Controlled Delay Active Queue Management

Abstract

   This document describes CoDel (Controlled Delay) -- a general
   framework that controls bufferbloat-generated excess delay in modern
   networking environments. CoDel consists of an estimator, a setpoint,
   and a control loop. It requires no configuration in normal Internet
   deployments.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

   The "persistently full buffer" problem has been discussed in the IETF
   community since the early 80s [RFC896]. The IRTF's End-to-End
   Research Group called for the deployment of Active Queue Management
   (AQM) to solve the problem in 1998 [RFC2309]. Despite this
   awareness, the problem has only gotten worse as growth in memory
   density per Moore's Law fueled an exponential increase in buffer pool
   size. Efforts to deploy AQM have been frustrated by difficult
   configuration and negative impact on network utilization. This
   "bufferbloat" problem [BLOAT] has become increasingly important
   throughout the Internet but particularly at the consumer edge. Queue
   management has become more critical due to increased consumer use of
   the Internet, mixing large video transactions with time-critical VoIP
   and gaming.

   An effective AQM remediates bufferbloat at a bottleneck while "doing
   no harm" at hops where buffers are not bloated. However, the
   development and deployment of AQM are frequently subject to
   misconceptions about the cause of packet queues in network buffers.
   Network buffers exist to absorb the packet bursts that occur
   naturally in statistically multiplexed networks. Buffers helpfully
   absorb the queues created by reasonable packet network behavior such
   as short-term mismatches in traffic arrival and departure rates that
   arise from upstream resource contention, transport conversation
   startup transients, and/or changes in the number of conversations
   sharing a link. Unfortunately, other less useful network behaviors
   can cause queues to fill, and their effects are not nearly as benign.
   Discussion of these issues and the reason why the solution is not
   simply "smaller buffers" can be found in [RFC2309], [VANQ2006],
   [REDL1998], and [CODEL2012]. To understand queue management, it is
   critical to understand the difference between the necessary, useful
   "good" queue and the counterproductive "bad" queue.

   Several approaches to AQM have been developed over the past two
   decades, but none have been widely deployed due to performance
   problems. When designed with the wrong conceptual model for queues,
   AQMs have limited operational range, require a lot of configuration
   tweaking, and frequently impair rather than improve performance.
   Learning from this past history, the CoDel approach is designed to
   meet the following goals:

   o Make AQM parameterless for normal operation, with no knobs for
      operators, users, or implementers to adjust.

   o Be able to distinguish "good" queue from "bad" queue and treat
      them differently, that is, keep delay low while permitting
      necessary bursts of traffic.

   o Control delay while insensitive (or nearly so) to round-trip
      delays, link rates, and traffic loads; this goal is to "do no
      harm" to network traffic while controlling delay.

   o Adapt to dynamically changing link rates with no negative impact
      on utilization.

   o Allow simple and efficient implementation (can easily span the
      spectrum from low-end access points and home routers up to high-
      end router hardware).

   CoDel has five major differences from prior AQMs: use of the local
   queue minimum to track congestion ("bad" queue), use of an efficient
   single state variable representation of that tracked statistic, use
   of packet sojourn time as the observed datum (rather than packets,
   bytes, or rates), use of a mathematically determined setpoint derived
   from maximizing network power [KLEIN81], and a modern state-space
   controller.

   CoDel configures itself based on a round-trip time metric that can be
   set to 100 ms for the normal, terrestrial Internet. With no changes
   to parameters, CoDel is expected to work across a wide range of
   conditions, with varying links and the full range of terrestrial
   round-trip times.

   CoDel is easily adapted to multiple queue systems as shown by
   [RFC8290]. Implementers and users SHOULD use the fq_codel multiple-
   queue approach as it deals with many problems beyond the reach of an
   AQM on a single queue.

   CoDel was first published in [CODEL2012] and has been implemented in
   the Linux kernel.

   Note that while this document refers to dropping packets when
   indicated by CoDel, it may be reasonable to ECN-mark packets instead.

2. Conventions and Terms Used in This Document

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and
   "OPTIONAL" in this document are to be interpreted as described in
   BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all
   capitals, as shown here.

The following terms are used in this document and are defined as
follows:

sojourn time: the amount of time a packet has spent in a particular
      buffer, i.e., the time a packet departs the buffer minus the
      time the packet arrived at the buffer. A packet can depart a
      buffer via transmission or drop.

standing queue: a queue (in packets, bytes, or time delay) in a
      buffer that persists for a "long" time, where "long" is on the
      order of the longer round-trip times through the buffer, as
      discussed in Section 4.2. A standing queue occurs when the
      minimum queue over the "long" time is non-zero and is usually
      tolerable and even desirable as long as it does not exceed some
      target delay.

bottleneck bandwidth: the limiting bandwidth along a network path.

3. Understanding the Building Blocks of Queue Management

At the heart of queue management is the notion of "good" queue and
"bad" queue and the search for ways to get rid of the "bad" queue
(which only adds delay) while preserving the "good" queue (which
provides for good utilization). This section explains queueing, both
good and bad, and covers the CoDel building blocks that can be used
to manage packet buffers to keep their queues in the "good" range.

