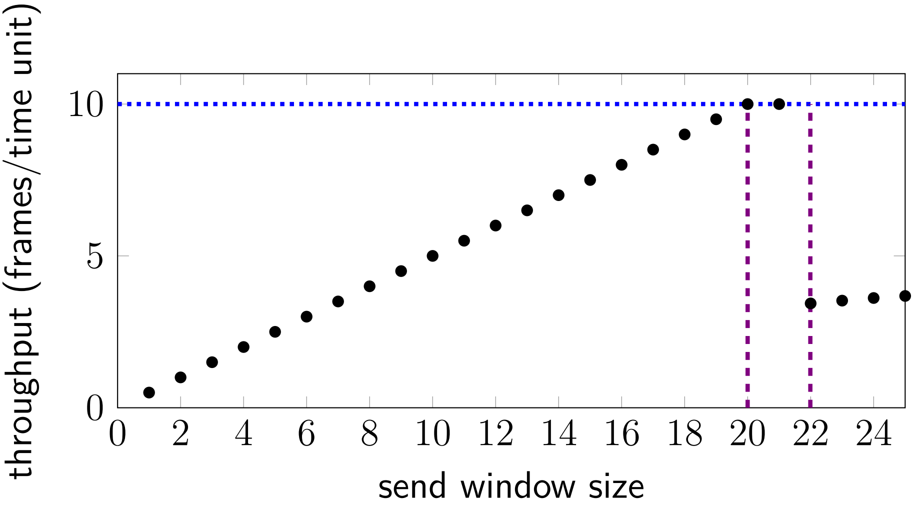
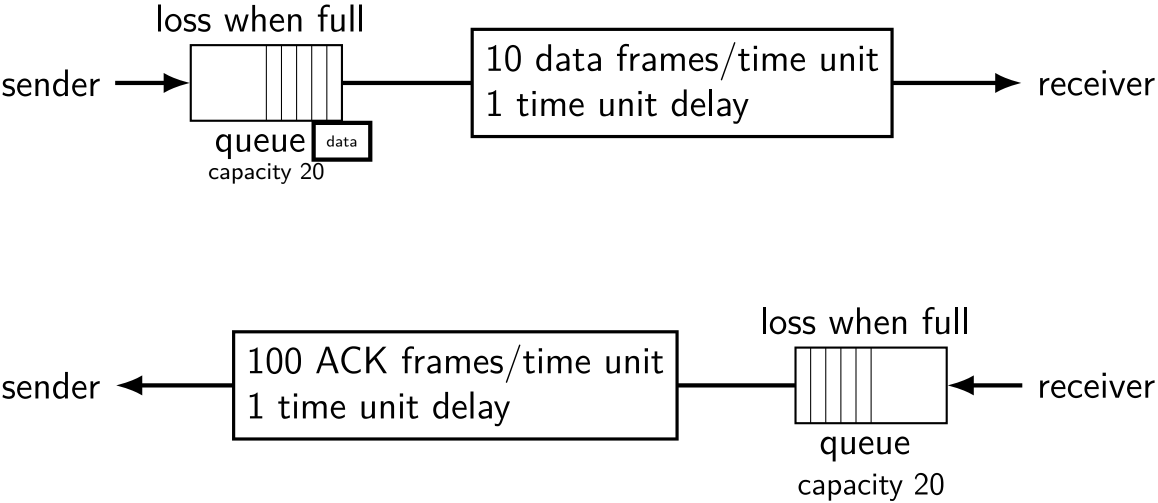
# Changelpg

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

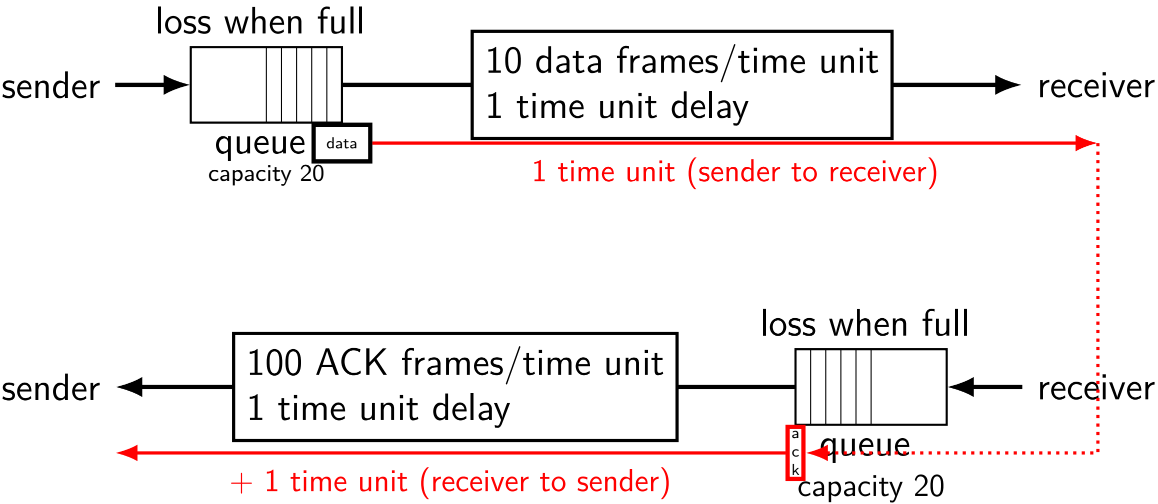
# throughput and window size



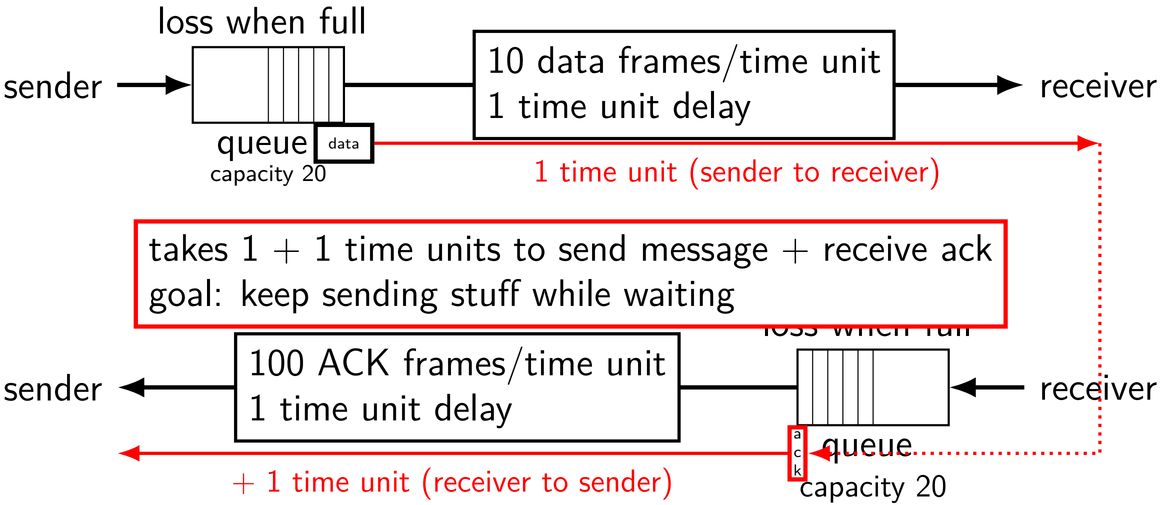
# packet transit time



# 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

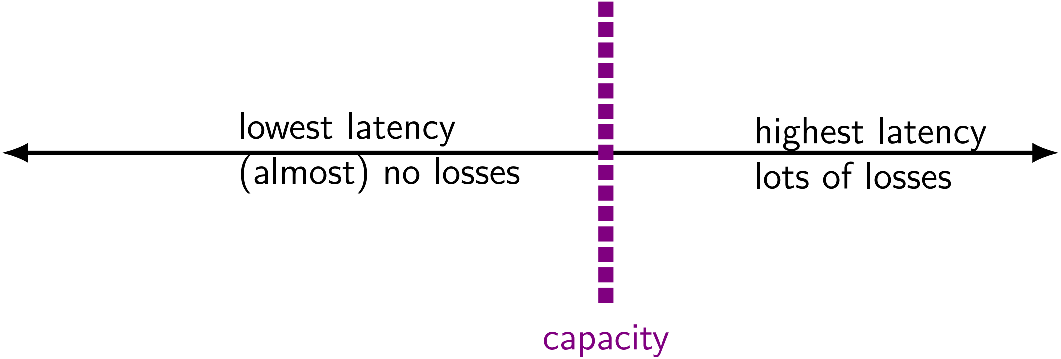
# window size tweaking

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

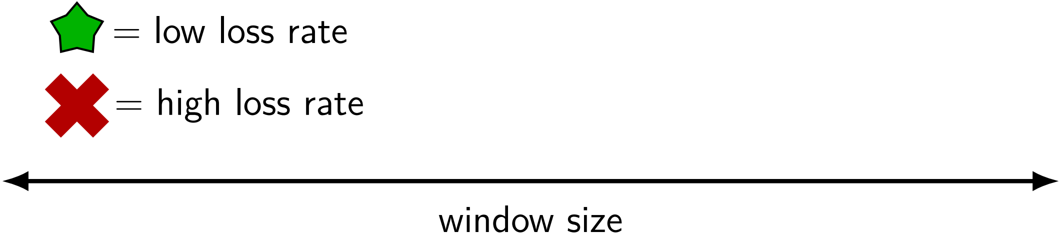
latency/loss rate increases when window size too big

