**Abstract**

# A Wire-Compatible TCP Implementation for Low-Latency Applications

Michael F. Nowlan

2014

Despite alternative transport protocols more suitable to latency-sensitive applications, TCP remains the *de facto* standard for Internet traffic, including many low-latency, interactive applications. As such, applications often tweak the TCP protocol to better suit their needs but run the risk that *any* change to TCP's wire format can cause reachability issues or complete failure. This work presents a modified TCP implementation to reduce end-to-end latency without modifying TCP's wire format.

Bufferbloat Resilient TCP (BBR) is a sender-side modification that detects and mitigates instances of so-called "bufferbloat" in a TCP connection. Large network buffers coupled with TCP's additive increase often result in standing queues and drastically higher round-trip times. BBR throttles the TCP sender by identifying bufferbloat through the correlation between sending rate and delay. A live production deployment shows BBR improves average end-to-end latency, bandwidth efficiency and retransmissions by orders of magnitude for a video serving platform.

Random and congestive packet loss increase the latency of subsequent packets, not just the lost packets, due to TCP's strict in-order delivery model. Unordered TCP ($u$TCP) is a receiver-side change that exposes to applications out-of-order data segments that the OS normally delays in socket buffers. This small change enables applications to build generic unordered datagram transports and improves end-to-end latency for interactive applications. Evaluations show that $u$TCP can approximate the latency performance of UDP for various application workloads.

# A Wire-Compatible TCP Implementation for Low-Latency Applications

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Michael F. Nowlan

Dissertation Director: Bryan A. Ford

May 2014

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I would like to thank my advisor, Bryan A. Ford, for his technical guidance and excellent example of how to pursue interesting research challenges. I also thank the members of my dissertation committee: Yang Richard Yang, Michael Fischer and Janardhan Iyengar, for providing guidance and intellectual stimulation throughout the last five years. Additionally, I would like to thank the MTF team, especially Van, Neal, Yuchung, Andreas and Eric, for augmenting my networking education despite my slow start. Also, I thank Jana, Obaid and Nabin for our joint efforts on Minion. I especially appreciate the support at Yale, both in the Decentralized and Distributed Systems group and the Computer Science Department as a whole.

I would like to thank my wife, Cheryl, for her unending support during these years and challenging me to work as hard as she does. To my parents, Laurie and Pat, and the rest of the Nowlan Gang, thanks for teaching me to think critically and reach my potential. Lastly, thank you to my undergraduate advisor, Brian M. Blake, for starting me on such a rewarding path.

This research was conducted in part with Government support under and awarded by DoD, Air Force Office of Scientific Research, and National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a. Additionally, this research was sponsored in part by Google Inc. and the National Science Foundation under grants CNS-0916413 and CNS-0916678. AMDG

# Chapter 1

# Introduction

TCP [54] remains the *de facto* standard for carrying Internet traffic and is often used as a general-purpose transport substrate [4, 45] for bulk transfer and interactive applications alike. Despite many alternatives better-equipped to transport certain application workloads, including UDP [46], SCTP [53], SST [23], and QUIC [47], developers continue to tweak TCP to better suit their objectives [21, 40].

TCP was originally designed to offer applications a convenient, high-level communication abstraction with semantics emulating Unix file I/O or pipes. As the Internet has evolved, however, TCP's original role of offering an *abstraction* has gradually been supplanted with a new role of providing a *substrate* for transport-like, application-level protocols such as SSL/TLS [18], ØMQ [2], SPDY [4], and WebSockets [60]. In this new *substrate* role, TCP's in-order delivery offers little value since application libraries are equally capable of implementing convenient abstractions. TCP's strict in-order delivery, however, prevents applications from controlling the *framing* of their communications [15, 23], and incurs a "latency tax" on content whose delivery must wait for the retransmission of a single lost TCP segment.

Due to the difficulty of deploying new transports today [24, 45, 50], applica-

tions rarely utilize new out-of-order transports such as SCTP [53] and DCCP [35]. UDP [46] is a popular substrate, but is still not universally supported in the Internet, leading even delay-sensitive applications such as the Skype telephony system [7] and Microsoft's DirectAccess VPN [17], to fall back on TCP despite its drawbacks.

Given the continued use of TCP as a substrate for latency-sensitive applications, we identify two TCP design features that can broadly impact a user's overall experience in a negative way by increasing the end-to-end latency. First, TCP's design goal to fully utilize available bandwidth causes it to gradually increase a flow's transmission rate until a loss event. This behavior coupled with large internal network buffers introduces the phenomenon of "bufferbloat" [25], characterized by standing queues and increased round-trip time (RTT).

Second, the aforementioned strict in-order delivery model TCP imposes means that small losses can have a disproportionate impact on the end-to-end latency. While bufferbloat's self-inflicted nature means that it can occur without triggering traditional congestion avoidance and control mechanisms [33], strict in-order delivery incurs the highest latency cost during typical congestion or random loss.

To overcome the above challenges, we present two TCP implementation changes: one that modifies the sender to avoid bufferbloat and another that modifies the receiver to deliver received data out-of-order. Bufferbloat Resilient TCP and *Unordered* TCP are both incrementally deployable at TCP endpoints independently of the other endpoint and make no changes to TCP's wire format.

## 1.1 Sender-Side Bufferbloat Resilient TCP

Bufferbloat Resilient TCP (BBR) is a minimalist patch to the sender-side TCP stack to mitigate the occurrence of bufferbloat. BBR tracks the growth of the bottleneck

router's queue and reduces the sender's transmission rate when the observed delay crosses a configurable threshold. This solution is incrementally deployable at the TCP sender and operates as a thin layer between the TCP sending module and its congestion control. Furthermore, BBR requires no changes to network infrastructure or queuing algorithms, nor does it introduce any changes to TCP's existing wire format. BBR is general enough to be ported to other transports as well.

We observe that bufferbloat occurs when the transmission rate grossly exceeds the path's available bandwidth but is generally orthogonal to the transport protocol. Any transport that overestimates the available bandwidth can enter a bufferbloated phase where it builds up a standing queue at the bottleneck. As a result, BBR measures the correlation between a flow's sending rate and the observed RTT. Absent competition on the path, TCP will saturate the bottleneck link, begin building a queue and enter bufferbloat. During bufferbloat (and prior to a loss event), the sending rate correlates almost perfectly with the observed RTT because any increase in rate causes a proportional increase in queuing delay. Thus, BBR identifies bufferbloat during periods of near perfect correlation and throttles the sender until the correlation drops below a configurable threshold.

## 1.2 Receiver-Side Unordered TCP

At the receiving side, we deploy *Unordered TCP* (*u*TCP), a small OS extension adding basic unordered delivery primitives to TCP, and two application-level protocols implementing datagram-oriented delivery services that function on either *u*TCP or unmodified TCP stacks. Together, the receiver-side tools comprise the Minion architecture, a low-level transport substrate using TCP atop which applications can build custom abstractions and delivery semantics.

*u*TCP addresses delays caused by TCP's receive buffering by enabling the application to receive out-of-order TCP segments immediately, without delaying their delivery until retransmissions fill prior holes. Designed for simplicity and deployability, these extensions add less than 600 lines to Linux's TCP stack.

Minion's application-level protocols, *uCOBS* and *uTLS*, build general datagram delivery services atop *u*TCP or TCP. Key challenges these protocols address are: (a) TCP offers no reliable out-of-band framing to delimit datagrams in a TCP stream; (b) *u*TCP cannot add out-of-band framing without changing TCP's wire protocol; and (c) common in-band TCP framing methods assume in-order processing. To make datagrams *self-delimiting* in a TCP stream, *u*COBS leverages *Consistent Overhead Byte Stuffing* (COBS) [13] to encode application datagrams with at most 0.4% expansion, while reserving a single byte value to delimit encoded datagrams.

## 1.3 Implementation and Evaluation Summary

We implemented BBR as a patch to Linux's TCP stack in about 360 lines of C code. In micro-benchmark evaluations, BBR correctly identifies instances of bufferbloat and improves end-to-end latency by 3 orders of magnitude compared to Linux's default TCP CUBIC implementation. In live experiments on a global content delivery network, BBR reduced end-to-end latency by roughly $6x$ for both mobile and desktop users while achieving median bandwidth efficiency of 95% compared to the baseline's 87%. Furthermore, BBR reduced dropped packets and, in turn, retransmissions by 66× versus the baseline TCP; in many cases BBR dropped no packets at all.

We implemented *u*TCP as a second patch to Linux's TCP stack in about 600 lines of C code. In addition to the *u*TCP kernel patch, we wrote several application-level libraries to ease the deployment of *u*TCP into existing applications. Applications

already using datagram-like messages, such as those relying on OpenSSL [43] for encryption, are able to quickly realize $u$TCP's latency benefits with minimal changes to their network stack.

Experiments with a $u$TCP prototype on Linux show several benefits for applications using TCP. $u$TCP can reduce application-perceived jitter of Voice-over-IP (VoIP) streams atop TCP, and increase perceptible-quality metrics [42]. Virtual private networks (VPNs) that tunnel IP traffic over SSL/TLS can double the throughput of some tunneled TCP connections by employing $u$TCP to prioritize and expedite tunneled ACKs. Web transports can cut the time before a page begins to appear by up to half, achieving the latency benefits of multistreaming transports [23, 38] while preserving the TCP substrate. Use of $u$COBS can incur up to 5× CPU load with respect to raw TCP, due to COBS encoding, but for secure connections, $u$TLS incurs less than 7% CPU overhead (and no bandwidth overhead) atop the baseline cost of TLS 1.1.

The contributions of this thesis are first an analysis of two TCP design decisions that account for significant latency increases under normal circumstances. Second, we develop a sender-side, wire-compatible change to avoid bufferbloat under conditions without competition. Third, we develop a receiver-side, wire-compatible change and accompanying application-level libraries to support out-of-order delivery in applications using TCP. Lastly, we experimentally evaluate both changes to demonstrate their effectiveness in reducing latency in TCP connections.

This thesis proceeds as follows: Chapter 2 analyzes TCP's send and receive semantics and discusses related work; Chapter 3 presents the design and algorithm of Bufferbloat Resilient TCP; Chapter 4 presents the design and integration of $u$TCP and the accompanying application-level libraries as a part of Minion; Chapter 5 describes the implementations of each change; Chapter 6 evaluates both changes

experimentally and Chapter 7 concludes.

# Chapter 2

# Background and Related Work

We motivate BBR and $u$TCP by examining TCP's design semantics and providing an analysis of how its behavior affects application-perceived latency. We first consider TCP's sending behavior when just a single flow occupies the bottleneck link. Then, we explore how a single flow behaves when competing flows share the bottleneck link. In both cases, TCP can unnecessarily increase end-to-end delay. Lastly, we discuss related work in the TCP design space as well as alternative protocols and user-space initiatives to reduce latency.

## 2.1 A TCP Flow without Competition

We consider the scenario when a TCP flow is the only flow on the bottleneck link (i.e., no competing flows). We motivate BBR by showing how TCP's default behavior induces bufferbloat in this scenario without competition. Then, based on our observations of TCP's sending behavior, we formulate high-level goals for a bufferbloat solution.

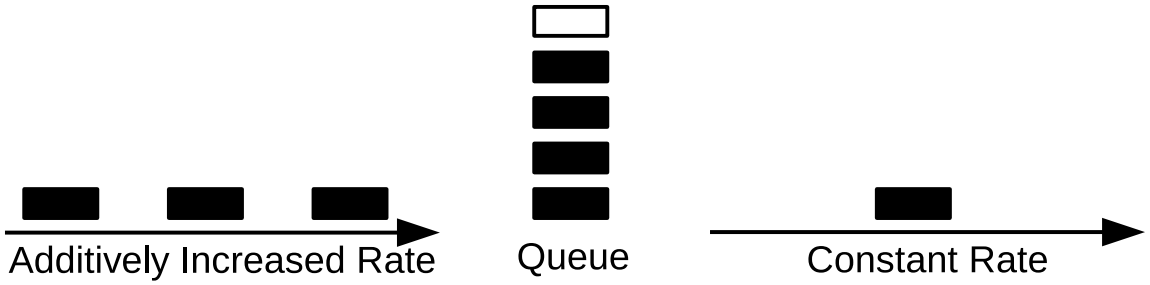



Figure 2.1: TCP's additively increased sending rate creates a faster link leading into a slower, constant rate link, which builds a standing queue and bufferbloat.

### 2.1.1 TCP's Additive Increase

At the endpoints, TCP aims to utilize the available network bandwidth fully and must adapt to dynamic network conditions. As a result, TCP uses a *congestion window*, or *cwnd*, to store the outstanding (i.e., unacknowledged) packets and tries to limit the *cwnd*'s size to the network's capacity. (The congestion window is generally proportional to the sending rate and we use the terms synonymously in this work.) Loss-based congestion control like Linux's default CUBIC grow the window until a loss event and then reduce it until recovering from the loss.

Within the network, operators deploy queues between links to handle the bursty nature of Internet traffic without incurring heavy packet loss. Without any queuing, a burst of packets could trigger a dropped packet, which automatically reduces TCP's sending rate even if the drop was random or due to fleeting congestion. As a result, networks always need some queuing to avoid severe under-utilization of the links. However, as larger queues are deployed, TCP mistakes the additional queuing capacity for bandwidth and continues increasing its *cwnd* until it experiences a loss. The result is a self-inflicted, standing queue at the bottleneck link, or bufferbloat.

Figure 2.1 depicts a typical bufferbloat scenario: TCP's additively increased sending rate causes it to exceed the speed of the slowest link on the path. The packets are not dropped because of the large buffer leading into the slower link and so a standing

queue forms. All future packets traversing this path experience higher queuing delay while the rate disparity persists.

Bufferbloat of this nature occurs in traditional consumer and data center networks alike. For example, large Internet Service Providers and Content Distribution Networks operate extremely fast, gigabit networks as part of the Internet's backbone before connecting to slower, "last mile" links leading to customers. The faster links near the content servers build up queues leading out to edge clients. Although the flow's throughput can remain high, its round-trip time is no longer a function of the path's minimum round-trip time, but rather a function of the bottleneck queue's size.

### 2.1.2 Bufferbloat in Theory

We can loosely classify a TCP flow into one of three stages at any moment based on whether its sending rate a) is below, b) matches or c) exceeds the path's capacity. As previously mentioned, over-buffering in the network can cause flows to enter and remain in phase c), even potentially *increasing* their rate while there. Typically, a new TCP flow will enter these three phases in succession, as dictated by TCP's additive increase [25].

Figure 2.2 depicts the three phases of a theoretical TCP CUBIC flow as it saturates a bottleneck link and enters bufferbloat broken down into two simplified graphs showing (i) RTT and (ii) Throughput as a function of the flow's *congestion window*. In this depiction, as in most cases of bufferbloat, the flow continuously increases its *congestion window* because the bottleneck link does not drop packets until after bufferbloat occurs.

On connection startup (a) the flow under-utilizes the link and continues to probe the bandwidth by increasing its *cwnd*. During this time, the sending rate does

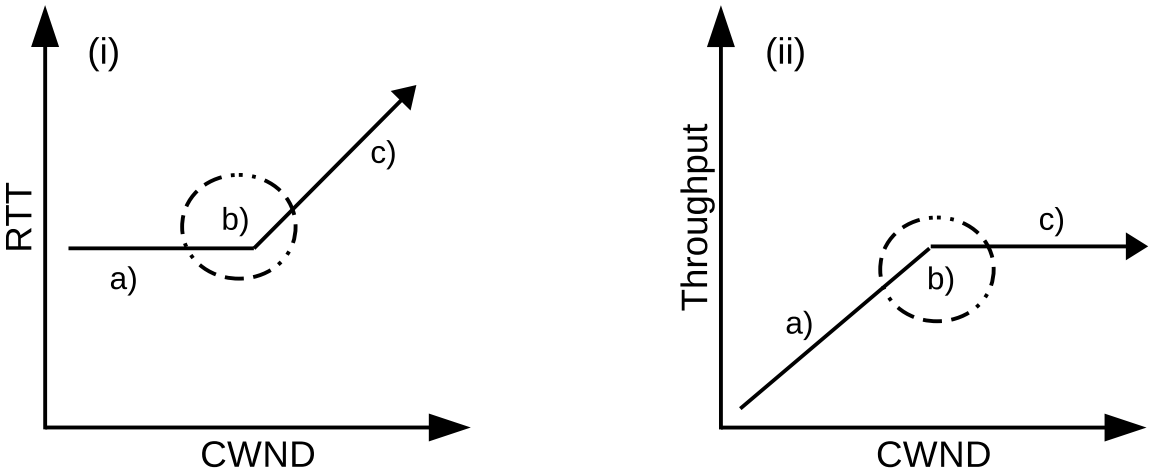



Figure 2.2: Three phases [a, b, c] of a TCP CUBIC flow as it enters bufferbloat with (i) RTT and (ii) Throughput shown as functions of the flow's *cwnd*. The circles mark the point at which the *cwnd* equals the path's BDP.

not exceed the link capacity so the observed RTT, graph (i), remains constant - the packets spend no time in the bottleneck queue. Also during phase (a), the flow experiences steadily increasing throughput proportional to the sending rate. Without queuing or dropped packets, the throughput is limited only by the sending rate while observed RTT remains constant.

Eventually the flow saturates the bottleneck link (b) and transmits at link capacity. This point marks the bandwidth-delay product (BDP) of the path and represents the ideal sending rate barring congestion. (Due to traffic burstiness, the ideal sending rate may be slightly above the BDP in order to achieve maximum link utilization.) This *cwnd* value also marks the inflection point of the discontinuity in both the RTT and Throughput graphs. Beyond this point, the RTT line no longer has a slope of zero but rather a slope proportional to the cwnd. Conversely, the Throughput stops increasing and instead remains constant with a slope of zero.

As the *congestion window* increases beyond the BDP (c), the steadily increasing RTT signals the increased time spent in the bottleneck queue and marks bufferbloat. Furthermore, the flow's throughput is now bounded by the network's capacity.

