Recursively Cautious Congestion Control

Radhika Mittal[*] *Justine Sherry*[*] *Sylvia Ratnasamy*[*] *Scott Shenker*[*†]

[*]*UC Berkeley* [†]*ICSI*

Abstract

TCP's congestion control is deliberately cautious, avoiding network overloads by starting with a small initial window and then iteratively ramping up. As a result, it often takes flows several round-trip times to fully utilize the available bandwidth. In this paper we propose RC3, a technique to quickly take advantage of available capacity from the very first RTT. RC3 uses several levels of lower priority service and a modified TCP behavior to achieve near-optimal throughputs while preserving TCP-friendliness and fairness. We implement RC3 in the Linux kernel and in NS-3. In common wide-area scenarios, RC3 results in over 40% reduction in average flow completion times, with strongest improvements – more than 70% reduction in flow completion time – seen in medium to large sized (100KB - 3MB) flows.

1 Introduction

We begin this paper by noting two facts about networks. First, modern ISPs run their networks at a relatively low utilization [18, 21, 26]. This is not because ISPs are incapable of achieving higher utilization, but because their networks must be prepared for link failures which could, at any time, reduce their available capacity by a significant fraction. Thus, most ISP networks are engineered with substantial headroom, so that ISPs can continue to deliver high-quality service even after failures.

Second, TCP congestion control is designed to be *cautious*, starting from a small window size and then increasing every round-trip time until the flow starts experiencing packet drops. The need for fairness requires that all flows follow the same congestion-control behavior, rather than letting some be cautious and others aggressive. Caution, rather than aggression, is the better choice for a uniform behavior because it can more easily cope with a heavily overloaded network; if every flow started out aggressively, the network could easily reach a congestion-collapsed state with a persistently high packet-drop rate.

These decisions – underloaded networks and cautious congestion control – were arrived at independently, but interact counter-productively. When the network is underloaded, flows will rarely hit congestion at lower speeds. However, the caution of today's congestion control algorithms requires that flows spend significant time ramping up rather than aggressively assuming that more bandwidth is available. In recent years there have been calls to increase TCP's initial window size to alleviate this problem but, as we shall see later in the paper, this approach brings only limited benefits.

In this paper we propose a new approach called *recursively cautious congestion control* (RC3) that retains the advantages of caution while enabling it to efficiently utilize the available bandwidth. The idea builds on a perverse notion of quality-of-service, called WQoS, in which we assume ISPs are willing to offer *worse* service if certain ToS bits are set in the packet header (and the mechanisms for doing so – priority queues, are present in almost all currently deployed routers). While traditional calls for QoS – in which better service is available at a higher price – have foundered on worries about equity (should good Internet service only be available to those who can pay the price?), pricing mechanisms (how do you extract payments for the better service?), and peering (how do peering arrangements cope with these higher-priced classes of service?), in our proposal we are only asking ISPs to make several worse classes of service available that would be treated as regular traffic for the purposes of charging and peering. Thus, we see fewer institutional barriers to deploying WQoS. Upgrading an operational network is a significant undertaking, and we do not make this proposal lightly, but our point is that many of the fundamental sources of resistance to traditional QoS do not apply to WQoS.

The RC3 approach is quite simple. RC3 runs, at the highest priority, the same basic congestion control algorithm as normal TCP. However, it also runs congestion control algorithms at each of the k worse levels of service; each of these levels sends only a fixed number of packets, with exponentially larger numbers at lower priority levels. As a result, all RC3 flows compete fairly at every priority level, and the fact that the highest priority level uses the traditional TCP algorithms ensures that RC3 does not increase the chances of congestion collapse. Moreover, RC3 can immediately "fill the pipe" with packets (assuming there are enough priority levels), so it can leverage the bandwidth available in underutilized networks.

We implemented RC3 in the Linux kernel and in the NS-3 network simulator. We find through experiments on both real and simulated networks that RC3 provides strong gains over traditional TCP, averaging 40% reduction in flow completion times over all flows, with strongest gains – of over 70% – seen in medium to large sized flows.

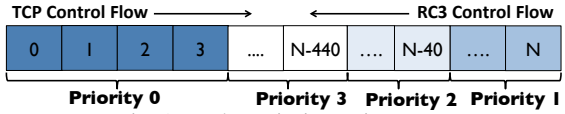


Fig. 1: Packet priority assignments.

2 Design

2.1 RC3 Overview

RC3 runs two parallel control loops: one transmitting at normal priority and obeying the cautious transmission rate of traditional TCP, and a second "recursive low priority" (RLP) control loop keeping the link saturated with low priority packets.

In the primary control loop, TCP proceeds as normal, sending packets in order from index 0 in the byte stream, starting with slow-start and then progressing to normal congestion-avoidance behavior after the first packet loss. The packets sent by this default TCP are transmitted at 'normal' priority – priority 0 (with lower priorities denoted by higher numbers).

In the RLP control loop, the sender transmits additional traffic from the same buffer as TCP to the NIC.[1] To minimize the overlap between the data sent by the two control loops, the RLP sender starts from the very *last* byte in the buffer rather than the first, and works its way towards the beginning of the buffer, as illustrated in Figure 1. RLP packets are sent at low priorities (priority 1 or greater): the first 40 packets (from right) are sent at priority 1; the next 400 are sent at priority 2; the next 4000 at priority 3, and so on.[2] The RLP traffic can only be transmitted when the TCP loop is not transmitting, so its transmission rate is the NIC capacity minus the normal TCP transmission rate.

RC3 enables TCP selective ACK (SACK) to keep track of which of low priority (and normal priority) packets have been accepted at the receiver. When ACKs are received for low priority packets, no new traffic is sent and no windows are adjusted. The RLP control loop transmits each low priority packet once and once only; there are no retransmissions. The RLP loop starts sending packets to the NIC as soon as the TCP send buffer is populated with new packets, terminating when its 'last byte sent' crosses with the TCP loop's 'last byte sent'. Performance gains from RC3 are seen only during the slow-start phase; for long flows where TCP enters congestion avoidance, TCP will keep the network maximally utilized with priority 0 traffic, assuming appropriately sized buffers [8]. If the bottleneck link is the edge link,

[1] As end-hosts support priority queueing discipline, this traffic will never pre-empt the primary TCP traffic.

[2] RC3 requires the packets to be exponentially divided across the priority levels to accommodate large flows within feasible number of priority bits. The exact number of packets in each priority level has little significance, as we shall see in § 5.1.2.

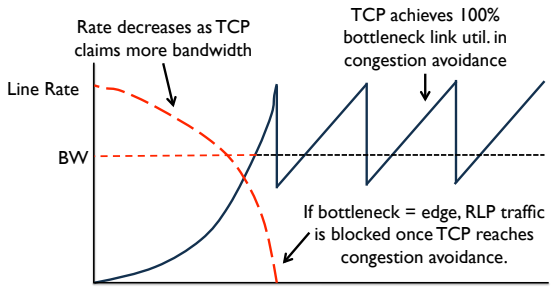


Fig. 2: Congestion window and throughput with RC3.

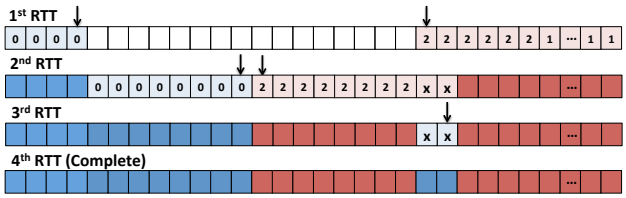


Fig. 3: Example RC3 transmission from §2.2.

high priority packets will pre-empt any packets sourced by the RLP directly at the end host NIC; otherwise the low priority packets will be dropped elsewhere in the network.

