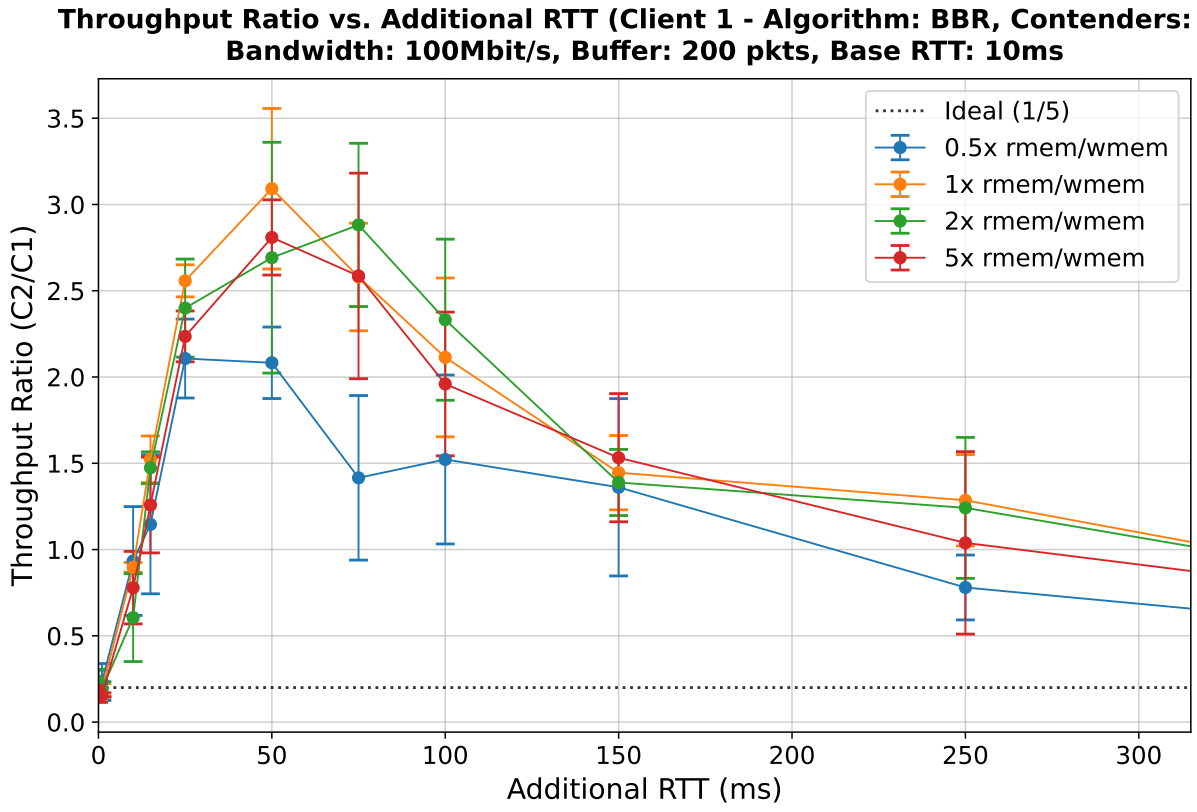


Figure 5.19: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` BBR flows, as a function of additional RTT, for different `client-2` window scales (100 Mbit/s, 200-packet buffer, 10 ms base RTT). See Table 5.19 for the corresponding peak-ratio summary.

Table 5.19: Peak throughput-share ratio summary corresponding to Figure 5.19.

Window Scale	Peak Add. Delay (ms)	Peak Ratio
0.5×	25	2.11
1×	50	3.09
2×	75	2.88
5×	50	2.81

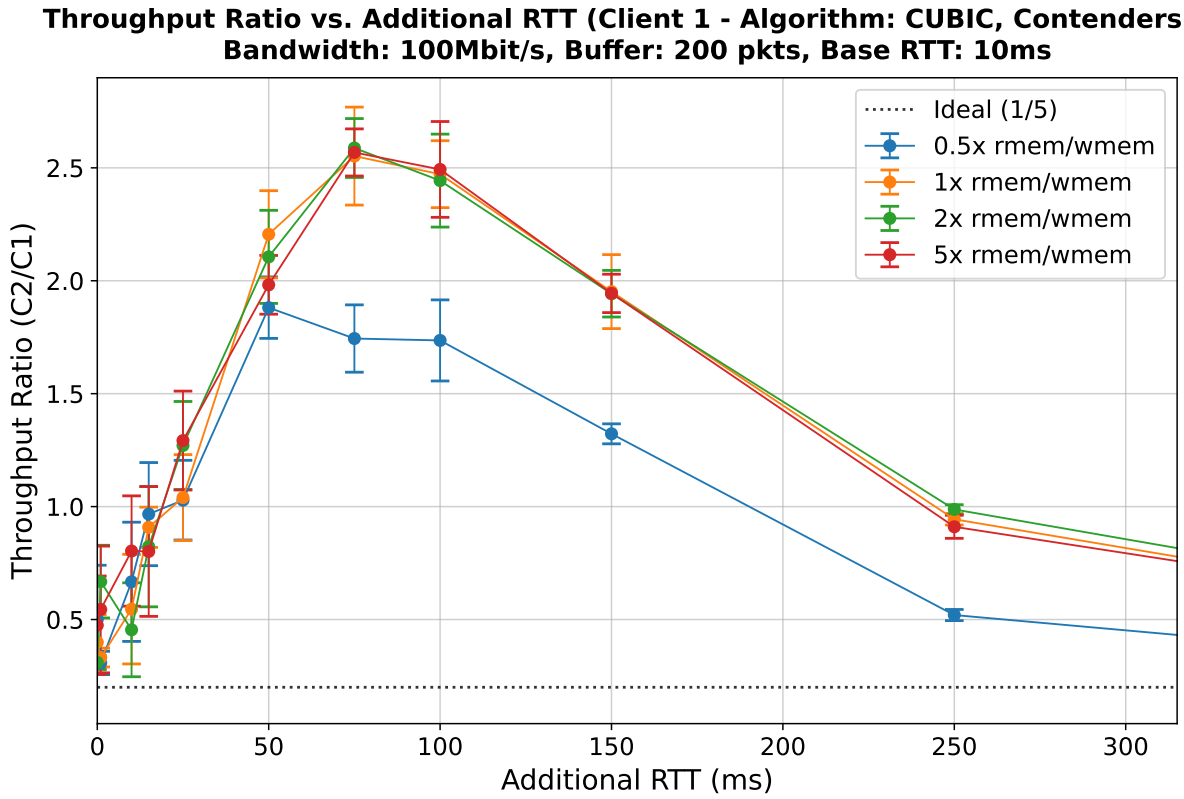


Figure 5.20: Throughput share of `client-2` (BBR, longer RTT) against *five* competing `client-1` CUBIC flows, as a function of additional RTT, for different `client-2` window scales (100 Mbit/s, 200-packet buffer, 10 ms base RTT). See Table 5.20 for the corresponding peak-ratio summary.

Table 5.20: Peak throughput-share ratio summary corresponding to Figure 5.20.

Window Scale	Peak Add. Delay (ms)	Peak Ratio
0.5×	50	1.88
1×	75	2.55
2×	75	2.59
5×	75	2.57

Figures 5.19 and 5.20 (numeric summary in Tables 5.19 and 5.20) repeat the same tests against five contenders. Just like in the one-contender case, increasing the window size doesn't improve the throughput ratio of `client-2` for both BBR-vs-BBR and CUBIC-vs-BBR runs. The throughput ratio we get at 2× and 5× the default value sit within noise of the ratio we get at the default value itself. The 0.5× case on the other hand now causes `client-2` to lose more throughput share compared to when it was only competing against one contender. For BBR-vs-BBR it grows from a 5.8% dip at one contender to a 31.7% dip at five contenders ($^{3.09}/_{0.2} = 15.45\times \rightarrow {}^{2.11}/_{0.2} = 10.55\times$). For CUBIC-vs-BBR the gap grows more modestly from 13.9% at the default window sizes to a 26.3% decrease of throughput ratio at 0.5× the window size ($^{2.55}/_{0.2} = 12.75\times \rightarrow {}^{1.88}/_{0.2} = 9.4\times$). We see the same pattern for every other base RTT we tested (5, 20, and 50 ms; full data in Section A.2). We have to note though that for BBR-vs-BBR against a single contender at 5 ms base RTT specifically, the average throughput ratio actually slightly increased when we decreased

the window size (10.52× at 0.5× vs. 10.37× at 1×), but as the value is again well inside of what we could call noise, and every other configuration we tested (including BBR-vs-BBR at five contenders, and CUBIC-vs-BBR at any contender count, both still at 5 ms) showed the usual drop, we didn't venture into this anomaly deeper.

We also did not vary the server's own window: the real workloads in Sections 5.3 and 5.4 run against servers we do not control anyway, so results from scaling the server's window here would not generalise to the actual attack scenario this thesis evaluates.

5.3 File Download Completion Time

Having established that the exploit works on synthetic `iperf3` throughput, we now switch on the ACK-delaying client itself (Section 4.2) and measure a real workload: two clients simultaneously download the same file (Go's `go1.22.3.linux-amd64.tar.gz`, ≈69 MB, from `dl.google.com`) over the real Internet (Real-Internet topology, Section 5.1.1). For all our file download tests we only competed against one contender on `client-1`, at a bottleneck of 40 Mbit/s with a 160-packet buffer and 1.5 ms of base RTT, resulting in a buffer-to-BDP ratio of 32× relative to the uninflated path. We chose such a deep buffer so the buffer itself would not cap our additional-RTT sweep too early; as Section 5.3.2 shows, though, once `client-2`'s `delay_acks`-inflated perceived RTT pushes its desired in-flight budget past what our 160-packet buffer (plus the tiny real pipe) can actually sustain – around 50 ms of additional delay – the buffer itself becomes the binding constraint, which is a direct contributor to the peak-then-decline shape the ratio traces at larger target delays. With this configuration fixed, we sweep the additional RTT from 0 to 1000 ms for each of the three strategies explained in Section 4.2.1. Our metric here is file download completion time (FCT), and we report the ratio $^{\mathrm{FCT}_{c1}}/_{\mathrm{FCT}_{c2}}$, where a ratio above 1 means that `client-2` finished faster than `client-1`, meaning that `client-2` got an advantage over `client-1` in throughput share at the bottleneck.

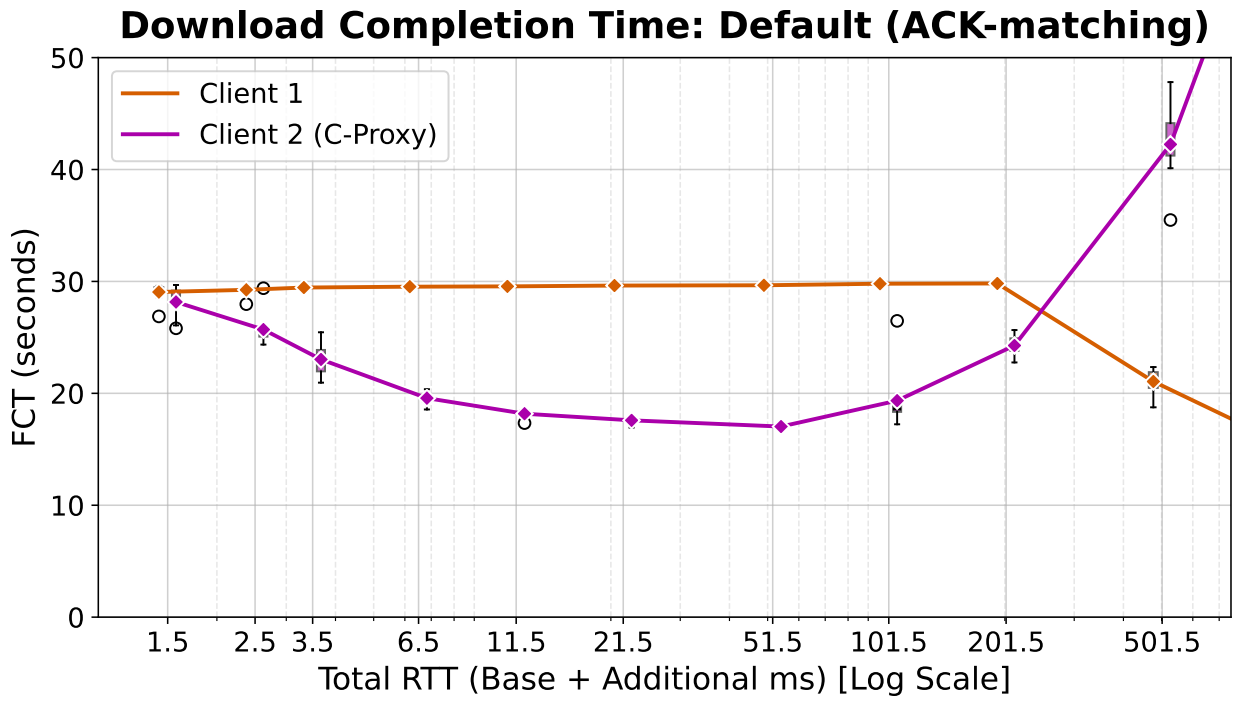


Figure 5.21: File download completion time of `client-1` vs. `client-2` over the real Internet, `client-2` running the **default (ACK-matching)** strategy, as a function of its target additional RTT (40 Mbit/s, 160-packet buffer, 1.5 ms base RTT, 32× BDP). The x-axis is cropped at 500 ms so the 1000 ms point does not compress the rest of the sweep; see Table 5.21 for that value.

Table 5.21: Numeric summary corresponding to Figure 5.21 (**default** strategy): mean FCT of `client-1` and `client-2` and their ratio $\mathrm{FCT_{c1}}/\mathrm{FCT_{c2}}$ for each target additional RTT.

