Reducing Web Latency: the Virtue of Gentle Aggression

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

[*]Department of Computer Science, University of Southern California

[†]Google Inc.

ABSTRACT

To serve users quickly, Web service providers build infrastructure closer to clients and use multi-stage transport connections. Although these changes reduce client-perceived round-trip times, TCP's current mechanisms fundamentally limit latency improvements. We performed a measurement study of a large Web service provider and found that, while connections with no loss complete close to the ideal latency of one round-trip time, TCP's timeout-driven recovery causes transfers with loss to take five times longer on average.

In this paper, we present the design of novel loss recovery mechanisms for TCP that judiciously use redundant transmissions to minimize timeout-driven recovery. *Proactive*, *Reactive*, and *Corrective* are three qualitatively-different, easily-deployable mechanisms that (1) proactively recover from losses, (2) recover from them as quickly as possible, and (3) reconstruct packets to mask loss. Crucially, the mechanisms are compatible both with middleboxes and with TCP's existing congestion control and loss recovery. Our large-scale experiments on Google's production network that serves billions of flows demonstrate a 23% decrease in the mean and 47% in 99th percentile latency over today's TCP.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*TCP*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Standards*; C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes

Keywords

TCP; Congestion Control; Web Latency; Internet Measurements; Packet Loss; Redundancy; Recovery

1. INTRODUCTION

Over the past few years, and especially with the mobile revolution, much economic and social activity has moved online. As such, user-perceived Web performance is now *the* primary metric for modern network services. Since bandwidth remains relatively cheap, Web latency is now the main impediment to improving user-perceived performance. Moreover, it is well known that Web latency inversely correlates with revenue and profit. For instance, Amazon estimates that every 100ms increase in latency cuts profits by 1% [26].

In response to these factors, some large Web service providers have made major structural changes to their service delivery infrastructure. These changes include a) expanding their backbones and PoPs to achieve proximity to their clients and b) careful re-engineering of routing and DNS redirection. As such, these service providers are able to ensure that clients quickly reach the nearest ingress point, thereby minimizing the extent to which the client traffic traverses the public Internet, over which providers have little control. To improve latency, providers engineer the capacity of and traffic over their internal backbones. As a final latency optimization, providers use *multi-stage* TCP connections to isolate internal access latency from the vagaries of the public Internet. Client TCP connections are usually terminated at a frontend server at the ingress to the provider's infrastructure. Separate backend TCP connections between frontend and backend servers complete Web transactions. Using persistent connections and request pipelining on both of these types of connections amortizes TCP connection setup and thereby reduces latency.

Despite the gains such changes have yielded, improvements through structural re-engineering have reached the point of diminishing returns [24], and the latency due to TCP's design now limits further improvement. Increasing deployment of broadband access—the average connection bandwidth globally was 2.8Mbps in late 2012, more than 41% of clients had a bandwidth above 4Mbps, and 11% had more than 10Mbps [2]—has significantly reduced transmission latency. Now, round-trip time (RTT) and the number of round trips required between clients and servers largely determine the overall latency of most Web transfers.

TCP's existing loss recovery mechanisms add RTTs, resulting in a highly-skewed client Web access latency distribution. In a measurement of billions of TCP connections from clients to Google services, we found that nearly 10% of them incur at least one packet loss, and flows with loss take on average five times longer to complete than those without any loss (Section 2). Furthermore, 77% of these losses are repaired through expensive retransmission timeouts (RTOs), often because packets at the tail of a burst were lost, preventing fast recovery. Finally, about 35% of these losses were single packet losses in the tail. Taken together, these measurements suggest that loss recovery dominates the Web latency.

In this paper, we explore *faster loss recovery* methods that are informed by our measurements and that leverage the trend towards multi-stage Web service access. Given the immediate benefits that these solutions can provide, we focus on deployable, minimal en-

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
Copyright 2013 ACM 978-1-4503-2056-6/13/08 ...$15.00.

hancements to TCP rather than a clean-slate design. Our mechanisms are motivated by the following design ideal: *to ensure that every loss is recovered within 1-RTT*. While we do not achieve this ideal, our paper conducts a principled exploration of three qualitatively-different, deployable TCP mechanisms that progressively take us closer to this ideal. The first mechanism, *Reactive*, retransmits the last packet in a window, enabling TCP to trigger fast recovery when it otherwise might have had to incur an RTO. *Corrective* additionally transmits a coded packet that enables recovery without retransmission in cases where a single packet is lost and *Reactive* might have triggered fast recovery. *Proactive* redundantly transmits each packet twice, avoiding retransmissions for most packets in a flow.

Along other dimensions, too, these approaches are qualitatively different. They each involve increasing levels of aggression: *Reactive* transmits one additional packet per window for a small fraction of flows, *Corrective* transmits one additional packet per window for all flows, while *Proactive* duplicates the window for a small portion of flows. Finally, each design leverages the multi-stage architecture in a qualitatively different way: *Reactive* requires only sender side changes and can be deployed on frontend servers, *Corrective* requires both sender and receiver side changes, while *Proactive* is designed to allow service providers to selectively apply redundancy for Web flows, which often are a minuscule fraction of the traffic relative to video on a backbone network. Despite the differences, these approaches face common design challenges: avoiding interference with TCP's fast retransmit mechanism, ensuring accurate congestion window adjustments, and co-existing with middleboxes.

We have implemented all three mechanisms in the Linux kernel. We deployed *Reactive* on frontend servers for production traffic at Google and have used it on hundreds of billions of flows, and we have experimented with *Proactive* for backend Web connections in a setting of interest for a month. In addition, we measured them extensively. Our evaluations of *Reactive* and *Proactive* use traces of several million flows and traffic to a wide variety of clients including mobile devices. We base the evaluation of *Corrective* on realistic loss emulation, as it requires both client and server changes and cannot be unilaterally deployed.

Our large-scale experiment in production with *Proactive* at the backend and *Reactive* at the frontend yielded a 23% improvement in the mean and 47% in 99th percentile latency over today's TCP. Our emulation experiment with *Corrective* yielded 29% improvement in 99th percentile latency for short flows with correlated losses. The penalty for these benefits is the increase in traffic per connection by 0.5% for *Reactive*, 100% for *Proactive*, and 10% for *Corrective* on average. Our experience with these mechanisms indicates that they can yield immediate benefits in a range of settings, and provide stepping stones towards the 1-RTT recovery ideal.

2. THE CASE FOR FASTER RECOVERY

In this section, we present measurements from Google's frontend infrastructure that indicate a pressing need to improve TCP recovery behavior. Web latency is dominated by TCP's startup phase (the initial handshake and slow start) and by time spent detecting and recovering from packet losses; measurements show about 90% of the connections on Web servers finish within the slow start phase, while the remaining experience long recovery latencies [40]. Recent work has proposed to speed up the connection startup by enabling data exchange during handshake [32] and by increasing TCP's initial congestion window [16]. However, mechanisms for faster loss recovery remain largely unexplored for short flows.

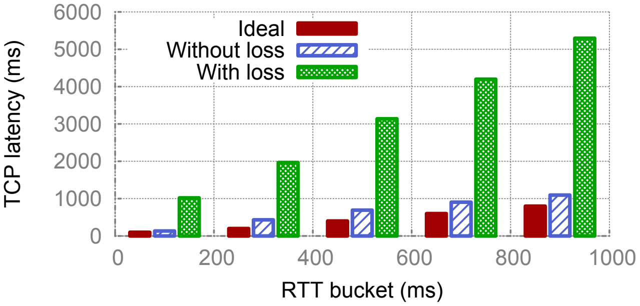


Figure 1: Mean TCP latency to transfer an HTTP response from Web server to a client. Measurements are bucketed by packet round-trip time between the frontend and the client.

Data Collection. We examine the efficacy of TCP's loss recovery mechanisms through measurements of billions of Web transactions from Google services excluding videos. We measure the types of retransmissions in a large data center which primarily serves users from the U.S. East coast and South America. We selected this data center because it has a mix of RTTs, user bandwidths, and loss rates. In addition, about 30% of the traffic it served is for cellular users. For ease of comparison, we also use the same data center to experiment with our own changes to loss recovery described in later sections. We collected Linux TCP SNMP statistics from Web servers and measured TCP latency to clients for one week in December 2012 and January 2013. Observations described here are consistent across several such sample sizes taken in different weeks and months. In addition, we also study packet loss patterns in transactions from two days of server-side TCP traces in 2012, of billions of clients accessing latency-sensitive services such as Web search from five frontend servers in two of our data centers. These measurements of actual client traffic allow us to understand the TCP-layer characteristics causing poor performance and to design our solutions to address them.

Loss makes Web latency 5 times slower. In our traces, 6.1% of HTTP replies saw loss, and 10% of TCP connections saw at least one loss.[1] The average (server) retransmission rate was 2.5%.