Figure 2 illustrates how the two loops interact: as the TCP sender ramps up, the RLP traffic has less and less 'free' bandwidth to take advantage of, until it eventually is fully blocked by the TCP traffic. Since the RLP loop does not perform retransmissions, it can leave behind 'holes' of packets which have been transmitted (at low priority) but never ACKed. Because RC3 enables SACK, the sender knows exactly which segments are missing and the primary control loop retransmits only those segments.[3] Once the TCP 'last byte sent' crosses into traffic that has already been transmitted by the RLP loop, it uses this information to retransmit the missing segments and ensure that all packets have been received. We walk through transmission of a flow with such a 'hole' in the following subsection.

2.2 Example

We now walk through a toy example of a flow with 66 packets transmitted over a link with an edge-limited delay-bandwidth product of 50 packets. Figure 3 illustrates our example.

In the first RTT, TCP sends the first 4 packets at priority 0 (from left); after these high priority packets are transmitted, the RLP loop sends the remaining 62 packets to the NIC – 40 packets at priority 1 and 22 packets at priority 2 (from right), of which 46 packets are transmitted by the NIC (filling up the entire delay-bandwidth product of 50 packets per RTT).

The 21st and 22nd packets from the left (marked as

[3] Enabling SACK allows selective retransmission for dropped low priority packets. However, RC3 still provides significant performance gains when SACK is disabled, despite some redundant retransmissions.

Xs), sent out at priority 2, are dropped. Thus, in the second RTT, ACKs are received for all packets transmitted at priority 0 and for all but packets 21 and 22 sent at lower priorities. The TCP control loop doubles its window and transmits an additional 8 packets; the RLP sender ignores the lost packets and the remaining packets are transmitted by the NIC at priority 2.

In the third RTT, the sender receives ACKs for all packets transmitted in the second RTT and TCP continues to expand its window to 16 under slow start. At this point, the TCP loop sees that all packets except 21st and 22nd have been ACKed. It, therefore, transmits only these two packets.

Finally, in the fourth RTT the sender receives ACKs for the 21st and 22nd packets as well. As all data acknowledgements have now been received by the sender, the connection completes.

3 Performance Model

Having described RC3 in §2, we now model our expected reduction in Flow Completion Time (FCT) for a TCP flow using RC3 as compared to a basic TCP implementation. We quantify gains as ((FCT with TCP) - (FCT with RC3)) / (FCT with TCP) – *i.e.* the percentage reduction in FCT [11]. Our model is very loose and ignores issues of queuing, packet drops, or the interaction between flows. Nonetheless, this model helps us understand some of the basic trends in performance gains. We extensively validate these expected gains in §5 and see the effects of interaction with other flows.

Basic Model: Let BW be the capacity of the bottleneck link a flow traverses, and u be the utilization level of that link. We define A, the available capacity remaining in the bottleneck link as $A = (1-u) \times BW$. Since RC3 utilizes *all* of the available capacity, a simplified expectation for FCTs under RC3 is $RTT + \frac{N}{A}$, where RTT is the round trip time and N is the flow size.

TCP does not utilize all available capacity during its slow start phase; it is only once the congestion window grows to $A \times RTT$, that the link is fully utilized. The slow start phase, during which TCP leaves the link partially idle, lasts $log(min(N, A \times RTT)/i)$ RTTs, with i being the initial congestion window of TCP. This is the interval during which RC3 can benefit TCP.

In Figure 4, the solid line shows our expected gains according to our model. Recall that i denotes the initial congestion window under TCP. For flow sizes $N < i$, RC3 provides no gains over a baseline TCP implementation, as in both scenarios the flow would complete in $RTT + \frac{N}{A}$. For flow sizes $i < N < A \times RTT$, the flow completes in 1 RTT with RC3, and $log(N/i)$ RTTs with basic TCP in slow start. Consequently, the reduction in FCT increases with N over this interval.

Once flow sizes reach $N > A \times RTT$, basic TCP reaches a state where it can ensure 100% link utilization after $log(A \times RTT/i)$ RTTs. Therefore, the improvements from RC3 become a smaller fraction of overall FCT with increasingly large flows; this reduction roughly follows $\frac{log(A \times RTT/i) \times RTT \times A}{N}$ (ignoring a few constants in the denominator).

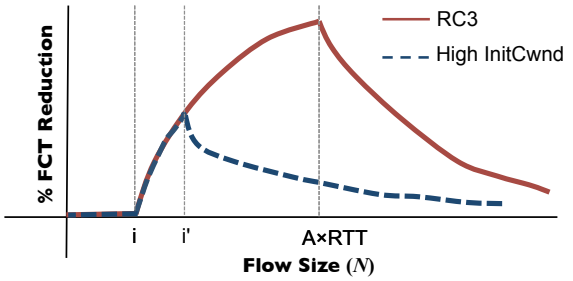


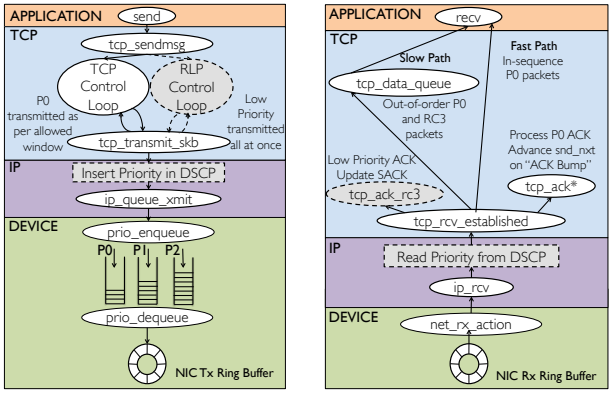
Fig. 4: Performance gains as predicted by a simple model for RC3 and an increased initial congestion window.

Parameter Sensitivity: The above model illustrates that improvements in FCTs due to RC3 are dependent primarily on three parameters: the flow size (N), the effective bandwidth-delay product ($A \times RTT$), and the choice of the initial congestion window (i). Peak improvements are observed when N is close to $A \times RTT$, because under these conditions the flow completes in 1 RTT with RC3 and spends its entire life time in slow start without RC3. When the delay-bandwidth product increases, both the optimal flow size (for performance improvement) increases, and the maximum improvement increases.

Adjusting *i*: There are several proposals [7, 12] to adjust the default initial congestion window in TCP to 10 or even more packets. Assume we adjusted a basic TCP implementation to use a new value, some i' as its initial congestion window. The dotted line in Figure 4 illustrates the gains from such an i'. When i' increases, the amount of time spent in slow start decreases to $log(min(N, A \times RTT)/i') \times RTT$. Flows of up to i' packets complete in a single RTT, but unless $i' = A \times RTT$ (hundreds of packets for today's WAN connections), adjusting the initial congestion window will always underperform when compared to RC3. However, there is good reason not to adjust i' to $A \times RTT$: without the use of low priorities, as in RC3, sending a large amount of traffic without cautious probing can lead to an increase in congestion and overall *worse* performance. Our model does not capture the impact of queueing and drops, however, in §5.1.4 we show via simulation how increasing the initial congestion window to 10 and 50 packets penalizes small flows in the network.

4 Linux Implementation

We implemented RC3 as an extension to the Linux 3.2 kernel on a server with Intel 82599EB 10Gbps NICs.



(a) Sending Data (b) Receiving Data and Acks

Fig. 5: Modifications to Linux kernel TCP stack.

Our implementation cleanly sits within the TCP and IP layers and requires minimal modifications to the kernel source code. Because our implementation does not touch the core TCP congestion control code, different congestion control implementations can easily be 'swapped out' while still retaining compatibility with RC3. After describing our implementation in §4.1, we discuss how RC3 interacts with other components of the networking stack in §4.2, including application buffering, QoS support, hardware performance optimizations, and SACK extensions.

