Congestion Control In The Internet

Part 2: Implementation

EPFL

Contents

6. Congestion Control in the Internet was initially only in TCP

Why?

Easy to add end-to-end congestion control to TCP, as TCP already maintains an end-to-end connection

How? In addition to what we discussed about TCP

- a TCP source adjusts its "rate" to the network-congestion status by:
  - *adjusting the sliding window*
  - using techniques like: *additive increase/multiplicative decrease (AIMD)*, and *slow start*
  - leveraging *implicit* or *explicit feedback* from network (according to the Decbit principle)
- this avoids congestion collapses and ensures *some sort of* fairness

Many congestion control algorithms

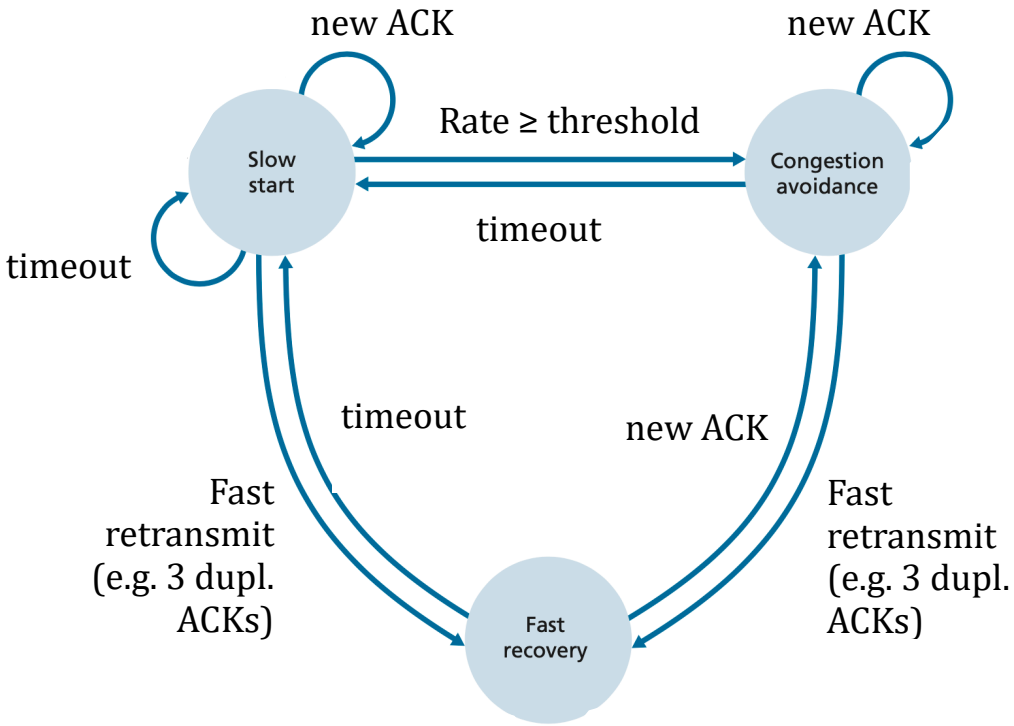
Popular ones are: TCP Reno (the most mature and well explored, widely used until recently)
TCP Cubic (widespread today in Linux servers)
Data Center TCP (Microsoft and Linux servers in data centers)
TCP-BBR (Google, Whatsapp, etc)

TCP Reno uses ~AIMD, Slow Start, implicit feedback

- Negative feedback = loss detected
  - *multiplicative decrease*
- Positive feedback = new (non-duplicate) ACK received
  - *multiplicative* or *additive increase* depending on the phase

- 3 phases or states:
  - Slow Start = (approx.) the same as theoretical slow start with *multiplicative increase* factor $w_0 \approx 2$ per RTT
  - Congestion Avoidance = *additive increase* with term $v_0 \approx +1$ MSS per RTT
  - Fast Recovery [see next]

- Transitions between states:
  - Initial state is Slow Start
  - Slow Start → Congestion Avoidance via a *threshold*
  - Any state → Slow Start if loss is detected via timeout
  - Any state → Fast Recovery if loss is detected via fast retransmit



What exactly TCP Reno does to adjust its sending rate?

- TCP Reno adjusts the sliding *window size*
- based on the approximation: $rate \approx \frac{W}{RTT}$

```
W = min (cwnd, offeredWindow)
```

`offeredWindow` = window advertized by other end in TCP's window field
`cwnd` = controlled by TCP congestion control

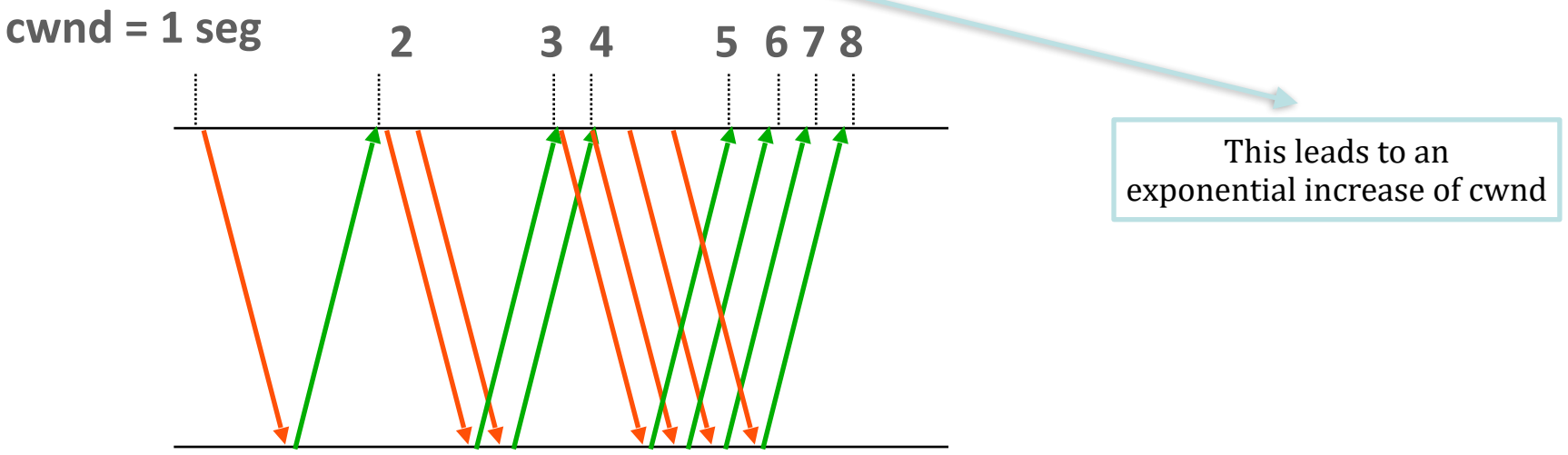
Slow Start by adjusting cwnd…

…multiplicative increase: (Slow Start)

- For the initial slow start, the target window `ssthresh` (for the target rate) is set to 64KB
- At each new (non-duplicate) ack received during slow start:

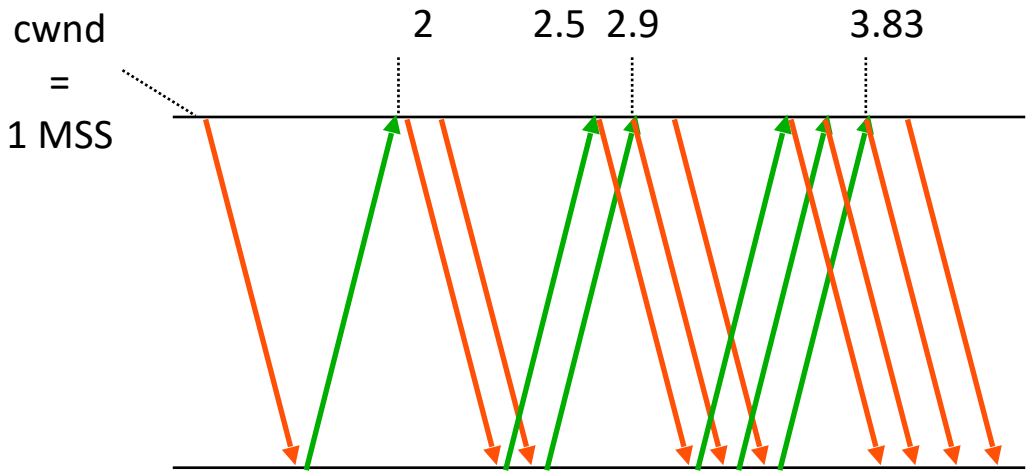
  `cwnd = cwnd + MSS (in bytes)`

  - if counted in packets, this would be: `cwnd = cwnd + 1 (in packets)`
  - i.e. a multiplicative increase with factor $w_0 = 2$ per RTT, *approximately*



- if cwnd ≥ ssthresh, then go to congestion avoidance

AIMD by adjusting cwnd…



additive increase: (Congestion avoidance)

- for every new (non-duplicate) ack received:

  `cwnd = cwnd + MSS × MSS/cwnd`

- if counted in packets, this would be:

  `cwnd+=1/cwnd,`

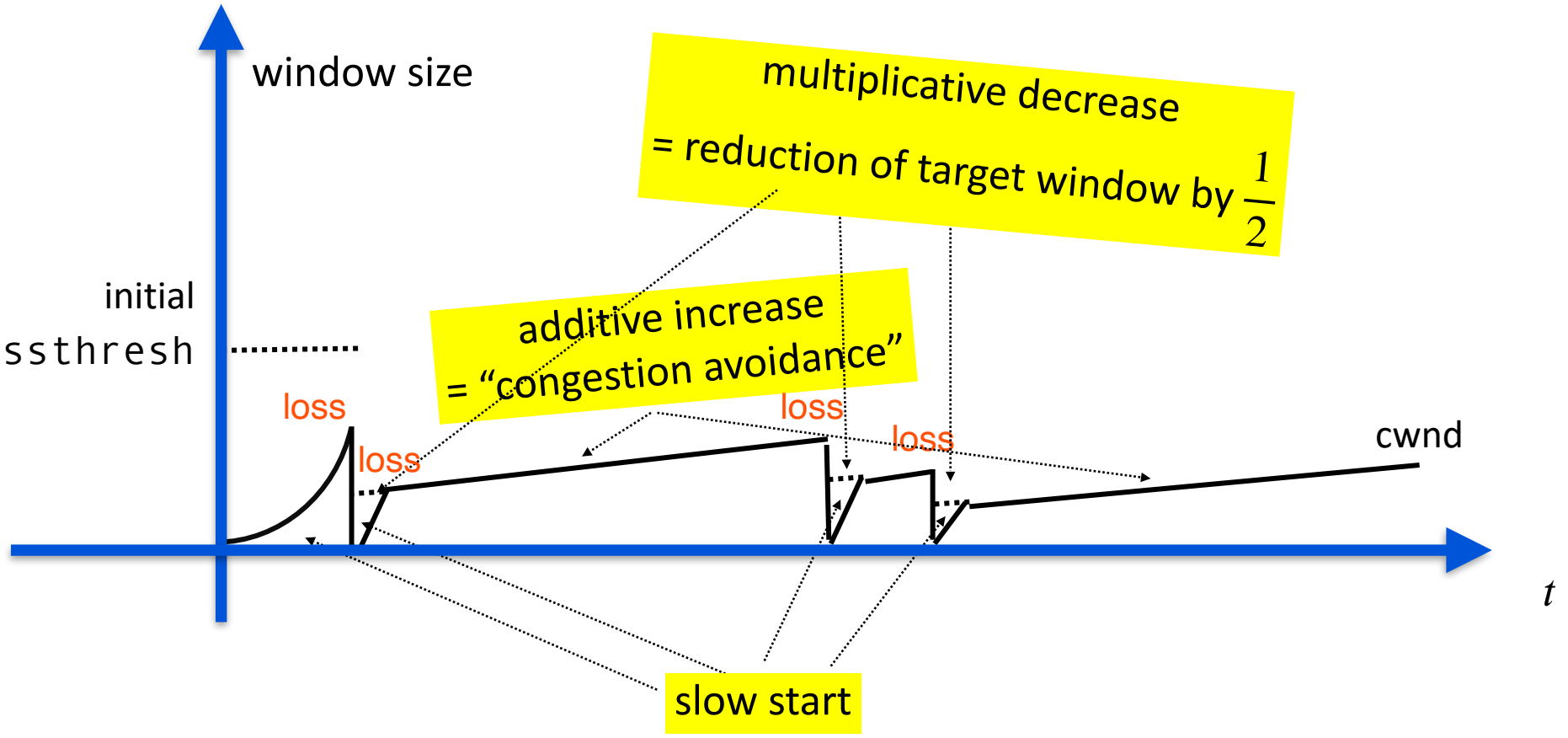
  slightly less than additive increase (< 1 MSS/RTT)

- other implementations also exist:
  - e.g. wait until cwnd bytes are ack'ed and then increment cwnd by 1 MSS

multiplicative decrease: (after detecting loss — negative feedback)

- `ssthresh = 0.5 × cwnd`
- `cwnd = 1MSS` (if timeout) ***or something else*** (if fast retransmit) [see Fast Recovery]

Example of congestion-window evolution without Fast recovery



Recall:

- there is a slow start phase initially and after every packet loss detected by timeout
- target window of slow start is called `ssthresh` («slow start threshold»)

Fast Recovery

Why?

- Loss detected by fast retransmit is not severe—we just want to apply multiplicative decrease with $u_1 = 0.5$
- but halving the cwnd is *not* a good approach;
  - formula "$rate \approx \frac{W}{RTT}$" is not true when there is a single isolated packet loss;
  - sliding window operation may even stop sending, if the first packet of a batch is lost

What?

- target window is halved: `ssthresh = 0.5 × cwnd`
- but congestion window is allowed to increase beyond the target window until the loss is repaired—it is increased by the value of duplicate ACKs
- *artificial increase* to keep sending segments

Algorithm:

When loss is detected by 3 duplicate ACKs at any phase:

```
ssthresh = 0.5 × current-cwnd
ssthresh = max (ssthresh, 2 × MSS)
cwnd = ssthresh + 3 × MSS
cwnd = min (cwnd, 64K)
```

Goto Fast Recovery

When in Fast Recovery, for each *duplicate ACK* received:

```
cwnd = cwnd + MSS (exp. increase)
cwnd = min (cwnd, 64K)
```

If loss is repaired

```
cwnd = ssthresh
```

Goto Congestion Avoidance

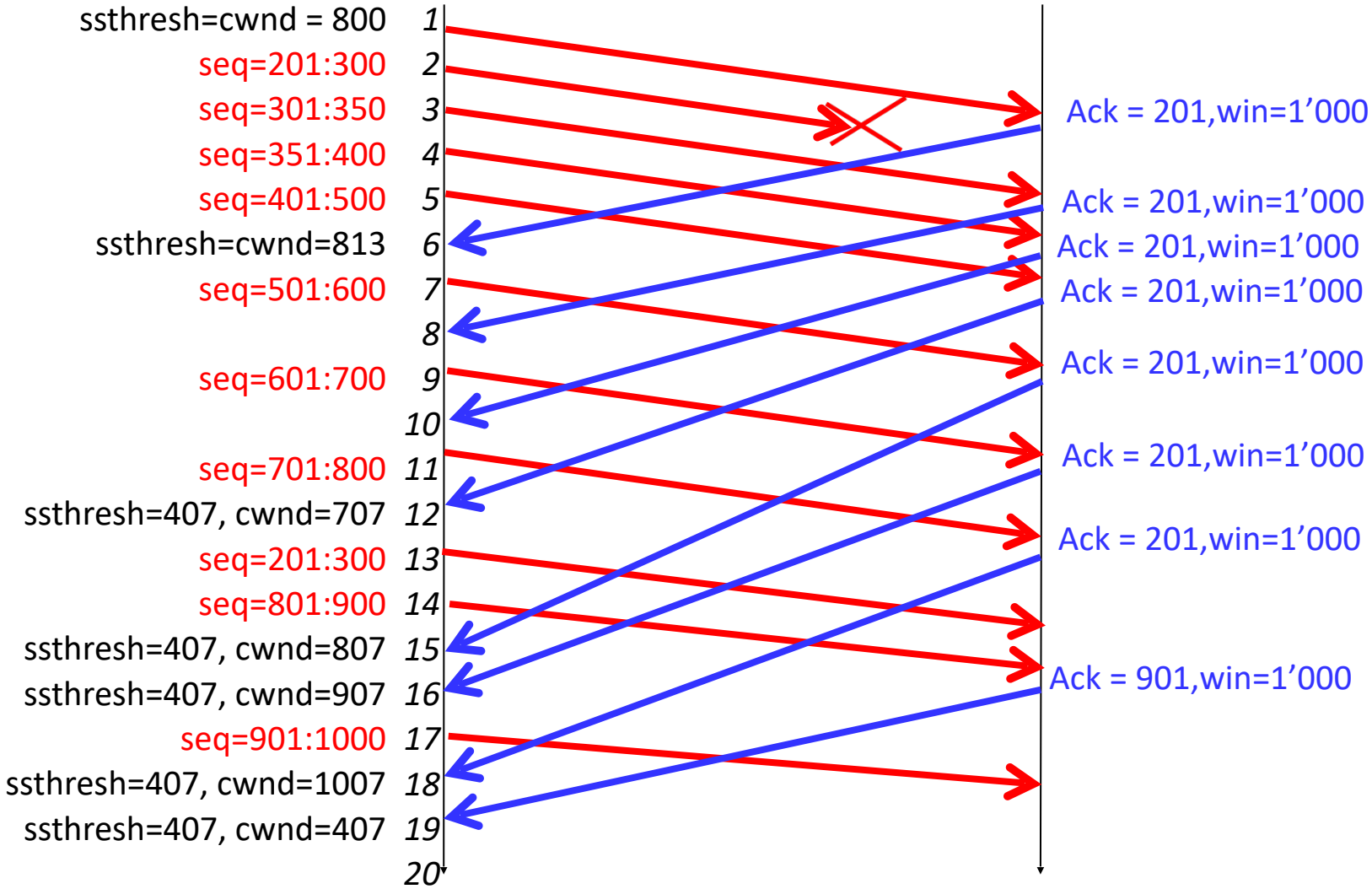
else (timeout)

Goto Slow Start

Fast Recovery Example

MSS = 100

TcpMaxDupACKs=3



During congestion avoidance:

$$\text{cwnd} \leftarrow \text{cwnd} + \frac{\text{MSS}^2}{\text{cwnd}}$$

At time 1, the sender is in “congestion avoidance” mode. The congestion window increases with every received non-duplicate ack (as at time 6). The target window (ssthresh) is equal to the congestion window.