### 2.1.3 Competition

The standing queue that results from bufferbloat increases the observed RTT regardless of the flow. However, it is true that the per-packet RTT is less problematic in some scenarios. For example, bulk downloads care more about flow completion time rather than per-packet RTTs. Thus, while a bulk download can trigger bufferbloat, it does not necessarily suffer due to the standing queue. However, consider the case when a user initiates a bulk download and immediately starts a web browsing session or some other interactive application. If the second TCP flow shares the bottleneck queue (e.g., last mile) then it will experience a significantly higher latency and worse user experience [25]. In addition to higher queuing delay, bursty traffic potentially experiences more dropped packets due to the queue nearing its capacity.

Currently, BBR's focus is on the initial bulk download that causes the standing queue. By keeping the queue shorter, competing interactive flows, such as web browsing, will experience lower end-to-end latency. However, in the scenario of multiple bulk downloads all competing for bandwidth, BBR achieves no better performance than TCP's CUBIC.

In theory, competition over the same path introduces noise in both the RTT and throughput plots in Figure 2.2 due the non-uniform nature of traffic bursts. In the presence of multiple flows, a new connection should not see the steady throughput increase as it ramps up its *cwnd* and its observed RTT will almost certainly not remain constant.

Ideally, the reduced correlation during competition should prevent BBR from clamping the sender and enable it to compete for bandwidth as aggressively as the other flows. In future work, we hope to be able to better compete for a fair share of the bottleneck bandwidth. However, this unfairness also affects standard TCP

CUBIC as well.

### 2.1.4 Goals

Based on the analysis above it is clear that a bufferbloat solution should throttle a flow's sending rate to the path's BDP *during periods without competition.* During competition, the flow should behave as aggressively as its competitors or, more simply, adhere to its congestion control algorithm. We identify three goals for our solution during periods where TCP would normally enter bufferbloat (i.e., no competition):

1. **Reduce Latency:** A flow's delay should approximate the path's minimum RTT, remaining a constant factor within the minimum round-trip time.

2. **Maintain Throughput:** A flow's throughput should not decrease.

3. **Reasonable Overhead:** CPU and memory overhead should not noticeably affect system performance. Additionally, deployment costs should be minimal so that the solution works equally well in both an "on by default" deployment or a more dynamic, application-controlled scenario.

We summarize the goals above by stating that a solution should accurately identify and mitigate instances of bufferbloat but be safe enough for an on-by-default deployment. A solution should only limit the sending rate when it is certain of bufferbloat's presence. At all other times, the control of the transmission rate should transparently defer to the congestion control algorithm.

## 2.2 TCP During Congestion

We now consider the scenario when multiple TCP flows share a bottleneck link. Current TCP CUBIC behavior dictates that the flows increase their sending rate, triggering a constant cycle of lost packets and retransmissions. However, a lost packet in TCP affects the delay for *all* subsequent packets until the successful retransmission. This design decision has significant consequences, which we now explore.

### 2.2.1 In-Order Delivery

TCP [54] was originally designed to offer applications a convenient, high-level communication abstraction with semantics emulating Unix file I/O or pipes. As such, TCP enforces strict in-order delivery of received packets to the application. While originally convenient, these semantics are not always necessary given that application-level libraries can just as easily impose packet ordering. Enforcing in-order delivery reduces the flexibility of applications running atop TCP.

During congestion, packet losses increase at the bottleneck router, triggering a decrease in TCP's sending rate (and helping to avoid bufferbloat). The receiver needs at least one full round-trip time to signal the lost packet and wait for its retransmission. During this time, successfully received packets ordered after the lost packet reside in TCP's buffers, waiting to be delivered. Thus, congestive losses in TCP increase the latency on both the lost packet and all subsequent packets.

Given the recent rise in application-level "transports" that use TCP as a generic packet substrate, this in-order delivery model can severely hurt user experience during congestion. Section 2.3 further discusses the use of TCP as a generic transport substrate, but we next focus on how TCP's in-order design affect any application during congestive loss.

### 2.2.2 TCP's Latency Tax

By delaying any segment's delivery to the application until all prior segments are received and delivered, TCP imposes a "latency tax" on all segments arriving within one round-trip time (RTT) after any single lost segment. We demonstrate this effect with a simple experiment in which a server transmits data to a client with random loss simulating congestion. We compare the application-perceived reception of data segments when using standard TCP and our new, *unordered* TCP, or *u*TCP, which we present and analyze in later sections.

Figure 2.3 illustrates TCP's latency tax, showing cumulative bytes delivered over time during a simple bulk transfer atop TCP versus the unordered *u*TCP substrate we introduce later. The transfer runs on a network path with 60ms RTT, with an unrealistically high 3% loss to make several losses appear in the short duration. From the application's perspective, all receive progress stops for one or more round-trip times after any segment's loss, while *u*TCP continues delivery without interruption.

Figure 2.4 shows a cumulative distribution function (CDF) plot of application-observed delivery latencies in an experiment suggestive of streaming video, where the sender transmits 1448-byte records at fixed 20ms intervals, over a path 100ms RTT and 2% random loss. Atop TCP, over 10% of records are delayed by at least one full RTT, waiting in TCP's receive queue for a prior segment's retransmission. Unordered delivery atop *u*TCP significantly delays fewer than 3% of records.

These experiments merely illustrate TCP's "latency tax"; we defer to later sections for further analysis of TCP's latency effects under more realistic conditions. For now, the important point is that this latency tax is a fundamental byproduct of TCP's in-order delivery model, and is irreducible, in that an application-level transport cannot "claw back" the time a potentially useful segment has wasted in

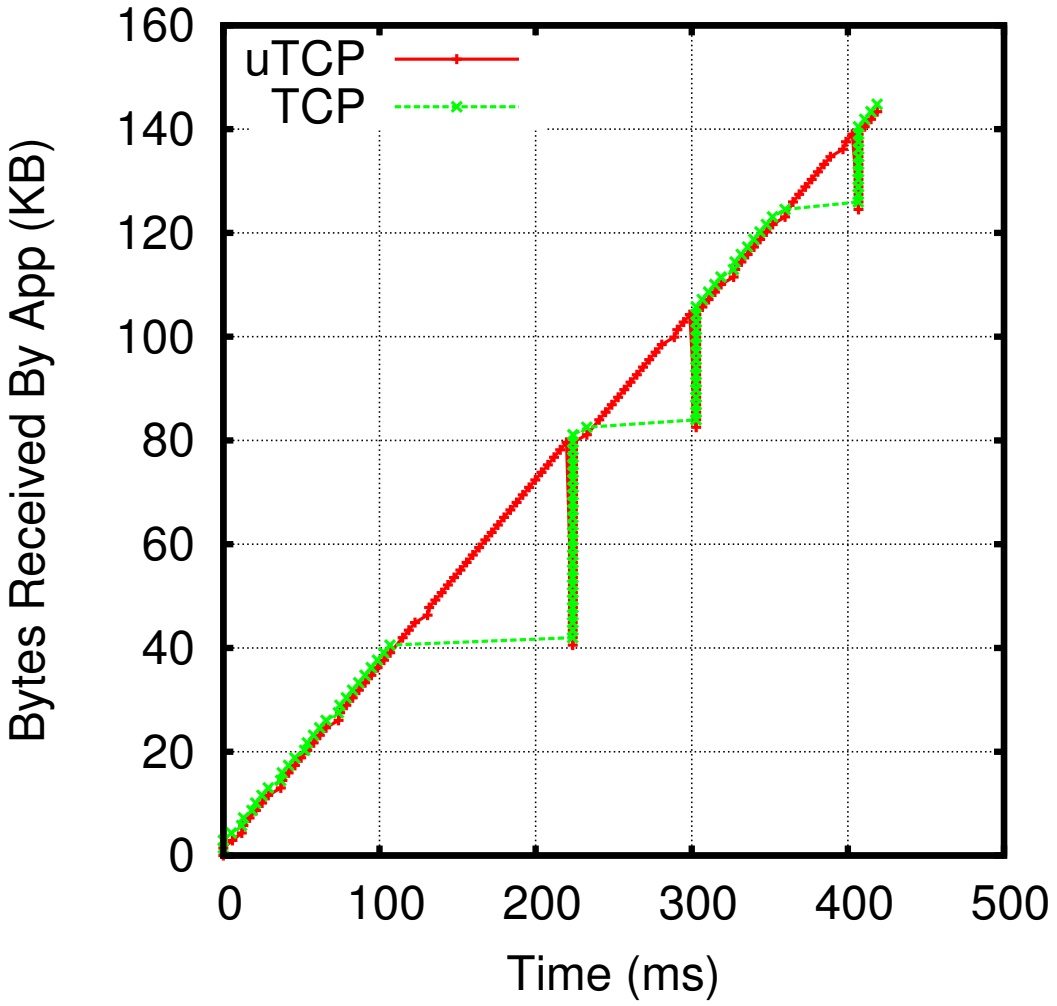



Figure 2.3: Bytes received by an application over time, in-order atop TCP versus out-of-order atop $u$TCP.

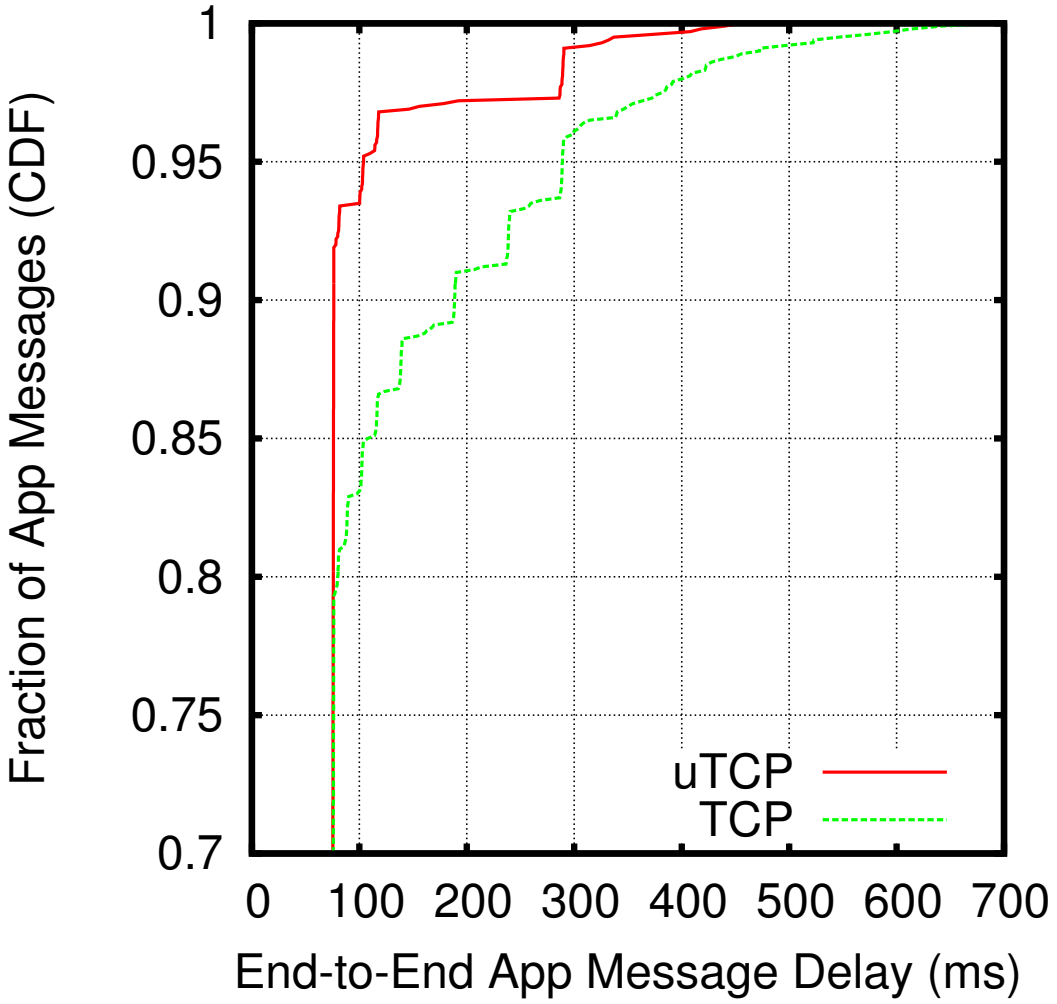



Figure 2.4: Cumulative distribution function plot of application-observed message delivery latency via in-order TCP versus out-of-order $u$TCP.

TCP's buffers. The best the application can do is simply to *expect* higher latencies to be common. A conferencing application can use a longer jitter buffer, for example, at the cost of increasing user-perceptible lag. Network hardware advances are unlikely to address this issue, since TCP's latency tax depends on RTT, which is lower-bounded by the speed of light for long-distance communications.

### 2.2.3 Why Not UDP?

As the only widely-supported transport with out-of-order delivery, UDP offers a natural substrate for application-level transports. Even applications otherwise well-suited to UDP's delivery model often favor TCP as a substrate, however. A recent study found over 70% of streaming media using TCP [29], and even latency-sensitive conferencing applications such as Skype often use TCP [7].

Network middleboxes support UDP widely but not *universally*. For this reason, latency-sensitive applications seeking maximal connectivity "in the wild" often fall back to TCP when UDP connectivity fails. Skype [7] and Microsoft's DirectAccess VPN [17], for example, support UDP but can masquerade as HTTP or HTTPS streams atop TCP when required for connectivity.

TCP can offer performance advantages over UDP as well. For applications requiring congestion control, an OS-level implementation in TCP may be more timing-accurate than an application-level implementation in a UDP-based protocol, because the OS kernel can avoid the timing artifacts of system calls and process scheduling [64]. Hardware TCP offload engines can optimize common-case efficiency in end hosts [37], and performance enhancing proxies can optimize TCP throughput across diverse networks [12, 14]. Since middleboxes can track TCP's state machine, they impose much longer idle timeouts on open TCP connections—nominally two hours [28]—whereas UDP-based applications must send keep-alives every two min-

utes to keep an idle connection open [6], draining power on mobile devices.

For applications, TCP versus UDP represents an "all-or-nothing" choice on the spectrum of services applications need. Applications desiring some but not all of TCP's services, such as congestion control but unordered delivery, must re-implement and tune all other services atop UDP or suffer TCP's performance penalties.

Without dismissing UDP's usefulness as a truly "least-common-denominator" substrate, we believe the factors above suggest that TCP will also remain a popular substrate—even for latency-sensitive applications that can benefit from out-of-order delivery—and that a deployable, backward-compatible workaround to TCP's latency tax can significantly benefit such applications.

### 2.2.4 Goals of $u$TCP

We summarize our receiver-side low latency goals as follows:

1. **Network compatibility:** Connections exploiting unordered delivery should still traverse any network path traversable by TCP, including any on-path middleboxes compatible with TCP semantics. This includes middleboxes that interpose on and split the end-to-end TCP connection, rewrite TCP sequence numbers, fragment or concatenate segments, introduce or drop TCP options, or affect the connection's behavior in other ways while preserving the integrity of the TCP byte-stream.

2. **Minimal deployment cost:** The receiver-side change should seek to minimize changes to kernel TCP stacks and APIs, offering only the minimum functionality needed to support out-of-order delivery. Other desirable features, such as multistreaming [53], ought to be implemented at application level if needed. Furthermore, unordered delivery should be incrementally deployable such that

a single endpoint can enable it without requiring negotiation or agreement from the sender.

## 2.3 Related Work

We highlight related work as it pertains to avoiding bufferbloat and reducing application-perceived latency.

*Reducing Bufferbloat:* There are many ways to potentially mitigate bufferbloat's effects but none have shown real traction throughout the Internet. For example, in-network solutions such as CoDel [39] and Random Early Detection [22] (RED) propose active queue management (AQM) techniques to prevent bufferbloat. While AQM schemes approach the problem at its source, they are slow to deploy at large-scale due to the usual barriers. Indeed, RED was proposed more than ten years ago and still has not seen widespread deployment. Furthermore, they leave proactive and/or willing end-users without an option for immediate improvement. Delay-based congestion control schemes, such as TCP Vegas [8], can avoid bufferbloat but have been eschewed in favor of loss-based schemes such as TCP CUBIC [30] (the default in Linux) due to their performance during competition. FAST TCP [61], is another delay-based scheme but falls under patent protection and ownership by Akamai Technologies [1].

Gettys' work [25] analyzes bufferbloat and proposes solutions such as AQM to manage buffers better, rather than modifying TCP directly to prevent bufferbloat. Remy [63] treats bandwidth allocation as an optimization problem and coordinates endpoints to achieve better bandwidth efficiency, but this work derives mostly from simulation and has not been tested in a live, production environment. Lastly, simpler approaches like Trickle [27], set a maximum sending rate for TCP flows to reduce

burst-induced loss, but work best when the sending rate can be tuned for specific applications, such as video streaming or data center to data center transfers.

*New transports for latency-sensitive apps:* Brosh et al. [9] model TCP latency, and identify the regions of operation for latency-sensitive apps with TCP. While some of the considerations apply, such as latency induced by TCP congestion control, $u$TCP extends the working region for such apps by eliminating delays at the receiver.

DCCP [35,36] provides an unreliable, unordered datagram service with negotiable congestion control. SCTP [53] provides unordered and partially-ordered delivery services to the application. Both DCCP and SCTP face large deployment barriers on today's Internet, however, and are thus not widely used.

New transports such as SST [23] and CUSP [55] run atop UDP to increase deployability, and UDP tunneling schemes have been proposed for standardized Internet transports as well [44, 56]. Many Internet paths block UDP traffic as well, however, as evidenced by the shift of popular VoIP applications such as Skype [7] and VPNs such as DirectAccess [17] toward tunneling atop TCP instead of UDP, despite the performance disadvantages.

*Message Framing over TCP:* Protocols such as HTTP [20], SIP [51], and iSCSI [52], can all benefit from out-of-order delivery, but use TCP for legacy and network compatibility reasons. All use simple type-length-value (TLV) encodings, which do not directly support out-of-order delivery even with $u$TCP, because they offer no reliable way to distinguish a record header from data in a TCP stream fragment. While COBS [13] represents an attractive set of characteristics for framing records to enable out-of-order delivery, other encodings such as BABS [11] also represent viable alternatives.

