CS 3205 COMPUTER NETWORKS

JAN-MAY 2020

LECTURES 21: 13TH APR 2020

Text book and section(s) covered in this lecture:
Book Kurose and Ross – Sections 3.6

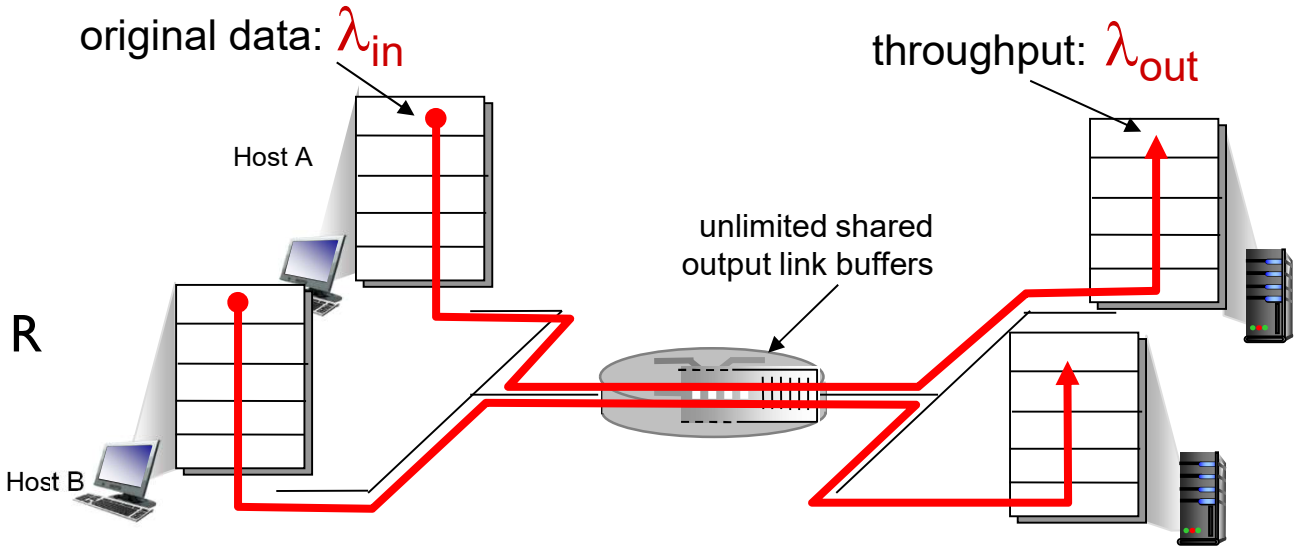
Principles of congestion control

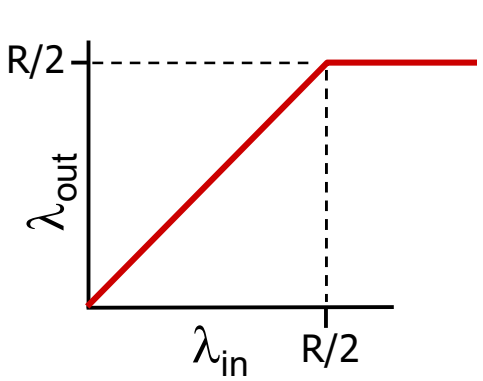
congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

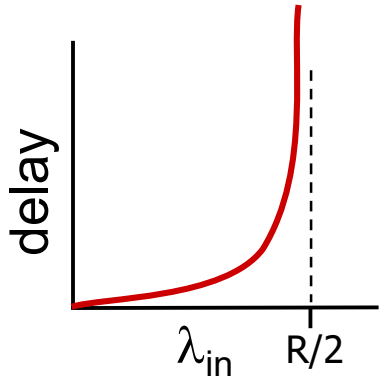
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission





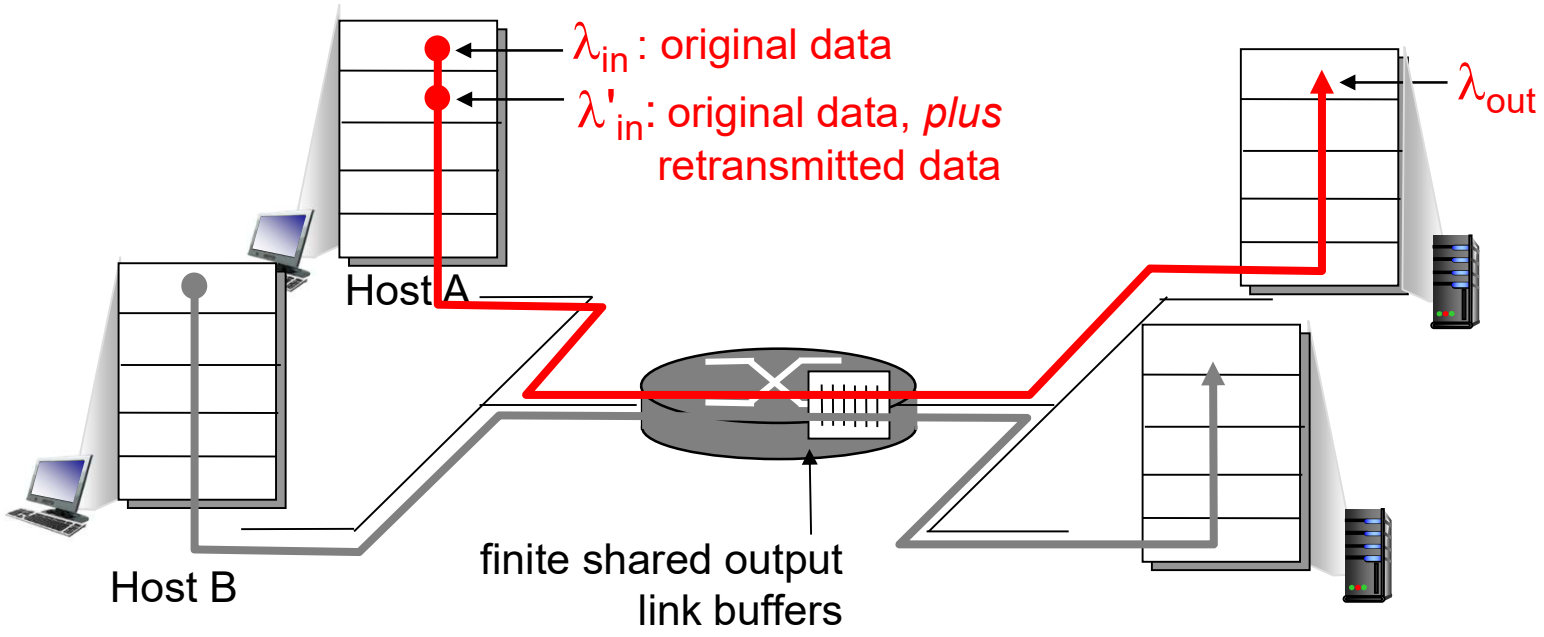
- maximum per-connection throughput: R/2



- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

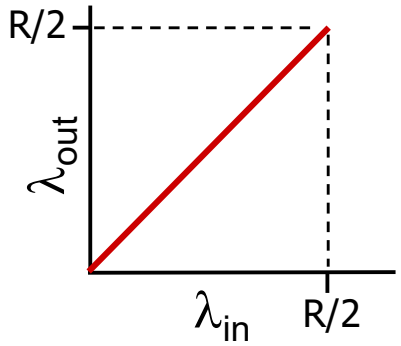
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

