# **How does Linux pacing work?**

Michael Welzl’s effort to \*really\* get it, all based on Linux kernel version 6.7.3.

## **User-facing: how to turn on, how to configure**

Knobs:

<https://sysctl-explorer.net/net/ipv4/> is nice, but it’s not a complete list!

Specifically, we have:

* for TSQ:  
  <https://sysctl-explorer.net/net/ipv4/tcp_limit_output_bytes/>
* for pacing:  
  <https://sysctl-explorer.net/net/ipv4/tcp_pacing_ss_ratio/> (default 200)  
  and  
  <https://sysctl-explorer.net/net/ipv4/tcp_pacing_ca_ratio/> (default 120)

When using a congestion control mechanism that does not by itself pace (i.e. require SK\_PACING\_NEEDED, see below), pacing is disabled by default. There are two ways to enable it:

1. by configuring the fq queue discipline, with a command like:  
   sudo tc qdisc add dev 10Ge root fq maxrate 10mbit  
   (undo: just delete, e.g. sudo tc qdisc del dev 10Ge root )
2. by using the SO\_MAX\_PACING\_RATE socket option. This works even without changing the qdisc (I see the default, which is pfifo\_fast). E.g., iperf can do this, when giving the client an option such as:  
   --fq-rate 10m

(which means 10Mbit/s)

The 10Mbit/s choice above is just an example - this number would serve as an upper limit for the pacing rate.

Aside from explaining how choosing one of these two approaches plays out (how one or the other is enabled), this document does not go into details on 1), which may not be 100% similar to 2). Variant 2), which is commonly called “internal” pacing, is the one that we describe in further detail here.

## 

## **Internals**

Based on: <https://elixir.bootlin.com/linux/latest/source>

An article that helped me better understand:

[https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/](https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/#:~:text=Packets%20added%20to%20the%20driver,the%20NIC%20for%20immediate%20transmission.)

**Some notes:** the text below uses 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” we are in). Also, this is about “general” TCP; things are different in several ways for BBR, which we ignore here. We also ignore special cases: retransmitting lost packets, sending SYN/FIN packets, sending a probe timeout… and also everything related to TSO etc. We want to know: in the most basic case, how are new packets clocked out by the pacing logic.

### **How and where is pacing enabled & sk\_pacing\_rate calculated**

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 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)

sk\_pacing\_rate is referenced in several places. One is: net/core/sock.c in the sock\_init\_data\_uid function, where we find:

sk->sk\_max\_pacing\_rate = ~0UL;

sk->sk\_pacing\_rate = ~0UL;

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.

and then, there’s the code in the sk\_setsockopt function under the case:

case SO\_MAX\_PACING\_RATE:

The code there 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. 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: obviously, it’s 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.

### **How and where is pacing applied**

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.

We don’t care about retransmissions, so let’s look at tcp\_write\_xmit.

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))) {

which will be active as long as there is something to send (i.e., SKBs are waiting to be sent in the socket writing queue) - and then, there are several function calls to check interrupting conditions in the loop, yielding “break” if one of these functions returns true.

Then, in this loop, tcp\_transmit\_skb is called, which just forwards the call to \_\_tcp\_transmit\_skb, which is the function that really sends a packet. 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 – various checks, header preparation, etc etc. It calls /include/linux/skbuff.h/skb\_set\_delivery\_time which sets skb->tstamp to the value that’s 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! So, TCP calculates a 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 a packet is sent, \_\_tcp\_transmit\_skb 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 a packet, in \_\_tcp\_transmit\_skb, the function tcp\_update\_skb\_after\_send is called. There, if pacing is done and more than 10 packets have been sent (hard coded number 10) then tp->tcp\_wstamp\_ns is updated. There’s a “minor annoyance” mentioned in a comment: tp->data\_segs\_out, which is used for the check against 10, overflows after 2^32 packets (i.e. we will get a burst of 10 packets at every 2^32 packets :-) ).

Calculating the time until the next packet is done by first calculating “len\_ns”, as:  
skb->len (the packet length) / rate (the sk\_pacing\_rate)

… which yields the time gap until the next packet.

Then, a minimum of len\_ns and a “credit” is taken, where the credit is tp->tcp\_wstamp\_ns minus a prior\_wstamp which is handed over to the function, and len\_ns is reduced by min(len\_ns/2, credit) to take OS jitter into account. This really just reduces the timer by the time to send half a packet, lower bound with the time gap since the previous packet. Then, len\_ns is added to tp->tcp\_wstamp\_ns.