# Chapter 3

# Sender-Side Changes

We now introduce Bufferbloat Resilient TCP (BBR) to prevent and mitigate the effects of bufferbloat. We briefly explain how BBR interposes on the TCP stack. Then, we show how the correlation between the sending rate and the observed RTT is used to signal bufferbloat. Next, we present the BBR algorithm which uses a dynamic correlation measurement to limit a TCP sender to the estimated BDP. Lastly, we describe key optimizations that improve the algorithm's robustness.

Figure 3.1 shows how BBR interacts with the sender's TCP implementation. BBR tries to identify the presence of bufferbloat based on outgoing packets and their associated incoming acknowledgment packet (ACK). BBR either controls the *cwnd* itself or calls the congestion control module as TCP normally would.

## 3.1 Delay Correlation

We continue with our observation of RTT as a function of the sending rate and consider the bufferbloat phase - when the sending rate exceeds the BDP and before any losses. In this phase, any sending rate builds a standing queue at the bottleneck, directly resulting in higher observed RTT. However, a steadily increasing, rather

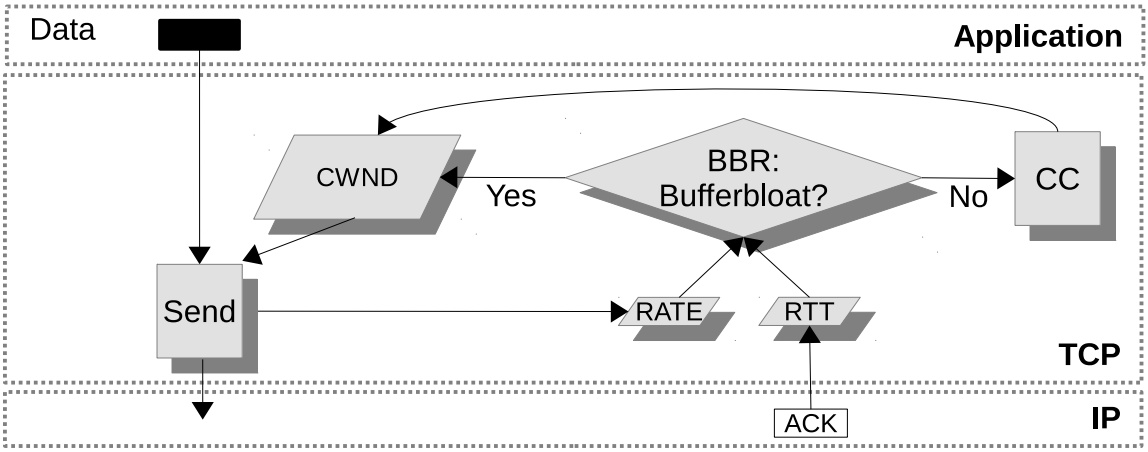



Figure 3.1: Sender-side TCP implementation with BBR interposed to control the *cwnd* only during periods of bufferbloat.

than constant, sending rate produces a proportional increase in the observed RTT. Thus, a proportionality exists between the growth of the sending rate and the RTT during bufferbloat. Measuring the correlation between these values over a series of observations captures their relationship, or dependency, in a way that their raw values do not. A strong positive correlation between the two implies the sending rate determines the latency (i.e., bufferbloat).

Next, we contrast the correlated state of bufferbloat above with the measurable correlation *prior* to the bufferbloat phase. Before a flow saturates a link, but while its sending rate increases, we expect the measured RTT over a series of packets to hold steady near the minimum observed RTT (e.g., usually the first packet sent on the path). As a result, the measured correlation between the changing sending rate and the "constant" RTT should be statistically insignificant. Fluctuations in RTT during this phase are inevitable due to noise and burstiness, but they should lack the proportionality measured by the correlation. Similarly, increased delay due to competing flows should lack the high correlation in proportionality present during bufferbloat. The bursty and non-uniform nature of competing traffic acts to combat the proportionality between sending rate and delay.

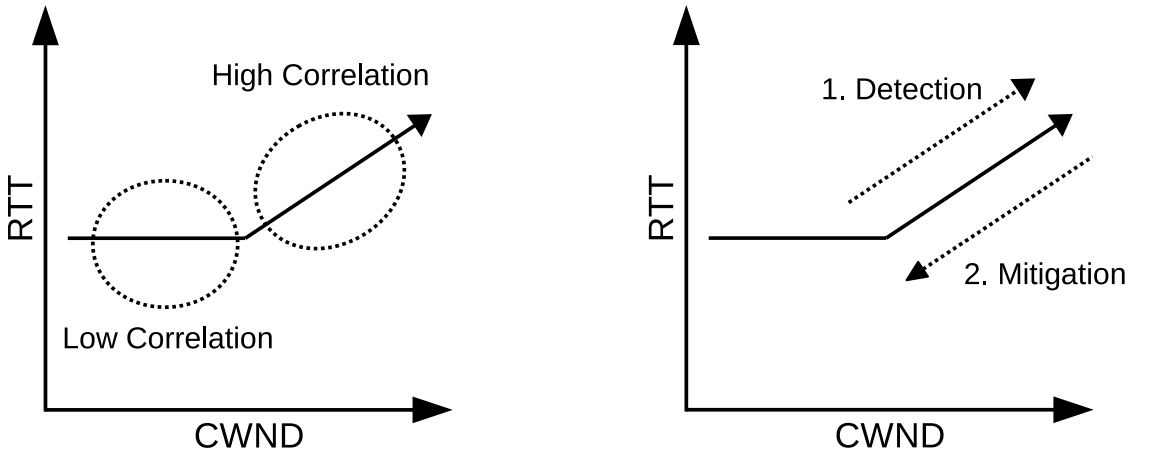



Figure 3.2: A depiction of the RTT as a function of TCP's *cwnd* before and during bufferbloat. The left graph annotates the correlation between the *cwnd* and RTT when the *cwnd* is less than or greater than the BDP. The right graph shows how BBR adjusts the *cwnd* during bufferbloat detection and mitigation.

Figure 3.2 depicts the delay correlation analysis described above in the left graph and shows how BBR tracks this correlation during bufferbloat in the right graph. BBR uses a sliding *window* of recent observations to calculate the correlation coefficient [62] where each observation consists of an approximation of the sending rate at the time of transmitting a packet and the RTT measured using the ACK for that packet. The *cwnd* is a good approximation for the momentary sending rate, though Section 3.2.2 presents a slightly improved approximation.

The graph on the right in Figure 3.2 shows how BBR reacts during bufferbloat. The correlation values range from $-1$ (perfect negative correlation) to 1 (perfect positive correlation). As the delay increases due to bufferbloat, the correlation approaches 1. BBR detects bufferbloat by comparing its calculated correlation to a configurable threshold (e.g., 90%) and declaring bufferbloat whenever the value exceeds the threshold.

Upon declaring bufferbloat, BBR *clamps* the *cwnd* (i.e., sending rate), preventing it from increasing further. Then, BBR reduces the *cwnd*, thereby "walking down"

the bufferbloat line in Figure 3.2. Conveniently, the correlation remains near 1 even as BBR throttles the *cwnd* because of the proportionality in the corresponding delay reduction. Finally, BBR releases the clamp on the *cwnd* once the correlation drops below the threshold, which occurs roughly when the *cwnd* equals the path's BDP.

## 3.2 Algorithm

BBR executes within the kernel's TCP stack implementation and stores state within the kernel's per-socket data structure. We discuss the per-socket state overhead in detail in Chapter 5. BBR has access to segments at their time of transmission as well as the reception time of the corresponding acknowledgment segments.

### 3.2.1 State

Different underlying mathematical models of correlation require different state. Currently, our proof-of-concept BBR implementation uses a Linear Least Squares (LLS) regression model for fitting a line of the form $y = mx + b$ to a series of $(x, y)$ observations. Each observation stores an approximation of the sending rate, *rate*, at the time of the packet's transmission and an observed RTT, *rtt*, recorded at the time of the ACK's reception. Additionally, BBR uses the regression line and observations to calculate the correlation coefficient between the $(x, y)$ variables as a measure of the error, or uncertainty, in the regression line.

The LLS model uses a circular array for storing the most recent observations and maintains running sums for several quantities used in the LLS calculation. Currently, the default observation window stores 32 observations but Section 3.4 describes alternative sizes. Maintaining running sums of quantities, such as $rate^2$, $rtt^2$, $rate{*}rtt$, etc., facilitates faster re-calculation of the LLS regression line.

### 3.2.2 Packet Transmission

Delay observations require both a *rate* and *rtt* value. The *rate* value should approximate the sending rate at the time of transmitting a packet. Rather than statically using the current *cwnd* as the *rate* value for every outgoing packet, BBR uses the more accurate calculation of "packets in flight". This value estimates the raw number of packets in the network by accounting for all transmitted and acknowledged packets.

Using packets in flight for the *rate* has the advantage that the first packet in a burst records a smaller rate than the last packet in the burst, which corresponds roughly to the expected size of the bottleneck queue that each packet will experience. Barring reordering, a later packet experiences a longer delay, so storing a higher *rate* for that packet improves the calculated correlation.

### 3.2.3 Update

The reception of an ACK creates a single new observation. Thus, if an ACK acknowledges two or more packets, BBR chooses the last, now-acknowledged packet that was sent and creates a single new observation using that packet's *rate* value. This avoids introducing errors due to delayed or stretch ACKs. This packet's *rate* (i.e., "inflight") value, stored at the time of the packet's transmission, serves as the observation's *rate* value. The observation's *rtt* value is the time between packet's transmission and its ACK's reception. TCP implementations already calculate a per-packet RTT in this fashion, so BBR can easily reuse it.

Initially, *bbr_update* stored every new observation but investigation showed that using a sampling approach improves robustness to bursty effects on consecutive observations. When the arrival of an ACK creates a new observation, BBR determines

**Algorithm 1** Store a new entry in the observation window.

```
1: function bbr_update(rate, rtt)
2:     wlen = length of window
3:     index = index of oldest observation in window
4:     sums = running sums for correlation calculation
5:     oldobs = window[index]
6:     subtract oldobs.{rate, rtt} from sums
7:     newobs = < rate, rtt >
8:     window[index] = newobs
9:     add newobs.{rate, rtt} to sums
10:    increment index modulus wlen
11: end function
```

if the observation should be stored or discarded based on a scaled probability distribution. As a result, every new ACK does not automatically trigger a call to *bbr_update*. The sampling method is described in detail in Section 3.4.

Algorithm 1 gives a basic pseudo-code for how *bbr_update* processes the new observation values. The high-level goal of *bbr_update* is to incorporate the new observation into the model's view of the network, but the implementation is model-dependent. Because the LLS model requires a window of observations, *bbr_update* uses a circular array to overwrite the oldest observation with the new one, and correspondingly updates the running sums by subtracting the old and adding the new observation's values.

### 3.2.4 Consult

LLS fits a regression line of the form $y = mx + b$ to the series of $(x, y)$ input coordinates such that the line minimizes the sum of the squared differences. Each input ordered pair is a $< rate, rtt >$ observation and the regression calculation solves for the slope, $m$, and the y-intercept, $b$. In addition to the regression line, BBR also uses the running sums to calculate the correlation coefficient [62], which describes

**Algorithm 2** Detect and react to bufferbloat.

```
1: function bbr_consult
2:     sums = running sums for correlation calculation
3:     R = correlation coefficient from sums
4:     if R < bufferbloat threshold then
5:         return
6:     end if
7:     m = regression slope from sums
8:     b = regression intercept from sums
   % Path's minimum RTT estimate, Section 3.3
9:     rate_est = (min_rtt - b) / m
   % bbr_dither presented in Section 3.4
10:    rate_est = bbr_dither(rate_est)
11:    cwnd = rate_est
12: end function
```

the proportion of the variance in *rtt* residuals accounted for by the regression line. Essentially, the correlation provides a good estimate for how much of the observed *rtt*'s variance is due to the change in the *rate*.

Algorithm 2 provides pseudo-code for the *bbr_consult* operation. The first steps calculate the correlation, $R$, and compare it to the configured bufferbloat threshold. If $R$ meets or exceeds the threshold then the slope and y-intercept are calculated. Now, we solve for $x$ (i.e., *rate*) using the slope, y-intercept and the minimum estimated RTT for the path. The minimum RTT estimate is introduced in a separate kernel patch and described in more detail in Section 3.3. The *rate_est* value provides an estimate for the *cwnd* size that yields an RTT close to the path's minimum RTT based on the past observations. We dither the *rate_est* and describe the dithering function in Section 3.4.

Upon detecting bufferbloat, BBR immediately sets the TCP *cwnd* to the window estimate, rather than gradually reducing the sending rate as in, for example, Proportional Rate Reduction [19]. Immediately changing the *cwnd* has two advantages.

First, it stops packet transmissions immediately and prevents further exacerbating the delay. When the transmissions resume with the reduced *cwnd*, they should experience shorter queuing delay. Second, drastic changes in sending rate should result in sharp differences in observed RTT, increasing the signal-to-noise ratio in the observations and improving the correlation calculation.

## 3.3 Minimum RTT Estimation

BBR's central goal is to limit the sending rate to approximately the path's BDP, as measured in packets (or bytes) per unit time. Ideally, the time unit is scaled by the path's propagation delay (i.e., minimum RTT) so that the *cwnd* simply equals the BDP estimate. As such, BBR uses a dynamic estimate of the connection's minimum RTT and introduces this estimate as a secondary patch because the current Linux TCP stack does not already implement it.

There are potentially many ways to estimate the path's minimum RTT. The simplest approach uses a single variable to store the smallest observed RTT during the lifetime of the connection. This approach is adequate for static paths absent any competition, however, it quickly loses value in the face of competing flows or dynamic path properties.

A better approach would be to track several estimates of the minimum RTT over a recent time window (e.g., 5 minutes) for the connection. Such a dynamic estimate allows the sender to detect underlying path changes. The more accurate the minimum RTT estimation, the better BBR performs, but BBR does not depend on any specific minimum RTT estimation. As a result, we introduce the minimum RTT code as a secondary patch.

## 3.4 Optimizations

We now describe several optimizations that improve BBR's performance. The correlation bufferbloat threshold determines when BBR declares bufferbloat and reduces the *cwnd*. Currently, BBR uses a threshold of 90% but testing in simulations and production showed BBR to be less sensitive to this value than expected. In many cases of bufferbloat the correlation remains higher than 98% and instances without bufferbloat exhibit correlation close to zero. Correlation values in the interval between 0 and 1 are generally fleeting as the connection is moving towards one end or the other (during periods without competition).

### 3.4.1 Observation Window

The current LLS regression model takes a calculation over a window, or series, of observations. A smaller window assigns more weight to each observation causing more volatility in the correlation measurement. A smaller window allows BBR to more quickly detect and react to bufferbloat but it makes it more susceptible to input noise. Conversely, a larger window reacts more slowly to bufferbloat but is more robust to noise due to the decreased weight each observation bears.

A more important characteristic than the window's size is its *membership* with respect to the *cwnd* (and, indirectly, the BDP). Ideally, the observation window provides a representative sample of data points from multiple *cwnd* rounds. As a result, BBR randomly selects observations based on the current *cwnd* so that the observation window contains a representative sample from the last two *cwnd* rounds. This sampling increases the information in the window by preventing a burst of very similar observations from filling up the entire window. For example, a window full of consecutive observations would be more susceptible to noisy measurements and may

overreact to temporary or bursty conditions.

### 3.4.2 Sufficient Excitation

After slow start, TCP increases its *cwnd* one packet per full *cwnd* of packets. However, after detecting bufferbloat, BBR prevents the *cwnd* from increasing above its window estimate as long as the correlation remains strong enough. Thus, BBR can induce a type of "steady state", where the *cwnd* does not change and each acknowledged packet triggers the transmission of one new packet.

Eventually, all observations in the observation window will record the same *rate* value (since BBR prevents changes to the *cwnd*). Furthermore, since the sending rate is constant, the observed RTT should stabilize causing the observation window to store the same $< rate, rtt >$ observation at every index. Such a series of data points yields a correlation of zero and forces BBR to release control of the *cwnd*, essentially triggering bufferbloat all over again. To prevent such a uniform observation window and the resulting oscillatory behavior, BBR dithers its window estimate to ensure *sufficient excitation*. Dithering the *cwnd* between a smaller and larger value allows BBR to continuously check for the presence of bufferbloat via the smaller and larger observed RTTs.

Algorithm 3 presents the *bbr_dither* function, which is called by *bbr_consult*. The goal of the dither function is simple: fill half of the observation window with samples of one *rate* value and the other half with a distinctly different (e.g., larger) value to maintain the information content in the observation window via sharply different *rates*. Thus, for observations in the first half of the window, *bbr_dither* returns a *cwnd* value just 2 packets larger than the calculated window estimate. This smaller dither value essentially tries to use the estimated ideal *cwnd*, but we increase it slightly to help avoid under-utilization of the link.

**Algorithm 3** Window-based dithering.

```
1: function bbr_dither(rate_est)
2:     wlen = length of window
3:     index = index of oldest observation in window
4:     if index < (wlen >> 1) then
5:         dither = 2
6:     else
7:         dither = MAX(10, rate_est >> 4)
8:     end if
9:     return rate_est + dither
10: end function
```

The second half of the observation window uses a significantly higher rate estimate to *probe* for the continued presence of bufferbloat. The larger dither value results in a higher observed RTT during bufferbloat, which maintains the correlation and allows BBR to continue throttling the sender. For paths with a small BDP, the high dither value of 10 additional packets represents a large increase in sending rate. For example, a 2Mbps path with one-way delay of 50ms has a BDP of roughly 8 TCP packets, producing low and high rate estimates of 10 and 18, respectively. For paths with a very large BDP, an increase of 10 packets may not yield sufficient excitation, so we scale the dither value to 6.25% of the rate estimate. This percentage worked well in testing, though we admit further investigation may be needed.

