TCP Pacing in the Linux Kernel

Michael Welzl
University of Oslo
Oslo, Norway
michawe@ifi.uio.no

***Abstract*—Pacing introduces short pauses in between data packets upon transmission; this is often beneficial. For example, pacing can reduce the chance of packet loss, and improve operation with very short network queues. For TCP, it has been available in the Linux kernel for many years, and it applies in the same way to all TCP congestion control variants that do not influence pacing by themselves (at the time of writing, only BBR is such an exception). Several papers offer a high level description of its operation; here, we give the first in-depth explanation of how Linux TCP pacing really works. We illustrate its behavior with a trace from a testbed experiment, and provide a companion guide that makes the relevant kernel code easier to understand.**

***Index Terms*—TCP, congestion control, pacing**

I. Introduction

The standard "Reno" [1] or, now much more common [2], "Cubic" [3] congestion control algorithm of TCP uses a credit called the "congestion window" (cwnd) to decide how many packets a sender is allowed to transmit at a given time. Allowing cwnd to grow quickly, as the "slow start" part of the congestion control algorithm does, normally causes the sender to emit a series of packets back-to-back, which increases the chance of loss when such a packet "burst" arrives at a bottleneck's queue.

One approach to address this problem is to introduce time gaps in between data packets by delaying their transmission—commonly called "pacing". Pacing was first described more than 30 years ago [4], and some evaluations studies around a decade later showed mixed results [5], [6]. Aside from this, for a long time, it was generally perceived as impossible to carry out pacing at realistic speeds in software, where the timing of TCP is normally determined. This has changed with the introduction of new innovations in OS kernels, such as high-speed timers, advances in sender-side segmentation offloading (TCP Segment Offload (TSO), the software-based Generic Segmentation Offload (GSO)) and a timing wheel [7]. TCP pacing is now popular, and some studies confirm usage "in the wild" [8], [9].

Pacing is also an integral element of the recent BBR congestion control algorithm [10]; aside from this, the pluggable congestion control frameworks of Linux and FreeBSD offer it as a general function for that applies to any of the available algorithms. In Linux, several elements of the kernel are involved in the timing of packet transmission—some of them have evolved as part of the "bufferbloat" activity which aims to avoid unnecessary buffering of packets either in routers or in the end host, since such buffers could cause unwanted delay for applications [11]. Specifically:

- *TCP Small Queues (TSQ)* limits how much data the sender enqueues for transmission, based on a feedback signal that is produced when the actual transmission happens at the Network Interface Card (NIC),
- *The queueing layer* multiplexes TCP connections and any other data coming from the transport layer, using the "FQ-CoDel" [12] queuing algorithm by default,
- *Byte Queue Limits (BQL)* limits the size of the NIC's hardware queue.

Many prior articles offer a high-level description of these elements, along with pacing itself, and their interaction [13]–[16]. However, to the best of our knowledge, none of these works offer an in-depth look at how pacing really operates, at a level that allows to calculate the actual time gaps that are inserted between packets. For this, a number of important questions are left unanswered in the literature—for example:

- Since cwnd represents a number of packets to be transmitted per round-trip time (RTT), if cwnd were constant, one could use it to uniformly disseminate the packets across the RTT. However, since cwnd itself changes during an RTT, and it may change quite significantly, e.g. in slow start, such a strategy could cause fewer packets to be emitted per RTT than the congestion control algorithm allows. How is this handled?
- Without pacing, the Initial Window (IW) can be a particularly large burst—nowadays, it is commonly 10 packets [17]. Delaying some of these packets may be especially useful [18], [19], but a pacing strategy may also suffer from the initial RTT estimate being particularly noisy, as it is normally only based on a single sample (the TCP handshake). What does Linux do?
- At high speeds, inserting a time gap between all packets becomes impossible in software. Instead, it may make sense to allow TSO to transmit a small burst of packets at once, but somehow limit the size of these bursts to just a handful of packets. How does Linux do this?

Researchers who seek to improve the currently deployed pacing approach need to understand the Linux implementation. This code deals with a complex problem, and it has undergone many changes over time; as a result, it is not easy to understand. This paper is here to help: in the next section, complementing the aforementioned prior work, we give a detailed account of the real behavior in Linux (kernel version 6.8.9). Then, in Section III, we show the resulting packet dynamics with an example run in a testbed. Finally,

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

for readers who want to "dig deeper", the appendix offers a description of the relevant parts of the Linux kernel that should make it much easier to understand the code.

