Battling Bufferbloat
An experimental comparison of four approaches to queue management in Linux

Toke Høiland-Jørgensen
toke@toke.dk

Master module project
Computer Science, Roskilde University
17th December 2012

Abstract

The term bufferbloat has been coined to describe the problem that occurs when computer network buffers misbehave, inducing unnecessary latency. If the buffers are too large, packets sit in the buffer queues instead of being dropped, and no signals of congestion reach the endpoints; meaning that they do not slow down in a timely manner, and the buffers stay full, severely increasing latency. This project aims to demonstrate the bufferbloat problem, and provide an overview of the current state of the art in controlling bufferbloat in the Linux operating system. To achieve this, a theoretical overview is combined with experimental data from a controlled testing environment, in which the test hosts are configured per the bufferbloat community guidelines to maximise the kernel queueing disciplines’ control of the packet queue.

On this setup three tests are conducted: A simple bidirectional TCP test, a Realtime Response Under Load (RRUL) test designed by the bufferbloat community, and a UDP flood test. Each test is repeated for each of the four tested queueing disciplines: the default pfifo_fast, the CoDel algorithm implementation (codel qdisc), the Stochastic Fairness Queueing (sfq) qdisc and finally the combination of fairness queueing and CoDel queue management as implemented in the fq_codel qdisc. Tests are run using the netperf-wrapper testing harness developed for this purpose by the author.

The test results correspond well with the expectations, and show that the default pfifo_fast qdisc suffers from decreased throughput and significantly increased latency under load, with all the other qdiscs providing orders of magnitude lower latency when the link is loaded. The results show that the CoDel algorithm can manage the queues of well-behaved TCP streams to some extent, providing somewhat lower latencies. However, ill-behaved streams, as tested in the UDP flood test, show the need for fairness queueing to distribute bandwidth among flows. The combination of fairness queueing with the CoDel algorithm, as implemented in the fq_codel qdisc, exhibits significantly lower latency under load than any of the other qdiscs when the number of concurrent bidirectional streams increase beyond one. Total throughput when using fq_codel is slightly lower than using sfq; however, for multiple flows the loss of total throughput is relatively small (2.2%) compared to the gains in latency (66%).

Overall, the results of using the fq_codel qdisc are encouraging, showing a decrease in latency under load to less than 20 milliseconds, from almost a second when using the default pfifo_fast qdisc.
## Contents

1 Introduction ................................................. 3

2 The Transmission Control Protocol ............................ 5
   2.1 Basic operation of TCP ................................. 5
   2.2 Congestion control ....................................... 6
   2.3 TCP’s interaction with bufferbloat ....................... 9

3 Packet transmission and queueing in the Linux kernel ....... 10
   3.1 The Linux IP network stack .............................. 10
   3.2 The Traffic Control subsystem ........................... 11
   3.3 Recent enhancements to the Linux kernel .............. 11

4 Queue management algorithms .................................. 13
   4.1 The default FIFO queueing algorithm (pfifo_fast) .... 13
   4.2 Stochastic Fairness Queueing (SFQ) ..................... 13
   4.3 Controlled Delay (CoDel) ............................... 14

5 Experimental work ............................................ 16
   5.1 The test setup ........................................... 16
   5.2 The netperf-wrapper testing harness .................... 16
   5.3 Test descriptions ...................................... 17

6 Results .................................................... 19
   6.1 Bidirectional TCP test ................................. 19
   6.2 RRUL test .............................................. 21
   6.3 UDP flood test ......................................... 23
   6.4 Summarising the results ................................ 25

7 Conclusions ................................................. 26

8 Further work ............................................... 27

9 References .................................................. 28
1 Introduction

Modern computer networks are incredibly complex, and decades of research has gone into ensuring they work reliably on the global internet scale in the face of unreliable transport links and varying bandwidth and latency. We rely on computer networks to transport huge quantities of data across the globe every second, and this works well, but not always without glitches. In recent years, as bandwidth-heavy applications such as peer-to-peer networks and web sites relying on user generated content have become more prevalent, especially the relatively slow residential broadband links have been used at full capacity, and interruptions in connectivity\(^1\) have become more common.

An overloaded link need not automatically result in dramatic increases in latency, and indeed the fact that it does, along with the reasons for it, have long been poorly understood. However, recent research has shown that the culprit is the buffers built in to every piece of network equipment $[\text{GN11}]$. The term *bufferbloat* has been coined to describe the problem that occurs when these buffers misbehave to induce unnecessary latency $[\text{Sta12}]$.

Some buffering is needed for the proper functioning of the network, to absorb bursts of traffic and ensure links are utilised to their full capacity. However, the network transport protocols rely on congestion along the network path being signalled to the endpoints in a timely manner (in the form of packet drops or explicit congestion notification) to adjust transmission rates to the available bandwidth. If the buffers are too large, packets sit in the buffer queues instead of being dropped, and no signals reach the endpoints; meaning that they do not slow down in a timely manner, and the buffers stay full. Furthermore, each new packet that enters the buffer has to wait for all packets queued before it to be transmitted before it can continue on its way. These two effects combine to create what is referred to as bufferbloat; and the result is a loss of interactivity that can cause applications to time out, and users to experience huge delays in their network usage. The amount of extra latency induced is a function of the available bandwidth (a fixed size buffer takes longer to transmit on a slower link), so the problem is most prevalent at the network edge where connections are slower.

To avoid the queues induced by the buffers creating problems for traffic, employing active queue management (AQM) has long been recommended practice $[\text{RFC2309}]$. However, historically the most prevalent AQM algorithms $[\text{FJ93; Fen+02; Fen+01}]$ required tuning for varying network conditions, leading to many networks running entirely without AQM $[\text{GN11}]$. This, combined with buffers becoming larger because the defaults are tuned for high-end connections not available to the majority of users, has exacerbated the bufferbloat problem to the point where it is now a major problem for many internet connections, making interactive applications such as teleconferencing and online gaming entirely infeasible over loaded links.

Recent work on finding ways to mitigate the bufferbloat problem has resulted in the development of a new AQM algorithm called ‘Controlled Delay’ (CoDel) that does not require tuning the same way previous algorithms have $[\text{NJ12}]$. Furthermore, an internet community has been established around creating tools for measuring and mitigating bufferbloat, as well as doing further research into the causes for it $[\text{Bloat}]$. Most of this work has been centred around improving buffer management and network latency in the Linux kernel, with various improvements appearing in recent versions, among them an implementation of the CoDel AQM algorithm.

This project aims to demonstrate the bufferbloat problem, and provide an overview of the current state of the art in controlling bufferbloat in the Linux operating system. To achieve this, a theoretical overview is combined with experimental data from a controlled testing environment. The report is structured into a background section, providing an overview of the workings of the most prevalent transport protocol employed on the internet, the Transmission Control Protocol, as well as outline how network packets are handled in the Linux kernel. Following this background section, the tested queue management algorithms are examined. Based on these theoretical sections, the experimental work is presented, comprised of the experimental setup, the

\(^1\) Or, more precisely, loss of interactivity due to high latency, perceived as interruptions.
question the experiments seek to answer, and the results. Finally, some concluding remarks on the state of bufferbloat mitigation and some suggestions for further work are presented.

One of the difficulties in measuring bufferbloat has been the lack of testing tools that adequately demonstrates the presence of bufferbloat. Various network testing tools exist, but several have to be run in concert and their results combined to show the problem. As part of the experimental work for this project, a testing harness has been developed, that allows several network tests to be run at once and have the results aggregated [Høi12]. A short description of the workings of this tool is given in the section on the experimental setup.
2 The Transmission Control Protocol

The Transmission Control Protocol (TCP), is the main transport protocol of the internet, providing reliable host-to-host communication over unreliable transport media [RFC793]. The protocol has had numerous aspects modified since it was originally defined in 1981, but the basic mode of operation remains the same. This section aims to provide an up-to-date overview of the workings of the modern TCP specification, focusing on the aspects of the protocol that is relevant for network congestion control.