Recall that the *rate* value is measured in terms of packets in flight, as described in Section 3.2.2. The packets in flight value accounts for both packets "on the wire" and in network queues. TCP respects this packets in flight value when determining whether or not to send more data. Thus, when transitioning the *cwnd* from the large dither value to the small, the sender cannot immediately transmit new packets at the lower rate. It must wait until the total number of packets in the network drops below the *cwnd*, thereby making "space" for the transmission of new packets. This increased delay before the transmission of new packets prevents the standing

queue from steadily increasing by waiting until it falls to the prescribed level before injecting more data.

## 3.5 Summary

BBR monitors the correlation between the rate of outgoing packets and their experienced round-trip delay. During times of high correlation, BBR throttles the sender to a rate closer to the path's estimated BDP. Lastly, BBR dithers its BDP estimate to ensure sufficient excitation of the correlation signal between *congestion window* and delay.

# Chapter 4

# Receiver-Side Changes

We now introduce *unordered* TCP, or $u$TCP, a receiver-side change to the TCP implementation that allows applications to receive data out-of-order. We first describe how $u$TCP interposes on the receiver's network stack, and then we present the design and implementation of $u$TCP. Lastly, we present two userspace libraries to aid application's in the integration and deployment of $u$TCP.

## 4.1 Architectural Components

We deploy $u$TCP as a patch to the TCP implementation but by itself, $u$TCP does not offer applications any convenient, high-level abstractions. As such, we package $u$TCP with several userspace (i.e., application-level) libraries to ease the integration and deployment of $u$TCP and call this suite of receiver protocols the Minion Architecture. Minion serves applications and higher application-level transports, by acting as a "packhorse" carrying raw datagrams as reliably and efficiently as possible across today's diverse and change-averse Internet.

Figure 4.1 illustrates the integration of $u$TCP into the network stack. Applications and higher application-level transports link in and use the $u$COBS or $u$TLS

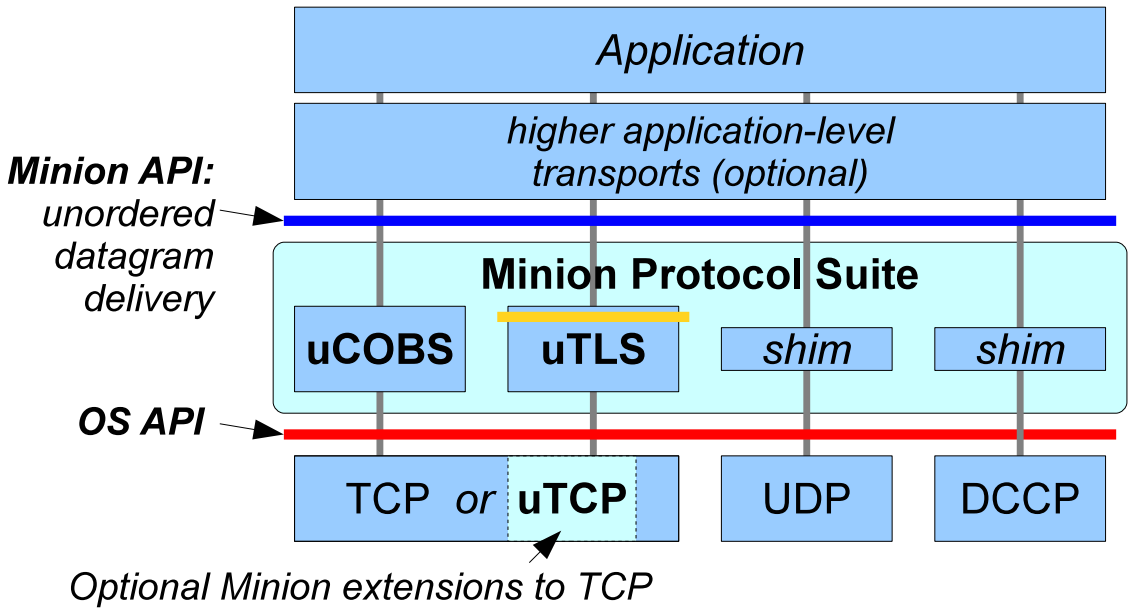



Figure 4.1: Receiver-side TCP implementation with $u$TCP.

libraries built atop $u$TCP in the same way as they already use existing application-level transports such as DTLS [49], the datagram-oriented analog of SSL/TLS [18].

While many protocols embed datagrams or application-level frames into TCP streams using delimiting schemes, to our knowledge Minion is the first application-level transport that, under suitable conditions, offers true unordered delivery atop TCP. Minion effectively offers relief from TCP's latency tax: the loss of one TCP segment in the network no longer prevents datagrams embedded in subsequent TCP segments from being delivered promptly to the application.

Minion consists of several application-level transport protocols, together with a set of optional enhancements to end hosts' OS-level TCP implementations.

Minion's enhanced OS-level TCP stack, which we call *uTCP* ("unordered TCP"), includes the receiver-side API feature supporting unordered delivery, which we describe in Section 4.2. This enhancement affects only the OS API through which application-level transports such as Minion interact with the TCP stack, and make *no* changes to TCP's wire protocol.

Minion's application-level protocol suite currently consists of *u*COBS, which implements unordered datagram delivery atop unmodified TCP or *u*TCP streams using COBS encoding [13] as described in Section 4.3; and *u*TLS, which adapts the traditionally stream-oriented TLS [18] into a secure unordered datagram delivery service atop TCP or *u*TCP. Minion also adds trivial shim layers atop OS-level datagram transports, such as UDP and DCCP, to give applications a consistent API for unordered delivery across multiple OS-level transports.

Minion currently leaves to the application the decision of *which* protocol to use for a given connection: e.g., *u*COBS or *u*TLS atop TCP/*u*TCP, or OS-level UDP or DCCP. Future work might address the negotiation of different transports during connection initialization. Many applications already incorporate simple negotiation schemes, for example, attempting a UDP connection first and falling back to TCP if that fails, and adapting these mechanisms to engage Minion's protocols according to application-defined preferences and decision criteria should be straightforward.

## 4.2 *u*TCP: Unordered TCP

Minion enhances the OS's TCP stack with API enhancements supporting unordered delivery in both TCP's send and receive paths, enabling applications to reduce transmission latency at both the sender- and receiver-side end hosts when both endpoints support *u*TCP. Since *u*TCP makes no changes to TCP's wire protocol, two endpoints need not "agree" on whether to use *u*TCP: one endpoint gains latency benefits from *u*TCP even if the other endpoint does not support it. Further, an OS may choose independently whether to support the sender- and receiver-side enhancements, and when available, applications can activate them independently.

*u*TCP does *not* seek to offer "convenient" or "clean" unordered delivery abstrac-

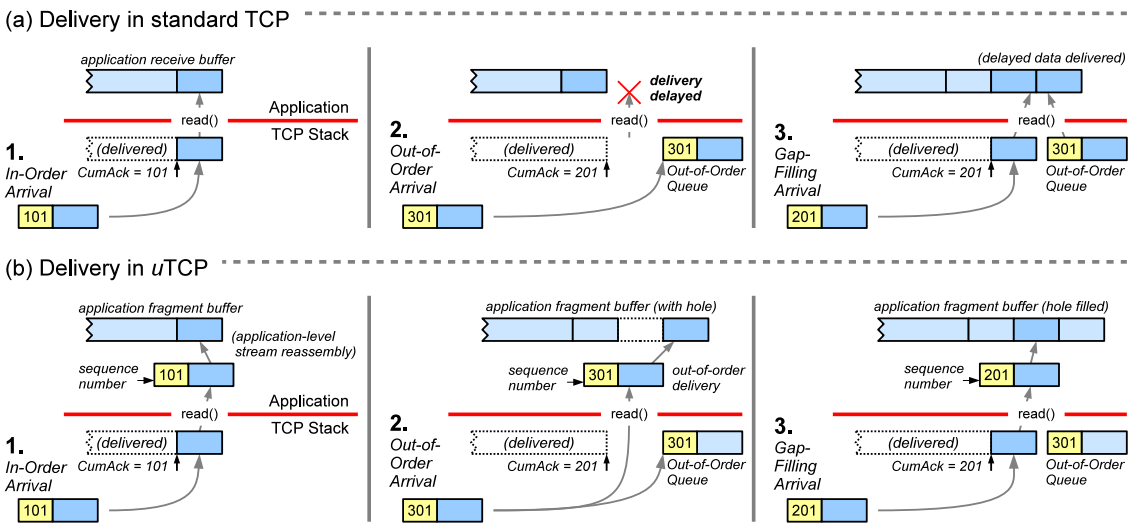



Figure 4.2: Delivery behavior of (a) standard TCP, and (b) *u*TCP, upon receipt of in-order and out-of-order segments.

tions directly at the OS API. Instead, *u*TCP's design is motivated by the goals of maintaining exact compatibility with TCP's existing wire-visible protocol and behavior, and facilitating deployability by minimizing the extent and complexity of changes to the OS's TCP stack. The design presented here is only one of many viable approaches, with different trade-offs, to supporting unordered delivery in TCP.

We describe *u*TCP's API enhancements in terms of the BSD sockets API, although *u*TCP's design contains nothing inherently specific to this API.

*u*TCP adds one new socket option affecting TCP's receive path, enabling applications to request immediate delivery of TCP segments received out of order. An application opens a TCP stream the usual way, via `connect()` or `accept()`, and may use this stream for conventional in-order communication before enabling *u*TCP. Once the application is ready to receive out-of-order data, it enables the new option `SO_UNORDERED` via `setsockopt()`, which changes TCP's receive-side behavior in two ways.

First, whereas a conventional TCP stack delivers received data to the applica-

tion only when prior gaps in the TCP sequence space are filled, the *u*TCP receiver makes data segments available to the application immediately upon receipt, skipping TCP's usual reordering queue. The application obtains this data via `read()` as usual, but the first data byte returned by a `read()` call may no longer be the one logically following the last byte returned by the prior `read()` call, in the byte stream transmitted by the sender. The data the *u*TCP stack delivers to the application in successive `read()` calls may skip forward and backward in the transmitted byte stream, and *u*TCP may even deliver portions of the transmitted stream multiple times. *u*TCP guarantees only that the data returned by one `read()` call corresponds to *some* contiguous sequence of bytes in the sender's transmitted stream, and that barring connection failure, *u*TCP will *eventually* deliver every byte of the transmitted stream at least once.

Second, when servicing an application's `read()` call, the *u*TCP receiver prepends a short header to the returned data, indicating the logical offset of the first returned byte in the sender's original byte stream. The *u*TCP stack computes this logical offset simply by subtracting the Initial Sequence Number (ISN) of the received stream from the TCP sequence number of the segment being delivered. Using this meta-data, the application can piece together data segments from successive `read()` calls into longer contiguous *fragments* of the transmitted byte stream.

Figure 4.2 illustrates *u*TCP's receive-side behavior, in a simple scenario where three TCP segments arrive in succession: first an in-order segment, then an out-of-order segment, and finally a segment filling the gap between the first two. With *u*TCP, the application receives each segment as soon as it arrives, along with the sequence number information it needs to reconstruct a complete internal view of whichever fragments of the TCP stream have arrived.

The *u*TCP receiver retains in its receive buffer the TCP headers of segments

received and delivered out-of-order, until its cumulative acknowledgment point moves past these segments, and generates acknowledgments and selective acknowledgments (SACKs) exactly as TCP normally would. The *u*TCP receiver does not increase its advertised receive window when it delivers data to the application out-of-order, so the advertised window tracks the cumulative in-order delivery point exactly as in TCP. In this fashion, *u*TCP maintains wire-visible behavior identical to TCP while delivering segments to the application out-of-order.

The *u*TCP receive path assumes the sender may be an unmodified TCP, and TCP's stream-oriented semantics allow the sending TCP to segment the sending application's stream at arbitrary points—independent of the boundaries of the sending application's `write()` calls, for example. Further, network middleboxes may silently *re-segment* TCP streams, making segment boundaries observed at the receiver differ from the sender's original transmissions [31]. An application using *u*TCP, therefore, must not assume anything about received segment boundaries. This is a key technical challenge to using *u*TCP reliably, and addressing this challenge is one function of *u*COBS and *u*TLS, described later.

## 4.3 *u*COBS: Simple Datagrams on TCP

Since *u*TCP's design attempts to minimize OS changes, its unordered delivery primitives do not directly offer applications a convenient, general-purpose datagram substrate. Minion's *u*COBS protocol bridges this semantic gap, building atop *u*TCP (or standard TCP) a lightweight datagram delivery service comparable to UDP or DCCP. This first section first introduces the challenge of delimiting datagrams, then presents *u*COBS' solution and discusses alternatives.

### 4.3.1 Self-Delimiting Datagrams for *u*TCP

Applications built on datagram substrates such as UDP generally assume the underlying layer preserves datagram boundaries. If the network fragments a large UDP datagram, the receiving host reassembles it before delivery to the application, and a correct UDP never merges multiple datagrams, or datagram fragments, into one delivery to the receiving application. TCP's stream-oriented semantics do not preserve any application-relevant frame boundaries within a stream, however. Both the TCP sender and network middleboxes can and do coalesce TCP segments or resegment TCP streams in unpredictable ways [31]. Conventional TCP applications, which send and receive TCP data in-order, commonly address this issue by delimiting application-level frames with some length-value encoding, enabling the receiver to locate the next frame in the stream from the previous frame's position and header content.

Since *u*TCP's receive path effectively just bypasses TCP's reordering buffer, delivering received segments to the application as they arrive, a stream fragment received out-of-order from *u*TCP may begin at any byte offset in the stream, and not at a frame boundary meaningful to the application. Since the receiver is by definition missing some data sent prior to this out-of-order segment, it cannot rely on preceding stream content to compute the next frame's position.

Reliable use of *u*TCP, therefore, requires that frames embedded in the TCP stream be *self-delimiting*: recognizable without knowledge of preceding or following data. A simple solution is to make frames fixed-length, so the receiver can compute the start of the next frame from the stream offset *u*TCP provides with out-of-order segments. *u*COBS is intended to offer a general-purpose datagram substrate, however, and many applications require support for variable-length frames.

If the application-level frames happen to be encoded so as never to include some "reserved" byte value, such as zero, then we could use that byte reserved value to delimit frames within $u$TCP streams. Since we wish $u$COBS to support general-purpose delivery of datagrams of variable length containing arbitrary byte values, however, $u$COBS must explicitly (re-)encode the application's datagrams in order to reserve some byte value to serve as a delimiter.

Any scheme that encodes arbitrary byte streams into strings utilizing fewer than 256 symbols will serve this purpose, such as the ubiquitous *base64* scheme, which encodes byte streams into strings utilizing only 64 ASCII symbols plus whitespace. Since base64 encodes three bytes into four ASCII symbols, however, it expands encoded streams by a factor of 4/3, incurring a 33% bandwidth overhead. Since $u$COBS needs to reserve only *one* byte value for delimiters, and not the large set of byte values considered "unsafe" in E-mail or other text-based message formats, base64 encoding is unnecessarily conservative for $u$COBS' purposes.

### 4.3.2 Operation of $u$COBS

To encode application datagrams efficiently, $u$COBS employs *consistent-overhead byte stuffing*, or COBS [13]. COBS is analogous to base64, except that it encodes byte streams to reserve only *one* distinguished byte value (e.g., zero), and utilizes the remaining 255 byte values in the encoding. COBS could in effect be termed "base255" encoding. By reserving only one byte value, COBS incurs an expansion ratio of at most 255/254, or 0.4% bandwidth overhead.

**Transmission:** When an application sends a datagram, $u$COBS first COBS-encodes the datagram to remove all zero bytes. $u$COBS then prepends a zero byte to the encoded datagram, appends a second zero byte to the end, and writes the encoded

and delimited datagram to the TCP socket. Since this sender-side encoding and transmission process operates entirely at application level within *u*COBS, and does not rely on any OS-level extensions on the sending host, *u*COBS operates even if the sender-side OS does not support *u*TCP.

The application can assign priorities to datagrams it submits to *u*COBS, however, and if the sender's OS does support the *u*TCP extensions, *u*COBS passes these priorities to the *u*TCP sender, enabling higher-priority datagrams to pass lower-priority datagrams already enqueued. Since *u*TCP respects application `write()` boundaries while reordering the send queue, *u*COBS preserves its delimiting invariant simply by writing each encoded datagram—with the leading and trailing zero bytes—in a single write.

**Reception:** At stream creation time, *u*COBS enables *u*TCP's receive-side extensions if available. If the receive-side OS does not support *u*TCP, then *u*COBS simply falls back on the standard TCP API, receiving, COBS-decoding, and delivering datagrams to the application in the order they appear in the TCP sequence space. (This may not be the application's original send order if the send-side OS supports *u*TCP.)

