Congestion Control In The Internet

Part 2: Implementation

EPFL

Contents

6. TCP Reno (widely used until recently, part of most other algorithms)
7. TCP Cubic (widespread today in Linux servers)
8. ECN and AQM, DC-TCP (Microsoft and Linux servers in data centers)
9. New Directions—TCP-BBR (Google, Whatsapp, etc)

Congestion Control in the Internet was initially only in TCP

Why?

TCP already maintains an e2e connection —> easy to apply e2e congestion control

How? At a high level:

- a TCP source adjusts its "rate" by:
  - adjusting the *sliding window:* `W = min (cwnd, advertized_Wnd)`

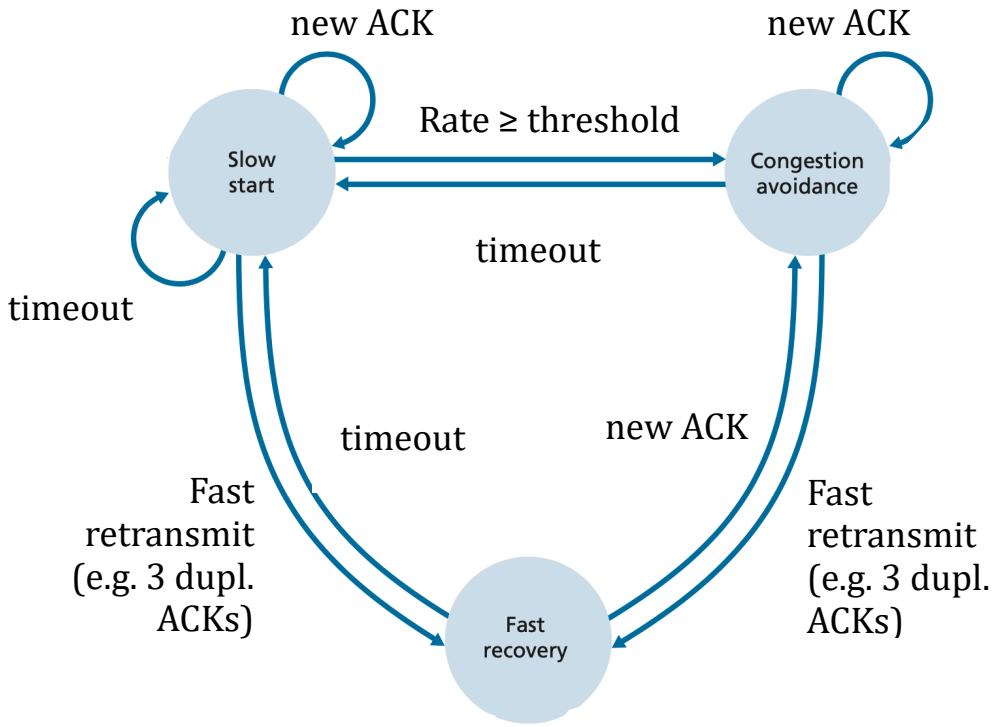
    based on: $rate \approx \frac{W}{RTT}$ (holding whenever there is no loss),
  - in response to *implicit* or *explicit feedback* from network (similarly to Decbit),

- this avoids congestion and ensures *some sort of* fairness

6. TCP Reno ≈ AIMD + Slow Start + implicit feedback

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

- 3 phases/states:
  - Slow Start (initial state) ≈ 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 = multiplicative decrease with factor $u_1 \approx 0.5$ after fast-retransmit event [see next]



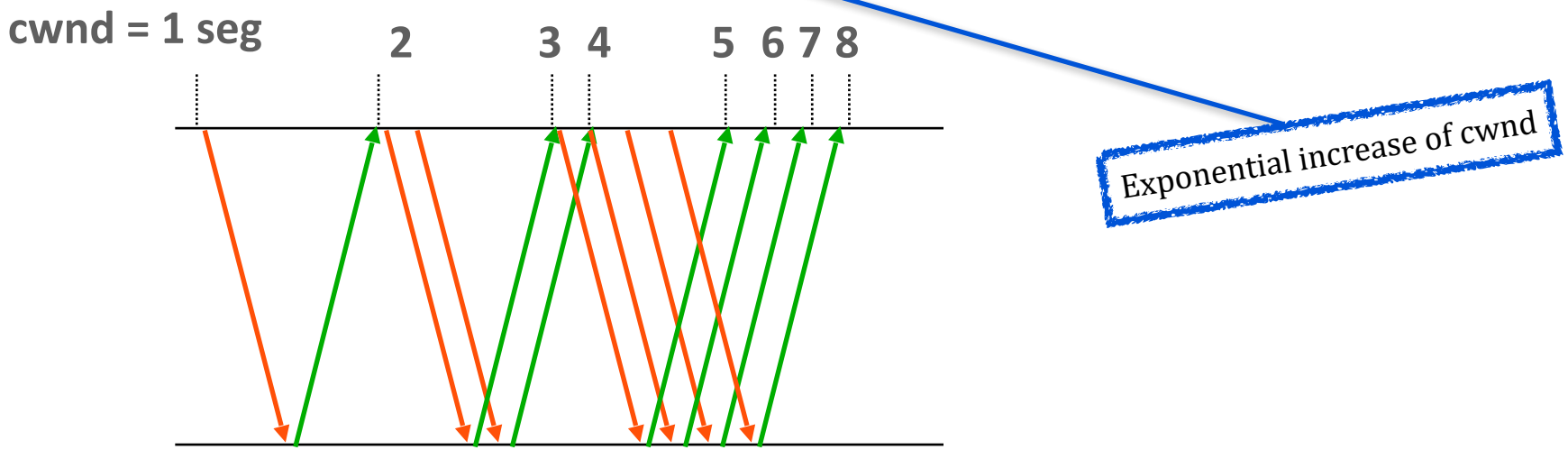
Slow Start by adjusting cwnd…

…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:

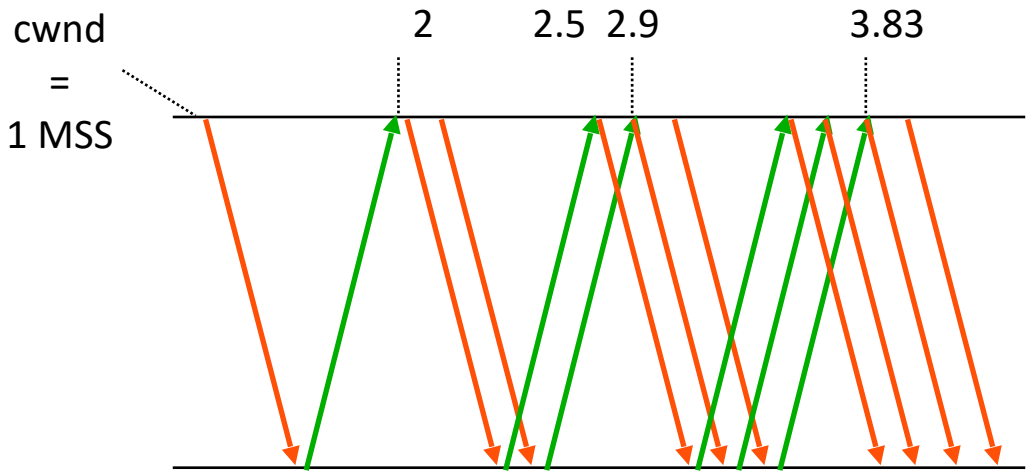
```
cwnd = cwnd + MSS (in bytes)
```

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



- when cwnd ≥ ssthresh, go to Congestion Avoidance phase

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,`

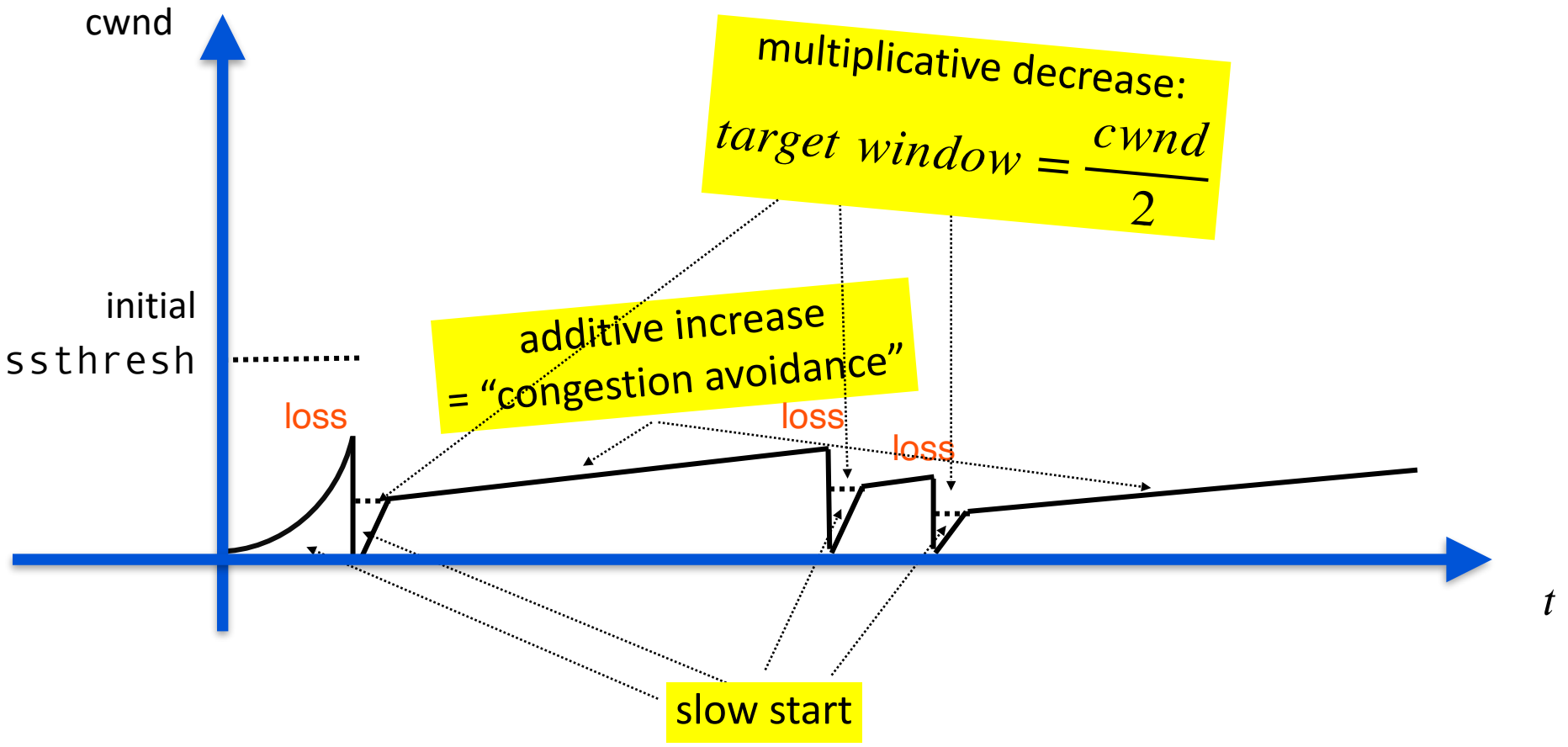
  slightly less than additive increase (< 1 MSS/RTT)