Packet queues form in buffers facing bottleneck links, i.e., where
the line rate goes from high to low or where many links converge.
The well-known bandwidth-delay product (sometimes called "pipe size")
is the bottleneck's bandwidth multiplied by the sender-receiver-
sender round-trip delay; it is the amount of data that has to be in
transit between two hosts in order to run the bottleneck link at 100%
utilization. To explore how queues can form, consider a long-lived
TCP connection with a 25-packet window sending through a connection
with a bandwidth-delay product of 20 packets. After an initial burst
of packets, the connection will settle into a 5-packet (+/-1)
standing queue; this standing queue size is determined by the
mismatch between the window size and the pipe size and is unrelated
to the connection's sending rate. The connection has 25 packets in
flight at all times, but only 20 packets arrive at the destination
over a round-trip time. If the TCP connection has a 30-packet
window, the queue will be 10 packets with no change in sending rate.
Similarly, if the window is 20 packets, there will be no queue, but
the sending rate is the same. Nothing can be inferred about the
sending rate from the queue size, and any queue other than transient
bursts only creates delays in the network. The sender needs to
reduce the number of packets in flight rather than the sending rate.

In the above example, the 5-packet standing queue can be seen to
contribute nothing but delay to the connection and thus is clearly
"bad" queue. If, in our example, there is a single bottleneck link
and it is much slower than the link that feeds it (say, a high-speed
Ethernet link into a limited DSL uplink), then a 20-packet buffer at
the bottleneck might be necessary to temporarily hold the 20 packets
in flight to keep the bottleneck link's utilization high. The burst
of packets should drain completely (to 0 or 1 packets) within a
round-trip time, and this transient queue is "good" queue because it
allows the connection to keep the 20 packets in flight and the
bottleneck link to be fully utilized. In terms of the delay
experienced, the "good" queue goes away in about a round-trip time,
while "bad" queue hangs around for longer, causing delays.

Effective queue management detects "bad" queue while ignoring "good"
queue and takes action to get rid of the "bad" queue when it is
detected. The goal is a queue controller that accomplishes this
objective. To control a queue, we need three basic components:

o Estimator - To figure out what we've got.

o Target setpoint - To know what we want.

o Control loop - If what we've got isn't what we want, we need a way
   to move it there.

3.1. Estimator

The estimator both observes the queue and detects when "good" queue
turns to "bad" queue and vice versa. CoDel has two parts to its
estimator: what is observed as an indicator of queue and how the
observations are used to detect "good"/"bad" queue.

Queue length has been widely used as an observed indicator of
congestion and is frequently conflated with sending rate. Use of
queue length as a metric is sensitive to how and when the length is
observed. A high-speed arrival link to a buffer serviced at a much
lower rate can rapidly build up a queue that might disperse
completely or down to a single packet before a round-trip time has
elapsed. If the queue length is monitored at packet arrival (as in
original Random Early Detection (RED)) or departure time, every
packet will see a queue with one possible exception. If the queue
length itself is time sampled (as recommended in [REDL1998]), a truer
picture of the queue's occupancy can be gained at the expense of
considerable implementation complexity.

The use of queue length is further complicated in networks that are
subject to both short- and long-term changes in available link rate
(as in WiFi). Link rate drops can result in a spike in queue length
that should be ignored unless it persists. It is not the queue
length that should be controlled but the amount of excess delay
packets experience due to a persistent or standing queue, which means
that the packet sojourn time in the buffer is exactly what we want to
track. Tracking the packet sojourn times in the buffer observes the
actual delay experienced by each packet. Sojourn time allows queue
management to be independent of link rate, gives superior performance
to use of buffer size, and is directly related to user-visible
performance. It works regardless of line rate changes or link
sharing by multiple queues (which the individual queues may
experience as changing rates).

Consider a link shared by two queues with different priorities.
Packets that arrive at the high-priority queue are sent as soon as
the link is available, while packets in the other queue have to wait
until the high-priority queue is empty (i.e., a strict priority
scheduler). The number of packets in the high-priority queue might
be large, but the queue is emptied quickly, and the amount of time
each packet spends enqueued (the sojourn time) is not large. The
other queue might have a smaller number of packets, but packet
sojourn times will include the waiting time for the high-priority
packets to be sent. This makes the sojourn time a good sample of the
congestion that each separate queue is experiencing. This example
also shows how the metric of sojourn time is independent of the
number of queues or the service discipline used and is instead
indicative of congestion seen by the individual queues.

How can observed sojourn time be used to separate "good" queue from
"bad" queue? Although averages, especially of queue length, have
previously been widely used as an indicator of "bad" queue, their
efficacy is questionable. Consider the burst that disperses every
round-trip time. The average queue will be one-half the burst size,
though this might vary depending on when the average is computed and
the timing of arrivals. The average queue sojourn time would be one-
half the time it takes to clear the burst. The average then would
indicate a persistent queue where there is none. Instead of
averages, we recommend tracking the minimum sojourn time; then, if
there is one packet that has a zero sojourn time, there is no
persistent queue.

A persistent queue can be detected by tracking the (local) minimum
queue delay packets experience. To ensure that this minimum value
does not become stale, it has to have been experienced recently,
i.e., during an appropriate past time interval. This interval is the
maximum amount of time a minimum value is considered to be in effect

and is related to the amount of time it takes for the largest
expected burst to drain. Conservatively, this interval SHOULD be at
least a round-trip time to avoid falsely detecting a persistent queue
and not a lot more than a round-trip time to avoid delay in detecting
the persistent queue. This suggests that the appropriate interval
value is the maximum round-trip time of all the connections sharing
the buffer.

Note that the following key insight makes computation of the local
minimum efficient: it is sufficient to keep a single state variable
that indicates how long the minimum has been above or below the
target value rather than retaining all the local values to compute
the minimum, which leads to both storage and computational savings.
We use this insight in the pseudocode for CoDel later in the
document.