Figure 1 depicts the TCP latency in the traces (the time between the first byte the server sent to its receipt of the lack ACK), separating the transfers that experienced loss from those that did not experience loss. The figure buckets the transfers by measured RTT and depicts the mean transfer latency for each bucket. For comparison, the figure also depicts the ideal transfer latency of one RTT. As seen in the figure, transfers without loss generally take little more than the ideal duration. However, transfers that experience loss take much longer to complete—*5 times longer on average*.

Finding: Flows without loss complete in essentially optimal time, but flows with loss are much slower. *Design implication*: TCP requires improved loss recovery behavior.

77% losses are recovered by timeout, not fast recovery. As suggested by the tail transfer latency in our traces, the time to recover from loss can dwarf the time to complete a lossless transfer.

In our traces, frontend servers recover about 23% of losses via fast retransmission—the other 77% require RTOs. This is because Web responses are small and tail drops are common. As a result, there are not enough duplicate ACKs to trigger fast recovery.[2]

Even worse, many timeouts are overly conservative compared to the actual network RTT. The sender bases the length of its RTO

[1] A TCP connection can be reused to transmit multiple responses.

[2] Linux implements early retransmit that requires only one duplicate ACK to perform fast recovery.

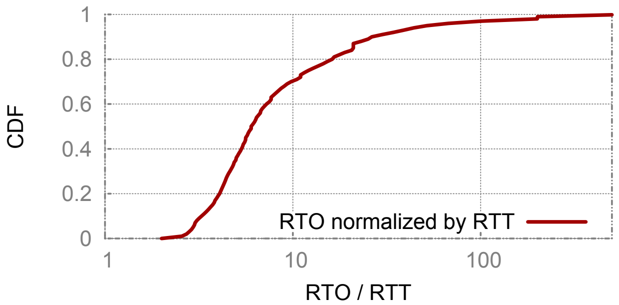


Figure 2: CDF of the measured RTOs normalized by RTT.

upon its estimate of the RTT and the variation in the RTT. In practice, this estimate can be quite large, meaning that the sender will not recover from loss quickly. In our traces we found that the median RTO is six times larger than the RTT, and the 99th percentile RTO is a whopping 200 times larger than the actual RTT, as shown in Figure 2. These high timeout values likely result from high variance in RTT, caused by factors such as insufficient RTT samples early in a flow and varying queuing delays in routers with large buffers [42]. In such cases, an RTO causes a severe performance hit for the client. Note that simply reducing the length of the RTO does not address the latency problem for two reasons. First, it increases the chances of spurious retransmissions. Based on TCP DSACK [10], our traces report that about 40% of timeouts are spurious. More importantly, a spurious RTO reduces the congestion window to one and forces a slow start, unnecessarily slowing the transfer of remaining data.

Finding: Servers currently recover from most losses using slow RTOs. *Design implication*: RTOs should be converted into fast retransmissions or, even better, TCP should recover from loss without requiring retransmission.

(Single) packet tail drop is very common. The duplicate acknowledgments triggered by packets received after a loss can trigger fast retransmission to recover the missing packet(s). The prevalence of RTOs in our traces suggests that loss mostly occurs towards the end of bursts. Figure 3 shows how likely a packet is to be lost, based on its position in the burst. We define a burst as a sequence of packets where the server sends each packet at most $500\mu s$ after the previous one. The figure shows that, with few exceptions, the later a packet occurs in a burst, the more likely it is to be lost. The correlation between position in a burst and the probability of loss may be due to the bursts themselves triggering congestive losses, with the later packets dropped by tail-drop buffers.

Figure 4 indicates, for flows experiencing loss, the probability of having at most two packet losses. For bursts of at most 10 packets, ∼35% experienced exactly one loss, and an additional 10% experienced exactly two losses.

Finding: Many flows lose only one or two consecutive packets, commonly at the tail of a burst. *Design implication*: Minimizing the impact of small amounts of tail loss can significantly improve TCP performance.

These findings confirm not only that tail losses are commonplace in modern networks, but that they can cause poor end-to-end latency. Next, we build upon these findings to develop mechanisms that improve loss recovery performance.

3. TOWARDS 1-RTT RECOVERIES

In this paper, we explore three qualitatively different TCP mechanisms, working towards the ideal of 1-RTT loss recovery. *Reactive* re-sends the last packet in a window, enabling TCP to trigger

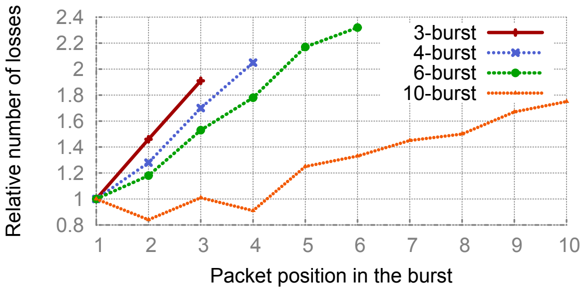


Figure 3: Relative probability of the x-th packet (in a burst) being lost compared to the probability of the first packet being lost (same burst). A line depicts the ratio for a fixed burst length derived from HTTP frontend-client traces.

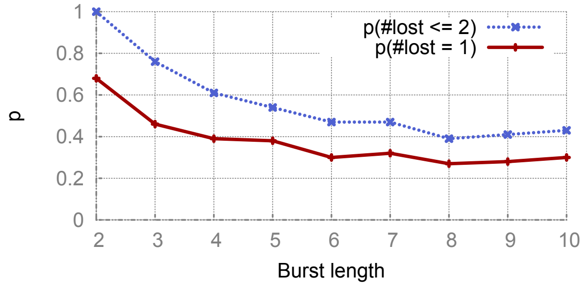


Figure 4: Probability of one (bottom line) or at most two (top line) packet losses in a burst for lossy bursts, derived from HTTP frontend-client traces.

fast recovery when it otherwise might have had to incur an RTO. *Corrective* transmits a coded packet that enables recovery without retransmission when a single packet in the coded block is lost. Finally, *Proactive* is 100% redundant: it transmits each data packet twice, avoiding retransmissions for almost all packets in a flow.

Our measurements from Section 2 motivate not just a focus on the 1-RTT recovery ideal, but have also informed these mechanisms. *Proactive* attempts to avoid loss recovery completely. Motivated by the finding that RTOs dominate loss recovery, *Reactive* effectively converts RTOs into fast retransmissions. *Corrective* is designed for the common case of a single packet loss. They were also designed as a progression towards the 1-RTT ideal: from fast recovery through more frequent fast retransmits in *Reactive*, to packet correction in *Corrective*, to recovery avoidance in *Proactive*. This progression reflects an increase in the level of aggression from *Reactive* to *Proactive* and the fact that each design is subsumed by the next: *Corrective* implicitly converts RTOs to fast retransmissions, and *Proactive* corrects more losses than *Corrective*.

Finally, these mechanisms were designed to be immediately deployable in a multi-stage Web service architecture, like that shown in Figure 5. Each of them makes relatively small changes to TCP, but different designs apply to different stages, with each stage having distinct constraints. *Reactive* requires sender side changes and can be deployed in the client-facing side of frontends to speed Web responses. *Proactive* requires both sender and receiver side changes and can be selectively applied on backends. Prompt loss recovery is relevant for backend connections because frontends deployed in remote, network-constrained locations can experience considerable loss: the average retransmission rate across all our backend connections on a particular day was 0.6% (max=16.3%). While *Proactive* adds high overhead, Web service traffic is a small fraction of overall traffic, so *Proactive*'s aggression adds negligible overhead (in our measurements, latency critical Web traffic is less than 1% of the overall video-dominated traffic). Finally, *Corrective* requires

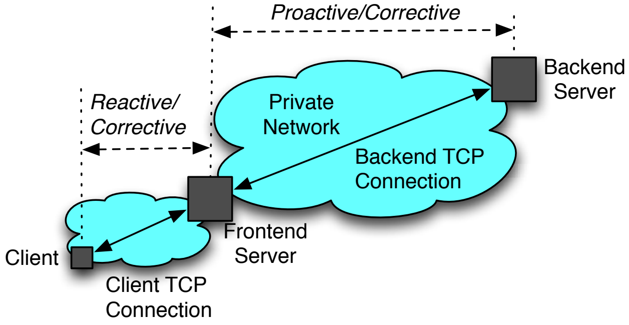


Figure 5: *Proactive* is applied selectively on certain transactions in the backend; *Reactive* can be deployed on client-facing side of frontends to speed Web responses; *Corrective* can apply equally to both client and backend connections.

Aggressiveness / **Phase**	Decreased slightly	Increased slightly	Increased greatly
Startup / Short flows		IW 10 [16] **Proactive**	
Steady state	Vegas [12]	CUBIC [18]	Relentless, Decongestion [28, 33]
Recovery	Moderation [22]	**Corrective**	
Timeout		**Reactive**	DDoS defense by offense [45]

Figure 6: The design space of transport mechanisms that are of a different aggressiveness than baseline.

sender and receiver side changes, and can apply equally to client or backend connections. These designs embed assumptions about the characteristics of losses observed today and about the structure of multi-stage architectures. Section 9 discusses the implications of these assumptions.