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

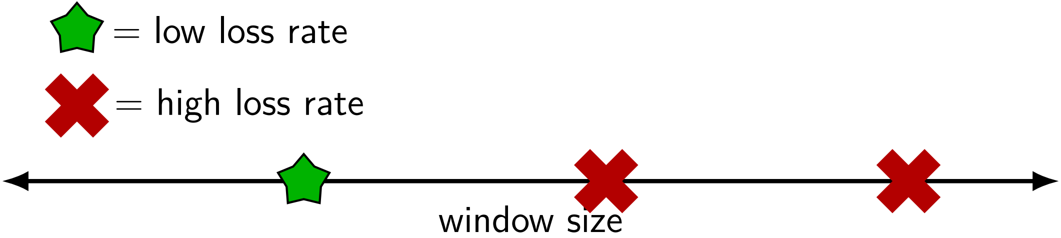


# try a bunch of things

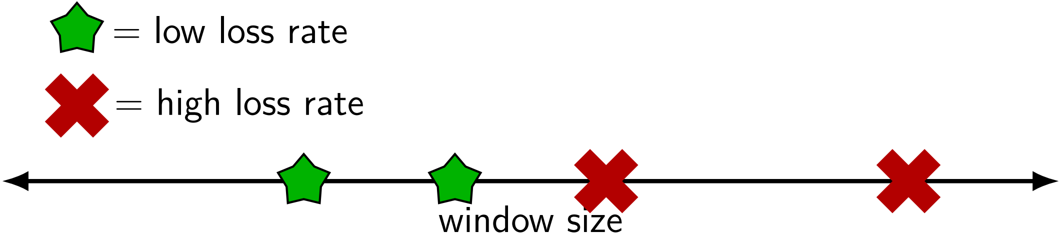
= low loss rate

= high loss rate

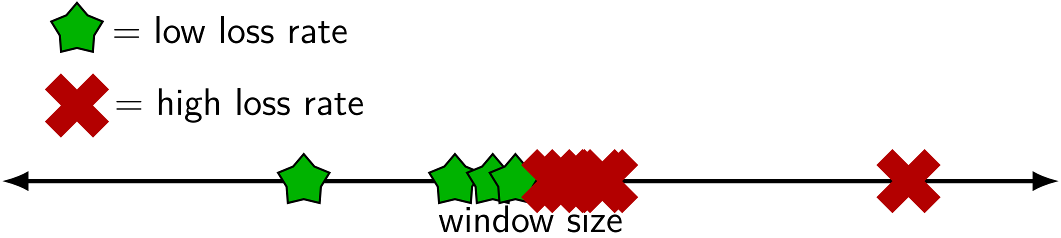
# try a bunch of things



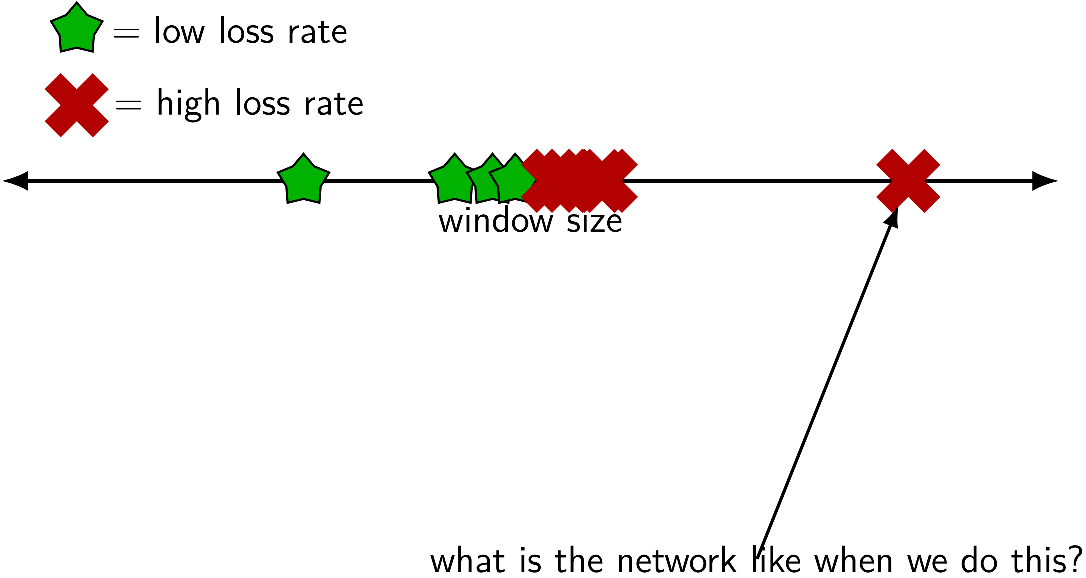
# try a bunch of things



# 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

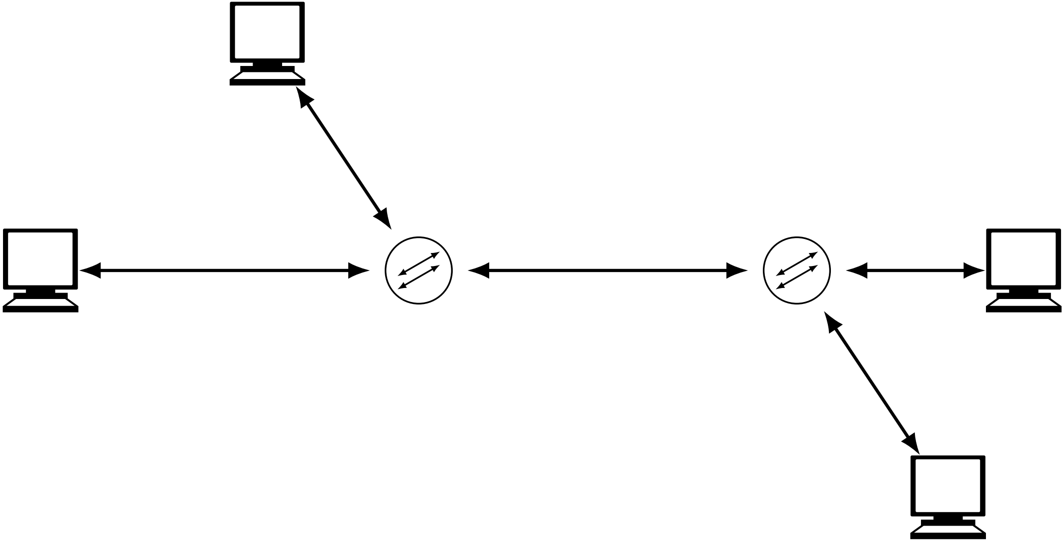
(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

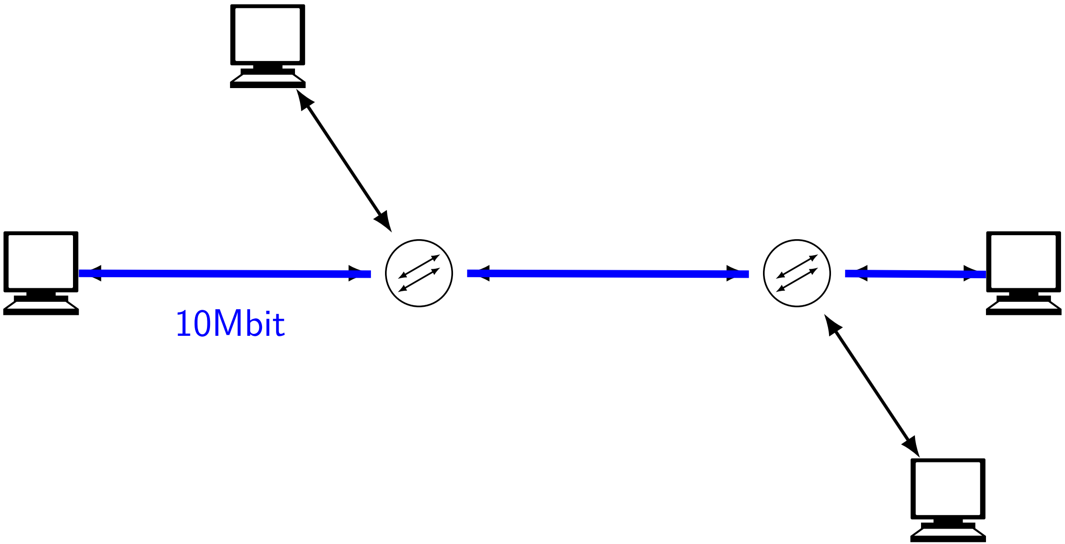
(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