These two parts, use of sojourn time as the observed value and the
local minimum as the statistic to monitor queue congestion, are key
to CoDel's estimator building block. The local minimum sojourn time
provides an accurate and robust measure of standing queue and has an
efficient implementation. In addition, use of the minimum sojourn
time has important advantages in implementation. The minimum packet
sojourn can only be decreased when a packet is dequeued, which means
that all the work of CoDel can take place when packets are dequeued
for transmission and that no locks are needed in the implementation.
The minimum is the only statistic with this property.

A more detailed explanation with many pictures can be found in
[TSV84].

3.2. Target Setpoint

Now that we have a robust way of detecting standing queue, we need a
target setpoint that tells us when to act. If the controller is set
to take action as soon as the estimator has a non-zero value, the
average drop rate will be maximized, which minimizes TCP goodput
[MACTCP1997]. Also, this policy results in no backlog over time (no
persistent queue), which negates much of the value of having a
buffer, since it maximizes the bottleneck link bandwidth lost due to
normal stochastic variation in packet interarrival time. We want a
target that maximizes utilization while minimizing delay. Early in
the history of packet networking, Kleinrock developed the analytic
machinery to do this using a quantity he called "power", which is the
ratio of a normalized throughput to a normalized delay [KLEIN81].

It is straightforward to derive an analytic expression for the
average goodput of a TCP conversation at a given round-trip time r
and target f (where f is expressed as a fraction of r). Reno TCP,
for example, yields:

goodput = r (3 + 6f - f^2) / (4 (1+f))

Since the peak queue delay is simply the product of f and r, power is
solely a function of f since the r's in the numerator and denominator
cancel:

power is proportional to (1 + 2f - 1/3 f^2) / (1 + f)^2

As Kleinrock observed, the best operating point (in terms of
bandwidth/delay trade-off) is the peak power point, since points off
the peak represent a higher cost (in delay) per unit of bandwidth.
The power vs. f curve for any Additive Increase Multiplicative
Decrease (AIMD) TCP is monotone decreasing. But the curve is very
flat for f < 0.1, followed by an increasing curvature with a knee
around f = 0.2, then a steep, almost linear fall off [TSV84]. Since
the previous equation showed that goodput is monotone increasing with
f, the best operating point is near the right edge of the flat top,
since that represents the highest goodput achievable for a negligible
increase in delay. However, since the r in the model is a
conservative upper bound, a target of 0.1r runs the risk of pushing
shorter RTT connections over the knee and giving them higher delay
for no significant goodput increase. Generally, a more conservative
target of 0.05r offers a good utilization vs. delay trade-off while
giving enough headroom to work well with a large variation in real
RTT.

As the above analysis shows, a very small standing queue gives close
to 100% utilization of the bottleneck link. While this result was
for Reno TCP, the derivation uses only properties that must hold for
any "TCP friendly" transport. We have verified by both analysis and
simulation that this result holds for Reno, Cubic, and Westwood
[TSV84]. This results in a particularly simple form for the target:
the ideal range for the permitted standing queue, or the target
setpoint, is between 5% and 10% of the TCP connection's RTT.

We used simulation to explore the impact when TCPs are mixed with
other traffic and with connections of different RTTs. Accordingly,
we experimented extensively with values in the 5-10% of RTT range
and, overall, used target values between 1 and 20 milliseconds for
RTTs from 30 to 500 ms and link bandwidths of 64 Kbps to 100 Mbps to
experimentally explore a value for the target that gives consistently

high utilization while controlling delay across a range of
bandwidths, RTTs, and traffic loads. Our results were notably
consistent with the mathematics above.

A congested (but not overloaded) CoDel link with traffic composed
solely or primarily of long-lived TCP flows will have a median delay
through the link that will tend to the target. For bursty traffic
loads and for overloaded conditions (where it is difficult or
impossible for all the arriving flows to be accommodated), the median
queues will be longer than the target.

The non-starvation drop inhibit feature dominates where the link rate
becomes very small. By inhibiting drops when there is less than an
(outbound link) MTU worth of bytes in the buffer, CoDel adapts to
very low bandwidth links, as shown in [CODEL2012].

3.3. Control Loop

Section 3.1 describes a simple, reliable way to measure "bad"
(persistent) queue. Section 3.2 shows that TCP congestion control
dynamics gives rise to a target setpoint for this measure that's a
provably good balance between enhancing throughput and minimizing
delay. Section 3.2 also shows that this target is a constant
fraction of the same "largest average RTT" interval used to
distinguish persistent from transient queue. The only remaining
building block needed for a basic AQM is a "control loop" algorithm
to effectively drive the queueing system from any "persistent queue
above the target" state to a state where the persistent queue is
below the target.

Control theory provides a wealth of approaches to the design of
control loops. Most of classical control theory deals with the
control of linear, time-invariant, Single-Input-Single-Output (SISO)
systems. Control loops for these systems generally come from a well-
understood class known as Proportional-Integral-Derivative (PID)
controllers. Unfortunately, a queue is not a linear system, and an
AQM operates at the point of maximum non-linearity (where the output
link bandwidth saturates, so increased demand creates delay rather
than higher utilization). Output queues are also not time invariant
since traffic is generally a mix of connections that start and stop
at arbitrary times and that can have radically different behaviors
ranging from "open-loop" UDP audio/video to "closed-loop" congestion-
avoiding TCP. Finally, the constantly changing mix of connections
(which can't be converted to a single "lumped parameter" model
because of their transfer function differences) makes the system
Multi-Input-Multi-Output (MIMO), not SISO.

Since queueing systems match none of the prerequisites for a
classical controller, a better approach is a modern state-space
controller with "no persistent queue" and "has persistent queue"
states. Since Internet traffic mixtures change rapidly and
unpredictably, a noise- and error-tolerant adaptation algorithm like
stochastic gradient is a good choice. Since there's essentially no
information in the amount of persistent queue [TSV84], the adaptation
should be driven by how long it has persisted.