Add. Delay (ms)	`client-1` FCT (s)	`client-2` FCT (s)	Ratio
0	29.05	28.16	1.032
1	29.25	25.70	1.138
2	29.45	23.03	1.279
5	29.53	19.57	1.509
10	29.56	18.18	1.626
20	29.63	17.59	1.685
50	29.66	17.03	1.741
100	29.79	19.35	1.540
200	29.82	24.27	1.228
500	21.06	42.25	0.498
1000	15.99	70.59	0.227

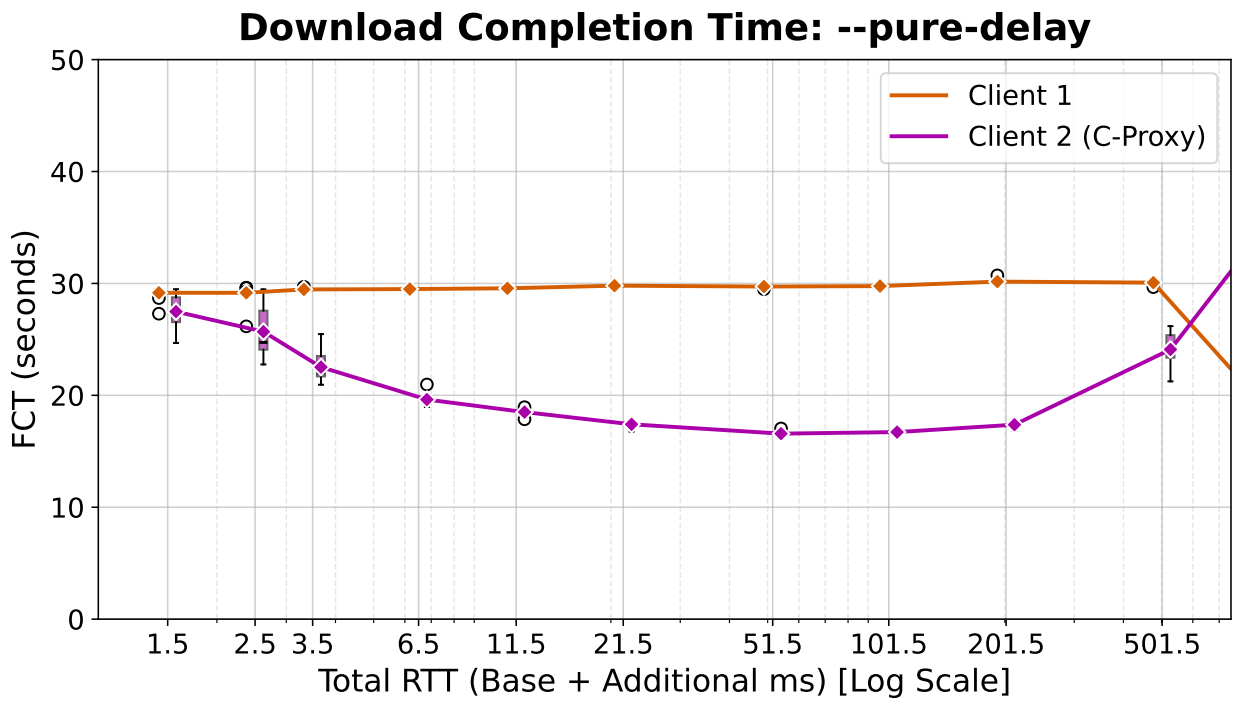


Figure 5.22: File download completion time of `client-1` vs. `client-2` over the real Internet, `client-2` running the `--pure-delay` strategy, as a function of its target additional RTT (40 Mbit/s, 160-packet buffer, 1.5 ms base RTT, 32× BDP). The x-axis is cropped at 500 ms so the 1000 ms point does not compress the rest of the sweep; see Table 5.22 for that value.

Table 5.22: Numeric summary corresponding to Figure 5.22 (`--pure-delay` strategy): mean FCT of `client-1` and `client-2` and their ratio $^{\mathrm{FCT_{c1}}}/_{\mathrm{FCT_{c2}}}$ for each target additional RTT.

Add. Delay (ms)	`client-1` FCT (s)	`client-2` FCT (s)	Ratio
0	29.16	27.48	1.061
1	29.16	25.69	1.135
2	29.46	22.52	1.308
5	29.49	19.62	1.503
10	29.56	18.50	1.598
20	29.80	17.41	1.711
50	29.72	16.58	1.793
100	29.76	16.71	1.781
200	30.16	17.38	1.736
500	30.07	24.09	1.248
1000	18.31	37.77	0.485

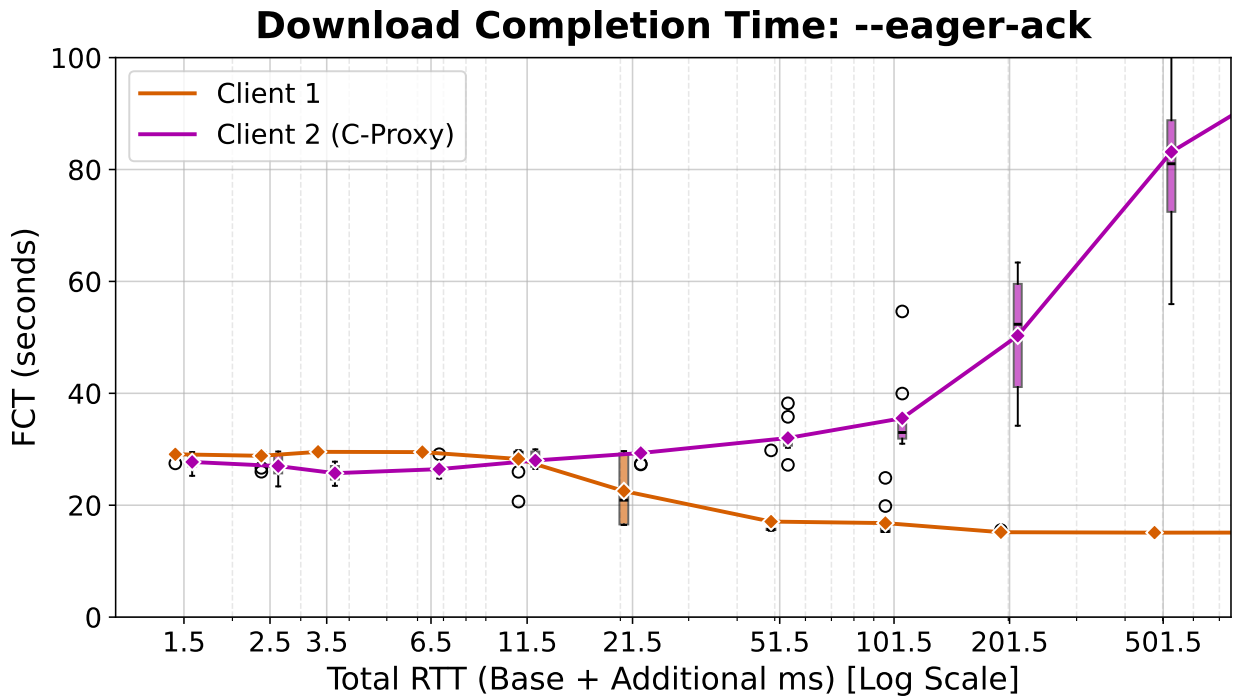


Figure 5.23: File download completion time of `client-1` vs. `client-2` over the real Internet, `client-2` running the `--eager-ack` strategy, as a function of its target additional RTT (40 Mbit/s, 160-packet buffer, 1.5 ms base RTT, 32× BDP). The x-axis is cropped at 500 ms so the 1000 ms point does not compress the rest of the sweep; see Table 5.23 for that value.

Table 5.23: Numeric summary corresponding to Figure 5.23 (`--eager-ack` strategy): mean FCT of `client-1` and `client-2` and their ratio $\mathrm{FCT_{c1}}/\mathrm{FCT_{c2}}$ for each target additional RTT.

Add. Delay (ms)	`client-1` FCT (s)	`client-2` FCT (s)	Ratio
0	29.10	27.73	1.050
1	28.83	27.00	1.068
2	29.54	25.73	1.148
5	29.49	26.45	1.115
10	28.26	28.00	1.009
20	22.53	29.33	0.768
50	17.07	32.01	0.533
100	16.81	35.55	0.473
200	15.18	50.29	0.302
500	15.10	83.15	0.182
1000	15.11	95.72	0.158

All three strategies reproduce the shape we would expect from Section 5.2: `client-2` finishes increasingly faster than `client-1` as target additional RTT grows, up to a peak, before the advantage erodes and eventually reverses at very large target delays. Where the three strategies differ is in whether the sophistication of the release logic actually pays off. We had expected the default ACK-matching strategy, which opportunistically stamps payload packets with an already-delayed ACK instead of always waiting for the fixed deadline like `--pure-delay` does, to noticeably

outperform simply delaying everything. Figures 5.21 and 5.22 (numeric summary in Tables 5.21 and 5.22) show that it does not: at the peak (50 ms target) `--pure-delay` is if anything slightly *better* (1.793× vs. 1.741×), and at 100 ms and 200 ms `--pure-delay` is clearly better (1.781× and 1.736× vs. 1.540× and 1.228× for the default strategy). Across the whole sweep, the two strategies track each other closely enough that we cannot say the additional bookkeeping in the default strategy buys us anything on the workload of downloading a file. Simply delaying every outgoing packet by a fixed amount is, empirically, just as good if not better.

`--eager-ack`, on the other hand, is a clear regression rather than a refinement. Figure 5.23 (with values in Table 5.23) shows that its ratio never climbs much above 1.1× even at its best point, 1.148× at 2 ms additional delay, is already back down to essential breakeven by 10 ms (1.009×), and drops clearly below 1 from 20 ms onward (0.768×) – right where the other two strategies are still climbing toward their peak of 1.6–1.8×. Past that point it keeps getting worse, bottoming out at 0.158× at 1000 ms additional delay, where it takes `client-2` over six times longer than `client-1` to finish the same download. Pairing outgoing payload with an ACK that has not yet reached its hold-back deadline releases that ACK to the server earlier than intended, directly undercutting the RTT sample the server's BBR sender measures for that packet. Since the whole exploit depends on inflating that RTT sample so BBR overestimates the BDP and admits more data in flight (Section 3.1.3), an ACK let out early hands back exactly the inflation the other two strategies preserve: the server sees a smaller RTT, computes a smaller BDP, and grants `client-2` a correspondingly smaller in-flight budget – consistent with the throughput regression observed here.

One pattern worth flagging: in all three tables (Tables 5.21 to 5.23), `client-1`'s own FCT drops at the largest target delays even though `client-1` is never touched by `delay_acks` at all. We believe this is simply the mirror image of `client-2`'s own collapse at those same delays, rather than an independent effect on `client-1`: once the target additional RTT grows large enough, the inflated RTT estimate stops paying off and `client-2`'s own throughput collapses instead (the same reversal already visible as the Ratio column dropping below 1). `client-1` and `client-2` share the same bottleneck for the whole run, so whatever bandwidth `client-2` fails to use as it starves itself becomes available to `client-1` – which is exactly where `client-1` speeds up. This tracks the data closely: for the default strategy `client-1` already speeds up at 500 ms, exactly where the ratio has already dropped below 1; for `--pure-delay`, where the ratio is still above 1 at 500 ms, `client-1`'s FCT has *not* dropped yet either, and only does so at 1000 ms once the ratio falls below 1 there too.

5.3.1 Packet and ACK Release Delay

To see *why* the three strategies behave so differently, we now look directly at how long `delay_acks` actually held each packet and ACK, using the per-packet logs it writes internally (Section 4.2.2). We pick the 10 ms target-delay configuration as a representative example and pool all 10 repeats of each strategy together (rather than looking at a single run) into fixed 1 ms-wide histograms (0–1 ms, 1–2 ms, and so on).

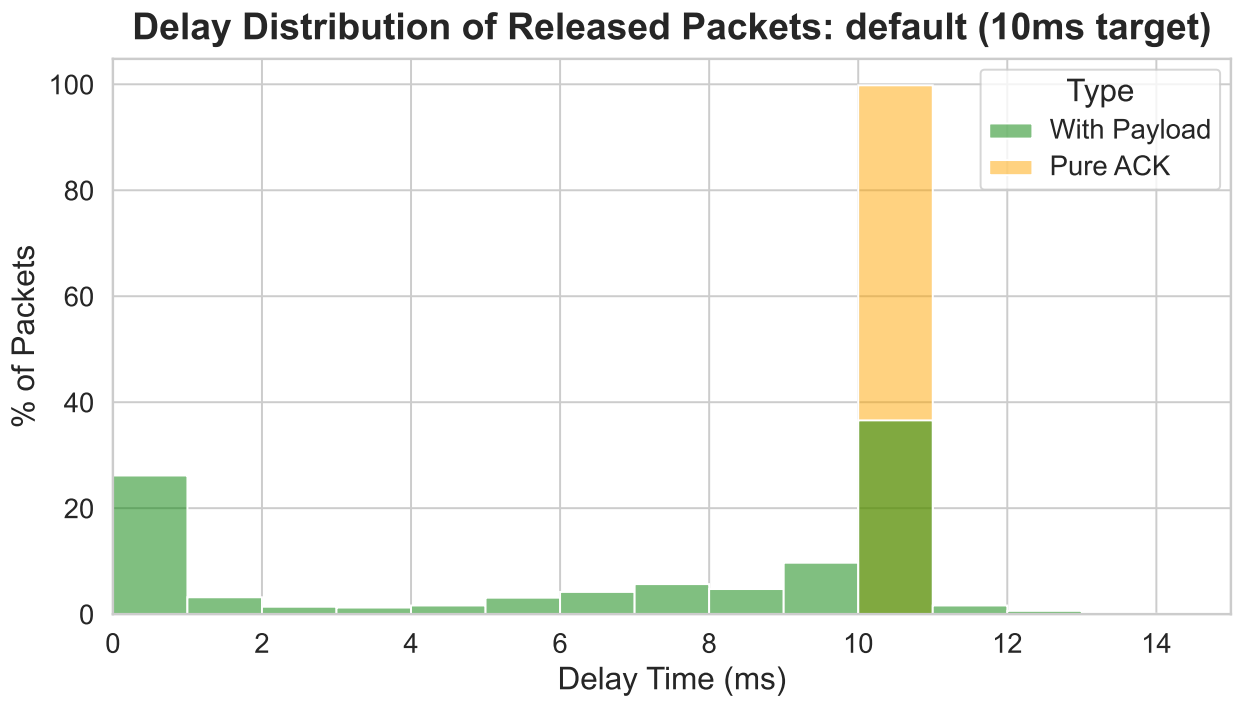


Figure 5.24: Delay distribution of released packets, **default** strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.24: Numeric summary corresponding to Figure 5.24 (**default** strategy, 10 ms target delay, pooled across 10 repeats).

	With Payload	Pure ACK
Released packets	1288	219838
Average delay (ms)	6.562	10.294

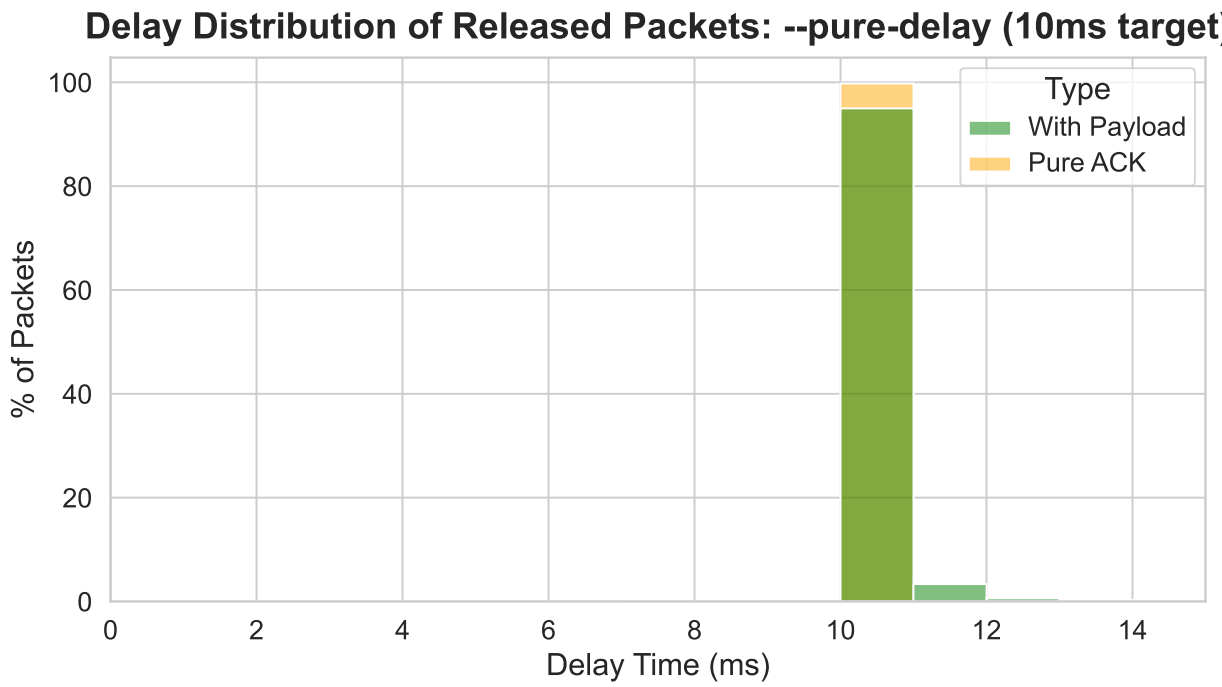


Figure 5.25: Delay distribution of released packets, `--pure-delay` strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.25: Numeric summary corresponding to Figure 5.25 (`--pure-delay` strategy, 10 ms target delay, pooled across 10 repeats).