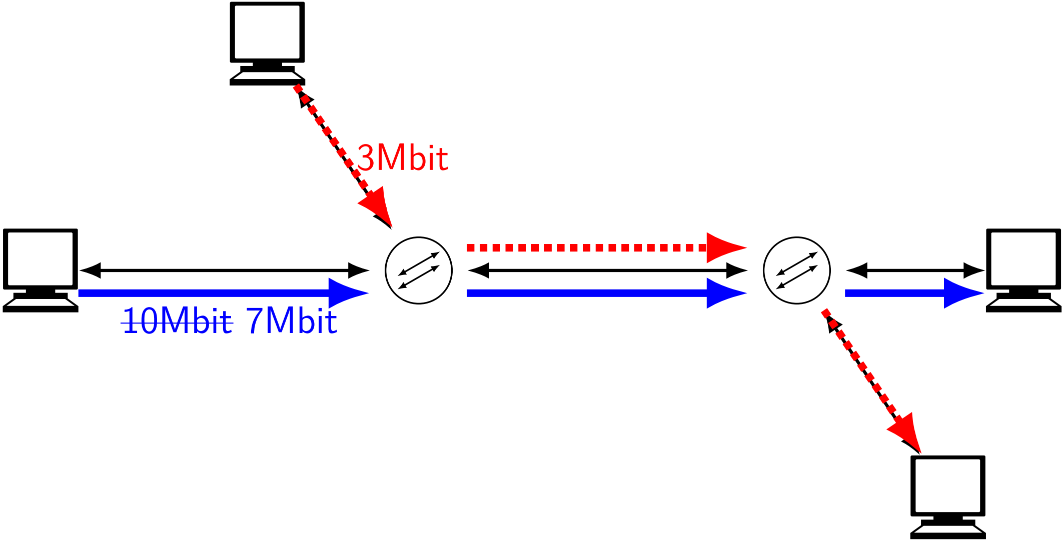
# changing cross-traffic



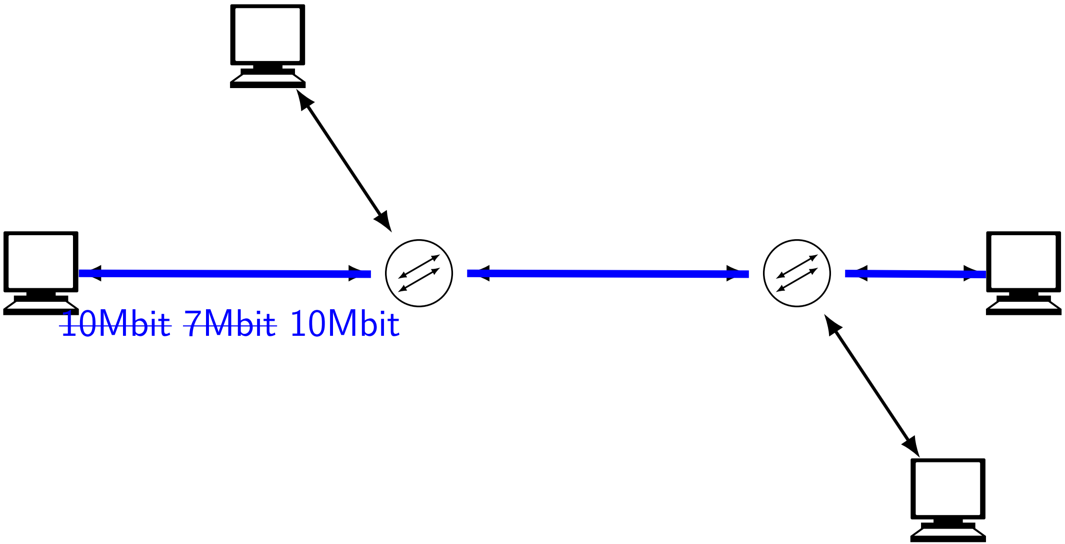
# changing cross-traffic



# 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?

# 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

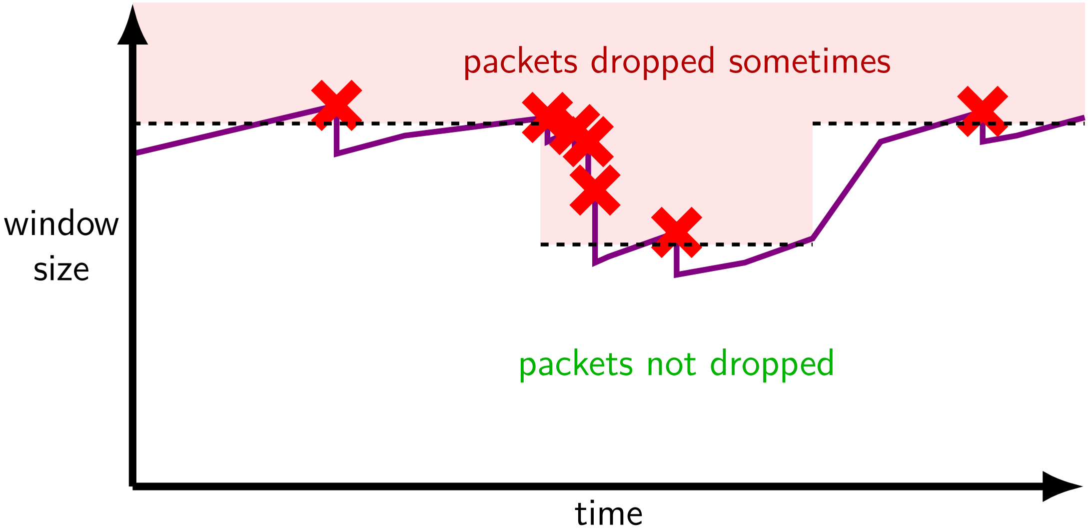
# handling steady state

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

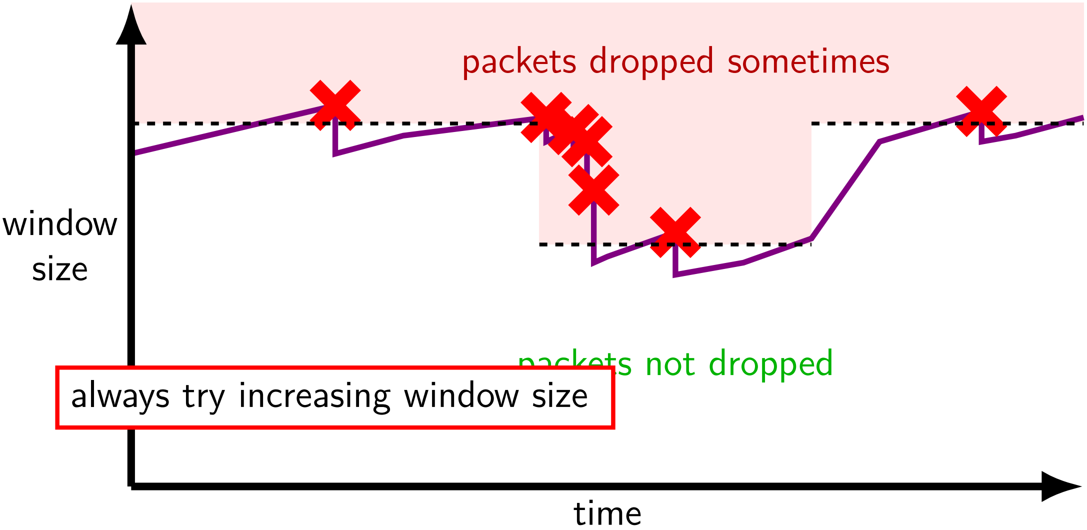
want to focus on how we react to changes

still going to use “experimentation” idea

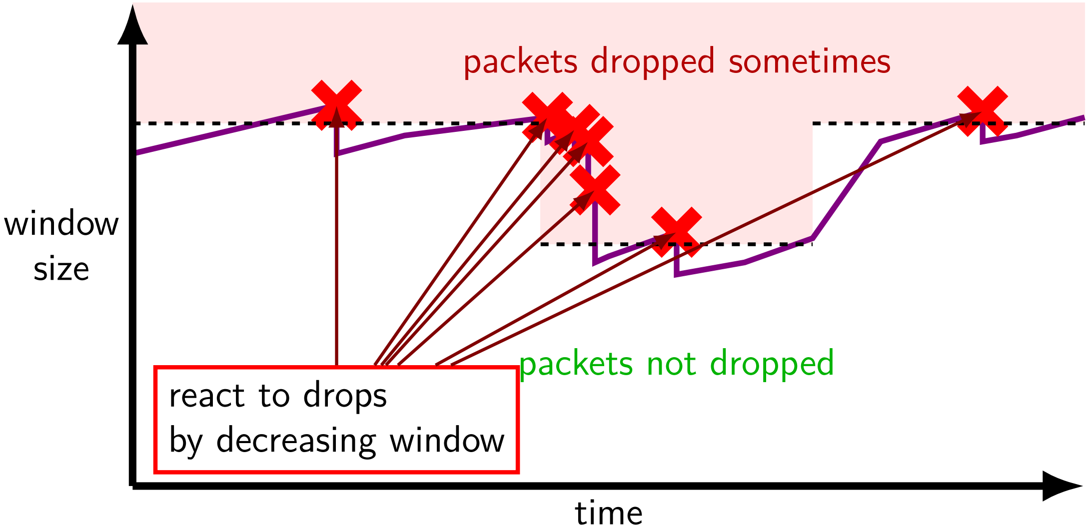
# window size experimenting



# 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?

# 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)
20 packets/second from other session

expected *loss rate* (% packets lost)? $\frac{100+20-50}{100+20} = 58\%$

expected % test flow packets lost? $58\%$

expected % other session packets lost? $58\%$

# the overloaded switch

let's say switch can handle 50 packets/second

but has:

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

expected *loss rate* (% packets lost)? $\dfrac{100 + 20 - 50}{100 + 20} = 58\%$

expected % test flow packets lost? $58\%$

expected % other session packets lost? $58\%$

...but I missed something

# a virtuous cycle

what is other session going to when 58% 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 $\rightarrow 10 + 58\% \cdot 10 + 58\%^2 \cdot 10 \ldots \approx 48$ 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)
20 packets/second from other session $\rightarrow 48$ with resends

expected *loss rate* (% packets lost)? $\dfrac{100+48-50}{100+48} = 66\%$

expected % test flow packets lost? $66\%$

expected % other session packets lost? $66\%$

# the overloaded switch (revised)

let's say switch can handle 50 packets/second

but has:

- 100 packets/second from test flow (checking window size)
- 20 packets/second from other session $\rightarrow 48$ with resends

expected *loss rate* (% packets lost)? $\dfrac{100+48-50}{100+48} = 66\%$

expected % test flow packets lost? $66\%$

expected % other session packets lost? $66\%$

- means that 48 pkt/sec is slight underestimate
- though realistically other session should slow down

# aside: latency (1)

58% packet loss $\rightarrow$ average packet sent 2.4 times

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

58% 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

# performance v load

# 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

$\approx 0.9$Mbyte/sec with 1500 byte packets

optimal rate is 10000 packets/second

$\approx 15$Mbyte/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

$\approx 0.9$Mbyte/sec with 1500 byte packets

optimal rate is 10000 packets/second

$\approx 15$Mbyte/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

# fast decrease

want to decrease quickly to get out of overload

original TCP heuristic: divide by two (minimum 1 packet)

# fast decrease

want to decrease quickly to get out of overload

original TCP heuristic: divide by two (minimum 1 packet)

exactly by two probably not important

important theoretically: approx. multiplicative decrease

- will help show okay behavior with multiple flows

# AIMD

additive increase + multiplicative decrease

basic of steady-state behavior

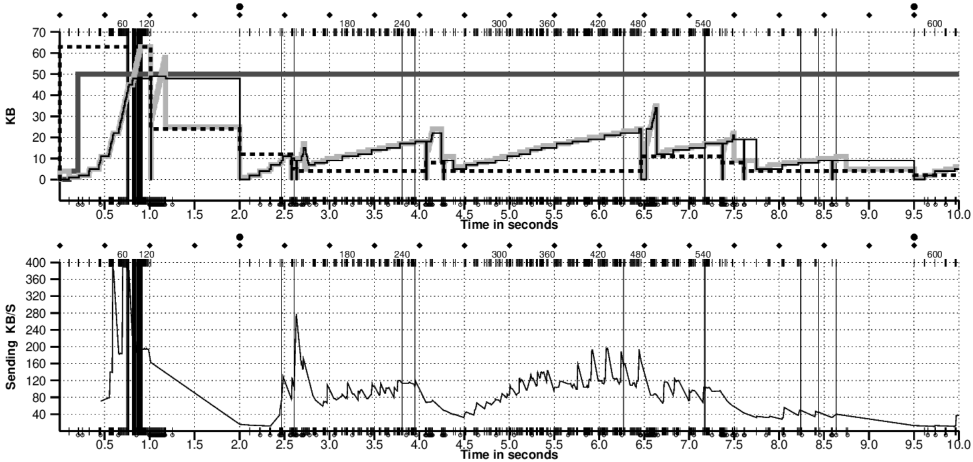



Figure 1: TCP Reno Trace Examples.

