Considerations when designing a congestion control

MPG[1,2]

[1]Simula Research Laboratory, Norway
[2]IFI, University of Oslo, Norway

July 17, 2015

This document is currently in a state of a rough tech report (work in progress), and is a request for comments.

1 About this document

In this document we discuss aspects of CC designs in regards to fairness towards network traffic produced by time-dependent applications, i.e., thin streams.

We also aim to document the significance of byte based vs. packet based accounting for thin streams through a study of FreeBSD and Linux versions with different properties. We argue that byte-based accounting is a framework that intrinsically allows for better tuning of CWND behavior for thin streams when the goal is to transmit the data as soon as possible. We also suggest work on developing proper limitations on byte-based behavior in order to avoid wasteful behavior for aggressive, but still application-limited, flows.

2 Introduction

Congestion control (CC) algorithms are used to adjust a TCP sender's send-rate according to the capacity of a network. This is done by using a congestion windows (CWND), defining the maximum amount of data a sender is currently allowed to transmit. This can both be implemented in terms of a number of MSS-sized segments or bytes, and current TCP standards leave the choice between byte-based or segment-based accounting as an implementation detail.

Linux uses a segment-based CWND, while FreeBSD implements a byte-based accounting. For greedy TCP flows, the difference between a byte-based and a segment-based accounting is negligible, as these flows mainly transmit MSS-sized segments. However, this is not the case for thin flows who frequently transmit smaller-sized segments where a segment-based accounting can cause issues.

The lack of a discussion on the impact either form of accounting may have, illustrates what we see as a symptomatic issue: focusing on how congestion control (CC) can limit aggressive (greedy) flows effectively and leaving the impact it may have on non-greedy flows largely ignored. We believe there is a systematic negligence of non-greedy flows in CC design, and this calls for an increased awareness.

The basis for today's CC originates from the congestion collapse in the early Internet[1, 2]. The underlying principle of today's CC algorithms is about preventing network congestion while still achieving a fair division of the available network capacity among competing flows. Decades of development in networking technology has lead to greatly increased bandwidths compared to the early days of the Internet. The design of CC algorithms are still mainly focused on limiting flows that try to probe the available bandwidth and do not consider flows that are limited by the application, such as online gaming and VoIP traffic. In practice, this means that these CC algorithms are in fact *not* being fair. In other words, today's CC algorithms are dependent on all flows trying to achieve the same throughput in order to achieve fairness, also known as the *max-min fairness*.

Flows that are application-limited may suffer from a disproportional reaction by the CC relative to the level of congestion they contribute to. These flows often have other requirements than achieving the highest possible throughput, such as lowest possible latency or sending at a fixed data-rate. Despite having been shown to be unfair, max-min fairness is still used as the fundamental fairness method[3].

3 Congestion Control fairness

The traditional loss-based CC mechanisms rely on loss detection for estimating the amount of congestion in the network. An inherent problem is that they must cause loss to find the limitations of the network. This also results in loss for other network streams sharing the same resources.

Thin streams will always lose when competing against greedy streams, because they do not actively seek more bandwidth, while still being "punished" as if they were greedy by having the CWND halved on fast retransmit. We argue that congestion control algorithms should use a fairness definition that does not consider bandwidth as its only fairness metric.

Research has been done on trying to find alternative CCs for TCP, that rely on other metrics than loss, or loss alone, to measure congestion.

3.1 Equation based Congestion Control

The need to define how a CC should behave in a formal way has lead to the development of equation based CCs, where the send rate is adjusted based on a function on some property, such as loss even rate. TCP is known to constantly and aggressively probe the network to find the maximum send rate. The primary goal of the equation based CCs is to avoid this, by finding and maintaining a steady send rate, that does not fluctuate as much as TCP's, while still reacting to network congestion.

3.1.1 TFRC

TFRC is a CC for non-TCP streams operating in a best-effort network, such as the Internet [4]. It is designed for applications that not only value maximum throughput, and it specifically aims to reduce the variations in throughput compared to TCP.