	With Payload	Pure ACK
Released packets	1314	234782
Average delay (ms)	10.326	10.293

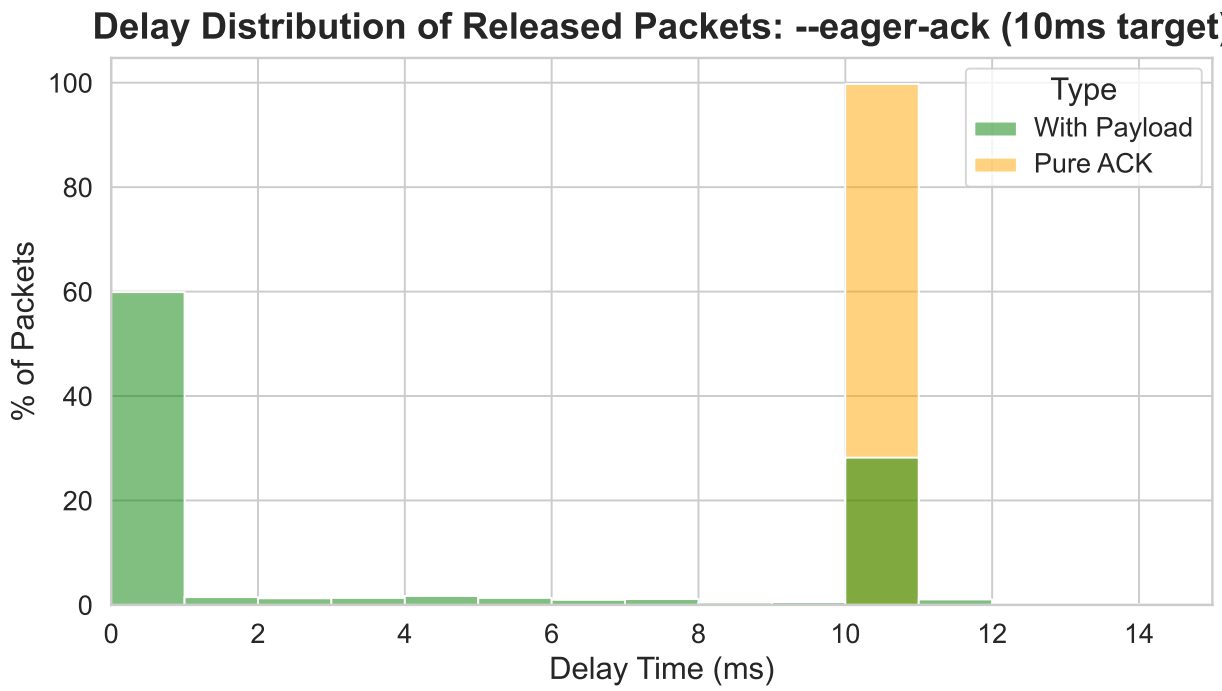


Figure 5.26: Delay distribution of released packets, `--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.26: Numeric summary corresponding to Figure 5.26 (`--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats).

	With Payload	Pure ACK
Released packets	1350	200056
Average delay (ms)	3.768	10.300

The three plots (Figures 5.24 to 5.26) confirm the mechanism directly. Under `--pure-delay` (Figure 5.25), every packet – payload or pure ACK – is queued and released unmodified exactly at the target delay by construction, and the payload average of 10.326 ms in Table 5.25 reflects exactly that: there is no path in this strategy for a payload packet to leave early. Under the default strategy (Figure 5.24), payload packets are released noticeably earlier on average (6.562 ms as seen in Table 5.24), because the opportunistic ACK-stamping described in Section 4.2.1 lets a payload packet go out as soon as *any* already-expired ACK becomes available, rather than waiting for its own deadline. `--eager-ack` (Figure 5.26) pushes this further still, to an average of 3.768 ms (Table 5.26), since it will also pair payload with an ACK that has not even expired yet. Pure-ACK packets, in contrast, sit at essentially the same $\approx$ 10.3 ms average in all three strategies – unsurprising, since none of the three strategies changes how a lone ACK (one with no payload attached to it) gets released. The default strategy's lower payload-release delay is a secondary property beyond the RTT-inflation trade-off itself; Section 6.3.3 discusses it further as future work.

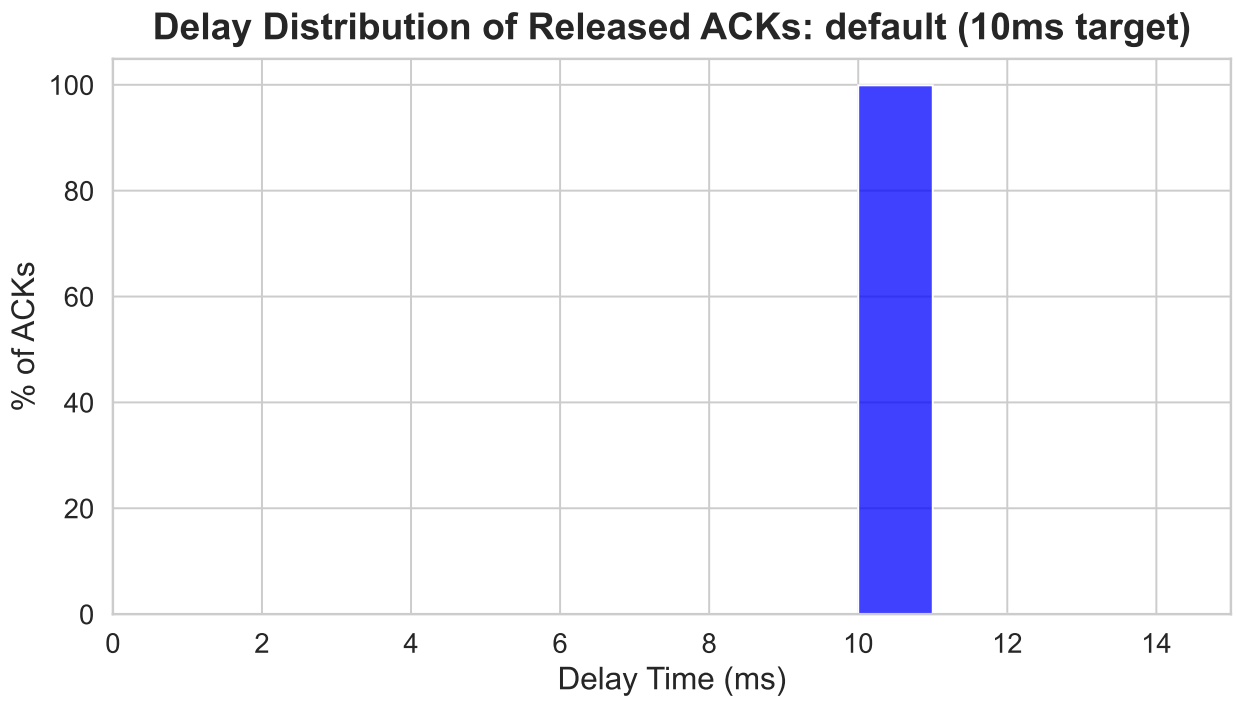


Figure 5.27: Delay distribution of released ACKs, **default** strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.27: Numeric summary corresponding to Figure 5.27 (**default** strategy, 10 ms target delay, pooled across 10 repeats).

	Value
Released ACKs	205629
Dropped ACKs	19088
Average delay (ms)	10.306
Minimum delay (ms)	10.000

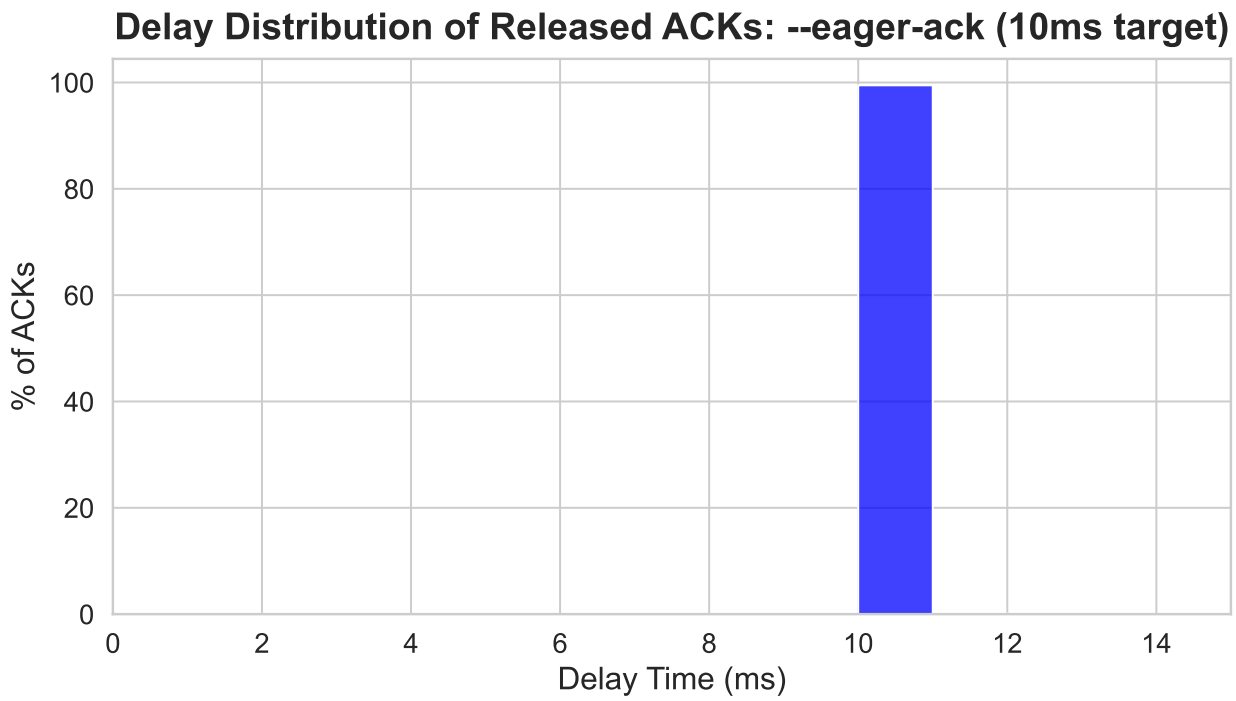


Figure 5.28: Delay distribution of released ACKs, `--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.28: Numeric summary corresponding to Figure 5.28 (`--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats).

	Value
Released ACKs	191265
Dropped ACKs	10994
Released before 10 ms target	759
Average delay (ms)	10.277
Minimum delay (ms)	0.001

There is no ACK-delay plot for `--pure-delay`: that strategy never inspects or re-stamps ACK numbers at all (Section 4.2.1), so no "released ACK" bookkeeping exists for it in the first place. Comparing the remaining two (Tables 5.27 and 5.28) tells a cleaner story than we originally expected. Of the 224 717 candidate ACKs seen under the default strategy, 19 088 (8.49%) were dropped rather than released – coalesced away because a later, more up-to-date ACK made them redundant before their hold-back deadline; the corresponding figure for `--eager-ack` is 10 994 dropped out of 202 259 candidates (5.44%). Under the default strategy, every single one of the 205 629 released ACKs was held for *at least* the full 10 ms target – the minimum recorded delay across all 10 repeats is exactly 10.000 ms, with zero exceptions. There is no early-release anomaly to explain away here: the "keep half the ACKs in reserve" bookkeeping (Section 4.2.1) never lets one out ahead of schedule. Under `--eager-ack`, by contrast, a small but real fraction of ACKs – 759 out of 191 265, or 0.397% – are released before the 10 ms target, some as early as 0.001 ms. This is the eager-pairing mechanism doing exactly what it is designed to do: an ACK that gets grabbed to stamp an outgoing payload packet before its own deadline expires is, by definition, released early,

and its logged delay time reflects that honestly.

The one place we do see values that make no physical sense is a handful of payload packets under `--eager-ack`: 6 out of 1350 have a recorded delay of a fraction of a millisecond *below zero* (as low as -0.013 ms), which would mean the packet was released before it ever arrived. Since these are both extremely rare (0.44% of payload packets in this strategy) and vanishingly small in magnitude, we attribute them to floating-point or clock-read noise right at the release boundary in `delay_acks`' timing instrumentation. There is no equivalent negative-delay case anywhere in the ACK data, which is where an actual logic error in the release ordering would be far more likely to show up.

5.3.2 Per-Flow Dynamics

The FCT sweep in Figure 5.21 and table 5.21 shows the default strategy reaching its peak advantage of $1.741\times$ at 50 ms target delay, yet `client-1`'s own FCT barely moves (29.66 s vs. a 29.05 s baseline at 0 ms additional delay). To see *how* `client-2` wins this much bandwidth without visibly hurting `client-1`, we zoom into a single representative repeat of exactly this configuration and look directly at the two clients' download flows at packet level, using the capture-and-analysis pipeline described in Section 5.1.3.

