Internet Congestion Control
Internet-Draft
Intended status: Informational
Expires: 23 April 2026

M. Welzl
University of Oslo
W. Eddy
Aalyria Technologies
V. Goel
Apple Inc.
M. T端xen
M端nster University of Applied Sciences

20 October 2025

Pacing in Transport Protocols
draft-irtf-iccrg-pacing-00

Abstract

Applications or congestion control mechanisms can produce bursty traffic which can cause unnecessary queuing and packet loss. To reduce the burstiness of traffic, the concept of evenly spacing out the traffic from a data sender over a round-trip time known as "pacing" has been used in many transport protocol implementations. This document gives an overview of pacing and how some known pacing implementations work.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://mwelzl.github.io/draft-iccrg-pacing/draft-irtf-iccrg-pacing.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-irtf-iccrg-pacing/.

Discussion of this document takes place on the Internet Congestion Control Research Group mailing list (mailto:iccrg@irtf.org), which is archived at https://mailarchive.ietf.org/arch/browse/iccrg. Subscribe at https://www.ietf.org/mailman/listinfo/iccrg/.

Source for this draft and an issue tracker can be found at https://github.com/mwelzl/draft-iccrg-pacing.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Copyright Notice

Table of Contents

1. Introduction 3
2. Conventions and Definitions 4
3. Motivations for Pacing 4
  3.1. Network Benefits 4
  3.2. End Host Benefits 5
  3.3. Other Motivations 5
4. Pacing: general considerations and consequences 5
  4.1. More likely to saturate a bottleneck 6
    4.1.1. Backing off after the increase 7
    4.1.2. Able to work with smaller queues 8
  4.2. Queue dynamics 8
  4.3. Getting good RTT estimates 9
  4.4. Mini-bursts and their trade-offs 9
  4.5. Application control 10
5. Implementation examples 10
  5.1. Linux TCP 10
  5.2. Apple OSes 11
  5.3. FreeBSD 12
  5.4. QUIC BBR implementations 15
6. Security Considerations 16

7. IANA Considerations 17
8. References 17
8.1. Normative References 17
8.2. Informative References 17
Acknowledgments 19
Change Log 19
Authors' Addresses 19

1. Introduction

Applications commonly generate either bulk data (e.g. files) or bursts of data (e.g. segments of media) that transport protocols deliver into the network based on congestion control algorithms.

RFCs describing congestion control generally refer to a congestion window (cwnd) state variable as an upper limit for either the number of unacknowledged packets or bytes that a sender is allowed to emit. This limits the sender's transmission rate at the granularity of a round-trip time (RTT). If the sender transmits the entire cwnd sized data in an instant, this can result in unnecessarily high queuing and eventually packet losses at the bottleneck. Such consequences are detrimental to users' applications in terms of both responsiveness and goodput. To solve this problem, the concept of pacing was introduced. Pacing allows to send the same cwnd sized data but spread it across a round-trip time more evenly.

Congestion control specifications always allow to send less than the cwnd, or temporarily emit packets at a lower rate. Accordingly, it is in line with these specifications to pace packets. Pacing is known to have advantages -- if some packets arrive at a bottleneck as a burst (all packets being back-to-back), loss can be more likely to happen than in a case where there are time gaps between packets (e.g., when they are spread out over the RTT). It also means that pacing is less likely to cause any sudden, ephemeral increases in queuing delay. Since keeping the queues short reduces packet losses, pacing can also yield higher goodput by reducing the time lost in loss recovery.

Because of its known advantages, pacing has become common in implementations of congestion controlled transports. It is also an integral element of the "BBR" congestion control mechanism [I-D.cardwell-iccrg-bbr-congestion-control].

