Making Linux TCP Fast

Yuchung Cheng **Neal Cardwell**

Google Inc.

{ycheng,ncardwell}@google.com

Abstract

We describe several major changes that have recently been made and are still underway within the Linux TCP implementation. The Linux send engine has been refactored to decouple loss recovery from congestion control, allowing the sending rate to be independent of packet loss. RACK loss recovery reworks loss detection to rest on a more general foundation of time-based analysis, instead of counting duplicate ACKs or sequence numbers. BBR congestion control sets the sending rate based on the network's delivery rate and limits the data in-flight based on the estimated BDP, instead of reacting only to packet losses. Both algorithms are founded on a high-performance packet scheduler. The scheduler sizes the TSO bursts and releases the packets based on the rate specified by BBR congestion control, and mixes packets from different flows. These tightly-coupled but modularly-structured mechanisms together produce a low-latency and high-throughput TCP stack.

Keywords

Linux, TCP, Protocol, Networking, Performance

Introduction

The Linux TCP stack is today's most feature-rich TCP implementation. It's widely used, running on more than a billion hosts, from powerful servers to cell phones and simple embedded systems. While TCP's wire protocol has changed little over the last 35 years, the internal algorithms continue to improve in order to suit modern networks and applications. The last publication covering the overall design of Linux TCP dates from 2002 [17]. Fast-forwarding to 2016, the design has evolved significantly, both to implement new features and to consolidate existing ones for simplicity. Instead of iterating through the dozens of new TCP protocol features added to Linux over the years, we focus on the architecture and design principles of some major performance-critical pieces: loss recovery, congestion control, segmentation, and pacing. We focus on the sender-side mechanisms and show how all these pieces work together with recent new algorithms written from scratch: RACK loss detection and the BBR congestion control algorithm.

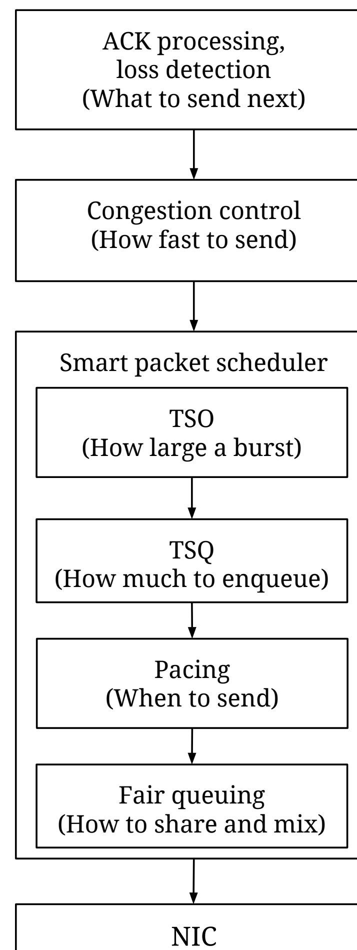

Figure 1: Linux TCP sender architecture

The Architecture of Linux TCP

Over the years, the Linux TCP implementation has been restructured to decouple functionalities, primarily to implement new features. Figure 1 shows the sender-side architecture of Linux TCP. As with most TCP implementations, with Linux most of the performance-critical decisions are made at the sender. The figure depicts the sequence of decisions that a Linux TCP sender makes as it prepares to send packets out on the wire.

The first functionality is the reliable delivery module, which decides what to send next. It processes the ACK and SACK information to track which packets were delivered. Using this information, the module also estimates if a packet was lost and should thus be retransmitted (before sending new data). It also tracks the send and delivery timing information to estimate the RTT and the delivery rate.

The second functionality is the congestion control, which decides the maximum amount of data to keep in-flight in the network (a.k.a. the congestion window), and how fast to send that data (pacing rate). This determines how quickly the sender clocks out packets when an application writes data into a socket or ACKs arrive. Linux TCP congestion control has a modular architecture, where the core decisions are made entirely by a kernel module. Linux has many different TCP congestion control algorithms, each available as a kernel module.