Streaming media and VoIP are such examples, where stable throughput is very important to maintain a smooth playback of video or audio. TFRC is designed to be "reasonably fair when competing for bandwidth with TCP flows," [5] where a reasonably fair stream is defined as "if its sending rate is generally within a factor of two of the sending rate of a TCP flow under the same conditions". Most importantly, TFRC is designed to avoid the drastic reduction of the CWND of TCP's multiplicative decrease phase, when loss is detected.

3.1.2 TFRC Small-Packet variant

RFC4828 defines TFRC-SP, a modification of TFRC for applications that send small packets, where the design goal is to achieve the same bandwidth as a TCP stream sending packets of size MSS [6]. It specifies that a minimum ITT of 10 ms should be enforced to prevent a stream from sending too many small packets. This limit is based on the requirement of VoIP applications, that do not send packets more often than 10 ms. Also, this limit is justified by the lack of applications used in in the current Internet that require to send small packets more often.

3.2 Binomial Congestion Control Algorithms

Another type of CC is the binomial congestion control algorithms [7]. They generalize TCP's additive increase and multiplicative decrease using two properties k and l, where an algorithm is TCP-compatible (TCP friendly) when it satisfies $k + l = 1$.

Additive increase is generalized by increasing inversely proportional to the power k of the current window, and multiplicative decrease, by decreasing proportional to the power k of the current window. This is formalized as

$$\begin{aligned} I &: \omega_{t+R} \leftarrow \omega_t + \alpha/\omega_t^k; \alpha > 0 \\ D &: \omega_{t+\delta_t} \leftarrow \omega_t - \beta\omega_t^l; 0 < \beta < 1 \end{aligned} \tag{1}$$

where I is the increase after receiving ACKs for a window within an RTT, D is the decrease of the window after a loss/congestion event. ω_t is the window size at time t, R is the RTT, and α and β are

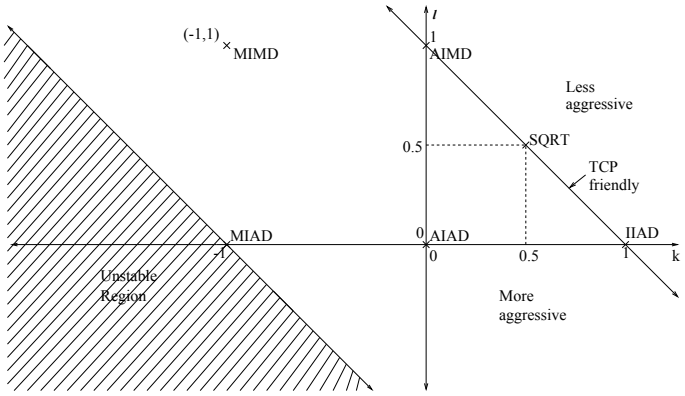


Figure 1: The k, l space of nonlinear controls from the binomial algorithms, with the $k + l = 1$ line showing the set of TCP-compatible controls. [7]

constants or functions of the current window size. Equation 1 generalizes the linear control algorithms:

$k = 0$, $l = 1$: AIMD (TCP)
$k = -1$, $l = 1$: MIMD (multiplicative increase/multiplicative decrease)
$k = -1$, $l = 0$: MIAD
$k = 0$, $l = 0$: AIAD

Figure 1 shows the k, l space of nonlinear controls, where it corresponds with the four linear algorithms, and the $k + l = 1$ line which is the set of TCP-compatible controls.

The idea is that by adjusting the k and l properties, the CC can be made to better fit the needs of applications such as audio and video streaming, where a drastic reduction of the send rate on loss events is very damaging.

An important difference with the binomial CCs compared to other equation based CCs such as TFRC is that it does not make use of a calculated loss rate.

4 Segment-based CWND

Properly accounting for the amount of acknowledged data is simple by looking at sequence numbers that are ACKed, however, when increasing the segment-based CWND it is impossible to increase the CWND by less than 1 MSS. Therefore, a segment-based CWND has an inherent short-coming in regards to streams that send smaller segments. The following sections discusses some issues specific to the segment based CWND implementation in the Linux kernel, and presents some results from a set of tests performed in a testbed.

