



Università
Ca' Foscari
Venezia

Master's Degree programme
in Computer Science and
Information Technology

Final Thesis

Integrating CoDel and Age-Based Scheduling: a hybrid queueing approach

Supervisor

Prof. Andrea Marin

Supervisor

Prof. Leonardo Maccari

Graduand

Giovanni Moschini

879808

Academic Year

2024/2025

Contents

1. Introduction	3
2. Active Queue Management	4
2.1. Random Early Detection	4
2.2. Controlled Delay	4
2.3. Flow Queue CoDel	10
2.4. Common Applications Kept Enhanced	11
2.5. Proportional Integral controller Enhanced	13
3. Age-based Scheduling and Size-Based Scheduling	18
3.1. Introduction to Queues and Scheduling	18
3.2. Size dependent disciplines	18
3.3. Threshold selection	20
4. Queue Implementation	21
4.1. Traffic Control	21
4.2. Basic queues	21
4.3. Multi-level queues	22
4.4. Priority scheduling	22
5. Experimental Setup	25
5.1. Hardware setup	25
5.2. Traffic generation	26
5.3. Antler	27
5.4. Queue Configuration	27
5.5. Test process	29
6. Experimental Results	30
6.1. Mean Flow Completion Time	30
6.2. Flow Completion Time versus Flow Size	32
6.3. Lorenz Curve Gap	36
7. Conclusions	39
A. Antler Modifications	40
References	45

1. Introduction

Active Queue Management (AQM) algorithms, such as CoDel [1] and FQ-CoDel, have been widely adopted to address issues like bufferbloat [2, 3] and to maintain low queueing delays. These algorithms dynamically manage queue lengths and provide mechanisms for fair bandwidth sharing among competing flows, but typically do not perform advanced scheduling beyond fair queueing or diffserv.

In parallel, research has demonstrated that size-based scheduling disciplines can significantly reduce the mean flow completion time, especially for workloads with heavy-tailed flow size distributions. Algorithms such as Least Attained Service [4] and Two-Level Processor Sharing [5] are notable examples.

Despite the individual benefits of AQM and age-based scheduling, their combination has not been thoroughly analysed in the literature. This thesis addresses this gap by investigating how AQM techniques and size-based scheduling can be combined to leverage the strengths of both approaches. Since these are orthogonal features of queueing algorithms, their integration is both logical and potentially beneficial.

To evaluate this combination, custom queueing solutions were developed using standard Linux utilities and benchmarked against each other. The analysis focused on Flow Completion Time across different algorithms, comparing age-based versus non-age-based solutions, queues with and without AQM, and flow-queueing versus single-queue configurations.

Unlike previous work [5] that implemented custom queueing disciplines in the kernel, this study used the traffic control (`tc`) and nftables (`nft`) Linux commands to rapidly prototype hybrid age-based AQM disciplines on a stable Linux distribution. Multi-band queues were assembled using `tc-prio` in combination with `tc-codel`, `tc-fq_codel`, `tc-pie`, and `tc-fq_pie`, while `tc-cake` and `tc-pfifo_fast` provided native support for multiple priorities. Nftables was used to direct packets into the appropriate priority band based on the number of bytes already sent by a flow, effectively implementing two-level processor sharing queues (2LPS).

The Antler network testing tool was extended to generate traffic with arbitrary distributions, including Pareto-distributed flows, enabling the evaluation of these queueing configurations under realistic workload conditions.

The following chapters introduce the relevant AQM algorithms and their motivations (section 2), discuss the principles and benefits of age-based scheduling (section 3), and detail the implementation of hybrid 2LPS queues with AQM using stable Linux features (section 4). The experimental setup and methodology are described in section 5, followed by an analysis of the collected results in section 6. The thesis concludes with a summary of findings and potential directions for future work in section 7.

2. Active Queue Management

The need for Active Queue Management (AQM) has long been recognized, initially as a solution to flow synchronization and persistently full queues [6], and more recently as a response to the bufferbloat problem [2, 3].

Traditional drop-tail algorithms only drop packets when the buffer is completely full. This behaviour causes buffers at bottleneck nodes to remain persistently full, leading to increased congestion and delay. In contrast, AQM algorithms aim to prevent this situation by proactively dropping packets before the buffer overflows.

To analyse how different AQM solutions interact with age-based scheduling, it is important to first understand their underlying mechanisms. This chapter provides an overview of the most significant algorithms, with special attention to the ones implemented in the Linux kernel, explaining the principles behind them, how they function, and highlighting their similarities and differences.

2.1. Random Early Detection

Random Early Detection (RED) [7] was one of the first algorithms developed to combat bufferbloat and was strongly recommended by the first RFC on AQM [6]. While RED can be effective, its adoption has been limited by the difficulty of configuring it correctly and the significant performance issues that can arise from misconfiguration.

It uses predictive models based on the current level of buffered packets to decide when to drop. Specifically, RED computes the average queue size using a low-pass filter, typically implemented as an exponentially weighted moving average. This average is then compared to two thresholds: a minimum and a maximum. If the average queue size is below the minimum threshold, packets are never dropped. If it is above the maximum, packets are always dropped. For values in between, packets are dropped with a probability that increases as the average queue size approaches the maximum threshold. This probabilistic dropping helps to signal congestion to senders before the buffer becomes completely full, reducing the likelihood of persistent queue build-up and improving overall network performance.

2.2. Controlled Delay

The Controlled Delay (CoDel) [1, 8] AQM strategy is a modern algorithm designed to provide a parameterless solution that can be easily deployed on typical Internet nodes. Its design goals include distinguishing between "good" and "bad" queues to treat them differently, controlling delay while allowing necessary traffic bursts, remaining insensitive to round-trip delays, link rates, and traffic loads, adapting to dynamically changing link rates, and enabling efficient implementations.

CoDel's operation is built around three main components: an estimator, a set-point, and a control loop.

Unlike earlier AQM solutions that use queue length as a congestion signal, CoDel measures congestion using packet sojourn time: the amount of time each packet spends in the queue from enqueue to dequeue. More specifically, CoDel tracks the minimum packet sojourn time over a configurable interval, chosen to be long enough for a "good" queue to drain. As buffers are meant to absorb and smooth out transient traffic spikes, it is acceptable for a flow to build up a queue temporarily to maintain high utilization. However, a queue that persists for longer than the network round-trip time of the flow suggests that an unnecessary amount of packets is being buffered, and congestion should be signalled.

A key question is when to consider a queue congested: this is the role of the set-point. Dropping packets as soon as the minimum sojourn time becomes non-zero would result in excessive drops and poor utilization. On the other hand, being too lenient would maximize utilization but fail to reduce delay, making the queue behave like a simple FIFO and not addressing bufferbloat. Therefore, it is crucial to choose an appropriate delay target to maximize utilization while minimizing delay. A useful metric to analyse this is "*power*", defined as the ratio between throughput and delay, as developed by Kleinrock [9].

The average goodput of a Reno TCP flow, given round-trip time r and target delay f expressed as fraction of r , can be expressed as:

$$goodput = r \frac{3 + 6f - f^2}{4(1 + f)} \quad (1)$$

Since the peak queue delay is just $f \cdot r$, *power* is simply a function of f , as the r in the numerator and denominator cancel out:

$$power = \frac{goodput}{fr} \propto \frac{1 + 2f - \frac{1}{3}f^2}{(1 + f)^2} \quad (2)$$

As Kleinrock observed, the best operating point is the peak *power* point. The second plot of Figure 1 shows that the *power* vs. f curve from Equation 2 is monotone decreasing, but remains very flat for $f < 0.1$. Since Equation 1, as shown in the first plot of Figure 1, is monotone increasing with f , the best operating point is near the right edge of the flat region. However, as the r in the model is a conservative upper bound, choosing a target of $0.05r$ is safer for shorter RTT connections while still providing a good utilization vs. delay trade-off.

For typical Internet links then, the recommended parameters are an interval of 100 ms and a target of 5 ms, as most Internet RTTs fall between 20 and 200 ms.

When congestion is detected, CoDel uses a control loop implemented as a state machine with a nominal state and a drop state. If the minimum delay stays above the target for more than an interval, CoDel enters the drop state, which is exited only when a packet with less than the target delay is dequeued. In the drop state, packets are dropped from the head of the queue to quickly signal congestion to the sender. The first drop occurs immediately upon entering the drop state, and subsequent drops are scheduled at decreasing intervals, with each interval divided by the square root of the number of packets dropped since entering the current phase. This increasing drop rate ensures that delay is eventually reduced without harming utilization through excessive early drops.

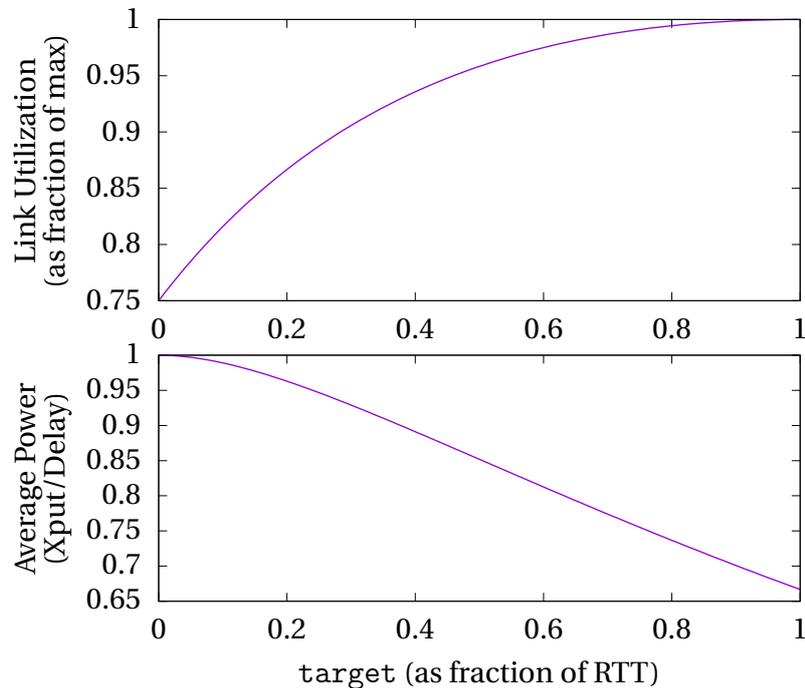


Figure 1: Relative goodput and average *Power* of a TCP Reno flow, as function of target expressed as fraction of round-trip time.

A major reason for CoDel’s effectiveness is its simplicity. The only state it requires, shown in Listing 1, is one timestamp to determine whether the minimum delay has been below the target within the interval, `first_above_time`, and four more variables to handle the drop control loop. Had CoDel chosen to use another metric, like mean or maximum delay, it would have required storing all observed values within the `interval`. As parameters, shown in Listing 2, CoDel needs an `interval`, which should be in the order of magnitude of the RTT of a typical flow, a `target`, usually set to 5% of `interval`, and the MTU of the link.

CoDel also requires storing a timestamp for each packet at enqueue time, so the sojourn time can be calculated when dequeuing. At enqueue, shown in Listing 3, the algorithm simply timestamps the packet and adds it to the underlying FIFO queue.

When a packet is retrieved from the underlying FIFO queue with the `dodequeue` function of Listing 4, the packet sojourn time of the packet is computed and used to detect congestion. The output flag `ok_to_drop` will be set if all the packets that have been dequeued in the past `interval` had a sojourn time above target.