II. Linux TCP pacing: what it really does

Unless the congestion control mechanism paces by itself (so far, that is only the case for BBR), pacing is disabled by default. There are two ways to enable it:

1) by configuring the `fq` queue discipline, with a command such as `sudo tc qdisc add dev 10Ge root fq maxrate 10mbit`
2) by using the `SO_MAX_PACING_RATE` socket option. This works even without changing the queue discipline. E.g., iperf can do this, when used with a client-side option such as: `--fq-rate 10m` (which means 10 Mbit/s). The rate that is given is an upper limit for the pacing rate; to just enable pacing without a limit, one can simply configure a very large number here.

Aside from explaining how choosing one of these two approaches plays out, we do not go into details on the first method. The second one, commonly called "internal" pacing, is the one that we describe here, and this is what it does:

- Independent of TCP's Initial Window (IW), which is also 10 by default, the first 10 packets are not paced. Later, 10 packets will be sent without pacing every 2^{32} packets.
- Packets are sent as bursts that are spread out via `sk_pacing_rate`. E.g., in slow start, `sk_pacing_rate` is: "200% of current rate (mss * cwnd / srtt)".
- There are various conditions that can limit the size of these bursts: not being allowed to send due to the send or congestion window; not having data from the application; even the use of TCP's "urgent" pointer. In case of a "normal", greedy sender application, and when the cwnd does not limit them, the size of these bursts is determined either by TSO or TSQ.
  - With TSO, data will be added to a socket buffer (`skb`) up to a limit of 1 ms worth of data (at `sk_pacing_rate`), but with a lower limit of 2 SMSS and an upper limit of 64 KByte). Also, more than the "1 ms worth of data" will be added for very close hosts (min. RTT of 3 ms by default, but adjustable via the `tcp_tso_rtt_log sysctl`). The role of TSQ is to ensure that `skb`'s are not actually handed over until $max(2, \texttt{sk\_pacing\_rate}/1024)$—i.e. 1 ms worth of data according to the pacing rate, but at least 2—packets have been sent out.
  - Without TSO, data are not added to an `skb` but each `skb` *is* a packet, and a loop will keep sending packets down the stack until TSQ applies its limit.
- The pacing interval between bursts is calculated as `skb->len (bytes) / sk_pacing_rate` (bytes/second). If the previous burst (`skb`) could not be transmitted when the pacing logic expects (e.g., due to a cwnd limitation), the pacing interval is halved. Without TSO, `skb->len` will be the length of a packet, but in the more common case with TSO or GSO, `skb->len` will typically reflect the amount of data that, at the current value of `sk_pacing_rate`, can be sent during 1 ms, and as a result, the pacing interval should be 1 ms in most cases. Deviations from the 1 ms limit can also happen when `skb->len` does not reflect the amount of data per millisecond at the current value of `sk_pacing_rate`:
  - because `sk_pacing_rate` is too small, and the 2*SMSS lower limit applies. Then, the time gap will be calculated as a constant (2*SMSS) / `sk_pacing_rate`, i.e. it will no longer be roughly a constant delay of 1 ms, but a gap that will become smaller as the pacing rate grows (and vice versa).
  - because `sk_pacing_rate` is too large, and the 64 KByte limit applies. Then, again, we have a constant (this time 64 KByte) / `sk_pacing_rate`, and a gap that will become smaller as the pacing rate grows (and vice versa).
  - because more data is added to the `skb` when the min. RTT < 3 ms (adjustable with the `tcp_tso_rtt_log` sysctl).

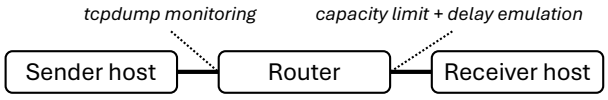

Fig. 1: The testbed used for the experiment.

III. Experiments

A test was done in the 3-node Linux testbed shown in Figure 1. The router was configured to emulate a 100 ms RTT and impose a bottleneck capacity limit of 50 Mbit/s, with a queue length of 25 packets (which should be irrelevant for this test). We configured the default IW value of 10 and selected congestion control "newreno" (which should not matter for this test as long as we do not choose BBR). We disabled delayed ACKs, kept TSO and GSO enabled, and used a packet size of 1500 bytes. In the following, as a simplification, all cwnd calculations are in packets (really, in the kernel, it is in bytes). We make the simplifying assumption: $SRTT = RTT$, where $SRTT$ is the smoothed, i.e. averaged, RTT, i.e. we assume no significant / sustained queue growth. We examine the behavior of one TCP sender with `iperf` (using the `--compatibility` flag to avoid a short initial message), and hence we do not expect the queuing system below TCP to play a role. The analysis focuses on the micro-bursts produced in the round after IW. In all later rounds, limitations from cwnd can occur due to ACKs arriving with the delay that was caused by pacing in prior rounds, making the overall behavior much harder to understand. We monitored traffic using `tcpdump` on the router's ingress link instead of the sender's outgoing interface, with TSO enabled, monitoring with `tcpdump` on the sender host does not show the actual packets that were sent out, but instead the larger segments that are handed over by TCP.