- 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:

- slow start is used initially and after every packet loss is 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 a multiplicative decrease of rate with $u_1 = 0.5$
- but halving the cwnd immediately is *not* a good approach;
  - formula "$rate \approx \frac{W}{RTT}$" is not true,
    when there is a single isolated packet loss;
  - source may even stop sending,
    if the first packet of a batch is lost

What?

- until loss is repaired, cwnd increases (beyond ssthresh) by 1MSS per duplicate ACK
- and only when loss is repaired, cwnd becomes half of its value before loss
- i.e. it's a *transient cwnd-increase phase*,
  used to keep sending segments

Algorithm:

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

```
ssthresh = 0.5 × 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 (artificial increase)
cwnd = min (cwnd, 64K)
```

If loss is repaired

```
cwnd = ssthresh
```

Goto Congestion Avoidance

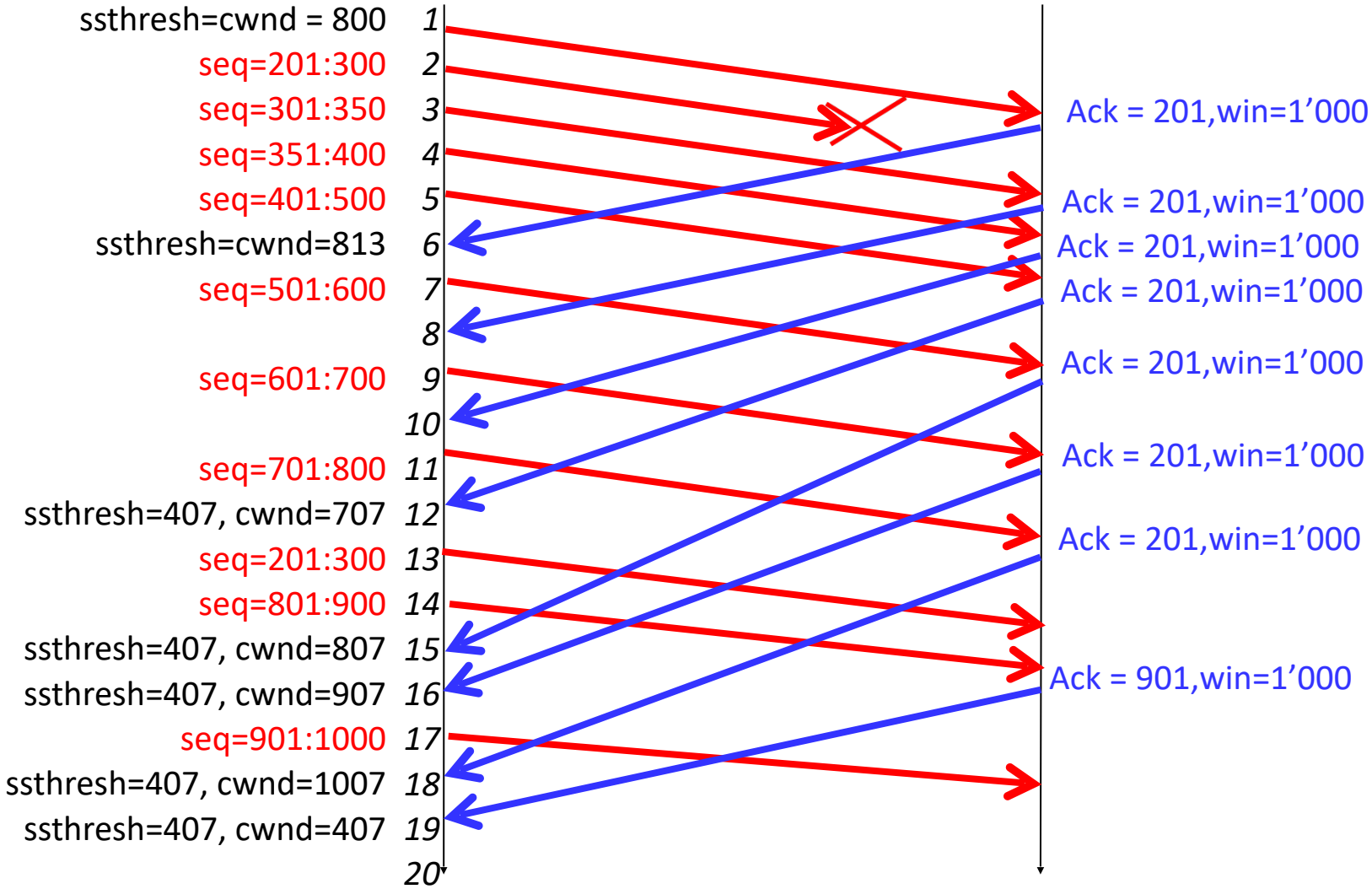
else (timeout)

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” phase. 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.

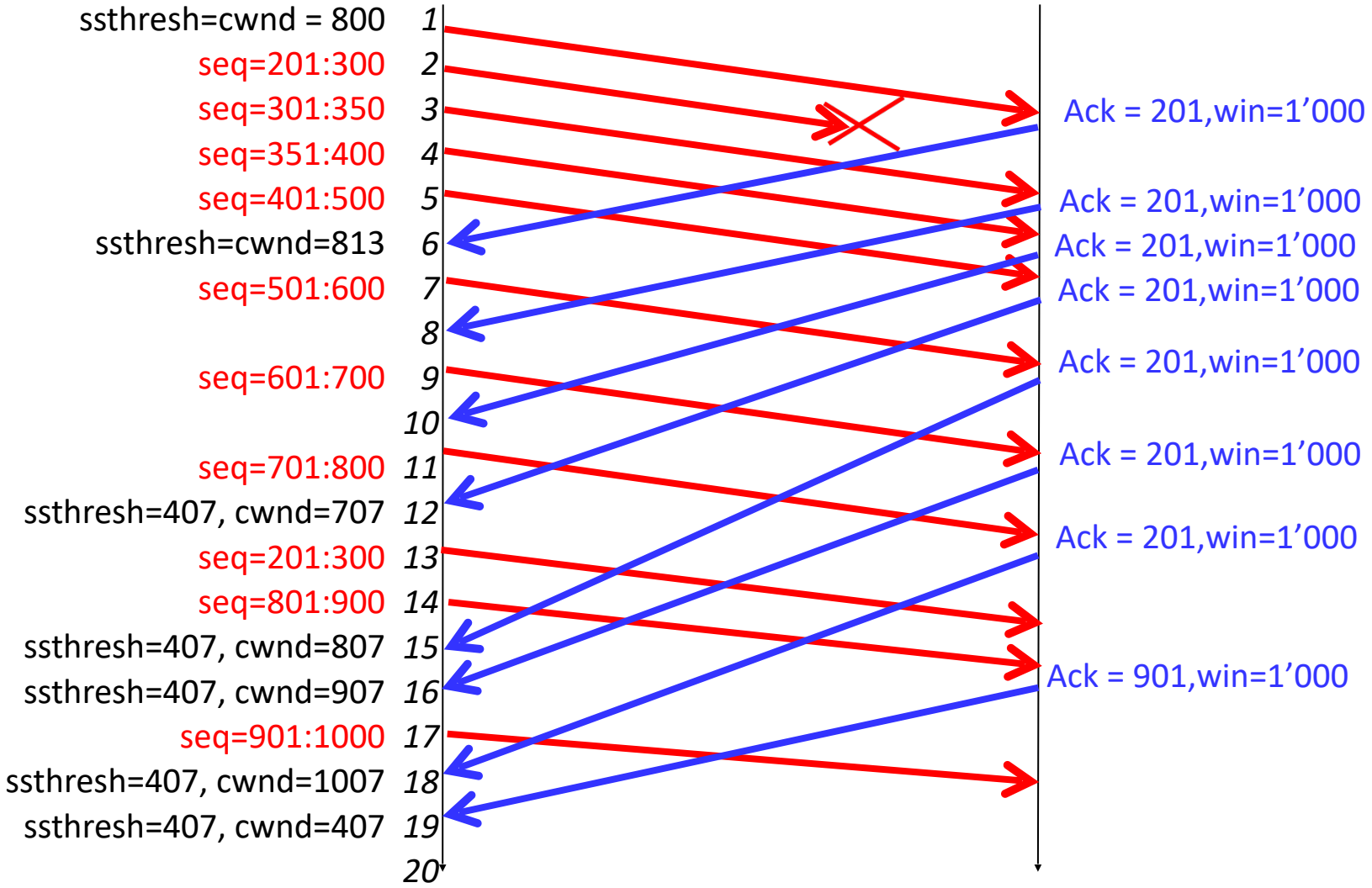
At time 12, its loss is detected by fast retransmit, i.e. reception of 3 duplicate acks. The sender goes into “fast recovery” mode. The target window is set to half the value of the congestion window; the congestion window is set to the target window plus 3 packets (one for each duplicate ack received).