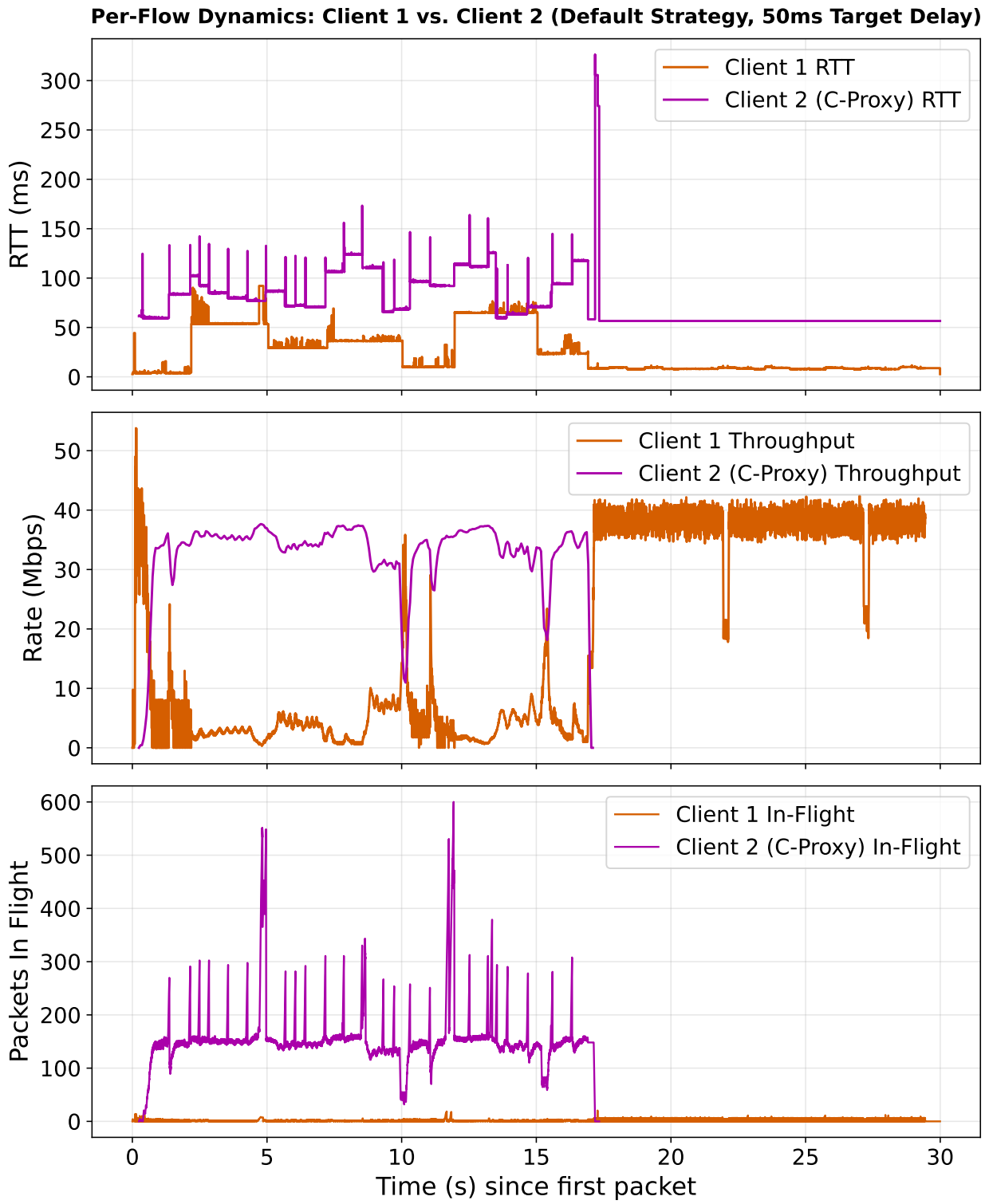


Figure 5.29: Round-trip time, throughput, and packets in flight of `client-1`'s and `client-2`'s download flows over time, for a single representative repeat of the default strategy at 50 ms target additional delay (the peak-ratio configuration from Table 5.21). Both clients start downloading the same file at $t = 0$.

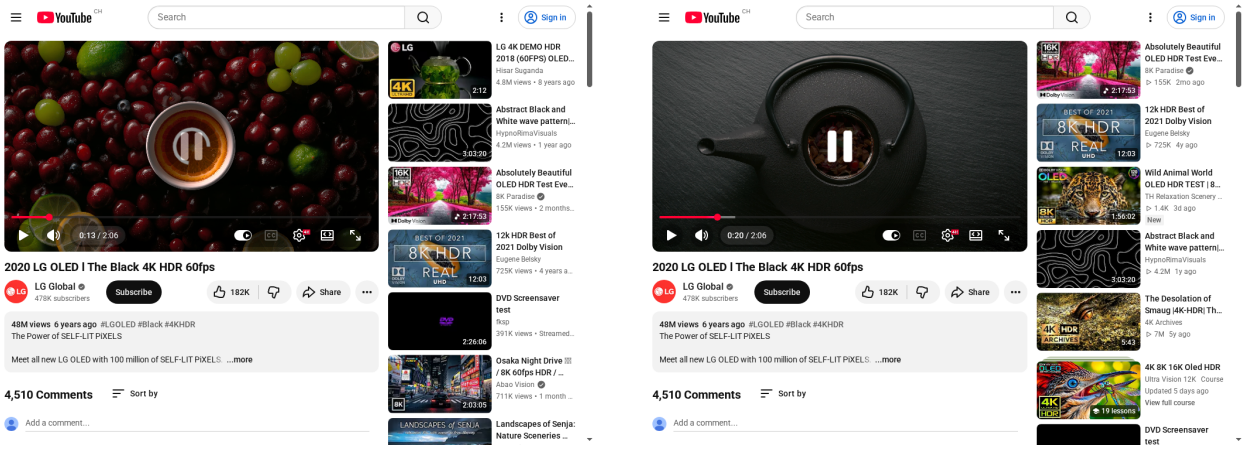
As Figure 5.29 shows, all three panels tell the same story in lockstep. `client-2`'s RTT (top panel) is inflated to a sawtoothing 60–170 ms for the whole run – `delay_acks` at work – while `client-1`'s stays low, close to the real 1.5 ms base RTT while occasionally also being increased to over 50 ms for the duration `client-2`'s file download is not complete yet. `client-1`'s throughput (middle panel) briefly overshoots to ≈53 Mbit/s in the first fraction of a second before collapsing – consistent with its tiny real RTT letting it race through BBR's `STARTUP` ramp (Section 3.1.2) far faster, in wall-clock time, than `client-2`'s inflated perceived RTT allows; `client-2`'s own `STARTUP` ramp is correspondingly slower, but once it catches up it saturates the ≈40 Mbit/s bottleneck (middle panel) and squeezes `client-1` down to a trickle for the rest of the run; correspondingly, `client-2` sustains a huge in-flight budget of 100–600 packets (bottom panel) against `client-1`'s near-zero, since BBR sizes its in-flight allowance to a multiple of (perceived RTT) × (bottleneck bandwidth) and `client-2`'s perceived RTT is roughly 50–100× larger. Notably, several of `client-2`'s in-flight spikes (up to 600 packets) overshoot what our topology can actually hold without dropping: the real pipe (5 packets at 40 Mbit/s and 1.5 ms base RTT) plus the 160-packet buffer caps out at 165 packets, equivalent to a perceived RTT of only $165 \times 8 \times 1500$ bits/40 Mbit/s ≈ 49.5 ms – almost exactly the 50 ms target delay this run was captured at. In other words, this specific configuration sits right at the edge of what the buffer can sustain, which is why the peak ratio occurs here rather than at a larger target delay: beyond this point, inflating the RTT further no longer buys `client-2` a proportionally larger sustainable in-flight window, it just pushes more of that window into outright drops. This holds until roughly $t \approx 17$ s, when `client-2`'s download finishes and all three of its curves collapse to zero at once – and `client-1` instantly claims the entire link, jumping straight to ≈40 Mbit/s and finishing shortly after, needing only a handful of packets in flight to do so given its own tiny real RTT. This is exactly why `client-1`'s FCT is untouched by `client-2`'s advantage: rather than the two clients sharing the bottleneck throughout the whole download and `client-1` suffering a lower average rate for its entire transfer, `client-2` takes almost the whole link while it is active and then gets out of the way entirely, handing the full bottleneck to `client-1` for the remainder of its download. `client-1`'s transfer is delayed in *when* it gets its bandwidth, not in *how much* of it it eventually gets.

This mirrors a phenomenon Ma et al. [10] document directly: in their Fig. 1, a 10 ms RTT flow running alone at first fully saturates a bottleneck, but "later when the 50-ms RTT flow joins, the 10-ms RTT flow quickly gives up bandwidth" [10], settling on a small fraction of its earlier rate. The mechanism they identify – a higher-RTT BBR flow probes for longer and floods proportionally more excess traffic into the queue, letting it dominate the backlog and delivery rate soon after it is present – is the same mechanism at work here, just induced synthetically by `delay_acks` rather than by a genuinely longer network path. The rapid takeover we observe is consistent with their finding that this bias develops quickly and does not require a large head start to become dominant – if anything, `client-1`'s own brief `STARTUP` lead is an even shorter head start than the established flow in their experiment, yet `client-2` still takes over just as fast.

5.4 Video Playback

Having studied file downloads, a workload where only the total completion time matters, we now turn to a latency-sensitive workload where the moment-to-moment delivery rate matters just as much as the total amount of data transferred: watching a YouTube video. Using the same Real-Internet topology, ACK-delaying client, and three release strategies as in Section 5.3, we again set `client-2` (running `delay_acks`) in competition against a single contender on `client-1`, this time at a bottleneck of 20 Mbit/s with an 80-packet buffer and 1.5 ms base RTT – a buffer-to-BDP ratio

of 32×, deliberately identical to the file-download configuration. As described in Section 5.1.4, each client runs a headless Selenium-driven Chrome instance that opens the same public YouTube video – a 4K/60fps HDR demo clip – forced to 2160p60 and plays it for a fixed 30-second wall-clock window; our metric is *playback progress*, i.e., how many seconds of the video the player actually advanced through in that window, read back from the player's own on-screen timer. Because playback only advances while the buffer holds enough data to keep playing, a lower playback progress within the same fixed real-time window is a direct symptom of rebuffering/stalling, making this a Quality of Experience (QoE) proxy in a way pure completion time is not: two clients could tie on total bytes delivered over 30 s and still differ sharply in perceived smoothness if one of them delivered its bytes in bursts separated by stalls. We again sweep the additional RTT from 0 to 1000 ms for each of the three strategies, and report the ratio $\text{progress}_{c2}/\text{progress}_{c1}$ – oriented the same way as the throughput-share metric in Section 5.2 rather than the inverted FCT ratio in Section 5.3, since more playback progress is unambiguously better here, just like more throughput. As in Section 5.3, we only ran this workload with a single contender on `client-1`; only the RTT-unfairness section (Section 5.2) additionally used 2 and 5 contenders. Figure 5.30 shows both clients' players at the end of one such run before we turn to the full sweep below.



(a) `client-1`: stopped at 0:13 / 2:06.

(b) `client-2`: stopped at 0:20 / 2:06.

Figure 5.30: Both clients' YouTube players at the moment the fixed 30 s playback window ends, for a single representative repeat of the default strategy at 100 ms target additional delay (20 Mbit/s, 80-packet buffer, 1.5 ms base RTT, single contender). `client-2` (running `delay_acks`) advances noticeably further into the video than `client-1` in the same fixed real-time window.

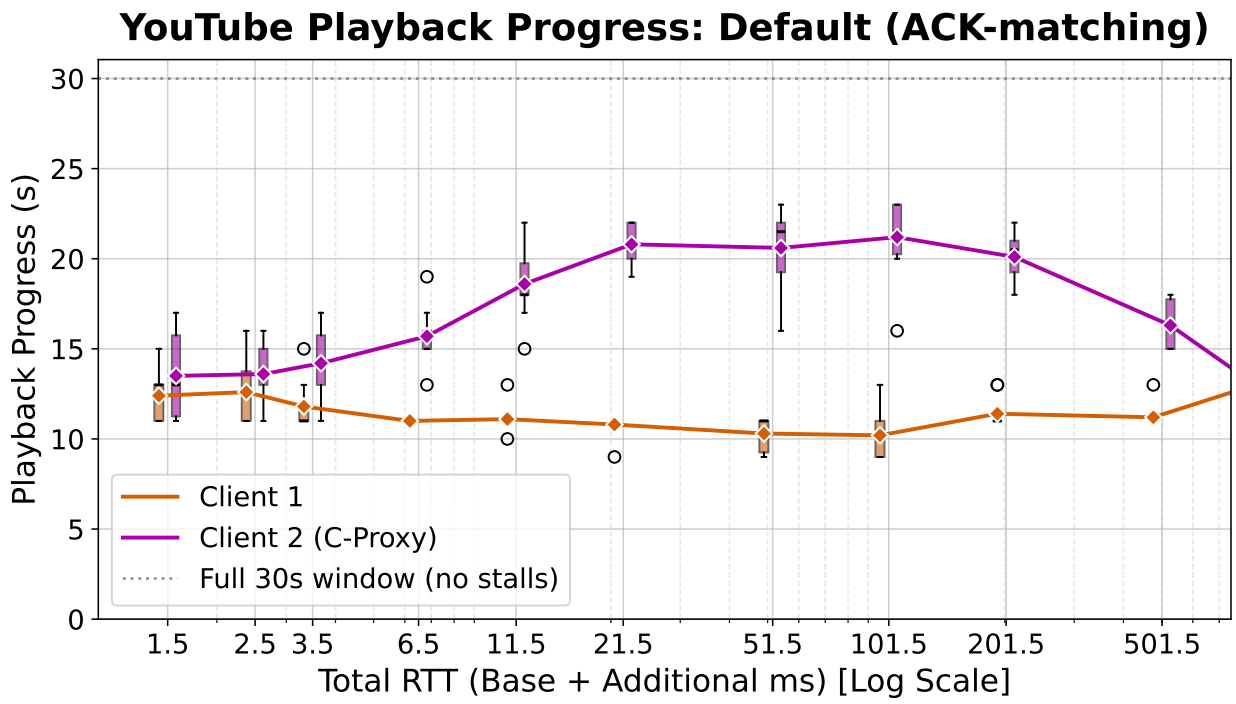


Figure 5.31: YouTube playback progress of `client-1` vs. `client-2` over the real Internet, `client-2` running the **default (ACK-matching)** strategy, as a function of its target additional RTT (20 Mbit/s, 80-packet buffer, 1.5 ms base RTT, 32× BDP). The x-axis is cropped at 500 ms so the 1000 ms point does not compress the rest of the sweep; see Table 5.29 for that value.

Table 5.29: Numeric summary corresponding to Figure 5.31 (**default** strategy): mean playback progress of `client-1` and `client-2` and their ratio $^{\text{progress}_{c2}}/_{\text{progress}_{c1}}$ for each target additional RTT.

