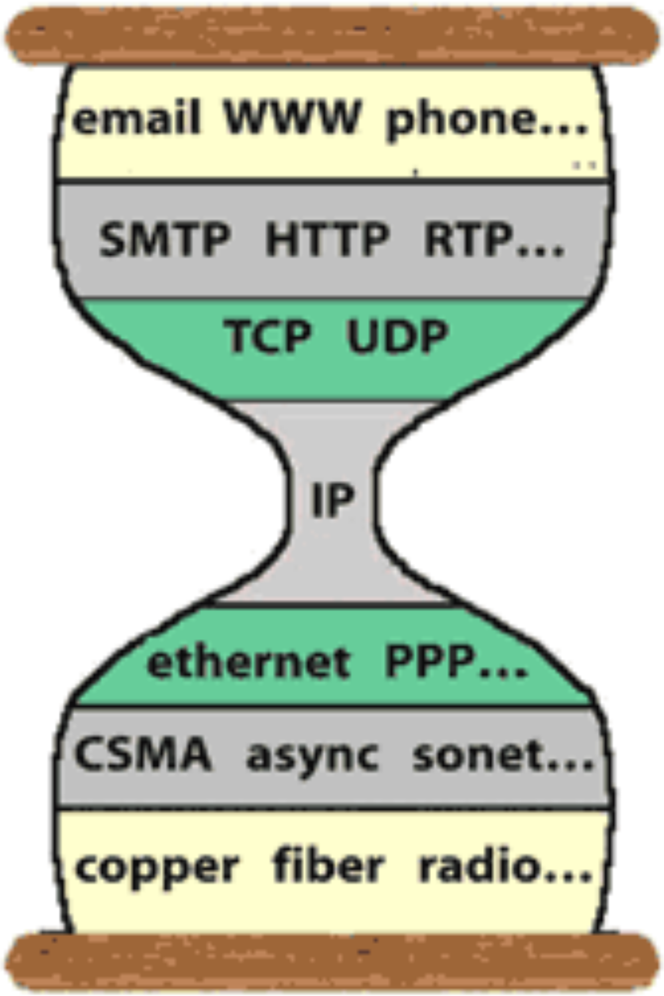
Lecture 7: Congestion Control



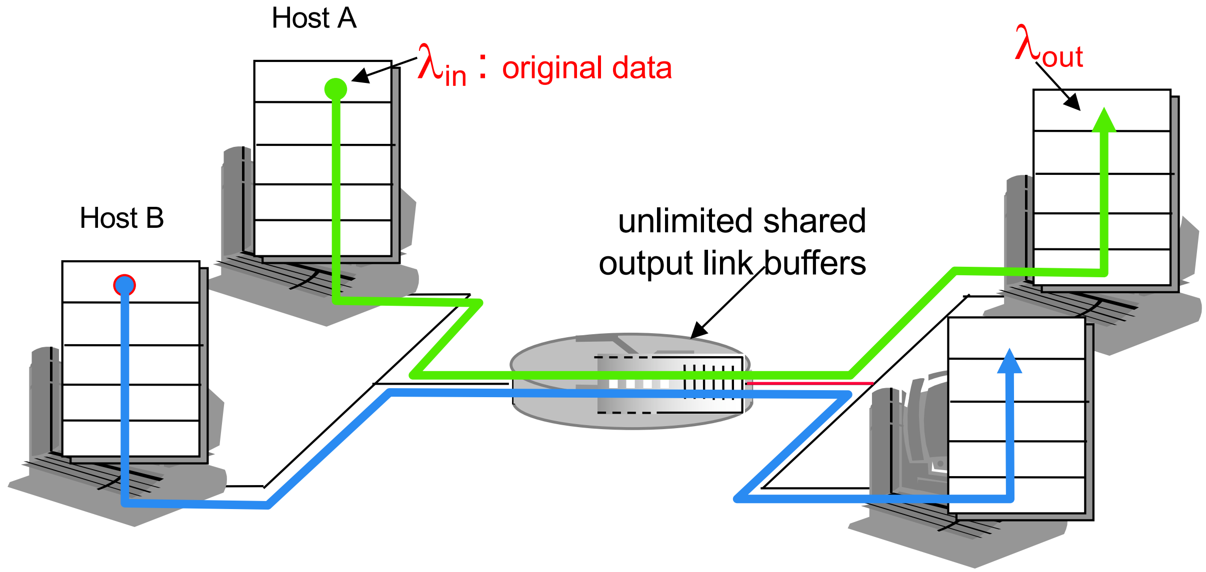
Chapter 3

3.6 Principles of congestion control

3.7 TCP congestion control

How network congestion happens

too many sources sending data too fast into the *network* at the same time

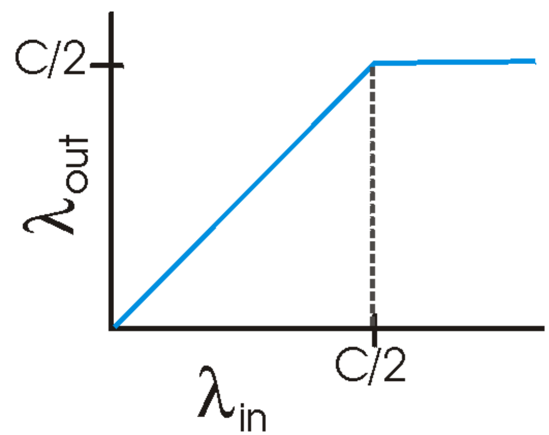


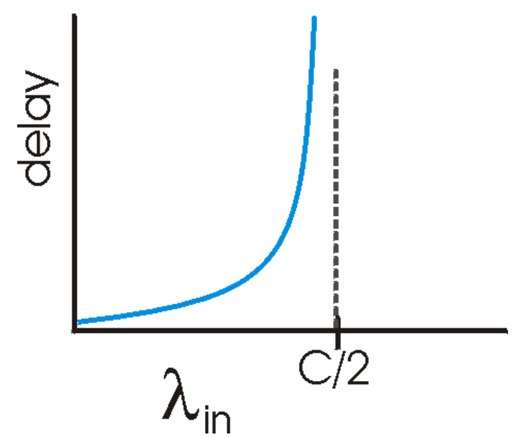
Scenario 1

- 2 senders, 2 receivers
- one router with *infinite* buffer
- no retransmission

When congested:

- Achieve maximum possible throughput
- long delays, unbounded

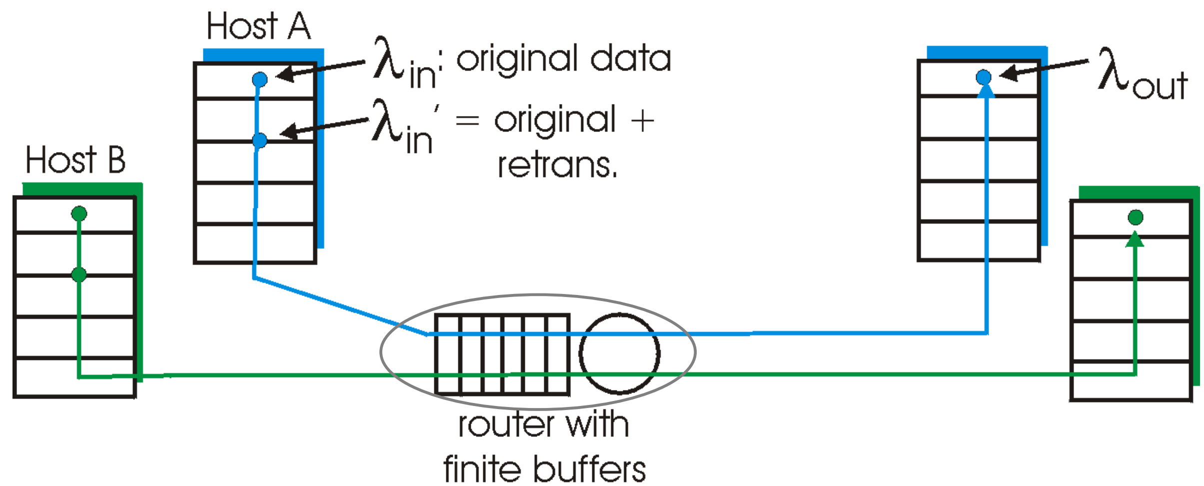




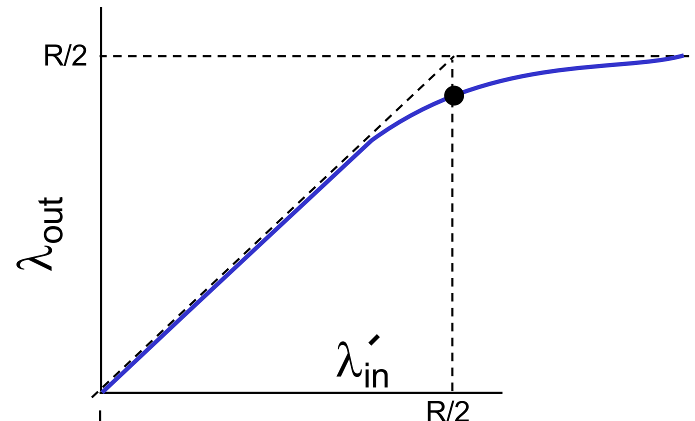
Congestion: scenario 2

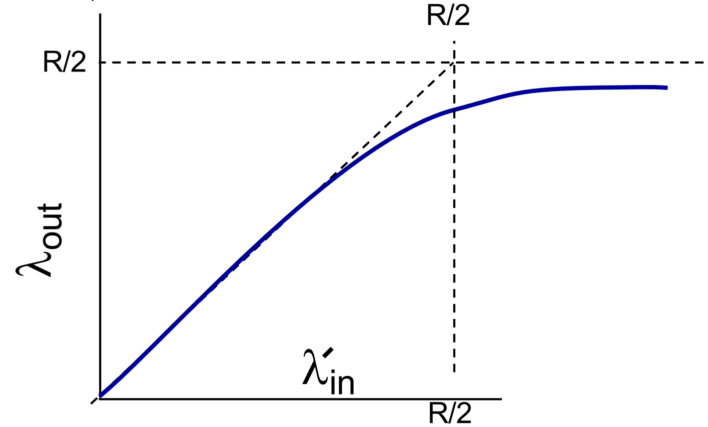
one router, ***finite*** **buffer**

senders *retransmit* when timeout

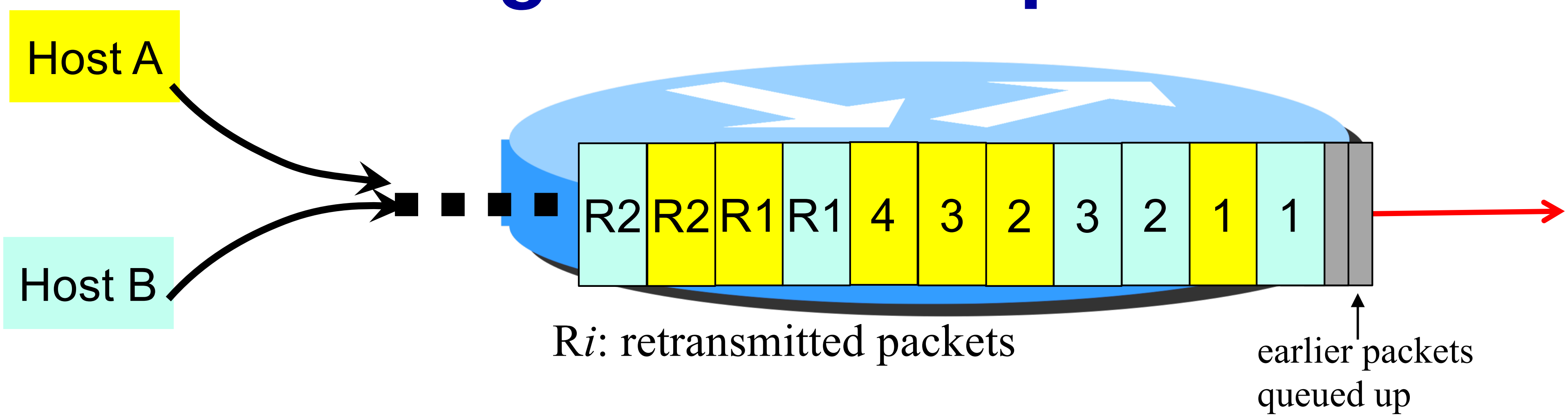


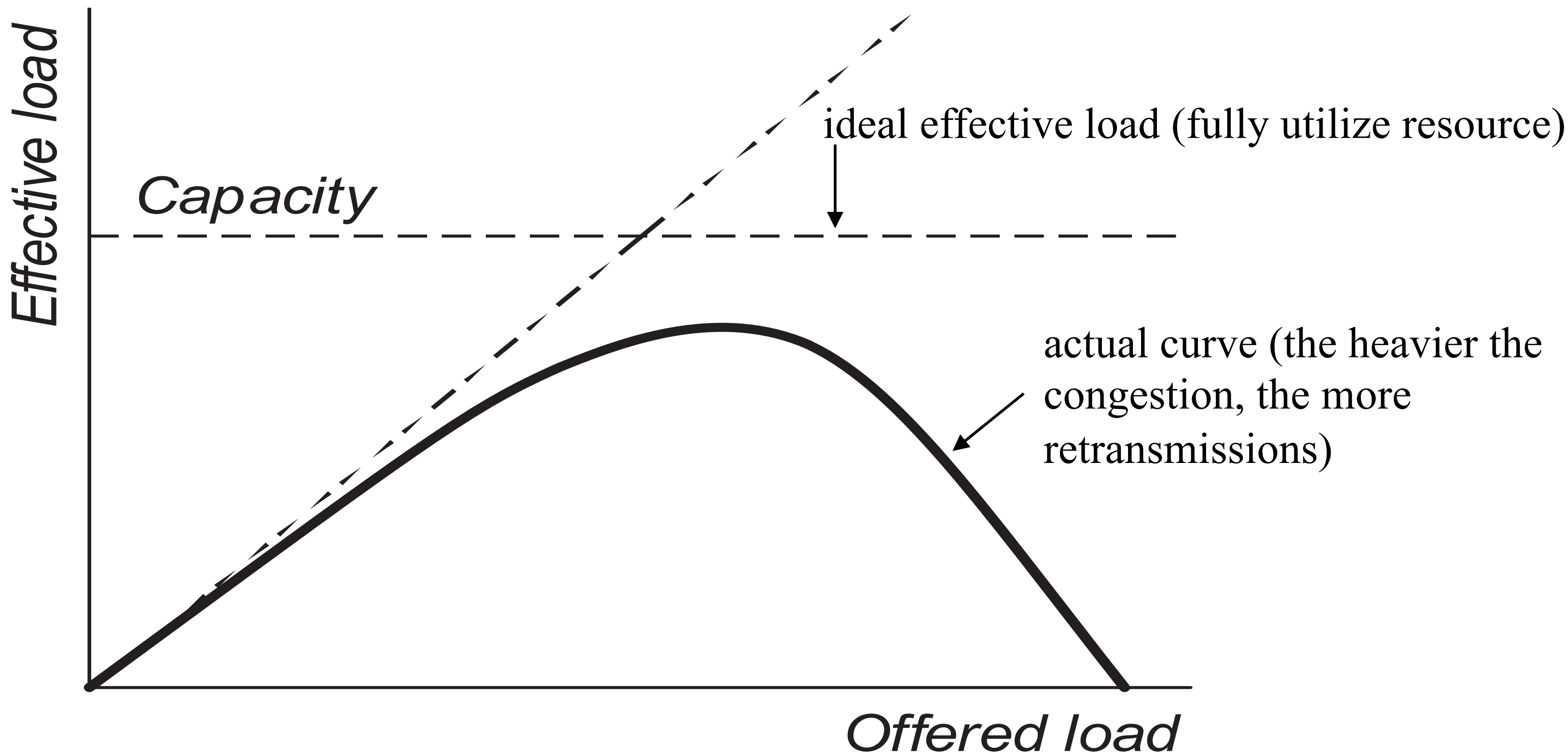
- Packets may get dropped at router due to buffer full
- **Known loss case**: sender only retransmits if a packet is known to be lost
- **Duplicates**: sender may time out prematurely and retransmit, *some duplicates* are delivered