4.1 TCP/IP in the Linux Kernel

We briefly provide high-level context for our implementation, describing the TCP/IP stack under the Linux 3.2 kernel. We expect that RC3 can be easily ported to other implementations and operating systems as well, but leave this task to future work.

Figure 5 illustrates the kernel TCP/IP architecture at a very high level, along with our RC3 extensions shaded in gray. The Linux kernel is as follows. When an application calls *send()*, the *tcp_sendmsg* function is invoked; this function segments the send buffer in to 'packets' represented by the socket buffer (*skb*) datastructure. By default, each *skb* represents one packet to be sent on the wire. After the data to be transmitted has been segmented in to *skb*s, it is passed to the core TCP logic, and then forwarded for transmission through the network device queue to the NIC.

On the receiver side, packets arrive at the NIC and are forwarded up to a receive buffer at the application layer. As packets are read in, they are represented once again as *skb* datatypes. Once packets are copied to *skb*s, they are passed up the stack through the TCP layer. Data arriving in-order is sent along the 'fast-path', directly to the application layer receive buffer. Data arriving out of order is sent along a 'slow path' to an out of order receive queue, where it waits for the missing packets to arrive, before being forwarded up in-order to the application layer.

We now describe how we extend these functions to support RC3.

4.1.1 Sending Data Packets

RC3 extends the send-side code in the networking stack with two simple changes, inserting only 72LOC in the TCP stack and 2LOC in the IP stack. The first change, in the TCP stack, is to invoke the RLP control loop once the data has been segmented in the *tcp_sendmsg* function. We leave all of the segmentation and core TCP logic untouched – we merely add a function call in *tcp_sendmsg* to invoke the RLP loop, as shown in Fig. 5.

The RLP loop then reads the TCP write queue iteratively from the tail end, reading in the packets one by one, marking the appropriate priority in the packet buffer, and then sending out the packet. The field *skb→priority* is assigned according to the sequence number of the packet: the RLP loop subtracts the packet's sequence number from the tail sequence number and then divides this value by the MSS. If this value is ≤ 40, the packet is assigned priority 1, if the value is ≤ 400 it is assigned priority 2, and so on. After the priority assignment, the *skb* packets are forwarded out via the *tcp_transmit_skb* function.

Our second change comes in the IP layer as packets are attached to an IP header; where we ensure that *skb→priority* is not overwritten by the fixed priority assigned to the socket, as in the default case. Instead, the value of *skb→priority* is copied to the DSCP priority bits in the IP header.

Overall, our changes are lightweight and do not interfere with core congestion control logic. Indeed, because the TCP logic is isolated from the RC3 code, we can easily enable TCP CUBIC, TCP New Reno, or any other TCP congestion control algorithms to run alongside RC3.

4.1.2 Receiving Data Packets and ACKs

Extending the receive-side code with RC3 is similarly lightweight and avoids modifications to the core TCP control flow. Our changes here comprise only of 46 LOC in the TCP stack and 1 LOC in the IP stack.

Starting from bottom to top in Figure 5, our first change comes as packets are read in off the wire and converted to *skb*s – here we ensure that the DSCP priority field in the IP header is copied to the *skb* priority field; this change is a simple 1 LOC edit.

Our second set of changes which lie up the stack within TCP. These changes separate out low priority packets from high priority in order to ensure that the high priority ACKing mechanism (and therefore the sender's congestion window) and other TCP variables remain unaffected by the low priority packets. We identify the low priority packets and pass them to the

out of order 'slow path' queue, using the unmodified function *tcp_data_queue*. We then call a new function, *tcp_send_ack_rc3*, which sends an ACK packet for the new data at the same priority the data arrived on, with the cumulative ACK as per the high priority traffic, but SACK tags indicating the additional low priority packets received. The priority is assigned in the field *skb→priority*, and the packets are sent out by calling *tcp_transmit_skb*, as explained in § 4.1.1.

The other modifications within the TCP receive code interpose on the handling of ACKs. We invoke *tcp_ack_rc3* on receiving a low priority ACK packet, which simply calls the function to update the SACK scoreboard (which records the packets that have been SACKed), as per the SACK tags carried by the ACK packet. We also relax the SACK validation criteria to update the SACK "scoreboard" to accept SACKed packets beyond *snd_nxt*, the sequence number up to which data has been sent out by the TCP control loop.

Typically when a new ACK is received, the stack double-checks that the received ACK is at a value less than *snd_nxt*, discarding the ACKs that do not satisfy this constraint. We instead tweak the ACK processing to update the *snd_nxt* value when a high-priority ACK is received for a sequence number that is greater than *snd_nxt*: such an ACK signals that the TCP sender has "crossed paths" with traffic transmitted by the RLP and is entering the cleanup phase. We advance the send queue's head and update *snd_nxt* to the new ACKed value and then allow TCP to continue as usual; we call this jump an "ACK bump."

While these changes dig shallowly in to the core TCP code, they do not impact our compatibility with various congestion control schemes.

4.2 Specific Implementation Features

We now discuss how RC3 interacts with some key features at all layers of the networking stack, from software to NIC hardware to routers and switches.

4.2.1 Socket Buffer Sizes

The default send and receive buffer sizes in Linux are very small - 16KB and 85KB respectively. Performance gains from RC3 are maximized when the entire flow is sent out in the first RTT itself. This requires us to make the send and receive buffers as big as the maximum flow size (up to a few MBs in most cases). Window scaling is turned on by default in Linux, and hence we are not limited by the 64KB receive window carried by the TCP header.

RC3 is nonetheless compatible with smaller send buffer sizes: every call to *tcp_sendmsg* passes a chunk of data to the RLP control loop, which treats that chunk, logically, as a new flow as far as priority assignment is concerned. We include a check to break the RLP control loop to ensure that the same packet is not transmitted twice by subsequent calls to *tcp_sendmsg*. Indeed, this behavior can help flows which resume from an application-layer imposed idle period.

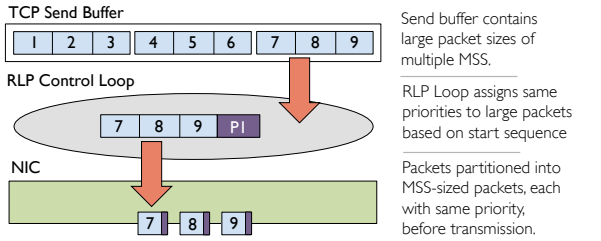


Fig. 6: RC3 combined with TSO.

4.2.2 Using QoS Support

RC3 is only effective if priority queueing is supported at both endhosts and the routers in the network.

Endhosts: We increase the Tx queue length at the software interface, to ensure that it can store all the packets forwarded by the TCP stack. The in-built traffic control functionality of Linux is used to set the queuing discipline (*qdisc*) as *prio* and map the packet priorities to queue 'bands'. The *prio* qdisc maintains priority queues in software, writing to a single NIC ring buffer as shown in Figure 5. Thus, when the NIC is free to transmit, a packet from band N is dequeued only if all bands from 0 to $N-1$ are empty. Up to 16 such bands are supported by the Linux kernel, which are more than enough for RC3.[4]

Routers: All modern routers today support QoS, where flow classes can be created and assigned to a particular priority level. Incoming packets can then be mapped to one of these classes based on the DSCP field. The exact mechanism of doing so may vary across different vendors. Although the ISPs may use the DSCP field for some internal prioritization, all we require them to do is to read the DSCP field of an incoming packet, assign a priority tag to the packet which can be recognized by their routers, and then rewrite the priority in the DSCP field when the packet leaves their network.

