Congestion Control

Prof. Anja Feldmann, Ph.D.

(Based on slide deck of Computer Networking, 7th ed., Jim Kurose and Keith Ross.)

Outline

- *Connection-oriented* transport: TCP
  - Quick refresher on TCP *Segment structure*
    - Sequence numbers & Acknowledgements
  - Reliable data transfer
  - Flow control
  - Connection management
- Congestion control
  - Principles
  - Mechanism

TCP: Congestion Control

Motivated by ARPANET congestion collapse

Underlying design principle: ***Packet conservation***

- At equilibrium, inject packet into network only when one is removed
- Basis for stability of physical systems

Why was this not working?

- Connection does not reach equilibrium
- Spurious retransmissions
- Resource limitations prevent equilibrium

TCP Congestion Control: Solutions

Reaching equilibrium

- Slow start

Eliminates spurious retransmissions

- Accurate RTO estimation
- Fast retransmit

Adapt to resource availability

- Congestion avoidance

TCP Congestion Control: Basics

Keep a congestion window, ***cwnd***

- Denotes how much the network can absorb

Sender's maximum window:

- Min. (advertised receiver window, *cwnd*)

Sender's actual window:

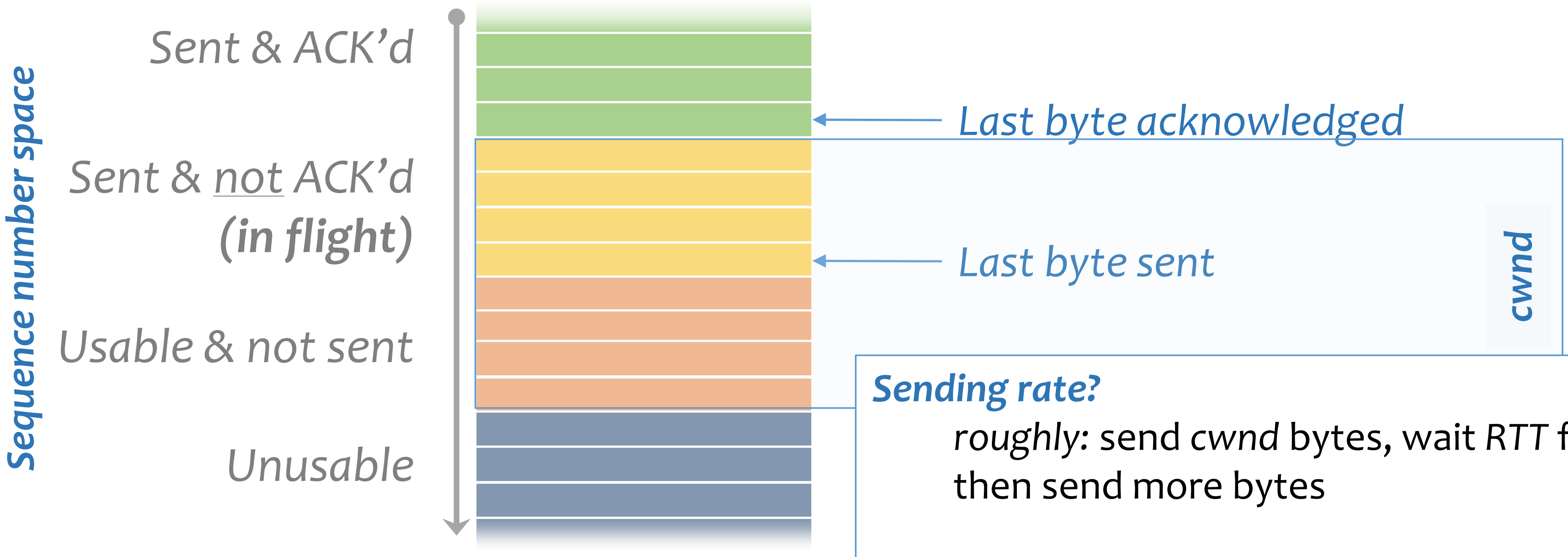
- Max. window – unacknowledged segments

If we have large actual window, should we send data in one shot?