The dequeue function of Listing 5 contains the state machine logic for the drop control loop. It branches depending on which state CoDel currently is. When in the drop state, it will first check whether no more congestion is detected and the drop state should be left, otherwise it

```

1 typedef struct {
2     queue_t queue;           \* underlying FIFO queue *\
3     time_t first_above_time = 0;
4     time_t drop_next = 0;
5     uint32_t count = 0;
6     uint32_t lastcount = 0;
7     flag_t dropping = false;
8 } codel_queue_t;

```

Listing 1: linuxkernel.]State necessary for each CoDel instance. Pseudocode based on the CoDel RFC [8] and the Linux kernel [10, Version 6.15, include/net/codel.h].

```

1 time_t TARGET = MS2TIME(5);
2 time_t INTERVAL = MS2TIME(100);
3 u_int MAXPACKET = 1500;

```

Listing 2: linuxkernel.]CoDel parameters. Pseudocode based on the CoDel RFC [8] and the Linux kernel [10, Version 6.15, include/net/codel.h].

will drop as many packets as the control law requires, and finally it will return the dequeued packet. When not in the drop state, it will enter it if congestion is detected, drop a packet and schedule the next drop, then return the next packet.

After each packet is dropped, the next drop time gets scheduled according to the control law, shown in Listing 6. The drop interval is first equal to `interval`, and gets shorter every time a packet is dropped without leaving the drop state, based the square root of the amount of dropped packets.

As CoDel adapts well to multiple queue systems, the CoDel RFC [8] recommends using a multiple-queue approach like FQ-CoDel or CAKE, instead of CoDel alone.

```

1 void codel_enqueue(codel_queue_t *q, packet_t* pkt)
2 {
3     pkt->tstamp = clock();
4     enqueue(&q->queue, pkt);
5 }

```

Listing 3: linuxkernel.]CoDel enqueue function. Pseudocode based on the CoDel RFC [8] and the Linux kernel [10, Version 6.15, net/sched/sch_codel.c].

```

1 typedef struct {
2     packet_t* p;
3     flag_t    ok_to_drop;
4 } dodequeue_result;
5
6 dodequeue_result codel_dodequeue(codel_queue_t *q, time_t now)
7 {
8     dodequeue_result r = { dequeue(&q->queue), false };
9     if (r.p == NULL) {
10         q->first_above_time = 0;
11         return r;
12     }
13
14     time_t sojourn_time = now - r.p->tstamp;
15     if (sojourn_time < TARGET || bytes() <= MAXPACKET) {
16         q->first_above_time = 0;
17     } else {
18         if (q->first_above_time == 0) {
19             q->first_above_time = now + INTERVAL;
20         } else if (now >= q->first_above_time) {
21             r.ok_to_drop = true;
22         }
23     }
24     return r;
25 }

```

Listing 4: linuxkernel.]CoDel dodequeue helper function. Pseudocode based on the CoDel RFC [8] and the Linux kernel [10, Version 6.15, net/sched/sch_codel.c].

```

1 packet_t* codel_dequeue(codel_queue_t *q)
2 {
3     time_t now = clock();
4     dodequeue_result r = codel_dodequeue(q, now);
5     uint32_t delta;
6
7     if (q->dropping) {
8         if (!r.ok_to_drop) {
9             q->dropping = false;
10        }
11        while (now >= q->drop_next && q->dropping) {
12            drop(r.p);
13            ++q->count;
14            r = codel_dodequeue(q, now);
15            if (! r.ok_to_drop) {
16                q->dropping = false;
17            } else {
18                q->drop_next = control_law(q->drop_next, q->count);
19            }
20        }
21    } else if (r.ok_to_drop) {
22        drop(r.p);
23        r = codel_dodequeue(q, now);
24        q->dropping = true;
25
26        delta = q->count - q->lastcount;
27        q->count = 1;
28        if ((delta > 1) && (now - q->drop_next < 16*INTERVAL))
29            q->count = delta;
30
31        q->drop_next = control_law(now, q->count);
32        q->lastcount = q->count;
33    }
34    return r.p;
35 }

```

Listing 5: linuxkernel.]q dequeue function. Pseudocode based on the CoDel RFC [8] and the Linux kernel [10, Version 6.15, include/net/codel_impl.c].

```

1 time_t control_law(time_t t, uint32_t count)
2 {
3     return t + INTERVAL / sqrt(count);
4 }

```

Listing 6: linuxkernel.]CoDel control law helper function. Pseudocode based on the CoDel RFC [8] and the Linux kernel [10, Version 6.15, include/net/codel_impl.c].

2.3. Flow Queue CoDel

Flow Queue CoDel (FQ-CoDel) [11] is the default queueing discipline in systemd based Linux distributions and is included in many router operating systems [12].

FQ-CoDel implements byte-based fairness between flows by maintaining a separate CoDel instance for each flow. Flows are identified using a hash table keyed by the 5-tuple: source and destination IP addresses, source and destination ports, and protocol. Each entry in the hash table contains an independent CoDel instance and the state required for deficit round-robin (DRR) scheduling. In addition to the hash table, FQ-CoDel keeps two lists of active flows called `new_flows` and `old_flows`, as shown in Listing 7.

```

1 typedef struct {
2     struct fq_codel_flow *flows;      /* flows table [FLOWS_CNT] */
3     struct list_head    new_flows;  /* list of new flows */
4     struct list_head    old_flows;  /* list of old flows */
5 } fq_codel_queue_t;
6
7 struct fq_codel_flow {
8     struct list_head flowchain;
9     int          deficit;
10    codel_queue_t queue;
11 };

```

Listing 7: linuxkernel]FQ-CoDel state variables. Pseudocode based on the Linux kernel [10, Version 6.15, net/sched/sch_fq_codel.c]

An FQ-CoDel instance takes as parameters the size of the hash-table `flows_cnt`, the DRR quantum, and the maximum queue limit `drop_overlimit`, shown in Listing 8, as well as the base CoDel parameters to initialize each flow queue.

At dequeue time, shown in Listing 9, these lists are polled in order, with `new_flows` given strict priority. When a packet is dequeued from a flow, the CoDel algorithm is executed, and the packet size is subtracted from the flow's deficit. If the deficit is exhausted, the flow is moved to the end of `old_flows` and its deficit is reset to the quantum parameter. If the buffer of a CoDel instance becomes empty, the flow is removed from the active list.

```

1 u_int FLOWS_CNT;          /* number of flows */
2 u_int QUANTUM;
3 u_int DROP_OVERLIMIT;

```

Listing 8: linuxkernel\FQ-CoDel parameters. Pseudocode based on the Linux kernel [10, Version 6.15, net/sched/sch_fq_codel.c]

This DRR scheduling ensures that all flows can transmit a quantum of bytes before any flow sends more, approximating byte-based fairness efficiently. The use of two active flow lists helps new flows start quickly and improves fairness for sparse flows. By always moving flows from `new_flows` to `old_flows` after use, even if empty, FQ-CoDel prevents a flow that sends individual packets with a high throughput from using more than its fair share of bandwidth.

At enqueue time, shown in Listing 10, the appropriate flow is determined by hashing the packet header fields. The packet is then enqueued in the corresponding CoDel instance; if the instance was previously empty, it is added to the end of `new_flows`. If the global packet limit is reached, the CoDel instance with the most packets is found by scanning `old_flows`, and half of its packets are dropped. A diagram of the FQ-CoDel flow state transitions is shown in Figure 2.

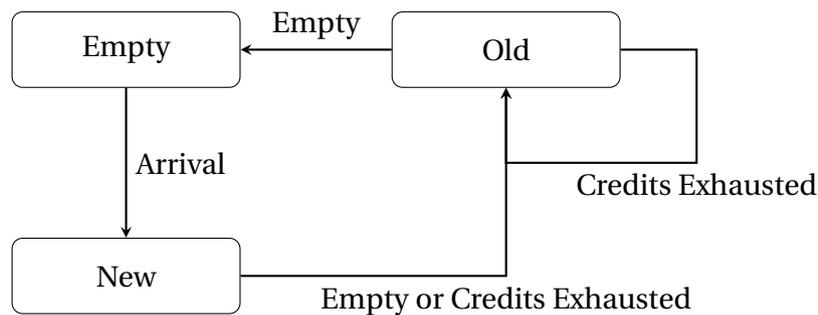


Figure 2: FQ-CoDel flow state diagram. [11]

Isolating flows in separate CoDel instances allows FQ-CoDel to penalize only the flows actually causing congestion, without dropping packets from sparse or low-volume flows. This parallelism is possible because CoDel uses packet sojourn time as its congestion metric, which works well with the variable throughput of a fair scheduler as flows start and stop.

2.4. Common Applications Kept Enhanced

Common Applications Kept Enhanced (CAKE) was developed as a successor to FQ-CoDel, building on its design while introducing several enhancements and new features.

One of the key improvements is the introduction of 8-way set associativity, which effectively eliminates hash collisions even with a very large number of flows. This significantly improves

```

1 packet_t* fq_codel_dequeue(fq_codel_queue_t *q)
2 {
3     struct fq_codel_flow *flow;
4     struct list_head *head;
5     packet_t *pkt;
6
7     begin:
8     head = &q->new_flows;
9     if (list_empty(head)) {
10        head = &q->old_flows;
11        if (list_empty(head))
12            return NULL;
13    }
14
15    flow = list_first_entry(head);
16
17    if (flow->deficit <= 0) {
18        flow->deficit += QUANTUM;
19        list_move_tail(&flow->flowchain, &q->old_flows);
20        goto begin;
21    }
22
23    pkt = codel_dequeue(&flow->queue);
24
25    if (!pkt) { /* queue is empty */
26        /* force a pass through old_flows to prevent starvation */
27        if ((head == &q->new_flows) && !list_empty(&q->old_flows))
28            list_move_tail(&flow->flowchain, &q->old_flows);
29        else
30            list_del_init(&flow->flowchain);
31        goto begin;
32    }
33
34    flow->deficit -= packet_size(pkt);
35    return pkt;
36 }

```

Listing 9: linuxkernel]FQ-CoDel dequeue function. Pseudocode based on the Linux kernel [10, Version 6.15, net/sched/sch_fq_codel.c]

```

1 void fqcodel_enqueue(fqcodel_queue_t *q, packet_t* pkt)
2 {
3     u_int idx;
4     struct fq_codell_flow *flow;
5
6     idx = fqcodel_classify(q, pkt);
7     flow = q->flows[idx];
8
9     codell_enqueue(flow->queue, pkt);
10
11     if (list_empty(&flow->flowchain)) {
12         list_add_tail(&flow->flowchain, q->new_flows);
13         flow->deficit = QUANTUM;
14     }
15
16     if (fqcodell_memory_usage(q) > DROP_OVERLIMIT)
17         fqcodell_drop_overlimit(q);
18 }

```

Listing 10: linuxkernel\FQ-CoDel enqueue function. Pseudocode based on the Linux kernel [10, Version 6.15, net/sched/sch_fq_codell.c]

flow isolation, though it introduces a small amount of additional delay due to the increased complexity. CAKE also includes an optional integrated traffic shaper, providing better performance compared to using an external shaper.

Another important feature is enhanced DiffServ support, allowing the configuration of multiple priority tiers—typically three or four—and scheduling flows based on their DiffServ markings. In this work, we leverage this feature to implement age-based scheduling, as described in subsection 4.3.