If the receive-side OS supports *u*TCP, then *u*COBS receives segments from *u*TCP in whatever order they arrive, then fits them together using the meta-data in *u*TCP's headers to form contiguous fragments of the TCP stream. The arrival of a TCP segment can cause *u*COBS to create a new fragment, expand an existing fragment at the beginning or end, or "fill a hole" between two fragments and merge them into one. The portion of the TCP stream before the receiver's cumulative-acknowledgment point, containing no sequence holes, *u*COBS treats as one large "fragment." *u*COBS scans the content of any new, expanded, or merged fragment for properly delimited records not yet delivered to the application. *u*COBS identifies a record by the

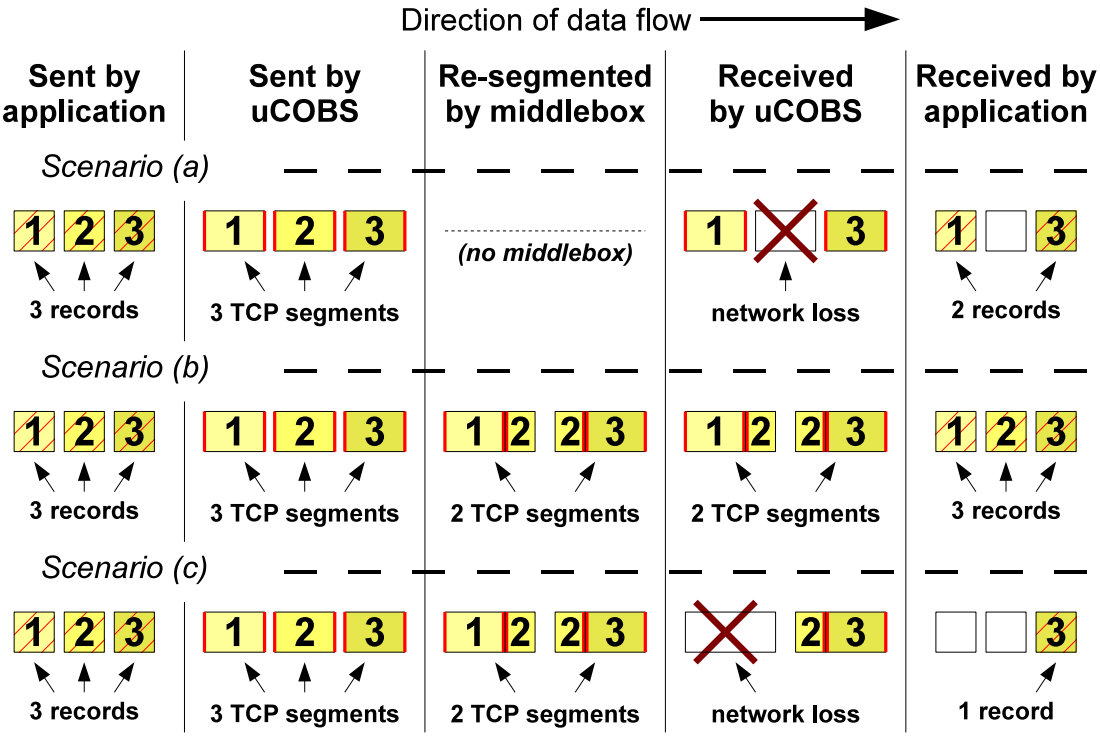



Figure 4.3: An example illustrating a *u*COBS transfer.

presence of two marker bytes surrounding a contiguous sequence of bytes containing no markers or holes. Once *u*COBS identifies a new record, it strips the delimiting markers, decodes the COBS-encoded content to obtain the original record data, and delivers the record to the application.

### 4.3.3 Why Two Markers Per Datagram?

For correctness alone, *u*COBS need only prepend *or* append a marker byte to each record—not both—but such a design could reduce performance by eliminating opportunities for out-of-order delivery. Consider Scenario (a) in Figure 4.3, in which an application sends three records. *u*COBS encodes these records and sends them via three `write()` calls, which TCP in turn sends in three separate TCP segments. In this scenario, no middleboxes re-segment the TCP stream in the network, but

the middle segment is lost. If the *u*COBS sender were only to *prepend* a marker at the start of each record, the *u*COBS receiver could not deliver record 1 immediately on receipt, since it cannot tell if record 1 extends into the following "hole" in sequence space. Similarly, if the sender were only to *append* a marker at the end of each record, then *u*COBS could not deliver segment 3 immediately on receipt, since record 3 might extend backwards into the preceding hole. By adding markers to both ends of each record, *u*COBS ensures that the receiver can deliver each record as soon as all of its segments arrive.

These markers enable *u*COBS to offer reliable out-of-order delivery even if network middleboxes re-segment the TCP stream. In Scenario (b) in Figure 4.3, for example, *u*COBS sends three records encoded into three TCP segments as above, but a middlebox re-segments them into two longer TCP segments, whose boundary splits record 2 into two parts. If neither of these segments are lost, then the *u*COBS receiver can deliver record 1 immediately upon receipt of the first TCP segment, and can deliver records 2 and 3 upon receipt of the second segment. If the first segment is lost as shown in Scenario (c), however, the *u*COBS receiver cannot deliver the missing record 1 or the partial record 2, but can still deliver record 3 as soon as the second TCP segment arrives.

## 4.4 *u*TLS: Secure Datagrams on TCP

While *u*COBS offers out-of-order delivery wire-compatible up to the TCP level, middleboxes often inspect and manipulate the *content* of TCP streams as well [48, 59]. All unencrypted network traffic today is, *de facto*, "fair game" for middleboxes—and streams exhibiting any "out of the ordinary" middlebox-visible behavior are likely to fail over *some* middleboxes [31]. An application's only way to protect "end-to-end"

communication in practice, therefore, is via end-to-end encryption and authentication. But network-layer mechanisms such as IPsec [34] face the same deployment challenges as new secure transports [23], and remain confined to the niche of corporate VPNs. Even VPNs are shifting from IPsec toward HTTPS tunnels [17], the only form of end-to-end encrypted connection almost universally supported on today's Internet. A network administrator or ISP might disable nearly any other port while claiming to offer "Internet access," but would be hard-pressed to disable HTTPS, today's foundation for E-commerce.

We could layer TLS atop *u*COBS, but TLS decrypts and delivers data only in-order, negating *u*TCP's benefit. We could also layer the datagram-oriented DTLS [49] atop *u*COBS, but the resulting (DTLS-encrypted *then* COBS-encoded) wire format would be radically different from TLS over TCP, and likely fail to traverse middleboxes expecting TLS, particularly on port 443.

The goal of *u*TLS, therefore, is to coax out-of-order delivery from the *existing* TCP-oriented TLS wire format, producing an encrypted datagram substrate indistinguishable on the wire from standard TLS connections (except via analysis of "side-channels" such as packet length and timing, which we do not address). Run on port 443, a *u*TLS stream is indistinguishable from HTTPS—regardless of whether the application actually uses HTTP headers, since the HTTP portion of HTTPS streams are TLS-encrypted anyway. Deployed this way, *u*TLS effectively offers an end-to-end protected substrate in the "HTTP as the new narrow waist" philosophy [45].

### 4.4.1 Design of *u*TLS

TLS [18] already breaks its communication into *records*, encrypts and authenticates each record, and prepends a header for transmission on the underlying TCP stream. TLS was designed to decrypt records strictly in-order, however, creating three chal-

lenges for *u*TLS:

- **Locating record headers out-of-order.** Since encrypted data may contain any byte sequence, there is no reliable way to differentiate a TLS header from record data in the TCP stream, as COBS encoding provides.
- **Encryption state chaining.** Some TLS ciphersuites chain encryption state across records, making records indecipherable until prior records are processed.
- **Record numbers used in MAC computation.** TLS includes a record number, which increases by 1 for each record, in computing the record's MAC. But the *u*TLS receiver may not know an out-of-order record's number: holes in TCP sequence space before the record could contain an unknown number of prior records.

To locate records out-of-order, *u*TLS first scans a received stream fragment for byte sequences that *may* represent the TLS 5-byte header: i.e., containing the correct record type and version, and a plausible length. While this scan may yield false positives, *u*TLS verifies the inferred header by attempting to decrypt and authenticate the record. If the cryptographic MAC check fails, instead of aborting the connection as TLS normally would, *u*TLS assumes a false positive and continues scanning.

Since TLS's MAC is designed to prevent resourceful adversaries from constructing a byte sequence the receiver could misinterpret as a record, and it is by definition at least as hard to find such a sequence "accidentally" as to forge one maliciously, TLS security should protect equally well against accidental false positives. One exception is when TLS is using its "null ciphersuite," which performs no packet authentication. With this ciphersuite, normally used only during initial key negotiation, *u*TLS disables out-of-order delivery to avoid the risk of accepting and delivering false records.

The only obvious solution to the second challenge above is to avoid ciphersuites that chain encryption state across records. Most ciphersuites before TLS 1.1 chain encryption state, unfortunately. Any stream cipher inherently does so, such as the RC4 cipher used in early SSL versions. Most recent ciphersuites use block ciphers in CBC mode. CBC ciphers do not inherently depend on chained encryption state, but do require an Initialization Vector (IV) for each record. Until recently, TLS produced each record's IV implicitly from the prior record's encryption state, making records interdependent.

To fix a security issue, however, TLS 1.1 block ciphers use explicit IVs, which the sender generates independently for each record and prepends to the record's ciphertext. As a side-effect, TLS 1.1 block ciphers support out-of-order decryption. Since TLS supports negotiation of versions and ciphersuites, *u*TLS simply leverages this process. An application can insist on TLS 1.1 with a block cipher to ensure out-of-order delivery support, or it can permit older ciphersuites to maximize interoperability, at the risk of sacrificing out-of-order delivery.

The third challenge is the implicit "pseudo-header" TLS uses in computing the MAC for each packet. This pseudo-header includes a "sequence number" that TLS increments once per *record*, rather than per *byte* as with TCP sequence numbers. When *u*TLS identifies a possible TLS record in a TCP fragment received out-of-order, the receiver knows only the byte-oriented TCP stream offset, and not the TLS record number. Since records are variable-length, unreceived holes prior to a record to be authenticated may "hide" a few large records or many smaller records, leaving the receiver uncertain of the correct record number for the MAC check.

To authenticate records out-of-order without modifying the TLS ciphersuite, therefore, *u*TLS attempts to *predict* the record's likely TLS record number, using heuristics such as the average size of past records, and may try several adjacent

record numbers to find one for which the MAC check succeeds. If $u$TLS fails to find a correct TLS record number, it cannot deliver the record out-of-order, but will still eventually deliver the record in-order.

## 4.5 Summary

$u$TCP simply exposes to applications data that might otherwise be delayed in the kernel's TCP socket buffers. Layering the Minion suite atop $u$TCP provides applications a diverse range of transport semantics. $u$COBS emulates a datagram transport such as UDP, while $u$TLS enables unordered datagram delivery within secure end-to-end communication, all without changing TCP's wire format.

# Chapter 5

# Implementation

We have developed prototype implementations of both BBR and Minion, including $u$TCP and Minion's application-linked libraries, $u$COBS and $u$TLS. We BBR and $u$TCP prototypes are Linux-specific, but we expect the API extensions it implements and the application-level libraries to be portable.

## 5.1 BBR Sender-Side Implementation

The current Linear Least Squares BBR implementation measures correlation over and fits a regression line to a series of delay observations. In terms of memory overhead, each socket buffer for outgoing packets stores the *rate* value so that it can be matched upon receiving its corresponding ACK. There are five running sums values, each a 64-bit value. Although the running sums of the observations' coordinates are enough to calculate the correlation, the full window must be stored in order to facilitate fast updates to the running sums. Each observation stores two 32-bit values.

The main determinant of the total memory overhead is the size of the observation window. An observation window of 32 samples incurs slightly more than 256 bytes of additional memory overhead per socket. We have tested observation windows as

large as 128 samples, which brings per-socket memory overhead to roughly 1KB. This is in addition to the extra *rate* integer in the socket buffer for each outgoing TCP segment, the memory footprint of which depends on the number of outstanding segments.

We introduce BBR as a patch to the Linux kernel's TCP stack. The patch touches approximately ten files and consists of approximately 360 lines of code added or changed, including comments. Our evaluations use the Linux kernel version 3.3 patched with BBR as well as the minimum RTT algorithm described in Section 3.3. We intend to open-source both patches.

Our current prototype does not work well during congestion, so applications or users can enable BBR via a Linux sysctl system call. Future work will address running BBR during competition so that an "on by default" deployment is possible.

## 5.2 Minion Receiver-Side Implementation

We briefly discuss the implementation of *u*TCP, *u*COBS and *u*TLS in our Linux-based Minion prototype.

**The *u*TCP Receiver in Linux:** The *u*TCP prototype adds about 240 lines and modifies about 50 lines of code in the Linux 2.6.34 kernel, to support the new `SO_UNORDERED` socket option. This extension involved two main changes. First, *u*TCP modifies the TCP code that delivers segments to the application, to prepend a 5-byte meta-data header to the data returned from each `read()` system call. This header consists of a 1-byte flags field and a 4-byte TCP sequence number. One flag bit is currently used, with which *u*TCP indicates whether it is delivering data in-order or out-of-order. Second, if TCP's in-order queue is empty, *u*TCP's `read()` path

checks and returns data from the out-of-order queue. To minimize kernel changes, segments remain in the out-of-order queue after delivery, so *u*TCP will eventually deliver the same data again in-order.

**The *u*COBS Library:** The *u*COBS prototype is a user-space library in C, amounting to ~700 lines of code [16]. *u*COBS presents simple `cobs_sendmsg()` and `cobs_recvmsg()` interfaces enabling applications to send and receive COBS-encoded datagrams, taking advantage of send-side prioritization and out-of-order reception depending on the presence of send- and receive-side OS support for *u*TCP, respectively.

**A *u*TLS Prototype Based on OpenSSL:** The *u*TLS prototype builds on OpenSSL 1.0.0 [43], adding ~550 lines of code and modifying ~40 lines [16]. Applications use OpenSSL's normal API to create a TLS connection atop a TCP socket, then set a new *u*TLS-specific socket option to enable out-of-order, record-oriented delivery on the socket. OpenSSL 1.0.0 unfortunately does not yet support TLS 1.1, the first TLS version that uses explicit Initialization Vectors (IVs), permitting out-of-order decryption. For experimentation, therefore, the *u*TLS prototype modifies OpenSSL's TLS 1.0 ciphersuite to use explicit IVs as in TLS 1.1. Since this change breaks OpenSSL's interoperability, our prototype is not suitable for deployment. We are currently porting *u*TLS to the next major OpenSSL release, which supports TLS 1.1.

**Implementation Complexity:** To evaluate the implementation complexity of *u*TCP and the related application-level code, Table 5.1 summarizes the source code changes *u*TCP makes to Linux's TCP stack in lines of code [16], the size of the standalone *u*COBS library, and the changes *u*TLS makes to OpenSSL's `libssl` library. The SSL/TLS total does not include OpenSSL's `libcrypt` library, which `libssl` requires but *u*TLS does not modify.

| | TCP | *u*TCP | DCCP | SCTP |
|---|---|---|---|---|
| Kernel Code | 12,982 | **565** (4.6%) | 6,338 | 19,312 |

| | *u*COBS | SSL/TLS | *u*TLS | DTLS |
|---|---|---|---|---|
| User Code | **732** | 31,359 | **586** (1.9%) | 4,734 |

Table 5.1: Code size of *u*TCP prototype as a delta to Linux's TCP stack, the *u*COBS library, and *u*TLS as a delta to `libssl` from OpenSSL. Code sizes of "native" out-of-order transports are included for comparison.

With only a 600-line change to the Linux kernel and less than 1400 lines of user-space support code, *u*TCP provides a delivery service comparable to Linux's 6,300-line native DCCP stack, while providing greater network compatibility. In user space, *u*TLS represents less than a 600-line change to the stream-oriented SSL/TLS protocol, contrasting with OpenSSL's 4,700-line implementation of DTLS, which runs only atop out-of-order transports such as UDP or DCCP.

# Chapter 6

# Evaluation

We experimentally evaluate both BBR and Minion based on their performance with regards to their stated goals from Chapter 2. For BBR, we compare it to the baseline TCP CUBIC implementation based on the metrics of end-to-end latency and bottleneck bandwidth efficiency during periods where a single flow occupies the bottleneck link (i.e., no competition). For Minion, we compare it to the baseline TCP CUBIC implementation based on the metrics of end-to-end latency and other user-perceived performance qualities, such as audio quality.

## 6.1 A BBR Sender

We deployed our proof-of-concept BBR implementation in both a simulated environment and a large, multinational production network. The micro-benchmark was run on real (i.e., non-virtual) machines using `tc` and `netem` to emulate network conditions. The video streaming experiments ran on a live, production CDN, where clients included real mobile and desktop users.

The micro-benchmark intends to give a concrete comparison of TCP's behavior versus BBR under simulated network conditions. The second experiment measures

various performance metrics for a video streaming platform, including end-to-end latency, bandwidth efficiency and retransmission rates from the perspective of the server, comparing BBR to the baseline TCP CUBIC.

### 6.1.1 BBR Micro-benchmark

Our first experiment compares the observed RTT with BBR versus TCP CUBIC. The goal is not to make a broad claim about BBR's real-world performance, but rather demonstrate how its behavior differs from Linux's default congestion control algorithm, TCP CUBIC [30].

This experiment performs a single 20-second TCP transfer, consisting of a client connecting to a server and transmitting as much bulk data as possible to the server within the time limit. The simulated network uses 10Mbps bandwidth, 10ms round-trip delay and a bottleneck queue of 1000 packets. Each flow ran in isolation without competing flows.

We chose 10Mbps for illustrative purposes because it is a common "last mile" speed for consumers but other simulations show BBR to work equally well for slow (e.g., 1Mbps) and fast (e.g., 1Gbps) links. Furthermore, our live production experiments (described in the next section) commonly observe a bottleneck bandwidth of 1Mbps.

Figure 6.1 shows the observed RTT (log-scale) BBR and CUBIC transfers. As expected, the TCP CUBIC flow steadily increases its sending rate until about 80% through the transfer. At this point, the full queue causes dropped packets. TCP responds by reducing its sending rate, resulting in a momentary decrease in observed RTT before it begins increasing its sending rate again.