4.1 Appropriate Byte Accounting for segment-based CWND

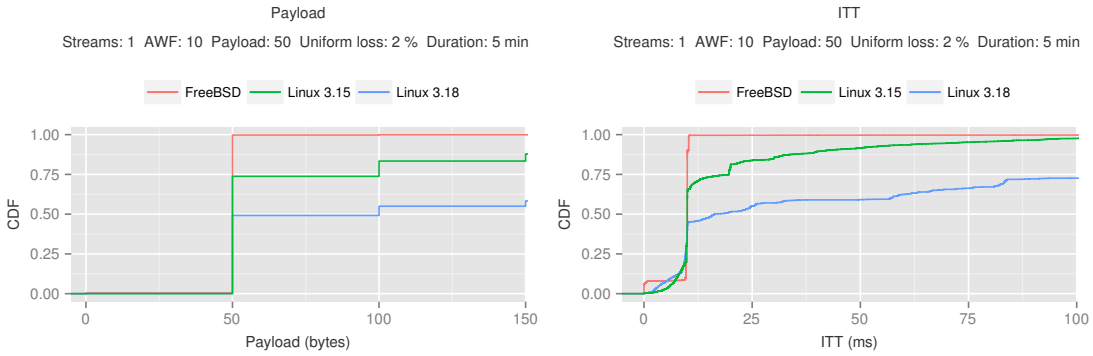
When packets in a thin stream are ACKed, they will lead to an increase of the CWND that does not correspond to the available link bandwidth confirmed by the successful packet transmission. When receiving an ACK on a packet with e.g. 120 bytes payload, the CWND will be increased as if the packet contained MSS worth of payload. RFC5681 recommends increasing the CWND using

$$cwnd += min(N, SMSS) \tag{2}$$

where N is the number of bytes ACKed. This is part of Appropriate Byte Counting (ABC) (rfc3465). However, ABC was removed from Linux in 2013 (`http://comments.gmane.org/gmane.linux.network/257896`).

4.2 Congestion Window Validation

Further, TCP implementations, such as in the Linux kernel, implement functionality to prevent unnecessarily growing the CWND. TCP Congestion Window Validation is a mechanism to validate the CWND



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
FreeBSD	fsender	1	9 / 9 / 9	50/50/50	3086/1448/2896/2898/2898/7242
Linux 3.15	rdbsender	1	18 / 18 / 18	96/96/96	10/1/7/9/13/20
Linux 3.18	rdbsender	1	51 / 51 / 51	264/264/264	3/1/3/3/3/32

(a)

Name	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
FreeBSD	30.5 K	563	559	11	1.5 M	1.5 M	53.6 K	45.4 K	1.95 / 1.87	8527
Linux 3.15	15.8 K	323	322	0	1.5 M	1.5 M	32.9 K	32.9 K	2.17 / 2.05	2799
Linux 3.18	5.8 K	130	128	0	1.5 M	1.5 M	36.8 K	36.8 K	2.44 / 2.26	473

(b)

Figure 2

on streams after application-limited periods [8]. The RFC also proposes not to grow the CWND for application-limited streams: "We propose that the TCP sender should not increase the congestion window when the TCP sender has been application-limited (and therefore has not fully used the current congestion window).".

In the context of thin streams it is necessary to be expressively precise to avoid mixing of segment-based and byte-based accounting. As thin streams do not send MSS-sized packets, they do not fully utilize the CWND in terms of bytes, however, they still require the CWND to be sufficiently large to allow them to transmit packets without increased sojourn times on the sender side.

4.2.1 New Congestion Window Validation (new CWV)

The new CWV implementation circumvents the issue (bug) described in the next section, as it ignores the `is_cwnd_limited` value in the test entirely.

4.3 Linux kernel bug that behaves as a forced Nagle

Changes applied in Linux 3.16 have changed how the CWND is controlled, which greatly affects latencies for thin streams [9]. In Linux 3.15, the function that increases the CWND in the CC would at call time simply test if the number of PIFs was not less than the CWND. If so, it would not increase the congestion window. This worked well for thin streams with Nagle disabled, because as long as the thin stream needed more PIFs it would increase the CWND.

The changes in Linux 3.16 moves the cwnd-limited test to the TCP output-engine, and remembers whether it was cwnd-limited when finished sending the last window (stored in `tp->is_cwnd_limited`). Commit ca8a22 in Linux 3.16 states:

This commit improves both of these, by:

1. Switching to a measurement-based approach where we explicitly track the largest number of packets in flight during the past window ("max_packets_out"), and remember whether we were cwnd-limited at the moment we finished sending that flight.