The second packet is lost.

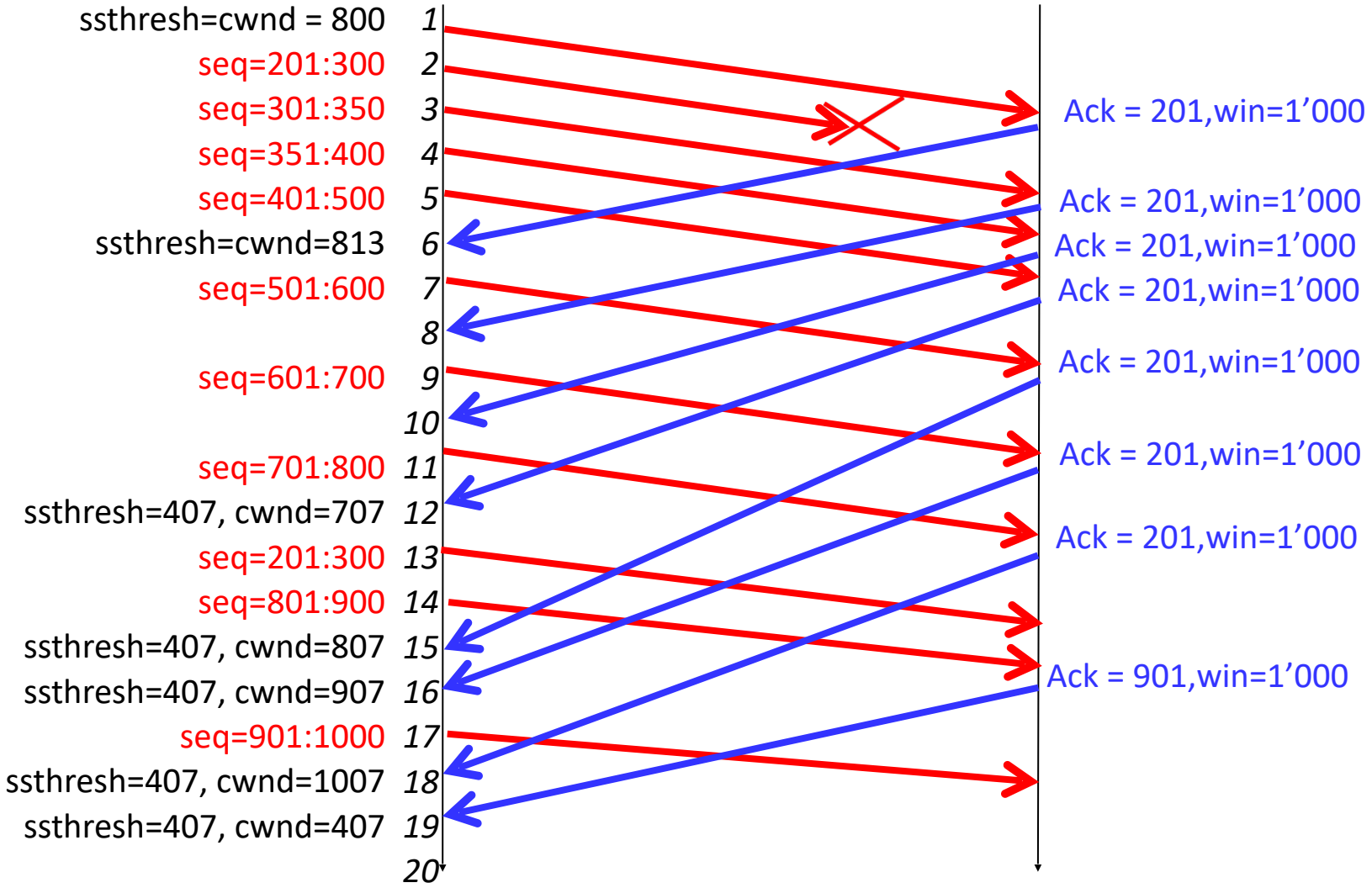
At time 12, its loss is detected by fast retransmit, i.e. reception of 3 duplicate acks. The sender goes into “fast recovery” mode. The target window is set to half the value of the congestion window; the congestion window is set to the target window plus 3 packets (one for each duplicate ack received).

At time 13 the source retransmits the lost packet. At time 14 it transmits a fresh packet. This is possible because the window is large enough. The window size, which is the minimum of the congestion window and the advertised window, is equal to 707. Since the last acked byte is 201, it is possible to send up to 907.

At times 15, 16 and 18, the congestion window is increased by 1 MSS, i.e. 100 bytes, by application of the fast recovery algorithm. At time 15, this allows to send one fresh packet, which occurs at time 17.

At time 19 the lost packet is acked, the source exits the fast recovery mode and enters congestion avoidance. The congestion window is set to the target window.

How many new segments of size 100 bytes can the source send at time 20 ?



Go to web.speakup.info
or
download speakup app
Join room
46045

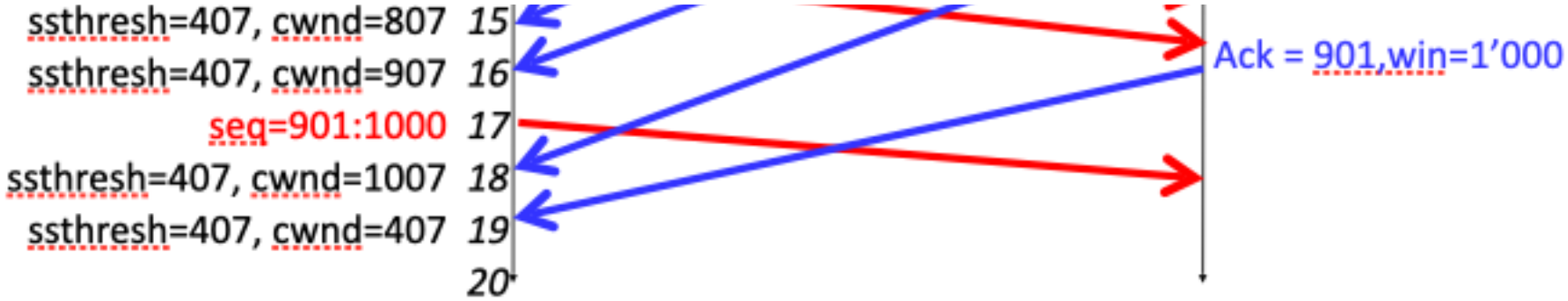
A. 1
B. 2
C. 3
D. 4
E. ≥ 5
F. 0
G. I don't know

Solution

Answer C

The congestion window is 407, the advertised window is 1000, and the last ack received is 901.

The source can send bytes 901 to 1308, the segment 901:1001 was already sent, i.e. the source can send 3 new segments of 100 bytes each.



TCP Reno — recap

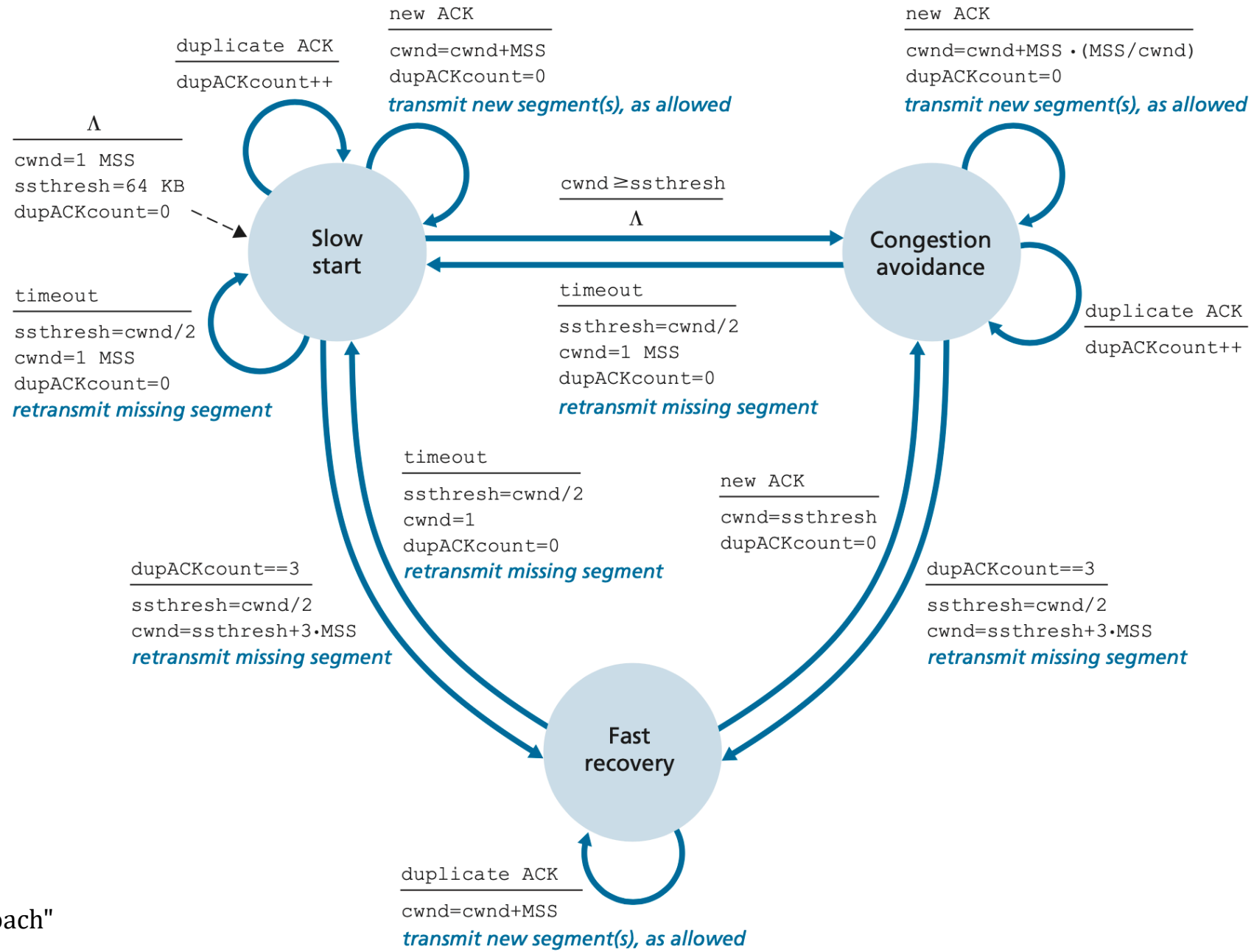


Figure from our textbook:
"Computer Networking: A top-down approach"
by J. Kurose and K. Ross

Assume a TCP flow uses WiFi with high loss ratio. Assume some packets are still lost in spite of WiFi retransmissions. When a packet is lost on the WiFi link…

A. The TCP source knows it is a loss due to channel errors and not congestion, therefore does not reduce the window
B. The TCP source thinks it is a congestion loss and reduces its window
C. It depends if the MAC layer uses retransmissions
D. I don't know

Go to web.speakup.info
or
download speakup app
Join room
46045

Solution

Answer B: the TCP source does not know the cause of a loss.

Side-effect:

A flow that experiences accidental losses on its wireless access link may ***never manage to get its fair share on another bottleneck link down its path***, because it will be constantly reducing its sending rate “thinking that it experiences congestion”.

Solutions:

Explicit Congestion Notification from the network [see later]

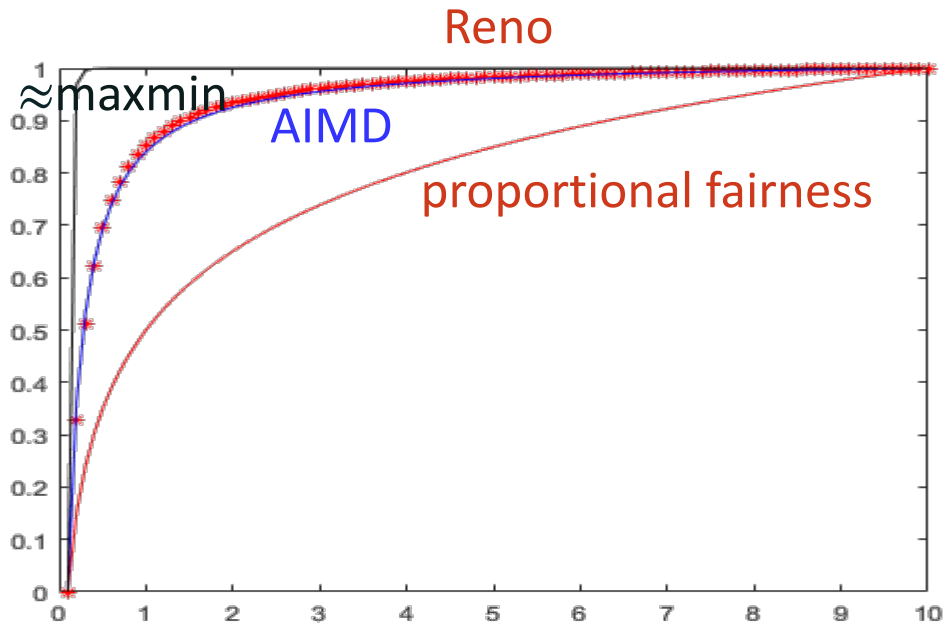
Dynamic (more sophisticated) coding at the physical layer to avoid errors on the wireless link

Fairness of TCP Reno

For *long lived flows*, the rates obtained with TCP Reno are as if they were distributed according to utility fairness, with utility of flow i given by $U_i(x_i) = \frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$

with x_i = rate (in MSSs) = W/τ_i, τ_i = RTT (see “Rate adaptation, Congestion Control and Fairness: A Tutorial")