2.1 Basic operation of TCP

The basic operation of TCP is described in terms of two hosts exchanging segments of data. This section does not go into detail with how the underlying network carries segments between hosts, but merely assumes that any given segment either reaches the destination in its entirety (at some point), or not at all.

A connection between two hosts is identified uniquely at each end by the host’s network address\(^2\) and a 16 bit port number, and is initiated by a three-way handshake between the hosts, synchronising their respective sequence numbers. The sequence numbers serve as the mechanism through which reliable data transport is accomplished through an unreliable network medium; this will be elaborated further below. A connection is full-duplex, but in the following it is described in terms of a sending and a receiving host; in practice both hosts can act as sender and receiver simultaneously.

2.1.1 Transmission of data

Having set up a connection, the sending host has picked an initial sequence number, communicated this to the receiving host, and received acknowledgement that the sequence number has been received. The sequence number is a 32-bit number\(^3\) used to count the bytes transmitted, so they can be acknowledged by the receiver. That is, starting from the initial sequence number, each data byte sent as part of the connection has a corresponding sequence number\(^4\); and only after having being acknowledged by the receiver is the data considered to be transmitted successfully.

To allow the sender to send larger portions of data at a time (thus utilising the network resources more efficiently), the receiver advertises a window of data that it is prepared to receive. This window is a number of bytes the receiver can handle at a time, given the available buffering and processing constraints. Data in excess of the window is assumed to be dropped, so the sender transmits no more data than allowed by the window before waiting for acknowledgement of the sent data.

If (as is often the case) a connection between two hosts has a capacity bottleneck along the way, the receiver window as communicated by TCP is not necessarily a very good indicator of the optimal rate of data transmission. To remedy this, a later addition to the TCP specification defines a congestion window, which acts as an estimate of the available bandwidth between the connected hosts. Instead of using just the advertised receiving window as the transmit limit, the minimum of the receiver window and the congestion window is used. The congestion control mechanisms are discussed in detail in section 2.2.

\(^2\)E.g. an IPv4 or IPv6 address.
\(^3\)The sequence numbers wrap around after exceeding the 32-bit storage limit; for the purpose of this discussion, however, they are considered to be logically increasing without bounds.
\(^4\)The SYN and FIN TCP protocol flags are considered part of the sequence number space, occupying one byte before and after the data sent over the lifetime of the connection, respectively.
2.1.2 Acknowledgement and retransmission

TCP uses a cumulative acknowledgement scheme, where the receiver sends packets that contain an acknowledgement sequence number, either by themselves or along with data going in the opposite direction. This number indicates the highest sequence number received so far. Upon receiving an acknowledgement of a sequence number, the sender can discard all data with sequence numbers less than the number received. Until an acknowledgement is received, sent data is held in a retransmission queue, to be resent if not acknowledged within a timeout. The timeout is set from the measured round trip time of the connection, taking into account the variance of the measured values [Jac88].

Because the acknowledgement scheme is cumulative, loss of one packet means subsequent packets cannot be acknowledged until the lost packet is retransmitted. This can lead to excessive retransmission and unnecessary load on the network. To mitigate this, a TCP extension has been developed that allows the receiver to send selective acknowledgements of blocks of received data with sequence numbers that are not cumulative with the data acknowledged in the traditional way [RFC2018].

2.2 Congestion control

In the second half of the 1980’s, it became clear that the original TCP specification lacked a mechanism to limit transmission speed in the face of network congestion, leading to dramatically diminished throughput [Jac88]. This lead to the development of a better congestion control mechanism for TCP, which was added as a requirement for hosts connected to the internet [RFC1122].

The congestion control specification has since been updated several times [RFC2001; RFC2581; RFC5681]. The specification covers a set of general principles for slow start and congestion avoidance, as well as fast retransmit and fast recovery to recover from packet loss. The general principles outlined in [RFC5681] and its predecessors is named the Reno congestion control after the name of the BSD operating system it was first introduced in. A subsequent update, that changes the behaviour during fast retransmit slightly, is called New Reno [RFC6582]. Several other congestion control algorithms have been devised; this section describes the original Reno implementation (and its New Reno update), as well as the current default congestion control algorithm of the Linux kernel, CUBIC [HRX08]. A sketch of the phases of Reno congestion avoidance is presented in figure 1. The phases on the figure are explained in the following sections.

2.2.1 Slow start and congestion avoidance

When data transfer is initiated, TCP probes the network by slowly increasing the transmission rate until congestion is detected (which is the slow start part of congestion control). Congestion is signalled by lower levels of the network stack, usually in the form of lower available bandwidth to the sender. Once congestion is detected, the sender reduces the transmission rate and continues to transmit at a reduced rate until the network is no longer congested. The congestion control algorithm then enters a linear growth phase, where the window size grows linearly with time. If no further congestion is detected, the window size will continue to grow beyond the initial window size.

![Figure 1: Sketch of the phases of congestion avoidance of a sample TCP session using the default Reno TCP implementation. (1) Exponential growth of initial slow start phase. (2) Second slow start after first congestion detected. (3) Linear growth of congestion avoidance phase. (4) Fast retransmit, showing first a static window during the first duplicate acknowledgements, then window adjustment followed by linear growth during window inflation. (5) Deflation of the window after acknowledgement of the retransmitted packets, followed by congestion avoidance.](image-url)
of dropped packets\(^5\). The TCP specification mandates initially setting the congestion window to between two and four segments of data, depending on the segment size [RFC3390]. However, in the current Linux implementation a value of 10 segments is used, following a draft proposal to increase the window to improve latency for short-lived connections that never get out of the slow start phase [Chu+12; Linux, include/net/tcp.h].

During slow start, the congestion window is increased by one packet for each packet acknowledged. That is, each time a packet is acknowledged two new packets are sent into the network: one to replace the one just received, and one due to the increased window size. This results in slow start exponentially increasing the transmission rate, and is seen in phase 1 on figure 1. The slow start mechanism continues to operate until either the congestion window reaches the slow start threshold, or congestion is detected. When congestion is detected, the slow start threshold is reset to half the amount of data currently in flight to the receiver (i.e. sent, but not acknowledged), the congestion window is reset to a size of one segment, and slow start is restarted (phase 2 in figure 1). Initially, the slow start threshold is set arbitrarily high, meaning that in practice it is first set when congestion is detected.

When the congestion window size reaches the slow start threshold, the sender switches to congestion avoidance mode, which is employed to maintain the transmission rate. In this mode the congestion window is increased not exponentially, but linearly by one segment each round trip time (phase 3 on figure 1). This allows the transmit rate to converge to the available bandwidth, while also detecting changes in bandwidth during the transfer. timeouts during congestion avoidance mode is handled the same way as during slow start (i.e. lowering the slow start threshold and restarting slow start).

### 2.2.2 Fast retransmit and fast recovery

In some cases it is possible to detect packet loss before the retransmission timeout occurs. This is the case if a packet is lost, but subsequent packets make it through to the receiver. In this case, the sender receives duplicate identical acknowledgements from the receiver, corresponding to each packet successfully received. This means that packets are being dropped (which is a sign of congestion), but data is still leaving the network. Fast retransmit and fast recovery consists of using this information to recover from the loss instead of restarting transmission using the slow start method.

Fast retransmit works by retransmitting a lost packet as soon as the loss is detected by duplicate acknowledgements arriving, and fast recovery works by continuing to transmit new data on subsequent duplicate acknowledgements, until the retransmitted packet is received. Fast retransmit kicks in on the third duplicate\(^6\): the slow start threshold is adjusted in the same way as when congestion is detected during slow start; but instead of resetting the congestion window and going back to slow start, the window is reset to the same value as the slow start threshold, i.e. half the amount of data in flight when congestion was detected. Furthermore, the first unacknowledged packet is re-sent, and the congestion window is enlarged by one packet; the latter is called inflating the window and is done to make room in the window for the extra packets sent during this fast retransmit phase. For each subsequent duplicate acknowledgement, fast recovery is employed by further inflation of the window by one segment, and the sending of new data. This is shown in phase 4 on figure 1, where the first duplicate acknowledgments cause the window to stay unchanged (since no new data is acknowledged), followed by a reset of the window to half the previous size, and a linear growth during window inflation.