4.2.3 Compatibility with TSO/LRO

TCP Segmentation Offload (TSO) and *Large Receiver Offload* (LRO) are two performance extensions within the Linux kernel that improve throughput through batching. TSO allows the TCP/IP stack to send packets comprising of multiple MSSes to the NIC, which then divides them into MSS sized segments before transmitting them on the link. LRO is the receiver counterpart which amasses the incoming segments into larger packets at the driver, before sending them higher up in the stack. TSO and LRO both improve performance by amortizing

[4] 16 priority levels is sufficient to support RC3 flow sizes on the order of a petabyte!

the cost of packet processing across packets in batches. Batched packets reduce the average per-packet processing CPU overhead, consequently improving throughput.

Figure 6 illustrates how RC3 behaves when TSO is enabled at the sender and a larger packet, comprising of multiple MSSes is seen by the RLP control loop. At first glance, TSO stands in the way of RC3: RC3 requires fine-grained control over individual packets to assign priorities, and the over sized packets passing through under TSO hinder RC3's ability to assign priorities correctly when it includes data from packets that should be assigned different priority classes. Rather than partitioning data within these extra large packets, we simply allow RC3 to label them according to the lowest priority of any data in the segment. This means that we might not strictly follow the RC3 design in §2 while assigning priorities for some large packets. However, such a situation can arise only when the MSSes in a large packet overlap with the border at which priority levels switch. Since the traffic is partitioned across priority level exponentially, such cases are infrequent. Further, the largest TSO packet is comprised of at most 64KB. Therefore, no more than 43 packets would be improperly labeled at the border between priority levels.

TSO batching leads to a second deviation from the RC3 specification, in that segments within a large packet are sent in sequence, rather than in reverse order. For example, in Figure 6, the segments in the packet are sent in order (7,8,9) instead of (9,8,7). Hence, although RC3 still processes *skb* packets from tail to front, the physical packets sent on the wire will be sent in short in-order bursts, each burst with a decreasing starting sequence number. Allowing the forward sequencing of packets within a TSO batch turns out to be useful when LRO is enabled at the receiver, where batching at the driver happens *only* if the packets arrive in order. As we'll show in §5.2, combining RC3 with TSO/LRO reduces the OS overhead of processing RC3 packets by almost 50%, and consequently leads to net gains in FCTs.

4.2.4 SACK Enhancements

Although RC3 benefits when SACK is enabled, it is incompatible with some SACK enhancements. Forward Acknowledgment (FACK) [24], is turned on by default in Linux. It estimates the number of outstanding packets by looking at the SACKed bytes. RC3 SACKed packets may lead the FACK control loop to falsely believe that all packets between the highest cumulative ACK received and the lowest SACK received are in flight. We therefore disable FACK to avoid the RC3 SACKed bytes from affecting the default congestion control behavior. Doing so does not penalize the performance in most cases, as the Fast Recovery mechanism continues transmitting unsent data after a loss event has occurred and partial ACKs are received, allowing lost packets to be efficiently detected by duplicate ACKs. DSACK [17] is also disabled, to avoid the TCP control loop from inferring incorrect information about the ordering of packets arriving at the receiver based on RC3 SACKs.

5 Experimental Evaluation

We now evaluate RC3 across several dimensions. In §5.1, we evaluate RC3 extensively using NS-3 simulations - §5.1.1, compares RC3's FCT reductions with the model we described in §3; §5.1.2 and §5.1.3 evaluate RC3's *robustness* and *fairness*, and §5.1.4 compares RC3's FCT reductions relative to other designs. We evaluate our Linux RC3 implementation in §5.2.

5.1 Simulation Based Evaluation

We evaluate RC3 using a wide range of simulation settings. Our primary simulation topology models the Internet-2 network consisting of ten routers, each attached to ten end hosts, with 1Gbps bottleneck bandwidth and 40ms RTT. It runs at 30% average link utilization [18, 21, 26]. The queue buffer size is equal to the delay-bandwidth product (RTT$\times$BW) in all cases, which is 5MB for our baseline. The queues do priority dropping and priority scheduling. All senders transmit using RC3 unless otherwise noted. Flow sizes are drawn from an empirical traffic distribution [6]; with Poisson interarrivals.

For most experiments we present RC3's performance relative to a baseline TCP implementation. Our baseline TCP implementation is TCP New Reno [16] with SACK enabled [9, 25] and an initial congestion window of 4 [7]; maximum segment size is set to 1460 bytes while slow start threshold and advertised received window are set to infinity.

5.1.1 Baseline Simulation

We first investigate the baseline improvements using RC3 and compare them to our modeled results from §3.

Validating the Model: Figure 7 compares the gains predicted by our model (§3) with the gains observed in our simulation. The data displayed is for 1Gbps bottleneck capacity, 40ms average RTT, and 30% load. Error bars plotting the standard deviation across 10 runs are shown; they sit very close to the average. For large flows, the simulated gains are slightly lower than predicted; this is the result of queueing delay which is not included in our model. For small flows – four packets or fewer – we actually see *better* results than predicted by the model. This is due to large flows completing sooner than with regular TCP, leaving the network queues more frequently vacant and thus decreasing average queueing delay for short flows. Despite these variations, the simulated and modeled results track each other quite closely: for all but the smallest flows, we see gains of 40–75%.

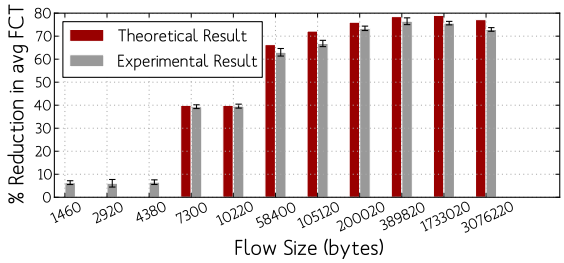
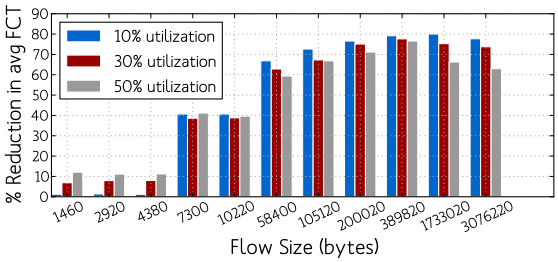


Fig. 7: Reduction in FCT as predicted by model vs simulations. (RTT×BW = 5MB, 30% average link utilization)

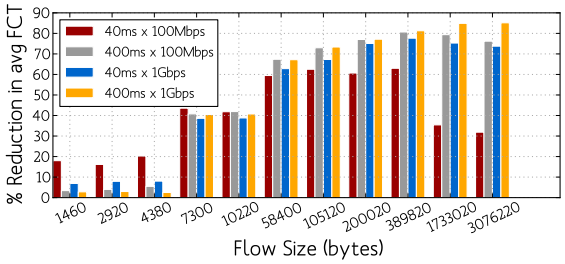


		Average Over Flows	Average Over Bytes
10% Load	Regular FCT (s)	0.125	0.423
	RC3 FCT (s)	0.068	0.091
	% Reduction	**45.56**	**78.36**
30% Load	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
50% Load	Regular FCT (s)	0.15	0.498
	RC3 FCT (s)	0.088	0.176
	% Reduction	**41.44**	**64.88**

Fig. 8: Reduction in FCT with load variation, with RTT×BW fixed at 5MB

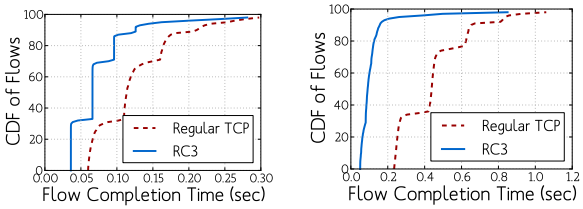
Link Load: Figure 8 shows FCT performance gains comparing RC3 to the base TCP under uniform link load of 10%, 30%, or 50%. RTT×BW is fixed at 5MB across all experiments. As expected, performance improvements decrease for higher average link utilization. For large flows, this follows from the fact that the available capacity ($A = (1-u) \times BW$) reduces with increase in utilization u. Thus, there is less spare capacity to be taken advantage of in scenarios with higher link load. However, for smaller flows, we actually see the opposite trend. This is once again due to reduced average queueing delays, as large flows complete sooner with most packets having lower priorities than the packets from the smaller flows.

