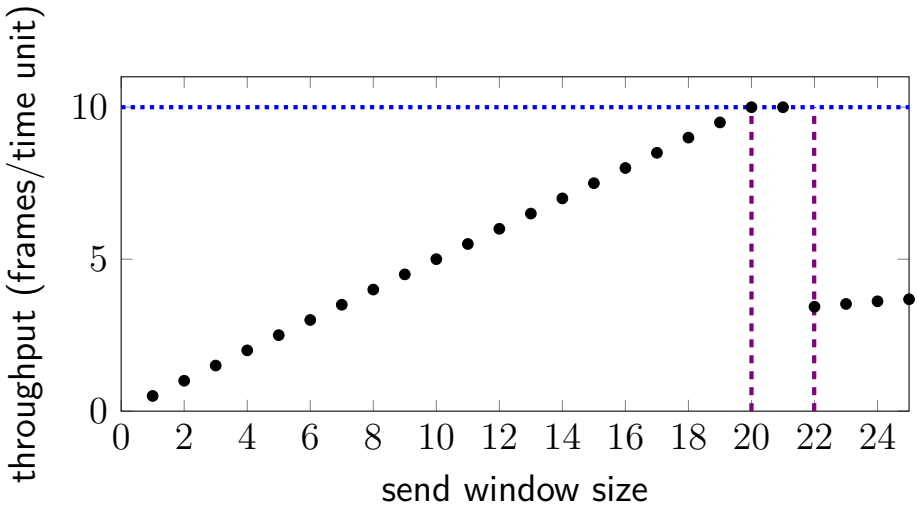
Changelog

1 Oct 2024: ECN timeline – don't have packet data modified going through switch

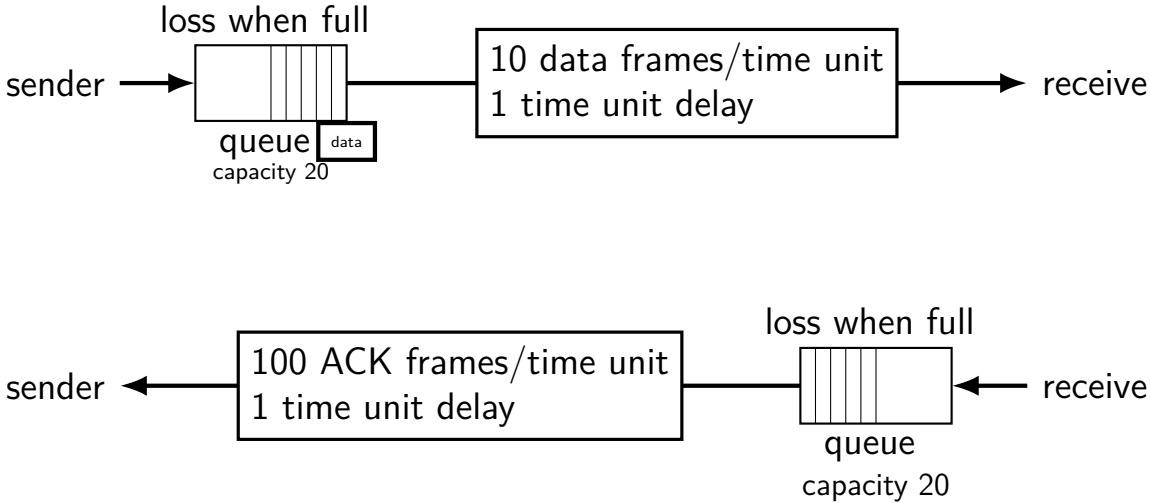
18 Sep 2025: reorder congestion collapse slides after cross-traffic slides

26 Sep 2025: correct fast recovery example to not have pkts-in-flight counting mistake at start of timeline

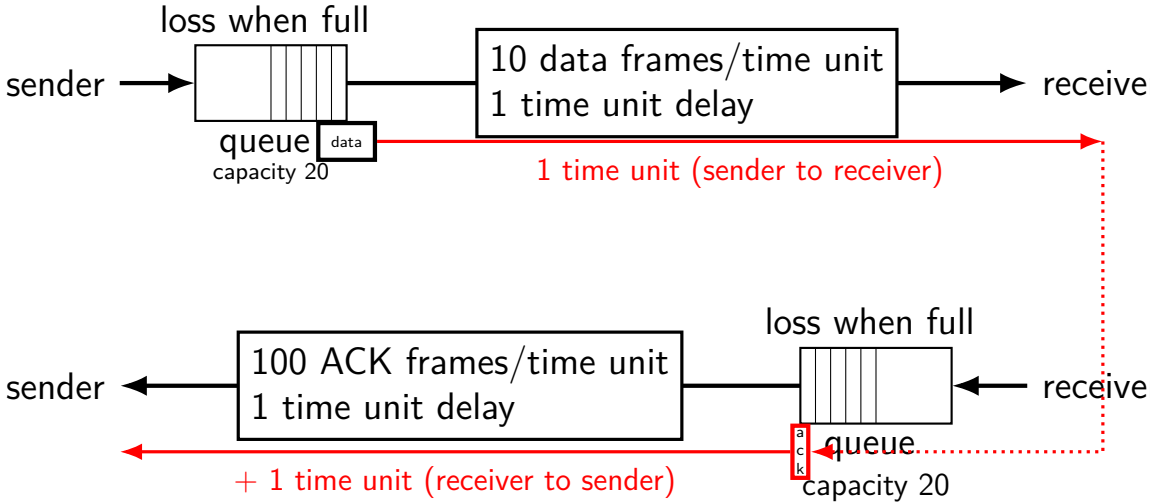
throughput and window size



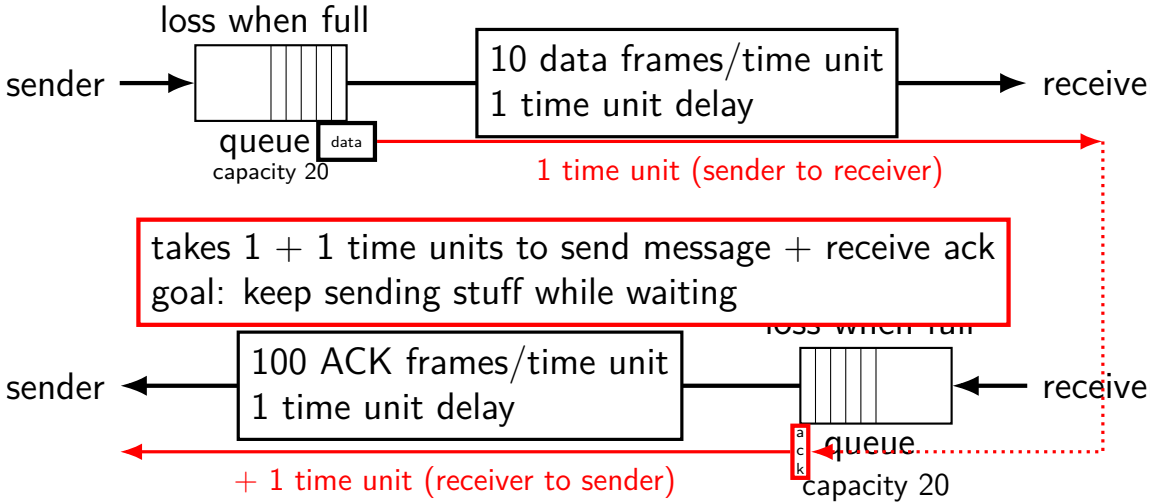
packet transit time



packet transit time



packet transit time



filling the pipe

round-trip time of 2 time units

from send data to receive ACK (assuming no queuing delay)

can send 10 data frames per time unit

= can send 20 data frames while waiting for ACK

filling the pipe

round-trip time of 2 time units

- from send data to receive ACK (assuming no queuing delay)

can send 10 data frames per time unit

= can send 20 data frames while waiting for ACK

"bandwidth-delay product"

- 10/time unit (banwidth) times 2 time unit (RTT = delay)

why optimal

in normal operation with window size W

- receive ACK for x (now $W - 1$ in flight)
- send packet $x + W$
- receive ACK for $x + 1$
- send packet $x + W + 1$
- ...

window size keeps W packets in flight

if links + queues can hold W packets — perfect!

number in flight on losses

window size W

let's say we lose packet x [only], sender might

- receive ACK for $x - 1$
- send packet $x + W - 1$
- receive ACK for x, x, x, ...
- resend packet x (guess it is lost)
- receive ACK for x, x, x, ...
- receive ACK for packet $x + W - 1$
- send packets $x + W$ through $x + W + W - 1$

number in flight on losses

window size W

let's say we lose packet x [only], sender might

- receive ACK for $x - 1$
- send packet $x + W - 1$
- receive ACK for x, x, x, ...
- resend packet x (guess it is lost)
- *receive ACK for x, x, x, ...*
- receive ACK for packet $x + W - 1$
- send packets $x + W$ through $x + W + W - 1$

lots of time where we are not sending packets
means network is underutilized

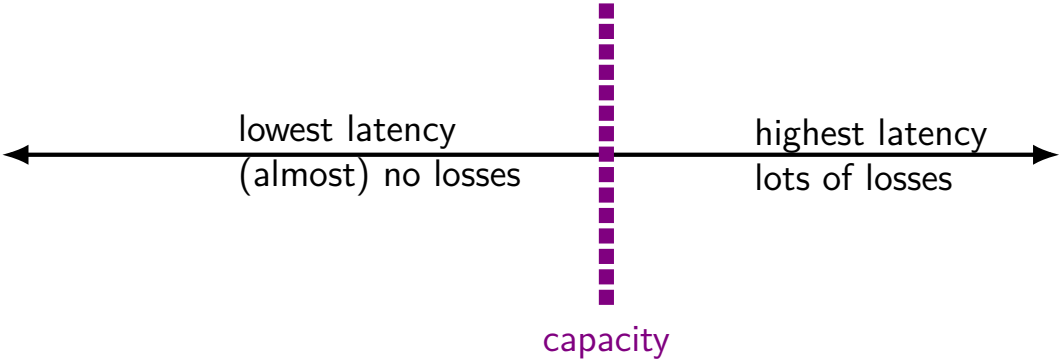
window size tweaking

window size imperfect proxy for # packets in flight

we'll ignore the difference for now

our goal for now: window size = number of packets to have in flight

finding window size empirically (1)



key insight

latency/loss rate increases when window size too big

latency/loss rate stable when window size not too big

for now, we'll focus on loss rate

- but you can do something similar with latency

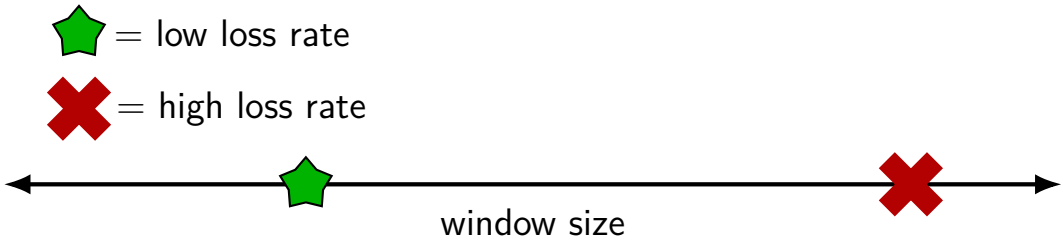
try a bunch of things

 = low loss rate

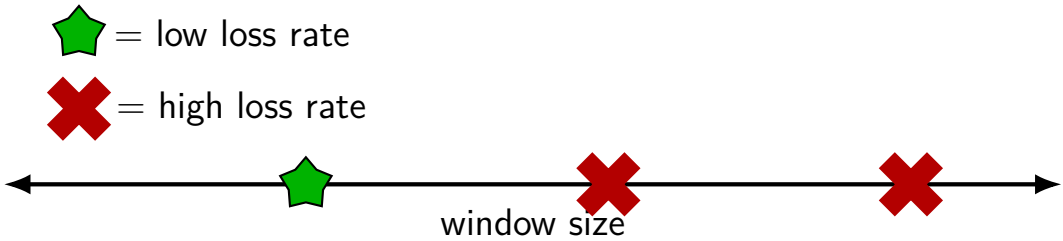
 = high loss rate

window size

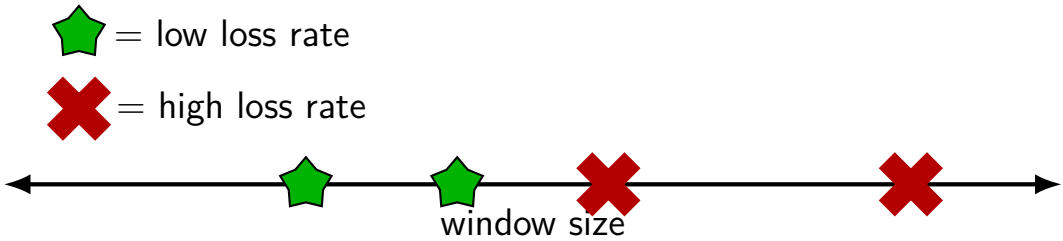
try a bunch of things



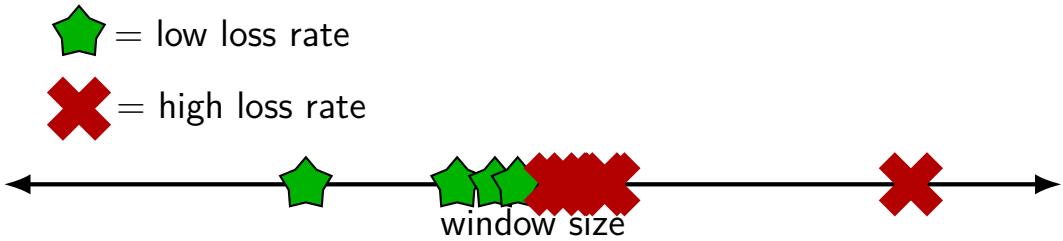
try a bunch of things



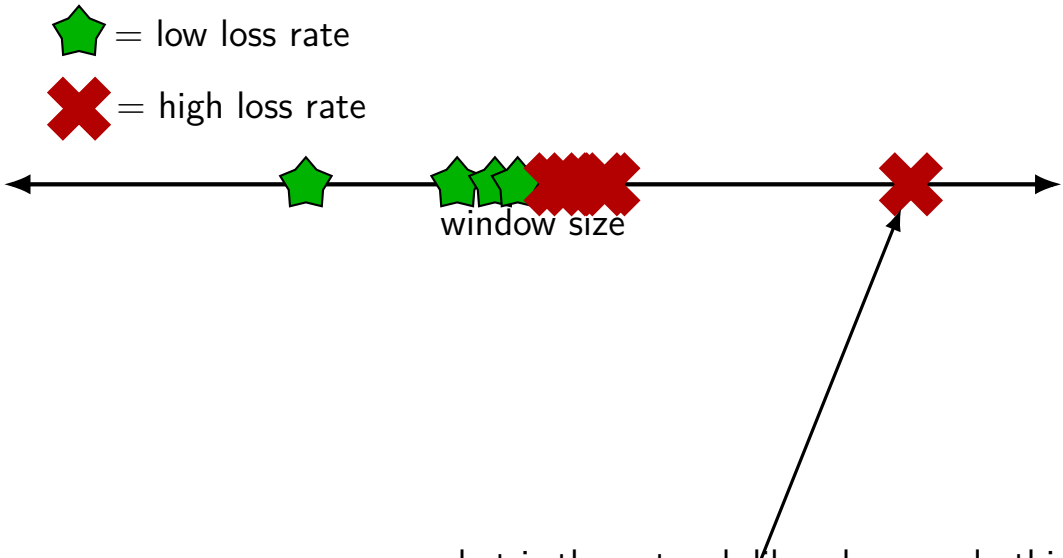
try a bunch of things



try a bunch of things



try a bunch of things



revisiting congestion collapse

Congestion Avoidance and Control

Van Jacobson*

University of California
Lawrence Berkeley Laboratory
Berkeley, CA 94720
van@helios.ee.lbl.gov

In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and three IMP hops) dropped from 32 Kbps to 40 bps. Mike Karels[1] and I were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. We wondered, in particular,

fixes from Jacobson's 1987 paper

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

the overloaded switch

let's say switch can handle 50 packets/second

but has:

- 100 packets/second from test flow sending as fast as it can
- 10 packets/second from other session

expected *loss rate* (% packets lost)?

expected % test flow packets lost?

expected other session packets lost?

modeling who gets dropped

it kinda does matter...

sending in big bursts or spread out ("pacing")?