Now we know that:

* a bulk of 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 related to pacing:

1. how is the timer (re-)scheduled?
2. how is the number of packets clocked out from the tcp\_write\_xmit while loop limited?

1) 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’s 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 wouldn’t (re-)schedule the pacing timer. **This is how “internal” 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 just go on with the transmission, if tp->tcp\_wstamp\_ns <= tp->tcp\_clock\_cache (which means that the current packet’s tcp\_wstamp\_ns is not in the future). When tcp\_pacing\_check is called for the very first time, tp->tcp\_wstamp\_ns has not yet been initialized (remember, it happens in tcp\_update\_skb\_after\_send), and this cause a return with false, allowing the next packet to be transmitted right away. So, there will always be a “burst” of 2 packets at the very beginning – but this makes no difference: as we will see, TSQ uses a minimum of 2 anyway.
* If none of these conditions caused tcp\_pacing\_check to exit, it means that 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.

2) other checks that will cause a “break” in the tcp\_write\_xmit while loop are: does congestion control allow to send more; is there space in the send window (related to a possible rwnd limitation); does the Nagle algorithm prevent us from sending more; some things related to TSO; a corner case with an “Argh” comment (“presumably a thread is sleeping..”), and, last but not least, TSQ: the function called tcp\_small\_queue\_check.

### **TSQ**

The core of TSQ is implemented in a function in tcp\_output.c called tcp\_small\_queue\_check. If this function returns true, it causes a break in a loop in tcp\_write\_xmit, and also interrupts something in tcp\_xmit\_retransmit\_queue (I’m not going into details because retransmissions are out of scope of this document).

This paper:

<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9541151>

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.

The limit is doubled when retransmitting.

If (and only 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 (minus one corner case where transmission is allowed after all), the function returns true, which will interrupt the outer while loop. 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 a packet 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 will be called when the skb is freed – from skb\_release\_head\_state() which is called from kfree\_skb() among other places. 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 hit the wire (as the skb is not freed until the TX completion interrupt comes back).

Here’s a quote from this paper: <https://ieeexplore.ieee.org/document/8581048>

“When the limit is reached, new data of a flow can be enqueued only when previously enqueued packets from the same flow have been dequeued by the NIC, and the packet memory freed. This works by triggering a completion signal that notifies the TSQ logic when the driver frees the memory allocated for a data packet.”

Finally, as this and other papers show, Byte Queue Limits (BQL) is just ensuring a limit on how much data is handed to the NIC, which means that it allows the queuing layer above to exercise better control. This is all “below” the TCP logic, and thus irrelevant for understanding pacing, except that it possibly influences the (destructor) “signal” to TSQ.

### **Summary**

When pacing is enabled via the socket option:

* Independent of the IW value, the first 10 (hardcoded) packets are not paced. Later, 10 packets will generally be sent without pacing every 2^32 packets.
* Packets are sent as bursts that are spread out via sk\_pacing\_rate. In slow start, for example, sk\_pacing\_rate is: “200 % of current rate (mss \* cwnd / srtt)”;
* The size of these bursts is determined by TSQ, which will set them to max(2, sk\_pacing\_rate/1024) ).

## 

## **What tests show**

I tested with Debian bookworm with an upgraded kernel, from “pc01” via “pc02” to “router” in our “old testbed”. The installed kernel for tests: uname -r shows:

6.7.1-zabbly+ (installed as described at [here](https://ubuntuhandbook.org/index.php/2023/08/install-latest-kernel-new-repository/))

sudo tc qdisc show:

qdisc noqueue 0: dev lo root refcnt 2

qdisc mq 0: dev enp36s0 root

qdisc pfifo\_fast 0: dev enp36s0 parent :4 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

qdisc pfifo\_fast 0: dev enp36s0 parent :3 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

qdisc pfifo\_fast 0: dev enp36s0 parent :2 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

qdisc pfifo\_fast 0: dev enp36s0 parent :1 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

qdisc pfifo\_fast 0: dev enp0s25 root refcnt 2 bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1

With newreno and cubic: default: no pacing.

The outcome of the two ways of enabling pacing, with the parameters mentioned above, is a little different! Generally, the size of bursts seems smaller when using socket-level pacing, but it’s hard to tell.

When TSO etc. are enabled, the pcap file taken at the sender shows perfect pacing, with a gap between every packet! … but this looks different at the router’s (“pc02”) ingress interface. It is best to disable TSO etc. to get a result that is “true”.