Add. Delay (ms)	`client-1` Progress (s)	`client-2` Progress (s)	Ratio
0	12.40	13.50	1.089
1	12.60	13.60	1.079
2	11.80	14.20	1.203
5	11.00	15.70	1.427
10	11.10	18.60	1.676
20	10.80	20.80	1.926
50	10.30	20.60	2.000
100	10.20	21.20	2.078
200	11.40	20.10	1.763
500	11.20	16.30	1.455
1000	13.30	11.70	0.880

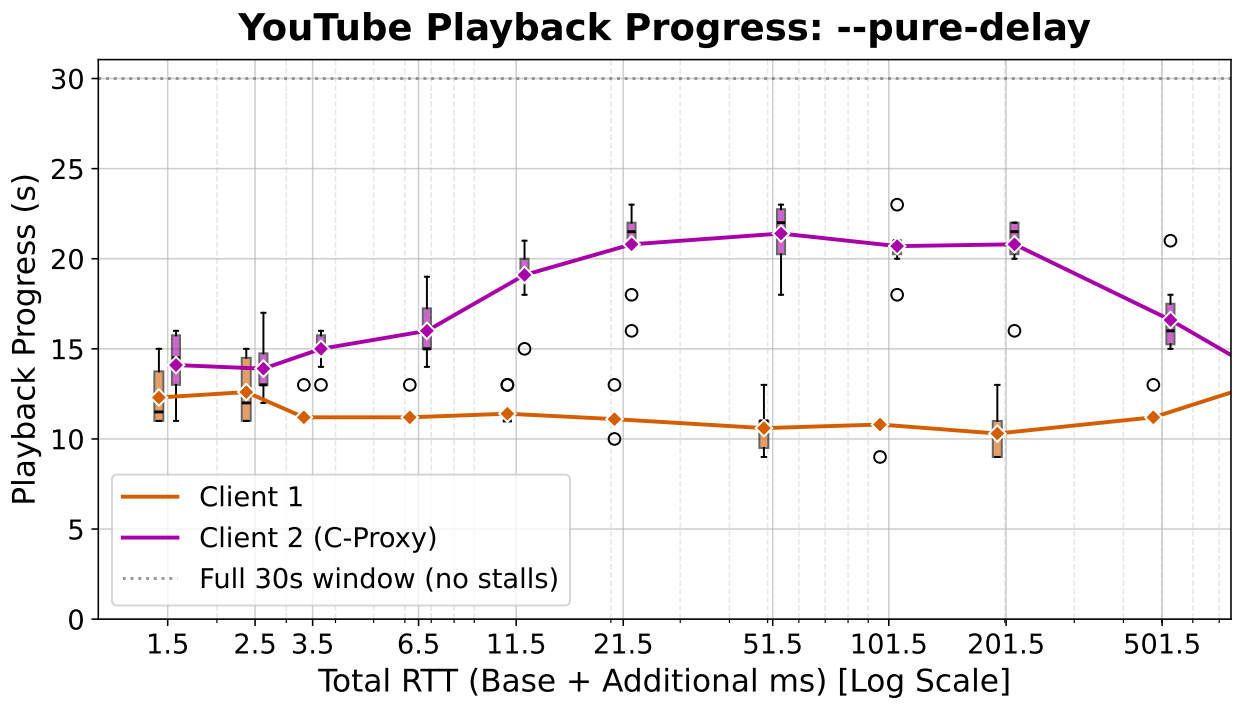


Figure 5.32: YouTube playback progress of `client-1` vs. `client-2` over the real Internet, `client-2` running the `--pure-delay` strategy, as a function of its target additional RTT (20 Mbit/s, 80-packet buffer, 1.5 ms base RTT, 32× BDP). The x-axis is cropped at 500 ms so the 1000 ms point does not compress the rest of the sweep; see Table 5.30 for that value.

Table 5.30: Numeric summary corresponding to Figure 5.32 (`--pure-delay` strategy): mean playback progress of `client-1` and `client-2` and their ratio $^{\text{progress}_{c2}}/_{\text{progress}_{c1}}$ for each target additional RTT.

Add. Delay (ms)	`client-1` Progress (s)	`client-2` Progress (s)	Ratio
0	12.30	14.10	1.146
1	12.60	13.90	1.103
2	11.20	15.00	1.339
5	11.20	16.00	1.429
10	11.40	19.10	1.675
20	11.10	20.80	1.874
50	10.60	21.40	2.019
100	10.80	20.70	1.917
200	10.30	20.80	2.019
500	11.20	16.60	1.482
1000	13.30	12.80	0.962

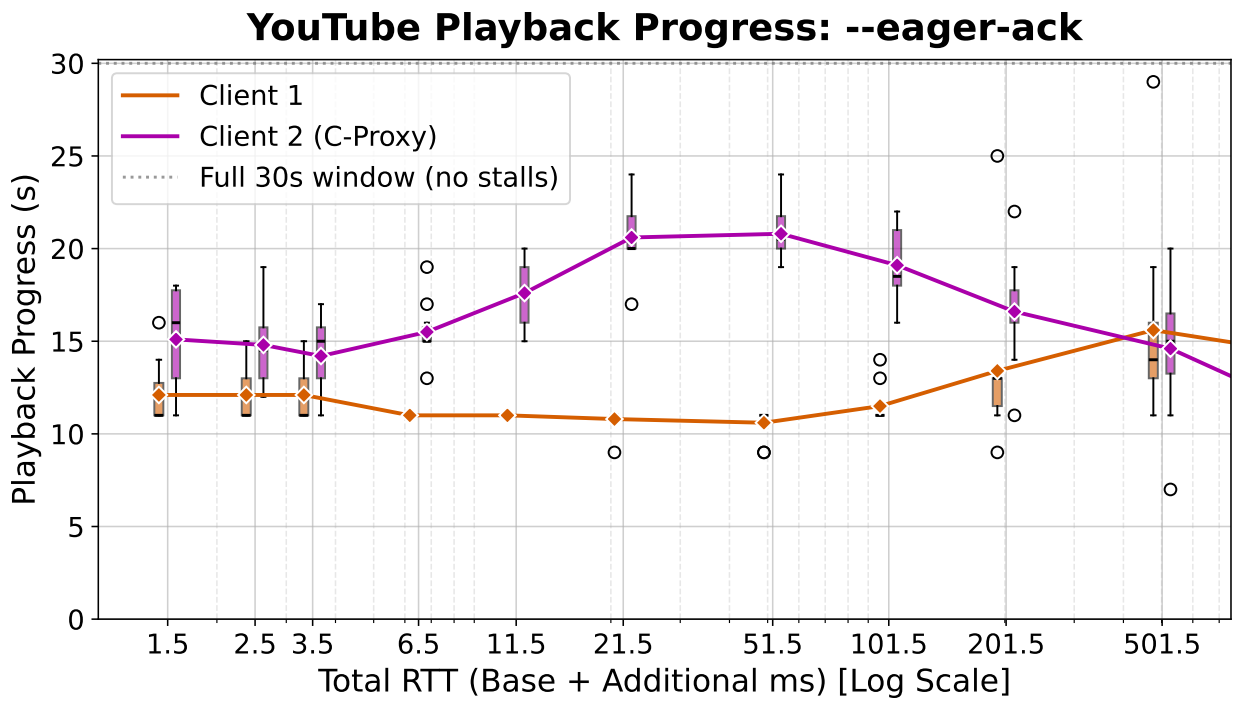


Figure 5.33: YouTube playback progress of `client-1` vs. `client-2` over the real Internet, `client-2` running the `--eager-ack` strategy, as a function of its target additional RTT (20 Mbit/s, 80-packet buffer, 1.5 ms base RTT, 32× BDP). The x-axis is cropped at 500 ms so the 1000 ms point does not compress the rest of the sweep; see Table 5.31 for that value.

Table 5.31: Numeric summary corresponding to Figure 5.33 (`--eager-ack` strategy): mean playback progress of `client-1` and `client-2` and their ratio $^{\text{progress}_{c2}}/_{\text{progress}_{c1}}$ for each target additional RTT.

Add. Delay (ms)	`client-1` Progress (s)	`client-2` Progress (s)	Ratio
0	12.10	15.10	1.248
1	12.10	14.80	1.223
2	12.10	14.20	1.174
5	11.00	15.50	1.409
10	11.00	17.60	1.600
20	10.80	20.60	1.907
50	10.60	20.80	1.962
100	11.50	19.10	1.661
200	13.40	16.60	1.239
500	15.60	14.60	0.936
1000	14.60	11.70	0.801

All three strategies again reproduce the shape we would expect from Sections 5.2 and 5.3: `client-2` pulls further ahead of `client-1` as target additional RTT grows, up to a peak, before the advantage erodes and eventually reverses at the largest delays. The peak itself is both flatter and closer between strategies than in the file-download case: the default strategy peaks at 2.078× at 100 ms (Table 5.29), `--pure-delay` reaches an essentially indistinguishable 2.019× at both 50 ms

and 200 ms (Table 5.30), and even `--eager-ack` still reaches 1.962× at 50 ms (Table 5.31) before its characteristic early regression sets in. This reinforces the finding from Section 5.3.1: the additional ACK-matching sophistication in the default strategy buys essentially nothing over simply delaying every outgoing packet by a fixed amount – if anything, the two strategies are even harder to tell apart here than they were for file downloads.

`--eager-ack` shows the same qualitative regression documented in Section 5.3, though blunted by the short, fixed 30 s test window: its ratio already turns down past 50 ms (Table 5.31), crossing below the break-even ratio of 1 already at 500 ms (0.936×) and settling at 0.801× by 1000 ms, where the file-download case had already collapsed to 0.158× at the very same target delay (Table 5.23). This is consistent with our explanation in Section 5.3: pairing outgoing payload with an ACK that has not yet reached its hold-back deadline releases that ACK early, undercutting the RTT sample the server's BBR sender bases its BDP estimate on (Section 3.1.3).

As in Section 5.3, `client-1`'s own playback progress rises again exactly where the ratio drops back toward or below 1: under the default strategy it climbs from a ≈10–12 s baseline to 13.30 s at 1000 ms, under `--pure-delay` to the same 13.30 s, and under `--eager-ack` – where the reversal happens earliest – it climbs sooner still, reaching 15.60 s already at 500 ms. The same explanation applies as before: once `client-2`'s inflated RTT estimate stops paying off, its own throughput collapses and hands the freed bandwidth back to `client-1`, which manifests here as more seconds of stall-free playback rather than a shorter completion time.

One caveat specific to this workload: the on-screen playback timer we read back only displays whole seconds, so individual samples are inherently coarser than the sub-second-resolution FCT measurements in Section 5.3; with only 10 repeats per configuration this limits how finely we can resolve the exact location of each strategy's peak, though it does not affect the overall shape of the curves or the conclusions drawn from them.

5.4.1 Packet and ACK Release Delay

As in Section 5.3.1, we look directly at how long `delay_acks` held each packet and ACK, using the same 10 ms target-delay configuration as the representative point and pooling all 10 repeats of each strategy into fixed 1 ms-wide histograms.

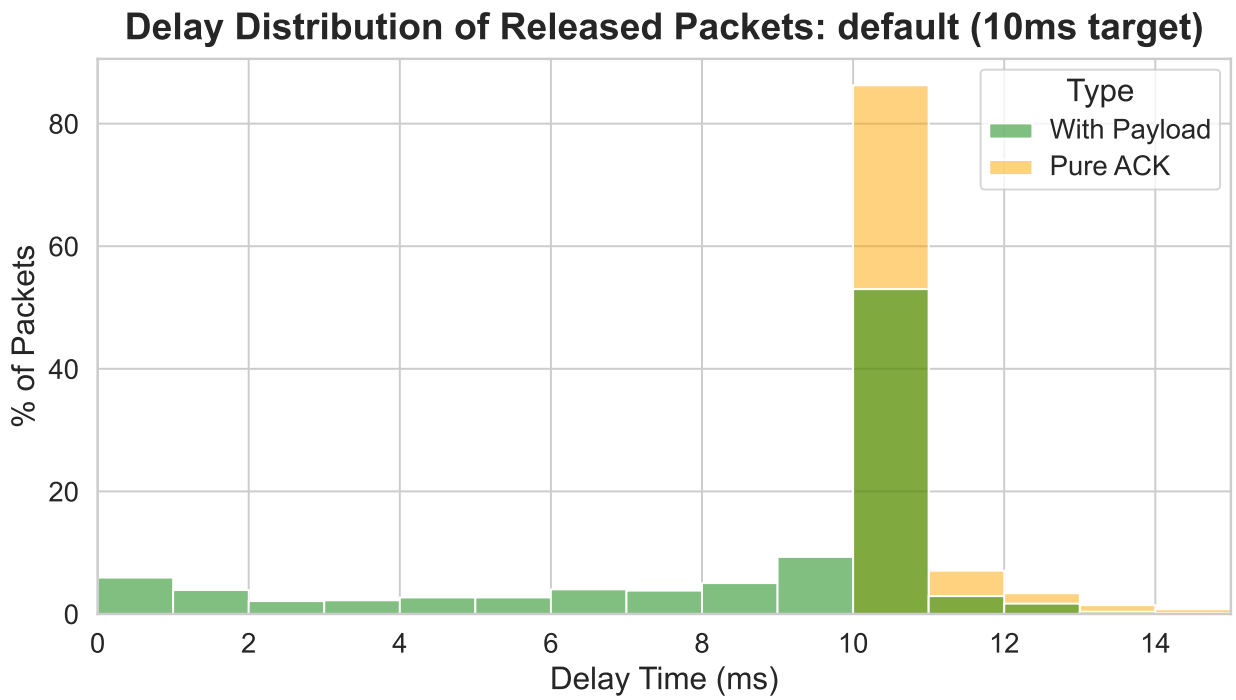


Figure 5.34: Delay distribution of released packets, **default** strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.32: Numeric summary corresponding to Figure 5.34 (**default** strategy, 10 ms target delay, pooled across 10 repeats).

	With Payload	Pure ACK
Released packets	4564	170620
Average delay (ms)	8.462	10.616

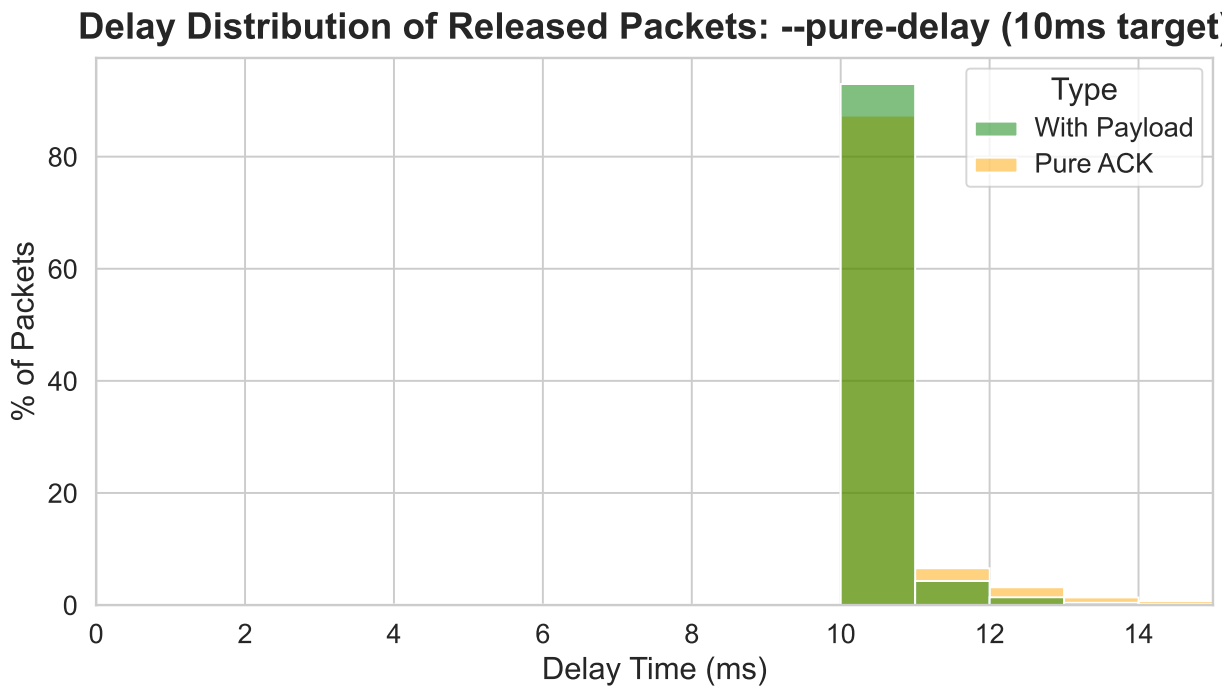


Figure 5.35: Delay distribution of released packets, `--pure-delay` strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.33: Numeric summary corresponding to Figure 5.35 (`--pure-delay` strategy, 10 ms target delay, pooled across 10 repeats).