A. Expected behavior

From the description in Section II, we expect to see the following behavior in the second round (i.e., when the IW was sent and, roughly an RTT later, the first ACK arrives, causing the pacing logic to insert its first time gap):

Burst 1: cwnd was updated after the ACK arrived; `sk_pacing_rate` = 2*cwnd/RTT = 22/0.1 = 220 packets/s, i.e. 0.22 packets/ms.
Burst size: using the min. of 2 packets.
Time gap: 2/220 = 0.009 s. Because this is the first pacing interval, the time ("`credit`") since last sending would be in the order of 100 ms, much more than `len_ns`/2, and so this interval is halved, yielding 0.0045 s.

Burst 2: cwnd was updated after more ACKs arrived: ACKs arrive at a rate that reflects the bottleneck's outgoing rate of 50 Mbit/s, i.e. 4166.6 packets per second. Thus, within the time gap of 0.0045 s, approximately 18 ACKs arrive. We have only sent 10 packets—thus, all remaining ACKs arrive within this time gap, and cwnd is now 20. `sk_pacing_rate` = 2*cwnd/RTT = 40/0.1 = 400 packets/s.
Burst size: 2 packets.
Time gap: 2/400 = 0.005 s. No more halving from now on.

This should stay the same for the following bursts. In this round, there should be 10 such bursts, each clocking out 2 packets. Without further arriving ACKs, this takes: 0.0045 s + 9*0.005 s = 0.0495 s, which is a bit less than half the RTT. After 0.0495 s, we should see nothing for about half an RTT, and then the next round should begin. There, we should again have a shorter gap after the first burst that is caused by halving, but this should be significantly shorter than the second gap because now, ACKs do not all arrive between the first and the second burst.

B. Test results

Figure 2 shows that, as expected, all bursts consist of two packets, and all ACKs arrive before the second burst. The first gap, 4.65 ms, is very close to the expected 4.5 ms, and so are the following ones (expected 5 ms). Nothing more happened for approximately half an RTT (50 ms), as expected, until the next packets shown in Figure 3: we see that the first time gap (2.46 ms) is approximately half the size of the following ones, which gradually become smaller as ACKs now arrive with time gaps (as a result of the way data packets were sent in the previous round, shown in Figure 2).

We conclude this section with a diagram that illustrates the benefit of pacing: Figure 4 shows the maximum cwnd that a single connection could attain at the end of slow start with and without it. In this scenario, we configured our 3-node testbed with an RTT of 30 ms, such that the bandwidth×delay product (BDP) was 125 packets, and we varied the queue length from 1 to 125 packets. Slow start terminates when packet loss happens; as the diagram shows, this generally occurs at a lower cwnd value with unpaced TCP than with paced TCP, i.e.

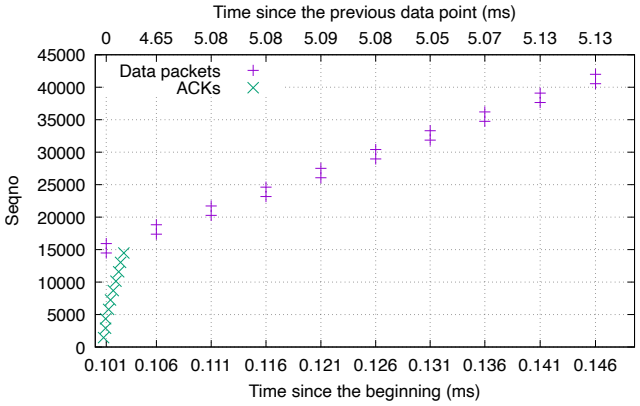

Fig. 2: Time sequence plot of all packets and ACKs in the second round. Time gaps are close to the calculated values: 4.5 ms (first) and 5 ms (all others).

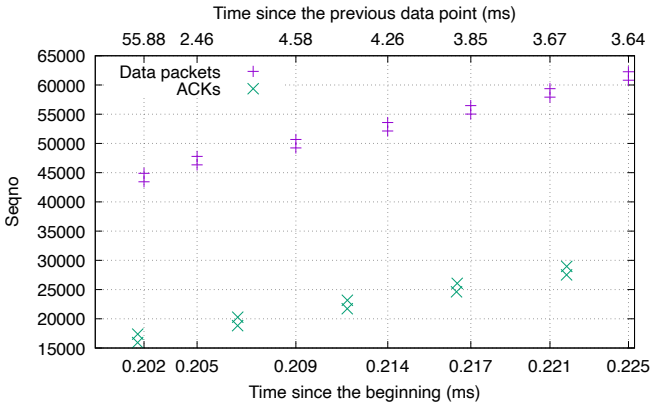

Fig. 3: The beginning of the third round. As expected, the first time gap is about half of the next, and then they shrink.

pacing successfully reduces the chance of burst loss, especially when the queue is small. The diagram also shows that TCP with pacing requires a smaller queue length to fully saturate the network with the maximum attainable cwnd, which is twice the number of packets in flight (BDP + the queue length).