2. Only allowing cwnd to grow to twice the number of outstanding packets ("max_packets_out") in slow start. In congestion avoidance mode we now only allow cwnd to grow if it was fully utilized.

The result is that after the CWND is reduced due to packet loss, the CWND does not grow back to the previous level. This prevents the sender from maintaining the number of PIFs it needs to transmit the data segments without being delayed in the TCP output queue. When the CWND does not grow, more data is buffered in the TCP output queue leading to bigger packets being sent, which effectively enforces a behavior similar to Nagle's Algorithm.

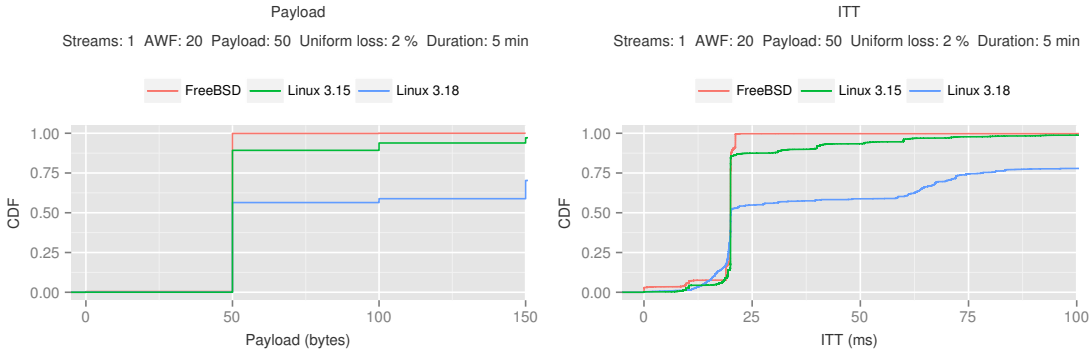
Figure 2 shows the results from a test comparing the new Linux kernel to the Linux 3.15 and FreeBSD 10.1. The RTT is 150 ms for the experiments. Subfigure 2a shows a CDF of the packet payload sizes, and subfigure 2b shows the ITTs of the packets sent from the sender hosts. In this test, each stream is produced by an application that writes 50 bytes with an application write frequency (AWF) of 10 ms.

The effect of the recent changes to the Linux kernel is evident by comparing the payloads and ITTs for Linux 3.18 to Linux 3.15. Worth noting is also a significant difference between Linux 3.15 and FreeBSD 10.1. From subfigure 2b we can see that FreeBSD 10.1 transmits twice as many packets as Linux 3.15, and Linux 3.15 transmits more than twice that of Linux 3.18.

We see similar effects in figure 3 from a test with application write frequency (AWF) of 20 ms. However, we can see that the difference between Linux 3.15 and FreeBSD 10.1 is much less prominent compared to the tests with AWF of 10 ms.

In figure 4, where the application are set up with AWF 50 ms, we see can see that the TCP types have similar performances, where the total number of packets sent is almost the same. The reason the changes to Linux 3.16 have less effect in this test is because the streams only request 3 PIFs ($150ms/50ms$), which the CWND of Linux 3.18 already allows to be sent, as seen from the CWND values for all the tests.

In all the tests, TCP_NODELAY was enabled which disables Nagle, however as the experiments show, Linux 3.18 enforces a behavior that produces an effect similar to Nagle, where parts of the data from the application is delayed in the output queue before being transmitted.

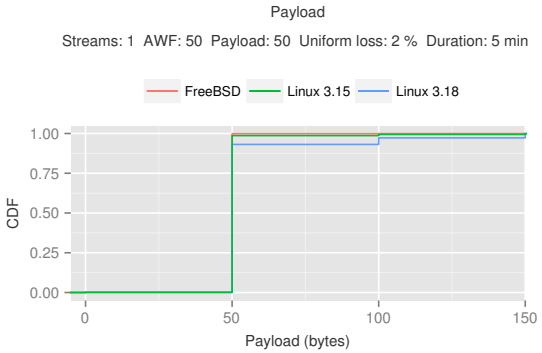


Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
FreeBSD	fsender	1	19 / 19 / 19	49/49/49	2928/1448/2598/2898/2898/5828
Linux 3.15	rdbsender	1	24 / 24 / 24	61/61/61	9/1/6/9/12/12
Linux 3.18	rdbsender	1	50 / 50 / 50	128/128/128	4/1/3/3/3/18