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Motivations for Pacing

Pacing is an old idea which did not see much deployment for decades. This may be due to the need for efficient fine-grain timers, which were not previously available in software. Also, at least one early analysis has documented disadvantages of pacing, primarily in terms of throughput [UnderstandingPacing]. At the time of writing, this article has become 25 years old; it is limited to Reno congestion control, and defines a pacing method that is not in line with any of the implementations that we document in Section 5. A part of the critical analysis in the article includes an example of a double back-off after slow start (fig. 8); the authors attribute the performance drop to synchronization, but it may instead be caused by the back-off factor beta being too large, as described in Section 4.1.1. In the latter case, it is not a problem with pacing per se.

There are several potential benefits to pacing, both for the end-host protocol stacks and within the network. This section provides a short summary of the motivations for performing pacing, with specific examples worked through in the following (Section 4).

3.1. Network Benefits

Senders generating large bursts create challenges for network queue management to maintain low latencies, low loss rates, and low correlated bursts of loss. This is described in more detail in Section 4.1, with examples.

A number of causes within the network may lead to "ACK compression", where the spacing of incoming packets (with new ACKs) becomes bunched. This could happen due to many factors, such as congestion at a bottleneck, packet send aggregation in the MAC layer or device drivers, etc. ACKs can also wind up being aggregated beyond the normal delayed ACK recommendation, such that instead of acknowledging one or two packets of data, a received ACK may cover many packets, and cause a large change in the congestion window, allowing many packets to be released in a burst (if pacing is not used). This can happen due to coalescing of ACKs or ACK "thinning" within the network, or as a means to deal with highly asymmetric connectivity.

In any case, a sender that performs pacing is not susceptible to ACK compression, aggregation, or thinning causing its own sending patterns to become bursty in turn, which allows the network more latitude in how ACKs are handled.

3.2. End Host Benefits

Pacing enables good operation with shorter queues, and can create an incentive to reduce the size of queues being configured within the network, leading to lower maximum latency for end host applications. When applications use pacing to limit the rate, this can also reduce latency irrespective of the size of the queues available in the network (see Section 4.5).

Improved RTT measurements can result from pacing, since samples are spread out further across time rather than clumped in time, as explained in Section 4.3.

At a receiver, processing of the received packets is impacted by pacing, since it will result in a more steady workload, rather than large incoming bursts of data. For some applications, pacing can therefore be beneficial in avoiding long periods where the system is busy devoted to processing incoming data, and unable to perform other work. However, some systems are designed such that incoming bursts are more efficient to process than a steadily paced stream, so benefits may differ.

3.3. Other Motivations

In some special situations, outside general Internet usage, the path properties may be well-known in advance (e.g. due to scheduling of capacity, etc.). In this case, senders should pace packets at the scheduled rates in order to efficiently utilize that capacity. In some of these cases, the rates may be very high, and any sender burstiness might require large expensive buffers within the network in order to accommodate bursts without losses. Situations where this applies may include supercomputing grids, private datacenter interconnection, and space mission communications [I-D.draft-many-tiptop-usecase].

4. Pacing: general considerations and consequences

This section explores pacing scenarios in more detail, explains considerations important for using and tuning pacing, and the resulting consequences.

4.1. More likely to saturate a bottleneck

We can distinguish between two reasons for packet losses that are due to congestion at a bottleneck with a DropTail (FIFO) queue:

1. A flight of N packets arrives. The amount of data in this flight exceeds the amount of data that can be transmitted by the bottleneck during the flight's arrival plus the queue length, i.e. some data do not fit into the queue.

2. The bottleneck is fully saturated. The queue is full, and packets drain from it more slowly than new packets arrive.

The second type of loss matches the typical expectation of a congestion control algorithm: the cwnd value when loss happens is indicative of the bottleneck being fully saturated. When the first type of loss happens, however, a sender's cwnd can be much smaller than the Bandwidth-Delay Product (BDP) of the path (the amount of data that can be in flight, ignoring the queue). In the absence of other traffic, the probability for the first type of loss to happen depends on the queue length and the ratio between the departure and the arrival rate during the flight's arrival. By introducing time gaps between the packets of a burst, this ratio is increased, i.e. the difference between the departure and the arrival rate becomes smaller, and the second type of loss is more likely.

