Three staggered-start TCP flows through PIE, codel and fq_codel bottleneck for TEACUP v0.4 testbed

Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 140630A
Swinburne University of Technology
Melbourne, Australia
garmitage@swin.edu.au

***Abstract*—This technical report summarises a basic set of staggered-start three-flow TCP experiments using pfifo (tail-drop), PIE, codel and fq_codel queue management schemes at the bottleneck router running Linux kernel 3.10.18. The goal is to illustrate plausible operation of our TEACUP testbed under teacup-v0.4.x using NewReno (FreeBSD) and CUBIC (Linux). Over a 10Mbps bottleneck and 20ms RTT path we observe that induced queuing delays drop dramatically (without significant impact on achieved throughput) when using PIE, codel or fq_codel at their default settings. Hashing of flows into separately scheduled queues allows fq_codel to provide smoother capacity sharing. Over a 200ms RTT path the three AQM schemes exhibited some drop-off in throughput relative to pfifo, suggesting a need to explore non-default settings in high RTT environments. The impact of ECN is a matter for future work. This report does *not* attempt to make any meaningful comparisons between the tested TCP algorithms, nor rigorously evaluate the consequences of using PIE, codel or fq_codel over a range of RTTs.**

***Index Terms*—TCP, codel, fq_codel, PIE, pfifo, experiments, testbed, TEACUP**

I. Introduction

CAIA has developed TEACUP[1] [1] to support a comprehensive experimental evaluation of different TCP congestion control algorithms. Our actual testbed is described in [2]. This report summarises a basic set of staggered-start three-flow TCP experiments that illustrate the impact of pfifo (tail-drop), PIE [3], codel [4] and fq_codel [5] queue management schemes at the bottleneck router.

The trials use six hosts to create three competing flows partially overlapping in time. The bottleneck in each case is a Linux-based router using `netem` and `tc` to provide independently configurable bottleneck rate limits and artificial one-way delay (OWD). The trials run over a small range of emulated path conditions using FreeBSD NewReno and Linux CUBIC. ECN is *not* used.

Over a path with 20ms base RTT and 10Mbps bottleneck we observe that using PIE, codel or fq_codel at their default settings provides a dramatic drop in overall RTT (without significant impact on achieved throughput) relative to using a pfifo bottleneck. The hashing of flows into separately managed queues allows fq_codel to provide smoother capacity sharing than either PIE or codel. When the path RTT rises to 200ms all three AQM schemes exhibited some drop-off in throughput relative to pfifo, suggesting a need to explore non-default settings in high RTT environments.

This report does *not* attempt to make any meaningful comparisons between the tested TCP algorithms, nor do we explore the differences between PIE, codel and fq_codel in any significant detail. The potential impact of ECN is also a subject for future work.

The rest of the report is organised as follows. Section II summarises the the testbed topology, physical configurations and emulated path characteristics for these trials. For similar paths and combinations of AQM, we look at the behaviour of three FreeBSD NewReno flows in Section III and three Linux CUBIC flows in Section IV. Section V concludes and outlines possible future work.

II. Testbed topology and test conditions

Trials involved three concurrent TCP connections each pushing data through a single bottleneck for 60 seconds. The path has different emulated delays, bottleneck speeds and AQM algorithms. Here we document the testbed topology, operating systems, TCP algorithms and path conditions.

[1] TCP Experiment Automation Controlled Using Python.

A. Hosts and router

Figure 1 (from [2]) shows a logical picture of the testbed's networks[2]. The router provides a configurable bottleneck between three hosts on network 172.16.10.0/24 and three hosts on network 172.16.11.0/24. In this report, hosts on 172.16.10/24 send traffic to hosts on 172.16.11/24.

The bottleneck router runs 64-bit Linux (openSUSE 12.3 with kernel 3.10.18 patched to run at 10000 Hz). Physically the router is a Supermicro X8STi motherboard with 4GB RAM, 2.80GHz Intel® Core™ i7 CPU, 2 x Intel 82576 Gigabit NICs for test traffic and 2 x 82574L Gigabit NICs for control traffic.

Each host is a triple-boot machine that can run 64-bit Linux (openSUSE 12.3 with kernel 3.9.8 and web10g patch [6]), 64-bit FreeBSD (FreeBSD 9.2-RELEASE #0 r255898) or 64-bit Windows 7 (with Cygwin 1.7.25 for unix-like control of the host). Physically each host is a HP Compaq dc7800, 4GB RAM, 2.33GHz Intel Core2 Duo CPU, Intel 82574L Gigabit NIC for test traffic and 82566DM-2 Gigabit NIC for control traffic.

See [2] for more technical details of how the router and each host was configured.

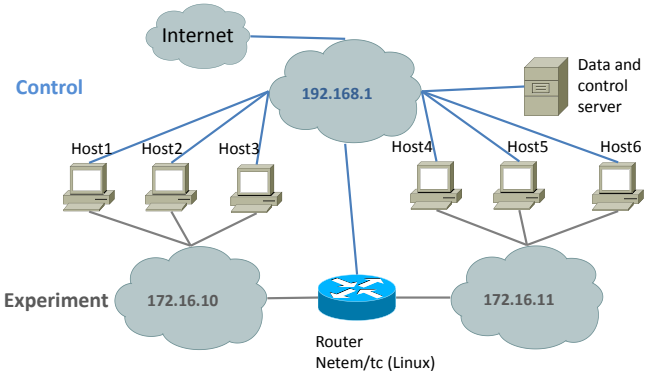


Figure 1: Testbed overview

B. Host operating system and TCP combinations

The trials used used two different operating systems and TCP algorithms.

- FreeBSD: Newreno
- Linux: CUBIC

[2]Each network is a switched gigabit ethernet VLAN on a single Dell PowerConnect 5324.

See Appendices A and B for details of the TCP stack configuration parameters used for the FreeBSD and Linux end hosts respectively.

C. Emulated path and bottleneck conditions

The bottleneck router uses `netem` and `tc` to concatenate an emulated path with specific one way delay (OWD) and an emulated bottleneck whose throughput is limited to a particular rate. Packets sit in a 1000-packet buffer while being delayed (to provide the artificial OWD), then sit in a separate "bottleneck buffer" of configurable size (in packets) while being rate-shaped to the bottleneck bandwidth.

1) Bottleneck AQM: We repeated each trial using the Linux 3.10.18 kernel's implementations of pfifo, PIE, codel and fq_codel algorithms in turn to manage the bottleneck buffer (queue) occupancy. To explore the impact of actual available buffer space, we set the total buffer size to either 180 packets[3] or 2000 packets.

As noted in Section V.D of [2], we compiled PIE into the 3.10.18 kernel from source [7] dated July 2nd 2013,[4] and included a suitably patched iproute2 (v3.9.0) [8].

As they are largely intended to require minimal operator control or tuning, we used PIE, codel and fq_codel at their default settings.[5] For example, when configured for a 180-packet buffer, the relevant qdisc details were:

PIE:

```
limit 180p target 20 tupdate 30
alpha 2 beta 20
```

codel:

```
limit 180p target 5.0ms interval 100.0ms
```

fq_codel:

```
limit 180p flows 1024 quantum 1514
target 5.0ms interval 100.0ms
```

See Appendix C for more details on the bottleneck router's AQM configuration.

[3] Significantly lower than the default codel and fq_codel buffer sizes of 1000 and 10000 packets respectively

[4]MODULE_INFO(srcversion, "1F54383BFCB1F4F3D4C7CE6")

[5]Except buffer size overridden by individual trial conditions.

2) Path conditions: This report covers the following emulated path and bottleneck conditions:

- 0% intrinsic loss rate[6]
- One way delay: 10 and 100ms
- Bottleneck bandwidth: 10Mbps
- Bottleneck buffer sizes: 180 and 2000 pkts
- ECN disabled on the hosts

These conditions were applied bidirectionally, using separate delay and rate shaping stages in the router for traffic in each direction. Consequently, the path's intrinsic (base) RTT is always twice the configured OWD. The 180 and 2000-packet bottleneck buffers were greater than the path's intrinsic BDP (bandwidth delay product).

D. Traffic generator and logging

Each concurrent TCP flow was generated using iperf 2.0.5 [9] on both OSes, patched to enable better control of the send and receive buffer sizes [10]. For each flow, iperf requests 600Kbyte socket buffers to ensure `cwnd` growth was not significantly limited by each destination host's maximum receive window.

Each 60-second flow was launched 20 seconds apart between the following host pairs:

- Host1→Host4 at $t = 0$
- Host2→Host5 at $t = 20$
- Host3→Host6 at $t = 40$

Data packets from all flows traverse the bottleneck router in the same direction. TCP connection statistics were logged using SIFTR [11] under FreeBSD and Web10g [6] under Linux. Packets captured at both hosts with tcpdump were used to calculate non-smoothed end to end RTT estimates using CAIA's passive RTT estimator, SPP [12], [13].

E. Measuring throughput

'Instantaneous' throughput is an approximation derived from the actual bytes transferred during constant (but essentially arbitrary) windows of time. Long windows smooth out the effect of transient bursts or gaps in packet arrivals. Short windows can result in calculated throughput that swings wildly (but not necessarilly meaningfully) from one measurement interval to the next. For this report we use a window two seconds wide, sliding forward in steps of 0.5 second.

[6]No loss beyond that induced by bottleneck buffer congestion.

III. Three NewReno flows, AQM, no ECN

This section illustrates how varying the AQM influences the observed throughput and overall RTT versus time when three NewReno flows share the bottleneck. ECN is disabled for these trials.

A. Throughput, `cwnd` and RTT versus time – two runs with pfifo queue management

By way of introduction we first review the result of three NewReno flows sharing a 10Mbps bottleneck using *pfifo* queue management, a 180-packet bottleneck buffer and either a 20ms RTT path or a 200ms RTT path. The main message here is that in real testbeds no two trial runs will be identical.

1) Three flows over a 20ms RTT path: Figure 2 illustrates how three NewReno flows behave during two runs over a 20ms RTT (10ms OWD) path.

Figures 2a (throughput vs time) and 2c (`cwnd` vs time) show the three flows very crudely sharing the botteneck capacity during periods of overlap in the first run. Flows 1 and 2 share *somewhat* equally from $t = 20$ to $t = 40$, but the sharing becomes noticeably unequal once Flow 3 joins at $t = 40$.

Figures 2b and 2d show a broadly similar chain of events in the repeat (second) run, with some differences. During the period from $t = 40$ to $t = 80$ Flow 2 has a greater share of overall capacity than it did in the first run.

Queuing delays impact on all traffic sharing the dominant congested bottleneck. Figures 2e and 2f show the RTT cycling between ~100ms and ~240ms, with the RTT experienced by each flow tracking closely over time within each run.[7]

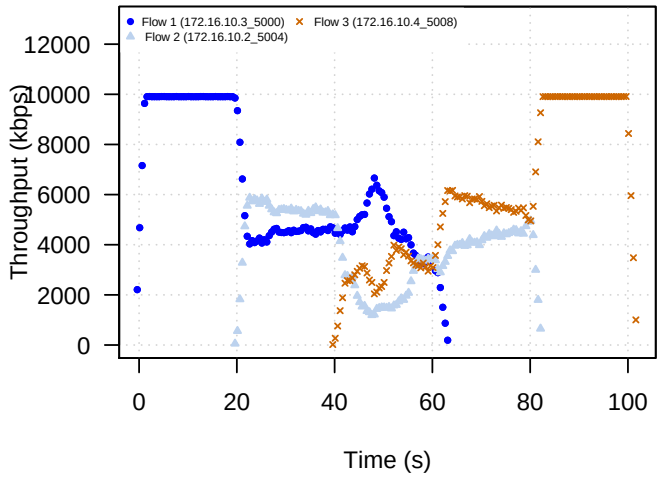
2) Three flows over a 200ms RTT path: Figure 3 repeats the previous experiment but this time with a 200ms RTT. As with the 20ms case, capacity sharing is crude to the point of being quite unequal.

Figures 3a and 3b show that Flow 1 is still able to reach full capacity relatively quickly before Flow 2 starts, and

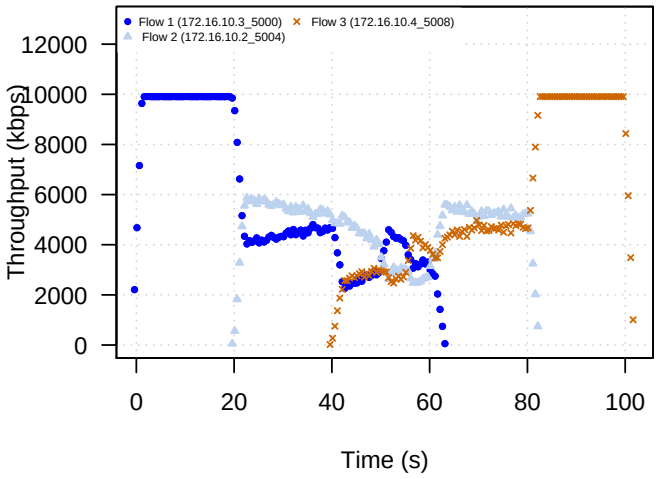
[7]At 10Mbps the 180-packet buffer full of 1500-byte TCP Data packets adds ~216ms to the OWD in the Data direction. The return (ACK) path's buffer is mostly empty, adding little to the RTT.

Flow 3 is (mostly) able to utilise the path's full capacity once Flows 1 and 2 cease.

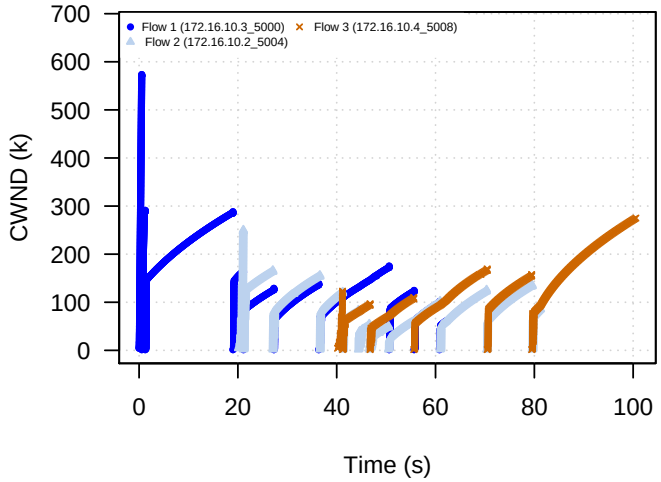
Relative to Figure 2, the impact of a 200ms base RTT is evident in the slower periodic cycling of `cwnd` vs time (Figures 3c and 3d) and total RTT vs time (Figures 3e and 3f). In this case, `cwnd` must rise to a far higher value to 'fill the pipe' and queuing delays increase RTT by up to ∼220ms over the base 200ms.



