Dummynet AQM v0.2 – CoDel, FQ-CoDel, PIE and FQ-PIE for FreeBSD's ipfw/dummynet framework

Rasool Al-Saadi, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 160418A
Swinburne University of Technology
Melbourne, Australia
ralsaadi@swin.edu.au, garmitage@swin.edu.au

***Abstract*—Controlled delay (CoDel) and Proportional Integral controller Enhanced (PIE) are active queue management (AQM) schemes designed to control bottleneck queueing delay. FlowQueue-CoDel (FQ-CoDel) is hybrid scheme that hashes flows into one of N queues, applies CoDel AQM on a per-queue basis and utilises modified Deficit Round Robin (DRR) scheduling to share link capacity between queues. We provide the first well-documented implementations for FreeBSD's ipfw/dummynet framework, and confirm the utility of the IETF's AQM working group's current CoDel, FQ-Codel and PIE Internet Drafts. We also introduce a prototype "FlowQueue-PIE" (FQ-PIE) implementation that combines FQ-CoDel's FlowQueuing with PIE's individual queue management. We experimentally compare our implementations against the current Linux CoDel, FQ-CoDel and PIE and show plausible results from FreeBSD FQ-PIE. Patches have been prepared for FreeBSD11-CURRENT and FreeBSD-10.x-RELEASE.**

***Index Terms*—AQM, Scheduler, CoDel, FQ-CoDel, PIE, FQ-PIE, FreeBSD, Dummynet, IPFW**

I. Introduction

Network routers use buffers to enhance routing performance and increase overall throughput by absorbing packet bursts and reducing packets drop. In recent years, routers have used oversized buffer due to low memory prices and this causes high latency in congested bottlenecks if traditional tail drop is used. Researchers have been attracted to find solutions to this problem by using active queue management (AQM) to manage bottlenecks buffers. AQM controls queue length by dropping/marking packets from bottleneck buffer when it becomes full or queue delay becomes over a threshold value. Some AQMs, such as RED (Random Early Detection) [1] and its variations, use queue occupancy as an indicator of how much a queue is congested. However, these types of AQMs are hard to configure and perform badly in certain scenarios [2].

CoDel (Controlled delay) [3] and Proportional Integral controller Enhanced (PIE) [4] are examples of a modern AQM designed to remedy these limitations by using queue delay, instead of queue length, to indicate standing queues. FQ-CoDel [5] is a hybrid scheme – flows are assigned to one of a pool of internal queues, each active queue runs an independent instance of CoDel, and a modified Deficit Round Robin (DRR) scheduler shares outbound link capacity between the active queues. Implementations of CoDel and FQ-CoDel have existed in the Linux kernel since version 3.5 [6] and PIE since version 3.14 [7]. FreeBSD11-CURRENT has had a CoDel implementation in the pf/ALTQ framework[1] since August 21st 2015 (merged into FreeBSD 10-STABLE[2] on April 16th 2016).

This report introduces v0.2 of our AQM implementations for FreeBSD's ipfw/dummynet framework [8]. Using current IETF Internet Drafts [3]–[5] we now have independent implementations of CoDel, FQ-Codel and PIE for FreeBSD. We have additionally taken PIE one step further and implemented a hybrid "FQ-PIE" that combines FQ-CoDel's FlowQueuing with PIE queue management on individual queues. Our initial FQ-PIE experiments show good sharing of capacity between competing flows, while achieving high throughput and low queueing delay. Our work helps usefully enhance FreeBSD and demonstrate that [3], [5] and [4] are clear enough for implementation.

Our v0.2 patchset[3] can be applied to FreeBSD-10.{0,1,2,3}-RELEASE and FreeBSD11-CURRENT r297692[4]. We confirm the functionality of our v0.2

[1] https://svnweb.freebsd.org/base?view=revision&revision=287009
[2] https://svnweb.freebsd.org/base?view=revision&revision=298133
[3] http://caia.swin.edu.au/freebsd/aqm/patches/dummynet-aqm-patch-0.2.tgz
[4] And potentially later revisions of FreeBSD11-CURRENT

implementation by comparing our results with Linux CoDel, FQ-CoDel and PIE implementation.

This technical report extends the v0.1 report [9], and is organised as follows: Section II includes basic information about ipfw/dummynet. Sections III, IV, V and VI contain detailed information about configuring CoDel, FQ-CoDel, PIE and FQ-PIE respectively. Section VII summarises how to apply, test and install our patch. Section VIII compares experimentally derived results from the FreeBSD and Linux CoDel, FQ-CoDel and PIE implementations, Section IX briefly demonstrates results with FreeBSD FQ-PIE, and we conclude in Section X.

II. IPFW/DUMMYNET

FreeBSD's IPFW firewall supports both IPv4 and IPv6, and allows filtering, redirecting, NAT, forwarding, and other operations on IP packets passing through network interfaces [10]. IPFW is tightly integrated with dummynet, providing traffic shaping, delay emulation, packet scheduling and queue management functionality.

Originally a tool to run experiments in an emulated network environment, Dummynet has been improved over time to support emulation of complex network configurations [11]. The latest version of Dummynet implements three queue management schemes (Drop Tail, RED and GRED) and supports dynamically loadable packet schedulers with many schedulers implementation including First In First Out (FIFO), Worst-case Weighted Fair Queueing (WF2Q+), Quick Fair Queueing (QFQ), and others.

Dummynet provides users with three objects:

1) **Pipe**: represents a link that supports traffic shaping and delay emulation.
2) **Queue**: represents a queue of packets managed by a queuing management scheme and connected to a packet scheduler.
3) **Scheduler**: represents a packet scheduler connected to a link and has one or more queues.

The user-space `ipfw` command is used to create, delete, configure and show these objects. For simplicity and compatibility reasons, dummynet creates additional objects when a certain objects is created. For example, when new pipe is created, queue and scheduler objects are created as well. For more details see the ipfw(8) FreeBSD Man Pages [12].

We implemented CoDel and FQ-CoDel in Dummynet due to its popularity and flexibility. To make the process of implementing new AQMs for Dummynet easier, we also added support for dynamically loadable AQM kernel module similar Dummynet schedulers.

An AQM module must define two mandatory functions (dequeue and enqueue) and could have a structure for state variables (for each queue) to store AQM internal variables and another for configuration parameters (for flowset) to store AQM configurations. In our implementation, CoDel is implemented as an AQM module to be configured for queues (or for implicit queues created when pipes are created) while FQ-CoDel is implemented as scheduler module that includes both FQ scheduler code and CoDel AQM code.

III. CODEL

CoDel drops or marks packets depending on packet sojourn time in the queue, and aims to achieve high throughput while controlling queue delay and to be nearly insensitive to flow RTT. It was designed to be parameterless and work properly on the Internet without changing its default configurations. However, the defaults in [3] are not always suitable (for example, when path RTT is high) and can degrade TCP throughput. Thus, our implementation provides options to change CoDel parameters for each pipe/queue individually as well as changing the defaults.

A. CoDel Parameters

CoDel has two primary parameters (`target` and `interval`) and one option (`[no]ecn`) to enable or disable Explicit Congestion Notification (ECN). `target` is the minimum acceptable persistent queue delay that CoDel allows. CoDel does not drop packets directly after packets sojourn time becomes higher than `target` but waits for `interval` before dropping. `interval` should be set to maximum RTT for all expected connection. `ecn` controls whether, for ECN-enabled TCP flows, CoDel marks or drops packets when queue delay become high.

Table I shows our CoDel configuration parameters, default values and `sysctl` control variables to change the default value.

B. CoDel Synopsis

CoDel is used with dummynet '`pipe`' or '`queue`' and can be configured through `ipfw` [12] interface. CoDel has the following synopsis:

```
ipfw pipe/queue x config [...] codel [
   target t] [interval t] [ecn | noecn]
```

where

`t` is time in seconds (s), milliseconds (ms) or microseconds (us). The default interpretation is milliseconds.

Table I: CoDel configuration parameters & options

Parameters	Default	Min	Max	sysctl variable names and units (to set defaults)
target	5ms	1us	5s	net.inet.ipdummynet.codel.target (microseconds)
interval	100ms	1us	5s	net.inet.ip.dummynet.codel.interval (microseconds)
Options				
[no]ecn	noecn			-

Note: any token after 'codel' is considered a CoDel parameter, so ensure all pipe/queue configuration options are written before 'codel'.

C. Examples of using CoDel

This subsection includes some examples of using CoDel with ipfw/dummynet. It should be noted that ipfw passes packets that match a classification rule of dummynet pipe/queue to next rule by default. Thus, a rule with 'allow' action should be added in some point after pipe/queue rule.

1) One pipe controlled by CoDel AQM (default configuration) and rate limit to 1 Mbits/s.

```
ipfw pipe 1 config bw 1mbits/s codel
ipfw add 100 pipe 1 ip from any to
   any
```

2) Two queues controlled by CoDel AQM using different CoDel configurations parameters. The pipe that queue 1 and 2 use has rate limit to 10 Mbits/s and 20ms emulated delay. In more details, queue 1 and 2 connected to an implicit WF2Q+ scheduler that use pipe 1 for traffic shaping and adding emulated delay.

```
ipfw pipe 1 config bw 10mbits/s delay
    20ms
ipfw queue 1 config pipe 1 codel
   target 7ms ecn
ipfw queue 2 config pipe 1 codel
   target 8ms interval 160ms ecn
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
ipfw add 200 queue 2 ip from
   172.16.0.0/16 to 172.16.0.0/16
```

3) Two queues - queue 1 controlled by CoDel AQM and queue 2 uses droptail. Both queues are connected to QFQ scheduler that uses pipe 1 for rate limit to 5 Mbits/s.

```
ipfw pipe 1 config bw 5mbits/s
ipfw sched 1 config pipe 1 type qfq
ipfw queue 1 config sched 1 codel
ipfw queue 2 config sched 1
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
ipfw add 200 queue 2 ip from
   172.16.0.0/16 to 172.16.0.0/16
```

IV. FQ-CoDel

As noted earlier, FQ-CoDel aims to control queuing delays while sharing bottleneck capacity relatively evenly among competing flows. FQ-CoDel's modified DRR (Deficit Round Robin) scheduler manages two lists of queues – old queues and new queues – to provide brief periods of priority to lightweight or short burst flows. FQ-CoDel's internal, dynamically created queues are controlled by separate instances of CoDel AQM (including separate state variables per queue).

The default parameters for FQ-CoDel in [5] are chosen to be generally useful. However, they are not always suitable so our implementation provides options to change FQ-CoDel parameters for each scheduler individually as well as changing the defaults.

A. FQ-CoDel Parameters

FQ-CoDel has five primary parameters (target, interval, quantum, limit and flows) and one option ([no]ecn) to enable or disable ECN. target, interval and [no]ecn are per-queue CoDel parameters described in section III-A. quantum is number of bytes a queue can be served before being moved to the tail of old queues list. limit is the hard size limit of all queues managed by an instance of the fq_codel scheduler. flows is number of flow queues that fq_codel creates and manages.