For all tests below: no delayed ACKs, no TSO etc., packet size 1500 bytes, all cwnd calculations in packets. Simplifying assumption: SRTT = RTT, i.e. no significant / sustained queue growth.

**Test 1:**

RTT = 500ms, IW 10:

Expectation:

The first round after IW:

sk\_pacing\_rate = 2\*cwnd/RTT = 40

Burst 1: 40 packets per second = 40/1024 packets per ms = 0.04 but at least 2, so 2 (and not going to get larger for a long time).

Time gap: 1/40 = 0.025 seconds.

Confirmed with wireshark.

**Test 2:**RTT = 300ms, IW 10:

Expectation:

The first round after IW:

sk\_pacing\_rate = 2\*cwnd/RTT = 66

Burst 1: 66 packets per second = 2000/1024 packets per ms = 0.065 but at least 2, so 2 (and not going to get larger for a long time).

Time gap: 1/66 = 0.01515 seconds.

Confirmed with wireshark.

**Test 3:**RTT = 6ms, IW 10:

Expectation:

The first round after IW:

sk\_pacing\_rate = 2\*cwnd/RTT = 3333.33

Burst 1: 3333 packets per second = 3333/1024 packets per ms = 3.25 but at least 2, so 3.

Time gap: 1/3333 = 0.0003 seconds.

Checked with wireshark - burst size confirmed, but some gaps are quite a bit larger now; this is probably due to our “old” testbed becoming imprecise at such timescales.

**Test 4:**

RTT = 10ms, IW 30:

Expectation:

10 packets should be sent without pacing, and then the remaining 20 packets in the IW will be paced.

The first round after IW:

sk\_pacing\_rate = 2\*cwnd/RTT = 6000

Burst 1: 6000 packets per second = 5.86 packets per ms = 5 but at least 2, so 5.

Time gap: 1/6000 = 0.00017 seconds.

Checked with wireshark – the first burst is 3 packets, the second 6, then 6, 4, 3, 5… this gets harder to understand. Also, the time gaps are significantly larger.

**Test 5:**

RTT = 500ms, IW 30:

Expectation:

30 packets should be sent without pacing (remember, the 10 packet rule only matters when ACKs already arrive while sending IW, which is clearly not the case here).

The first round after IW:

sk\_pacing\_rate = 2\*cwnd/RTT = 120

Burst 1: 120 packets per second = 0.1 packets per ms = 0 but at least 2, so 2.

Time gap: 1/120 = 0.00833 seconds.

Checked with wireshark - the first burst is 7 packets, then staying at 4, and the time gap is much smaller, in the order of 0.0025 seconds or so.

*Note: in the calculations above, cwnd should actually be larger by 1 because the cong. update happens before the sk\_pacing\_rate update, and this all begins when the first ACK arrives. We ignore this, it’s a minor detail.*

# **Not directly about Linux pacing… but relevant**

## **How to disable delayed ACKs?**

It doesn’t work with the ip route quickack = 1 thing anymore.

However: it’s a socket option, and iperf allows to use it. In the iperf manpage, it’s listed as a client-specific option, but it’s also possible to supply it on the server… why not just use it on both sides. The option is: “----tcp-quickack”

In teacup (the scripting environment that we use), it needs to be supplied as an “extra parameter”. Delayed ACKs get enabled when specifying iperf with this line, for example:  
traffic\_iperf = [ ('0.0', '3', " start\_iperf, client='pc01', server='router', port=5003,"

" duration=V\_duration, extra\_params\_server='--tcp-quickack', extra\_params\_client='--tcp-quickack' "),

]

ncat does not support this option, but it could be added to the source code:

<https://github.com/nmap/nmap/tree/master/ncat>

## **How to disable TSO etc.?**

sudo ethtool -K interfacename tso off;

sudo ethtool -K interfacename gso off;

sudo ethtool -K interfacename lro off;

sudo ethtool -K interfacename gro off;

sudo ethtool -K interfacename ufo off;

… ignore warnings, some may be old / changed. Note: teacup disables all of these things anyway.

## **How to change IW?**

Confirmed, this still works as described here:   
<https://www.cdnplanet.com/blog/tune-tcp-initcwnd-for-optimum-performance/#change-initcwnd>

Just make sure to change the correct entry, and note: teacup changes this!

## **How to monitor?**

When considering that level of detail, it may be worth taking a pcap from the sender-facing interface of the router instead of taking it directly at the sender. E.g., at the sender, the effect of TSO on vs. TSO off looked different.