BBR, on the other hand, has a brief spike in RTT at the very beginning of the transfer while its observation window fills. Soon thereafter, the correlation is

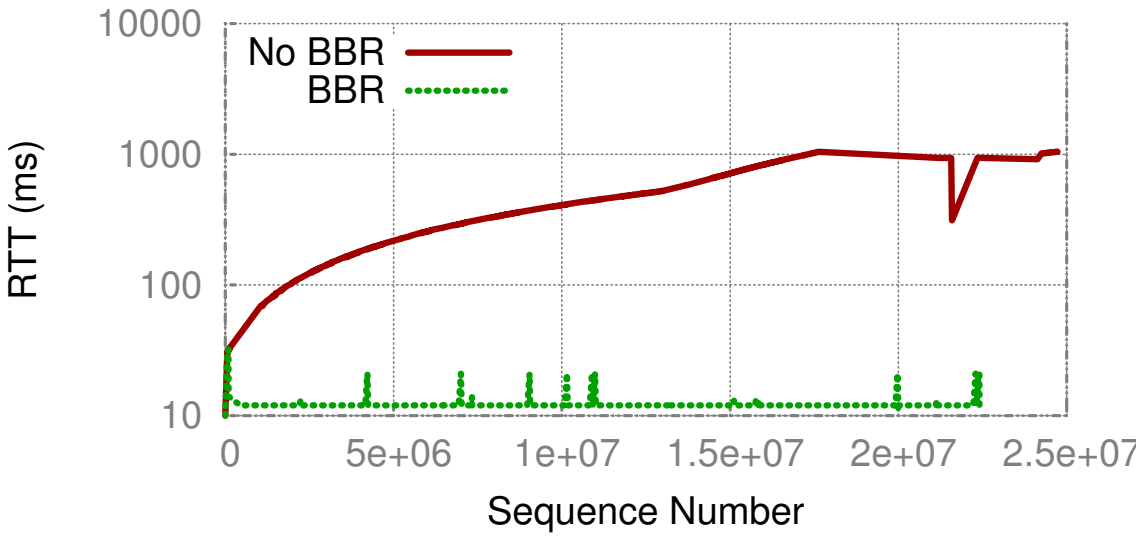



Figure 6.1: BBR produces a much lower, and consistent, observed RTT (log-scale) during a 20-second TCP transfer compared to TCP CUBIC. Flows ran separately and without competition.

strong enough to declare bufferbloat and the throttling of the sending rate begins. Periodically, BBR shows small increases in RTT during periods of the high dither value. This allows BBR to continually check that bufferbloat is still present. This experiment used an observation window of 32 samples.

Figure 6.2 shows a CDF of the observed RTT for the same 20-second TCP transfer, which highlights that BBR brings a majority of the RTT samples to within a small factor of the minimum RTT of 10ms. The average RTT for regular TCP was 493ms, while BBR's was 13.5ms, producing a roughly 2 orders of magnitude improvement.

Another interesting characteristic of BBR in this transfer is that it avoids packet drops due to a full bottleneck queue. Through trace analysis, we verified that TCP suffered over 100 retransmissions for this transfer, while BBR did not have a single packet retransmission. While this experiment represents an isolated and somewhat unrealistic environment, it serves as an example of how BBR behaves during

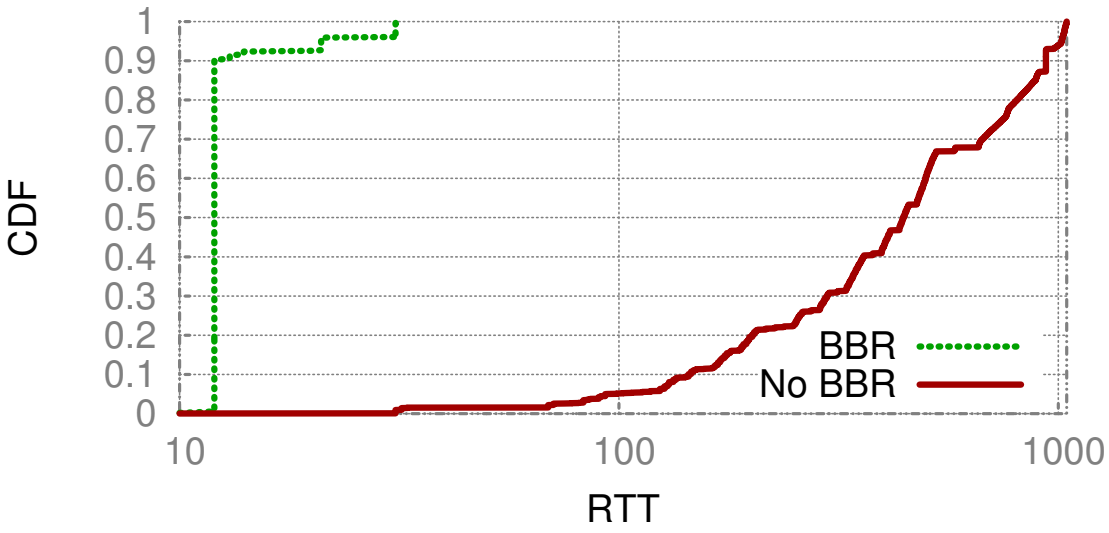



Figure 6.2: Cumulative distribution function plot of the observed RTT during a 20-second TCP transfer shows BBR produces drastically lower end-to-end latency.

bufferbloat.

### 6.1.2 Video Streaming

Multimedia applications such as video and audio streaming pose an interesting challenge for TCP because they use more bandwidth than HTTP transfers carrying text or static content yet they are usually interactive in nature. Furthermore, adaptive bit-rate encoding schemes can dynamically vary the resolution of the media based on available bandwidth but require minimal end-to-end latency to respond quickly enough. The user experience of such interactive applications relies on keeping the round-trip delay as close to the minimum as possible.

Video streaming represents a scenario where BBR is particularly useful because of the tendency for the application to be the only active flow on the bottleneck link. Intuitively, if a viewer is actually watching the video, he or she may not have other simultaneous flows. Though perhaps too simplistic an explanation, BBR's performance results in this scenario show that it does not suffer (i.e., lose throughput)

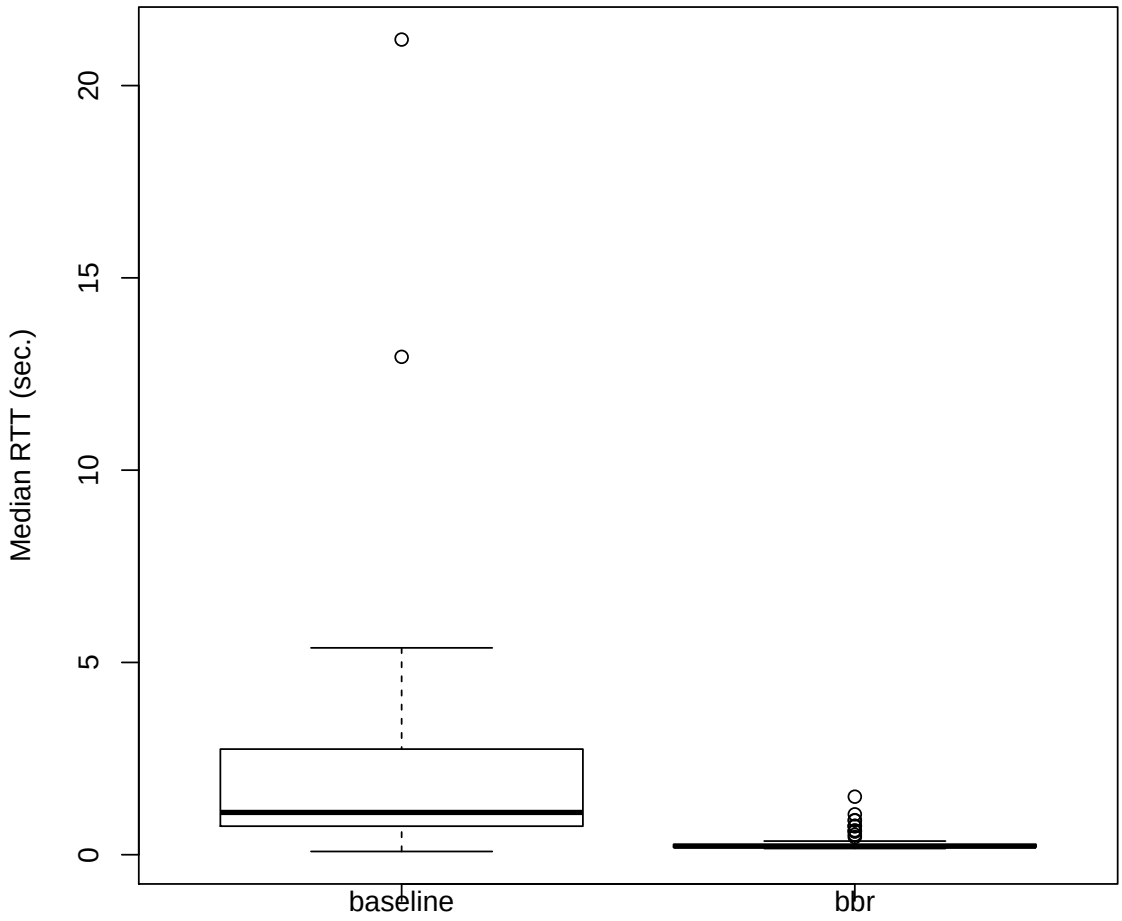



Figure 6.3: Boxplots comparing end-to-end latency for BBR video streams versus baseline TCP show BBR's significantly improved average latency and reduced variance.

from competing traffic as it does in simulations.

To evaluate BBR in the video streaming scenario, we deployed it in a live production video streaming platform and compared its performance to the baseline of TCP CUBIC. The video platform serves both mobile and desktop users alike. We used TCP traces of each session to measure the end-to-end latency, the bandwidth efficiency and the rate of retransmissions. We present results from this experiment taken over a single day comparing 56 BBR sessions to 126 baseline TCP CUBIC sessions with equivalent bandwidth and application conditions.

**End-to-End Latency** - Figure 6.3 shows boxplots of the end-to-end latency observed by the video server for connections using BBR and the regular baseline TCP CUBIC. The average, mean and quartiles are given after removing outliers, though outliers are marked by circles. The average latency for the baseline TCP was approximately 2 seconds compared to 0.34 for BBR, which constitutes roughly a $6x$ reduction. Median latency for BBR was $5x$ smaller at 0.22 seconds versus 1.1 seconds for baseline TCP.

Anecdotally interesting are the two baseline TCP flows that experienced extreme median latency of 13 and 21 seconds! BBR had none such extreme outliers, with all sessions observing median latency well below 3 seconds. Additionally, the inter-quartile range for BBR was a much smaller 0.06 seconds compared to the baseline's 2 seconds. The plot indicates that BBR's throttling of the sending rate reduces the median latency variance significantly compared to the baseline TCP.

**Bandwidth Efficiency** - We next evaluate BBR in the video streaming scenario based on its bandwidth efficiency compared to the baseline TCP CUBIC. Bandwidth efficiency is defined as application-level throughput, or "goodput", divided by the bottleneck link's bandwidth. For these traces, the bottleneck bandwidth was 1Mbps.

Figure 6.4 presents a comparison of BBR streams versus the baseline TCP CUBIC in terms of bandwidth efficiency. Once again, BBR exhibits a much smaller variance than the baseline TCP, never delivering below roughly 85% for any one connection. On the other hand, baseline TCP has extreme outliers with one as low as 20%. More importantly, baseline TCP's range of values from minimum to maximum (after removing outliers) spans from roughly 30% to 100%, while BBR's is a much tighter 85% to 100%.

The mean and median bandwidth efficiency for BBR were both 95%. Furthermore, BBR's inter-quartile range was 4%, which means BBR delivered bandwidth

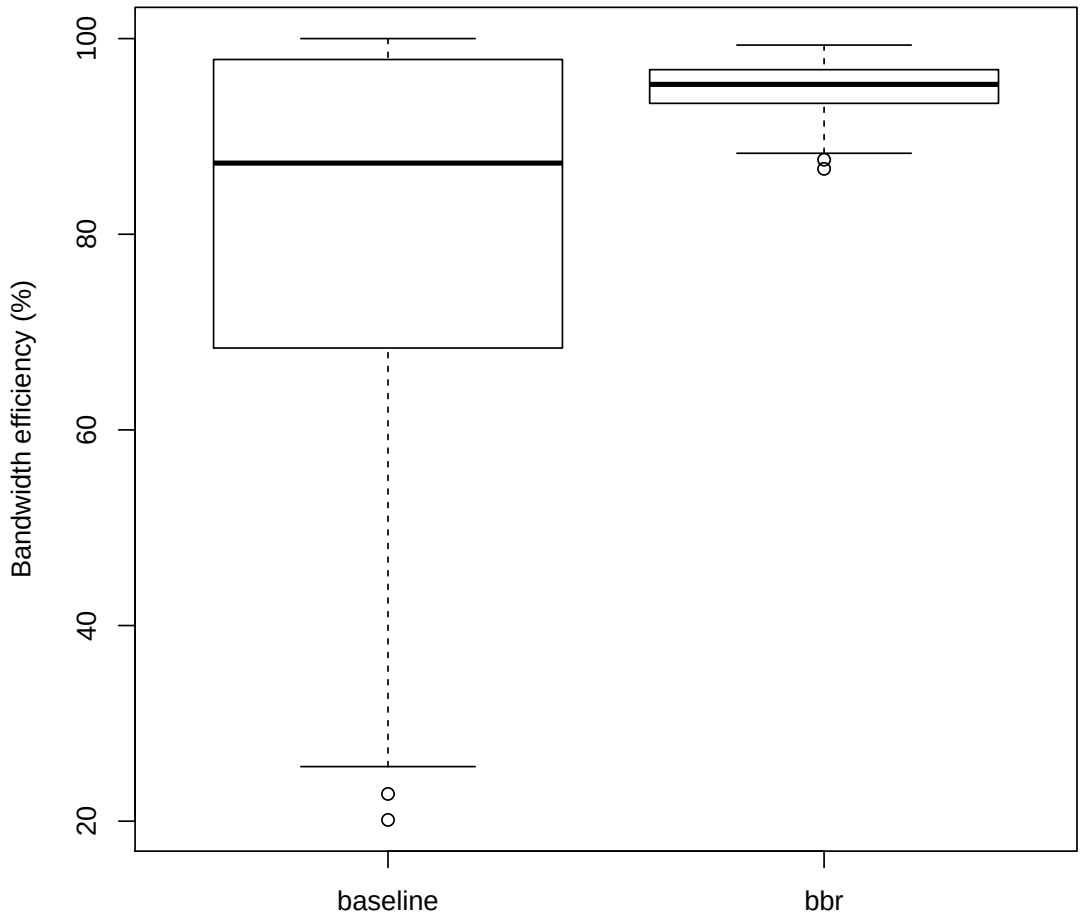



Figure 6.4: Boxplots comparing bandwidth efficiency for BBR video streams versus baseline TCP show BBR results in higher average bandwidth efficiency and reduced variance.

efficiency at or above 93% for three quarters of all sessions. When compared with the baseline, these results represent noticeable improvements in user experience. Baseline TCP averaged 80% bandwidth efficiency for comparable sessions with a median of 87%. Its inter-quartile range was 30%, nearly $7x$ larger than BBR's observed variance in bandwidth efficiency.

**Retransmissions** - We now evaluate BBR versus the baseline TCP in terms of retransmissions due to lost packets. Recall that the baseline TCP CUBIC implementation continues to increase its sending rate until there is a lost packet. In the single flow scenario it is possible for this behavior to be the sole cause of dropped packets and retransmissions. On the other hand, by limiting the sending rate to the estimate of the bottleneck bandwidth, BBR effectively eliminates the possibility of a full queue causing packet drops.

Figure 6.5 shows boxplots of the retransmission rate for video streams using BBR versus the baseline TCP. The BBR sessions experience almost no retransmissions for a majority of flows as the median retransmit rate is 0. The BBR retransmission rate contrasts sharply with the baseline's median rate of 2.7% of packets, representing orders of magnitude improvement for BBR. The average retransmit rate for BBR of 0.2% similarly is orders of magnitude better than the baseline TCP average rate of 13%.

It is perhaps counter-intuitive that even without competition the baseline TCP sessions experience such a high retransmit rate. However, because CUBIC only responds to losses, it must *initiate* losses by transmitting beyond the path's capacity. Furthermore, once the queue is full it is likely that the bottleneck router drops a burst of consecutive packets before the sender becomes aware of the losses. The subsequent recovery and repetition produces the familiar "sawtooth" pattern of TCP's sending rate.

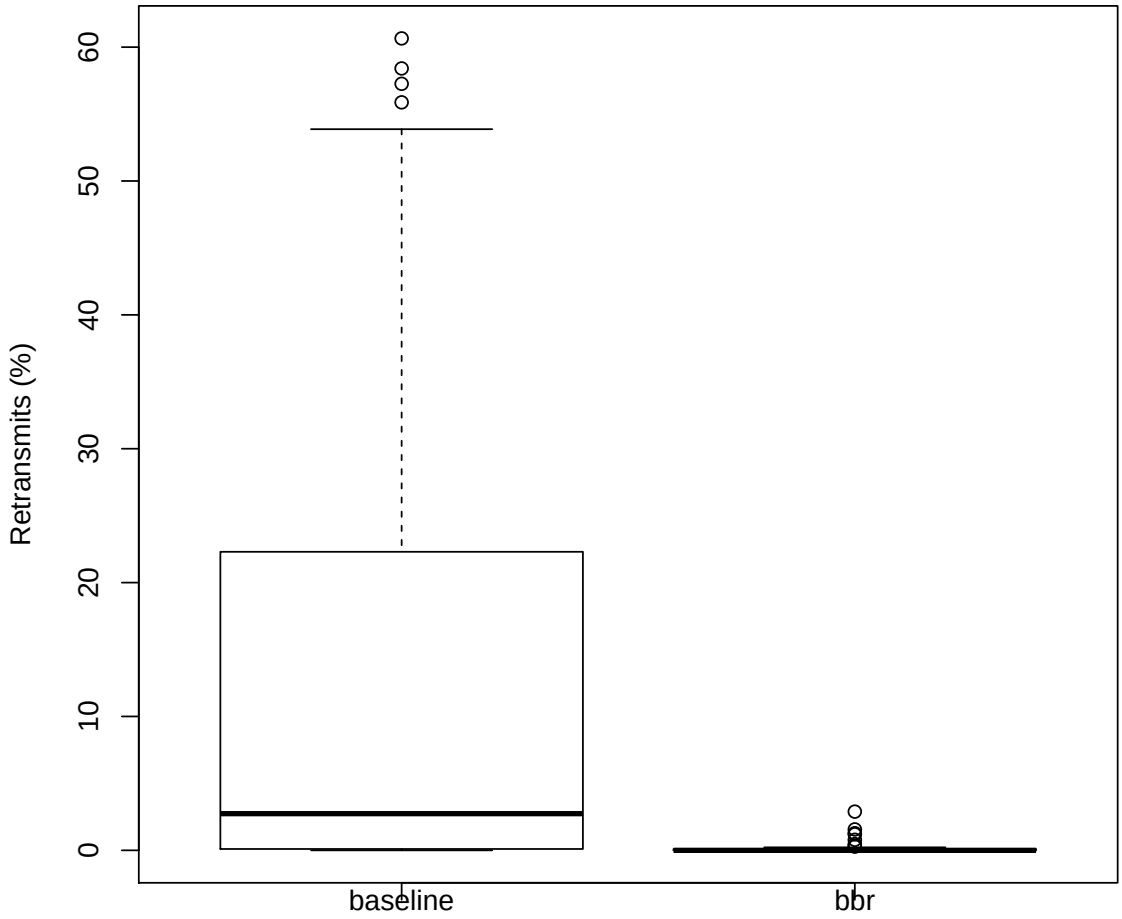



Figure 6.5: Boxplots comparing the retransmit rate for BBR video streams versus baseline TCP show almost no dropped packets or retransmissions on most BBR flows.