For *flows that have same RTT*, the fairness of TCP is *between max-min and proportional fairness*, closer to proportional fairness:



rescaled utility functions;
RTT = 100 ms
maxmin approx. is $U(x) = 1 - x^{-5}$

TCP Reno and RTT

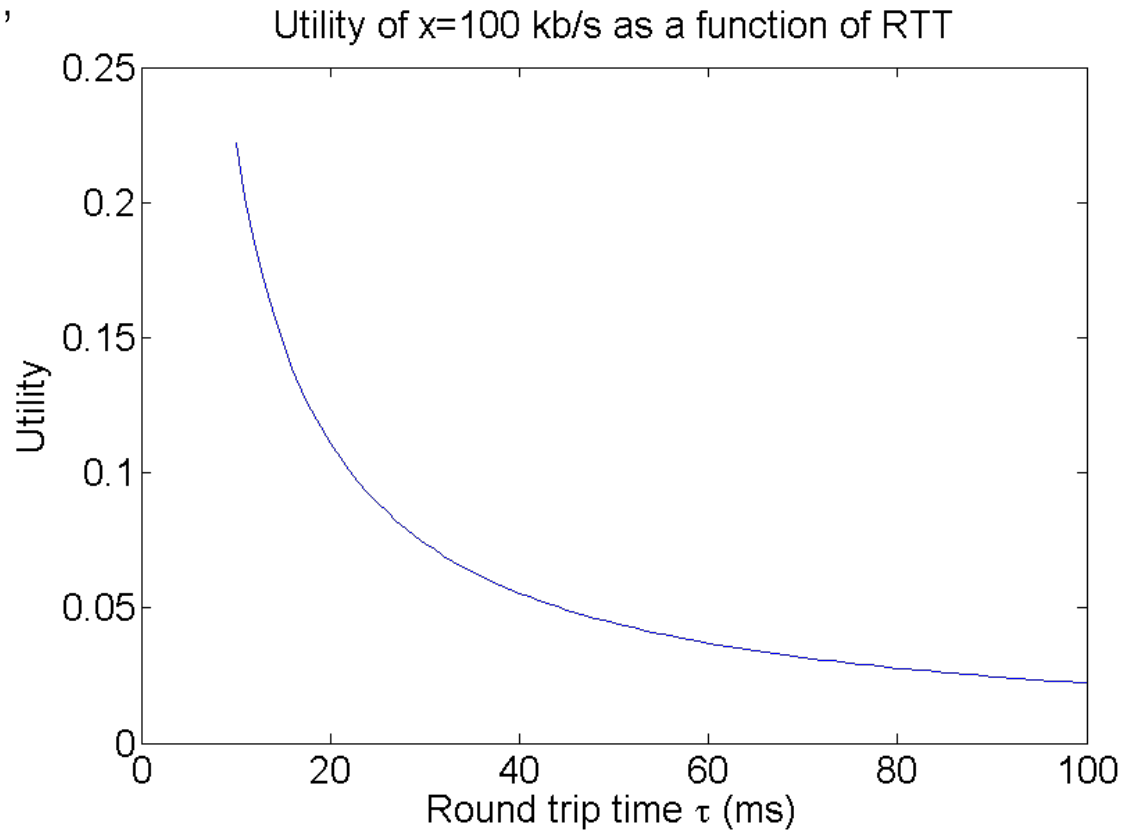
TCP Reno tends to distribute rate so as to maximize utility of source i given by

$$U_i(x_i) = \frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$$

The utility U depends on the roundtrip time τ_i;

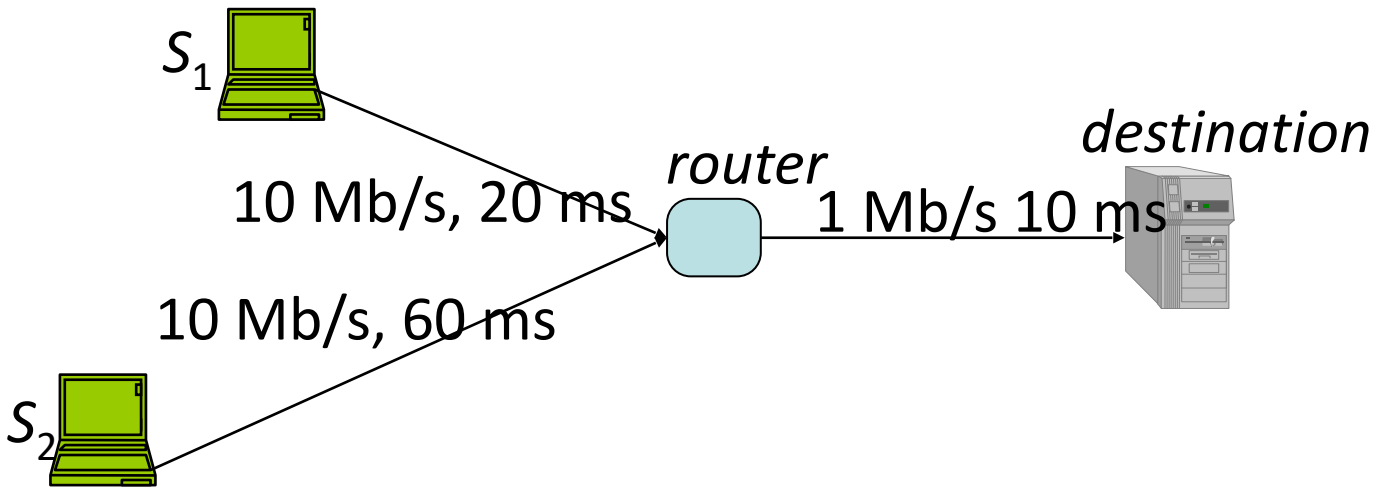
The utility U is a *decreasing function of RTT*

What does this imply ?



S_1 and S_2 send to destination using one TCP connection each, RTTs are 60ms and 140ms. Bottleneck is link « router-destination ». Who gets more ?

A. S_1 gets a higher throughput
B. S_2 gets a higher throughput
C. Both get the same
D. I don't know



Go to web.speakup.info
or
download speakup app
Join room
46045

Solution

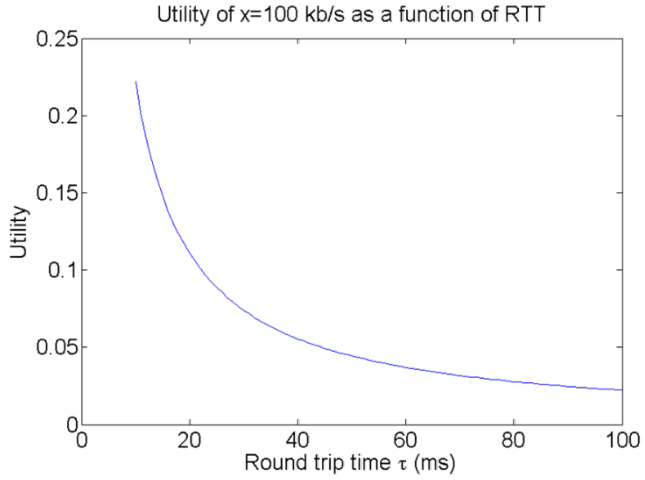
For long lived flows, the rates obtained with TCP are as if they were distributed according to utility fairness, with utility of flow i given by $U(x_i) = \frac{\sqrt{2}}{\tau_i}\arctan\frac{x_i\tau_i}{\sqrt{2}}$

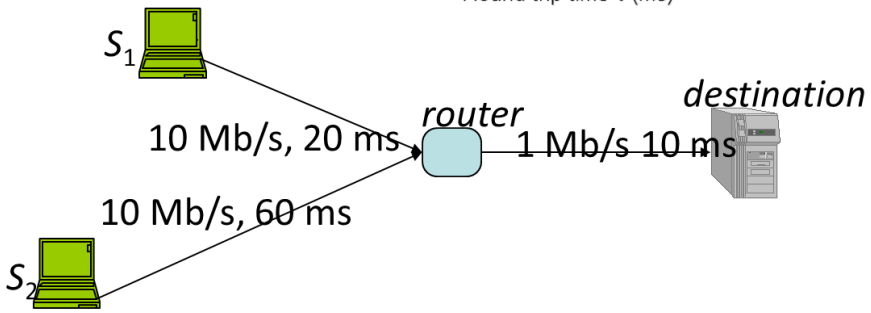
S_1 has a smaller RTT than S_2

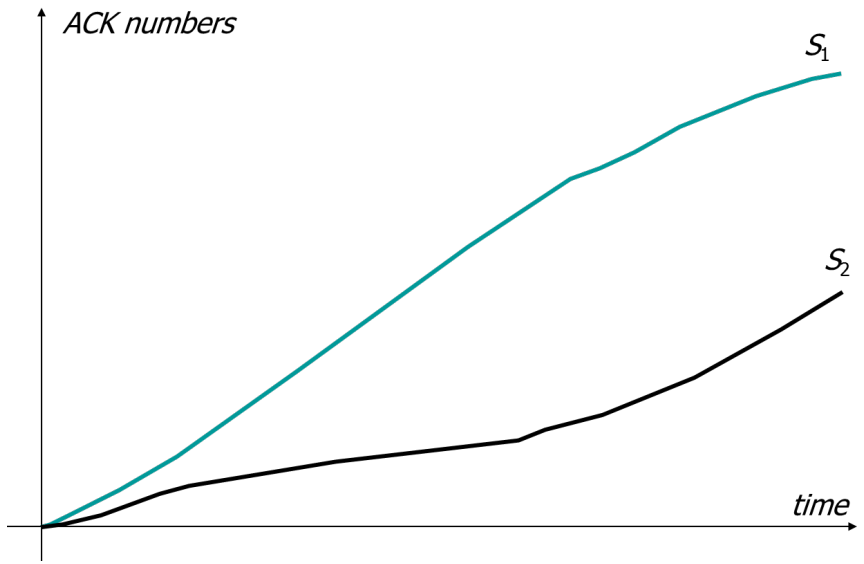
The utility is less when RTT is large; therefore, TCP tries less hard to give a high rate to sources with large RTT.

So, **S_2 gets less**.

Answer A.







The RTT Bias of TCP Reno

With TCP Reno, two competing flows with different RTTs are not treated equally

- flow with large RTT obtains less
- a (practical) explanation: additive increase is one packet per RTT (instead of one packet per constant time interval); so a flow with a *smaller RTT* can *"open" the window faster*.

A flow that uses ***many hops*** obtains less rate because of two combined factors:

1. If this flow goes over many congested links, it uses more resources. The mechanic of TCP Reno that is close to proportional fairness leads to this source having less rate, which is *desirable* in view of the theory of fairness.
2. If this flow has simply a larger RTT, then things are different. The mechanics of additive increase leads to this source having less rate,
   which is an *undesired* bias in the design of TCP Reno.

TCP Reno

Loss - Throughput Formula

Consider a ***large*** TCP flow size (many bytes to transmit).

Assume we observe that, in average, a fraction q of packets is lost (or marked with ECN).

The throughput should be close to $\theta = \frac{MSS\ 1.22}{RTT\sqrt{q}}$.

Formula **assumes**:

transmission time is negligible compared to RTT,

losses are rare and occur periodically,

time spent in Slow Start and Fast Recovery is negligible.

[see “Rate adaptation, Congestion Control and Fairness: A Tutorial”]

Guess the ratio between the throughputs θ_1 and θ_2 of S_1 and S_2 (assume: same MSS, same loss prob, and negligible transmission/ processing delays for both flows)

A. $\theta_1 = \frac{3}{7}\theta_2$

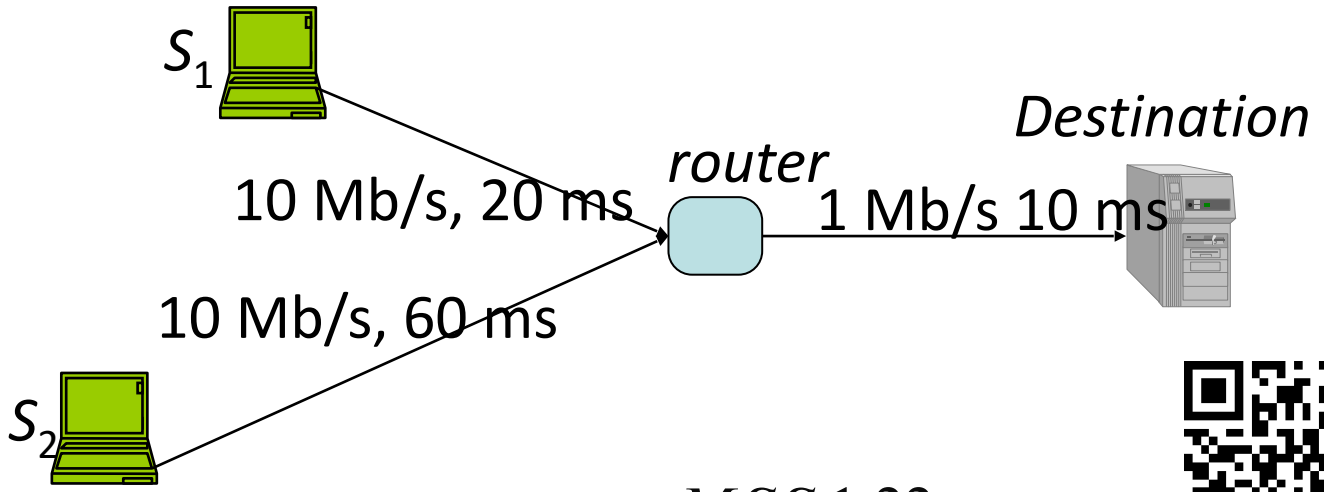
B. $\theta_1 = \theta_2$

C. $\theta_1 = \frac{7}{3}\theta_2$

D. $\theta_1 = \frac{10}{3}\theta_2$

E. None of the above

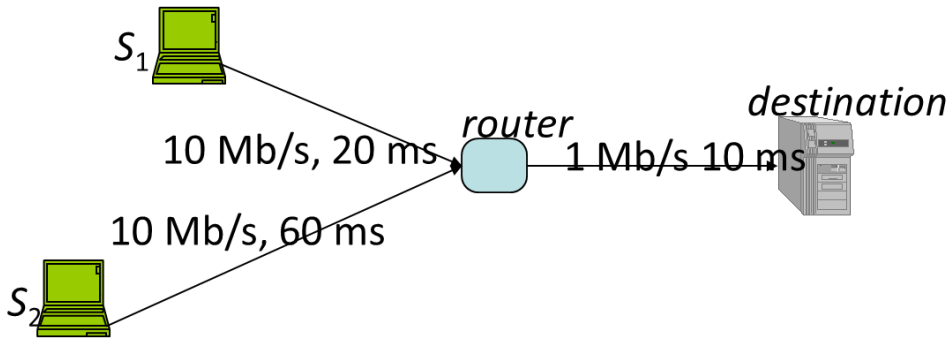
F. I don't know

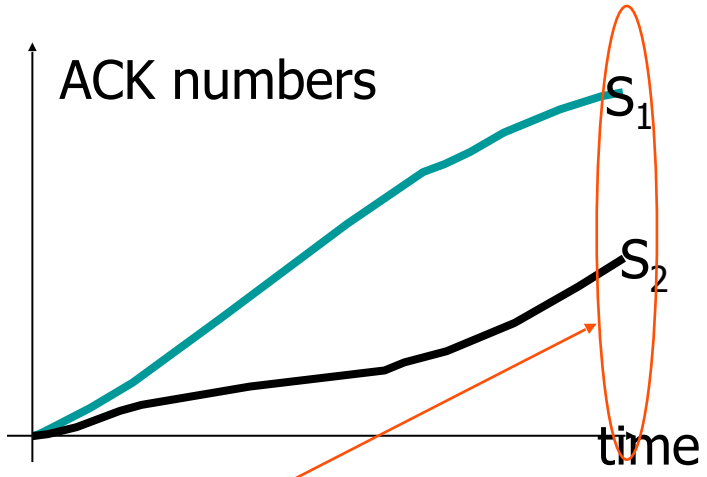


$$\theta = \frac{MSS\ 1.22}{RTT\sqrt{q}}$$

Go to web.speakup.info
or
download speakup app
Join room
46045

Solution





If processing time is negligible and router drops packets in the same proportion for all flows, then throughput is proportional to 1/RTT, thus

$$\frac{\theta_1}{\frac{1}{\tau_1}} = \frac{\theta_2}{\frac{1}{\tau_2}} \qquad \text{i.e.} \qquad \theta_1 = \frac{7}{3}\,\theta_2$$

Answer C.