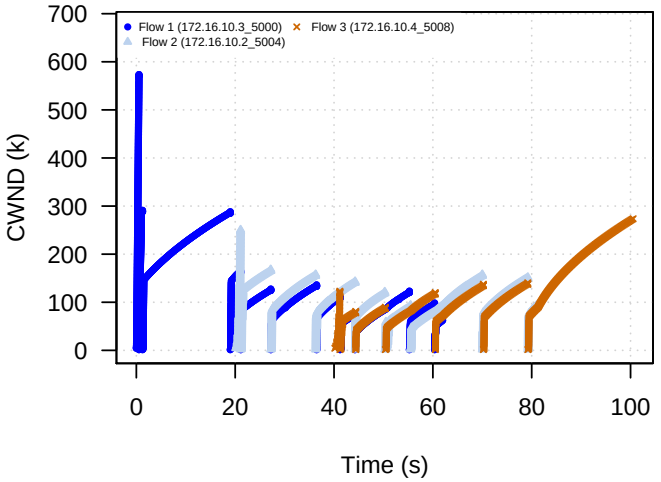
(a) Throughput vs time – first run



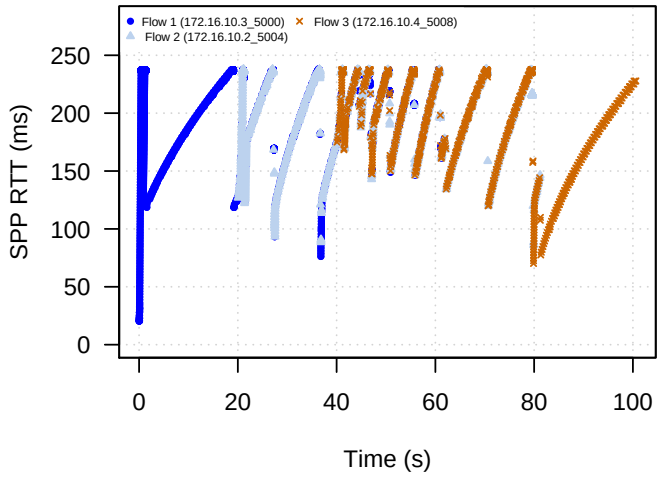
(b) Throughput vs time – second run



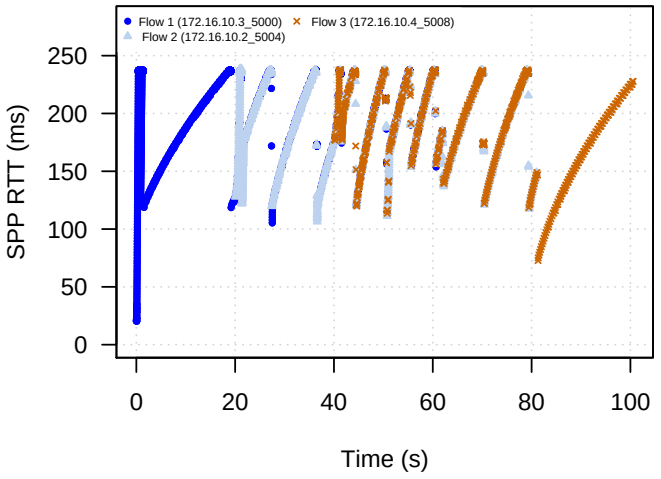
(c) `cwnd` vs time – first run



(d) `cwnd` vs time – second run

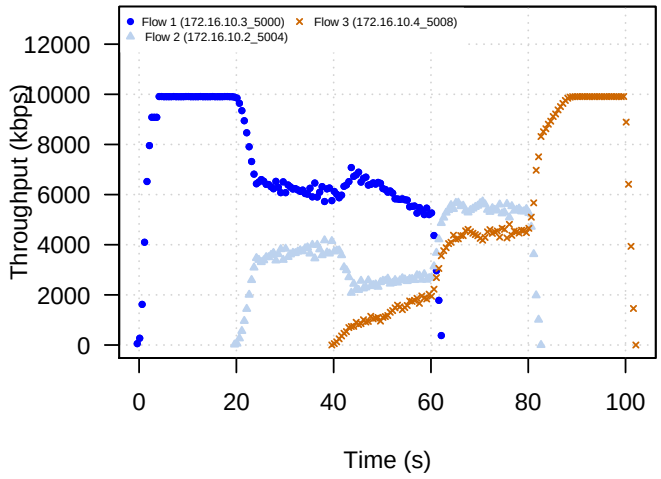


(e) Total RTT vs time – first run

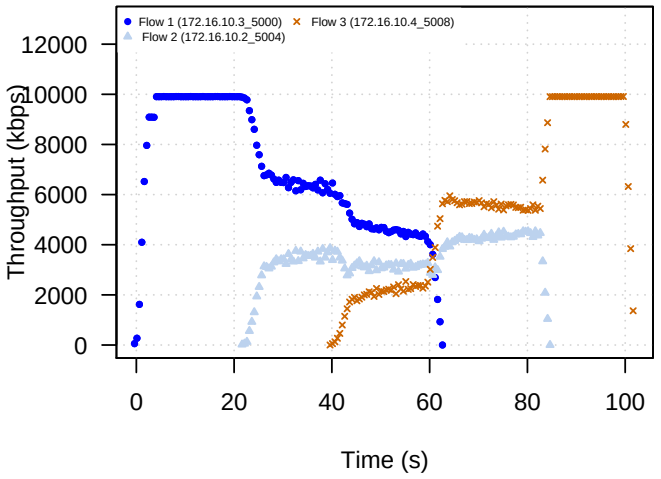


(f) Total RTT vs time – second run

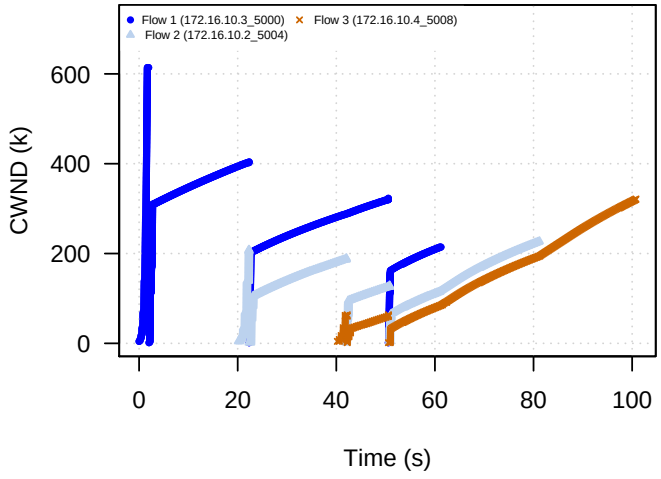
Figure 2: Two separate runs of three overlapping FreeBSD NewReno flows over a *20ms* RTT path with 10Mbps rate limit and 180-packet *pfifo* bottleneck buffer (no ECN).



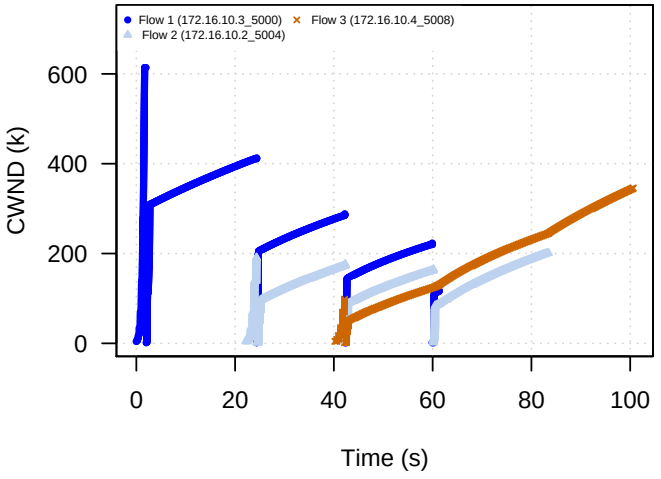
(a) Throughput vs time – first run



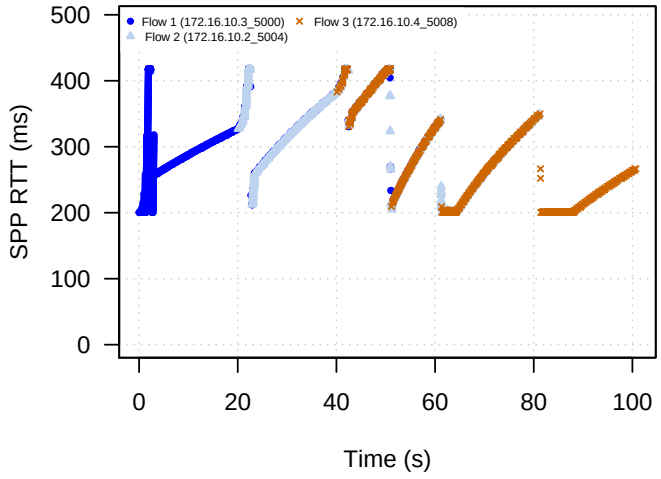
(b) Throughput vs time – second run



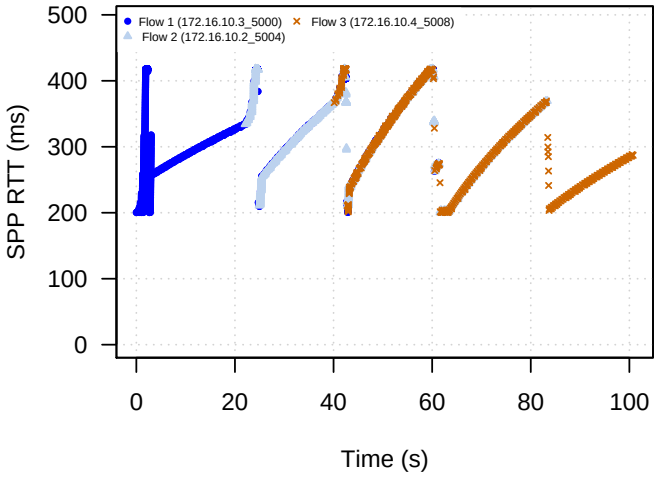
(c) `cwnd` vs time – first run



(d) `cwnd` vs time – second run



(e) Total RTT vs time – first run



(f) Total RTT vs time – second run

Figure 3: Two separate runs of three overlapping FreeBSD NewReno flows over a *200ms* RTT path with 10Mbps rate limit and 180-packet *pfifo* bottleneck buffer (no ECN).

B. Throughput and RTT versus time – one NewReno run with PIE, codel and fq_codel

Next we look at the impact of PIE, codel and fq_codel on flow behaviour over time.[8] First we look at a path having a 20ms RTT, a 10Mbps bottleneck rate limit and either a 180-packet or 2000-packet bottleneck buffer. Then we look at the same path with a 200ms RTT and 180-packet bottleneck buffer. Relative to the pfifo case (Figures 2e and 2f), all three AQM schemes provide better capacity sharing with significantly reduced bottleneck queuing delays.

1) Using a 180-packet bottleneck buffer @ 20ms RTT: Through the PIE bottleneck, Figure 4a show all three flows sharing broadly equally during periods of overlap, with moderate fluctuations over time. A similar result is observed in Figure 4c when traversing a codel bottleneck. In contrast, Figure 4e shows that using an fq_codel bottleneck results in each flow achieving quite consistent throughput and balanced sharing.

Figures 4b, 4d and 4f illustrate the RTT experienced by each flow over time using PIE, codel and fq_codel respectively. Ignoring the brief spikes induced during slow-start as Flows 2 and 3 begin, PIE sees a somewhat wider range of RTT than codel or fq_codel (within ∼30ms of base RTT for PIE and within ∼10–15ms of base RTT for codel and fq_codel).

2) Using a 2000-packet bottleneck buffer @ 20ms RTT: Section III-B1's use of a 180-packet buffer is significantly lower than the default codel and fq_codel buffer sizes of 1K and 10K packets respectively. Figure 5 shows a repeat of the trials in Section III-B1 (Figure 4), but now with the bottleneck buffer bumped up from 180 to 2000 packets to look for any differences.

Visual inspection shows that overall the results are essentially the same. PIE, codel and fq_codel all meet their goals of keeping the RTTs low despite the significantly higher bottleneck buffer space (and with much the same range of absolute values as in Figure 4) . Throughput and capacity sharing also appears reasonably similar to what was achieved with a 180-packet buffer.

3) Using 180-packet bottleneck buffers @ 200ms RTT: Figure 6 show the performance over time when the path's base RTT is bumped up to 200ms.[9] Total RTT in Figures 6b, 6d and 6f is not unlike the 20ms case – all show RTT spiking at the start of each flow, then (mostly) stabilising much closer to the path's base RTT.

Figures 6a, 6c and 6e show an interesting result relative to the pfifo case in Figure 3. In all three cases Flow 1 struggles to reach full path capacity during its first 20 seconds, and Flow 3 struggles during the final 20 seconds. A likely explanation can be found in Figures 7a, 7b and 7c (`cwnd` vs time) – Flow 1 switches from slow-start (SS) to congestion avoidance (CA) mode at a much lower `cwnd` than was the case with pfifo.

Using PIE, Flow 1's `cwnd` initially spikes to ∼600kB, but this is quickly followed by two drops, resulting in CA mode taking over at ∼150kB (much lower than the switch to CA mode at ∼300kB in Figure 3c).[10] In both the codel and fq_codel cases Flow 1's `cwnd` rises to ∼270kB before being hit with a drop and switching to CA mode at ∼135kB.