- bursts can overload queues even though average rate is low

how switch's queue works?

- queue size (handling bursts), way to choose what to drop

random or fixed intervals between sending?

modeling who gets dropped

it kinda does matter...

sending in big bursts or spread out ("pacing")?

- bursts can overload queues even though average rate is low

how switch's queue works?

- queue size (handling bursts), way to choose what to drop

random or fixed intervals between sending?

but we'll *simplify*, assuming—

- a flow's arrivals are randomly spaced
- drops hit packets at random
- queue is "pretty big"

the overloaded switch

let's say switch can handle 50 packets/second

but has:

- 100 packets/second from test flow (checking window size)
- 10 packets/second from other session

expected *loss rate* (% packets lost)? $\frac{100 + 10 - 50}{100 + 10} = 54\%$

expected % test flow packets lost? 54%

expected % other session packets lost? 54%

the overloaded switch

let's say switch can handle 50 packets/second

but has:

- 100 packets/second from test flow (checking window size)
- 10 packets/second from other session

expected *loss rate* (% packets lost)? $\frac{100 + 10 - 50}{100 + 10} = 54\%$

expected % test flow packets lost? 54%

expected % other session packets lost? 54%

...but I missed something

a virtuous cycle

what is other session going to when 54% of its packets are lost?

- probably resend them

what about when resent packets are lost?

- probably resent again

if other session doesn't slow down, then...

10 pkt/s $\rightarrow 10 + 54\% \cdot 10 + 54\%^2 \cdot 10 \ldots \approx 22$ pkt/s

the overloaded switch (revised)

let's say switch can handle 50 packets/second

but has:

100 packets/second from test flow (checking window size)
10 packets/second from other session $\rightarrow$ 22 with resends

expected *loss rate* (% packets lost)? $\frac{100 + 22 - 50}{100 + 22} = 59\%$

expected % test flow packets lost? 59%

expected % other session packets lost? 59%

the overloaded switch (revised)

let's say switch can handle 50 packets/second

but has:

- 100 packets/second from test flow (checking window size)
- 10 packets/second from other session $\rightarrow$ 22 with resends

expected *loss rate* (% packets lost)? $\frac{100 + 22 - 50}{100 + 22} = 59\%$

expected % test flow packets lost? 59%

expected % other session packets lost? 59%

- means that 22 pkt/sec is slight underestimate
- though realistically other session should slow down

aside: latency (1)

589

need one round-trip time (RTT) to detect loss

- probably from duplicate ACK
- if detecting via timeout, probably longer

so need 1.4 RTTs (detecting loss 1.4 times) extra time

mean latency $= \frac{1.4\text{RTTs}}{0.5\text{RTTs}}$ times normal $= 2.8$ times normal

aside: high-percentile latency

59% packet loss

about 10% of time need more than 4 retransmissions

about 5% of the time need more than 5 retransmissions

about 1% of the time need more than 8 retransmissions

sliding windows and retransmissions

assuming that other session doesn't slow down

sliding window approach slows down on losses

sliding window throughput collapse

let's say doing sliding window with 100 packet window

if 1% of the time, we need to resend a packet 8 times, then

probably need around 8 RTTs to send all 100 packets in window

sliding window throughput collapse

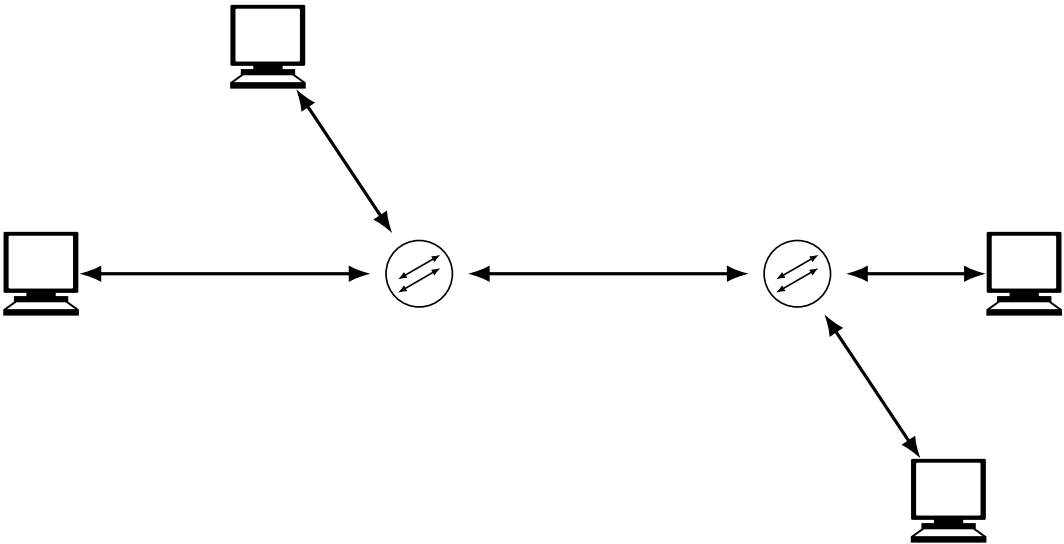
let's say doing sliding window with 100 packet window

if 1% of the time, we need to resend a packet 8 times, then

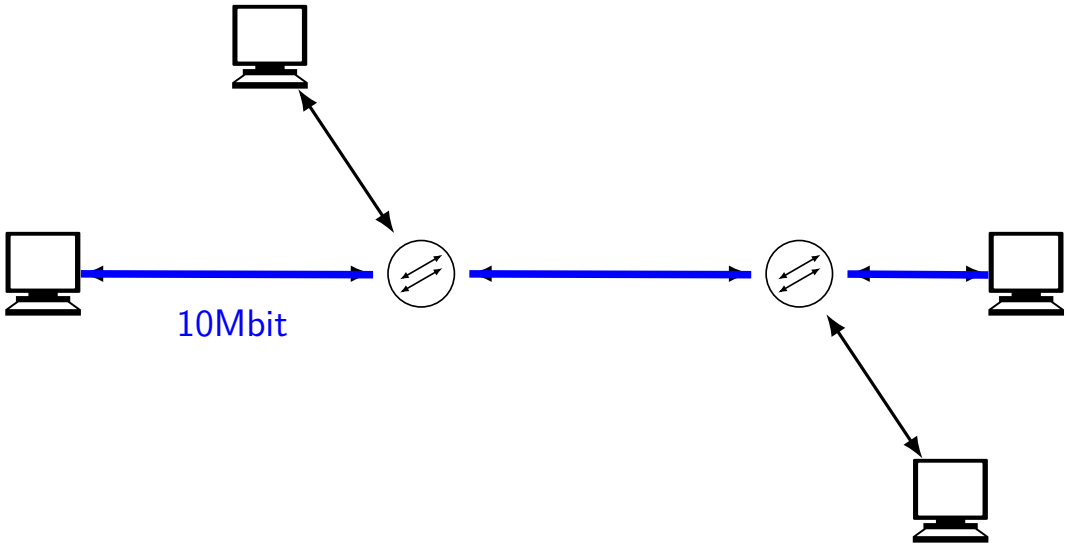
probably need around 8 RTTs to send all 100 packets in window

$\approx$ 8 times slower with same window size

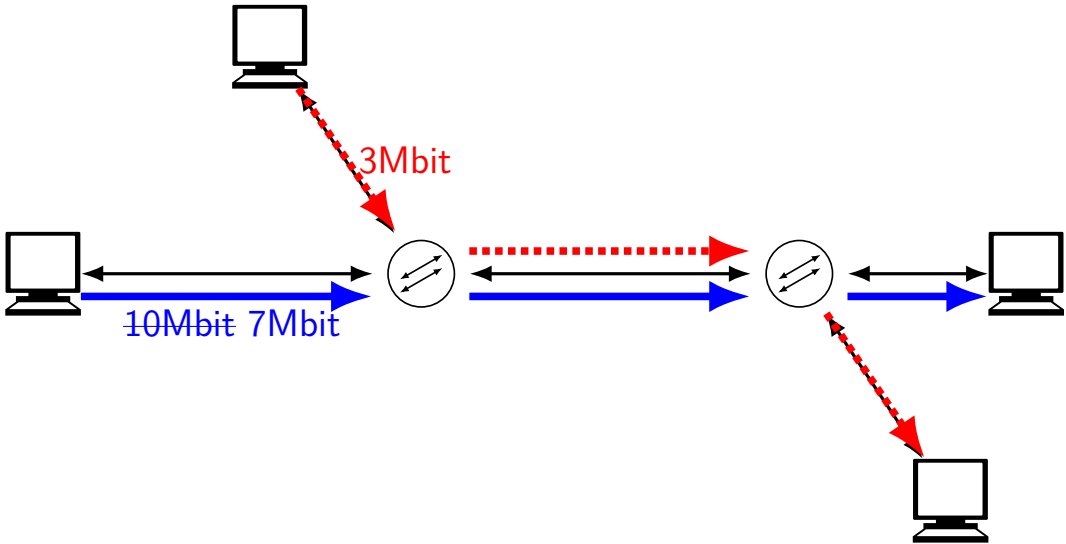
changing cross-traffic



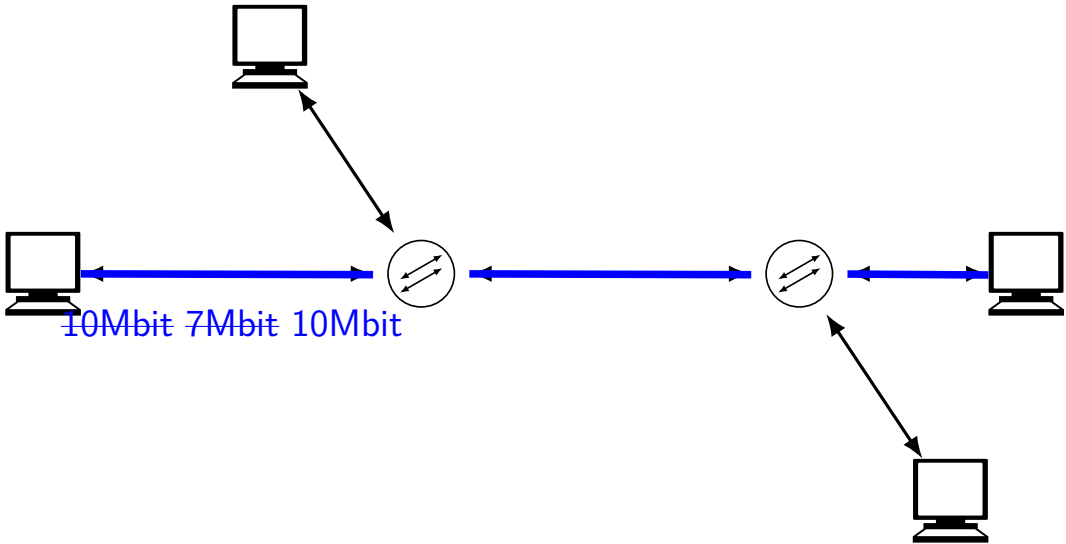
changing cross-traffic



changing cross-traffic



changing cross-traffic



adapting to cross-traffic

available bandwidth will change

previous example: 3Mbit lost/added from other flow

need to adapt to lost bandwidth

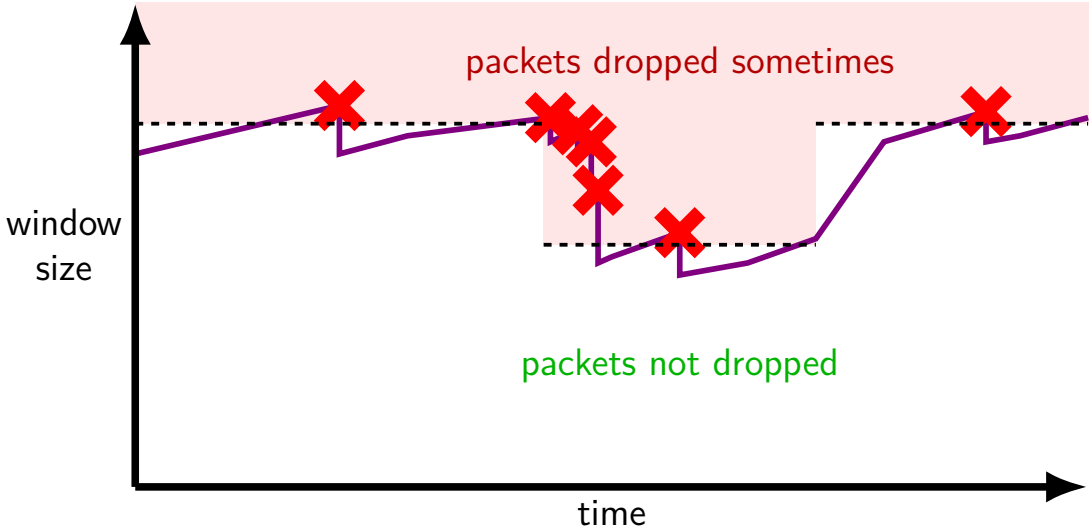
need to detect new available bandwidth

other flow's bandwidth?