Additional improvements include a refined CoDel implementation, a more user-friendly command line interface, improved ECN support, and framing compensation. Further details can be found on the Bufferbloat Wiki [13].

2.5. Proportional Integral controller Enhanced

Proportional Integral controller Enhanced (PIE) is another AQM solution designed to address the bufferbloat problem [14, 15]. Its goal is to achieve results and ease of deployment similar to CoDel, while retaining the ease of implementation and scalability found in RED.

Like RED, PIE separates its logic into two parts: one algorithm computes the statistics needed to determine the drop probability `drop_prob`, and another decides when to actually drop packets. However, PIE adopts packet sojourn time as its congestion metric, following CoDel,

rather than the queue length used by RED.

```
1 typedef struct {  
2     queue_t queue;          \* underlying FIFO queue *\br/>3     double drop_prob;  
4     time_t burst_allowance;  
5     time_t qdelay_old;  
6 } pie_queue_t;
```

Listing 11: linuxkernel.]PIE state variables. Pseudocode based on the PIE RFC [15] and the Linux kernel [10, Version 6.15, include/net/pie.h].

PIE performs its packet dropping decisions during the enqueue operation, as illustrated in Listing 12, which is the same approach RED uses. When a packet arrives, PIE checks if `drop_prob` exceeds a certain threshold. If so, the packet is randomly dropped (or an ECN mark is sent) according to this probability. Alternatively, some implementations may avoid randomness by accumulating the drop probability and dropping a packet whenever the accumulator exceeds one. If the packet is not dropped, it is appended to the underlying FIFO queue.

At dequeue time, shown in Listing 13, PIE simply returns packets from the head of the FIFO queue, while updating the internal statistics that influence the drop probability.

The drop probability `drop_prob` is computed regularly, with a period of `T_UPDATE`, using the function shown in Listing 14. It is updated using both the distance between queue delay and delay target `QDELAY_REF` and the first derivative of the delay, linearly combined using two parameters, `ALPHA` and `BETA`, as shown in Equation 3. This delta is then harshly shrunk when the starting drop probability is very small, to prevent it from spiking too suddenly. This approach allows PIE to react more quickly when delay starts increasing, and to reduce the drop rate when delay is decreasing. Queue delay can be measured using timestamps (the approach shown in the pseudocode), as in CoDel, or estimated using Little's Law with the measured service rate and queue size.

$$p_{drop} = p_{drop} + \alpha(\text{delay} - \text{target}) + \beta(\text{delay} - \text{old_delay}) \quad (3)$$

As with CoDel, PIE is not typically deployed on its own. Instead, it is intended to be combined with a flow isolation mechanism, such as in FQ-PIE [16], or with other approximate fairness solutions.

```

1 void pie_enqueue(pie_queue_t *q, packet_t *pkt) {
2     if (q->drop_prob == 0 && q->qdelay < QDELAY_REF/2
3         && q->qdelay_old < QDELAY_REF/2) {
4         q->burst_allowance = MAX_BURST;
5     }
6     if (q->burst_allowance == 0 && pie_drop_early(q)) {
7         drop(pkt);
8     } else {
9         pkt->tstamp = clock();
10        enqueue(&q->queue_, pkt);
11    }
12 }
13
14 bool pie_drop_early(pie_queue_t *q) {
15     //Safeguard q to be work conserving
16     if ((q->qdelay_old < QDELAY_REF/2 && q->drop_prob < 0.2)
17         || (q->queue_.byte_length() <= 2 * MEAN_PKTSIZE) ) {
18         return false;
19     }
20
21     return random() < q->drop_prob;
22 }

```

Listing 12: linuxkernel.]PIE enqueue function. Pseudocode based on the PIE RFC [15] and the Linux kernel [10, Version 6.15, net/sched/sch_pie.c].

```
1 packet_t* pie_dequeue(pie_queue_t *q) {
2     packet_t *pkt = dequeue(&q->queue_);
3     q->qdelay = clock() - pkt->tstamp;
4     return pkt;
5 }
```

Listing 13: linuxkernel.]PIE dequeue function. Pseudocode based on the PIE RFC [15] and the Linux kernel [10, Version 6.15, net/sched/sch_pie.c].

```

1 void pie_calculate_drop_prob(pie_queue_t *q) {
2     time_t p = ALPHA * (q->qdelay - QDELAY_REF) + \
3         BETA * (q->qdelay - q->qdelay_old);
4
5     if (q->drop_prob < 0.000001) {
6         p /= 2048;
7     } else if (q->drop_prob < 0.00001) {
8         p /= 512;
9     } else if (q->drop_prob < 0.0001) {
10        p /= 128;
11    } else if (q->drop_prob < 0.001) {
12        p /= 32;
13    } else if (q->drop_prob < 0.01) {
14        p /= 8;
15    } else if (q->drop_prob < 0.1) {
16        p /= 2;
17    } else {
18        p = p;
19    }
20
21    q->drop_prob += p;
22
23    // Exponentially decay drop prob when congestion goes away
24    if (q->qdelay == 0 && q->qdelay_old == 0) {
25        q->drop_prob *= 0.98;
26    }
27
28    // Bound drop probability
29    if (q->drop_prob < 0)
30        q->drop_prob = 0.0
31    if (q->drop_prob > 1)
32        q->drop_prob = 1.0
33
34    q->qdelay_old = q->qdelay;
35
36    q->burst_allowance =
37        max(0, q->burst_allowance - T_UPDATE);
38 }

```

Listing 14: linuxkernel.]PIE drop probability calculator function. Pseudocode based on the PIE RFC [15] and the Linux kernel [10, Version 6.15, net/sched/sch_pie.c].

3. Age-based Scheduling and Size-Based Scheduling

The way packets and flows are scheduled has a significant impact on both latency and fairness. Traditional queueing disciplines like FIFO treat all flows equally, but can perform poorly when flow sizes are highly variable. By leveraging information about the age or size of flows, it is possible to design scheduling algorithms that significantly improve flow completion times, especially under heavy or bursty loads.

This section introduces the theoretical foundations of age-based and size-based scheduling, explains their relevance to TCP traffic, and discusses practical approaches for implementing these disciplines in real systems.

3.1. Introduction to Queues and Scheduling

The behaviour of TCP flows over an interface configured to schedule packets according to a FIFO discipline can be modelled as an $M/G/1/PS$ queue of flows in queueing theory. This stands for Markovian arrivals, General service time, 1 server, Processor Sharing.

A Markovian (or Poisson) arrival process is characterized by inter-arrival times of flows that are independent identically distributed exponential random variables. This is applicable to TCP flows in many network scenarios, as the initiation of new TCP connections by users or applications often occurs independently and at random times. When aggregated across many users and services, the superposition of these independent connection arrivals closely approximates a Poisson process, making it a suitable and widely used model for analysing TCP flow-level dynamics in queueing theory [17].

The reason why TCP flows experience a Processor Sharing discipline is two-fold. First, the packets from each flow tend to arrive in the queue over time, mixing them and resulting in multiple flows receiving service in the same interval of time. Second, when multiple TCP flows compete for bandwidth they eventually converge to a fair sharing of bandwidth. However, this assumes that all flows are always active and ignores burstiness or idle periods, which can occur in real traffic.

While the $M/G/1/PS$ model provides a useful baseline, it does not exploit information about flow sizes or ages, which can be leveraged for better performance.

3.2. Size dependent disciplines

When additional information about job sizes is available, either exactly or as a probability distribution, it becomes possible to take effective scheduling decisions. This is especially important under heavy load and with highly skewed or long-tailed job size distributions, where FIFO queues tend to have much worse mean response times compared to age-based schedulers.

The following sections review key size-based and age-based scheduling disciplines, their

theoretical properties, and practical considerations for their implementation.

3.2.1. Shortest Remaining Processing Time

Shortest Remaining Processing Time (SRPT) is the optimal scheduling discipline for minimizing average job completion time [18]. SRPT is a pre-emptive, work-conserving discipline. It always serves the job with the least remaining work, based on its total size and the amount of service it has already received.

Despite its optimality, SRPT is rarely used in practice because it requires knowing the exact size of each job in advance, which is often not possible, such as when scheduling network packets. Additionally, SRPT can cause starvation for large jobs under heavy load.

If we relax the requirement of knowing exact job sizes and instead assume we know the statistical distribution of job sizes, we can use a discipline called Shortest Expected Remaining Processing Time (SERPT). SERPT prioritizes jobs based on the expected remaining work, given the service they have already received. This is equivalent to prioritizing jobs with the highest hazard rate.

3.2.2. Least Attained Service

For long-tailed job size distributions, which have a monotonically decreasing hazard rate, an age-based scheduling approach is equivalent to SERPT. The most straightforward implementation of this idea is the Least Attained Service (LAS) discipline [4]. LAS works by always prioritizing jobs that have received the least amount of service so far, using Processor Sharing among jobs with equal attained service.

The main drawback of LAS is its implementation complexity. It requires maintaining a separate FIFO queue for every flow, as well as a priority queue to schedule flows according to their age or attained service.

To address this, a common practical approximation is Multi-Level Processor Sharing (MLPS). MLPS uses a fixed number of Processor Sharing queues, each corresponding to a range of attained service (or "age"). These queues are served in strict priority order: jobs start in the highest-priority queue and move to lower-priority queues as they accumulate more service. This approach reduces complexity while still capturing most of the benefits of LAS.

3.2.3. Two Level Processor Sharing

Two Level Processor Sharing (2LPS or PS+PS) is a special case of MLPS that uses only two levels of priority. It is parametrized using a service threshold a . Jobs that have received service less than or equal to a are placed in the high-priority queue and share service among themselves. Once a job has received more service than a , it is moved to the low-priority queue and is only served when there are no high-priority jobs left. This means that jobs

smaller than a always receive high-priority service, while larger jobs start with high priority and are demoted once they exceed the threshold.

Adding more priority levels can make the approximation closer to the ideal LAS discipline, but with diminishing returns and increased implementation complexity. With just two levels and a well-chosen threshold a , most of the performance benefits of LAS can be achieved at minimal cost [5].

To implement 2LPS for TCP flows, it is sufficient to maintain two FIFO packet queues and direct packets to the appropriate queue based on the size of the flow they belong to. This can be done by counting either the number of packets or bytes transmitted so far. In our work, we use the connection tracking capabilities of a stateful firewall to achieve this, as described in more detail in subsection 4.4.

Furthermore, by using different queueing disciplines instead of FIFO for the two levels, such as the AQM disciplines described in section 2, we can create 2LPS variants that support AQM. Since these AQM disciplines can still be modelled as Processor Sharing from the perspective of TCP flows, the theoretical results for 2LPS should still apply to these variants.

3.3. Threshold selection

While 2LPS is easier to implement than LAS, it does require choosing a good threshold to be effective. The first step in selecting one is to fit the observed flow sizes to a generalized hyperbolic (GH) distribution. The GH distribution is flexible enough to approximate any positive-valued empirical distribution, making it suitable for modelling real-world flow sizes.

To achieve this, we first use PhFit [19] to fit flow size data to a phase-type (PH) distribution in canonical form. We then convert this PH distribution into an equivalent GH distribution. Finally, we can solve the M/GH/1/2LPS queue by optimizing the expected response time for jobs in the system. Previous works have shown that this approach is both feasible and computationally efficient [5], and have also provided a toolkit to implement it [20].