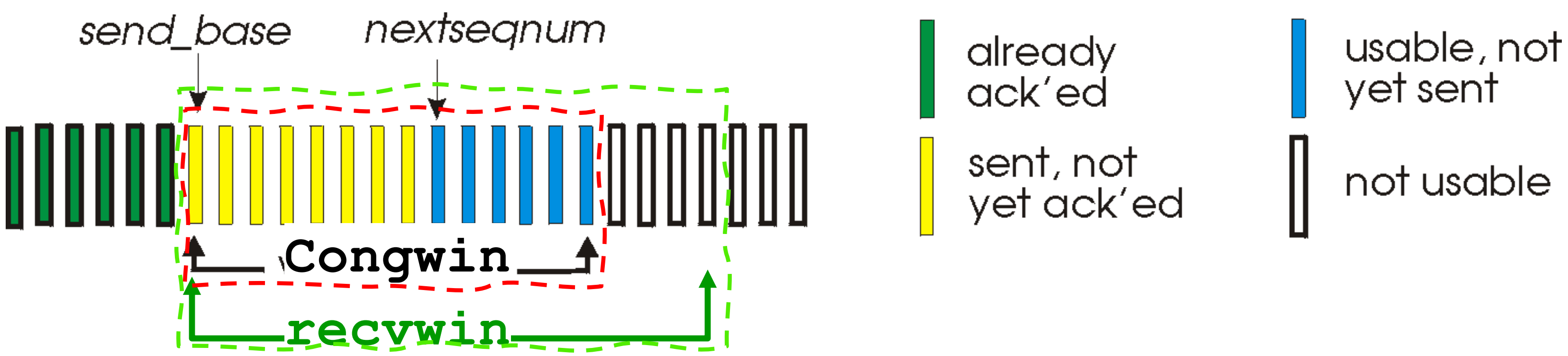
Congestion Collapse





TCP Congestion Control

- Add a ***congestion control window*** (**`cwnd`**) on top of the flow-control window
  - Sender limits: **`LastByteSent-LastByteAcked`** ≤ **`cwnd`**



- How to adjust **`cwnd`** size based on network traffic load?
  - Infer network congestion by observed packet losses

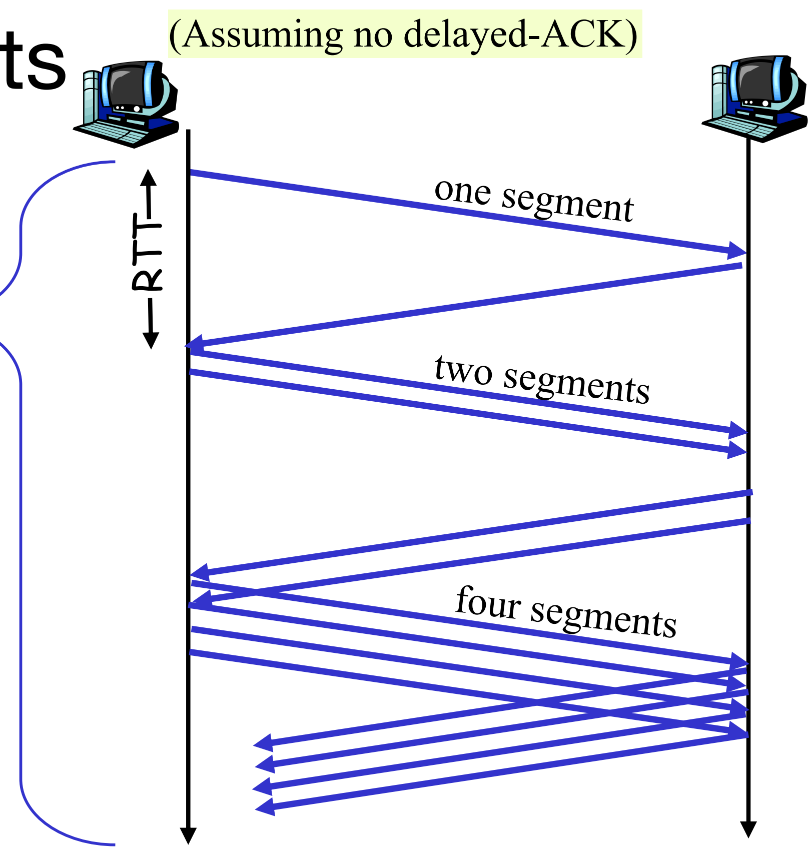
Congestion Control (CC) Window Adjustment

- Two phases:
  - slow start: set CC window (cwnd) size to 1 *segment*
    - Start slow but *rapidly* increase CC window size
  - congestion avoidance
    - Slowly but continuously increase CC window size
- Use Slow-Start Threshold (ssthresh) to define the boundary between these two phases
  - When cwnd < ssthresh: in slow-start phase, increase cwnd quickly
  - When cwnd ≧ ssthresh: in congestion avoidance phase, increase cwnd by one segment per RTT

TCP Slow Start

Objective: gauge the pipeline size quickly

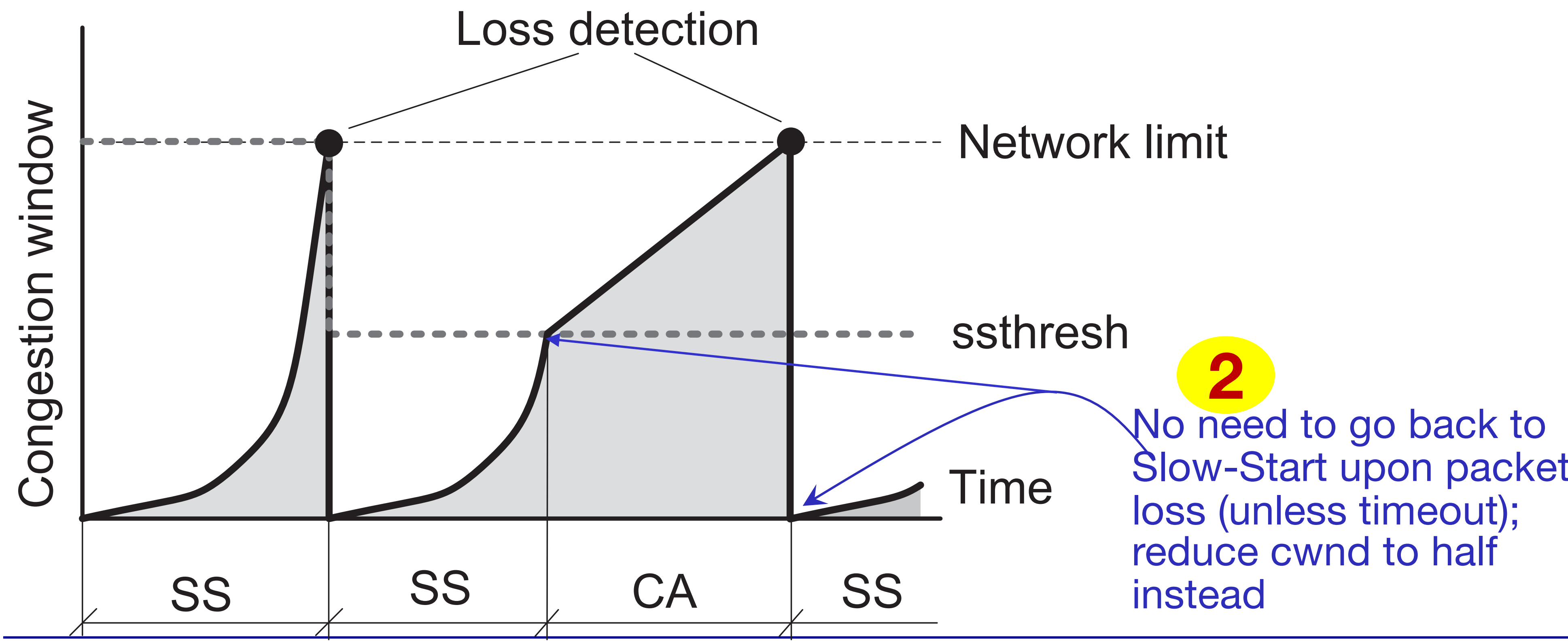
1. Set cwnd = 1 MSS (max. segment size, in bytes)
   - i.e. cwnd = 1 segment worth of bytes
2. Send cwnd-allowed segments
3. If receive an ack ← which moves the cumulative ACK forward
   - cwnd = cwnd + 1 segment
     - more segment can be sent now
4. If timeout [cwnd have gone too far]
   - ssthresh = cwnd / 2
   - cwnd = 1 MSS [reset cwnd to 1 segment]
   - goto step 2