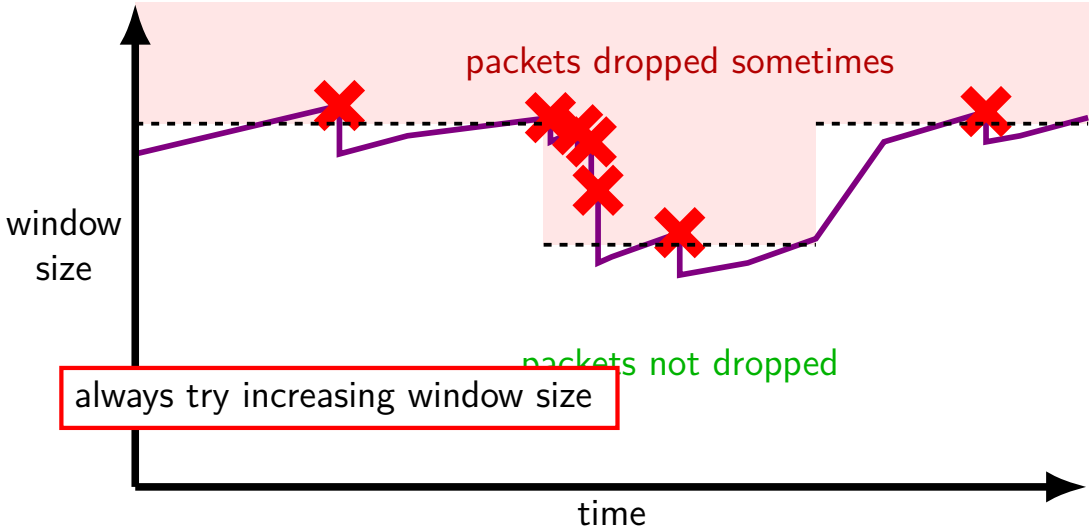
for now, we'll pretend other flows don't react to us

later topic: what happens when both reacting?

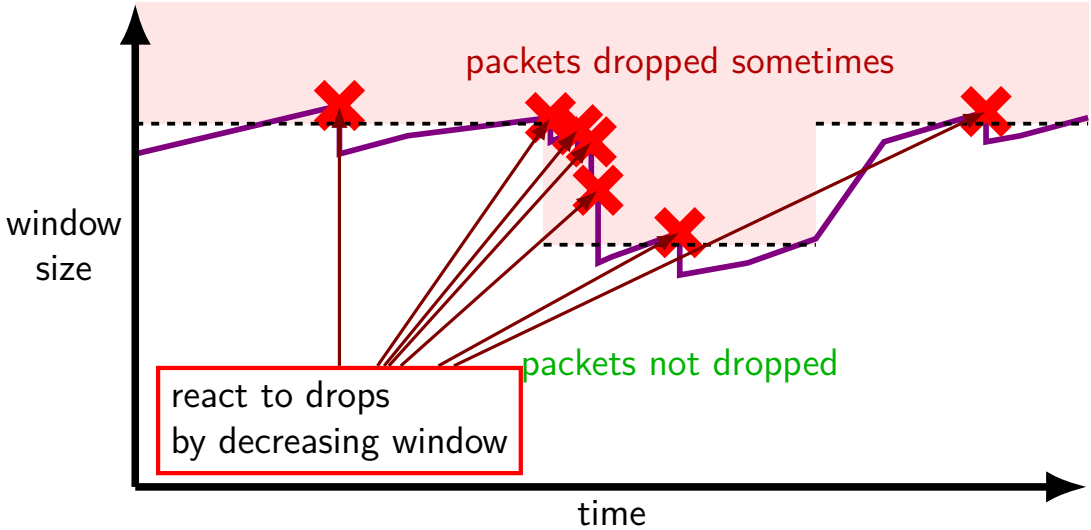
window size experimenting



window size experimenting



window size experimenting



increase/decrease strategy

default to increasing window size

react to packet drops by decreasing window size

assumption: few “non-congestion” packet losses

increase/decrease strategy

default to increasing window size

react to packet drops by decreasing window size

assumption: few "non-congestion" packet losses

increase/decrease strategy

default to increasing window size

react to packet drops by decreasing window size

- assumption: few "non-congestion" packet losses

big topic: how fast to do each?

questions to help decide that:

- what happens if we increase too fast? too slow?
- what happens if we decrease too fast? too slow?

slow increase

want to increase *slowly* to avoid overload

original TCP: $+1$ packet/round trip time

slow increase

want to increase *slowly* to avoid overload

original TCP: $+1$ packet/round trip time

$+1$ certainly not optimal choice, but okay heuristic

important theoretically: approx. *additive* increase

helps ensure good behavior with multiple connections
(we'll talk later about why)

exercise: convergence time (1)

suppose: 50 ms round trip time

initially sending at 600 packets/second
≈ 0.9Mbyte/sec with 1500 byte packets

optimal rate is 10000 packets/second
≈ 15Mbyte/sec with 1500 byte packets

'standard' TCP increase of 1 packet/RTT

how long to get there?

exercise: convergence time (1)

suppose: 50 ms round trip time

initially sending at 600 packets/second
≈ 0.9Mbyte/sec with 1500 byte packets

optimal rate is 10000 packets/second
≈ 15Mbyte/sec with 1500 byte packets

'standard' TCP increase of 1 packet/RTT

how long to get there?

current: 30 packets/RTT (= window size 30)

need to get to: 500 packets/RTT

will take $500 - 30 = 470$ round trips $\approx$ 23500 ms $\approx$ 24 s

fixing bad convergence time

TCP's additive increase is very slow for "high bandwidth-delay" networks

two things make this better:

not in additive increase mode at start of connection

- "slow start" we'll talk about later

more adaptive increase for modern TCP variants

- e.g. FAST TCP, CUBIC TCP, ...
- heuristics to increase faster when appropriate

heurstic: fast decrease (v1)

want to decrease quickly to get out of overload

original TCP heuristic: reset window to (minimum) 2 segments

TCP Tahoe (1)

first version of TCP congestion control

track two variables

- cwnd = congestion window
- `ssthresh` = 'slow start threshold'

three modes of operation:

congestion avoidance (cwnd $\geq$ ssthresh; no known loss)

recovery (known loss)

"slow start" (cwnd $<$ ssthresh)

TCP Tahoe (2)

congestion avoidance (cwnd $\geq$ ssthresh; no known loss)

- represents "steady state" — our focus for now
- no loss: cwnd $+=$ 1 segment per round-trip-time
- on loss: cwnd $=$ $\sim$2 segments; ssthresh $=$ cwnd$/2$

recovery (known loss)

- retransmit packet after third duplicate ACK of previous packet (or timeout)

"slow start" (cwnd $<$ ssthresh)

- intuition: not in "steady state" yet
- want to get there quickly, but with overloading network
- we'll give details later

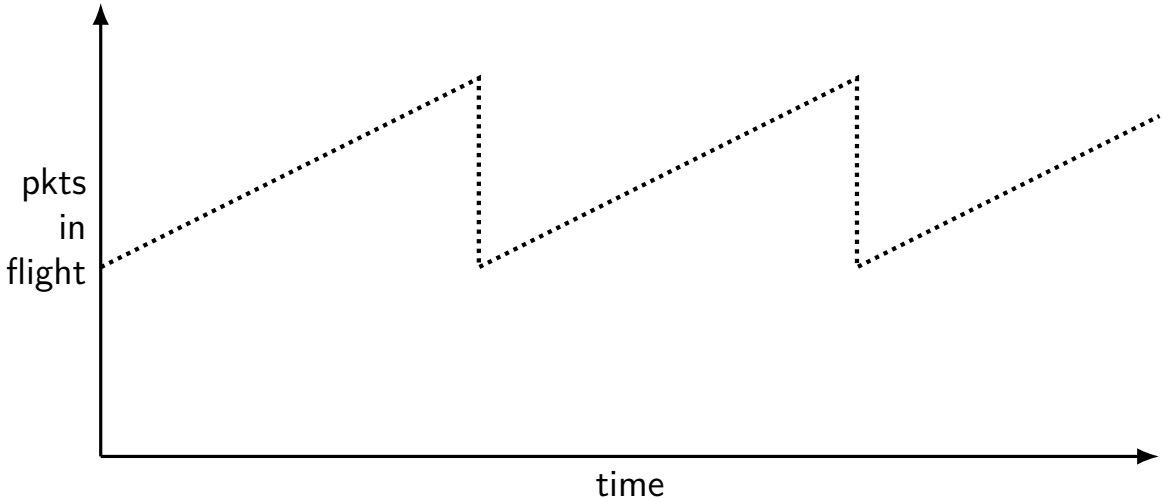
plan

really don't want to reset to beginning on loss

- even if we have a way to get to good size quickly

would like to less extreme decrease: cwnd $=$ cwnd$/2$

sawtooth pattern



Linux's ss command

```
$ ss --info --numeric --tcp state established
...
Recv-Q  Send-Q   Local Address:Port          Peer Address:Port
0       2583232  128.143.71.27:5201          128.143.63.240:55926
cubic wscale:7,7 rto:204 rtt:1.891/0.125 ato:40 mss:1448 pmtu:1500
rcvmss:536 advmss:1448 cwnd:309 ssthresh:64 bytes_sent:152936312
bytes_acked:152698840 bytes_received:37 segs_out:105620 segs_in:7024
data_segs_out:105619 data_segs_in:1 send 1.89Gbps lastrcv:1312
pacing_rate 2.27Gbps delivery_rate 996Mbps delivered:105456
busy:1308ms unacked:164 rcv_space:14600 rcv_ssthresh:64076
notsent:2345760 minrtt:0.219
```

`cubic` = which congestion control algorithm

cwnd = 309 segments; `ssthresh` = 64 segments

cwnd = 309 segments $\times$ 1488B max segment size = 437 KiB

packet rate $\approx$ cwnd / rtt = 224 MiB/sec = 1879 Mbit/s

fixes from Jacobson's 1987 paper

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

handling steady state

most of the time we should be at approx. correct window size

want to focus on how we react to changes

still going to use “experimentation” idea

TCP (New)Reno

base for 'modern' TCP (changes from Tahoe highlighted)

congestion avoidance (cwnd $\geq$ ssthresh; no known loss)

- no loss: cwnd $+=$ 1 segment per round-trip-time
- *on loss (dup ACK): cwnd = cwnd/2; ssthresh = new cwnd*
- on loss (timeout): cwnd $=$ 2 segments

recovery (known loss)

- resend immediately on third dup ACK
- *send new data based on self-clocking idea ("fast recovery")*

"slow start" (cwnd $<$ ssthresh)

- (details later)

fast decrease (v2)

resetting window size to 1 is too slow

compromise:

- loss detected without timeout: divide window size by two
- loss detected with timeout: decrease to 1

fast decrease (v2)

resetting window size to 1 is too slow

compromise:

- loss detected without timeout: divide window size by two
- loss detected with timeout: decrease to 1

exactly by two probably not important

important theoretically: *multiplicative* decrease

- will help show okay behavior with multiple flows

AIMD

result: additive increase + multiplicative decrease

preview: good theoretical properties for *sharing connections*

doesn't matter much what additive/multiplicative factor is!

but: not quite what current Internet does

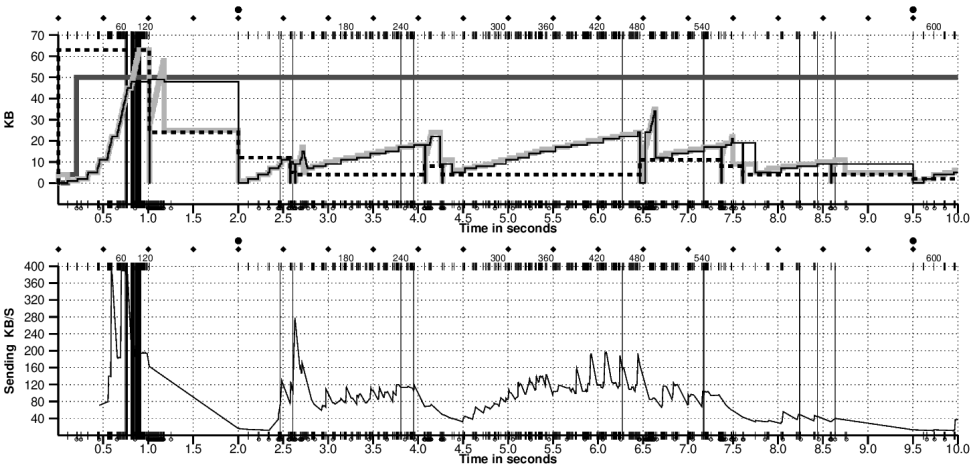


Figure 1: TCP Reno Trace Examples.

from Brakmo, O'Malley, and Peterson, "TCP Vegas: New techniques for congestion detection and avoidance"

top thick, light-grey line = congestion window; dotted = slow start threshold

non-congestion losses

we were ignoring non-congestion losses

suppose *1%* loss rate from transmission errors

if huge bandwidth, 50 ms RTT

with TCP heuristics (+1 packet/RTT, half on loss)...

normal window size? achieved bandwidth (pkts/sec)?

non-congestion losses

we were ignoring non-congestion losses

suppose *1%* loss rate from transmission errors

if huge bandwidth, 50 ms RTT

with TCP heuristics (+1 packet/RTT, half on loss)...

normal window size? achieved bandwidth (pkts/sec)?

window size increases for about 100 packets, then halves

starting at window size 8:

8, 9 (17 total), 10 (27), 11 (38), 12 (50), 13 (63), 14 (77), 15 (92), 16 (>100)

→ window size fluctuates from around 8 to 16

non-congestion losses

we were ignoring non-congestion losses

suppose *1%* loss rate from transmission errors

if huge bandwidth, 50 ms RTT

with TCP heuristics (+1 packet/RTT, half on loss)...

normal window size? achieved bandwidth (pkts/sec)?

window size increases for about 100 packets, then halves

starting at window size 8:

8, 9 (17 total), 10 (27), 11 (38), 12 (50), 13 (63), 14 (77), 15 (92), 16 (>100)

→ window size fluctuates from around 8 to 16

12 pkts/50 ms = 240 pkts/sec

non-congestion losses and congestion control

significant non-congestion losses $\rightarrow$ very bad performance with most congestion control

reason why wireless, etc. often does its own acknowledgements and resending

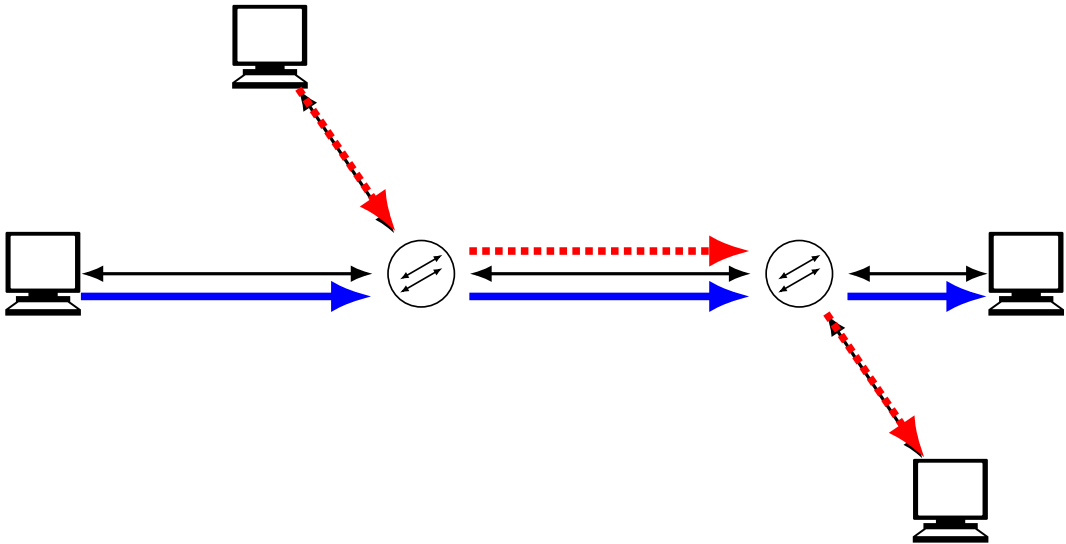
congestion: sharing

want to consider multiple flows

key questions:

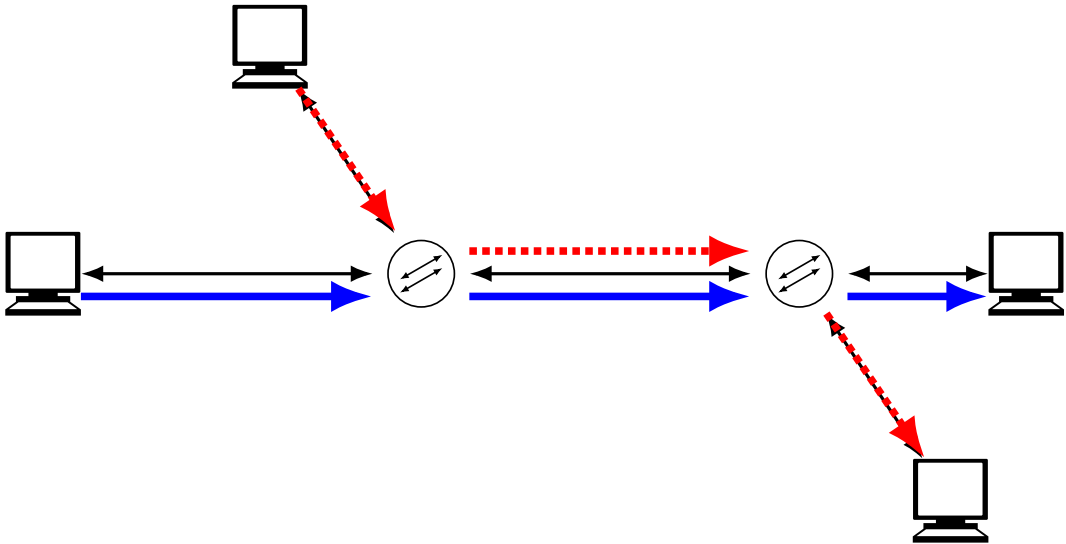
- is it stable if both flows changing window sizes?
- is there one winner/loser?
- is the winner/loser who we want it to be?

exericse: what should happen?



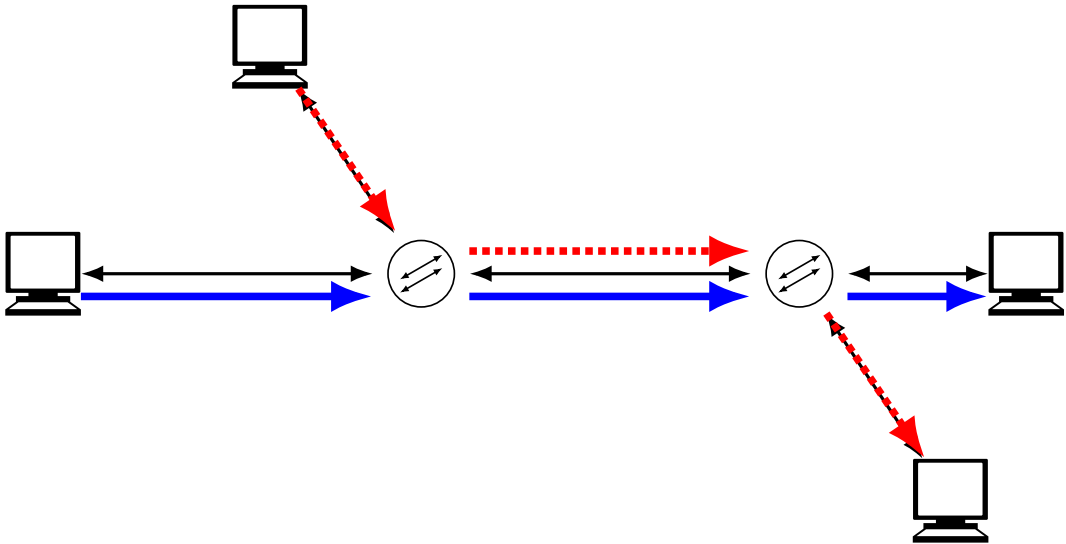
two connections on shared link

exericse: what should happen?



two connections on shared link

exericse: what should happen?



two connections on shared link

showing AIMD

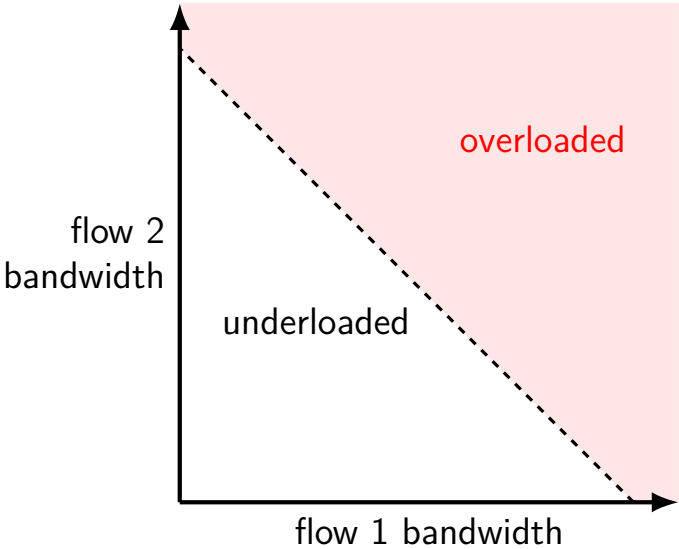
slides based on Chiu and Jain, "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks"'

1989 paper

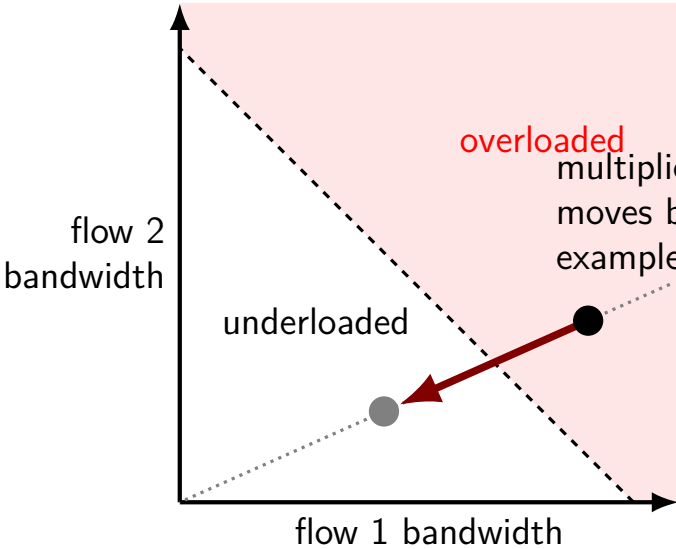
you might notice 1989 is well after TCP was in use

(kinda deployed without all the theory being developed...
...and it's still not really a solved problem)

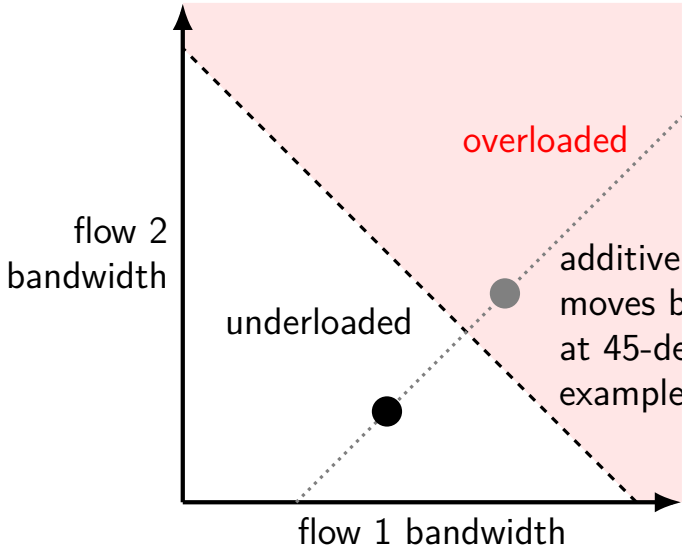
picturing sharing



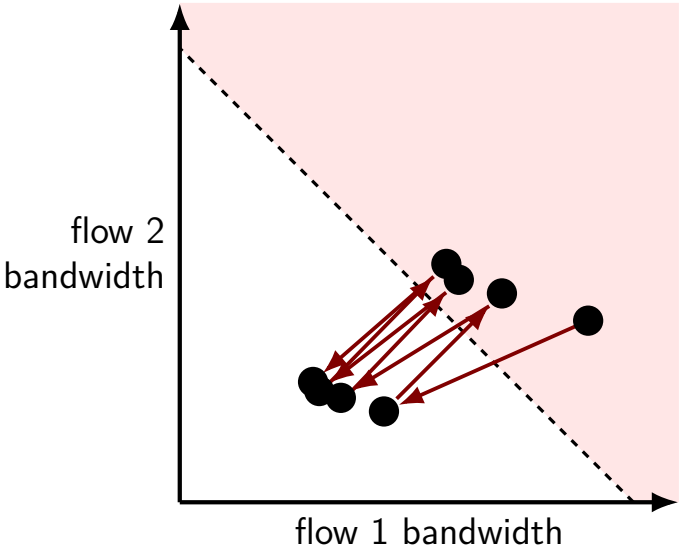
picturing sharing



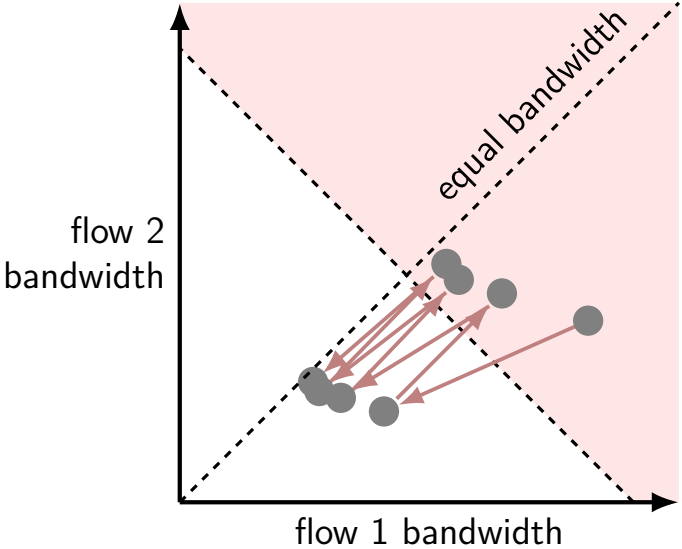
picturing sharing



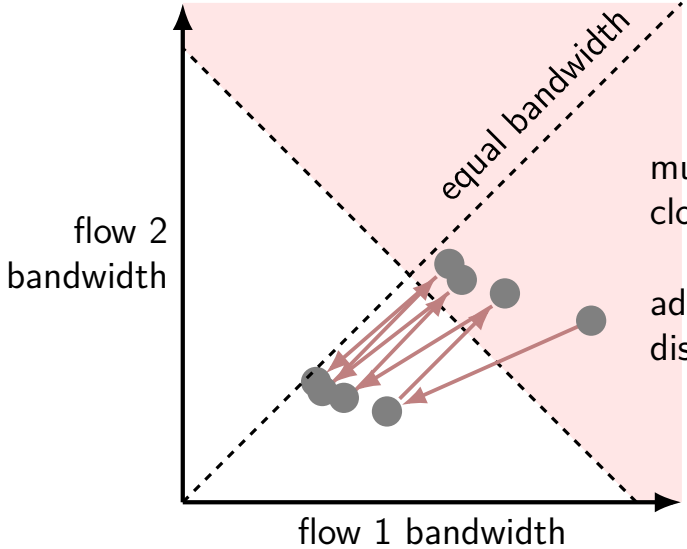
picturing sharing



picturing sharing



picturing sharing



multiplicative decrease brings closer to equal bandwidth line

additive increase keeps same distance from line

some assumptions we made

...that are probably not true

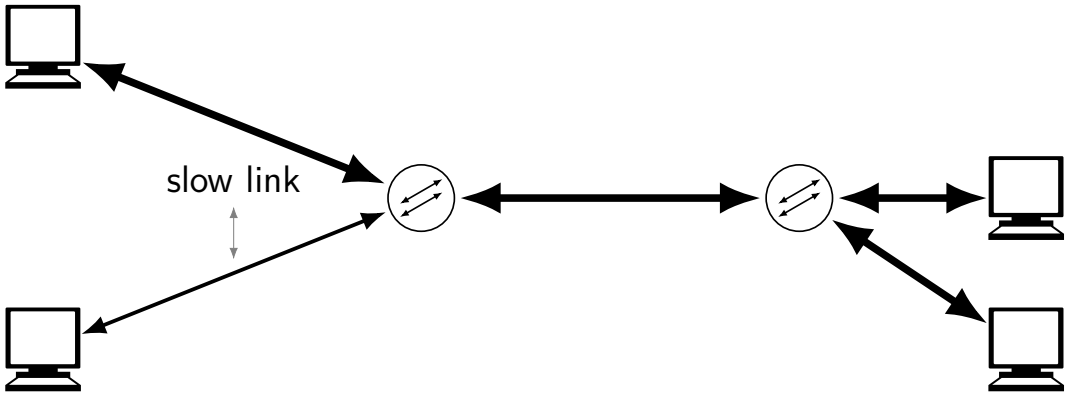
both flows experience drops equally when network overloaded

- higher-bandwidth flow more likely to see drops?
- depends on how switches manage queues?
- depends on how 'bursty' hosts are?