The optimal threshold a primarily depends on the job size distribution, which is usually somewhat stable over time, and to a lesser extent on the system load. Therefore, it is possible to compute a suitable threshold using a recent traffic trace and a high load parameter. For deployment in a real system, it would be best to periodically recompute this threshold to adapt to changing traffic patterns. This could be accomplished by running a background process that collects flow size statistics and updates the threshold as needed.

In our experiments, we use a static threshold computed from the distribution used to generate the traffic and the load factor specific to each experiment.

4. Queue Implementation

For our experiments, we chose six queueing disciplines, across three different AQM strategies:

- FIFO (no AQM);
- CoDel, FQ-CoDel and CAKE;
- PIE and FQ-PIE.

We tested each of them by themselves, as well as an age-based two-level variant of them. In order to configure them on our test hosts, we used a combination of Traffic Control and nftables.

4.1. Traffic Control

The `tc` command is used to configure Traffic Control in the Linux kernel. This subsystem is responsible for shaping network traffic, scheduling packet transmission, policing incoming traffic, and dropping excess packets when necessary.

A queueing discipline (qdisc) is a core component of this system. When a packet is ready to be sent to an interface, the kernel places it into the associated qdisc, as specified by the `tc` configuration. Packets are then dequeued from the qdisc and handed over to the network adapter driver for transmission.

Normal classless qdiscs implement common queueing algorithms, such as FIFO, fair queueing and various AQM solutions, and may include support for diffserv or traffic shaping.

Classful qdiscs can support multiple classes, each with its own qdisc. When a packet is dequeued from a classful qdisc, it recursively dequeues from its child disciplines, according to its implementation. For enqueueing, `tc` allows the use of filters to map packets to specific classes. More advanced filtering can be achieved using external tools, such as nftables, to classify packets.

Additionally, `tc` provides the ability to retrieve queueing statistics, including the total number of bytes or packets transmitted and dropped. This also applies to queues associated with classes, which is particularly useful for debugging complex configurations.

4.2. Basic queues

As all the basic queueing disciplines we want to test are already implemented in the Linux kernel, it is enough for us to use the traffic control utility to configure them on an interface.

The `tc-pfifo` and `tc-pfifo_fast` modules provide implementations for the FIFO discipline, with a configurable packet limit. The `tc-codel` and `tc-fq_codel` modules provide implementations for CoDel and FQ-CoDel respectively, and allow setting values for the interval and target parameters. The `tc-cake` module implements CAKE, which can be

configured with an RTT parameter, from which the interval and target used in CoDel are derived. Finally, the `tc-pie` and `tc-fq_pie` modules provide implementations for PIE and FQ-PIE, with a configurable delay target and update interval. Shown in Listing 15 is an example on how to configure an interface to use FQ-CoDel.

```
1 tc qdisc add dev enp0s31f6 root handle 1: fq_codel
```

Listing 15: Example traffic control configuration for a simple FQ-CoDel queue.

4.3. Multi-level queues

As `tc-pfifo_fast` and `tc-cake` support diffserv out of the box, they can handle scheduling traffic with different priorities without any extra effort. On the other hand, the other four have no support for any kind of priority scheduling. To be able to implement a multi-level queueing discipline, where each priority level is handled by either `tc-codel`, `tc-fq_codel`, `tc-pie`, or `tc-fq_pie`, we can use `tc-prio`.

The `tc-prio` module implements a classful discipline that allows configuring an arbitrary amount of priority bands, each with its own queueing discipline. The classes are dequeued with strict priority, with the lowest class index first. For example, configuring `tc-prio` with three classes handled by `tc-pfifo` would result in a queueing discipline equivalent to `tc-pfifo_fast`.

As such, to implement multi-level CoDel, FQ-CoDel, PIE, and FQ-PIE, we use `tc-prio` with classes configured to be `tc-codel`, `tc-fq_codel`, `tc-pie`, and `tc-fq_pie` respectively, while to implement multi-level FIFO and CAKE disciplines we can use directly `tc-pfifo_fast` and `tc-cake` respectively. An example on how to configure `tc-prio` with FQ-CoDel is shown in Listing 16.

```
1 tc qdisc add dev enp0s31f6 root handle 1: prio bands 3
2 tc qdisc add dev enp0s31f6 parent 1:1 handle 10: fq_codel
3 tc qdisc add dev enp0s31f6 parent 1:2 handle 20: fq_codel
4 tc qdisc add dev enp0s31f6 parent 1:3 handle 30: fq_codel
```

Listing 16: Example traffic control configuration for a multi-level FQ-CoDel queue.

4.4. Priority scheduling

Once we have queues that support multiple priority levels, it is necessary to actually schedule packets in the right class. To this end we use `nftables`, configured as shown in Listing 18.

To assign a class to a packet, `nftables` allows setting packet meta-information. In particular, by assigning a value to the `meta priority` of a packet it is possible to direct it to a specific sub-queue configured on the interface or to a specific priority band.

As what `meta` priority values map to depends on the configured queue, we need a different configuration for each setup. When using a setup built on `tc-prio`, the `meta` priority value should correspond to the handle of the `tc-prio` queueing discipline followed by the class. If the handle of the root `tc-prio` is 1, a priority value of 1:1 will have higher priority, and a value of 1:2 will have lower priority. When using `tc-pfifo_fast`, the minor number of the `meta` priority should instead correspond to a type of service class, which under default configuration would map a value 1:6 to the highest priority class, and a value of 1:0 to the middle priority class. Finally, when using `tc-cake`, the minor number is interpreted as a tin number, so 1:2 maps to the best-effort tin, and 1:1 to the bulk traffic tin, which has lower priority.

To then dynamically assign a `meta` priority value to each packet, we can use the connection tracking of nftables. In particular, the `ct bytes` value represents how many bytes have been already sent by the connection the current packet belongs to. By mapping ranges of `ct bytes` to specific `meta` priority values, we can achieve priority scheduling depending on the age of the flow.

To update the threshold it is enough to replace the `map`, which can be done in one command, as shown in Listing 17.

```
1 nft add map inet prioritize multiqueue { typeof ct bytes : meta
    priority; flags interval; elements={ 0-$threshold : 1:2, * : 1:1 }}
```

Listing 17: Nftables command to update threshold.

```

1 table inet prioritize {
2   map multiqueue {
3     # types of the data in the map,
4     # from bytes measured by conntrack
5     # to priority levels
6     typeof ct bytes : meta priority
7     # map supports interval ranges
8     flags interval
9     # two ranges with a fixed threshold
10    elements = {
11      0-6331208 : 1:1,
12      * : 1:2,
13    }
14  }
15
16  chain y {
17    type filter hook output priority 0; policy accept;
18    # set the priority in the metadata of the packet
19    # based on the data extracted from ct bytes, and mapped
20    # using the map we defined before
21    meta priority set ct bytes map @multiqueue
22  }
23 }

```

Listing 18: Example nftables configuration for a tc-prio setup.

5. Experimental Setup

To analyse and compare the performance of these hybrid algorithms, we measured the Flow Completion Time (FCT) of TCP flows under heavy load. We chose to use synthetic traffic on real hardware, as this provides a controlled yet high-fidelity environment. An initial attempt using virtual machines was unsuccessful, as we were unable to satisfactorily limit the throughput of the virtual interfaces.

We conducted our experiments on two different setups to observe how results scale between 1 Gbps and 10 Gbps. However, the higher throughput setup required us to adjust the traffic generation parameters to avoid running into CPU limitations. In both cases, we used bounded Pareto distributed traffic. A long-tailed traffic distribution is both realistic and well-suited to age-based algorithms. The upper bound allows us to control the variance in the collected data, enabling us to produce cleaner statistics.

5.1. Hardware setup

The first setup consists of two bare-metal nodes, a "client" and a "server", each with an Intel Core i5-6600 CPU, 16 GB of RAM, and an integrated Intel I219-LM 1 Gbps network interface. They are directly connected to each other by an Ethernet cable using the integrated interfaces, referred to as "test" interfaces, and are controlled via separate network links established with USB Ethernet adapters.

The second setup is very similar, but uses slightly different hardware. The two nodes have 8 GB of RAM each, and the "client" and "server" have respectively an Intel Core i5-4590S CPU and an Intel Core i5-3470 CPU. As test interfaces, they each have installed an Intel X540-AT2 dual 10 Gbps PCIe Ethernet card, which are directly connected to each other by an Ethernet cable.

Listed in Table 1 are the base performance values of the interfaces for each setup. It includes the mean RTT measured with the ping utility on 100 samples, and the throughput measured using iperf over a single 60 seconds TCP stream.

Setup	Ethernet Controller	Link Mode	RTT	Throughput
1	Intel I219-LM	1000baseT/Full	0.710 ms	864 Mbits/sec
2	Intel X540-AT2	10000baseT/Full	0.270 ms	9.34 Gbits/sec

Table 1: Test link base performance.

Test traffic is run through the test link, established using the test interfaces. On this link, the machine designated "client" is assigned the IP 192.168.42.1, while the machine designated "server" is assigned the IP 192.168.42.2.

Following recommendations from the bufferbloat team [21], many hardware offloads have been disabled on the test interfaces, as to not introduce extra latency during the tests.

These include RX and TX checksumming, scatter-gather, TCP segmentation and generic fragmentation, generic receive offload, and VLAN acceleration.

5.2. Traffic generation

To generate traffic, we use a closed-loop test configuration. It consists in N simulated actors, that independently and concurrently each alternate a thinking phase with an action phase. During the thinking phase, an actor sleeps for an exponentially distributed random amount of time, to then enter the action phase. During the action phase, an actor uploads a random amount of data using TCP, and returns to the thinking phase on flow completion.

To simulate a long-tailed traffic distribution, the amount of data to upload is sampled from a bounded Pareto random variable. Together with the exponential thinking time, this results in traffic with a Poisson arrival process and Pareto job size. The stability condition is always guaranteed in a closed-loop test, and utilization can be tuned by changing the amount of actors or the parameters of the arrival process or job distribution. The parameters used to configure traffic generation are shown in Table 2.

Setup	Arrival Process		Flow Size Distribution			Load Factor
	N Actors	Thinking Time	α	Min. Size	Max. Size	
1	80	1 s	1.2	300 KiB	1 GiB	0.88
2	150	0.35 s	1.1	373 KiB	62 GiB	0.92

Table 2: Traffic generation parameters.

For the first setup, which uses a 1 Gbps link, we configured the test with 80 actors. Each actor has a mean thinking time of 1 second. The job sizes are sampled from a bounded Pareto distribution with shape parameter $\alpha = 1.2$, and bounded between 300 KiB and 1 GiB. This configuration results in a load factor of 0.88 with respect to the nominal link rate of the interface.

For the second setup, with a 10 Gbps link, we used 150 actors and set the mean thinking time to 0.35 seconds. The job sizes are again sampled from a bounded Pareto distribution, with shape parameter $\alpha = 1.1$, and bounded between 373 KiB and 62 GiB. This setup achieves a load factor of 0.92.

Before settling on these parameters, we experimented with a range of different configurations. When the load factor was too low, the choice of scheduling algorithm had little impact on performance. Conversely, when the load factor was too high, performance degraded rapidly for all algorithms. Reducing the minimum job size increased the number of flows per unit time, which led to higher CPU usage. Increasing the maximum job size raised the run-to-run variance, since very large flows became increasingly rare. Additionally, due to implementation details, the RAM required by the configuration parser scales with the number of actors N , making it impractical to test much larger values.