The third functionality is a high-performance packet scheduling system, which implements several mechanisms that are tightly coupled together. First, TSO autosizing selects the size of chunks of data that TCP hands off to lower layers to transmit in a single burst. TSO autosizing strikes a balance between desiring larger chunks for CPU efficiency and desiring smaller chunks to launch smaller bursts into the network. TCP Small Queues (TSQ) performs local flow control to reduce latency by limiting the amount of data in the qdisc. Pacing and fair queuing are both implemented in the fq/pacing qdisc; they mix TSO packets from different flows and release them according to the rate specified by their congestion control.

The following sections discuss each piece of the Linux TCP architecture in turn.

Detect losses quickly by tracking time

Over the years, Linux has implemented a rich set of loss recovery algorithms [2, 4, 1, 3, 15, 16]. Nevertheless, together they still fail to address these problems:

- Lost retransmissions due to traffic policers [9] and burst losses often cause retransmissions to be lost again.
- Tail drops at the end of application data units.
- Frequent reordering, where often the tail packet is reordered before the rest, so the reordering degree measured in terms of packet distance is a full congestion window.

Prior fast recovery algorithms counting duplicate acknowledgments [2] or sequences [15] often fail to trigger ACK-driven repairs in the above situations. The repair would run in a stop-and-wait process and eventually stall the congestion control due to the slow delivery process.

Instead of counting out-of-order packets, a better approach is to monitor the transmission time of every packet not yet delivered. The goal is to retransmit lost packets quickly: within an RTT plus a small window to accommodate reordering (e.g., a fraction of RTT). In addition, the sender may send (or retransmit) one "probe" packet after an RTT of radio silence, to solicit an ACK to trigger ACK-driven repairs. The repair only resorts to the conservative retransmission timeout (RTO) and resetting cwnd to 1 packet when every packet was dropped to the floor, which signals extreme congestion, or even link failure.

The implementation of that strategy is RACK [6] plus TLP [8]. The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. RACK uses a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences. With the Tail Loss Probe (TLP) approach, after 1.5 RTT of radio silence the sender sends a "scout" packet to solicit an ACK to generate a delivery event so RACK can continue.

Once loss recovery has determined which packets were lost, and so the sender knows what to send next, then congestion control estimates how fast those packets should be sent.

Congestion Control

Finding a good operating point

Since the 1980s, the Internet has largely used loss-based congestion control algorithms (mainly Reno [14] and its successor CUBIC [13]), which use loss as a signal to slow down. However, loss-based congestion control is ill-suited to many of today's networks. Increasingly, packet loss does not signal a good operating point for congestion control. On the last-mile links of today's Internet, loss-based congestion control causes the infamous bufferbloat problem [11], since it keeps the often bloated buffers there near full, often causing seconds of needless queuing delay. On today's high-speed long-haul links using commodity switches with shallow buffers, loss-based congestion control has abysmal throughput because it overreacts to losses caused by transient traffic bursts briefly filling the buffers even when the link may be mostly unutilized.

Since packet loss does not signal a good operating point for congestion control, ideally senders would find the best operating point themselves, largely independent of the losses they see. The best operating point for a network involves flows (a) sending at a rate matching the bandwidth available to the connection, and (b) maintaining a volume of data in flight in the network that matches the BDP (bandwidth-delay product, or bandwidth * (round-trip propagation delay)). That maximizes throughput (fills the pipes) while minimizing delay (keeping queues empty) [10].

Finding this operating point has been elusive, largely because it is impossible to measure bandwidth and round-trip propagation delay simultaneously: to measure the pipe's bandwidth, the sender must send increasingly faster until it estimates the pipe is full, which creates some amount of queue, which means the sender can no longer measure the

round-trip propagation delay.

BBR: Bottleneck Bandwidth and RTT

BBR ("Bottleneck Bandwidth and RTT") [5] is a new congestion control algorithm added for Linux TCP in Linux v4.9. It tackles the problem of finding a good operating point by sequentially probing bandwidth and RTT, and using those measurements to maintain an explicit model estimating the bandwidth and round-trip propagation delay of the pipe.

