Congestion Control In The Internet

Part 2: Implementation

JY Le Boudec

2020

Contents

6. Congestion Control in the Internet is in TCP

TCP is used to avoid congestion in the Internet

in addition to what was shown about TCP, a TCP source adjusts its window to the congestion status of the Internet
(slow start, congestion avoidance)

this avoids congestion collapse and ensures some fairness

TCP sources interpret losses as a negative feedback

UDP sources have to implement their own congestion control

Some UDP sources imitate TCP : “TCP friendly”

Some UDP sources (e.g. QUIC) implement the same code as TCP congestion control

TCP Reno, New Reno, Vegas, etc

The congestion control module of TCP exists in n versions;

Popular versions are

TCP Reno with SACK (historic version, also in QUIC)

TCP Cubic (widespread today in Linux servers)

Data Center TCP (Microsoft and Linux servers)

TCP Reno Congestion Control
Uses ≈AIMD and Slow Start

TCP adjusts the window size based on the approximation rate $\approx \frac{W}{RTT}$

W = min (cwnd, offeredWindow)
offeredWindow = window obtained by TCP's window field
cwnd = controlled by TCP congestion control

Negative feedback = loss, positive feedback = ACK received

increase is ≈ additive (≈+1 MSS per RTT),

Multiplicative Decrease ($u_1 = 0.5$)

Slow start with increase factor $w_0 = 2$ per round trip time (approx.)

Loss detected by timeout → slow start

Loss detected by fast retransmit → fast recovery (see next)

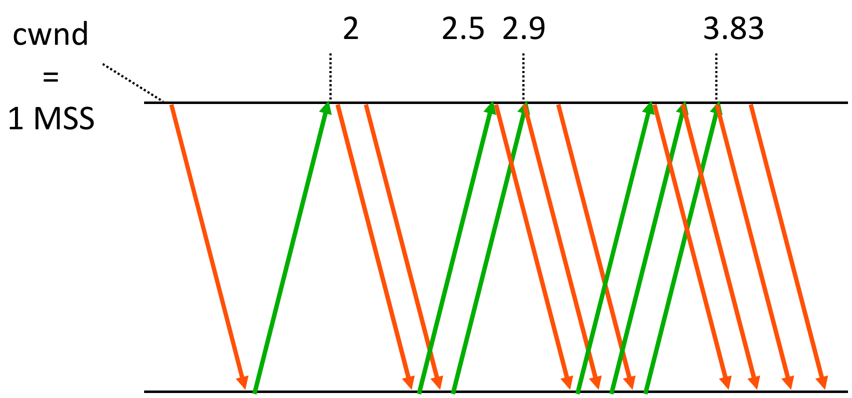
TCP Implementations of…

Multiplicative decrease:

ssthresh = 0.5 × cwnd

Additive increase:

for every ack received cwnd = cwnd + MSS×MSS / cwnd
(if we counted in packets, this would be cwnd+=1/cwnd



this is slightly less than additive increase
other implementations exist: for example: wait until the cwnd bytes are acked and then increment cwnd by 1 MSS

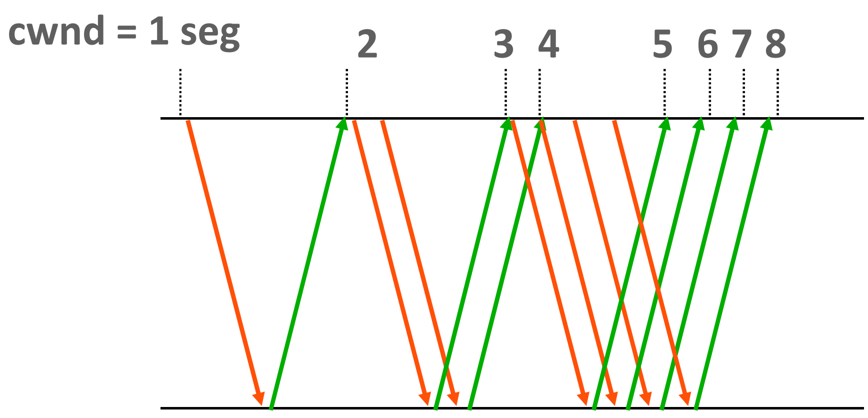
How TCP approximates…

…multiplicative increase : (Slow Start)

non dupl. ack received during slow start ->

cwnd = cwnd + MSS (in bytes) (1)

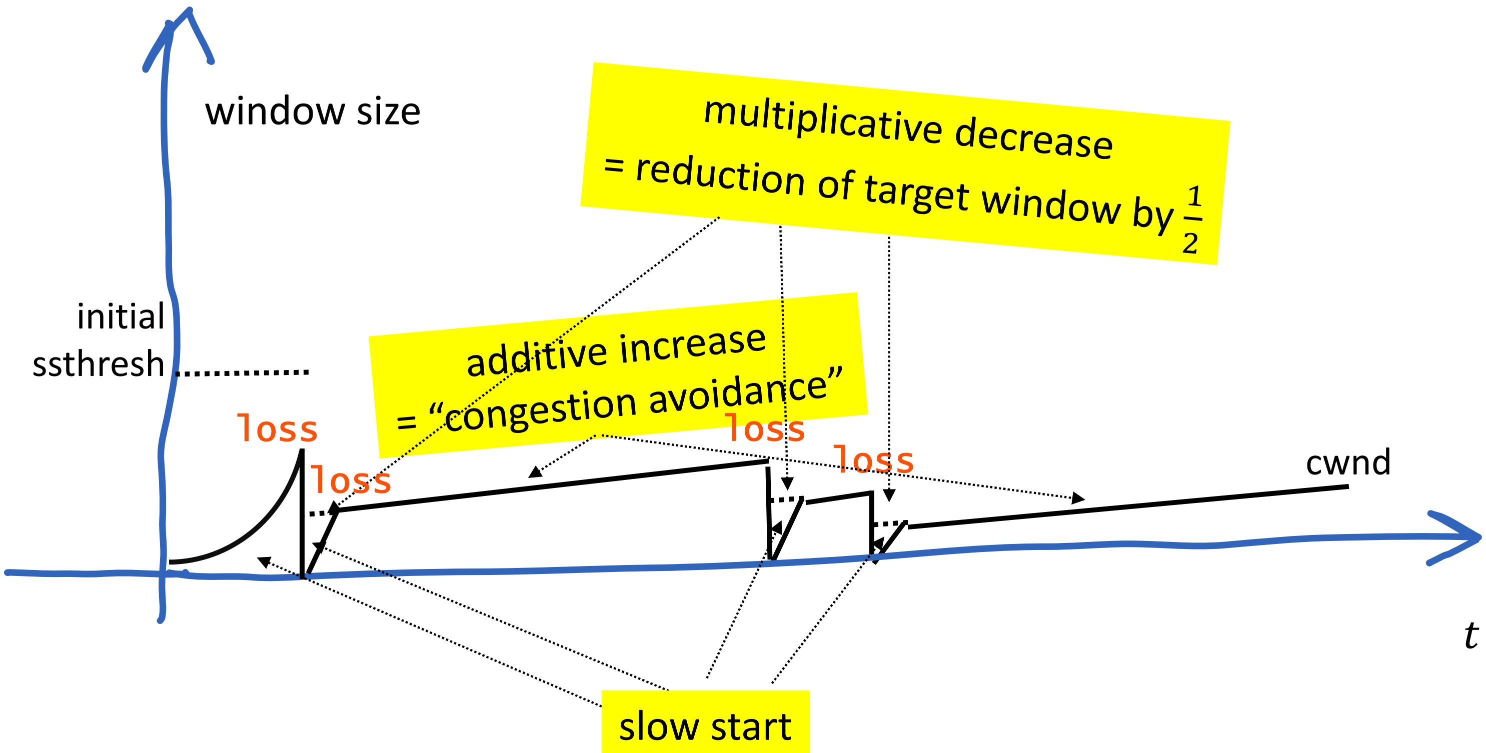
if cwnd = ssthresh then go to congestion avoidance



(1) is equivalent in packets to

cwnd = cwnd + 1 (in packets)

AIMD and Slow Start



target window of slow start is called ssthresh («slow start threshold»)

there is a slowstart phase initially and after every packet loss detected by timeout

Fast Recovery

Slow start used when we assume that the network condition is new or abruptly changing

i.e. at beginning and after loss detected by timeout

In all other packet loss detection events, slow start is not used, but "fast recovery" is used instead

- Problem to be solved: the formula "rate $\approx \frac{W}{RTT}$" is not true when there is a packet loss – sliding window operation may stop sending

With Fast Recovery

- target window is halved
- But congestion window is allowed to increase beyond the target window until the loss is repaired

Fast Recovery Details

When loss is detected by 3 duplicate acks

ssthresh = 0.5 × current-size

ssthresh = max (ssthresh, 2 × MSS)

cwnd = ssthresh + 3 × MSS (exp. increase)

cwnd = min (cwnd, 64K)

For each duplicated ACK received

cwnd = cwnd + MSS (exp. increase)

cwnd = min (cwnd, 64K)