Remaining consistent with the previous two performance metrics, BBR drastically reduces the variance compared to the baseline. BBR's inter-quartile range of 0.1% is an improvement of approximately $220x$ over the baseline TCP's range of 22%. As previously stated, these video streaming sessions often appear to be the only flow on the bottleneck path, which means BBR's reduced sending rate achieves compelling improvements.

Due to the observation window of the current LLS implementation, the 1KB per-socket memory overhead is not negligible. However, the live production servers showed no negative impact due to the additional state. Furthermore, the reduced retransmissions and increased bandwidth efficiency in the experiments suggest that the overhead is worth it in video streaming scenarios.

## 6.2 A Minion Receiver

We now evaluate Minion's receiver-side enhancements via experiments designed to approximate realistic application scenarios. All experiments were run across three Intel PCs running Linux 2.6.34: between two machines representing end hosts, a third machine interposes on the path and uses dummynet [10] to emulate various network conditions. To minimize well-known TCP delays fairly for both TCP and *u*TCP, we enabled Linux's "low latency" TCP code path via the `net.ipv4.tcp_low_latency` sysctl, and disabled Nagle's algorithm.

### 6.2.1 CPU Costs

We first explore *u*TCP's costs, with and without record encoding and extraction via *u*COBS and *u*TLS, for a 30MB bulk transfer on a path with 60ms RTT for several loss rates.

Figure 6.6(a) shows CPU costs including application-level encoding/decoding, atop standard TCP ("COBS") and atop $u$TCP ("$u$COBS"), for several loss rates at both sender and receiver. The lighter part of each bar represents user time and the darker part represents kernel time. These results are normalized to the performance of raw TCP, with no application-level encoding or decoding.

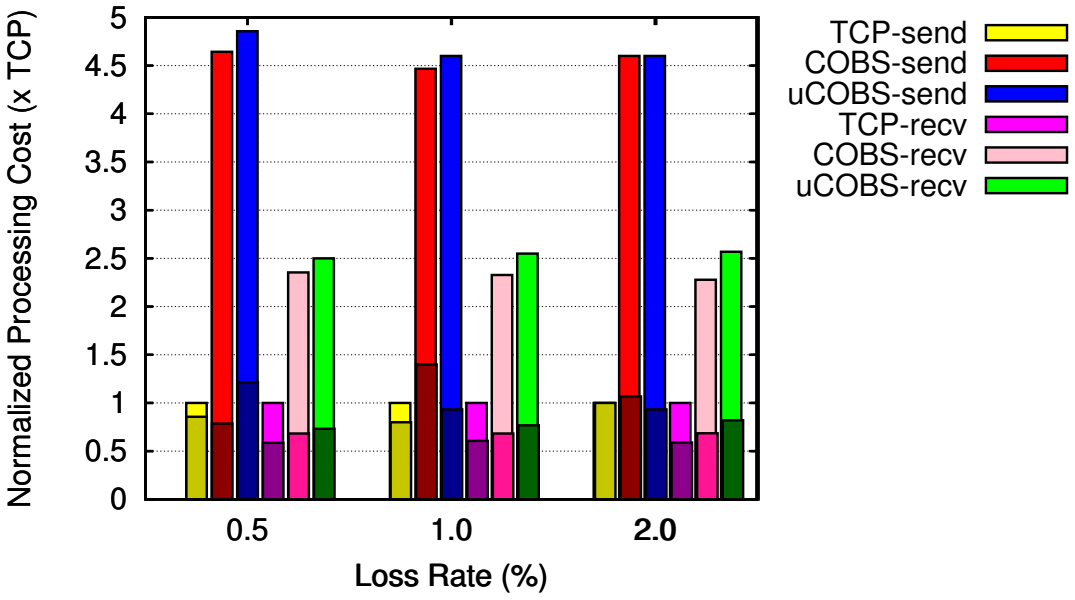
COBS encoding/decoding barely affects kernel CPU use but incurs some application-level CPU cost. This cost is partly due to the encoding itself, and partly because the libraries are not yet well-optimized.

Figure 6.6(b) shows the CPU costs of $u$TLS relative to TLS. At the sender, the CPU costs are identical, since there is nothing that $u$TLS does differently than TLS, and since the CPU cost of using $u$TCP is practically the same as with TCP. The user-space cost for the $u$TLS receiver is generally higher than TLS, since the $u$TLS receiver does more work in processing out-of-order frames than the TLS receiver, but this cost remains within 7% of the TLS receiver's cost.

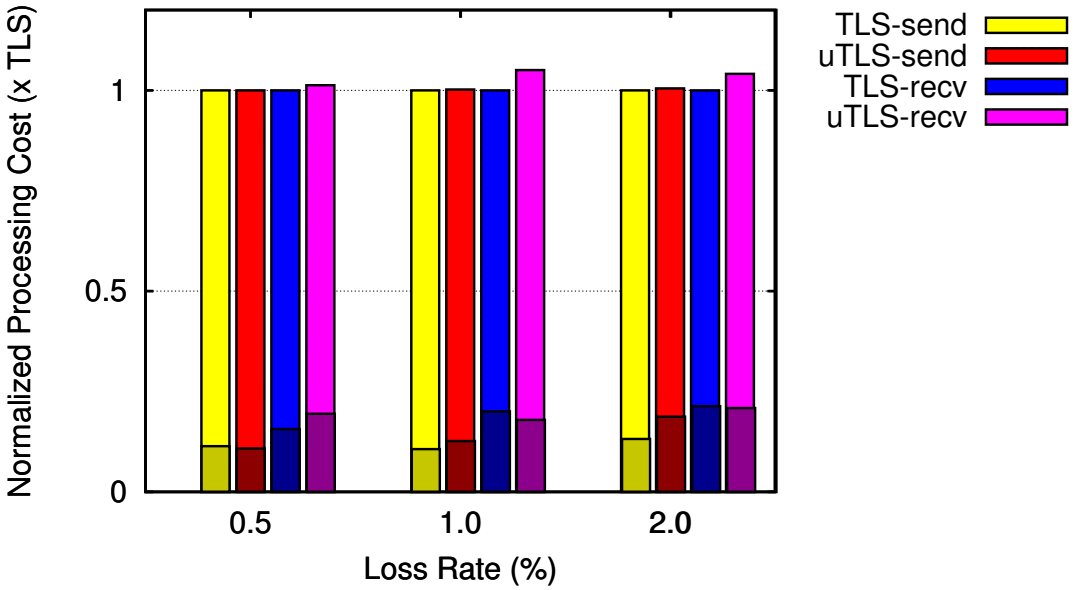
The bandwidth penalty of $u$COBS encoding is barely perceptible, under 1%. TLS's bandwidth overhead, up to 10%, is due to TLS headers, IVs, and MACs; $u$TLS adds no bandwidth overhead beyond standard TLS 1.1.

### 6.2.2 Conferencing Applications

We now examine a real-time Voice-over-IP (VoIP) scenario. A test application uses the SPEEX codec [57] to encode a WAV file using ultra-wideband mode (32kHz), for a 256kbps average bit-rate, and transmit voice frames at fixed 20ms intervals. Network bandwidth is 3Mbps and RTT is 60ms, realistic for a home broadband connection. To generate losses more realistically representing network contention, we run a varying number of competing TCP file transfers, emulating concurrent web browsing sessions or a BitTorrent download, for example.




(a) COBS/uCOBS encoding costs




(b) TLS/uTLS costs

Figure 6.6: CPU costs of using an application with TCP, COBS, and uCOBS at different loss rates.

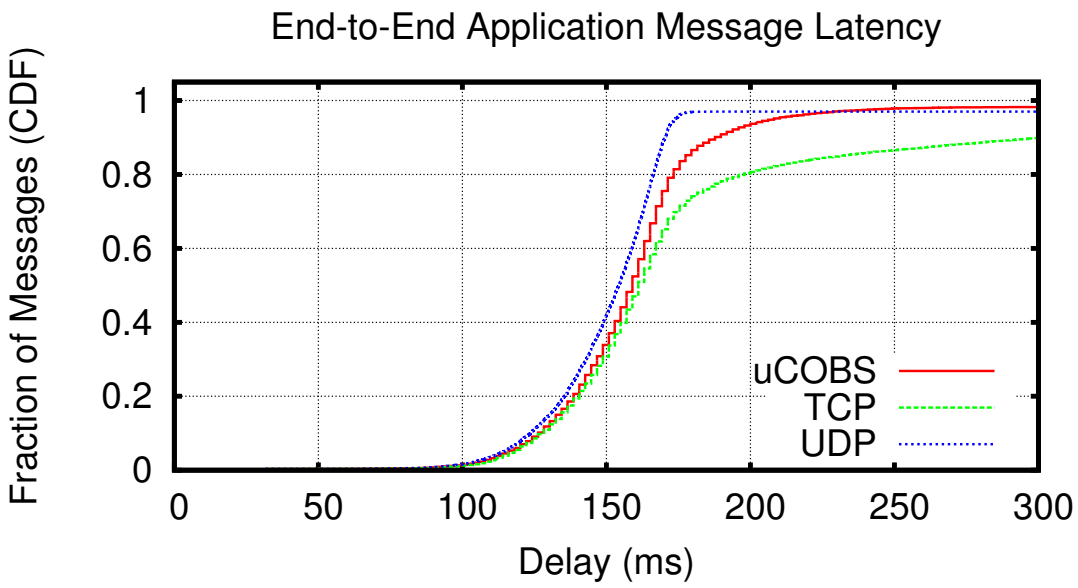



Figure 6.7: CDF of end-to-end latency in VoIP frames.

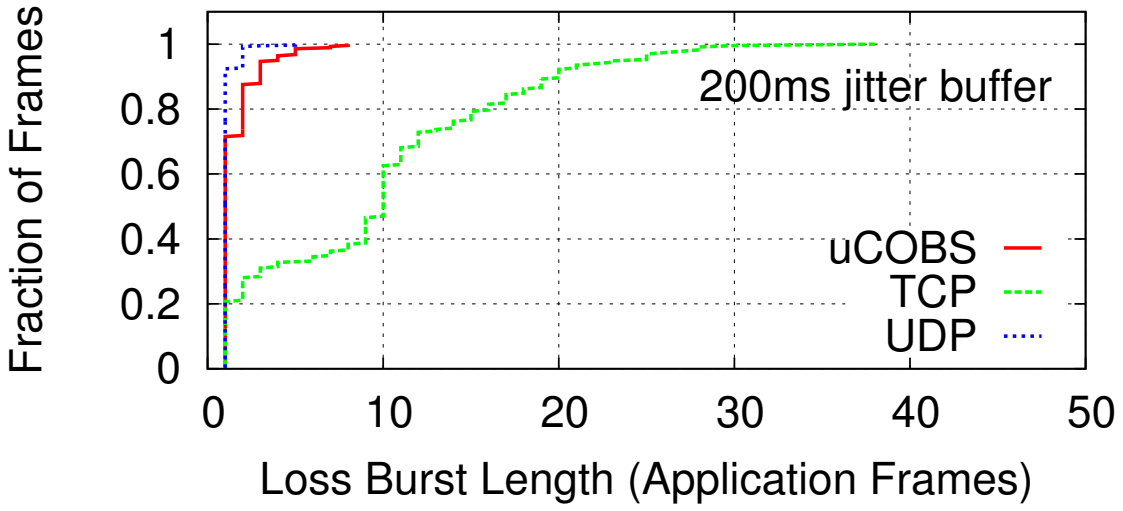



Figure 6.8: CDF of codec-perceived loss-burst size with TLV encoded frames over TCP, UDP, and uCOBS.

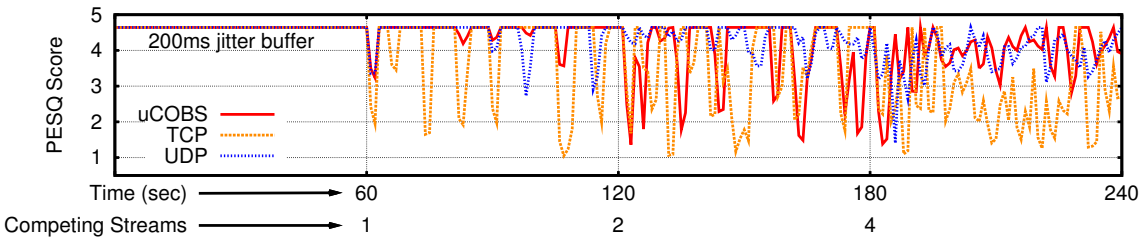



Figure 6.9: Moving PESQ score of VoIP call under increasing bandwidth competition.

This is a simplistic scenario for experimental purposes. Real VoIP applications, which we intend to evaluate in future work, often determine bit-rate based on network conditions. Real applications may also implement loss recovery mechanisms atop UDP, which may improve perceived voice quality when using UDP.

**Latency:** Figure 6.7 shows a CDF of one-way per-frame latency perceived by the receiving application, under heavy contention from 4 competing TCP streams. All three transports suffer major delays. 4% of UDP frames do not arrive at all, since UDP does not retransmit. 95% of frames sent with *u*COBS over *u*TCP arrive within 200ms, compared to 80% of TCP frames.

**Burst Losses:** VoIP codecs such as SPEEX can interpolate across one or two missing frames, but are sensitive to burst losses or delays, which yield user-perceptible blackouts. An application's susceptibility to blackouts depends on its jitter buffer size: a larger buffer increases the receiver's tolerance of burst losses or delays, but also increases effective round-trip delay, which can add user-perceptible "lag" to all interactions.

The CDF in Figure 6.8 shows the prevalence of different lengths of burst losses experienced by the receiver in a typical VoIP call. A burst loss is a series of consecutive voice frames that miss their designated playout time, due either to loss or delay.

A 200ms jitter buffer of 3× the path RTT might seem generous, but the ITU's recommended maximum transmission time of 400ms [5] allows for a larger buffer with these network conditions. Now the differences between $u$COBS and TCP are quite pronounced, with 80% of burst losses atop $u$COBS being 3 or fewer packets, nearly matching that of UDP. Meanwhile 40% of TCP's bursts are greater than 10 packets, producing highly-perceptible 1/5-second pauses.

**Perceptual Audio Quality:** To illustrate the impact of unordered delivery on VoIP quality, we use Perceptual Evaluation of Speech Quality (PESQ) [42] to measure audio reproduction quality, by comparing the audio stream reproduced by SPEEX at the receiver against that of an ideal run with no lost or delayed frames. This experiment uses the freely available International Telecommunication Union (ITU) reference implementation [41] for 862.2 PESQ score. PESQ scores range from roughly 1 (degraded audio is inaudible) to 4.644 (perfect match, no degradation).

We transmit a 4 minute VoIP call using the same 60ms RTT and 3Mbps link as before, with two jitter buffer sizes. Competing streams begin at 1-minute intervals, eventually causing significant degradation of audio quality with 4 competing streams.

We compare the received audio files to an "ideal" audio file with no dropped frames. Each data point represents the PESQ score for the next 2 seconds, representing a reasonable measure of the momentary audio quality.

Our testing shows that audio quality degrades immediately once competing streams begin. Our goal is to show the user experience throughout the call as competing streams choke out the bandwidth. In such a scenario, we expect to see the audio quality rebound after the initial collision with competing TCP streams, which themselves back off due to their losses.

Figure 6.9 plots PESQ quality scores for 2-second sliding time windows over a rep-

resentative 4-minute call, comparing transmission via $u$COBS, TCP, and UDP. The effect of network contention becomes apparent even with only one competing stream, but unordered delivery makes this impact much smaller on $u$COBS or UDP than on TCP. $u$COBS sometimes performs better than UDP, in fact, when $u$TCP successfully retransmits a lost segment within the jitter buffer's time window, whereas UDP never retransmits. (Some UDP applications employ application-level retransmission schemes [3], especially for control data.) Like TCP, $u$COBS shows greater volatility than UDP with higher contention, due to TCP congestion control effects that $u$TCP preserves (though congestion control can be disabled). Similarly, the "back-off" of the *competing* streams enables the transports to rebound after the initial contention of 4 competing streams.

We see that even 1 competing stream causes significant drops for TCP, while $u$COBS and UDP exhibit only minor dips in quality. In the third minute of the call (2 competing streams), $u$COBS shows volatility compared to UDP. We attribute this to the congestion control of $u$COBS and TCP. The competing streams themselves use TCP's back-off mechanism, interleaving windows of high and low bandwidth for the VoIP call. $u$COBS still suffers from this less than TCP, however, due to its ability to deliver data out-of-order. Meanwhile, TCP's decreasing fidelity beyond 1 competing stream makes it potentially a non-option under any bandwidth competition.