5.3. Antler

To run the tests we used Antler [22], a tool for network and congestion control testing written in the Go language, which allows the definition of tests using configuration files written in the CUE language. The test configuration is kept on a coordinator machine, which parses it and connects to the test nodes via SSH to start the processes necessary to run the test. Test data collected by the remote machines is sent back over the SSH connection to be post-processed and stored by the coordinator machine.

An Antler test consists of a tree-like combination of configurable pre-defined runners. A runner can, for example, execute shell commands on a node, generate TCP or UDP traffic, or execute other runners sequentially or in parallel.

As Antler was built to execute shorter tests, with a predetermined amount of flows, we had to extend the set of available runners in order to implement closed-loop tests. We implemented a `ClosedLoopActor` runner which alternates an exponentially distributed thinking phase with executing a configured sub-runner, quitting after a set amount of time. We also implemented a `Random` runner, which takes as parameters a list of runners and their associated weights. It selects and runs one of the specified runners, sampling according to the given probabilities. Listing 19 shows how we combined these runners to configure a closed-loop test. Other runners present include `Parallel`, which allows us to instantiate N independent `ClosedLoopActors`, and `StreamClient`, which runs a single TCP upload connection with the specified length.

To sample the exponential thinking time we use the `ExpFloat64()` function from the standard `rand` module, called by the `ClosedLoopActor` runner and adjusted to have the desired mean thinking time. To sample the flow sizes, we use inverse transform sampling instead. The desired Pareto CDF is pre-computed, then discretized into M levels, 249 for the 1 Gbps setup and 199 for the 10 Gbps setup. In the Antler test config we load the M samples and corresponding M probabilities, which will be used by the `Random` runner. Then, when it needs to choose what size of flow to start, the runner samples a uniform random value between 0 and 1, and uses binary search on the cumulative probabilities to determine which sub-runner to execute. Full source code of the new runners is included in Appendix A.

5.4. Queue Configuration

While CoDel claims to be parameterless for common Internet deployments, this setup is very much an outlier when considering the RTT of usual Internet connections. As such we had to reduce both `interval` and `target`. We landed on values of 10 ms for `interval` down from 100 ms default, and 0.5 ms `target`, corresponding to 5% of `interval`. While a lower `interval` could have been more appropriate for our sub 1 ms RTT link, CoDel authors do not recommend going below 10 ms on a general purpose Linux kernel build.

CAKE only supports specifying the RTT, and automatically sets the `target` to 5% of the configured value. We configured it to use a value of 10 ms, to match the other CoDel based algorithms.

```

1 {Parallel: [
2   for j in list.Range(0, _nActors, 1)
3   {ClosedLoopActor:{
4     ThinkingTime: _thinkingTime
5     Duration: _duration
6     Seed: _seed*_nActors*2 + j*2
7     Random: {
8       Seed: _seed*_nActors*2 + j*2+1
9       Run: [
10        for len in _connsizespareto
11        {StreamClient: {
12          Addr: _rig.serverAddr
13          Upload: {
14            Flow: "a\(j)"
15            CCA: "cubic"
16            Length: len
17            Duration: "0s"
18          }
19        }},
20      ]
21      Weights: _connprobpareto
22    }
23  }},
24 ]},

```

Listing 19: Antler runner configuration for a closed-loop test.

For PIE, the configurable values include a delay target, similar to that of CoDel, and an update interval, both defaulting to 15 ms. We set the target to 0.5 ms to match CoDel, and the update interval to 3 ms.

To select the priority thresholds for the two-level algorithms we used the method described in subsection 3.3. As the two setups have different traffic distributions, we set a threshold of 6.33 MB for the 1 Gbps setup and 19.5 MB for the 10 Gbps setup.

5.5. Test process

As our goal is comparing the performance of different queueing disciplines, we need to run the same test for each algorithm under test. The queueing disciplines we are testing are FIFO, CoDel, FQ-CoDel, CAKE, PIE and FQ-PIE. For each of these disciplines, we test the queue normally as well as a two-level variant.

Each test in a set of twelve uses the same seed to generate traffic, and runs for 6 hours with the first setup, and 4 hours with the second. The first hour of data for the first setup, and half hour of data for the second setup, is discarded as warm-up period. This was determined using a cumulative moving average analysis: the test duration was chosen based on when the mean flow completion time stabilized, and the startup period was set by observing when the relative performance of the different algorithms stopped changing significantly.

Every test run follows these steps:

1. the traffic control queueing disciplines and nftables configurations on the test interfaces are reset;
2. interface hardware offloads are disabled;
3. the traffic control queueing discipline for the test interface on the client node is configured to use the algorithm under test;
4. when testing a two-level variant, nftables is configured with the right priority scheduling settings, using a pre-computed static threshold;
5. performance monitoring is started on both nodes;
6. on the server node, a TCP server is started;
7. on the client node, N parallel actor runners are run for the configured test duration;
8. finally, all test and performance data is sent to the coordinator machine.

A set of tests repeats all of these steps for each queueing discipline, using the same seed to generate traffic. All collected data is stored on disk for analysis.

6. Experimental Results

After running three test sets for the first setup and two for the second, each with a different seed, we can begin to analyse the collected data. We look at Flow Completion Time (FCT), as well as throughput fairness.

The algorithms under test can be grouped in three main ways. First, we compare two-level queueing disciplines to their non-age-based implementations. Second, we examine the effects of CoDel and PIE AQM compared to having no AQM. Finally, we compare flow-queueing algorithms to their single-queue counterparts.

One notable limitation of this setup is real-time traffic, or protocols for which the "flow as a job" model does not apply, such as Voice over IP or game synchronization packets. While the AQM solutions under test are designed to keep latency in check, penalizing long-running flows that are not bandwidth-intensive but are sensitive to latency may lead to performance degradation.

6.1. Mean Flow Completion Time

The first interesting metric to consider is mean Flow Completion Time, which is what an age-based scheduling algorithm aims to optimize. It is obtained by taking the mean of the completion time over each flow. We highlight the difference between each age-based algorithm and its base version by computing the relative reduction in mean FCT between the two, shown in the result tables as Gain, defined as $T_{age}/T_{base} - 1$.

Algorithm	Two-Level version		Base version		Gain
	Mean FCT	95% CI	Mean FCT	95% CI	
FIFO	123	1.46	154.3	0.748	-20 %
CAKE	127.2	1.37	142.1	0.906	-10 %
FQ-CoDel	119.4	1.5	137.6	0.966	-13 %
CoDel	118.1	1.47	122.2	0.971	-3.3 %
FQ-PIE	126	1.14	138.7	0.824	-9.1 %
PIE	119.4	1.27	130.9	0.813	-8.8 %

Table 3: Mean Flow Completion Time for the 1 Gbps setup. All data is in milliseconds. Gain computed as relative FCT reduction between the base version and the two-level version.

The results for the 1 Gbps setup, listed in Table 3 and plotted for clarity in Figure 3, show that the mean Flow Completion Time for two-level variants of algorithms is always better than that of their base versions. When considering age versus non-age algorithm pairs, the biggest improvements of over 20% can be seen for FIFO, as it has the worst base performance, followed by FQ-CoDel with a 13% improvement, CAKE with 10%, PIE and FQ-PIE with around 9%, and finally CoDel with only a 3.3% reduction in FCT.

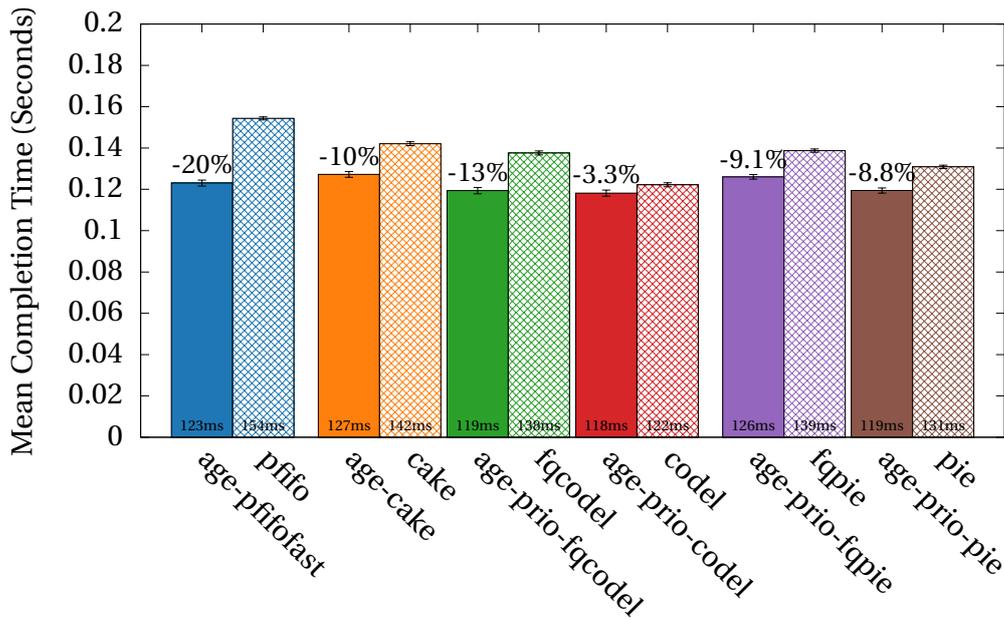


Figure 3: Mean Flow Completion Time for the 1 Gbps setup, organized by algorithm. Pairs of data points show data for the same AQM solution, with and without two level scheduling. Lower is better. Error shown as 95% confidence intervals on the mean.

The best overall algorithms are two-level CoDel, two-level FQ-CoDel, and two-level PIE all within error of each other, followed by base CoDel, and two-level FIFO.

When looking at AQM solutions, all non-age AQM algorithms are better than the non-age non-AQM FIFO algorithm, with CoDel slightly outperforming PIE. In this setup, the flow-queue algorithms perform worse than their single-queue versions, for both CoDel and PIE, with at best two-level FQ-CoDel coming within error of two-level CoDel. CAKE is slightly worse than FQ-CoDel in both the age and non-age scenarios, despite being a very similar algorithm.

All the age-based algorithm tend to have similar performance, with the two-level versions of CoDel, FQ-CoDel, FIFO, and PIE, having mean FCT within 4% of each other. Age-based AQM solutions still slightly beat age-based FIFO, but two-level CAKE and two-level FQ-PIE are slightly worse than no AQM (two-level FIFO) in this setup.

With this kind of load, CoDel by itself is able to match the performance of the two-level FIFO implementation as well as beat that of two-level CAKE and FQ-PIE, but is still slightly worse than two-level CoDel.

For the 10 Gbps setup, results listed in Table 4 and visible in Figure 4, the situation is slightly different. Most noticeably, CAKE behaves erratically, with FCT that are 5 to 15 times larger than the other algorithms. This behaviour is repeatable, but we were unable to explain what causes it, as both goodput and CPU usage are also significantly lower when using CAKE or its age-based variant. While the behaviour of CAKE is very much an outlier, the other

Algorithm	Two-Level version		Base version		Gain
	Mean FCT	95% CI	Mean FCT	95% CI	
FIFO	42.08	0.438	45.98	0.374	-8.5 %
CAKE	184.8	10.3	662.5	22.9	-72 %
FQ-CoDel	38.93	0.503	40.09	0.476	-2.9 %
CoDel	40.04	0.465	43.95	0.405	-8.9 %
FQ-PIE	38.13	0.529	41.81	0.487	-8.8 %
PIE	38.17	0.514	42.23	0.462	-9.6 %

Table 4: Mean Flow Completion Time for the 10 Gbps setup. All data is in milliseconds. Gain computed as relative FCT reduction between the base version and the two-level version.

disciplines are more stable, and thus worth analysing.