4) Discussion: Neither PIE nor codel can eliminate interactions between the flows, as they manage all flows in a common queue. Nevertheless they are clearly better than pfifo in terms of reducing the queuing delays, keeping RTT low for both 180-packet and 2000-packet bottleneck buffers.

The behaviour with fq_codel is a little more complex. Individual flows are hashed into one of 1024 different internal queues, which are serviced by a modified form of deficit round robin scheduling. Codel is used to manage the sojourn times of packets passing through each internal queue. Figure 4e reflects the fact that each flow was mapped into a separate codel-managed queue, and hence received an approximately equal share of the bottleneck bandwidth.[11]

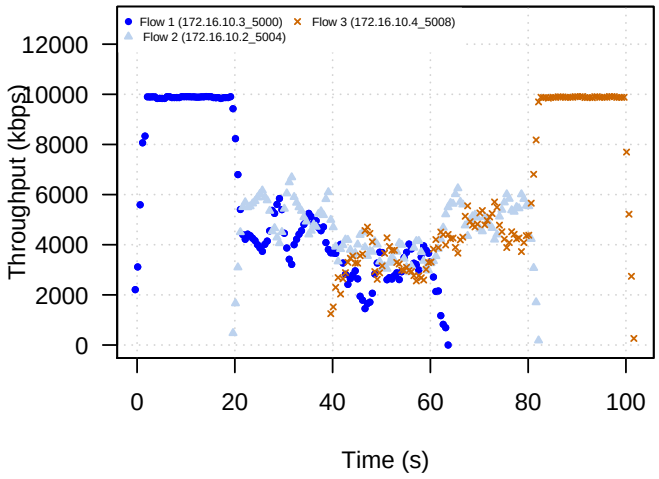
Figure 6 illustrates the potential value of tuning AQM parameters for paths having high RTTs. Although Figure 7 showed each AQM keeping total RTTs low, throughput clearly takes a hit due to the AQMs triggering a switch from SS to CA modes 'too early'.

[8] Keeping in mind that detailed evaluations of PIE, codel and fq_codel are a matter for future study.

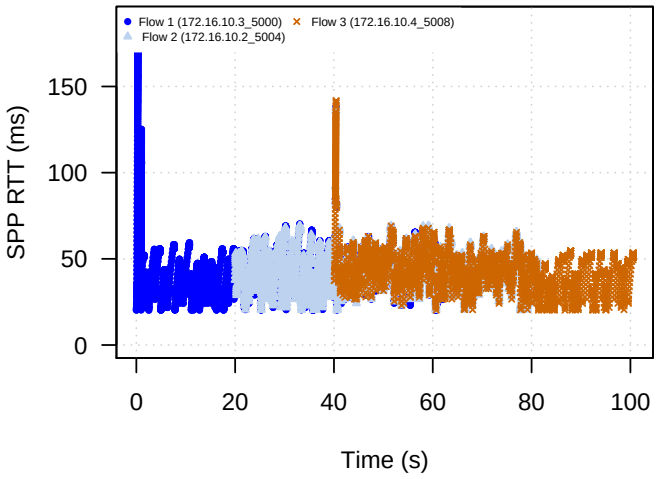
[9] Noting that codel's defaults are set assuming a 100ms RTT [4].

[10] The same double-drop was observed in a 2nd run of this trial.

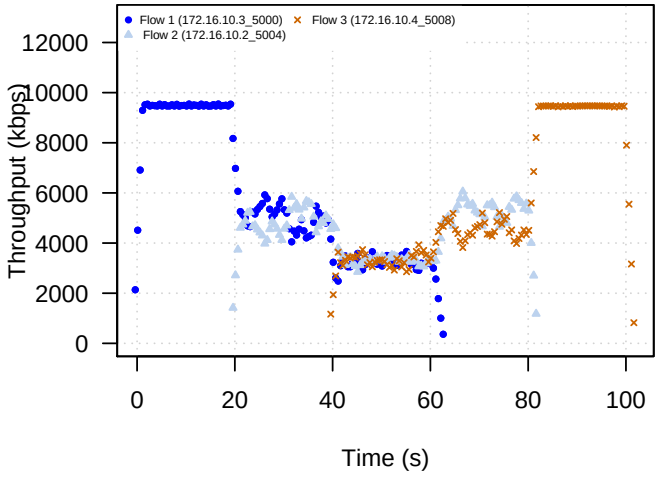
[11] The impact of different flows being hashed into the same internal queue is a matter for future study



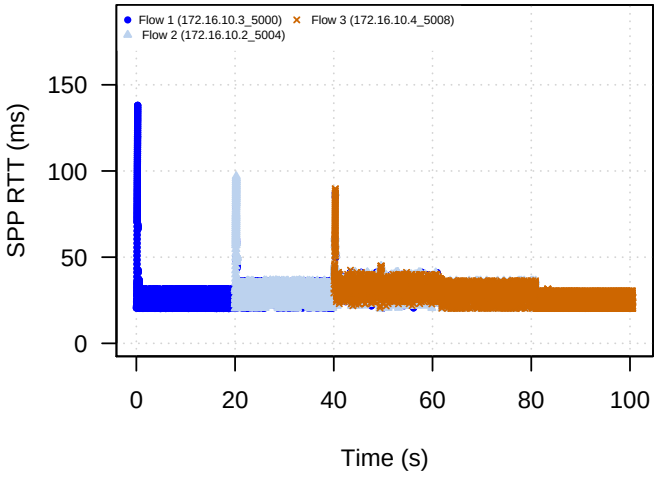
(a) Throughput vs time – PIE



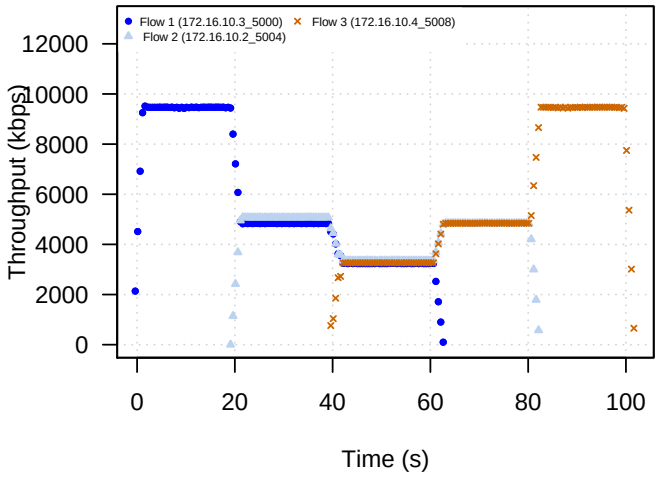
(b) Total RTT vs time – PIE



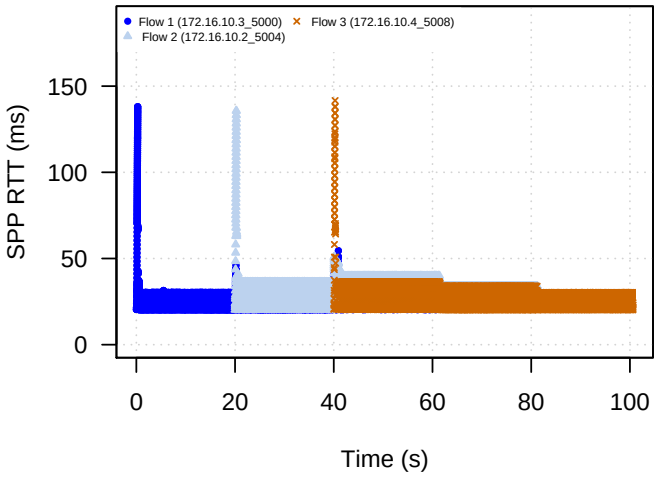
(c) Throughput vs time – codel



(d) Total RTT vs time – codel

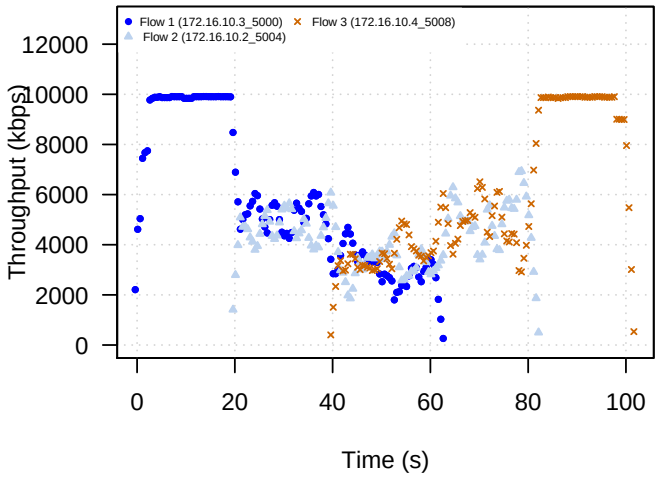


(e) Throughput vs time – fq_codel

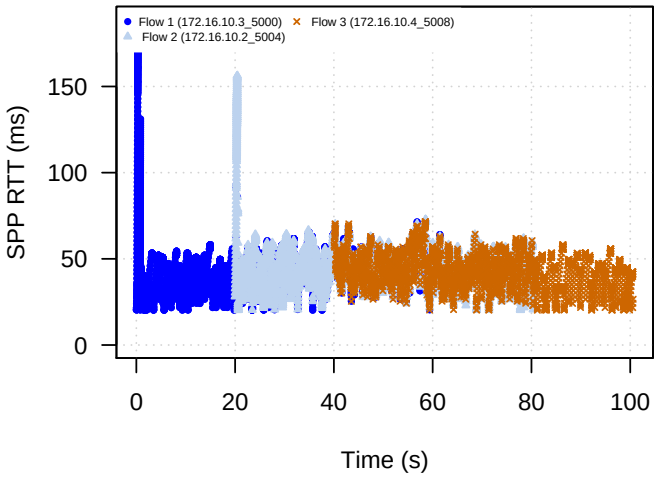


(f) Total RTT vs time – fq_codel

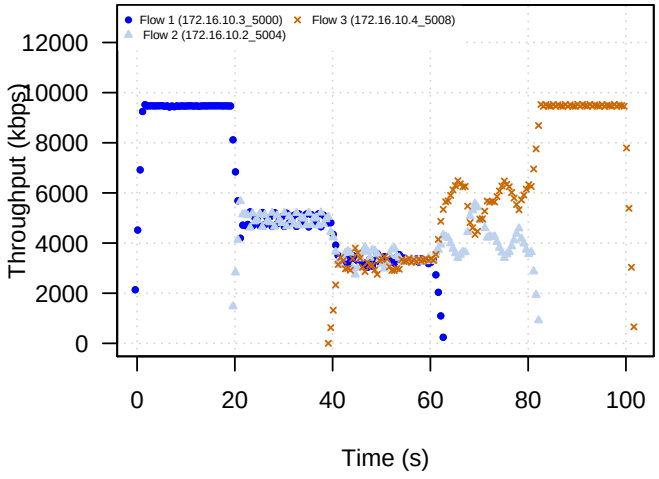
Figure 4: Three overlapping FreeBSD NewReno flows over a *20ms* RTT path with 10Mbps rate limit and 180-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel* (no ECN)



