TCP Reno

window-based
Loss-based

Four Stage:

① slow start
② congestion avoidance
③ fast retransmission
④ fast recovery

ideal TCP self-clocking

for each ack, cwnd + 1

⇓

double cwnd in each RTT

Once congestion

cwnd = cwnd / 2

then in each RTT. cwnd + 1

Resend Data[n]

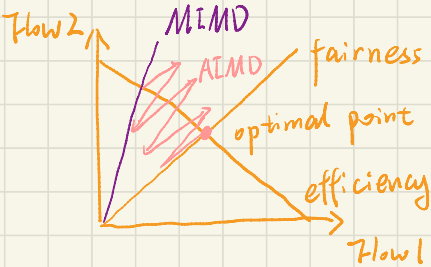
when receiving 3 dup Ack[n-1] ⇑

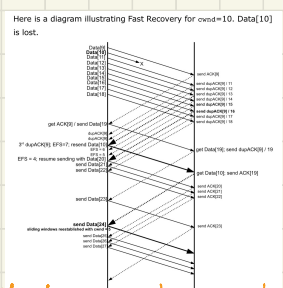
(lost Data[n] but received n+1, n+2...)

Received	Response
Data[1]	ACK[1]
Data[3]	ACK[1]
Data[4]	ACK[1]
Data[2]	ACK[4]
Data[6]	ACK[4]
Data[5]	ACK[6]

cwnd = packets in flight

= bandwidth-delay product





Why MIMD cannot reach optimal point?

AIMD → 1/2 cwnd, if 1/3, delay will decrease but utilization ↓?

Cons

① Bursty traffic when Acks come quickly

② under-utilization in lossy networks (why for Reno but not for others) (why Reno?)

③ high delay from deep buffers ← bufferbloat window-based?

④ synchronize flows ← caused by AIMD?

Active Queue Management

goal — reduce excessive queuing delays => buffer bloat

three drop modes:

① Droptail (FIFO)

② Drophead
-> to inform sender congestion as early as possible

③ Random

TCP Reno connections are happiest when the queue capacity at the bottleneck router exceeds the bandwidth × delay transit capacity.

why happiest? If not, it cannot fully utilize the bw?

RED (Random Early Detection)

The actual RED algorithm does two things. First, the base drop probability – p_{base} – rises steadily from a minimum queue threshold q_{min} to a maximum queue threshold q_{max} (these might be 40% and 80% respectively of the absolute queue capacity); at the maximum threshold, the drop probability is still quite small. The base probability p_{base} increases linearly in this range according to the following formula, where p_{max} is the maximum RED-drop probability; the value for p_{max} proposed in [FJ93] was 0.02.

$$p_{base} = p_{max} \times (avg\_queuesize - q_{min})/(q_{max} - q_{min})$$

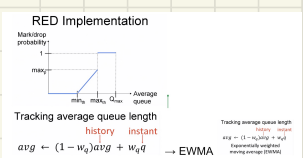
Second, as time passes after a RED drop, the actual drop probability p_{actual} begins to rise, according to the next formula:

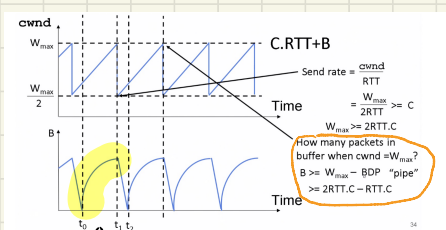
$$p_{actual} = p_{base} / (1 - count \times p_{base})$$

Here, count is the number of packets sent since the last RED drop. With count=0 we have $p_{actual} = p_{base}$, but p_{actual} rises from then on with a RED drop guaranteed within the next $1/p_{base}$ packets. This provides a

what does Pactual mean?

Once drop starts,
the packets drops as Pactual
until Avg_queue changes?





$$\text{Send rate} = \frac{cwnd}{RTT}$$

why slope of buffer change decreases?

– Queue increases linearly,
but queuing delay ↑ -> RTT ↑

– Increase the same amount,
it takes more time

CoDeL

"Sojourn Time"

measure queue utilization with the time the packets spend in the queue

Queue size is not a good measure?

Implementing CoDeL (2012)

- Buffer overflow ⇒ drop packet (as per normal)
- Track local minimum queue delay (time)
- Start dropping when delay > target
- Next drop time is decreased in inverse proportion to the square root of the number of drops since we started dropping
- When delay < target, stop dropping

measure the minimum value of queue utilization in a interval

a little larger than $RTT_{no\ load}$

drop time = $\langle 1/\sqrt{2}, 1/\sqrt{3}, 1/\sqrt{4} \rangle \times$ Interval

How can we measure fairness?

Jain's index: $f = \frac{(\sum_{i=1}^{N} x_i)^2}{(n \sum_{i=1}^{N} x_i^2)}$

Max-min fairness: Zero-sum game

Allocation that maximizes the minimum throuhput among all flows

fairness
- per-flow state
- long flow vs short flow
- Bursty flow
- flow last time

TCP Vegas

delay-based

expected throughput = $\frac{cwnd}{RTT}$ = sending rate?

Westwood+ below. TCP Vegas estimates RTT_{noLoad} by the minimum RTT (RTT_{min}) encountered during the connection. The "ideal" cwnd that just saturates the bottleneck link is $BWE \times RTT_{noLoad}$. Note that BWE will be much more volatile than RTT_{min}; the latter will typically reach its final value early in the connection, while BWE will fluctuate up and down with congestion (which will also act on RTT, but by increasing it).