Name	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
FreeBSD	15.4 K	328	332	0	766 K	743 K	23.5 K	23.5 K	2.13 / 2.15	2882
Linux 3.15	12.4 K	262	261	0	762 K	744 K	17.9 K	17.9 K	2.35 / 2.12	1868
Linux 3.18	5.9 K	120	119	0	759 K	744 K	15.4 K	15.4 K	2.04 / 2.02	412

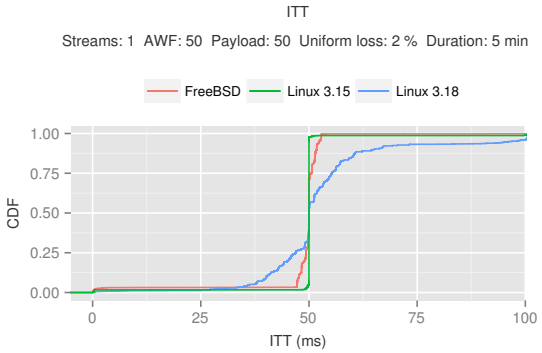
(a) (b)

Figure 3



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
FreeBSD	fsender	1	48 / 48 / 48	50/50/50	2811/1448/2248/2898/2898/5794
Linux 3.15	rdbsender	1	50 / 50 / 50	51/51/51	7/1/6/8/8/10
Linux 3.18	rdbsender	1	53 / 53 / 53	54/54/54	3/1/3/3/3/10

(a)



Name	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
FreeBSD	6.1 K	128	121	0	304 K	297 K	6.3 K	6.3 K	2.09 / 2.10	657
Linux 3.15	5.9 K	117	116	0	304 K	298 K	6.1 K	6.1 K	2.01 / 1.97	583
Linux 3.18	5.6 K	138	137	0	305 K	298 K	7.5 K	7.5 K	2.46 / 2.48	501

(b)

Figure 4

5 Byte-based CWND

In the Linux kernel's segment-based implementation, the minimum value for the CWND is 1, which permits $1 * MSS$ bytes in flight. However, as the CWND represents the maximum number of MSS-sized packets in flight, a stream that tries to transmit data segments of 10 bytes is only allowed to have one segment in flight when the CWND is 1.

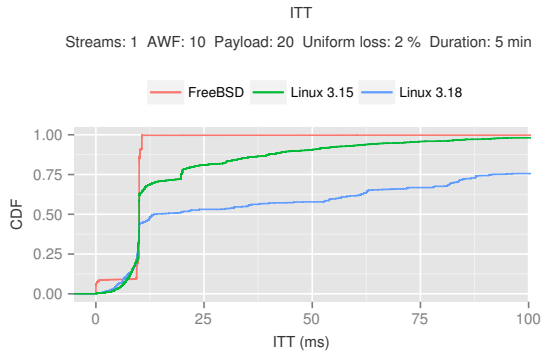
FreeBSD's byte-based implementation enforces the same minimum CWND in terms of $1 * MSS$ bytes in flight. However, there is no limit on the number of packets used to transmit these $1 * MSS$ bytes. Streams that send small sized packets are therefore treated vastly different in a byte-based CWND implementation compared to a segment-based CWND implementation.

Figure 5 shows the results from a test with AWF of 10 ms with 20 bytes written per call. From the table in 5a we can see that the number of packets sent ("Sent p") is significantly higher for FreeBSD 10.1 (31.5K) than for Linux 3.15 (15.3K) and Linux 3.18 (5.8K).

This is due to the segment-based CWND on Linux holding back data segments to a large degree. Figure 5b shows the sojourn time measured for each segment, from the time it was written to the kernel by the application until it was transmitted for the first time. From the table we can see that 50% of the data segments sent by the application on Linux 3.15 have a sojourn time of 9 ms or more, and on Linux 3.18, 50% have a sojourn time of 45 ms or more. Compared to FreeBSD 10.1 where all the segments have a sojourn time of 1 ms, it is clear that services depending on thin streams may benefit greatly by using FreeBSD instead of Linux.