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

# 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

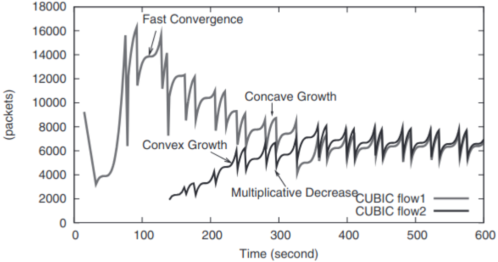
big idea: faster increase when further away from window size of last loss

- cubic function with saddle at that window size

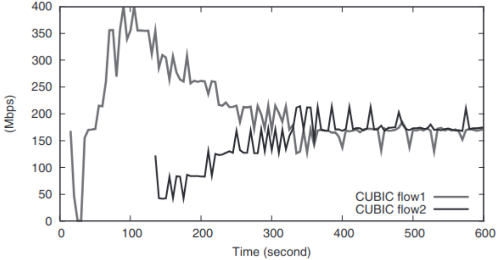
intuition:

- search faster if away from “steady state”
- avoid excess losses from ‘probing’ if at “steady state”

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.**

# 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)
- $\rightarrow$ 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

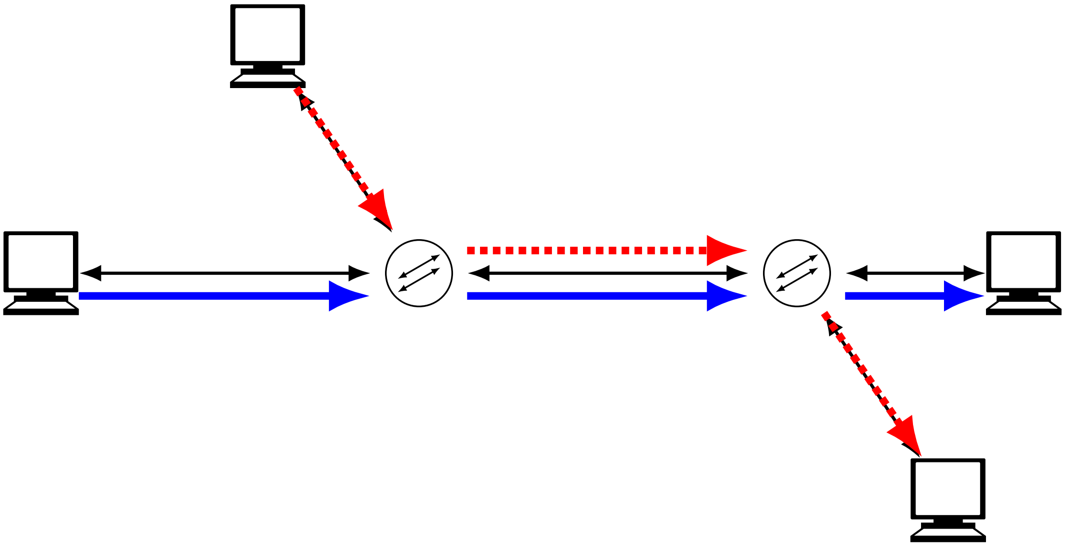
# congestion: sharing

want to consider multiple flows

key questions:

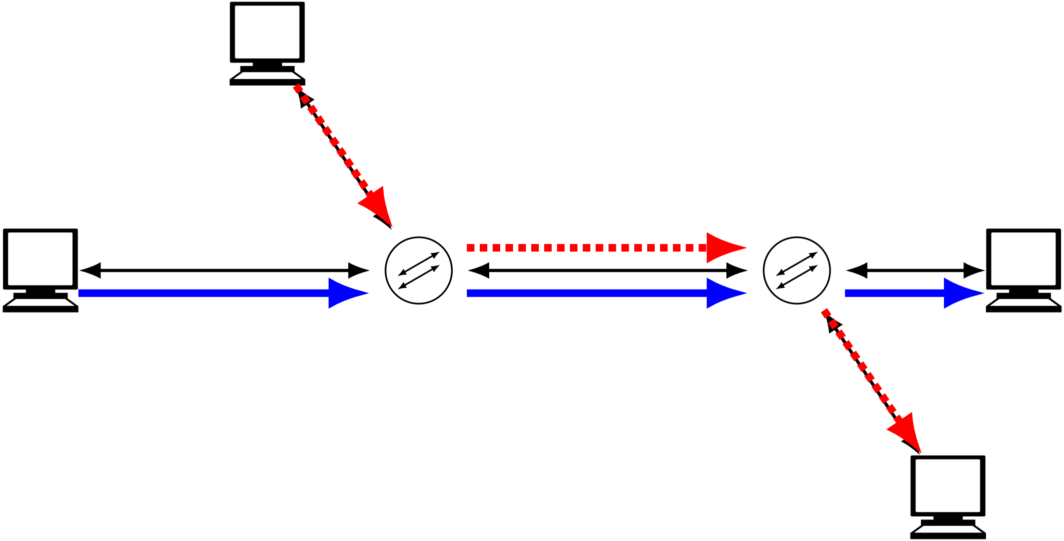
- is it stable if both flows changing window sizes?
- is there one winner/loser?
- is the winner/loser who we want it to be?

# exericse: what should happen?



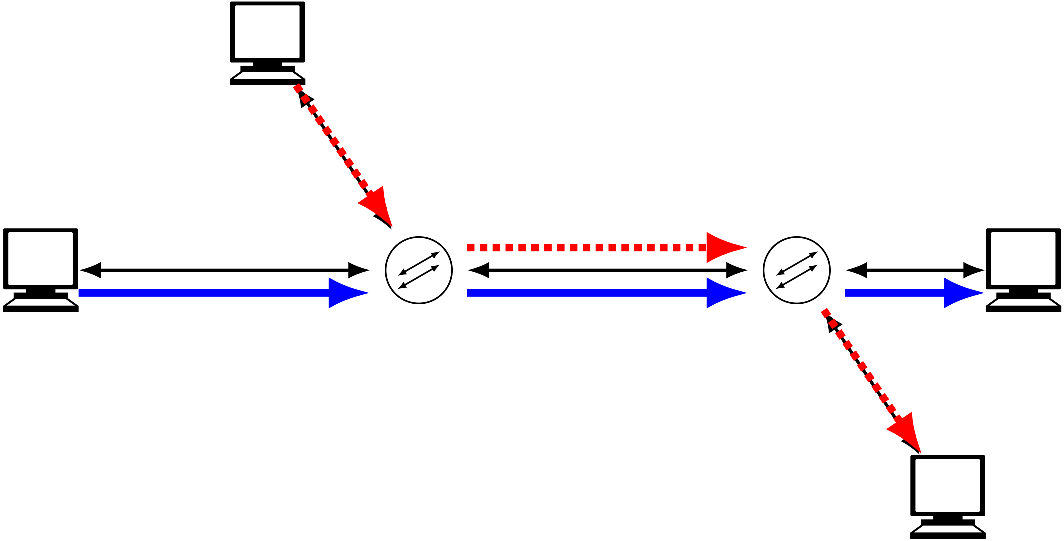
two connections on shared link

# exericse: what should happen?



two connections on shared link

# exericse: what should happen?



two connections on shared link

# showing AIMD

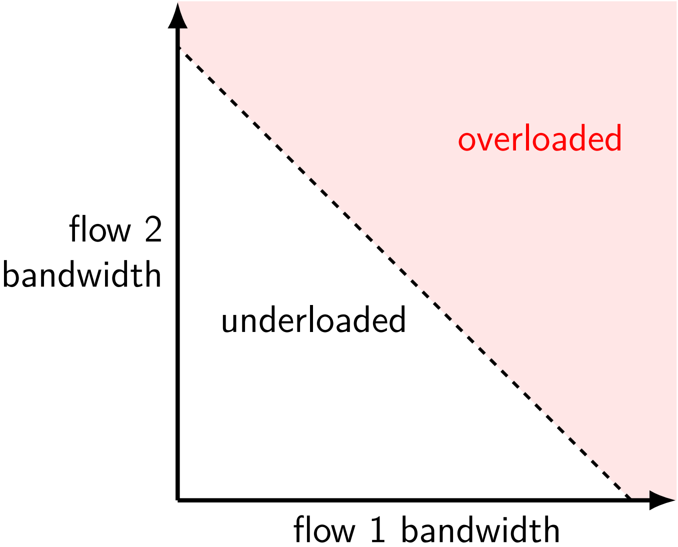
slides based on Chiu and Jain, “Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks”’

1989 paper

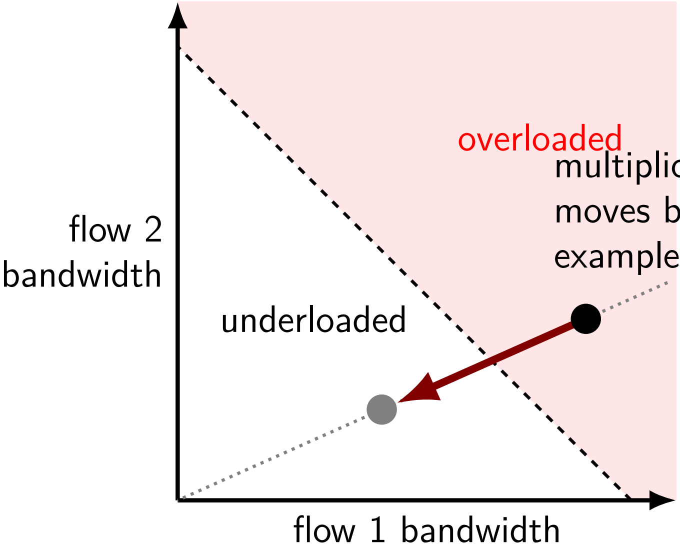
you might notice 1989 is well after TCP was in use

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

# picturing sharing

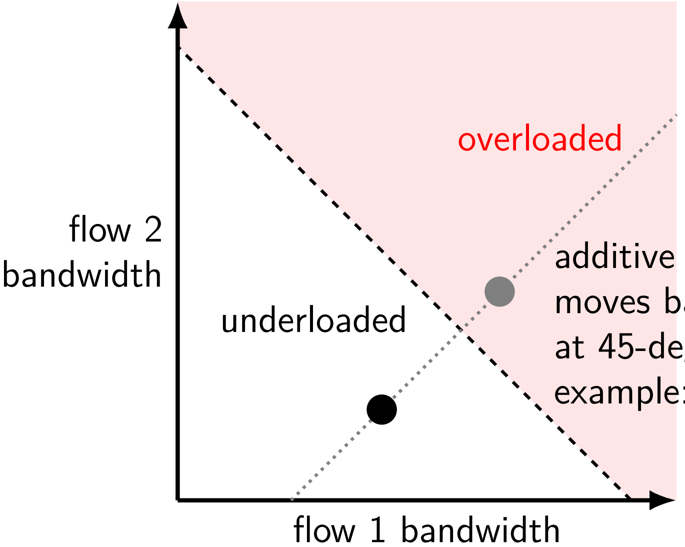


# picturing sharing



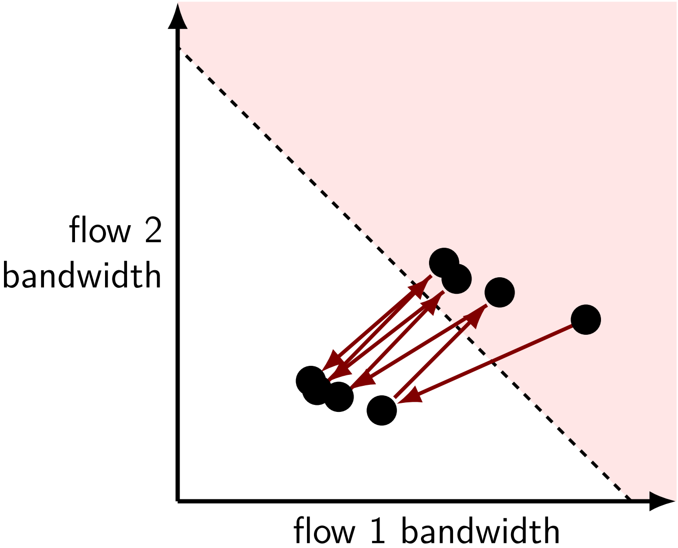

multiplicative decrease
moves bandwidths along line to origin
example: (90%, 40%) to (45%, 20%)