Despite the differences between these approaches, they face several common challenges that arise in adding redundancy to TCP. First, a redundantly transmitted data packet might trigger additional ACKs and consequently fast retransmissions. Second, when a redundantly transmitted data packet masks a loss, the congestion control algorithms must react to the loss. Finally, any changes to TCP must co-exist with middleboxes [21]. In subsequent sections, we present the design of each of our mechanisms and describe how they address these challenges.

In the broader context (Figure 6) of other schemes that have attempted to be more or less aggressive than TCP, our designs occupy a unique niche: leveraging gentle aggression for loss recovery. As our results show, this degree of aggression is sufficient to achieve latency reduction without introducing network instability (e.g., by increasing loss rates).

4. REACTIVE

In this section we present our *Reactive* algorithm, a technique to mitigate retransmission timeouts (RTOs) that occur due to tail losses. *Reactive* sends probe segments to trigger duplicate ACKs to attempt to spur fast recovery more quickly than an RTO at the end of a transaction. *Reactive* requires only sender-side changes and does not require any TCP options.

The design of *Reactive* presents two main challenges: a) how to trigger an unmodified client to respond to the server with appropriate information so as to help plug tail losses using fast recovery, and b) how to avoid circumventing TCP's congestion control. After we describe the basic *Reactive* mechanism, we then outline an algorithm to detect the cases in which *Reactive* plugs a hole. We will show that the algorithm makes the sender aware that a loss had occurred so it performs the appropriate congestion window reduction. We then discuss how *Reactive* enables a TCP sender to recover *any* degree of tail losses via fast recovery.

Reactive algorithm. The *Reactive* algorithm allows a sender to quickly detect tail losses without waiting for an RTO.[3] The risk of a sender incurring a timeout is high when the sender has not received any acknowledgments for some time but is unable to transmit any further data either because it is application-limited (out of new data to send), receiver window-limited (rwnd), or congestion window-limited (cwnd). In these circumstances, *Reactive* transmits probe segments to elicit additional ACKs from the receiver. *Reactive* is applicable only when the sender has thus far received in-sequence ACKs and is not already in any state of loss recovery. Further, it is designed for senders with Selective Acknowledgment (SACK) enabled because the SACK feedback of the last packet allows senders to infer whether any tail segments were lost [11, 29].

The *Reactive* algorithm triggers on a newly defined probe timeout (PTO), which is a timer event indicating that an ACK is overdue on a connection. The sender sets the PTO value to approximately twice the smoothed RTT and adjusts it to account for a delayed ACK when there is only one outstanding segment. The basic version of the *Reactive* algorithm transmits one probe segment after a PTO if the connection has outstanding unacknowledged data but is otherwise idle, i.e. it is not receiving any ACKs or is cwnd/rwnd/application-limited. The transmitted segment—the *loss probe*—can be either a new segment if available and the receive window permits, or a retransmission of the most recently sent segment, (i.e., the segment with the highest sequence number). In the case of tail loss, the ACK for the probe triggers fast recovery. In the absence of loss, there is no change in the congestion control or loss recovery state of the connection, apart from any state related to *Reactive* itself.

Pseudocode and Example. Algorithm 1 gives pseudocode for the basic *Reactive* algorithm. FlightSize is the amount of in-network outstanding data and WDT is the worst-case delayed ACK timer. The key part of the algorithm is the transmission of a probe packet in Function *handle_pto*() to elicit an ACK without waiting for an RTO. It retransmits the last segment (or new one if available), such that its ACK will carry SACK blocks and trigger either SACK-based [11] or Forward Acknowledgment (FACK)-based fast recovery [29] in the event of a tail loss.

Next we provide an example of how *Reactive* operates. Suppose a sender transmits ten segments, 1 through 10, after which there is no more new data to transmit. A probe timeout is scheduled to fire two RTTs after the transmission of the tenth segment, handled by *schedule_pto*() in Algorithm 1. Now assume that ACKs for segments one through five arrive, but segments six through ten at the tail are lost and no ACKs are received. Note that the sender (re)schedules its probe timer relative to the last received ACK (Function *handle_ack*()), which is for segment five in this case. When the probe timer fires, the sender retransmits segment ten (Function *handle_pto*())—this is the key part of the algorithm. After an RTT, the sender receives an acknowledgement for this

[3] In the rest of the paper, we'll use the term "tail loss" to generally refer to either drops at the tail end of transactions or a loss of an entire window of data or acknowledgments.

Algorithm 1: *Reactive*.

```
% Called after transmission of new data in Open state.
Function schedule_pto():
    if $FlightSize > 1$ then $PTO \leftarrow 2 \times RTT$;
    else if $FlightSize == 1$ then $PTO \leftarrow 1.5 \times RTT + WDT$;
    $PTO = \min(PTO, RTO)$
    Conditions:
      (a) Connection is in open state
      (b) Connection is cwnd- and/or application-limited
      (c) Number of consecutive PTOs $\leq 2$
      (d) Connection is SACK-enabled
    if all conditions hold then Arm timer with $PTO$;
    else Rearm timer with RTO;
Function handle_pto():
    if previously unsent segment exists then
        Transmit new segment
        $FlightSize \leftarrow FlightSize + segment\ size$
    else Retransmit last segment;
    $schedule\_pto()$
Function handle_ack():
    Cancel existing $PTO$
    $schedule\_pto()$
```

packet that carries SACK information indicating the missing segments. The sender marks the missing segments as lost (here segments six through nine) and triggers FACK-based recovery. Finally, the connection enters fast recovery and retransmits the remaining lost segments.

Detecting recovered losses. If the only loss was the last segment, there is the risk that the loss probe itself might repair the loss, effectively masking it from congestion control. *Reactive* includes a loss detection mechanism that detects, by examining ACKs, when the retransmission probe might have masked a loss; *Reactive* then enforces a congestion window reduction, thus complying with the mandatory congestion control.[4]

The basic idea of *Reactive* loss detection is as follows. Consider a *Reactive* retransmission "episode" where a sender retransmits N consecutive *Reactive* packets, all for the same tail packet in a flight. Suppose that an episode ends when the sender receives an acknowledgment above the $SND.NXT$ at the time of the episode. We want to make sure that before the episode ends the sender receives N "*Reactive* dupacks", indicating that all N *Reactive* probe segments were unnecessary, so there was no hole that needed plugging. If the sender gets less than N "*Reactive* dupacks" before the end of the episode, it is likely that the first *Reactive* packet to arrive at the receiver plugged a hole, and only the remaining *Reactive* packets that arrived at the receiver generated dupacks. In the interest of space, we omit the pseudocode for this mechanism.

Note that delayed ACKs complicate the picture since a delayed ACK implies that the sender will receive fewer ACKs than would normally be expected. To mitigate this complication, before sending a loss probe retransmission, the sender should attempt to wait long enough that the receiver has sent any delayed ACKs that it is withholding. Our sender implementation features such a delay.

If there is ACK loss or a delayed ACK, then this algorithm is conservative, because the sender will reduce cwnd when in fact there was no packet loss. In practice this is acceptable, and potentially even desirable: if there is reverse path congestion then reducing cwnd is prudent.

Pattern	*Reactive* scoreboard	Mechanism
`AAAL`	`AAAA`	*Reactive* loss detection
`AALL`	`AALS`	Early retransmit
`ALLL`	`ALLS`	Early retransmit
`LLLL`	`LLLS`	FACK fast recovery
`>=5 L`	`..LS`	FACK fast recovery

Table 1: Recovery behavior with *Reactive* packets for different tail loss scenarios (A = ACKed segment, L = lost segment, S = SACKed segment). The TCP sender maintain the received SACK blocks information in a data structure called scoreboard. The *Reactive* scoreboard shows the state for each segment after the *Reactive* packet was ACKed.

Implementation. We implemented *Reactive* in Linux kernels 2.6 and 3.3. In line with our overarching goal of keeping our mechanisms simple, the basic *Reactive* algorithm is 110 lines of code and the loss detection algorithm is 55 ($\sim$0.7% of Linux TCP code).

Initially we designed *Reactive* to send a zero window probe (ZWP) with one byte of new or old data. The acknowledgment from the ZWP would provide an additional opportunity for a SACK block to detect loss without an RTO. Additional losses can be detected subsequently and repaired with SACK-based fast recovery. However, in practice sending a single byte of data turned out to be problematic to implement in Linux TCP. Instead we opted to send a full segment to probe at the expense of the slight complexity required to detect the probe itself masking losses.

The *Reactive* algorithm allows the source to transmit one or two PTOs. However, one of the design choices we made in our implementation is to not use consecutive probe timeouts, since we observed that over 90% of the latency gains by *Reactive* are achieved with a single probe packet. Finally, the worst case delayed ACK timer we use is 200ms. This is the delayed ACK timer used in most of the Windows clients served from our Web server.

Reactive is also described in the IETF draft [15] and is on by default in mainline Linux kernels [14].