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 87072

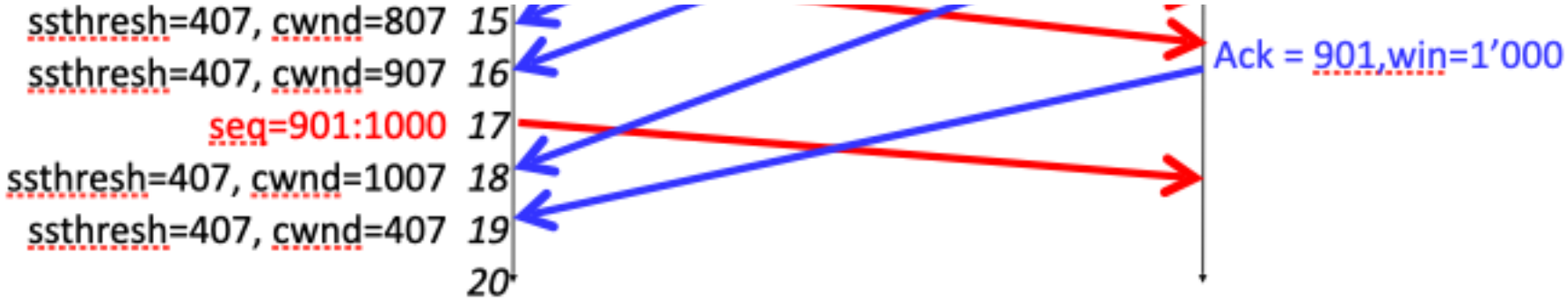
A. 1
B. 2
C. 3
D. 4
E. ≥ 5
F. 0
G. I don't know

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

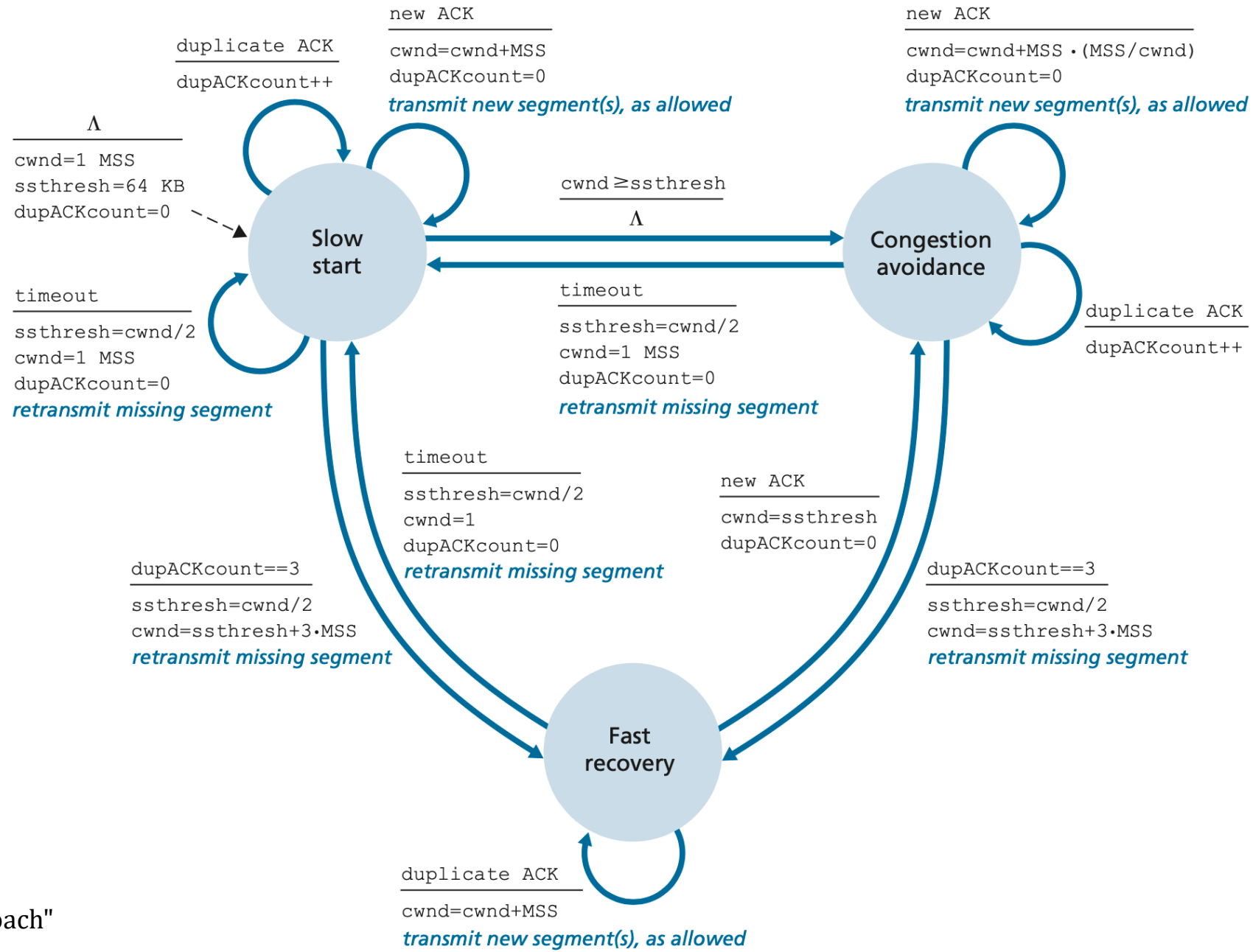


Figure from our textbook:
"Computer Networking: A top-down approach"
by J. Kurose and K. Ross

Assume a TCP flow goes over a WiFi link with high loss rate, where packets are lost despite WiFi (layer-2) 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 87072

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]

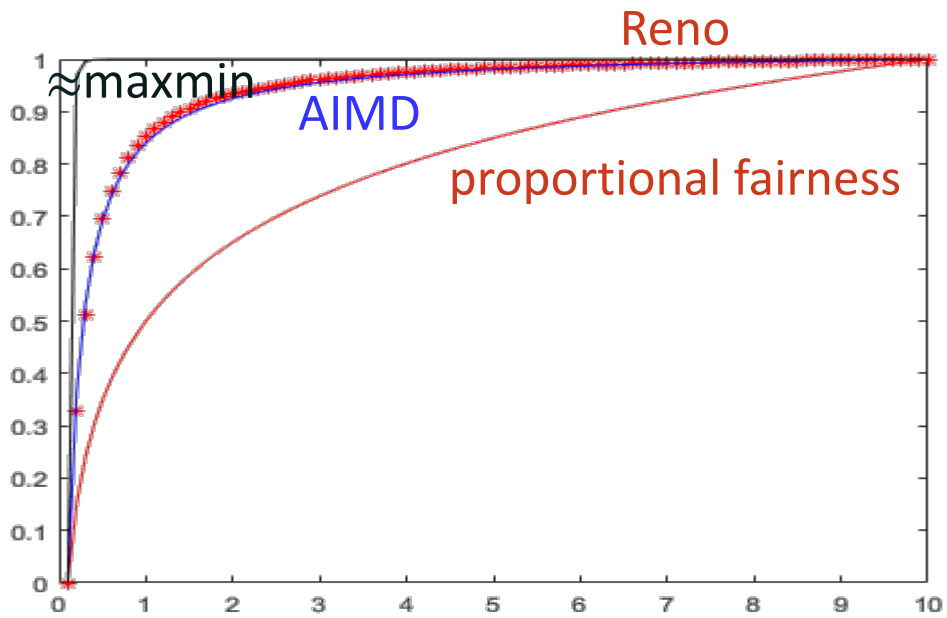
Dynamic error-correction coding at the physical layer to avoid errors on the wireless link

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

where 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, and very close to theoretical AIMD (with additive term 1MSS per RTT, and multiplicative factor 0.5):