# picturing sharing

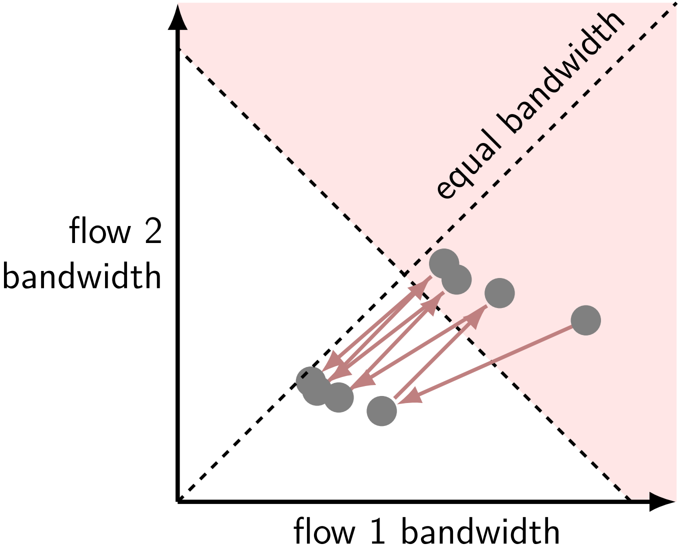



additive decrease
moves bandwidths
at 45-degree angle
example: (45%, 20%) to (55%, 30%)

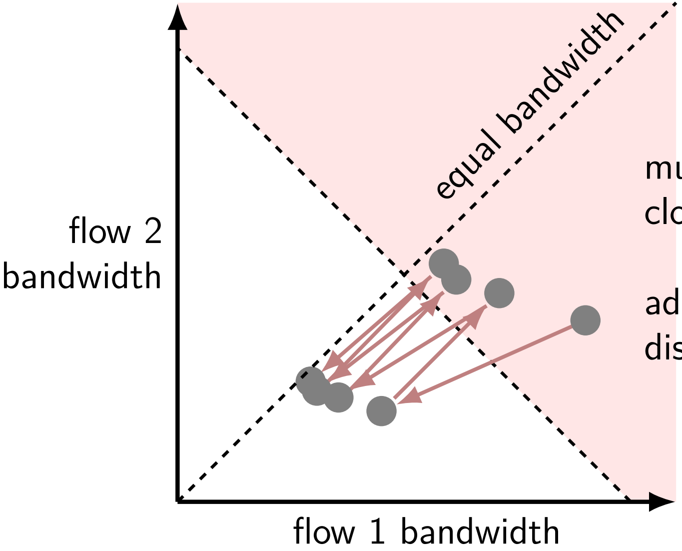
# picturing sharing



# picturing sharing



# picturing sharing




multiplicative decrease brings closer to equal bandwidth line

additive increase keeps same distance from line

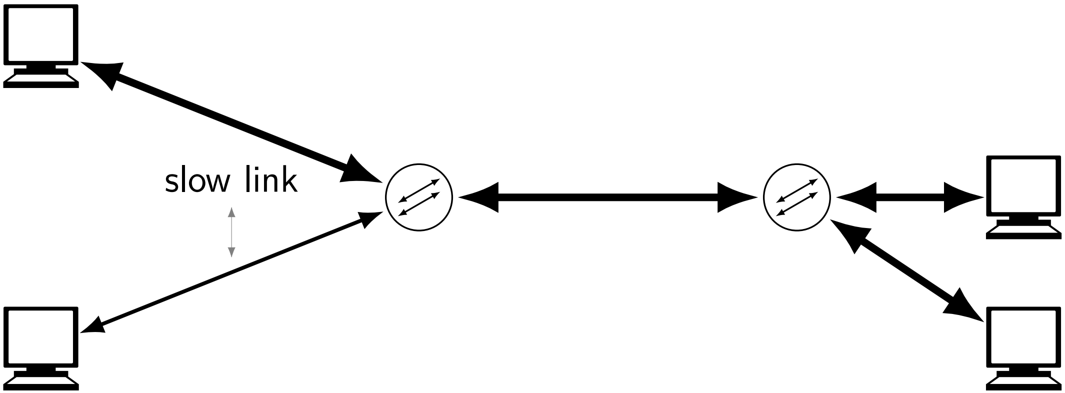
# some assumptions we made

...that may not always be true

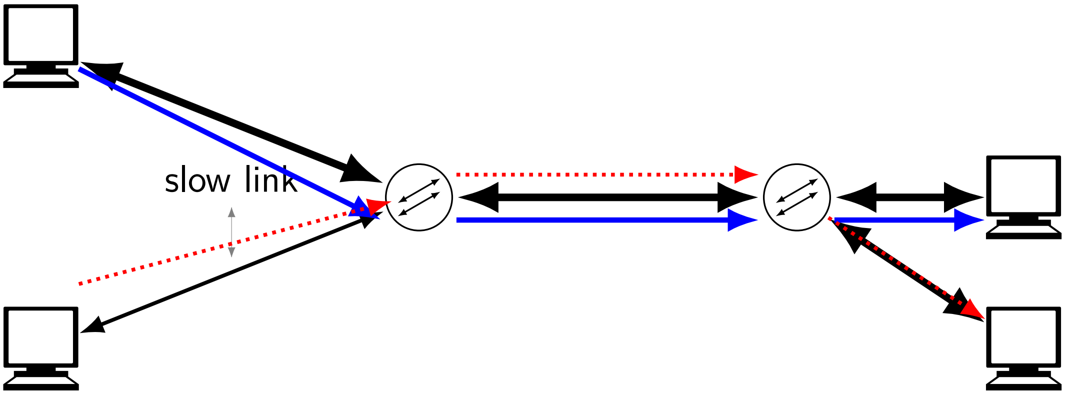
both flows experience drops when network overloaded

same additive increase factor (for 45 degree angle)