TCP Reno — shortcomings

- RTT *bias* – not nice for users far away from the source
- Periodic *losses* must occur, not nice for applications (e.g video streaming).
- TCP controls the window, not the rate. Large *bursts* typically occur when packets are released by host following e.g. a window increase – not nice for queues in the internet, makes non smooth behavior.
- *Self inflicted delay*: if network buffers (in routers and switches) are large, TCP first fills buffers before adapting the rate. The RTT is increased unnecessarily. Buffers are constantly full, which reduces their usefulness (*bufferbloat syndrome*) and increases delay for all users. *Interactive, short flows experience large latency* when buffers are large and full.

Congestion control in UDP Applications

UDP applications that can adapt their rate have to implement congestion control.

One method is to use the *congestion control module* of TCP:
e.g. QUIC, which is over UDP, uses Cubic's congestion control (in its original version) or Reno's congestion control (in the standard version).

Another method (e.g. for videoconferencing application) is to control the rate by computing the rate that TCP Reno would obtain.
E.g.: *TFRC (TCP-Friendly Rate Control)* protocol

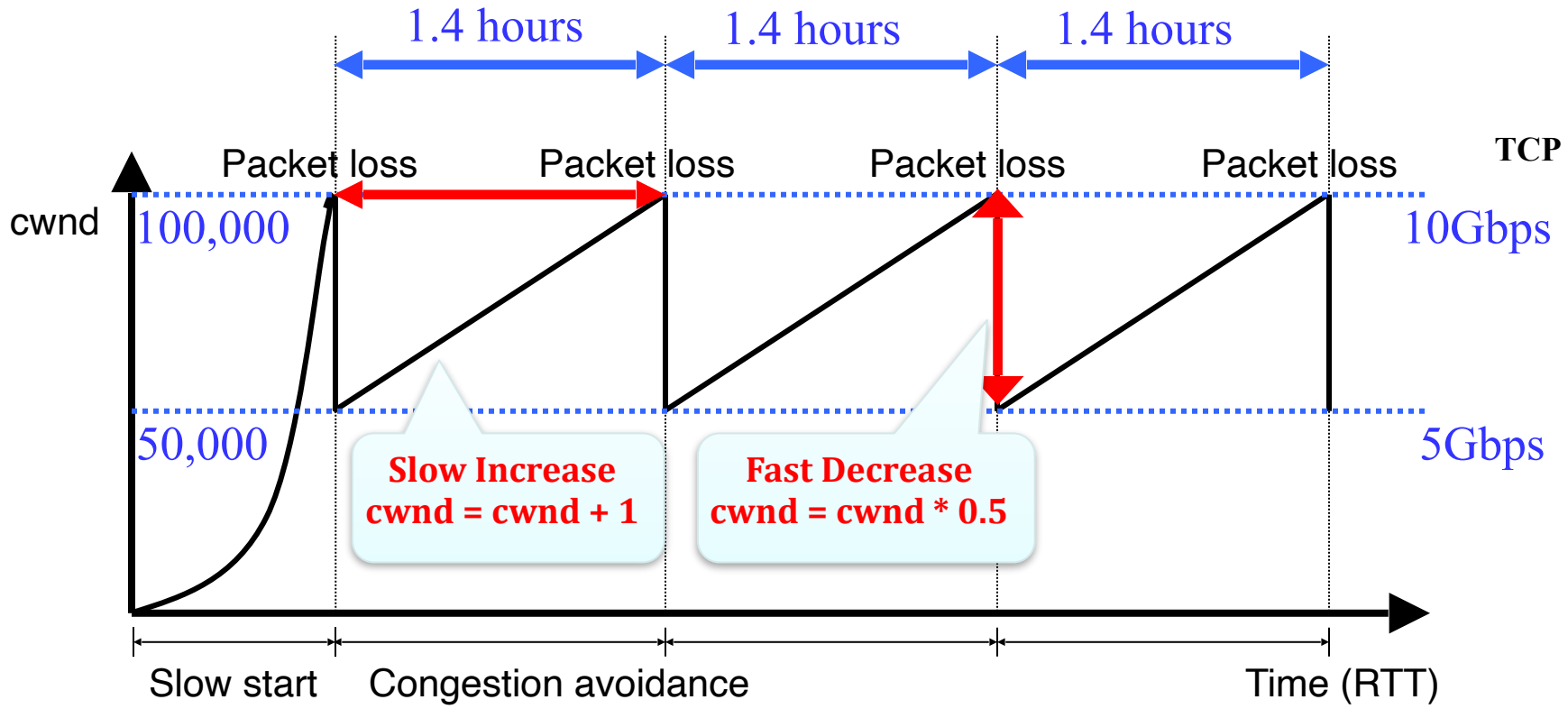
application adapts the sending rate (by modifying the coding rate for audio and video)

feedback is received in form of count of lost packets, used by source to estimate drop probability q

source sets rate to $x = \frac{MSS\ 1.22}{RTT\sqrt{q}}$ *(TCP Reno's loss throughput formula)*

7. TCP Cubic: Improving performance in Long Fat Networks (LFNs)

- In an LFN, additive increase can be *too slow*



(slide from Presentation: "Congestion Control on High-Speed Networks", Injong Rhee, Lisong Xu, Slide 7)

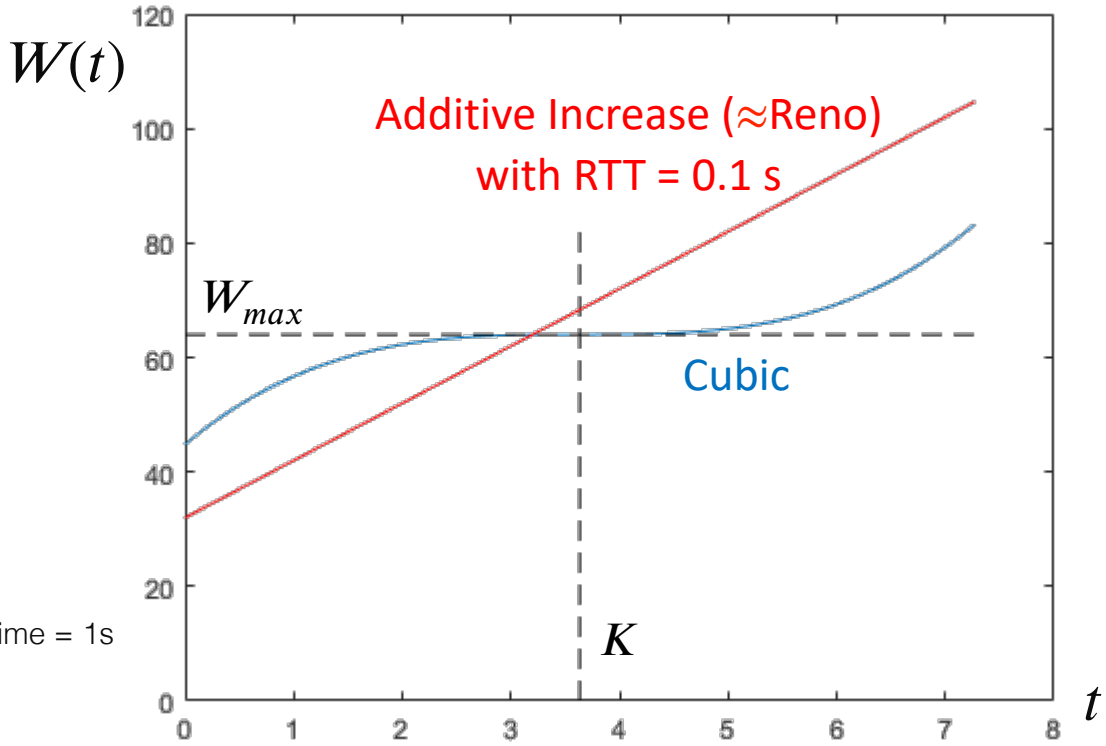
the figure ***assumes:*** *congestion avoidance implementing a strict additive increase of 1 MSS per RTT, losses are detected by fast retransmit, but the "fast recovery" phase is not used, MSS = 1250B, RTT = 100 msec.*

TCP Cubic modifies Reno

Why? increase TCP rate faster on LFNs
How? Cubic keeps the same slow start, fast recovery phases as Reno, but:

- during congestion avoidance, the increase is not additive but *cubic*
- multiplicative Decrease with factor ×0.7 (instead of ×0.5)

Say congestion avoidance is entered
at time $t_0 = 0$ and let
$W_{max} =$ value of cwnd, when loss is detected.
Let $W(t) = W_{max} + 0.4(t - K)^3$
with K such that $W(0) = 0.7\ W_{max}$.
Then the window increases like
$W(t)$ until a loss occurs again.



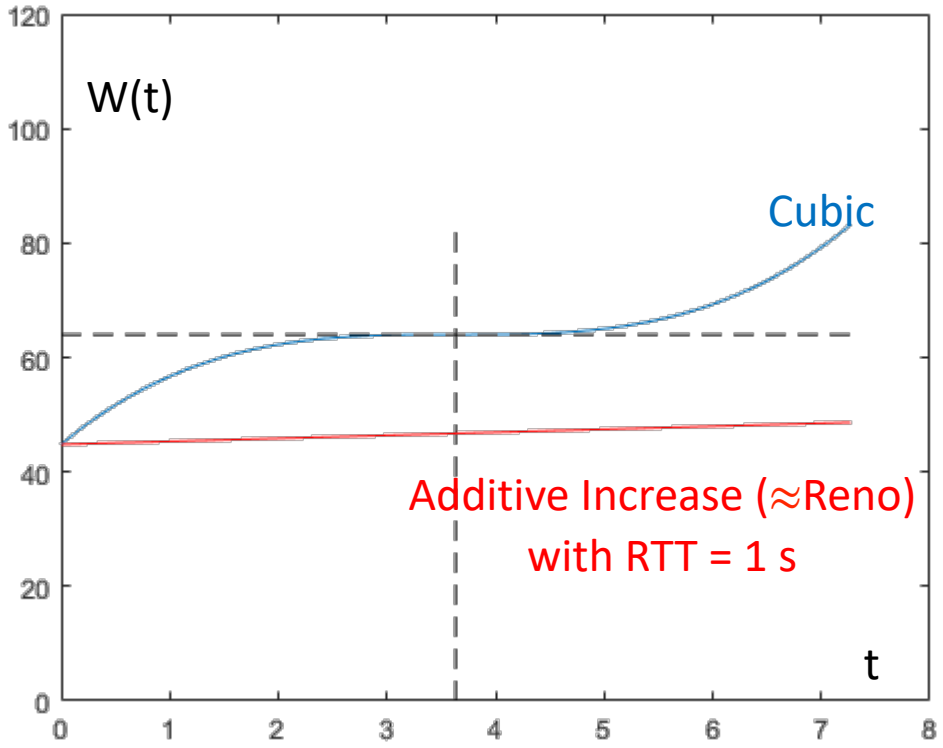
Units are: data = 1MSS; time = 1s

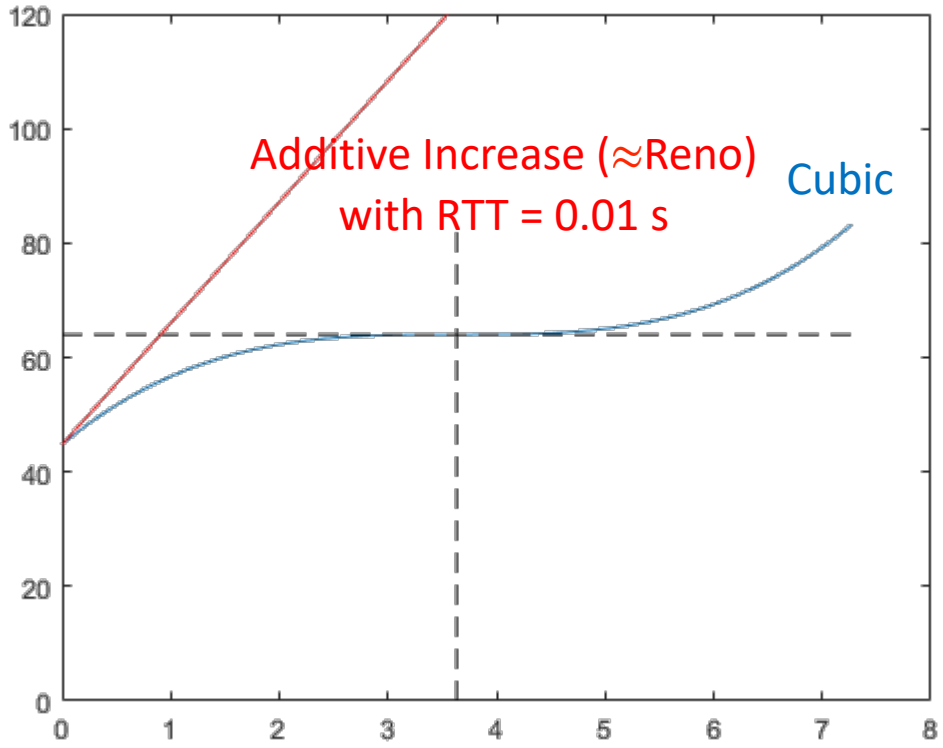
How does this compare to Reno?

Cubic increases window in concave way until reaches W_{max}, then increases in a convex way

$W(t)$ is *independent from RTT*, but

- it opens *faster* than Reno when RTT is large (long networks),
- but may be *slower* when RTT is small (non-LFNs)





Cubic's Window Increase

Cubic is always at least as fast as a hypothetical Reno (i.e. AIMD) with additive increase term r_{cubic} MSS per RTT (instead of 1) and multiplicative decrease $\beta_{cubic} = 0.7$.

Formally:

$$W_{CUBIC}(t) = \max\{W(t), W_{AIMD}(t)\},$$

where

$$W_{AIMD}(t) = W(0) + r_{cubic}\frac{t}{RTT} \text{ and}$$

r_{cubic} is computed s.t. this hypothetical Reno has the *same loss-throughput formula* as standard Reno: $\Rightarrow r_{cubic} = 3\frac{1-\beta_{cubic}}{1+\beta_{cubic}} = 0.529$

➡ *Cubic's throughput ≥ Reno's throughput*
with equality when RTT or bandwidth-delay product is small (i.e. when in non-LFNs)

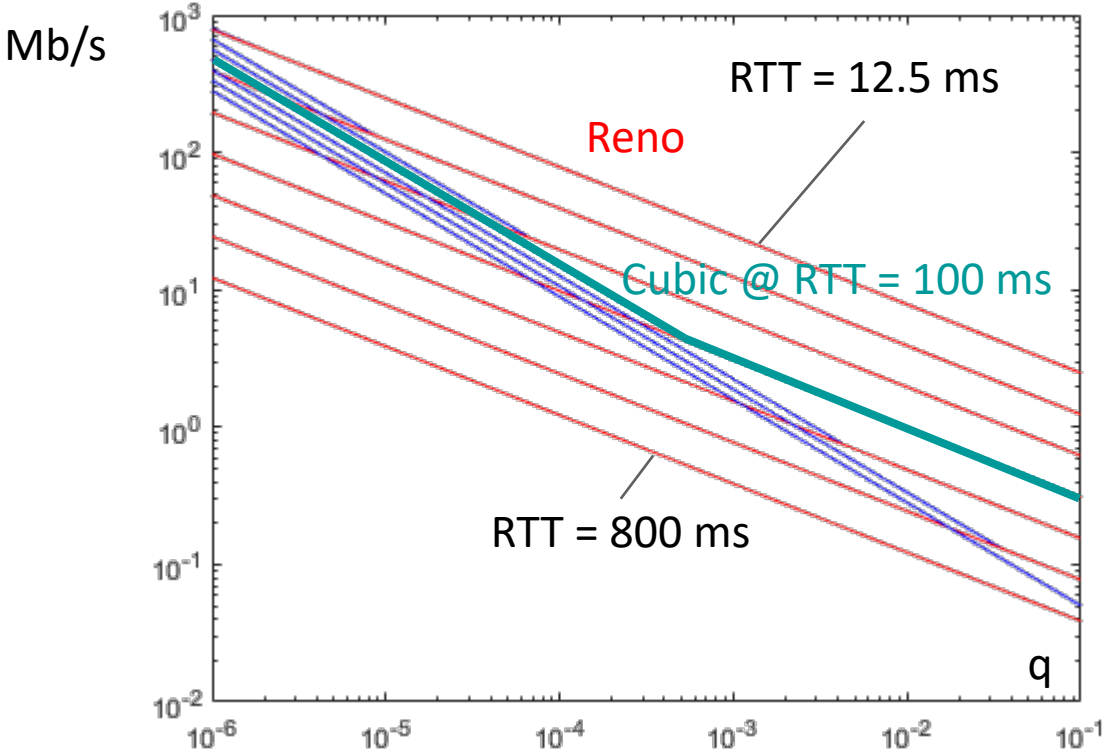
Cubic's Loss throughput formula

Given the ***same assumptions*** as for TCP Reno:

$$\theta \approx \max\left(\frac{1.054}{RTT^{0.25} q^{0.75}}, \frac{1.22}{RTT\sqrt{q}} \right)$$ in MSS per second.