Every two-level discipline is still better than its base variant, but the reduction in FCT is smaller compared to the 1 Gbps setup.

Overall, the best performers are two-level FQ-PIE and two-level PIE, closely followed by two-level FQ-CoDel. The base performance of FQ-CoDel is also very good, as it very close the performance of the other age-based AQM algorithms and only 3% behind that of two-level FQ-CoDel.

PIE and FQ-PIE have very similar performance, and both see about 8% difference between the two-level and base versions.

The worst performing age-based algorithm is two-level FIFO. It is an 8.5% improvement compared to its base version, which also has the worst performance out of the non-age algorithms.

While the 1 Gbps favoured non-flow-queueing AQM algorithms, CoDel and PIE, in the 10 Gbps setup FQ-CoDel and FQ-PIE have better performance.

6.2. Flow Completion Time versus Flow Size

By looking at how the flow completion time varies depending on the flow size, we can better understand where the performance differences of subsection 6.1 come from.

The 1 Gbps setup is shown in Figure 5. The most noticeable feature of the plot, which also serves as good validation for our setup, is the sharp angle at the 6 MB mark for curves corresponding to two-level algorithms. This is where the size threshold is located, and it separates flows that only receive high priority service from flows that eventually end up in the low priority level of age-based queues.

Overall, the two-level disciplines always have better performance than all the normal disciplines for flows that are smaller than the threshold. Soon after the threshold, the performance of two-level disciplines degrades, to finally become always worse than that of normal

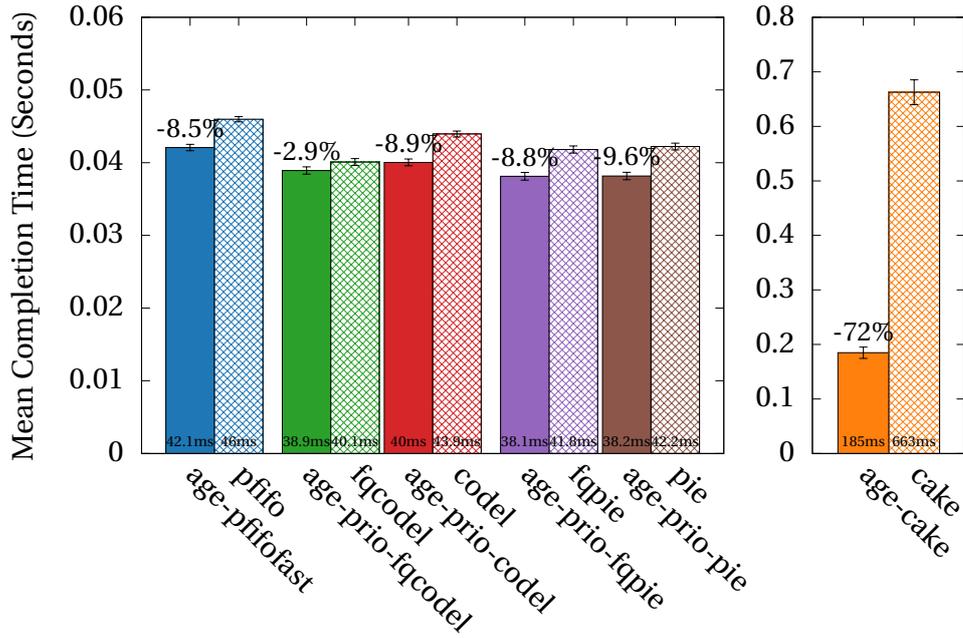


Figure 4: Mean Flow Completion Time for the 10 Gbps setup, organized by algorithm. Pairs of data points show data for the same AQM solution, with and without two level scheduling. Lower is better. Error shown as 95% confidence intervals on the mean.

disciplines around the 20 MB mark.

Starting from two-level CoDel and two-level FQ-CoDel, the top performers of the 1 Gbps setup, we can see that their behaviour is indistinguishable for flows larger than the threshold, where they have the best performance over the two-level algorithms. For flows smaller than the threshold, FQ-CoDel holds the advantage on flows smaller than 800 KB, while CoDel is better in the 800 KB to 6 MB range. From this we can deduce that the effect of flow-queueing is most beneficial to the very small flows.

Two-level PIE behaves identically to two-level FIFO before the threshold, but beats it in the 7 MB to 30 MB range. These two algorithms together have the best overall performance for flows smaller than the threshold, and perform worse than the other two-level algorithms for flows larger than the threshold.

Overall, while two-level FIFO, CoDel, FQ-CoDel and PIE all have comparable mean performance, we can see that the two CoDel based solutions are more balanced, and two-level FIFO and PIE favour smaller flows more strongly.

Two-level FQ-PIE has consistently poor performance when compared to other two-level algorithms, except for the biggest flows.

Similarly, two-level CAKE is consistently worse than two-level FQ-CoDel by a constant factor for flows smaller than the threshold, to eventually become equivalent after the 20 MB mark. When comparing normal CAKE to normal FQ-CoDel, a similar behaviour emerges. Normal

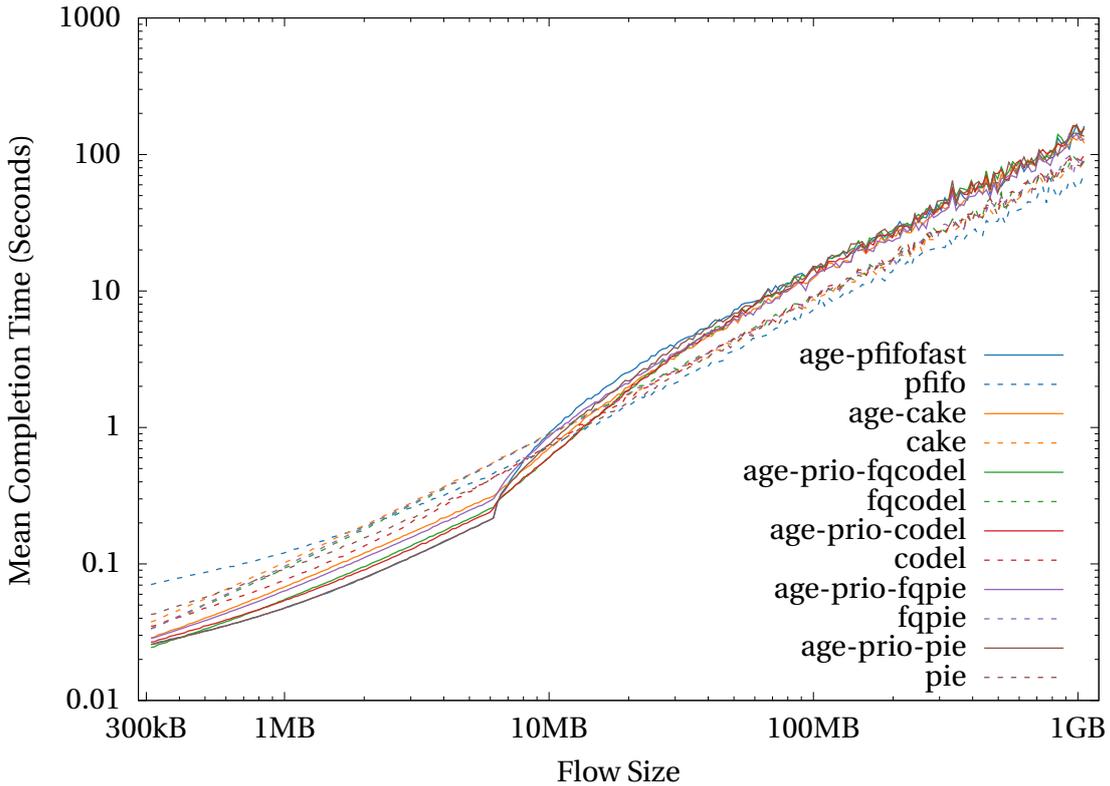


Figure 5: Mean Flow Completion Time versus Flow Size for the 1 Gbps setup, organized by algorithm. Lower is better.

CAKE is worse than normal FQ-CoDel by a constant factor up to around the 1 MB mark, the gap then shrinks, until they have equal performance after the 5 MB mark. A potential explanation to this performance gap in otherwise very similar algorithms, could be found in the fact that CAKE is slightly more complex, and thus slightly slower at processing packets, by a constant amount. As the RTT of our setup is very small, this extra delay is not insignificant, and ends up affecting the slow-start phase of the TCP connections.

Two-level FIFO has the best performance for flows below the threshold, and the worst performance for flows above it. Inversely, normal FIFO is by far the worst discipline for flows smaller than 1 MB, to then become the best for flows larger than 20 MB. These two algorithms are the most extreme, swapping places around the threshold.

In the 10 Gbps setup, shown in Figure 6, the threshold, and therefore the sharp angle for two-level disciplines, is located at 19 MB. The gap between algorithms is narrower, which matches the results shown in subsection 6.1. At the same time, the total range of observed completion times has widened, with lower lows and higher highs compared to the 1 Gbps setup. Data for CAKE and two-level CAKE is not shown for readability.

Looking at the age-based algorithms first, two-level PIE and two-level FQ-PIE have a slight advantage over the other two-level solutions on the smallest flows up to the threshold, with

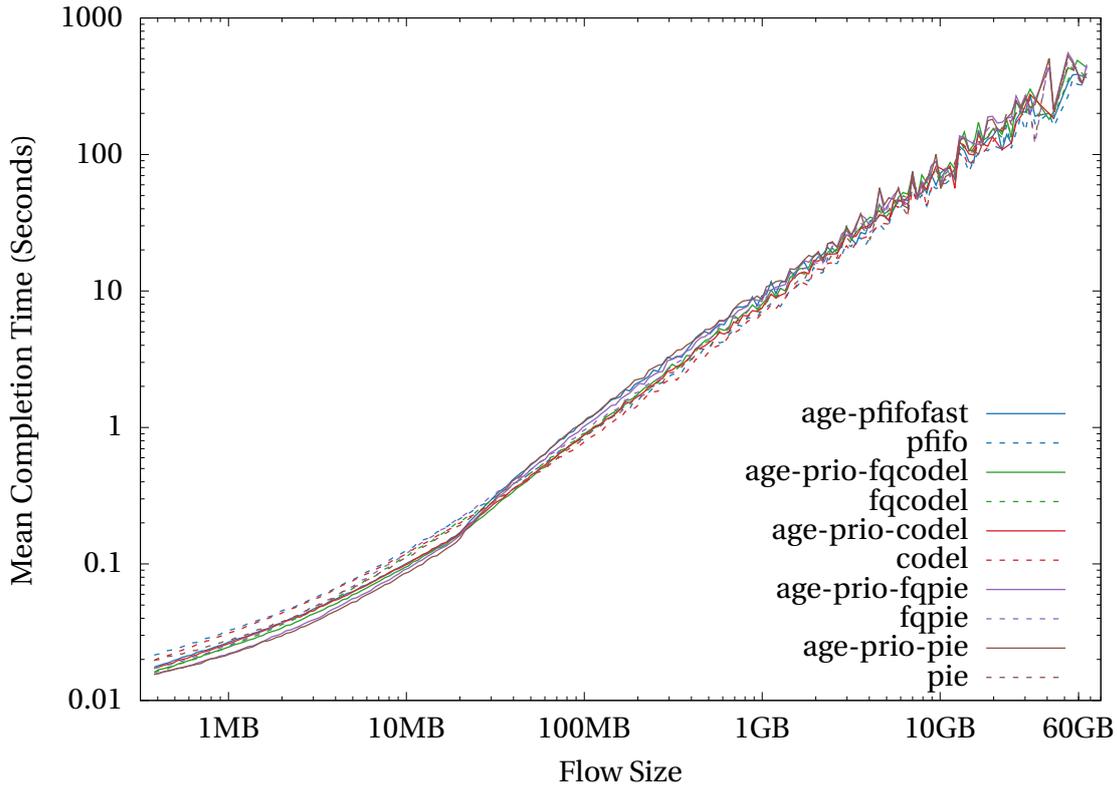


Figure 6: Mean Flow Completion Time versus Flow Size for the 10 Gbps setup, organized by algorithm. Lower is better.

two-level FQ-CoDel third best in this region. Two-level FIFO, two-level PIE and two-level FQ-PIE have similar performance after the threshold, and are slightly worse than the other three. Where two-level FQ-CoDel and two-level CoDel shine most is between the threshold and below the 500 MB mark, where they beat the other age-based algorithms and are very similar to the best non-age algorithms. The behaviours of all the age-based algorithms end up converging at the 800 MB mark, and remain similar thereafter.