	With Payload	Pure ACK
Released packets	4616	181044
Average delay (ms)	10.401	10.580

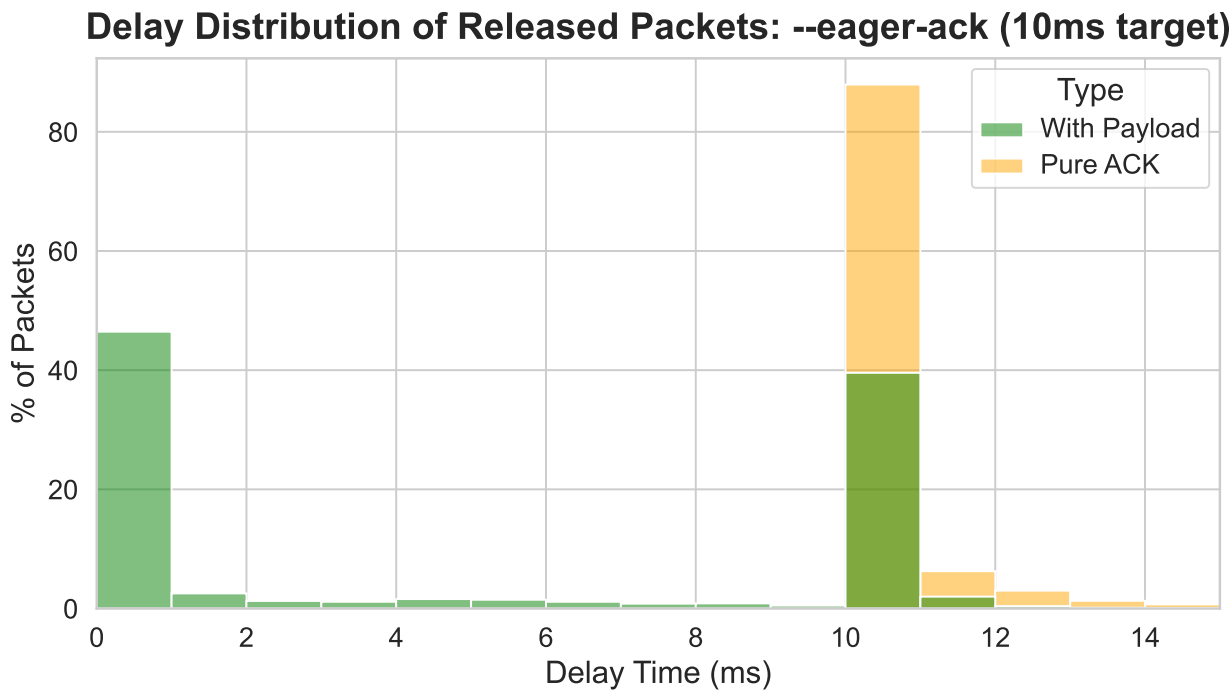


Figure 5.36: Delay distribution of released packets, `--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.34: Numeric summary corresponding to Figure 5.36 (`--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats).

	With Payload	Pure ACK
Released packets	4423	165110
Average delay (ms)	5.065	10.568

The same pattern from Section 5.3.1 shows up again, though compressed: under `--pure-delay` every packet is queued and released exactly at the target delay by construction, and the payload average of 10.401 ms in Table 5.33 reflects that directly. Under the default strategy, payload packets are again released earlier on average (8.462 ms, Table 5.32) thanks to the opportunistic ACK-stamping described in Section 4.2.1, and `--eager-ack` pushes this earlier still, to 5.065 ms (Table 5.34). The gap between the default strategy and `--pure-delay`, however, is noticeably narrower here ($\approx$ 1.9 ms) than it was for file downloads ($\approx$ 3.8 ms, Tables 5.24 and 5.25) – consistent with the playback-progress sweep already showing the two strategies as harder to tell apart in this workload. Pure-ACK packets again sit at essentially the same $\approx$ 10.6 ms average across all three strategies, since none of them changes how a lone ACK gets released.

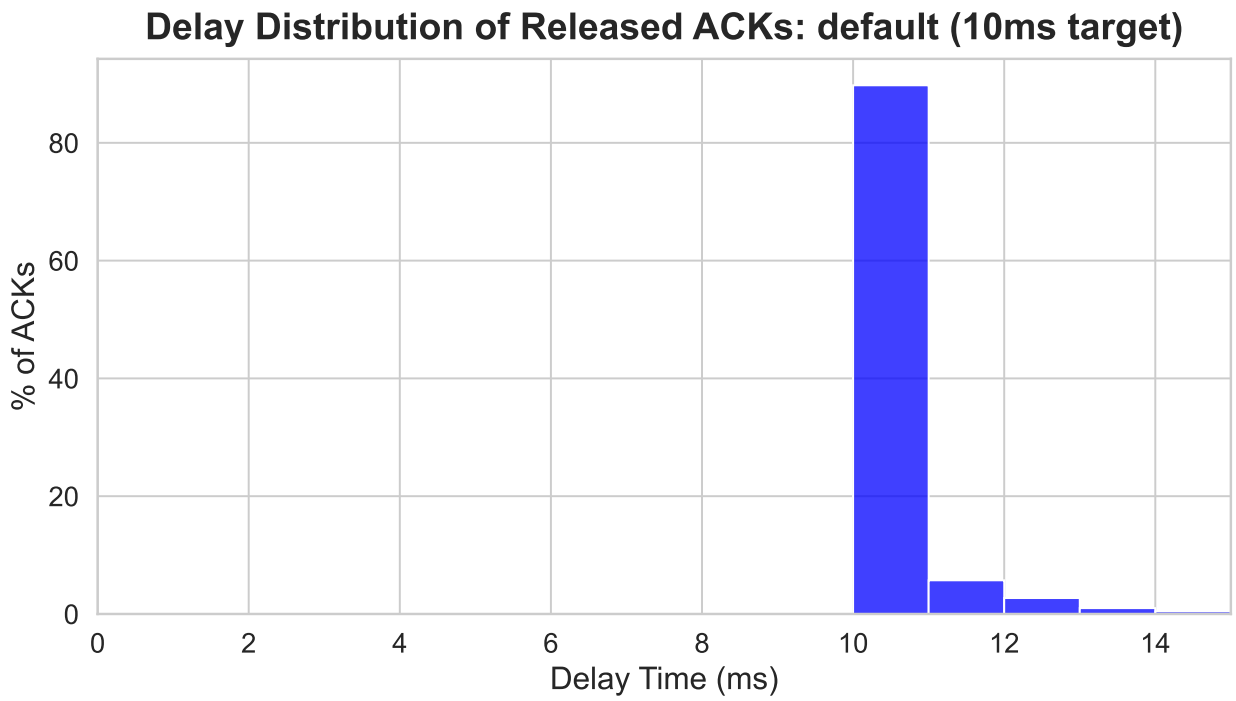


Figure 5.37: Delay distribution of released ACKs, **default** strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.35: Numeric summary corresponding to Figure 5.37 (**default** strategy, 10 ms target delay, pooled across 10 repeats).

	Value
Released ACKs	139620
Dropped ACKs	31830
Average delay (ms)	10.515
Minimum delay (ms)	10.000

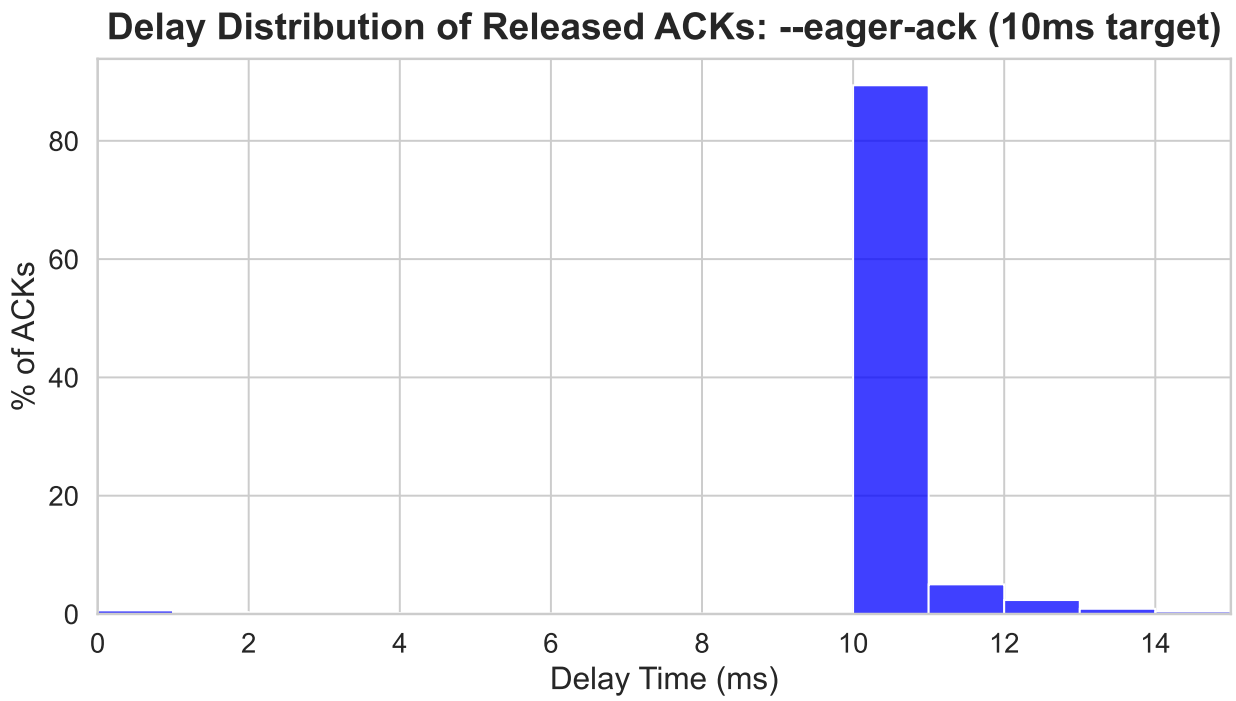


Figure 5.38: Delay distribution of released ACKs, `--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats.

Table 5.36: Numeric summary corresponding to Figure 5.38 (`--eager-ack` strategy, 10 ms target delay, pooled across 10 repeats).

	Value
Released ACKs	140938
Dropped ACKs	24057
Released before 10 ms target	2414
Average delay (ms)	10.351
Minimum delay (ms)	−0.022

As before, there is no ACK-delay plot for `--pure-delay`, since that strategy never inspects or re-stamps ACK numbers at all. Comparing the remaining two (Tables 5.35 and 5.36) reproduces the same qualitative story as Section 5.3.1, but at a noticeably higher drop rate: 18.57% of candidate ACKs are coalesced away under the default strategy and 14.58% under `--eager-ack`, both roughly double the file-download figures of 8.49% and 5.44% (Tables 5.27 and 5.28). We suspect this reflects the bursty, chunk-fetch-then-pause traffic pattern typical of adaptive video streaming – a workload that repeatedly stops and restarts would naturally produce more closely-spaced, mutually-superseding ACKs for `delay_acks` to coalesce than a file download's steady stream does, though we have not confirmed this against request-level timing. The default strategy's minimum released-ACK delay is again exactly the 10 ms target with zero early releases, while `--eager-ack` again releases a small fraction early – 2414 out of 140938, or 1.713% (versus 0.397% for file downloads) – consistent with the same eager-pairing mechanism, just triggered somewhat more often by this workload's burstier packet arrivals. The minimum recorded delay under `--eager-ack` here is slightly negative (−0.022 ms, versus a smallest value of 0.001 ms for file downloads); given its tiny magnitude and

the fact that releasing ACKs before they arrive is not something `delay_acks` was designed to do, we attribute this to the same clock-read or floating-point error at the release boundary discussed for the handful of negative-delay payload packets in Section 5.3.1.

5.4.2 Per-Flow Dynamics

To see *how* `client-2` converts its RTT-inflation advantage into extra playback progress, we zoom into a single representative repeat of the default strategy at 100 ms target additional delay – the peak-ratio configuration from Table 5.29 – and look directly at the two clients' flows at packet level, using the same capture-and-analysis pipeline described in Section 5.1.3. Both clients use a single continuous TCP flow for the whole run in this repeat.

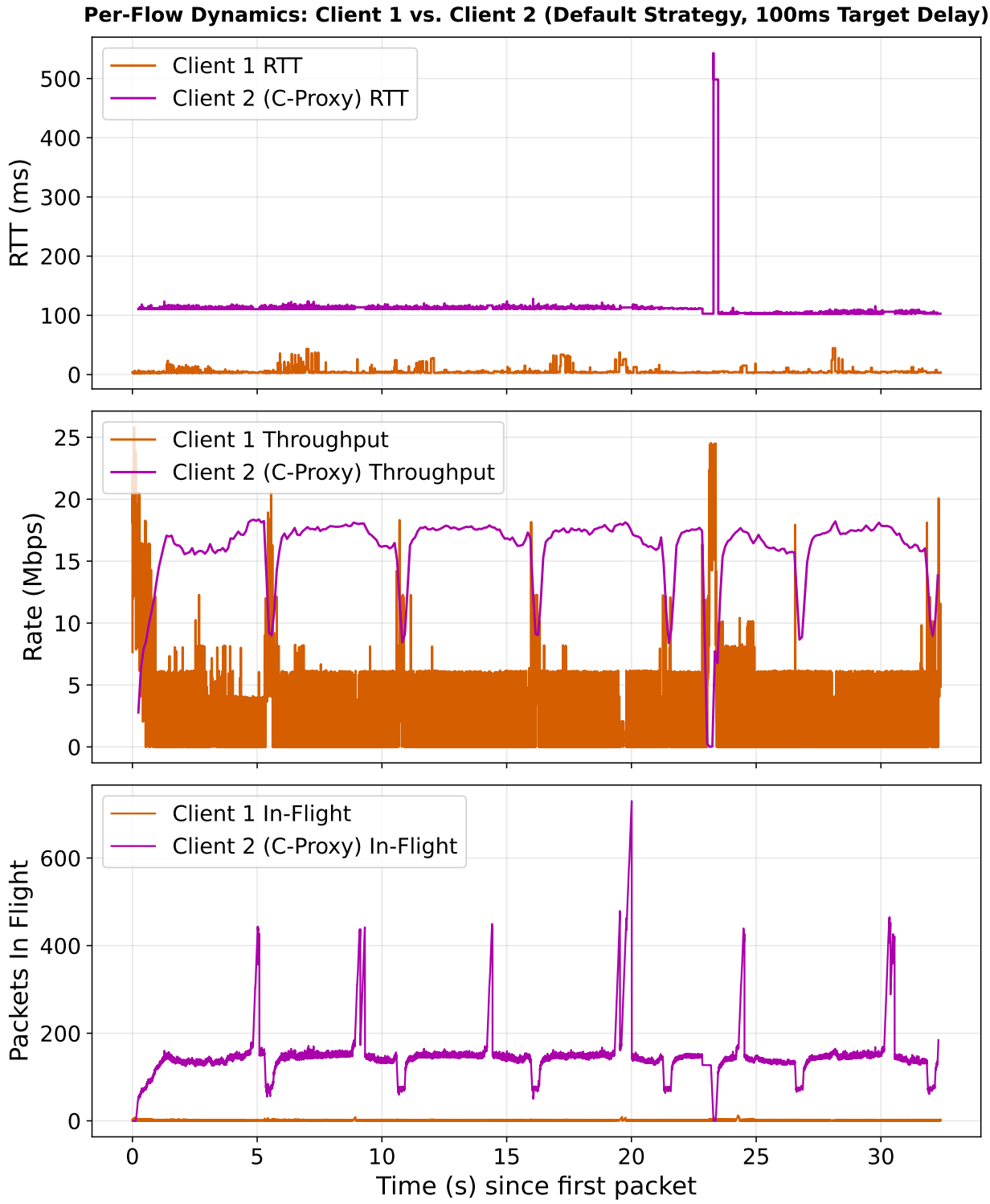


Figure 5.39: Round-trip time, throughput, and packets in flight of `client-1`'s and `client-2`'s video flows over time, for a single representative repeat of the default strategy at 100 ms target additional delay.