RTT×BW: Figure 9 shows the FCT reduction due to RC3 at varying RTT×BW. In this experiment we adjusted RTTs and bandwidth capacities to achieve RTT×BW of 500KB (40ms×100Mbps), 5MB (40ms×1Gbps and 400ms×100Mbps) and 50MB(400ms×1Gbps). As discussed in §3, the performance improvement increases with increasing RTT×BW, as the peak of the curve in Figure 4 shifts towards the right. The opposite trend for very short flows is repeated here as well.



		Average Over Flows	Average Over Bytes
100Mbps Bottleneck 40ms avg. RTT	Regular FCT (s)	0.167	0.691
	RC3 FCT (s)	0.11	0.442
	% Reduction	**33.98**	**36.05**
100Mbps Bottleneck 400ms avg. RTT	Regular FCT (s)	0.948	3.501
	RC3 FCT (s)	0.567	0.783
	% Reduction	**40.29**	**77.62**
1 Gbps Bottleneck 40ms avg. RTT	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
1 Gbps Bottleneck 400ms avg. RTT	Regular FCT (s)	0.971	3.59
	RC3 FCT (s)	0.558	0.569
	% Reduction	**42.45**	**84.17**

Fig. 9: Reduction in average FCT with variation in RTT×BW with 30% average link utilization



(a) Flow size 7.3KB (b) Flow size 1.7MB

Fig. 10: Cumulative Distribution of FCTs. (RTT×BW = 5MB, 30% average link utilization)

Summary: Overall, RC3 provides strong gains, reducing flow completion times by as much as 80% depending on simulation parameters. These results closely track the results predicted by the model presented in §3.

5.1.2 Robustness

In the prior section, we evaluated RC3 within a single context. We now demonstrate that these results are robust, inspecting RC3 in numerous contexts and under different metrics. Many of the results in this section are summarized in Table 1.

Performance at the Tails: Our previous figures plot the average and standard deviation of flow completion times; in Figures 10(a) and (b) we plot the full cumulative distribution of FCTs from our Internet-2 experiments for two representative flow sizes, 7.3KB and 1.7MB.[5] We

[5]The 'jumps' or 'banding' in the CDF are due to the uniform link latencies in the simulation topologies. Paths of two hops had an RTT of 40, paths of three hops had an RTT of 60, and so on. A flow which completes in some k RTTs while still under slow start thus completes

		Average Over Flows	Average Over Bytes
Default: Internet-2	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
Telstra Topology	Regular FCT (s)	0.159	0.510
	RC3 FCT (s)	0.084	0.111
	% Reduction	**47.07**	**78.13**
RedClara Topology	Regular FCT (s)	0.17	0.429
	RC3 FCT (s)	0.097	0.098
	% Reduction	**42.78**	**77.16**
ESNet Topology	Regular FCT (s)	0.207	0.478
	RC3 FCT (s)	0.137	0.0976
	% Reduction	**33.91**	**79.58**
2000 Workload	Regular FCT (s)	0.0871	0.238
	RC3 FCT (s)	0.0704	0.079
	% Reduction	**13.83**	**66.73**
Link Heterogeneity	Regular FCT (s)	0.159	0.541
	RC3 FCT (s)	0.087	0.141
	% Reduction	**45.35**	**73.89**

Table 1: RC3 performance in robustness experiments.

see in these results that performance improvements are provided across the board at all percentiles; even the 1^{st} and 99^{th} percentiles improve by using RC3.

Topology: We performed most of our experiments on a simulation topology based off a simplified model of the Internet-2 network. To verify that our results were not somehow biased by this topology, we repeated the experiment using simulation topologies derived from the Telstra network, the Red Clara academic network, and the complete ESNet [1, 4], keeping the average delay as 40ms and the bottleneck bandwidth as 1Gbps. All three topologies provided results similar to our initial Internet-2 experiments: average FCTs for Telstra improved by 47.07%, for Red Clara, by 42.78%, and for ESNet by 33.91%.

Varying Workload Distribution: Our baseline experiments use an empirical flow size distribution [6]. A noticeable property of the flow size distribution in our experiments is the presence of of very large flows (up to a few MBs) in the tail of the distribution. We repeated the Internet-2 experiment with an empirical distribution from a 2000 [2] study, an era when average flow sizes were smaller than today. Here we saw that the average FCT improved by only 13.83% when averaged over all flows. When averaging FCT gains weighted by bytes, however, we still observe strong gains for large flows resulting in a reduction of 66.73%.

Link Heterogeneity: We now break the assumption of uniform link utilization and capacity: in this experiment we assigned core link bandwidths in the Internet-2 topology to a random value between 500Mbps and 2Gbps. We observed that FCTs in the heterogenous experiment were higher than in the uniform experiment, for both TCP and RC3. Nevertheless, the penalty to TCP was worse, resulting in a stronger reduction in flow completion times, when averaged across flows.

in approximately $k * RTT$ time. This created fixed steps in the CDF, as per the RTTs

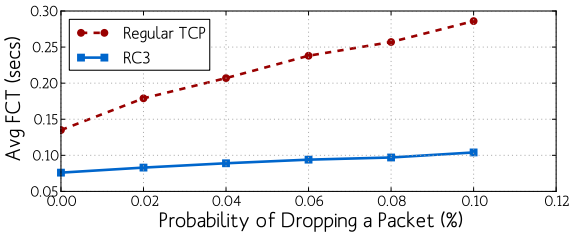
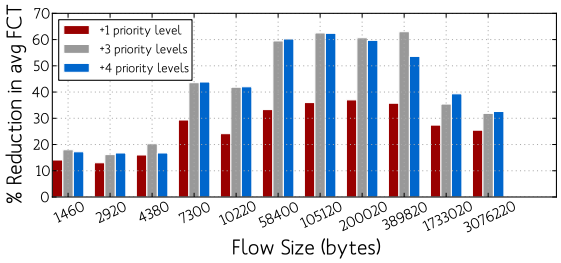


Fig. 11: Average FCTs with increasing arbitrary loss rate. (RTT×BW = 5MB, 30% average link utilization)



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.167	0.691
1 RC3 Priority Level	FCT	0.126	0.496
	%	**24.55**	**28.22**
3 RC3 Priority Levels (40, 400, 4000)	FCT	0.11	0.442
	%	**33.98**	**36.05**
4 RC3 Priority Levels (10, 100, 1000, 10000)	FCT	0.112	0.434
	%	**32.94**	**37.19**

Fig. 12: Reduction in FCT with varying priority levels. (RTT×BW = 500KB, 30% average link utilization)

Loss Rate: Until now all loss has been the result of queue overflows; we now introduce random arbitrary loss and investigate the impact on RC3. Figure 11 shows flow completion times for TCP and RC3 when arbitrary loss is introduced for 0.02-0.1% of packets. We see that loss strongly penalizes TCP flows, but that RC3 flows do not suffer nearly so much as TCP. RC3 provides even stronger gains in such high loss scenarios because each packet essentially has two chances at transmission. Further, since the RLP loop ignores ACKs and losses, low priority losses do not slow the sending rate.