Consider the two extremes of traffic behavior: a single, open-loop
UDP video stream and a single, long-lived TCP bulk data transfer. If
the average bandwidth of the UDP video stream is greater than the
bottleneck link rate, the link's queue will grow, and the controller
will eventually enter "has persistent queue" state and start dropping
packets. Since the video stream is open loop, its arrival rate is
unaffected by drops, so the queue will persist until the average drop
rate is greater than the output bandwidth deficit (= average arrival
rate - average departure rate); the job of the adaptation algorithm
is to discover this rate. For this example, the adaptation could
consist of simply estimating the arrival and departure rates and then
dropping at a rate slightly greater than their difference, but this
class of algorithm won't work at all for the bulk data TCP stream.
TCP runs in closed-loop flow balance [TSV84], so its arrival rate is
almost always exactly equal to the departure rate -- the queue isn't
the result of a rate imbalance but rather a mismatch between the TCP
sender's window and the source-destination-source round-trip path
capacity (i.e., the connection's bandwidth-delay product). The
sender's TCP congestion avoidance algorithm will slowly increase the
send window (one packet per round-trip time) [RFC5681], which will
eventually cause the bottleneck to enter "has persistent queue"
state. But, since the average input rate is the same as the average
output rate, the rate deficit estimation that gave the correct drop
rate for the video stream would compute a drop rate of zero for the
TCP stream. However, if the output link drops one packet as it
enters "has persistent queue" state, when the sender discovers this
(via TCP's normal packet loss repair mechanisms), it will reduce its
window by a factor of two [RFC5681]; so, one round-trip time after
the drop, the persistent queue will go away.

If there were N TCP conversations sharing the bottleneck, the
controller would have to drop O(N) packets (one from each
conversation) to make all the conversations reduce their window to
get rid of the persistent queue. If the traffic mix consists of
short (<= bandwidth-delay product) conversations, the aggregate
behavior becomes more like the open-loop video example since each
conversation is likely to have already sent all its packets by the
time it learns about a drop so each drop has negligible effect on
subsequent traffic.

The controller does not know the number, duration, or kind of
conversations creating its queue, so it has to learn the appropriate
response. Since single drops can have a large effect if the degree
of multiplexing (the number of active conversations) is small,
dropping at too high a rate is likely to have a catastrophic effect
on throughput. Dropping at a low rate (< 1 packet per round-trip
time) and then increasing the drop rate slowly until the persistent
queue goes below the target is unlikely to overdrop and is guaranteed
to eventually dissipate the persistent queue. This stochastic
gradient learning procedure is the core of CoDel's control loop (the
gradient exists because a drop always reduces the (instantaneous)
queue, so an increasing drop rate always moves the system "down"
toward no persistent queue, regardless of traffic mix).

The "next drop time" is decreased in inverse proportion to the square
root of the number of drops since the drop state was entered, using
the well-known non-linear relationship of drop rate to throughput to
get a linear change in throughput [REDL1998][MACTCP1997].

Since the best rate to start dropping is at slightly more than one
packet per RTT, the controller's initial drop rate can be directly
derived from the estimator's interval. When the minimum sojourn time
first crosses the target and CoDel drops a packet, the earliest the
controller could see the effect of the drop is the round-trip time
(interval) + the local queue wait time (the target). If the next
drop happens any earlier than this time (interval + target), CoDel
will overdrop. In practice, the local queue waiting time tends to
vary, so making the initial drop spacing (i.e., the time to the
second drop) be exactly the minimum possible also leads to
overdropping. Analysis of simulation and real-world measured data
shows that the 75th percentile magnitude of this variation is less
than the target, so the initial drop spacing SHOULD be set to the
estimator's interval (i.e., initial drop spacing = interval) to
ensure that the controller has accounted for acceptable congestion
delays.

Use of the minimum statistic lets the controller be placed in the
dequeue routine with the estimator. This means that the control
signal (the drop) can be sent at the first sign of "bad" queue (as
indicated by the sojourn time) and that the controller can stop
acting as soon as the sojourn time falls below the target. Dropping
at dequeue has both implementation and control advantages.

4. Overview of the CoDel AQM

   CoDel was initially designed as a bufferbloat solution for the
   consumer network edge. The CoDel building blocks are able to adapt
   to different or time-varying link rates, be easily used with multiple
   queues, have excellent utilization with low delay, and have a simple
   and efficient implementation.

   The CoDel algorithm described in the rest of this document uses two
   key variables: TARGET, which is the controller's target setpoint
   described in Section 3.2, and INTERVAL, which is the estimator's
   interval described in Section 3.3.

   The only setting CoDel requires is the INTERVAL value, and as 100 ms
   satisfies that definition for normal Internet usage, CoDel can be
   parameter-free for consumer use. To ensure that link utilization is
   not adversely affected, CoDel's estimator sets TARGET to one that
   optimizes power. CoDel's controller does not drop packets when the
   drop would leave the queue empty or with fewer than a Maximum
   Transmission Unit (MTU) worth of bytes in the buffer. Section 3.2
   shows that an ideal TARGET is 5-10% of the connection round-trip time
   (RTT). In the open terrestrial-based Internet, especially at the
   consumer edge, we expect most unbloated RTTs to have a ceiling of 100
   ms [CHARB2007]. Using this RTT gives a minimum TARGET of 5 ms and
   INTERVAL of 100 ms. In practice, uncongested links will see sojourn
   times below TARGET more often than once per RTT, so the estimator is
   not overly sensitive to the value of INTERVAL.