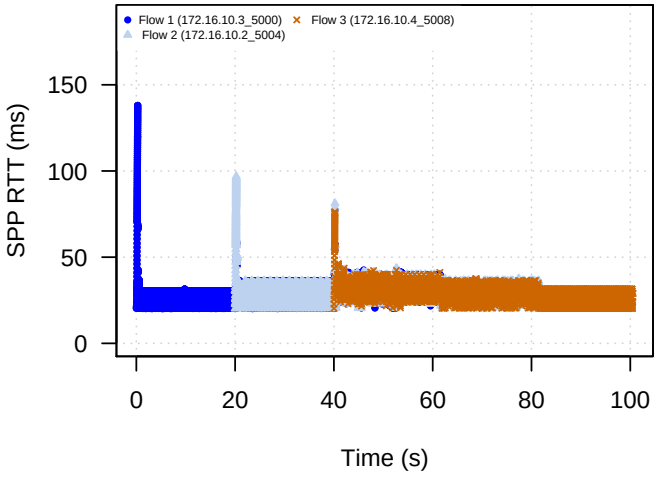
(a) Throughput vs time – PIE



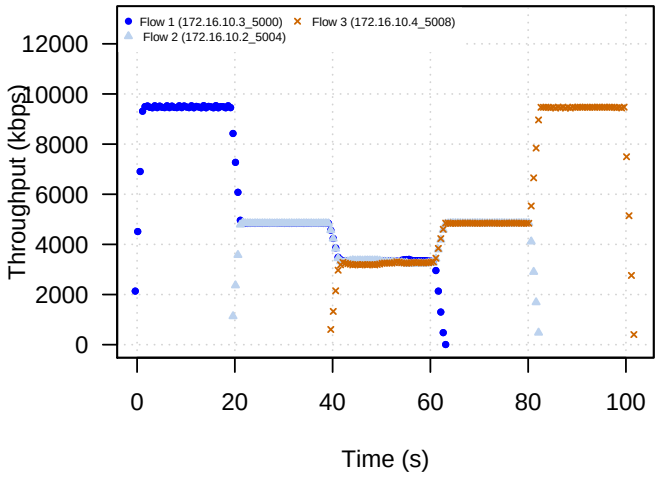
(b) Total RTT vs time – PIE



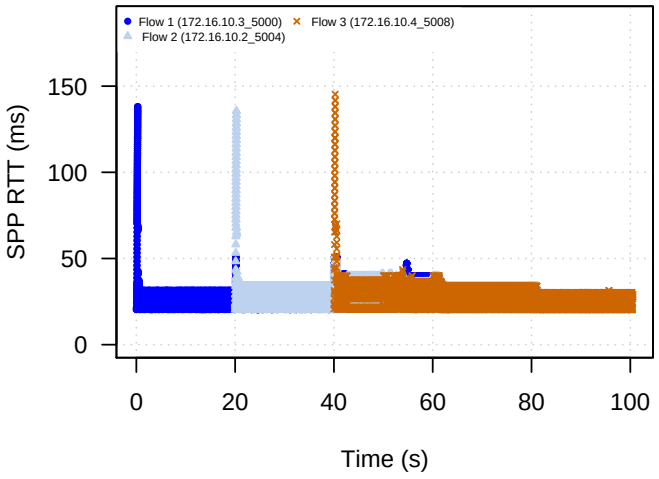
(c) Throughput vs time – codel



(d) Total RTT vs time – codel

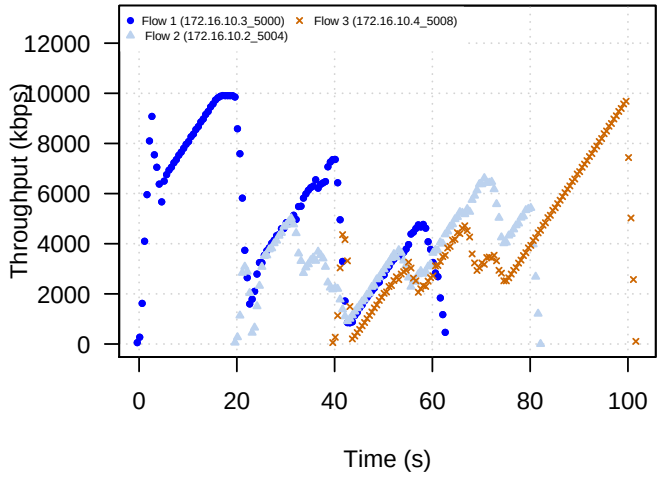


(e) Throughput vs time – fq_codel

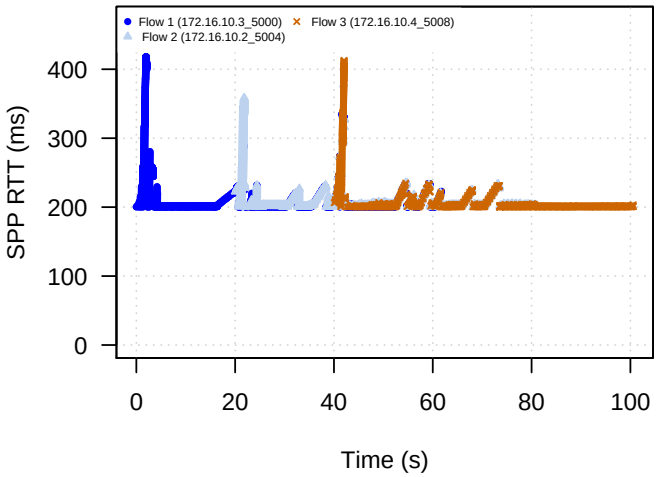


(f) Total RTT vs time – fq_codel

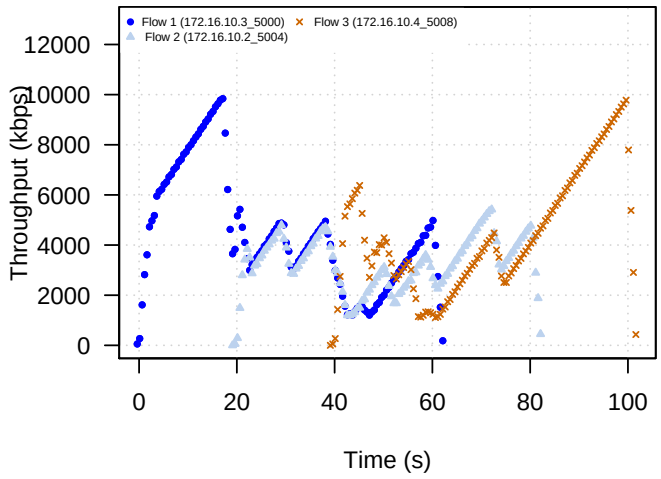
Figure 5: Three overlapping FreeBSD NewReno flows over a *20ms* RTT path with 10Mbps rate limit and 2000-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel* (no ECN)



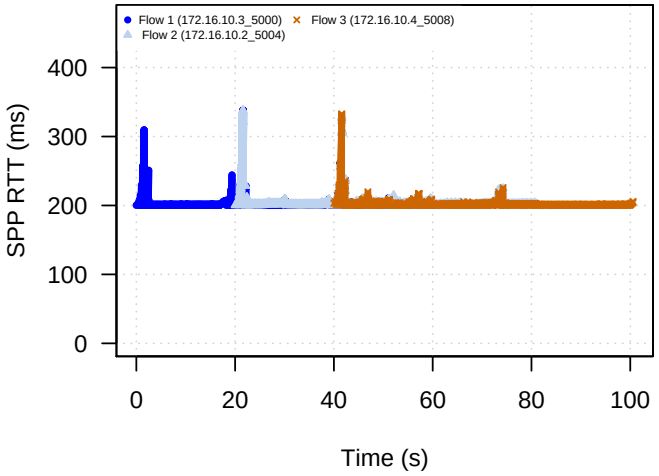
(a) Throughput vs time – PIE



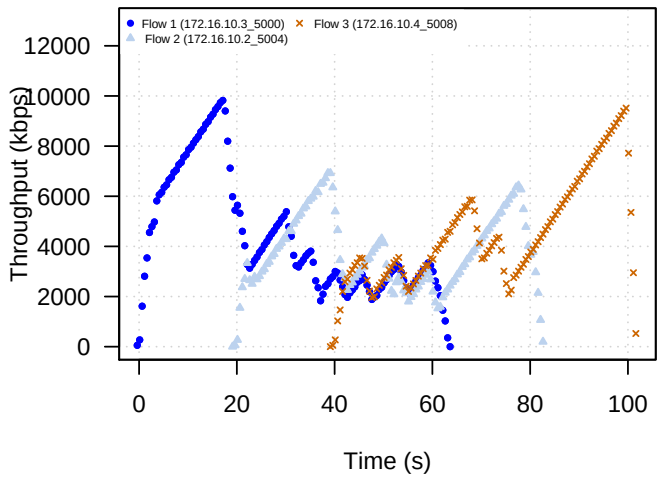
(b) Total RTT vs time – PIE



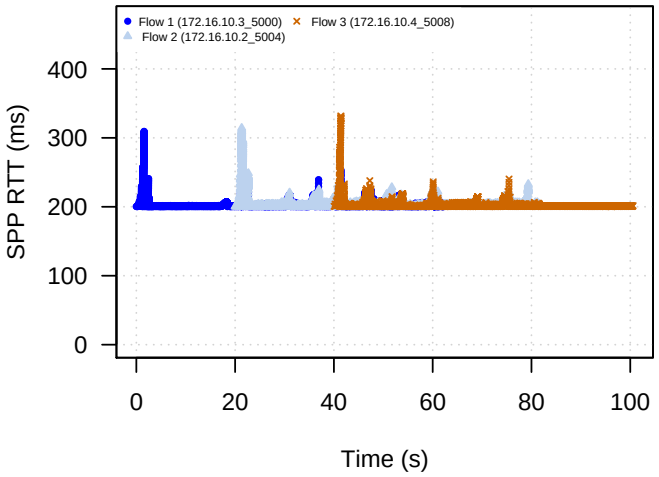
(c) Throughput vs time – codel



(d) Total RTT vs time – codel

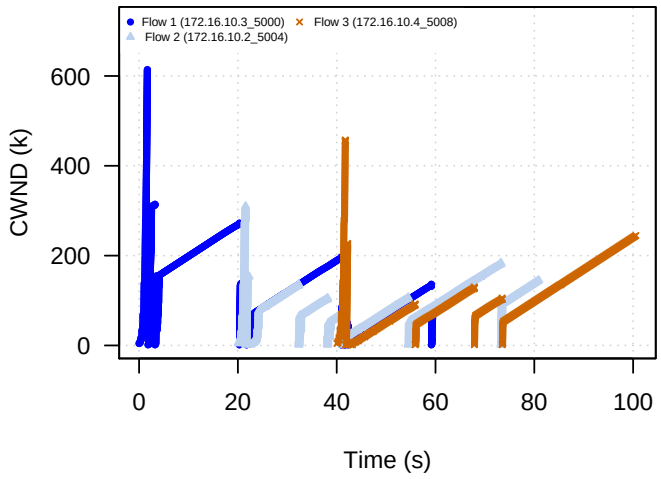


(e) Throughput vs time – fq_codel

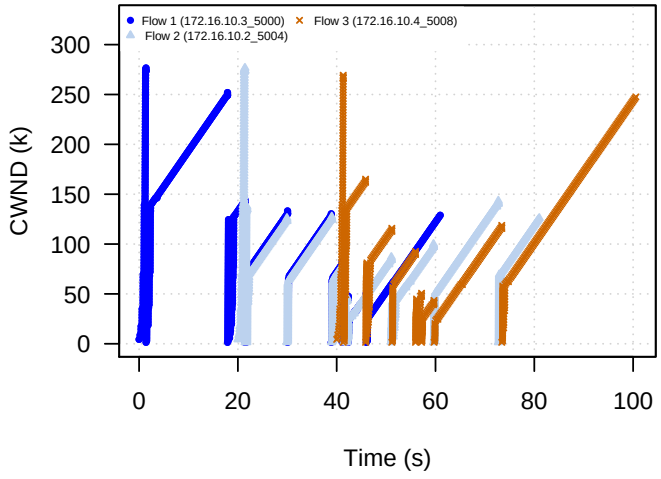


(f) Total RTT vs time – fq_codel

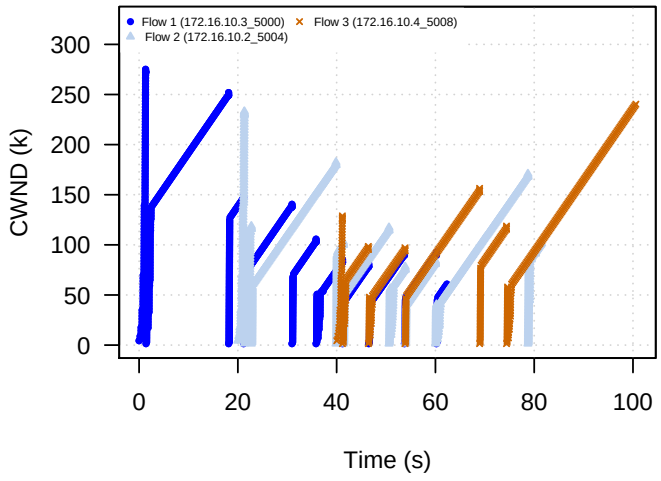
Figure 6: Three overlapping FreeBSD NewReno flows over a *200ms* RTT path with 10Mbps rate limit and 180-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel* (no ECN)



(a) cwnd vs time – PIE



(b) cwnd vs time – codel



(c) cwnd vs time – fq_codel

Figure 7: Three overlapping FreeBSD NewReno flows over a *200ms* RTT path at 10Mbps and 180-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel*

IV. THREE CUBIC FLOWS, AQM, NO ECN

This section illustrates how varying the AQM influences the observed throughput and overall RTT versus time when three CUBIC flows share the bottleneck. ECN is disabled for these trials.

A. Throughput, `cwnd` and RTT versus time – two runs with pfifo queue management

By way of introduction we first review the result of three CUBIC flows sharing a 10Mbps bottleneck using *pfifo* queue management, a 180-packet bottleneck buffer and either a 20ms RTT path or a 200ms RTT path. As with the NewReno trials, no two CUBIC trial runs are identical.

1) Three flows over a 20ms RTT path: Figure 8 illustrates how three CUBIC flows behave during two runs over a 20ms RTT (10ms OWD) path.

Figures 8a (throughput vs time) and 8c (`cwnd` vs time) show the three flows very poorly sharing the botteneck capacity during periods of overlap in the first run. Flows 1 dominates Flow 2 from $t = 20$ to $t = 40$, with Flow 2 only slowly gaining a share of bandwidth. The sharing remains noticeably unequal after Flow 3 joins at $t = 40$.

Figures 8b and 8d show a broadly similar chain of events in the repeat (second) run, differing only in small details (e.g. from $t = 40$ to $t = 80$ Flow 2 has a greater share of overall capacity than it did in the first run relative to Flow 3). No doubt a third run would differ again in the precise sequence of events.

Queuing delays impact on all traffic sharing the dominant congested bottleneck. Figures 8e and 8f show the RTT cycling up to ∼240ms, with the RTT experienced by each flow tracking closely over time within each run.

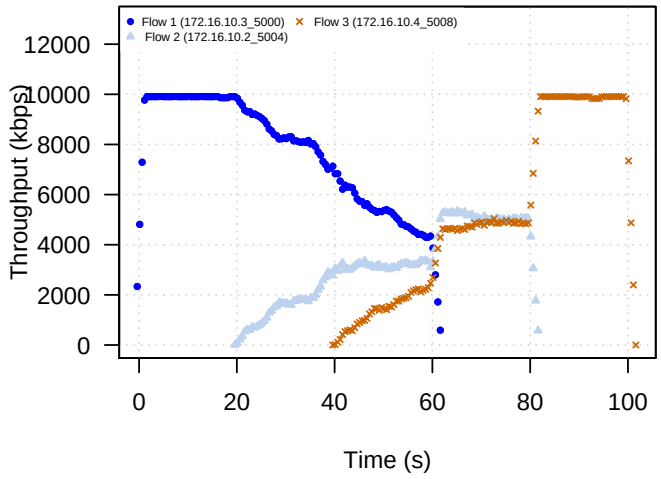
2) Three flows over a 200ms RTT path: Figure 9 repeats the previous experiment but this time with a 200ms RTT. As with the 20ms case, capacity sharing is crude to the point of being quite unequal. But we also see an interesting artefact of how Linux allocates buffer space at the receiver.

Figures 9a and 9b show that Flow 1 is still able to reach full capacity relatively quickly before Flow 2 starts, and Flow 3 is (mostly) able to utilise the path's full capacity once Flows 1 and 2 cease. However, Figures 9c and 9d reveal that `cwnd` now behaves a little differently to what we observed in Figure 8.