As can be seen in Figure 5.39, `client-2` sustains a ≈17 Mbit/s median throughput (peak

≈18.4 Mbit/s), interrupted by six throughput dips at 5.3, 10.6, 16.0, 21.3, 26.5, and 31.8 s – ≈5.2–5.4 s apart – plus one much larger event at 22.9 s. During the six regular dips, in-flight falls from a ≈145-packet baseline to 50–66 packets (≈35–45% of baseline); RTT stays flat at ≈112 ms for the first three dips and at ≈104 ms for the last three. This matches BBRv2/v3's `PROBE_RTT` behaviour (Section 3.2.1): ≈5 s interval, congestion window cut to roughly half the estimated BDP rather than BBRv1's near-empty-pipe floor. `client-1` holds a low, bursty ≈0–8 Mbit/s baseline for the whole run, spiking to ≈19–25 Mbit/s exactly during `client-2`'s dips.

The 112 ms→104 ms RTT floor shift first appears at 22.6 s, shortly before the large event, and persists for the rest of the run. Packet-level inspection shows `client-2` released no ACK between 22.822 s and 23.265 s (443 ms); all traffic on the flow stops for 339 ms within that window (22.825–23.164 s, no packets in either direction, no RST) – the only silence gap over 100 ms in the entire capture, unlike any of the six regular dips. The server retransmits the exact segment (same sequence number, same 842 bytes) it had already sent 353 ms earlier at 23.164 s, consistent with its retransmission timer firing after not seeing an ACK. The 460–540 ms RTT samples at 23.265 s are not the retransmitted segment itself (`match_acks`, Section 5.1.3, excludes retransmits by design); they are 83 packets the server sent between 22.72–22.81 s, all acknowledged at once by the single ACK that ends the 443 ms gap, each computed against its own, much earlier send time.

We believe this reflects an issue in `delay_acks`' own ACK-release logic rather than genuine network loss, as `client-1` had no issues. A thorough investigation of the tool's behaviour under conditions like this one is left to future work.

5.5 No Contender & Unconstrained

The file-download (Section 5.3) and video-playback results (Section 5.4) so far both measure `client-2`'s advantage relative to a competing `client-1` sharing the same bottleneck. This section asks what happens without that rival: a single client, alone on an unconstrained real-Internet path, downloads the same file as in Section 5.3 while `delay_acks` runs against it. We sweep the same target additional RTT from 0 to 1000 ms across the same three release strategies (Section 4.2.1), 10 repeats each. With only one client, there is no second flow to compare against and thus no ratio to report, hence our only metric here is the client's own FCT.

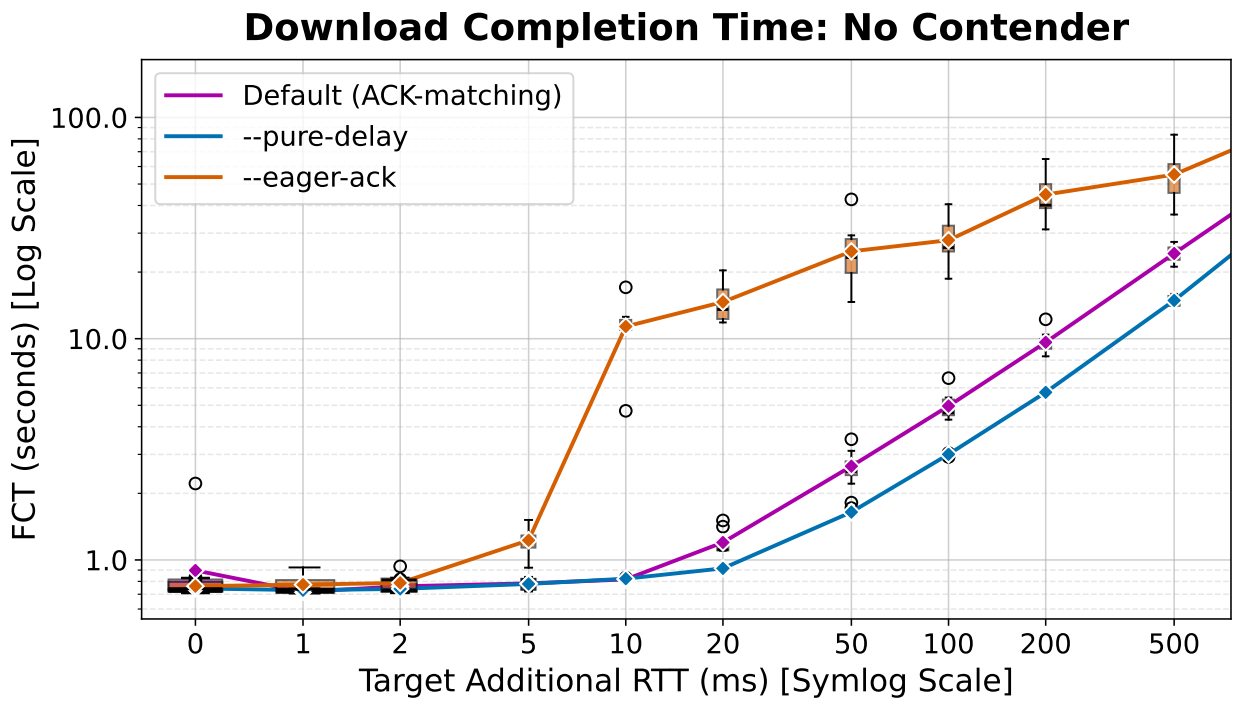


Figure 5.40: File download completion time of a single, uncontended client over the real Internet, as a function of its target additional RTT, for all three release strategies. y-axis is log-scaled – `--eager-ack`'s values span nearly two orders of magnitude more than the other two strategies, which a linear axis would flatten into illegibility at the low end.

Table 5.37: Numeric summary corresponding to Figure 5.40: mean FCT (s) of the single uncontended client, per strategy, for each target additional RTT.

Add. Delay (ms)	Default	`--pure-delay`	`--eager-ack`
0	0.90	0.74	0.76
1	0.72	0.73	0.77
2	0.76	0.74	0.79
5	0.78	0.78	1.23
10	0.82	0.82	11.35
20	1.20	0.92	14.65
50	2.65	1.65	24.82
100	4.96	3.00	27.88
200	9.63	5.72	44.80
500	24.27	14.89	55.19
1000	48.48	33.36	84.73

For all three strategies (Section 4.2.1), FCT never significantly drops below its own 0 ms baseline anywhere in the sweep. It only ever grows as target additional RTT increases, with the exception of the 0 ms point in our default strategy, which we attribute to randomly increased download time due to varying network conditions during our experiment. So even though we are most certainly competing against other flows in the Internet on the way to Google's servers, we aren't getting more bandwidth as we increase our own RTT towards Google's server. On the other hand this experiment

gives us the cleanest possible confirmation that the exploit's gains in Sections 5.3 and 5.4 come specifically at a competing flow's expense in a highly contested bottleneck rather than BBR simply granting the inflated client more bandwidth out of thin air.

The three strategies (Section 4.2.1) still separate from each other exactly as in Section 5.3, just without any upside to offset the cost. `--pure-delay` and the default strategy track each other closely at small target delays (within noise through 10 ms), then `--pure-delay` pulls clearly ahead from 20 ms onward – e.g. 14.89 s vs. 24.27 s at 500 ms – the same pattern already seen in Tables 5.21 and 5.29, where the default strategy's extra ACK-matching bookkeeping bought nothing over uniformly delaying every packet. `--eager-ack` again fares far worse than either, but here the collapse is immediate and severe rather than gradual: mean FCT jumps from 1.23 s at 5 ms to 11.35 s at 10 ms, roughly 14× the other two strategies at that same point, and it never recovers, reaching 84.73 s at 1000 ms (vs. 48.48 s and 33.36 s for the default and `--pure-delay` strategies, respectively) as seen in Table 5.37. The mechanism is the same early-ACK-release problem discussed in Section 5.3: undercutting the server's RTT sample tells the server that our `RTprop` is smaller and as explained in Section 3.1.3 will cause the server to reduce the pacing of the outgoing data.

Chapter 6

Discussion & Conclusion & Future Work

6.1 Discussion

Throughout this thesis we have largely followed the framing inherited from Chapter 2: BBR has an RTT-unfairness bias, and `delay_acks` (Chapter 4) exploits it. That framing undersells what `delay_acks` actually does: it does not correct a bias but instead mounts an attack. One receiver uses it to unilaterally take bandwidth from a specific victim flow it shares a bottleneck with, without that victim's consent, without its knowledge, and without giving anything back to the network. Section 5.5 makes this concrete: remove the victim flow, and the whole mechanism has no upside left at all. Inflating the RTT sample never buys back so much as one millisecond of file download completion time in the experiments run, but instead only ever costs the attacker latency and reduces its received throughput.

Whether this attack can be fixed or not follows directly from where the vulnerability actually comes from. Section 3.1.3 already establishes that BBR's RTT estimate is nothing more than the smallest RTT sample the sender has itself measured from its own packets' departure and acknowledgement timestamps, with no mechanism in the protocol distinguishing a genuinely delayed ACK from a deliberately withheld one. Ma et al. [10] make the same point about the sender's blind spot towards a manipulated RTT sample (Section 2.3) already shortly after BBR was proposed. The two mitigations Chapter 2 already surveys both work around this blind spot rather than closing it. Ma et al.'s own BBQ variant [10] bounds how much excess a longer-RTT flow can inject per probing cycle, but gives the sender no way to actually tell a genuine long-RTT flow from a manipulated one, only a smaller window in which either can do damage. Gomez et al.'s P4-programmable switch [6] sidesteps the blind spot entirely by measuring RTT from a network vantage point the receiver cannot influence, at the cost of requiring custom, non-standard hardware deployed in the path. Neither closes the sender-side verification gap without either accepting a softer form of the same bias or moving the fix out of the sender's hands altogether.

That leaves a choice between two differently sized problems. Cardwell et al.'s own conclusion to the original BBR paper notes that BBR "runs purely on the sender and does not require changes to the protocol, receiver, or network, making it incrementally deployable" [2] – precisely the property that let Google roll out BBRv2 and then BBRv3 (Section 3.2) without coordinating anything beyond its own sender-side codebase. A sender-side fix such as BBQ inherits this same incremental deployability; a network-side fix such as Gomez et al.'s [6] requires convincing operators along the path an attacker might choose to deploy new, non-standard hardware, across an Internet whose

infrastructure ownership is about as fragmented as it gets. Put simply: it is easier to reinvent BBR than it is to reinvent the Internet, and any realistic path to closing this gap almost certainly runs through the sender's own algorithm rather than through the network underneath it.

It is worth emphasising that this thesis actually demonstrates that the attack works against BBR as currently deployed, rather than only against a fully controlled testbed. Section 5.3, Section 5.4, and Section 5.5 all ran over the Real-Internet topology (Section 5.1.1) against Google's own production servers, `dl.google.com` and YouTube. The periodic throughput dips Section 5.4.2 observes are evidence, per Sections 3.2 and 3.2.1, that the sender on the other end was specifically running BBRv3. The parametric sweep in Section 5.2 which maps out how the effect depends on base RTT, buffer size, bottleneck bandwidth, and window scaling, by contrast, is purely emulated and exercises only the mainline kernel's BBRv1 (Section 5.1.1). The claim that this attack works against infrastructure Google actually runs in production rests on the file download and video playback results, not on the parametric sweep's specific numbers.

6.2 Conclusion

This thesis set out to ask whether deliberately inflating the RTT sample a BBR sender observes translates into a real advantage. We first evaluated how big of an advantage we can expect under varying circumstances in a controlled parametric sweep (Section 5.2), before building `delay_acks` (Chapter 4) to test the advantage of making real senders deployed on the Internet perceive an inflated RTT while running realistic workloads (Sections 5.3 to 5.5).

The clearest empirical finding to take out of the evaluation is that delaying ACKs is a double-edged sword: it only pays off if there is a contending flow with bandwidth to steal from in the first place. Against a contender, all three release strategies developed (Section 4.2.1) buy the client that is running `delay_acks` a real competitive advantage over a contender in file-download time (Section 5.3) and YouTube playback progress over a fixed and synchronised time window (Section 5.4), but Section 5.5 shows that this advantage disappears entirely the moment the rival flow is removed: FCT never significantly drops below its own 0 ms baseline anywhere in the sweep performed no matter which of the three release strategies was used. Inflating the RTT sample only ever costs the attacker latency when there is no one else's bandwidth to take. As Section 5.5 itself puts it, this is the cleanest possible confirmation that the exploit's gains come specifically at a competing flow's expense, in a contested bottleneck, rather than from BBR simply granting the inflated client more bandwidth out of thin air – direct empirical support for the Discussion's framing of `delay_acks` as an attack rather than a general-purpose throughput trick.

Section 5.2.1 shows that the usefulness of this attack is dependent on the base RTT towards the server. According to our synthesised experiments the advantage is largest at low base RTT and shrinks sharply as base RTT grows: a peak ratio of 10.57× at 5 ms base RTT against a single BBR contender falls to 2.07× at 50 ms RTT towards the server, with a fixed bottleneck bandwidth and buffer depth (Table 5.1). Our hypothesis for why the advantage shrinks at higher base RTT is that the contending flow computes its own in-flight budget from its own RTT too (Section 3.1.3), and BDP = bandwidth × RTT grows with RTT on its own, without any manipulation. With this, at a high base RTT, the contender is already claiming a larger share of the bottleneck buffer for itself before the attacker inflates anything, leaving less room to take. At a low base RTT, the contender's own baseline budget is small, so the same amount of injected delay makes a much bigger relative difference. We have not measured this directly – it is a plausible explanation for the shape Section 5.2.1 shows, not a confirmed mechanism.

Taken together, `delay_acks` demonstrates a real, reproducible weakness in how BBR trusts its

own RTT samples. It is easy to trigger from an unmodified receiver, and it reaches production BBR deployments on the real Internet. But its payoff depends entirely on a contending flow to take bandwidth from, and that payoff shrinks as base RTT grows.

6.3 Future Work

After writing this thesis many questions stay open and there is too little time to test our theories with empirical measurements. That's why we propose the following points as potential future work that can be done in this field.