For example, consider a network path with a bottleneck capacity of 50 Mbit/s, a queue length of 15000 bytes (or 10 packets of size 1500 bytes) and an RTT of 30 ms. Assume that all packets emitted by the sender have a size of 1500 bytes. Then, the BDP equals 125 packets. The bottleneck of this network path is fully saturated when a (BDP + queue length) amount of bytes are in flight: 135 packets.

In this network, the first type of loss can happen as follows: say, N=40 packets arrive at this bottleneck at a rate of 100 Mbit/s. In an otherwise empty network and assuming an initial window of 10 packets and no delayed ACKs, this occurs in the third round of slow start without pacing, provided that the capacities of all links before the bottleneck are at least 100 Mbit/s. In this case, an overshoot will occur: packets are forwarded with half their arrival rate, i.e. less than 20 packets can be forwarded during the burst's arrival. The remaining 20 (or more) packets cannot fit into the 10-packet queue. A cwnd of 40 packets is much smaller than the (BDP + queue) limit of 135 packets, and the bottleneck is not fully saturated.

Let us now assume that the flight of 40 packets is instead paced, such that the arrival rate only mildly exceeds the departure rate -- e.g., they arrive at a rate of 60 Mbit/s. When the last packet of this flight arrives at the bottleneck, the bottleneck should already have forwarded 5/6 * 39 = 32.5 packets. Since only complete packets can be sent, the bottleneck has really forwarded 32 packets, and the remaining 40-32 = 8 packets fit in the queue. No loss occurs.

This example explains how pacing can enable a rate increase to last longer than without pacing. This makes it more likely that a bottleneck is saturated, such that cwnd reflects the BDP plus the queue length (loss type 2).

4.1.1. Backing off after the increase

The two loss types explained in Section 4.1 require a different back-off factor to allow the queue to drain and congestion to dissipate. Specifically, in the single-sender single-bottleneck example above, when a slow start overshoot occurs as loss type 2, a back-off function such as: ssthresh = cwnd * beta with beta >= 0.5 is guaranteed to cause a second loss after the end of loss recovery. This is because, when cwnd exceeds a fully saturated bottleneck (i.e., cwnd > BDP + queue length), cwnd will have grown further by another (BDP + queue length) by the time the sender learns about the loss. In this case, beta = 0.5 will cause ssthresh to exceed (BDP + queue length) again.

Since pacing makes loss type 2 more likely, beta < 0.5 may be a better choice after slow start overshoot when pacing is used.

The following example illustrates this: consider a TCP sender that transmits data across a single bottleneck router that uses a FIFO queue towards a TCP receiver; assume that the sender is paced well, and only loss type 2 happens. For simplicity, we consider the congestion window in units of segments rather than bytes. The initial window (IW) is 10 segments, and the path's capacity limit (the BDP plus the bottleneck queue length) equals 30 segments. In slow start, after receiving ACKs for the first 10 segments, the sender will have transmitted 20 more segments and the value of cwnd will be 20. After receiving ACKs for these 20 segments, the sender will have transmitted 40 more segments and the value of cwnd will be 40. The first 30 of the 40 newly transmitted segments will pass through the bottleneck, but the remaining 10 segments will be dropped. The ACKs that are caused by the first 30 segments then cause the sender to transmit another 60 segments, and cwnd will be increased to 70. When the first of these 60 segments arrive at the receiver, they cause DupACKs; when these DupACKs arrive the sender, backing off with beta=0.5 yields ssthresh = 35, which is more than the path's capacity limit, and another loss will occur.

4.1.2. Able to work with smaller queues

The probability of loss type 1 in Section 4.1 is indirectly proportional to the queue length. Pacing therefore enables a rate increase to continue with a smaller queue at the bottleneck than in the case without pacing.