- **No**, use *ACKs* to clock sending new data

TCP: Congestion Window

- ***cwnd*** is *dynamic*; function of perceived network congestion
- Sender limits transmission: *last-byte-sent – last-byte-ack'd ≤ cwnd*



Sending rate?

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

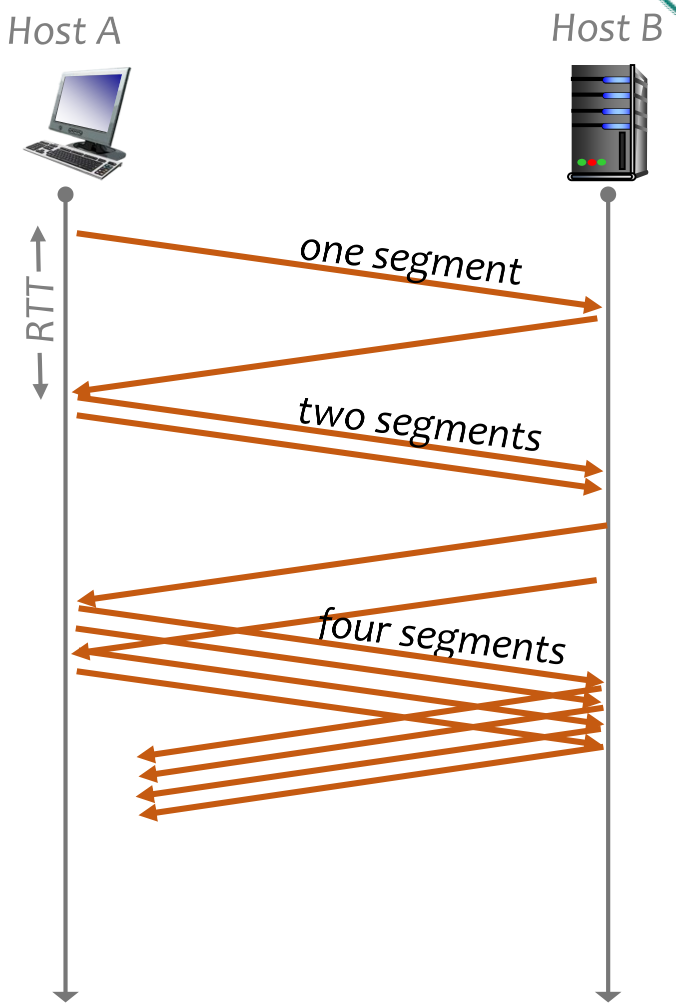
rate ≈ cwnd/RTT *(in bytes/s)*

TCP: Slow start

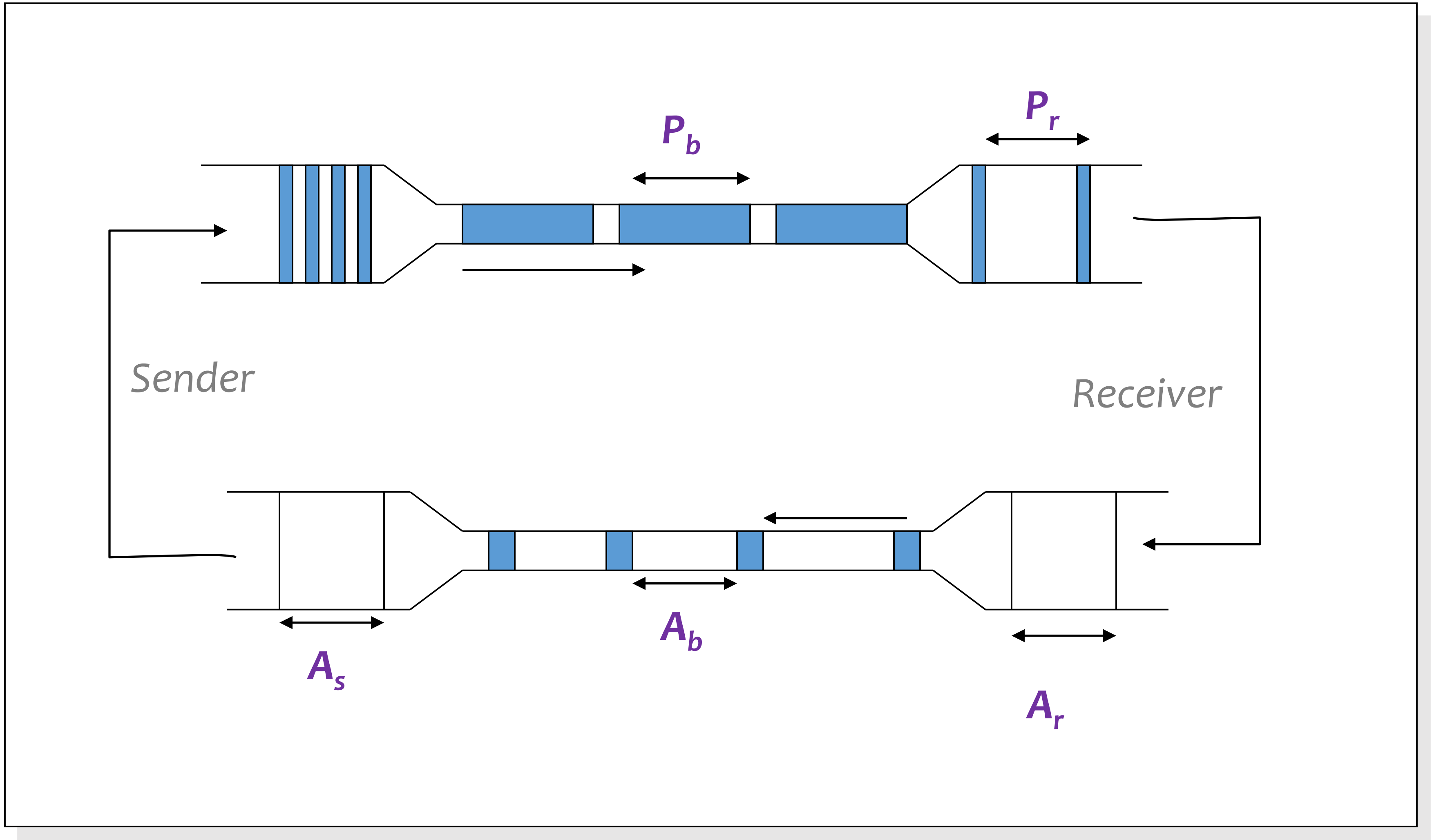
When connection begins, increase rate exponentially until first loss event:

- initially *cwnd* = **1 MSS**
- double *cwnd* *every RTT*
- done by incrementing *cwnd* for every ACK received

"slow start" *misleading*: initial rate is slow, but it ramps up exponentially fast!



TCP Self-Clocking



TCP: Detecting and 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

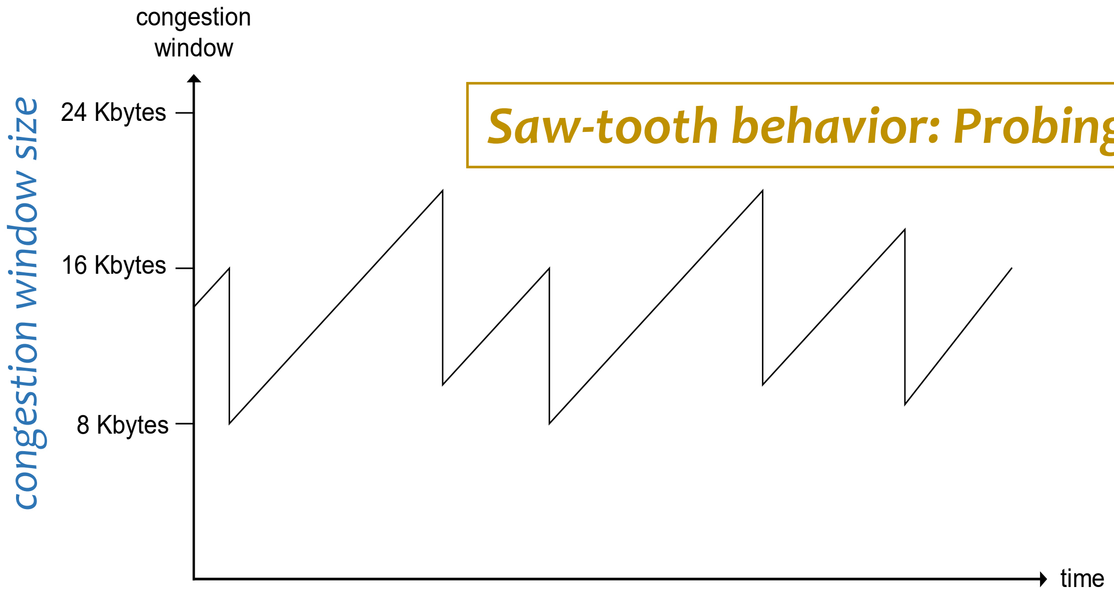
- dup ACKs indicate network capable of delivering some segments
- cwnd is cut in half window then grows linearly

TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)

TCP Congestion Control: AIMD

Additive increase, multiplicative decrease (AIMD)

- *Approach*: Increase transmission rate (*window size*)
  Probe 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

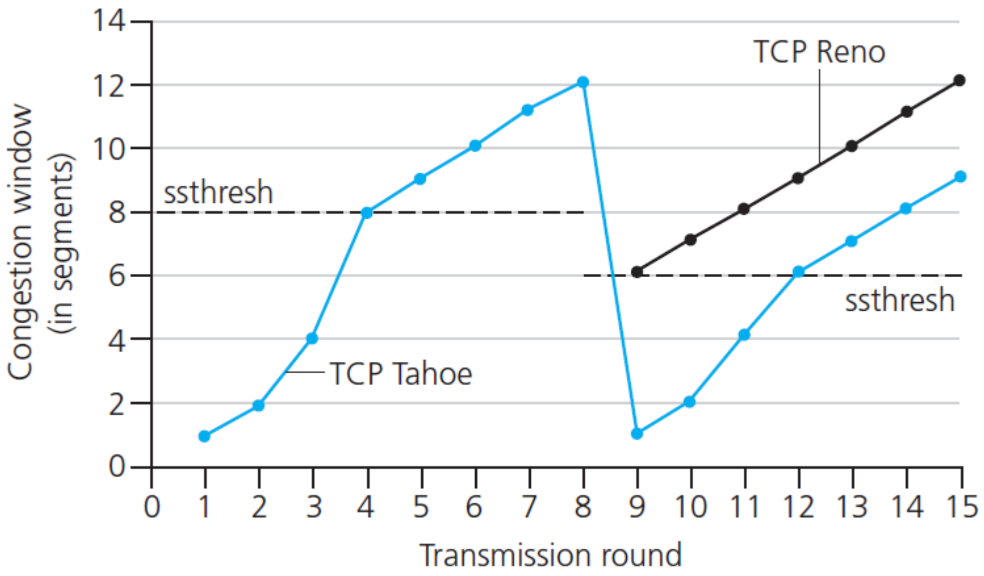


Saw-tooth behavior: Probing for bandwidth

TCP: From slow start to cong. avoidance

When should the exponential increase switch to linear?