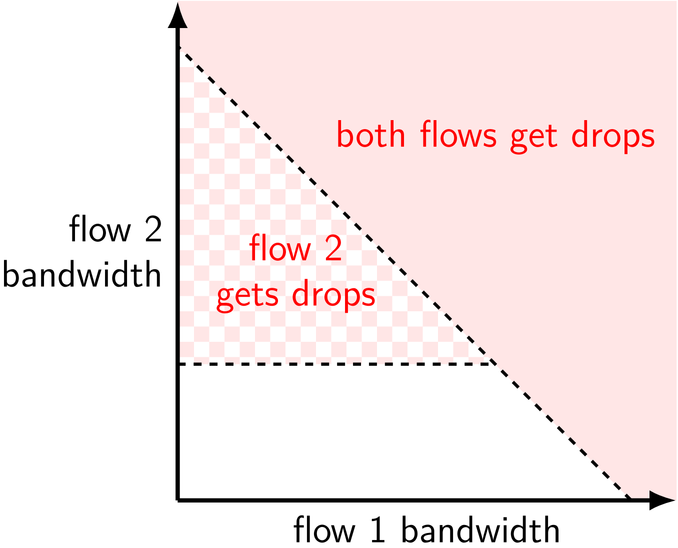
# a scenario



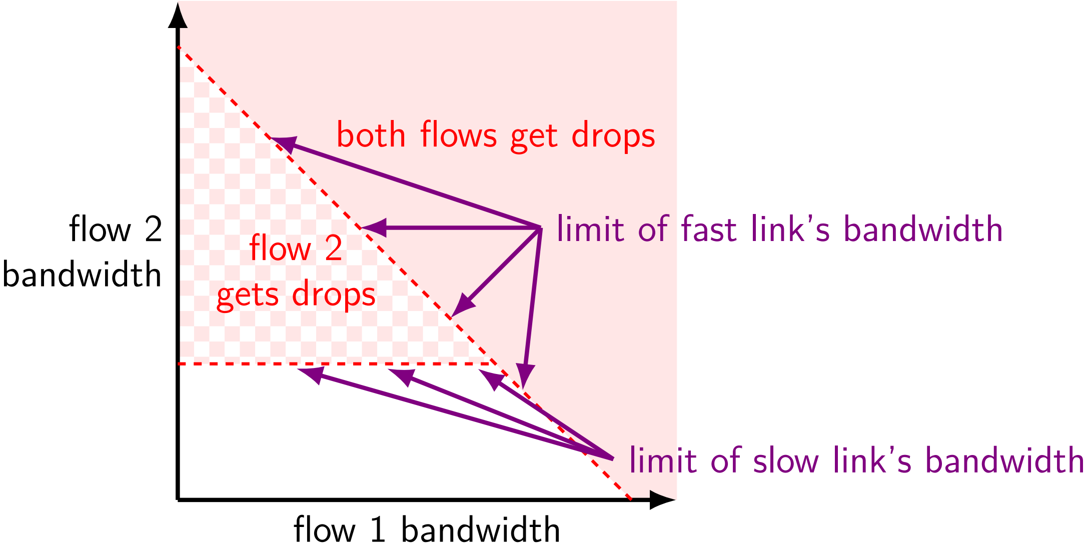
# a scenario



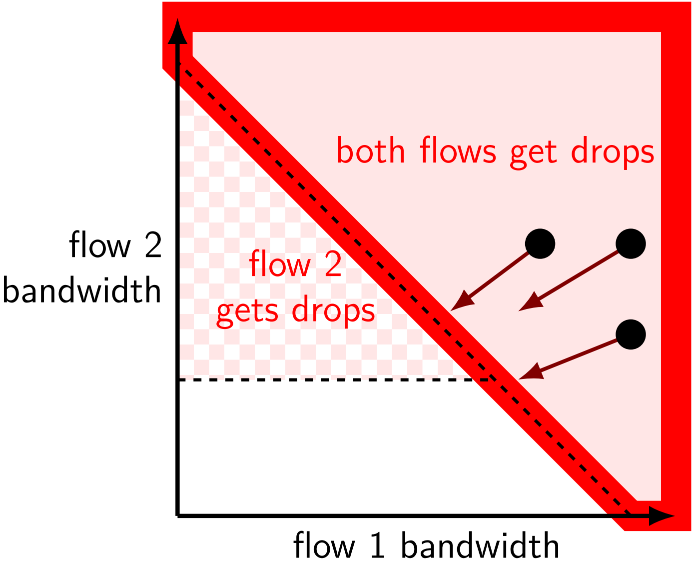
# slow link + mixed traffic



# slow link + mixed traffic

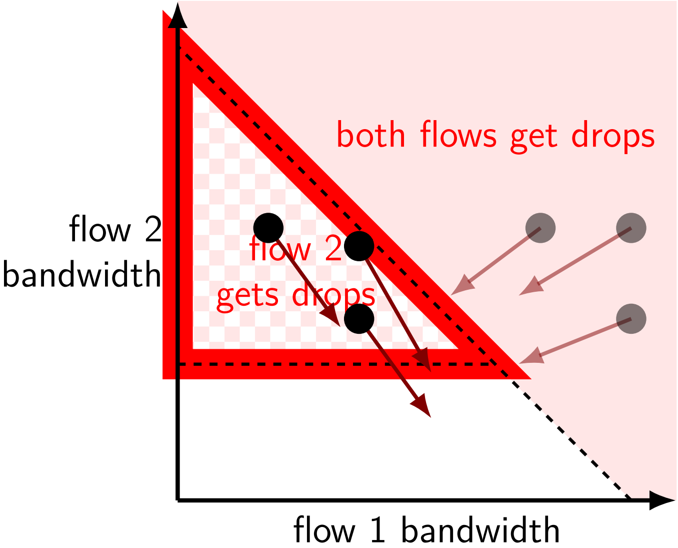


# slow link + mixed traffic



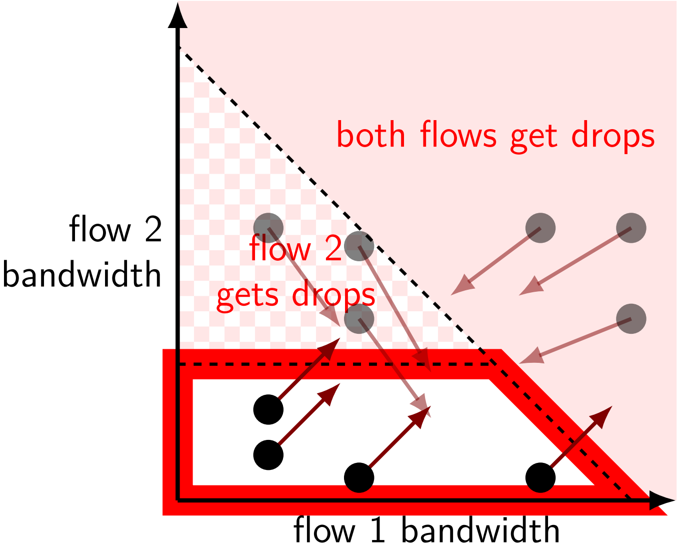

if both flows see drops,
multiplicative decrease
(toward origin)

# slow link + mixed traffic



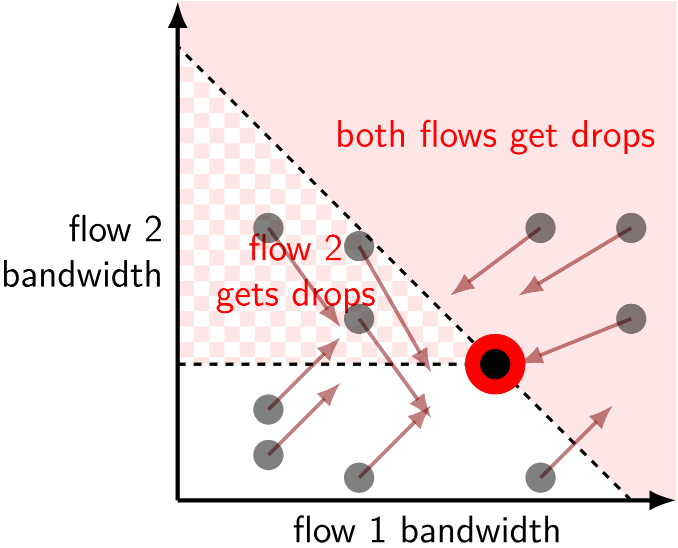

if only flow 2 see drops,
it decreases bandwidth and
flow 1 increase bandwidth

# 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

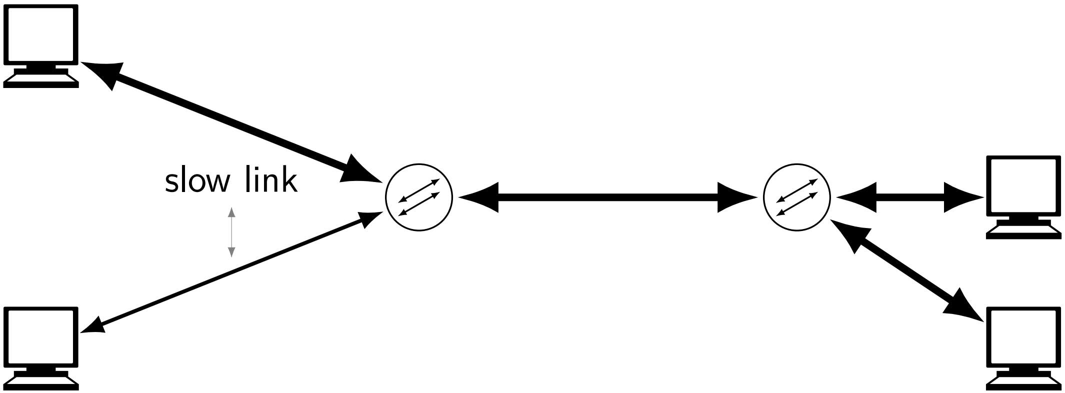
unfair allocation = someone gets much less than others

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



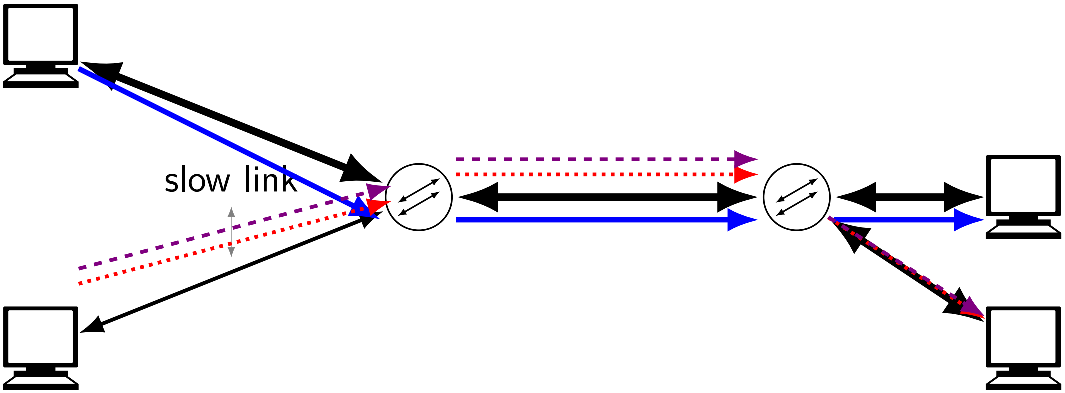

let's say slow link has 5 MByte/s capacity, other links 20MByte/s

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

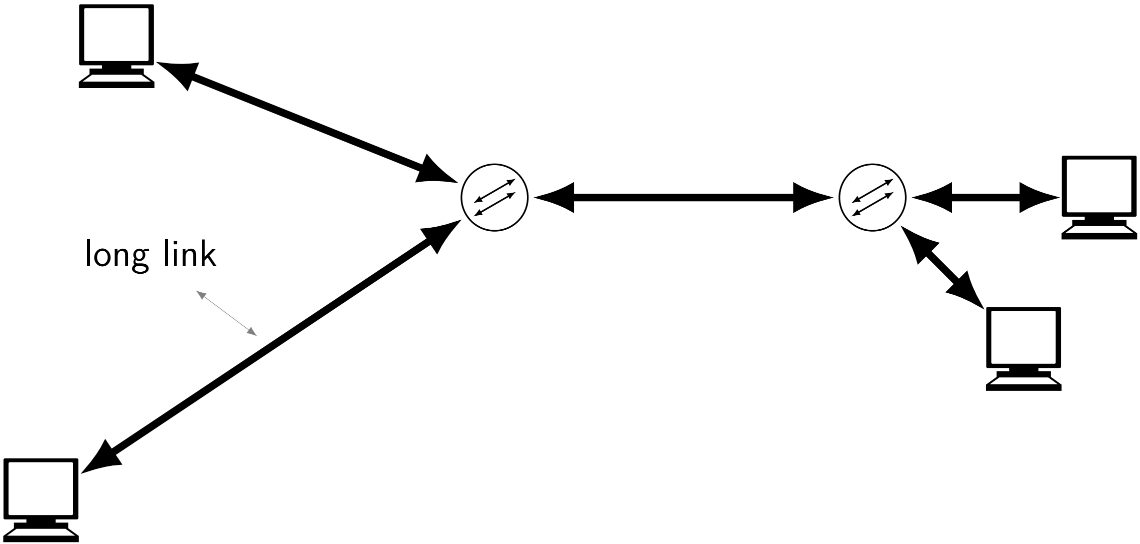
more common metric, but for scenarios where equal allocation makes sense

if $x_i$ is i'th flow's share:

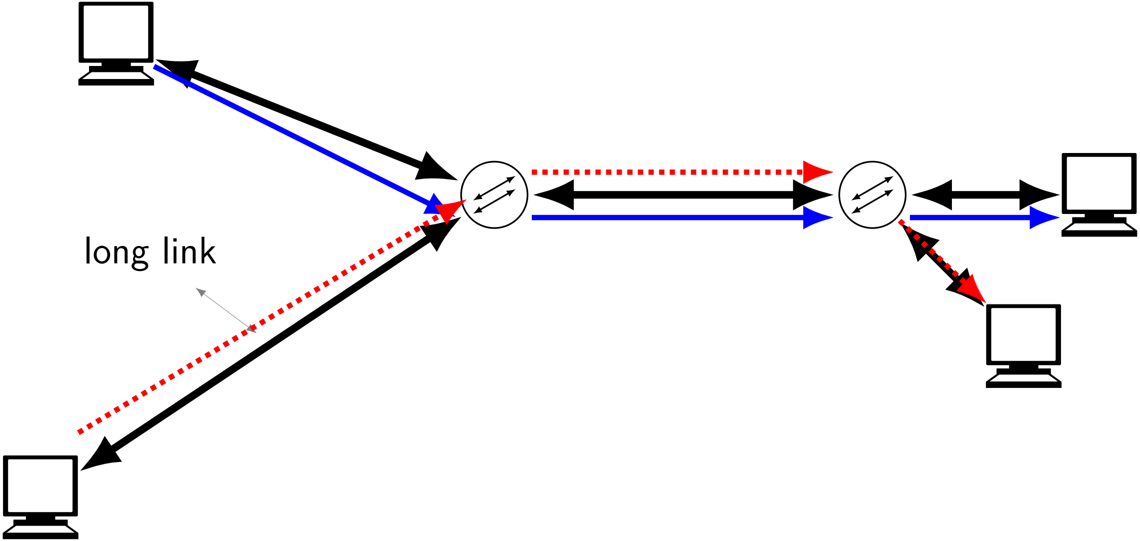
$$\frac{\left(\sum x_i\right)^2}{n \cdot \sum x_i^2}$$

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?