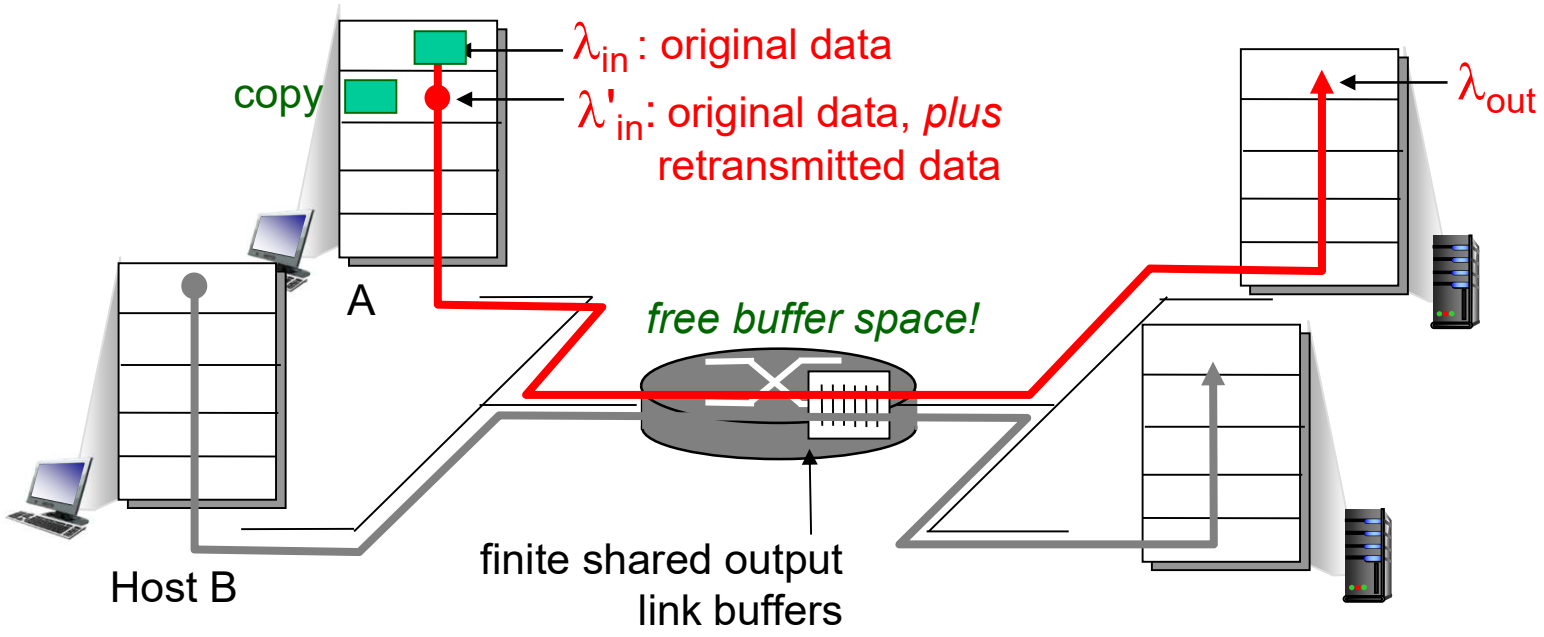


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available



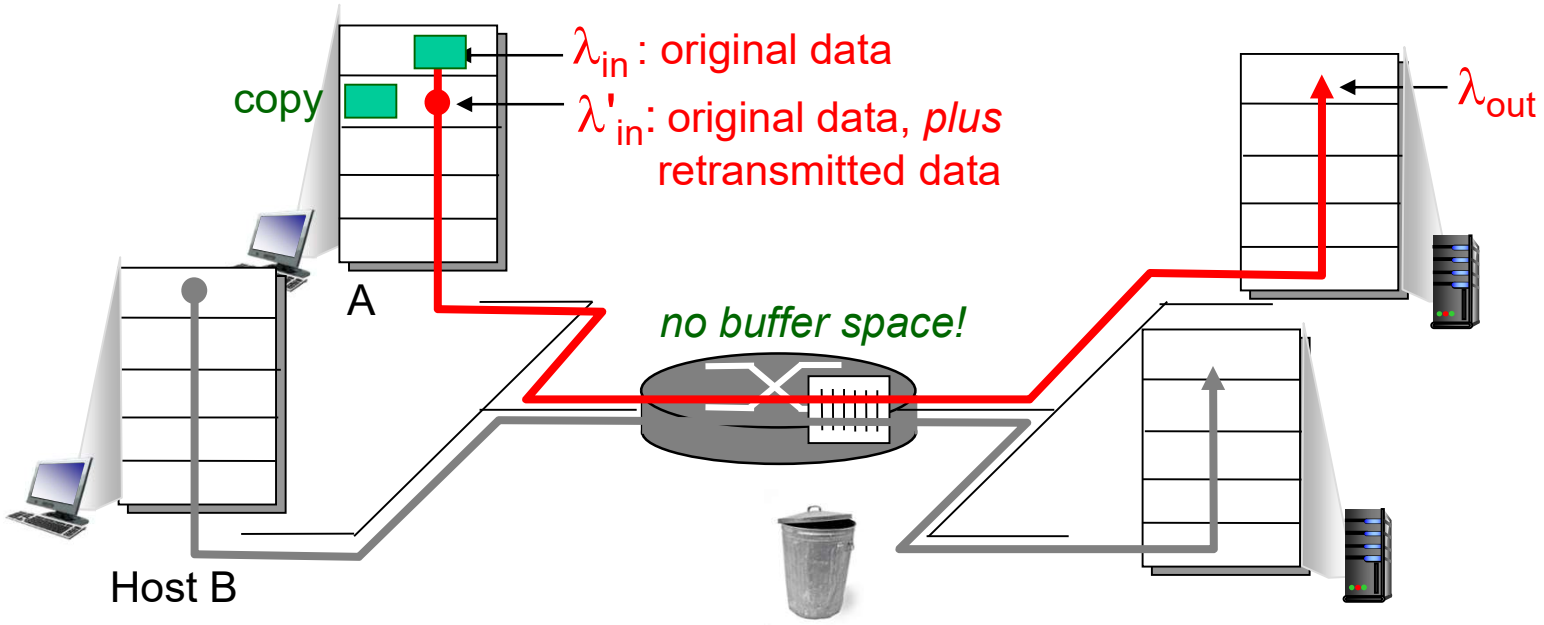


Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

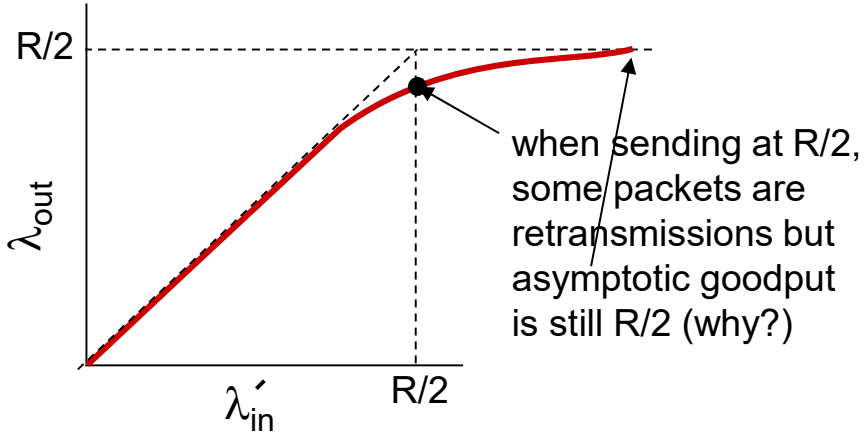


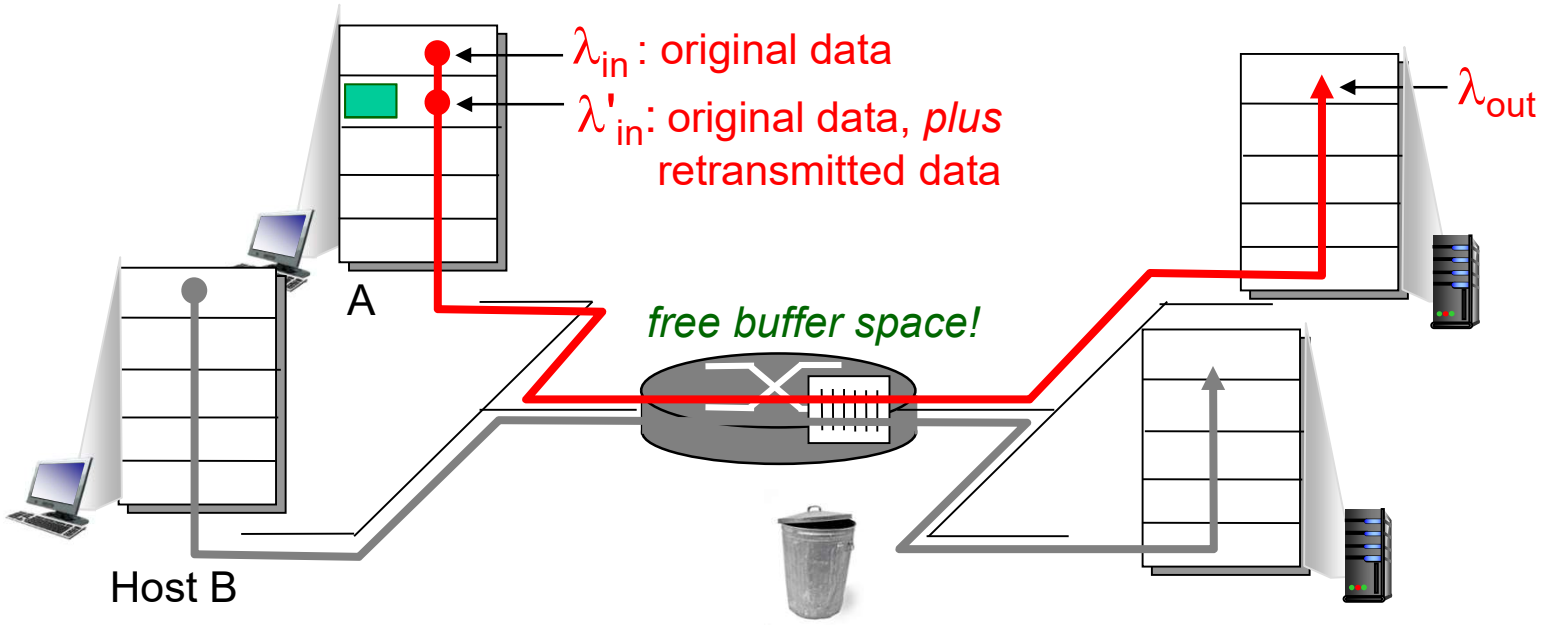
Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