4.2. Queue dynamics

When it enters the queue at a network bottleneck, unpaced traffic causes more sudden, drastic delay growth than paced traffic, and has a higher risk of packet loss, as discussed in Section 4.1. Paced traffic, on the other hand, can cause a bottleneck queue to grow more slowly and steadily, incurring delay growth over a longer time interval. Aside from the direct problems that delay can cause, such sustained queue and delay growth is also more likely to provoke an Active Queue Management (AQM) algorithm to drop packets or mark them using Explicit Congestion Notification (ECN). This is because AQM algorithms are commonly designed to allow short, transient traffic bursts to pass unharmed, but react upon longer-term average queue growth.

4.3. Getting good RTT estimates

Since pacing algorithms generally attempt to spread out packets evenly across an RTT, it is important to have a good RTT estimate. Especially in the beginning of a transfer, when sending the initial window, the only RTT estimate available may be from the connection establishment handshake. Being based on only one sample, this is a very unreliable estimate. Moreover, a new transport connection may be preceded by a longer period of quiescence on the path between two endpoints than there might normally occur when a connection is active. Such a silence period can provoke behavior of lower layers that increases the RTT. For example, idle periods commonly cause a handshake procedure on 5G links before communication can continue, inflating the RTT.

Thus, using this sample to pace the initial window can cause the pacing rate to become unnecessarily low. This may be the reason why the Linux TCP implementation does not pace the first 10 packets (see Section 5.1). As a possible improvement, the initial RTT estimate could also be based on a previous connection (temporal sharing) or on another ongoing connection (ensemble sharing) [RFC9040].

4.4. Mini-bursts and their trade-offs

Generally, hardware can perform better on large blocks of data than on multiple small data blocks (fewer copy operations). Hardware offload capabilities such as TCP Segment Offload (TSO) and Generic Segmentation Offload (GSO) are popularly used in cases with high data rates or volumes (e.g. datacenters, hyperscaler servers, etc.) and important to efficiency in compute and power budgets. When using TSO and GSO efficiently, there will be large writes between software and hardware. Since the hardware itself does not typically perform pacing, this results in burstiness, or if the sending software is trying to perform pacing, it could defeat the goal of efficiently using the offload hardware. A strategy to work with this is to avoid pacing every single packet, but instead pace such that a pause is introduced between batches of some packets that are sent as a mini-burst. Such a strategy is implemented in Linux, for example.

At the receiving side, offload techniques like Large Receive Offload (LRO), Generic Receive Offload (GRO), interrupt coalescing, and other features may also be impacted by pacing. Paced packets reduce the ability to group together incoming hardware frames and packets for upper layer processing, but end systems may be tuned to handle incoming mini-bursts and maintain some efficiency.

Clearly, the size of mini-bursts embeds some trade-offs. Even mini-bursts that are very short in terms of time when they leave the sender may cause significant delay further away on an Internet path, where the link capacity is smaller. For example, consider a server that is connected to a 100 Gbps link, serving a client that is behind a 15 Mbps bottleneck link. If that server emits bursts that are 50 kbyte long, the duration of these bursts at the server-side link is negligible (4.1 microseconds). When they reach the bottleneck, however, their duration becomes as large as 27.3 milliseconds. This is an argument for minimizing the size of mini-bursts. On the other hand, wireless link layers such as WiFi can benefit from having more than one packet available at the local send buffer, to make use of frame aggregation methods. This can significantly reduce overhead, and allow a wireless sender to make better use of its transmission opportunity; eliminating these benefits with pacing may in some cases be counter-productive. This is an argument for making the size of mini-bursts larger.

4.5. Application control

When an application produces data at a certain (known) bitrate, it can be beneficial to make use of pacing to limit the transport bitrate on this basis such that it is not exceedingly large. The Linux and FreeBSD applications allow the application to set an upper limit.

For example, frame based video transmission typically generates data chunks at a varying size at regular intervals. Such an application could request a data chunk to be spread over the interval. This would allow a more sustained data transmission at a lower rate than a transport protocol’s congestion control might choose, rather than using a shorter time period within the interval with a high rate. This has the benefit that queue growth is less likely, i.e. this form of pacing can reduce latency.