Multiplicative Increase per RTT

Slow Start <u>with Congestion Avoidance</u>

- Set cwnd = 1 packet, and initialize **`ssthresh`**
  - default: initialize ssthresh to the flow control window size
- When **`cwnd < ssthresh`**: in Slow Start phase
- when **`cwnd ≥ ssthresh`**: in Congestion Avoidance phase
  - increase cwnd by one packet per round-trip time 1

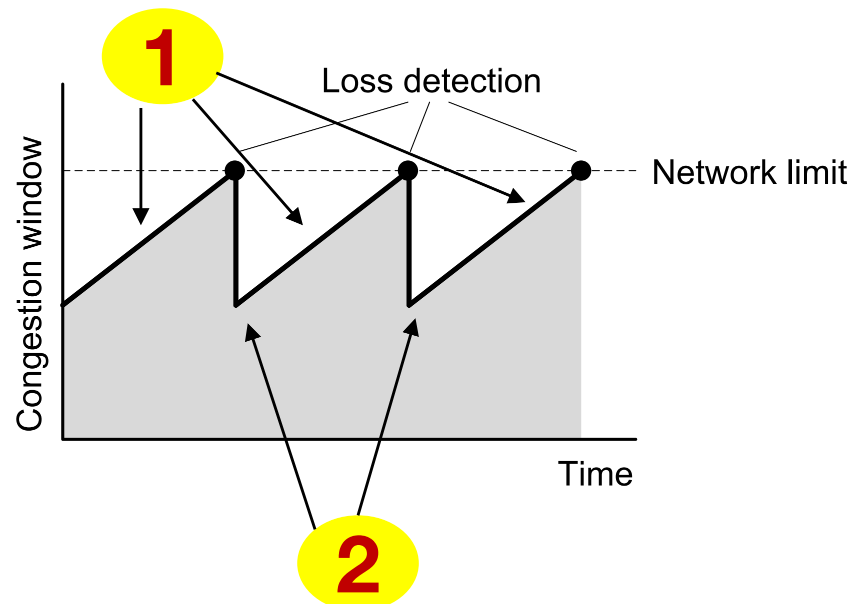


Congestion Avoidance:
Additive Increase, Multiplicative Decrease (AIMD)

Without moving cwnd back to single segment

Objective: cautiously probe for unused resources, quickly recover from overshoot

- Send cwnd-allowed segments
  - If all sent segments *in the last RTT* time period get ACKed
    - cwnd = cwnd + 1 segment
  - Else if 3 dup-ACKs
    - cwnd = cwnd / 2
  - Else if timeout
    - cwnd = 1 segment



From the TCP lecture:

TCP Fast Retransmit

- RTO set to a relatively long value
  - Detect loss by timeout → long delay before retransmit
- Detect packet loss by duplicate ACKs
  - When a segment is lost, next arrival at receiver is out of order
  - Receiver sends an ack with the seq# of the last in-order arrival (cumulative ACK)
- When sender receives 3 duplicate ACKs carrying #n: assumes the segment of seq#(n) is lost
  - Why 3 dup-ACKs: avoid false alarm due to out-of-order packet delivery

→fast retransmit: resend the segment without waiting for timeout

- Resending one segment only; also restart the retransmission timer

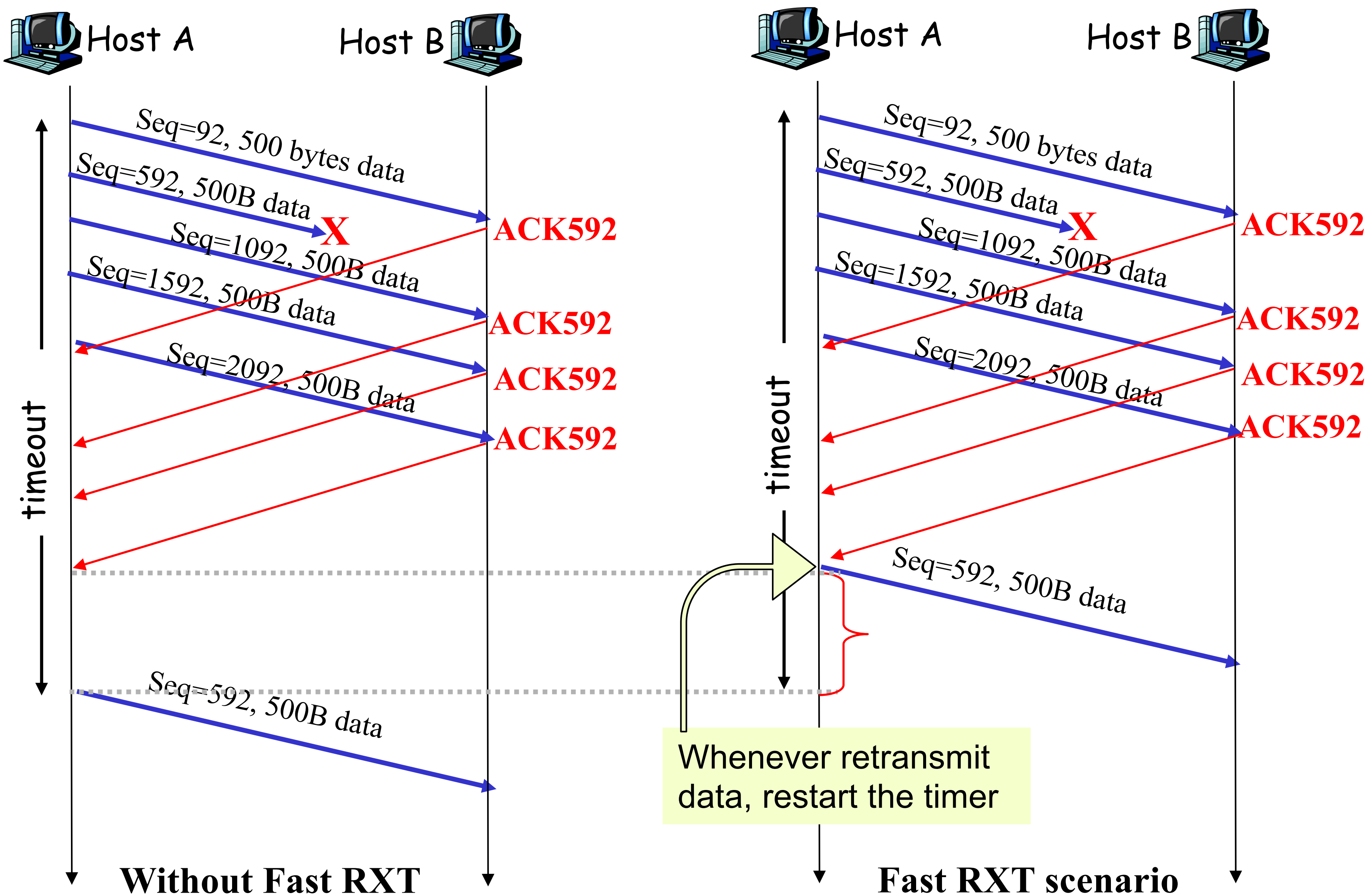
Congestion Avoidance

Objective: in steady state, the sender gently probe for unused resources

- Send cwnd packets
- If receives an ack **1**
  - cwnd(i) = cwnd(i-1) + (#bytes in 1 segment)/cwnd(i-1)
- If detect loss by 3 duplicate ACKs: *packets continue to arrived at receiver* → network not badly jammed
  - cwnd = ssthresh = cwnd / 2 **2**

Additive Increase, Multiplicative Decrease (AIMD)

TCP fast retransmit example



Early Congestion Notification (ECN)

FYI

- ECN-capable hosts set ECT (0 or 1) bits in IP header `(ECT: ECN Capable Transport)`
- When a router is getting overloaded: set the 2 ECN bits to 11
- TCP receiver: set an “ECN-Echo” (ECE) flag in the ACK packet going to the sender
- TCP sender: cut cwnd to half
  - congestion avoidance)

```
+-----+-----+
| ECN FIELD |
+-----+-----+
   0     1         ECT(1)
   1     0         ECT(0)
   1     1         CE
```

In IP header

sender can use either 01 or 10;
routers sets to 11 to indicate congestion.
These 2 bits are copied on return ACK pkt

TCP Throughput

important

- What's TCP throughout as a function of window size and RTT?
- Ignore slow start: let W = window-size when loss occurs
  - When window is W: throughput = W / RTT
  - Just after loss

    window → W/2, throughput → W/2RTT
  - (rough estimate) Average throughout: 0.75 W/RTT