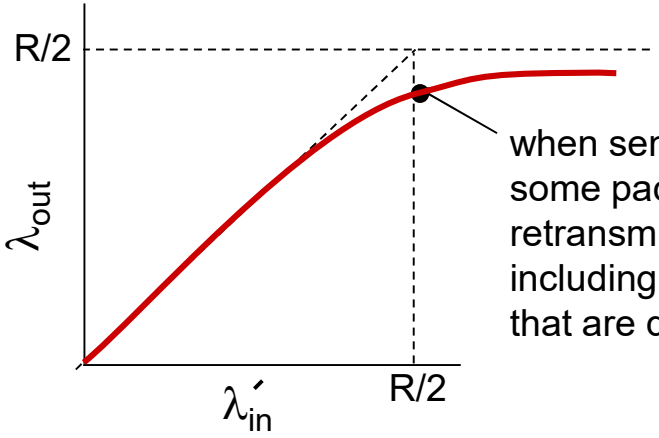


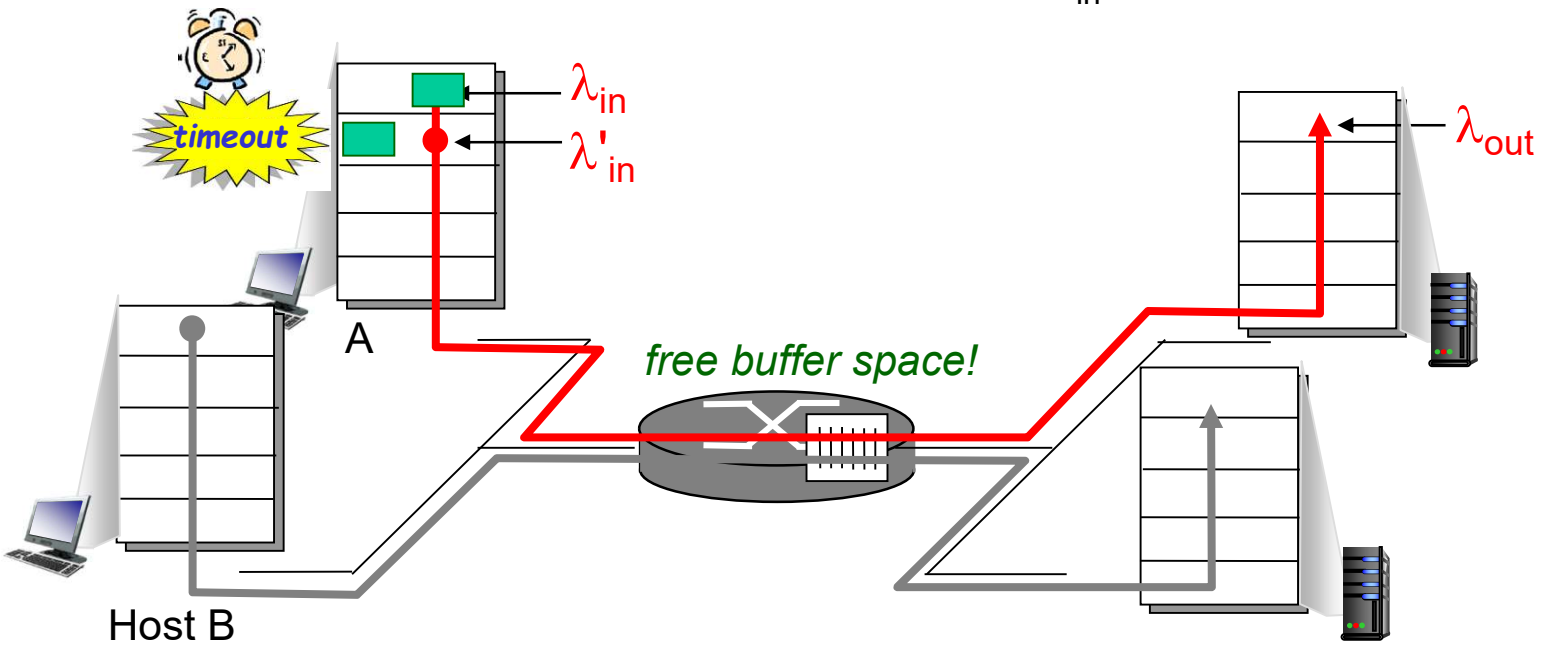


Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

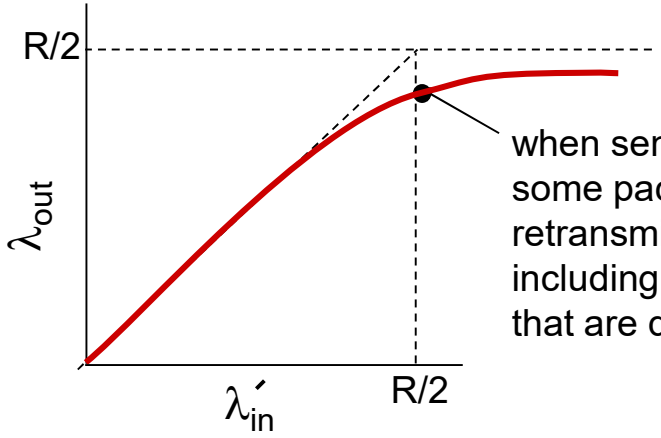




Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

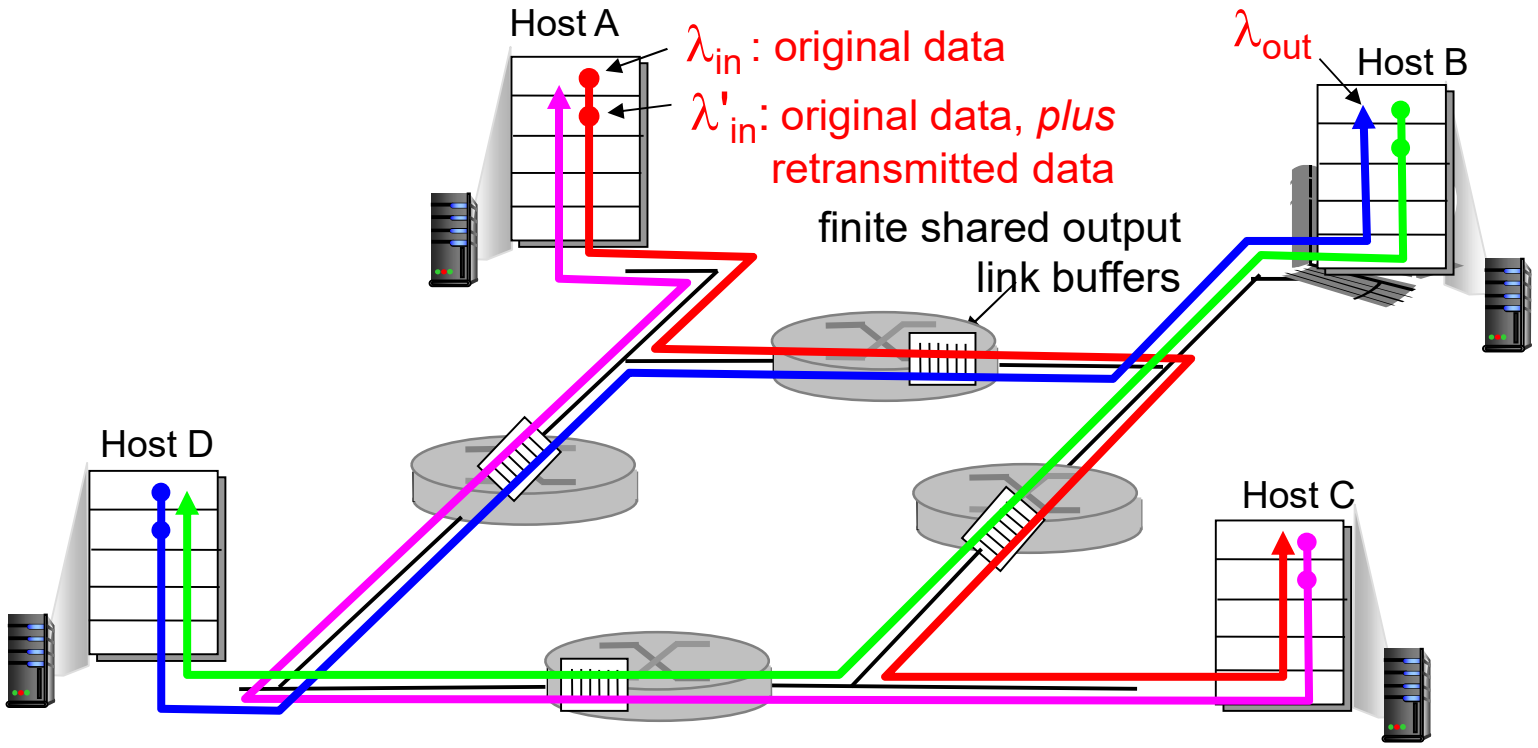
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

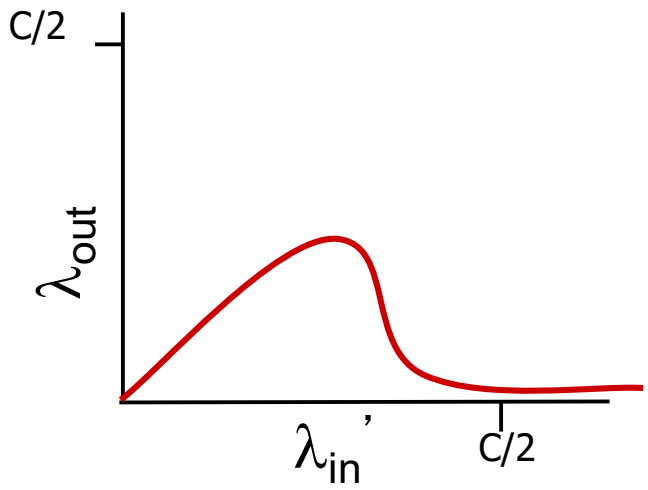
- four senders
- multihop paths
- timeout/retransmit

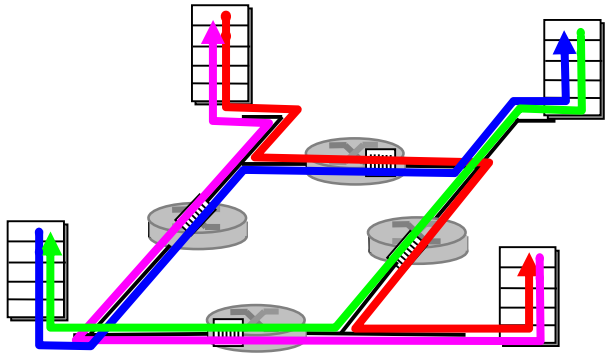
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Causes/costs of congestion: scenario 3





another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

two broad approaches towards congestion control:

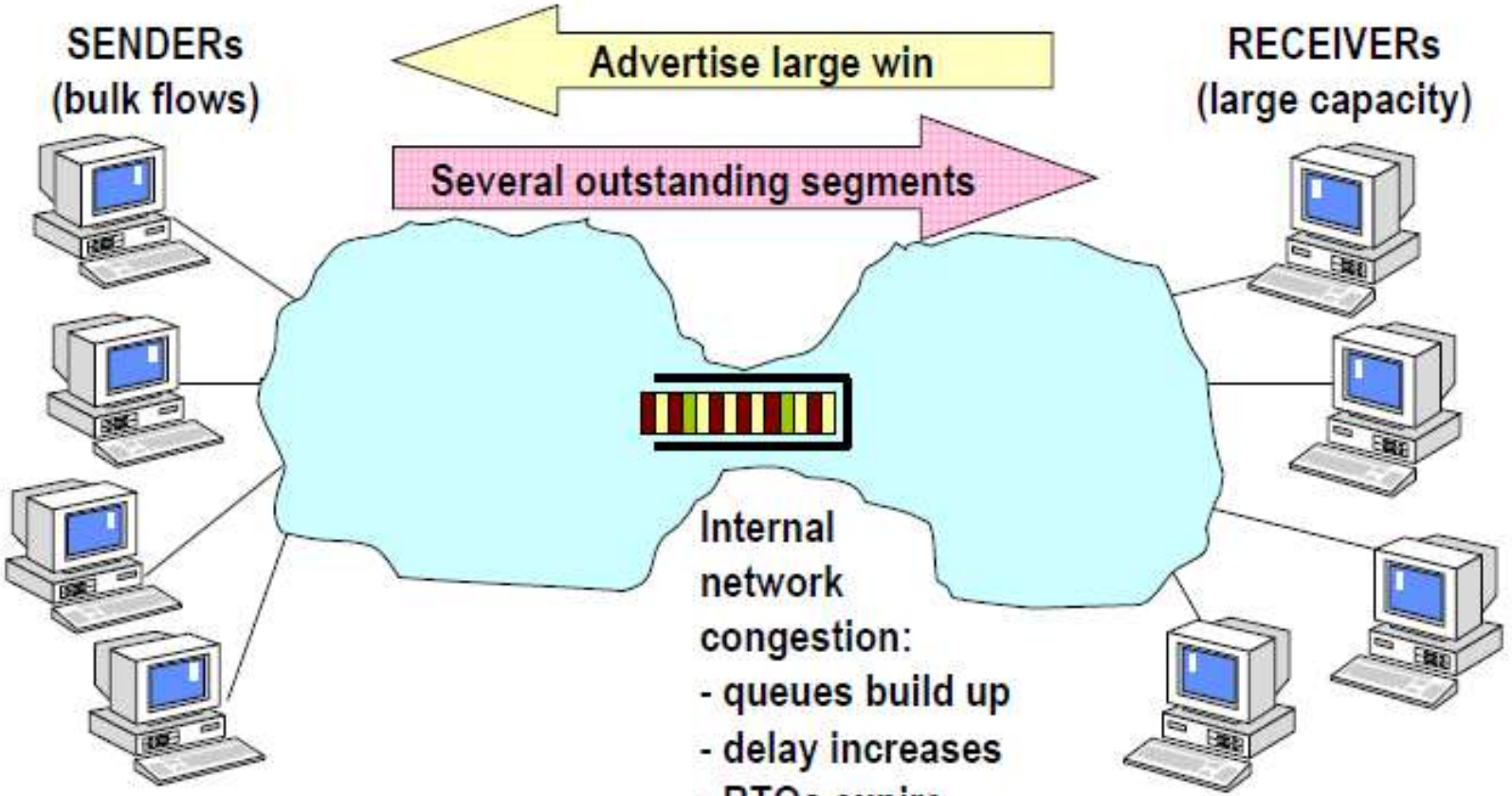
end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

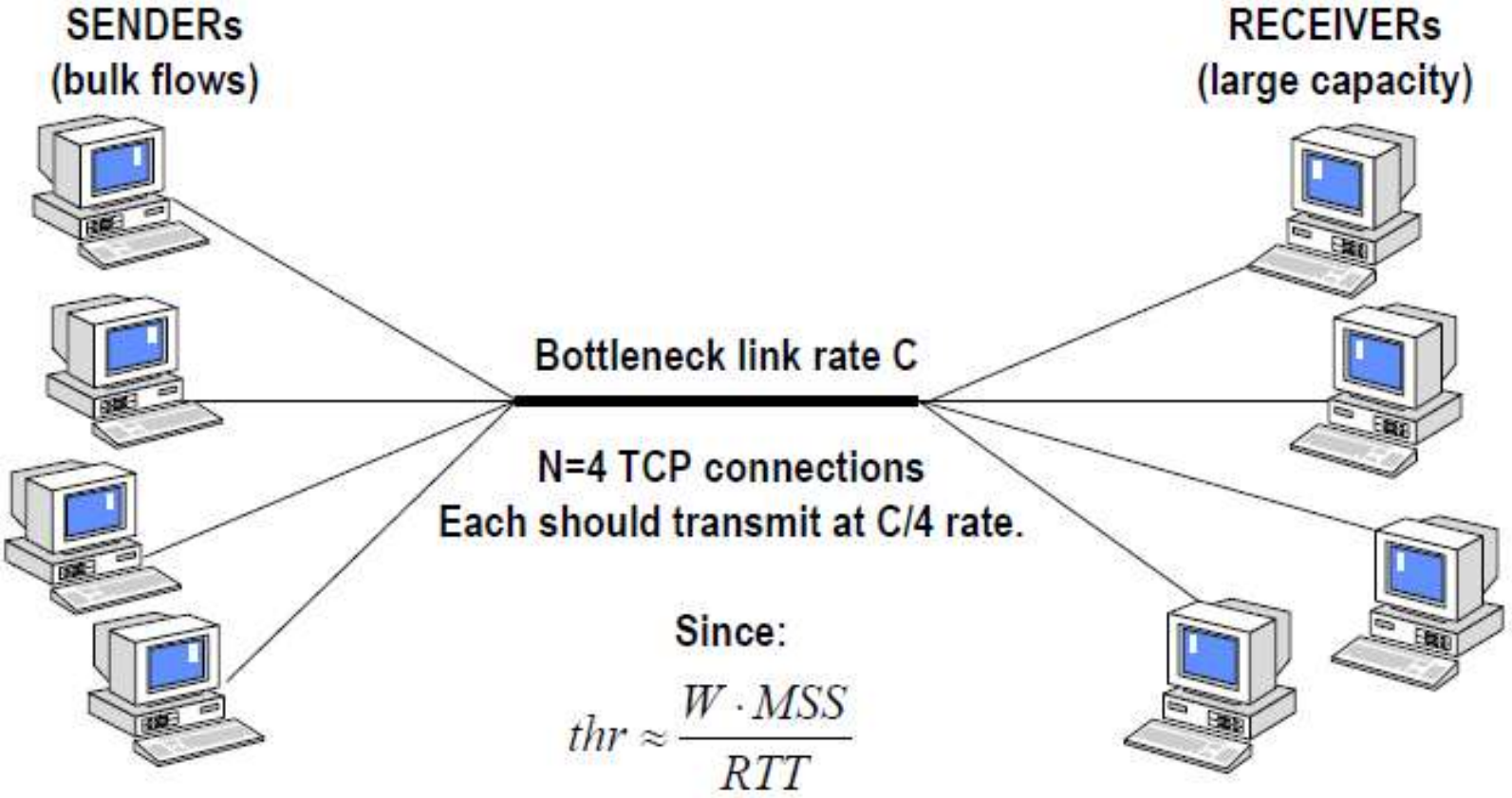
network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

The problem of congestion



The goal of congestion control



Each should adapt W accordingly…
How sources can be lead to know the RIGHT value of W??