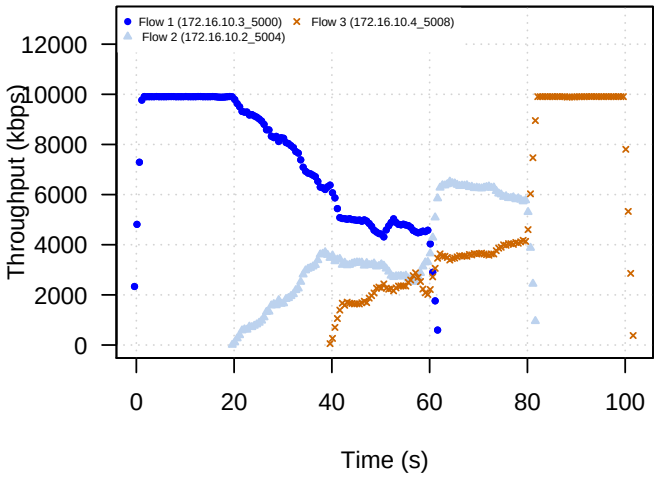
Although our patched iperf requests a 600kB receiver buffer, the Linux kernel advertises a smaller maximum receiver window. Consequently, from $t = 0$ to $t = 20$ Flow 1's `cwnd` rises quickly then stabilises at just over 400kB – a level sufficient to 'fill the pipe' (and achieve full path utilisation) but insufficient to overflow the bottleneck's pfifo buffer. Thus we do not see any loss-induced cyclical `cwnd` behaviour during this period. After $t = 80$ only Flow 3 remains, with its own `cwnd` similarly capped at a value high enough to fill the pipe but unable to over-fill the bottleneck buffer.

Between $t = 20$ and $t = 80$ the combination of flows means the aggregate traffic is now capable of overfilling the bottleneck buffer, leading to classic, cyclical `cwnd` behaviour. Figures 9e and 9f show the expected RTT variations during this time, as queuing delays increase RTT by up to ∼220ms over the base 200ms.

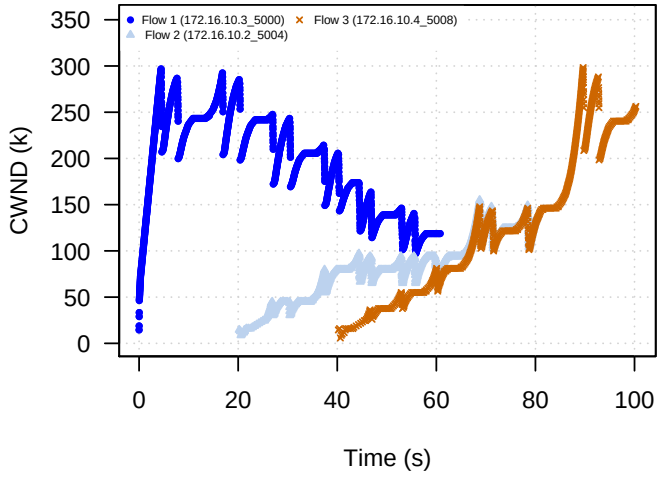
Closer inspection reveals that the RTT variations between $t = 0$ and $t = 20$ and after $t = 80$ are largely due to the source sending line-rate bursts towards the bottleneck. Figure 10 provides a close-up of the RTT in the first few seconds of Flow 1. Between $t = 0$ and $t = 1.5$ we see six bursts of RTT ramping up from ∼200ms to increasingly higher values during the initial growth of `cwnd`. These occur once every 200ms. Then every ∼116ms from roughly $t = 2.0$ onwards the RTT repeatedly ramps up from ∼228ms to ∼338ms.



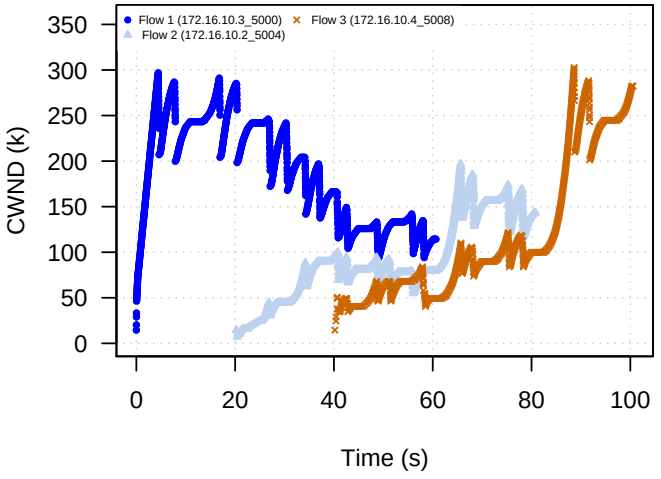
(a) Throughput vs time – first run



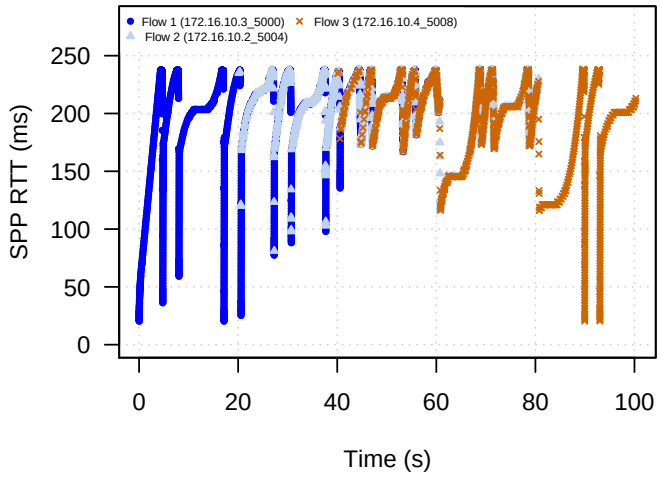
(b) Throughput vs time – second run



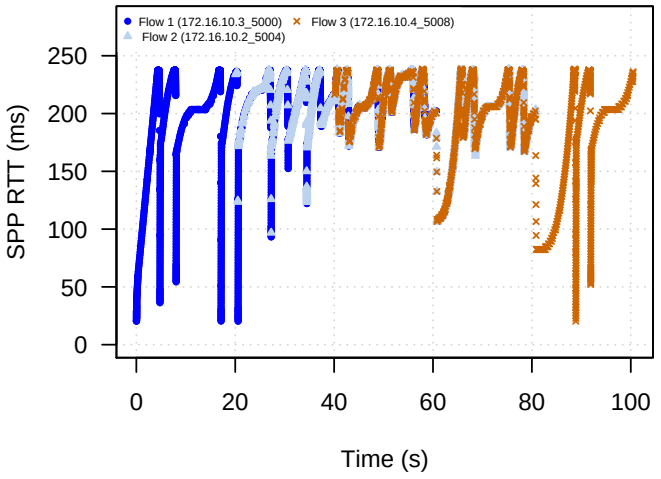
(c) `cwnd` vs time – first run



(d) `cwnd` vs time – second run

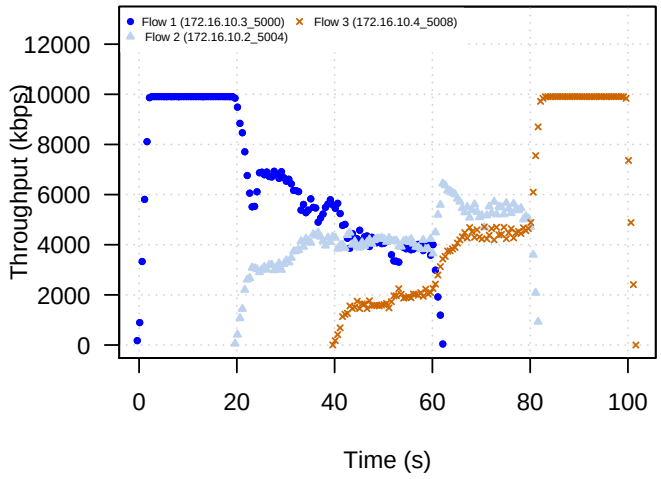


(e) Total RTT vs time – first run

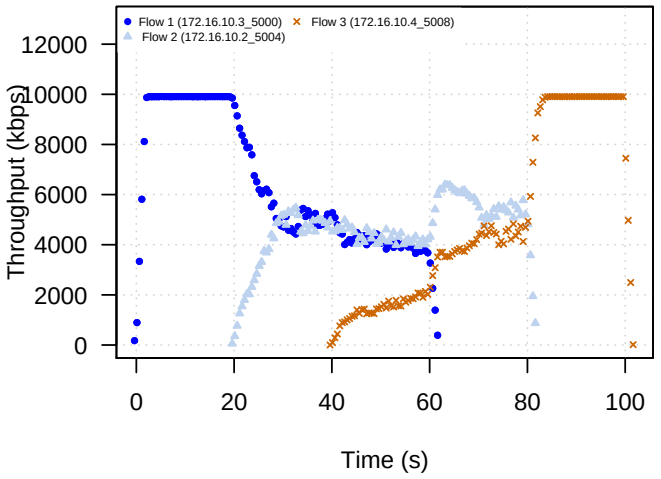


(f) Total RTT vs time – second run

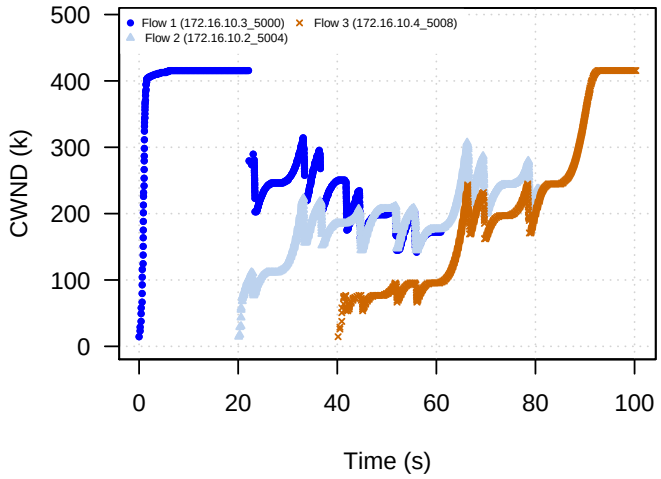
Figure 8: Two separate runs of three overlapping Linux CUBIC flows over a *20ms* RTT path with 10Mbps rate limit and 180-packet *pfifo* bottleneck buffer (no ECN).



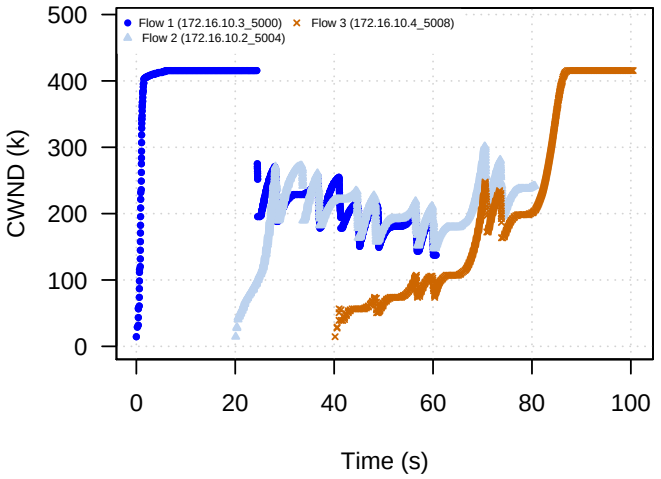
(a) Throughput vs time – first run



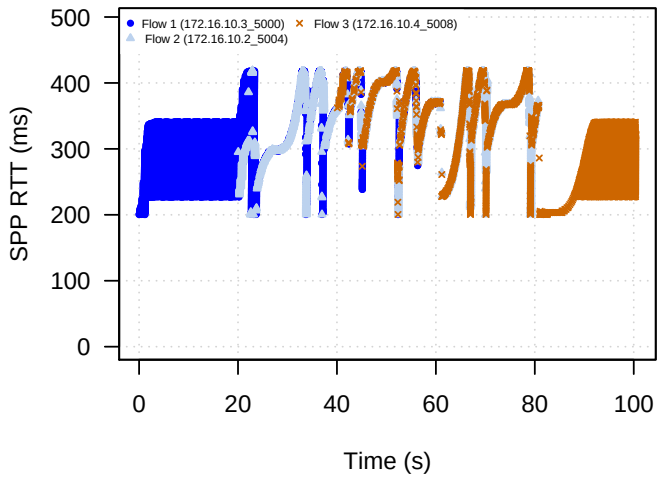
(b) Throughput vs time – second run



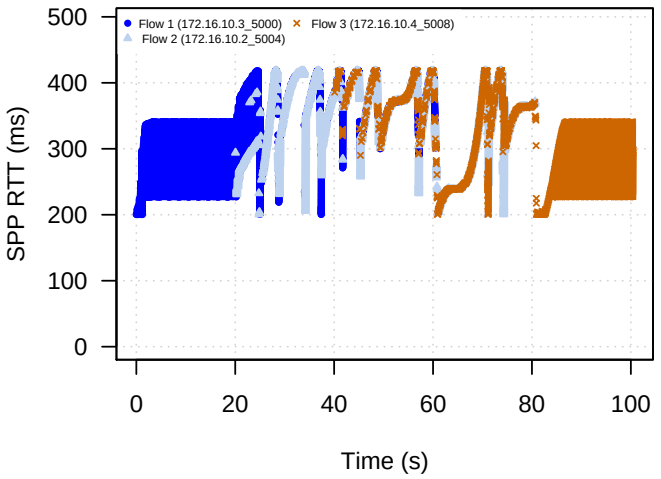
(c) `cwnd` vs time – first run



(d) `cwnd` vs time – second run



(e) Total RTT vs time – first run



(f) Total RTT vs time – second run

Figure 9: Two separate runs of three overlapping Linux CUBIC flows over a *200ms* RTT path with 10Mbps rate limit and 180-packet *pfifo* bottleneck buffer (no ECN).