Spreading over the interval needs to be done with some caution; "ideally" spreading data across the entire interval risks that some of data will not arrive in time, e.g. when delays are introduced by other traffic. SCReAM and SAMMY pace packets at a somewhat higher rate (50% in case of SCReAM) to reduce this risk [I-D.draft-johansson-ccwg-rfc8298bis-screamv2-03], [Sammy].

5. Implementation examples

5.1. Linux TCP

Pacing was first implemented in Linux kernel version 3.12 in 2013. The following description is based on Linux kernel version 6.12.

There are two ways to enable pacing in Linux: 1) via a socket option, 2) by configuring the FQ queue discipline. We describe case 1.

Independent of the value of the Initial Window (IW), the first 10 (hardcoded) packets are not paced. Later, 10 packets will generally be sent without pacing every 2^32 packets.

Every time an ACK arrives, a pacing rate is calculated, as: factor * MSS * cwnd / SRTT, where "factor" is a configurable value that, by default, is 2 in slow start and 1.2 in congestion avoidance. MSS is the sender maximum segment size [RFC5681], and SRTT is the smoothed round-trip time [RFC6298]. The sender transmits data in line with the calculated pacing rate; this is approximated by calculating the rate per millisecond, and generally sending the resulting amount of data per millisecond as a small burst, every millisecond. As an exception, the per-millisecond amount of data can be a little larger when the peer is very close, depending on a configurable value (per default, when the minimum RTT is less than 3 milliseconds).

If the pacing rate is smaller than 2 packets per millisecond, these bursts will become 2 packets in size, and they will not be sent every millisecond but with a varying time delay (depending on the pacing rate). If the pacing rate is larger than 64 Kbyte per millisecond, these bursts will be 64 Kbyte in size, and they will not be sent every millisecond but with a varying time delay (depending on the pacing rate). Bursts can always be smaller than described above, or be "nothing", if a limiting factor such as the receiver window (rwnd) [RFC5681] or the current cwnd disallows transmission. If the previous packet was not sent when expected by the pacing logic, but more than half of a pacing gap ago (e.g., due to a cwnd limitation), the pacing gap is halved.

This description is based on the longer Linux pacing analysis text in [LinuxPacing].

5.2. Apple OSes

Pacing was added to Apple OS as a private API in iOS 17 and macOS 14. In its current form, an application or transport protocol computes and sets the desired transmit timestamp on a per packet basis and sends it to the pacing module in AQM. The packets are queued in the AQM until the current time becomes greater than or equal to corresponding packet’s transmit timestamp. There is an upper limit of 3 seconds for how long the AQM will hold a queued packet before sending it.

The above simplicity in the kernel allows upper layer protocols or applications to set a transmit timestamp in a manner that is suitable for them. For example, a stream based protocol like TCP might pace packets differently than a video conferencing app.

5.3. FreeBSD

FreeBSD has the infrastructure to support multiple TCP stacks. Each TCP stack has a tcp_output() function, which handles most of the sending of TCP segments. The default stack does not support pacing and its tcp_output() may be called whenever

1. a TCP segment is received or
2. a TCP timer (like the retransmission or delayed ACK timer) runs off or
3. the application provides new data to be sent

and sends as many TCP segments as is allowed by the congestion and flow control resulting in burst of TCP segments. However, this also allows to make use of TCP Segment Offload (TSO), which reduces the CPU load.

The RACK [RACK] and BBR stacks both support pacing by leveraging the TCP High Precision Timer System (HPTS) [HPTS], which is a kernel loadable module available in FreeBSD 14 and higher. The tcp_output() function of a TCP stack which supports pacing will not send as much as is allowed by congestion and flow control, but may only send a micro burst and schedule itself for being called after the inter-burst send time using the HPTS. The RACK stack supports an application setting a pacing rate and a maximum burst size using TCP socket options. The RACK stack then uses these values to compute the actual micro burst size and the inter-burst send time.