TCP – IETF RFCs

- TRANSMISSION CONTROL PROTOCOL - RFC 793 (Sept 1991)
- TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms – RFC 2001, (Jan 1997)
- TCP Congestion Control – RFC 2581(April 1999)
  - TCP Congestion Control – RFC 5681 (Sept 2009)
- Defending against Sequence Number Attacks – RFC 6528 (Feb 2012)
- The Addition of Explicit Congestion Notification (ECN) to IP – RFC 3168 (Sept 2001)

- A Roadmap for Transmission Control Protocol (TCP) Specification Documents – RFC 7414 (Feb 2015)

History of congestion control

➔ **Before 1986: the Internet meltdown!**

- ⇨ No mechanisms employed to react to internal network congestion

➔ **1986: Slow Start + Congestion avoidance**

- ⇨ Van Jacobson, TCP Berkeley
- ⇨ Proposes idea to make TCP reactive to congestion

➔ **1988: Fast Retransmit (TCP Tahoe)**

- ⇨ Van Jacobson, first implemented in 1988 BSD Tahoe release

➔ **1990: Fast Recovery (TCP Reno)**

- ⇨ Van Jacobson, first implemented in 1990 BSD Reno release

➔ **1995-1996: TCP NewReno**

- ⇨ Floyd (based on Hoe's idea), RFC 2582
- ⇨ Today the de-facto standard

TCP approach for detecting and controlling congestion

- **IP protocol does not implement mechanisms to detect congestion in IP routers**
  - →Unlike other networks, e.g. ATM
- **necessary indirect ways (TCP is an end-to-end protocol)**
- **TCP approach: congestion detected by lack of acks**
  - » couldn't work efficiently in the 60s & 70s (error prone transmission lines)
  - » OK in the 80s & 90s (reliable transmission)
  - » what about wireless networks???
- **Controlling congestion: use a SECOND window (congestion window)**
  - →Locally computed at sender
  - →Outstanding segments: min(receiver_window, congestion_window)

Starting a TCP transmission

➔A new offered flow may suddenly overload network nodes

- ⇨ receiver window is used to avoid recv buffer overflow
- ⇨ But it may be a large value (16-64 KB)

➔Idea: slow start

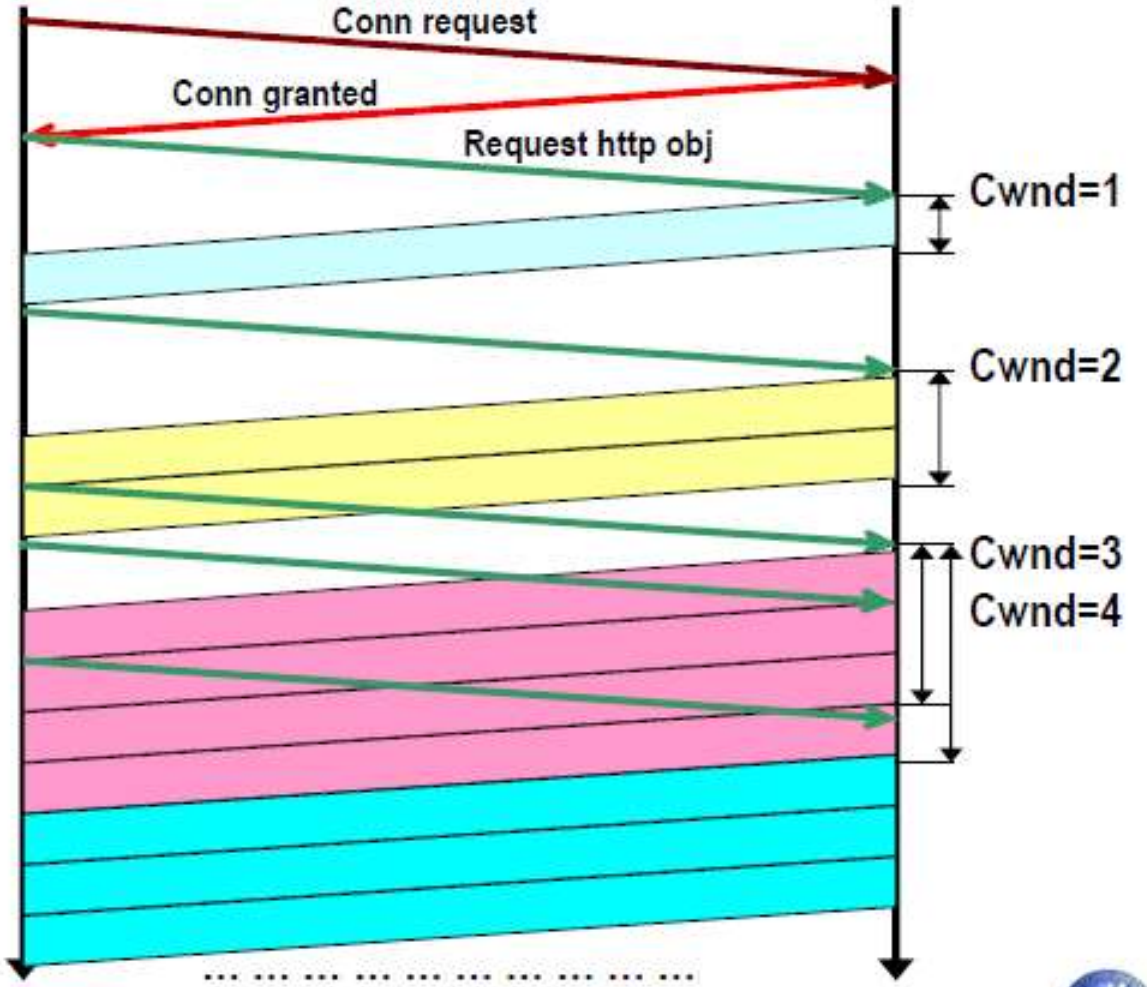
- ⇨ Start with small value of cwnd
- ⇨ And increase it as soon as packets get through
  - » Arrival of ACKs = no packet losts = no congestion

➔Initial cwnd size:

- ⇨ Just 1 MSS!
- ⇨ Recent (1998) proposals for more aggressive starts (up to 4 MSS) have been found to be dangerous

Slow start – exponential increase

- First start: set congestion window
  cwnd = 1MSS
- send cwnd segments
  - assume cwnd <= receiver win
- upon successful reception:
  - Cwnd +=1 MSS
  - i.e. double cwnd every RTT
  - until reaching receiver window advertisement
  - OR a segment gets lost



Detecting congestion and restarting

- **Segment gets lost**
  - Detected via RTO expiration
  - Indirectly notifies that one of the network nodes along the path has lost segment
    - Because of full queue
- **Restart from cwnd=1 (slow start)**
- **But introduce a supplementary control: slow start threshold**
  - sstresh = max(cwnd/2, 2MSS)
  - The idea is that we now KNOW that there is congestion in the network, and we need to increase our rate in a more careful manner…
  - ssthresh defines the "congestion avoidance" region

Congestion avoidance

➔**If cwnd < ssthresh**

⇨Slow start region: Increase rate exponentially

➔**If cwnd >= ssthresh**

⇨Congestion avoidance region : Increase rate linearly

⇨At rate 1 MSS per RTT

→Practical implementation:

cwnd += MSS*MSS/cwnd

→Good approximation for 1 MSS per RTT

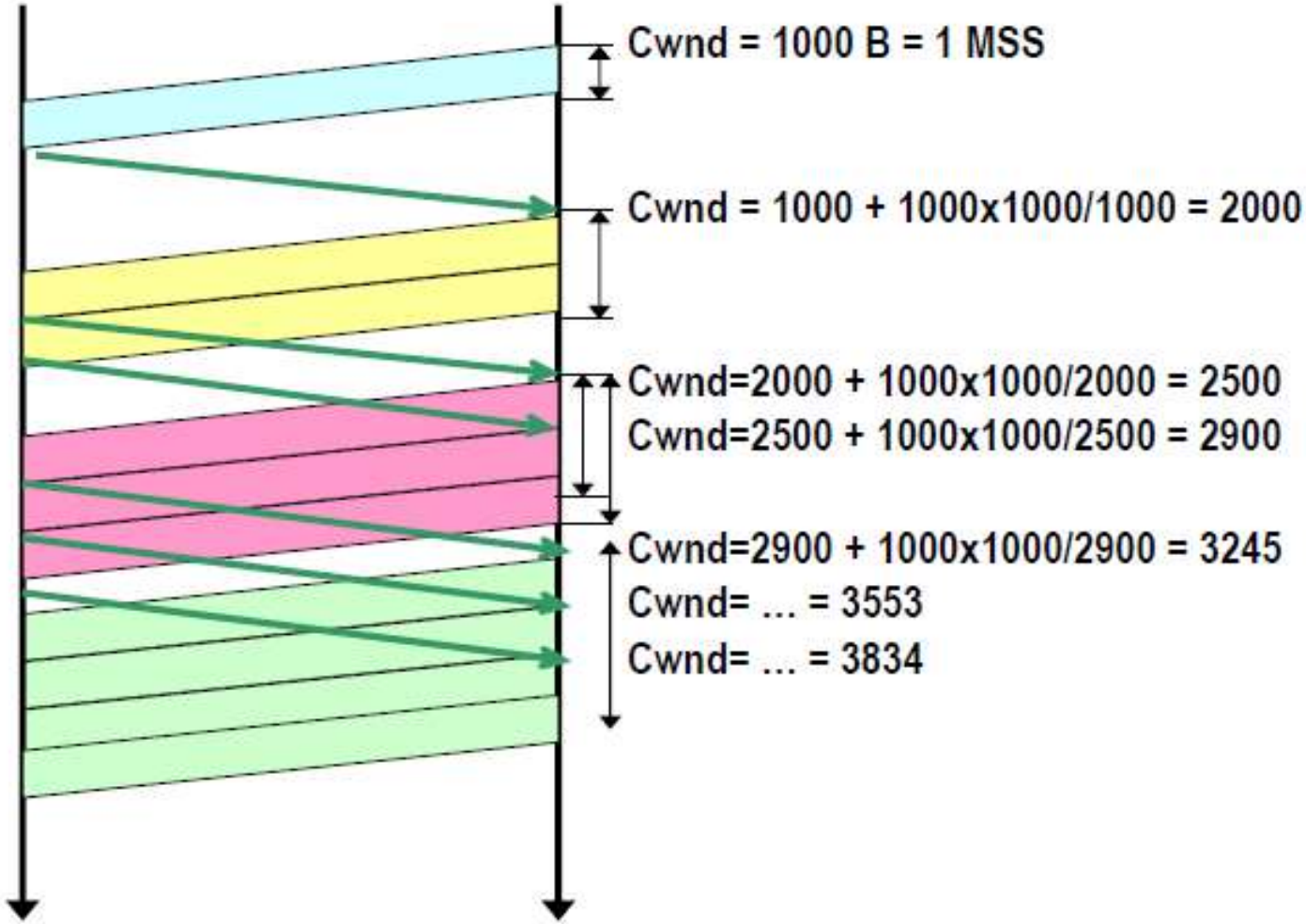
→Alternative (exact) implementations: count!!