utility functions for RTT = 100 ms

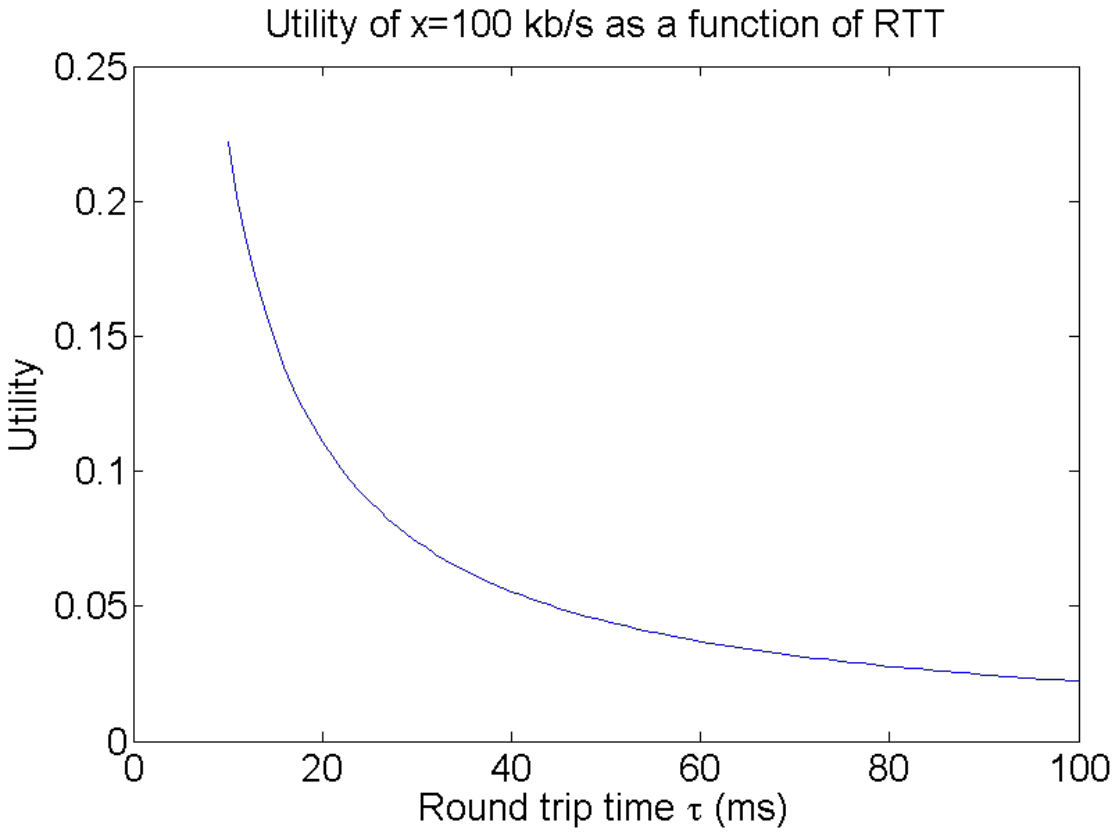
max-min approx. is $U(x) = 1 - x^{-5}$

Dependence on RTT

Utility U is an increasing function of rate x_i, but a *decreasing function of RTT*:

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

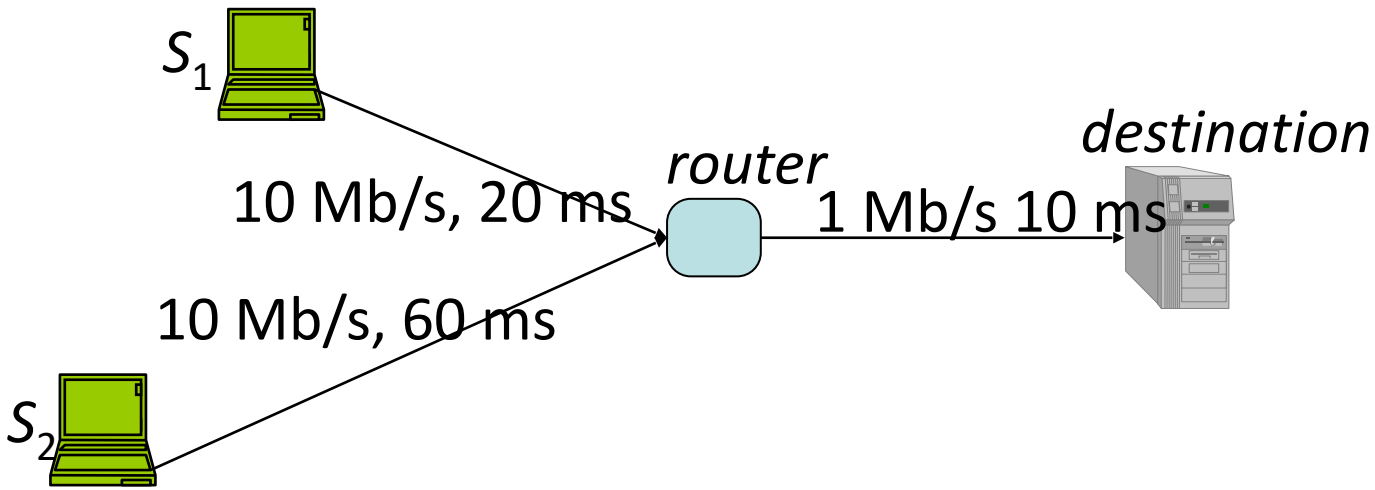
What does this imply?



- *When allocating rates to maximize the aggregate utility, it seems a waste to allocate high rates to flows with large RTTs*

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 87072

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

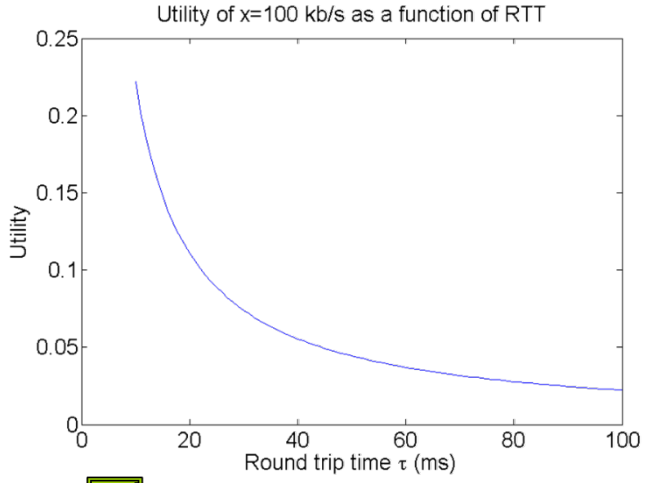
The utility is less when RTT is large; hence, TCP tries less hard to give a high rate to sources with large RTT.

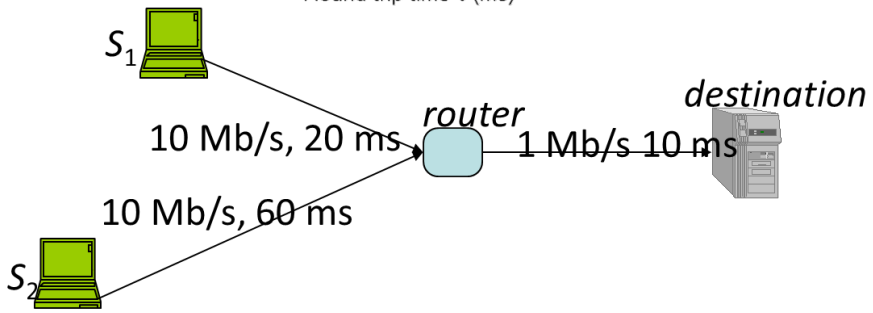
S_2 has a larger RTT than S_1.

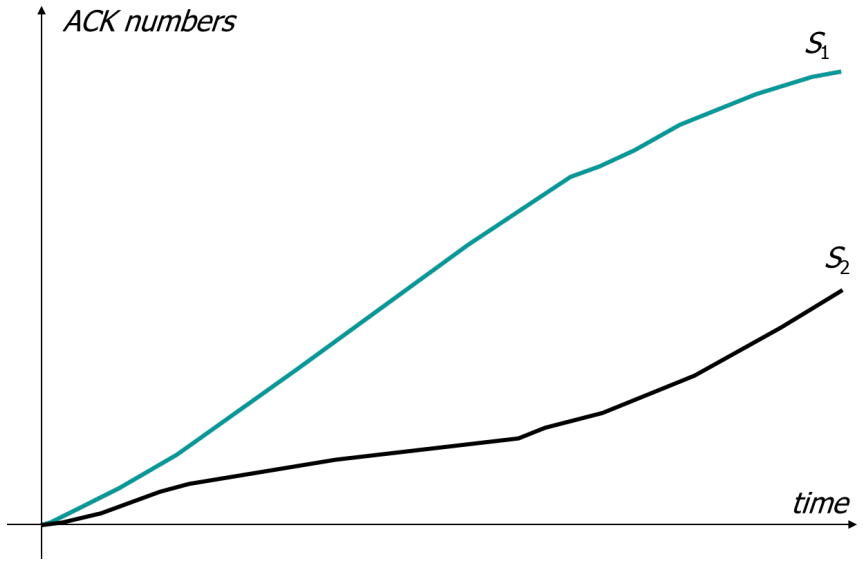
So, S_2 gets lower rate.

A more *practical explanation*:
New ACKs arrive at S_1 faster than at S_2, S_1 opens its window faster, hence achieves a higher rate.

Answer A.







The RTT Bias of TCP Reno

Competing flows with different RTTs are not treated equally

- flow with large RTT attains less rate
- [practical explanation:] additive increase is ~1MSS per RTT (instead of 1MSS per time unit); so a flow with a *smaller RTT* can *"open" the window faster*.

➡ A flow that uses ***many links*** attains less rate because of 2 reasons:

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 attaining less rate, which is *desirable* in view of the theory of fairness.
2. If this flow has a larger RTT, then things are different. The mechanics of additive increase leads to this source attaining 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.