As in 8.3.2 RTT Calculations, any TCP sender can estimate queue utilization as

$$queue\_use = cwnd - BWE \times RTT_{noLoad} = cwnd \times (1 - RTT_{noLoad}/RTT_{actual})$$

TCP Vegas then adjusts cwnd regularly to maintain the following:

$$\alpha \le queue\_use \le \beta$$

which is the same as

$$BWE \times RTT_{noLoad} + \alpha \le cwnd \le BWE \times RTT_{noLoad} + \beta$$

Typically α = 2-3 packets and β = 4-6 packets. We increment cwnd by 1 if cwnd falls below the lower limit (*eg* if BWE has increased). Similarly, we decrement cwnd by 1 if BWE drops and cwnd exceeds $BWE \times RTT_{noLoad}$ +

limit cwnd close to the number of packets currently in flight

(measure and control the amount of extra data in transit)

queue size

BBR

main idea — manipulate sending rate

rate-based

⇓

reduce its sending rate in response to decreases in BWE

BWE (Bandwidth Estimate)

↓

maximum rate recorded over the past ten RTTs

How to measure?

STARTUP

↳ pacing_gain = 2.89 ($2/\ln 2$)

↳ Until an additional RTT yields no improvement in BWE

DRAIN

↳ pacing_rate = $1/2.89$

PROBE_BW

Behaviour

① Set base sending rate to BWE (Update every 8 RTTs)

⇓

purpose

② Set cwnd target to $2 \times BWE \times RTT_{min}$

why?

WHY

To let bottleneck queue = transit capacity

BWE ↑ ← if BWE ↓, 0.75 → 1.25 → 1.0 ?

→ pacing_gain = 1.25

⇓

new rate = 1.25 × BWE (last 1 RTT)

⇓

IF → RTT is also 1.25 times larger

⇓

pacing_gain = 0.75

new rate = 0.75 × BWE (last 1 RTT)

⇓

pacing_gain = 1.0 (last 6 RTT)

After STARTUP, PROBE_BW and PROBE_RTT, which comes first?

BBR is TCP friendly?

PROBE_RTT

Till in-flight pkt drops to $2 \times BWE \times RTT_{min}$

① # of packets drops to 4

② stays 1 RTT_{actual}

③ Return to PROBE_BW

(negligible "wasted" throughput

200ms out of 10 sec)

↑ 1 RTT

↑ check period

In any RTT, we can either measure bottleneck bandwidth or RTT_{min}

To test BWE, packets in flight > transit capacity

→ use ACK rate to test?

To test RTT_{min}, packets in flight < transit capacity

→ "transit capacity" here is only this flow?

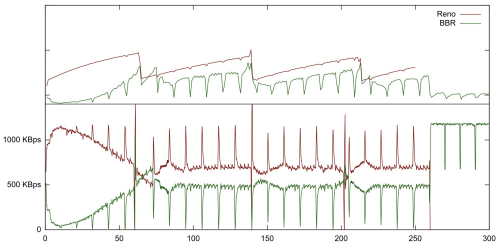
Here is a concrete example of BWE increase. To simplify the analysis, we will assume TCP BBR's FlightSize is BWE × RTT_{min}, dropping the factor of 2. Suppose a TCP BBR connection and a TCP Reno connection share a bottleneck link with a bandwidth of 2 packets/ms. The RTT_{min} (= RTT_{noLoad}) of each connection is 80 ms, making the transit capacity 160 packets. Finally, suppose that each connection has 80 packets in flight, exactly filling the transit capacity but with no queue utilization (so RTT_{min} = RTT_{actual}). Over the course of the eight-RTT `pacing_gain` cycle, the Reno connection's `cwnd` rises by 8, to 88 packets. This means the total queue utilization is now 8 packets, divided on average between BBR and Reno in the proportion 80 to 88.

Now the BBR cycle with `pacing_gain`=1.25 arrives; for the next RTT, the BBR connection has 80×1.25 = 100 packets in flight. The total number of packets in flight is now 188. The RTT climbs to 188/2 = 94 ms, and the next BBR BWE measurement is 100 packets in 94 ms, or 1.064 packets/ms (the precise value may depend on exactly when the measurement is recorded). For the following RTT, `pacing_gain` drops to 0.75, but the higher BWE persists. For the rest of the `pacing_gain` cycle, TCP BBR calculates a base rate corresponding to 1.064 × 80 = 85 packets in flight per RTT, which is close to the TCP Reno `cwnd`. See also exercise 14.0.

competitive rate. Suppose, for example, that in the BBR-vs-Reno scenario above, Reno has gobbled up a total of 240 spots in the bottleneck queue, thus increasing the RTT for both connections to (240+80)/2 = 160. During a PROBE_RTT cycle, TCP BBR will drop its link utilization essentially to zero, but TCP Reno will still have 240 packets in transit, so TCP BBR will measure RTT_{min} as 240/2 = 120 ms. After the PROBE_RTT phase is over, TCP BBR will increase its sending rate by 50% over what it had been when RTT_{min} was 80.

The lower part of the diagram shows each connection's share of the 10 Mbps (1.25 MBps) bottleneck bandwidth. The upper part shows the number of packets "in flight" (for TCP Reno, outside of Fast Recovery, that is of course `cwnd`). The Reno sawtooth pattern is clearly visible.

A dominant feature of the graph is the spikes every 10 seconds (down for BBR, correspondingly up for Reno) caused by TCP BBR's periodic PROBE_RTT mode.



Known Issues: BBR

- Not friendly with Reno/CUBIC
- Impossible to estimate RTTmin ⇒ **cwnd** = 2 x BDP ⇒ High delay
- Loss rate can be quite high for small buffers
- PROBE_RTT causes significant throughput reduction