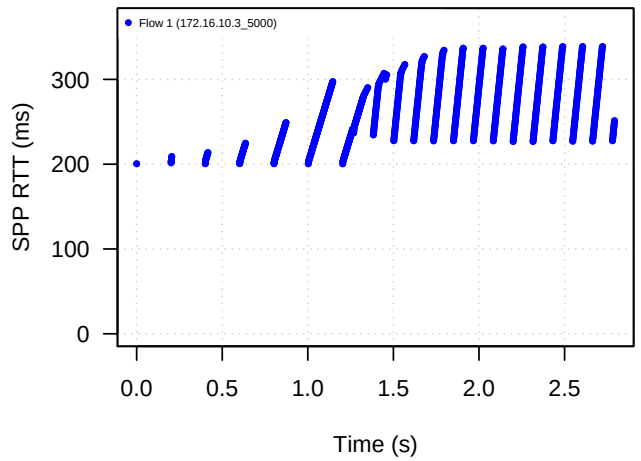


Figure 10: Closeup of Figure 9e from $t = 0$ to $t = 2.5$ (Flow 1 alone, not overfilling the bottleneck's buffer)

B. Throughput and RTT versus time – one CUBIC run with PIE, codel and fq_codel

Next we look at the impact of PIE, codel and fq_codel on flow behaviour over time in Figure 11. Again, the path has a base 20ms RTT (10ms OWD), a 10Mbps bottleneck rate limit and either a 180-packet or 2000-packet bottleneck buffer. Relative to the pfifo case (Figures 8e and 8f), all three AQM schemes provide better capacity sharing with significantly reduced bottleneck queuing delays.

1) Using a 180-packet bottleneck buffer: Figure 11a show all three flows sharing broadly equally during periods of overlap through a PIE bottleneck, with moderate fluctuations over time. A similar result is observed in Figure 11c when traversing a codel bottleneck. In contrast, Figure 11e shows that using an fq_codel bottleneck results in each flow achieving quite consistent throughput and almost 'perfect' sharing.

Figures 11b, 11d and 11f illustrate the RTT experienced by each flow over time using PIE, codel and fq_codel respectively. Ignoring the brief spikes induced during slow-start as Flows 2 and 3 begin, PIE sees a somewhat wider range of RTT than codel or fq_codel (within ∼30ms of base RTT for PIE and within ∼10–15ms of base RTT for codel and fq_codel).

2) Using a 2000-packet bottleneck buffer: As noted in Section III-B2, a 180-packet buffer is significantly smaller than the recommended defaults for codel and fq_codel. Figure 12 repeats Section IV-B's trials with the bottleneck buffer increased to 2000 packets.

Visual inspection shows that overall the results are essentially the same. PIE, codel and fq_codel all meet their goals of keeping the RTTs low despite the significantly higher bottleneck buffer space (and with much the same range of absolute values as in Figure 11) . Throughput and capacity sharing also appears reasonably similar to what was achieved with a 180-packet buffer.

3) Using 180-packet bottleneck buffers @ 200ms RTT: Figure 13 show the performance over time when the path's base RTT is bumped up to 200ms.[12] Total RTT in Figures 13b, 13d and 13f is not unlike the 20ms case – all show RTT spiking at the start of each flow, then (mostly) stabilising much closer to the path's base RTT.[13]

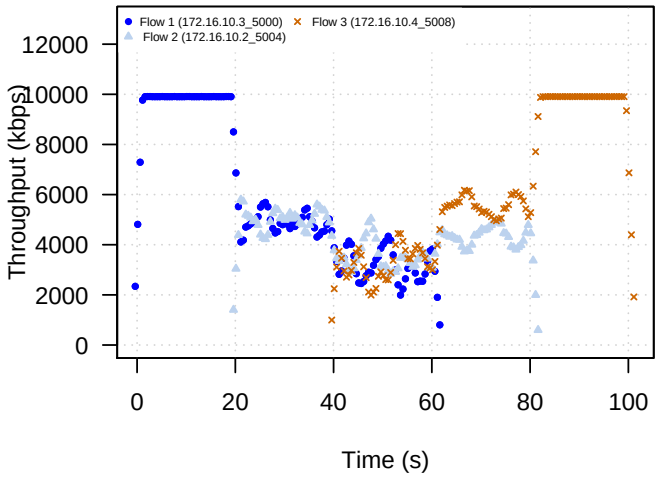
Figures 13a, 13c and 13e show an interesting result relative to the pfifo case in Figure 9. In all three cases Flow 1 struggles to reach full path capacity during its first 20 seconds, and Flow 3 struggles during the final 20 seconds. A likely explanation can be found in Figures 14a, 14b and 14c (`cwnd` vs time) – Flow 1 switches from slow-start (SS) to congestion avoidance (CA) mode at a much lower `cwnd` than was the case with pfifo.

Using PIE, Flow 1's `cwnd` initially spikes to ∼390kB, but this is quickly followed by two drops, resulting in CA mode taking over at ∼150kB (lower than the level required for full path utilisation). In both the codel and fq_codel cases Flow 1's `cwnd` rises to ∼340kB before being hit with a drop and switching to CA mode at ∼150kB.

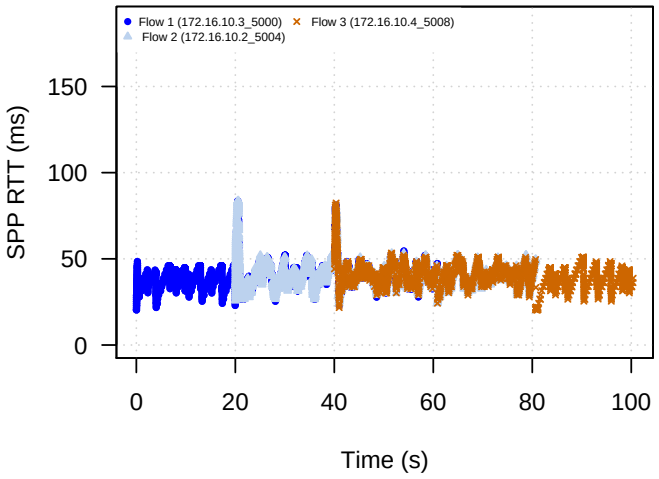
4) Discussion: As noted in Section III-B4, we again see all three AQM algorithms providing significant reduction in queuing delays. And fq_codel again provides balanced capacity sharing as individual flows are hashed into separate codel-managed queues, then serviced by a modified form of deficit round robin scheduling. And again we see the potential value of tuning AQM parameters for paths having high RTTs. Although Figure 14 showed each AQM keeping total RTTs low, throughput takes a hit due to the AQMs triggering a switch from SS to CA modes 'too early'.

[12] Noting that codel's defaults are set assuming a 100ms RTT [4].

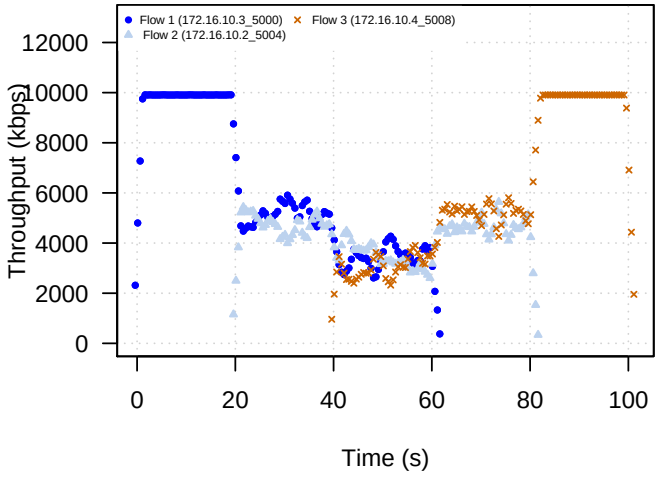
[13] The less smooth PIE results in Figure 13b need further investigation.



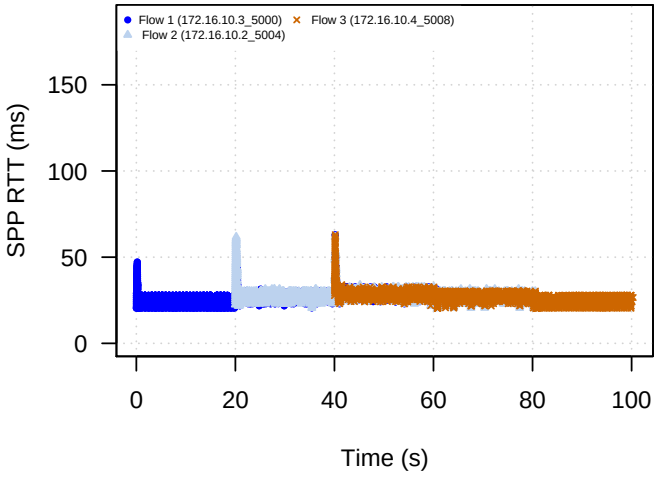
(a) Throughput vs time – PIE



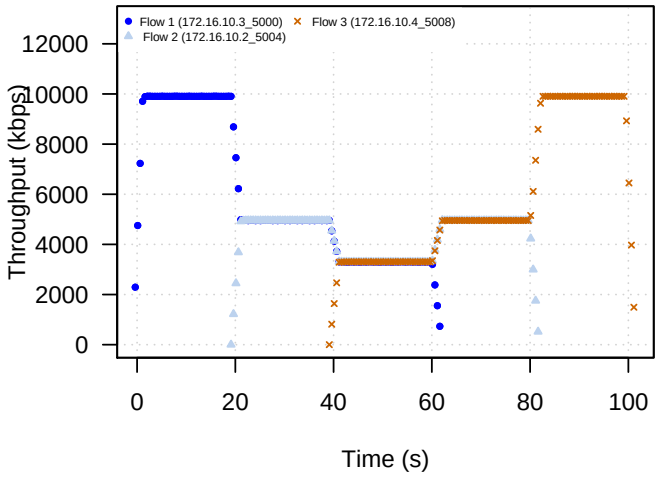
(b) Total RTT vs time – PIE



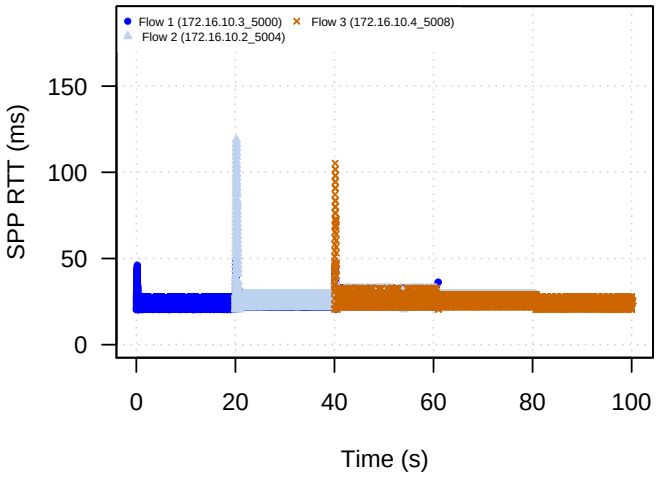
(c) Throughput vs time – codel



(d) Total RTT vs time – codel

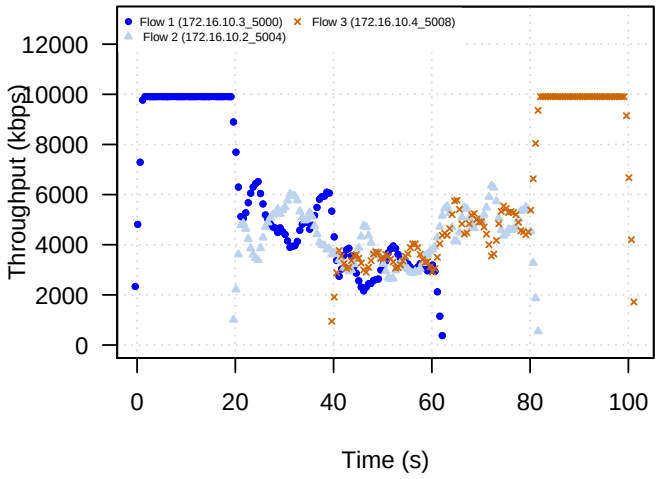


(e) Throughput vs time – fq_codel

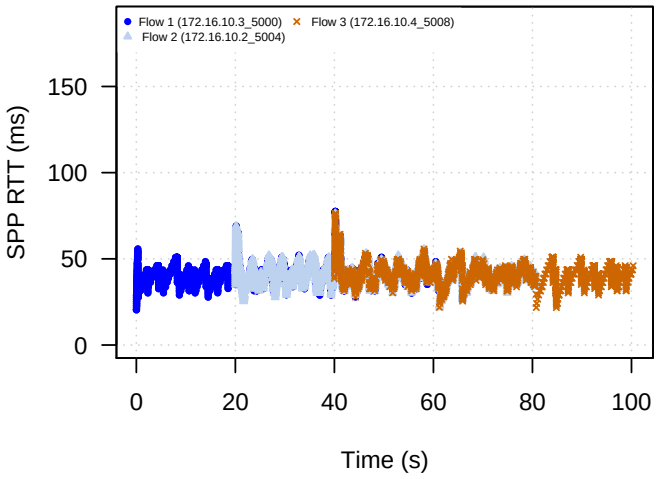


(f) Total RTT vs time – fq_codel

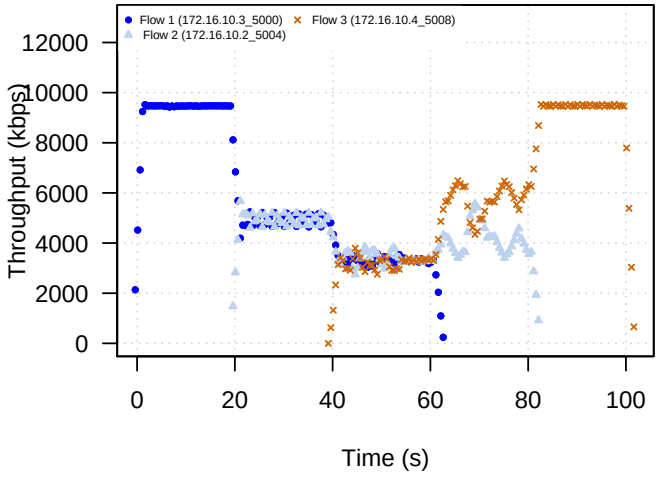
Figure 11: Three overlapping Linux CUBIC flows over a *20ms* RTT path with 10Mbps rate limit and 180-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel* (no ECN)