So:

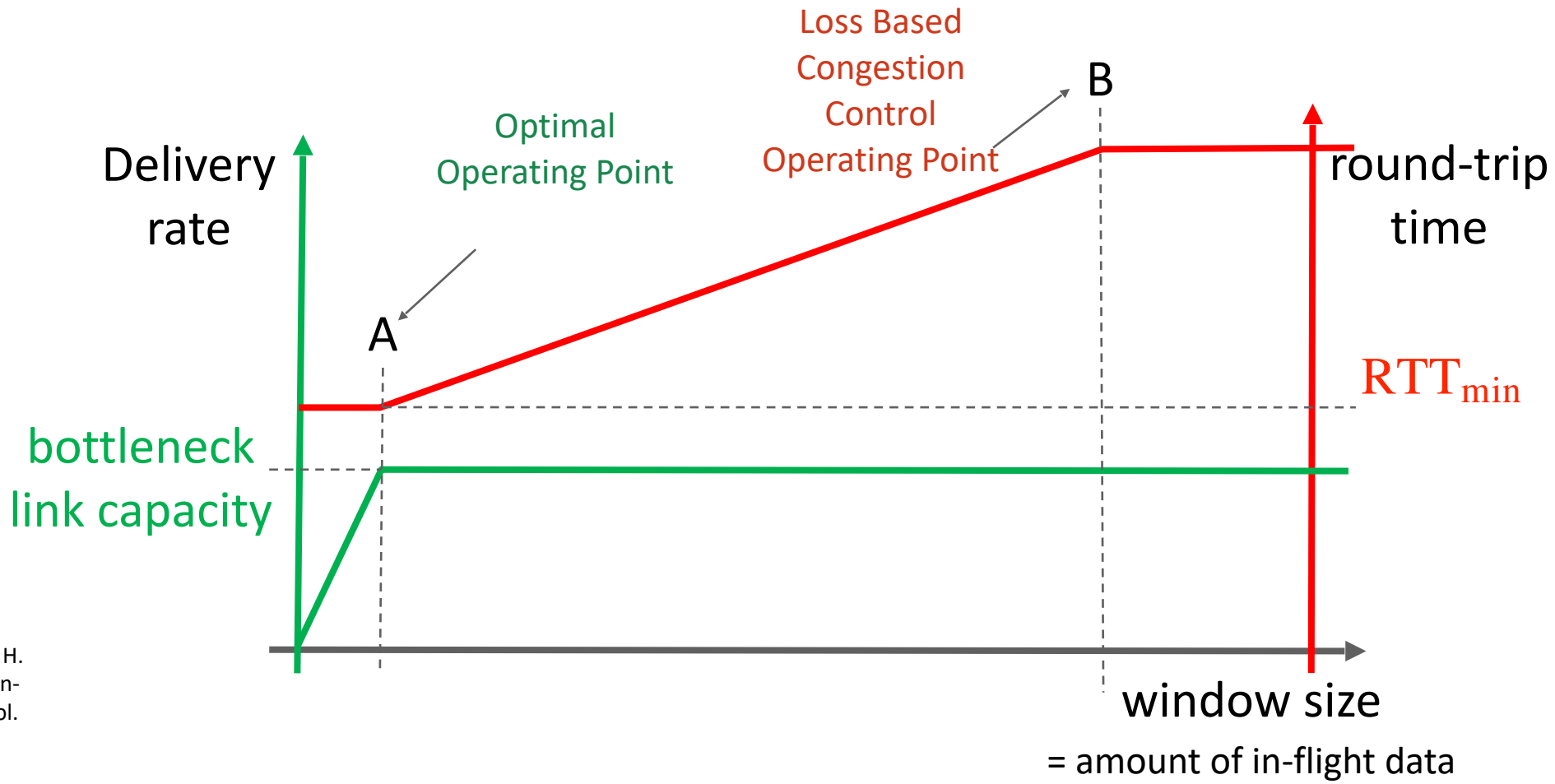
- Cubic's formula is same as Reno for small RTTs and small BW-delay products
- but a TCP Cubic connection gets *more throughput* than TCP Reno when bit-rate and RTT are large



Other details: computation of W_{max} uses a more complex mechanism called "fast convergence" - see Latest IETF Cubic RFC / Internet Draft or http://elixir.free-electrons.com/linux/latest/source/net/ipv4/tcp_cubic.c

8. Tackling the Bufferbloat Syndrome with ECN and AQM

Using loss as congestion feedback has a major drawback = *self-inflicted delay*: *increased latencies* and buffers are not well utilized due to *bufferbloat syndrome*.



From : N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," ACM Queue, vol. 14, no. 5, pp. 50:20–50:53, Oct. 2016.

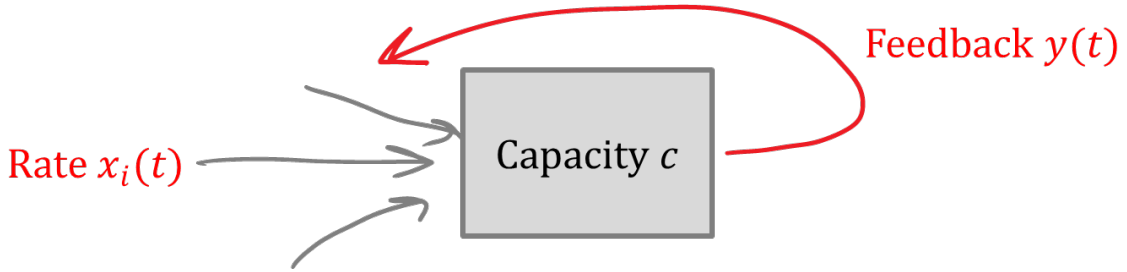
from [Hock et al, 2017] Mario Hock, Roland Bless, Martina Zitterbart, “Experimental Evaluation of BBR Congestion Control”, ICNP 2017:

The previous figure illustrates that if the amount of inflight data (i.e. the window size) is just large enough to fill the available bottleneck link capacity, the bottleneck link is fully utilized and the queuing delay is zero or close to zero. This is the optimal operating point (A), because the bottleneck link is already fully utilized at this point. If the amount of inflight data is increased any further, the bottleneck buffer gets filled with the excess data. The delivery rate, however, does not increase anymore. The data is not delivered any faster since the bottleneck does not serve packets any faster and the throughput stays the same for the sender: the amount of inflight data is larger, but the round-trip time increases by the corresponding amount. Excess data in the buffer is useless for throughput gain and a queuing delay is caused that rises with an increasing amount of inflight data. Loss-based congestion controls shift the point of operation to (B) which implies an unnecessary high end-to-end delay, leading to “bufferbloat” in case the buffer sizes are large.

ECN - Explicit Congestion Notification…

…aims at avoiding bufferbloat

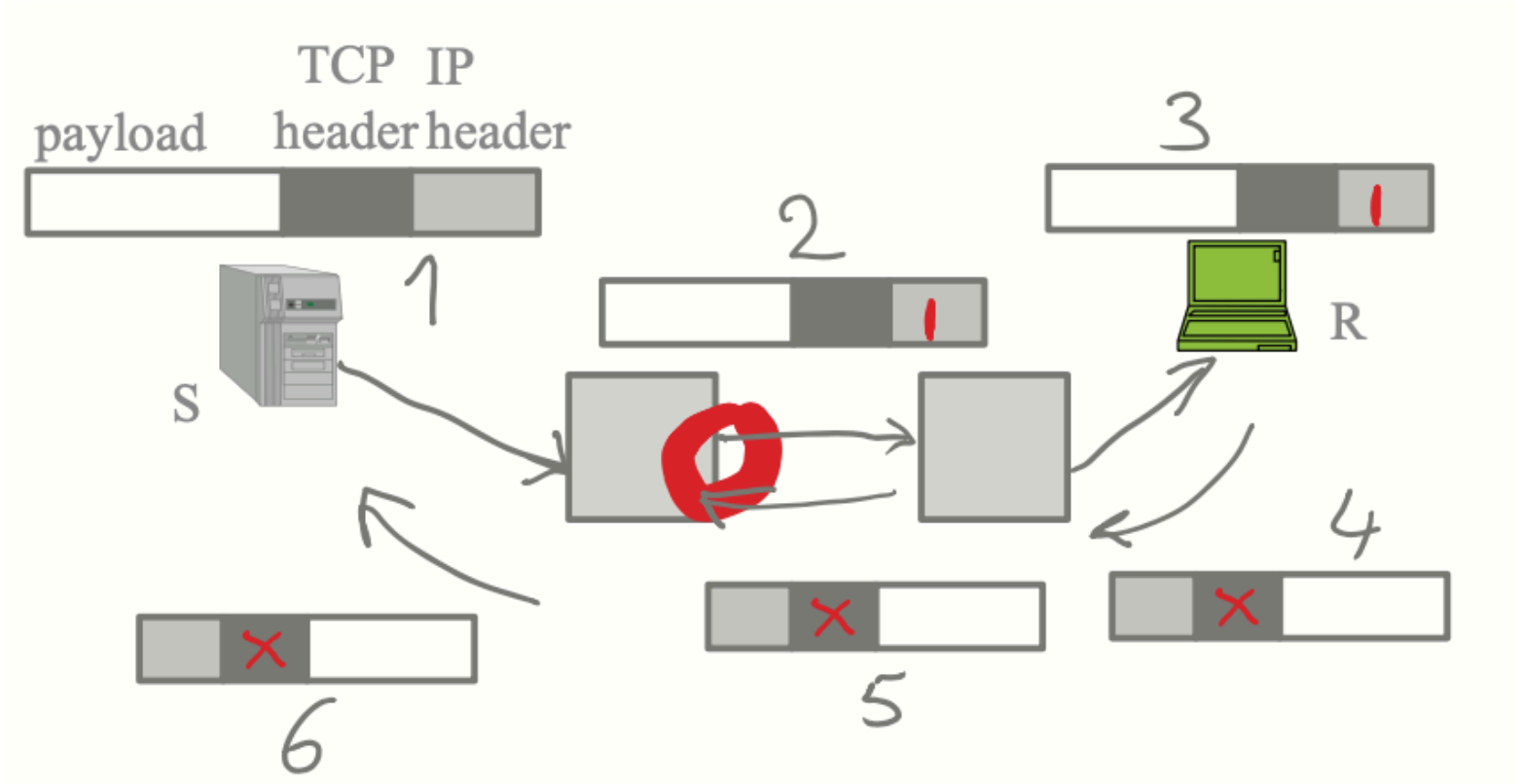
What? Network signals congestion without dropping packets (similarly to DECbit)



How?

- IP router experiencing congestion *marks* packet instead of dropping
- TCP destination *echoes back* the mark to the source
- TCP source *interprets* an echoed marked packet as if there was a loss detected by fast retransmit

Example



After 6: Source applies multiplicative decrease to cwnd, as if there was a loss detected by fast retransmit

Steps in the previous slide:

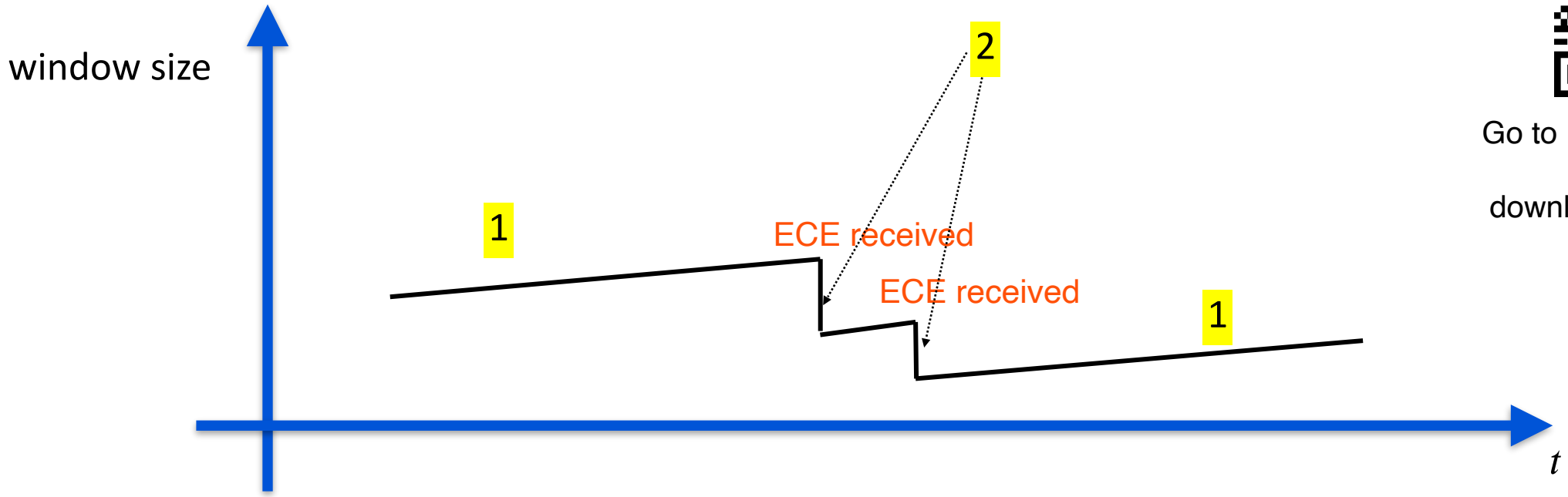
1. S sends a packet using TCP
2. Packet is received at congested router buffer; router marks the Congestion Experienced (CE) bit in IP header
3. Receiver sees CE in received packet and set the ECN Echo (ECE) flag in the TCP header of packets sent in the reverse direction
4. Packets with ECE are forwarded towards the source
5. Packets with ECE are forwarded towards the source
6. Packets with ECE are received by source.
7. Source applies multiplicative decrease of the congestion window.

   Source sets the Congestion Window Reduced (CWR) flag in TCP header.

   The receiver continues to set the ECE flag until it receives a packet with CWR set.

   Multiplicative decrease is applied *only once per window of data* (typically, multiple packets are received with ECE set inside one window of data).