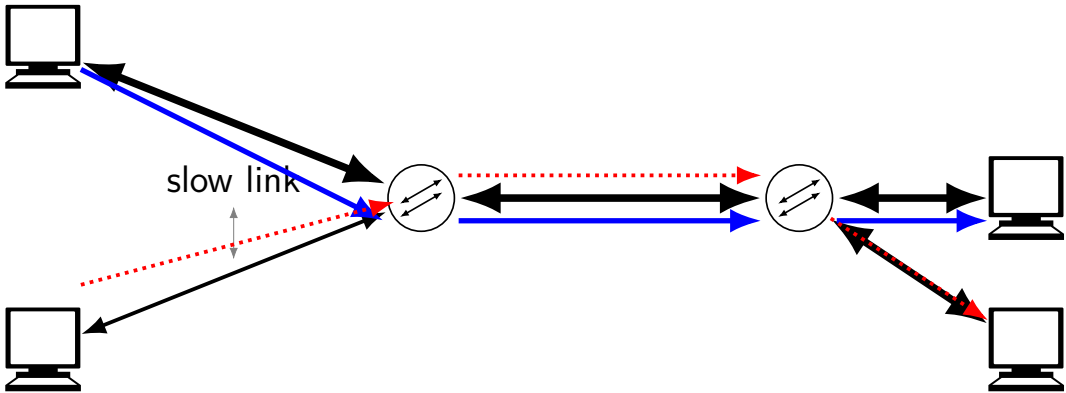
same additive increase factor (for 45 degree angle)

- TCP so far: $+1$ segment/round-trip-time
- round trip time, segment size may vary between flows

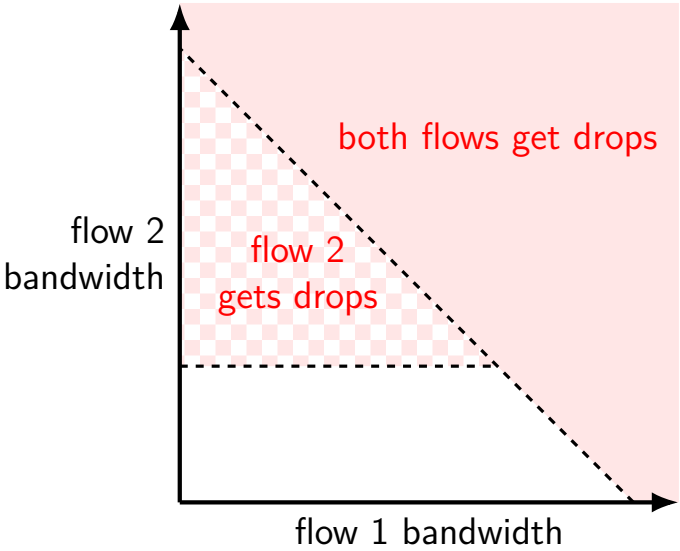
a scenario



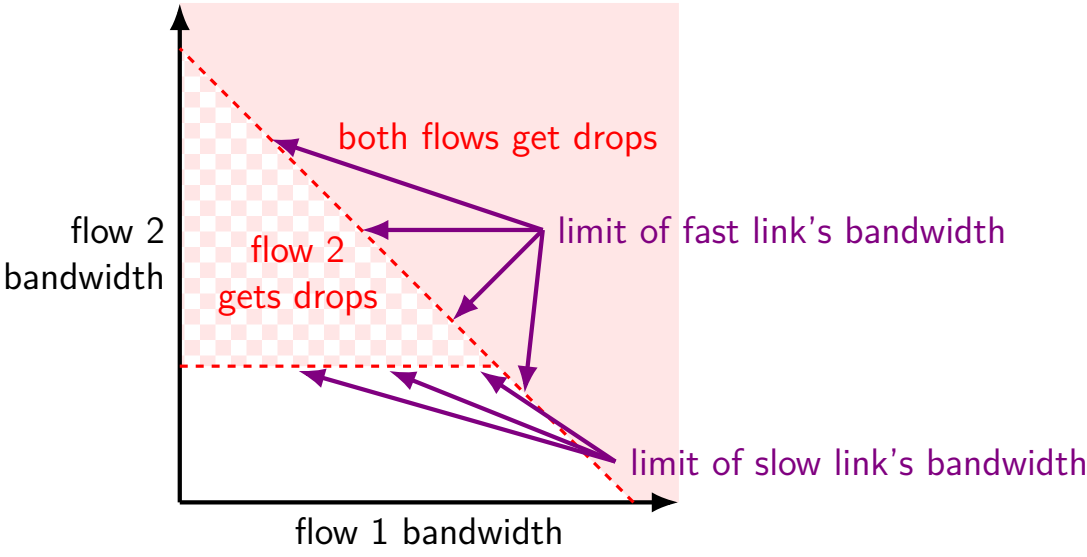
a scenario



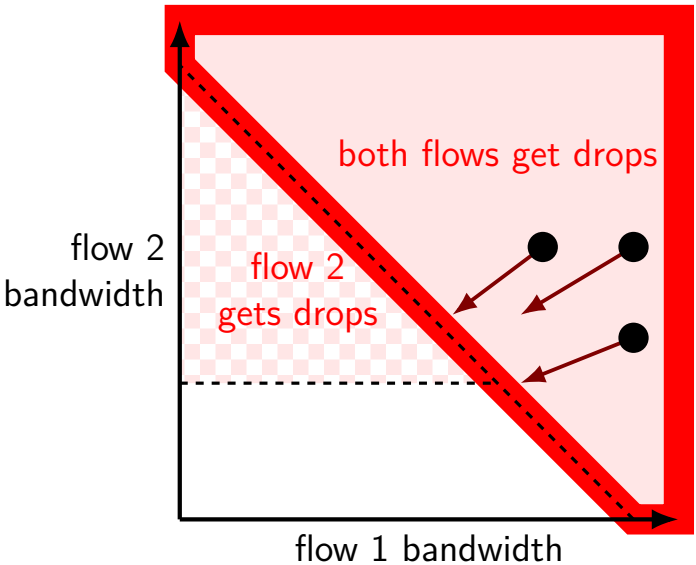
slow link + mixed traffic



slow link + mixed traffic

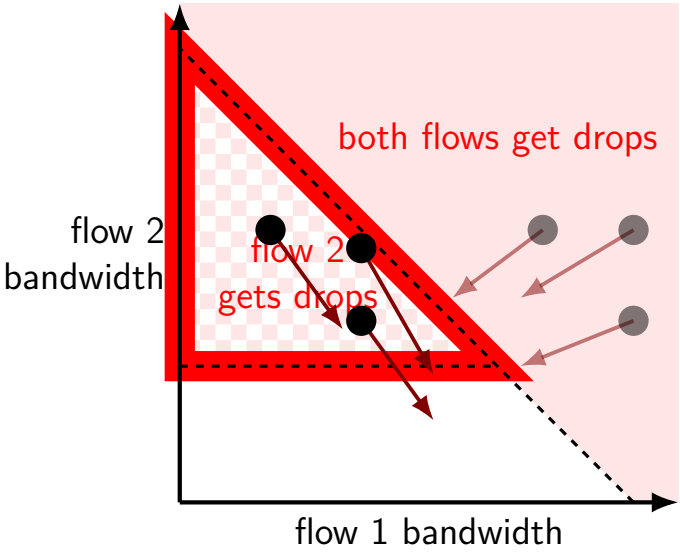


slow link + mixed traffic



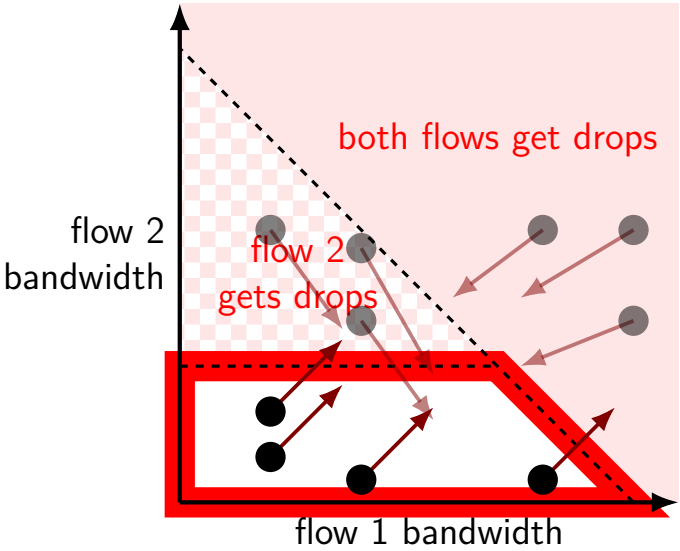
if both flows see drops, multiplicative decrease (toward origin)

slow link + mixed traffic



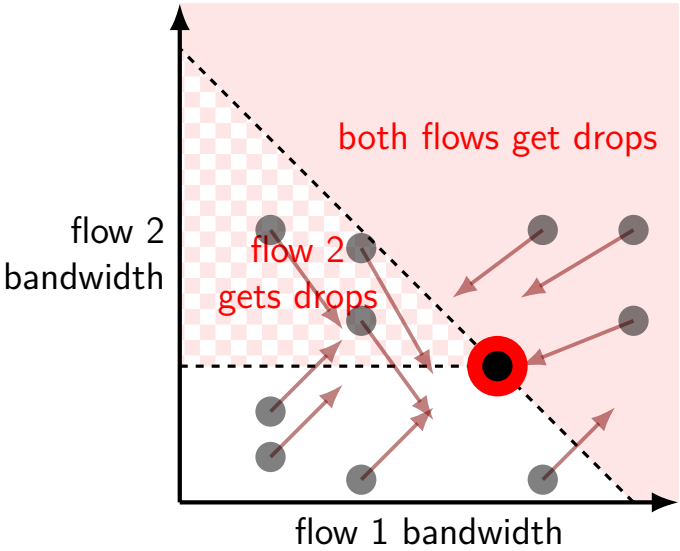
if only flow 2 see drops,
it decreases bandwidth and
flow 1 increase bandwidth

slow link + mixed traffic



if both flows see no drops,
both increase additively
(45 degree angle)

slow link + mixed traffic



result: flow 2 reaches limit of slow link
flow 1 gets the rest of the bandwidth

fairness metrics

would like to say both allocations are ‘fair’

easy when ideal allocation is equal, but that’s not always the case

perhaps not equal, but most equal we can give on network

would like some way of formalizing this

fairness intuition

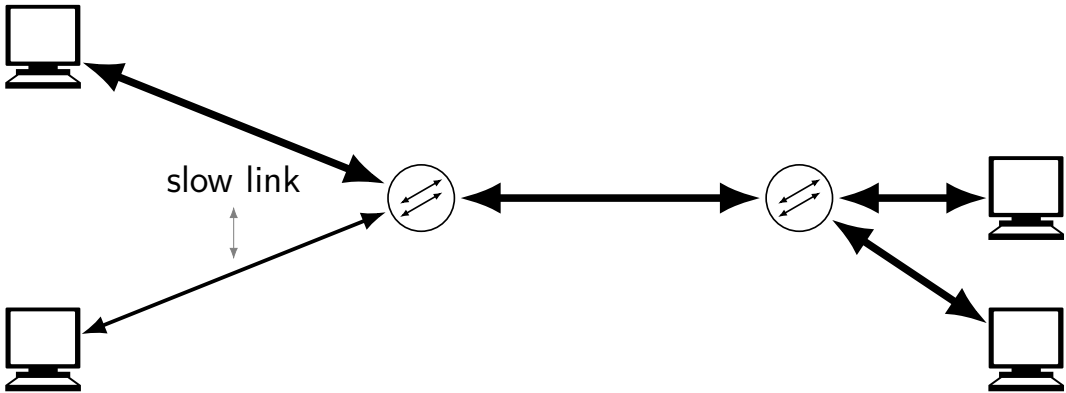
unfair allocation = someone gets much less than others

consequence: let's look for "starved" flow

if we can add to one flow...
and only hurt flows that are slower than it...
then that's "unfair"

idea called *min-max fairness*

exercise



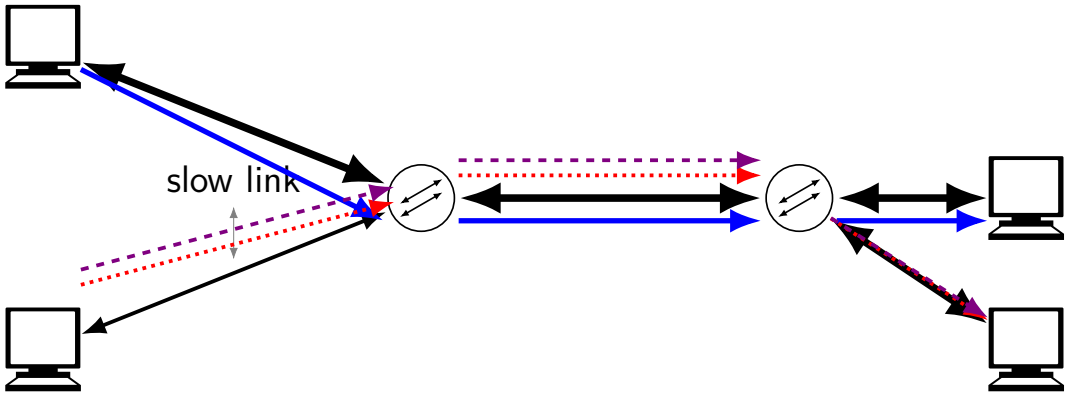
let's say slow link has 5 MByte/s capacity, other links 20MByte/s

why are these fair/unfair? (by min-max fairness)

solid = 10MByte, dotted = 2MByte/s, dashed = 3MByte/s

solid = 16MByte, dashed = 2MByte/s, dashed = 2MByte/s

exercise



let's say slow link has 5 MByte/s capacity, other links 20MByte/s

why are these fair/unfair? (by min-max fairness)

solid = 10MByte, dotted = 2MByte/s, dashed = 3MByte/s

solid = 16MByte, dashed = 2MByte/s, dashed = 2MByte/s

Jain's fairness index

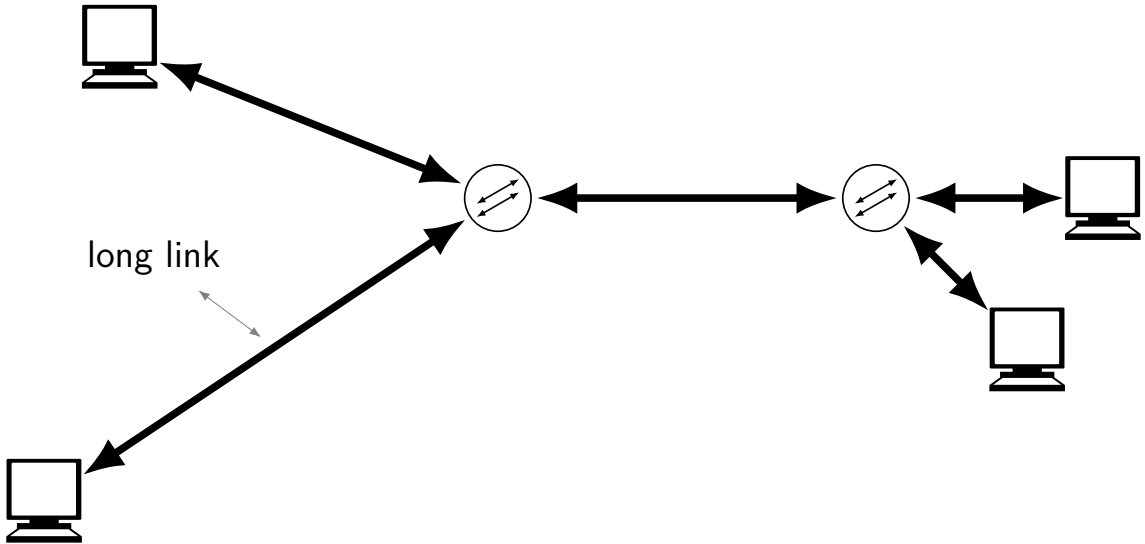
more common metric, but for scenarios where equal allocation makes sense

if x_i is i'th flow's share:

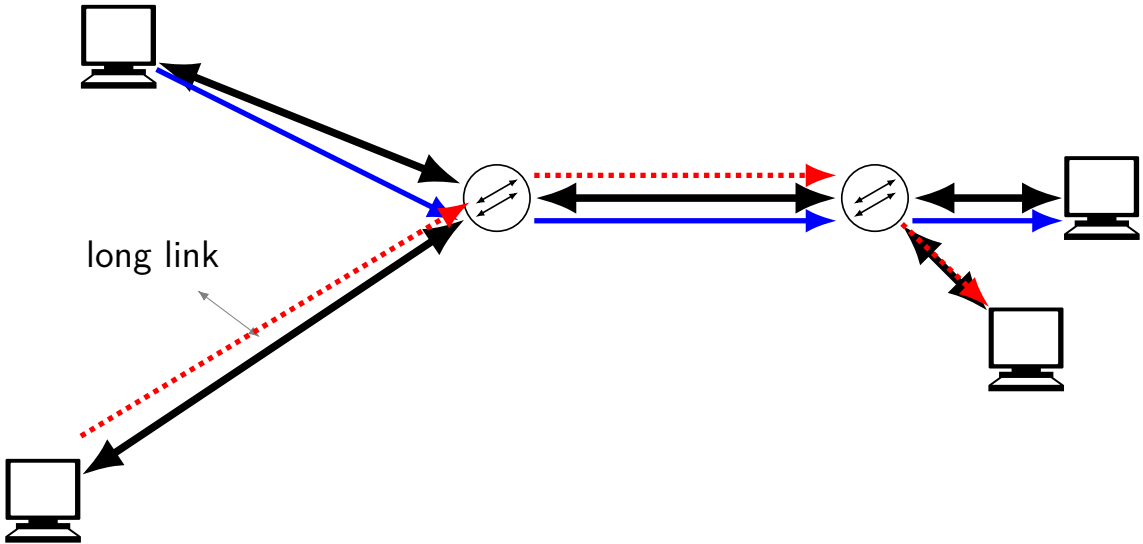
$$\frac{\left(\sum x_i\right)^2}{n \cdot \sum x_i^2}$$

approaches 1 when allocations equal, $\frac{1}{n}$ if one flow gets everything

long links



long links



exercise: window size?

flow 1: 500 packets/sec, 50 ms round trip

flow 2: 500 packets/sec, 100 ms round trip

exercise: what window size achieves this for each flow?

exercise: window size?

flow 1: 500 packets/sec, 50 ms round trip

flow 2: 500 packets/sec, 100 ms round trip

exercise: what window size achieves this for each flow?

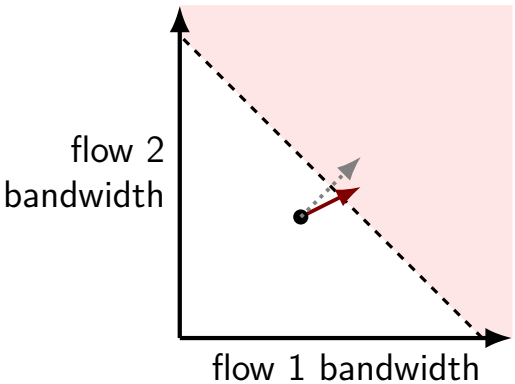
1: window size 25; 2: window size 50

revisiting additive increase

TCP: add +1 to window size each round trip time

flow 1: window 25+1 → 26 pkt/50 ms = 520 pkt/sec

flow 2: window 50+1 → 51 pkt/100 ms = 510 pkt/sec



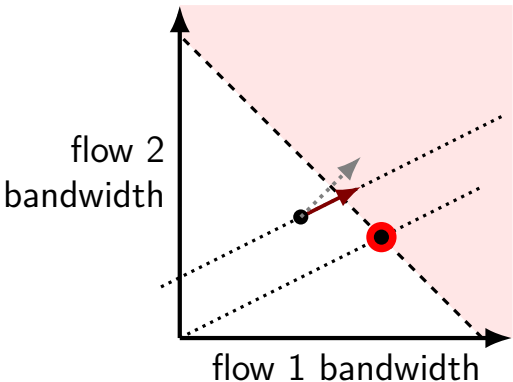
flow 1 increases faster
than flow 2 increases
not 45-degree angle anymore

revisiting additive increase

TCP: add +1 to window size each round trip time

flow 1: window 25+1 → 26 pkt/50 ms = 520 pkt/sec

flow 2: window 50+1 → 51 pkt/100 ms = 510 pkt/sec



in equilibrium
flow 1 gets more bandwidth

other unfairness

lower round-trip gets more bandwidth

can also get more bandwidth by...

using more connections ('independent' windows)

adding more than + 1 packet to window size/RTT

alternate congestion control

usually *more aggressive* than "normal" TCP

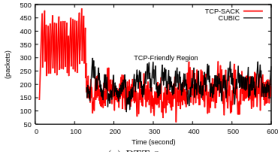
- examples we'll talk about later: CUBIC, Compound TCP
  (common defaults on recentish OSes)
- needed to fully utilize bandwidth on fast network

concern: are these "stealing" all the bandwidth from normal TCP?

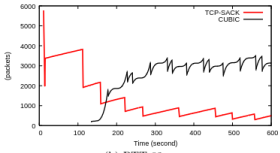
"TCP-friendly"

- should not make TCP used alongside them behave poorly

examples: TCP-unfriendliness



(a) RTT 8ms.



(b) RTT 82ms.

Figure 5: One CUBIC flow and one TCP-SACK flow. Bandwidth is set to 400Mbps.

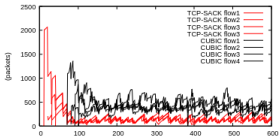


Figure 6: Four TCP-SACK flows and four CUBIC flows over 40ms RTT

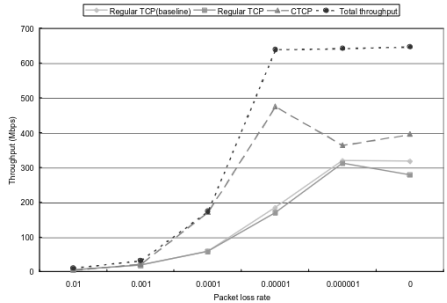


Figure 11. Throughput of CTCP and Regular TCP flows when competing for same bottleneck.

from Ha, et al, "CUBIC: A New TCP-Friendly High-Speed TCP Variant"
and Tan, et al, "A Compound TCP Approach for High-speed and Long Distance Networks"

a theoretical result

for RTT-unfairness, standard TCP with selective acknowledgments:[1]

$$\text{throughput} \approx \text{constant} \times \frac{\text{packet size}}{\text{RTT}\sqrt{\text{loss rate}}}$$

[1] Mathis et al, "The Macroscopeic Behavior of the TCP Congestion Avoidance Algorithm" (1997)

empirical results

some results from Philip (IMC'21)[2]

with same congestion control algorithm + RTT, Jain's fairness index > 0.99

CUBIC takes 70-80% of throughput when competing with equal number of traditional TCP flows

- recall: major change is cubic increase curve instead of additive (linear) increase

[2] Philip, Ware, Athapathu, Sherry, Sekar, "Revisting TCP Congestion Control Throughput Models & Fairness Properties At Scale" (IMC'21)

fixes from Jacobson's 1987 paper

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

“slow start”

not very well named

exponential(ish) increase to find window size

used on connection setup or some losses

history: original TCP set high window size

slow start is “slow” because it starts at a small size always

...but it grows *really quickly*

“slow start”

set window size = 1 packet

increase by one packet for each ACK

...until first packet loss

use window size before first window size

slow start effect

suppose we never leave slow start in connection between A and B and:

A sends 4 packets to B

after receiving 4 packets, B sends 8 packets to A

after receiving those packets A sends 1 packet to B

how many round-trip times does this take?

TCP Tahoe

congestion avoidance (cwnd $\geq$ ssthresh)

no loss: cwnd $+=$ 1 segment per round-trip-time
on loss: ssthresh $=$ cwnd$/2$; cwnd $=$ $\sim$2 segments

"slow start" (cwnd $<$ ssthresh)

no loss: cwnd $+=$ 1 segment per newly ack'd segment
on loss: ssthresh $=$ cwnd$/2$; cwnd $=$ cwnd$/2$;

recovery (known loss)

retransmit packet after third duplicate ACK of previous packet :or timeout

TCP (New)Reno [with slow start]

congestion avoidance (cwnd $\geq$ ssthresh; no known loss)

- no loss: cwnd $+=$ 1 segment per round-trip-time
- on loss (dup ACK): cwnd $=$ cwnd$/2$; ssthresh $=$ new cwnd
- on loss (timeout): cwnd $=$ 2 segments

recovery (known loss)

- resend immediately on third dup ACK
- send new data based on self-clocking idea ("fast recovery")

"*slow start*" (cwnd $<$ ssthresh)

- no loss: cwnd $+=$ 1 segment per newly ack'd segment
- on loss: ssthresh $=$ cwnd$/2$; cwnd $=$ cwnd$/2$;

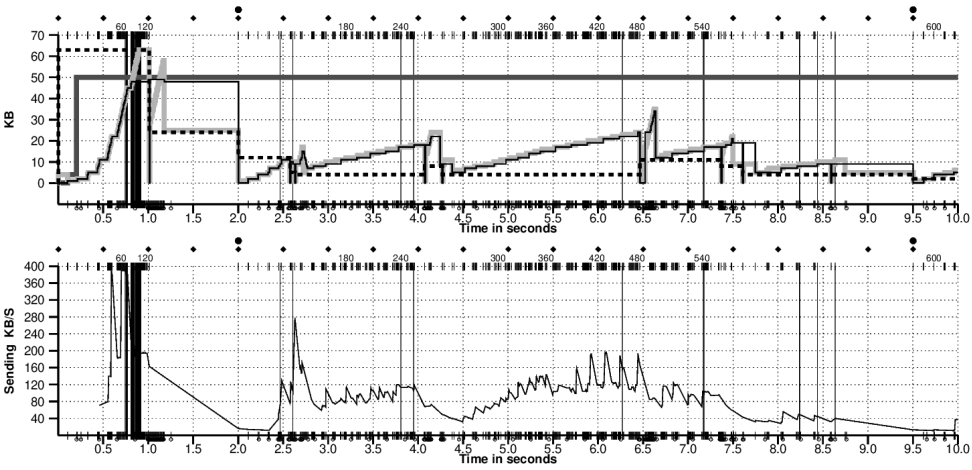


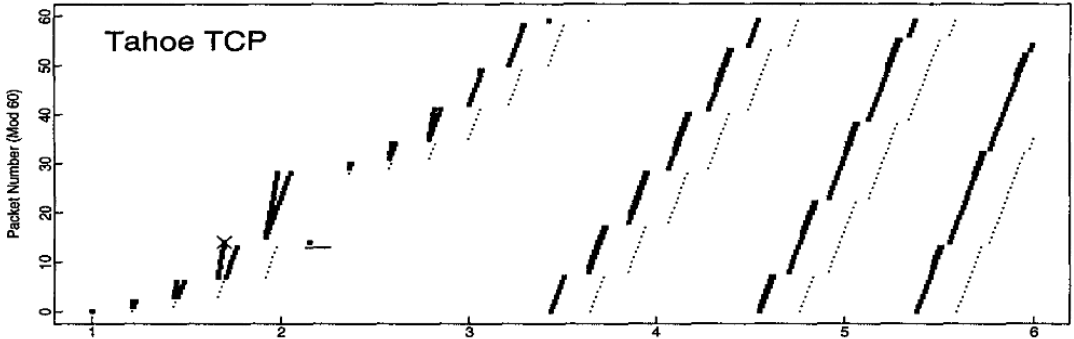
Figure 1: TCP Reno Trace Examples.

from Brakmo, O'Malley, and Peterson, "TCP Vegas: New techniques for congestion detection and avoidance"

top thick, light-grey line = congestion window; dotted = slow start threshold

TCP ‘Tahoe’ w/ one loss

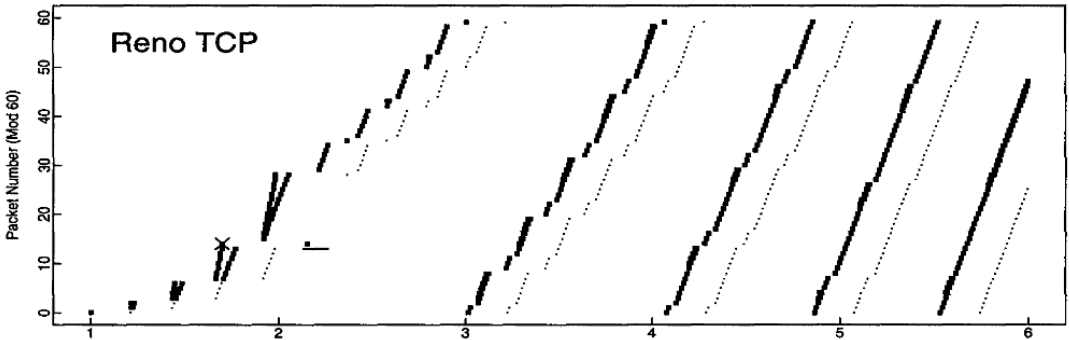
from Kevin Fall and Sally Floyd, “Simulation-based Comparisons of Tahoe, Reno, and SACK TCP”



TCP Tahoe = slow start, fast retransmit, no fast recovery

TCP ‘Reno’ w/ one loss

from Kevin Fall and Sally Floyd, “Simulation-based Comparisons of Tahoe, Reno, and SACK TCP”



TCP Reno = slow start, fast retransmit/recovery

fixes from Jacobson's 1987 paper

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

fast retransmit

if large window + data packet 2 is lost, then sender will see

ACK 0, *ACK 1, ACK 1, ACK 1, ACK 1, ACK 1*

duplicate ACKs indicate missing packet 2

shouldn't wait for timeout

fast retransmit

if large window + data packet 2 is lost, then sender will see

ACK 0, *ACK 1, ACK 1, ACK 1, ACK 1, ACK 1*

duplicate ACKs indicate missing packet 2

shouldn't wait for timeout

→ TCP heuristic: retransmit immediately after ∼3 duplicate ACKs

not 1 duplicate ACK to tolerate some reordering

also some other details (we'll talk later)

fast retransmission

TCP calls this idea of retransmission from duplicate ACKs "fast retransmission"

means we typically don't wait for timeouts

...assuming just one loss

fast retransmit delays

recv'd	sent
—	data 0-7
ACK 0	
	data 8
ACK 1	
	data 9
ACK 1	
ACK 1	
	data 2
ACK 1	
ACK 1	
ACK 1	
ACK 1	

selective ACKs

selective ACK: ACK 1, plus I got 3–5 and 7–8

can use as another duplicate ACK signal:

example:
resend 6 immediately after
3x “plus I got 3–5 and 7–...”