Priority Assignment: Our design assigns packets across multiple priorities, bucketed exponentially with 40 packets at priority 1, 400 at priority 2, and so on. We performed an experiment to investigate the impact of these design choices by experimenting with an RC3 deployment when 1, 3, or 4 additional priority levels were enabled; the results of these experiments are plotted in Fig. 12. We see that dividing traffic over multiple priority levels provides stronger gains than with only one level of low priority traffic. The flows which benefit the most from extra priorities are the medium-sized flows which, without RC3, require more than one RTT to complete during slow start. A very slight difference is seen in

performance gains when bucketing packets as (10, 100, 1000, 10000) instead of (40, 400, 4000).

Application Pacing: Until now, our model application has been a file transfer where the entire contents of the transfer are available for transmission from the beginning of the connection. However, many modern applications 'pace' or 'chunk' their transfers. For example, after an initial buffering phase YouTube paces video transfers at the application layer, transmitting only a few KB of data at a time proportional to the rate that the video data is consumed. To see the effect of RC3 on these type of flows, we emulated a YouTube transfer [30] with a 1.5MB buffer followed by 64KB chunks sent every 100ms. Ultimately, RC3 helped these video connections by decreasing the amount of time spent in buffering by slightly over 70% in our experimental topology. This means that the time between when a user loads the page and can begin video playback decreases while using RC3. However, in the long run, large videos did not complete transferring the entire file any faster with RC3 because their transfer rate is dominated by the 64KB pacing.

Summary: In this section, we examined RC3 in numerous contexts, changing our experimental setup, looking at alternative application models, and investigating the tail distribution of FCTs under RC3 and TCP. In all contexts, RC3 provides benefits over TCP, typically in the range of 30-75%. Even in the worst case context we evaluated, when downlink rather than uplink capacities bottlenecked transmission, *RC3 still outperformed baseline TCP by 10%*.

5.1.3 RC3 and Fairness

In this subsection we ask, *is RC3 fair?* We evaluate two forms of fairness: how RC3 flows of different sizes interact with each other, and how RC3 interacts with concurrent TCP flows.

RC3 with RC3: It is well-known that TCP in the long run is biased in that its bandwidth allocations benefit longer flows over short ones. We calculated the effective throughput for flows using TCP or RC3 in our baseline experiments (Figure 13). TCP achieves near-optimal throughput for flow sizes less than 4 packets, but throughput is very low for medium-sized flows and only slightly increases for the largest (multiple-MB) flows. RC3 maintains substantially high throughput for all flow sizes, having a slight relative bias towards medium sized flows.

RC3 with TCP: To evaluate how RC3 behaves with concurrent TCP flows, we performed an experiment with mixed RC3 and TCP flows running concurrently. We allowed 50% of end-hosts attached to each core router in our simulations (say in set *A*) to use RC3, while the remaining 50% (set *B*) used regular TCP. Overall, FCTs

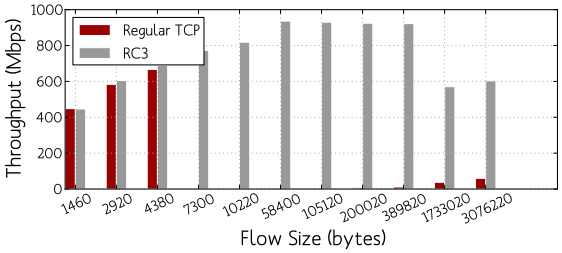


Fig. 13: Median Flow Throughput

for both RC3 and TCP were lower than in the same setup where all flows used regular TCP. Thus, RC3 is not only fair to TCP, but in fact improves TCP FCTs by allowing RC3 flows to complete quickly and 'get out of the way' of the TCP flows.

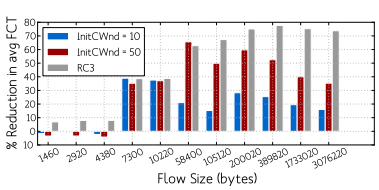
5.1.4 RC3 In Comparison

We now compare the performance gains of RC3 against some other proposals to reduce TCP flow completion times.

Increasing Initial Congestion Window: Figure 14(a) compares the performance gains obtained from RC3 with the performance gains from increasing the baseline TCP's initial congestion window (InitCwnd) to 10 and 50. For most flow sizes, especially larger flows, RC3 provides stronger improvements than simply increasing the initial congestion window. When averaging across all flows, RC3 provides a 44% reduction in FCT whereas increasing the InitCwnd to 10 reduces the FCT by only 13% and 50 reduces it by just 24%. Further, for small flow sizes ($\leq$ 4 packets), increasing the InitCwnd actually introduces a *penalty* due to increased queueing delays. RC3 never makes flows do worse than they would have under traditional TCP. These results confirm our expectations from §3.

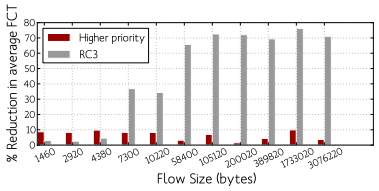
Traditional QoS: An alternate technique to improve FCTs is to designate certain flows as 'critical' and send those flows using unmodified TCP, but at higher priority. We annotated 10% of flows as 'critical'; performance results for the critical flows alone are showed in Fig. 14(b). When the 'critical' 10% of flows simply used higher priority, their average FCT reduces from 0.126 seconds to 0.119 seconds; while the non-critical flows suffered a very slight (<2%) penalty. When we repeated the experiment, but used RC3 for the critical flows (leaving the rest to use TCP), the average FCT reduced from 0.126 seconds to 0.078 seconds, as shown in Figure 14(b) . Furthermore, non-critical flows showed a slight (<1%) *improvement*. This suggests that it is better to be able to send an unrestricted amount of traffic, albeit at low priority, than to send at high priority at a rate limited by TCP.

RCP: Finally, we compare against RCP, an alternative transport protocol to TCP. With RCP, routers calculate average fair rate and signal this to flows; this allows flows



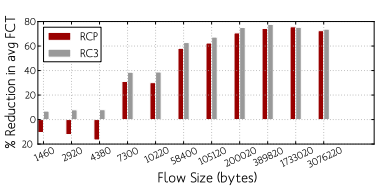
		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.135	0.443
i = 10	FCT	0.117	0.362
	%	**13.21**	**17.87**
i = 50	FCT	0.102	0.272
	%	**24.33**	**38.24**
RC3	FCT	0.076	0.114
	%	**43.54**	**74.35**

(a) Increased InitCwnd



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.126	0.435
QoS	FCT	0.119	0.411
	%	**5.33**	**5.64**
RC3	FCT	0.078	0.12
	%	**38.31**	**72.43**

(b) Traditional QoS



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.135	0.443
RCP	FCT	0.088	0.117
	%	**33.86**	**73.58**
RC3	FCT	0.076	0.114
	%	**43.54**	**74.35**

(c) RCP

Fig. 14: RC3 as compared to three alternatives. All FCTs are reported in seconds; % shows percent reduction from baseline. (RTT×BW = 5MB, 30% average link utilization)

to start transmitting at an explicitly allocated rate from the first (post-handshake) RTT, overcoming TCP's slow start penalty. We show the performance improvement for RCP and RC3 in Fig. 14(c). While for large flows, the two schemes are roughly neck-to-neck, RCP actually imposes a penalty for the very smallest (1-4 packet) flows, in part because RCP's explicit rate allocation enforces *pacing* of packets according to the assigned rate, whereas with traditional TCP (and RC3), all packets are transmitted back to back. These results show that RC3 can provide FCTs which are usually comparable or even better than those with RCP. Further, as RC3 can be deployed on legacy hardware and is friendly with existing TCP flows, it is a more deployable path to comparable performance improvements.