Table II shows our FQ-CoDel configuration parameters, default values and sysctl control variables to change the default value.

B. FQ-CoDel synopsis

fq_codel is used with dummynet scheduler object ('schd') and can be configured through ipfw interface. fq_codel has the following synopsis:

```
ipfw sched x config [...] type fq_codel
    [target t] [interval t] [ecn |
   noecn] [quantum n] [limit n] [flows
   n]
```

Table II: FQ-CoDel configuration parameters & options

Parameters	Default	Min	Max	sysctl variable names and units (to set defaults)
target	5ms	1us	5s	net.inet.ip.dummynet.fqcodel.target (microseconds)
interval	100ms	1us	5s	net.inet.ip.dummynet.fqcodel.interval (microseconds)
quantum	1514	1	9000	net.inet.ip.dummynet.fqcodel.quantum (bytes)
limit	10240	1	20480	net.inet.ip.dummynet.fqcodel.limit (packets)
flows	1024	1	65536	net.inet.ip.dummynet.fqcodel.flows (queues)
Options				
[no]ecn	ECN			-

where

- `t` is time in seconds (s), milliseconds (ms) or microseconds (us). The default interpretation is milliseconds.
- `n` is an integer number

Note: any token after '`fq_codel`' is considered an FQ-CoDel parameter, so ensure all other scheduler configuration options come before '`fq_codel`'.

C. Examples of using FQ-CoDel

This subsection includes some examples of using fq_codel with ipfw/dummynet. Note that `ipfw` passes packets that match a classification rule of dummynet pipe/queue to next rule by default. Thus, a rule with '`allow`' action should be added in some point after pipe/queue rule.

1) One scheduler with one queue, 2048 fq_codel sub-queues, target 7ms and quantum 2000 bytes

```
ipfw pipe 1 config bw 10mbits/s
ipfw sched 1 config pipe 1 type
   fq_codel target 7ms quantum 2000
   flows 2048
ipfw queue 1 config sched 1
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
```

2) One scheduler with two queues (1024 fq_codel sub-queues by default), interval 150ms, ECN enabled

```
ipfw pipe 1 config delay 10ms
ipfw sched 1 config pipe 1 type
   fq_codel interval 150ms ecn
ipfw queue 1 config sched 1
ipfw queue 2 config sched 1
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
ipfw add 200 queue 2 ip from
   172.16.0.0/16 to 172.16.0.0/16
```

V. PIE

PIE drops or marks packets depending on calculated drop probability *p* during en-queue process, with the aim of achieving high throughput while keeping queue delay low. At regular time intervals (*tupdate*) a background process (re)calculates *p* based on queue delay deviations from *target* and queue delay trends. PIE approximates current queue delay by using a departure rate estimation method, or (optionally) by using a packet timestamp method similar to CoDel. PIE was designed to work properly on the Internet using the default configurations. However, the defaults in [4] are not appropriate for all network environments such as in datacenters due to very low latency. Moreover, PIE configurations should be tunable for studying and evaluation purposes. Thus, our implementation provides options to change PIE parameters away from the defaults for each pipe/queue individually.

A. PIE Parameters

PIE has a number of configurable parameters and options derived from [4] :

1) **`target:`** The acceptable persistent queue delay. Drop probability increases as queue delay increases higher than `target`.
2) **`tupdate:`** The frequency of drop probability re-calculation.
3) **`alpha` and `beta`** are drop probability weights.
4) **`max_burst:`** The maximum period of time that PIE does not drop/mark packets.
5) **`max_ecnth:`** When ECN is enabled, PIE drops packets instead of marking them when drop probability becomes higher than ECN probability threshold `max_ecnth`.

PIE has the following options:

1) **`[no]ecn:`** enable (**`ecn`**) or disable (**`noecn`**) ECN marking for ECN-enabled TCP flows.
2) **`[no]capdrop:`** enable (**`capdrop`**) or disable (**`nocapdrop`**) cap drop adjustment.

3) **[no]derand:** enable (**derand**) or disable (**noderand**) drop probability de-randomisation. De-randomisation eliminates the problem of dropping packets too close or too far.
4) **onoff** enable turing PIE on and off depending on queue load. PIE tunes on when over 1/3 of queue becomes full.
5) **[dre|ts]:** Calculate queue delay using departure rate estimation (**dre**) or timestamps (**ts**).

Table III shows our PIE configuration parameters and options, default values and `sysctl` control variables to change the default value.

B. PIE Synopsis

PIE is used with dummynet '`pipe`' or '`queue`' and can be configured through `ipfw` [12] interface. PIE has the following synopsis:

```
ipfw pipe/queue x config [...] pie [
   target t] [tupdate t] [alpha m] [beta
   m] [max_burst t] [max_ecnth m] [ecn |
   noecn] [capdrop | nocapdrop] [drand |
   nodrand] [onoff] [dre | ts]
```

where

`t` is time in second (s), millisecond (ms) or microsecond (us). The default interpretation is milliseconds.

m is a real number

Note: any token after 'pie' is considered a PIE parameter, so ensure all pipe/queue configuration options are written before 'pie'.

C. Examples of using PIE

This subsection includes some examples of using PIE with ipfw/dummynet. It should be noted that `ipfw` passes packets that match a classification rule of dummynet pipe/queue to next rule by default. Thus, a rule with '`allow`' action should be added in some point after pipe/queue rule.

1) One pipe controlled by PIE AQM (default configuration) and rate limit to 1 Mbits/s.

```
ipfw pipe 1 config bw 1mbits/s pie
ipfw add 100 pipe 1 ip from any to
   any
```

2) Two queues controlled by PIE AQM using different PIE configurations parameters. The pipe that queue 1 and 2 use has rate limited to 10 Mbits/s and 20ms emulated delay. In more details, queue 1 and 2 connected to an implicit WF2Q+ scheduler that use pipe 1 for traffic shaping and adding emulated delay.

```
ipfw pipe 1 config bw 10mbits/s delay
    20ms
ipfw queue 1 config pipe 1 pie target
    25ms ecn
ipfw queue 2 config pipe 1 pie target
    20ms tupdate 30ms ecn
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
ipfw add 200 queue 2 ip from
   172.16.0.0/16 to 172.16.0.0/16
```

3) Two queues - queue 1 controlled by PIE AQM and queue 2 uses CoDel. Both queues are connected to QFQ scheduler that uses pipe 1 for rate limited to 5 Mbits/s.

```
ipfw pipe 1 config bw 5mbits/s
ipfw sched 1 config pipe 1 type qfq
ipfw queue 1 config sched 1 pie
ipfw queue 2 config sched 1 codel
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
ipfw add 200 queue 2 ip from
   172.16.0.0/16 to 172.16.0.0/16
```

VI. FQ-PIE

In the absence of any normative reference implementation or Internet Draft, our implementation of FQ-PIE is the combination of FQ-CoDel's FlowQueue logic with PIE queue management on every dynamically created sub-queue. The goals are similar to FQ-CoDel – control queuing delays while sharing bottleneck capacity relatively evenly among competing flows. We set each instance of PIE to use timestamps (`ts`) rather than departure rate estimation (dre) in the context of FQ-PIE, as there have been doubts raised as to the accuracy of dre in such a context [4].

Our implementation uses the same default parameters for FQ-PIE as in [5], and provides options to change FQ-PIE parameters for each scheduler individually as well as changing the defaults.

A. FQ-PIE Parameters

Table IV shows our FQ-PIE configuration parameters and options, default values and `sysctl` control variables to change the default value. Many are equivalent to PIE parameters described in section V-A. The remaining parameters are borrowed from FQ-CoDel: `quantum` is number of bytes a queue can be served before being moved to the tail of old queues list, `limit` is the hard size limit of all queues managed by an instance of the fq_pie scheduler and `flows` is the number of flow queues that fq_pie creates and manages.

Table III: PIE configuration parameters/options

Parameter	Default	Min	Max	sysctl variable names and units (to set defaults)
target	15ms	1us	5s	net.inet.ipdummynet.pie.target (microseconds)
tupdate	15ms	1us	5s	net.inet.ip.dummynet.pie.tupdate (microseconds)
alpha	0.125	0	7	net.inet.ip.dummynet.pie.alpha (by 1000)
beta	1.25	0	7	net.inet.ip.dummynet.pie.beta (by 1000)
max_burst	150ms	0	10s	net.inet.ip.dummynet.pie.max_burst (microseconds)
max_ecnth	0.1	0	1	net.inet.ip.dummynet.pie.max_ecnth (by 1000)
Options				
[no]ecn	noecn			-
[no]capdrop	capdrop			
[no]derand	derand			
onoff	no onoff			
[dre|ts]	dre			

Table IV: FQ-PIE configuration parameters/options

Parameter	Default	Min	Max	sysctl variable names and units (to set defaults)
target	15ms	1us	5s	net.inet.ipdummynet.fqpie.target (microseconds)
tupdate	15ms	1us	5s	net.inet.ip.dummynet.fqpie.tupdate (microseconds)
alpha	0.125	0	7	net.inet.ip.dummynet.fqpie.alpha (by 1000)
beta	1.25	0	7	net.inet.ip.dummynet.fqpie.beta (by 1000)
max_burst	150ms	0	10s	net.inet.ip.dummynet.fqpie.max_burst (microseconds)
max_ecnth	0.1	0	1	net.inet.ip.dummynet.fqpie.max_ecnth (by 1000)
quantum	1514	1	9000	net.inet.ip.dummynet.fqpie.quantum (bytes)
limit	10240	1	20480	net.inet.ip.dummynet.fqpie.limit (packets)
flows	1024	1	65536	net.inet.ip.dummynet.fqpie.flows (queues)
Options				
[no]ecn/noecn	noecn			-
[no]capdrop/nocapdrop	capdrop			
[no]derand/nodrand	derand			
onoff	no onoff			
[dre|/ts]	ts			

B. FQ-PIE Synopsis

FQ-PIE is used with dummynet 'pipe' or 'queue' and can be configured through ipfw [12] interface. FQ-PIE has the following synopsis:

```
ipfw sched x config [...] type fq_pie [
   target t] [tupdate t] [alpha m] [beta
   m] [max_burst t] [max_ecnth m] [
   quantum n] [limit n] [flows n] [ecn |
   noecn] [capdrop | nocapdrop] [drand |
   nodrand] [onoff] [dre | ts]
```

where

- t is time in second (s), millisecond (ms) or microsecond (us). The default interpretation is milliseconds.
- n is an integer number
- m is a real number

Note: any token after 'fq_pie' is considered a FQ-PIE parameter, so ensure all pipe/queue configuration options are written before 'fq_pie'.