When loss is repaired

cwnd = ssthresh

Goto congestion avoidance

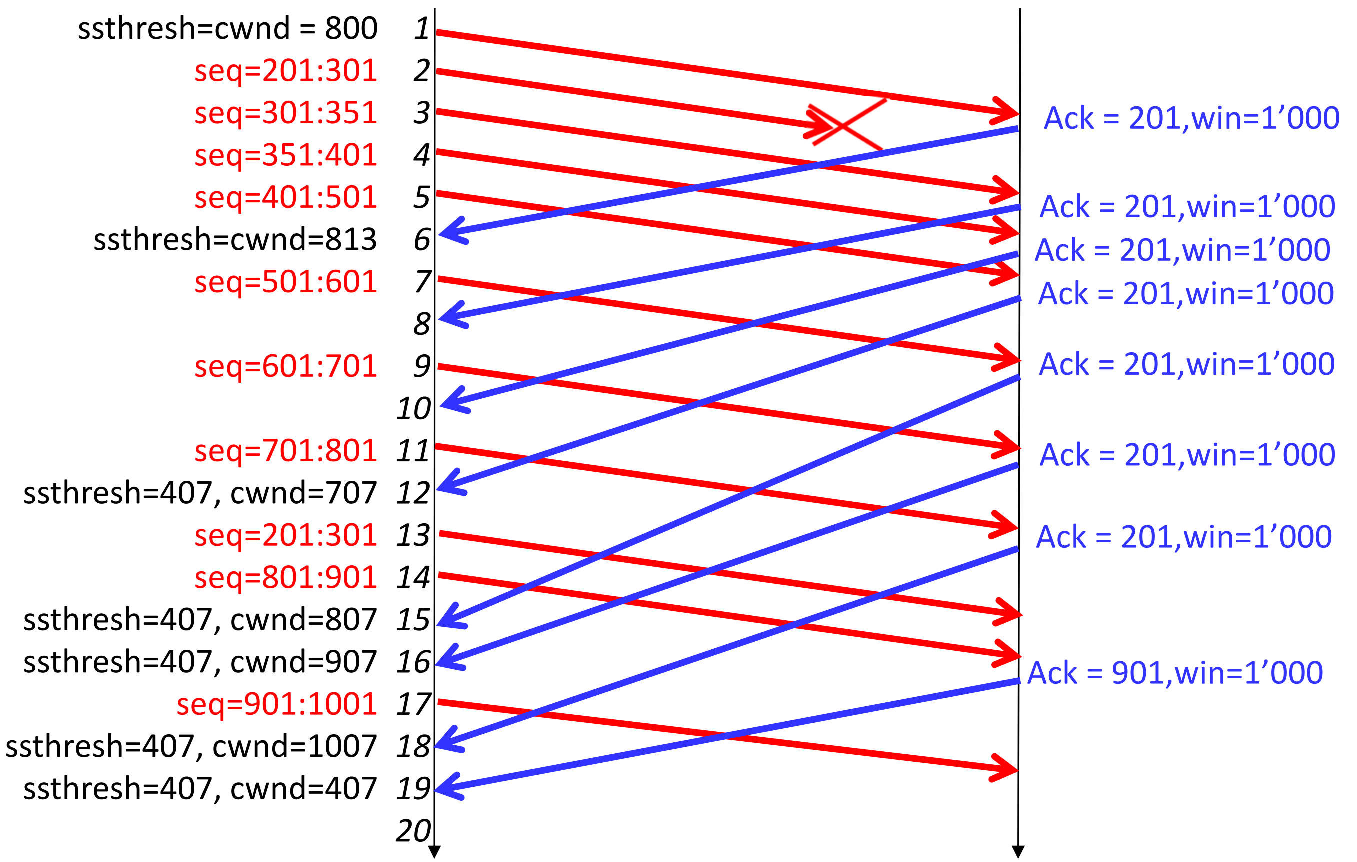
Fast Recovery Example

TcpMaxDupACKs=3

During congestion avoidance:

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

MSS = 100



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).

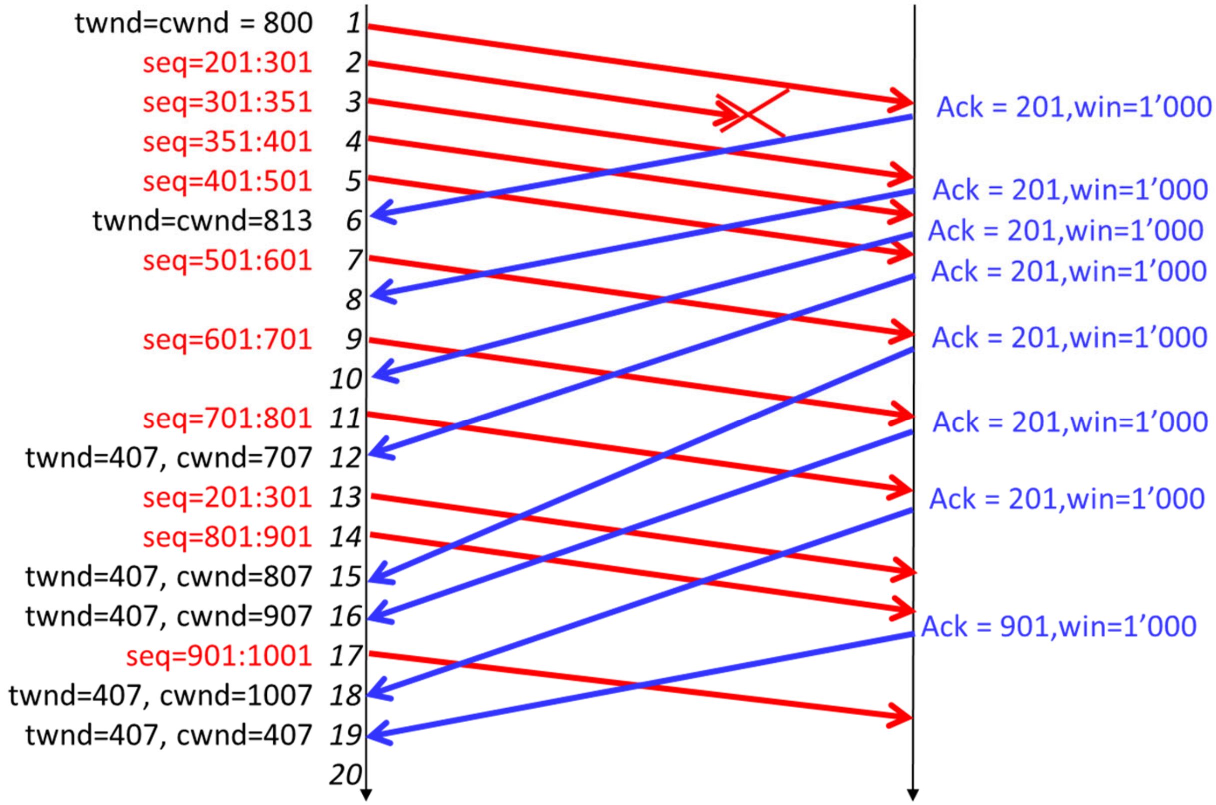
At time 13 the source retransmits the lost packet. At time 14 it transmits a fresh packet. This is possible because the window is large enough. The window size, which is the minimum of the congestion window and the advertised window, is equal to 707. Since the last acked byte is 201, it is possible to 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 18 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 ?

A. 1
B. 2
C. 3
D. 4
E. $\geq$ 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.

Assume a TCP flow uses WiFi with high loss ratio. Assume some packets are 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

Solution

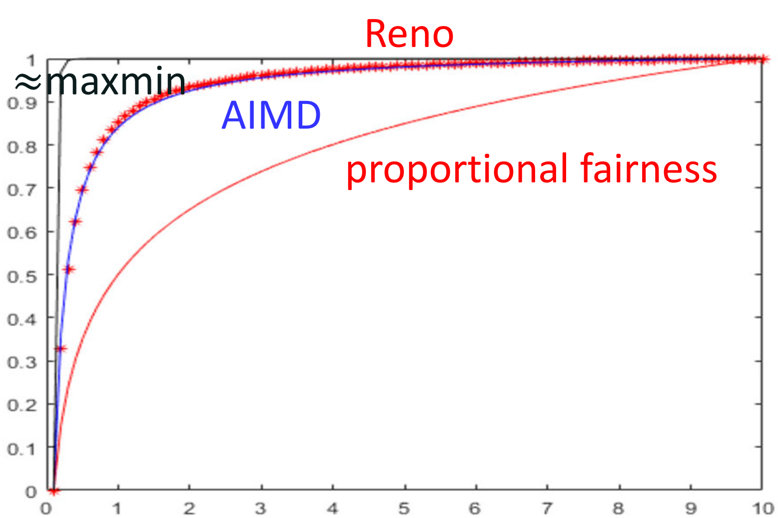
Answer B: the TCP source does not know the cause of a loss.

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" https://ica1www.epfl.ch/PS_files/LEB3132.pdf)

For sources that have same RTT, the fairness of TCP is between maxmin fairness and proportional fairness, closer to proportional fairness.



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

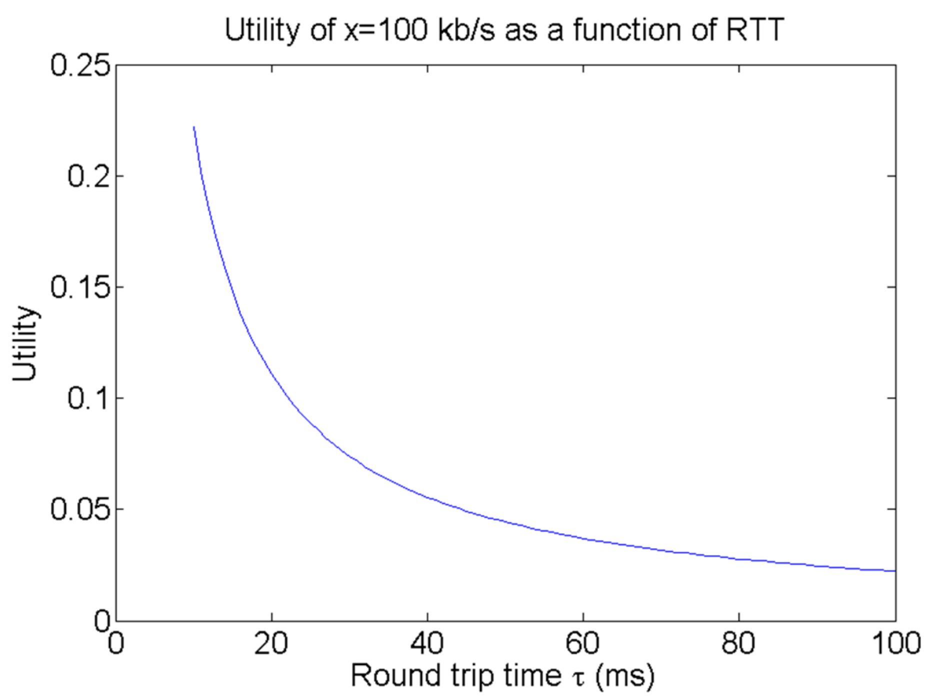
TCP Reno and RTT

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;

The utility U is a decreasing function of τ

What does this imply ?