IV. Conclusion

Prior literature offers a high-level overview of TCP pacing in Linux, which is certainly sufficient for many readers. Complementing these descriptions, we have provided a deeper look at how TCP pacing really operates inside the kernel. We believe that this can be of interest to a wide variety of people: a designer of a new congestion control algorithm who plans to test the algorithm in Linux; someone who wants to improve pacing itself; someone who intends to evaluate various algorithms against each other; someone who wants to understand whether to enable pacing for a specific application, and how to best tune its parameters.

We invite readers who would like to understand the kernel specifics to continue with the appendix.

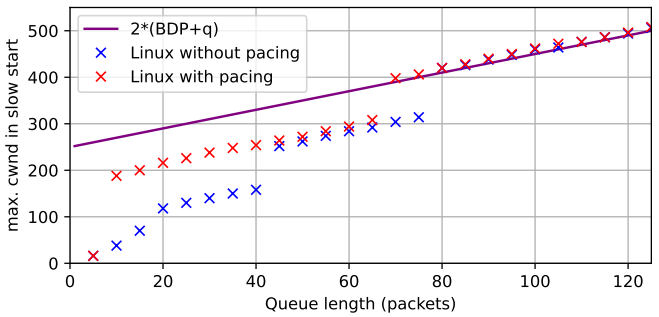

Fig. 4: Maximum cwnd in slow start.

V. Acknowledgments

The author would like to thank Kristjon Ciko, Toke Høiland-Jørgensen, and especially Neal Cardwell, who invested a large amount of time to help with this work.

References

[1] E. Blanton, D. V. Paxson, and M. Allman, "TCP Congestion Control." RFC 5681, Sept. 2009.

[2] A. Mishra, L. Rastogi, R. Joshi, and B. Leong, "Keeping an eye on congestion control in the wild with nebby," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, (New York, NY, USA), p. 136–150, Association for Computing Machinery, 2024.

[3] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert, "CUBIC for Fast and Long-Distance Networks." RFC 9438, Aug. 2023.

[4] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," in *Proc. SIGCOMM '91*, pp. 133–147, 1991.

[5] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," in *IEEE INFOCOM 2000*, vol. 3, pp. 1157–1165 vol.3, March 2000.

[6] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge, *A simulation study of paced TCP*. NASA/CR-2000-209416, 2000.

[7] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End Hosts," in *ACM SIGCOMM '17*, (New York, NY, USA), pp. 404–417, ACM, 2017.

[8] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger, "TCP revisited: A fresh look at TCP in the wild," in *Proc. 9th ACM SIGCOMM Conf. on Internet Measurement*, IMC '09, pp. 76–89, 2009.

[9] J. Rüth and O. Hohlfeld, "Demystifying TCP initial window configurations of content distribution networks," in *2018 Network Traffic Measurement and Analysis Conf. (TMA)*, June 2018.

[10] N. Cardwell, Y. Cheng, S. H. Yeganeh, I. Swett, and V. Jacobson, "BBR Congestion Control," Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-02, IETF, Mar. 2022. Work in Progress.

[11] J. Gettys, "Bufferbloat: Dark buffers in the internet," *IEEE Internet Computing*, vol. 15, no. 3, pp. 96–96, 2011.

[12] T. Høiland-Jørgensen, P. McKenney, dave.taht@gmail.com, J. Gettys, and E. Dumazet, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm." RFC 8290, Jan. 2018.

[13] Y. Cheng and N. Cardwell, "Making Linux TCP Fast," in *NetDev 1.2*, Oct. 2016.

[14] C. A. Grazia, N. Patriciello, T. Høiland-Jørgensen, M. Klapez, M. Casoni, and J. Mangues-Bafalluy, "Adapting TCP Small Queues for IEEE 802.11 Networks," in *IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2018.

[15] C. A. Grazia, M. Klapez, and M. Casoni, "The New TCP Modules on the Block: A Performance Evaluation of TCP Pacing and TCP Small Queues," *IEEE Access*, vol. 9, pp. 129329–129336, 2021.

[16] T. Høiland-Jørgensen, *Bufferbloat and Beyond: Removing Performance Barriers in Real-World Networks*. Phd thesis, Karlstad University, Faculty of Health, Science and Technology, Example City, CA, 2018. Available at https://bufferbloat-and-beyond.net.