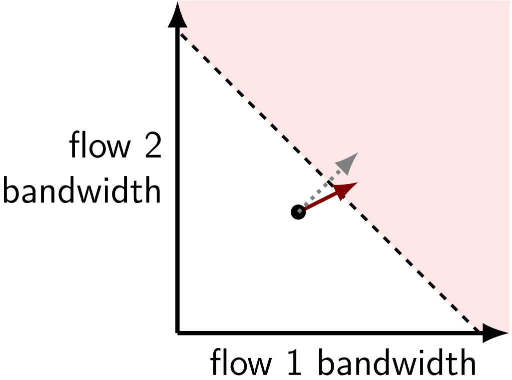
1: window size 25; 2: window size 50

# revisiting additive increase

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

flow 1: window $25+1 \rightarrow 26$ pkt/50 ms $=$ 520 pkt/sec

flow 2: window $50+1 \rightarrow 51$ pkt/100 ms $=$ 510 pkt/sec



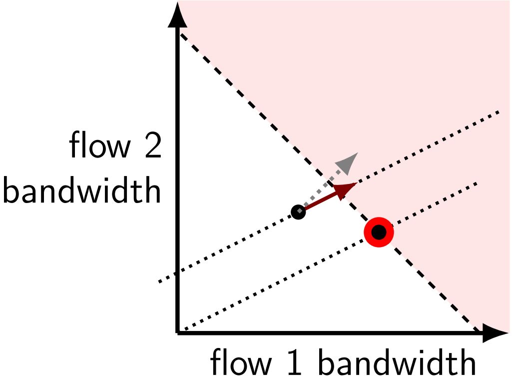

flow 1 increases faster
than flow 2 increases
not 45-degree angle anymore

# 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

lots of changes to congestion control

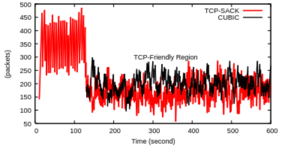
some used in modern TCP implementations

on Internet, need to be compatible with "normal" TCP

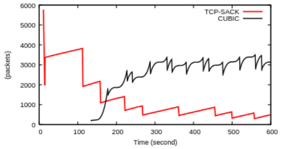
"TCP-friendly"

should not make TCP used alongside them behave poorly

# examples: checking versus TCP




(a) RTT 8ms.



(b) RTT 82ms.

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

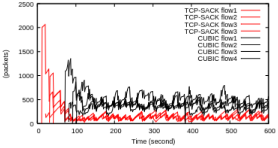



**Figure 6: Four TCP-SACK flows and four CUBIC flows over 40ms RTT**

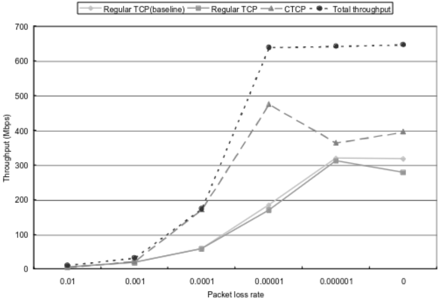



Figure 11. Throughput of CTCP and Regular TCP flows when competing for same bottleneck.

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

# 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"

was actually not done in early versions of TCP

but problem: what to do with congestion window

solution called 'fast recovery'

# self-clocking and dup-ACKs

without losses, sender sends one new packet per ACK

keeps number of packets in network constant

but duplicate ACKs are exception (say window size 6):

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

# alternate explanation

sender stopped sending while receiving duplicate ACKs

but we know *most messages got there*

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

# TCP's fast retransmission

on third duplicate ACK:

resend packet,

do multiplicative decrease, AND THEN

temporarily add packet to window for each dup ACK

- send packets to replace received packet
- (if allowed by multiplicative-decreased window)

reset window size back when 'new' ACK

# self-clocking and fast retransmit

adjust window size to keep packets in flight constant:

| recv'd | sent | count packets in flight | send window size (range) |
|---|---|---|---|
| — | data 0-5 | 6 | 6 (0-5) |
| ACK 0 | | 5 | 6 (1-6) |
| | data 6 | 6 | 6 (1-6) |
| ACK 1 | | 5 | 6 (2-7) |
| | data 7 | 6 | 6 (2-7) |
| ACK 1 | | 5 | 6 (2-7) |
| ACK 1 | | 4 | 6 (2-7) |
| ACK 1 | | 3 | 6 (2-7) |
| | data 2 | 4 | 8 (2-9) |
| | data 8 | 5 | 8 (2-9) |
| | data 9 | 6 | 8 (2-9) |
| ACK 1 | | 5 | 9 (2-10) |
| 1 | data 10 | 5 | 9 (2-10) |

# 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

problem is that additive increase doesn’t find capacity quickly

exponential(ish) increase to find *initial* window size

# “slow start” on connection begin

set window size $=$ 1 packet

increase by one packet for each ACK

…until first packet loss

then revert to additive increase

actually slower at increasing

# “slow start” later

keep track of window size after multiplicative decrease

called `ssthresh`

- probably variable name in BSD code for this

use slow start when window size lower than ssthresh

but how can that happen?

- need something other than multiplicative decrease

# decrease versus reset

on duplicate ACK (most common case):

do multiplicative decrease

on timeout:

reset window size to 1 packet

after timeout, use “slow start”

…until ssthresh reached or congestion

intuition: don’t assume halving is enough

intuition: find correct lower window size faster

# 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?

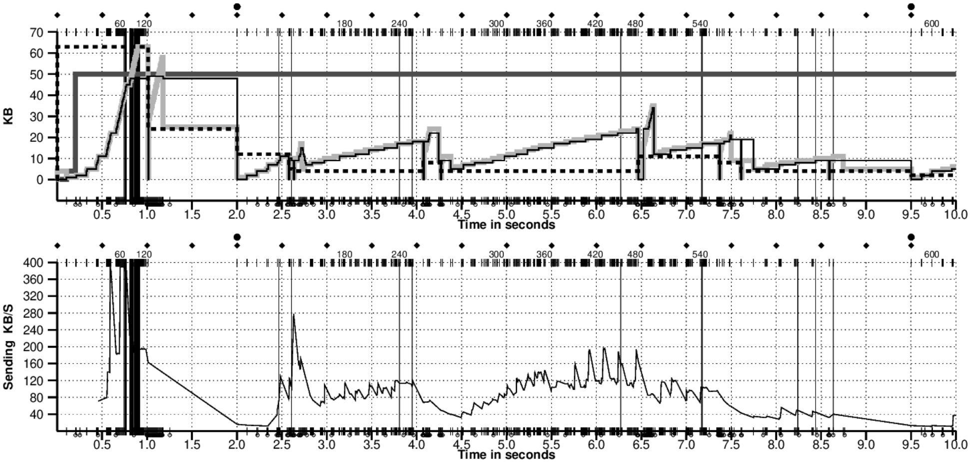


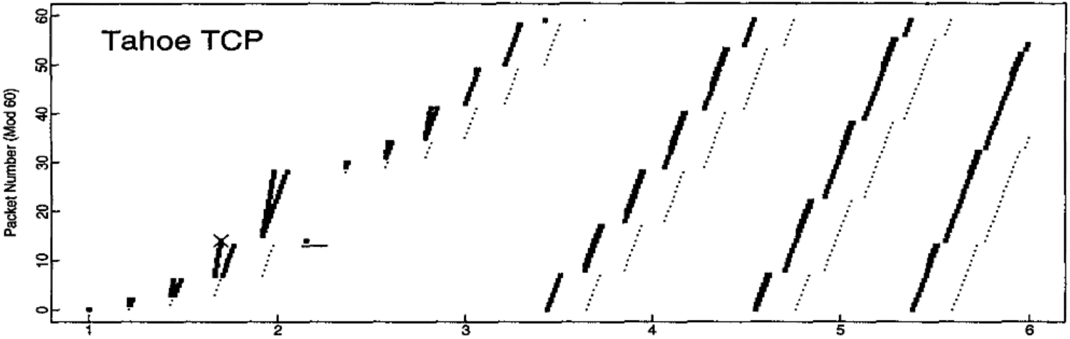

Figure 1: TCP Reno Trace Examples.

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

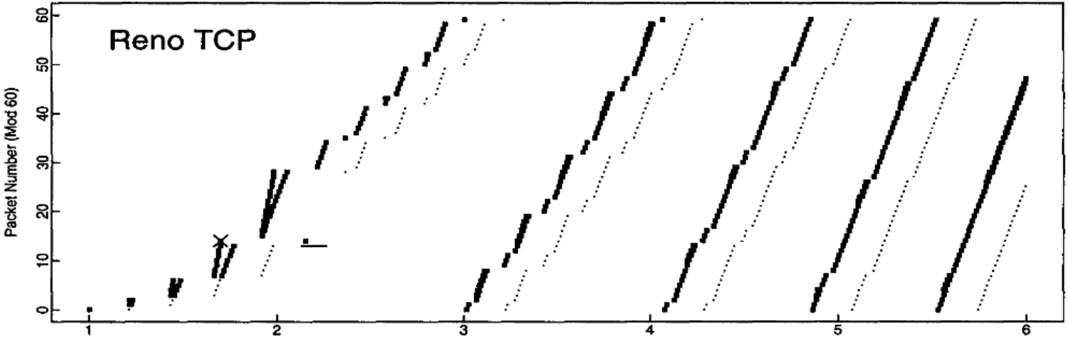
from Kevin Fall and Sally Floyd, “Simulation-based Comparisons of Tahoe, Reno, and SACK TCP”




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

# 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.)

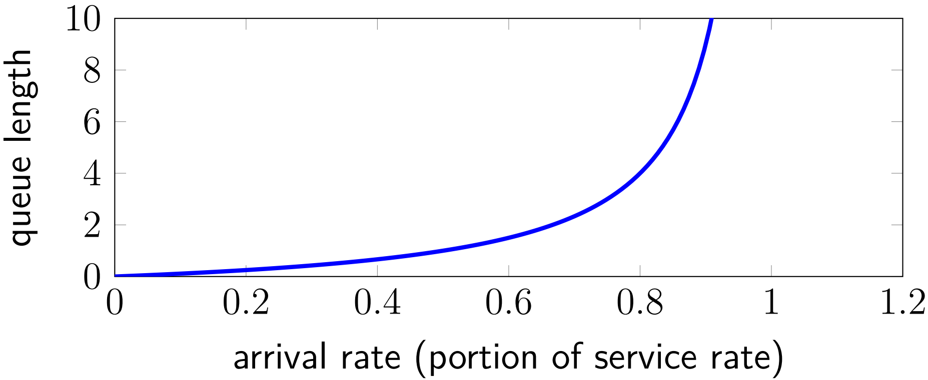
1 — one "server" (thing that can process packets)