Summary

- Congestion control is a necessary tool to avoid congestion collapse
  - congestion collapse: increasing load →further decreasing goodput
- Classic TCP congestion control approaches: end host adaptation
  - Don't rely on network help, try to estimate network state using losses
    - More advanced schemes also estimate by delays, delay changes
- Classic TCP congestion controls have two main stages
  - Slow Start to quickly ramp up sending
  - Congestion Avoidance to maintain sending

Summary: TCP Congestion Control Actions

1. a TCP connection starts with slow start
   - cwnd = 1 segment
   - ssthresh assigned an initial value
2. when cwnd < ssthresh: slow-start
   - when in slow-start: increase cwnd by 1 segment for every ACK received that advances the cumulative acknowledgment value
3. when cwnd ≧ ssthresh: congestion avoidance
   - when in congestion avoidance: increase cwnd by 1 segment per RTT (or after successful delivery of a windowful of segments)
4. After loss detected: ssthresh = cwnd/2
   - if detected by 3 dup-ACKs: cwnd = cwnd/2
   - if detected by retransmission timeout: cwnd = 1 segment

Schedule Rebase

Midterm coverage (1/6 – 1/29)

Project 2 related, FYI (→ 2/3)

Mon	**1/6** Intro & BW & delay & socket **1**	**1/13** HTTP & DNS **2**	**1/20** Martin Luther King Jr. Day	**1/27** TCP **3**	**2/3** Security 101
Wed	**1/8** HTTP **2**	**1/15** DNS **2**	**1/22** Transport **3**	**1/29** Congestion Control **3**	**2/5** **Midterm**

“Modern” Transport, FYI (→ QUIC)

6 7 8 9 10

Mon	**2/10** **QUIC**	**2/17** Presidents’ Day	**2/24** Routing algorithms & protocols **5**	**3/3** Routing in the Internet **5**	**3/10** Hubs and switches **6**	**3/21: Final Exam**
Wed	**2/12** Internet Protocol (IP) **4**	**2/19** Addressing, NAT, IPv6 **4**	**2/26** Routing algorithms & protocols **5**	**3/5** Link layer (Ethernet) **6**	**3/12** Course review	

- The big yellow numbers indicate the chapter numbers in the textbook.

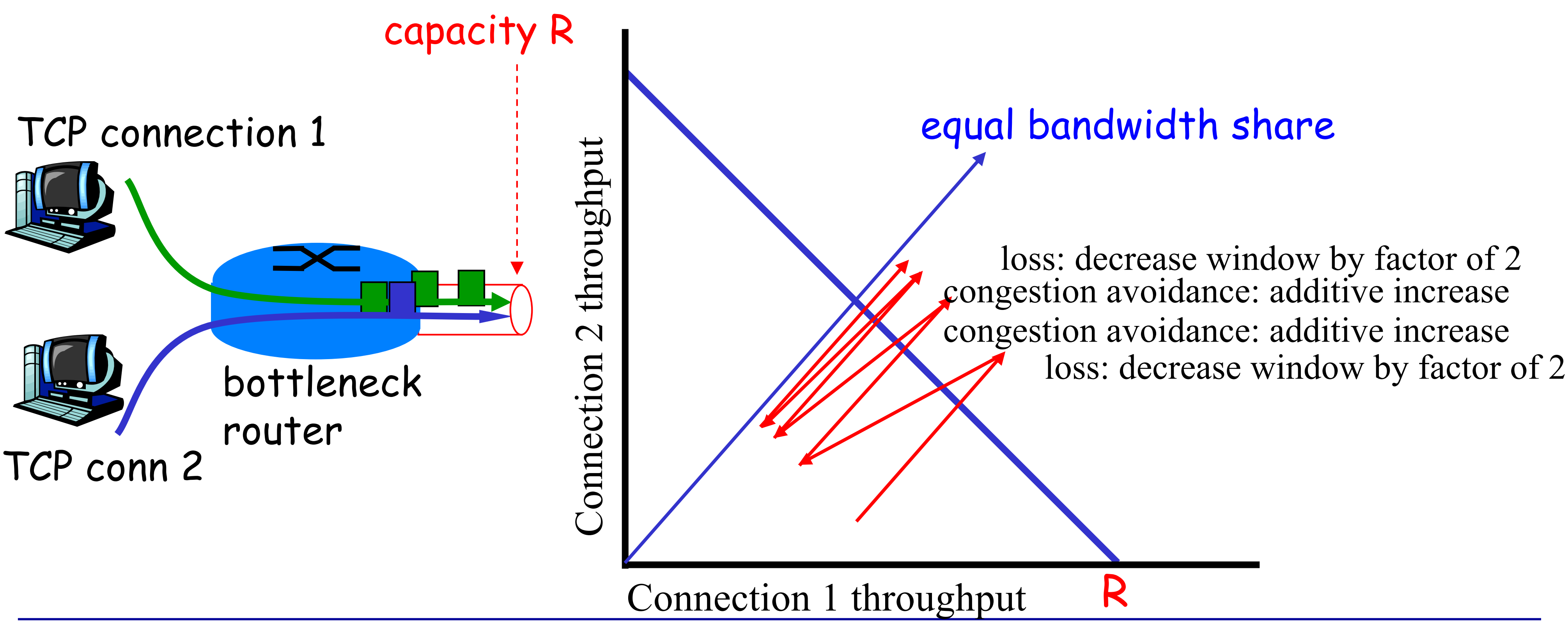
Is TCP congestion control fair?

FYI

Fairness: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity

Example: 2 competing connections, same RTT

- Additive increase gives slope of 1
- multiplicative decrease decreases throughput proportionally



Midterm next Wednesday

- in-person midterm

Summary: TCP sender congestion control

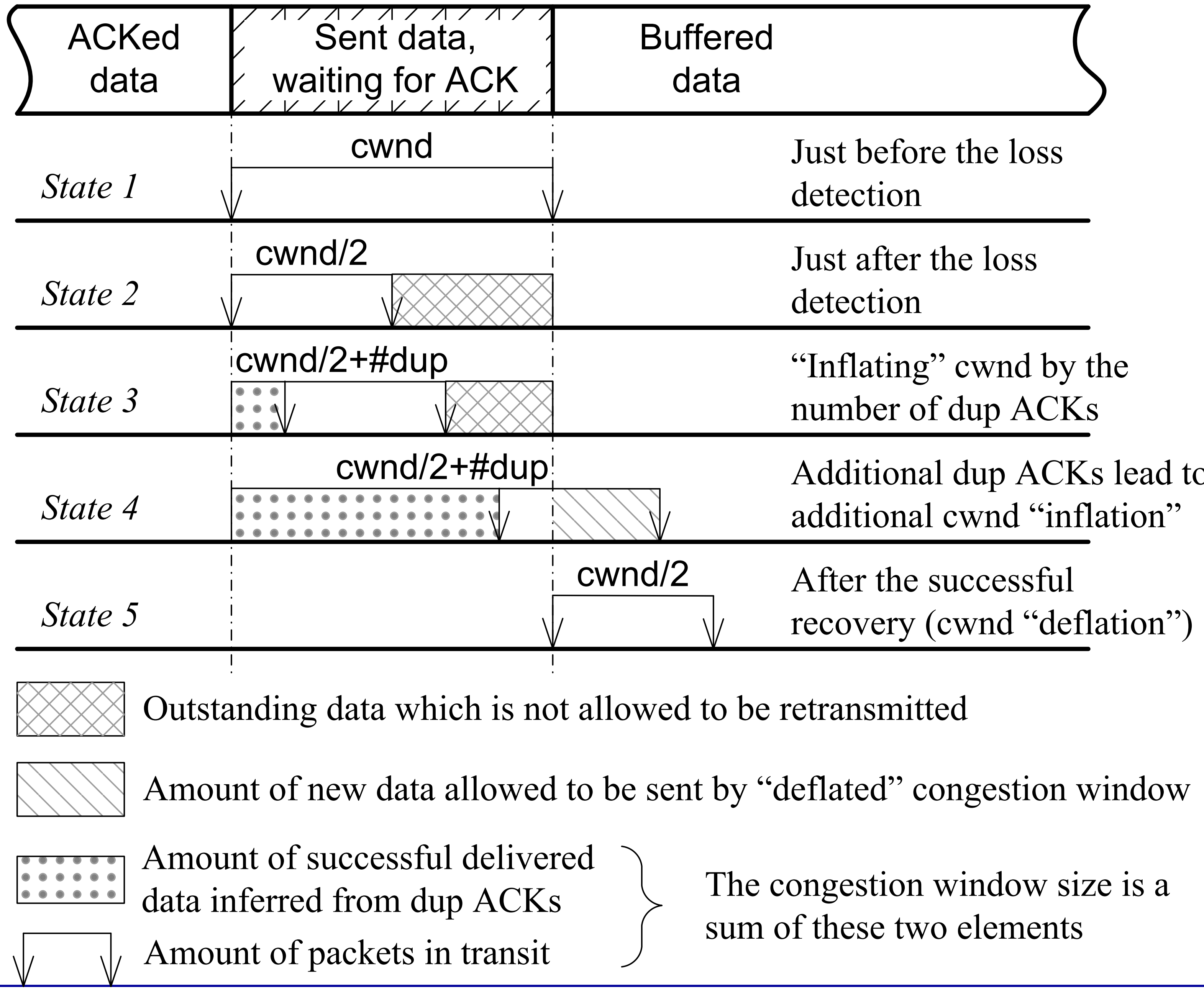
State	Event	TCP Sender Action	Commentary
Slow Start (SS)	Received ACK for previously unacked data	CongWin = CongWin + MSS If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	Received ACK for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by 3 duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