[for the proof 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. $\theta_1 = \frac{3}{7}\theta_2$

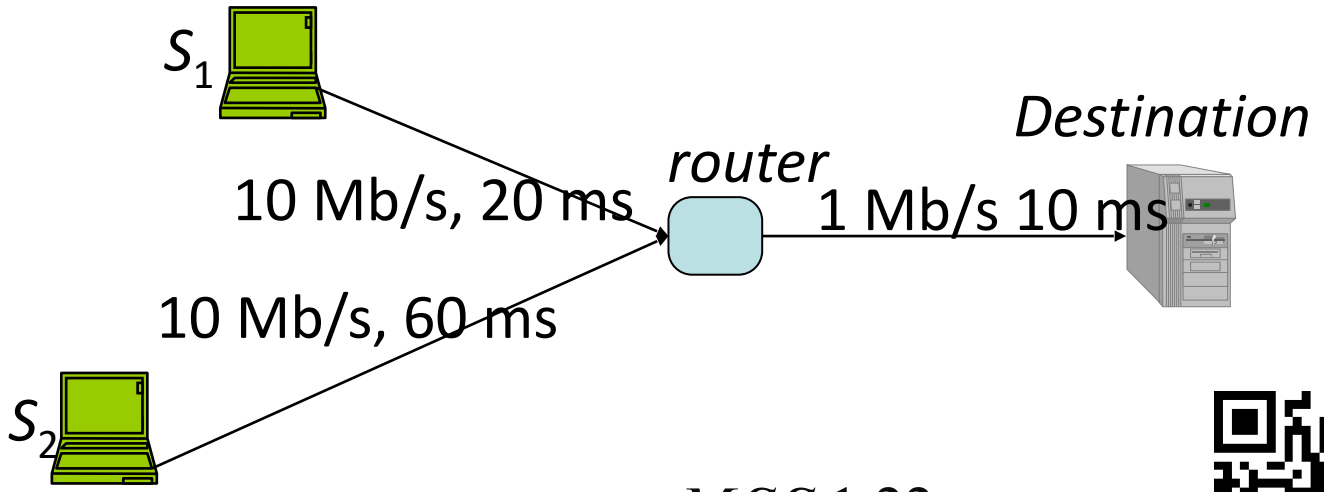
B. $\theta_1 = \theta_2$

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

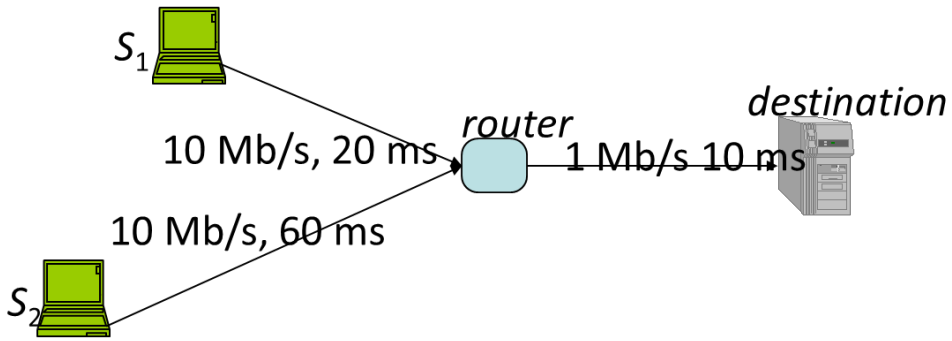


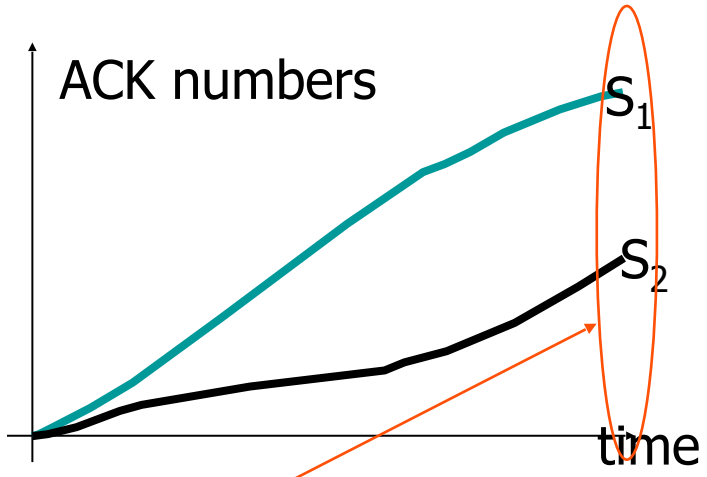
$$\theta = \frac{MSS\ 1.22}{RTT\sqrt{q}}$$

Go to web.speakup.info or download speakup app

Join room 87072

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

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

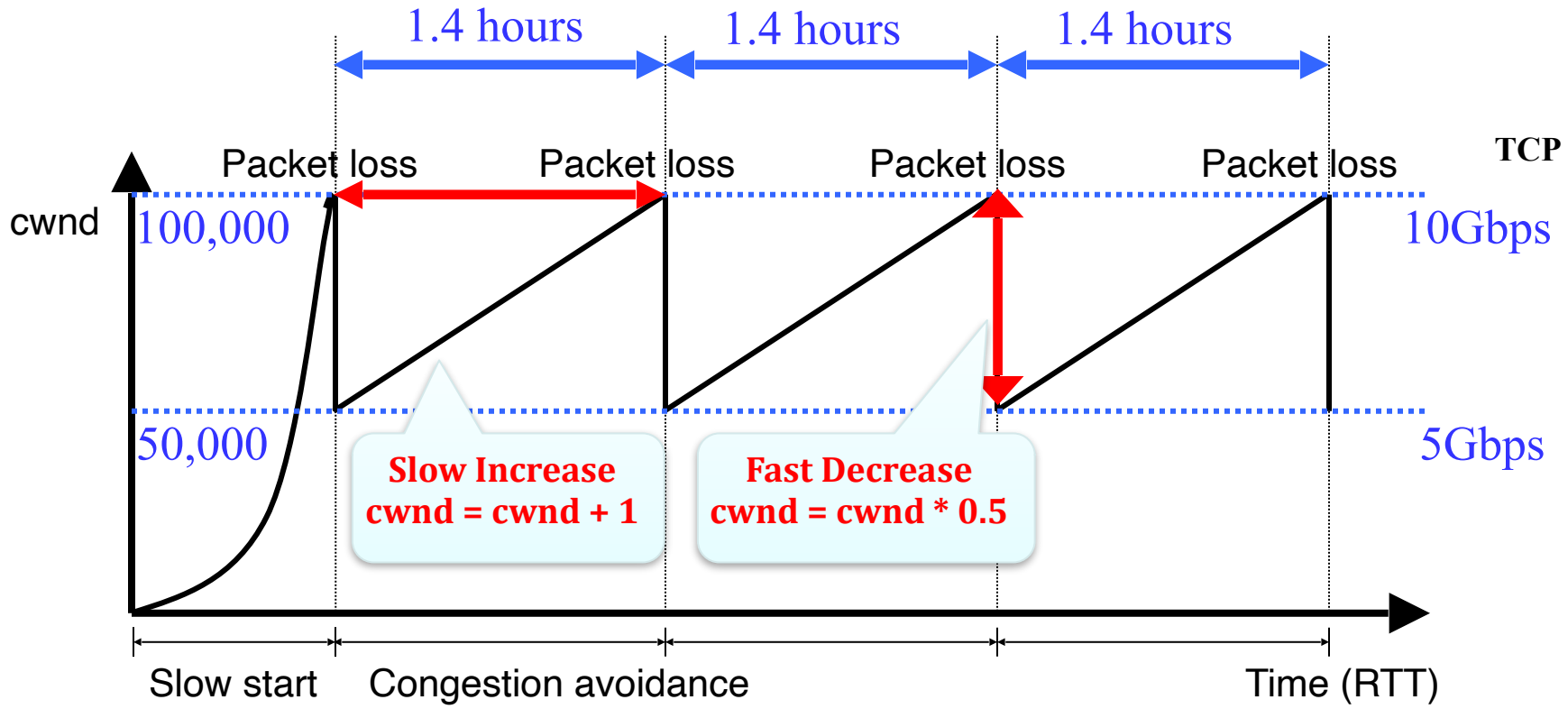
Answer C.

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, non-smooth/bursty behavior
- *Bufferbloat syndrome*: Reno first fills buffers before adapting the rate
  - ➡ Buffers are constantly (almost) full — their usefulness is cancelled
  - ➡ RTT increases unnecessarily — self-inflicted delays for all flows
  - ➡ *Interactive, short flows experience large latencies* when buffers are large (and full)

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, increase is *cubic* (not additive)
- multiplicative decrease with factor ×0.7 (instead of ×0.5)

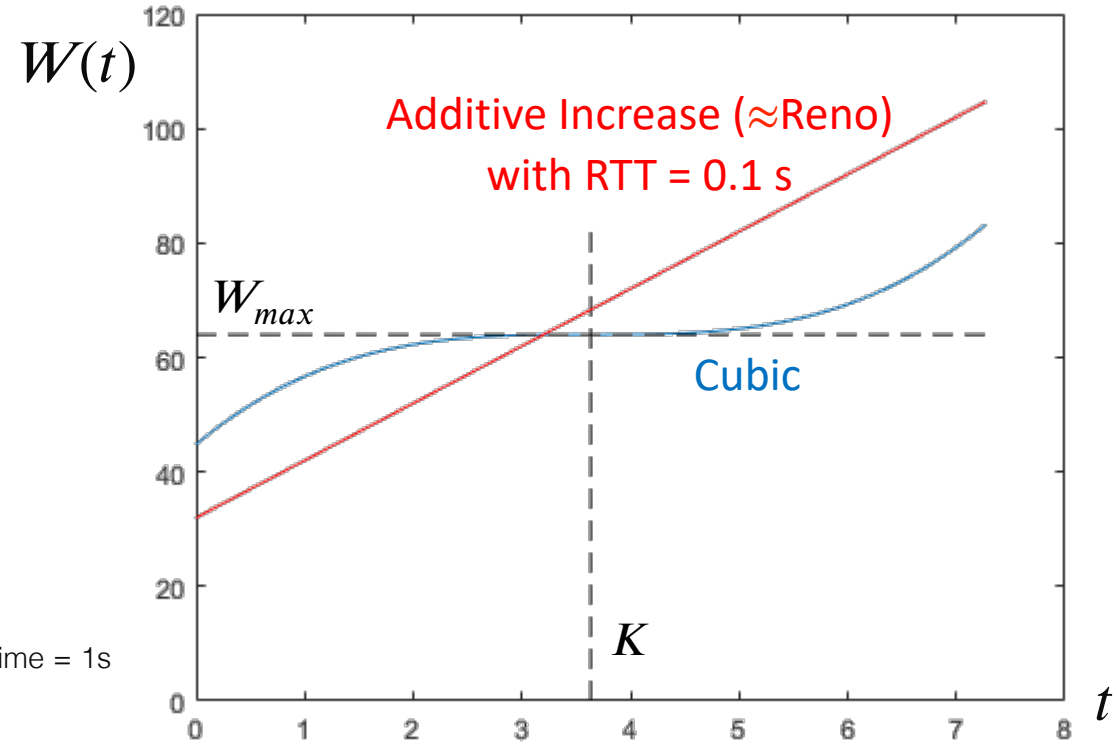
What?

