AQM and Packet Scheduling
Internet-Draft
Intended status: Informational
Expires: September 4, 2014

T. Hoeiland-Joergensen
Karlstad University
P. McKenney
IBM Linux Technology Center
D. Taht
Teklibre
J. Gettys
Bell Labs
E. Dumazet
Google, Inc.

March 3, 2014

FlowQueue-Codel
draft-hoeiland-joergensen-aqm-fq-codel-00

Abstract

This memo presents the FQ-CoDel hybrid packet scheduler/AQM algorithm, a critical tool for fighting bufferbloat and reducing latency across the Internet.

FQ-CoDel mixes packets from multiple flows and reduces the impact of head of line blocking from bursty traffic. It provides isolation for low-rate traffic such as DNS, web, and videoconferencing traffic. It improves utilisation across the networking fabric, especially for bidirectional traffic, by keeping queue lengths short; and it can be implemented in a memory- and CPU-efficient fashion across a wide range of hardware.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 4, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 3
1.1. Terminology and concepts 3
1.2. Informal summary of FQ-CoDel 4
2. CoDel 5
3. Flow Queueing 5
4. FQ-CoDel Parameters and Data Structures 6
4.1. Parameters 6
4.1.1. Interval 6
4.1.2. Target 6
4.1.3. Packet limit 7
4.1.4. Quantum 7
4.1.5. Flows 7
4.1.6. ECN 7
4.2. Data structures 8
4.2.1. Internal sub-queues 8
4.2.2. New and old queues lists 8
5. The FQ-CoDel scheduler and AQM interactions 8
5.1. Enqueue 8
5.2. Dequeue 9
6. Implementation considerations 11
6.1. Probability of hash collisions 11
6.2. Memory Overhead 11
6.3. Per-Packet Timestamping 12
6.4. Other forms of "Fair Queueing" 12
6.5. Differences between CoDel and FQ-CoDel behaviour 12
7. Security Considerations 13

8. IANA Considerations 13
9. Acknowledgements 13
10. Conclusions 13
11. References 13
11.1. Normative References 14

11.2. Informative References 14
11.3. URIs 14
Authors' Addresses 14

1. Introduction

The FQ-CoDel algorithm is a combined packet scheduler and AQM developed as part of the bufferbloat-fighting community effort. It is based on a modified Deficit Round Robin (DRR) queue scheduler, with the CoDel AQM algorithm operating on each sub-queue. This document describes the combined algorithm; reference implementations are available for ns2 and ns3 and it is included in the mainline Linux kernel as the FQ-CoDel queueing discipline.

The rest of this document is structured as follows: The rest of this section gives some concepts and terminology used in the rest of the document, and gives a short informal summary of the FQ-CoDel algorithm. Section 2 gives an overview of the CoDel algorithm. Section 3 covers the DRR portion. Section 4 defines the parameters and data structures employed by FQ-CoDel. Section 5 describes the working of the algorithm in detail. Section 6 describes implementation considerations. Section 10 concludes.

1.1. Terminology and concepts

Flow: A flow is typically identified by a 5-tuple of source IP, destination IP, source port, destination port, and protocol. It can also be identified by a superset or subset of those parameters, or by mac address, or other means.

Queue: A queue of packets represented internally in FQ-CoDel. In most instances each flow gets its own queue; however because of the possibility of hash collisions, this is not always the case. In an attempt to avoid confusion, the word 'queue' is used to refer to the internal data structure, and 'flow' to refer to the actual stream of packets being delivered to the FQ-CoDel algorithm.

Scheduler: A mechanism to select which queue a packet is dequeued from.

CoDel AQM: The Active Queue Management algorithm employed by FQ-CoDel.

DRR: Deficit round-robin scheduling.

Quantum: The maximum amount of bytes to be dequeued from a queue at once.

1.2. Informal summary of FQ-CoDel

FQ-CoDel is a _hybrid_ of DRR [DRR] and CODEL [CODEL2012], with an optimisation for sparse flows similar to SQF [SQF2012]. We call this "Flow Queueing" rather than "Fair Queueing" as flows that build a queue are treated differently than flows that do not.