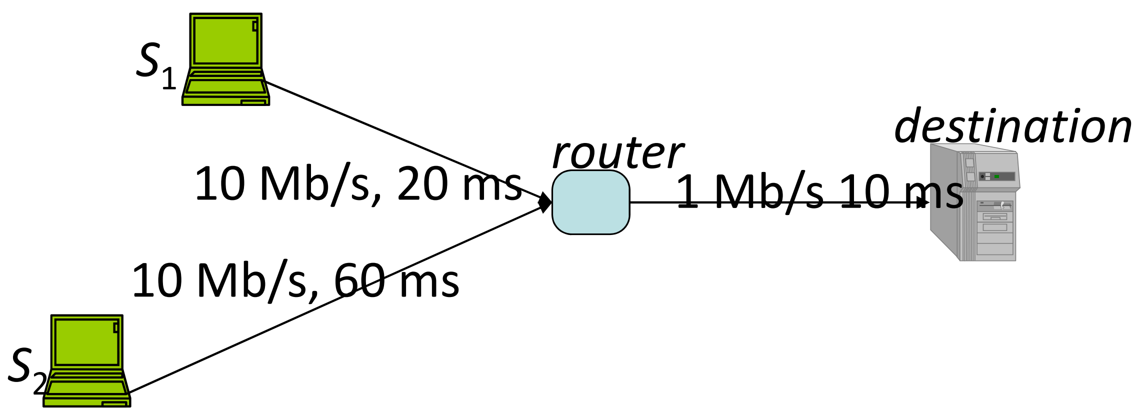
$\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ send to destination using one TCP connection each, RTTs are 60ms and 140ms. Bottleneck is link « router-destination ». Who gets more ?

A. S_1 gets a higher throughput

B. S_2 gets a higher throughput

C. Both get the same

D. I don't know



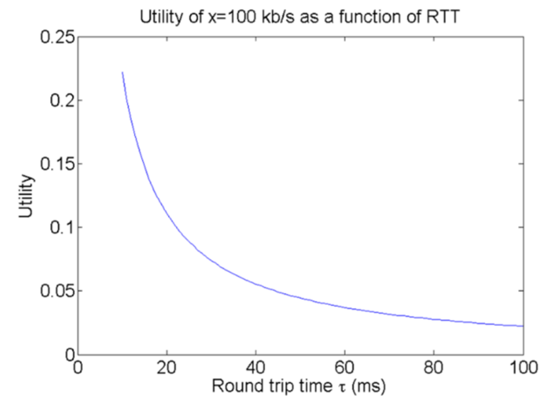
Solution

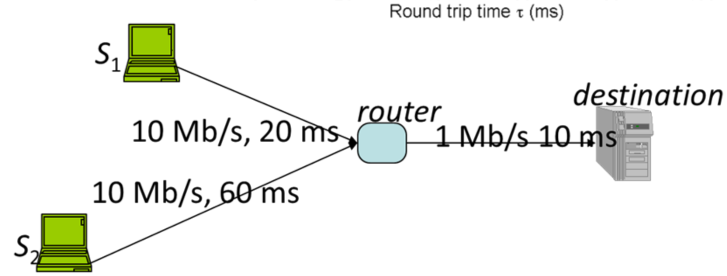
For long lived flows, the rates obtained with TCP are as if they were distributed according to utility fairness, with utility of flow i given by $U(x_i) = \frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$

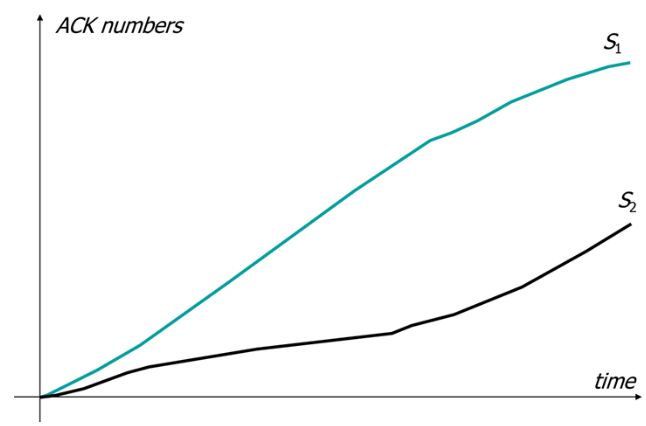
S_1 has a smaller RTT than S_2

The utility is less when RTT is large, therefore TCP tries less hard to give a high rate to sources with large RTT. S_2 gets less.

Answer A.







The RTT Bias of TCP Reno

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

source with large RTT obtains less

A source that uses *many hops* obtains less rate because of two combined factors, one is desired, the other happens by accident:

1. this source uses more resources. The mechanic of proportional fairness leads to this source having less rate – this is desirable in view of the theory of fairness.
2. this source has a larger RTT. The mechanics of additive increase leads to this source having less rate – this is an undesired bias in the design of TCP Reno.

   Cause is : additive increase is one packet per RTT (instead of one packet per constant time interval).

TCP Reno
Loss - Throughput Formula

Consider a *large* TCP connection (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 negligible compared to RTT, losses are rare, time spent in Slow Start and Fast Recovery negligible, losses occur periodically.

(see “Rate adaptation, Congestion Control and Fairness: A Tutorial" https://ica1www.epfl.ch/PS_files/LEB3132.pdf)

Guess the ratio between the throughputs θ_1 and θ_2 of S_1 and S_2

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

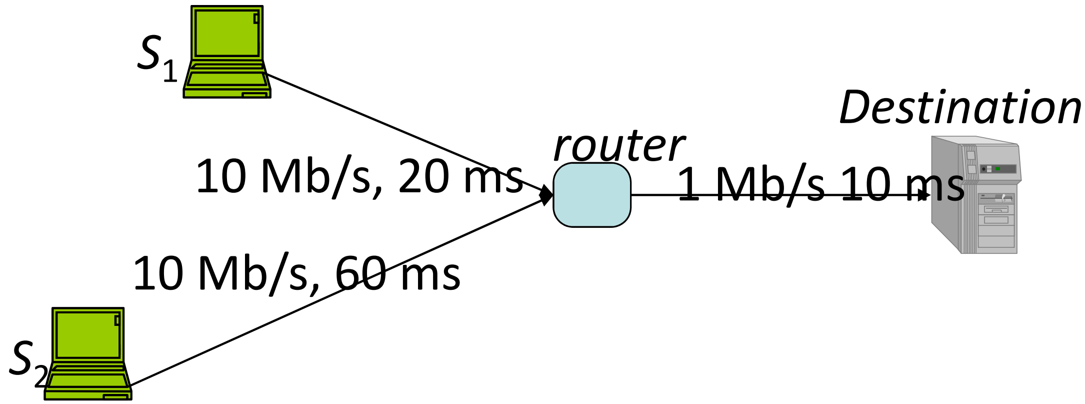
B. $\theta_1 = \theta_2$

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

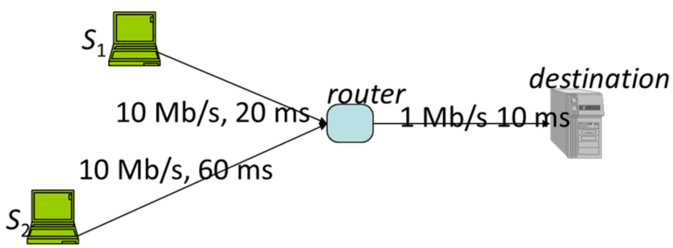
D. $\theta_1 = \frac{10}{3}\theta_2$

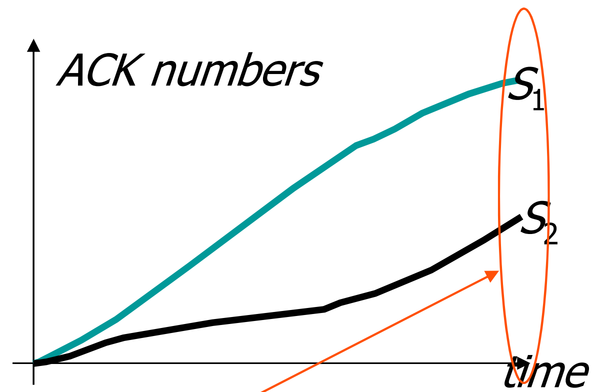
E. None of the above

F. I don't know



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

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

Answer C.

7. TCP Cubic

TCP Reno serves as the reference for congestion control in the Internet as it was the first mature implementation of congestion control.

TCP Reno has a number of shortcomings. Can you cite a few ?