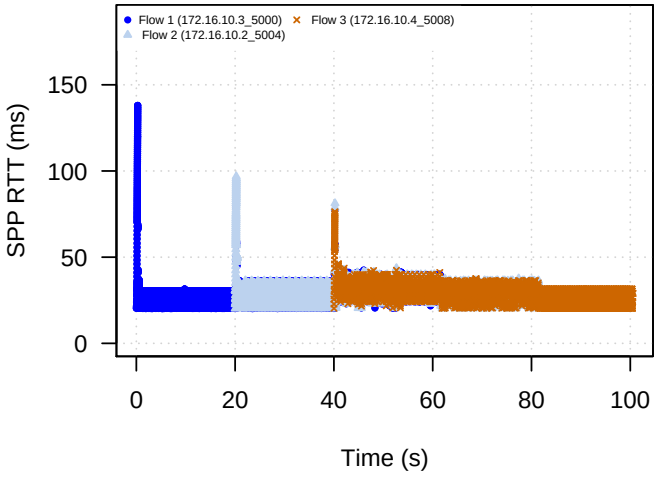
(a) Throughput vs time – PIE



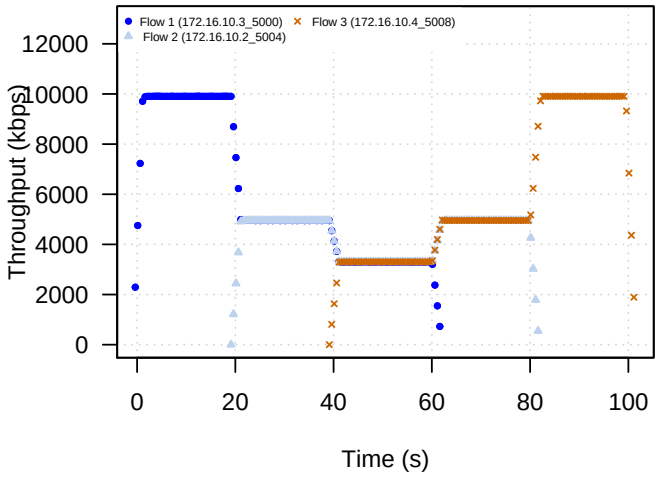
(b) Total RTT vs time – PIE



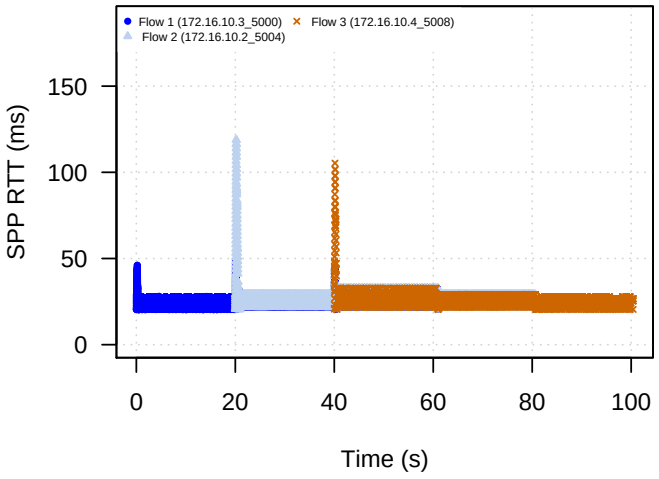
(c) Throughput vs time – codel



(d) Total RTT vs time – codel

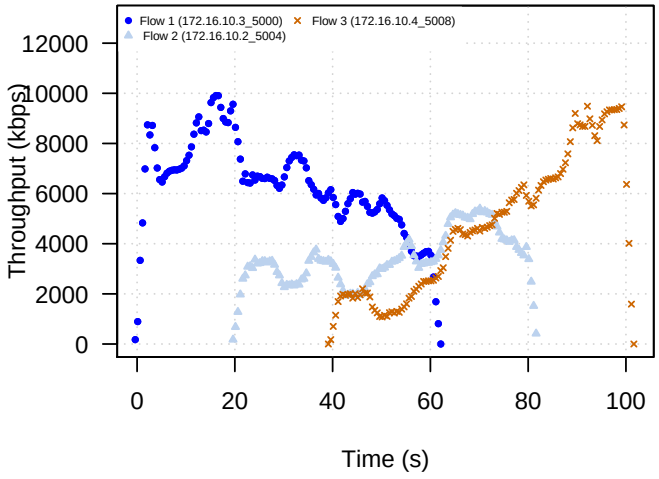


(e) Throughput vs time – fq_codel

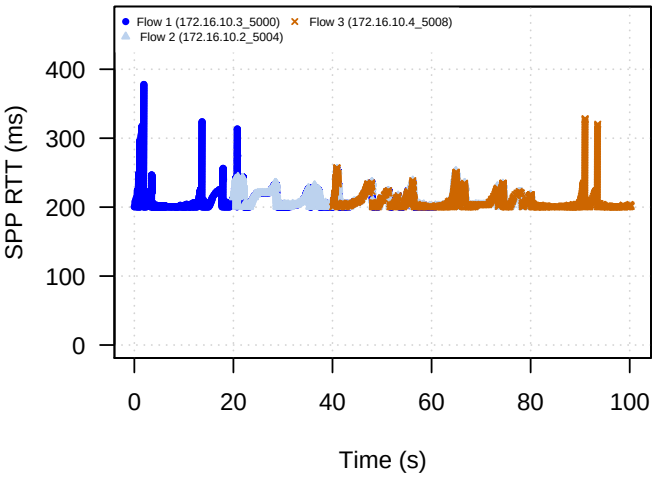


(f) Total RTT vs time – fq_codel

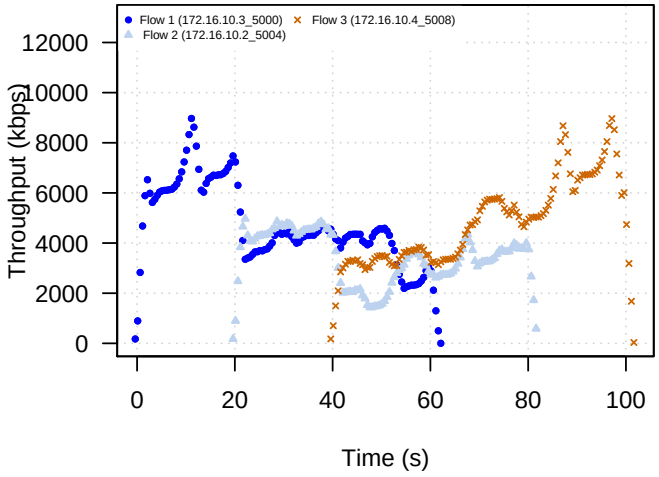
Figure 12: Three overlapping Linux CUBIC flows over a *20ms* RTT path with 10Mbps rate limit and 2000-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel* (no ECN)



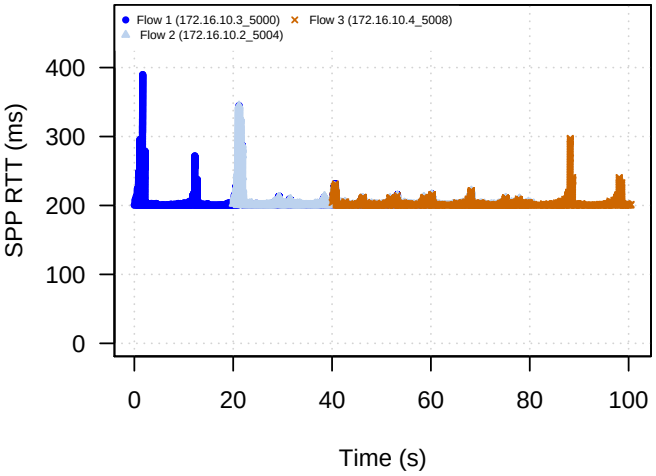
(a) Throughput vs time – PIE



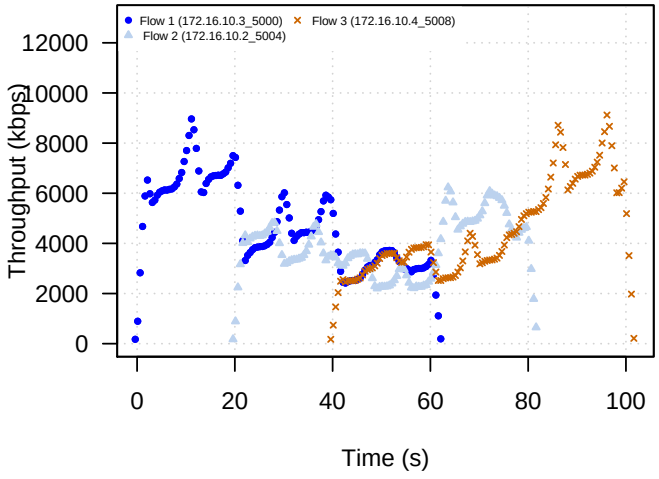
(b) Total RTT vs time – PIE



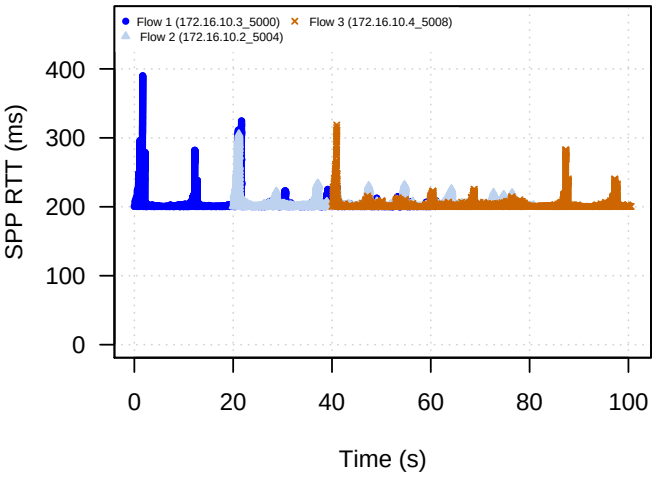
(c) Throughput vs time – codel



(d) Total RTT vs time – codel

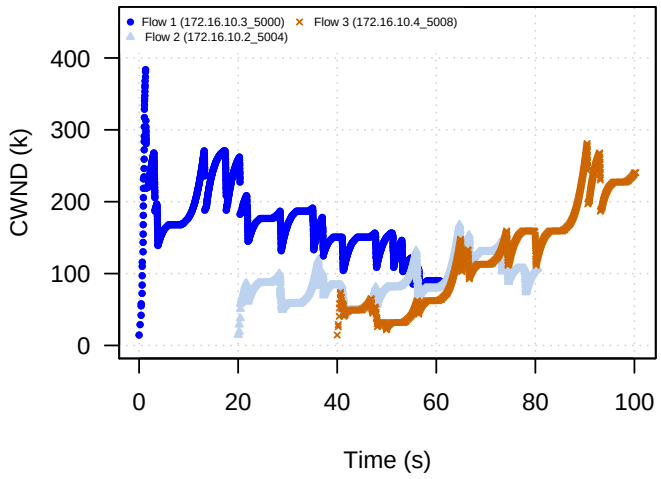


(e) Throughput vs time – fq_codel

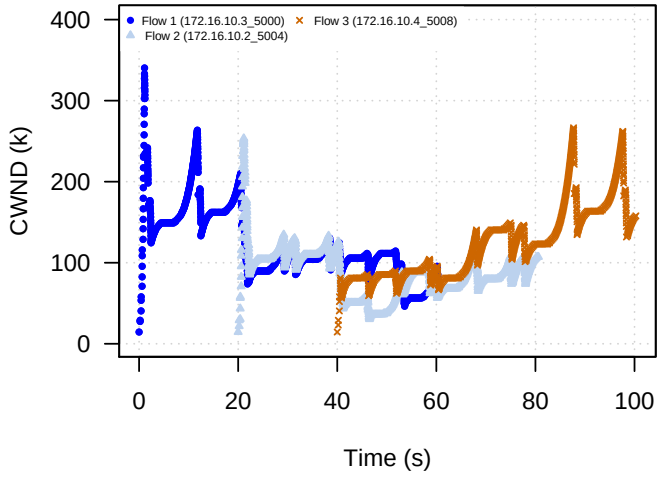


(f) Total RTT vs time – fq_codel

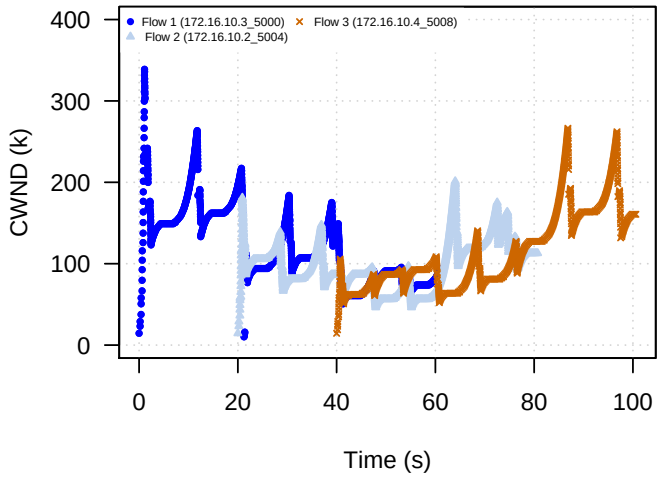
Figure 13: Three overlapping Linux CUBIC flows over a *200ms* RTT path with 10Mbps rate limit and 180-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel* (no ECN)



(a) cwnd vs time – PIE



(b) cwnd vs time – codel



(c) cwnd vs time – fq_codel

Figure 14: Three overlapping Linux CUBIC flows over a *200ms* RTT path at 10Mbps and 180-packet bottleneck buffer managed by *PIE*, *codel* or *fq_codel*

V. Conclusions and Future Work

This report describes some simple, staggered-start three-flow TCP tests run on the CAIA TCP testbed using TEACUP v0.4.x. We provide preliminary observations as to the consequences of three overlapping TCP flows sharing a 10Mbps bottleneck in the same direction. We compare results where the bottleneck uses pfifo (tail-drop), PIE, codel or fq_codel queue management schemes with default settings. This report is *only* intended to illustrate the plausibly correct behaviour of our testbed when three flows share a symmetric path.