Suppose congestion avoidance is entered at time $t_0 = 0$ and let $W_{max} =$ value of cwnd, when loss was detected.

`cwnd` starts increasing as:

$$W(t) = W_{max} + 0.4(t - K)^3 ,$$

where K is such that $W(0) = 0.7\ W_{max}$



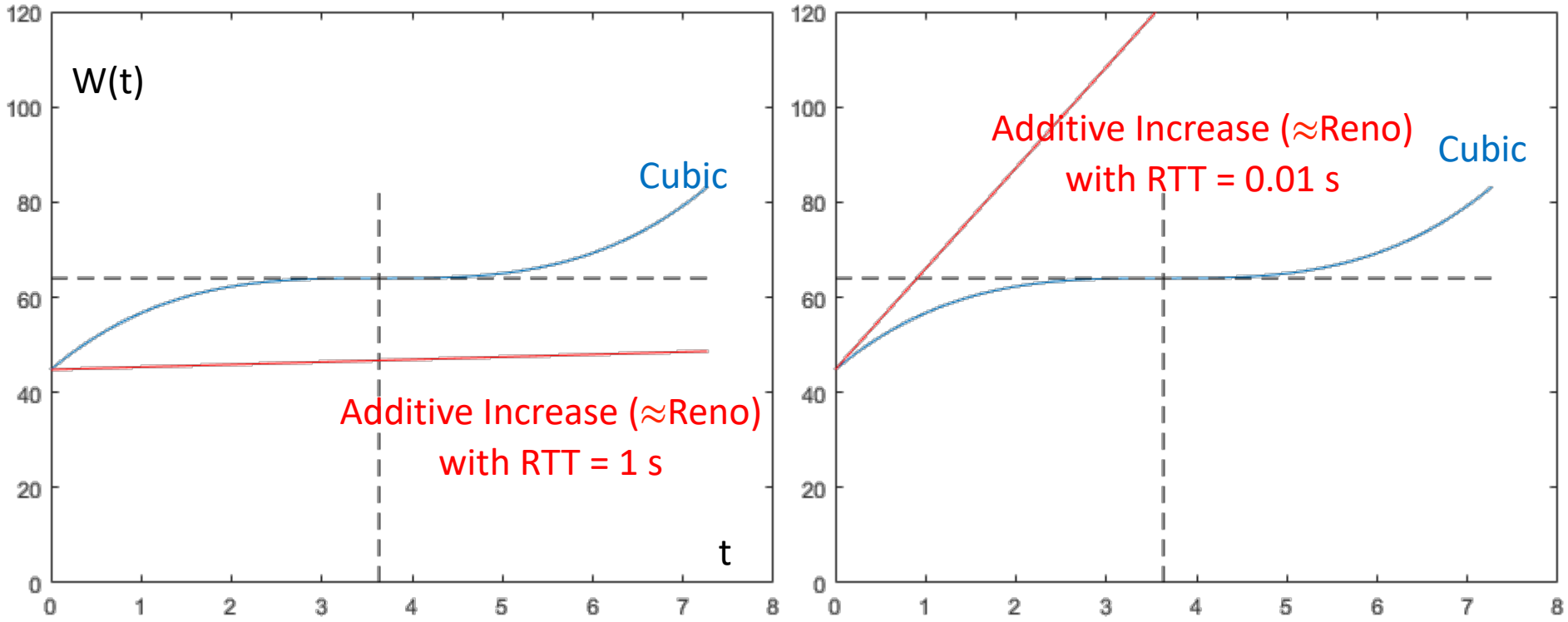
Units are: data = 1MSS; time = 1s

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*

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



Cubic's Window

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

where:

- $W_{AIMD}(t) = W(0) + r_{cubic}\frac{t}{RTT}$ is the cwnd of a hypothetical AIMD that achieves the *same loss-throughput formula* as standard Reno, but applies Cubic's multiplicative decrease factor $\beta_{cubic} = 0.7$ (as opposed to Reno's factor ≈ 0.5).
- For that AIMD, the additive term compensating for the smaller multiplicative decrease is: $r_{cubic} = 3\frac{1-\beta_{cubic}}{1+\beta_{cubic}} = 0.529$ MSS (as opposed to Reno's term ≈ 1 MSS).

Benefits:

➡ Cubic uses max cwnd of 2 options: additive increase or cubic increase (with the same multiplicative decrease)

➡ Cubic's throughput ≥ Reno's throughput
equality holds when RTT or bandwidth-delay product are small (i.e. whenever not in LFNs)

➡ Cubic is always at least "as fast as standard Reno"

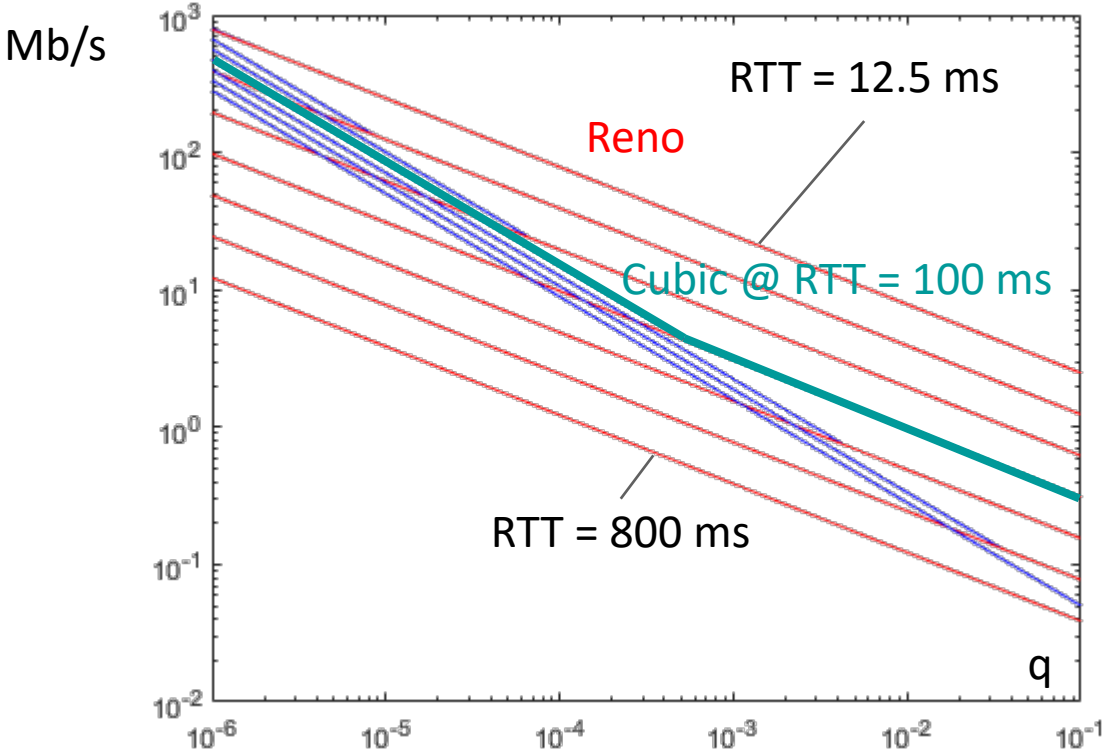
Cubic's Loss throughput formula

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:

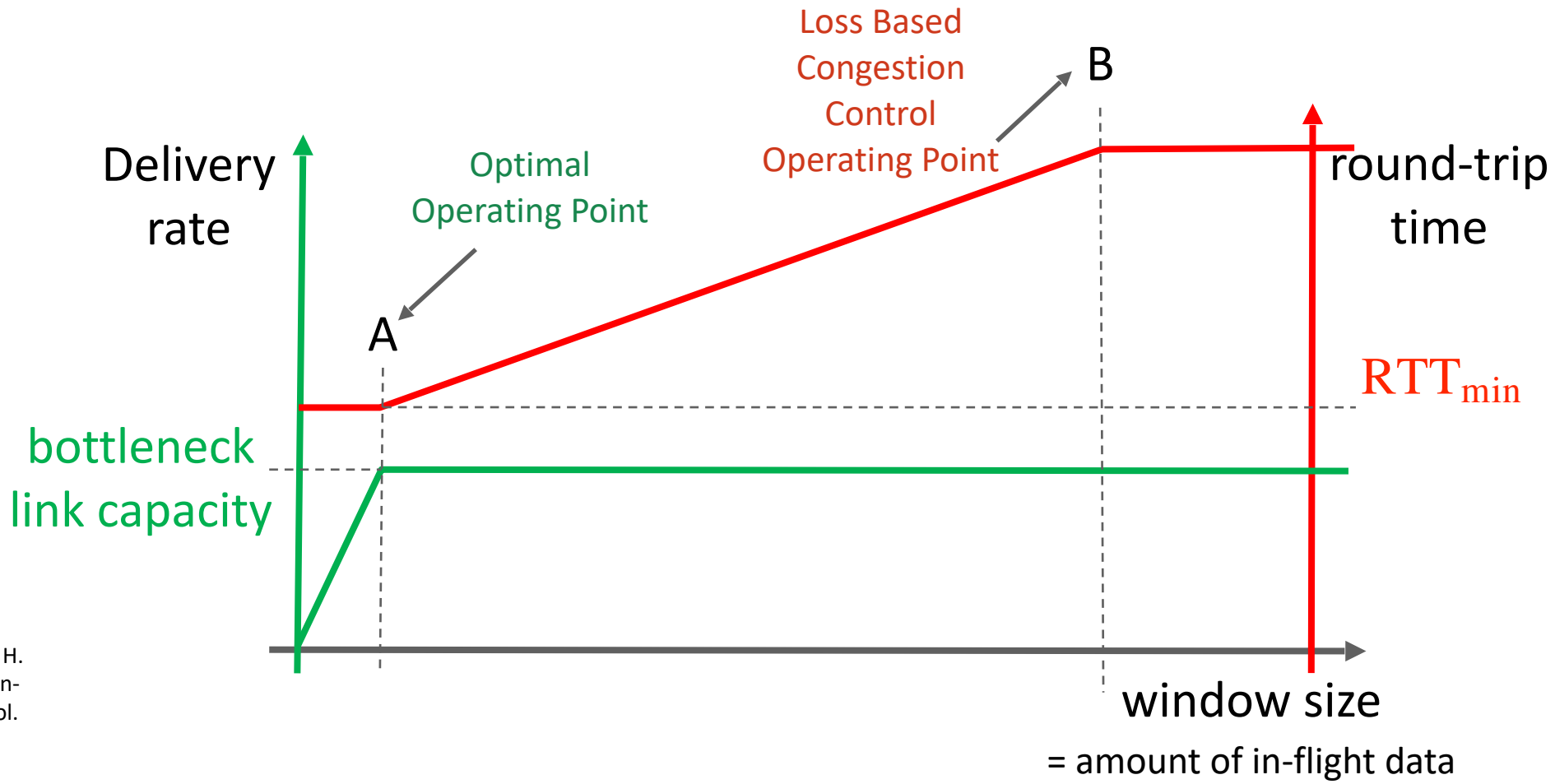
- Cubic's formula is same as Reno for small RTTs and small BW-delay products
- but a TCP Cubic gets *more throughput* than TCP Reno when bit-rate and RTT are large