normal packets in flight

recv'd	sent	count of packets in flight
—	data 0-7	8
ACK 0		7
	data 8	8
ACK 1		7
	data 9	8
ACK 2		7
	data 10	8
ACK 3		7
	data 11	8
ACK 4		7
	data 12	8

window size $\sim$ packets in flight normally

duplicate ACK waste

window size $\sim$ packets in flight normally

but duplicate ACKs are exception (say window size 8)

recv'd	sent	count of packets in flight
—	data 0-7	8
ACK 0		7
	data 8	8
ACK 1		7
	data 9	8
ACK 1		7
ACK 1		6
	data 2	5
ACK 1		4
ACK 1		3
ACK 1		2

problem: need to wait until ACK of 2 to make progress

alternate explanation

sender stopped sending while receiving duplicate ACKs

but we know *most messages got there*

means our usage of network doesn't reflect our window size

want to transmit new packets in response to these ACKs

even though out of 'normal' window

TCP's fast recovery

estimate *packets in flight*

- with no losses, cwnd already does this!
- problem: needs adjustment there are losses

transmit new packets when packets in flight $<$ cwnd

- two implementation ideas:
- temporarily 'inflate' cwnd
- track effective send window separately

fast recovery example

recv'd	sent	est. packets in flight	send window size (range)	cwnd = target packets in flight
—	data 0-7	8	8 (0-7)	8
ACK 0		7	8 (0-7)	8
	data 8	8	8 (1-8)	8
ACK 1		7	8 (1-8)	8
	data 9	8	8 (2-9)	8
ACK 1 (+3)		7	8 (2-9)	8
ACK 1 (+3-4)		6	8 (2-9)	8
ACK 1 (+3-5)		5	8 (2-9)	4
	data 2	6	8 (2-9)	4
ACK 1 (+3-6)		5	8 (2-9)	4
ACK 1 (+3-7)		4	8 (2-9)	4
ACK 1 (+3-8)		3	8 (2-9)	4
	data 10	4	9 (2-10)	4
ACK 1 (+3-9)		3	9 (2-10)	4
	data 11	4	10 (2-11)	4

during recovery, "inflate" effective send window size

try to make actual packets in flight match new congestion window size

the reverse path

so far: assuming congestion on sender to receiver path

but we can also have congestion in other direction

- network becomes overloaded with ACKs

hopefully rare because ACKs are small, but...

but worth some special mitigations

delayed ACKs

RFC 1122 (Requirements for Internet Hosts — Communication Layers)

> "A host that is receiving a stream of TCP data segments can increase efficiency...by sending fewer than one ACK (acknowledgment) per data segment received; this is known as a "delayed ACK"..."

usually enabled these days

adds some latency, so Linux lets you disable on per-connection basis

diversion: some queuing theory

queuing theory: applied probability

talks about how queues work

applies to networks and anything else with “waiting in line”

queue measurements

arrival rate

service time (amount of time after waiting in line)

utilization = arrival rate / service time

if single thing can be processed at a time, then max utilization = 100%

- higher implies “infinitely” long queues

$M/M/1/\infty$ queue

next slides: results for $M/M/1/\infty$ queue

M (memoryless) — random arrival (exponential dist.)

M — random service time (exponential dist.)

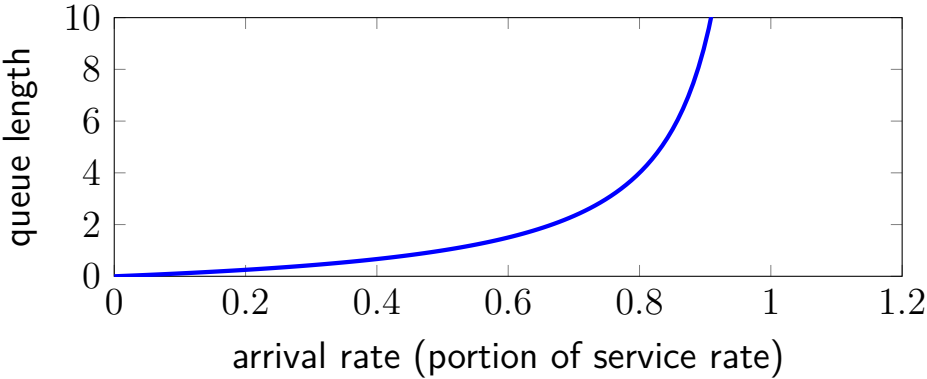
1 — one "server" (thing that can process packets)

∞ — unlimited queue length

$M/M/1/\infty$ queue length

mean queue length

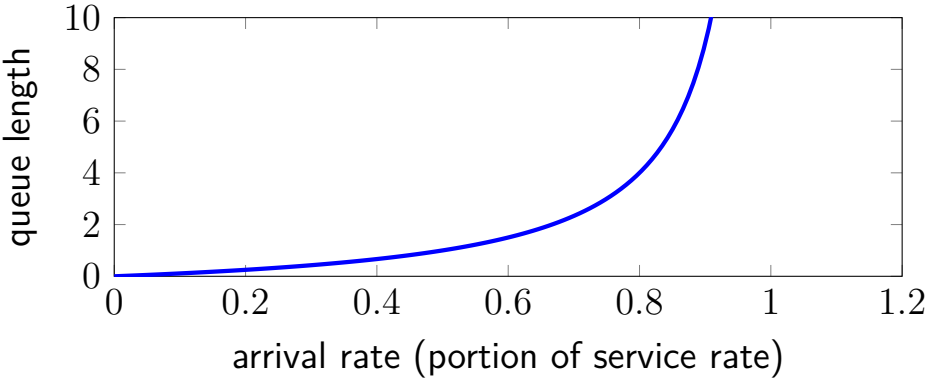
$$\frac{\text{arrival rate}}{\text{service rate} - \text{arrival rate}}$$



$M/M/1/\infty$ queue length

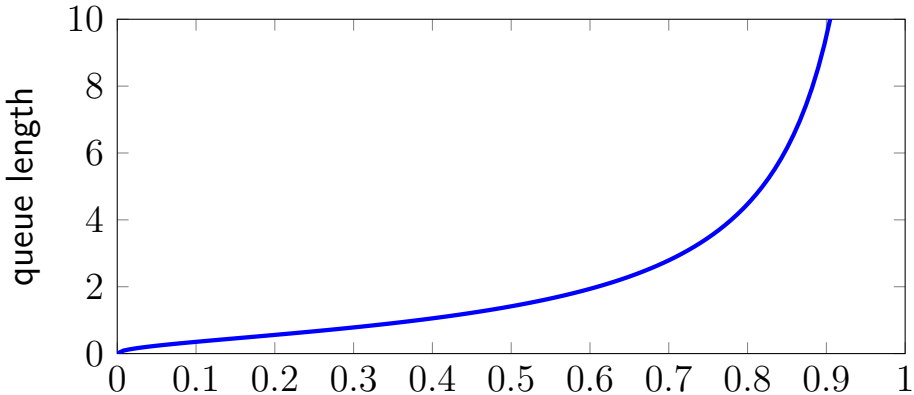
mean queue length

$$\frac{\text{arrival rate}}{\text{service rate} - \text{arrival rate}}$$

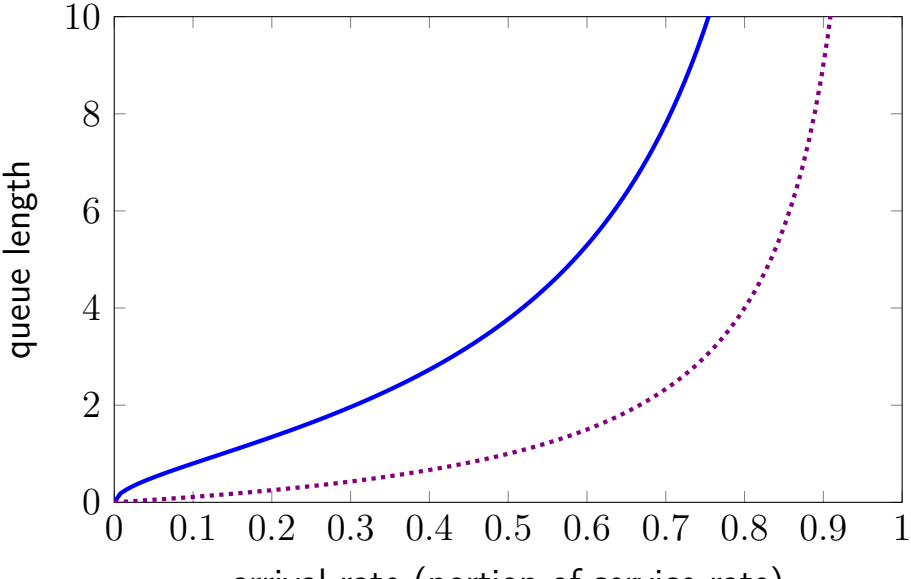


M/M/1/∞ queue length std. deviation

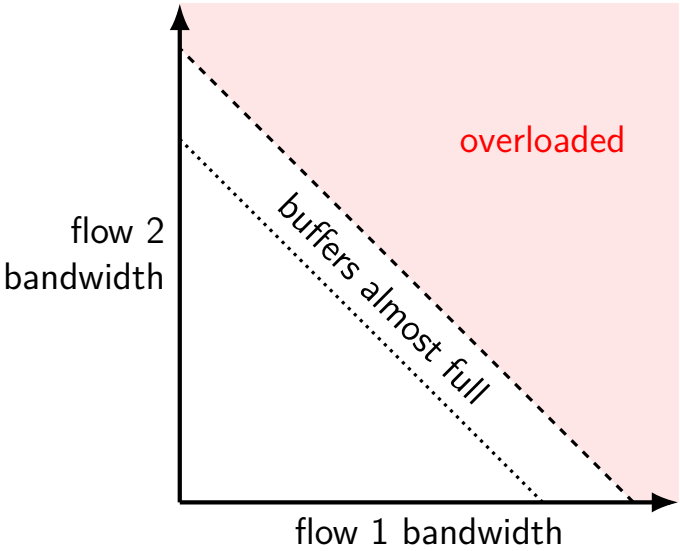
$$\sqrt{\frac{\text{utilization}}{(1-\text{utilization})^2}}$$



approx 95th pctile v mean queue length

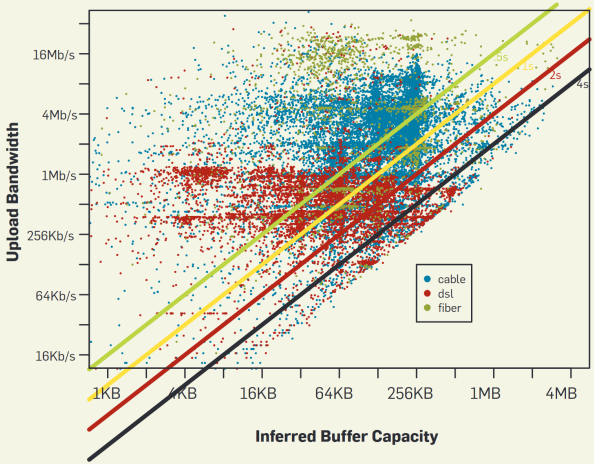


filling buffers



big buffers? (in 2011 or so)

Figure 5. Plot reproduced from ICSI's Netalyzr studies.



Jim Gettys and Kathleen Nichols,

"Bufferbloat: Dark Buffers in the Internet" (CACM, Jan 2012)

problems with big buffers

high latency — bad for some applications

slower response to congestion

- 1 second round trip time = 1 second to detect congestion
- more likely to have ‘congestion collapse’

avoiding big buffers

multiple fixes (that can be combined):

use smaller buffers?

- simpliest solution

detect congestion without full buffer...

- by choosing when/which packets to drop better?
- by using something other than drops?

avoiding big buffers

multiple fixes (that can be combined):

use smaller buffers?

- simpliest solution

detect congestion without full buffer...

- by choosing when/which packets to drop better?
- by using something other than drops?

avoiding big buffers

multiple fixes (that can be combined):

use smaller buffers?

- simpliest solution

detect congestion without full buffer...

- *by choosing when/which packets to drop better?*
- by using something other than drops?

avoiding big buffers

multiple fixes (that can be combined):

use smaller buffers?

- simpliest solution

detect congestion without full buffer...

- by choosing when/which packets to drop better?
- *by using something other than drops?*

fixes from Jacobson's 1987 paper

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

timeout setting

goal in setting timeouts:

timeout triggering almost always means dropped packet

to do this want *highest likely round trip time*

original TCP heuristic: twice RTT estimate

RTT variation exercise (1)

let's say 1 ms tranmission delay + 20 ms propogation delay

and queue depth ranges 'randomly' from 1 to 10

exercise: round-trip-time?

RTT variation exercise (1)

let's say 1 ms tranmission delay + 20 ms propogation delay

and queue depth ranges 'randomly' from 1 to 10

exercise: round-trip-time?

- 42 ms with no queue
- +1 ms per queue depth
- 43 to 52 ms

RTT variation exercise (2)

let's say 1 ms tranmission delay + 10 ms propogation delay

and queue depth ranges 'randomly' from 10 to 40

exercise: round-trip-time?

RTT variation exercise (2)

let's say 1 ms tranmission delay + 10 ms propogation delay

and queue depth ranges 'randomly' from 10 to 40

exercise: round-trip-time?