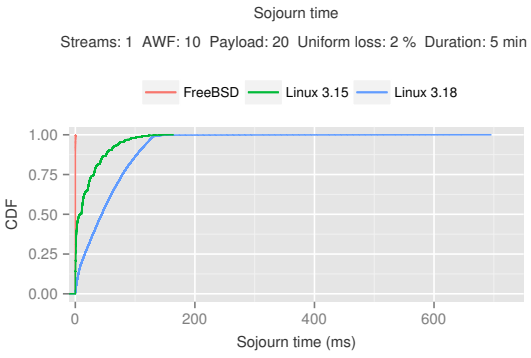
Figure 5c shows both the per-segment (written by the application) latency values as well as the sojourn time plotted for each of the three hosts.

Figure 6 shows the results from tests with an AWF of 20 ms. In these tests the gap in performance between Linux 3.15 and FreeBSD 10.1 is much smaller than the previous tests. However, from 6b we can see that 10% of the segments fro Linux 3.15 have a sojourn time of at least 31 ms. For Linux 3.18, we can see that 50% of the segments have a sojourn time of at least 33 ms, and 10% have a sojourn time of at least 89 ms.



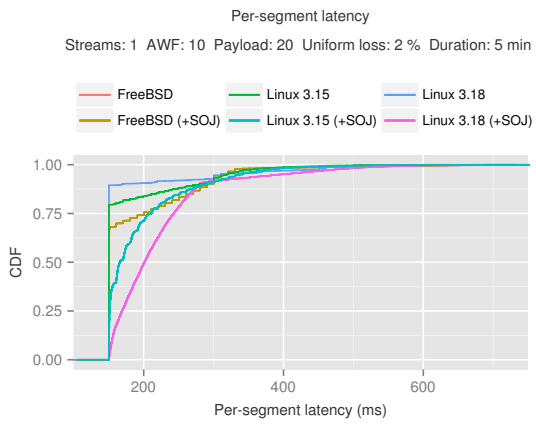
Name	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
FreeBSD	31.5 K	620	622	0	626 K	595 K	31.4 K	31.4 K	1.97 / 2.03	9103
Linux 3.15	15.3 K	333	332	0	608 K	595 K	12.6 K	12.6 K	2.07 / 2.18	2861
Linux 3.18	5.8 K	111	110	0	607 K	595 K	12.0 K	12.0 K	1.97 / 1.90	438

(a)



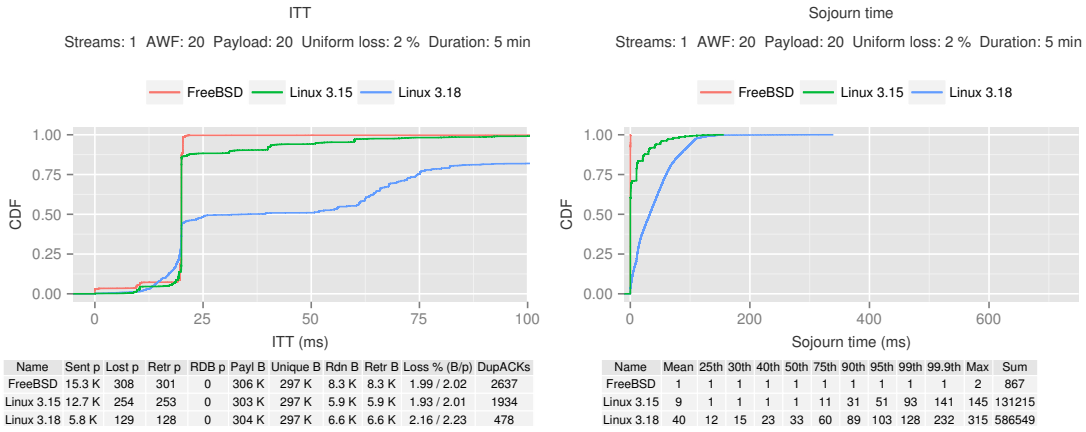
Name	Mean	25th	30th	40th	50th	75th	90th	95th	99th	99.9th	Max	Sum
FreeBSD	1	1	1	1	1	1	1	1	1	1	2	1669
Linux 3.15	20	1	1	3	9	31	56	75	112	142	153	572537
Linux 3.18	51	16	22	33	45	79	109	121	132	348	645	1502580