Recovery of any N-degree tail loss. *Reactive* remedies discontinuity in today's loss recovery algorithms wherein a single segment loss in the middle of a packet train can be recovered via fast recovery while a loss at the end of the train causes a retransmission timeout. With *Reactive*, a segment loss in the middle of a train as well as at the tail triggers the same fast recovery mechanisms. When combined with a variant of the early retransmit mechanism [4], *Reactive* enables fast recovery instead of an RTO for any degree of N-segment tail loss as shown in Table 1.[5]

5. CORRECTIVE

Reactive recovers from tail loss without incurring (slow) RTOs, and it does so without requiring client-side changes, but it does not eliminate the need for recovery. Instead, it still requires the sender to recognize packet loss and retransmit. *Proactive* achieves 0-RTT loss recovery, but it has limited applicability, since it doubles bandwidth usage. Further, this level of redundancy may be overkill in many settings–our measurements in Section 2 found that many bursts lose only a single packet.

In this section, we explore a middle way–a mechanism to achieve 0-RTT recovery in common loss scenarios. Our approach, *Corrective*, requires both sender and receiver changes (like *Proactive*, unlike *Reactive*) but has low overhead (like *Reactive*, unlike *Proactive*). Instead of complete redundancy, we employ forward error

[4] Since we observed from our measurements that a significant fraction of the hosts that support SACK do not support DSACK [10], the *Reactive* algorithm for detecting such lost segments relies only on the support of basic SACK.

[5] The variant we propose is to allow an early retransmit in the case where there are three outstanding segments that have not been cumulatively acknowledged and one segment that has been fully SACKed.

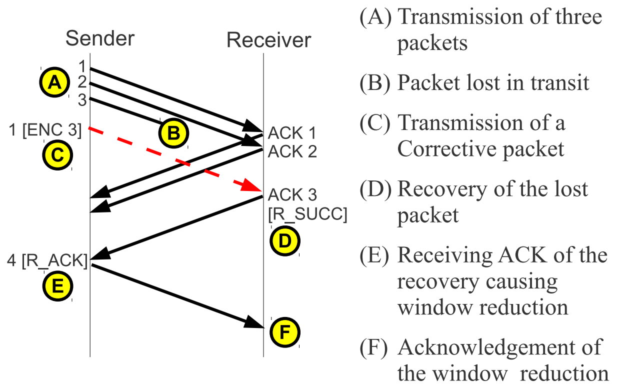

Sender
Receiver
1
2
3
A
1 [ENC 3]
B
C
ACK 1
ACK 2
ACK 3
[R_SUCC]
D
4 [R_ACK]
E
F
(A) Transmission of three packets
(B) Packet lost in transit
(C) Transmission of a Corrective packet
(D) Recovery of the lost packet
(E) Receiving ACK of the recovery causing window reduction
(F) Acknowledgement of the window reduction

Figure 7: Timeline of a connection using *Corrective*. The flow shows regular (solid) and *Corrective* packets (dashed), sequence/ACK numbers, and *Corrective* option values (terms in brackets).

correction (FEC) within TCP. The sender transmits extra FEC packets so that the receiver can repair a small number of losses.

While the use of FEC for transport has been explored in the past, for example in [9, 41, 43], to our knowledge we are the first to place FEC within TCP in a way that is incrementally deployable across today's networks. Our goal is to achieve an immediate decrease in Web latency, and thus enhancing native TCP with FEC is important. However, this brings up significant challenges that we now discuss.

Corrective encoding. The sender and receiver negotiate whether to use *Corrective* during TCP's initial handshake. If both hosts support it, *every* packet in the flow will include a new TCP option, the *Corrective* option. We then group sequences of packets and place the XOR of their payloads into a single *Corrective* checksum packet. Checksums have low CPU overhead relative to other coding schemes like Reed-Solomon codes [35]; while such algorithms provide higher recovery rates than checksums in general, our measurements indicated that many bursts experience only a single loss, and so a checksum can recover many losses.

Corrective groups together all packets seen within a time window, up to a maximum of sixteen MSS bytes of packets. It aligns the packets along MSS bytes boundaries to XOR them into a single *Corrective* payload. Because no regular packet carries a payload of more than MSS bytes, this encoding guarantees that the receiver can recover any single packet loss. *Corrective* delays transmitting the encoded packet by $\frac{\text{RTT}}{4}$ since our measurements indicate that this minimizes the probability of losing both, a regular packet and the XOR packet that encodes it.

Incorporating loss correction into TCP adds a key challenge. TCP uses a single sequence number space to provide an ordered and reliable byte stream. Blocking on reliable delivery of *Corrective* packets is counter to our goal of reducing latency. For this reason, a *Corrective* packet uses the same sequence number as the first packet it encodes. This prevents reliability for *Corrective* packets and avoids the overhead of encoding the index of the first encoded byte in a separate header field. The *Corrective* packet sets a special `ENC` flag in its *Corrective* option signaling that the payload is encoded which allows the receiver to distinguish a *Corrective* packet from a regular retransmission (since they both have the same sequence number). The option also includes the number of bytes that the payload encodes.

Corrective recovery. To guarantee that the receiver can recover any lost packet, the *Corrective* module keeps the last 15 ACKed MSS blocks buffered, even if the application layer has already consumed these blocks.[6] Since a *Corrective* packet encodes at most 16 MSS blocks, the receiver can then recover any single lost packet by computing the XOR of the *Corrective* payload and the buffered blocks in the encoding range. To obtain the encoding range, the receiver combines the sequence number of the *Corrective* packet (which is set to be the same as the sequence number of the first encoded byte) and the number of bytes encoded (which is part of the *Corrective* TCP option).

Corrective reception works as follows. Once the receiver establishes that the payload is encoded (by checking the `ENC` flag in the *Corrective* option), it checks for holes in the encoded range. If it received the whole sequence, the receiver drops the *Corrective* packet. Otherwise, if it is missing at most MSS continuous bytes, the receiver uses the *Corrective* packet to recover the subsequence and forward it to the regular reception routine, allowing 0-RTT recovery. If too much data is missing for the *Corrective* packet to recover, the receiver sends an explicit duplicate ACK. This ACK informs the sender that a recovery failed and denotes which byte ranges were lost[7] via an `R_FAIL` flag and value in the *Corrective* option. The sender marks the byte ranges as lost and triggers a fast retransmit. Thus, even when immediate recovery is not possible, *Corrective* provides the same benefit as *Reactive*.

If the receiver were to simply ACK a recovered packet, it would mask the loss and circumvent congestion control during a known loss episode. Since TCP connections may be reused for multiple HTTP transactions, masking losses can hurt subsequent transfers. To prevent this behavior, we devised a mechanism similar to explicit congestion notification (ECN) [34]. Upon successful *Corrective* recovery, the receiver enables an `R_SUCC` flag in the *Corrective* option in each outgoing ACK, signaling a successful recovery. Once the sender sees this flag, it triggers a cwnd reduction. In addition, it sets an `R_ACK` flag in the *Corrective* option of the next packet sent to the receiver. Once the receiver observes `R_ACK` in an incoming packet, indicating that the sender reduced the congestion window, it disables `R_SUCC` for future packets. Figure 7 shows a sample packet with a successful *Corrective* recovery.

Implementation. We implemented our prototype in Linux kernel versions 2.6 and 3.2 in 1674 lines of code (∼7.3% of the Linux TCP codebase). Our implementation is modularized and makes minimal changes to the existing kernel. This separation has made it easy, for example, to port *Corrective* to the Linux stack for Android devices. We plan to make our implementation publicly available.

6. PROACTIVE

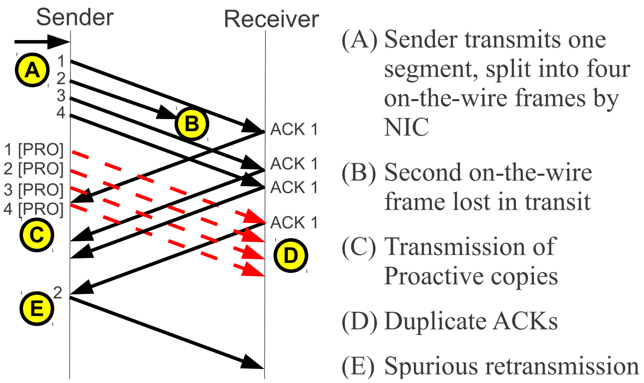
Proactive takes the aggressive stance of proactively transmitting copies of each TCP segment. If at least one copy of the segment arrives at the receiver then the connection proceeds with no delay. The receiver can discard redundant segments. While sending duplicate segments can potentially increase congestion and consequently decrease goodput and increase response time, *Proactive* is designed only for latency-sensitive services on networks where these services occupy a small percentage of the total traffic. While repeating packets is less efficient than sophisticated error correction coding schemes, we designed *Proactive* to keep the additional complexity of TCP implementation at a minimum while achieving significant latency improvements.

While intuitively simple, the implementation of *Proactive* has some subtleties. A naive approach would be to send one copy of every segment, or two instances of every segment.[8] If the destination

[6] Packets received out-of-order are already buffered by default.

[7] We can say that the packets were lost with confidence since the *Corrective* packet transmissions are delayed (as described earlier).