[17] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, "Increasing TCP's Initial Window." RFC 6928, Apr. 2013.

[18] R. Sallantin, C. Baudoin, E. Chaput, F. Arnal, E. Dubois, and A.-L. Beylot, "Initial spreading: A fast Start-Up TCP mechanism," in *38th Annual IEEE Conference on Local Computer Networks*, 2013.

[19] N. Kuhn, E. Stephan, G. Fairhurst, R. Secchi, and C. Huitema, "Convergence of Congestion Control from Retained State," Internet-Draft draft-ietf-tsvwg-careful-resume-10, IETF, July 2024. Work in Progress.

Appendix A
Code

This appendix is written as a companion guide, meant to be read alongside the actual kernel code (e.g., using a cross reference such as https://elixir.bootlin.com/linux/latest/source).

In the following, we use the style `filename/function_name` to refer to a function. Unless a longer path is given, the file is in `net/ipv4/`, and if no file name is mentioned, the function is in the most recently mentioned file (i.e., that file is still the "context" of the discussion). Note also that this description is about "general" TCP; things are different in several ways for BBR, which we ignore here. We also ignore many special cases, such as retransmitting lost packets or sending control (SYN/FIN, etc.) packets. We assume that TSO and GSO are enabled, and we talk about them synonymously (when TSO is not available, GSO will always be used). The goal is to understand: in the most basic (but realistic) case, how are new packets clocked out by the pacing logic?

A. How the kernel enables pacing and calculates the rate

Whenever an ACK arrives (not counting SYN-ACK etc.), as part of ACK processing, `tcp_input.c/tcp_cong_control` is called. This function updates the congestion control state and then, for congestion control algorithms that do not have their own cong_control handler (e.g., CUBIC, Reno, and generally the majority of congestion control algorithms) calls tcp_update_pacing_rate. Simply put, for the first slow start, this does, as a comment says: `/* set sk_pacing_rate to 200 % of current rate (mss*cwnd/srtt) */`

(but limited to `sk_max_pacing_rate`). The 200 % value here comes from the `tcp_pacing_ss_ratio` sysctl.

`sk_pacing_rate` is referenced in several places. One is `net/core/sock.c/sock_init_data_uid`, where we find:

```
sk->sk_max_pacing_rate = ~0UL;
sk->sk_pacing_rate = ~UL;
WRITE_ONCE(sk->sk_pacing_shift, 10);
```

The `sk_pacing_shift` value of 10 is a factor meant for right-shifting (i.e., roughly a division by 1000), and used in some places to determine the rate per millisecond.

Then, there is also the code in the `sk_setsockopt` function under `case SO_MAX_PACING_RATE`: it changes `sk_pacing_status` from `SK_PACING_NONE` (if that was, indeed, the old status) to `SK_PACING_NEEDED`, and then sets `sk_pacing_rate` to the minimum of its current value and the supplied value. This is how using the socket option enables pacing. The definition of `sk_pacing_status` in `include/net/sock.h` has a comment saying "see enum sk_pacing", which is:

```
enum sk_pacing {
        SK_PACING_NONE = 0,
        SK_PACING_NEEDED = 1,
        SK_PACING_FQ = 2,
};
```

`sk_pacing_status` is used in various places: its value is `SK_PACING_NEEDED` (which means "internal", i.e. done by TCP "itself") for BBR, and Cubic reads it (as well as `sk_pacing_rate`) as an input for HyStart.

B. Preparing things and TSO auto-sizing

`tcp.c/tcp_init_sock` calls `tcp_timer.c/tcp_init_xmit_timers` which initializes some timers, including the one for pacing, with these lines:

```
hrtimer_init(&tcp_sk(sk)->pacing_timer,
    CLOCK_MONOTONIC, HRTIMER_MODE_ABS_PINNED_SOFT);
tcp_sk(sk)->pacing_timer.function = tcp_pace_kick;
```

So, when the pacing timer fires, `tcp_output.c/tcp_pace_kick` is called. `tcp_pace_kick` calls `tcp_tsq_handler`, which calls `tcp_tsq_write` (these multiple calls are needed to deal with the timer, lock the socket, etc). `tcp_tsq_write` calls `tcp_xmit_retransmit_queue` if retransmissions are needed, and then in any case calls `tcp_write_xmit`.

Let us now take a closer look at `tcp_write_xmit`. It begins with MTU probing, which we ignore here. With the way this function is called from `tcp_tsq_write`, the parameter `push_one` is 0, which means that the function will not only push one packet. Instead, this while loop:

```
while ((skb = tcp_send_head(sk))) {
```

will be active as long as there is something to send (i.e., data are waiting to be sent in the socket write queue)—and then, there are several function calls to check interrupting conditions in the loop, causing a `break` if one of these calls returns `true`.