   When the estimator finds a persistent delay above TARGET, the
   controller enters the drop state where a packet is dropped, and the
   next drop time is set. As discussed in Section 3.3, the initial next
   drop spacing is intended to be long enough to give the endpoints time
   to react to the single drop and therefore SHOULD be set to a value
   equal to INTERVAL. If the estimator's output falls below TARGET, the
   controller cancels the next drop and exits the drop state. (The
   controller is more sensitive than the estimator to an overly short
   INTERVAL value, since an unnecessary drop would occur and lower link
   utilization). If the next drop time is reached while the controller
   is still in drop state, the packet being dequeued is dropped, and the
   next drop time is recalculated.

   Additional logic prevents re-entering the drop state too soon after
   exiting it and resumes the drop state at a recent control level, if
   one exists. This logic is described more precisely in the pseudocode
   below. Additional work is required to determine the frequency and
   importance of re-entering the drop state.

Note that CoDel AQM only enters its drop state when the local minimum
sojourn delay has exceeded TARGET for a time period long enough for
normal bursts to dissipate, ensuring that a burst of packets that
fits in the pipe will not be dropped.

4.1. Non-starvation

CoDel's goals are to control delay with little or no impact on link
utilization and to be deployed on a wide range of link bandwidths,
including variable-rate links, without reconfiguration. To keep from
making drops when it would starve the output link, CoDel makes
another check before dropping to see if at least an MTU worth of
bytes remains in the buffer. If not, the packet SHOULD NOT be
dropped; therefore, CoDel exits the drop state. The MTU size can be
set adaptively to the largest packet seen so far or can be read from
the interface driver.

4.2. Setting INTERVAL

The INTERVAL value is chosen to give endpoints time to react to a
drop without being so long that response times suffer. CoDel's
estimator, TARGET, and control loop all use INTERVAL. Understanding
their derivation shows that CoDel is the most sensitive to the value
of INTERVAL for single long-lived TCPs with a decreased sensitivity
for traffic mixes. This is fortunate, as RTTs vary across
connections and are not known a priori. The best policy seems to be
to use an INTERVAL value slightly larger than the RTT seen by most of
the connections using a link, a value that can be determined as the
largest RTT seen if the value is not an outlier (use of a 95-99th
percentile value should work). In practice, this value is not known
or measured (however, see Appendix A for an application where
INTERVAL is measured). An INTERVAL setting of 100 ms works well
across a range of RTTs from 10 ms to 1 second (excellent performance
is achieved in the range from 10 ms to 300 ms). For devices intended
for the normal terrestrial Internet, INTERVAL SHOULD have a value of
100 ms. This will only cause overdropping where a long-lived TCP has
an RTT longer than 100 ms and there is little or no mixing with other
connections through the link.

4.3. Setting TARGET

TARGET is the maximum acceptable persistent queue delay above which
CoDel is dropping or preparing to drop and below which CoDel will not
drop. TARGET SHOULD be set to 5 ms for normal Internet traffic.

The calculations of Section 3.2 show that the best TARGET value is
5-10% of the RTT, with the low end of 5% preferred. Extensive
simulations exploring the impact of different TARGET values when used

with mixed traffic flows with different RTTs and different bandwidths
show that below a TARGET of 5 ms, utilization suffers for some
conditions and traffic loads; above 5 ms showed very little or no
improvement in utilization.

Sojourn times must remain above the TARGET for INTERVAL amount of
time in order to enter the drop state. Any packet with a sojourn
time less than TARGET will reset the time that the queue was last
below TARGET. Since Internet traffic has very dynamic
characteristics, the actual sojourn delay experienced by packets
varies greatly and is often less than TARGET unless the overload is
excessive. When a link is not overloaded, it is not a bottleneck,
and packet sojourn times will be small or nonexistent. In the usual
case, there are only one or two places along a path where packets
will encounter a bottleneck (usually at the edge), so the total
amount of queueing delay experienced by a packet should be less than
10 ms even under extremely congested conditions. This net delay is
substantially lower than common current queueing delays on the
Internet that grow to orders of seconds [NETAL2010] [CHARB2007].

Regarding the roles of TARGET and the minimum-tracking INTERVAL, note
that TARGET SHOULD NOT be increased in response to lower layers that
have a bursty nature, where packets are transmitted for short periods
interspersed with idle periods where the link is waiting for
permission to send. CoDel's estimator will "see" the effective
transmission rate over an INTERVAL amount of time, and increasing
TARGET only leads to longer queue delays. On the other hand, where a
significant additional delay is added to the intrinsic RTT of most or
all packets due to the waiting time for a transmission, it is
necessary to increase INTERVAL by that extra delay. TARGET SHOULD
NOT be adjusted for such short-term bursts, but INTERVAL MAY need to
be adjusted if the path's intrinsic RTT changes.

4.4. Use with Multiple Queues

CoDel is easily adapted to multiple queue systems. With other
approaches, there is always a question of how to account for the fact
that each queue receives less than the full link rate over time and
usually sees a varying rate over time. This is what CoDel excels at:
using a packet's sojourn time in the buffer completely circumvents
this problem. In a multiple-queue setting, a separate CoDel
algorithm runs on each queue, but each CoDel instance uses the packet
sojourn time the same way a single-queue CoDel does. Just as a
single-queue CoDel adapts to changing link bandwidths [CODEL2012], so
does a multiple-queue CoDel system. As an optimization to avoid
queueing more than necessary, when testing for queue occupancy before
dropping, the total occupancy of all queues sharing the same output
link SHOULD be used. This property of CoDel has been exploited in

fq_codel [RFC8290], which hashes on the packet header fields to
determine a specific bin, or sub-queue, for the packet and runs CoDel
on each bin or sub-queue, thus creating a well-mixed output flow and
obviating issues of reverse path flows (including "ack compression").