[8] We use the term *copies* to differentiate them from duplicate segments that TCP sends during retransmission.



(A) Sender transmits one segment, split into four on-the-wire frames by NIC

(B) Second on-the-wire frame lost in transit

(C) Transmission of Proactive copies

(D) Duplicate ACKs

(E) Spurious retransmission

Figure 8: Timeline of a *Proactive* connection with TSO enabled that loses a segment. While the *Proactive* copy recovers the loss, the sender retransmits the segment due to three duplicate ACKs.

receives both data segments it will send two ACKs, since the reception of an out-of-order packet triggers an immediate ACK [6]. The second ACK will be a duplicate ACK (i.e., the value of the ACK field will be the same for both segments). Since modern Linux TCP stacks use duplicate SACKs (DSACK) to signal sequence ranges which were received more than once, the second ACK will also contain a (D)SACK block. This duplicate ACK does not falsely trigger fast recovery because it only acknowledges old data and does not indicate a hole (i.e., missing segment).

However, many modern network interface controllers (NICs) use TCP Segmentation Offloading (TSO) [21]. This mechanism allows TCP to process segments which are larger than MSS, with the NIC taking care of breaking them into MTU-sized frames.[9] For example, if the sender-side NIC splits a segment into K on-the-wire frames, the receiver will send back $2K$ ACKs . If $K >$ `dupthresh` and SACKs are disabled or some segments are lost, the sender will treat the duplicate ACKs as a sign of congestion and enter a recovery mode. This is clearly undesirable since it slows down the sender and offsets *Proactive*'s potential latency benefits. Figure 8 illustrates one such a spurious retransmission.

To avoid spurious retransmissions we disable TSO for the flows that use *Proactive* and enlist the receiver to identify original/copied segments reordered by or lost in the network.

Specifically, the sender marks the copied segments by setting a flag in the header using one of the reserved but unused bits. Then, a receiver processes incoming packets as follows. If the flag is set, the packet is only processed if it was not received before (otherwise it is dropped). In this case an ACK is generated. If the flag is not set, the packet will be processed if it was not received before or if the previous packet carrying the same sequence did not have the flag set either. These rules will prevent the generation of duplicate ACKs due to copied segments while allowing duplicate ACKs that are due to retransmitted segments. In addition to copying data segments, *Proactive* can be configured to copy SYN and pure ACK segments for an added level of resiliency.

We implemented *Proactive* in Linux kernels 2.6 and 3.3 with the new module comprising 358 lines of code, or ∼1.6% of the Linux TCP codebase.

7. THE ROLE OF MIDDLEBOXES

We aim to make our modules usable for most connections in today's Internet, despite on-path middleboxes [21]. *Reactive* is fully compatible with middleboxes since all *Reactive* packets are either retransmissions of previously sent packets or the next in-order segment. *Proactive* uses reserved bits in the TCP header for copied segments which can trigger middleboxes that discard packets with non-compliant TCP flags. However, in our experience, possibly due to the widespread use of reserved bits and the position of frontend servers relative to middleboxes, we did not observe this effect in practice.

Corrective introduces substantial changes to TCP which could lead to compatibility issues with middlebox implementations that are unaware of the *Corrective* functionality. Our goal is to ensure a graceful fallback to standard TCP in situations where *Corrective* is not usable. Even if hosts negotiate *Corrective* during the initial handshake, it is possible for a middlebox to strip the option from a later packet. To be robust to this, if either host receives a packet without the option, it discards the packet and stops using *Corrective* for the remainder of the connection, so hosts don't confuse *Corrective* packets with regular data packets. Some middleboxes translate sequence numbers to a different range [21], and so *Corrective* uses only relative sequence numbers to convey metadata (such as the encoding range) between endpoints. We have also designed, but have not yet fully implemented, solutions for other middlebox issues. Some devices rewrite the ACK number for a recovered sequence since they have not seen this sequence before. To solve this problem, the sender would retransmit the recovered sequence, even though it is not needed by the other endpoint anymore, to plug this "sequence hole" in the state of the middlebox. Solutions to other issues include *Corrective* checksums to detect if a middlebox rewrites payloads for previously seen sequences, as well as introducing additional identifier information to the *Corrective* option to cope with packet coalescing or splitting.

We could have avoided middlebox issues by implementing *Corrective* above or below the transport layer. Integrating it into TCP made it easier to leverage TCP's option negotiation (so connections can selectively use *Corrective*) and its RTT estimates (so that the *Corrective* packet transmission can be timed correctly). It also eased buffer management, since *Corrective* can leverage TCP's socket buffers; this is especially important since buffer space is at a premium in production Web servers.

Ideally, middlebox implementations would be extended to be aware of our modules. For *Proactive*, adding support for the TCP flag used is sufficient. *Corrective* on the other hand requires new logic to distinguish between regular payloads and *Corrective*-encoded payloads based on the *Corrective* option and flags used. In particular, stateful middleboxes need this functionality to properly update the state kept for *Corrective*-enabled connections.

8. EVALUATION

Next we evaluate the performance gains achieved by *Reactive*, *Corrective*, and *Proactive* in our experiments. We begin with results from a combined experiment running *Proactive* for backend connections and *Reactive* for client connections. We then describe detailed experiments for each of the mechanisms in order of increasing aggressiveness. First, we describe our experimental setup.

8.1 Experimental setup

We performed all of our Web server experiments with *Reactive* and *Proactive* in a production data center that serves live user traffic for a diverse set of Web applications. The Web servers run Linux 2.6 using default settings, except that ECN is disabled. The servers terminate user TCP connections and are load balanced by steering new connections to randomly selected Web servers based on the server and client IP addresses and ports.

Calibration measurements over 24-hour periods show that

[9] For now assume that each of these on-the-wire packets generates an ACK and that the network does not lose or reorder any messages.

SNMP and HTTP latency statistics agree within 0.5% between individual servers. This property permits us to run N-way experiments concurrently by changing TCP configurations on groups of servers. A typical A/B experiment runs on four or six servers with half of them running the experimental algorithm while the rest serve as the baseline. Note that multiple simultaneous connections opened by a single client are likely to be served by Web servers with different A/B configurations. These experiments were performed over several months.

The primary latency metric that we measure is *response time (RT)* which is the interval between the Web server receiving a client's request to the server receiving an ACK for the last packet of the response. We are also interested in retransmission statistics and the overhead from each scheme. Linux is a fair baseline comparison because it implements the latest loss recovery techniques in TCP literature and IETF RFCs.[10]

8.2 End-to-end evaluation

Since our overarching goal is to reduce latency for Web transfers in real networks, we first present our findings in experiments using both *Reactive* and *Proactive* in an end-to-end setting.

The end-to-end experiment involves multi-stage TCP flows as illustrated in Figure 5. The backend server resides in the same data center described in Section 2, but user requests are directed to nearby frontend servers that then forward them to the backend server. The connections between the backend and frontend servers use *Proactive*, while the connections between the end users and the frontend nodes use *Reactive*.[11] The baseline servers used standard TCP for both backend and client connections.

We measure RT, which includes the communication between the frontend and backend servers. Table 2 shows that, over a two-day period, the experiment yielded a 14% reduction in average latency and a substantial 37% improvement in the 99th percentile, We noticed that the baseline retransmission rate over the backend connections was 5.5% on Day 1 of the experiment and 0.25% on Day 2. The redundancy added by *Proactive* effectively reduced the retransmission rate to 0.02% for both days. Correspondingly, the mean response time reduction on Day 1 was 21% (48% for the 99th percentile) and 4% on Day 2 (9% for the 99th percentile). Results from another 15-day experiment between a different frontend-backend server pair demonstrated a 23.1% decrease in mean response time (46.7% for the 99th percentile). The sample sizes for the second experiment were ∼2.6 million queries while the retransmission rates for the baseline and *Proactive* were 0.99% and 0.09%, respectively.

Taken in perspective, such a latency reduction is significant: consider that an increase in TCP's initial congestion window to ten segments—a change of much larger scope—improved the average latency by 10% [16]. We did not measure the impact of 23% response latency reduction on end-user experience. Emulations with Corrective in section 8.4.2, show the browser's *render start time* metric. Ultimately, user latency depends not just on TCP but also on how browsers use the data – including the order that clients issue requests for style sheets, scripts and images, image scaling and compression level, browser caching, DNS lookups and so on. TCP's job is to deliver the bits as fast as possible to the browser.

Quantile	Linux	*Proactive* + *Reactive*	
25	362	-5	*-1%*
50	487	-11	*-2%*
90	940	-173	*-18%*
99	5608	-2058	*-37%*
Mean	700	-99	*-14%*
Sample size	186K	243K	

Table 2: RT comparison (in ms) for Linux baseline and *Proactive* combined with *Reactive*. The two rightmost columns show the relative latency w.r.t the baseline. This experiment was enabled only for short Web transfers, due to its increased overhead.