FQ-CoDel stochastically classifies incoming packets into different sub-queues by hashing the 5-tuple of IP protocol number and source and destination IP and port numbers, perturbed with a random number selected at initiation time. Each queue is managed by the CoDel queueing discipline. Packet ordering within a queue is preserved, since queues have FIFO ordering.

The FQ-CoDel algorithm consists of two logical parts: the scheduler which selects which queue to dequeue a packet from, and the CoDel AQM which works on each of the queues. The subtleties of FQ-CoDel are mostly in the scheduling part, whereas the interaction between the scheduler and the CoDel algorithm are fairly straight forward:

At initialisation, each queue is set up to have a separate set of CoDel state variables. By default, 1024 queues are created. The current implementation supports anywhere from one to 64K separate queues, and each queue maintains the state variables throughout its lifetime, and so acts the same as the non-FQ CoDel variant would. This means that with only one queue, FQ-CoDel behaves essentially the same as CoDel by itself.

On dequeue, FQ-CoDel selects a queue from which to dequeue by a two-

tier round-robin scheme, in which each queue is allowed to dequeue up to a configurable quantum of bytes for each iteration. Deviations from this quantum is maintained as a deficit for the queue, which serves to make the fairness scheme byte-based rather than a packet-based. The two-tier round-robin mechanism distinguishes between "new" queues (which don't build up a standing queue) and "old" queues, that have queued enough data to be around for more than one iteration of the round-robin scheduler.

This new/old queue distinction has a particular consequence for queues that don't build up more than a quantum of bytes before being visited by the scheduler: Such queues are removed from the list, and then re-added as a new queue each time a packet arrives for it, and so will get priority over queues that do not empty out each round (except for a minor modification to protect against starvation, detailed below). Exactly how much data a flow has to send to keep its queue in this state is somewhat difficult to reason about, because it depends on both the egress link speed and the number of concurrent flows. However, in practice many things that are

beneficial to have prioritised for typical internet use (ACKs, DNS lookups, interactive SSH, HTTP requests, ARP, ICMP, VoIP) _tend_ to fall in this category, which is why FQ-CoDel performs so well for many practical applications.

This scheduling scheme has some subtlety to it, which is explained in detail in the remainder of this document.

2. CoDel

CoDel is described in the the ACM Queue paper, CODEL [CODEL2012], and Van Jacobson's IETF84 presentation [IETF84]. The basic idea is to control queue length, maintaining sufficient queueing to keep the outgoing link busy, but avoiding building up the queue beyond that point. This is done by preferentially dropping packets that remain in the queue for "too long".

When each new packet arrives, its arrival time is stored with it. Later, when it is that packet's turn to be dequeued, CoDel computes its sojourn time (the current time minus the arrival time). If the sojourn time for packets being dequeued exceeds the _target_ time for a time period of at least the (current value of) _interval_, one or

more packets will be dropped (or marked, if ECN is enabled) in order to signal the source endpoint to reduce its send rate. If the sojourn still remains above the target time, the value of interval be lowered, and additional packet drops will occur on a schedule computed from an inverse-square-root control law until either (1) the queue becomes empty or (2) a packet is encountered with a sojourn time that is less than the target time. Upon exiting the dropping mode, CoDel caches the last calculated interval (applying varying amounts of hysteresis to it), to be used as the starting point on subsequent re-entries into dropping mode.

The _target_ time is normally set to about five milliseconds, and the initial _interval_ is normally set to about 100 milliseconds. This approach has proven to be quite effective in a wide variety of situations.

CoDel drops packets at the head of a queue, rather than at the tail.

3. Flow Queueing

FQ-CoDel's DRR scheduler is byte-based, employing a deficit round-robin mechanism between queues. This works by keeping track of the current byte _deficit_ of each queue. This deficit is initialised to the configurable quantum; each time a queue gets a dequeue opportunity, it gets to dequeue packets, decreasing the deficit by the packet size for each packet, until the deficit runs into the

negative, at which point it is increased by one quantum, and the dequeue opportunity ends.