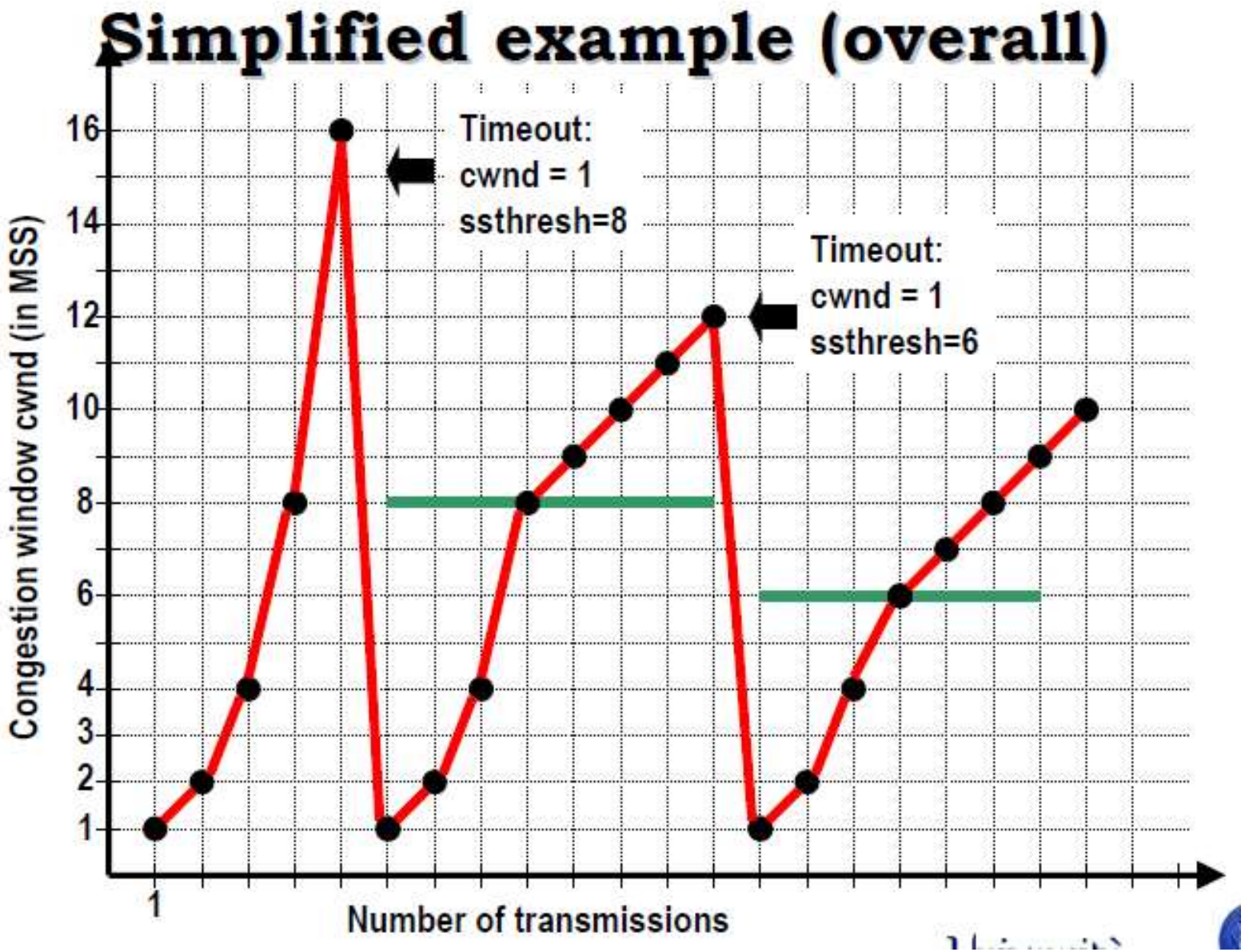
➔**Which initial ssthresh?**

» ssthresh initially set to 65535: unreachable!

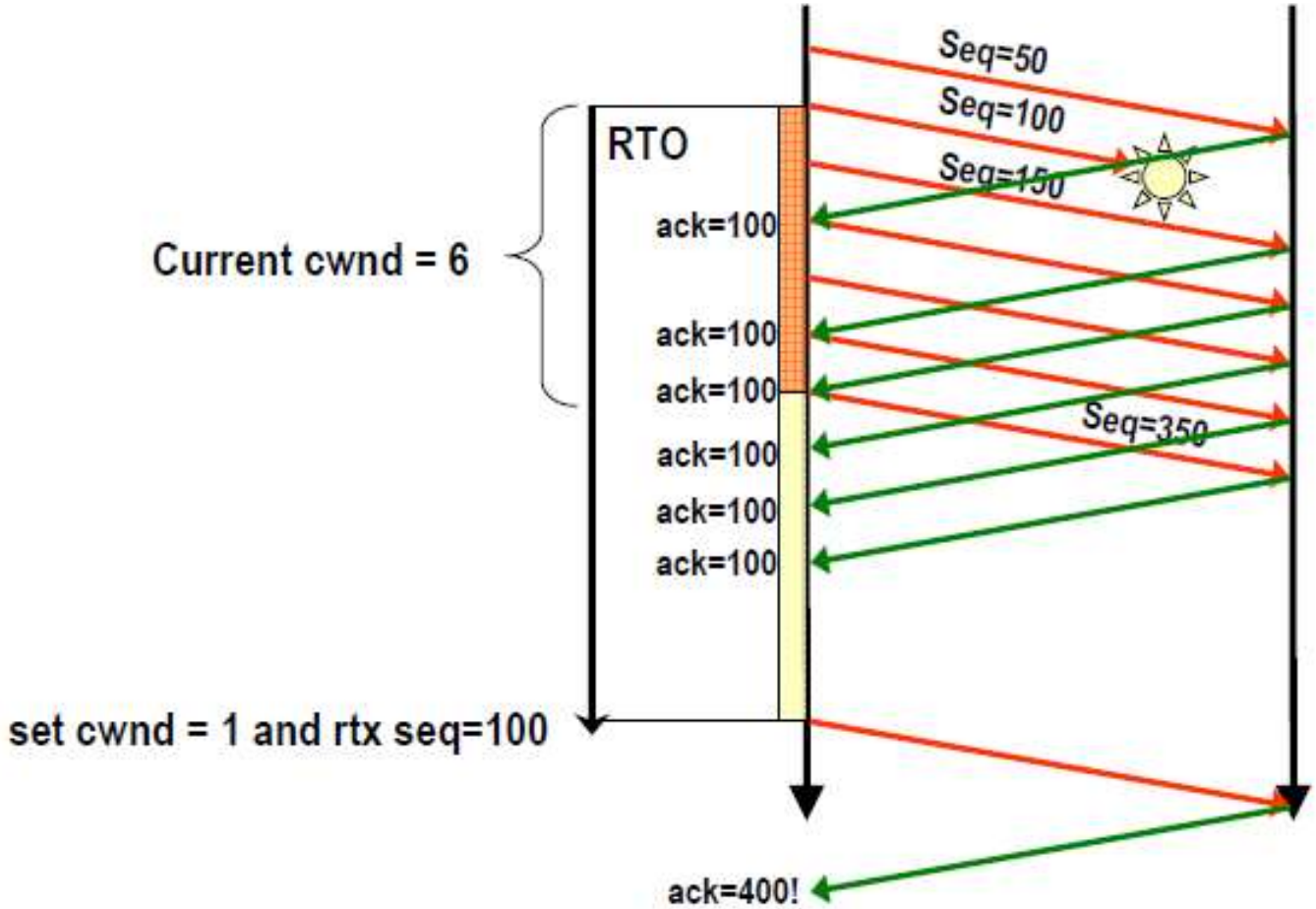
In essence, congestion avoidance is flow control imposed by sender while advertised window is flow control imposed by receiver

Congestion avoidance example



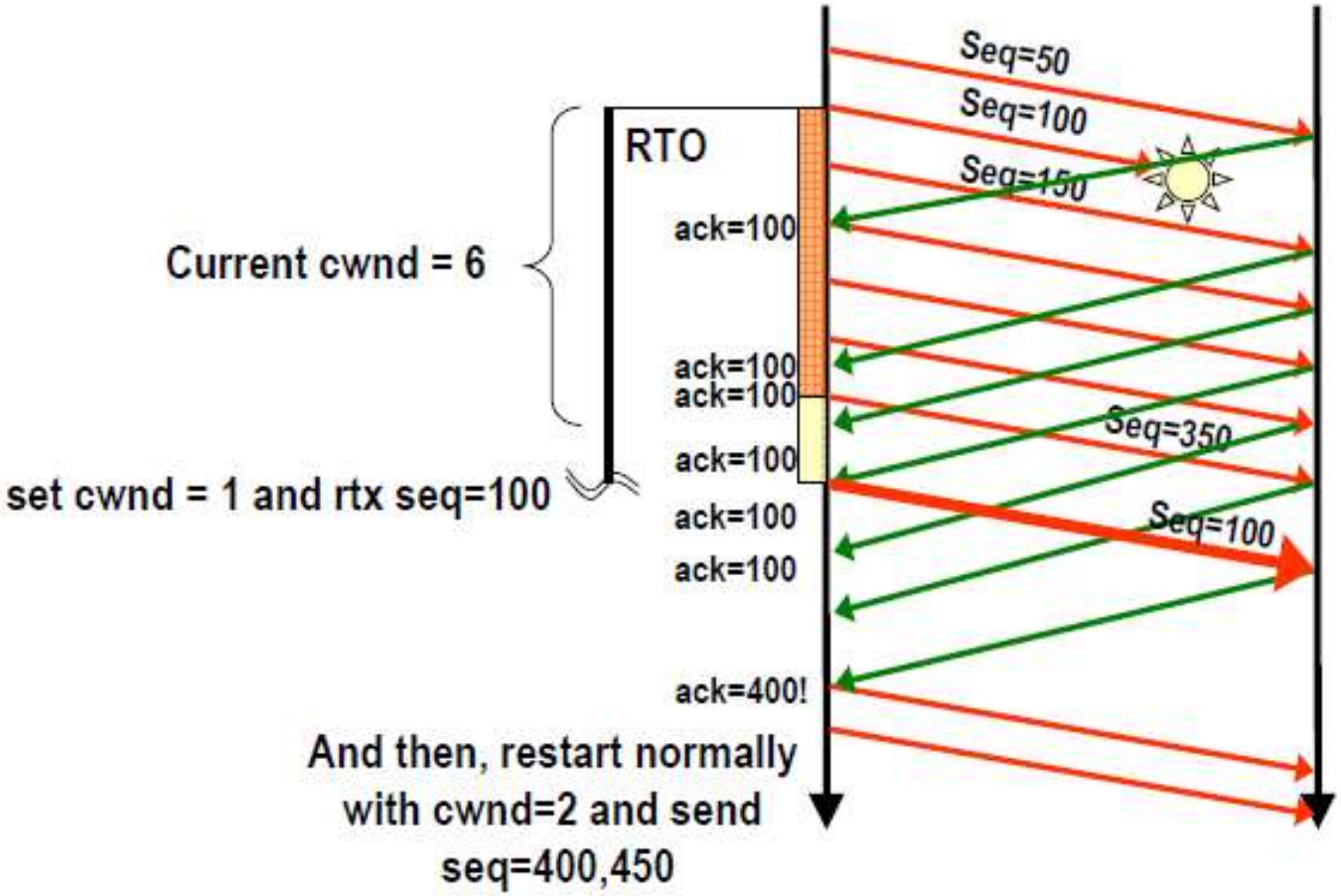

Simplified example (overall)
Timeout:
cwnd = 1
ssthresh=8
Timeout:
cwnd = 1
ssthresh=6
Congestion window cwnd (in MSS)
16
14
12
10
8
6
4
3
2
1
1
Number of transmissions

What happens AFTER RTO?
(without fast retransmit)



And then, restart normally with cwnd=2 and send seq=400,450

TCP TAHOE
(with fast retransmit)



Same as before, but shorter time to recover packet loss!