Before discussing the various `break` conditions, note that this while loop sends one `skb` per iteration, where one `skb` corresponds to one segment *without* TSO. However, when TSO is active, a call to `tcp_tso_segs` (just above the while loop) figures out how many segments should really be filled into an `skb`, and stores the result in a variable called `max_segs`. We will get back to how this variable is used later. `tcp_tso_segs` first obtains a minimum number of segments, default 2 (from the `tcp_min_tso_segs` sysctl, which can be overruled by the congestion control algorithm—this is currently only done by BBR, to allow to go down to 1 depending on `sk_pacing_rate`). Then, the function calls `tcp_tso_autosize`.

The comment above `tcp_tso_autosize` explains the point of this function:

```
/* Return how many segs we'd like on a TSO packet,
 * depending on current pacing rate, and how close the peer is.
 *
 * Rationale is:
 * - For close peers, we rather send bigger packets to reduce
 *   cpu costs, because occasional losses will be repaired fast.
 * - For long distance/rtt flows, we would like to get ACK clocking
 *   with 1 ACK per ms.
 *
 * Use min_rtt to help adapt TSO burst size, with smaller min_rtt resulting
 * in bigger TSO bursts. We we cut the RTT-based allowance in half
 * for every 2^9 usec (aka 512 us) of RTT, so that the RTT-based allowance
 * is below 1500 bytes after 6 * ~500 usec = 3ms.
 */
```

This function determines the allowed number of bytes (in a variable called `bytes`) and returns `bytes/smss` (as non-float, i.e. rounded down). SMSS is the sender's maximum segment size. `bytes` is first set as `sk_pacing_rate` in milliseconds, i.e. it is the amount of data that is sent at this pacing rate in a millisecond. A variable `r` gets the value of the minimum RTT, right shifted by the `tcp_tso_rtt_log` sysctl (default 9). If `r < sizeof(unsigned int)` in bits, then bytes is increased further by the maximum available size, again right-shifted by `r`.

Let us see what this means. To simplify, we assume that `tcp_tso_rtt_log` is 10, just like `sk_pacing_shift`. Just like `bytes` is the amount of bytes per ms, `r` would now also be in ms. Commonly, `sizeof(unsigned int)` = 2^{32}, so the "if" check would require $r < 32$ ms before deciding to increase `bytes`. If `bytes` is indeed increased, we need to consider by how much. The maximum available size for TSO is defined by `sk->sk_gso_max_size`, and this value is specific to the net device; it is normally 64 Kbytes. `64 >> r` Kbytes are therefore added to bytes. So, if the minimum RTT is, e.g., 4 ms, `64 >> 4` = 4 KBytes are added.

However, the default value of `tcp_tso_rtt_log` is really 9. This means that `r` is twice the RTT in ms. Thus, twice the minimum RTT in milliseconds must be smaller than 32 ms for this condition to succeed, or, in other words, it requires min. RTT < 16 ms. For example, if the minimum RTT is 4 ms, `64 >> 8` (instead of `64 >> 4`) = 0.25 KBytes are added to `bytes`. If it is 2 ms, `64 >> 4` = 4 KByte are added to `bytes`. If it is 3 ms, `64 >> 6` = 1 KByte is added to bytes—since this is below the most common MSS, this is already a rough lower boundary above which nothing will be added (significantly lower than the 16 ms that are required for this branch of the code to even be executed).

Finally, `sk->sk_gso_max_size` (64 Kbytes) is applied as an upper limit for `bytes` in any case.

To summarize: `tcp_tso_autosize` will return the number of segments that can fit within 1 ms in most cases, lower bound by 2 (the default of the `tcp_min_tso_segs` sysctl), and upper bound by 64 KByte. Within these bounds, the number will be increased by some KBytes when the minimum RTT is very small—by default, less than 3 ms, but this value (and with it, the number of bytes to add) is controllable, via the `tcp_tso_rtt_log` sysctl.

C. Sending, with limitations

We are still in `tcp_write_xmit`. Initially, we mentioned this while loop, which transmits `skb`'s until no more are available, but with function calls where a returning `true` causes to break the loop:

```
while ((skb = tcp\_send\_head(sk))) {
```

Towards the very end of this loop, `tcp_transmit_skb` is called. This forwards the call to `__tcp_transmit_skb`, which is the function that really sends an `skb` (one or, with TSO, multiple packets). The comment above it says: *This routine actually transmits TCP packets queued in by*

tcp_do_sendmsg(). This is used by both the initial transmission and possible later retransmissions.