Perhaps the most important feature shown by these experiments is the effect of the jitter size on $u$COBS, and the lack of effect for TCP. Increasing the jitter buffer from 150 to 200ms, particularly under 4 competing streams, improves $u$COBS significantly more than it improves TCP. This speaks to the fundamental difference between the two highlighted in Figure 6.8: that TCP's bursty losses are ill-suited to real-time applications. Because a single lost packet delays subsequent packets, repeatedly dropped packets under heavy bandwidth contention are helpless with

any reasonable jitter buffer size. Breaking the in-order requirement of delivery is essential to transmission quality in these situations, making *u*COBS more suitable for real-time applications.

### 6.2.3 Multistreaming Web Transfers

We now explore building concurrency atop *u*TCP's unordered message service; we build a *multistreaming* abstraction that provides multiple independent and ordered message streams *within* a single *u*TCP connection. While both the need for and the benefits of concurrency at the transport layer have been well known [23, 26, 38, 53], to our knowledge, this is the first time that true multistreaming has been built using TCP.

Our prototype implementation of a multistreaming library in user-space atop *u*TCP, which we call *Multistreamed TCP* or *ms*TCP, uses a 16-bit stream identifier, and a 16-bit stream sequence number to order chunks within a stream. An application specifies a stream identifier when using *ms*TCP's `ms_sendmsg()` API; *ms*TCP then breaks every application message down into a series of *chunks*—the *ms*TCP-defined smallest multiplexable unit of data within a connection. Chunks are prepended with a 16-byte chunk header, which includes the stream identifier for the message and the stream sequence number, and then finally transmitted using *u*COBS. At the receiver, *ms*TCP's `ms_recvmsg()` receives unordered chunks from the underlying *u*COBS, parses the chunk header, and adds each chunk to its stream's *stream-queue*—`ms_recvmsg()` maintains one queue for each stream identifier that it encounters in the received chunks—the chunks are then delivered to the application in-order within their respective streams. Note that *ms*TCP's API can be trivially extended to using *u*TLS as well.

We use the Web as an example application to illustrate the behavior of *ms*TCP.

HTTP/1.1 addressed the inefficiency of short-lived TCP streams through pipelining over persistent connections. Since *ms*TCP attempts to offer the benefits of persistent streams with the simplicity of the one-transaction-per-stream model. We now compare the performance of HTTP/1.0 with parallel object requests over *ms*TCP against the behavior of pipelined HTTP/1.1 over a persistent TCP connection, under a simulated web workload.

For this test we simulate a series of web page loads, each page consisting of a "primary" HTTP request for the HTML, followed by a batch of "secondary" requests for embedded objects such as images. As the simulation's workload we use a fragment of the UC Berkeley Home IP web client traces available from the Internet Traffic Archive [32]. We sort the trace by client IP address so that each user's activities are contiguous, then we use only the order and sizes of requests to drive the simulation, ignoring time stamps. Since the traces do not indicate which requests belong to one web page, the simulation approximates this information by classifying requests by extension into "primary" (e.g., '`.html`' or no extension) and "secondary" (e.g., '`gif`', '`.jpg`', '`.class`'), and then associating each contiguous run of secondary requests with the immediately preceding primary request. The simulation pessimistically assumes that the browser cannot begin requesting secondary objects until it has downloaded the primary object completely, but at this point it can in theory request all of the secondary objects in parallel. The experimental setup uses a link with 1.5Mbps bandwidth in each direction and with a 60ms RTT, typical of browsing over a residential connection.

Figure 6.10 shows a scatter-plot of total time to load the entire page in the top three graphs, and the bottom three graphs show the average time to load the first byte of an object within each page—the expected time for an object to start being rendered at the browser. The X-axis represents total webpage size. The dark curves

show median times, computed across webpages in log-sized buckets of webpage size—the solid curve shows the median for HTTP/1.1 over TCP and the dashed curve for HTTP/1.0 over multistreaming.

The total time to transfer the pages remains the same; *ms*TCP's bit-overheads do not affect application-observed throughput noticeably. *ms*TCP however, shows much lower latency in loading the first byte of objects, since the objects' chunks are interleaved within the persistent connection by *ms*TCP.

We note that the results in Figure 6.10 show the end-to-end impact on web browsing of *ms*TCP's application-level message chunking and multiplexing in addition to the benefits of *u*TCP's out-of-order delivery. These latency savings thus represent the potential savings when web frameworks like SPDY [4] use *u*TCP, and make HTTP/HTTPS more usable as a general purpose substrate for deploying latency-sensitive applications [45].

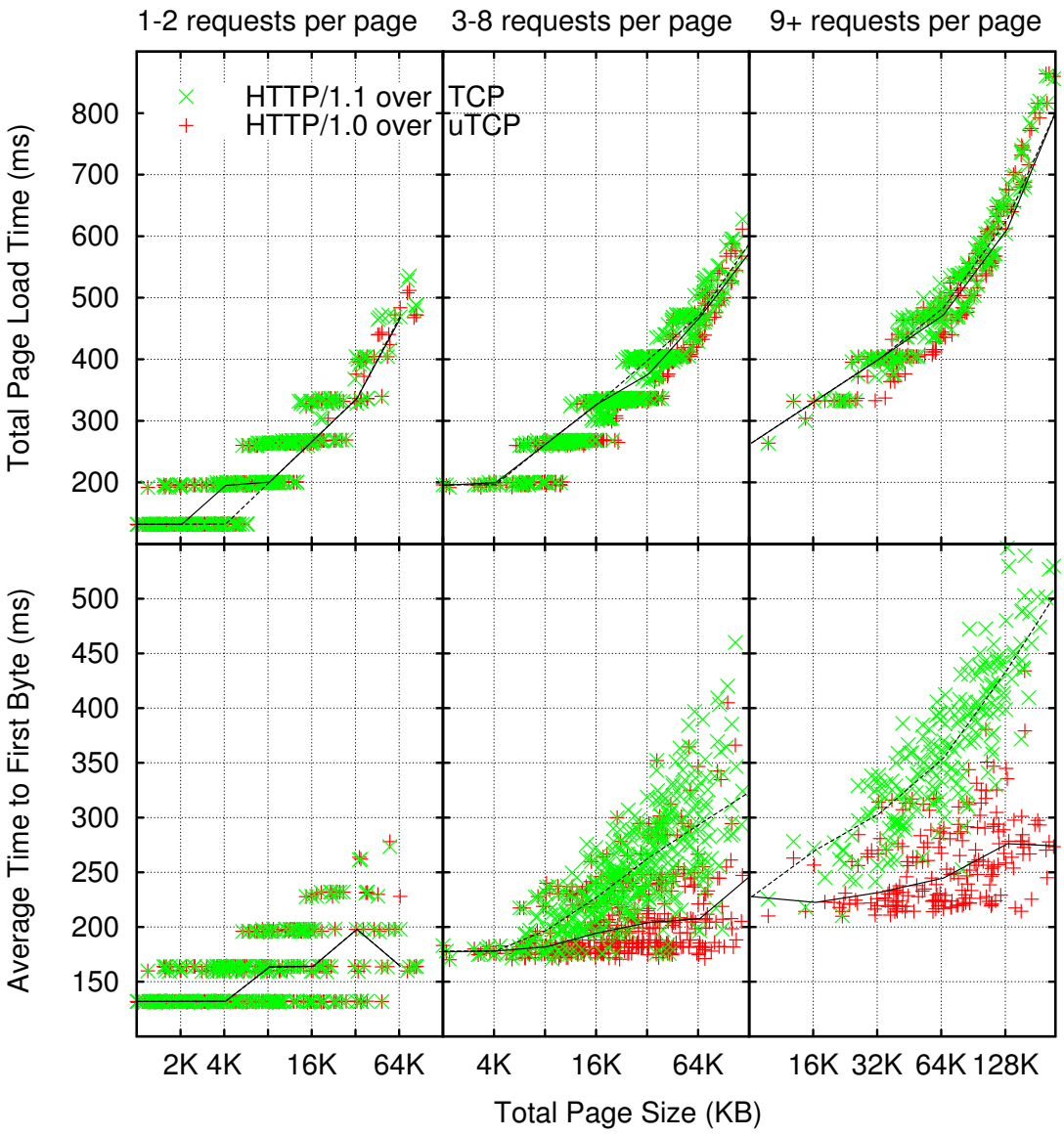



Figure 6.10: Pipelined HTTP/1.1 over a persistent TCP connection, vs. Parallel HTTP/1.0 over $ms$TCP.

# Chapter 7

# Conclusion and Future Work

For better or worse, TCP remains the most common substrate for application-level protocols and frameworks, many of which require low end-to-end latency for acceptable user experience. In this work, we have presented two small changes to TCP's implementation that reduce end-to-end latency without changing TCP's on-the-wire format. By maintaining TCP's existing wire format, both changes are incrementally deployable and backward-compatible with endpoints and network middleboxes that do not support the features. While both changes improve performance in certain circumstances, future work remains to show that both BBR and *u*TCP are safe for broad deployment in nearly all application scenarios.

Based on our experimental evaluations, the LLS implementation accurately identifies instances of bufferbloat when the flow has no competition. This is consistent with BBR's goal to reduce self-inflicted increased latency. However, during periods of competition at the bottleneck link, a flow's goal of low end-to-end latency changes and becomes instead a goal of maximizing throughput. It is this second goal that BBR struggles with during competition, though it is a problem that plagues TCP CUBIC itself.

If BBR is already throttling a sender and a new CUBIC flow appears, the CUBIC flow quickly fills up the bottleneck queue and takes most of the throughput. BBR's first challenge is detecting the presence of a competing flow through a decrease in correlation. We observe in testing that the correlation does, in fact, fall below the required threshold but by this time, the competing CUBIC flow's sending rate vastly exceeds BBR's rate.

TCP's inherent *unfairness* is the more fundamental issue. Even if BBR is not throttling a sender (and behaves as regular CUBIC), if its transmission starts after an existing CUBIC flow, it will never ramp up its sending rate to share the throughput equally. Thus, this is not unique to BBR- when two CUBIC flows compete, generally whichever flow starts first quickly amplifies its slightly larger sending rate into an extreme disparity. TCP's autocatalytic nature causes a small increase in sending rate to perpetuate a larger increase, and so on.

Addressing this inherent unfairness is a steep challenge for any congestion control scheme as it must simultaneously act greedily against the incumbent CUBIC while also trying to supplant it with a more conservative approach. Our intuition is that BBR should be able to achieve this balance but we leave this to future work.

Future work on Minion should attempt to build out more application-level support for *u*TCP and unordered delivery. However, adoption of Minion will always be limited by the widespread application assumption that TCP delivers strictly in-order. Despite all standardized OS-level transports since TCP, including UDP [46], RDP [58], DCCP [35], and SCTP [53], supporting out-of-order delivery, applications continue to rely upon TCP and its semantics. Furthermore, the Internet's evolution has created strong barriers [24, 45, 50] against the widespread deployment of new transports other than the original TCP and UDP.

In conclusion, TCP and the side effects of its design continue to affect the user

experience for low-latency applications, if not their choice in transport. Rather than fight against TCP's cultural inertia, this work instead takes the approach that minor endpoint-only changes can help TCP remain a valuable transport substrate for current and future low-latency applications.

# Bibliography


[1] Akamai technologies. `http://www.akamai.com`.

[2] ØMQ: The intelligent transport layer. `http://www.zeromq.org`.

[3] OpenArena project. `http://openarena.ws/`.

[4] SPDY: An experimental protocol for a faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[5] ITU. Recommendation G.114: One-way transmission time, May 2003.

[6] F. Audet, ed. and C. Jennings. Network address translation (NAT) behavioral requirements for unicast UDP, January 2007. RFC 4787.

[7] Salman A. Baset and Henning Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *IEEE INFOCOM*, April 2006.

[8] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[9] Eli Brosh, Salman Abdul Baset, Vishal Misra, Dan Rubenstein, and Henning Schulzrinne. The delay-friendliness of TCP for real-time traffic. *IEEE Transactions on Networking*, 18(5):1478–1491, 2010.

[10] M. Carbone and L. Rizzo. Dummynet Revisited. *ACM CCR*, 40(2), April 2010.

[11] Jaime S. Cardoso. Bandwidth-efficient byte stuffing. In *IEEE ICC 2007*, 2007.

[12] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and Issues, February 2002. RFC 3234.

[13] Stuart Cheshire and Mary Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, September 1997.

[14] Cisco. Rate-Based Satellite Control Protocol, 2006.

[15] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM*, pages 200–208, 1990.

[16] Al Danial. Counting Lines of Code, ver. 1.53. `http://cloc.sourceforge.net/`.

[17] Joseph Davies. DirectAccess and the thin edge network. *Microsoft TechNet Magazine*, May 2009.

[18] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, August 2008. RFC 5246.

[19] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for TCP. In *Internet Measurement Conference (IMC)*, November 2011.

[20] R. Fielding et al. Hypertext transfer protocol – HTTP/1.1, June 1999. RFC 2616.

[21] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh

Govindan. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM*, August 2013.

[22] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *Transactions on Networking*, 1(4):1063–6692, August 1993.

[23] Bryan Ford. Structured streams: a new transport abstraction. In *ACM SIGCOMM*, August 2007.

[24] Bryan Ford and Janardhan Iyengar. Breaking up the transport logjam. In *7th Workshop on Hot Topics in Networks (HotNets-VII)*, October 2008.

[25] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *ACM queue*, November 2011. http://queue.acm.org/detail.cfm?id=2071893.

[26] Jim Gettys and Henrik Frystyk Nielsen. SMUX Protocol Specification, July 1998. `http://www.w3.org/TR/WD-mux`.

[27] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate limiting YouTube video streaming. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2012.

[28] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT behavioral requirements for TCP, October 2008. RFC 5382.

[29] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, Oliver Spatscheck, and Xiaodong Zhang. Delving into Internet streaming media delivery: a quality and resource utilization perspective. In *Internet Measurement Conference (IMC)*, October 2006.

[30] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *Operating Systems Review*, 42(5), July 2008.

[31] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend TCP? In *Internet Measurement Conference*, November 2011.

[32] The Internet traffic archive. `http://ita.ee.lbl.gov/`.

[33] Van Jacobson. Congestion avoidance and control. pages 314–329, August 1988.

[34] S. Kent and K. Seo. Security architecture for the Internet protocol, December 2005. RFC 4301.

[35] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), March 2006. RFC 4340.

[36] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: Congestion control without reliability. In *ACM SIGCOMM (SIGCOMM)*, 2006.

[37] Jeff Mogul. TCP offload is a dumb idea whose time has come. In *HotOS IX*, May 2003.

[38] Preethi Natarajan et al. SCTP: An innovative transport layer protocol for the Web. In *15th World Wide Web Conference (WWW)*, May 2006.

[39] Kathleen Nichols and Van Jacobson. Controlling queue delay. *ACM queue*, May 2012. http://queue.acm.org/detail.cfm?id=2209336.

[40] Michael F. Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. Fitting square pegs through round pipes: Unordered delivery wire-compatible with TCP and TLS. April 2012.

[41] ITU-T Telecommunication Standardization Sector of ITU. Recommendation p.862 (2001) amendment 2 (11/05) reference implementation and conformance

testing, November 2005. http://www.itu.int/rec/T-REC-P.862-200511-I!Amd2/en.

[42] ITU-T Telecommunication Standardization Sector of ITU. Wideband extension to recommendation p.862 for the assessment of wideband telephone networks and speech codecs, November 2007.

[43] OpenSSL project. http://www.openssl.org/.

[44] T. Phelan. DCCP Encapsulation in UDP for NAT Traversal (DCCP-UDP), August 2010. Internet-Draft draft-ietf-dccp-udpencap-02 (Work in Progress).

[45] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP as the narrow waist of the future Internet. In *9th ACM Workshop on Hot Topics in Networks (HotNets-IX)*, October 2010.

[46] J. Postel. User datagram protocol, August 1980. RFC 768.

[47] QUIC: Multiplexed stream transport over UDP. http://blog.chromium.org/2013/06/experimenting-with-quic.html.

[48] Charles Reis et al. Detecting in-flight page changes with web tripwires. In *5th Symposium on Networked System Design and Implementation (NSDI)*, April 2008.

[49] E. Rescorla and N. Modadugu. Datagram transport layer security, April 2006. RFC 4347.

[50] J. Rosenberg. UDP and TCP as the new waist of the Internet hourglass, February 2008. Internet-Draft (Work in Progress).

[51] J. Rosenberg et al. SIP: session initiation protocol, June 2002. RFC 3261.

[52] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI), April 2004. RFC 3720.

[53] R. Stewart, ed. Stream control transmission protocol, September 2007. RFC 4960.

[54] Transmission control protocol, September 1981. RFC 793.

[55] Wesley W. Terpstra, Christof Leng, Max Lehn, and Alejandro P. Buchmann. Channel-based unidirectional stream protocol (CUSP). In *IEEE INFOCOM Mini Conference*, March 2010.

[56] Michael Tuexen and Randall Stewart. UDP Encapsulation of SCTP Packets, January 2010. Internet-Draft draft-tuexen-sctp-udp-encaps-05 (Work in Progress).

[57] Jean-Marc Valin. The speex codec manual version 1.2 beta 3, December 2007. `http://www.speex.org/`.

[58] David Velten, Robert Hinden, and Jack Sax. Reliable data protocol, July 1984. RFC 908.

[59] Nevena Vratonjic, Julien Freudiger, and Jean-Pierre Hubaux. Integrity of the web content: The case of online advertising. In *Workshop on Collaborative Methods for Security and Privacy*, August 2010.

[60] W3C. The WebSocket API (draft), 2011. `http://dev.w3.org/html5/websockets/`.

[61] David X. Wei et al. FAST TCP: Motivation, architecture, algorithms, performance. *Transactions on Networking*, 14(6), December 2006.

[62] Eric W. Weisstein. Correlation coefficient. http://mathworld.wolfram.com/CorrelationCoefficient.html.

[63] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated congestion control. In *ACM SIGCOMM*, August 2013.

[64] Marko Zec, Miljenko Mikuc, and Mario Zagar. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. In *SoftCOM*, 2002.