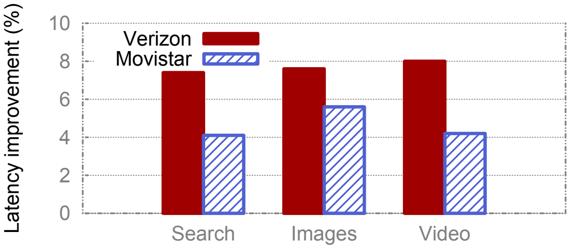


Figure 9: Average latency improvement (in %) of HTTP responses with *Reactive* vs. baseline Linux for two mobile carriers. Carriers and Web applications are chosen because of their large sample size.

To understand where the improvements come from, we elaborate on the performance of each of the schemes in the following subsections.

8.3 Reactive

Using our production experimental setup, we measured *Reactive*'s performance relative to the baseline in Web server experiments spanning over half a year. The results reported below represent a week-long snapshot. Both the experiment and baseline used the same kernels, which had an option to selectively enable *Reactive*. Our experiments included the two flavors of *Reactive* discussed above, with and without loss detection support. The results reported here include the combined algorithm with loss detection. All other algorithms such as early retransmit and FACK based recovery are present in both the experiment and baseline.

Table 3 shows the percentiles and average latency improvement of key Web applications, including responses without losses. The varied improvements are due to different response-size distributions and traffic patterns. For example, *Reactive* helps the most for Images, as these are served by multiple concurrent TCP connections which increase the chances of tail segment losses.[12] There are two takeaways: the average response time improved up to 6% and the 99th percentile improved by 10%. Also, nearly all of the improvement for *Reactive* is in the latency tail (post-90th percentile).

Figure 9 shows the data for mobile clients, with an average improvement of 7.2% for Web search and 7.6% for images transferred over Verizon.

The reason for *Reactive*'s latency improvement becomes apparent when looking at the difference in retransmission statistics shown in Table 4—*Reactive* reduced the number of timeouts by 14%. The largest reduction in timeouts is when the sender is in the *Open* state in which it receives only in-sequence ACKs and no duplicate ACKs, likely because of tail losses. Correspondingly, RTO-triggered retransmissions occurring in the slow start phase reduced by 46% relative to baseline. *Reactive* probes converted timeouts to fast recoveries, resulting in a 49% increase in fast recovery events.

[10] This includes SACK, F-RTO, RFC 3517, limited-transmit, dynamic duplicate ACK threshold, reordering detection, early retransmit algorithm, proportional rate reduction, FACK based threshold recovery, and ECN.

[11] For practical reasons, we did not include *Corrective* in this experiment as it requires changes to client devices that we did not control.

[12] It is common for browsers to use four to six connections per domain, and for Web sites to use multiple subdomains for certain Web applications.

	Google Web Search			Images			Google Maps		
Quantile	Linux	*Reactive*		Linux	*Reactive*		Linux	*Reactive*	
25	344	-2	*-1%*	74	0		59	0	
50	503	-5	*-1%*	193	-2	*-1%*	155	0	
90	1467	-43	*-3%*	878	-65	*-7%*	487	-18	*-3%*
99	14725	-760	*-5%*	5008	-508	*-10%*	2882	-290	*-10%*
Mean	1145	-32	*-3%*	471	-29	*-6%*	305	-14	*-4%*
Sample size	5.7M	5.7M		14.8M	14.8M		1.64M	1.64M	

Table 3: Response time comparison (in ms) of baseline Linux vs. *Reactive*. The *Reactive* columns shows relative latency w.r.t. the baseline.

Retransmission type	Linux	*Reactive*	
Total # of Retransmission	107.5M	-7.3M	*-7%*
Fast Recovery events	5.5M	+2.7M	*+49%*
Fast Retransmissions	24.7M	+8.2M	*+33%*
Timeout Retrans.	69.3M	-9.4M	*-14%*
Timeout On Open	32.4M	-8.3M	*-26%*
Slow Start Retrans.	13.5M	-6.2M	*-46%*
cwnd undo events	6.1M	-3.7M	*-61%*

Table 4: Retransmission statistics in Linux and the corresponding delta in the *Reactive* experiment. *Reactive* results in 14% fewer timeouts and converts them to fast recovery.

Also notable is a significant decrease in the number of spurious timeouts, which explains why the experiment had 61% fewer cwnd *undo* events. The Linux TCP sender [38] uses either DSACK or timestamps to determine if retransmissions are spurious and employs techniques for undoing cwnd reductions. We also note that the total number of retransmissions decreased 7% with *Reactive* because of the decrease in spurious retransmissions.

We also quantified the overhead of sending probe packets. The probes accounted for 0.48% of all outgoing segments. This is a reasonable overhead even when contrasted with the overall retransmission rate of 3.2%. 10% of the probes are new segments and the rest are retransmissions, which is unsurprising given that short Web responses often do not have new data to send [16]. We also found that, in about 33% of the cases, the probes themselves plugged the only hole, and the loss detection algorithm reduced the congestion window. 37% of the probes were not necessary and resulted in a duplicate acknowledgment.

A natural question that arises is a comparison of *Reactive* with a shorter RTO such as 2 × RTT. We did not shorten the RTO on live user tests because it induces too many spurious retransmissions that impact user experience. Tuning the RTO algorithm is extensively studied in literature and is complementary to *Reactive*. Our own measurements show very little room exists in fine-tuning the RTO estimation algorithm. The limitations are: 1) packet delay is becoming hard to model as the Internet is moving towards wireless infrastructure, and 2) short flows often do not have enough samples for models to work well.

8.4 Corrective

In contrast to *Reactive* and *Proactive*, we have not yet deployed *Corrective* in our production servers since it requires both server and client support. We evaluate *Corrective* in a lab environment.

8.4.1 Isolated flows

Experimental setup. We directly connected two hosts that we configured to use the *Corrective* module. We used the `netem` module to emulate a 200 ms RTT between them and emulated both fixed loss rates and correlated loss. In correlated loss scenarios, each packet initially had a drop probability of 0.01 and we raised the loss probability to 0.5 if the previous packet in the burst was lost. We chose these parameters to approximate the loss patterns observed in the data collection described earlier (see Section 2). We used `netperf` to evaluate the impact of *Corrective* on various types of connections. We ran each experiment 10,000 times with *Corrective* disabled (baseline) and 10,000 times with it enabled. All percentiles shown in tables for this evaluation have margins of error $< 2\%$ with 95% confidence.

***Corrective* substantially reduces the latency for short bursts in lossy environments.** In Table 5a we show results for queries using 40 byte request and 5000 byte response messages, similar to search engine queries. These queries are isolated which means that the hosts initiate a TCP connection, then the client sends a request, and the server responds, after which the connection closes. Table 5b gives results for pre-established TCP connections; here we measure latency from when the client sends the request. Both tables show the relative latency improvement when using *Corrective* for correlated losses and for a fixed loss rate of 2%.

When we include handshakes, *Corrective* reduces average latency by 4–10%, depending on the loss scenario, and reduces 90th percentile latency by 18–28%. Because hosts negotiate *Corrective* as part of the TCP handshake, SYN losses are not corrected which can lead to slow queries if the SYN is lost. If we pre-establish connections, as would happen when a server sends multiple responses over a single connection, *Corrective* packets cover the entire flow, and in general *Corrective* provides high latency reductions in the 99th percentile as well.

Existing work demonstrates that transmitting all SYN packets twice can reduce latency in cases of SYN loss [44].[13] For our correlated loss setting, on queries that included handshakes, we found that adding redundant SYN transmissions to standard TCP reduces the 99th percentile latency by 8%. If we use redundant SYN transmission with *Corrective*, the *combined* reduction reaches 17%, since the two mechanisms are complementary.

***Corrective* provides less benefit over longer connections.** Next, using established connections, we evaluate *Corrective*'s performance when transferring larger responses. While still reducing 90th percentile latency by 7% to 10% (Table 5c), *Corrective* provides less benefit in the tail on these large responses than it did for small responses. The benefits diminish as the minimum number of RTTs necessary to complete the transaction increases (due to the message size). As a result, the recovery of losses in the tail no longer dominates the overall transmission time. *Corrective* is better suited for small transfers common to today's Web [16].

8.4.2 Web page replay

Experimental setup. In addition to the synthetic workloads, we used the Web-page-replay tool [1] and dummynet [13] to replay resource transfers for actual Web page downloads through controlled, emulated network conditions. We ran separate tests for

[13] These redundant transmissions are similar to *Proactive* applied only to the SYN packet.

Quantile	Random	Correlated
50	0%	0%
90	-28%	-24%
99	0%	-15%
Mean	-8%	-4%

(a) Short transmission with connection establishment (Initial handshake, 40 byte request, 5000 byte response)

Quantile	Random	Correlated
50	0%	0%
90	-37%	0%
99	-52%	-29%
Mean	-13%	-7%

(b) Short transmission without connection establishment (40 byte request, 5000 byte response)

Quantile	Random	Correlated
50	-13%	0%
90	-10%	0%
99	-5%	-9%
Mean	-10%	-1%

(c) Long transmission without connection establishment (40 byte request, 50000 byte response)

Table 5: Latency reduction with *Corrective* for random and correlated loss patterns under varying connection properties.