This means that if one queue contains packets of size quantum/3, and another contains quantum-sized packets, the first queue will dequeue three packets each time it gets a turn, whereas the second only dequeues one. This means that flows that send small packets are not penalised by the difference in packet sizes; rather, the DRR scheme approximates a (single-)byte-based fairness queueing. The size of the quantum determines the scheduling granularity, with the tradeoff from too small a quantum being scheduling overhead. For small bandwidths, lowering the quantum from the default MTU size can be advantageous.

Unlike DRR there are two sets of flows - a "new" list for flows that

have not built a queue recently, and an "old" list for flow-building queues.

4. FQ-CoDel Parameters and Data Structures

This section goes into the parameters and data structures in FQ-CoDel.

4.1. Parameters

4.1.1. Interval

The _interval_ parameter has the same semantics as CoDel and is used to ensure that the measured minimum delay does not become too stale. The minimum delay MUST be experienced in the last epoch of length interval. It SHOULD be set on the order of the worst-case RTT through the bottleneck to give end-points sufficient time to react.

The default interval value is 100 ms.

4.1.2. Target

The _target_ parameter has the same semantics as CoDel. It is the acceptable minimum standing/persistent queue delay for each FQ-CoDel Queue. This minimum delay is identified by tracking the local minimum queue delay that packets experience.

The default target value is 5 ms, but this value SHOULD be tuned to be at least the transmission time of a single MTU-sized packet at the prevalent egress link speed (which for e.g. 1Mbps and MTU 1500 is ~15ms).

4.1.3. Packet limit

Routers do not have infinite memory, so some packet limit MUST be enforced.

The _limit_ parameter is the hard limit on the real queue size, measured in number of packets. This limit is a global limit on the number of packets in all queues; each individual queue does not have

an upper limit. When the limit is reached and a new packet arrives for enqueue, a packet is dropped from the head of the largest queue (measured in bytes) to make room for the new packet.

The default packet limit is 10240 packets, which is suitable for up to 10GigE speeds.

4.1.4. Quantum

The _quantum_ parameter is the number of bytes each queue gets to dequeue on each round of the scheduling algorithm. The default is set to 1514 bytes which corresponds to the Ethernet MTU plus the hardware header length of 14 bytes.

In TSO-enabled systems, where a "packet" consists of an offloaded packet train, it can presently be as large as 64K bytes. In GRO-enabled systems, up to 17 times the TCP max segment size (or 25K bytes).

4.1.5. Flows

The _flows_ parameter sets the number of sub-queues into which the incoming packets are classified. Due to the stochastic nature of hashing, multiple flows may end up being hashed into the same slot.

This parameter can be set only at load time since memory has to be allocated for the hash table in the current implementation.

The default value is 1024.

4.1.6. ECN

ECN is _enabled_ by default. Rather than do anything special with misbehaved ECN flows, FQ-CoDel relies on the packet scheduling system to minimise their impact, thus unresponsive packets in a flow being marked with ECN can grow to the overall packet limit, but will not otherwise affect the performance of the system.

It can be disabled by specifying the _noecn_ parameter.

4.2. Data structures

4.2.1. Internal sub-queues

The main data structure of FQ-CoDel is the array of sub-queues, which is instantiated to the number of queues specified by the _flows_ parameter at instantiation time. Each sub-queue consists simply of an ordered list of packets with FIFO semantics, two state variables tracking the queue deficit and total number of bytes enqueued, and the set of CoDel state variables. Other state variables to track queue statistics can also be included: for instance, the Linux implementation keeps a count of dropped packets.

Queue space is shared: there's a global limit on the number of packets the queues can hold, but not one per queue.

4.2.2. New and old queues lists

FQ-CoDel maintains two lists of active queues, called "new" and "old" queues. Each list is an ordered list containing references to the array of sub-queues. When a packet is added to a queue that is not currently active, that queue becomes active by being added to the list of new queues. Later on, it is moved to the list of old queues, from which it is removed when it is no longer active. This behaviour is the source of some subtlety in the packet scheduling at dequeue time, explained below.

5. The FQ-CoDel scheduler and AQM interactions

This section describes the operation of the FQ-CoDel scheduler and AQM. It is split into two parts explaining the enqueue and dequeue operations.