__tcp_transmit_skb takes care of many things, including various checks and header preparation. It calls /include/linux/skbuff.h/skb_set_delivery_time which sets skb->tstamp to the value that is handed over by __tcp_transmit_skb, which is tp->tcp_wstamp_ns, the "departure time for next sent data packet" according to the comment next to its definition in tcp.h. In the end, skb->tstamp is also used in net/ipv4/ip_output.c, e.g. to assign the right delivery time to multiple fragments when doing fragmentation.

Note: both skb->tstamp and sk_pacing_rate are also used in net/sched/sch_fq.c/fq_dequeue, where FQ pacing is implemented. This shows us that TCP calculates the current rate and writes transmission timestamps per packet, and the FQ implementation then reads these values and acts upon them. Also, sk->sk_pacing_status is checked here, and compared with the value SK_PACING_FQ.

So where does tp->tcp_wstamp_ns come from?

Whenever an skb is sent, tcp_output.c/__tcp_transmit_skb first stores the current value of tp->tcp_wstamp_ns in a variable called prior_wstamp, and then ensures that tp->tcp_wstamp_ns is at least as large as tp->tcp_clock_cache, which reflects the current time. After successful transmission of an skb, tcp_update_skb_after_send is called. There, if pacing is enabled and at least 10 (a hard coded number) packets have been sent, then tp->tcp_wstamp_ns is updated. A "minor annoyance" is mentioned in a comment: tp->data_segs_out, which is used for the check against 10, overflows after 2^{32} packets (i.e. there can be a burst of 10 packets every 2^{32} packets). Calculating the time until the next skb (remember, not always only one, but possibly multiple packets when TSO is enabled) is done by first calculating len_ns, as: skb->len (the "length of actual data") / rate (the sk_pacing_rate), which yields the time gap until the next skb.

Then, a credit is calculated as tp->tcp_wstamp_ns minus the prior_wstamp that is handed over to the function. This is the time between the expected sending time and the current time. Under "normal" pacing conditions, this credit should be minuscule. It is subtracted from len_ns—as a comment indicates, to take OS jitter into account. However, instead of simply carrying out this subtraction, the code really does: len_ns -= min(len_ns/2, credit), which means that, whenever credit is a much larger number than expected, the calculated time gap (len_ns) is halved. This happens when the last skb was not sent as calculated by pacing—e.g., because of a limitation due to cwnd or the send window. Then, len_ns is added to tp->tcp_wstamp_ns.

The above skb->len / sk_pacing_rate equation yields a dynamic gap between packet bursts that depends on the pacing rate when TSO is disabled. When TSO is enabled, this is different: remember from the description of tcp_tso_autosize that skb->len is typically autosized to be skb->len = sk_pacing_rate*1 ms. Hence, the pacing interval between skbs (bursts) should typically be the burst length / pacing_rate = skb->len (bytes) / sk_pacing_rate (bytes/sec) = sk_pacing_rate*1 ms / sk_pacing_rate = 1 ms.

Exceptions to this fixed 1 ms pacing gap calculation are:

- The minimum RTT is below 3 ms (or the tcp_tso_rtt_log sysctl has been adjusted to change this value). In this case, some extra bytes are added to the skb, such that skb->len is larger, and the gap becomes larger too.
- If sk_pacing_rate is large enough that sk_pacing_rate*1 ms hits the 64 KByte ceiling, then the skb size will be fixed at 64 KByte and the interval will shrink or expand in proportion to the pacing rate.
- If the sk_pacing_rate is small enough that sk_pacing_rate*1 ms hits the 2 SMSS floor (or the tcp_min_tso_segs sysctl was adjusted to change this value), then the skb size will be fixed at 2 SMSS and the interval will shrink or expand in proportion to the pacing rate.
- For some reason, there is not enough data available to fill the skb to really contain the equivalent of sk_pacing_rate*1 ms bytes.

Now we know that:

- a bulk of packets (one or multiple skbs—and with TSO, even one skb can represent multiple packets) is transmitted when a timer is fired,
- the time between the packets is calculated in tcp_update_skb_after_send, after every transmitted packet, and stored in tcp_wstamp_ns.

This leaves us with two questions:

1) how is the timer (re-)scheduled?
2) what limits the number of skb's that are clocked out from the tcp_write_xmit while loop?

Item 1) above is done in tcp_pacing_check, which is one of the functions called from within the tcp_write_xmit while loop. As with other such checks, if tcp_pacing_check returns "true", a "break" will terminate the loop. It is worth looking at this function closer:

- The function exits with "false" right away based if a call to tcp_needs_internal_pacing determines that the sk_pacing_status is not SK_PACING_NEEDED. This means that, without pacing, or with SK_PACING_FQ, the loop would just continue, and it would at least not be limited by tcp_pacing_check, which also would not (re-)schedule the pacing timer. **This is how "internal" TCP pacing only happens if sk_pacing_status is indeed set to SK_PACING_NEEDED.**
- The next "if" in tcp_pacing_check will also cause an exit with "false", i.e. allow the tcp_write_xmit while loop to continue transmitting, if tp->tcp_wstamp_ns

<= tp->tcp_clock_cache (which means that the current skb's tcp_wstamp_ns is not in the future).
- If none of these conditions caused tcp_pacing_check to exit, we should either wait for the pacing timer or re-schedule it. Accordingly, if no pacing timer is currently active, the pacing timer is scheduled to tp->tcp_wstamp_ns, and in any case, at this point, the function returns "true", causing a "break" in the outer loop.

