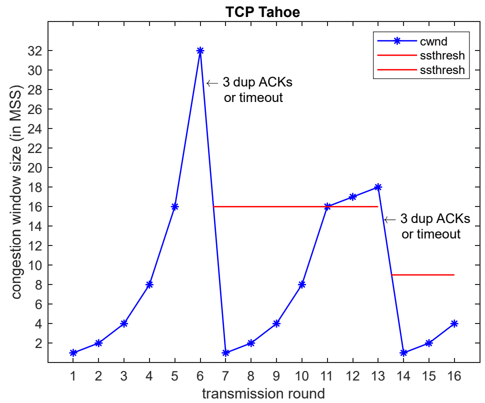
Chapter 3: roadmap

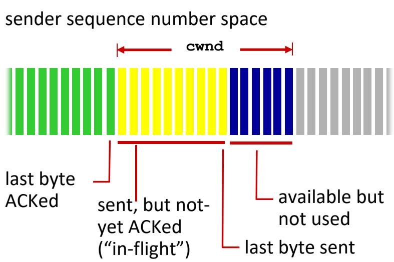
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

TCP congestion control: overview

- sender adjusts cwnd dynamically to limit the tx rate (cwnd ∝ rate)
- sender perceives congestion by
  - timeout and 3 duplicate ACKs
- TCP Tahoe:
  - "slow start" phase
    - start with cwnd = 1 (MSS)
    - if no loss occurs, double cwnd every RTT (until cwnd reaches ssthresh and goes to "congestion avoidance")
    - if loss occurs, reset cwnd = 1
  - "congestion avoidance" phase
    - if no loss occurs, cwnd grows linearly
    - if loss occurs, go to "slow start"



TCP congestion window: details

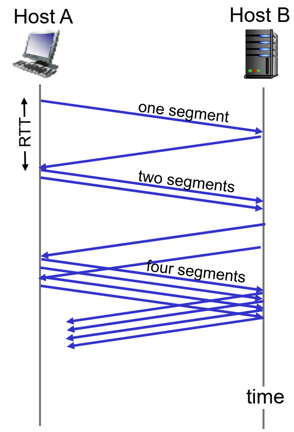


- TCP sender limits transmission: `LastByteSent – LastByteAcked ≤ cwnd`
  - *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

TCP slow start: details

- 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
- initial rate is slow, but ramps up exponentially fast



Improvement for congestion avoidance: AIMD

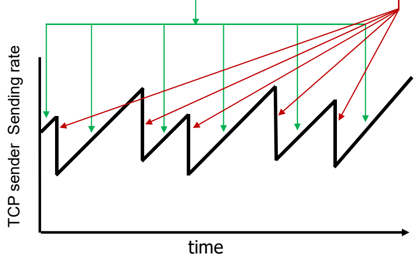
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event in a AIMD manner

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

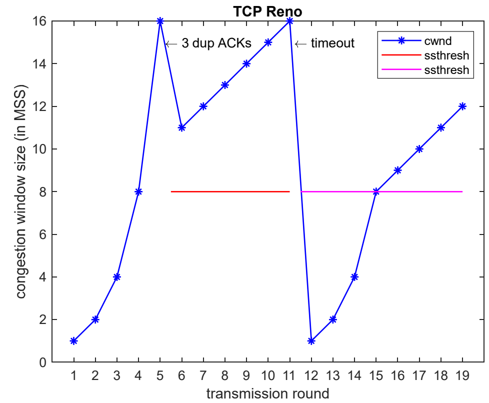
cut sending rate in half at each loss event



TCP Reno

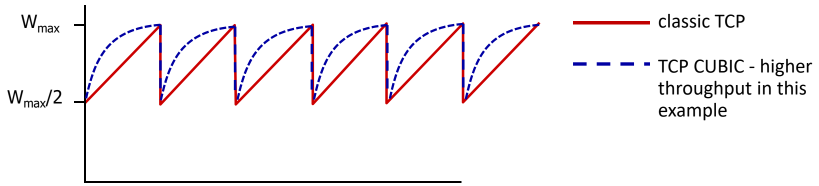
TCP Reno: cwnd is

- Cut to 1 MSS when loss detected by **timeout**
  - then enter "slow start"
  - ssthresh ← cwnd/2
    cwnd ← 1 · MSS
- Cut in half on loss detected by **3 duplicate ACK**
  - then enter "congestion avoidance"
  - more precisely,
    ssthresh ← cwnd/2
    cwnd ← cwnd/2 + 3 · MSS



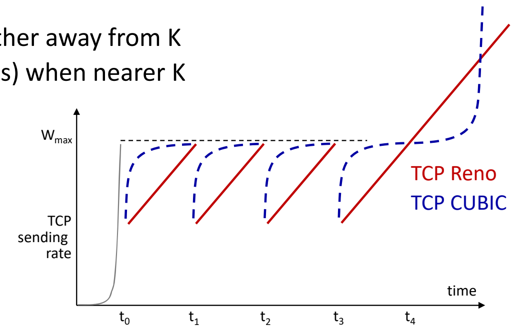
TCP CUBIC (default in Linux)

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



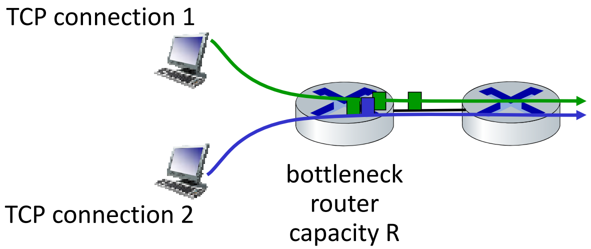
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



TCP fairness

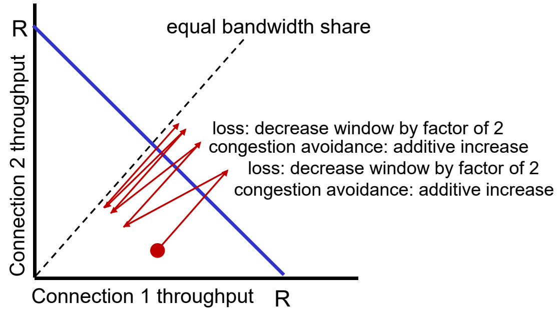
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Reno Fair?

Example: two competing TCP connections:

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



Is TCP fair?

A: Yes, under idealized assumptions:
- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be “fair”?

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
- there is no “Internet police” policing use of congestion control

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 1R/10
  - new app asks for 11 TCPs, gets 11R/20