Assume TCP with ECN is used and there is no packet loss. Put correct labels…



Go to web.speakup.info
or
download speakup app
Join room
46045

A. 1= CA, 2 = SS
B. 1 = SS, 2 = MD
C. 1 = CA, 2 = MD
D. I don't know

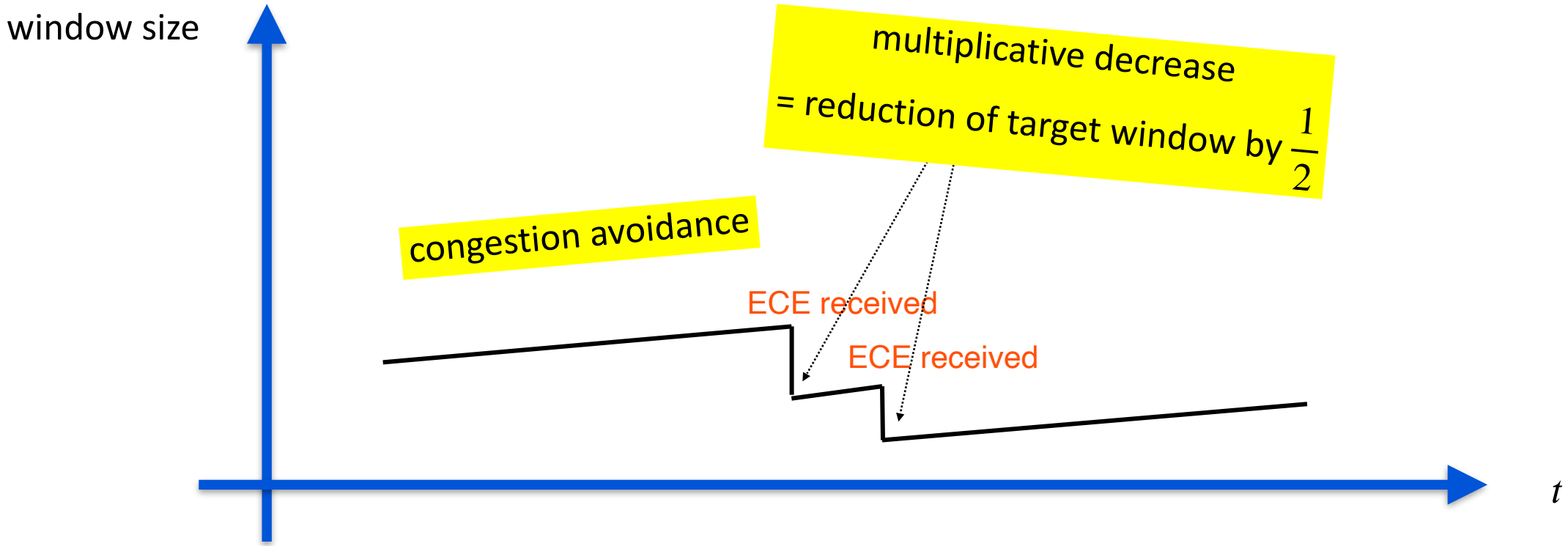
CA: congestion avoidance

SS: slow start = multiplicative increase

MD: multiplicative decrease

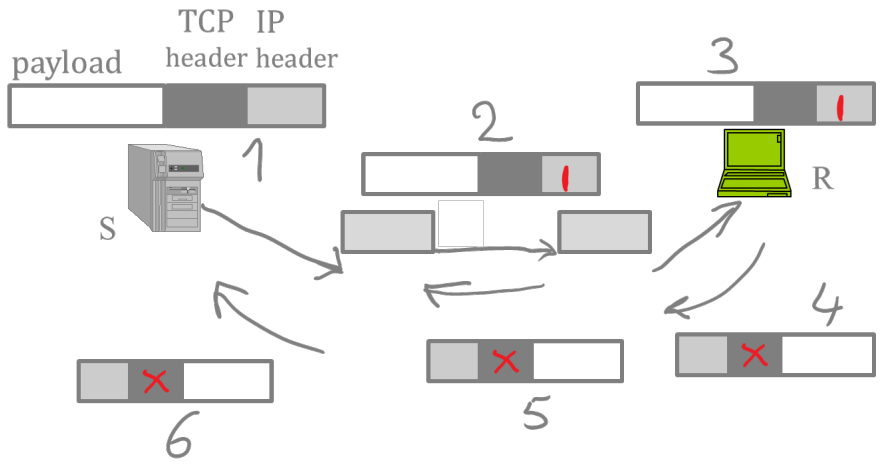
Solution

Answer C



Recall: Slow start's multiplicative increase results in an exponential growth of the cwnd.
So, no slow start phase is shown in this figure.

ECN flags in IP and TCP headers



2 bits in IP header, 4 possible codewords:

00 = non ECN Capable (non ECT)
01 or 10 = ECN capable ECT(0) and ECT(1)
historically used at random; today used to differentiate congestion control (TCP Cubic vs DCTCP)
11 = used by routers to signal that congestion is experienced (CE)
If congested, router marks ECT(0) or ECT(1) packets; but *discards non ECT packets*

2 bits in TCP header but as separate flags:

- ECE (echo) is set by R to inform S about congestion.
- CWR (congestion window reduced) set by S to inform R that ECE was received and R.
- When receiving ECE, S *reduces window only once* per RTT and sets the CWR flag in TCP headers. R sets ECE flag in all TCP headers until CWR is received or if a new CE packet is received.

ECN requires Active Queue Management

Why? decide when to mark a packet with ECN, and more generally, avoid buffer bloat syndrome

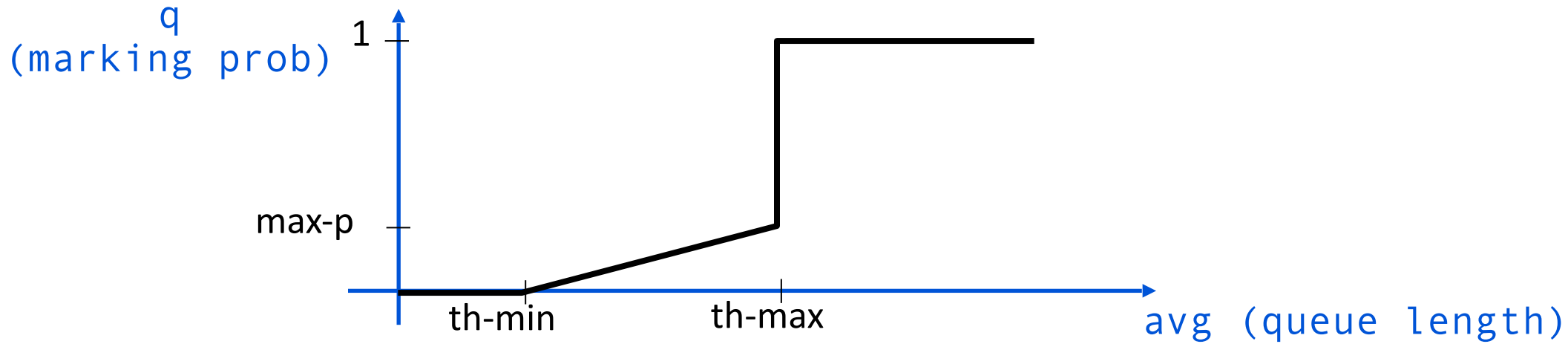
How? E.g. with a RED (Random Early Detection) queue:

- Queue estimates its average queue length

$$avg \leftarrow \alpha \times measured + (1 - \alpha) \times avg$$

- Incoming packet is marked with probability given by RED curve (see figure)

  a uniformization procedure is also applied to prevent bursts of marking events



See the difference from *passive* queue management = drop a packet only when queue is full = "Tail Drop"

But…Active Queue Management does not require ECN

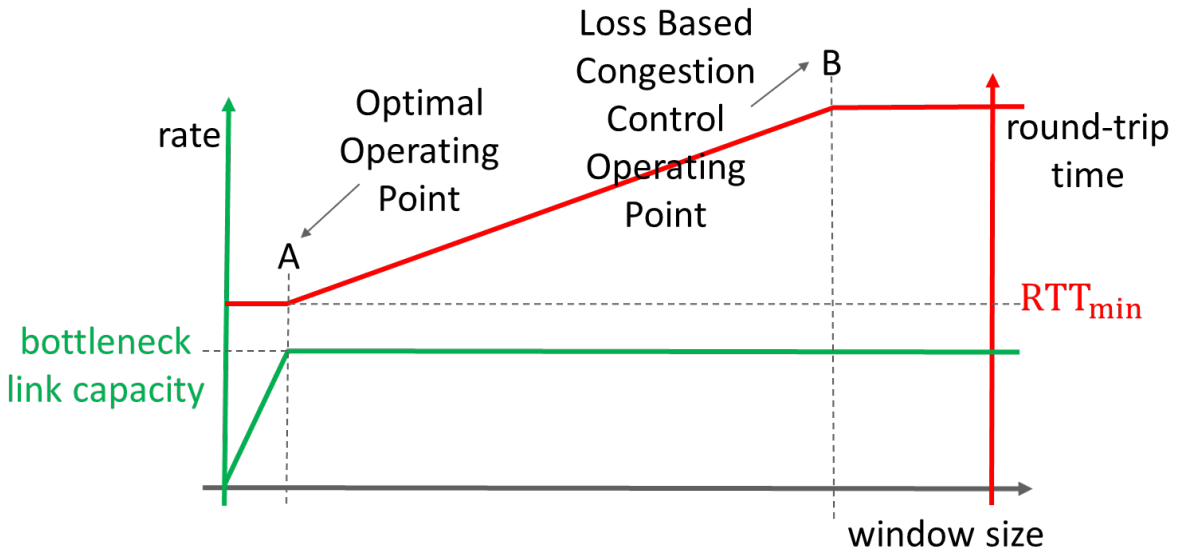
AQM can also be applied even if ECN is not supported

e.g. with RED, a packet is *dropped* with probability q computed by the RED curve

- packet may be discarded even if there is some space available!

Expected benefits in this case:

- *mitigate bufferbloat* – reduce latency
- *avoid irregular drop patterns*, as the drop probability affects all flows in the same way



In the context of packet dropping (instead of ECN), RED can be replaced by the more recent variant called CoDel (RFC 8289).

In a network where all flows use TCP with ECN and all routers support ECN, we expect that …

A. there is no packet loss
B. there is significantly less packet loss due to congestion in both switches and routers
C. there is significantly less packet loss due to congestion in routers
D. none of the above
E. I don't know

Go to web.speakup.info
or
download speakup app
Join room
46045

Solution

Answer C

We expect that routers (almost) do not drop packets due to congestion if all TCP sources use ECN

However there might be congestion losses in switches (especially the ones in large networks or Internet exchange points—IXPs), and there might be non-congestion losses (transmission errors)

Data Centers and congestion control

What is a data center?

a room with lots of racks of PCs and switches

where many distributed apps are running: e.g. youtube, CFF.ch, switchdrive, etc

What is special about data centers?

- most traffic is TCP
- very small latencies (10-100 μs)
- lots of bandwidth
- various traffic patterns coexist:
  - internal traffic (distributed computing) and
  - external traffic (user requests and their responses)
  - many short flows with low latency requirements (user queries, mapReduce communication)
  - some jumbo flows with huge volume (backup, synchronizations) may use an entire link

Given what you have learnt so far, what would you choose for TCP flows *inside* a data center ?

A. TCP Reno, no ECN no RED
B. TCP Reno and ECN
C. TCP Cubic, no ECN no RED
D. TCP Cubic and ECN
E. I don't know

Go to web.speakup.info
or
download speakup app
Join room
46045

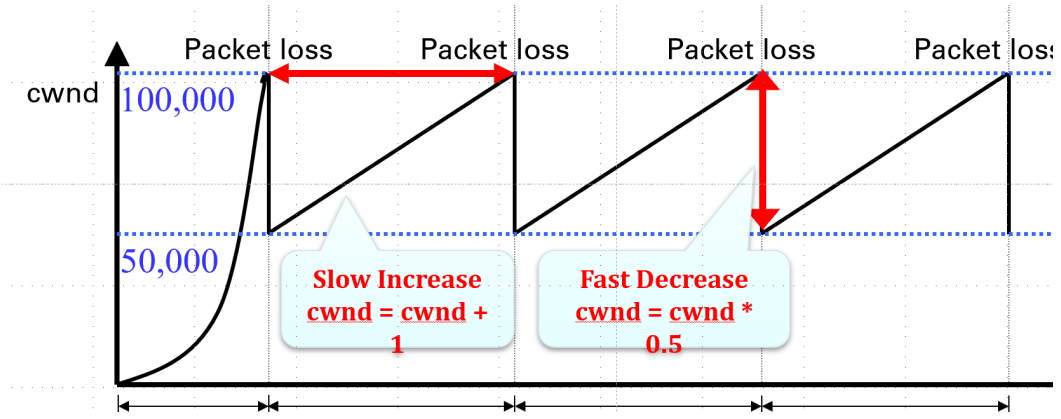
Solution

Answer D (also B could work)

- Cubic has better performance than Reno when bandwidth-delay product is large, which may occur in data centers.
  Also Cubic performs at least as good as Reno in *any* case.
- Without ECN there will be bufferbloat, which means high latency for short flows

Standard operation of ECN (e.g. with Reno or Cubic) still has *drawbacks for jumbo* flows in data center settings:

multiplicative decrease by 50% or 30% is still abrupt ⇒
throughput inefficiency



Data Center TCP

Why ? Improve performance for jumbo flows when ECN is used

How ?

Avoid brutal multiplicative decrease of 50% (of Reno) or 30% (of Cubic)

Instead, TCP source estimates *prob of congestion p* from ECN echoes

- ECN echo is modified so that:
  the proportion of CE marked ACKs $\approx$ the probability of congestion p
- Multiplicative decrease is $\times \beta_{DCTCP} = \left(1 - \frac{p}{2}\right)$

In a data center: two large TCP flows compete for a bottleneck link; one uses DCTCP, the other uses Cubic/ECN. Both have same RTT.

A. Both get roughly the same throughput
B. DCTCP gets much more throughput
C. Cubic gets much more throughput
D. I don't know

Go to web.speakup.info
or
download speakup app
Join room
46045

Solution

Answer B.

If latency is very small, Cubic with ECN has same throughput performance as Reno with ECN, i.e. same as AIMD with multiplicative decrease = $\times 0.5$ and window increase of 1 packet per RTT during congestion avoidance.

DCTCP is similar, in particular has same window increase, but with multiplicative decrease = $\times \left(1 - \frac{p}{2}\right)$ so the multiplicative decrease is always less.

DCTCP decreases less and increases the same, therefore it is *more aggressive*.

In other words, DCTCP competes *unfairly* with other TCPs; this is why it *cannot be deployed outside* data centers (or other controlled environments).

Inside data centers, care must be given to separate the DCTCP flows (i.e. the internal flows) from other flows. This can be done with class-based queuing [see next].

9. Beyond Loss/ECN Based Congestion Control

TCP-BBR

Per Class Queuing

Evolution of Buffer Drain Time in the Internet

Buffer Drain Time = buffer capacity / link rate

To keep buffer drain time constant, the product (memory speed × memory size)

should scale faster than link rate, which is technologically not feasible.

- Access network (1 Gb/s): buffer drain time is 10s of seconds = buffer is "large" w.r.t. rate
  ⇒ Bufferbloat unless ECN is used

But

- In internet core links (100 Gb/s, 1 Tb/s):

  buffer drain time decreases, is now fraction of ms, much less than RTT = buffer is "small"
  ⇒ impossible to react correctly within round trip time

  ⇒ feedback control may be inadequate

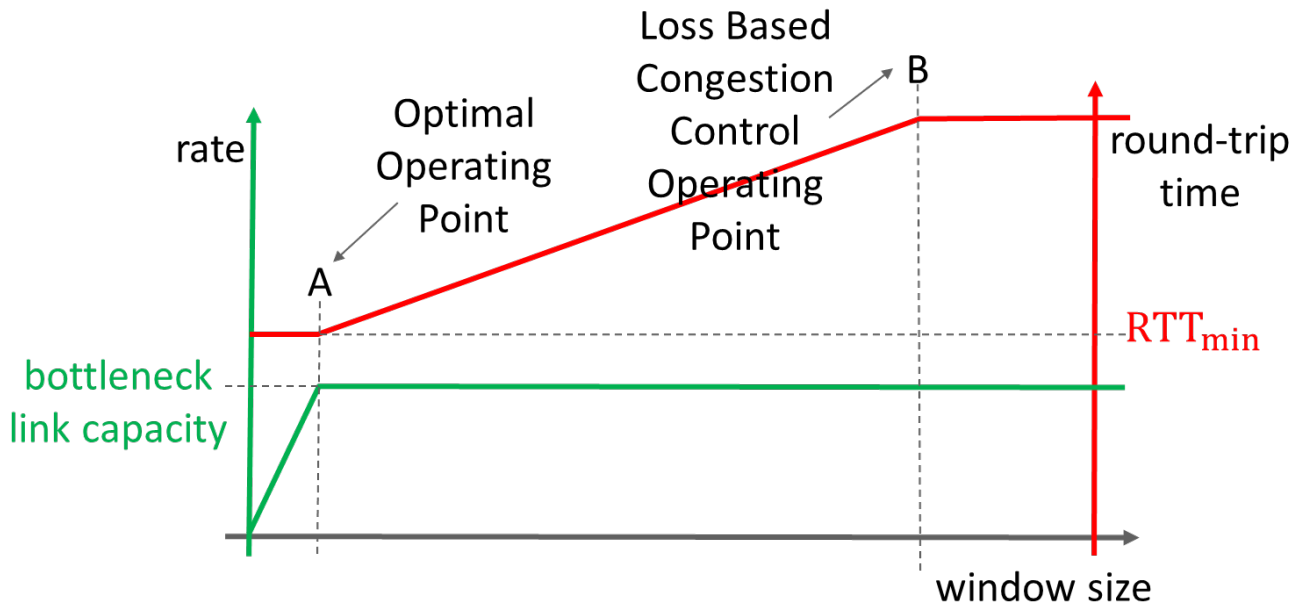
TCP-BBR
Bottleneck Bandwidth and RTT

TCP-BBR published by Google in 2016 [Caldwell et al 2016]

What ? Avoid per packet feedback, target *maximum throughput* with *minimal delay*

How ? BBR-TCP source:

1. estimates the bottleneck bandwidth and the min RTT separately
2. controls directly the rate (not the window) using *pacing* (= implementing a packet spacer) that tries to keep amount of inflight data close to `bottleneck bandwidth × minRTT` (optimal operating point)



N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," ACM Queue, vol. 14, no. 5, pp. 50:20–50:53, Oct. 2016

BBRv1 operation — simplified

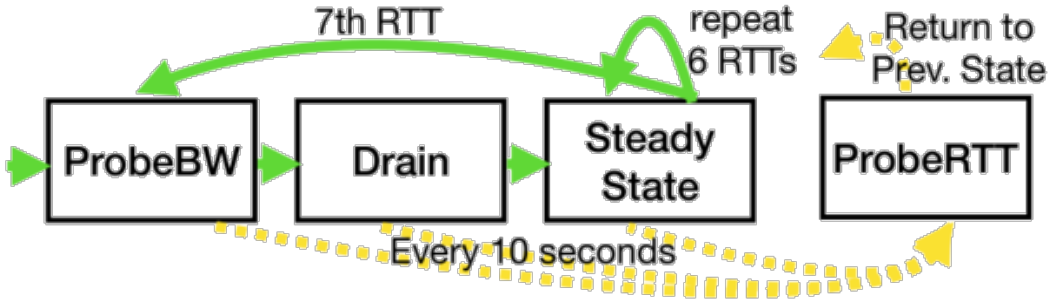


Figure from:
Ware, R., Mukerjee, M.K., Seshan, S. and Sherry, J., 2019, October. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the internet measurement conference* (pp. 137-143).