Quantile	Linux	*Proactive*	
25	372	-9	*-2%*
50	468	-19	*-4%*
90	702	-76	*-11%*
99	1611	-737	*-46%*
Mean	520	-65	*-13%*
Sample size	260K	262K	

Table 6: RT comparison (in ms) for Linux baseline and *Proactive*. The *Proactive* columns shows the relative latency vs. the baseline. *Proactive* was enabled only for short Web transfers, due to its increased overhead.

Web pages tailored for desktop and mobile clients. The tests for desktop clients emulated a cable connection with 5Mbit/s downlink and 1Mbit/s uplink bandwidth and an RTT of 28ms. The tests for mobile clients emulated a 3G mobile connection with 2Mbit/s downlink and 1Mbit/s uplink bandwidth and an RTT of 150ms.[14] In all tests, we simulated correlated losses as described earlier.

We tested a variety of popular Web sites, and *Corrective* substantially reduced the latency distribution in all cases. For brevity, we limit our discussion to two representative desktop Web sites, a simple page with few resources (Craigslist, 5 resources across 5 connections, 147KB total) and a content-rich page requiring many resources from a variety of providers (New York Times, 167 resources across 47 connections, 1387KB total).

Figure 10 shows the cumulative latency distributions for these websites requested in a desktop environment. The graphs confirm that *Corrective* can improve latency significantly in the last quartile. For example, the New York Times website takes 15% less time until the first objects are rendered on the screen in the 90th percentile. *Corrective* also significantly improves performance in a lossy mobile environment as well. For example, fetching the mobile version of the New York Times website takes 2793ms instead of 3644ms (-23%) in the median, and 3819ms instead of 4813ms (-21%) in the 90th percentile.

8.5 Proactive

While Section 8.2 presented results when *Reactive* in the client connection is used in conjunction with *Proactive* in the backend connection, in this section we report results using only *Proactive* in the backend connections. We conducted the experiments in production datacenters serving live user traffic, as described in Section 8.1. Table 6 presents the reduction in response time that *Proactive* achieves for short Web transfers by masking many of the TCP losses on the connection between the CDN node and the backend server. Specifically, while the average retransmission rate for the baseline was 0.91%, the retransmission rate for *Proactive* was only 0.13%. Even though the difference in retransmission rates may not seem significant, especially since the baseline rate is already small, *Proactive* reduces tail latency (99-th percentile) by 46%.

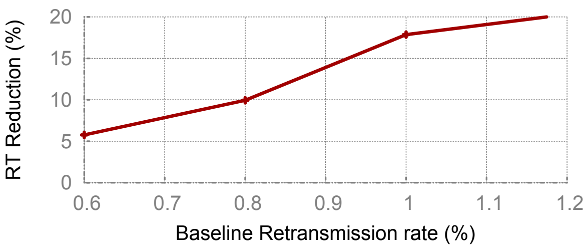


Figure 11: Reduction in response time achieved by *Proactive* as a function of baseline retransmission rate.

What is not obvious from Table 6 is the sensitivity of response time to losses and consequently the benefit that *Proactive* brings by masking these losses. The performance difference between the two days of the experiment in Section 8.2 hinted at this sensitivity. Here we report results across one week, allowing a more systematic evaluation of the relationship between baseline retransmission rate and response time reduction. Figure 11 plots the reduction in response time as a function of the baseline retransmission rate. Even though the baseline retransmission rate increases only modestly across the graph, *Proactive*'s reduction of the average response time grows from 6% to 20%.

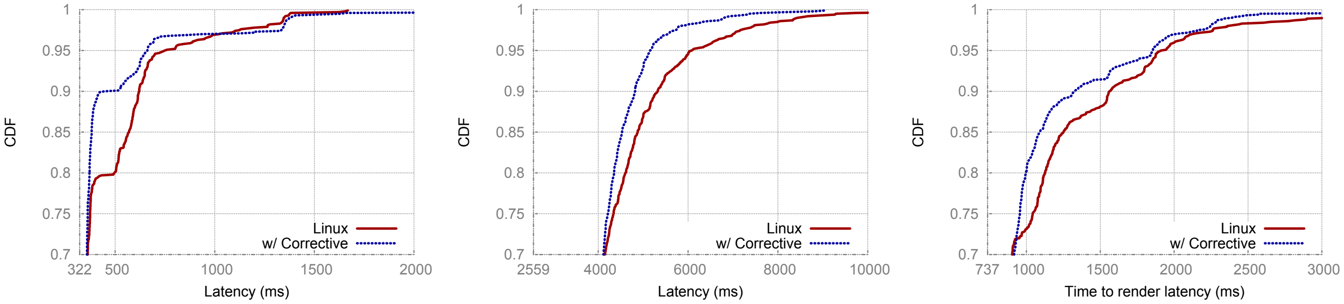
9. DISCUSSION

The 1-RTT Recovery Ideal. Even if the mechanisms described in this paper do not achieve the ideal, they make significant progress towards the 1-RTT recovery ideal articulated in Section 1. We did not set out to conquer that ideal; over the years, many loss recovery mechanisms have been developed for TCP, and yet, as our measurements show, there was still significant room for improvement. An open question is: is it possible to introduce enough redundancy in TCP (or a clean-slate design) to achieve 1-RTT recovery without adding instability, in order to effectively scale the recovery mechanisms with growing bandwidths?

When should Gentle Aggression be used? A transport's job is to provide a fast pipe to the applications without exposing complexity. In that vein, the level of aggression that makes use of fine grained information like RTT or loss is best decided by TCP – examples are *Reactive* and the fraction of extra *Corrective* packets. At a higher level, the application decides whether to enable *Proactive* or *Corrective*, based on its knowledge of the traffic mix in the network.

Multi-stage connections. Our designs leverage multi-stage Web access to provide different levels of redundancy on client and backend connections. Some of our designs, like *Proactive* (but not *Corrective* or *Reactive*), may become obsolete if Web accesses were engineered differently in the future. We see this as an unlikely event: we believe the relationship between user perceived latency and revenue is fundamental and becoming increasingly important with the use of mobile devices, and so the large, popular Web service providers will always have incentive to build out backbones in order to engineer low latency.

[14] The connection parameters are similar to the ones used by `http://www.webpagetest.org`.



(a) Completion time for Craigslist (b) Completion time for New York Times (c) Render start time for New York Times

Figure 10: CDFs ($y \geq 0.7$) for Web site downloads on a desktop client with a cable connection and a correlated loss pattern. The first label on each x-axis describes the ideal latency observed in a no-loss scenario.

Loss patterns. We based the designs of our TCP enhancements on loss patterns observed in today's Internet. How likely is it that these loss patterns will persist in the future? First, we note that at least one early study pointed out that a significant number (56%) of recoveries incurred RTOs [7], so at least one of our findings appears to have existed over a decade and a half ago. Second, networks that use network-based congestion management, flow isolation, Explicit Congestion Notification, and/or QoS can avoid most or all loss for latency critical traffic. Such networks exist but are rare in the public Internet. In such environments, tail losses may be less common, making the mechanisms in this paper less useful. In these settings, *Reactive* is not detrimental since it responds only on an impending timeout, and *Corrective* can also be adapted to have this property. So long as there is loss, these techniques help trim the tail of the latency distribution, and *Proactive* could still be used in targeted environments. Moreover, while such AQM deployments have been proposed over the decades, history suggests that we are still many years away from a loss-free Internet.

Coexistence with legacy TCP. In our large scale experiments with *Reactive* and *Proactive*, clients were served with a mix of experiment and baseline traffic. We monitored the baseline with and without the experiment and observed no measurable difference between the two. This is not surprising: even though *Proactive* doubles the traffic, it does so for a small fraction of traffic without creating instabilities. Likewise, the fraction of traffic increased by *Reactive* is smaller than 0.5% – comparable to connection management control traffic. Both *Reactive* and *Corrective*, which we recommend using over the public Internet, are well-behaved in that they appropriately reduce the congestion window upon a loss event even if the lost packet is recovered. *Corrective* increases traffic by an additional 10%, similar to adding a new flow(s) on the link; since emulation is unlikely to give an accurate assessment of the impact of this overhead on legacy TCP, we plan to evaluate this in future work using a large-scale deployment of *Corrective*.

10. RELATED WORK

The study of TCP loss recovery in real networks is not new [7, 25, 36, 40]. Measurements from 1995 showed that 85% of timeouts were due to insufficient duplicate ACKs to trigger Fast Retransmit [25], and 75% of retransmissions happened during timeout recovery. A study of the 1996 Olympic Web servers estimated that SACK might only eliminate 4% of timeouts [7]. The authors invented *limited transmit*, which was standardized [5] and widely deployed. An analysis of the Coral CDN service identified loss recovery as one of the major performance bottlenecks [40].

Similarly, improving loss recovery is a perennial goal, and such improvements fall into several broad categories: better strategies for managing the window during recovery [7, 20, 29], detecting and compensating for spurious retransmissions triggered by reordering [10, 27], disambiguating loss and reordering at the end of a stream [39], and improving the retransmit timer estimation.