On the arrival of each ACK, BBR measures the current delivery rate over the last round trip, and feeds this through a windowed max-filter to estimate the recent bottleneck bandwidth. Conversely, it uses a windowed min-filter to estimate the recent round trip propagation delay. The max-filtered bandwidth and min-filtered RTT estimates form BBR's model of the network pipe.

Using its model, BBR sets control parameters to govern its sending behavior. The primary control is the pacing rate: BBR paces alternately slightly faster or slower than the estimated bottleneck bandwidth. The conventional congestion window (cwnd), which limits the number of packets in flight, is BBR's secondary control. BBR sets the cwnd to a small multiple of the estimated BDP, in order to allow full utilization and bandwidth probing while bounding the potential queue at the bottleneck.

When a BBR connection starts, it performs an exponential search to quickly probe the bottleneck bandwidth (doubling its sending rate each round trip, like slow start). However, instead of continuing until it loses a packet, or until delay or ACK spacing reaches some threshold (like CUBIC's Hystart [12]), it uses its model of the pipe: it estimates the pipe is full when the estimated bandwidth has stopped growing. At that point, it exits its initial rapid-growth phase and reduces its pacing rate to drain the estimated queue. Then BBR enters steady state.

In steady state, BBR first paces faster to probe for more bandwidth, and then paces slower to drain any queue that created if no more bandwidth was available, and then cruises at the estimated bandwidth to utilize the pipe without creating excess queue. Occasionally, if needed, it sends significantly slower to probe the round-trip propagation delay.

Benefits of BBR's approach

At one end of the spectrum, consider shallow-buffered networks. Since BBR uses its model of the pipe's delivery rate instead of backing off based on random incidental losses, it is resilient to the random packet loss that is common with coincident bursts in shallow buffers. For example, while CUBIC requires less than one loss per 30 million packets to fully utilize a 1Gbps link with 100ms RTT, BBR can fully utilize links at up to 15% loss rates. At typical WAN loss rates, the difference is stark: in a 30-second bulk transfer with a 10 Gbps bottleneck, 100ms RTT, and 1% packet loss rate, CUBIC gets around 3.3 Mbps, and BBR gets 9150 Mbps (more than 2700x higher).

At the other end of the spectrum, consider the bloated buffers common in today's last-mile links. Here, BBR offers low latency, because its model of bandwidth and round-trip propagation delay allow it to maintain a number of packets in flight that closely matches the BDP, fully utilizing the path with small queues and low latency. For example, consider a path with a 10 Mbps bottleneck link and 40ms RTT, with a (not uncommon) 1000-packet bottleneck buffer. BBR can fully utilize this link, but with a median RTT 25x lower than CUBIC (43 ms instead of 1.09 secs).

High-performance packet scheduler

We have presented the protocol-level algorithms designed to achieve high networking performance (high bandwidth, low queue, and fast loss recovery) from a single flow's perspective. However, these algorithms demand an efficient packet scheduler that can release the packets into the network at the time specified by the congestion control as precisely as possible. At the same time, the packets from different flows should be mixed in a way that is fair and reduces head-of-line blocking. In addition, the software solution should be CPU-efficient and maintain short local queues on the sender. We will show how Linux's networking stack meets these challenging goals with TSO autosizing, pacing, fair queuing, and TCP Small Queues (TSQ) tightly working together.

TSO autosizing

To allow low CPU utilization even at high speeds, today's Linux TCP senders typically employ segmentation offload mechanisms. With segmentation offloading, TCP deals with chunks larger than the packets on the wire, and then the NIC (with TSO) or Linux (with GSO) segments the chunks into MTU-sized packets.