As expected, NewReno and CUBIC induced significant queuing delays when sharing a pfifo-based bottleneck. Using PIE, codel and fq_codel at their default settings resulted in observed RTT dropping dramatically without significant impact on achieved throughput. Due to its hashing of flows into separate queues, fq_codel provides smoother capacity sharing between the three flows than either codel or PIE.

One topic for further analysis is the way in which default settings for pie, codel and fq_codel caused some sub-optimal performance when the path RTT was 200ms. (Initial analysis suggests this occurs due to the AQMs triggering a switch from slow start to congestion avoidance mode earlier than is desirable on such a path.) Another topic for further analysis is to explore the impact of enabling ECN on the end hosts.

Detailed analysis of why we see these specific results is a subject for future work. Future work will also include varying any PIE and fq_codel parameters that may plausibly be tuned, and explore both asymmetric path latencies and asymmetric path bottleneck bandwidths with concurrent (competing) TCP flows. Future work may also attempt to draw some conclusions about which of the tested TCP and AQM algorithms are 'better' by various metrics.

Acknowledgements

TEACUP v0.4.x was developed at CAIA by Sebastian Zander, as part of a project funded by Cisco Systems and titled "Study in TCP Congestion Control Performance In A Data Centre". This is a collaborative effort between CAIA and Mr Fred Baker of Cisco Systems.

Appendix A
FreeBSD Host TCP stack configuration

For the NewReno and CDG trials:

uname

```
FreeBSD newtcp3.caia.swin.edu.au 9.2-RELEASE FreeBSD
9.2-RELEASE #0 r255898: Thu Sep 26 22:50:31 UTC 2013
root@bake.isc.freebsd.org:/usr/obj/usr/src/sys/GENERIC
amd64
```

System information from sysctl

- kern.ostype: FreeBSD
- kern.osrelease: 9.2-RELEASE
- kern.osrevision: 199506
- kern.version: FreeBSD 9.2-RELEASE #0
- r255898: Thu Sep 26 22:50:31 UTC 2013
- root@bake.isc.freebsd.org:
- /usr/obj/usr/src/sys/GENERIC

net.inet.tcp information from sysctl

- net.inet.tcp.rfc1323: 1
- net.inet.tcp.mssdflt: 536
- net.inet.tcp.keepidle: 7200000
- net.inet.tcp.keepintvl: 75000
- net.inet.tcp.sendspace: 32768
- net.inet.tcp.recvspace: 65536
- net.inet.tcp.keepinit: 75000
- net.inet.tcp.delacktime: 100
- net.inet.tcp.v6mssdflt: 1220
- net.inet.tcp.cc.available: newreno
- net.inet.tcp.cc.algorithm: newreno
- net.inet.tcp.hostcache.purge: 0
- net.inet.tcp.hostcache.prune: 5
- net.inet.tcp.hostcache.expire: 1
- net.inet.tcp.hostcache.count: 0
- net.inet.tcp.hostcache.bucketlimit: 30
- net.inet.tcp.hostcache.hashsize: 512
- net.inet.tcp.hostcache.cachelimit: 15360
- net.inet.tcp.recvbuf_max: 2097152
- net.inet.tcp.recvbuf_inc: 16384
- net.inet.tcp.recvbuf_auto: 1
- net.inet.tcp.insecure_rst: 0
- net.inet.tcp.ecn.maxretries: 1
- net.inet.tcp.ecn.enable: 0
- net.inet.tcp.abc_l_var: 2
- net.inet.tcp.rfc3465: 1
- net.inet.tcp.experimental.initcwnd10: 0
- net.inet.tcp.rfc3390: 1
- net.inet.tcp.rfc3042: 1
- net.inet.tcp.drop_synfin: 0
- net.inet.tcp.delayed_ack: 1
- net.inet.tcp.blackhole: 0
- net.inet.tcp.log_in_vain: 0
- net.inet.tcp.sendbuf_max: 2097152
- net.inet.tcp.sendbuf_inc: 8192
- net.inet.tcp.sendbuf_auto: 1
- net.inet.tcp.tso: 0
- net.inet.tcp.path_mtu_discovery: 1
- net.inet.tcp.reass.overflows: 0
- net.inet.tcp.reass.cursegments: 0
- net.inet.tcp.reass.maxsegments: 1680
- net.inet.tcp.sack.globalholes: 0
- net.inet.tcp.sack.globalmaxholes: 65536
- net.inet.tcp.sack.maxholes: 128
- net.inet.tcp.sack.enable: 1
- net.inet.tcp.soreceive_stream: 0
- net.inet.tcp.isn_reseed_interval: 0
- net.inet.tcp.icmp_may_rst: 1
- net.inet.tcp.pcbcount: 6
- net.inet.tcp.do_tcpdrain: 1
- net.inet.tcp.tcbhashsize: 512
- net.inet.tcp.log_debug: 0
- net.inet.tcp.minmss: 216
- net.inet.tcp.syncache.rst_on_sock_fail: 1
- net.inet.tcp.syncache.rexmtlimit: 3
- net.inet.tcp.syncache.hashsize: 512
- net.inet.tcp.syncache.count: 0
- net.inet.tcp.syncache.cachelimit: 15375
- net.inet.tcp.syncache.bucketlimit: 30

- net.inet.tcp.syncookies_only: 0
- net.inet.tcp.syncookies: 1
- net.inet.tcp.timer_race: 0
- net.inet.tcp.per_cpu_timers: 0
- net.inet.tcp.rexmit_drop_options: 1
- net.inet.tcp.keepcnt: 8
- net.inet.tcp.finwait2_timeout: 60000
- net.inet.tcp.fast_finwait2_recycle: 0
- net.inet.tcp.always_keepalive: 1
- net.inet.tcp.rexmit_slop: 200
- net.inet.tcp.rexmit_min: 30
- net.inet.tcp.msl: 30000
- net.inet.tcp.nolocaltimewait: 0
- net.inet.tcp.maxtcptw: 5120

APPENDIX B
LINUX HOST TCP STACK CONFIGURATION

For the Cubic trials:

uname

```
Linux newtcp3.caia.swin.edu.au 3.9.8-desktop-web10g
#1 SMP PREEMPT Wed Jan 8 20:20:07 EST 2014 x86_64
x86_64 x86_64 GNU/Linux
```

System information from sysctl

- kernel.osrelease = 3.9.8-desktop-web10g
- kernel.ostype = Linux
- kernel.version = #1 SMP PREEMPT Wed Jan 8 20:20:07 EST 2014

net.ipv4.tcp information from sysctl

- net.ipv4.tcp_abort_on_overflow = 0
- net.ipv4.tcp_adv_win_scale = 1
- net.ipv4.tcp_allowed_congestion_control = cubic reno
- net.ipv4.tcp_app_win = 31
- net.ipv4.tcp_available_congestion_control = cubic reno
- net.ipv4.tcp_base_mss = 512
- net.ipv4.tcp_challenge_ack_limit = 100
- net.ipv4.tcp_congestion_control = cubic
- net.ipv4.tcp_cookie_size = 0
- net.ipv4.tcp_dma_copybreak = 4096
- net.ipv4.tcp_dsack = 1
- net.ipv4.tcp_early_retrans = 2
- net.ipv4.tcp_ecn = 0
- net.ipv4.tcp_fack = 1
- net.ipv4.tcp_fastopen = 0
- net.ipv4.tcp_fastopen_key = e8a015b2-e29720c6-4ce4eff7-83c84664
- net.ipv4.tcp_fin_timeout = 60
- net.ipv4.tcp_frto = 2
- net.ipv4.tcp_frto_response = 0
- net.ipv4.tcp_keepalive_intvl = 75
- net.ipv4.tcp_keepalive_probes = 9
- net.ipv4.tcp_keepalive_time = 7200
- net.ipv4.tcp_limit_output_bytes = 131072
- net.ipv4.tcp_low_latency = 0
- net.ipv4.tcp_max_orphans = 16384
- net.ipv4.tcp_max_ssthresh = 0
- net.ipv4.tcp_max_syn_backlog = 128
- net.ipv4.tcp_max_tw_buckets = 16384
- net.ipv4.tcp_mem = 89955 119943 179910
- net.ipv4.tcp_moderate_rcvbuf = 0
- net.ipv4.tcp_mtu_probing = 0
- net.ipv4.tcp_no_metrics_save = 1
- net.ipv4.tcp_orphan_retries = 0
- net.ipv4.tcp_reordering = 3
- net.ipv4.tcp_retrans_collapse = 1
- net.ipv4.tcp_retries1 = 3
- net.ipv4.tcp_retries2 = 15
- net.ipv4.tcp_rfc1337 = 0
- net.ipv4.tcp_rmem = 4096 87380 6291456
- net.ipv4.tcp_sack = 1
- net.ipv4.tcp_slow_start_after_idle = 1
- net.ipv4.tcp_stdurg = 0
- net.ipv4.tcp_syn_retries = 6
- net.ipv4.tcp_synack_retries = 5
- net.ipv4.tcp_syncookies = 1
- net.ipv4.tcp_thin_dupack = 0
- net.ipv4.tcp_thin_linear_timeouts = 0
- net.ipv4.tcp_timestamps = 1
- net.ipv4.tcp_tso_win_divisor = 3
- net.ipv4.tcp_tw_recycle = 0
- net.ipv4.tcp_tw_reuse = 0
- net.ipv4.tcp_window_scaling = 1
- net.ipv4.tcp_wmem = 4096 65535 4194304
- net.ipv4.tcp_workaround_signed_windows = 0

tcp_cubic information from /sys/module

- /sys/module/tcp_cubic/parameters/beta:717
- /sys/module/tcp_cubic/parameters/hystart_low_window:16
- /sys/module/tcp_cubic/parameters/fast_convergence:1
- /sys/module/tcp_cubic/parameters/initial_ssthresh:0
- /sys/module/tcp_cubic/parameters/hystart_detect:3
- /sys/module/tcp_cubic/parameters/bic_scale:41
- /sys/module/tcp_cubic/parameters/tcp_friendliness:1
- /sys/module/tcp_cubic/parameters/hystart_ack_delta:2
- /sys/module/tcp_cubic/parameters/hystart:1
- /sys/module/tcp_cubic/version:2.3

APPENDIX C
LINUX ROUTER CONFIGURATION

The bottleneck router is an 8-core machine, patched (as noted in Section V.D of [2]) to tick at 10000Hz for high precision packet scheduling behaviour.

uname

```
Linux newtcp5.caia.swin.edu.au
3.10.18-vanilla-10000hz #1 SMP PREEMPT Fri
Nov 8 20:10:47 EST 2013 x86_64 x86_64 x86_64
GNU/Linux
```

System information from sysctl

- kernel.osrelease = 3.10.18-vanilla-10000hz
- kernel.ostype = Linux
- kernel.version = #1 SMP PREEMPT Fri Nov 8 20:10:47 EST 2013

Bottleneck / AQM configuration

As noted in Section III.H of [1], we use separate stages to apply artificial delay and rate shaping respectively and separate pipelines for traffic flowing in either direction through the bottleneck router. The selected AQM is applied on ingress to the rate shaping section (the 3.10.18 kernel's pfifo, PIE or fq_codel). For example, when configured for a 180-packet bottleneck buffer:

qdisc when using PIE:

```
qdisc pie 1002: dev ifb0 parent 1:2 limit 180p
target 20 tupdate 30 alpha 2 beta 20
```

qdisc when using codel:

```
qdisc codel 1002: dev ifb0 parent 1:2 limit 180p
target 5.0ms interval 100.0ms ecn
```

qdisc when using fq_codel:

```
qdisc fq_codel 1002: dev ifb0 parent 1:2 limit
180p flows 1024 quantum 1514 target 5.0ms interval
100.0ms ecn
```

REFERENCES

[1] S. Zander, G. Armitage, "TEACUP v0.4 – A System for Automated TCP Testbed Experiments," Centre for Advanced Internet Architectures, Swinburne University of Technology, Tech. Rep. 140314A, March 2014. [Online]. Available: http://caia.swin.edu.au/reports/140314A/CAIA-TR-140314A.pdf

[2] S. Zander, G. Armitage, "CAIA Testbed for TCP Experiments," Centre for Advanced Internet Architectures, Swinburne University of Technology, Tech. Rep. 140314B, March 2014. [Online]. Available: http://caia.swin.edu.au/reports/140314B/CAIA-TR-140314B.pdf

[3] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. V. Steeg, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem," February 2014. [Online]. Available: http://tools.ietf.org/html/draft-pan-aqm-pie-01

[4] K. Nichols and V. Jacobson, "Controlled Delay Active Queue Management," March 2014. [Online]. Available: http://tools.ietf.org/html/draft-nichols-tsvwg-codel-02

[5] T. Høiland-Jørgensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "FlowQueue-Codel," March 2014. [Online]. Available: http://tools.ietf.org/html/draft-hoeiland-joergensen-aqm-fq-codel-00

[6] "The Web10G Project." [Online]. Available: http://web10g.org

[7] "PIE AQM source code." [Online]. Available: ftp://ftpeng.cisco.com/pie/linux_code/pie_code

[8] "iproute2 source code." [Online]. Available: http://www.kernel.org/pub/linux/utils/net/iproute2

[9] "iperf Web Page." [Online]. Available: http://iperf.fr/

[10] "NEWTCP Project Tools." [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools.html

[11] L. Stewart, "SIFTR – Statistical Information For TCP Research." [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=siftr

[12] S. Zander and G. Armitage, "Minimally-Intrusive Frequent Round Trip Time Measurements Using Synthetic Packet-Pairs," in *The 38th IEEE Conference on Local Computer Networks (LCN 2013)*, 21-24 October 2013.

[13] A. Heyde, "SPP Implementation," August 2013. [Online]. Available: http://caia.swin.edu.au/tools/spp/downloads.html