C. Examples of using FQ-PIE

This subsection includes some examples of using fq_pie with ipfw/dummynet. Note that ipfw passes packets that match a classification rule of dummynet pipe/queue to next rule by default. Thus, a rule with 'allow' action should be added in some point after pipe/queue rule.

1) One fq_pie scheduler, 2048 fq_pie subqueues, target 10ms, tupdate 10ms, and quantum 2000 bytes. ECN is disabled

```
ipfw pipe 1 config bw 10mbits/s
ipfw sched 1 config pipe 1 type
   fq_pie target 10ms tupdate 10ms
   quantum 2000 flows 2048 noecn
ipfw queue 1 config sched 1
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
```

2) One scheduler with two queues (1024 fq_pie subqueues by default), ECN enabled, maximum ECN threshold 50% and no capdrop.

```
ipfw pipe 1 config delay 10ms
ipfw sched 1 config pipe 1 type
   fq_pie max_ecnth 0.5 nocapdrop
ipfw queue 1 config sched 1
ipfw queue 2 config sched 1
ipfw add 100 queue 1 ip from
   192.168.0.0/16 to 192.168.0.0/16
ipfw add 200 queue 2 ip from
   172.16.0.0/16 to 172.16.0.0/16
```

VII. APPLYING THE PATCH/TESTING/INSTALLATION

We have tested our FreeBSD11-based v0.2 patch [8] against FreeBSD11-CURRENT r297692, although it may also work with earlier or later builds. To build r297692 from source you'll need to checkout FreeBSD11-CURRENT r297692 source tree using:

```
svn checkout -r 295345 svn://svn.freebsd.
   org/base/head/ /usr/src/
```

We also provide a version for FreeBSD 10.x-RELEASE (10.0, 10.1, 10.2, 10.3) as a separate patch file that combines our CoDel/FQ-CoDel/PIE/FQ-PIE code with ECN marking code backported from FreeBSD11-CURRENT r266941.

Once the patch is applied, you only need to (re)build the `dummynet.ko` kernel module and `ipfw` userland command (rather than rebuild a complete kernel and world from source). As `root` user do the following steps:

A. Applying the patch

Apply the patch as follows:

1) Extract the patch file

```
tar -xvf dummynet-aqm-patch-0.2.tgz -
   C /usr/src/ cd /usr/src
```

2) Apply the patch
   a) For FreeBSD11-CURRENT

```
patch -p1 < dummynet-aqm-patch
   -0.2/freebsd11-r297692.patch
```

   b) For FreeBSD 10.x-RELEASE

```
patch -p1 < dummynet-aqm-patch
   -0.2/freebsd10.x.patch
```

3) Copy `ip_dummynet.h` to /usr/include/netinet.

```
cp /usr/src/sys/netinet/ip_dummynet.h
    /usr/include/netinet/
```

4) Build ipfw userland.

```
cd /usr/src/sbin/ipfw
make
```

5) Build dummynet kernel module

```
cd /usr/src/sys/modules/dummynet/
make
```

B. Testing the patched ipfw/dummynet

Use the following steps to check whether the patch applied cleanly and built both userland ipfw and dummynet kernel module:

1) Check if old dummynet is already loaded

```
kldstat | grep dummynet
```

2) If dummynet is already loaded, unload it.

```
kldunload dummynet
```

3) Load the patched dummynet.ko into FreeBSD kernel

```
kldload /usr/src/sys/modules/dummynet
   /dummynet.ko
```

4) Check debug messages using '`dmesg`' command:

```
dmesg | grep 'CODEL\|PIE'
```

The Output should be something like:

```
load_dn_sched dn_aqm PIE loaded
load_dn_aqm dn_aqm CODEL loaded
load_dn_sched dn_sched FQ_CODEL
   loaded
load_dn_sched dn_sched FQ_PIE loaded
```

5) Use the patched ipfw interface (specify a full pathname when use ipfw):

```
/usr/src/sbin/ipfw/ipfw pipe 1 config
    codel
/usr/src/sbin/ipfw/ipfw pipe 1 show
```

The Output should be something like:

```
00001: unlimited 0 ms burst 0
q131073 50 sl. 0 flows (1 buckets)
   sched 65537 weight 0 lmax 0 pri 0
   AQM CoDel target 5ms interval 100
   ms
sched 65537 type FIFO flags 0x0 0
   buckets 0 active
```

C. Installing the patched ipfw and dummynet.ko

To use the patched ipfw/dummynet by default, install them as follow:

1) Install ipfw interface, ignore any error if appears.

```
cd /usr/src/sbin/ipfw
make install
```

2) Install dummynet kernel module

```
cp /usr/src/sys/modules/dummynet/
    dummynet.ko /boot/kernel/
```

3) (Optional) To avoid the warning "KLD '/boot/kernel/dummynet.ko' is newer than the linker.hints file", regenerate kernel loader hints with:

```
kldxref /boot/kernel
```

VIII. Experimental comparisons of CoDel, FQ-CoDel and PIE in Linux and FreeBSD

We used a TEACUP-based [13] testbed[5] to compare the behaviour of our implementation with a Linux implementation. Figure 1 shows our testbed's network topology, with three hosts[6], one bottleneck router[7] and a control host. Experiment network links are 1Gbps Ethernet, while the Control network used separate 100Mbps Ethernet connections to each machine.

Each endhost could be booted into FreeBSD10.1-RELEASE (for NewReno flows) or Linux 4.2 (for CUBIC flows) while the control host ran FreeBSD 10.1. The bottleneck router (with the AQM under test) was booted into FreeBSD10.1-RELEASE[8] or Linux 4.1 as required. When running CoDel, FQ-CoDel and PIE under the Linux bottleneck router, TEACUP uses the `netem` and `tc` modules to emulate our target RTT and bottleneck bandwidths.[9]

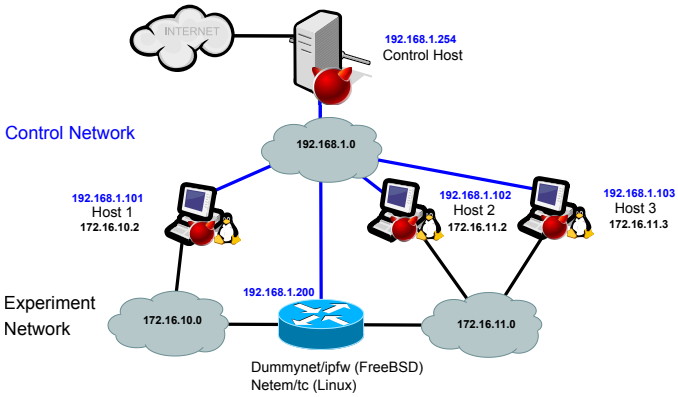


Figure 1: TEACUP Testbed topology

[5]TEACUP source is at https://sourceforge.net/projects/teacup/

[6]Intel Core 2 Duo @ 3GHz, 4GiB RAM, 1Gbps NICs

[7]Intel Core 2 Duo @ 2.33GHz, 1Gbps NICs

[8]We chose FreeBSD 10.1 because FreeBSD11-CURRENT is the development branch with lots of debugging code enabled

[9]See section IV-B of [13] for details on how TEACUP configures `netem` and `tc` for this purpose.

A. Linux CoDel vs FreeBSD CoDel

In all our CoDel comparison experiments we used `iperf` to generate one NewReno (using FreeBSD) or CUBIC (using Linux) TCP flow running for 60 seconds between Hosts 2 and 3 (Figure 1). The router applied independent instances of CoDel to 10Mbit/sec bottlenecks in each direction, with each CoDel buffer set to 1000 packets. The router also emulated either 20ms or 40ms of underlying path RTT.

Scenario 1: NewReno over CoDel @ 20ms RTT

In scenario 1 the end hosts booted into FreeBSD and ran TCP NewReno over a 20ms RTT path. CoDel was configured with target 5 ms, interval 100 ms and no ECN. Figure 2 shows throughput[10], CWND and smoothed TCP RTT versus time for both FreeBSD and Linux implementations of CoDel. Our implementation behaves very similarly to CoDel under Linux.

Scenario 2: NewReno over CoDel @ 40ms RTT

Scenario 2 is the same as scenario 1 except that the emulated path RTT is 40 ms. Figure 3 shows throughput, CWND and smoothed TCP RTT versus time, and again the behaviour of our implementation is very similar to CoDel under Linux.

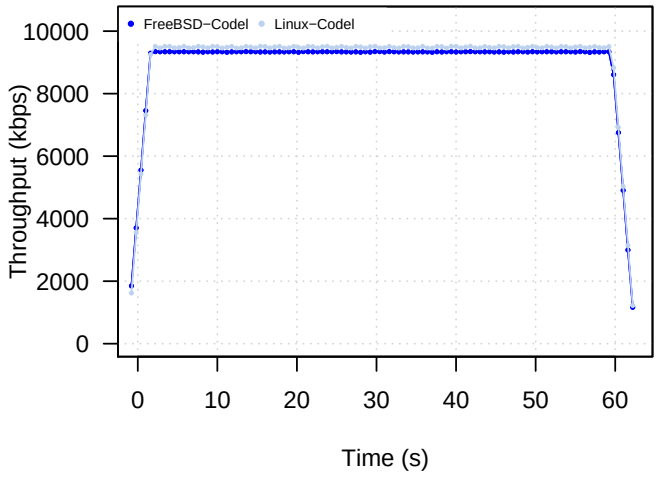
Scenario 3: CUBIC over CoDel @ 20ms RTT

In scenario 3 the end hosts booted into Linux and ran TCP CUBIC over a 20ms RTT path. CoDel was configured with target 10ms, interval 100ms and ECN enabled. We checked number of dropped packet and confirmed that ECN is functional with nothing dropped during the experiment. Figure 4 shows throughput, CWND, smoothed TCP RTT versus time. Again our implementation behaves similarly to CoDel under Linux.

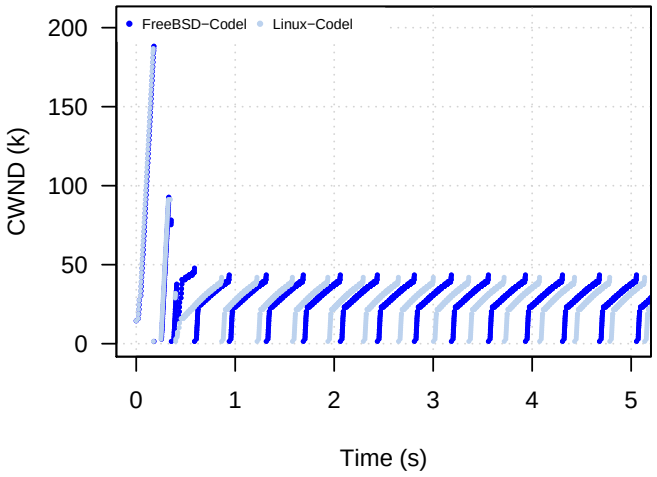
B. Linux FQ-CoDel vs FreeBSD FQ-CoDel

Similar to the CoDel experiments, we compared our FQ-CoDel implementation results with Linux FQ-CoDel to illustrate their similarities. In this case we used multiple instances of `iperf` under Linux to generate four TCP CUBIC flows with staggered start and end times (starting at t=0, 10, 20, 30 seconds and each flow lasting for 60 seconds). Each instance of fq-codel was configured for target 5ms, interval 100ms, no ECN, quantum 1514 bytes and with 10240 packets of bottleneck buffering shared by 1024 fq_codel sub-queues.