Solution

RTT bias – not nice for users in New Zealand

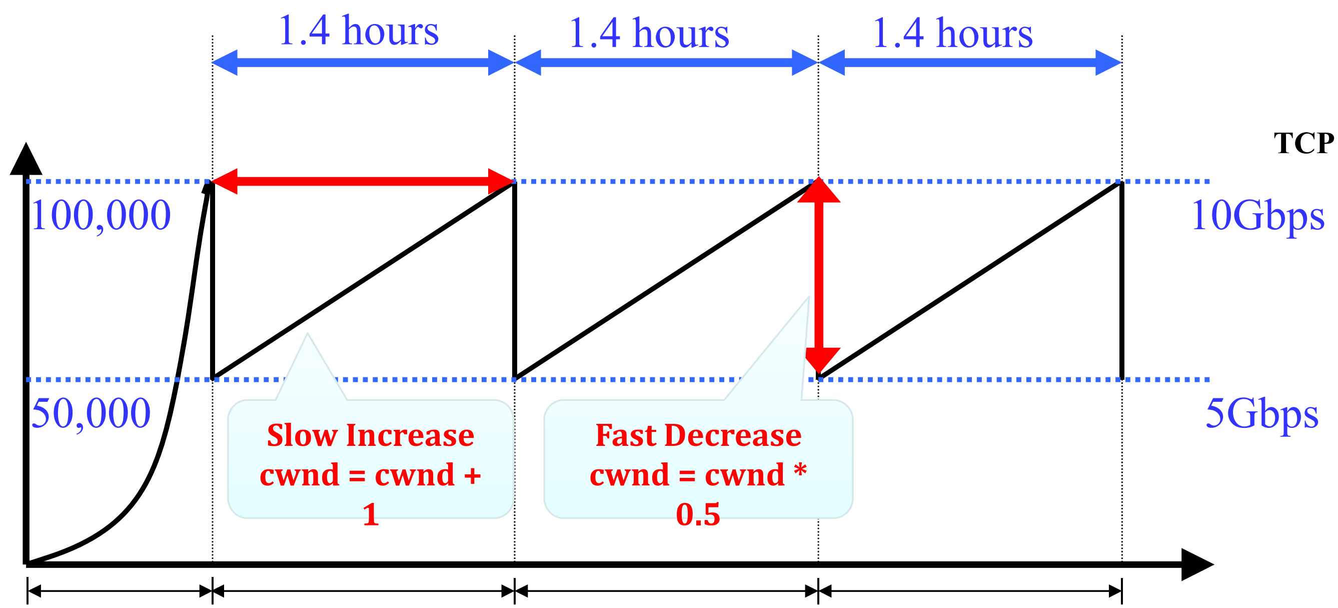
Periodic losses must occur, not nice for application (e.g video streaming).

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 behaviour.

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) and increases delay for all users. Interactive, short flows see large latency when buffers are large and full.

Long Fat Networks (LFNs)

In an LFN, additive increase is too slow



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

the figure assumes congestion avoidance implements a strict additive increase, losses are detected by fast retransmit and ignores the "fast recovery" phase. MSS = 1250B, RTT = 100 msec

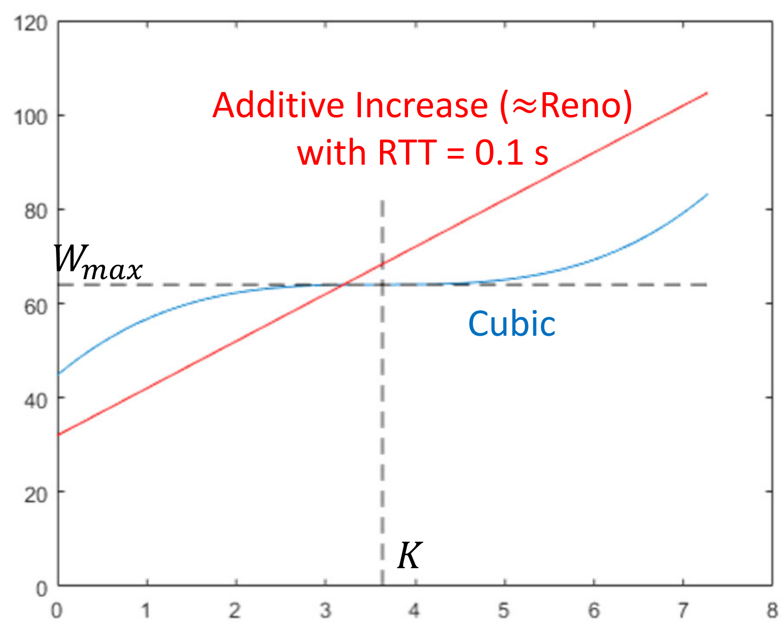
TCP Cubic modifies Congestion Control

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

- Multiplicative Decrease is ×0.7 (instead of ×0.5)
- During congestion avoidance, the increase is not additive but cubic

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.

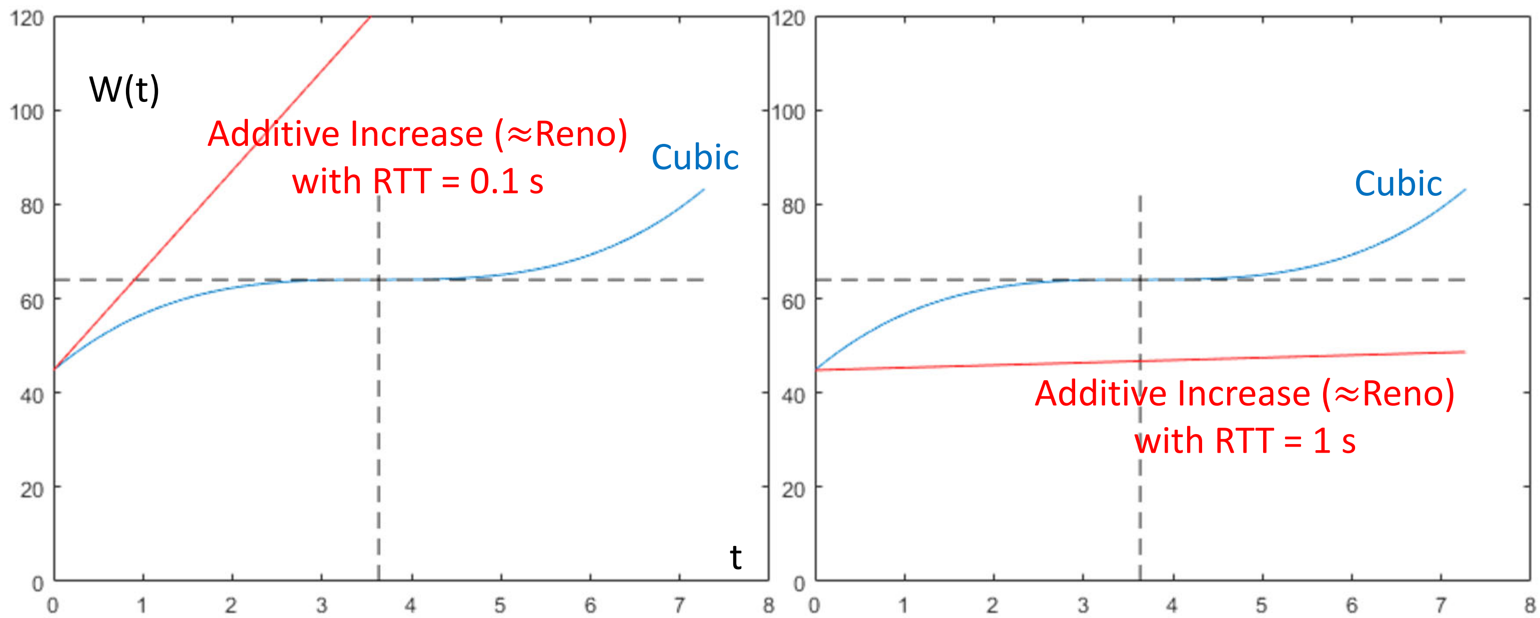
Units are : data = 1MSS; time = 1s



Cubic versus Reno

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

Cubic's window function is independent of RTT;
is slower than Reno when RTT is small, larger when RTT is large



The Cubic Window Increase

Cubic makes sure it is at least as fast as additive increase with an additive increase term r_{cubic} (discussed later):

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

if $W(t) < W_{AIMD}(t)$ then Cubic replaces $W(t)$ by $W_{AIMD}(t)$

⇒ Cubic’s window ≥ AIMD’s window

⇒ When RTT or bandwidth-delay product is small, Cubic does the same as a modified Reno with additive increase r_{cubic} MSS per RTT (instead of 1) and multiplicative decrease $\beta_{cubic} = 0.7$.

r_{cubic} is computed such that this modified Reno has the same loss-throughput formula as standard Reno ⇒ $r_{cubic} = 3\frac{1-\beta_{cubic}}{1+\beta_{cubic}} = 0.529$

⇒ Cubic’s throughput ≥ Reno’ throughput with equality when RTT or bandwidth-delay product is small

Cubic's Other Bells and Whistles

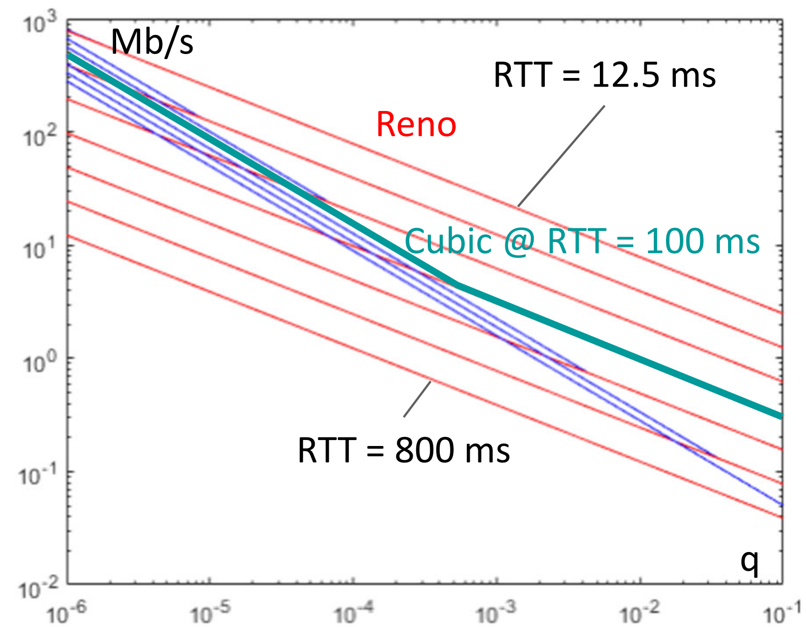
Cubic's Loss throughput formula

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

in MSS per second.