5.1. Enqueue

The packet enqueue mechanism consists of three stages: classification into a sub-queue, timestamping and bookkeeping, and optionally dropping a packet when the total number of enqueued packets goes over the maximum.

When a packet is enqueued, it is first classified into the appropriate sub-queue. By default, this is done by hashing on the 5-tuple of IP protocol, and source and destination IP and port numbers, permuted by a random value selected at initialisation time, and taking the hash value modulo the number of sub-queues. However, an implementation MAY also specify a configurable classification scheme along a wide variety of other possible parameters such as mac

address, diffserv, firewall and flow specific markings, etc. (the Linux implementation does so in the form of the 'tc filter' command).

If a custom filter fails, classification failure results in the packet being dropped and no further action taken. By design the standard filter cannot fail.

Additionally, the default hashing algorithm presently deployed does decapsulation of some common packet types (6in4, IPIP, GRE 0), mixes IPv6 IP addresses thoroughly, and uses Jenkins hash on the result.

Once the packet has been successfully classified into a sub-queue, it is handed over to the CoDel algorithm for timestamping. It is then added to the tail of the selected queue, and the queue's byte count is updated by the packet size. Then, if the queue is not currently active (i.e. if it is not in either the list of new or the list of old queues), it is added to the end of the list of new queues, and its deficit is initiated to the configured quantum. Otherwise it is added to the old queue list.

Finally, the total number of enqueued packets is compared with the configured limit, and if it is _above_ this value (which can happen since a packet was just enqueued), a packet is dropped from the head of the queue with the largest current byte count. Note that this in most cases means that the packet that gets dropped is different from the one that was just enqueued, and may even be from a different queue.

5.2. Dequeue

Most of FQ-CoDel's work is done at packet dequeue time. It consists of three parts: selecting a queue from which to dequeue a packet, actually dequeuing it (employing the CoDel algorithm in the process), and some final bookkeeping.

For the first part, the scheduler first looks at the list of new queues; for each queue in that list, if that queue has a negative deficit (i.e. it has already dequeued at least a quantum of bytes), its deficit is increased by one quantum, and the queue is put onto the end of the list of old queues, and the routine selects the next queue and starts again.

Otherwise, that queue is selected for dequeue. If the list of new queues is empty, the scheduler proceeds down the list of old queues in the same fashion (checking the deficit, and either selecting the queue for dequeuing, or increasing the deficit and putting the queue

back at the end of the list).

After having selected a queue from which to dequeue a packet, the CoDel algorithm is invoked on that queue. This applies the CoDel control law, and may discard one or more packets from the head of that queue, before returning the packet that should be dequeued (or nothing if the queue is or becomes empty while being handled by the CoDel algorithm).

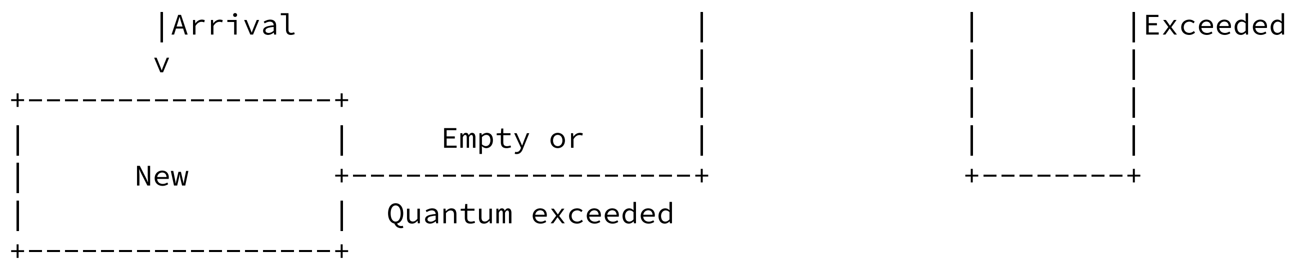
Finally, if the CoDel algorithm did not return a packet, the queue is empty, and the scheduler does one of two things: if the queue selected for dequeue came from the list of new queues, it is moved to the end of the list of old queues. If instead it came from the list of old queues, that queue is removed from the list, to be added back (as a new queue) the next time a packet arrives that hashes to that queue. Then (since no packet was available for dequeue), the whole dequeue process is restarted from the beginning.