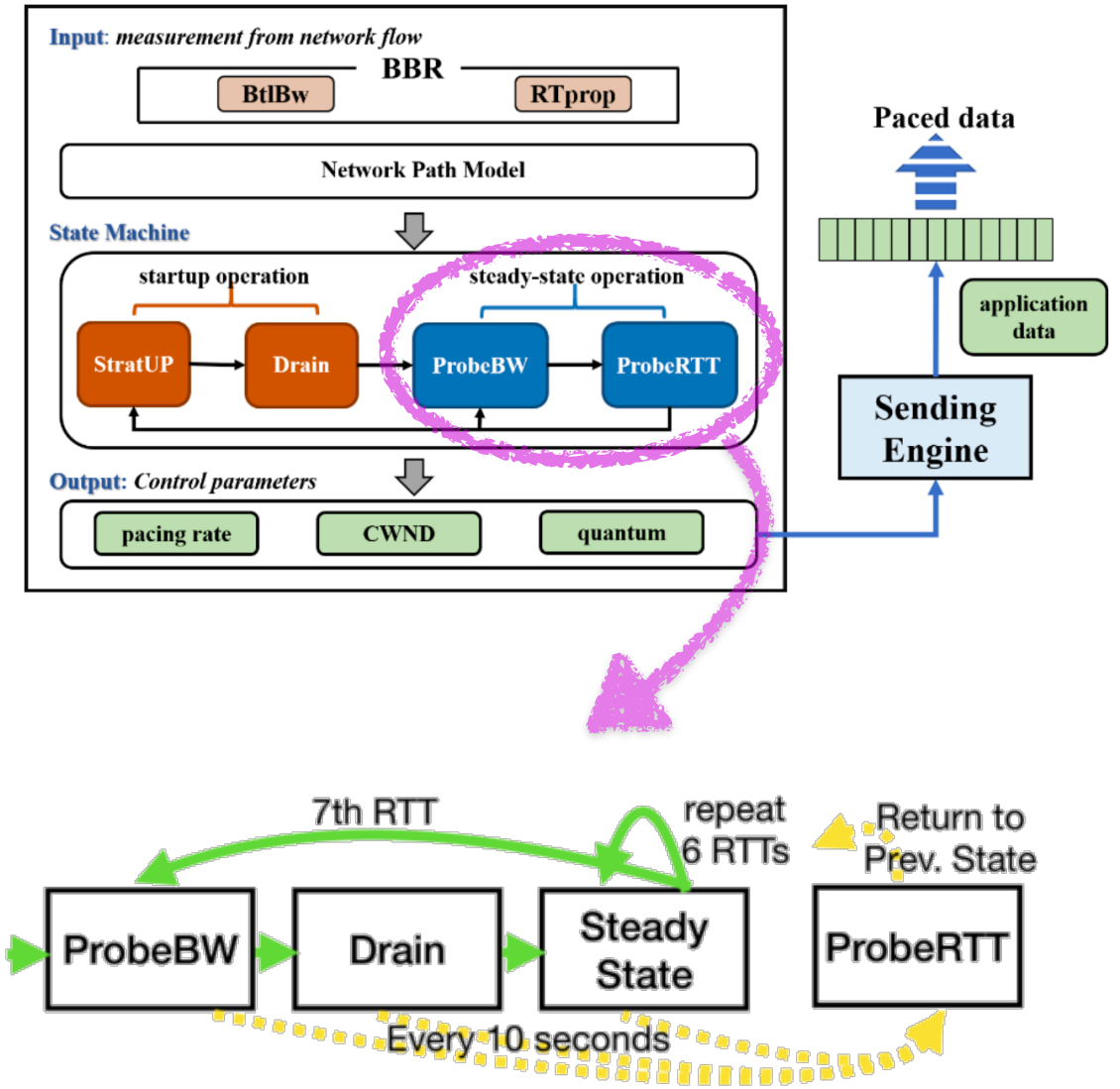
- source views network as a single link (the bottleneck link)
- estimates min RTT by taking the min over the last 10 sec = RTprop
- estimates bottleneck rate (bandwidth); b_r= max of delivery rate over last 10 RTTs; delivery rate = amount of ACKed data per Δt
- sends data at rate $b_r \times c(t)$
  where $c(t) = 1.25; 0.75, \ ; 1; 1; 1; 1; 1; 1$ is the pacing gain; i.e., c(t) is 1.25 during one RTprop, then 0.75 during one RTprop, then 1 during 6 RTprops ("probe bandwidth" followed by "drain excess" followed by steady state)
- if no new RTprop value for 10 seconds, the source enters Probe RTT state: sends only 4 packets to drain any possible queue and get a real estimation of the RTprop
- for safety, max data in flight is limited to $2 \times b_r \times RTT_{est}$ and by the offered window
- there is also a startup phase (similar to CUBIC and Reno) with exponential increase of rate
- no reaction to losses or ECN

…BBRv1 in more detail

1) Overview: The main objective of BBR is to ensure that the bottleneck remains saturated but not congested, resulting in maximum throughput with minimal delay. Therefore, BBR estimates bandwidth as maximum observed delivery rate BtlBw and propagation delay RTprop as minimum observed RTT over certain intervals. Both values cannot be measured simultaneously, as probing for more bandwidth increases the delay through the creation of a queue at the bottleneck and vice-versa. Consequently, they are measured separately.

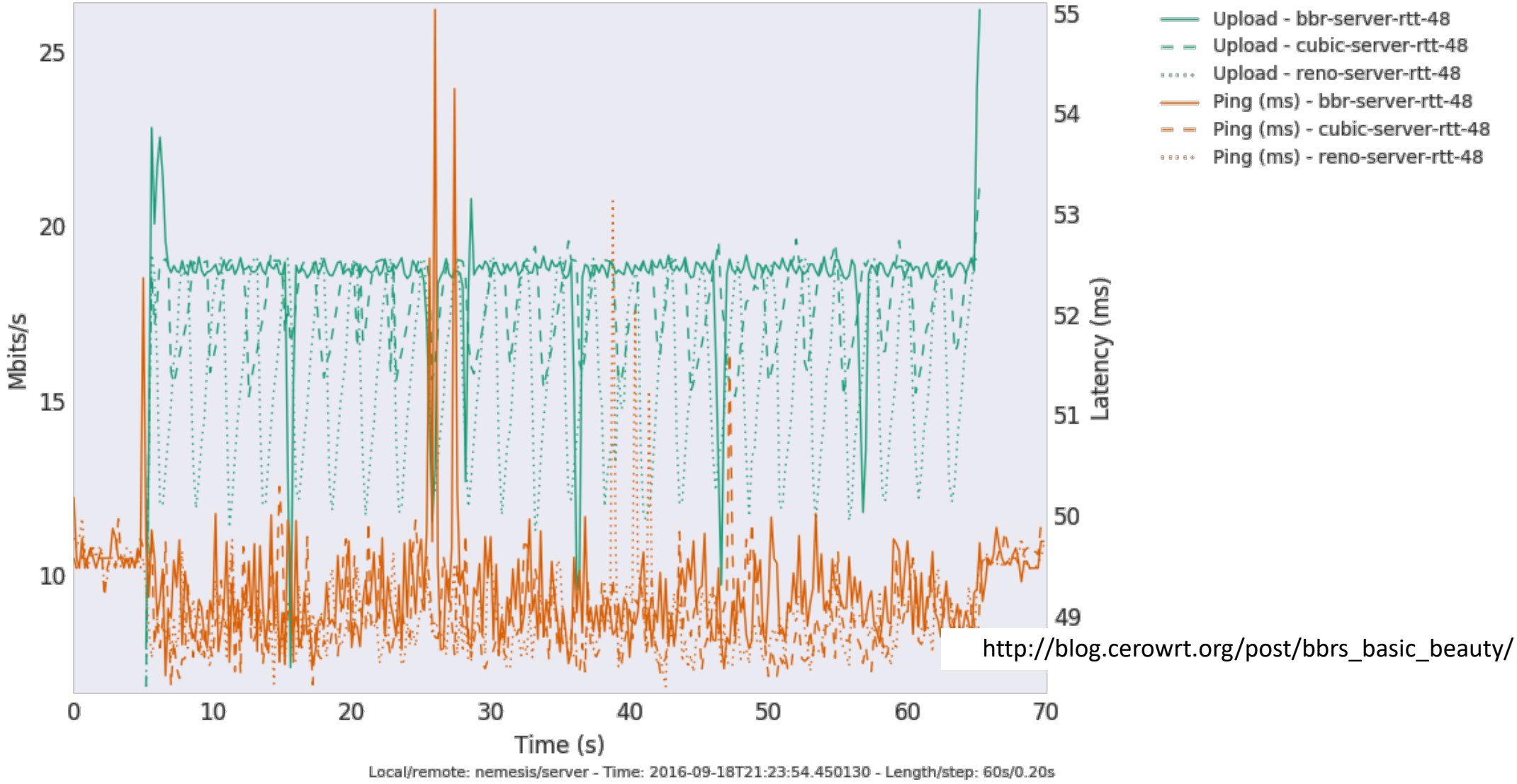
To control the amount of data sent, BBR uses pacing gain. This parameter, most of the time set to one, is multiplied with BtlBw to represent the actual sending rate.

2) Phases: The BBR algorithm has four different phases: **Startup**, **Drain**, **Probe Bandwidth**, and **Probe RTT**. The first phase adapts the exponential Startup behavior from CUBIC by doubling the sending rate with each round-trip. Once the measured bandwidth does not increase further, BBR assumes to have reached the bottleneck bandwidth. Since this observation is delayed by one RTT, a queue was already created at the bottleneck. BBR tries to Drain it by temporarily reducing the pacing gain. Afterwards, BBR enters the Probe Bandwidth phase in which it probes for more available bandwidth. This is performed in eight cycles, each lasting RTprop: First, pacing gain is set to 1.25, probing for more bandwidth, followed by 0.75 to drain created queues. For the remaining six cycles BBR sets the pacing gain to 1. BBR continuously samples the bandwidth and uses the maximum as BtlBw estimator, whereby values are valid for the timespan of ten RTprop. After not measuring a new RTprop value for ten seconds, BBR stops probing for bandwidth and enters the Probe RTT phase. During this phase the bandwidth is reduced to four packets to drain any possible queue and get a real estimation of the RTT. This phase is kept for 200 ms plus one RTT. If a new minimum value is measured, RTprop is updated and valid for ten seconds.



Performance of BBRv1

Google and other data center companies report
improvement on throughput (green curve), the latency measurement here is irrelevant and should be ignored



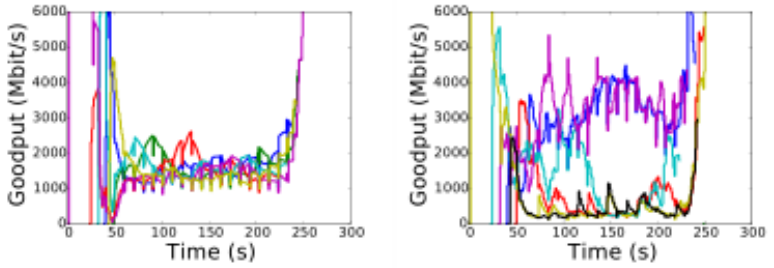
http://blog.cerowrt.org/post/bbrs_basic_beauty/

Performance of BBRv1

But…BBRv1 takes no feedback from network – no reaction to loss or ECN

- [Hock et al, 2017] find that BBRv1's estimated bottleneck bandwidth ignores how many flows are competing → fairness issues with:
  - BBR flows of different RTTs and
  - BBR versus other TCPs
- [Ware et al, 2019] find that in-flight cap, designed as a safety mechanism, is determinant

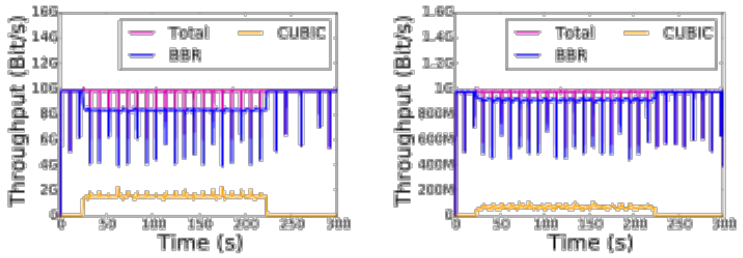
Google proposed BBRv2 and BBRv3 to address these and other shortcomings…



(a) 10 Gbit/s large buffer (b) 10 Gbit/s small buffer

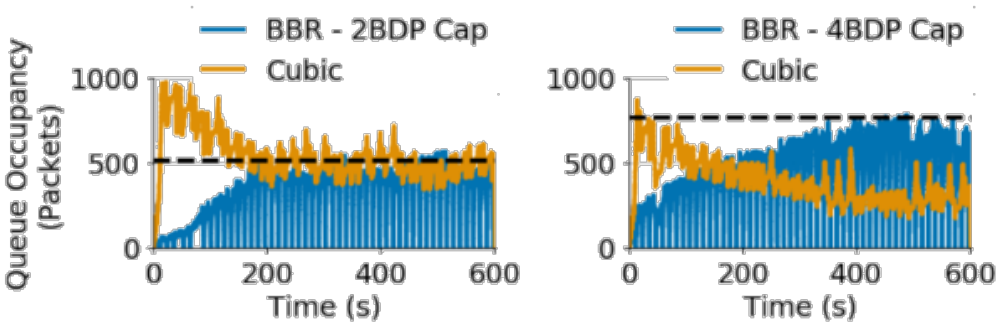
Fig. 7: Goodput of six BBR flows

Hock, M., Bless, R. and Zitterbart, M., 2017, October. Experimental evaluation of BBR congestion control. In 2017 IEEE 25th International Conference on Network Protocols (ICNP) (pp. 1-10). IEEE.



(a) 10 Gbit/s (b) 1 Gbit/s

Fig. 16: BBR vs. CUBIC TCP (small buffer)



(a) 2 BDP in-flight cap (b) 4 BDP in-flight cap

Figure 5: BBR vs Cubic in a 10Mbps×40ms testbed with a 32 BDP queue. Black dashed line is the model (5).

Ware, R., Mukerjee, M.K., Seshan, S. and Sherry, J., 2019, October. Modeling bbr's interactions with loss-based congestion control. In Proceedings of the internet measurement conference (pp. 137-143).