The non-age-based flow-queueing disciplines, FQ-CoDel and FQ-PIE, perform as well as any two-level algorithm up to the 3 MB mark. From then until after the threshold they have a worse performance than the two-level algorithms, and finally from around the 30 MB mark they have a performance equal to the best of the two-level algorithms. Between them, FQ-CoDel consistently holds a steady advantage.

Non-age-based FIFO and CoDel have very similar behaviours, with CoDel coming out slightly on top. They are significantly worse than the other algorithms up until the 10 MB mark, but they are slightly better than the other algorithms for the bigger flows after the 100 MB mark.

Overall, the most significant differentiators are age scheduling and flow-queueing, with the choice of AQM having less of an impact, suggesting that we might need better tuning for the

CoDel and PIE parameters.

6.3. Lorenz Curve Gap

The Lorenz Curve Gap is the maximum euclidean distance between the Lorenz Curve and the curve of maximum fairness, rescaled by its maximum value [23]. It is computed over the mean throughput of each of the flows, obtained using flow size and completion time. A Lorenz Curve Gap value of 0 would indicate that all the flows experienced equal mean throughput, while a value close to 1 would indicate that a very small minority of flows received the vast majority of bandwidth. Figure 7 shows, for example, the Lorenz Curve for CoDel in the 1 Gbps setup, highlighting the gap.

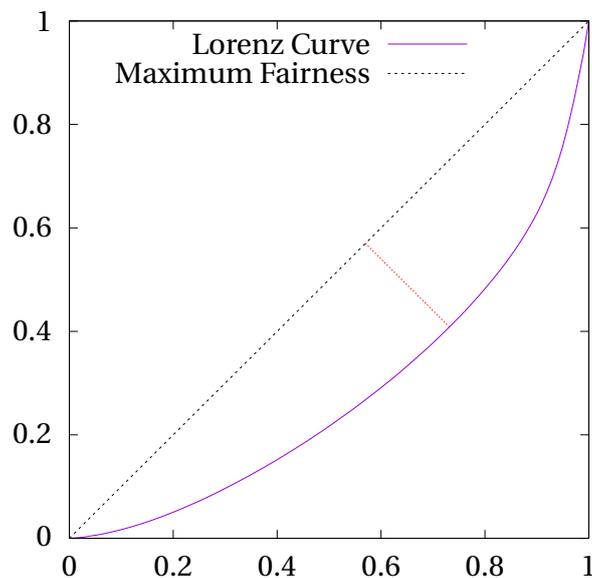


Figure 7: Lorenz Curve for CoDel in the 1 Gbps setup.

Due to the slow-start effect caused by TCP congestion control algorithms, even a scheduler that enforces byte-based fairness would be unable to achieve a perfect score when the flow distribution mixes very short with larger flows.

A plot of the Lorenz Curve Gap for the 1 Gbps setup is shown on Figure 8. In all six cases the two-level algorithms show better fairness than their base versions. The rankings and relative scores of the algorithms are very similar to the results for the mean flow completion time of subsection 6.1.

It appears that unfairly favouring small flows, when they are the least greedy, ends up offsetting the fact that they use less than their fair share of bandwidth due to slow-start, overall improving fairness across all flows.

The situation is slightly different in the 10 Gbps setup, shown in Figure 9. Other than the two CAKE algorithms behaving like outliers, CoDel scores better than two-level CoDel, and

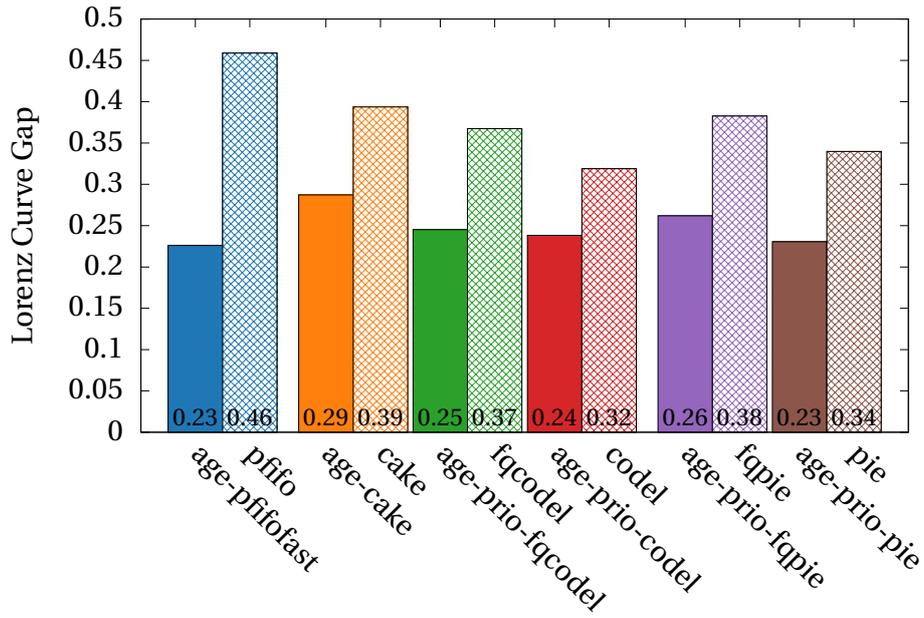


Figure 8: Lorenz Curve Gap for the 1 Gbps setup, computed on the mean throughput, organized by algorithm. Values are between 0 and $1 - \frac{1}{n}$, lower is better.

PIE scores better than two-level PIE, which is opposite to what happens in the 1 Gbps setup. With FIFO, FQ-CoDel and FQ-PIE the two-level version scores better than the non-age-based one, which matches the behaviour seen in the 1 Gbps setup.

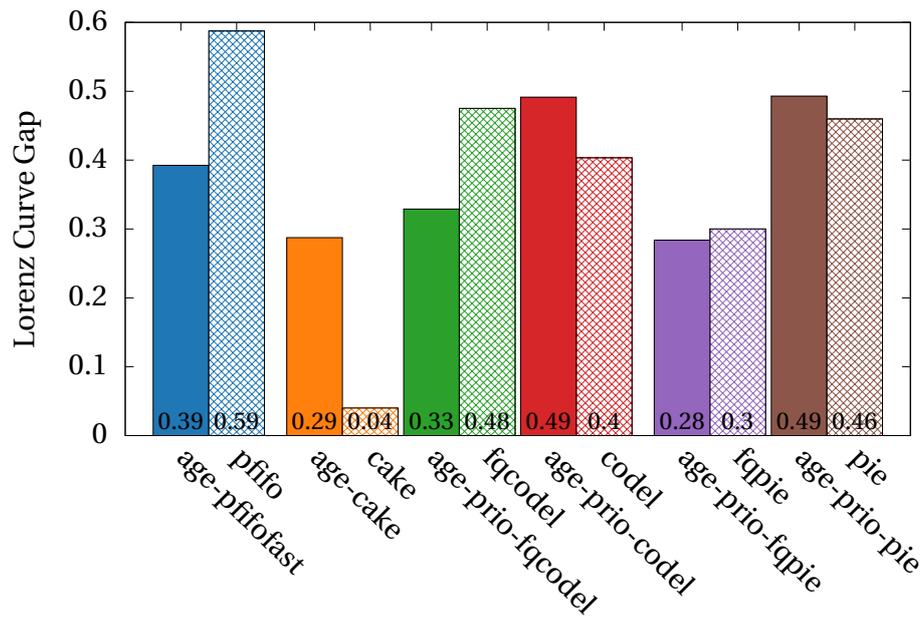


Figure 9: Lorenz Curve Gap for the 10 Gbps setup, computed on the mean throughput, organized by algorithm. Values are between 0 and $1 - \frac{1}{n}$, lower is better.

7. Conclusions

In this work, the combination of 2LPS scheduling and Active Queue Management (AQM) was explored to determine whether these techniques can complement each other. The results show that both approaches improve Flow Completion Time under heavy load with heavy-tailed traffic, and that using them together yields even greater improvements. This suggests that modern networks could benefit from deploying age-based scheduling alongside AQM.

To reach these conclusions, closed-loop tests were conducted on a variety of algorithms using Antler, a network performance testing tool that was extended for this purpose. Algorithms already implemented in the Linux kernel, such as FIFO, CoDel, and PIE, as well as their flow-queueing versions FQ-CoDel, FQ-PIE, and CAKE, were compared against 2LPS queues constructed from stable Linux components like the `tc -prio` queueing discipline and `nftables`. Experiments were run on two different setups to examine how results vary between 1 Gbps and 10 Gbps links.

In both test setups, age-based scheduling consistently led to significant improvements in Flow Completion Time across all base queues. All AQM solutions also outperformed the FIFO queue, further confirming their effectiveness. The best overall performance was achieved by two-level FQ-CoDel, which was tied for best in the 1 Gbps setup and also had good results in the 10 Gbps setup.

A notable exception was CAKE, which consistently misbehaved in the 10 Gbps setup, failing to produce useful data, despite no other system metrics indicating a problem. Additionally, the 10 Gbps environment exhibited greater instability in general, with higher run-to-run variance compared to the 1 Gbps setup.

While the findings are promising, further research is needed before these techniques can be widely adopted in real-world deployments. First, the analysis should be expanded to include other performance metrics, such as latency. Special attention should be given to real-time protocols, where latency is critical and the "flow as a job" model described in section 3 may not apply. Such flows may be non-greedy and require minimal bandwidth, but if they persist long enough, the current queue implementation would eventually move them to the low priority band, potentially degrading their performance.

Another direction for future work is a fully in-kernel implementation, using the flow-queueing mechanisms of FQ-CoDel and FQ-PIE to isolate and schedule flows according to an age-based discipline, while applying AQM to each flow independently. This kind of implementation would also be able to explore other age-based scheduling algorithms, such as LAS.