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 delays*: *increased latencies* and not-well-utilized buffers 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.

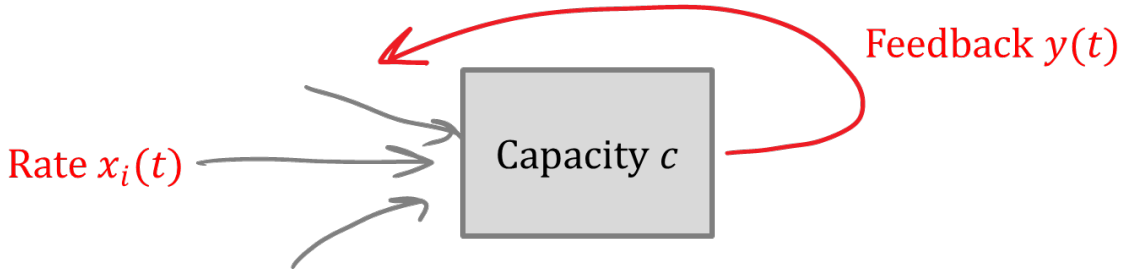
from [Hock et al, 2017] Mario Hock, Roland Bless, Martina Zitterbart, “Experimental Evaluation of BBR Congestion Control”, ICNP 2017:

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…

…tackles bufferbloat

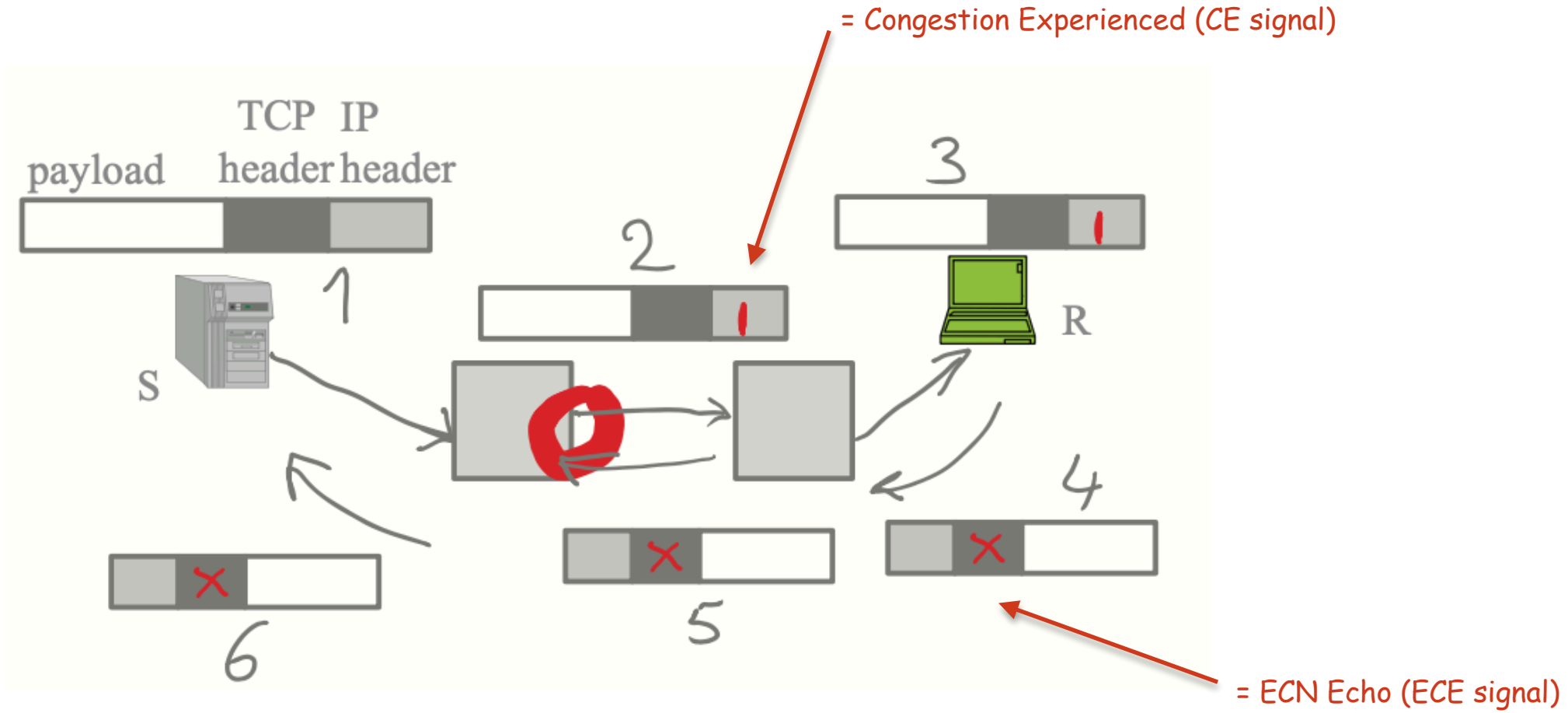
What? Network signals congestion without dropping packets (similarly to DECbit)



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



At t=6 (ECE received):
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.

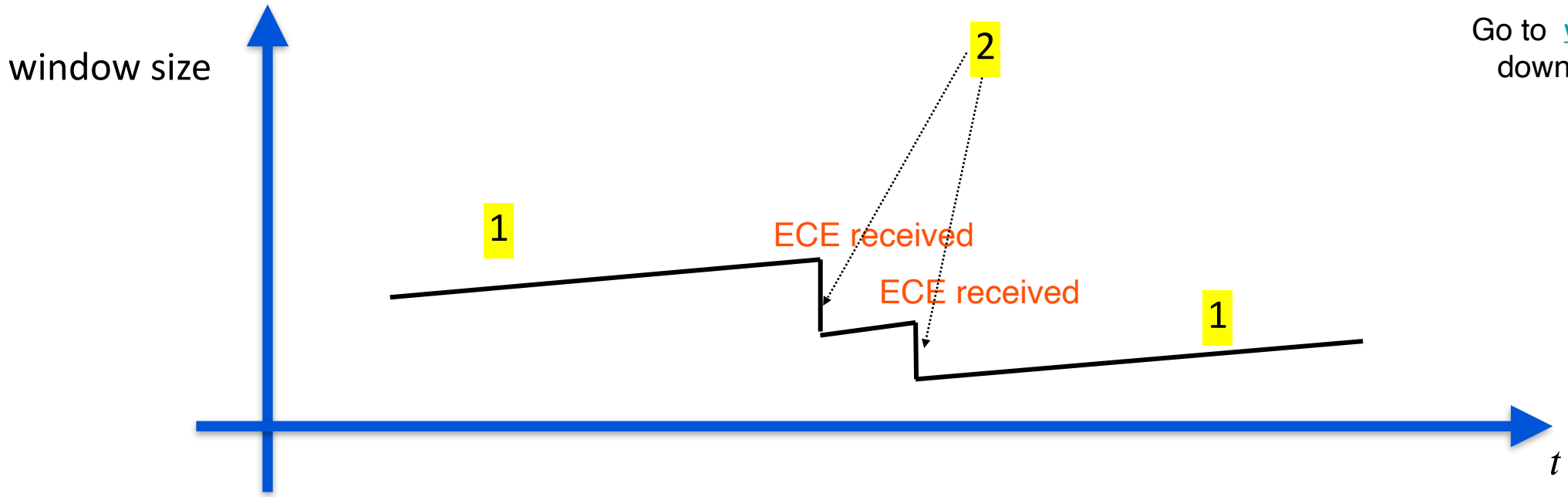
   The receiver continues to set the ECE flag until it receives a packet with CWR set.