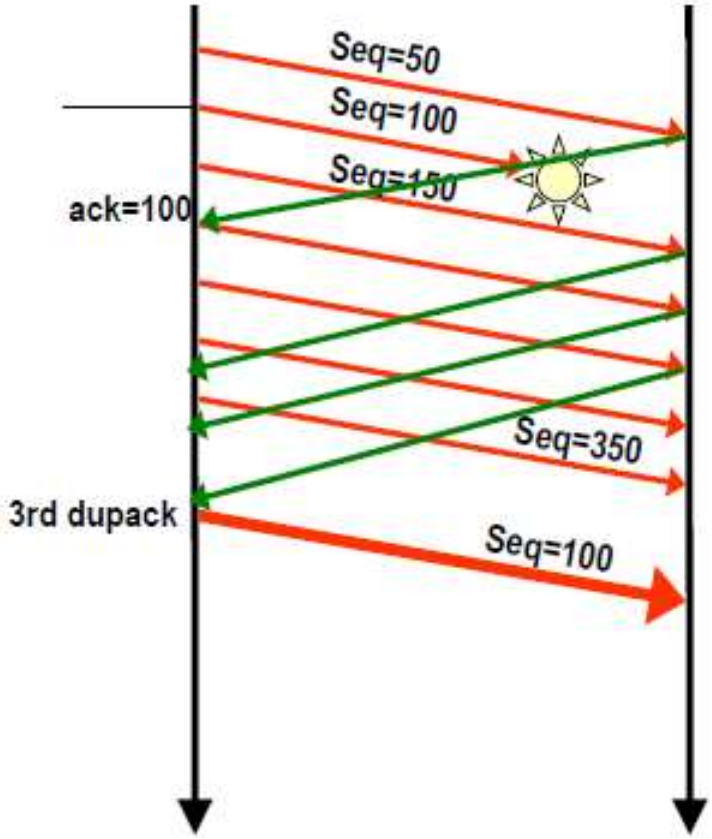
Motivations for fast recovery

FAST RECOVERY:

- ⇨ The phase following fast retransmit (3 duplicate acks received)

- ⇨ TAHOE approach: slow start, to protect network after congestion

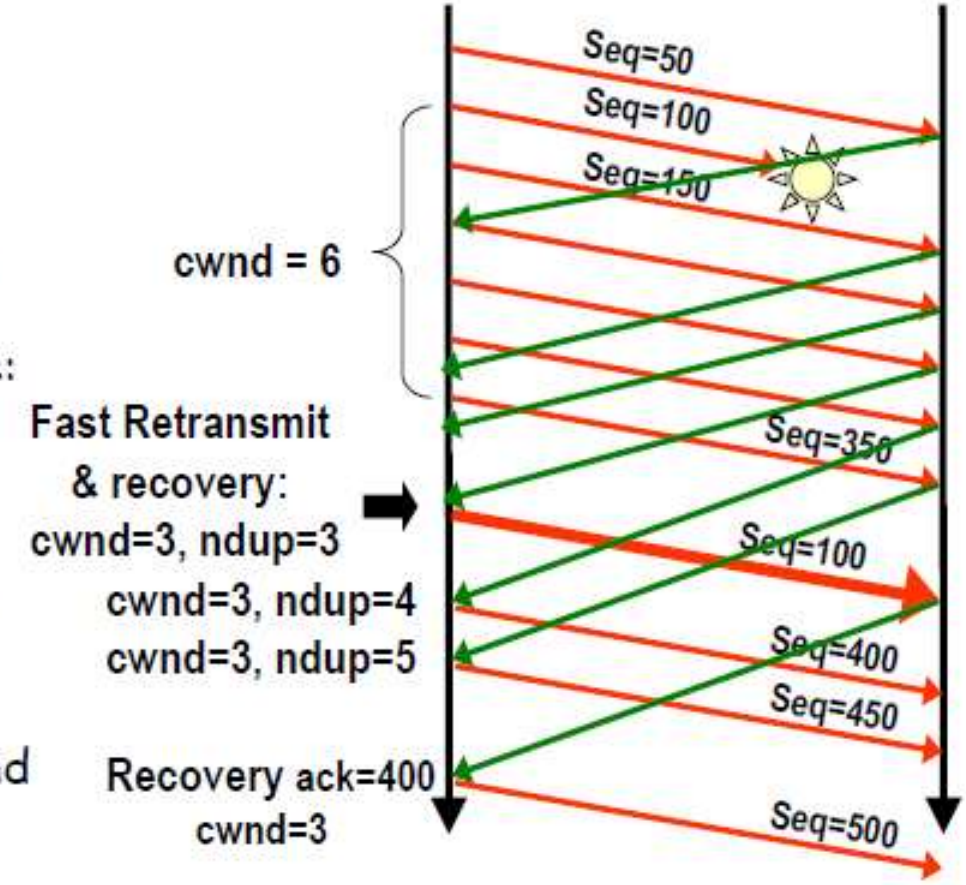
- ⇨ However, since subsequent acks have been received, no hard congestion situation should be present in the network: slow start is a too conservative restart!



Fast recovery rules

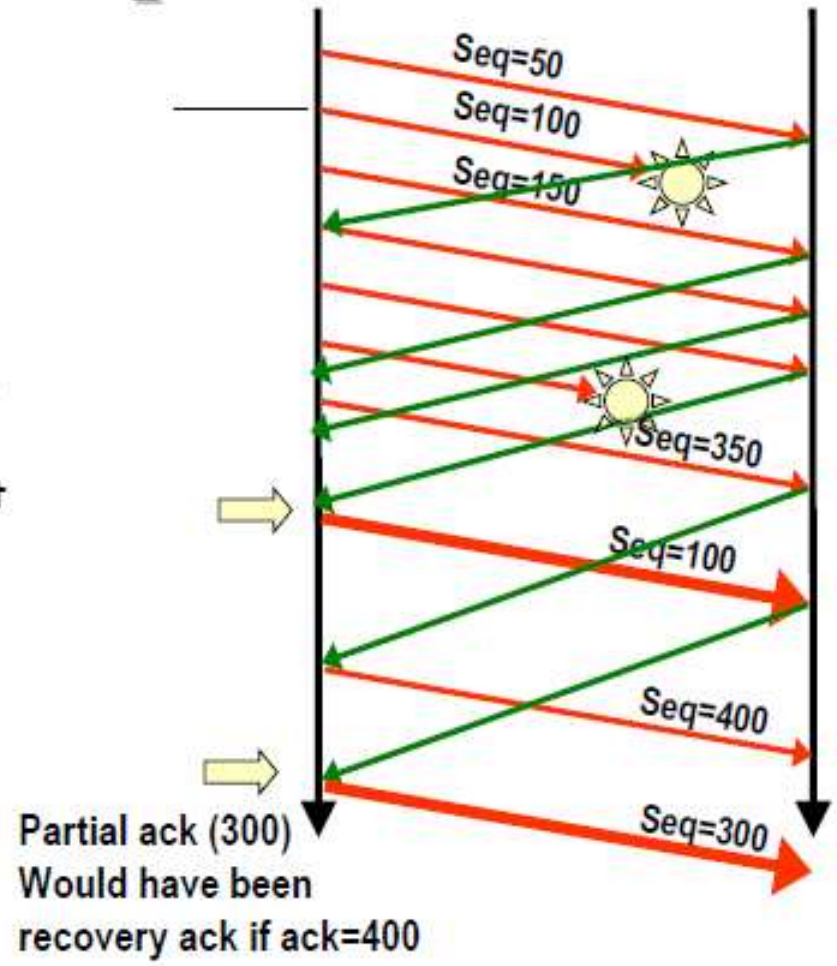
FAST RECOVERY RULES:

- ⇨ Retransmit lost segment
- ⇨ **Set cwnd = ssthresh = cwnd/2**
- ⇨ **Restart with congestion avoidance (linear)**
- ⇨ start fast recovery phase:
  - ⇨ Set counter for duplicate packets ndup=3
  - ⇨ Use "inflated" window: w = cwnd+ndup
  - ⇨ Upon new dup_acks, increase ndup, not cwnd (and send new data)
  - ⇨ Upon recovery ack, "deflate" window setting ndup=0



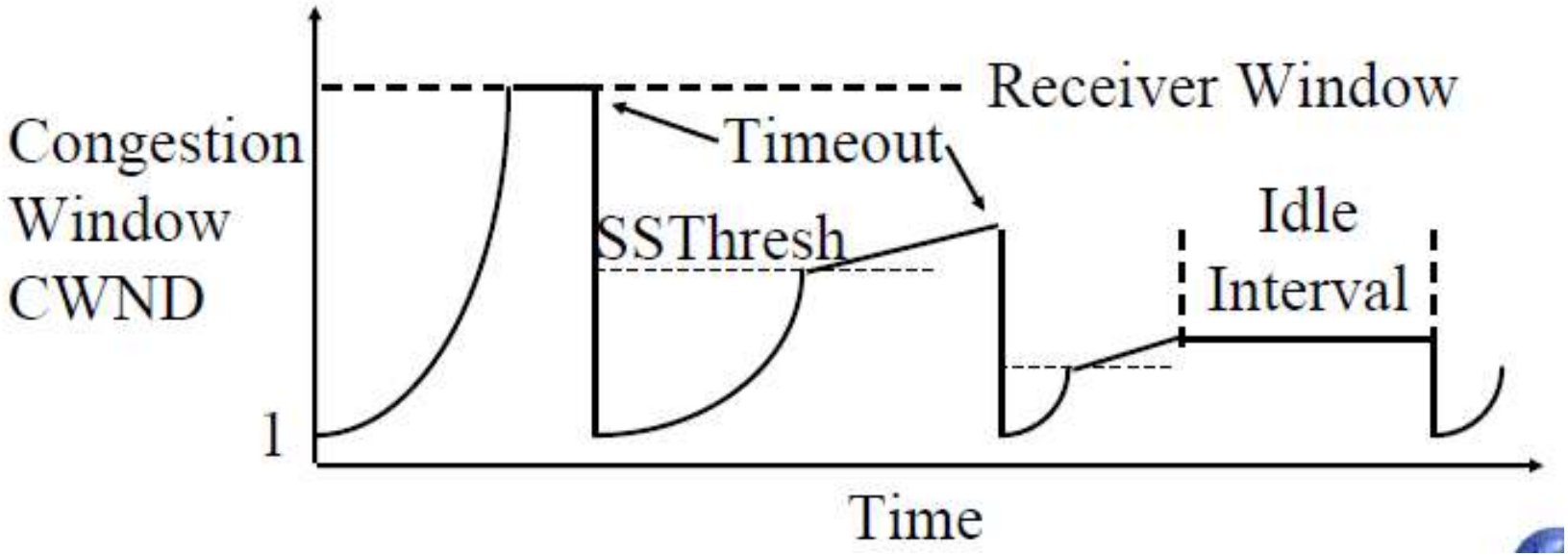
What about multiple losses?