Finally, although the test setup used real hardware and software, the round-trip time of the test network was extremely low compared to the Internet. Since both TCP congestion control and AQM algorithms are sensitive to RTT, experiments with a more realistic network size would provide a better understanding of these techniques in practice.

A. Antler Modifications

```
1 // Run represents the information needed to coordinate the execution of runners.
2 // Using the Serial, Parallel and Child fields, Runs may be arranged in a tree
3 // for sequential, concurrent and child node execution.
4 //
5 // Run must be created with valid constraints, i.e. each Run must have exactly
6 // one of Serial, Parallel, Child or a Runners field set. Run is not safe for
7 // concurrent use, though Parallel Runs execute safely, concurrently.
8 type Run struct {
9     // Serial lists Runs to be executed sequentially
10    Serial Serial
11
12    // Parallel lists Runs to be executed concurrently
13    Parallel Parallel
14
15    // Schedule lists Runs to be executed on a schedule.
16    Schedule *Schedule
17
18    // ClosedLoopActor alternates a thinking phase with a Run phase.
19    ClosedLoopActor *ClosedLoopActor
20
21    // Random executes a random Run from a list of Runs.
22    Random *Random
23
24    // Child is a Run to be executed on a child Node
25    Child *Child
26
27    // Runners is a union of the available runner implementations.
28    //
29    // NOTE: In the future, this may be an interface field, if CUE can be made
30    // to choose a concrete type without using a field for each runner.
31    Runners
32 }
33
34 // ClosedLoopActor is a runner that alternates an exponentially distributed
35 // thinking phase with a Run phase.
36 type ClosedLoopActor struct {
37     // Duration is the lifetime of the runner.
38     Duration metric.Duration
39
40     // ThinkingTime is the mean time of the thinking phase.
41     ThinkingTime metric.Duration
42
43     // Run is the Run to execute every time the thinking phase ends.
44     Run
```

```

45
46 RandomRunner
47 }
48
49 // flowIndexCtxKey is the context key used to store what iteration a
50 // ClosedLoopActor is on, so that a unique Flow name can be generated for each
51 // iteration.
52 type flowIndexCtxKey struct {}
53
54 // do alternates waiting for a thinking phase to end with executing the Run.
55 func (a *ClosedLoopActor) do(ctx context.Context, arg runArg, ev chan event) (
56   ofb Feedback, ok bool) {
57   ofb = Feedback{}
58   ok = true
59   var i int
60
61   a.initRandom()
62
63   dc := ctx.Done()
64   end := time.After(time.Duration(a.Duration))
65   w := time.After(a.nextWait())
66
67   for ok {
68     select {
69     case <-dc:
70       return
71     case <-end:
72       return
73     case <-w:
74       ctx := context.WithValue(ctx, flowIndexCtxKey{}, i)
75       var fb Feedback
76       fb, ok = a.Run.run(ctx, arg, ev)
77       w = time.After(a.nextWait())
78       i++
79
80       if e := ofb.merge(fb); e != nil {
81         ok = false
82         rr := arg.rec.WithTag(typeBaseName(a.Run))
83         ev <- errorEvent{rr.NewError(e), false}
84       }
85     }
86   }
87   return
88 }
89
90 // nextWait returns the next wait time, which is a random duration

```

```

91 // exponentially distributed with a mean of ThinkingTime.
92 func (a *ClosedLoopActor) nextWait() time.Duration {
93     return time.Duration(a.rand.ExpFloat64() * float64(a.ThinkingTime))
94 }
95
96 // validate returns the validation error from the Run.
97 func (a *ClosedLoopActor) validate() (err error) {
98     if err = a.Run.Validate(); err != nil {
99         return
100     }
101     return
102 }
103
104 // Random executes a random Run from a list of Runs.
105 type Random struct {
106     // Run is a list of Runs to sample from.
107     Run []Run
108
109     // Weights is a list of weights for each Run. If nil, the Runs are
110     // selected uniformly. If not nil, the Runs are selected with a
111     // probability proportional to the weight of each Run.
112     Weights []float64
113
114     cumulativeWeights []float64
115     RandomRunner
116 }
117
118 // do executes a random Run from the list of Runs.
119 func (r *Random) do(ctx context.Context, arg runArg, ev chan event) (
120     ofb Feedback, ok bool) {
121     r.initRandom()
122     run := r.sampleRun()
123     ofb, ok = run.run(ctx, arg, ev)
124     return
125 }
126
127 // sampleRun returns a random Run from the list of Runs. If Weights is nil,
128 // a random Run is selected uniformly. If Weights is not nil, a random Run is
129 // selected with a probability proportional to the weight of each Run.
130 func (r *Random) sampleRun() (run *Run) {
131     if len(r.Run) == 0 {
132         return nil
133     }
134     if r.Weights == nil {
135         // when no weights are specified, select a random run uniformly
136         run = &r.Run[r.rand.Intn(len(r.Run))]

```

```

137 } else {
138 // lazy init cumulative weights
139 if r.cumulativeWeights == nil {
140     r.cumulativeWeights = make([]float64, len(r.Weights))
141     r.cumulativeWeights[0] = r.Weights[0]
142     for i := 1; i < len(r.Weights); i++ {
143         r.cumulativeWeights[i] = r.cumulativeWeights[i-1] + r.Weights[i]
144     }
145 }
146 // obtain a random value in the range of [0, sum(weights))
147 randVal := r.rand.Float64() * r.cumulativeWeights[len(r.cumulativeWeights)-1]
148
149 // binary search for the first weight that is greater than the random value
150 low, high := 0, len(r.cumulativeWeights)-1
151 for low < high {
152     mid := (low + high) / 2
153     if r.cumulativeWeights[mid] <= randVal {
154         low = mid + 1
155     } else {
156         high = mid
157     }
158 }
159 run = &r.Run[low]
160 }
161 return
162 }
163
164 // validate returns the first validation error from each of the Runs.
165 // If Weights is not nil, it must have the same number of elements as Run.
166 func (r *Random) validate() (err error) {
167     for _, r := range r.Run {
168         if err = r.Validate(); err != nil {
169             return
170         }
171     }
172     if r.Weights != nil && len(r.Run) != len(r.Weights) {
173         err = fmt.Errorf("number_of_weights_(%d)_must_match_number_of_runs_(%d)",
174             len(r.Weights), len(r.Run))
175         return
176     }
177     return
178 }
179
180 // RandomRunner is a base type for runners that use a random number generator.
181 type RandomRunner struct {
182     // Seed is the seed for the random number generator. If 0, the current

```

```
183 // time is used as the seed.
184 Seed int64
185
186 // rand is the random number generator.
187 rand *rand.Rand
188 }
189
190 // initRandom initializes the random number generator.
191 func (r *RandomRunner) initRandom() {
192     if r.rand == nil {
193         if r.Seed == 0 {
194             r.Seed = time.Now().UnixNano()
195         }
196         r.rand = rand.New(rand.NewSource(r.Seed))
197     }
198 }
```

References

- [1] K. M. Nichols and V. Jacobson, “Controlling queue delay,” *Commun. ACM*, vol. 55, no. 7, pp. 42–50, 2012. [Online]. Available: <https://doi.org/10.1145/2209249.2209264>
- [2] J. Gettys and K. M. Nichols, “Bufferbloat: Dark buffers in the internet: Networks without effective aqm may again be vulnerable to congestion collapse.” *Queue*, vol. 9, no. 11, pp. 40–54, Nov. 2011. [Online]. Available: <https://doi.org/10.1145/2063166.2071893>
- [3] —, “Bufferbloat: dark buffers in the internet,” *Commun. ACM*, vol. 55, no. 1, pp. 57–65, Jan. 2012. [Online]. Available: <https://doi.org/10.1145/2063176.2063196>
- [4] I. Rai, E. Biersack, and G. Urvoy-keller, “Size-based scheduling to improve the performance of short tcp flows,” *IEEE Network*, vol. 19, no. 1, pp. 12–17, 2005.
- [5] A. Marin, S. Rossi, and C. Zen, “Size-based scheduling for tcp flows: Implementation and performance evaluation,” *Computer Networks*, vol. 183, p. 107574, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128620312172>
- [6] L. Zhang, D. C. Partridge, S. Shenker, J. T. Wroclawski, D. K. K. Ramakrishnan, L. Peterson, D. D. D. Clark, G. Minshall, J. Crowcroft, R. T. Braden, D. S. E. Deering, S. Floyd, D. B. S. Davie, V. Jacobson, and D. D. Estrin, “Recommendations on Queue Management and Congestion Avoidance in the Internet,” RFC 2309, Apr. 1998. [Online]. Available: <https://www.rfc-editor.org/info/rfc2309>
- [7] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.
- [8] K. M. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, “Controlled Delay Active Queue Management,” RFC 8289, Jan. 2018. [Online]. Available: <https://www.rfc-editor.org/info/rfc8289>
- [9] L. Kleinrock and R. Gail, “An invariant property of computer network power,” *IEEE International Conference on Communications*, vol. 3, pp. 63.1.1–63.1.5, 1981. [Online]. Available: <https://www.lk.cs.ucla.edu/data/files/Gail/power.pdf>
- [10] L. Torvalds. Linux kernel. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git>
- [11] T. Høiland-Jørgensen, P. McKenney, dave.taht@gmail.com, J. Gettys, and E. Dumazet, “The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm,” RFC 8290, Jan. 2018. [Online]. Available: <https://www.rfc-editor.org/info/rfc8290>
- [12] CoDel Overview. [Online]. Available: <https://www.bufferbloat.net/projects/codel/wiki/>
- [13] Cake - Common Applications Kept Enhanced. [Online]. Available: <https://www.bufferbloat.net/projects/codel/wiki/Cake/>
- [14] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg, “Pie: A lightweight control scheme to address the bufferbloat problem,” in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, 2013, pp. 148–155.

- [15] R. Pan, P. Natarajan, F. Baker, and G. White, “Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem,” RFC 8033, Feb. 2017. [Online]. Available: <https://www.rfc-editor.org/info/rfc8033>
- [16] M. P. Tahiliani, “Flow Queue PIE: A Hybrid Packet Scheduler and Active Queue Management Algorithm,” Internet Engineering Task Force, Internet-Draft draft-tahiliani-tsvwg-fq-pie-01, Mar. 2025, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/draft-tahiliani-tsvwg-fq-pie/01/>
- [17] H. H. Liu, *Software Performance and Scalability: A Quantitative Approach*. Hoboken, New Jersey: John Wiley & Sons, Ltd, 2009.
- [18] L. Schrage, “A proof of the optimality of the shortest remaining processing time discipline,” *Operations Research*, vol. 16, no. 3, pp. 687–690, 1968. [Online]. Available: <https://www.jstor.org/stable/168596>
- [19] A. Horváth and M. Telek, “Phfit: A general phase-type fitting tool,” in *Computer Performance Evaluation: Modelling Techniques and Tools*, T. Field, P. G. Harrison, J. T. Bradley, and U. Harder, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 82–91.
- [20] A. Marin, S. Rossi, and C. Zen, “A matlab toolkit for the analysis of two-level processor sharing queues,” in *Quantitative Evaluation of Systems*, M. Gribaudo, D. N. Jansen, and A. Remke, Eds. Cham: Springer International Publishing, 2020, pp. 144–147.
- [21] Best Practices for Benchmarking CoDel and FQ CoDel (and almost any other network subsystem!). [Online]. Available: https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel/
- [22] P. Heist. Antler. [Online]. Available: <https://github.com/heistp/antler>
- [23] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.