(b)



(c)

Figure 5

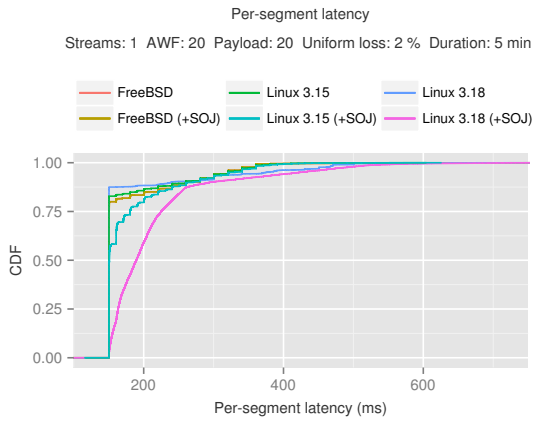


Name	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
FreeBSD	15.3 K	308	301	0	306 K	297 K	8.3 K	8.3 K	1.99 / 2.02	2637
Linux 3.15	12.7 K	254	253	0	303 K	297 K	5.9 K	5.9 K	1.93 / 2.01	1934
Linux 3.18	5.8 K	129	128	0	304 K	297 K	6.6 K	6.6 K	2.16 / 2.23	478

(a)

Name	Mean	25th	30th	40th	50th	75th	90th	95th	99th	99.9th	Max	Sum
FreeBSD	1	1	1	1	1	1	1	1	1	1	2	867
Linux 3.15	9	1	1	1	1	11	31	51	93	141	145	131215
Linux 3.18	40	12	15	23	33	60	89	103	128	232	315	586549

(b)



(c)

Figure 6

5.1 Potential issues

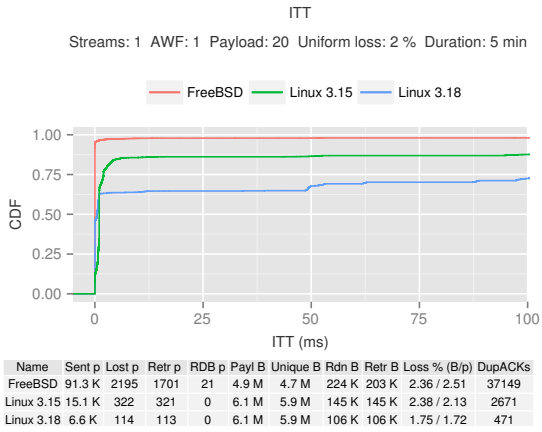
Having no limitation on the number of packets used to transmit the $1 * MSS$ bytes of data may seem unfair. Generating more packets to transfer the same amount of data will cause more overhead considering the ratio between goodput/throughput per packet and the headers required for each packet.

Figure 7 shows the results from tests where the application write frequency (AWF) has been reduced to 1 ms. The results show that the difference between numbers of packets transmitted is much bigger (table in figure 7a) compared to the tests with 10 ms AWF. The FreeBSD 10.1 stream sends 6 times as many packets as the Linux 3.15 stream, and 13 times as many as Linux 3.18. One might expect that this would show similar results as the previous tests, where the lower sojourn times on the FreeBSD 10.1 stream give lower latencies. From 7 we see that this is not the case at all, where the latencies are in fact significantly higher on the FreeBSD 10.1 stream for most segments. This shows that it is not necessarily beneficial for the latency to transmit as many packets as possible to ensure a minimal sojourn time.

If sending more packets to transmit the same amount of data is considered unfair, or too wasteful of the network resources, mechanisms to account for this could be implemented. By keeping track of the number of packets in flight, it is possible to include an overhead factor when deciding whether more data should be sent. If there are e.g. 20 packets in flight, $20 \times headersize$ could be added to the calculation when testing if the CWND allows more data to be sent. This would make a difference when the CWND is small, and the number of packets in flight is large relative to the CWND size, however, for a greedy stream with a large CWND, and few packets in flight, it would not alter the current behavior in a significant manner.