Cubic's formula is same as Reno for small RTTs and small BW-delay products.

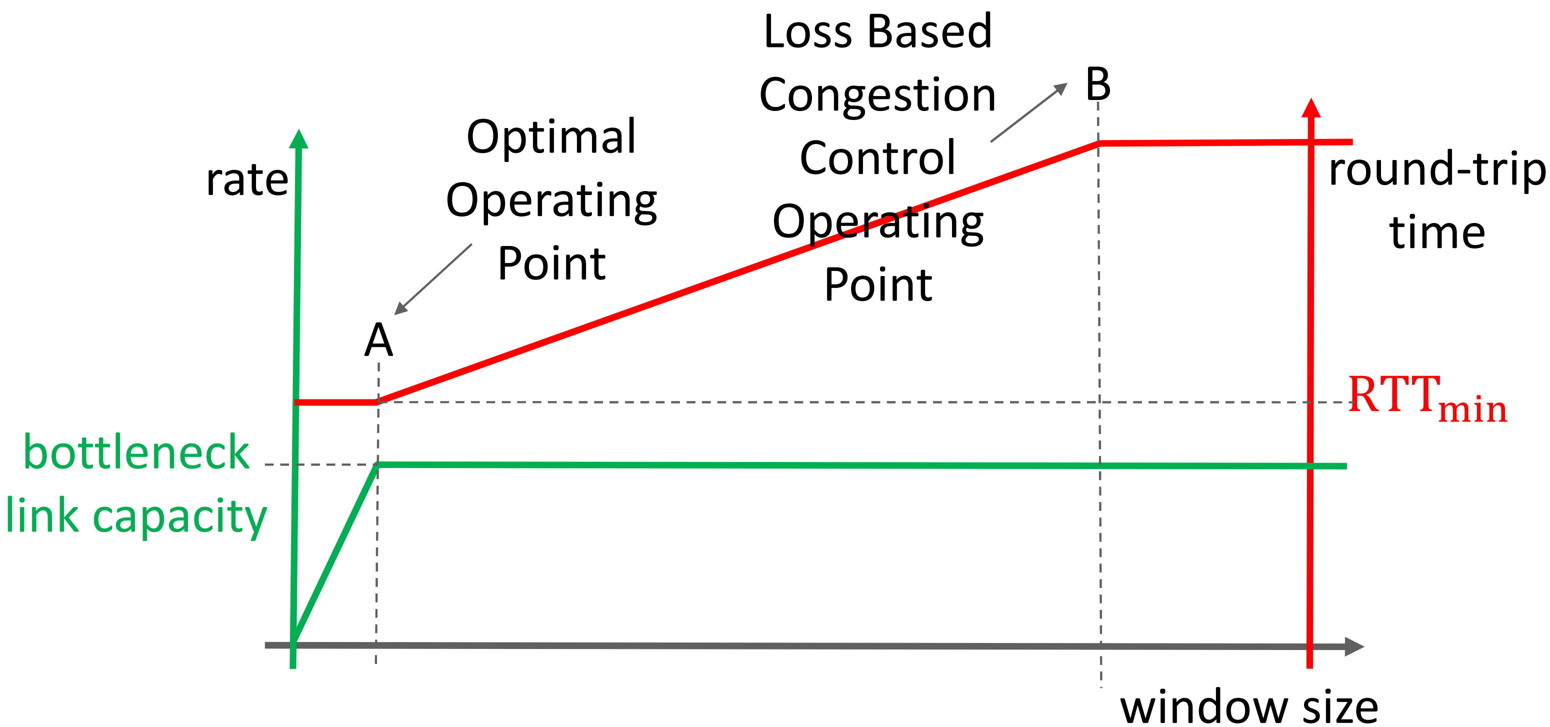
A TCP Cubic connection gets more throughput than TCP Reno when bit-rate and RTT are large



Other Cubic details: W_{max} computation 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. ECN and AQM: The Bufferbloat Syndrom

Using loss as a congestion indication has major drawback: losses to application + latency due to bufferbloat.



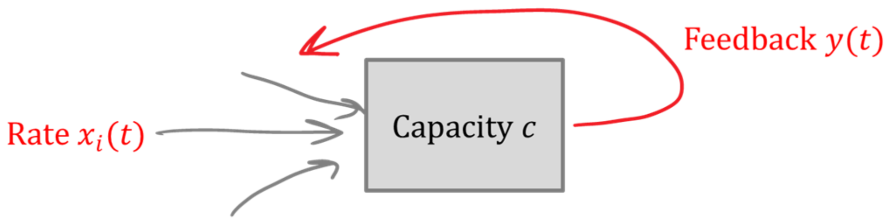
From : N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-Based Congestion Control,” ACM Queue, vol. 14, no. 5, pp. 50:20–50:53, Oct. 2016.

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 is just large enough to fill the available bottleneck link capacity, the bottleneck link is fully utilized and the queuing delay is still 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 and RED

Explicit Congestion Notification (ECN) aims at avoiding these problems

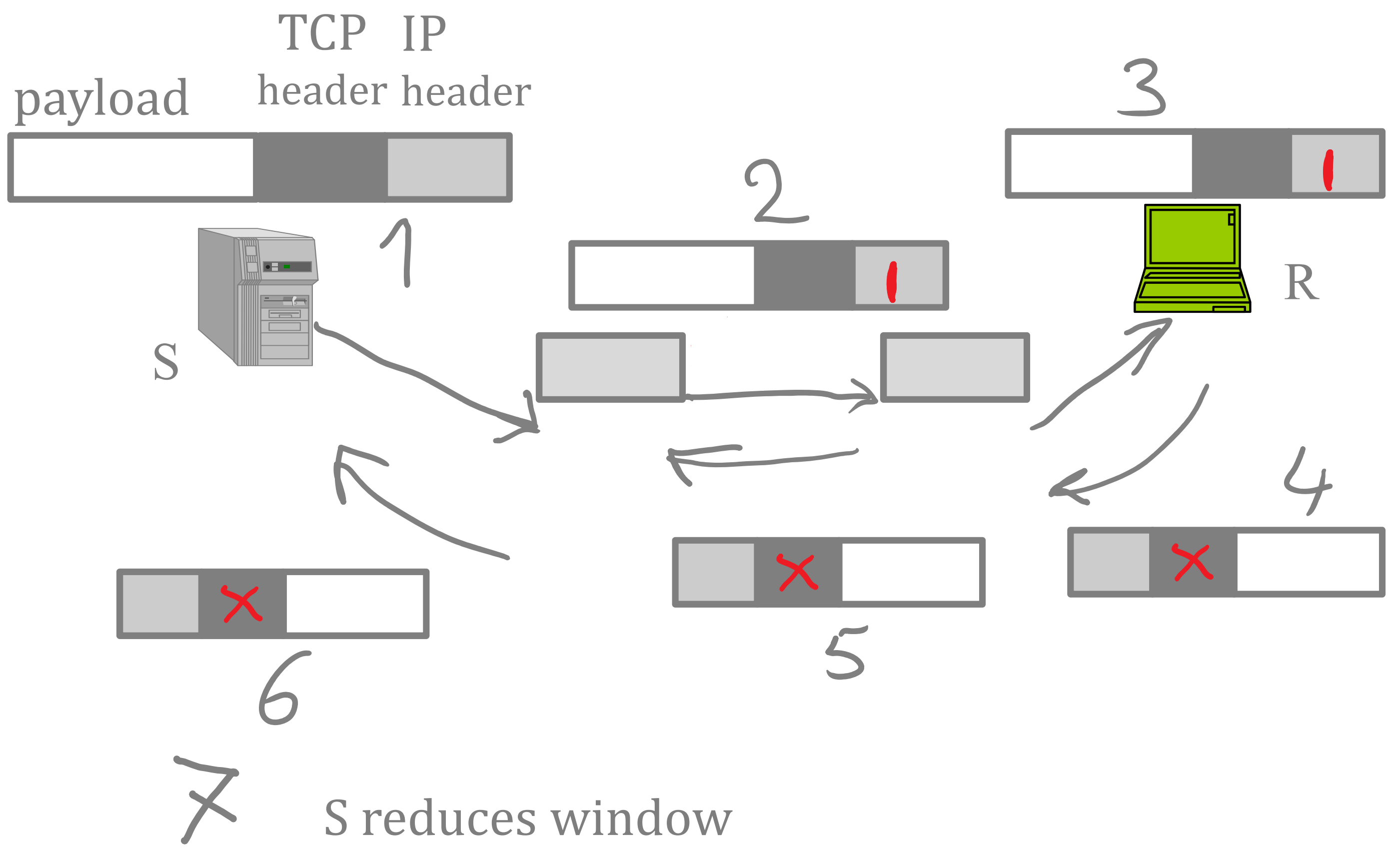


What ? signal congestion without dropping packets (≈ DECbit)

How ? router marks packet instead of dropping
TCP destination echoes the mark back to source

At the source, TCP interprets a marked packet as if there would be a loss detected by fast retransmit

Explicit Congestion Notification (ECN)