A Bit of The History of TCP

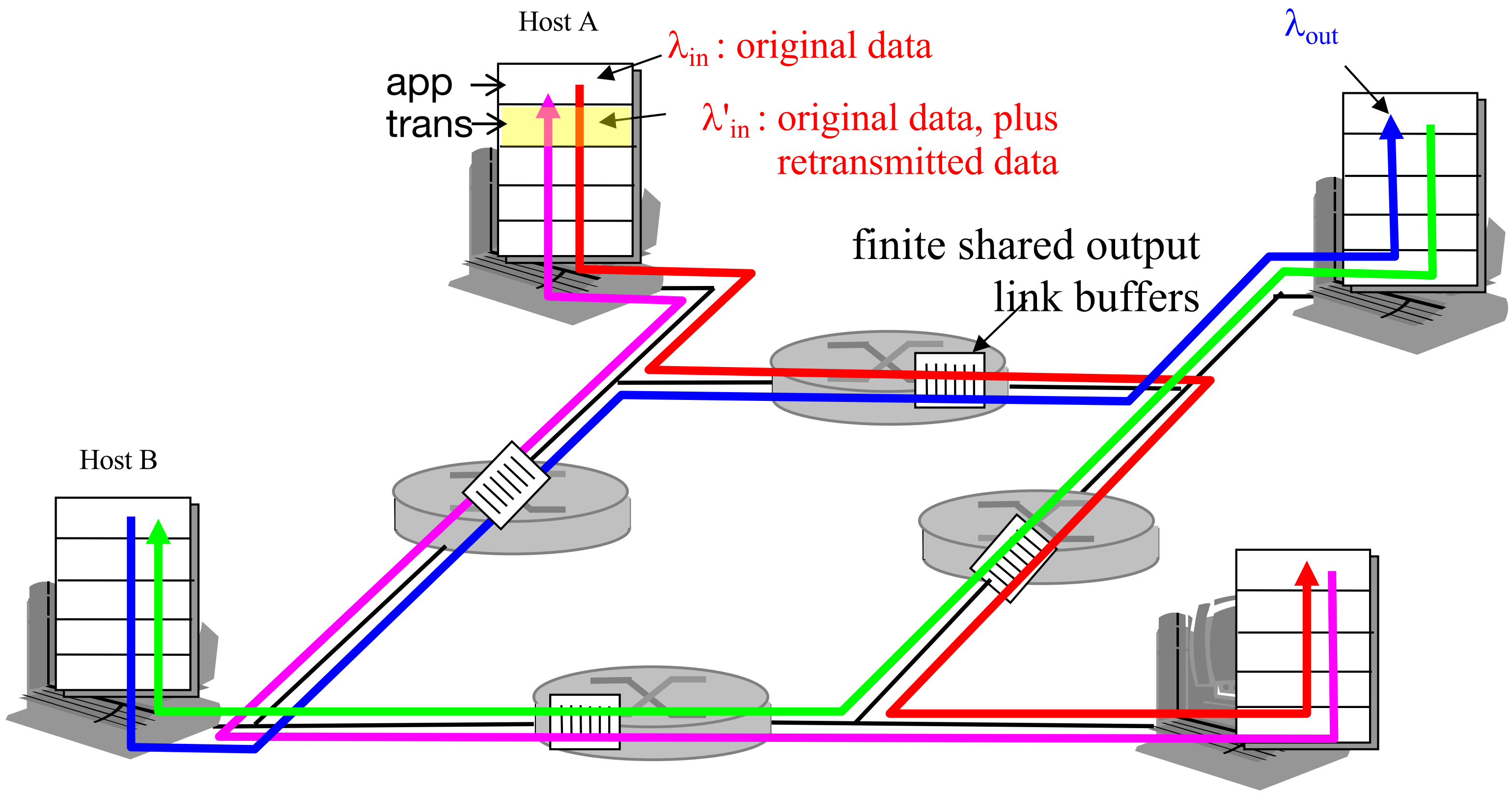
- 1974: 3-way handshake
- 1978: TCP and IP split into TCP/IP
- 1983 January 1: ARPAnet switches to TCP/IP
- **1986: Internet started seeing congestion collapses**
- 1987-1988: Van Jacobson fixes TCP, publishes a seminal paper (TCP-Tahoe)
  "Congestion Avoidance and Control"
  http://ccr.sigcomm.org/archive/1995/jan95/ccr-9501-jacobson.pdf
- 1990: added fast retransmit and fast recovery (TCP-Reno)

Another Illustration of Fast Recovery/Retransmit (Reno)

FYI



Congestion scenario 3



- Unneeded (superfluous) retransmissions
  - multiple copies of same packets go through overloaded links, reduce effective throughput
- When a packet is dropped, any "upstream transmission capacity" used for that packet was wasted

Congestion Control (CC)

(from textbook) Two basic approaches to CC:

End-to-end congestion control: no explicit feedback from network

- Hosts infer congestion from observed loss or delay

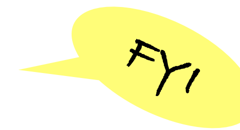
Network-assisted congestion control: routers provide feedback to end hosts

- A single bit congestion indication

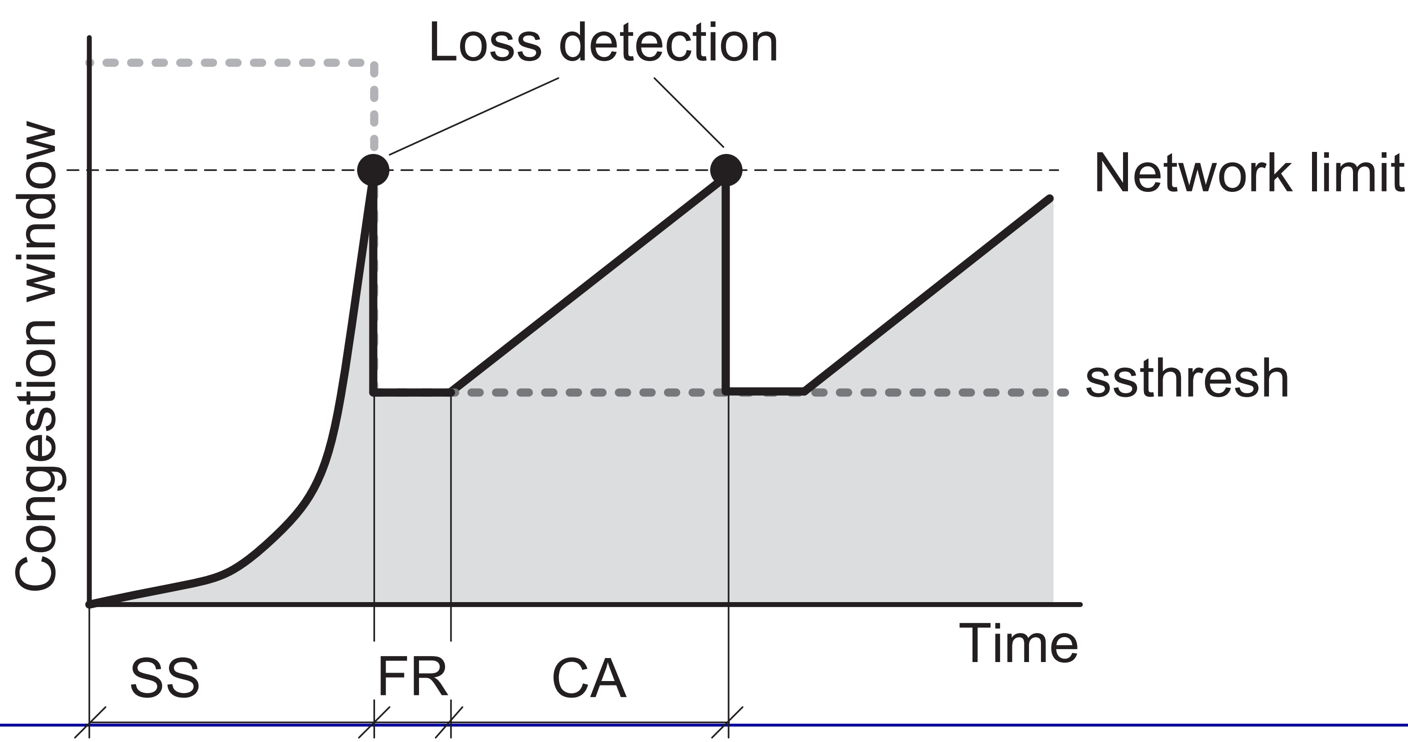
FYI: there is a 3rd and better approach: let the network *regulates* traffic to avoid congestion