If, instead, the scheduler _did_ get a packet back from the CoDel algorithm, it updates the byte deficit for the selected queue before returning the packet as the result of the dequeue operation.

The step that moves an empty queue from the list of new queues to the list of old queues before it is removed is crucial to prevent starvation. This is because otherwise the queue can reappear (the next time a packet arrives for it) before the list of old queues is visited; this can go on indefinitely even with a small number of active flows, if the flow providing packets to the queue in question transmits at just the right rate. This is prevented by first moving the queue to the list of old queues, forcing a pass through that, and thus preventing starvation.

The resulting migration of queues between the different states is summarised in the following state diagram:

```
+-----------------+                +--------------------+
|                 |     Empty      |                    |
|     Empty       |<---------------+       Old          +-----+
|                 |                |                    |     |
+-------+---------+                +--------------------+     |
        |                             ^              ^        |Quantum
```



6. Implementation considerations

6.1. Probability of hash collisions

Since the Linux FQ-CoDel implementation by default uses 1024 hash buckets, the probability that (say) 100 VoIP sessions will all hash to the same bucket is something like ten to the power of minus 300. Thus, the probability that at least one of the VoIP sessions will hash to some other queue is very high indeed.

Conversely, the probability that each of the 100 VoIP sessions will get its own queue is given by (1023!/(924!*1024^99)) or about 0.007; so not all that probable. The probability rises sharply, however, if we are willing to accept a few collisions. For example, there is about an 86% probability that no more than two of the 100 VoIP sessions will be involved in any given collision, and about a 99% probability that no more than three of the VoIP sessions will be involved in any given collision. These last two results were computed using Monte Carlo simulations: Oddly enough, the mathematics for VoIP-session collision exactly matches that of hardware cache overflow.

6.2. Memory Overhead

FQ-CoDel can be implemented with a very low memory footprint (less than 64 bytes per queue on 64 bit systems). These are the data structures used in the Linux implementation:

```
struct codel_vars {
    u32                count;
    u32                lastcount;
    bool               dropping;
```

```
    u16               rec_inv_sqrt;
    codel_time_t      first_above_time;
    codel_time_t      drop_next;
    codel_time_t      ldelay;
};

struct fq_codel_flow {
    struct sk_buff    *head;
    struct sk_buff    *tail;
    struct list_head  flowchain;
    int               deficit;
    u32               dropped; /* number of drops (or ECN marks) on this flow *
    struct codel_vars cvars;
};
```

The master table managing all queues looks like this:


```
  struct fq_codel_sched_data {
      struct tcf_proto *filter_list;  /* optional external classifier */
      struct fq_codel_flow *flows;    /* Flows table [flows_cnt] */
      u32              *backlogs;     /* backlog table [flows_cnt] */
      u32              flows_cnt;     /* number of flows */
      u32              perturbation;  /* hash perturbation */
      u32              quantum;       /* psched_mtu(qdisc_dev(sch)); */
      struct codel_params cparams;
      struct codel_stats cstats;
      u32              drop_overlimit;
      u32              new_flow_count;

      struct list_head new_flows;     /* list of new flows */
      struct list_head old_flows;     /* list of old flows */
  };
```

6.3. Per-Packet Timestamping

The CoDel portion of the algorithm requires per-packet timestamps be stored along with the packet. While this approach works well for software-based routers, it cannot be easily retrofitted to devices that do most of their processing in silicon.

Also, timestamping functions in the core OS need to be very efficient.

6.4. Other forms of "Fair Queueing"

Much of the scheduling portion of FQ-CoDel is derived from DRR. SFQ-based versions have also been produced and tested in ns2. Other forms of Fair Queueing, such as WFQ or QFQ, have not been thoroughly explored.