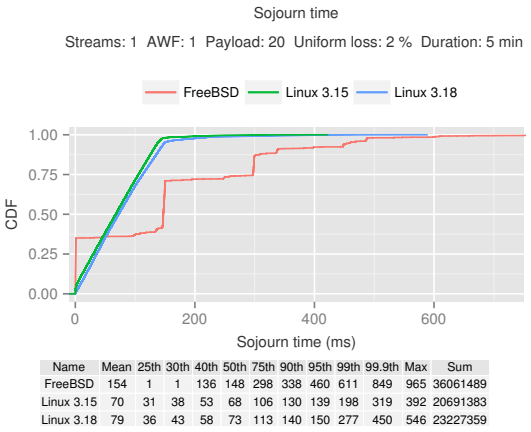
5.2 Benefits with byte-based CWND

As the CWND in Linux is measured in units of MSS-sized packets, it is currently not possible to represent fractions of such units. With byte-based CWND this is not an issue as the CWND is represented in the smallest possible unit available, i.e. bytes. This allows to represent sub-MSS CWND out of the box, instead of the need to alter the segment-based CWND in Linux such as discussed in "Scaling TCP's Congestion Window for Small Round Trip Times" [10].



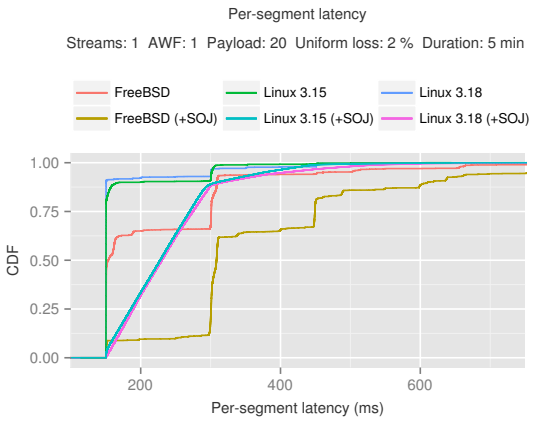
Name	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
FreeBSD	91.3 K	2195	1701	21	4.9 M	4.7 M	224 K	203 K	2.36 / 2.51	37149
Linux 3.15	15.1 K	322	321	0	6.1 M	5.9 M	145 K	145 K	2.38 / 2.13	2671
Linux 3.18	6.6 K	114	113	0	6.1 M	5.9 M	106 K	106 K	1.75 / 1.72	471

(a)



Name	Mean	25th	30th	40th	50th	75th	90th	95th	99th	99.9th	Max	Sum
FreeBSD	154	1	1	136	148	298	338	460	611	849	965	36061489
Linux 3.15	70	31	38	53	68	106	130	139	198	319	392	20691383
Linux 3.18	79	36	43	58	73	113	140	150	277	450	546	23227359

(b)



(c)

Figure 7

References

[1] V. Jacobson, "Congestion avoidance and control," *SIGCOMM Comput. Commun. Rev.*, vol. 18, pp. 314–329, Aug. 1988.

[2] J. Nagle, "Congestion Control in IP/TCP Internetworks." RFC 896, Jan. 1984.

[3] B. Briscoe, "Flow rate fairness: Dismantling a religion," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 63–74, Mar. 2007.

[4] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," *SIGCOMM Comput. Commun. Rev.*, vol. 30, pp. 43–56, Aug. 2000.

[5] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification." RFC 5348 (Proposed Standard), Sept. 2008.

[6] S. Floyd and E. Kohler, "TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant." RFC 4828 (Experimental), Apr. 2007.

[7] D. Bansal and H. Balakrishnan, "Binomial congestion control algorithms," in *INFOCOM*, 2001.

[8] M. Handley, J. Padhye, and S. Floyd, "TCP Congestion Window Validation." RFC 2861 (Experimental), June 2000.

[9] Git commits, "Changes to CWND growth." Available: http://git.kernel.org/linus/ca8a22 and http://git.kernel.org/linus/e114a7.

[10] B. Briscoe, "Scaling TCP's congestion window for small round trip times," May 2015. RITE internal document.