Summary: RC3 outperforms traditional QoS and increasing the initial congestion windows; performance with RC3 is on par with RCP without requiring substantial changes to routers.

5.2 Evaluating RC3 Linux Implementation

We now evaluate RC3 using our implementation in the Linux kernel. We extended the Linux 3.2 kernel as described in §4. We did our baseline experiments using both New Reno and CUBIC congestion control mechanisms. We set the send and receive buffer sizes to 2GB, to ensure that an entire flow fits in a single window. We keep the default initial congestion window of 10 [12] in the kernel unchanged.

Our testbed consists of two servers each with two 10Gbps NICs connected to a 10Gbps Arista datacenter switch. As the hosts were physically adjacent, we used *netem* to increase the observed link latency to 10ms, which reflects a short WAN latency.

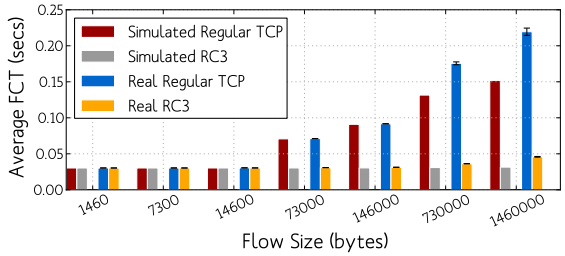
Baseline: Flows with varying sizes were sent from one machine to another. Figure 15(a) shows FCTs with RC3 and baseline TCP implementation in Linux compared to RC3 and baseline TCP NS-3 simulations (with the initial congestion windows set to 10), both running with 10Gbps bandwidth and 20ms RTT. The figure reflects averages over 100 samples.

Overall, RC3 continues to provide strong gains over the baseline TCP design, however, our results in implementation do not tightly match our simulated results from NS. The baseline TCP implementation in Linux performs *worse* than in simulation because of delayed ACK behavior in Linux: when more than two segments are ACKed together, it still only generates an increase in congestion window proportional to two packets being ACKed. This slows down the rate at which the congestion window can increase. The RC3 FCT is slightly higher in Linux than in simulation for large flows because of the extra per-packet overhead in receiving RC3 packets: recall from §4 that RC3 packets are carried over the Linux 'slow path' and thus have slightly higher per-packet overhead.

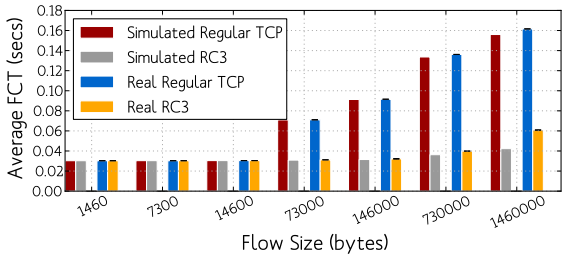
In Figure 15(b), we repeat the same experiment with only 1Gbps bandwidth set by the token bucket filter queueing discipline (retaining 10ms latency through *netem*). In this experiment, results track our simulations more closely. TCP deviates little from the baseline because the arrival rate of the packets ensures that at most two segments are ACKed by the receiver via delayed ACK, and thus the congestion window increases at the correct rate. Overall, we observe that RC3 in implementation continues to provide gains proportional to what we expect from simulation.

While these graphs show the result for New Reno, we repeated these experiments using TCP CUBIC and the FCTs matched very closely to New Reno, since both have the same slow start behavior.

Endhost Correctness: Priority queueing is widely deployed in the OS networking stack, NICs, and routers,

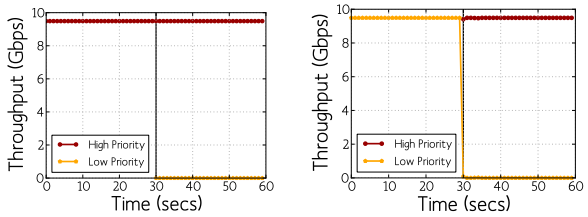


(a) 20ms×10Gbps



(b) 20ms×1Gbps

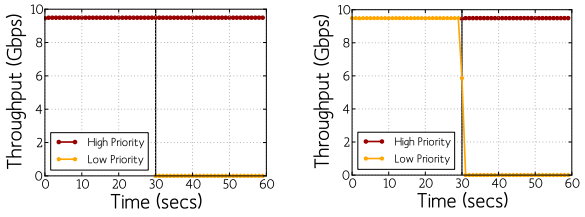
Fig. 15: FCTs for implementation vs. simulation



(a) Low Priority Starts after High Priority

(b) High Priority Starts after Low Priority

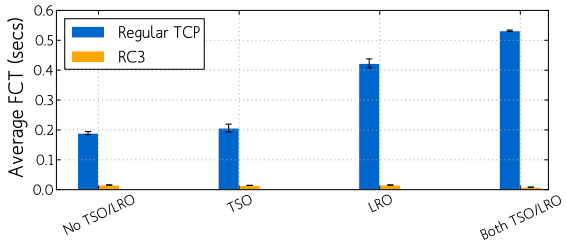
Fig. 16: Correctness of Priority Queuing in Linux



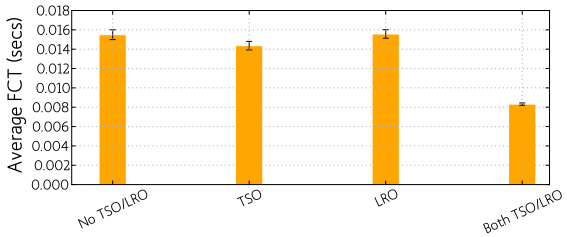
(a) Low Priority Starts after High Priority

(b) High Priority Starts after Low Priority

Fig. 17: Correctness of the Priority Queuing in the Switch



(a) Comparing FCTs for Regular TCP with RC3



(b) Zooming in to observe trends for RC3 FCT

Fig. 18: FCTs for Regular TCP and RC3 with TSO/LRO (20ms×10Gbps)

but is often unused. We now verify the correctness of the prio queueing discipline in Linux. We performed our experiments with iPerf [3] using the default NIC buffer size of 512 packets and with segment offload enabled to achieve a high throughput of 9.5Gbps. All packets in an iPerf flow were assigned the same priority level – this experiment *does not* use RC3 itself. All flows being sent to a particular destination port were marked as priority 1 by changing the DSCP field in the IP header. We connected two endhosts directly, with one acting as the iPerf client sending simultaneous flows to the connected iPerf server (a) with a low priority flow beginning after a high priority flow has begun, and (b) with a high priority flow beginning after a low priority flow has begun. Figure 16 shows that the priority queueing discipline behaves as expected.

Switch Correctness: We extended our topology to connect three endhosts to the switch, two of which acted as iPerf clients, sending simultaneous flows as explained above to the third endhost acting as the iPerf server. Since the two senders were on different machines, prioritization was done by the router. Figure 17 shows that priority queueing at the switch behaves as expected.

Segment and Receiver Offload: In §4.2.3 we discussed how RC3 interacts with segment and receiver offload; we now evaluate the performance of RC3 when combined with these optimizations. For this experiment, we used the same set up, as our baseline and sent a 1000 packet flow without TSO/LRO, with each enabled independently, and with both TSO and LRO enabled. Figure 18 plots the corresponding FCTs excluding the connection set-up time.

For baseline TCP, we see that TSO and LRO each cause a performance penalty in our test scenario. TSO hurts TCP because the increased throughput also increases the number of segments being ACKed with one single delayed ACK, thus slowing the rate at which the congestion window increases. LRO aggravates the same problem by coalescing packets in the receive queue, once again leading them to be ACKed as a batch.