Per-class Queuing

Routers classify packets (using an access list)

- each class is guaranteed a dedicated *queue* and a weight —> hence a *rate*
- classes may exceed the guaranteed rate by *borrowing* from other classes if spare capacity exists

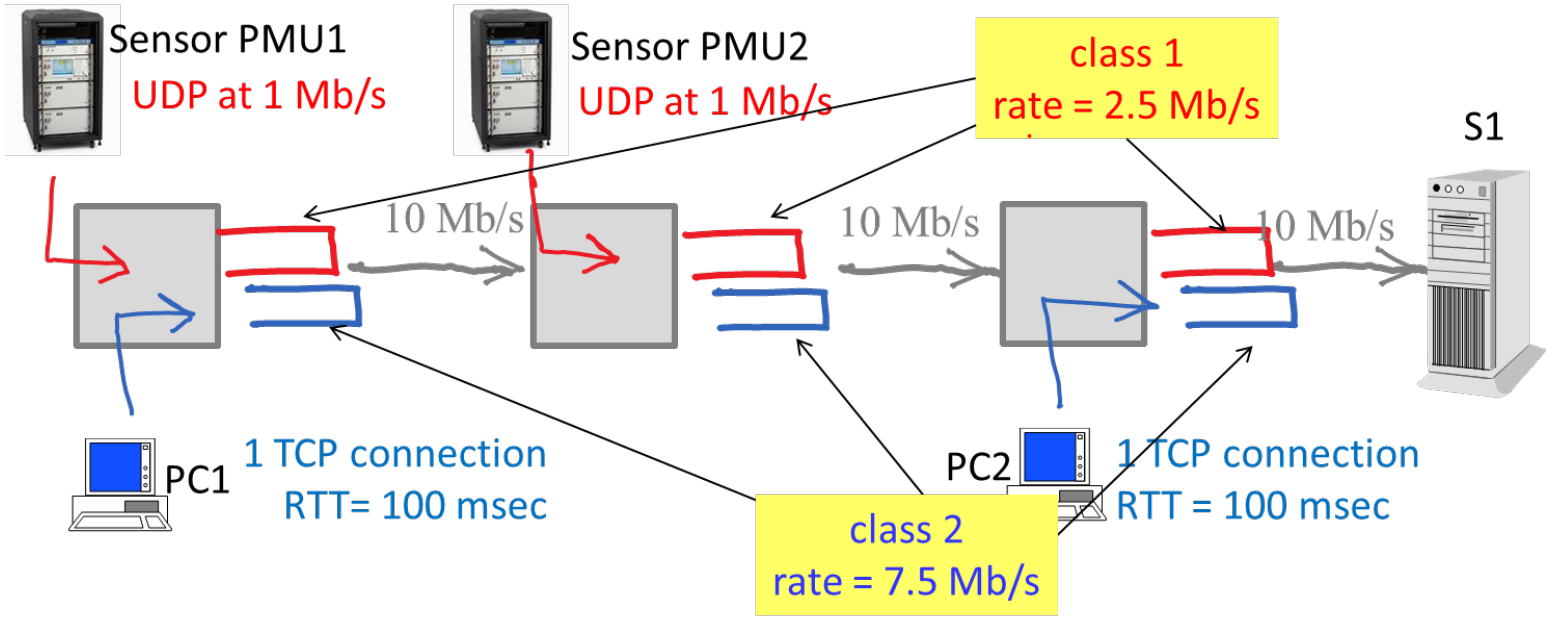
It is implemented in routers with dedicated queues for every class and a *scheduler* such as Weighted Round Robin (WRR) or Deficit Round Robin (DRR).

- WRR and DRR have one queue per class.
- At every round, queues are visited in sequence.
- WRR serves w_i packets of class i in one round. DRR serves q_i bits of class i in one round.

Used in

- enterprise or industrial networks to support non-congestion-controlled flows (e.g. real-time flows);
- provider networks to separate customers / isolate suspicious flows (network virtualization).

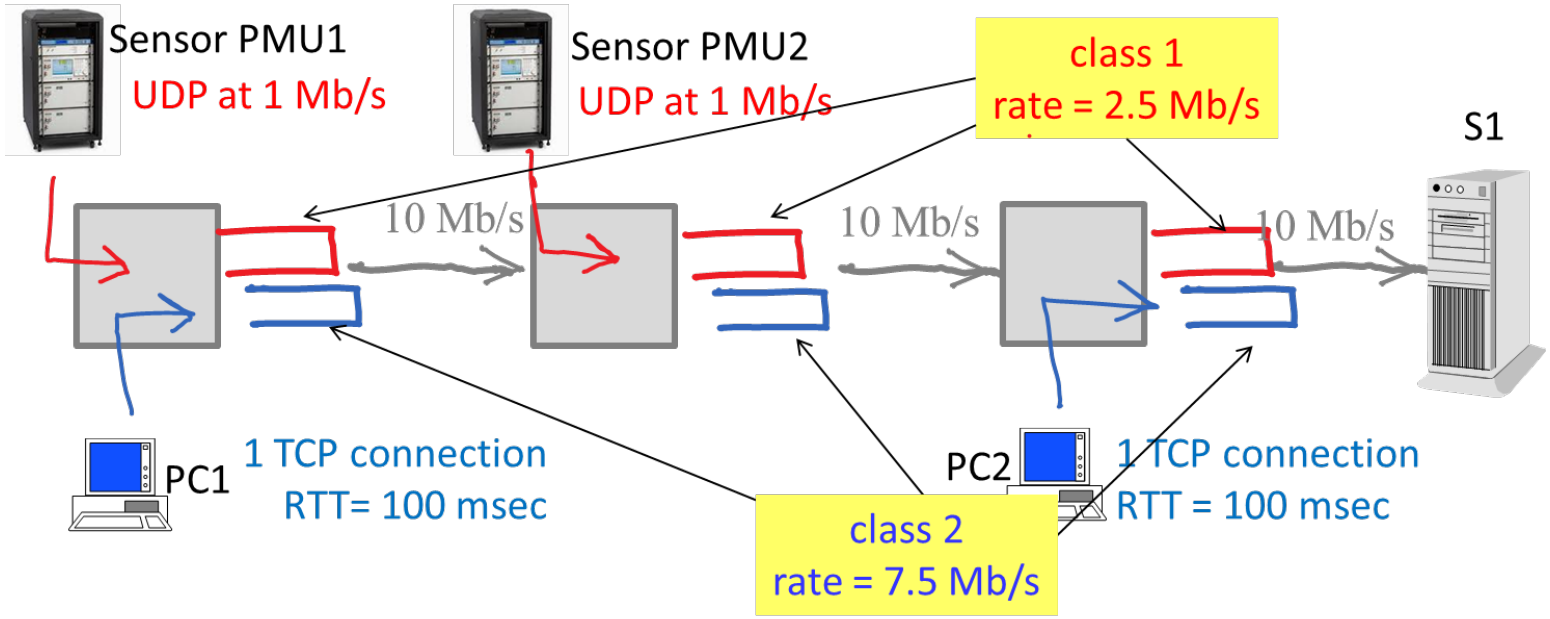
Example of Class-Based Queuing



Class 1 for PMUs (power measurement units) is guaranteed a rate of 2.5 Mb/s; it can exceed this rate by *borrowing* capacity available from the total 10 Mb/s if class 2 does not need it.

Class 2 for PCs is guaranteed a rate of 7.5 Mb/s; it can exceed this rate by *borrowing* capacity available from the total 10 Mb/s if class 1 does not need it.

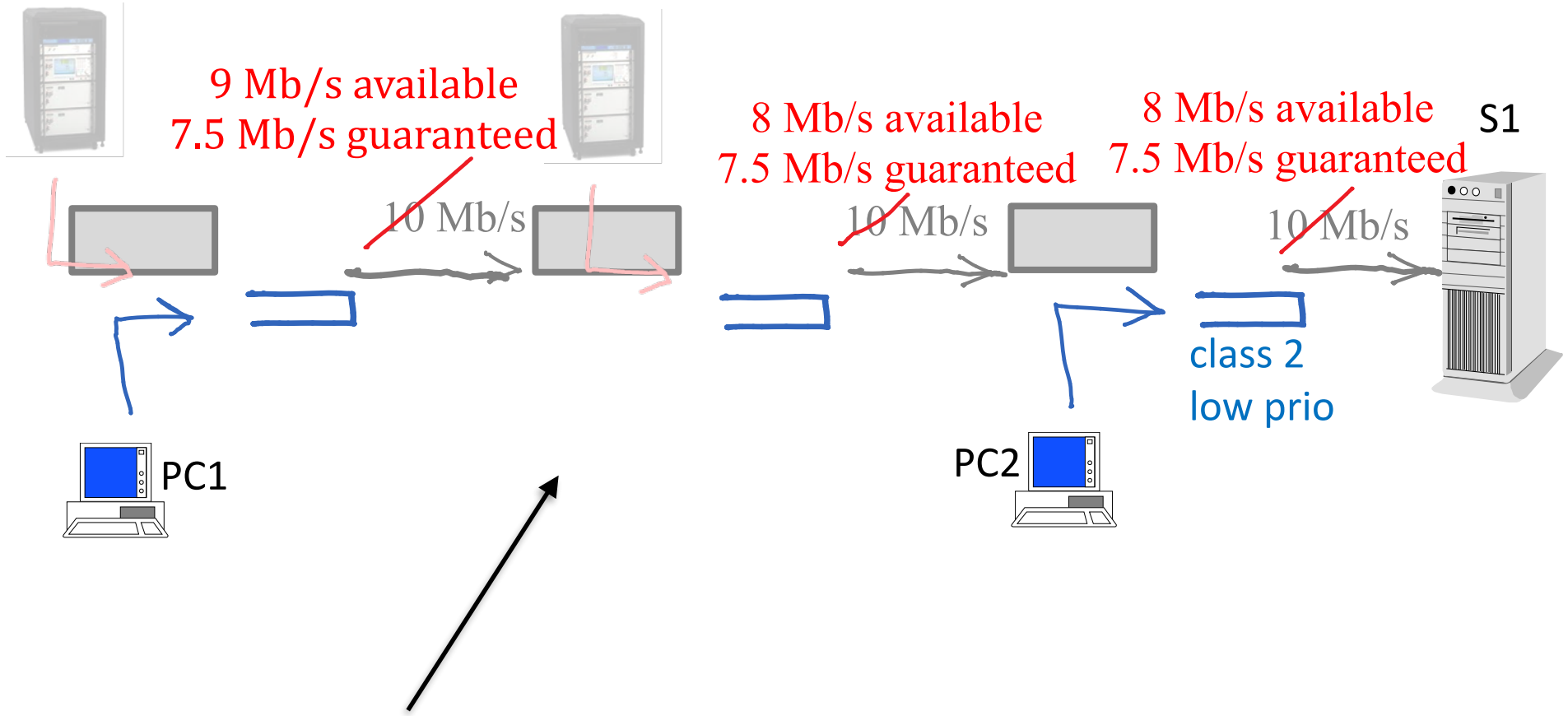
Suppose PMUs behave properly as expected.
Which rates will PC1 and PC2 achieve, if their RTTs are equal?



A. 5 Mb/s each
B. 4 Mb/s each
C. PC1: 5 Mb/s, PC2: 3 Mb/s
D. I don't know

Go to web.speakup.info
or
download speakup app
Join room
46045

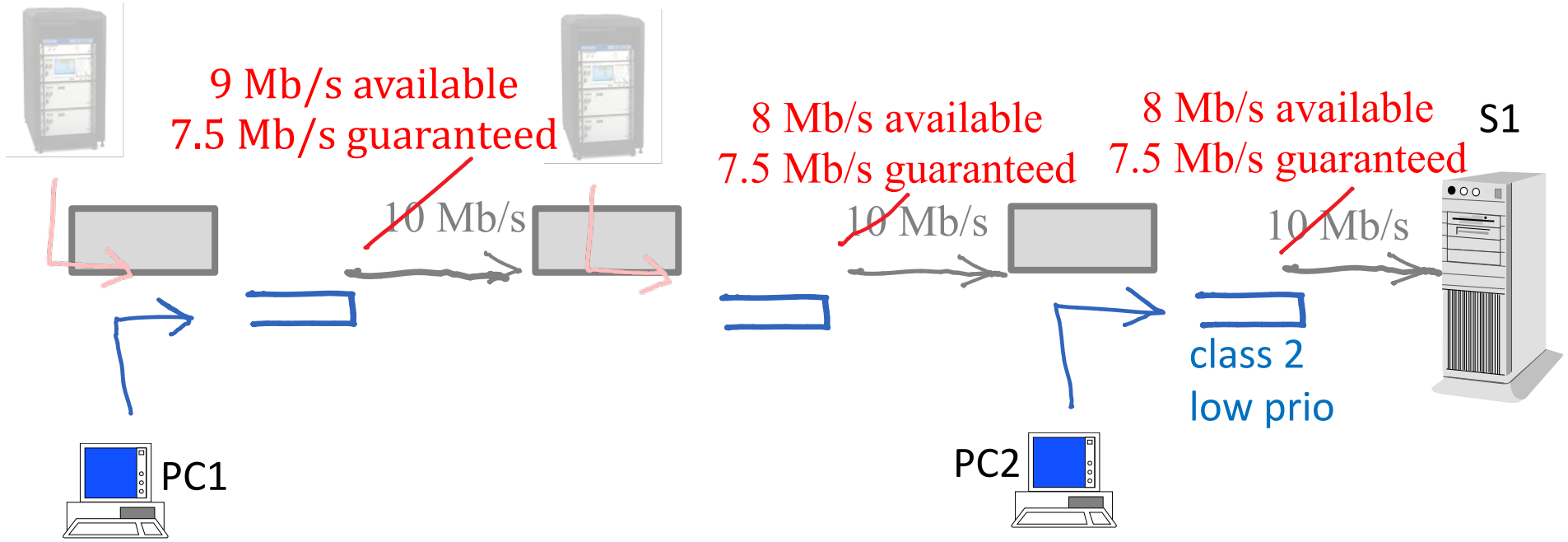
Solution



PC1 and PC2 see this network.

Since PMU1 and PMU2 stream at 1 Mb/s and class 2 may borrow the remaining capacity, the available capacities for class 2 are: 9 Mb/s, 8 Mb/s and 8 Mb/s.

Solution



TCP allocates rates x_1 and x_2 so as to maximize $U(x_1) + U(x_2)$ where U is the utility function of TCP; the function U is the same for PC1 and PC2 because RTTs are the same.

The constraints are $x_1 \leq 9$ Mb/s, $x_1 \leq 8$, $x_1 + x_2 \leq 8$ Mb/s.

Thus TCP solves a utility optimization problem: maximize $U(x_1) + U(x_2)$ subject to $x_1 + x_2 \leq 8$ Mb/s

By symmetry, $x_1 = x_2 = 4$ Mb/s

You can also check max-min fair allocation ($x_1 = x_2 = 4$ Mb/s) and proportionally fair allocation ($x_1 = x_2 = 4$ Mb/s) .

Answer B.

The Future of Congestion Control

In the past, most TCP versions have relied on loss or ECN as negative signal. Some versions also relied on delay only (TCP Vegas) or use delay as well as loss (PCC).

Congestion control today wants to also achieve “per-flow fairness”. But each flow may use a different congestion control algorithm.

- Is fairness achieved? Is every flow "TCP friendly”?
- Is the “flow” the right abstraction/fairness-actor?
- What are the alternatives?

Brown, L., Ananthanarayanan, G., Katz-Bassett, E., Krishnamurthy, A., Ratnasamy, S., Schapira, M. and Shenker, S., 2020, November. On the future of congestion control for the public internet. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (pp. 30-37).

Traffic isolation (e.g. with per-class traffic shapers or per-class queuing) is a possible future alternative; packet dropping/ECN marking becomes a function of the traffic aggregate/class a packet belongs to.

But does this comply with network neutrality regulations (= ISPs provide no competitive advantage to specific apps/services, either through pricing or QoS)? How could network neutrality be maintained?

Conclusion

Congestion control is in TCP or in QUIC (a form of congestion-controlled UDP).

Traditional TCP uses:

- the window to control the amount of traffic: additive increase or cubic (as long as no loss); multiplicative decrease (following a loss).
- loss as congestion signal.

Too much buffer is as bad as too little buffer—bufferbloat provokes large latency for interactive flows.

- ECN can avoid this – it replaces loss by an explicit congestion signal; but it is partly deployed in the Internet; although it is part of Data Center TCP.
- TCP-BBR aims at avoiding this by pacing traffic:
  it estimates the available bottleneck bandwidth and the min RTT
  and it tries to keep amount of inflight data close to bottleneck bandwidth × minRTT

Per-Class-based queuing can ***separate*** flows in enterprise networks or classes of flows in provider networks.