- TCP Reno optimized for single loss
- Performance drawbacks with multiple losses in same window
- Improvement: NewReno
  - Distinguish recovery ack from partial ack
    - →Equal to the recovery ack, but does not recover for all ndup segments
  - Does not exit fast recovery when partial ack received
  - Retransmit segment immediately following partial ack, assuming it was lost



Idle periods

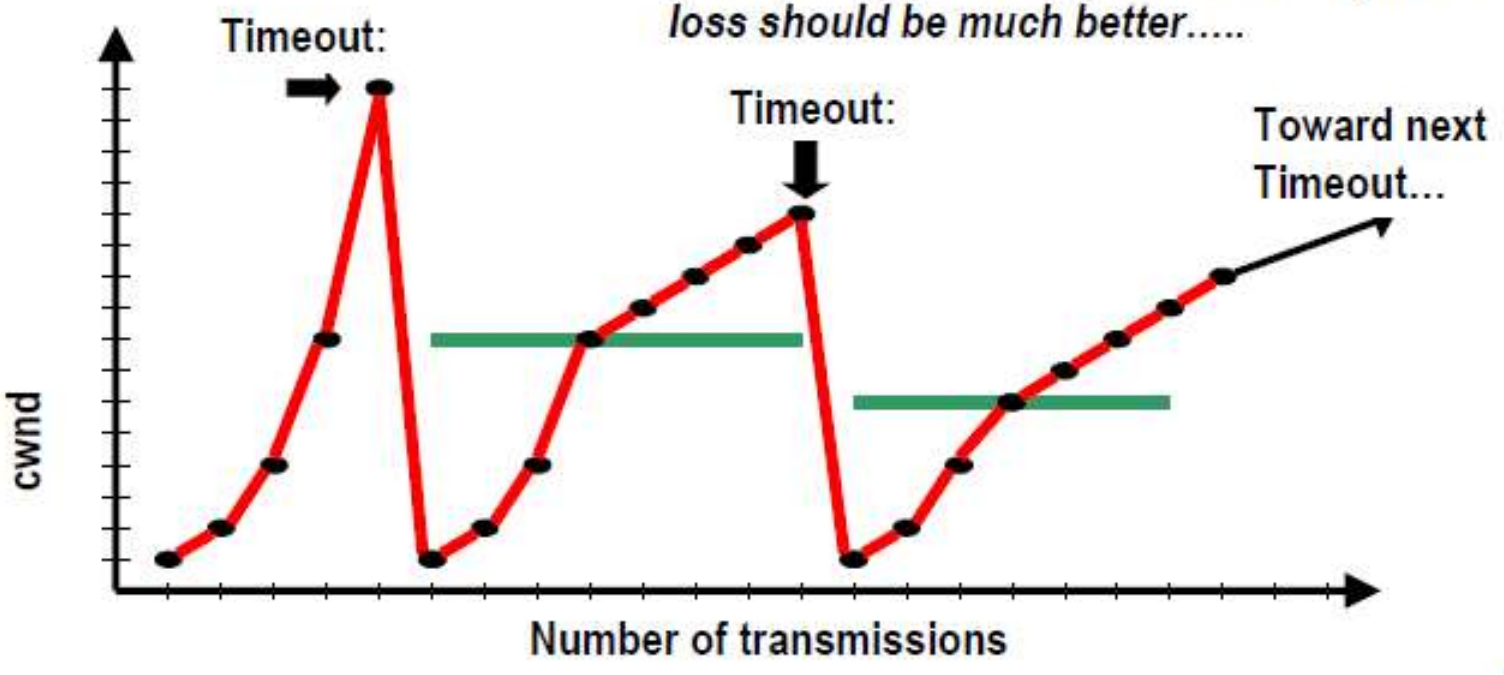
➔ **After a long idle period (exceeding one RTO), reset the congestion window to one.**



Further TCP issues

Timeout = packet loss occurrence in an internal network router
TCP (both Tahoe & Reno) does not AVOID packet loss
Simply REACTS to packet loss

CONCLUSION: a TCP able to AVOID packet loss should be much better.....



TCP Vegas (1995)

➔ **Avoids packet loss by predicting it!**

- ⇨ Approach: monitor RTT
- ⇨ when RTT shows increase, deduce that congestion is going to occur
- ⇨ and thus preventively reduce cwnd
- ⇨ but not down to as low as slow start

➔ **A problem: DOES NOT WORK WHEN OTHER TERMINALS USE TAHOE/RENO!!!!**

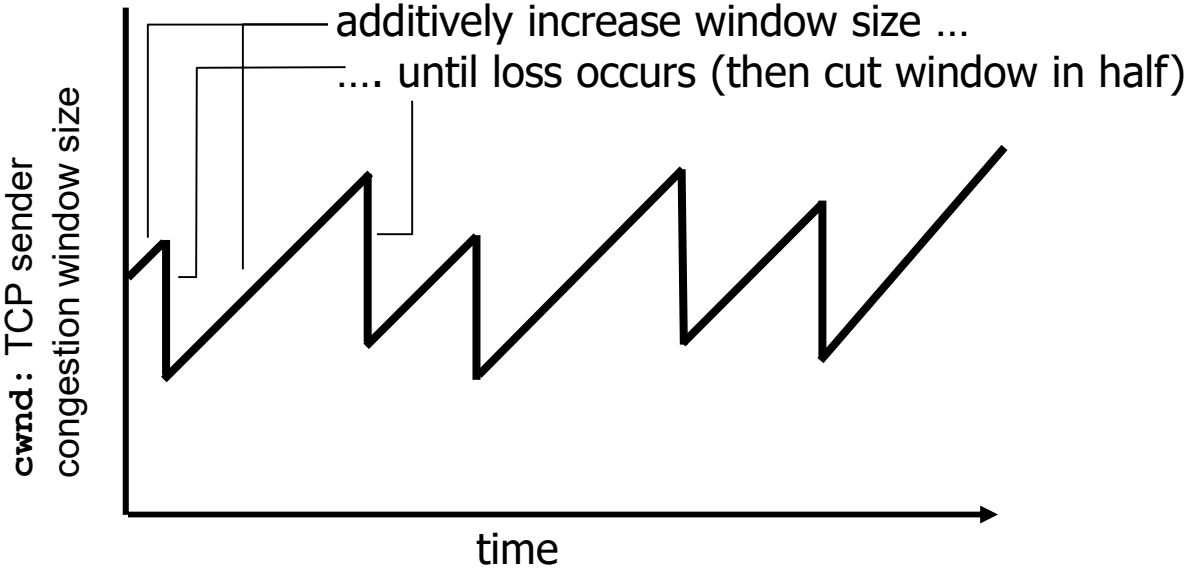
- ⇨ Vegas reduces rate to avoid congestion
- ⇨ while Tahoe/Reno grab the available bandwidth!!

A typical problem in Internet Protocol design: need to live with legacy apps and protoc

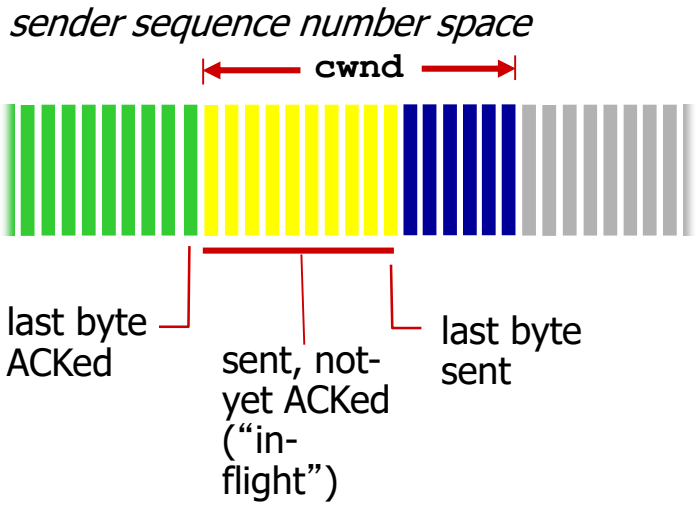
TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: details



- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}$$

- **cwnd** is dynamic, function of perceived network congestion

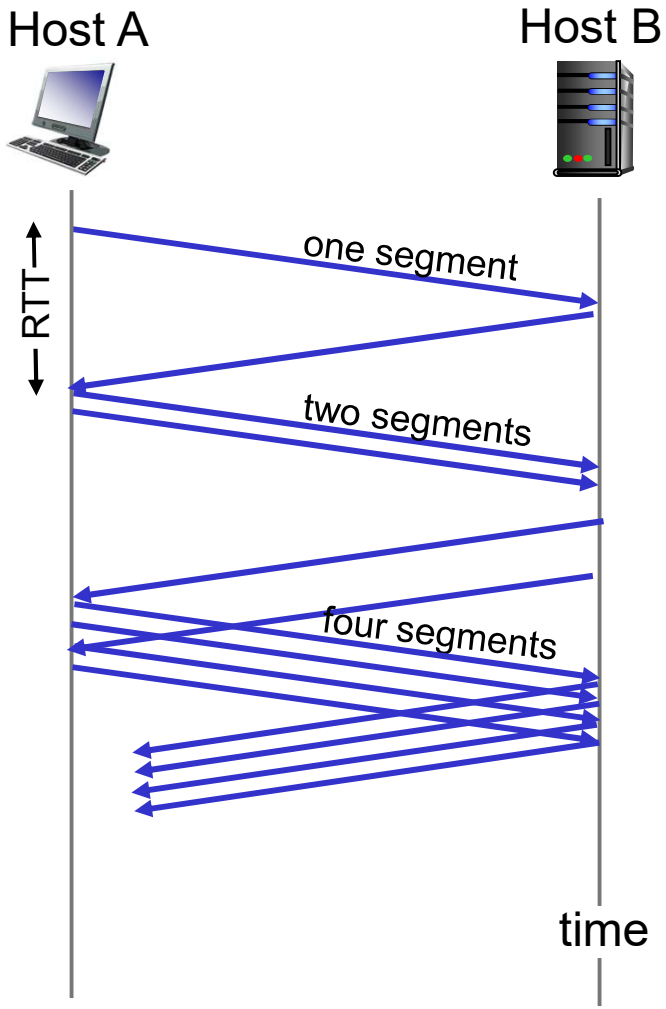
TCP sending rate:

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



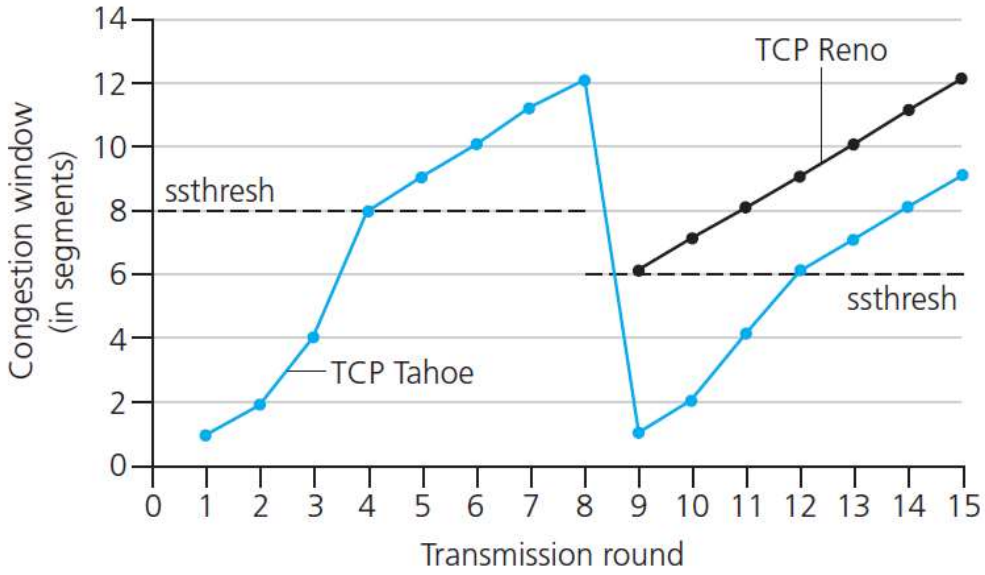
TCP: detecting, reacting to loss

- loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **`cwnd`** is cut in half window then grows linearly
- TCP Tahoe always sets **`cwnd`** to 1 (timeout or 3 duplicate acks)

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **`cwnd`** gets to 1/2 of its value before timeout.