The following IPPROTO_TCP-level socket options are used to control static pacing:

Option Name	Data Type	Semantic
TCP_RACK_PACE_RATE_SS	uint64_t	Pace rate in B/s during slow start
TCP_RACK_PACE_RATE_CA	uint64_t	Pace rate in B/s during congestion avoidance
TCP_RACK_PACE_RATE_REC	uint64_t	Pace rate in B/s during recovery
TCP_RACK_PACE_ALWAYS	int	Enable/Disable pacing
TCP_RACK_PACE_MAX_SEG	int	Micro burst size in number of full sized segments

Table 1: Socket Options

The first three options can be used to control the pace rate in B/s. It is possible to specify individual pace rates for slow start, congestion avoidance, and recovery. When initializing one of the three pace rates, the other two pace rates are also initialized to the same rate. With the fourth socket option static pacing can be enabled and disabled. The last socket option allows to control the size of the micro burst in full sized segments. The default value is 40.

The following packetdrill-script illustrates the behaviour of a sender using a pace rate of 12 Mb/s and a micro burst size of 4 full sized segments. packetdrill is available in the FreeBSD ports collection. Please note that 12 Mb/s correspond to 1.5MB/s. Since FreeBSD takes the size of the IP packet into account this corresponds to 1000 full sized segments on a path with an MTU of 1500 bytes. The script uses a round trip time of 50 ms.

```
--ip_version=ipv4

 0.000 ‘kldload -n tcp_rack‘
+0.000 ‘kldload -n cc_newreno‘
+0.000 ‘sysctl kern.timecounter.alloweddeviation=0‘

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                                     pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0
+0.000 < S      0:0(0)                  win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.     0:0(0)        ack     1 win 65535 <mss 1460,sackOK,eol,eol>
+0.050 <  .     1:1(0)        ack     1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_LOG, [TCP_LOG_STATE_CONTINUAL], 4) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_RATE_SS, [1500000], 8) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_MAX_SEG, [4], 4) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_ALWAYS, [1], 4) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 >  .     1:1461(1460)  ack     1 win 65535
+0.000 >  .  1461:2921(1460)  ack     1 win 65535
+0.000 >  .  2921:4381(1460)  ack     1 win 65535
+0.000 >  .  4381:5841(1460)  ack     1 win 65535
+0.004 >  .  5841:7301(1460)  ack     1 win 65535
+0.000 >  .  7301:8761(1460)  ack     1 win 65535
+0.000 >  .  8761:10221(1460) ack     1 win 65535
+0.000 >  . 10221:11681(1460) ack     1 win 65535
+0.004 >  . 11681:13141(1460) ack     1 win 65535
+0.000 > P. 13141:14601(1460) ack     1 win 65535
+0.042 <  .     1:1(0)        ack  2921 win 65535
+0.000 <  .     1:1(0)        ack  5841 win 65535
+0.004 <  .     1:1(0)        ack  8761 win 65535
+0.000 <  .     1:1(0)        ack 11681 win 65535
+0.004 <  .     1:1(0)        ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0)    ack     1 win 65535
+0.050 < F.      1:1(0)       ack 14602 win 65535
+0.000 >  . 14602:14602(0)    ack     2
```

The HPTS is optimized for handling a large number of TCP connections and the tcp_output() function of the RACK stack is also optimized for being called more often than the tcp_output() function of the default stack. This allows to use TSO in combination with TCP pacing.

This subsystem underpins recently published research by Netflix and Stanford into application-informed pacing at scale [Sammy].

5.4. QUIC BBR implementations

Pacing capability is expected in QUIC senders. While standard QUIC congestion control [RFC9002] is based on TCP Reno, which is not defined to include pacing (but also does not prohibit it), QUIC congestion control requires either pacing or some other burst limitation (Section 7.7 of [RFC9002]). BBR congestion control implementations are common in QUIC stacks, and pacing is integral to BBR, so this document focuses on it.