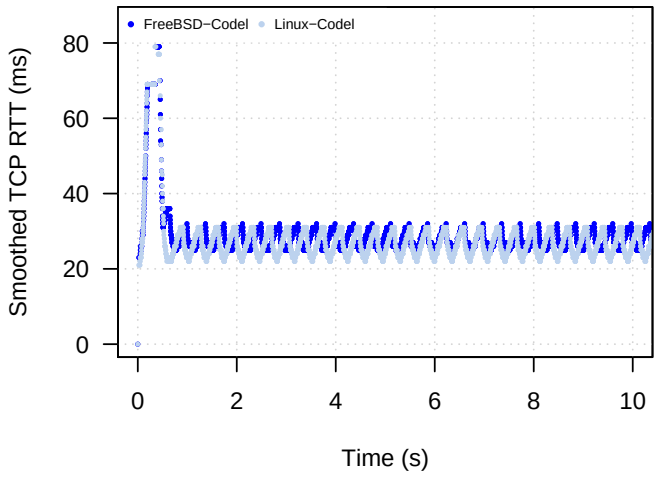
[10]All CoDel experiments throughputs were calculated using 3 seconds window moving forward in steps of 0.6 sec



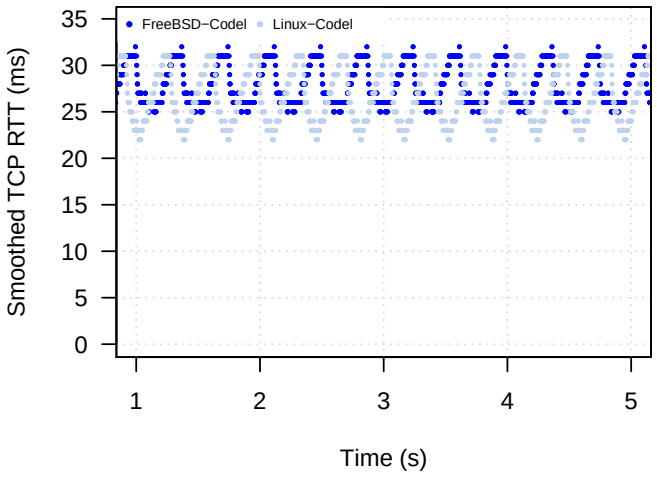
(a) Throughput vs. Time



(b) CWND vs. Time from t=0 to t=5



(c) smoothed TCP RTT vs. Time for first 10 seconds



(d) smoothed TCP RTT vs. Time from t=1 to t=5

Figure 2: One FreeBSD NewReno flow, 20ms RTT path, 10Mbps rate limit and 1000-packet bottleneck buffer. Linux and FreeBSD CoDel configured for target 5ms, interval 100ms, ECN disabled (Scenario 1)

Scenario 4: 10Mbit/sec bottleneck

In scenario 4 the bottleneck was configured for 10Mbit/sec. Figure 5 shows throughput[11], CWND and smoothed TCP RTT versus time for the experiment. As with CoDel, our FQ-CoDel implementation behaves similarly to the Linux implementation.

Scenario 5: 1Mbit/sec bottleneck

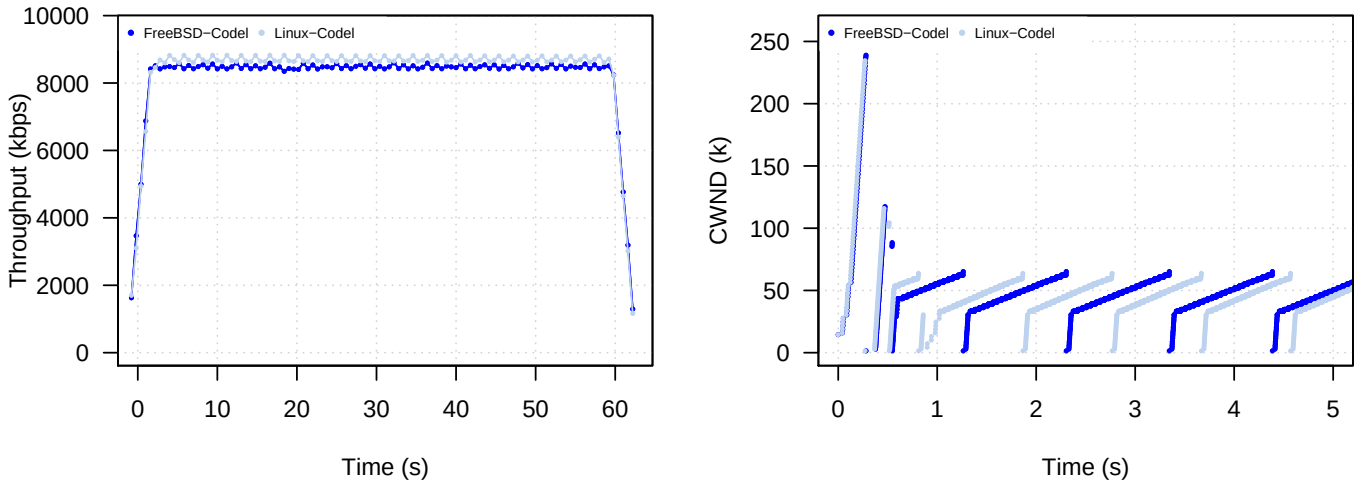
Scenario 5 repeats scenario 4 but with a 1Mbit/sec bottleneck. Figure 6 shows throughput, CWND and smoothed TCP RTT versus time for the experiment. For reasons we have yet to determine, our FQ-CoDel implementation seems to actually share the limited 1Mbps among multiple flows more consistently than the Linux implementation.[12]

C. Linux PIE vs FreeBSD PIE

Our FreeBSD PIE is based on the latest (version 06) PIE Internet Draft [4], whilst the current Linux PIE implementation (as of kernel 4.5) is based on an earlier PIE Internet Draft. Consequently, Linux PIE uses different default parameters (20ms target, 30ms tupdate and 100ms max_burst) and does not reflect subsequent

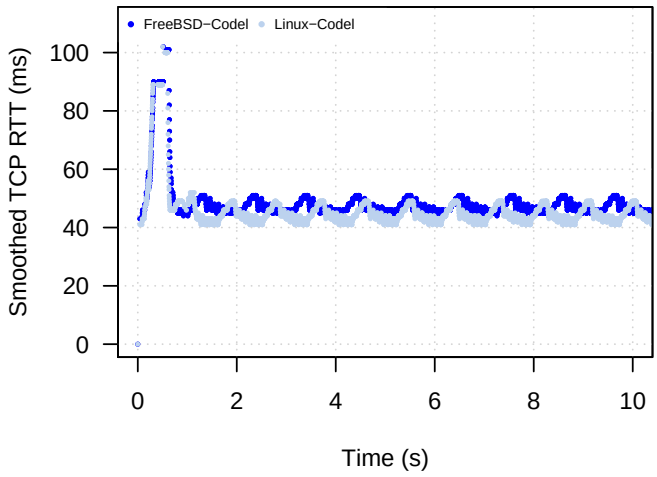
[11] All FQ-CoDel experiments throughputs were calculated using a 1.5 sec window moving forward in steps of 0.3 sec

[12] Whether our FQ-CoDel implementation's behaviour is more *correct* is a separate question

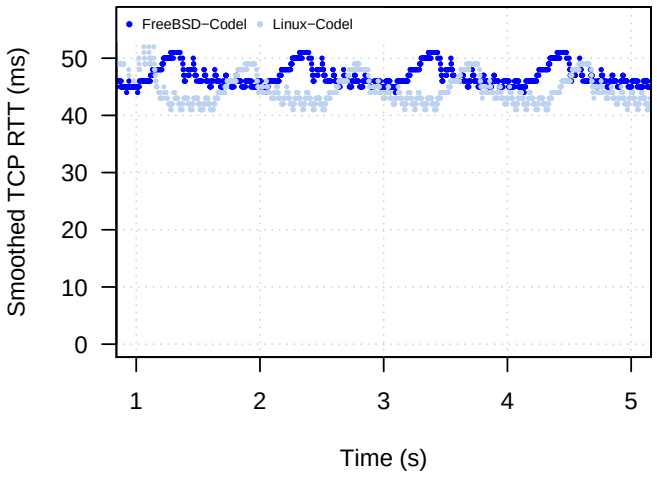


(a) Throughput vs. Time

(b) CWND vs. Time from t=0 to t=5



(c) smoothed TCP RTT vs. Time for first 10 seconds



(d) smoothed TCP RTT vs. Time from t=1 to t=5

Figure 3: One FreeBSD NewReno flow, 40ms RTT path, 10Mbps rate limit and 1000-packet bottleneck buffer. Linux and FreeBSD CoDel configured for target 5ms, interval 100ms, ECN disabled (Scenario 2)

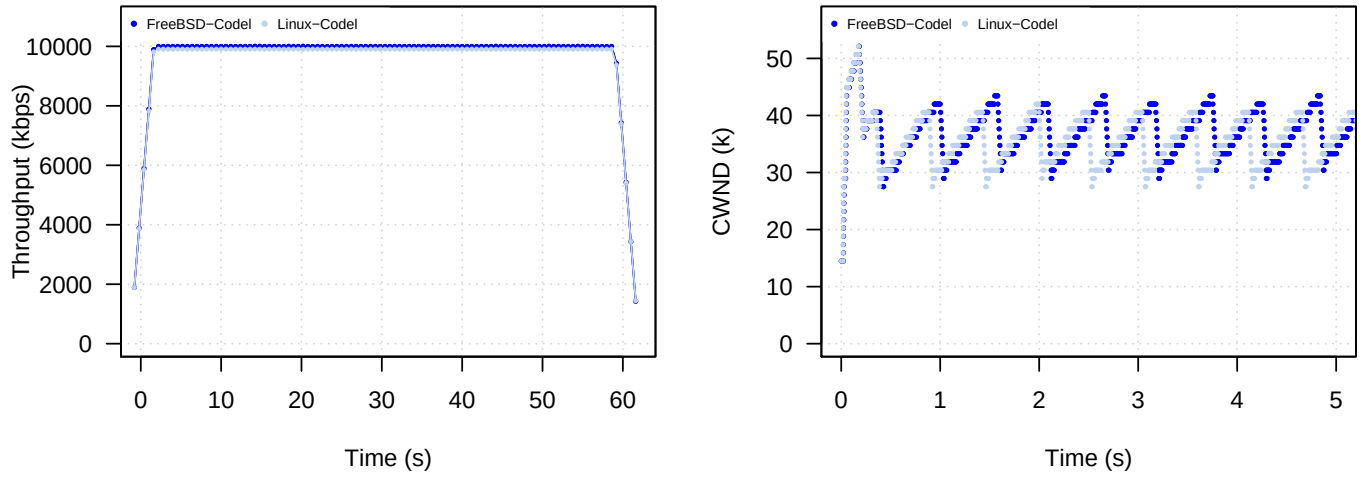
changes to the "auto-tune drop probability" algorithm in [4]. For greater similarity, we set our FreeBSD PIE parameters to match the defaults used by current Linux PIE for all experiments. Even so, FreeBSD PIE showed consistently higher burst tolerance (particularly during TCP slow-start) by following the [4] PIE specification.[13]

In all our PIE comparison experiments we used `iperf` to generate one NewReno (using FreeBSD) or CUBIC (using Linux) TCP flow running for 60 seconds between Hosts 2 and 3 (Figure 1). The router applied independent instances of PIE to 10Mbit/sec bottlenecks in each direction, with each PIE buffer set to 1000 packets. The router also emulated either 20ms or 40ms of underlying path RTT.