When a non-duplicate acknowledgement is received, the congestion window is reset back to the value it had before the fast retransmit was initiated (i.e. it is set to the value before it was inflated, hence this is called deflating the window), and the fast recovery mode is ended, going back to congestion avoidance mode (phase 5 on figure 1). In the original Reno implementation only a non-duplicate acknowledgement of all outstanding

\(^5\)Another possible way to signal congestion at the IP layer is by the use of explicit congestion notification as per [RFC3168].

\(^6\)The first two duplicate acknowledgements can trigger an independent limited transmit loss recovery [RFC3042].
data triggered the return to normal congestion avoidance behaviour. This meant that a partial acknowledgement of new data (indicating another lost packet) would keep the sender in fast retransmit until a timeout of the second lost packet occurred. The New Reno algorithm changes this behaviour to ‘do the right thing’ when receiving a partial acknowledgement, i.e. retransmit the next lost packet as indicated by the partial acknowledgement [RFC6582].

2.2.3 The CUBIC algorithm

The CUBIC algorithm [HRX08] is the default TCP congestion control implementation in the Linux kernel. It replaces the default TCP congestion control algorithm described above with a different algorithm based on a cubic function of the time since the last congestion event. The cubic function (illustrated in figure 2) has a rapidly increasing concave part for small time values, a plateau set at the previous maximal congestion window size, and finally a convex interval for increasing bandwidth above the maximum.

Instead of adjusting the congestion window as a function of previous values as each packet is acknowledged, the CUBIC algorithm re-computes the congestion window size at each step, using the cubic function calibrated so that the plateau is at the previous maximal congestion window size. This allows the congestion window to respond quickly to changes in available bandwidth; and because it is based on time rather than number of acknowledged packets, the window adjustment is independent of round-trip time, meaning that connections with higher round-trip times adjust as quickly as those with shorter.

The CUBIC implementation in the Linux kernel employs a different slow start algorithm than the one described above. This algorithm is called Hybrid Slow Start (or HyStart), and is designed to improve the slow start phase of TCP, especially for high bandwidth-delay products [HR11]. The problem that HyStart is trying to solve, is that the exponential increase of regular slow start tends to overshoot the actual available bandwidth, causing many packets to be dropped. This places unnecessary load on both end hosts and network, and can take a long time to recover from. The problem is most severe when the overshoot happens late in the slow start process, when the exponential curve is steeper, which happens when the bandwidth-delay product is large.

The basic function of the HyStart algorithm is the same as the regular slow start, but two different heuristics are applied to estimate the proper time to exit slow start mode (before packet loss occurs). Using these heuristics, TCP can transition to the congestion avoidance phase earlier, minimising the load on the network without significant performance loss.

One heuristic uses the combined round trip time of the initial burst of packets occurring at each round trip time of the slow start phase. This relies on the fact that at the beginning of a TCP session, an initial burst of packets is sent at once (corresponding to the initial congestion window). These packets travel through the network to the receiver, and are acknowledged in turn. By measuring the total delay between acknowledgement of the first and last packets of this burst, the time it took the packets to be transmitted can be estimated. This information can be used to estimate the total bandwidth of the transmission path, and by combining this with the measured minimum round-trip time of the connection, the total bandwidth-delay product of the connection can be estimated. This estimated value is used as a heuristic, in that slow start is
ended when the congestion window has increased beyond half the estimate, rather than when packet loss occurs.

Depending on the congestion conditions of the path, the first heuristic can have a varying accuracy. To improve accuracy, a second heuristic is employed. This heuristic measures delay over time and exits if the delay increases above a fixed threshold between consecutive round-trip times. To avoid errors in the estimation, delay is not measured on consecutive packets, but rather on consecutive packet bursts. This relies on the fact that TCP sends out a burst of packets, waits for acknowledgement, then sends out a second burst, and so on. The reasoning is that if the first packets of each burst are delayed, this delay will be the result of general path congestion, and congestion cause by the burst itself, and so using these initial packets in each burst, provides a more accurate estimate of the path delay characteristics.

2.3 TCP's interaction with bufferbloat

TCP relies on timely congestion notification to adjust its transmission rate to the available bandwidth. Bufferbloat means that packets are buffered instead of dropped, sometimes for lengthy periods of time. In the meantime, TCP continues sending packets, causing even further queue buildup at the bottlenecks. When packet drop then occurs, more packets are dropped than necessary, and the exponential back-off mechanism of TCP causes a sharp drop in transmission speed, freeing up bandwidth. This in turn causes TCP to increase its bandwidth again and, because of bufferbloat, no timely congestion notification reaches the sending host, so the bandwidth is once again increased to too high a level, causing another exponential back-off.

This behaviour can repeat indefinitely and cause throughput degradation as well as latency problems. When multiple TCP streams share the same buffer (e.g. because they go through the same bottleneck router), the back-off behaviour has a tendency to synchronise because the flows see packet drops at the same time (when the buffer fills). This causes all the flows to throttle back their transmission rates simultaneously, amplifying the effect. The synchronisation behaviour is known as TCP global synchronisation, and can be seen as a characteristic sawtooth pattern in bandwidth graphs.
3 Packet transmission and queueing in the Linux kernel

The Linux network stack is quite feature rich and devices running Linux are commonly employed at many different points in the network, from routers and access points to laptop computers and embedded devices. This section provides a high-level overview of the main components of the network stack with a focus on how IP packets generated by applications or forwarded from other hosts are transmitted and queued. The description of the network stack is based on [Ben06].

The aim of this survey of the network stack is to provide some context for how the queueing algorithms described in the next section interact with the rest of the stack when they are implemented in Linux. The aim of AQM algorithms is to control the queue, so it is important that the level they operate at (called the traffic control subsystem in Linux) actually is the place queuing occurs. Two relatively recent additions to the Linux kernel have been made with exactly this goal in mind: Byte Queue Limits [Her11; Cor11] and TCP Small Queues [Dum12] limit queueing at the level below and above the traffic control subsystem respectively. The workings of these additions are described in a separate subsection below.

3.1 The Linux IP network stack

Figure 3 provides an overview of the Linux IP stack. Incoming packets enter the stack from the device driver. The device driver has read in the packet from the hardware, decoded it and determined that it is an IP packet, and passes the decoded packet, stripped of its layer 2 header, to the IP stack. The IP stack first carries out a series of sanity and firewall policy checks, and subsequently passes it through the routing subsystem to determine whether the packet is destined for the local host or whether it should be forwarded. In the former case the packet is defragmented (if relevant) and passed to the relevant layer 4 protocol (typically TCP, UDP or ICMP) or application based on the contents of the packet. In the latter case, the packet is passed to the forwarding subsystem.

The forwarding subsystem checks system policies and packet header fields to make sure that the packet can actually be forwarded, before passing it on through the firewall to the outgoing transmission control subsystem. Packets generated by applications and layer 4 protocols on the local system are passed through the routing subsystem to determine their next hop destination, and are then passed on through the firewall to the traffic control subsystem. One of the recent enhancements to the Linux kernel, TCP Small Queues, operates at this level, limiting the amount of data TCP sends down the stack at once (see description in section 3.3.2).

---

7 The Linux network stack supports several other protocols apart from IP, but describing them is out of the scope of this section. The packets from these other protocols end up in the traffic control subsystem along with the IP packets.

8 The firewall subsystem (called netfilter) has a separate set of hooks at the forwarding and application/layer 4 input and output points, but these are omitted here to avoid cluttering the figure.