Pacing in QUIC stacks commonly involves:

1. Access to lower-level (e.g. OS and hardware) capabilities needed for effective pacing.

2. Managing additional timers related to pacing, along with those already needed for retransmission, and other events.

3. Details of the actual pacing algorithm (e.g. granularity of bursts allowed, etc.).

Examples of different approaches to dealing with these challenges in ways that work on multiple operating systems and hardware platforms can be found in open source QUIC stacks, such as Google’s QUIC implementation and Meta’s "mvfst". These provide examples for some of the concepts discussed below.

Unlike TCP implementations that typically run within the operating system kernel, QUIC implementations more typically run in user space and are thus faced with more challenges regarding timing and coupling with the underlying protocol stack and hardware needed to achieve pacing. For instance, if an application trying to do pacing is running on a highly loaded system, it may often "wake up late" and miss the times that it intends to pace packets.

When a large amount of data needs to be sent, pacing naively could result in an excessive number of timers to be managed and adjusted along with all of the other timers that the QUIC stack and rest of the application require. The Hashed Hierarchical Timing Wheel [VL87] provides one approach for such cases, but implementations may also simply schedule the next send event based on the current pacing rate, and then schedule subsequent events as needed, rather than adjusting timers for them. In any case, typically a pacing algorithm should allow for some amount of burstiness, in order to efficiently use the hardware as well as to be responsive for bursty (but low overall rate) applications, and to avoid excessive timer management.

Pacing can be done based on different approaches such as a token-based or tokenless algorithm. For instance, a tokenless algorithm (e.g. as used in mvfst) might compute a regular interval time and batch size (number of packets) to be released every interval and achieve the pacing rate. This allows specific future transmissions to be scheduled. In contrast, a token-based algorithm accumulates tokens to permit transmission based on the pacing rate, using a "leaky bucket" to control bursts. In this case the size of bursts may be more granular, depending on how much time has elapsed between evaluations.

The additional notion of "burst tokens" (or other burst allowance) may be present in order to rapidly transmit data if coming out of a quiescent period (e.g. when a flow has been application-limited without data to send, e.g. as used in Google’s implementation). A number of burst tokens, representing packets that can be sent unpaced, is initialized to some value (e.g. 10) when a flow starts or becomes quiescent. If burst tokens are available, outgoing packets are sent immediately, without pacing, up to the limit permitted by the congestion window, and the burst tokens are depleted by each packet sent. The number of burst tokens is reduced to zero on congestion events. When coming out of quiescence, it is set to the minimum of the initial burst size, or the amount of packets that the congestion window (in bytes) represents.

There may be additional "lumpy tokens" that further allow unpaced packets after the burst tokens have been consumed, and the congestion window does not limit sending. The amount of lumpy tokens that might be present is determined using heuristics, generally limiting to a small number of packets (e.g. 1 or 2).

6. Security Considerations

While congestion control designs, including aspects such as pacing, could result in unwanted competing traffic, they do not directly result in new security considerations.

Transport protocols that provide authentication (including those using encryption), or are carried over protocols that provide authentication, can protect their congestion control algorithm from network attack. This is orthogonal to the congestion control rules.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[I-D.cardwell-iccrg-bbr-congestion-control]
Cardwell, N., Cheng, Y., Yeganeh, S. H., Swett, I., and V. Jacobson, "BBR Congestion Control", Work in Progress, Internet-Draft, draft-cardwell-iccrg-bbr-congestion-control-02, 7 March 2022, <https://datatracker.ietf.org/doc/html/draft-cardwell-iccrg-bbr-congestion-control-02>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

8.2. Informative References

[HPTS] Stewart, R. and M. T端xen, "Pacing in the FreeBSD TCP Stack", FreeBSD Journal , October 2024, <https://freebsdfoundation.org/wp-content/uploads/2024/11/stewart-adventures.pdf>.