Scenario 6: NewReno over PIE @ 20ms RTT

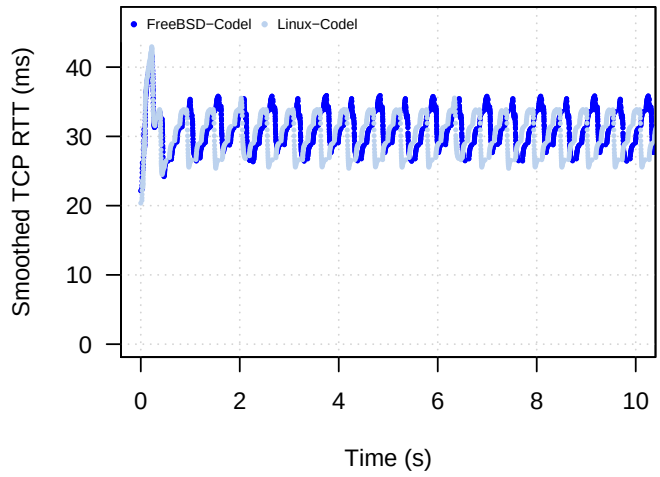
In scenario 6 the end hosts booted into FreeBSD and ran TCP NewReno over a 20ms RTT path. PIE was configured with target 20ms, tupdate 30ms, max_burst 100ms and no ECN. Figure 7 shows throughput, CWND and smoothed TCP RTT versus time for both FreeBSD and Linux implementations of PIE. Allowing for the previously noted increase in burst tolerance, both implementations produce broadly similar results.

[13] Much closer alignment of burst tolerance was seen when we reverted FreeBSD PIE's "auto-tune drop probability" to match the older version used by Linux PIE

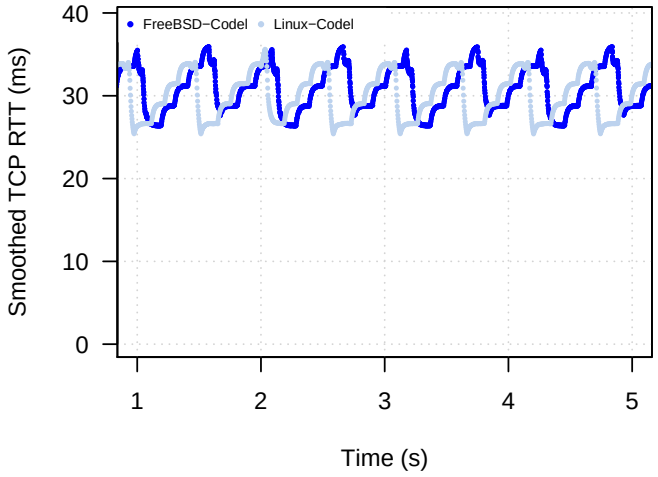


(a) Throughput vs. Time

(b) CWND vs. Time from t=0 to t=5



(c) smoothed TCP RTT vs. Time for first 10 seconds



(d) smoothed TCP RTT vs. Time from t=1 to t=5

Figure 4: One Linux CUBIC flow, 20ms RTT path, 10Mbps rate limit and 1000-packet bottleneck buffer. Linux and FreeBSD CoDel configured for target 10ms, interval 100ms, ECN enabled (Scenario 3)

Scenario 7: NewReno over PIE @ 40ms RTT

Scenario 7 is the same as scenario 6 except that the emulated path RTT is 40 ms. Figure 8 shows throughput, CWND and smoothed TCP RTT versus time, and again the behaviour of our implementation is broadly similar to PIE under Linux (aside from the increased burst tolerance).

Scenario 8: CUBIC over PIE @ 20ms RTT

In scenario 8 the end hosts booted into Linux and ran TCP CUBIC over a 20ms RTT path. PIE was configured with target 20ms, tupdate 30ms, max_burst 100ms and ECN *enabled*. We checked number of dropped packet and confirmed that ECN is functional with nothing dropped during the experiment. Figure 9 shows throughput, CWND, smoothed TCP RTT versus time, and again the behaviour of our implementation is broadly similar to PIE under Linux (aside from the increased burst tolerance).

IX. FreeBSD FQ-PIE

There is no official Linux FQ-PIE implementation to compare with, so here we illustrate that our definition FQ-PIE exhibits plausibly expected and useful behaviour. We used multiple instances of `iperf` under Linux to generate four TCP CUBIC flows with staggered start and end times (starting at t=0, 10, 20, 30 seconds and each flow lasting for 60 seconds). Each instance of

FQ-PIE was configured for target 15ms, tupdate 15ms, max_burst 100ms, no ECN, quantum 1514 bytes and with 10240 packets of bottleneck buffering shared by 1024 fq_pie sub-queues.

Scenario 9: 10Mbit/sec bottleneck

In scenario 9 the bottleneck was configured for 10Mbit/sec. Figure 10 shows throughput[14], CWND and smoothed TCP RTT versus time for the experiment. In this scenario FreeBSD FQ-PIE achieves good throughput, capacity sharing and reasonable queueing delay.

Scenario 10: 1Mbit/sec bottleneck

Scenario 10 repeats scenario 9 but with a 1Mbit/sec bottleneck. Figure 11 shows throughput, CWND and smoothed TCP RTT versus time for the experiment. Even though the RTT spikes are higher than Scenario 9 (due to the lower 1Mbit/sec bottleneck speed), FreeBSD FQ-PIE achieves good throughput, capacity sharing and reasonable queueing delay.

X. Conclusions and Future Work

This report focuses on Dummynet AQM v0.2 – our FreeBSD implementation of CoDel,FQ-CoDel, PIE and FQ-PIE in ipfw/dummynet framework. We summarise the AQM and Dummynet context, provide instructions for applying the patch, and present preliminary experimental results. Our independent implementations of [3], [5] and [4] behave similar to equivalent Linux kernel 4.1 implementations. In addition, we demonstrate that our definition and implementation of FQ-PIE seems to have potential as a PIE-based alternative to FQ-CoDel.

Our experimental testing has been limited, aiming primarily to confirm plausible similarity in behaviours and (indirectly) confirm the clarity of the current Internet Drafts. The unexpectedly good capacity sharing exhibited by our FreeBSD implementation (relative to Linux FQ-CoDel) when faced with low (1Mbps) bottleneck rates is a topic for future investigation.

XI. Acknowledgements

This project has been made possible in part by a gift from the Comcast Innovation Fund.

[14] All FQ-PIE experiments throughputs were calculated using a 1.5 sec window moving forward in steps of 0.3 sec

References

[1] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *Networking, IEEE/ACM Transactions on*, vol. 1, no. 4, pp. 397–413, Aug 1993.

[2] K. Nichols and V. Jacobson, "A modern aqm is just one piece of the solution to bufferbloat," *ACM Queue Networks*, vol. 10, no. 5, 2012. [Online]. Available: http://queue.acm.org/detail.cfm?id=2209336

[3] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," IETF Draft, March 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-codel-03

[4] R. Pan, P. Natarajan, F. Baker, G. White, B. VerSteeg, M. Prabhu, C. Piglione, and V. Subramanian, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem," April 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-pie-06

[5] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "FlowQueue-Codel," IETF Draft, March 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06

[6] *Bufferbloat Wiki*. [Online]. Available: http://www.bufferbloat.net/projects/codel/wiki

[7] V. Subramanian, *"PIE AQM scheme"*, Kernel commit log, Jan 2014. [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=d4b36210c2e6ecef0ce52fb6c18c51144f5c2d88

[8] R. Al-Saadi and G. Armitage, "Implementing AQM in FreeBSD." [Online]. Available: http://caia.swin.edu.au/freebsd/aqm

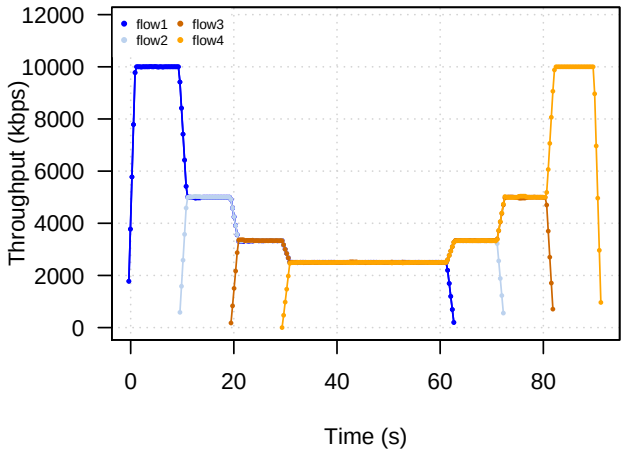
[9] R. Al-Saadi and G. Armitage, "Dummynet AQM v0.1 - CoDel and FQ-CoDel for FreeBSD's ipfw/dummynet framework," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160226A, 26 February 2016. [Online]. Available: http://caia.swin.edu.au/reports/160226A/CAIA-TR-160226A.pdf

[10] *IPFW - FreeBSD Handbook*. The FreeBSD Documentation Project, 2015. [Online]. Available: https://www.freebsd.org/doc/handbook/firewalls-ipfw.html