9 Layer 4 protocols can be implemented in user space by applications, and applications can send and receive raw IP packets. Both these two cases are omitted from figure 3, but for the purposes of this discussion the packets are handled the same: they go through the routing and everything below it on the figure in the same way.
The Traffic Control subsystem (described in more detail below in section 3.2) controls queueing of packets before they are passed to the device driver for transmission, and features various queueing algorithms and policies. The subsystem also has extensive filtering and policy capabilities, acting on and altering previous routing decisions as well as filtering based on packet contents. Traffic Control passes packets on to the device driver according to policies and capacity. The other recent enhancement, Byte Queue Limits (see section 3.3.1 below), works at this level, between Traffic Control and the device driver, allowing the kernel to control how many packets are passed on to the device driver by byte counts rather than packet count. Since packets can vary quite a bit in size, accounting by bytes allows more fine-grained control, limiting the amount of unmanaged queue occurring inside the device driver itself.

3.2 The Traffic Control subsystem

The Traffic Control subsystem [Bro06] provides extensive packet control capabilities to the Linux kernel. It consists of a kernel subsystem to handle the traffic and a user-space tool (the tc command) to configure it.

Configuration of the traffic control system is done per network interface. The basic configuration unit is the ‘queueing discipline’ or qdisc attached to an interface. Whenever the upper network layers of the kernel needs to send a packet to an interface, it is enqueued to the qdisc attached to the interface, and the qdisc is subsequently asked to dequeue packets into the network driver. Qdiscs can implement different queueing algorithms, or they can contain classes, which allows packets to be classified into a hierarchical tree structure of sub-qdiscs, through which packets travel, optionally controlled by filters configured at each node in the tree.

This structure allows for elaborate queueing control schemes, most of which are outside the scope of this section. The main feature of the traffic control subsystem relevant for this discussion is the qdiscs that implement the various queueing algorithms described in section 4.

3.3 Recent enhancements to the Linux kernel

As mentioned above, two recent additions to the Linux kernel have improved the ability of the Traffic Control subsystem to manage queues, by limiting queueing in other places of the stack. The TCP Small Queues (TSQ) patch works in the TCP subsystem, limiting the amount of packets that TCP sends down the stack at once, and Byte Queue Limits (BQL) works at the device driver level, limiting the amount of data passed on to the device driver at once. Both these enhancements, and their impact on queueing, are described below. The places they are operating at in the IP stack are marked on figure 3.

3.3.1 Byte Queue Limits

Byte Queue Limits (BQL) [Her11; Cor11] was introduced in version 3.3 of the Linux kernel. It introduces a layer above the device drivers themselves that keeps track of the number of bytes queued in the driver. This allows the queueing layer to limit the transmission of packets to the driver at the lowest possible level that does not impact transmission speed, keeping the main queueing of packets at the traffic control layer, where the queue can be managed to reduce latency. Before the BQL changes, the queue in the driver was managed in number of packets, which can vary in size. Tom Herbert, the author of BQL, puts it thus:

‘Hardware queueing limits are typically specified in terms of a number hardware descriptors, each of which has a variable size. The variability of the size of individual queued items can have a very wide range. For instance with the e1000 NIC the size could range from 64 bytes to 4K (with TSO enabled). This variability makes it next to impossible to choose a single queue limit that prevents starvation and provides lowest possible latency.’ [Her11]
BQL requires support in the device driver to do the accounting, so getting network drivers to add support is an ongoing effort. However, benchmark results for drivers that do support it show a marked improvement in latency under load, regardless of the queueing algorithm employed at the traffic control layer [Her11].

### 3.3.2 TCP Small Queues

TCP Small Queues (TSQ) [Dum12] has been introduced in version 3.6 of the Linux kernel. It is an enhancement to the TCP stack that changes the way TCP passes packets on to the IP layer and subsequent queues. Before the change, TCP passed packets down the stack as fast as possible, meaning that the packets ended up queued at the traffic control layer when they could not be sent immediately. TSQ introduces a limit that makes TCP stop passing packets to the lower layer when a configurable amount of data (by default set to 128KB) is in transit. The sending of subsequent packets is then deferred until the packets already passed down the stack have actually been sent. Eric Dumazet, the author of TSQ, states its purpose as follows:

> 'TSQ goal is to reduce number of TCP packets in xmit queues (qdisc & device queues), to reduce RTT and cwnd bias, part of the bufferbloat problem.' [Dum12]

Keeping the data in the TCP layer instead of dumping packets into the traffic control system allows TCP to react faster to congestion events and retransmissions, the number of packets sent, as seen from the TCP subsystem’s point of view, more closely resembles reality. Additionally, congestion is signalled directly to TCP, avoiding unnecessary packet drops. Benchmarks have shown a marked improvement in latency with the addition of TSQ regardless of queueing algorithm [Dum12].
4 Queue management algorithms

Active queue management plays an important role in minimising buffers and reducing latency. The Linux kernel contains implementations of a variety of queue management algorithms that can be installed into the traffic control subsystem. Many of them require tuning of their parameters to the bandwidth and latency conditions they are employed under, and if tuned incorrectly can severely impair network performance. This has led administrators to disable queue management for fear of breaking things, and has severely hampered deployment of queueing algorithms [GN11]. For this reason, the focus of this project is on parameter-less queueing algorithms. Specifically, four queueing algorithms are tested: the default unmanaged first-in-first-out (FIFO) queue, the Stochastic Fairness Queueing (SFQ) algorithm, the Controlled Delay (CoDel) algorithm and its fair queueing variant, fq_codel. These algorithms are described in the following sections.

The text below refers literature descriptions where available, and the Linux source code [Linux] for specific details not available elsewhere.

4.1 The default FIFO queueing algorithm (pfifo_fast)

The default queueing discipline used in the Linux kernel is a simple FIFO queue called pfifo_fast [Linux, net/sched/sch_generic.c]. The queue contains three priority bands so packets with higher priority (as set per the sending socket or IP TOS field) are dequeued before packets with lower demand. Other than that, packets are dequeued in the same order that they are enqueued, and the total queue length (the total for all three bands) is determined by the network interface transmit queue length (typically 1000 packets for Ethernet devices). If the queue is full, a packet is simply dropped on enqueue.

This behaviour is referred to as tail drop, as opposed to head drop where a packet is dropped from the front of the queue when the queue fills up. Using head drop makes it easier for TCP to notice the congestion, because the subsequently transmitted packets will trigger duplicate acknowledgements; whereas when using tail drop, all the packets already in the queue has to be sent before the packets triggering the duplicate acknowledgement arrives.

4.2 Stochastic Fairness Queueing (SFQ)

The SFQ algorithm implemented in the Linux kernel is an implementation of the algorithm described in [McK90]. The goal of fairness queueing is to give each distinct user\(^{10}\) of the network a fair share of available bandwidth, fair in this context taken to mean 'approximately equal'. The stochastic part of SFQ comes form the fact that instead of keeping track of all active flows and their share of the bandwidth, flows are hashed into a number of buckets, each of which has its own queue. These queues are served in a round-robin fashion when packets are dequeued. For Linux, this means that the source and destination address and ports are concatenated together with a random seed, and this value is hashed to find the queue the packet should be enqueued to.

This hashing scheme provides an approximation of fairness queueing as long as no hash collisions occur, and has the advantage that it is less computationally expensive, and takes up less memory keeping track of states than straight fairness queueing. To mitigate unfairness incurred by hash collisions, the number of buckets in the hash table can be adjusted (default Linux uses 1024 hash buckets, but limits the number of concurrently active flows to 128), and the hash function can optionally be perturbed at a configurable interval (by changing the seed; this is off by default) [Linux, net/sched/sch_sfq.c].