   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 87072



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

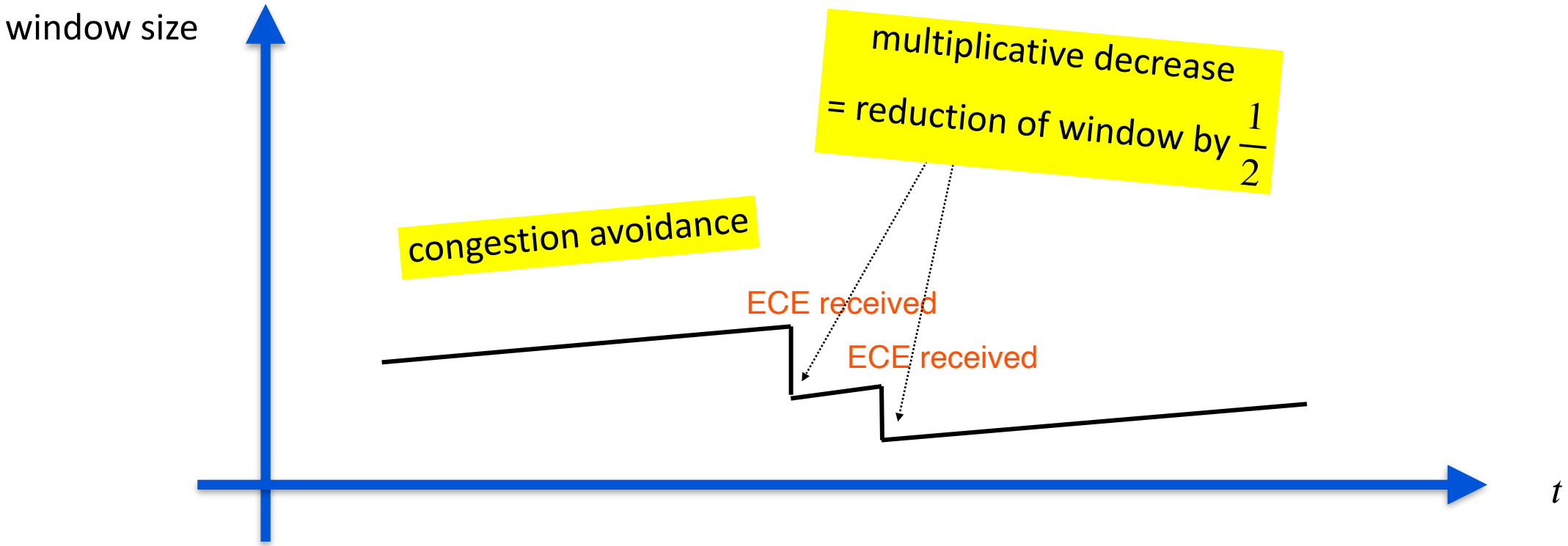
CA: congestion avoidance

SS: slow start = multiplicative increase

MD: multiplicative decrease

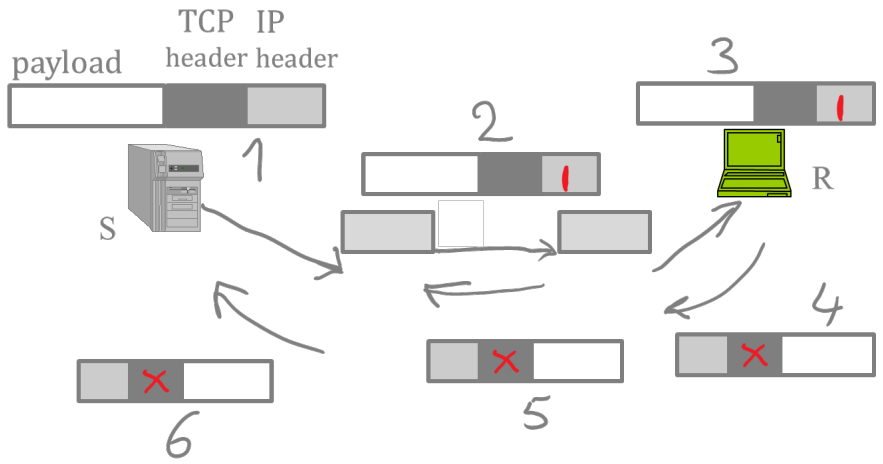
Solution

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; and *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.
- Upon 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 bufferbloat syndrome

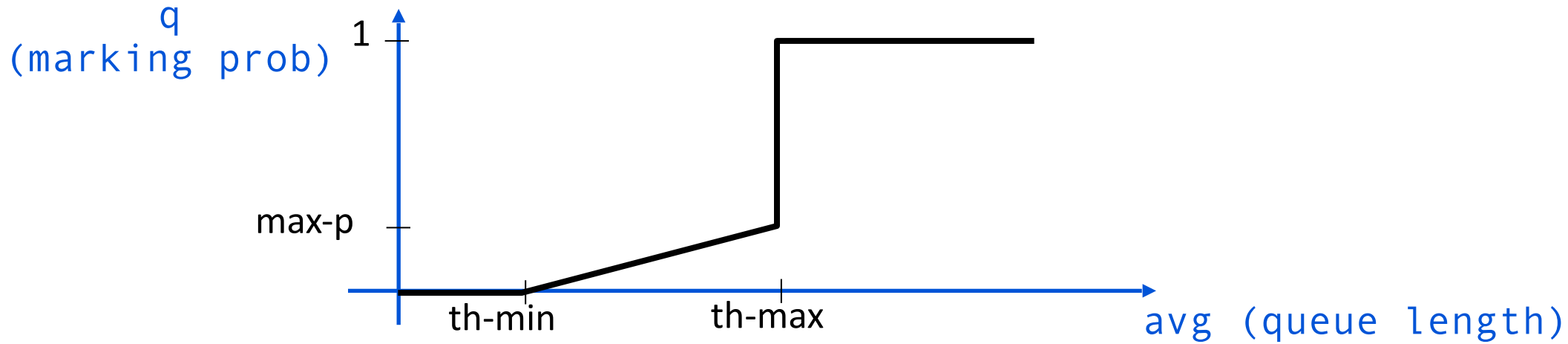
How? E.g. with a RED (Random Early Detection) queue:

- 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

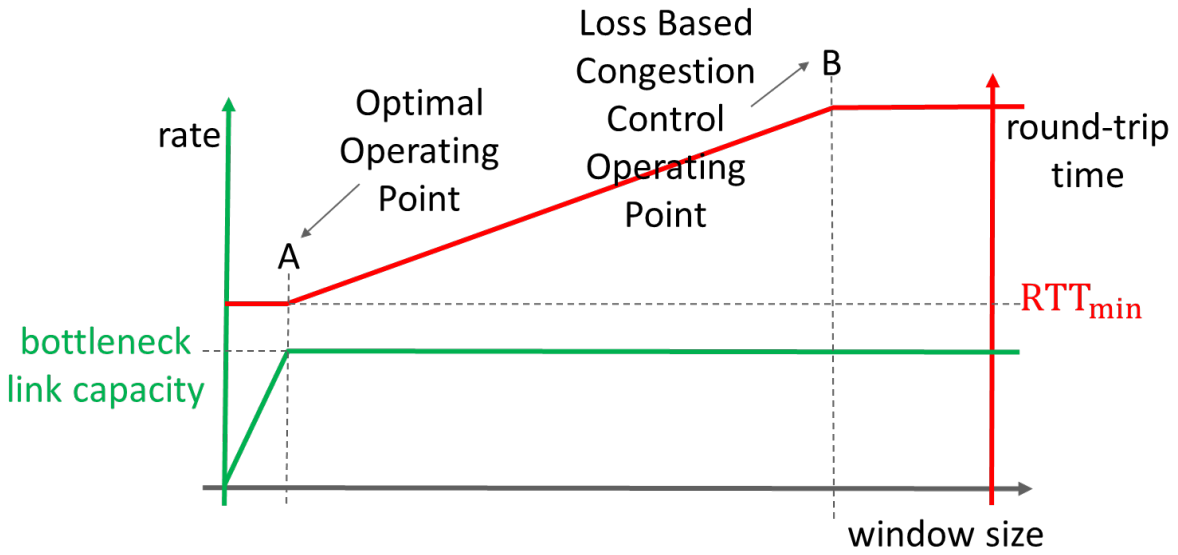


See the difference from *passive* queue management = drop a packet only when queue is full = "Tail Drop"

But…Active Queue Management does not require ECN

- AQM can be applied even if ECN is not used
  e.g. with RED, a packet is *dropped* with probability q computed by the RED curve

- Expected benefits in this case:
  - *Mitigate bufferbloat* – reduce latencies
  - *Avoid irregular drop patterns:*
    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 87072

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 propagation delays (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 87072

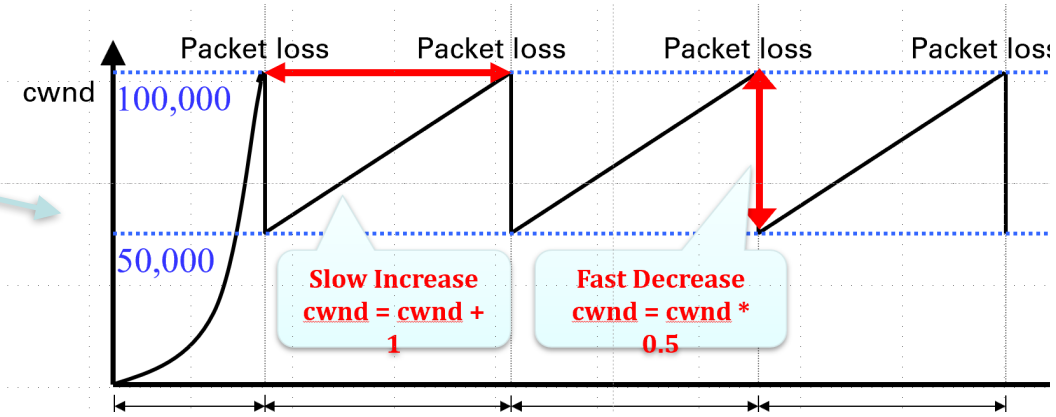
Solution

Answer D (also B could work)

- Cubic has better performance than Reno when bandwidth-delay product is large, which may happen in data centers.
  But Cubic performs at least as good as Reno in (almost) *every* case, so why not choosing Cubic anyways?
- Without ECN there will be bufferbloat, which means high latency for short flows

Standard operation of Cubic (or Reno) with ECN has *drawbacks for jumbo* flows in data centers:

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



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 ECE 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 87072

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].