4.5. Setting Up CoDel

   CoDel is set for use in devices in the open Internet. An INTERVAL
   setting of 100 ms is used, TARGET is set to 5% of INTERVAL, and the
   initial drop spacing is also set to the INTERVAL. These settings
   have been chosen so that a device, such as a small WiFi router, can
   be sold without the need for any values to be made adjustable,
   yielding a parameterless implementation. In addition, CoDel is
   useful in environments with significantly different characteristics
   from the normal Internet, for example, in switches used as a cluster
   interconnect within a data center. Since cluster traffic is entirely
   internal to the data center, round-trip latencies are low (typically
   <100 us) but bandwidths are high (1-40 Gbps), so it's relatively easy
   for the aggregation phase of a distributed computation (e.g., the
   Reduce part of a Map/Reduce) to persistently fill and then overflow
   the modest per-port buffering available in most high-speed switches.
   A CoDel configured for this environment (TARGET and INTERVAL in the
   microsecond rather than millisecond range) can minimize drops or
   Explicit Congestion Notification (ECN) marks while keeping throughput
   high and latency low.

   Devices destined for these environments MAY use a different value for
   INTERVAL, where suitable. If appropriate analysis indicates, the
   TARGET MAY be set to some other value in the 5-10% of INTERVAL, and
   the initial drop spacing MAY be set to a value of 1.0 to 1.2 times
   INTERVAL. But these settings will cause problems, such as
   overdropping and low throughput, if used on the open Internet, so
   devices that allow CoDel to be configured SHOULD default to the
   Internet-appropriate values given in this document.

5. Annotated Pseudocode for CoDel AQM

   What follows is the CoDel algorithm in C++-like pseudocode. Since
   CoDel adds relatively little new code to a basic tail-drop FIFO
   queue, we have attempted to highlight just these additions by
   presenting CoDel as a sub-class of a basic FIFO queue base class.
   The reference code is included to aid implementers who wish to apply
   CoDel to queue management as described here or to adapt its
   principles to other applications.

Implementors are strongly encouraged to also look at the Linux kernel
version of CoDel -- a well-written, well-tested, real-world, C-based
implementation. As of this writing, it is available at
https://github.com/torvalds/linux/blob/master/net/sched/sch_codel.c.

5.1. Data Types

time_t is an integer time value in units convenient for the system.
The code presented here uses 0 as a flag value to indicate "no time
set."

packet_t* is a pointer to a packet descriptor. We assume it has a
tstamp field capable of holding a time_t and that the field is
available for use by CoDel (it will be set by the enqueue routine and
used by the dequeue routine).

queue_t is a base class for queue objects (the parent class for
codel_queue_t objects). We assume it has enqueue() and dequeue()
methods that can be implemented in child classes. We assume it has a
bytes() method that returns the current queue size in bytes. This
can be an approximate value. The method is invoked in the dequeue()
method but shouldn't require a lock with the enqueue() method.

flag_t is a Boolean.

5.2. Per-Queue State (codel_queue_t Instance Variables)

```
time_t first_above_time_ = 0; // Time to declare sojourn time above
                              // TARGET
time_t drop_next_ = 0;        // Time to drop next packet
uint32_t count_ = 0;          // Packets dropped in drop state
uint32_t lastcount_ = 0;      // Count from previous iteration
flag_t dropping_ = false;     // Set to true if in drop state
```

5.3. Constants

```
time_t TARGET = MS2TIME(5);     // 5 ms TARGET queue delay
time_t INTERVAL = MS2TIME(100); // 100 ms sliding-minimum window
u_int maxpacket = 512;          // Maximum packet size in bytes
                                // (SHOULD use interface MTU)
```

5.4. Enqueue Routine

   All the work of CoDel is done in the dequeue routine. The only CoDel
   addition to enqueue is putting the current time in the packet's
   tstamp field so that the dequeue routine can compute the packet's
   sojourn time. Note that packets arriving at a full buffer will be
   dropped, but these drops are not counted towards CoDel's
   computations.

```
   void codel_queue_t::enqueue(packet_t* pkt)
   {
       pkt->tstamp = clock();
       queue_t::enqueue(pkt);
   }
```

5.5. Dequeue Routine

   This is the heart of CoDel. There are two branches based on whether
   the controller is in drop state: (i) if the controller is in drop
   state (that is, the minimum packet sojourn time is greater than
   TARGET), then the controller checks if it is time to leave drop state
   or schedules the next drop(s); or (ii) if the controller is not in
   drop state, it determines if it should enter drop state and do the
   initial drop.

```
   packet_t* CoDelQueue::dequeue()
   {
       time_t now = clock();
       dodequeue_result r = dodequeue(now);
       uint32_t delta;

       if (dropping_) {
           if (! r.ok_to_drop) {
               // sojourn time below TARGET - leave drop state
               dropping_ = false;
           }
           // Time for the next drop.  Drop current packet and dequeue
           // next.  If the dequeue doesn't take us out of dropping
           // state, schedule the next drop.  A large backlog might
           // result in drop rates so high that the next drop should
           // happen now, hence the 'while' loop.
           while (now >= drop_next_ && dropping_) {
               drop(r.p);
               ++count_;
               r = dodequeue(now);
               if (! r.ok_to_drop) {
                   // leave drop state
                   dropping_ = false;
```

```
                } else {
                    // schedule the next drop.
                    drop_next_ = control_law(drop_next_, count_);
                }
            }
        // If we get here, we're not in drop state.  The 'ok_to_drop'
        // return from dodequeue means that the sojourn time has been
        // above 'TARGET' for 'INTERVAL', so enter drop state.
        } else if (r.ok_to_drop) {
            drop(r.p);
            r = dodequeue(now);
            dropping_ = true;

            // If min went above TARGET close to when it last went
            // below, assume that the drop rate that controlled the
            // queue on the last cycle is a good starting point to
            // control it now.  ('drop_next' will be at most 'INTERVAL'
            // later than the time of the last drop, so 'now - drop_next'
            // is a good approximation of the time from the last drop
            // until now.) Implementations vary slightly here; this is
            // the Linux version, which is more widely deployed and
            // tested.
            delta = count_ - lastcount_;
            count_ = 1;
            if ((delta > 1) && (now - drop_next_ < 16*INTERVAL))
                count_ = delta;

            drop_next_ = control_law(now, count_);
            lastcount_ = count_;
        }
        return (r.p);
    }
```