$\infty$ — unlimited queue length

# M/M/1/∞ queue length

mean queue length

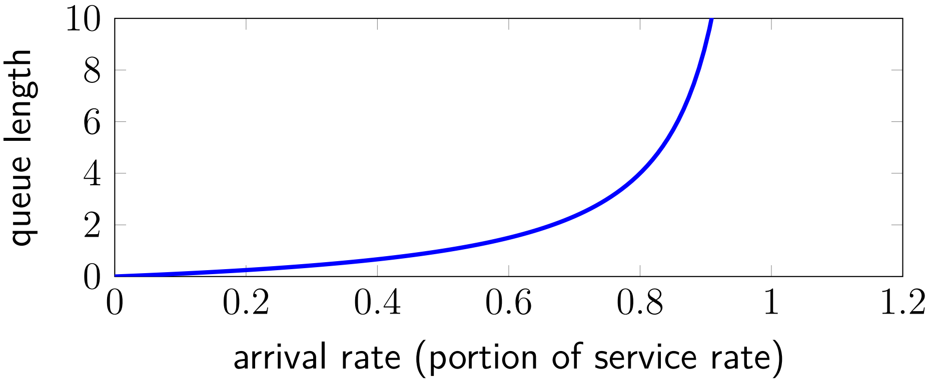
$$\frac{\text{arrival rate}}{\text{service rate} - \text{arrival rate}}$$



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

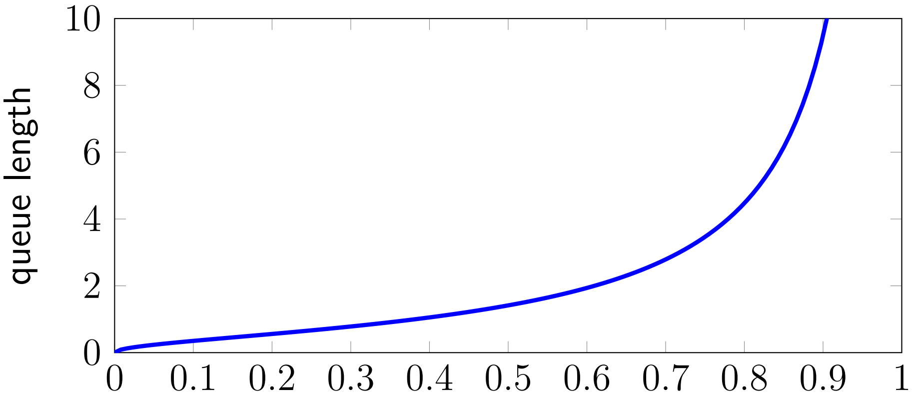
mean queue length

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

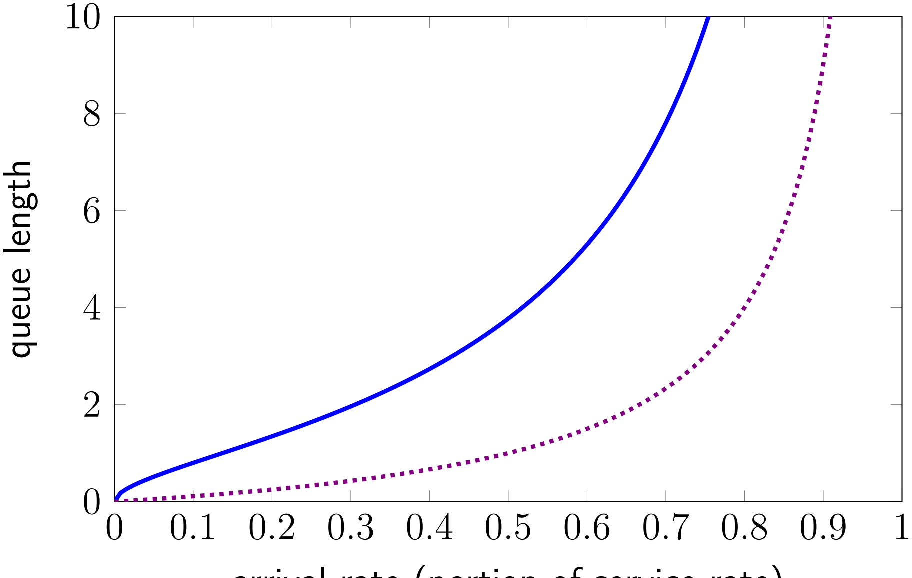


# M/M/1/$\infty$ queue length std. deviation

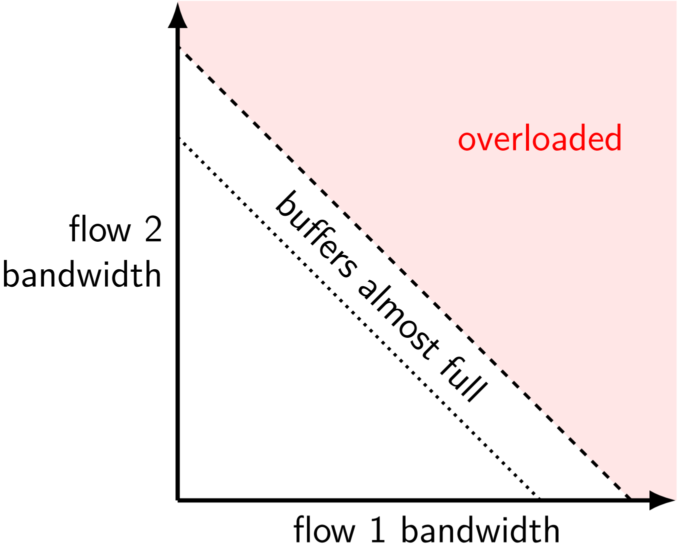
$$\sqrt{\frac{\text{utilization}}{(1-\text{utilization})^2}}$$



# approx 95th pctile v mean queue length

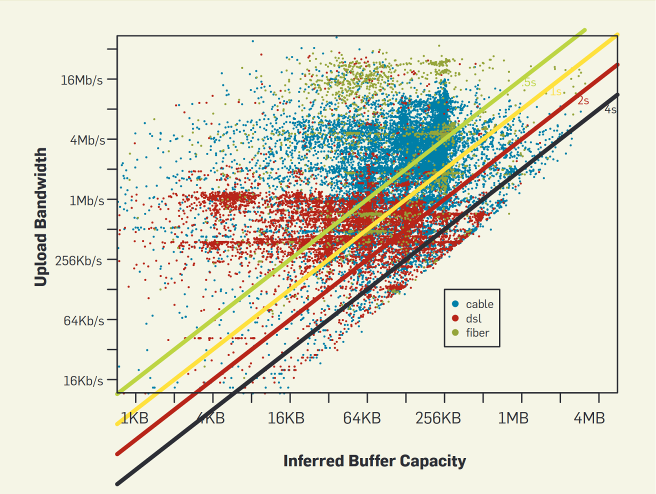


# filling buffers



# 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 $\times$ 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 $i$th retransmission is $2^i \times$ base timeout

intuition: avoids overloading network by being a lot less aggressive

not aware of good theoretically results in TCP context

- famous result that this type of backoff is good for things like deciding when to retransmit on sahred wireless link
(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)

# some connected questions

do we really need packet loss?

what does congestoin control due to latency?

# 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

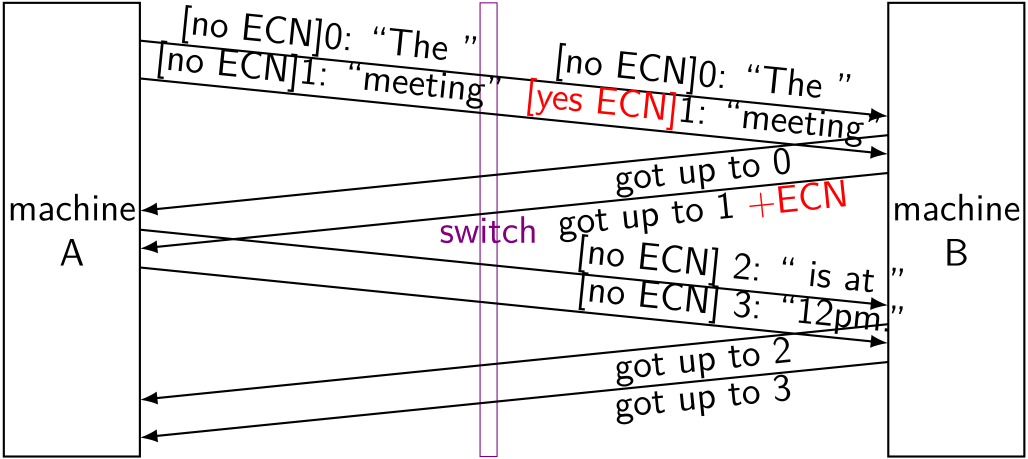
RFC 3168 says:

`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 $\alpha$

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

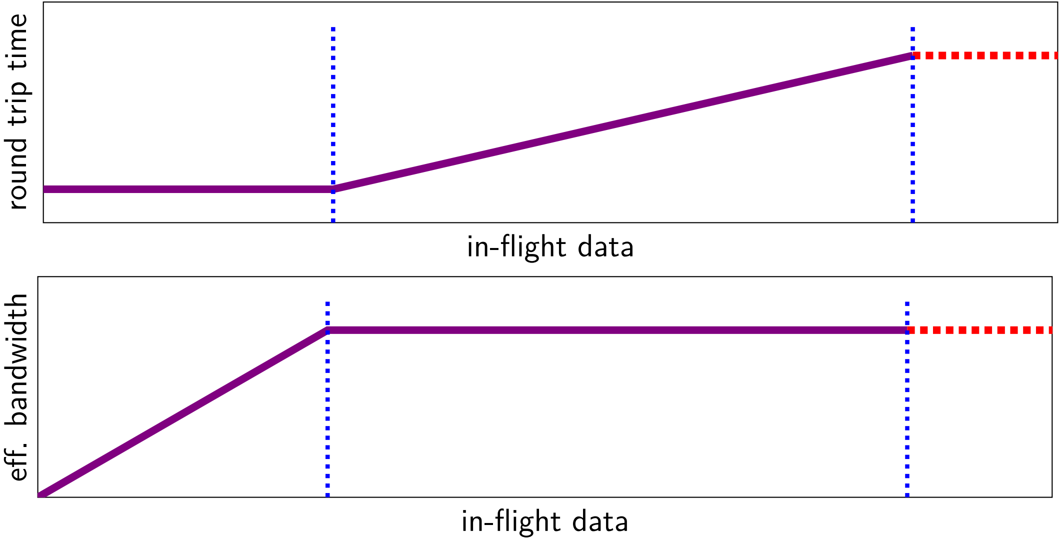
some alternate ideas:

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 ≈ 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

# 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)

# backup slides