S reduces window

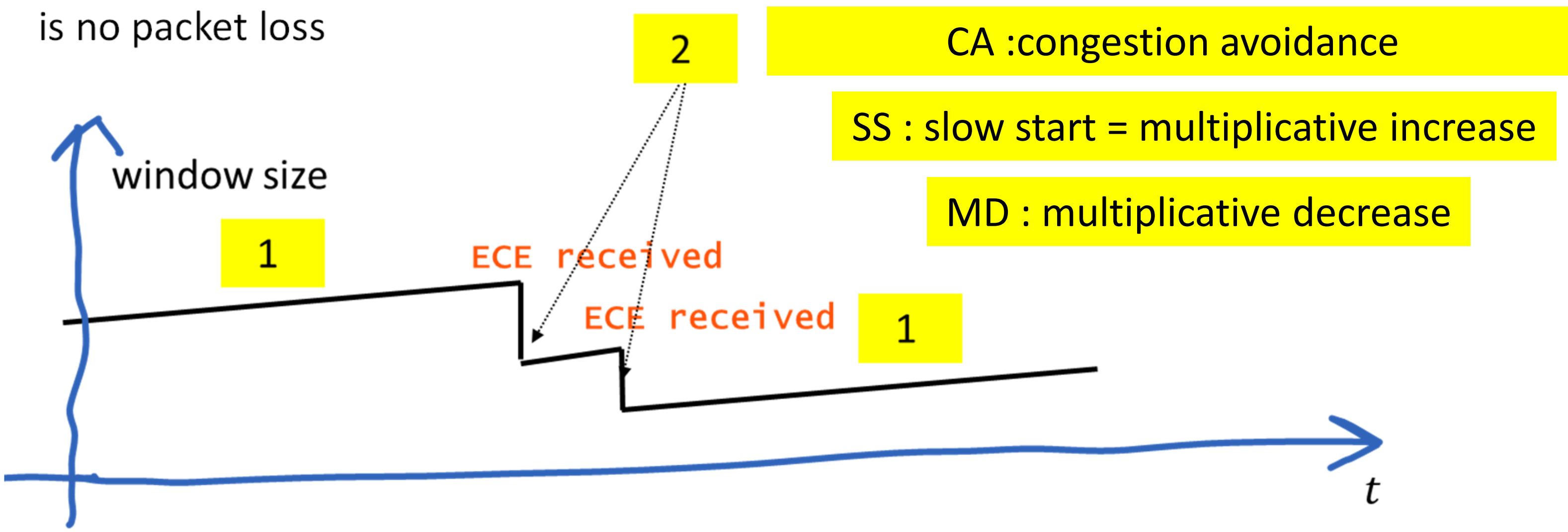
1. S sends a packet using TCP
2. Packet is received at congested router buffer; router marks the Congestion Experienced (CE) bit in IP header
3. Receiver sees CE in received packet and set the ECN Echo (ECE) flag in the TCP header of packets sent in the reverse direction
4. 5,6 Packets with ECE are received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

Put correct labels

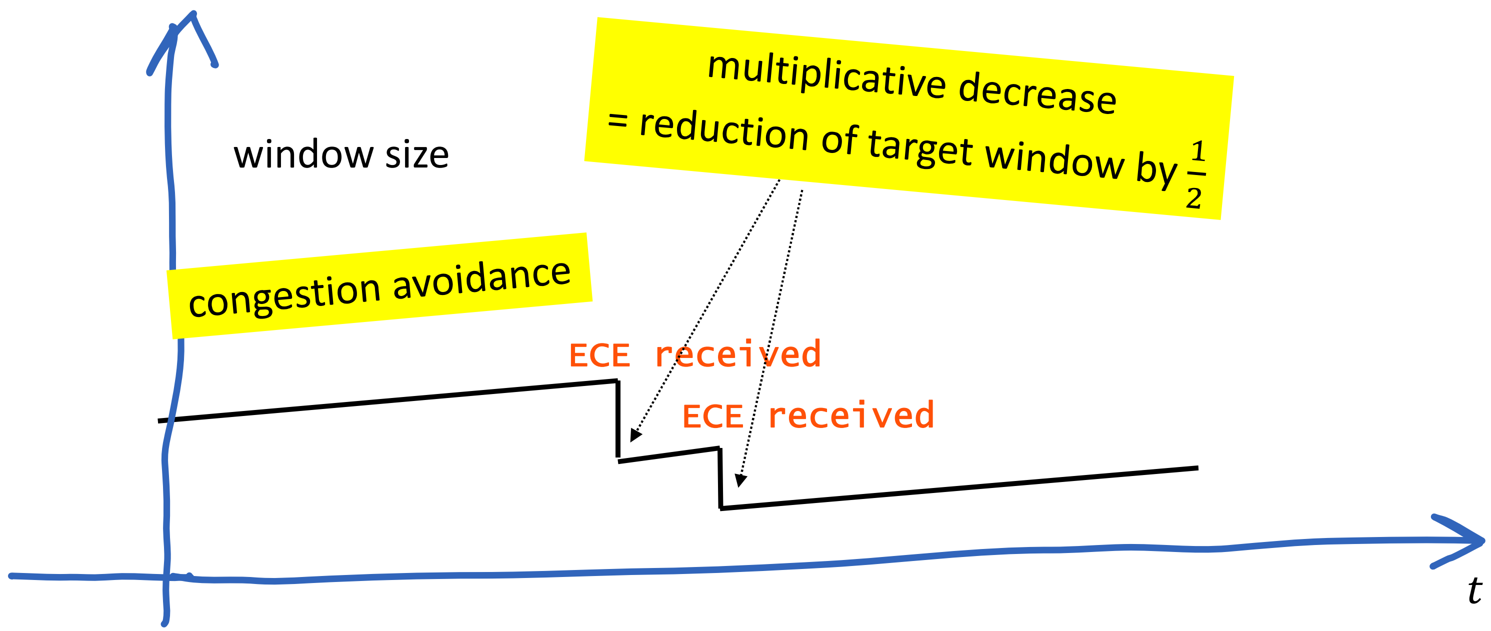
assume TCP with ECN is used and there is no packet loss



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

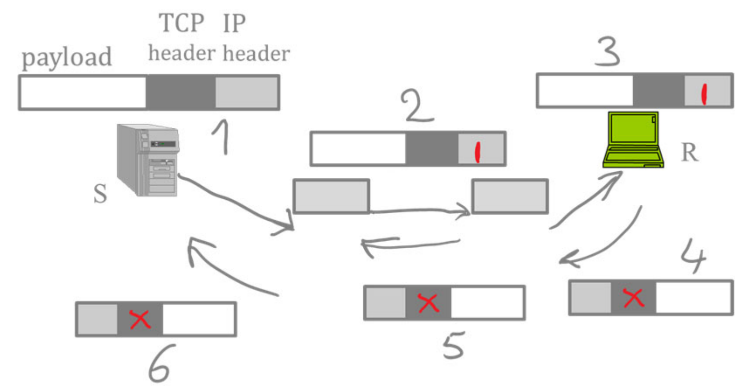
Solution

Answer C



ECN flags in IP and TCP headers

2 bits in IP header : 4 possible codepoints:



- non ECN Capable (non ECT)
- ECN capable ECT(0) and ECT(1)
  - historically used at random; today used to differentiate congestion control (TCP Cubic vs DCTCP)
- ECN capable and congestion experienced (CE)
- If congested, router marks ECT(0) or ECT(1) packets; discards non ECT packets

2 bits in TCP header

- ECE is set by R to inform S of congestion
- CWR (congestion window reduced) set by S to inform R that ECE was received and R can stop sending ECE until receiver receives a TCP header with CWR set
- When receiving ECE, S reduces window only once per RTT. R sets ECE in all TCP headers until CWR is received or until new CE packet received.

RED (Random Early Detection)

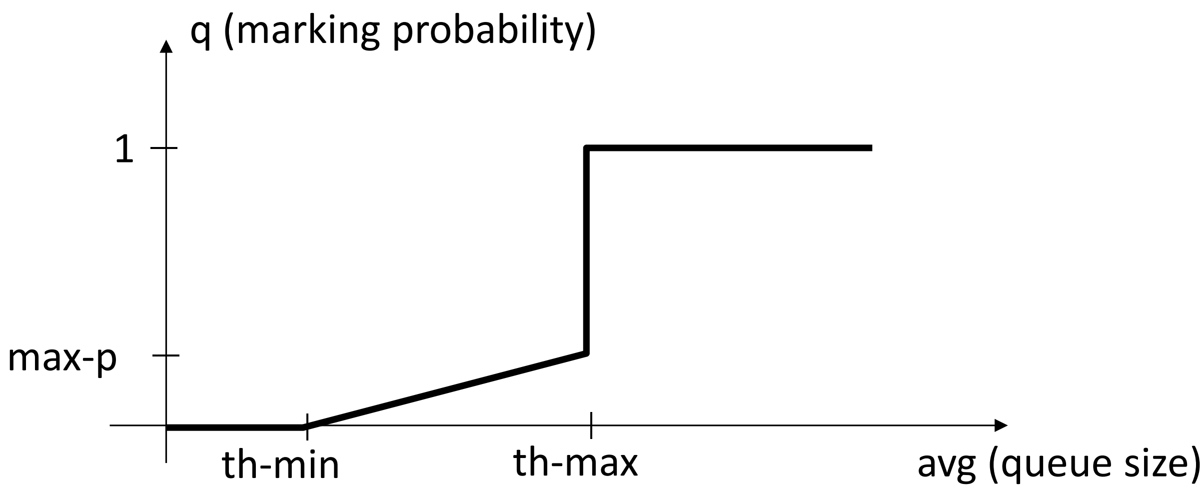
Why ? when to mark a packet with ECN

How ? queue estimates its average queue length

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

incoming packet is marked with probability given by RED curve

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



RED is an example of Active Queue Management (AQM)

Active Queue Management

AQM can also be applied even if ECN is not supported

In such a case, e.g. with RED, a packet is dropped with proba q computed by the RED curve

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

Expected benefit

- avoid bufferbloat – reduce latency
- avoid irregular drop patterns

Contrast to passive queue management = drop a packet when queue is full = "Tail Drop"

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 no packet loss due to congestion
C. there is no packet loss due to congestion in routers
D. none of the above
E. I don’t know

Solution

Answer C

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

However there might be congestion losses in bridges, and there might be non-congestion losses (transmission errors)

9. Other Cool Stuff

Data Center TCP

TCP-BBR

Per Class Queuing

TCP-friendly apps

Data Centers and TCP

What is a data center ?

a room with lots of racks of PCs and switches

youtube, CFF.ch, switchdrive, etc

What is special about data centers ?