Regarding item 2), other checks that will cause a "break" in the tcp_write_xmit while loop are, in order of appearance:

- Does congestion control allow to send more? This check is done via a call to tcp_cwnd_test.
- Is there space in the send window (related to a possible receiver window (rwnd) limitation)? This check is done via a call to tcp_snd_wnd_test.
- Does the Nagle algorithm prevent us from sending more? This check is done via a call to tcp_nagle_test.
- Should we collect more data for TSO? This check is done via a call to tcp_tso_should_defer. This function defers sending (by returning "true") only if all of the following conditions are true:
  - the last write (as seen in tcp_wstamp_ns, which is updated in __tcp_transmit_skb, i.e. when skb's are really sent) is at most 1 ms ago.
  - the available sending limit (min(send window, cwnd minus packets in flight)) is below max_segs. Note that we already mentioned max_segs earlier: It was determined by tcp_tso_segs, which called tcp_tso_autosize. The idea here is to immediately send when the maximum size has been reached.
  - this limit is below a certain configurable (sysctl tcp_tso_win_divisor, default 3) fraction of min(cwnd, send window) or, if tcp_tso_win_divisor is 0, it is below 3 (the explanation for this number, from a comment, is: "Receiver should ACK every other full sized frame, so if we have space for more than 3 frames then send now").
  - no more data is expected to become available (the application has stopped sending, e.g. indicated by a set FIN flag).
  - no packet is in flight (because the retransmission queue is empty).
  - judging from the time since the last packet that was sent, the next ACK is, as a comment explains: "likely to come too late (half srtt)".
- If the size limit allowed by the send window is exceeded (determined with a call to tcp_mss_split_point), can the data be correctly split (tso_fragment), i.e. can the skb be trimmed to the right length and the remaining data be put in a new packet? This requires the allocation of a new skb for whatever does not fit in the current packet.
- As the penultimate check, we have TCP Small Queues (TSQ): the function called tcp_small_queue_check.
- Finally, there is a check against a corner case with an "Argh" comment (the skb is empty—"presumably a thread is sleeping").

D. TCP Small Queues (TSQ)

TSQ limits the amount of data TCP hands over to the queuing system, based on a feedback signal that indicates when packets were sent. This is done to avoid delay from uncontrolled growth of buffers ("bufferbloat"). The core of TSQ is implemented in tcp_output.c/tcp_small_queue_check. If this function returns "true", it causes a break in the while loop in tcp_write_xmit.

About the limit, reference [15] says: "the limit is set between 2 and the configurable sysctl limit_output_bytes (128 KB by default), and dynamically chosen in this range as: sk->tcp_pacing_rate >> 10 (right-shift by 10 bits = approx. divided by 1000) which corresponds to the amount of data transmitted in 1 ms at the current value of tcp_pacing_rate."

This is correct, except that the sysctl value limit_output_bytes *only applies when pacing is disabled*. This means: with pacing enabled, the limit is max(2, tcp_pacing_rate >> 10). The limit is doubled when retransmitting.

If the optional, default off, tcp_tx_delay socket option is used, some extra bytes are added to the limit.

Then, essentially, if the limit is exceeded, the function returns "true", which will interrupt the outer while loop (except for a corner case where transmission is allowed after all). The "exceeding the limit" check is done with:

```
if (refcount\_read(&sk->sk\_wmem\_alloc) > limit)
```

which relates to this line from __tcp_transmit_skb that is executed every time an skb is sent from the loop in tcp_write_xmit:

```
refcount\_add(skb->truesize, &sk->sk\_wmem\_alloc);
```

skb->truesize is subtracted from sk->sk_wmem_alloc in tcp_wfree, which is set as the skb->destructor in __tcp_transmit_skb. This is called when the skb is freed—from net/core/skbuff.c/skb_release_head_state which is called from various places in the same file. Basically, this ensures that the TCP stack can keep track of how many bytes are outstanding for a particular flow all the way until the packet has reached "the wire" (as the skb is not freed until the TX completion interrupt comes back). When that happens (a packet is transmitted), the refcount_read call reads a different value, serving as a completion signal to TSQ that allows it to enqueue data again.