TCP's slow RTO recovery is known to be a bottleneck. For example, Griwodz and Halvorsen showed that repeated long RTOs are the main cause of game unresponsiveness [17]. Petlund et al. [31] propose to use a linear RTO, which has been incorporated in the Linux kernel as a non-default socket option for "thin" streams. This approach still relies on receiving duplicate ACKs and does not address RTOs resulting from tail losses. Mondal and Kuzmanovic further argue that exponential RTO backoff should be removed because it is not necessary for the stability of Internet [30]. In contrast, *Reactive* does not change the RTO timer calculation or exponential backoff and instead leaves the RTO conservative for stability but sends a few probes before concluding the network is badly congested. F-RTO reduces the number of spurious timeout retransmissions [37]. It is enabled by default in Linux, and we used it in all our experiments. F-RTO has close to zero latency impact in our end-user benchmarks, because it is rarely triggered. It relies on availability of new data to send on timeout, but typically tail losses happen at the end of an HTTP or RPC-type response. *Reactive* does not require new data and hence does not have this limitation. Early Retransmit [4] reduces timeouts when a connection has received a certain number of duplicate ACKs. F-RTO and Early Retransmit are both complementary to *Reactive*.

In line with our approach, Vulimiri et al. [44] make a case for the use of redundancy in the context of the wide-area Internet as an effective way to convert a small amount of extra capacity into reduced latency. RPT introduces redundancy-based loss protection with low traffic overhead in content-aware networks [19]. Studies targeting low-latency datacenters aim to reduce the long tail of flow completion times by reducing packet drops [46, 3]. However their design assumptions preclude their deployment in the Internet.

Applying FEC to transport (at nearly every layer) is an old idea. Sundararajan et al. [41] suggested placing network coding in TCP, and Kim et al. [23] extended this work by implementing a variant over UDP while mimicking TCP capabilities to applications (mainly for high-loss wireless environments). Among others, Baldantoni et al. [9] and Tickoo et al. [43] explored extending TCP to incorporate FEC. None of these, to our knowledge address the issues faced when building a real kernel implementation with today's TCP stack, nor do they address middleboxes tampering with packets. Finally, Maelstrom is an FEC variant for long-range communication between data centers leveraging the benefits of combining and encoding data from multiple sources into a single stream [8].

11. CONCLUSION

Ideally packet loss recovery would take no more than one RTT. We are far from this ideal with today's TCP loss recovery. *Reactive*, *Corrective* and *Proactive* are simple, practical, easily deployable, and immediately useful mechanisms that progressively move us closer to this ideal by judiciously adding redundancy. In some cases, they can reduce 99th percentile latency by 35%. *Reactive* is enabled by default in mainline Linux. Our plan is to also integrate the remaining mechanisms in mainline operating systems such as Linux, with the aim of making the Web faster.

Acknowledgments We thank our shepherd, Sachin Katti, and the reviewers for their helpful comments, Matt Mathis for his insightful discussions on TCP's loss recovery as well as Brad Dux, Mario Fanelli, Pavlos Papageorge, and Tina Wong for their help in deploying *Proactive*. Tobias Flach and Ethan Katz-Bassett performed some of this work while employed temporarily at Google. This work was funded in part by the National Science Foundation (NSF) under grant number 905596. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

12. REFERENCES

[1] Web Page Replay. `http://code.google.com/p/web-page-replay/`.

[2] Akamai. The State of the Internet (3rd Quarter 2012), 2012. `http://www.akamai.com/stateoftheinternet/`.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of SIGCOMM*, 2010.

[4] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. Early retransmit for TCP and SCTP, May 2010. RFC 5827.

[5] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit, January 2001. RFC 3042.

[6] M. Allman, V. Paxson, and E. Blanton. TCP congestion control, September 2009. RFC 5681.

[7] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *Proc. of INFOCOM*, 1998.

[8] M. Balakrishnan, T. Marian, K. P. Birman, H. Weatherspoon, and L. Ganesh. Maelstrom: transparent error correction for communication between data centers. *IEEE/ACM Trans. Netw.*, 19(3), June 2011.

[9] L. Baldantoni, H. Lundqvist, and G. Karlsson. Adaptive end-to-end FEC for improving TCP performance over wireless links. In *Proc. of Conf. on Commun.*, June 2004.

[10] E. Blanton and M. Allman. Using TCP DSACKs and SCTP duplicate TSNs to detect spurious retransmissions, February 2004. RFC 3708.

[11] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP, 2012. RFC 6675.

[12] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *ACM Comput. Commun. Rev.*, August 1996.

[13] M. Carbone and L. Rizzo. Dummynet revisited. *ACM Comput. Commun. Rev.*, 40(2), 2010.

[14] N. Dukkipati. tcp: Tail Loss Probe (TLP). `http://lwn.net/Articles/542642/`.

[15] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses, Feburary 2013. draft-dukkipati-tcpm-tcp-loss-probe-01.

[16] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *ACM Comput. Commun. Rev.*, 40, 2010.

[17] C. Griwodz and P. Halvorsen. The fun of using TCP for an MMORPG. In *Proc. of NOSSDAV*, 2006.

[18] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5), July 2008.

[19] D. Han, A. Anand, A. Akella, and S. Seshan. RPT: Re-architecting Loss Protection for Content-Aware Networks. In *Proc. of NSDI*, 2012.

[20] J. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. *ACM Comput. Commun. Rev.*, August 1996.

[21] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. of IMC*, 2011.

[22] A. Hughes, J. Touch, and J. Heidemann. Issues in TCP Slow-Start Restart after Idle, December 2001. draft-hughes-restart-00.

[23] M. Kim, J. Cloud, A. ParandehGheibi, L. Urbina, K. Fouli, D. Leith, and M. Medard. Network Coded TCP (CTCP). arXiv:1212.2291.

[24] R. Krishnan, H. V. Madhyastha, S. Jain, S. Srinivasan, A. Krishnamurthy, T. Anderson, and J. Gao. Moving Beyond End-to-End Path Information to Optimize CDN Performance. In *Proc. of IMC*, 2009.

[25] D. Lin and H. Kung. TCP fast recovery strategies: Analysis and improvements. In *Proc. of INFOCOM*, 1998.

[26] G. Linden. Make Data Useful. `http://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt`, 2006.

[27] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *(ACM) Comp. Commun. Rev.*, 30(1), January 2000.

[28] M. Mathis. Relentless Congestion Control, March 2009. draft-mathis-iccrg-relentless-tcp-00.txt.

[29] M. Mathis and J. Mahdavi. Forward acknowledgment: refining TCP congestion control. *ACM Comput. Commun. Rev.*, 26(4), August 1996.

[30] A. Mondal and A. Kuzmanovic. Removing exponential backoff from TCP. *ACM Comput. Commun. Rev.*, 38(5), September 2008.

[31] A. Petlund, K. Evensen, C. Griwodz, and P. Halvorsen. TCP enhancements for interactive thin-stream applications. In *Proc. of NOSSDAV*, 2008.

[32] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of CoNEXT*, 2011.

[33] B. Raghavan and A. Snoeren. Decongestion Control. In *Proc. of HotNets*, 2006.

[34] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP, September 2001. RFC 3042.

[35] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journ. of the Soc. for Industr. and Appl. Math.*, 8(2), jun 1960.

[36] S. Rewaskar, J. Kaur, and F. D. Smith. A performance study of loss detection/recovery in real-world TCP implementations. *Proc. of ICNP*, 2007.

[37] P. Sarolahti, M. Kojo, K. Yamamoto, and M. Hata. Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP, September 2009. RFC 5682.

[38] P. Sarolahti and A. Kuznetsov. Congestion Control in Linux TCP. In *Proc. of USENIX*, 2002.

[39] R. Scheffenegger. Improving SACK-based loss recovery for TCP, November 2010. draft-scheffenegger-tcpm-sack-loss-recovery-00.txt.

[40] P. Sun, M. Yu, M. J. Freedman, and J. Rexford. Identifying Performance Bottlenecks in CDNs through TCP-Level Monitoring. In *SIGCOMM Workshop on Meas. Up the Stack*, August 2011.

[41] J. Sundararajan, D. Shah, M. Medard, S. Jakubczak, M. Mitzenmacher, and J. Barros. Network Coding Meets TCP: Theory and Implementation. *Proc. of the IEEE*, 99(3), March 2011.

[42] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband Internet Performance: A View from the Gateway. *ACM Comput. Commun. Rev.*, 41(4), 2011.

[43] O. Tickoo, V. Subramanian, S. Kalyanaraman, and K. Ramakrishnan. LT-TCP: End-to-End Framework to improve TCP Performance over Networks with Lossy Channels. In *Proc. of IWQoS*, 2005.

[44] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is less: reducing latency via redundancy. In *Proc. of HotNets*, 2012.

[45] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *Proc. of SIGCOMM*, 2006.

[46] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: reducing the flow completion time tail in datacenter networks. In *Proc. of SIGCOMM*, 2012.