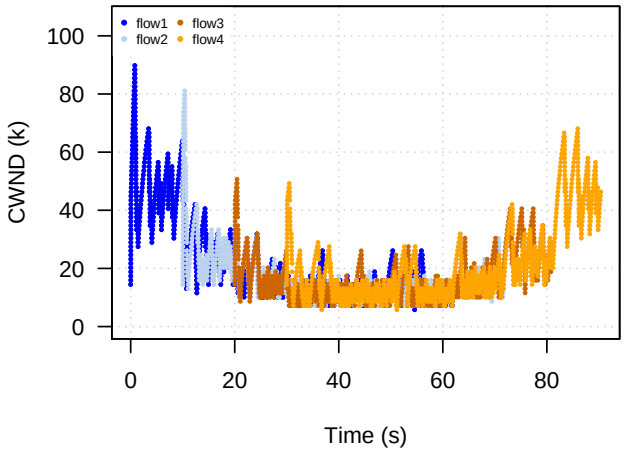
[11] M. Carbone and L. Rizzo, "Dummynet revisited," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 12–20, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1764873.1764876

[12] *IPFW(8)*, FreeBSD System Manager's Manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?ipfw(8)

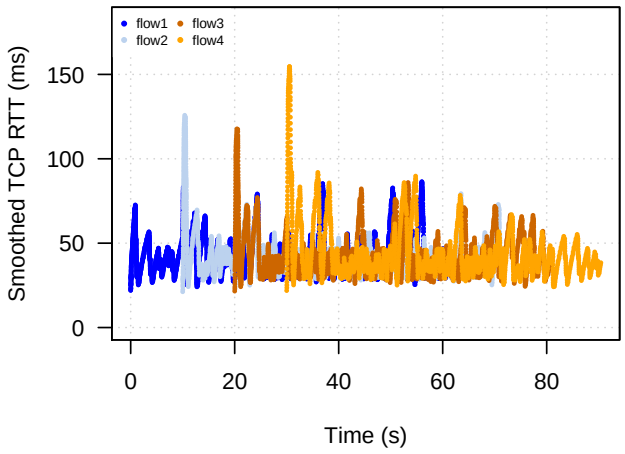
[13] S. Zander and G. Armitage, "TEACUP v1.0 - A System for Automated TCP Testbed Experiments," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 150529A, 2015. [Online]. Available: http://caia.swin.edu.au/reports/150529A/CAIA-TR-150529A.pdf



(a) Throughput vs. Time

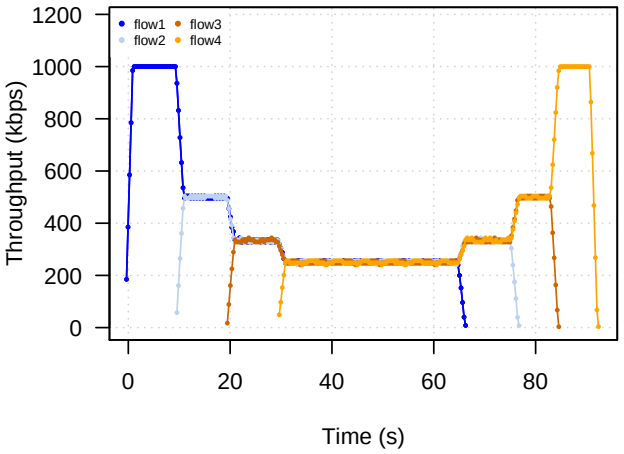


(b) CWND vs. Time

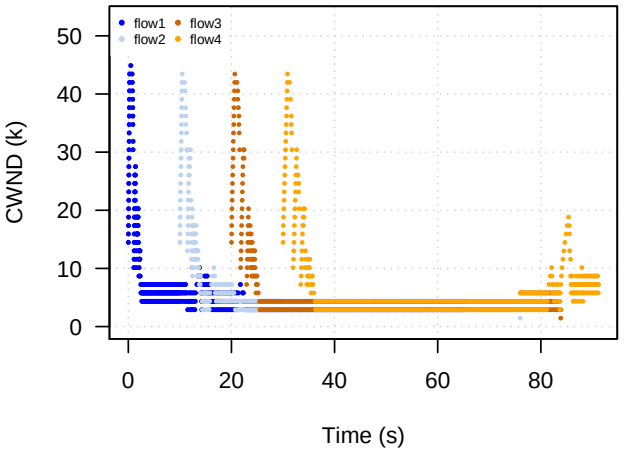


(c) smoothed TCP RTT vs. Time

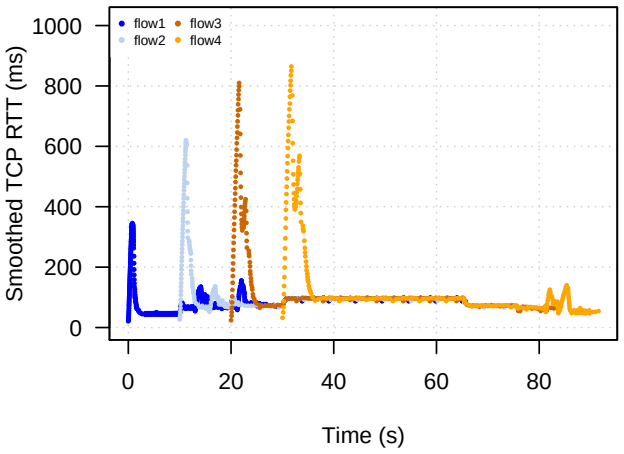
Figure 10: Four Linux CUBIC flows, 20ms RTT path, 10Mbps limit. FreeBSD FQ-PIE configured for target 15ms, tupdate 15ms max_burst 150ms, 1024 queues, quantum 1514 bytes, 10240-packet buffering, ECN disabled (Scenario 9)



(a) Throughput vs. Time

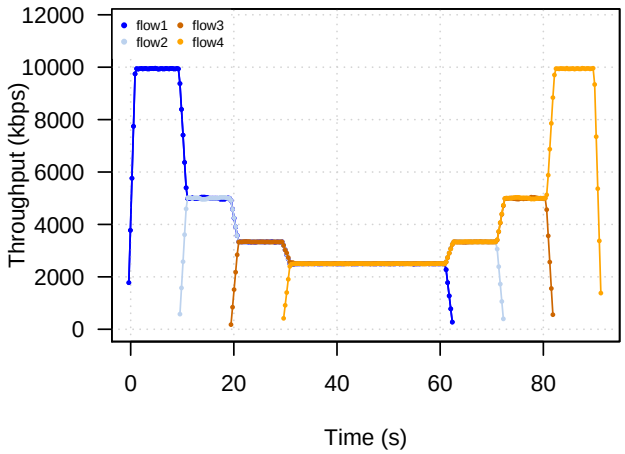


(b) CWND vs. Time

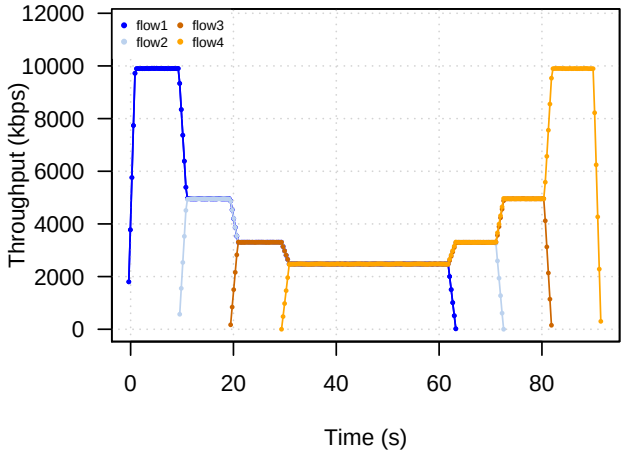


(c) smoothed TCP RTT vs. Time

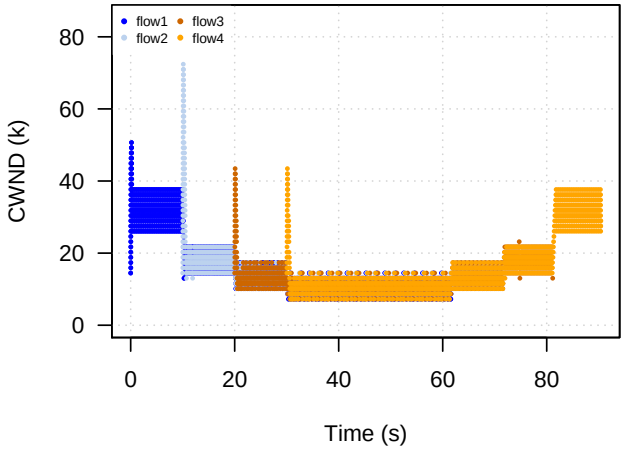
Figure 11: Four Linux CUBIC flows, 20ms RTT path, 1Mbps limit. FreeBSD FQ-PIE configured for target 15ms, tupdate 15ms max_burst 150ms, 1024 queues, quantum 1514 bytes, 10240-packet buffering, ECN disabled (Scenario 10)



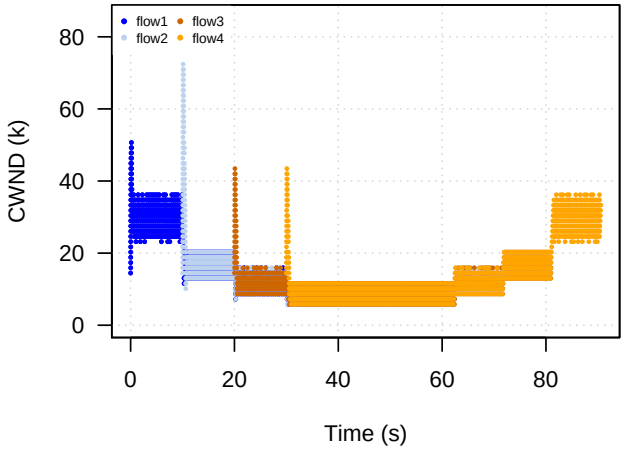
(a) FreeBSD fq_codel Throughput vs. Time