- But *an IP network cannot do it*

TCP Fast Recovery

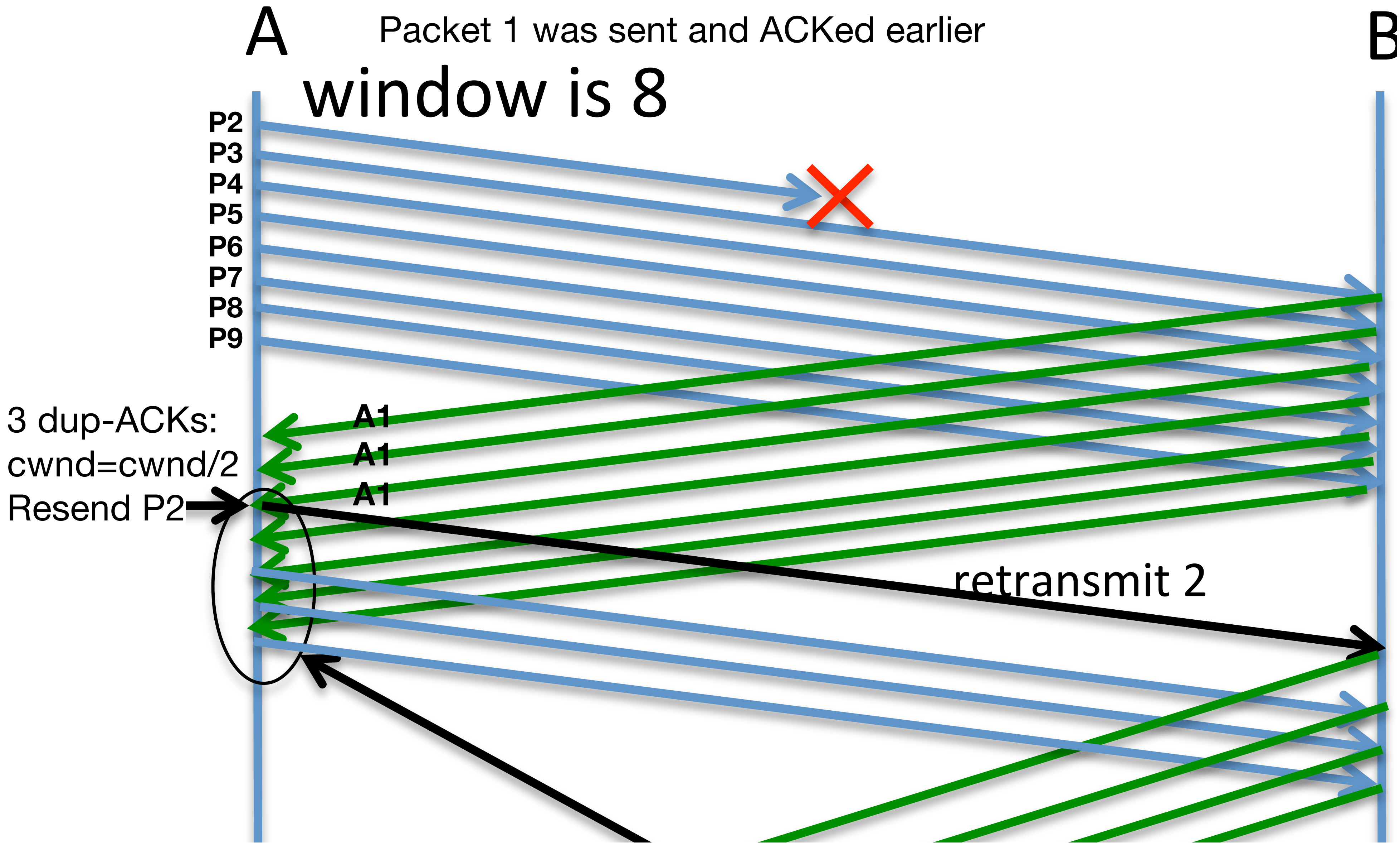


- cwnd: aims to limit the number of packets *inside network*
- Whenever a duplicate ACK arrives → a packet is out of network → increase cwnd by 1 segment (cwnd inflation)
- When the lost segment is ACKed: deflate cwnd to the right size

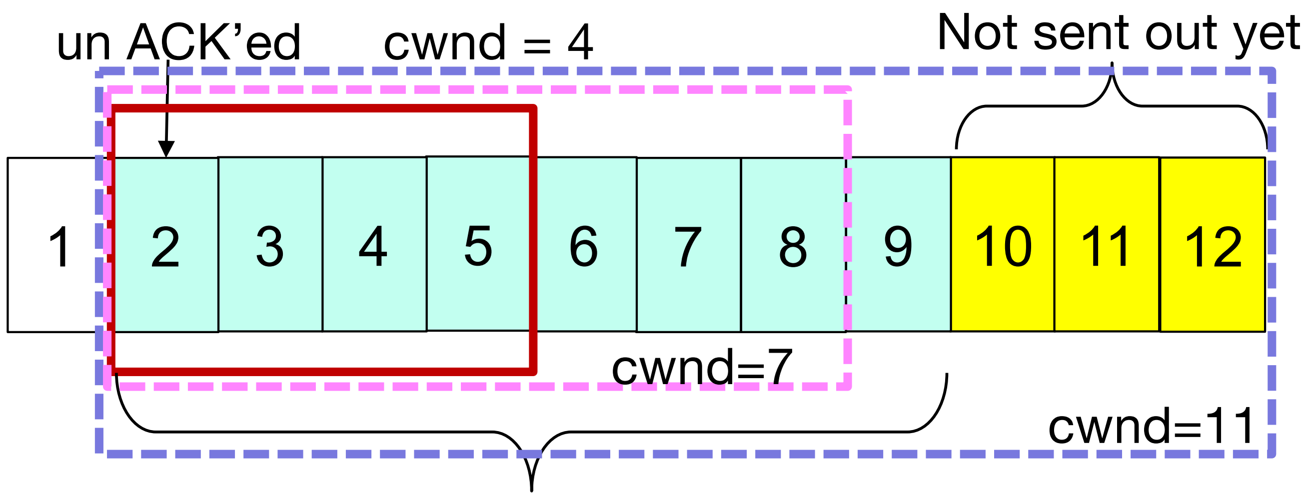


cwnd = limit on # of packets *inside network*

FYI



The current situation:



cwnd = 4, should allow 4 packets in the network

But we cannot slide the window to the right to allow more transmission

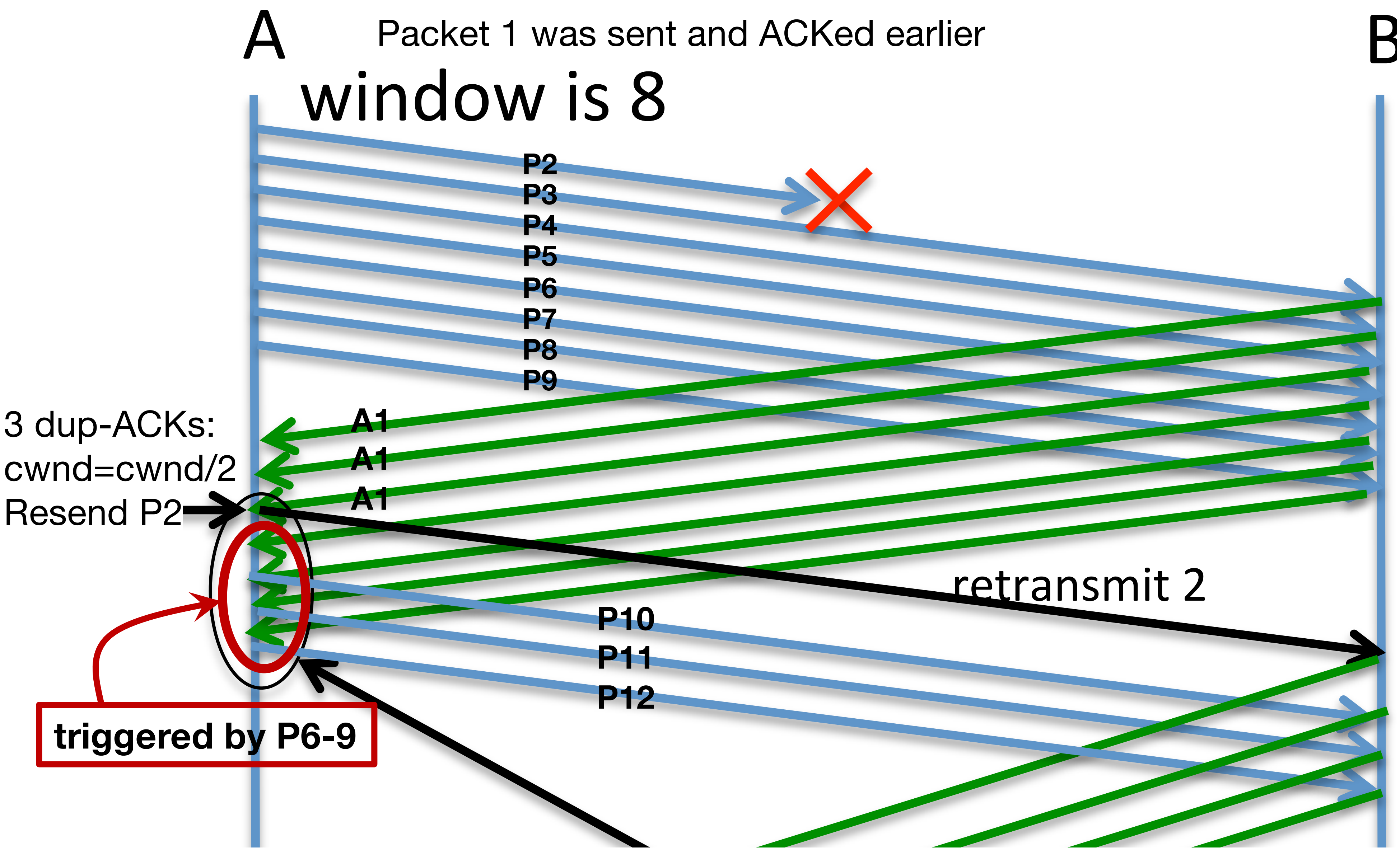
Why?

How to fix it 3 dup-ACKs inform us that 3 packets have been out of network
Inflate cwnd by 3 pkts→ cwnd = 4+3 = 7 (still nothing new can go yet)

Receive next dup-ACK (triggered by P6): cwnd =8: still can't send new packet
Receive next 3 dup-ACKs (triggered by P7-9): cwnd=11, sends P10-12

cwnd = limit on # of packets *inside network*

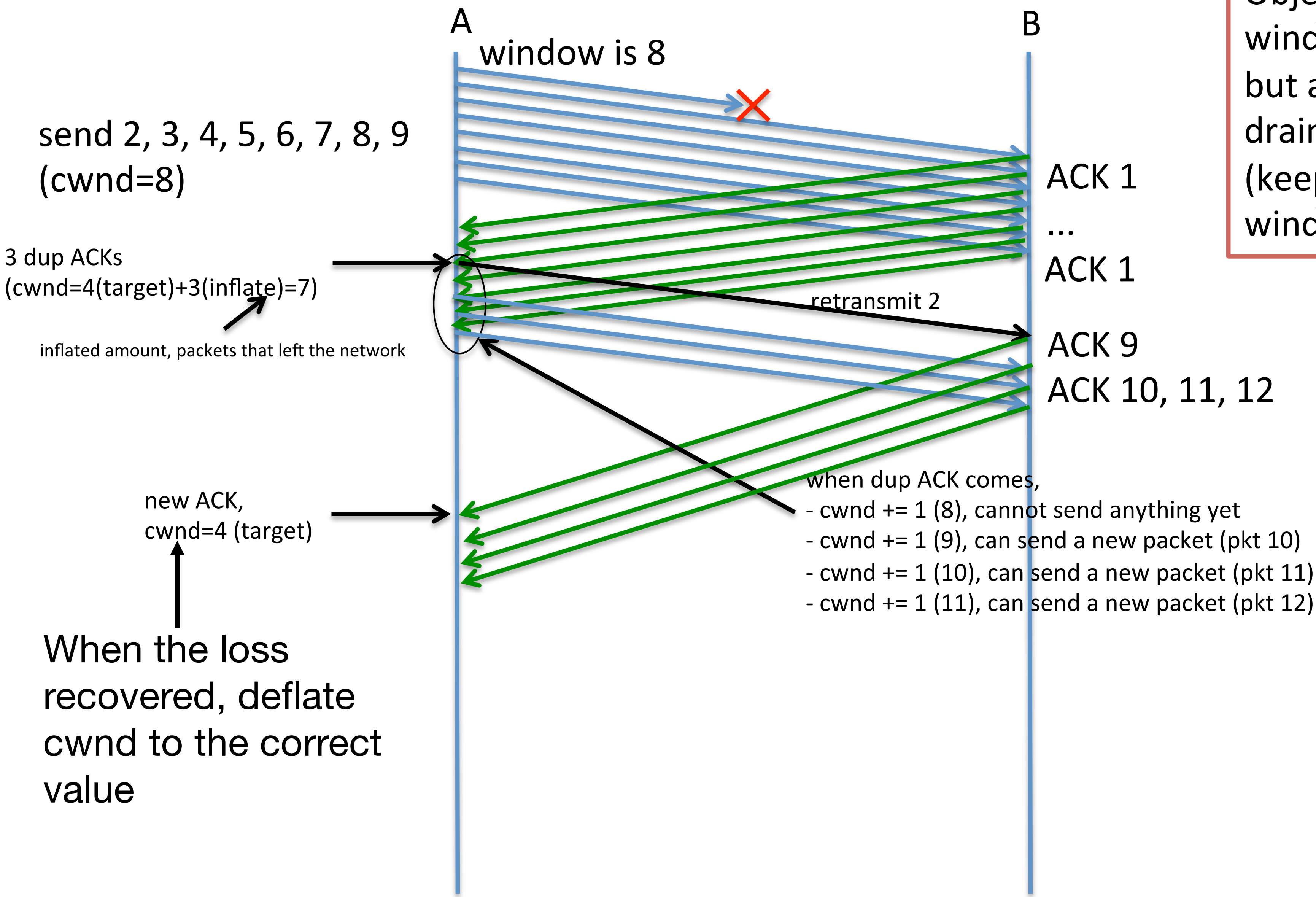
FYI



FYI

Fast Retransmit / Fast Recovery

Objective: cut window by half, but avoiding draining the pipe (keep ~half of window in flight)



Need better than loss-based congestion detection

FYI

- network traffic can be in one of 3 states
  - Under-Utilized: traffic load < link capacity, no queue
  - Over-Utilized: traffic load > link capacity, queues form
  - Saturated: queues full, packet loss occurs
- Loss-based control systems probe upward to the Saturated point, then try to back off quickly to assumed Under-Utilized state, to the let the queues drain
- Optimal traffic control: at the point of state change from Under to Over-utilized, not to reach the Saturated point