Implementation:

- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

Case study: ATM ABR congestion control

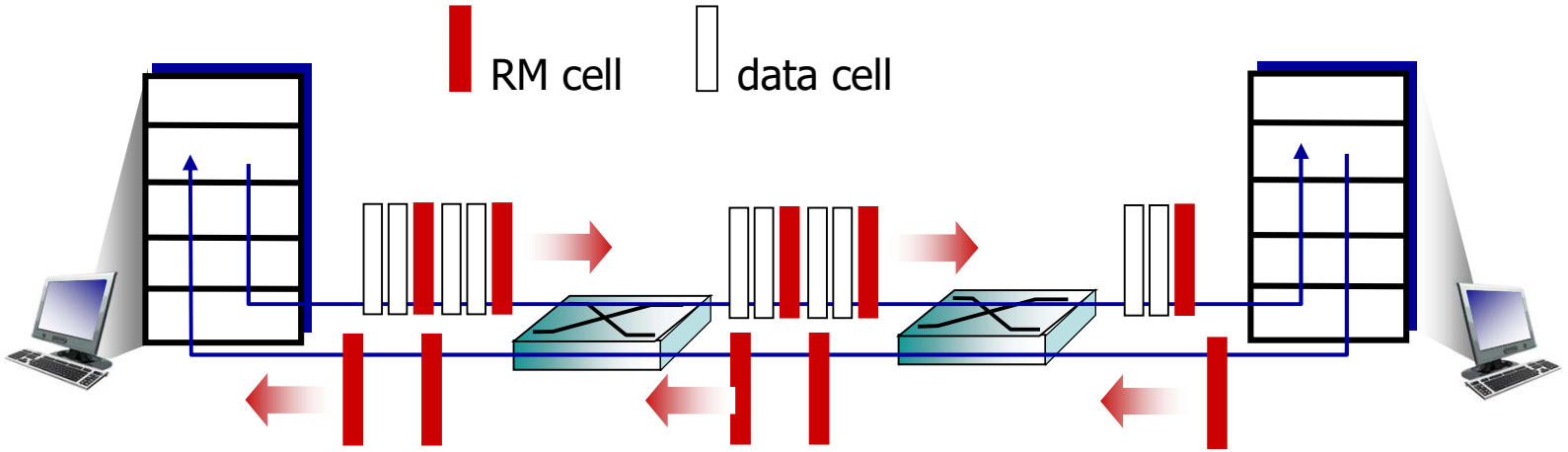
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

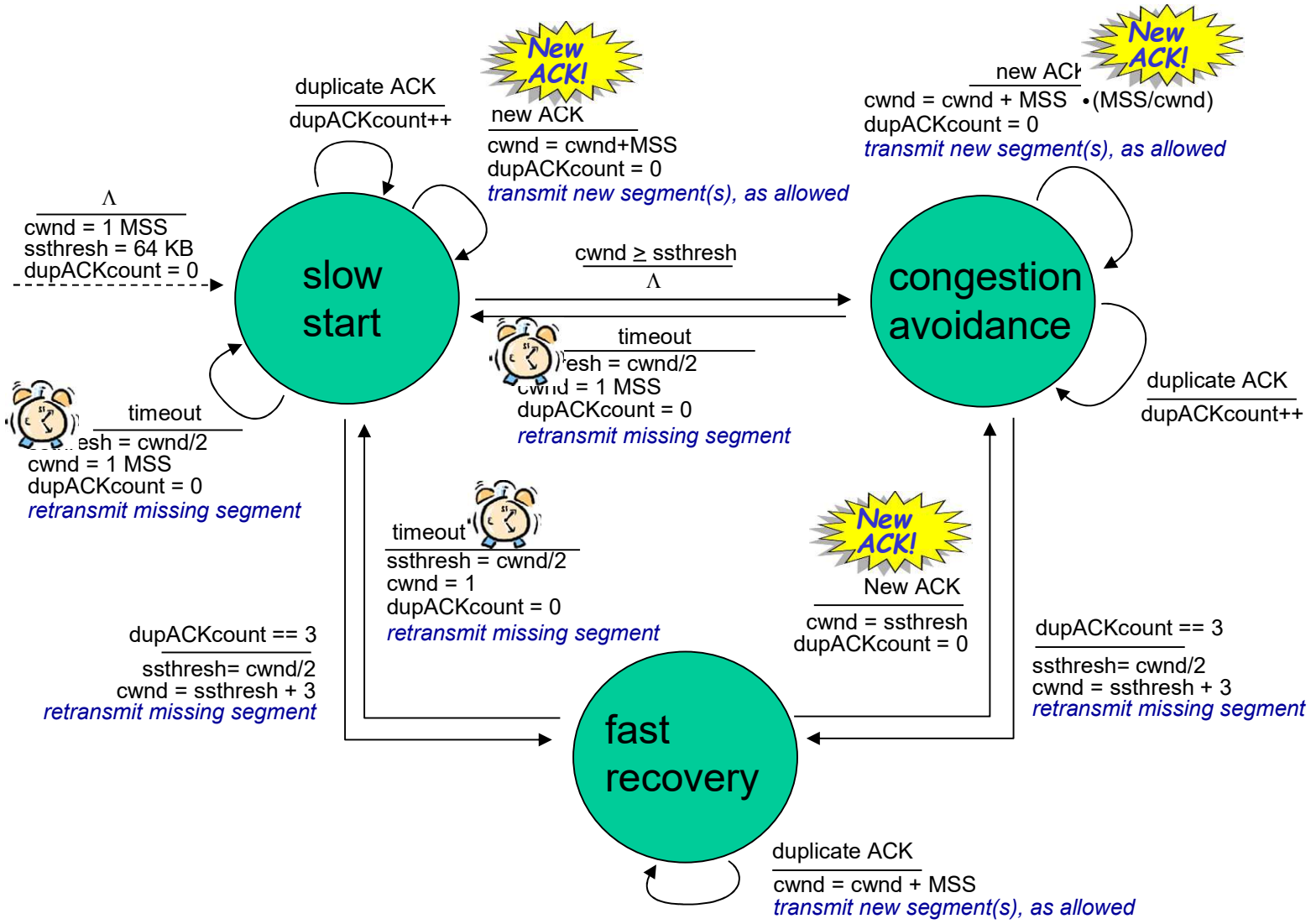
- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - *NI bit:* no increase in rate (mild congestion)
  - *CI bit:* congestion indication
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

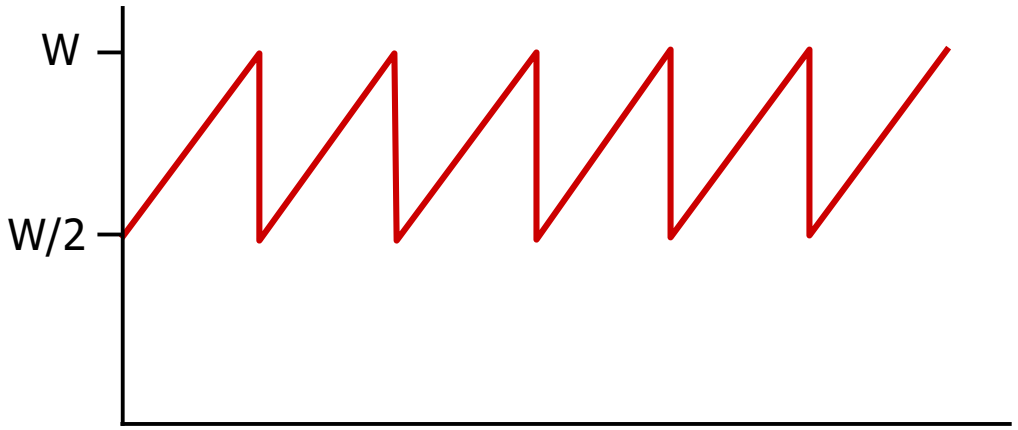
Summary: TCP Congestion Control



TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP Futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

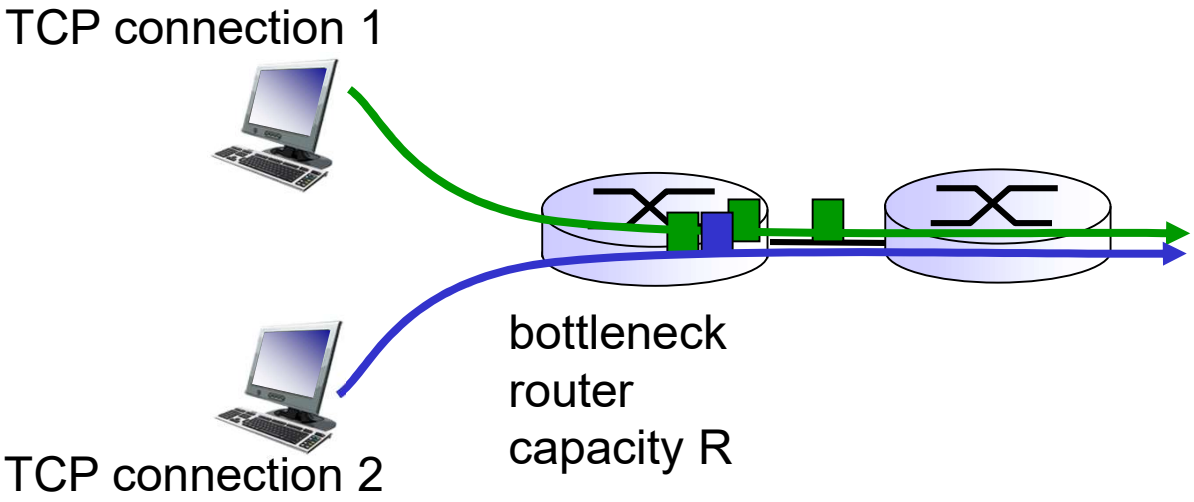
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of L = $2 \cdot 10^{-10}$ *– a very small loss rate!*

- new versions of TCP for high-speed

TCP Fairness

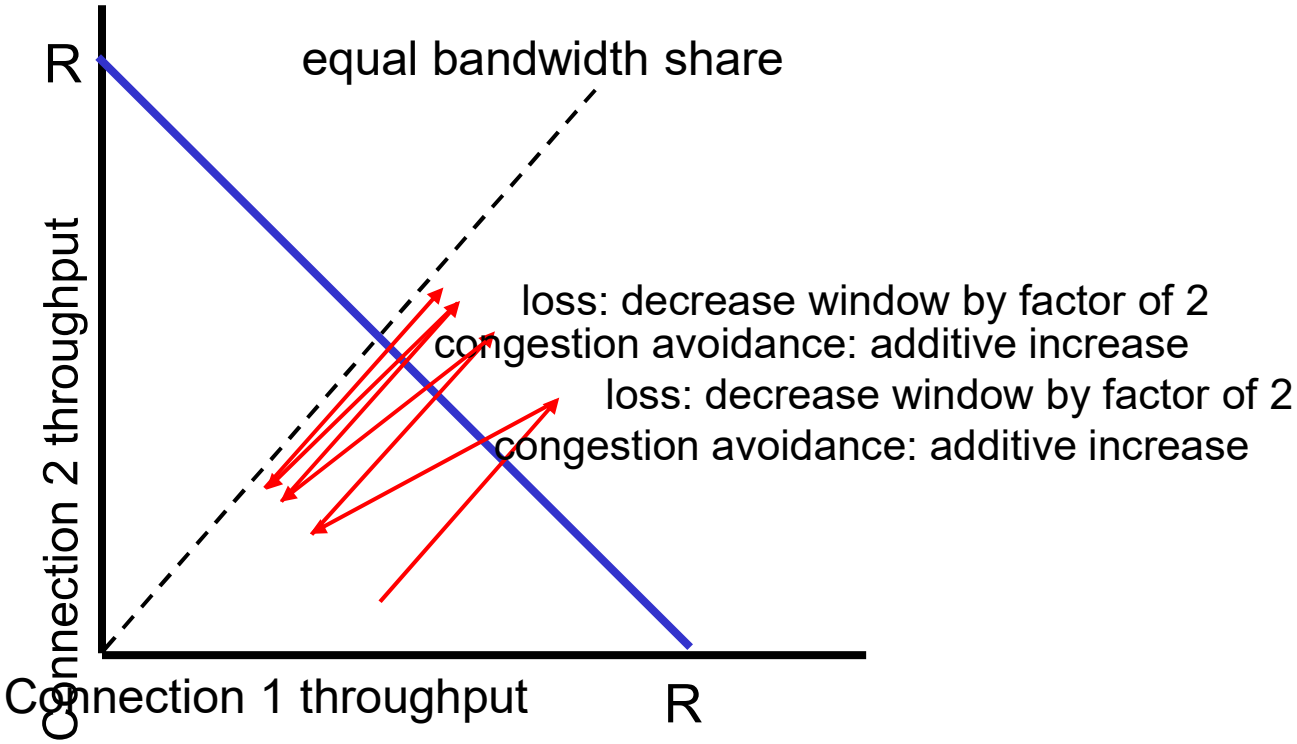
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2