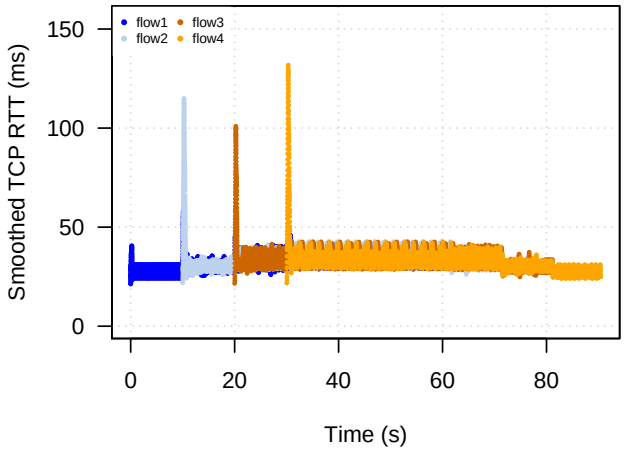
(b) Linux fq_codel Throughput vs. Time



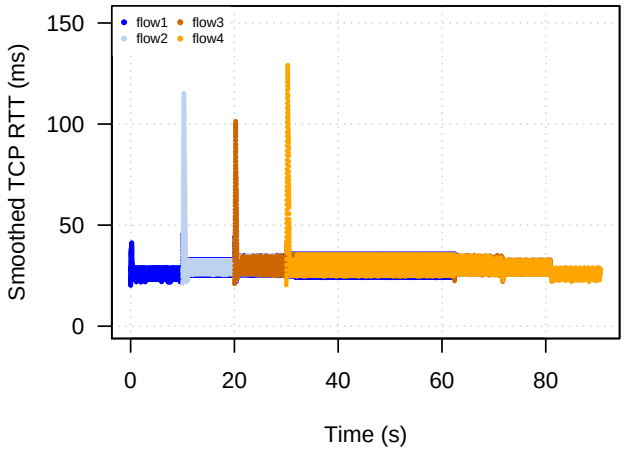
(c) FreeBSD fq_codel CWND vs. Time



(d) Linux fq_codel CWND vs. Time

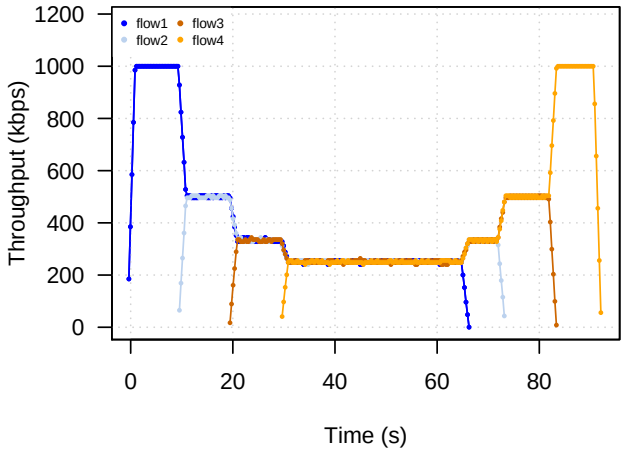


(e) FreeBSD fq_codel smoothed TCP RTT vs. Time

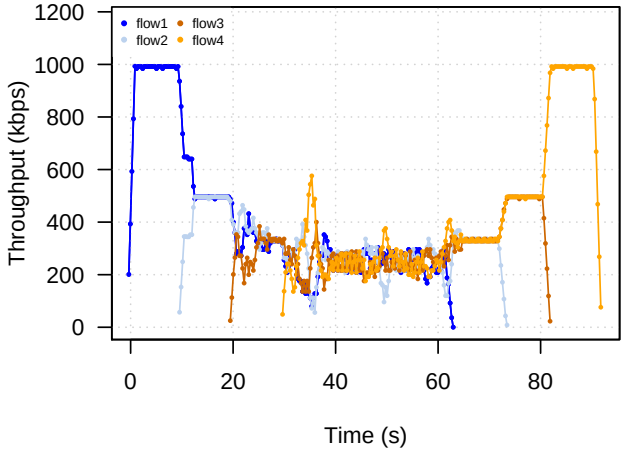


(f) Linux fq_codel smoothed TCP RTT vs. Time

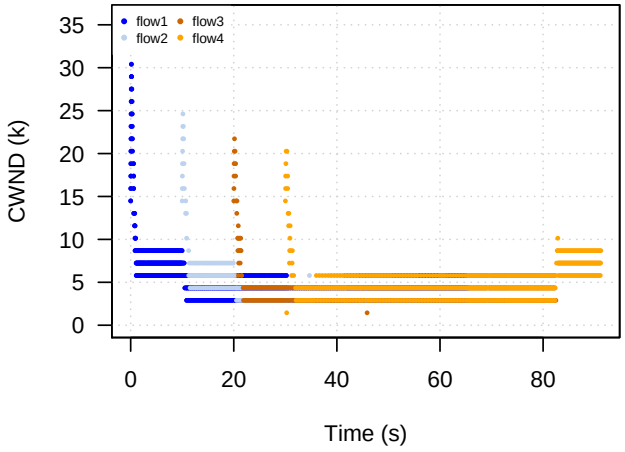
Figure 5: Four Linux CUBIC flows, 20ms RTT path, 10Mbps limit. Both FQ-CoDels configured for target 5ms, interval 100ms, 1024 queues, quantum 1514 bytes, 10240-packet buffering, ECN disabled (Scenario 4)



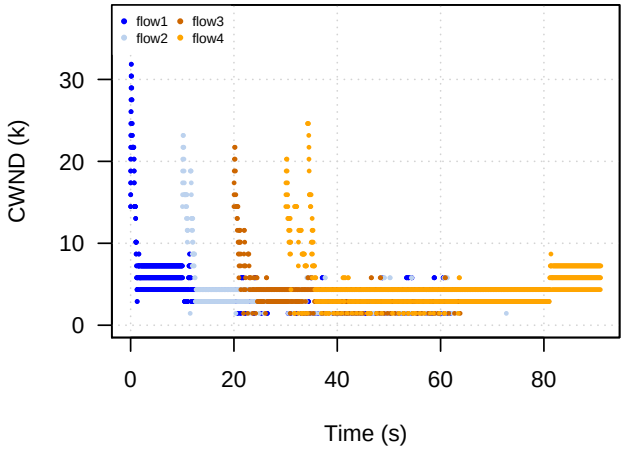
(a) FreeBSD fq_codel Throughput vs. Time



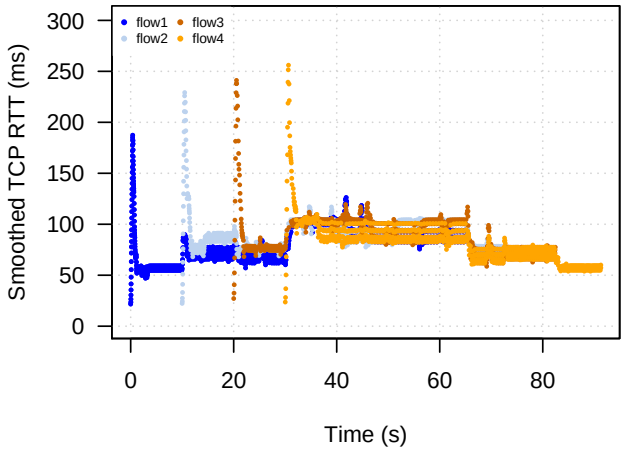
(b) Linux fq_codel Throughput vs. Time



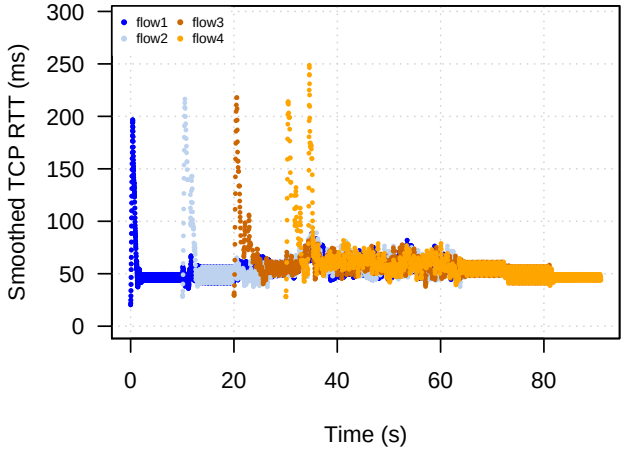
(c) FreeBSD fq_codel CWND vs. Time



(d) Linux fq_codel CWND vs. Time

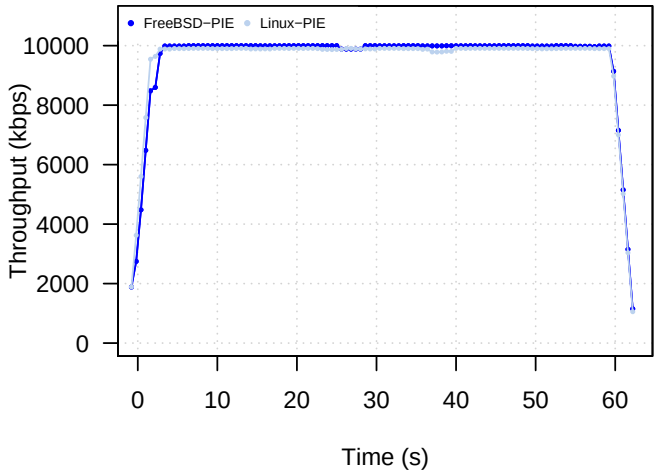


(e) FreeBSD fq_codel smoothed TCP RTT vs. Time

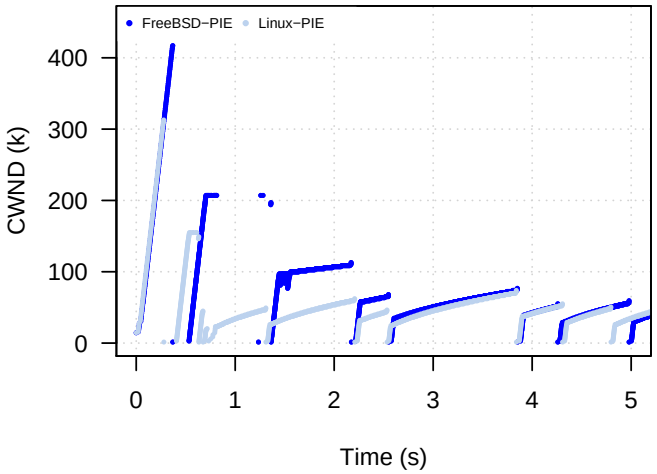


(f) Linux fq_codel smoothed TCP RTT vs. Time

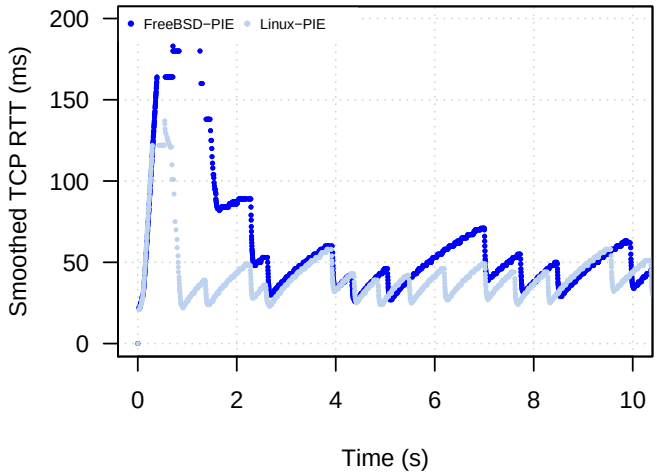
Figure 6: Four Linux CUBIC flows, 20ms RTT path, 1Mbps limit. Both FQ-CoDels configured for target 5ms, interval 100ms, 1024 queues, quantum 1514 bytes, 10240-packet buffering, ECN disabled (Scenario 5)