\(^{10}\)For some definition of user appropriate to the situation. The implementation in the Linux kernel provides fairness between individual flows, designated by their source and destination IP addresses, IP protocol number and source and destination (TCP or UDP) port number, or optionally a user configured filter setting [Linux, net/sched/sch_sfq.c].
The SFQ algorithm protects against single flows taking up all the available bandwidth, but otherwise does nothing to manage the queue size: the queues used for each hash bucket are a fixed size FIFO queues, with a default limit of 127 packets queued in total for all buckets. When the queue is full, packets are dropped from the flow with the longest active queue, giving a high probability of dropping packets from flows that use up a lot of bandwidth, rather than smaller flows. By default tail drop is employed, but the drop behaviour can optionally be switched to head drop.

4.3 Controlled Delay (CoDel)

CoDel is a relatively new queueing algorithm developed specifically to tackle part of the bufferbloat problem [NJ12]. It works by trying to distinguish between two different types of queue, termed good queue and bad queue. Good queue is the kind of queue that absorbs bursts in traffic to even out transmission speeds and keep the network interface busy. Bad queue is long-standing queue of a more permanent nature, that keeps buffers unnecessarily full and contributes to delay. The aim of CoDel is to get rid of the latter kind, while allowing the former to occur.

CoDel distinguishes between these two kinds of queue by looking at the minimum queue seen over an interval. If this queue does not drop below a certain threshold in the time that it takes flows to react to congestion signals (i.e. roughly one round-trip time), packets should be dropped to signal congestion and make flows back off. As the queue stays longer above the threshold, packets are dropped at an increasing rate, tuned to achieve a linear decrease in TCP sending rates.

As mentioned previously, older AQM algorithms requires tuning to the bandwidth and/or latency operating conditions they are employed in. CoDel sidesteps this issue by directly measuring the latency of the packets instead of using queue size as its threshold parameter. To achieve this, packets are marked with arrival time on enqueue, and on dequeue this time stamp is used to calculate the time the packet spent in the queue - the packet sojourn time. Because this quantity is directly measured from the individual packets, it is very easy to compute, and it does not depend on a bandwidth parameter. The only parameter that the CoDel algorithm keeps track of is the time the minimum sojourn time has been above or below the threshold value. Additionally, the only work done by the algorithm at enqueue time is adding the time stamp; all the rest is done at dequeue time. This means that no locking is required for the implementation.

Experimentation with the parameters have shown that a value of 5 ms for the minimum sojourn time, and a value of 100 ms for the delay before dropping packets work well for a long range of round trip times and bandwidths, making the algorithm virtually parameter-less for the most common connection conditions, at least at edge networks [NJ12].

4.3.1 The fairness queueing CoDel variant (fq_codel)

Apart from the basic implementation of the CoDel algorithm, the Linux kernel contains a hybrid of SFQ and CoDel, named simply fq_codel [Linux, net/sched/sch_fq_codel.c]. Since CoDel itself works on a single queue, it does not enforce any fairness between flows, and relies entirely on flows being well-behaved (i.e. turning down their transmission rates when packets are dropped) to control congestion. This is the reason the CoDel algorithm itself is only considered ‘part of’ the solution to bufferbloat by its authors.

fq_codel seeks to remedy this lack by combining the stochastic fairness operation of SFQ with the CoDel queue management algorithm. Conceptually, fq_codel is a variant of SFQ that uses CoDel to control the queues at each hash bucket, thereby combining the fairness benefits of SFQ with the queue management of CoDel. Apart from this, the fairness part of fq_codel differs from SFQ in several respects:

- fq_codel keeps track of the number of bytes in each queue rather than the number of packets. This allows it to better cooperate with the BQL mechanism described in section 3.3.1. A configurable quantum parameter designates how many bytes each flow is allowed to send per dequeue operation.
Unlike SFQ, *fq_codel* distinguishes between new and old flows. When a flow is first added to the hash table, it is designated as a new flow. Upon dequeueing, all new flows are searched for packets to dequeue before any of the old flows. Whenever a flow has sent more than its quantum of bytes, it is moved to the end of the old flows list. If the flow’s queue becomes empty before having transmitted its quantum of bytes, it is removed from the list, and a subsequent packet in the same flow will re-add the flow to the new flows list\(^\text{11}\).

This behaviour gives newly started flows priority and lets short-lived flows (such as for example DNS lookups) finish quickly, which can help interactivity. Additionally, flows that transfer a sufficiently small amount of data (relative to the other flows) that they are removed from the list of old flows, can be re-added as a new flow, allowing long-lived flows with sufficiently modest bandwidth requirements to be prioritised over bigger flows.

* fq_codel uses head drop instead of tail drop.

* fq_codel does not perturb the hash function periodically the way SFQ (optionally) does. Instead, it selects a random value to feed into the hash function along with the packet details upon qdisc initialisation. This in effect does the perturbation once when the qdisc is initialised.

\(^{11}\text{In fact this description is simplified a bit: When a new flow empties out, it is really put at the end of the old flow list, and only removed when it is re-encountered on that list and is still empty. However, this mechanism is somewhat counter-intuitive; and in practice, flows that transmit slowly enough to empty entirely before hitting the quantum limit will not have transmitted any packets when it is re-encountered on the old flows list, so the common behaviour corresponds to the simplified description.}\)
5 Experimental work

The goal of the experiments outlined in this section is to examine the effectiveness of the state of the art bufferbloat mitigation measures in the Linux kernel. This is done by running a set of tests that show the network behaviour under load and compare the behaviour of the different algorithms. More specifically, latency under load is tested with queueing handled by each of pfifo_fast, SFQ, CoDel andfq_codel enabled.

This section explains the test setup and testing tools used and outlines the different tests being run, including their purpose and expected results. The next section will go over the results and compare them with the expectations.

5.1 The test setup

The tests are run in a controlled testing environment consisting of three networked regular desktop computers (see figure 4). The three computers are networked together so that one of them acts as a router between the other two. The connection between the test server and the router is artificially limited to 10 mbit/sec (through use of the ethtool command to modify the Ethernet speed negotiation procedure). This is done to simulate the typical environment in a home network setup, where clients have a fast connection to the router, which is then connected through a slower link to the internet at large, causing buffer buildup at the router.

In a real-world internet scenario, the server accessed by a client would not normally be located right at the other end of the bottleneck. However, queueing occurs primarily at the bottleneck, so this difference should not make a substantial difference. Since the test server is right next to the bottleneck, queueing should occur at the traffic control level in the kernel, which gives direct feedback to the TCP stack. This is different from the client, where the bottleneck is at the router, and so the direct feedback to TCP is lessened by the extra hop. This could manifest itself in the TCP flows going from the server to the client adjusting themselves quicker to the available bandwidth than those going the other way.

Another significant difference between the test setup and a real-world internet usage scenario is the baseline latency between the hosts. In the test setup, baseline latency from client to server is less than one millisecond, whereas on the wider internet a baseline latency of 40-60 milliseconds is common. On the one hand this could make algorithms such as the CoDel algorithm that may assume a longer baseline latency behave sub-optimally. On the other hand, the low baseline latency makes it readily apparent when latency is induced by queueing, and not by some other accidental property of the path between test client and server.

For the tests, the test computers have been setup according to current bufferbloat community best practices [TG12]. This means that all hardware offload features are turned off, to allow all packet handling to happen in the kernel. Furthermore, the BQL limits have been set to a maximum of one packet (1514 bytes, including the Ethernet header). The purpose of both of these configuration options is to enable as much of the packet queueing to happen in the kernel, allowing the queueing disciplines maximum control of the queues.

5.2 The netperf-wrapper testing harness

The main benchmarking tool used for the performance tests is the Netperf benchmark [Jon12]. It supports various modes of sending TCP and UDP streams between a client and server and measuring the throughput and/or roundtrip time. However, diagnosing bufferbloat requires several such streams be run at the same
time (at minimum, a roundtrip time measurement while another stream loads up the link), and no tools existed for running such tests and aggregating the results into a single data set.