6.5. Differences between CoDel and FQ-CoDel behaviour

CoDel can be applied to a single queue system as a straight AQM, where it converges towards an "ideal" drop rate (i.e. one that minimises delay while keeping a high link utilisation), and then optimises around that control point.

The scheduling of FQ-CoDel mixes packets of competing flows, which acts to pace bursty flows to better fill the pipe. Additionally, a new flow gets substantial "credit" over other flows until CoDel finds an ideal drop rate for it. However, for a new flow that exceeds the configured quantum, more time passes before all of its data is delivered (as packets from it, too, are mixed across the other existing queue-building flows). Thus, FQ-CoDel takes longer (as measured in time) to converge towards an ideal drop rate for a given

new flow, but does so within fewer delivered _packets_ from that flow.

Finally, FQ-CoDel drops packets from the largest flows sooner and more accurately than CoDel alone, and it is more responsive to changes in bandwidth, and in number of flows, than CoDel alone.

7. Security Considerations

There are no specific security exposures associated with FQ-CoDel. Some exposures present in current FIFO systems are in fact reduced (e.g. simple minded packet floods).

8. IANA Considerations

This document has no actions for IANA.

9. Acknowledgements

Our deepest thanks to Eric Dumazet (author of FQ-CoDel), Kathie Nichols, Van Jacobson, and all the members of the bufferbloat.net effort.

10. Conclusions

FQ-CoDel is a very general, efficient, nearly parameterless active queue management approach combining flow queueing with CoDel. It is a critical tool in solving bufferbloat.

FQ-CoDel's default settings SHOULD be modified for other special-purpose networking applications, such as for exceptionally slow links, for use in data centres, or on links with inherent delay greater than 800ms (e.g. satellite links).

On-going projects are: improving FQ-CoDel with more SFQ-like behaviour for lower bandwidth systems, improving the control law, optimising sparse packet drop behaviour, etc..

In addition to the Linux kernel sources, ns2 and ns3 models are available. Refinements (such as NFQCODEL [1]) are being tested in the CeroWrt effort.

11. References

11.1. Normative References

[RFC0896] Nagle, J., "Congestion control in IP/TCP internetworks", RFC 896, January 1984.

[RFC0970] Nagle, J., "On packet switches with infinite storage", RFC 970, December 1985.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, April 1998.

11.2. Informative References

[CODEL2012] Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <http://queue.acm.org/detail.cfm?id=2209336>.

[DRR] Shreedhar, M. and G. Varghese, "Efficient Fair Queueing Using Deficit Round Robin", June 1996, <http://users.ece.gatech.edu/~siva/ECE4607/presentations/DRR.pdf>.

[SFQ] McKenney, P., "Stochastic Fairness Queuing", September 1990, <http://www2.rdrop.com/~paulmck/scalability/paper/sfq.2002.06.04.pdf>.

[SQF2012] Bonald, T., Muscariello, L., and N. Ostallo, "On the impact of TCP and per-flow scheduling on Internet Performance - IEEE/ACM transactions on Networking", April 2012, <http://perso.telecom-paristech.fr/~bonald/Publications_files/BMO2011.pdf>.

11.3. URIs

[1] http://www.bufferbloat.net/projects/cerowrt/wiki/nfq_codel

Authors' Addresses

Toke Hoeiland-Joergensen
Karlstad University
Dept. of Computer Science
Karlstad 65188

Sweden

Email: toke.hoiland-jorgensen@kau.se

Paul McKenney
IBM Linux Technology Center
1385 NW Amberglen Parkway
Hillsboro, OR 97006
USA

Email: paulmck@linux.vnet.ibm.com
URI: http://www2.rdrop.com/~paulmck/

Dave Taht
Teklibre
2104 W First street
Apt 2002
FT Myers, FL 33901
USA

Email: d@taht.net
URI: http://www.teklibre.com/

Jim Gettys
Bell Labs
21 Oak Knoll Road
Carlisle, MA 01741
USA

Email: jg@freedesktop.org
URI: https://en.wikipedia.org/wiki/Jim_Gettys

Eric Dumazet
Google, Inc.
1600 Amphitheater Pkwy
Mountain View, Ca 94043
USA

Email: edumazet@gmail.com