In contrast, RC3's FCTs improve when RC3 is combined with TSO and LRO. TSO and LRO do little to change the performance of RC3, when enabled independently, but when combined they allow chunks of packets to be processed together in batches at the receiver. This reduces the overhead of packet processing by almost 50%, resulting in better overall FCTs.

6 Discussion

Deployment Incentives: For RC3 to be widely used requires ISPs to opt-in by enabling the priority queueing that already exists in their routers. As discussed in the introduction, we believe that giving worse service, rather than better service, for these low priority packets alleviates some of the concerns that has made QoS so hard to offer (in the wide area) today. WQoS is safe and backwards compatible because regular traffic will never be penalized and pricing remains unaffected. Moreover, since RC3 makes more efficient use of bandwidth, it allows providers to run their networks at higher utilization, while still providing good performance, resulting in higher return in investment for their network provisioning.

Partial Support: Our simulations assume that all routers support multiple priorities. If RC3 is to be deployed, it must be usable even when the network is in a state of partial deployment, where some providers but not all support WQoS. When traffic crosses from a network which supports WQoS to a network which does not, a provider has two options: either drop all low priority packets before they cross in to the single-priority domain (obviating the benefits of RC3), or allow the low priority packets to pass through (allowing the packets to subsequently compete with normal TCP traffic at high priority). Simulating this latter scenario, we saw that average FCTs still improved for all flows, from using RC3 when 20% of routers did not support priorities; when 50% of routers did not support priorities small flows experienced a 6-7% FCT penalty, medium-sized flows saw slightly weaker FCT reductions (around 36%), and large flows saw slightly stronger FCT reductions (76-70%).

Middleboxes: Middleboxes which keep tight account of in-flight packets and TCP state are a rare but growing attribute of today's networks. These devices directly challenge the deployment of new protocols; resolving this challenge for proposals like RC3 and others remains an open area of research [13, 20, 27, 29, 31].

Datacenters and Elsewhere: As we've shown via model (§3) and simulation (§5), the benefits of RC3 are strongest in networks with large RTT$\times$BW. Today's datacenter networks typically do not fit this description: with microsecond latencies, RTT$\times$BW is small and thus flows today can very quickly reach 100% link utilization. Nevertheless, given increasing bandwidth, RTT$\times$BW may not remain small forever. In simulations on a fat-tree datacenter topology with (futuristic) 100Gbps links, we observed average FCT improvements of 45% when averaged over flows, and 66% when averaged over bytes. Thus, while RC3 is not a good fit for datacenters today, it may be in the future.

Future: Outside of the datacenter, RTT$\times$BW is already large – and increasing. While increasing TCP's initial congestion window may mitigate the problem in the short term, given the inevitable expansion of available bandwidth, the problem will return again and again with any new choice of new initial congestion window. Our solution, while posing some deployment hurdles, has the advantage of being able to handle future speeds without further modifications.

7 Related Work

Router-assisted Congestion Control: Observing TCP's sometimes poor ability to ensure high link utilization, some have moved away from TCP entirely, designing protocols which use explicit signaling for bandwidth allocation. RCP [11] and XCP [22] are effective protocols in this space. Along similar lines, TCP QuickStart [15] uses an alternate slow-start behavior, which actively requests available capacity from the routers using a new IP Option during the TCP handshake. Using these explicitly supplied rates, a connection can skip slow start entirely and begin sending at its allocated rate immediately following the TCP handshake. Unlike RC3, these algorithms require new router capabilities.

Alternate TCP Designs: There are numerous TCP designs that use alternative congestion avoidance algorithms to TCP New Reno [10, 14, 19, 32, 35]. TCP CUBIC [19] and Compound TCP [32] are deployed in Linux and Windows respectively. Nevertheless, their slow-start behaviors still leave substantial wasted capacity during the first few RTTs – consequently, they could just as easily be used in RC3's primary control loop as TCP New Reno. Indeed, in our implementation we also evaluated TCP CUBIC in combination with RC3.

TCP FastStart [28] targets back-to-back connections, allowing a second connection to re-use cached Cwnd and RTT data from a prior connection. TCP Remy [34] uses machine learning to generate the congestion control algorithm to optimize a given objective function, based on prior knowledge or assumptions about the network. RC3 improves flow completion time even from cold start and without requiring any prior information about the network delay, bandwidth or other parameters.

TCP-Nice [33] and TCP-LP [23] try to utilize the excess bandwidth in the network by using more aggressive back-off algorithms for the low-priority background traffic. RC3 also makes use of the excess bandwidth, but by explicitly using priority queues, with a very different aim of reducing the flow completion time for all flows.

Use of Low Priorities: PFabric [5] is a recent proposal for datacenters that also uses many layers of priorities and ensures high utilization. However, unlike RC3, PFabric's flow scheduling algorithm is targeted exclusively at the datacenter environment, and would not work in the wide-area case.

8 Acknowledgements

We would like to thank all our colleagues in UC Berkeley, for their help and feedback - in particular Sangjin Han, Jon Kuroda, David Zats, Aurojit Panda and Gautam Kumar. We are also very thankful to our anonymous Hotnets 2013 and NSDI 2014 reviewers for their helpful comments and to our shepherd Prof. Srinivasan Seshan for his guidance in shaping the final version of the paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1106400.

References

[1] CAIDA Internet Topology Data Kit. `http://goo.gl/QAbecc`.

[2] Internet Traffic Flow Size Analysis. `http://net.doit.wisc.edu/data/flow/size/`.

[3] iPerf. `http://iperf.sourceforge.net/`.

[4] Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[5] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing Datacenter Packet Transport. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[6] M. Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 2013.

[7] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390.

[8] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. ACM SIGCOMM*, 2004.

[9] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517.

[10] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *ACM SIGCOMM Computer Communication Review*, 1994.

[11] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[12] 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 SIGCOMM Computer Communication Review*, 2010.

[13] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: the Virtue of Gentle Aggression. In *Proc. SIGCOMM*, 2013.

[14] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649.

[15] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782.

[16] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582.

[17] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883.

[18] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-Level Traffic Measurements from the Sprint IP Backbone. *IEEE Network*, 2003.

[19] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 2008.

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

[21] S. Iyer, S. Bhattacharyya, N. Taft, and C. Diot. An Approach to Alleviate Link Overload as Observed on an IP Backbone. In *Proc. IEEE INFOCOM*, 2003.

[22] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIGCOMM*, 2002.

[23] A. Kuzmanovic and E. W. Knightly. TCP-LP: Low-priority service via end-Point congestion Control. *In IEEE/ACM ToN*, 2006.

[24] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining tcp congestion control. *ACM SIGCOMM Computer Communication Review*, 1996.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018.

[26] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proc. USENIX NSDI*, 2008.

[27] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Fordy. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-compatible with TCP and TLS. In *Proc. USENIX NSDI*, 2012.

[28] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *Proc. IEEE Global Internet Conference (GLOBECOM)*, 1998.

[29] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proc. USENIX NSDI*, 2012.

[30] A. Rao, A. Legout, Y. sup Lim, D. Towlsley, C. Barakat, and W. Dabbous. Network Characteristics of Video Streaming Traffic. In *Proc. ACM CoNeXT*, 2011.

[31] C. Rotsos, H. Howard, D. Sheets, R. Mortier, A. Madhavapeddy, A. Chaudhry, and J. Crowcroft. Lost In the Edge: Finding Your Way With Signposts. In *Proc. USENIX FOCI*, 2013.

[32] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proc. IEEE INFOCOM*, 2006.

[33] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp nice: A mechanism for background transfers. In *Proc. USENIX OSDI*, 2002.

[34] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *Proc. ACM SIGCOMM*, 2013.

[35] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *INFOCOM 2004*.