To remedy this lack, and to be able to conduct the experiments outlined in the following sections in a reproducible way, a testing harness for running these kinds of concurrent tests has been developed by the author. The tool, netperf-wrapper [Høl12], is implemented as a Python program that runs various benchmarking tools concurrently, aggregating the results. A test can be specified by means of a configuration file detailing which benchmark application(s) to run. The output of the benchmarking tools is then parsed and turned into aggregated time series data.

The aggregation relies on the benchmarking tools being able to produce time stamped output. The output for each benchmark command is parsed into a sequence of time stamped data points, and the various data series are aligned to the same time points using simple linear interpolation to the aligned time steps, from the closest time steps of each command output. The end result is a time series data set, with the data points for each benchmark command aligned to the same time values. To this can be added various computed result sets, such as sums and averages of other command outputs.

The aggregated data sets can be exported to various data formats for processing in other tools, or they can be output directly as graphs of the data. Each data set is saved for posterior use upon successful completion of a test run, and each test configuration can specify several different plot configurations for the same data point. This allows easy reproduction of tests, and comparison of results between testing sessions or software or configuration changes. The graphs in the results section are all produced using this tool.

Development of the netperf-wrapper tool has been guided by inputs from the bufferbloat online community [Bloat], and the tool is used for diagnosing bufferbloat in various contexts in the community12.

5.3 Test descriptions

To test the performance of the various queueing algorithms, three tests are conducted, described below. Each test is repeated for each of the four queueing algorithms: the default pfifo_fast, the CoDel algorithm, the SFQ algorithm, and the fq_codel queueing discipline. Since the object of the tests is to test the baseline default performance of the implementations in the current Linux kernel, any qdisc parameters are left at their defaults. The queueing disciplines are changed on all four network interfaces involved (two on the router, one on each of the other hosts), to create the best possible environment for the queueing algorithms to control queue delay.

The overall question these tests seek to answer is the following:

‘Which of the four queueing disciplines (pfifo_fast, codel, sfq or fq_codel) performs best (at their default settings) at mitigating increased latency under load, and how effective is this mitigation overall?’

5.3.1 The bidirectional TCP test

This test consists of running two concurrent TCP streams between the client and server (one in each direction), while simultaneously measuring ICMP ping latency. The ping measurements are started a few seconds before the TCP streams, to better show the change in latency as the TCP streams ramp up their bandwidth.

The expected behaviour with this test is a sharp increase in latency as the TCP streams start up. This should be most marked in the case of the default pfifo_fast qdisc, where queueing introduces a substantial delay. Using the codel qdisc should improve on this as the queue management makes the TCP flows back off and manage their transmission rate to avoid queueing. The sfq qdisc should improve latency by having the ping

12A special thanks to Dave Taht for testing and input in the development of the netperf-wrapper tool.
packets served intermittently with the TCP flows, instead of having to share the queue with them, but no additional queue management is employed to have the TCP flows adjust their rate. Since there is only three total flows, the separate queueing may in fact produce better result that then CoDel queue management on its own. Finally, fq_codel, combining separate queues with CoDel queue management, should give the best of both worlds: ping flows being served faster due to the separate queues, while at the same time all queues are kept at a reasonable level, ensuring lower latency overall.

5.3.2 The Real-time Response Under Load (RRUL) test

This test is the current prototype of one of the bufferbloat tests under development in the bufferbloat community [Tah12]. It is similar to the bidirectional TCP test, except that it consists of running several concurrent connections: four in each direction. This should better approximate a real-world busy network where several connections are active simultaneously. The various flows each use a different quality of service marking of the packets, which allows testing for behaviour of routers in the presence of such a marking. The RRUL test also incorporates a UDP packet round-trip measurement in addition to the ICMP ping measurement. This is done to compare the round-trip behaviour of the different protocols on the internet.

Since no quality of service differentiation is done in the test setup, and no difference in handling between protocols is present, some of the features of the RRUL test are less relevant for this testing setup. As such, the test is included for its other features, i.e. the multiple concurrent streams. The test is used unmodified to better compare the results with other uses of the RRUL test, and for the purposes of this test, the extra features can be simply ignored, except that the extra bandwidth consumed by the additional round-trip measurement streams are accounted for in the total throughput data.

The expected behaviour of the RRUL test is very similar to the bidirectional TCP test; however, the increased number of flows should render the multiple queue approach of sfq less efficient on its own, since there are more queues to serve on each round of packet dequeuing. This should emphasise the benefit of the combined approach of fq_codel.

5.3.3 UDP flood test

This test is simply comprised of one stream sending UDP packets as fast as they can go, along with a ping. Since there is no rate limiting employed for the UDP packets, algorithms that attempt to signal endpoints to slow down (such as CoDel) have no effect. The object of this test is to show exactly this; that prioritising traffic at the bottleneck (such as with fair queueing) is necessary to achieve reasonable latency in the face of ill-behaved flows (which the UDP flood is meant to simulate). As such, it is expected that this test will show straight CoDel provides latency behaviour roughly equivalent to the default pfifo_fast qdisc, while SFQ and fq_codel also exhibit behaviour similar to each other (since their fairness queueing implementations are quite similar).
6 Results

This section presents the test results for each of the three tests performed. The results are included for each test in turn, in the form of graphs of the bandwidth and latency during the tests, as generated by the netperf-wrapper testing harness. Each of the following subsections describes the results of the relevant test and compares them with the expected values, as outlined in section 5. Finally, a general summary of the test results is presented.

The raw data is available online at http://akira.ruc.dk/~tohojo/bufferbloat/.

6.1 Bidirectional TCP test

The results of the bidirectional TCP test are shown in figures 5 and 6 on the following page, which show the ping and bandwidth over time and the cumulative distribution functions of the ping values, respectively. Logarithmic scales are employed to show the large variance in values between the different tests. The total throughput values for each qdisc are shown in table 1.

<table>
<thead>
<tr>
<th>qdisc</th>
<th>Upload</th>
<th>Download</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfifo_fast</td>
<td>566.47</td>
<td>231.72</td>
<td>798.19</td>
</tr>
<tr>
<td>codel</td>
<td>530.95</td>
<td>526.07</td>
<td>1057.02</td>
</tr>
<tr>
<td>sfq</td>
<td>552.38</td>
<td>551.34</td>
<td>1103.72</td>
</tr>
<tr>
<td>fq_codel</td>
<td>546.25</td>
<td>543.71</td>
<td>1089.96</td>
</tr>
</tbody>
</table>

Figure 5 shows how the ping times sharply increase from the initial sub-millisecond times as soon as the TCP flows start (delayed five seconds). When using the default pfifo_fast qdisc, this increase is particularly marked, where ping times increase rapidly to around 100 milliseconds, and then continues to increase for the duration of the TCP streams, peaking at almost a second of latency. This is a clear sign of bufferbloat: the ping packets have to pass through a large queue in both directions, causing very large delays.

The throughput on the download stream (blue line) when using the pfifo_fast is also degraded. This is most likely due to TCP acknowledgments being delayed, and the TCP congestion control algorithm scaling back appropriately. That this happens only on the downstream link, corresponds well with the fact that the download stream is controlled by the server, which is right at the bottleneck, and so gets more direct feedback into the TCP stack, allowing TSQ to more effectively limit the sending rate. The TCP stack controlling the upload stream on the other hand, having an additional link before the bottleneck, gets less timely feedback, and so the bandwidth usage is more even, but still somewhat erratic. The total throughput, as seen in table 1, is slightly higher for the upload stream, but much lower for the download stream, leading to a low total throughput.

Employing the CoDel qdisc improves results significantly: ping times now only go up to between 20 and 30 milliseconds, and the bandwidth utilisation in both directions is stable around the link capacity of 10 mbps. This corresponds well with the expected results, namely that timely signalling to the TCP stacks to slow down their sending rates improves latency by avoiding bufferbloat. Throughput is affected slightly for the upload stream (corresponding to a 5.5% decrease), but the total throughput is improved by 32.5% compared to the base pfifo_fast case.