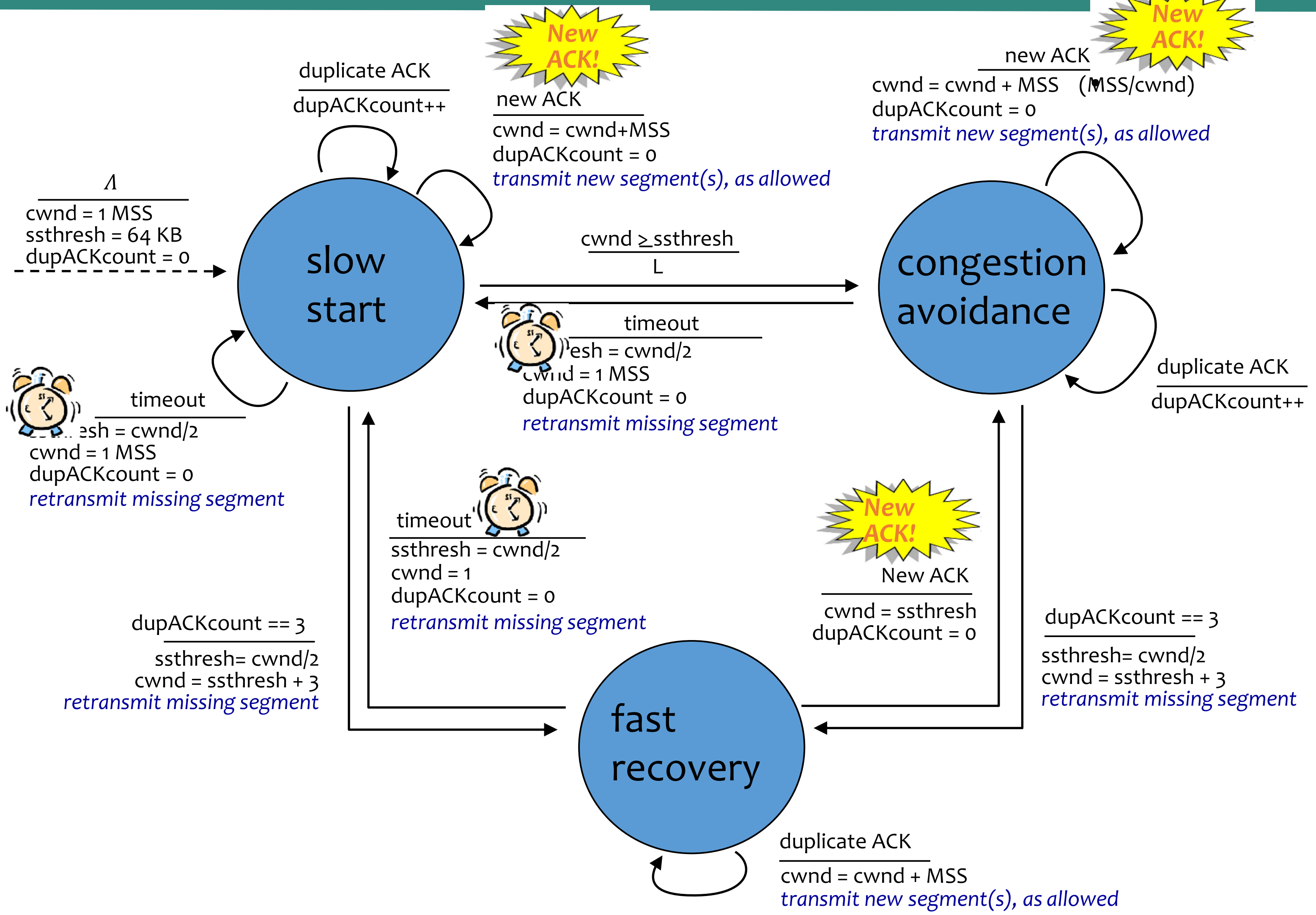
- When *cwnd* gets to ***one half*** of its value before *timeout*.



Implementation:

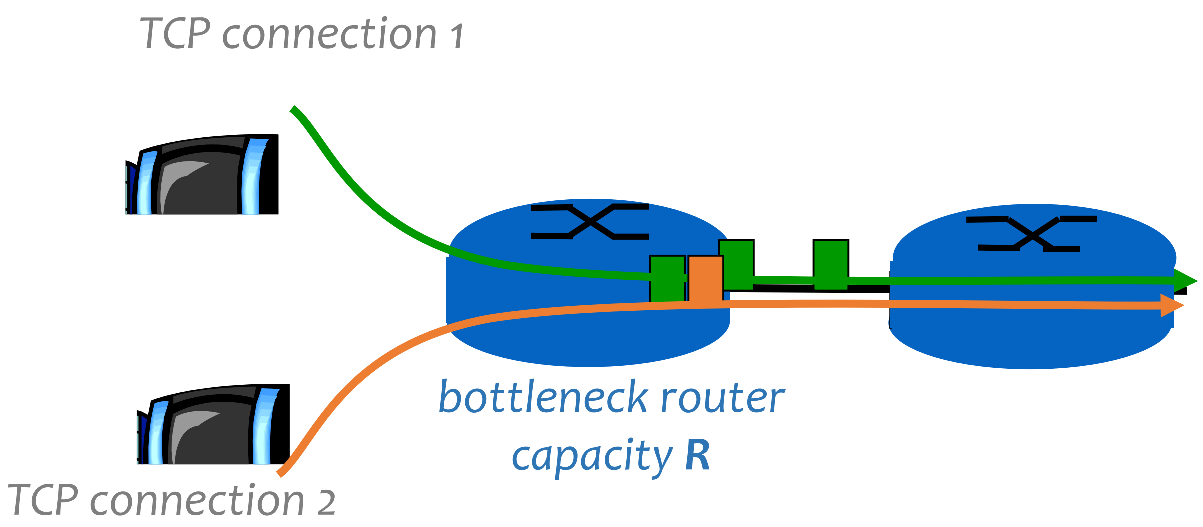
- Variable ***ssthresh***
- On a loss, ***ssthresh*** is set to *one half* of *cwnd* of the value just before the loss event