6.3.1 Why Do the Curves Have This Shape? A Proposed Delay Decomposition

Every sweep in Section 5.2 traces the same qualitative shape: the throughput-share ratio rises as additional delay grows, peaks, and then declines. Sections 5.2.1 to 5.2.4 explain this shape informally, in terms of a buffer that eventually can no longer sustain the inflated in-flight window BBR's model calls for. We propose a more general, unifying account of why this shape appears at all. Decompose the total round-trip time D a BBR sender actually measures into three additive terms,

$$D = \texttt{RTT}_{min} + \text{QueueingDelay}_{\text{natural}} + \delta,$$

where $\texttt{RTT}_{\text{min}}$ is the path's true propagation delay, $\text{QueueingDelay}_{\text{natural}}$ is whatever standing queue already exists at the bottleneck independent of the attack, and δ is the additional delay `delay_acks` deliberately injects. Figure 6.1 sketches three regions this decomposition suggests. In **Region A**, $\text{QueueingDelay}_{\text{natural}}$ stays negligible and δ dominates the growth in D; every additional millisecond of δ translates almost directly into a larger admitted in-flight budget, and the ratio climbs. In **Region B**, the larger in-flight budget the client inflating its RTT claims begins to fill the bottleneck buffer itself, so $\text{QueueingDelay}_{\text{natural}}$ starts growing alongside δ rather than staying negligible. The two terms compete, and the ratio's rate of growth slows until it peaks. In **Region C**, the buffer saturates and $\text{QueueingDelay}_{\text{natural}}$ takes over as the dominant term in D. Further increases in δ mostly add pure latency rather than additional admitted in-flight budget, and the ratio declines. This is consistent with the buffer-relative-to-BDP exhaustion argument already made informally throughout Sections 5.2 and 5.3.2.

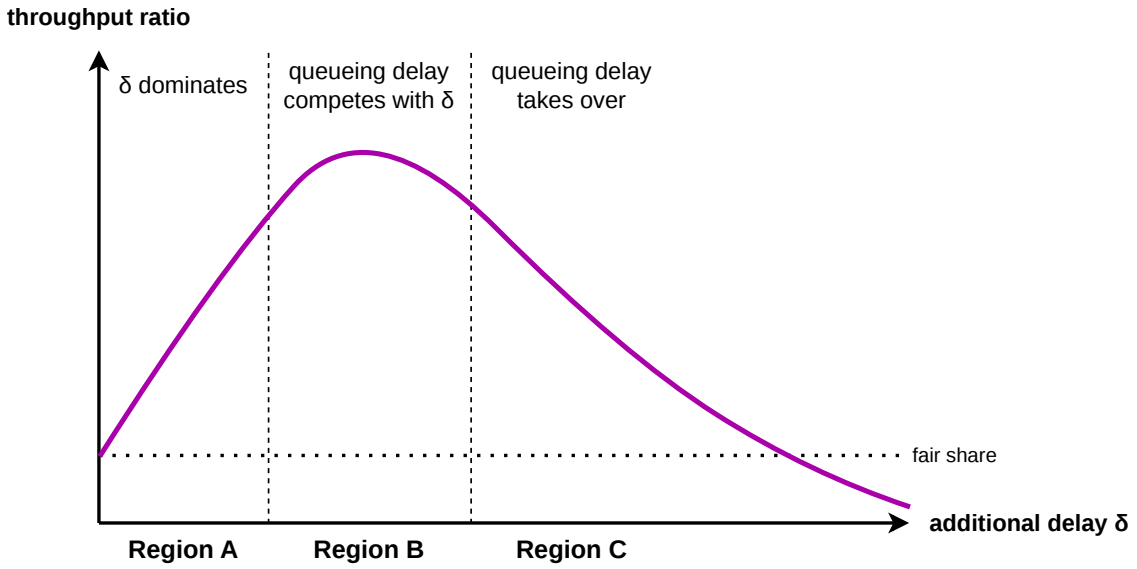


Figure 6.1: Conceptual sketch (not fitted to data) of the proposed delay decomposition $D = \texttt{RTT}_{min} + \text{QueueingDelay}_{\text{natural}} + \delta$. In Region A, δ dominates D and the ratio rises with it; in Region B, natural queueing delay begins competing with δ's contribution, and the two roughly balance around the peak; in Region C, natural queueing delay from a saturated buffer takes over and the ratio declines despite δ still growing. This model has not been directly measured against data – see the discussion below.

We are explicit that this model is proposed, not validated. The pipeline used throughout this thesis (explained in Section 5.1.3) reconstructs total RTT, throughput, and in-flight packet counts from packet timestamps; it never separately measures $\text{QueueingDelay}_{\text{natural}}$ apart from $\texttt{RTT}_{\text{min}}$ and δ, so the split above cannot be recovered from data already collected. Confirming it directly would need new instrumentation at the bottleneck itself, for example, sampled `qdisc` queue-occupancy statistics, or timestamps taken at both ends of the bottleneck rather than only at the clients and server. If it holds, the payoff would be to actually be able to predict where the region boundaries and the peak sit as a function of the parameters of the bottleneck, instead of only being able to explain them individually and after the fact, as we do now.

6.3.2 Sensitivity to the `PROBE_RTT` Variant

A related open question is whether the peak's location is sensitive to which `PROBE_RTT` variant the server runs and for how long the RTT is probed by the underlying BBR version. Section 3.2.1 establishes that BBRv1 – the only version our Emulated server topology exercises, since its shared kernel exposes only the standard mainline `tcp_bbr` module (Section 5.1.1) – forcibly cuts the congestion window to a hard, BDP-independent floor of four packets every 10 s, "for at least 200 ms and one round trip" [2], fully draining the bottleneck queue on a fixed schedule regardless of how full it is. BBRv2 and BBRv3 instead only halve the current BDP estimate every 5 s, a much shallower cut that may leave a standing queue only partially drained. Since every result in Section 5.2 only ever exercises BBRv1's aggressive full-drain behaviour, it is an open question whether repeating the same parametric sweep against a BBRv2 or BBRv3 sender would shift where the peaks sit,

or change how sharply the ratio declines afterward. A persistent, never-fully-emptied queue under the shallower cut could plausibly interact differently with the Region B/C balance proposed above than BBRv1's periodic full reset does.

6.3.3 A Secondary Benefit for the Client's Own Outgoing Traffic

Section 5.3.1 shows that the default (ACK-matching) strategy releases the client's own outgoing payload packets earlier, on average, than `--pure-delay` does – 6.562 ms vs. 10.326 ms mean payload delay at the 10 ms target-delay configuration (Tables 5.24 and 5.25), a gap of roughly 3.8 ms; `--eager-ack` goes further still, to 3.768 ms, a gap of roughly 6.6 ms against `--pure-delay` (Table 5.26). This is a secondary property of the release strategies (Section 4.2.1), distinct from simply trading a less-inflated apparent RTT for less added latency: opportunistically stamping payload with an already-expired ACK, rather than waiting out the full hold-back window every time, could plausibly help application response time for any latency-sensitive outgoing traffic from the client itself, by not delaying outgoing payload longer than necessary. Neither workload evaluated in this thesis actually tests this: both file downloads (Section 5.3) and video playback (Section 5.4) are receiver-heavy, sending only a handful of small outgoing payload packets against a large volume of pure ACKs. At the 10 ms additional RTT configuration and under the default strategy, just 1288 payload packets against 219 838 pure-ACK packets for file downloads (Table 5.24, a ratio of well over 170:1) and 4564 payload packets against 170 620 pure-ACK packets for video playback (Table 5.32, a ratio of over 37:1) were logged, which makes this potential benefit barely exercised by either workload studied here. An upload-heavy or bidirectional workload (a video call, a file upload, an interactive chat protocol) would be needed to actually quantify whether this few-millisecond advantage in payload-release delay translates into a measurable response-time or quality-of-experience benefit.

6.3.4 Robustifying `delay_acks`

Section 5.4.2 documents one further open thread worth flagging here: an anomalous multi-hundred-millisecond gap in `client-2`'s ACK stream during one otherwise-ordinary video-playback run that appears to stem from `delay_acks`' own release logic rather than genuine network loss – a gap that section already leaves to future work rather than investigating further. Robustifying that release logic under bursty, stop-start traffic – plausibly the pattern video playback produces, unlike a file download's steadier stream – is a natural next step.

A second, unrelated sign that the release logic may not yet be doing exactly what it was designed to do: under `--eager-ack`, an outgoing payload packet should almost always find some already-received ACK to pair with immediately, so mean payload-release delay should sit close to 0 ms. Instead, Section 5.3.1 measures a mean payload delay of 3.768 ms for `--eager-ack` at the 10 ms target-delay configuration (Table 5.26) – small, but clearly not zero. This is worth auditing directly in `delay_acks`' own code as future work, rather than assuming the current implementation already behaves exactly as `--eager-ack`'s design intends.

Bibliography

[1] Abrol, A., Murali Mohan, P., Truong-Huu, T., and Gurusamy, M. Bbr congestion control algorithms: Evolution, challenges and future directions. *ACM Comput. Surv. 58*, 9 (Feb. 2026).

[2] Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. Bbr: Congestion-based congestion control. *ACM Queue 14, September-October* (2016), 20 – 53.

[3] Cardwell, N., Cheng, Y., Yang, K., Morley, D., Hassas Yeganeh, S., Jha, P., Seung, Y., Jacobson, V., Swett, I., Wu, B., and Vasiliev, V. BBRv3: Algorithm bug fixes and public internet deployment. IETF 117 CCWG presentation slides, July 2023. States that, as of July 2023, BBRv3 is already the TCP congestion control for all Google.com public-internet traffic, while YouTube TCP (and QUIC for google.com/YouTube) was still under small-percentage A/B testing against BBRv1 at that time.

[4] Cardwell, N., Cheng, Y., Yang, K., Morley, D., Hassas Yeganeh, S., Jha, P., Seung, Y., Jacobson, V., Swett, I., Wu, B., and Vasiliev, V. BBRv3: Algorithm overview and google's public internet deployment. IETF 119 CCWG presentation slides, March 2024. States that BBRv3 is the congestion control algorithm for all Google.com and YouTube public-internet traffic as of March 2024.

[5] Claypool, S., Claypool, M., Chung, J., and Li, F. Sharing but not caring – performance of TCP BBR and TCP CUBIC at the network bottleneck. In *Proceedings of the 15th Advanced International Conference on Telecommunications (AICT)* (2019).

[6] Gomez, J., Kfoury, E. F., Crichigno, J., and Srivastava, G. Improving tcp fairness in non-programmable networks using p4-programmable data planes. In *2024 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)* (2024), pp. 102–107.

[7] Ha, S., Rhee, I., and Xu, L. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 64–74.

[8] Hock, M., Bless, R., and Zitterbart, M. Experimental evaluation of bbr congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (2017), pp. 1–10.

[9] Kozu, T., Akiyama, Y., and Yamaguchi, S. Improving rtt fairness on cubic tcp. In *2013 First International Symposium on Computing and Networking* (2013), pp. 162–167.

[10] Ma, S., Jiang, J., Wang, W., and Li, B. Fairness of congestion-based congestion control: Experimental evaluation and analysis, 2017.

[11] Mishra, A., Tiu, W. H., and Leong, B. Are we heading towards a bbr-dominant internet? In *Proceedings of the 22nd ACM Internet Measurement Conference* (New York, NY, USA, 2022), IMC '22, Association for Computing Machinery, p. 538–550.

[12] Scholz, D., Jaeger, B., Schwaighofer, L., Raumer, D., Geyer, F., and Carle, G. Towards a deeper understanding of tcp bbr congestion control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops* (2018), pp. 1–9.

[13] Zhang, S. An evaluation of bbr and its variants, 2019.

Appendix A

Additional Data

A.1 Buffer/BDP Ratio Sweep – Full Peak Data

Section 5.2.4 only plots the 1.0×-BDP buffer sweep in full; the 2.0× and 5.0× sweeps referenced in that discussion are tabulated here instead, in the same peak-additional-delay/peak-ratio format used throughout Section 5.2.

	1 Contender		5 Contenders	
Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio	Peak Add. Delay (ms)	Peak Ratio
10	100	2.15	250	0.44
20	150	3.00	250	0.53
50	50	6.28	150	1.60
100	50	8.71	50	2.75
200	50	12.43	50	5.21
500	10	9.76	10	4.13
1000	2.5	1.76	5	1.36

Table A.1: Peak throughput-share ratio vs. bandwidth with the buffer fixed at 2.0× the BDP at every bandwidth (12 ms base RTT), `client-1` using BBR.

	1 Contender		5 Contenders	
Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio	Peak Add. Delay (ms)	Peak Ratio
10	50	4.19	100	1.83
20	150	4.74	100	2.86
50	50	6.13	50	3.32
100	75	3.02	75	2.57
200	50	0.64	50	1.38
500	25	0.60	50	0.26
1000	10	0.71	25	0.17

Table A.2: Peak throughput-share ratio vs. bandwidth with the buffer fixed at 2.0× the BDP at every bandwidth (12 ms base RTT), `client-1` using CUBIC.

	1 Contender		5 Contenders	
Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio	Peak Add. Delay (ms)	Peak Ratio
10	75	3.71	75	0.70
20	100	5.80	150	1.08
50	100	8.10	150	2.77
100	50	9.62	100	4.66
200	50	13.86	50	5.22
500	10	9.85	15	3.77
1000	10	1.25	10	1.15

Table A.3: Peak throughput-share ratio vs. bandwidth with the buffer fixed at 5.0× the BDP at every bandwidth (12 ms base RTT), `client-1` using BBR.

	1 Contender		5 Contenders	
Bandwidth (Mbit/s)	Peak Add. Delay (ms)	Peak Ratio	Peak Add. Delay (ms)	Peak Ratio
10	100	4.32	75	2.23
20	100	4.45	150	2.88
50	150	2.94	150	2.47
100	50	0.96	75	1.37
200	50	0.65	50	0.59
500	25	0.61	25	0.19
1000	10	0.71	25	0.17

Table A.4: Peak throughput-share ratio vs. bandwidth with the buffer fixed at 5.0× the BDP at every bandwidth (12 ms base RTT), `client-1` using CUBIC.

A.2 Window Scale Sweep – Full Peak Data

Section 5.2.5 only plots the window-scale sweep at 10 ms base RTT in full; the same sweep at 5, 20, and 50 ms base RTT is tabulated here instead. Unlike the other peak tables in this thesis, only the peak throughput-share ratio is shown (dropping the peak additional delay column) since, unlike buffer size, bandwidth, or buffer/BDP ratio, the additional delay at which the peak occurs does not shift in any consistent way with window scale.

	BBR-vs-BBR		CUBIC-vs-BBR	
Window Scale	1 Contender	5 Contenders	1 Contender	5 Contenders
0.5×	10.52	2.94	2.55	1.72
1×	10.37	3.46	2.94	2.50
2×	10.33	3.67	2.95	2.49
5×	10.78	3.47	2.97	2.53

Table A.5: Peak throughput-share ratio vs. window scale at 5 ms base RTT (100 Mbit/s, 200-packet buffer).

	BBR-vs-BBR		CUBIC-vs-BBR	
Window Scale	1 Contender	5 Contenders	1 Contender	5 Contenders
0.5×	7.62	2.20	2.51	1.89
1×	8.02	2.53	2.94	2.40
2×	7.62	2.67	2.97	2.54
5×	7.66	2.33	2.84	2.53

Table A.6: Peak throughput-share ratio vs. window scale at 20 ms base RTT (100 Mbit/s, 200-packet buffer).

	BBR-vs-BBR		CUBIC-vs-BBR	
Window Scale	1 Contender	5 Contenders	1 Contender	5 Contenders
0.5×	1.56	0.31	2.35	2.27
1×	2.02	0.37	2.77	2.66
2×	1.96	0.27	2.82	2.84
5×	2.01	0.44	2.73	2.98

Table A.7: Peak throughput-share ratio vs. window scale at 50 ms base RTT (100 Mbit/s, 200-packet buffer).