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

What is your preferred combination 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

Solution

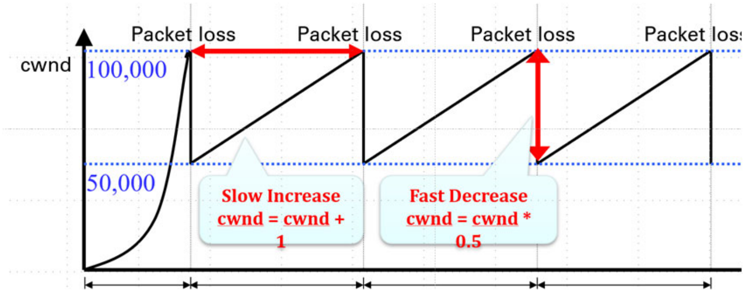
Answers B or D

Without ECN there will be bufferbloat, which means high latency for short flows

Cubic has better performance than Reno when bandwidth-delay product is large, which may occur in data centers.

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. Same as Reno except avoids the brutal multiplicative decrease by 50%.

How ?

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

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

Solution

Answer B.

If latency is very small, Cubic with ECN has same 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; 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.

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 $\times$ memory size) should scale faster than link rate, which is technologically not feasible.

- In internet core links (100 Gb/s, 1 Gb/s): buffer drain time decreases, is now fraction of ms, much less than RTT
  $\Rightarrow$ impossible to react correctly within round trip time $\Rightarrow$ feedback control is inadequate
- Access network (1 Gb/s): buffer drain time increases to 10s of second
  $\Rightarrow$ Bufferbloat unless ECN is used

TCP-BBR
Bottleneck Bandwidth and RTT

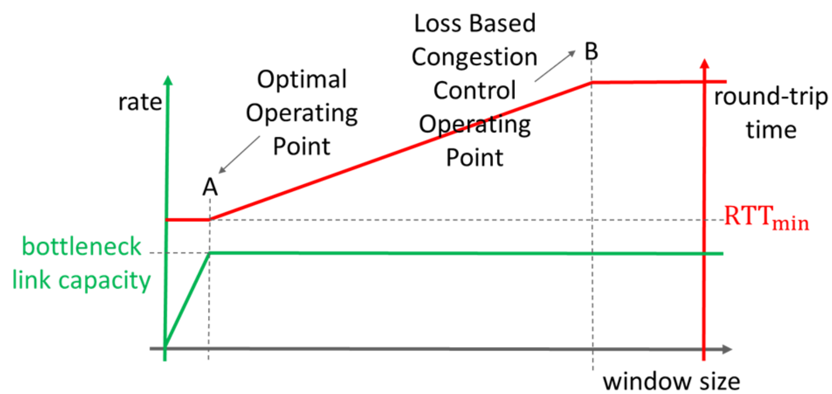
TCP-BBR published by Google in 2016

What ? Avoid per packet feedback

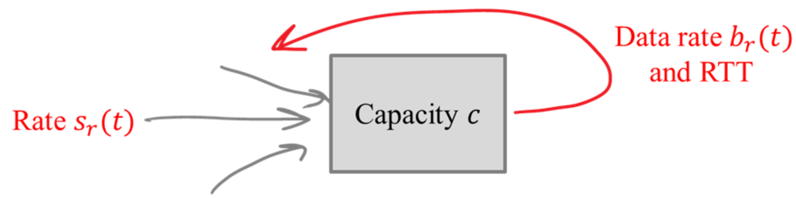
How ? BBR-TCP source:

1. controls rate (not window) (*pacing*)
2. estimates the bottleneck bandwidth and the min RTT separately
3. 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



Operation of BBRv1



source views network as a single link
(the bottleneck link)

estimates RTT by taking the min over the last 10 sec

estimates bottleneck rate (bandwidth) b_r= max of delivery rate over last 10 RTTs; delivery rate = amount of acked data per Δt

send data at rate $b_r \times c(t)$
where $c(t) = 1.25; 0.75, ; 1; 1; 1; 1; 1; 1$ i.e. c(t) is 1.25 during one RTT, then 0.75 during one RTT,then 1 during 6 RTTs ("probe bandwidth" followed by "drain excess" followed by steady state)

data is paced using a spacer at the source

max data in flight is limited to $2 \times b_r \times RTT_{est}$ and by the offered window

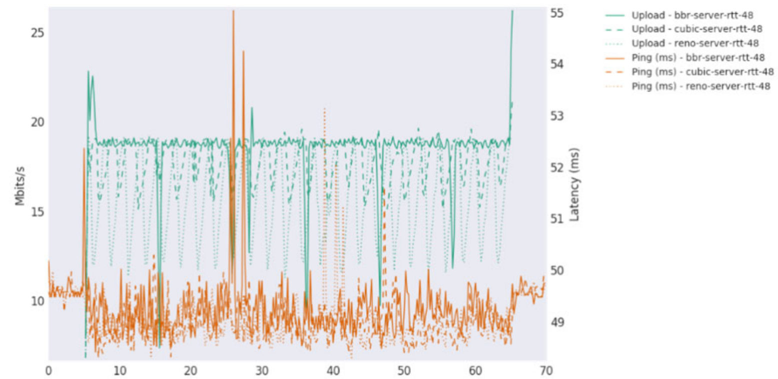
there is also a special startup phase with exponential increase of rate

no reaction to losses or ECN

Performance of BBRv1

Google and other data center companies report improvement on throughput

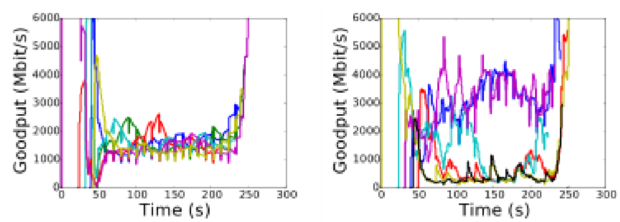
But…BBRv1 takes no feedback from network – no reaction to loss or ECN



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

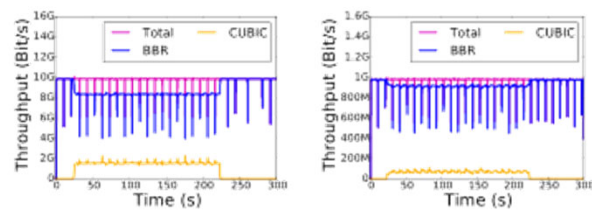
[Hock et al, 2017] find that BBRv1's estimated bottleneck bandwidth ignores how many flows are competing

-- fairness issues may exist inside BBRv1 flows with different RTTs and versus other TCPs



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

Fig. 7: Goodput of six BBR flows :



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

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

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 theoretical analysis is still to be performed.

Google started BBRv2 to address these and other shortcomings – stay tuned !

Class Based Queuing

In general, all flows compete in the Internet using the congestion control method of TCP. It is possible to modify the competition and separate flows using per-class queuing.

E.g. routers classify packets (using an access list)

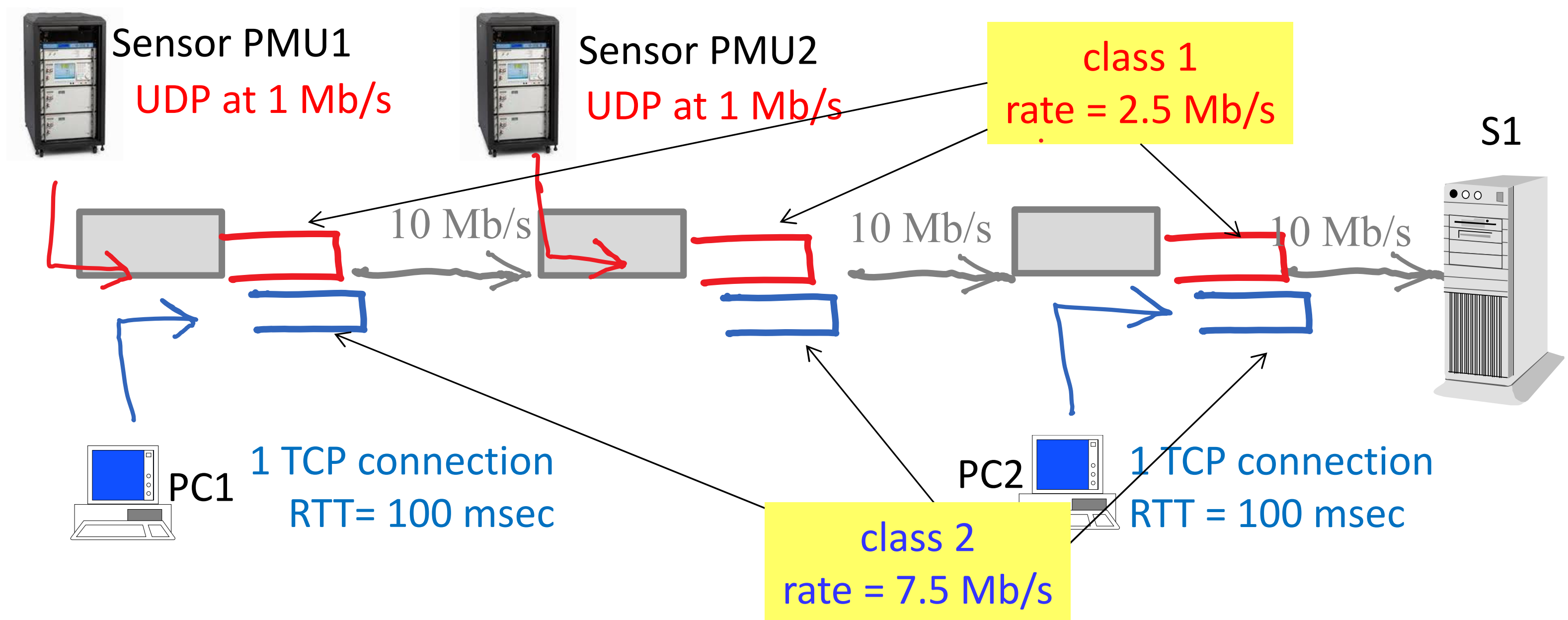
- each class is guaranteed a rate
- classes may exceed the guaranteed rate by *borrowing* from other classes if there is spare capacity

This 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.