The results look even better for the sfq and fq_codel qdiscs. Here latency is stable just below 20 milliseconds, and throughput is completely stable, very close to the link capacity. The sfq qdisc exhibits a slight tendency to small spikes in traffic rates on the upload side. These double spikes correspond to the TCP CUBIC...
Figure 5: Bandwidth and ping plots for the bidirectional TCP test. The red lines are ping times (on the right axis), and the blue and green lines are download and upload (respectively), on the left axis.

Figure 6: CDF plot of ping time distributions for the bidirectional TCP test.
growth function when the transmission rate is lowered, and subsequently pushed back up to probe for more bandwidth. The small waves on the transmission rate correspond to the continuous adjustments, while the larger spikes are the times when multiple packets are dropped at once (which happen somewhat randomly due to timing variance in the system). The download stream does not exhibit these spikes because the TCP stack is directly at the bottleneck, and so get more timely congestion information than the upload stream does. The \texttt{fq\_codel} qdisc has a smaller tendency to drop multiple packets at once due to the ongoing TCP flow rate management of the CoDel algorithm, which is seen by the absence of the bandwidth spikes.

Throughput on the \texttt{sfq} qdisc is 2.5\% lower on upload, but 38.2\% higher in total. For \texttt{fq\_codel}, upload throughput is 3.6\% lower, and total throughput is 36.4\% higher, showing a clear overall advantage of employing either qdisc over the default. The 1\% difference between \texttt{sfq} and \texttt{fq\_codel} corresponds roughly to the throughput overhead of the CoDel algorithm cited in \cite{NJ12}.

The graph of the ping times shown in figure 6 makes it easier to directly compare the latency behaviour of the different qdiscs. The figure shows the cumulative distribution of the ping time values from the interval of the test where the TCP streams are active. The \texttt{pfifo\_fast} qdisc experiences severe latency degradation, while the others fare somewhat better. This corroborates the interpretation that managing queues using CoDel improves the latency tremendously, but fairness queueing is even better, with \texttt{fq\_codel} showing a slight, but insignificant edge over \texttt{sfq} behaviour.

In summary, the bidirectional TCP test shows results that correspond quite well to the expectations of the test setup. That is, the default \texttt{pfifo\_fast} qdisc shows severe latency degradation when the link is loaded. Queue management using CoDel improves on this, but fairness queueing has an even larger effect, improving both latency and total throughput. And finally, the latency difference between the \texttt{sfq} and \texttt{fq\_codel} qdiscs is insignificant with the small number of flows employed in this test, while \texttt{sfq} has a small (1\%) advantage in total throughput.

### 6.2 RRUL test

The results for the RRUL test are shown in figures 7 and 8 on the next page. As with the results for the bidirectional TCP test, a logarithmic scale is employed to better show the full range of values for the ping tests. However, since the \texttt{pfifo\_fast} values are so obviously worse than the rest, they are left off the CDF plot on figure 8 so better compare the results for the other qdiscs; this makes it possible to do away with the logarithmic scale. The total throughput for the RRUL test is shown in table 2. The overhead column indicates the total bandwidth consumed by the UDP round-trip measurement streams; since these work by counting roundtrips per second, the amount of bandwidth they consume is (unlike the ICMP ping tests) not insignificant, and furthermore varies as a function of the latency.

<table>
<thead>
<tr>
<th>qdisc</th>
<th>Upload</th>
<th>Download</th>
<th>Overhead</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{pfifo_fast}</td>
<td>565.65</td>
<td>437.55</td>
<td>0.19</td>
<td>1003.39</td>
</tr>
<tr>
<td>\texttt{codel}</td>
<td>530.80</td>
<td>530.09</td>
<td>0.11</td>
<td>1061.00</td>
</tr>
<tr>
<td>\texttt{sfq}</td>
<td>551.02</td>
<td>546.67</td>
<td>5.93</td>
<td>1103.62</td>
</tr>
<tr>
<td>\texttt{fq_codel}</td>
<td>535.63</td>
<td>534.83</td>
<td>9.89</td>
<td>1080.35</td>
</tr>
</tbody>
</table>

Similar to the bidirectional TCP test, the RRUL test shows the severely degraded performance of the default \texttt{pfifo\_fast} qdisc when the link is loaded. As with the previous test, throughput is degraded in the download direction and the latency suffers under load.
Figure 7: Bandwidth and ping plots for the RRUL test. For each test, the top plot shows download speed, the middle one shows upload speed and the bottom plot shows ping times. The bold black lines indicate average values for all streams, while the coloured lines are the individual stream values.

Figure 8: CDF plot of ping time distributions for the RRUL test. The \texttt{pfifo\_fast qdisc}, exhibiting ping times orders of magnitude higher than the other tests, is left out on this plot.
As with the bidirectional TCP test, employing CoDel queue management improves latency conditions significantly, and also stabilises the aggregate throughput (as can be seen on the graphs by the flat average lines). Upload throughput suffers about the same as for the bidirectional TCP test (6.2%), while the improvement of total throughput is somewhat lower (5.7%). The individual TCP flows are still showing signs of TCP synchronisation (as is seen in the sawtooth pattern); in the face of multiple flows, CoDel on its own is insufficient to completely mitigate the effects of bufferbloat, however the fairness between flows is improved compared to pfifo_fast.

The results for sfq and fq_codel are also quite similar to the previous results, including the bandwidth spikes of sfq and more stable behaviour of fq_codel in both aggregate and individual stream behaviour. This includes the upload streams showing a larger variance than the download streams due to the extra hop and related longer feedback time to the TCP stack. As with the CoDel qdisc, upload throughput is a bit lower (2.5% and 5.4% respectively), while total throughput is a bit higher (10.0% and 7.7% respectively), compared to the base pfifo_fast case. In this test, the throughput advantage of sfq over fq_codel is slightly higher (at 2.2%) than in the bidirectional TCP test.

However, compared to the bidirectional TCP test, the difference in latency of sfq and fq_codel is quite a bit larger, as is clearly seen in figure 8. With the increased number of streams of the RRUL test, the queue management approach of fq_codel (i.e. the combination of the old/new queue handling and the CoDel queue management) gives a significant advantage over that of sfq. This is shown in the significant (66%) increase in ping times from fq_codel to sfq, which corresponds well with the expected result; the relative advantage of fq_codel over straight sfq should continue to improve as the number of streams get bigger. While the throughput penalty is also a bit larger than in the bidirectional TCP test, the gains in latency behaviour clearly outweighs the loss in throughput.

Another difference from the bidirectional TCP is the initial spikes in aggregate bandwidth seen on all four tests just after the streams start up. This is most likely caused by TCP failing to react fast enough as the streams hit the bandwidth limit. All the employed qdiscs have quite high values for their maximum queue size, allowing the TCP stacks to fill up queues before the queue management algorithms have time to react, which causes these initial spikes in transmission rates.

Overall, the RRUL test results correspond well with the expected values, and shows that the combination of fairness queueing and active queue management, as implemented in fq_codel, provides a significant advantage in bufferbloat mitigation. The effect of fq_codel is, with the increased number of simultaneous streams in this test, significantly better than either fairness queueing or CoDel queue management in isolation. While sfq provides slightly better throughput than fq_codel (2.2%), it comes at a hefty cost in latency (66%).

6.3 UDP flood test

The results for the UDP flood test are shown in figures 9 and 10 on the following page. The graphs show what happens in the presence of a misbehaved stream (i.e. one without the bandwidth adjustment features of TCP). The UDP stream is configured to transmit at 100 mbps, and does not adjust itself according to dropped packets or any sort of feedback. The bandwidth on the graphs do not represent the number of bytes received by the server, but only what is sent by the client; for this reason, no aggregate bandwidth values are provided. The CoDel variants exhibit a slight variance in consumed bandwidth, due to its dropping behaviour (at the client, not the bottleneck). However, the stream is not significantly hampered by this, and continues to transmit at several times the available (bottleneck) bandwidth.