5.6. Helper Routines

   Since the degree of multiplexing and nature of the traffic sources is
   unknown, CoDel acts as a closed-loop servo system that gradually
   increases the frequency of dropping until the queue is controlled
   (sojourn time goes below TARGET). This is the control law that
   governs the servo. It has this form because of the sqrt(p)
   dependence of TCP throughput on drop probability. Note that for
   embedded systems or kernel implementation, the inverse sqrt can be
   computed efficiently using only integer multiplication.

```
   time_t codel_queue_t::control_law(time_t t, uint32_t count)
   {
       return t + INTERVAL / sqrt(count);
   }
```

Next is a helper routine that does the actual packet dequeue and
tracks whether the sojourn time is above or below TARGET and, if
above, if it has remained above continuously for at least INTERVAL
amount of time. It returns two values: a Boolean indicating if it is
OK to drop (sojourn time above TARGET for at least INTERVAL) and the
packet dequeued.

```
typedef struct {
    packet_t* p;
    flag_t ok_to_drop;
} dodequeue_result;

dodequeue_result codel_queue_t::dodequeue(time_t now)
{
    dodequeue_result r = { queue_t::dequeue(), false };
    if (r.p == NULL) {
        // queue is empty - we can't be above TARGET
        first_above_time_ = 0;
        return r;
    }

    // To span a large range of bandwidths, CoDel runs two
    // different AQMs in parallel.  One is based on sojourn time
    // and takes effect when the time to send an MTU-sized
    // packet is less than TARGET.  The 1st term of the "if"
    // below does this.  The other is based on backlog and takes
    // effect when the time to send an MTU-sized packet is >=
    // TARGET.  The goal here is to keep the output link
    // utilization high by never allowing the queue to get
    // smaller than the amount that arrives in a typical
    // interarrival time (MTU-sized packets arriving spaced
    // by the amount of time it takes to send such a packet on
    // the bottleneck).  The 2nd term of the "if" does this.
    time_t sojourn_time = now - r.p->tstamp;
    if (sojourn_time_ < TARGET || bytes() <= maxpacket_) {
        // went below - stay below for at least INTERVAL
        first_above_time_ = 0;
    } else {
        if (first_above_time_ == 0) {
            // just went above from below. if still above at
            // first_above_time, will say it's ok to drop.
            first_above_time_ = now + INTERVAL;
        } else if (now >= first_above_time_) {
            r.ok_to_drop = true;
        }
    }
    return r;
}
```

5.7. Implementation Considerations

   time_t is an integer time value in units convenient for the system.
   Resolution to at least a millisecond is required, and better
   resolution is useful up to the minimum possible packet time on the
   output link; 64- or 32-bit widths are acceptable but with 32 bits the
   resolution should be no finer than 2^{-16} to leave enough dynamic
   range to represent a wide range of queue waiting times. Narrower
   widths also have implementation issues due to overflow (wrapping) and
   underflow (limit cycles because of truncation to zero) that are not
   addressed in this pseudocode.

   Since CoDel requires relatively little per-queue state and no direct
   communication or state sharing between the enqueue and dequeue
   routines, it is relatively simple to add CoDel to almost any packet
   processing pipeline, including forwarding engines based on
   Application-Specific Integrated Circuits (ASICs) or Network
   Processors (NPUs). One issue to consider is dodequeue()'s use of a
   'bytes()' function to determine the current queue size in bytes.
   This value does not need to be exact. If the enqueue part of the
   pipeline keeps a running count of the total number of bytes it has
   put into the queue, and the dequeue routine keeps a running count of
   the total bytes it has removed from the queue, 'bytes()' is simply
   the difference between these two counters (32-bit counters should be
   adequate). Enqueue has to update its counter once per packet queued,
   but it does not matter when (before, during, or after the packet has
   been added to the queue). The worst that can happen is a slight,
   transient underestimate of the queue size, which might cause a drop
   to be briefly deferred.

6. Further Experimentation

   We encourage experimentation with the recommended values of TARGET
   and INTERVAL for Internet settings. CoDel provides general,
   efficient, parameterless building blocks for queue management that
   can be applied to single or multiple queues in a variety of data
   networking scenarios. CoDel's settings may be modified for other
   special-purpose networking applications.

7. Security Considerations

   This document describes an active queue management algorithm for
   implementation in networked devices. There are no known security
   exposures associated with CoDel at this time.

8. IANA Considerations

   This document does not require actions by IANA.

9. References

9.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

   [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
              2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,
              May 2017, <https://www.rfc-editor.org/info/rfc8174>.

9.2. Informative References

   [BLOAT]   Gettys, J. and K. Nichols, "Bufferbloat: Dark Buffers in
              the Internet", Communications of the ACM, Volume 55, Issue
              1, DOI 10.1145/2063176.2063196, January 2012.