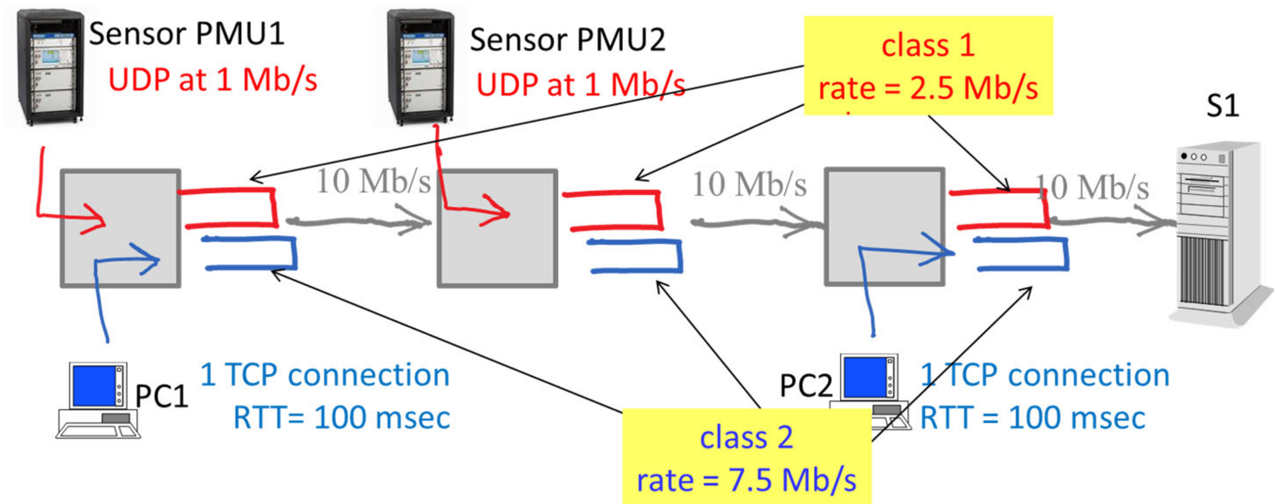
Used in enterprise or industrial networks to support non congestion controlled flows (e.g. real-time flows); in provider networks to separate customers / isolate suspicious flows (network virtualization).

Example of Class-Based Queuing



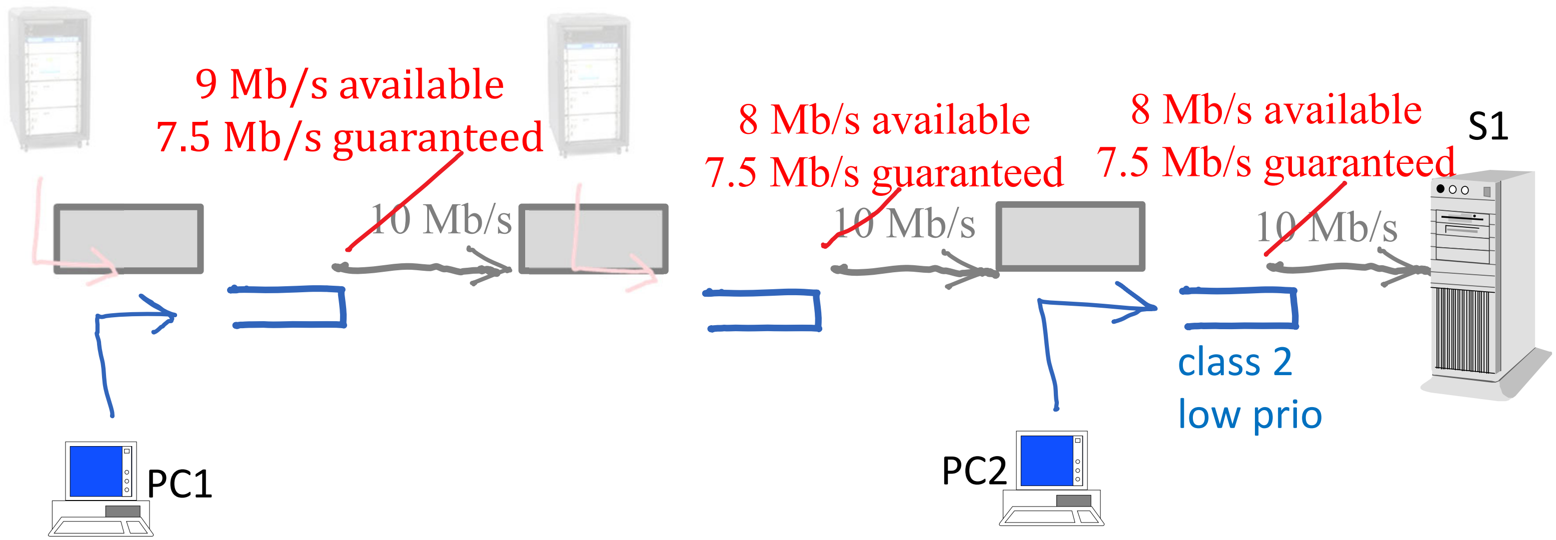
Class 1 is guaranteed a rate of 2.5 Mb/s; can exceed this rate by borrowing capacity available from the total 10 Mb/s if class 2 does not need it. Class 2 is guaranteed a rate of 7.5 Mb/s; can exceed this rate by borrowing capacity available from the total 10 Mb/s if class 1 does not need it

Which rates will PC1 and PC2 achieve ?



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

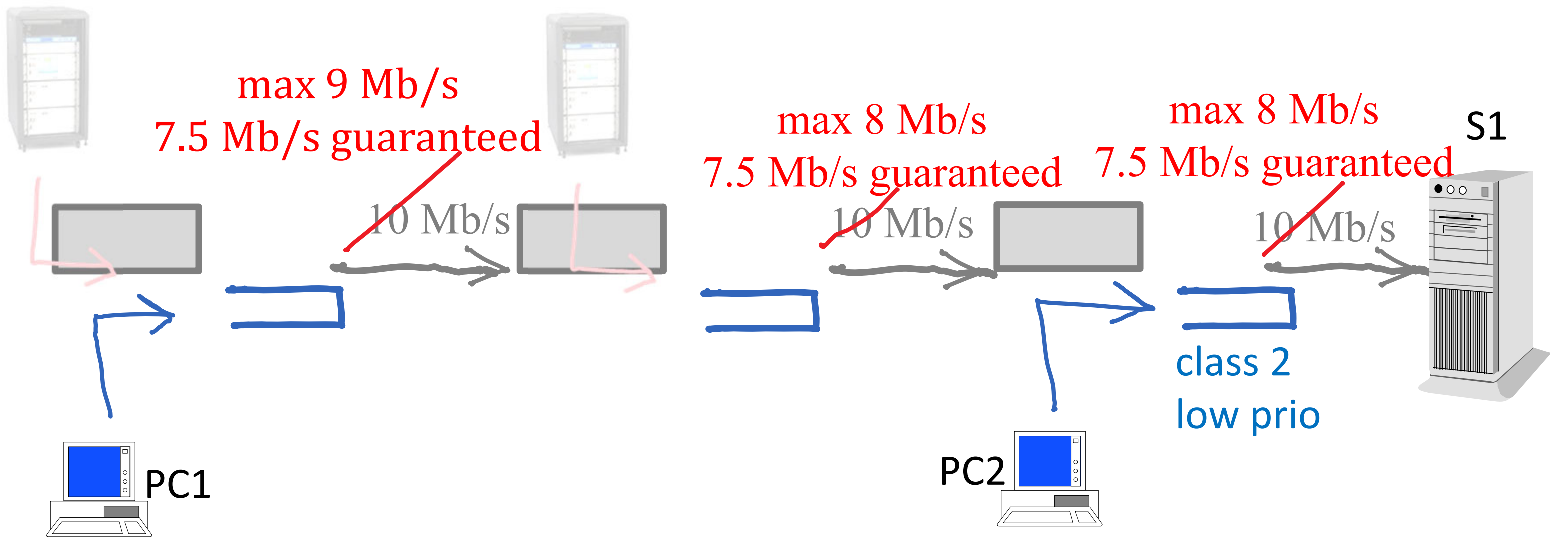
Solution



PC1 and PC2 see this network ↑

Since PMU1 and PMU2 stream at 1 Mb/s and class 2 may borrow, 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, $x_1 + x_2 \leq 8$ Mb/s, $x_1 + x_2 \leq 8$ Mb/s.

Thus TCP solves the 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.

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

Conclusion

Congestion control is in TCP or in a TCP-friendly UDP application.

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

TCP uses loss as congestion signal.

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

ECN can be used to avoid bufferbloat – it replaces loss by an explicit congestion signal; partly deployed in the internet; part of Data Center TCP.

Class based queuing is used to separate flows in enterprise networks or classes of flows in provider networks.