TCP congestion control: Summary



TCP Fairness

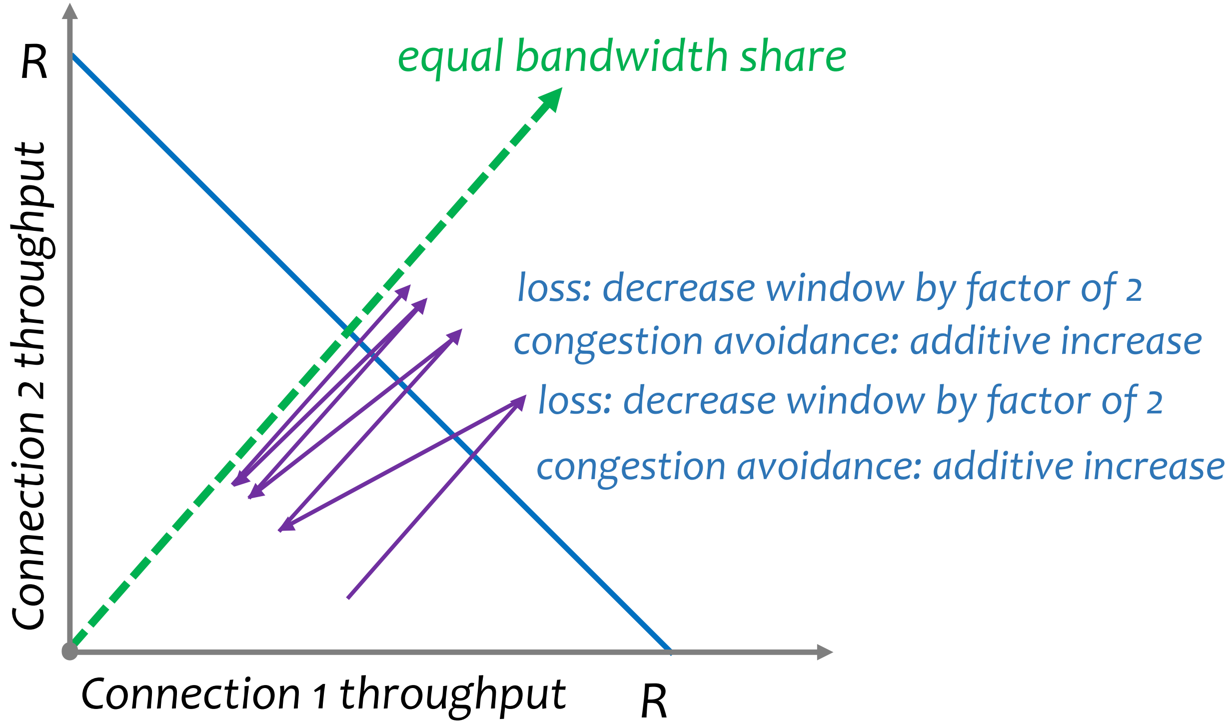
Fairness goal: If N TCP sessions share same *bottleneck* link, each should get 1/N of link capacity



Why is TCP fair? (Ideal Case)

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



Assumptions for TCP Fairness

- Window under consideration is large enough
- Same RTT
- Similar TCP parameters
- Enough data to send
- ...

Outline

- *Connection-oriented* transport: TCP
- Congestion control
  - Principles
  - Mechanism
- Up next: TCP variants