12 ms with no queue
+1 ms per queue depth
22 to 52 ms

how does original timeout do?

works well when queuing delay small relative to other delays

works poorly when queuing delay high

...because queuing delay won't be consistent!

new timeout formula

estimate mean deviation of RTT (= difference from average)

similar exponentially weighted moving average

timeout = RTT estimate + 2 × RTT deviation estimate

fixes from Jacobson's 1987 paper

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

normal backoff

problem: what if we have multiple timeouts

let's say timeout is 1 time unit

transmit at 1 time unit, 2 time units, 3 time units, 4 time units, etc.

problem: if the network is overloaded *from retransmissions* won't stop it

> ...but window size reduction should make number of packets retransmitted *per connection* low
> (so probably not so important with corrected window size management?)

exponential backoff

instead of:

transmit at 1 time unit, 2 time units, 3 time units, 4 time units, etc.

do something like:
transmit at 1 time unit, 3 time units, 7 time units, 15 time units, etc.

exponential backoff theory

for binary exponential backoff

- timeout for ith retransmission is $2^i \times$ base timeout

intuition: avoids overloading network by being a lot less aggressive

- what you 'should' do for repeated timeouts due to congestion

not aware of good theoretical results in TCP context

- famous result that this type of backoff is good for things like deciding when to retransmit on shared wired/wireless medium (Goodman et al, "On Stability of Ethernet")

“traditional” TCP variant names

everything we’ve talked about as standard = NewReno

Tahoe — slow start + redo slow start on any loss + fast retransmit

Reno — Tahoe + halve window size on dup ACKs

NewReno — Reno + fast recovery (send extra during fast retransmit)

SACK — NewReno + use selective acknowledgments

more recent TCP variants

BIC, CUBIC — loss-based schemes that vary increase/decrease algorithm

Vegas, BBR, FAST, Compound, Westwood — schemes that use latency/bandwidth to detect congestion

(later topic)

(and there are many, many more)

CUBIC: default congestion control today

default in Linux (since 2006), OS X (since 2014), Windows (since 2019)

- sysadmin has other options they can configure
- can be changed on connection-by-connection basis

includes improvements to recovery we'll talk about later

but mostn notably has non-additive increase

increase intuition

'standard' TCP basically has two increase algorithms:

slow start — quickly explore window sizes

congestion avoidance — slowly probe for maximum

abrupt transition between these is suspect

- should have gradual exploration $\rightarrow$ fine-tuning switch

heuristic for when to use slow start-style is suspect

- half way from last loss $\approx$ guess?

CUBIC increase algorithm

big idea: faster increase when further away from window size of last loss

- cubic function with saddle point at that window size
- function parameterized by time since loss

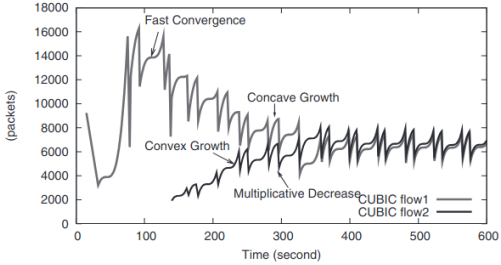
intuition for 'steady state':

- guess correct window size is close to last loss
- quickly get to around that window size
- search slowly/precisely around that size to fine tune size

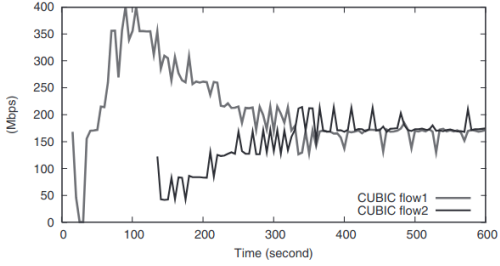
intuition for non-steady state:

- try to quickly probe for roughly how small/big it is
- worry about getting more precise size on later 'pass'

from Ha, Rhee, and Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant"



(a) CUBIC window curves.



(b) Throughput of two CUBIC flows.

Figure 4: Two CUBIC flows with 246ms RTT.

other CUBIC changes

different multiplicative decrease factor (divide by 1.4)

other congestion signals

so far: detecting congestion via drops

- need data to go missing
- transmitting redundant data
- filling up buffers causing high latency

some alternate ideas:

have switches/routers ‘mark’ packets

latency from longer queues

other congestion signals

so far: detecting congestion via drops

- need data to go missing
- transmitting redundant data
- filling up buffers causing high latency

some alternate ideas:

have switches/routers 'mark' packets

latency from longer queues

other congestion signals

so far: detecting congestion via drops

- need data to go missing
- transmitting redundant data
- filling up buffers causing high latency

some alternate ideas:

have switches/routers ‘mark’ packets

latency from longer queues

other congestion signals

so far: detecting congestion via drops

- need data to go missing
- transmitting redundant data
- filling up buffers causing high latency

some alternate ideas:

have switches/routers 'mark' packets

latency from longer queues

ECN

explicit congestion notification

when buffer ‘close’ to full

switches set ‘congestion experienced’ (CE) signal in some packets

goal: congestion signal *instead of* packet drops

avoid all the retransmission, hopefully

still have fallback to dropping packets

ECN and TCP/IP

congestion experience (CE) signal in IP heaer

when ACKing, "return" CE signal with ACK

ECN echo (ECE) TCP flag on ACK packets

when sender sees ECE flag, confirm reciept by setting "congestion window reduced" (CWR) flag until ECE flag stops being set

ECN opt-in

two bits in IP header

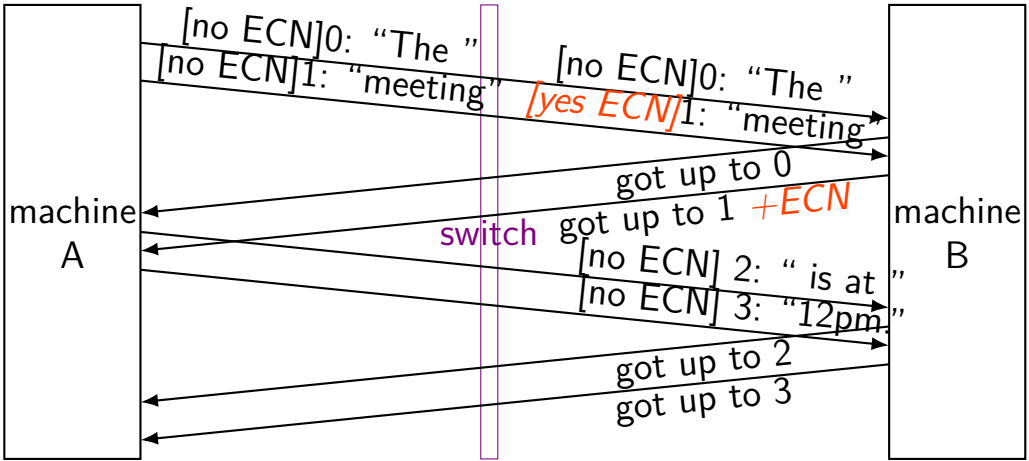
RFC 3168 says:

`00` = not-ECN capable (default)

`01`, `10` = ECN-capable (set by TCP/etc. implementation)

`11` = congestion experienced

ECN timeline



data sent has place for ECN bit to be placed

switch modifies ECN bit *if buffer close to full*

reacting to ECN marks

multiple options for using ECN marks

simplest idea:

adjust window as if packet was dropped

...but don't need to resend data

ECN deployment

ECN proposed in 2001

13 years later: around 56% support on websites[3]

16 years later:

- around 80% support on websites[4]
- around 0.2% of servers disallow connection when ECN requested

20 years later:

- around 86% support on websites[5]
- around 4% of paths strip ECN signals, including notable ISPs/cloud providers/etc.
- around 7.5% of connections (from sampled Universities) enable ECN

[3] Trammel et al, "Enabling Internet-Wide Deployment of Explicit Congestion Notification"

[4] Kühlewind et al, "Tracing Internet Path Transparency"

[5] Lim et al, "A Fresh Look at ECN Traversal in the Wild"

example: DCTCP

DataCenter TCP (2010)

intended for datacenters

- high bandwidth, low latency networks

based on explicit congestion notification

...but uses different multiplicative decrease strategy

measure portion of packets marked recently α

decrease by factor of $1 - \alpha/2$

- respond gradually to congestion
- start responding early (packets marked when queues far from full)

other congestion signals

so far: detecting congestion via drops

- need data to go missing
- transmitting redundant data
- filling up buffers causing high latency

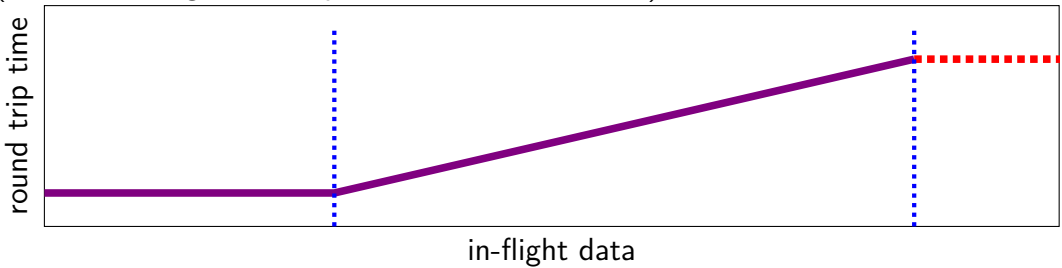
some alternate ideas:

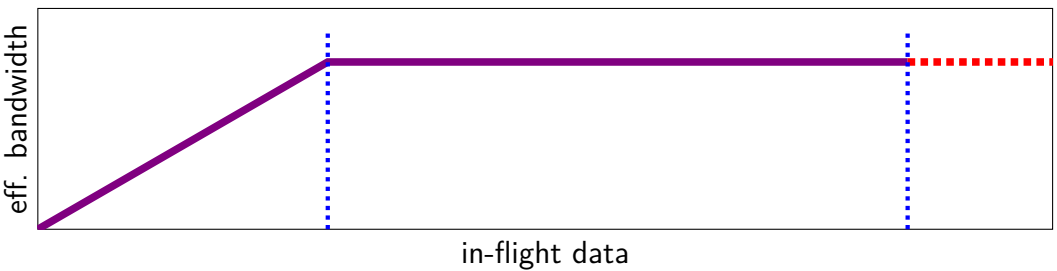
have switches/routers ‘mark’ packets

latency from longer queues

some intuition (based on BBR)

(based on Google's BBR presentation to the IETF)





very different congestion control

fuller queues $\rightarrow$ higher latency

fuller queues $\rightarrow$ throughput same as window increases

very different congestion control

fuller queues $\rightarrow$ higher latency

fuller queues $\rightarrow$ throughput same as window increases

strategy: monitor throughput/latency to detect full queues

goal: fill link without making queue grow (much) in size

‘Vegas’-style congestion control (1)

record “base” round-trip time

- connection start or lowest observed

“ideal” throughput should be one window / base round trip time

- (Vegas paper calls this “expected” throughput)
- what would happen with no queuing delay

“actual” throughput $\approx$ one window / actual round trip time

‘Vegas’-style congestion control (2)

measured ideal+actual throughput (prev. slide)

mainly using idea of ‘base’ round trip time

goal: control what “ideal” - actual throughput is

if 0, queues are probably empty, can increase window

if large, queues are too big, decrease window

Compound TCP

combines Vegas and ‘normal’ TCP congestion control

track seperate ‘delay’ and ‘congestion’ window

- congestion window uses standard TCP algorithm
- delay window based on Vegas-like increase in RTT detection

effective window size based delay+congestion window

was default on Windows for many years

Westwood TCP

Vegas = watch latency versus window size

intuition: if window size $\uparrow \implies$ latency $\uparrow$, window size too high

alternate idea: bandwidth versus window size

intuition: if window size $\uparrow \implies$ bandwidth unchanged, window size too high

Westwood TCP:

estimate actual bandwidth based on ACK rate

set ssthresh based on bandwidth + min RTT

BBR-style congestion control

if queues are empty, larger window:

latency stays the same and throughput increases

if queues are filling, larger window:

throughput stays the same and latency increases

BBR-style congestion control

if queues are empty, larger window:

latency stays the same and throughput increases

if queues are filling, larger window:

throughput stays the same and latency increases

BBR-style congestion control

if queues are empty, larger window:

latency stays the same and throughput increases

if queues are filling, larger window:

throughput stays the same and latency increases

BBR-style congestion control

if queues are empty, larger window:

latency stays the same and throughput increases

if queues are filling, larger window:

throughput stays the same and latency increases

observe effect of sending more/fewer packets periodically

estimate 'boundary' based on observed latency/throughput

keep window size near boundary most of the time

BBR

congestion control algoirithm out of Google published c. 2016

apparently deployed (at least at some point) on their servers

not great fairness results with traditional TCP

Philip (IMC'21)[6] claims one BBR flow takes 40% of throughput when competing with thousands of CUBIC or NewReno flows

[6] Philip, Ware, Athapathu, Sherry, Sekar, "Revisting TCP Congestion Control Throughput Models & Fairness Properties AT Scale" (IMC'21)

2019ish measurement data

Table 5. Distribution of variants.

Variant	Websites	Proportion
CUBIC [15]	6,139	30.70%
BBR [4]	3,550	17.75%
BBR G1.1	167	0.84%
YeAH [2]	1,162	5.81%
CTCP [34]/Illinois[22]	1,148	5.74%
Vegas [3]/Veno [13]	564	2.82%
HTCP [21]	560	2.80%
BIC [37]	181	0.90%
New Reno [28]/HSTCP [12]	160	0.80%
Scalable [20]	39	0.20%
Westwood [7]	0	0.00%
Unknown	3,535	17.67%
Short flows	1,493	7.46%
Unresponsive websites	1,302	6.51%
Total	20,000	100%

Mishra et al, "The Great Internet TCP Congestion Control Census"

2023ish measurement data

Table 3: Classification results for websites by CDN websites. The values after the slashes are after a clustering step on traces within each CDN.

CDN	BBR	BIC	CDG	CUBIC	Highspeed	HTCP	NV	Reno	Vegas	Westwood	Yeah	Unknown	All Invalid	RTT > 85ms	Unresponsive	Total
Akamai	470/491	0	3/4	4	0	0	0	0	0	0	0	115/91	189	36	233	1050
Cloudflare	1233/1595	0	6/7	5/6	0	0	0	1	0	0	0	824/460	394	55	989	3507
Cloudfront	530/545	0	9/10	7/10	0	0	3/2	0	0	0	0	74/56	78	10	121	832
Fastly	21/25	1/13	3	25/130	0	0	0/1	0	0	0	0	174/52	26	3	30	283
Google	29	0	1	2	0	0	2	0	0	0	1	230	37	18	66	386
Other CDN	28/32	2/0	1	53/92	0	0	1	0	0	0	0	72/31	54	41	226	478
No CDN	116/122	3/0	8/9	89/116	3/5	2	5	4/3	1/0	3	0	146/115	127	1205	1752	3464
Total	2427/2839	6/13	31/35	185/360	3/5	2	11	5/4	1/0	3	1	1635/1035	905	1368	3417	10000

obtaining these measurements

Ware et al (2024) technique: estimate queue occupancy

- add artificial slow link on measurement path
- measure number of packets queued over time
- shape of graph indicates which congestion control algorithm

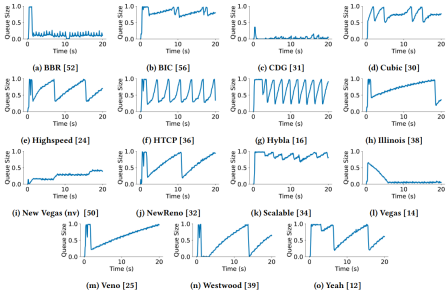


Figure 23: Example CCAnalyzer CCA training sample traces from AWS-Virginia (5bw-85rtt-64q setting)

backup slides

“slow start” later

heuristic: slow start if window size is ‘small’ compared to last packet loss

compromise: find correct size v. overload network

method:

set `ssthresh` = bytes in flight / 2 on packet loss

use slow start when window size < `ssthresh`