[HPTSCode] "tcp_hpts.c", October 2024, <https://github.com/freebsd/freebsd-src/blob/main/sys/netinet/tcp_hpts.c#L31-L99>.

[I-D.draft-johansson-ccwg-rfc8298bis-screamv2-03]
Johansson, I., Westerlund, M., and M. K端hlewind, "Self-Clocked Rate Adaptation for Multimedia", Work in Progress, Internet-Draft, draft-johansson-ccwg-rfc8298bis-screamv2-03, 3 March 2025, <https://datatracker.ietf.org/doc/html/draft-johansson-ccwg-rfc8298bis-screamv2-03>.

[I-D.draft-many-tiptop-usecase]
Blanchet, M., Eddy, W., and M. Eubanks, "IP in Deep Space: Key Characteristics, Use Cases and Requirements", Work in Progress, Internet-Draft, draft-many-tiptop-usecase-03, 18 June 2025, <https://datatracker.ietf.org/doc/html/draft-many-tiptop-usecase-03>.

[LinuxPacing]
Welzl, M., "TCP Pacing in the Linux Kernel", IEEE ICNC 2025 , 17 February 2025, <https://folk.universitetetioslo.no/michawe/research/publications/icnc2025-pacing.pdf>.

[RACK] Stewart, R. and M. T端xen, "RACK and Alternate TCP Stacks for FreeBSD", FreeBSD Journal (January/February 2024) , February 2024, <https://freebsdfoundation.org/our-work/journal/browser-based-edition/networking-10th-anniversary/rack-and-alternate-tcp-stacks-for-freebsd/>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/rfc/rfc5681>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/rfc/rfc6298>.

[RFC9002] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/rfc/rfc9002>.

[RFC9040] Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", RFC 9040, DOI 10.17487/RFC9040, July 2021, <https://www.rfc-editor.org/rfc/rfc9040>.

[Sammy] Spang, B., Kunamalla, S., Teixeira, R., Huang, T., Armitage, G., Johari, R., and N. McKeown, "Sammy: Smoothing Video Traffic to be a Friendly Internet Neighbor", ACM SIGCOMM '23: Proceedings of the ACM SIGCOMM 2023 Conference , 1 September 2023, <https://doi.org/10.1145/3603269.3604839>.

[UnderstandingPacing]
Aggarwal, A., Savage, S., and T. Anderson, "Understanding the performance of TCP pacing", IEEE Infocom 2000 , March 2000, <https://doi.org/10.1109/INFCOM.2000.832483>.

[VL87] Varghese, G. and T. Tauck, "Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility", DOI 10.1145/37499.37504, 1 November 1987, <https://doi.org/10.1145/37499.37504>.

Acknowledgments

The authors would like to thank Grenville Armitage, Ingemar Johansson and Eduard Vasilenko for suggesting improvements to this document.

Change Log

* -00 was the first individual submission for feedback by ICCRG.

* -01 adds Michael Tuexen as new co-author, and adds the following new descriptions:

  - a first version of text for Apple OSes

  - a first version of text for FreeBSD

* -02 adds a reference for Linux pacing, removes a comment on Linux SRTT calculation ("TODO check": it was checked, nothing to change), adds more discussion text:

  - queue dynamics / AQM interactions

  - initial RTT calculation, thanks to Ingemar Johansson

  - mini-bursts and their trade-offs, thanks to Eduard Vasilenko

  - ... and now we also have an ACK section for such thanks

  - -03:

    o re-structures the parts on general considerations and adds more text in them

    o elaborates on application control

Authors' Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
0316 Oslo
Norway
Email: michawe@ifi.uio.no

URI: http://welzl.at/

Wesley Eddy
Aalyria Technologies
Avon, OH 44011,
United States of America
Email: wes@aalyria.com

Vidhi Goel
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America
Email: vidhi_goel@apple.com

Michael T端xen
M端nster University of Applied Sciences
Stegerwaldstrasse 39
48565 Steinfurt
Germany
Email: tuexen@fh-muenster.de