In Linux, TSO autosizing selects the size of these chunks of data (represented in Linux as an skb) that TCP hands off to lower layers to transmit in a single burst. TSO autosizing has to make a trade-off here. On one hand, it desires larger chunks to increase CPU efficiency, by making fewer trips down through the entire network stack, and receiving fewer transmit completion interrupts from the NIC. On the other hand, it desires smaller chunks to launch smaller bursts into the network, leading to less data in the queues of the network, which reduces queuing latency and packet loss rates. TSO autosizing strikes a balance by sizing the bursts to 1 millisecond times the sending rate instructed by the congestion control (bounded to a max of 64KB). This allows fast flows (512 Mbps or faster) to send full 64KB bursts while slow flows send proportionally smaller bursts.

Pacing

While many of today's NICs have some mechanism to spread packets, BBR - and likely future congestion control algorithms - requires a degree of fine-grained pacing support that most NICs do not yet support. Linux's solution for pacing is the efficient fq/pacing qdisc, which uses a per-qdisc nanosecond-granularity timer to schedule packets. The pacing component takes the chunks created by TSO autosizing and spreads them out in time, placing gaps of silence in between the chunks, according to the sending rate specified on the socket by the congestion control module. Every packet's earliest release time is (almost) precisely specified by its congestion control module; that release time changes dynamically.

Fair queuing

Fair queuing, as its name implies, provides fair bandwidth sharing between flows. It does this by providing the abstraction of a separate queue for each flow, scheduling TSO chunks from each queue in a round-robin fashion that is fair on a byte-by-byte basis [7]. Fair queuing greatly improves fairness and reduces head-of-line blocking among flows. Furthermore, like pacing, fair queuing increase traffic entropy by mixing flows and thus spreading out packets from each given flow; this can lead to lower queuing delays and fewer packet drops in the middle of the network.

TCP Small Queues (TSQ)

TCP Small Queues (TSQ) performs local flow control, limiting the amount of data in the queues on the sending host. Like TSO autosizing, TSQ has its own balancing act, keeping the queues small to reduce latency and head-of-line blocking (HoLB), while keeping the queues just large enough to ensure the queues on the sending host - including both qdisc and NIC transmit queues - can feed the qdisc layer and NIC fast enough to reach full utilization.

To strike this balance and implement this flow control scheme, TSQ logic is invoked first at the beginning and second at the end of the lifetime of an skb. First, when TCP is making a decision about whether to hand off the skb to the lower layers, TSQ allows no more than 262 KB (4 full-size TSO skbs) bytes to be enqueued locally in the sending host (including the optional qdisc layer, and NIC transmit queue). Second, when the NIC finishes transmitting an skb, just before freeing the skb it makes a callback up into TCP to let it know that the packet has left the machine; at that point, the TSQ logic allows TCP to hand off another TSO skb to the lower layers. Each transmit completion can allow TCP to hand off another TSO skb, keeping the flow in balance while the sender host queues remain small and the NIC remains fully utilized.

BBR's relation to Pacing, Fair Queuing, TSQ, and TSO

By design, BBR strives to reduce the bottleneck queue in the network with its per-flow congestion control algorithm, which estimates the bandwidth available to the flow and uses pacing (rate-shaping) queues on the sending host to move the bottleneck to the sending host, where waiting packets cannot cause network queuing delays or packet loss. This means pacing is critical for BBR. As such, BBR relies on fq/pacing in Linux and would greatly benefit from future hardware support for such fine-grained packet scheduling.

Pacing is particularly crucial when an idle flow restarts. Traditional TCP congestion control algorithms needed an "ACK clock", a stream of incoming ACKs, to tell it how fast it could safely send; so when restarting from idle they would face a dilemma: either start from scratch and slow-start from a small initial cwnd, or skip that and send the entire congestion window at once, leading to line-rate bursts into the network, causing both long queues and high losses. Pacing solves that dilemma: pacing allows a congestion control algorithm (like BBR) that has a bandwidth estimate to release packets at the estimated bottleneck rate after an idle period, allowing the restarting flow to quickly fill the pipe without causing queuing delay or packet loss.

Because BBR is designed to move the bottleneck to the sending host's queues, this increases the importance of fair queuing and TCP Small Queues. These mechanisms greatly improve fairness and queueing across flows when the sending host itself is the bottleneck (e.g., a host serving hundreds of thousands of TCP flows).