The effect this behaviour has on round-trip latency is, of course, the interesting part of this test. Latency suffers tremendously under both the pfifo_fast and codel qdiscs, codel providing only a slight improvement. In fact, for most of the test, pings are lost entirely due to an overloaded bottleneck link, and so data points are missing throughout (which also explains the blocky appearance of the CDF plot on figure 10). On
Figure 9: Bandwidth and ping plots for the UDP flood test. The red lines are ping times (on the right axis), and the blue lines are download speed (on the left axis).

Figure 10: CDF plot of ping time distributions for the UDP flood test.
the other hand, the fairness queueing qdiscs (sfq and fq_codel) show exactly equivalent latency behaviour, with ping times around 10 milliseconds. This corresponds well to the latency being around 20 milliseconds in the bidirectional TCP stream (which had two streams), and indicates a linear growth in ping times from fairness queueing as a function of the number of streams, which is to be expected from the round-robin dequeueing behaviour of the fairness queueing implementations.

Overall, the UDP flood test results correspond well with the expectations, showing the need for fairness queueing in the face of ill-behaved streams. It also shows that the queue managing behaviour of CoDel has no effect on streams that do not react to packet drops by lowering their transmission rate, and so the type of fairness queueing is not important, just the fact that fairness queueing provides service for competing streams in a way that prohibits ill-behaving streams from hogging all the available bandwidth. This is also very much in line with the expectations for the results of the test.

6.4 Summarising the results

All of the tests exhibit behaviour that conform quite well to the expectations. It is very clear that the default pfifo_fast qdisc exhibits severe latency degradation when the available bandwidth is utilised fully. On a 10 mbps link, response times under load can go from the sub-millisecond latencies of a local network to almost a full second. It can be further noted that this behaviour persists in the face of the general enhancements in the recent Linux kernels described in section 3.3.

However, the results also show that this behaviour is quite fixable. The alternative queueing disciplines all show lower latency under load. The CoDel algorithm can indeed manage TCP queues, but struggles to completely contain latency increases as the number of streams go up. Likewise, fairness queueing in itself improves the situation dramatically, and is essential especially in the face of ill-behaved streams that do not react to congestion indications.

Finally, combining the queue management behaviour of CoDel with fairness queueing, in the form of the fq_code1 qdisc, gives the largest overall improvement in latency behaviour under load. The fq_code1 qdisc achieves results that range from just as good as any of the other tested qdiscs (in the UDP flood test), to significantly better (in the RRUL multi-stream test). And compared to the default pfifo_fast qdisc, the results of fq_code1 (as well as that of the other tested alternative qdiscs) is orders of magnitude better.

The total throughput of fq_code1 is slightly lower than that of the sfq qdisc (which has the highest overall throughput of all the tested qdiscs), however this is made up for in latency gains: for the RRUL test (with many simultaneous streams), fq_code1 suffers 2.2% in throughput, while gaining 66% in latency. For the single stream bidirectional TCP test, the results are not so clear; here, the gain in latency of fq_code1 is negligible, but the loss in throughput persists, though it is reduced to 1%.

While all the tested alternative qdiscs provide a very significant improvement over the default pfifo_fast qdisc, the latency is still orders of magnitude worse under load than in the baseline unloaded case. However, this should probably be attributed to the extremely low latency of the baseline case. In the common case of usage on the wider internet, the base load latency is in most cases significantly higher than that of a local network, and so the relative latency under load compared to the unloaded case should be better. Or to put it another way, the added latency under load is constant as a function of the bandwidth and not related to the baseline latency, and adding 10-20 milliseconds of round-trip latency to a stream going out on the wider internet is hardly noticeable in the wider scheme of things.

Finally, it is noted that these results are achieved without any tuning of the algorithms to the test conditions. The only tuning of the test setup that has been done is done at the levels above and below the actual queueing disciplines, in order to allow the algorithms to exert maximal control. It is therefore reasonable to expect that tuning of the algorithms themselves might improve the results even further; especially when considering that the defaults in most cases are fairly high and so more suited to higher bandwidths. However, testing the algorithms tuned to various parameters is out of the scope of this project, and is left as further work.
7 Conclusions

This project examines the state of the art of the bufferbloat mitigation features in version 3.6 of the Linux kernel. This is done by conducting a range of experiments designed to exhibit the bufferbloat problem and the various mitigation features, in the form of different queueing disciplines, in the kernel. The tests are performed in a controlled setting where the test hosts are configured to maximise the queueing disciplines’ control of the packet queue.

On this setup three tests are conducted: A simple bidirectional TCP test, a Realtime Response Under Load (RRUL) test designed by the bufferbloat community, and an UDP flood test. Each test is repeated for each of the four tested queueing disciplines: the default pfifo_fast, the CoDel algorithm implementation (codel qdisc), the Stochastic Fairness Queueing (sfq) qdisc and finally the combination of fairness queueing and CoDel queue management as implemented in the fq_codel qdisc.

The test results correspond well with the expectations, and show that the default pfifo_fast qdisc suffers from decreased throughput and significantly increased latency under load, with all the other qdiscs providing orders of magnitude lower latency when the link is loaded. The results show that the CoDel algorithm can manage the queues of well-behaved TCP streams to some extent, providing somewhat lower latencies. However, ill-behaved streams, as tested in the UDP flood test, show the need for fairness queueing to distribute bandwidth among flows. The combination of fairness queueing with the CoDel algorithm, as implemented in the fq_codel qdisc, exhibits significantly lower latency under load than any of the other qdiscs when the number of concurrent bidirectional streams increase beyond one. Total throughput when using fq_codel is slightly lower than using sfq; however, for multiple flows, the loss of total throughput is relatively small (2.2%) compared to the gains in latency (66%).

Overall, the results of using the fq_codel qdisc are encouraging, showing a decrease in latency under load to less than 20 milliseconds, from almost a second when using the default pfifo_fast qdisc.
8 Further work

While the results presented in this report are encouraging and show that it is possible to mitigate the bufferbloat problem quite effectively already in the current version of the Linux kernel, much work remains to be done. This section outlines some of the areas that require further research.

The tests presented here show encouraging results, but are done in a fairly controlled setting. Much testing of the algorithms in real-world conditions is needed to better understand the behaviour of the algorithms in varying conditions. This also includes testing and tuning of the various parameters of the qdiscs in order to find the parameters that work best over the range of bandwidths and network conditions encountered in real-world usage (these conditions, of course, also need to be identified). Similarly, alternative variants of the \texttt{fq\_codel} qdisc have been proposed (called \texttt{efq\_codel} and \texttt{nffq\_codel}), the behaviour of which also needs more testing. Finally, the difference in throughput between the different qdiscs need further testing to be better understood.

Some of the enhancements in the Linux kernel presented in section 3.3 (i.e. BQL) are primarily oriented towards Ethernet and related technologies. Similar work needs to be done in relation to other network interface technologies (such as ADSL and cable modem connections), or alternatively a different approach needs to be employed that can make sure control is kept at the qdisc level. BQL itself is also an ongoing effort because of the required driver level support.

Wireless networks present an entirely separate problem, where much testing and queue management research is needed. Wireless technologies provide several different forms of packet batching and offloading that interfere with buffer management, but are required to attain reasonable throughput. Finding ways to combine these technologies with buffer management features is required to tackle latency issues over wireless networks.

The nature of the queue management algorithms require them to be present where the queues are, i.e. at the bottleneck links. This requires a massive deployment effort, which is hampered by consumer equipment being hard or impossible to update, and expensive to replace. Additional technologies to artificially limit the bandwidth to gain control of the queue exists (such as the \texttt{htb} qdisc in the Linux kernel), but deploying them in a way that does not require extensive and error-prone manual tuning is an ongoing effort.

All in all, though, while much work remains to be done, tackling the bufferbloat problem does appear to be tractable within the foreseeable future.


9 References