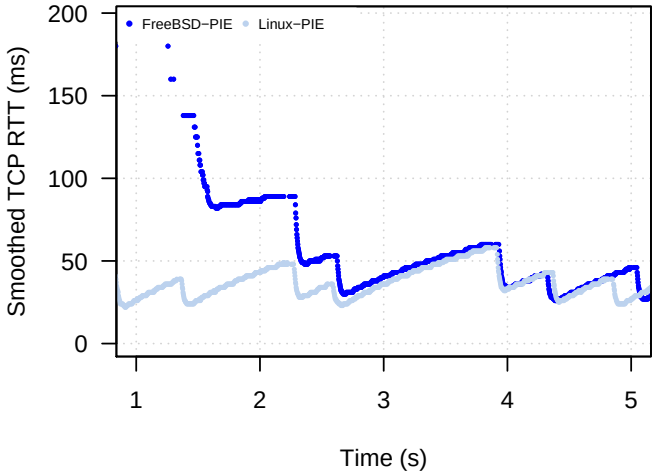
(a) Throughput vs. Time



(b) CWND vs. Time from t=0 to t=5

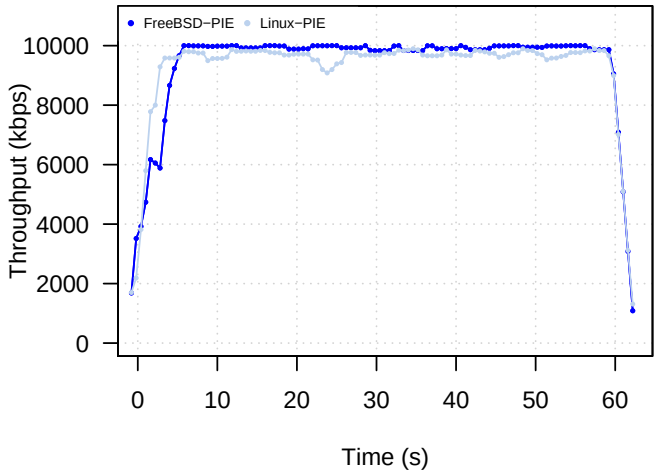


(c) smoothed TCP RTT vs. Time for first 10 seconds

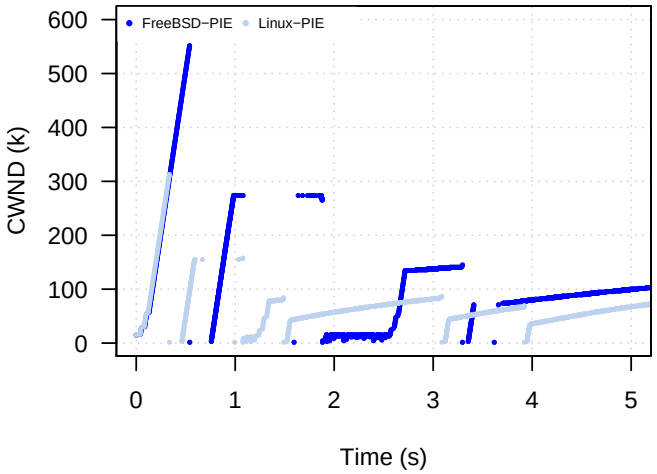


(d) smoothed TCP RTT vs. Time from t=1 to t=5

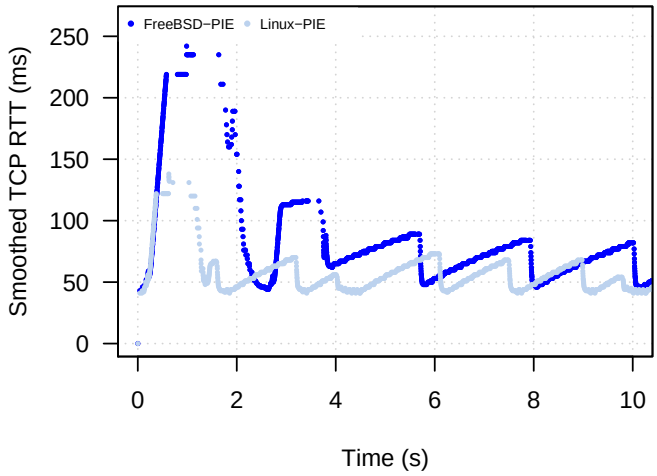
Figure 7: One FreeBSD NewReno flow, 20ms RTT path, 10Mbps rate limit and 1000-packet bottleneck buffer. Linux and FreeBSD PIE configured for target 20ms, tupdate 30ms, max_burst 100ms, ECN disabled (Scenario 6)



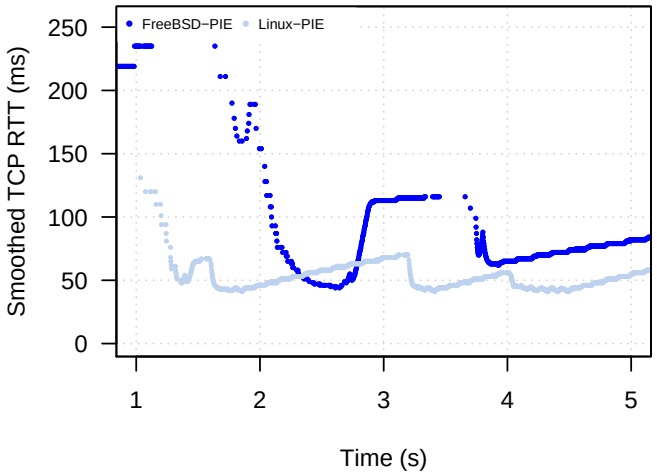
(a) Throughput vs. Time



(b) CWND vs. Time from t=0 to t=5

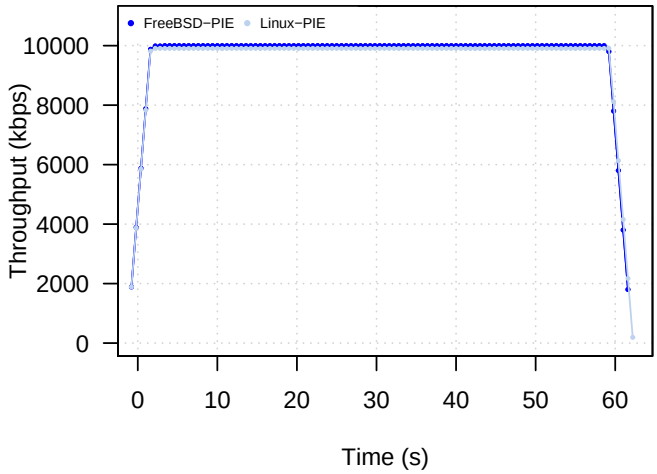


(c) smoothed TCP RTT vs. Time for first 10 seconds

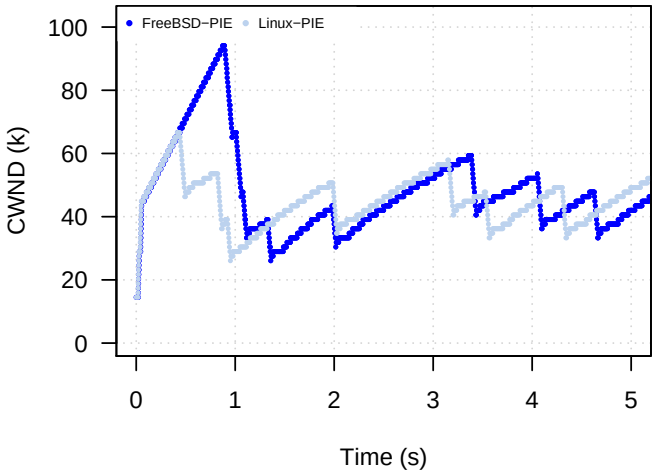


(d) smoothed TCP RTT vs. Time from t=1 to t=5

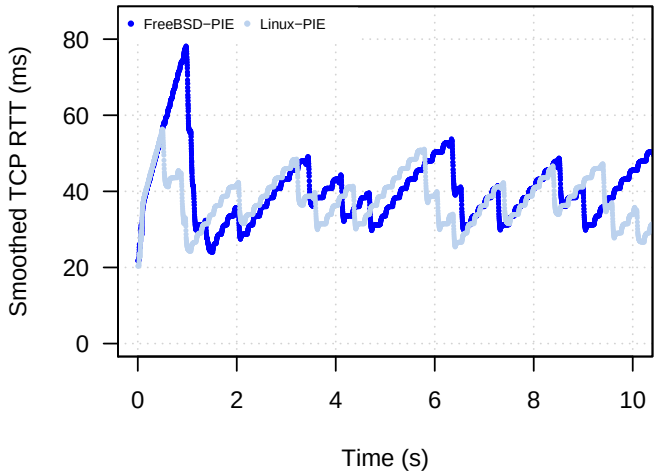
Figure 8: One FreeBSD NewReno flow, 40ms RTT path, 10Mbps rate limit and 1000-packet bottleneck buffer. Linux and FreeBSD PIE configured for target 20ms, tupdate 30ms, max_burst 100ms, ECN disabled (Scenario 7)



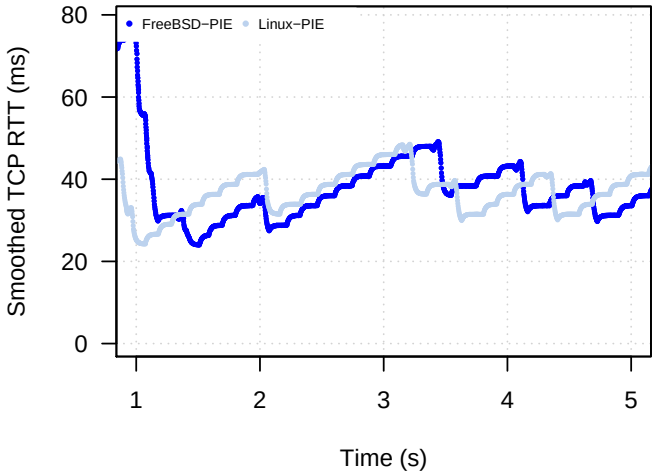
(a) Throughput vs. Time



(b) CWND vs. Time from t=0 to t=5



(c) smoothed TCP RTT vs. Time for first 10 seconds



(d) smoothed TCP RTT vs. Time from t=1 to t=5

Figure 9: One Linux CUBIC flow, 20ms RTT path, 10Mbps rate limit and 1000-packet bottleneck buffer. Linux and FreeBSD PIE configured for target 10ms, tupdate 30ms, interval 20ms, ECN enabled (Scenario 8)