   [CHARB2007]
              Dischinger, M., Haeberlen, A., Gummadi, K., and S. Saroiu,
              "Characterizing Residential Broadband Networks",
              Proceedings of the 7th ACM SIGCOMM Conference on Internet
              Measurement, DOI 10.1145/1298306.1298313, October 2007.

   [CODEL2012]
              Nichols, K. and V. Jacobson, "Controlling Queue Delay",
              ACM Queue, Volume 10, Issue 5,
              DOI 10.1145/2208917.2209336, May 2012.

   [KLEIN81] Kleinrock, L. and R. Gail, "An Invariant Property of
              Computer Network Power", Proceedings of the International
              Conference on Communications, June 1981,
              <http://www.lk.cs.ucla.edu/data/files/Gail/power.pdf>.

   [MACTCP1997]
              Mathis, M., Semke, J., Mahdavi, J., and T. Ott, "The
              Macroscopic Behavior of the TCP Congestion Avoidance
              Algorithm", ACM SIGCOMM Computer Communications
              Review, Volume 27, Issue 3, pp. 67-82,
              DOI 10.1145/263932.264023, July 1997.

   [NETAL2010]
              Kreibich, C., Weaver, N., Paxson, V., and B. Nechaev,
              "Netalyzr: Illuminating the Edge Network", Proceedings of
              the 10th ACM SIGCOMM Conference on Internet Measurement,
              DOI 10.1145/1879141.1879173, November 2010.

   [REDL1998] Nichols, K., Jacobson, V., and K. Poduri, "RED in a
             Different Light", Technical Report, Cisco Systems,
             September 1999, <http://citeseerx.ist.psu.edu/viewdoc/
             summary?doi=10.1.1.22.9406>.

   [RFC896]   Nagle, J., "Congestion Control in IP/TCP Internetworks",
             RFC 896, DOI 10.17487/RFC0896, January 1984,
             <https://www.rfc-editor.org/info/rfc896>.

   [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, DOI 10.17487/RFC2309, April 1998,
             <https://www.rfc-editor.org/info/rfc2309>.

   [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion
             Control", RFC 5681, DOI 10.17487/RFC5681, September 2009,
             <https://www.rfc-editor.org/info/rfc5681>.

   [RFC8290] Hoeiland-Joergensen, T., McKenney, P., Taht, D.,
             Gettys, J., and E. Dumazet, "The Flow Queue CoDel Packet
             Scheduler and Active Queue Management Algorithm",
             RFC 8290, DOI 10.17487/RFC8290, January 2018,
             <https://www.rfc-editor.org/info/rfc8290>.

   [TSV84]   Jacobson, V., "CoDel", IETF 84, Transport Area Open
             Meeting, July 2012,
             <http://www.ietf.org/proceedings/84/slides/
             slides-84-tsvarea-4.pdf>.

   [VANQ2006] Jacobson, V., "A Rant on Queues", Talk at MIT Lincoln
             Labs, Lexington, MA, July 2006,
             <http://www.pollere.net/Pdfdocs/QrantJul06.pdf>.

Appendix A. Applying CoDel in the Data Center

   Nandita Dukkipati and her group at Google realized that the CoDel
   building blocks could be applied to bufferbloat problems in data-
   center servers, not just to Internet routers. The Linux CoDel
   queueing discipline (qdisc) was adapted in three ways to tackle this
   bufferbloat problem.

   1. The default CoDel action was modified to be a direct feedback
       from qdisc to the TCP layer at dequeue. The direct feedback
       simply reduces TCP's congestion window just as congestion control
       would do in the event of drop. The scheme falls back to ECN
       marking or packet drop if the TCP socket lock could not be
       acquired at dequeue.

   2. Being located in the server makes it possible to monitor the
       actual RTT to use as CoDel's interval rather than making a "best
       guess" of RTT. The CoDel interval is dynamically adjusted by
       using the maximum TCP round-trip time (RTT) of those connections
       sharing the same qdisc/bucket. In particular, there is a history
       entry of the maximum RTT experienced over the last second. As a
       packet is dequeued, the RTT estimate is accessed from its TCP
       socket. If the estimate is larger than the current CoDel
       interval, the CoDel interval is immediately refreshed to the new
       value. If the CoDel interval is not refreshed for over a second,
       it is decreased to the history entry, and the process is
       repeated. The use of the dynamic TCP RTT estimate allows the
       interval to adapt to the actual maximum value currently seen and
       thus lets the controller space its drop intervals appropriately.

   3. Since the mathematics of computing the setpoint are invariant, a
       TARGET of 5% of the RTT or CoDel interval was used here.

   Non-data packets were not dropped, as these are typically small and
   sometimes critical control packets. Being located on the server,
   there is no concern with misbehaving users as there would be on the
   public Internet.

   In several data-center workload benchmarks, which are typically
   bursty, CoDel reduced the queueing latencies at the qdisc and thereby
   improved the mean and 99th-percentile latencies from several tens of
   milliseconds to less than one millisecond. The minimum tracking part
   of the CoDel framework proved useful in disambiguating "good" queue
   versus "bad" queue, which is particularly helpful in controlling
   qdisc buffers that are inherently bursty because of TCP Segmentation
   Offload (TSO).

Acknowledgments

Authors' Addresses

   Kathleen Nichols
   Pollere, Inc.
   PO Box 370201
   Montara, CA 94037
   United States of America

   Email: nichols@pollere.com

   Van Jacobson
   Google

   Email: vanj@google.com

   Andrew McGregor (editor)
   Google

   Email: andrewmcgr@google.com

   Janardhan Iyengar (editor)
   Google

   Email: jri@google.com