Having a good estimate of the bottleneck bandwidth, unlike other congestion control algorithms, BBR can decide on the optimal TSO size required to saturate the link, at acceptable CPU usage, without causing collateral damage and congestion.

Conclusion

Over the history of the Internet, there have been numerous changes to the networking ecosystem, including applications, protocols, software defined networking, kernel bypass, and faster hardware. However, for decades, TCP has remained the main vehicle for carrying most of the bytes on the Internet. Although its wire protocol remains largely the same for legacy reasons, the internal algorithms in the Linux TCP implementation have evolved dramatically, making Linux TCP better and faster.

Within the Linux TCP implementation, we've described several large-scale changes that have recently been made, and are still underway. The RACK loss recovery algorithm is reworking loss recovery to rest on a foundation of time-based analysis, to recover losses quickly by making maximal use of all the information available to the sender, even in the face of the reordering and retransmissions common today. The Linux send engine has been refactored to decouple loss recovery from congestion control, allowing the sending rate to be independent of packet loss. This enabled BBR congestion control, which sets the sending rate based on the network's delivery rate, and limits the data in-flight based on the estimated BDP. This, in turn, moves the rate-shaping bottleneck to the queues of the sending host, which reduces queuing delays and packet loss in the network, and leverages the already-excellent packet scheduling infrastructure in the Linux networking stack: TSO autosizing, pacing, fair queuing, and TCP Small Queues.

The ideas behind the recent innovations in the Linux TCP stack are widely applicable, to most any networking transport or OS. The Linux TCP implementation will continue to evolve, and in particular there is ongoing work to deploy, test, and improve BBR; to pitch in, join the mailing list `https://groups.google.com/forum/#!forum/bbr-dev`.

Acknowledgments

We would like to thank our collaborators and colleagues in the Make TCP Fast project at Google. We'd also like to thank the members of the Linux networking community, who have together created and maintained a world-class networking code base.

Authors' Biographies

Yuchung Cheng has been a software engineer at Google since 2007. He leads the Make TCP Fast project at Google and is passionate to renovate the 35-year-old protocol. His work includes BBR congestion control, RACK recovery, Fast Open, and Tail Loss Probe. His resarch page is at `http://research.google.com/pubs/author27276.html`

Neal Cardwell has been a software engineer at Google since 2002. He worked on TCP performance in graduate school, including the TCP Vegas implementation for Linux, and is thrilled to be working on congestion control and loss recovery once again, since 2011. His work includes BBR congestion control, the packetdrill network testing tool, and involvement with RACK and Tail Loss Probe. His research page is at `http://research.google.com/pubs/NealCardwell.html`

References

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

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

[3] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton. Improving the robustness of TCP to non-congestion events., August 2006. RFC 4653.

[4] 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.

[5] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. *Queue*, 14(5), 2016.

[6] Y. Cheng and N. Cardwell. Rack: a time-based fast loss detection algorithm for tcp, 2016. IETF Draft, draft-ietf-tcpm-rack-00.

[7] J. Corbet. Tso sizing and the fq scheduler. `https://lwn.net/Articles/564978/`.

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

[9] T. Flach, P. Papageorge, A. Terzis, L. Pedrosa, Y. Cheng, T. Karim, E. Katz-Bassett, and R. Govindan. An internet-wide analysis of traffic policing. In *SIGCOMM*, pages 468–482. ACM, 2016.

[10] R. Gail and L. Kleinrock. An invariant property of computer network power. In *Conference Record, International Conference on Communications*, pages 63–1, 1981.

[11] J. Gettys and K. Nichols. Bufferbloat: dark buffers in the internet. *Communications of the ACM*, 55(1):57–65, 2012.

[12] S. Ha and I. Rhee. Taming the elephants: New tcp slow start. *Computer Networks*, 55(9):2092–2110, 2011.

[13] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[14] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

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

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

[17] P. Sarolahti and A. Kuznetsov. Congestion control in linux tcp. In *USENIX Annual Technical Conference, FREENIX Track*, pages 49–62, 2002.