Transport layer: overview

- Transport services and protocols
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control
- Evolution of transport-layer functionality

TCP congestion control

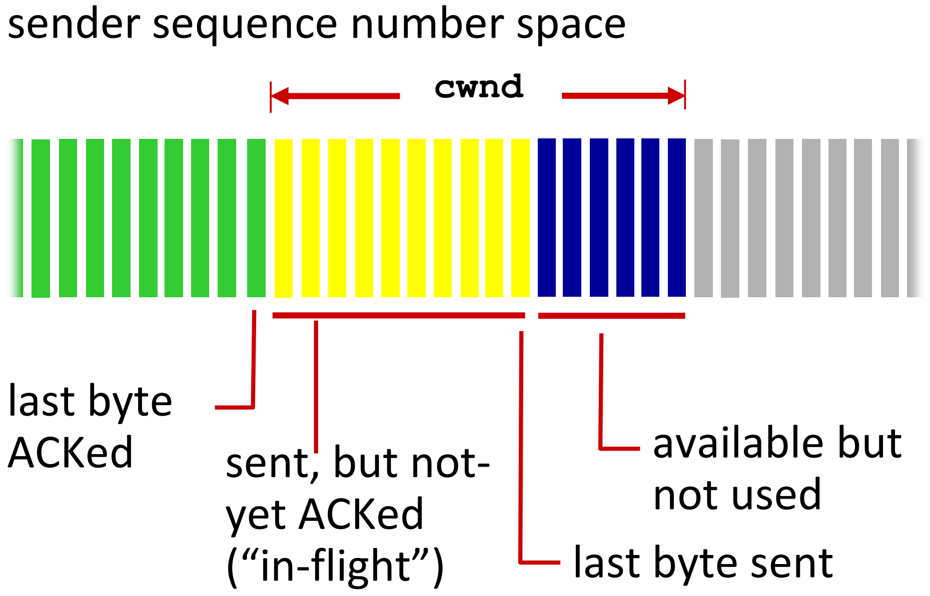
- how does a TCP sender limit its sending rate?
- how does a TCP sender perceive a congestion?
- what algorithm is used to change the sending rate?

```
LastByteSent - LastByteAcked ≤ min{cwnd, rwnd}
```

TCP congestion control: details

- slow start
- congestion avoidance
- fast recovery

TCP congestion control: details



TCP sending behavior:

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

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

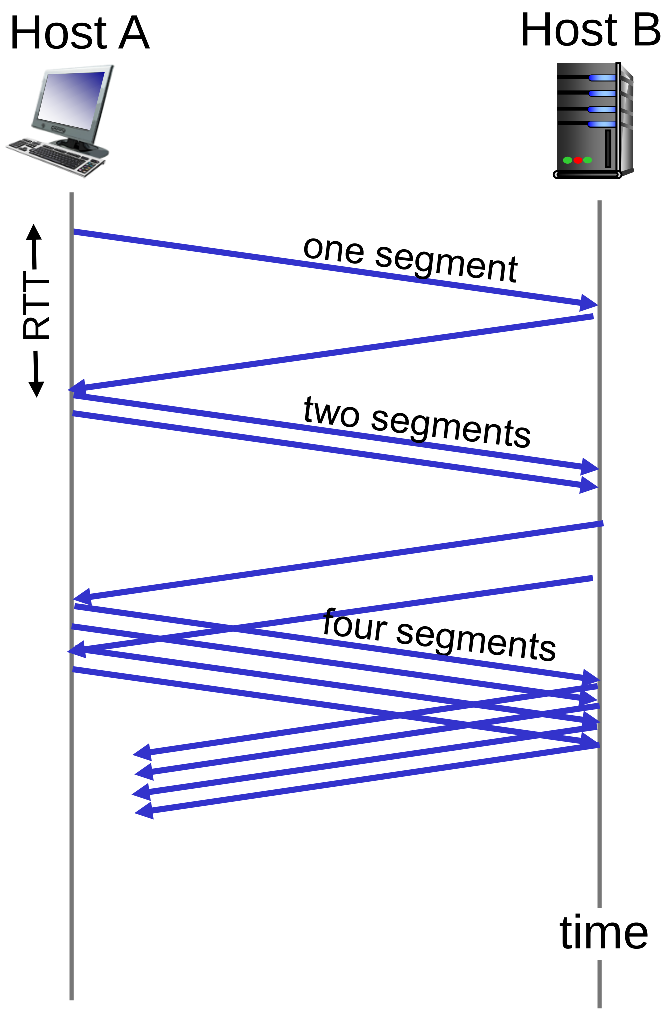
MSS=500 bytes & RTT=200 msec

I.S.R. = 500/200 = 20 kbps

- TCP sender limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

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



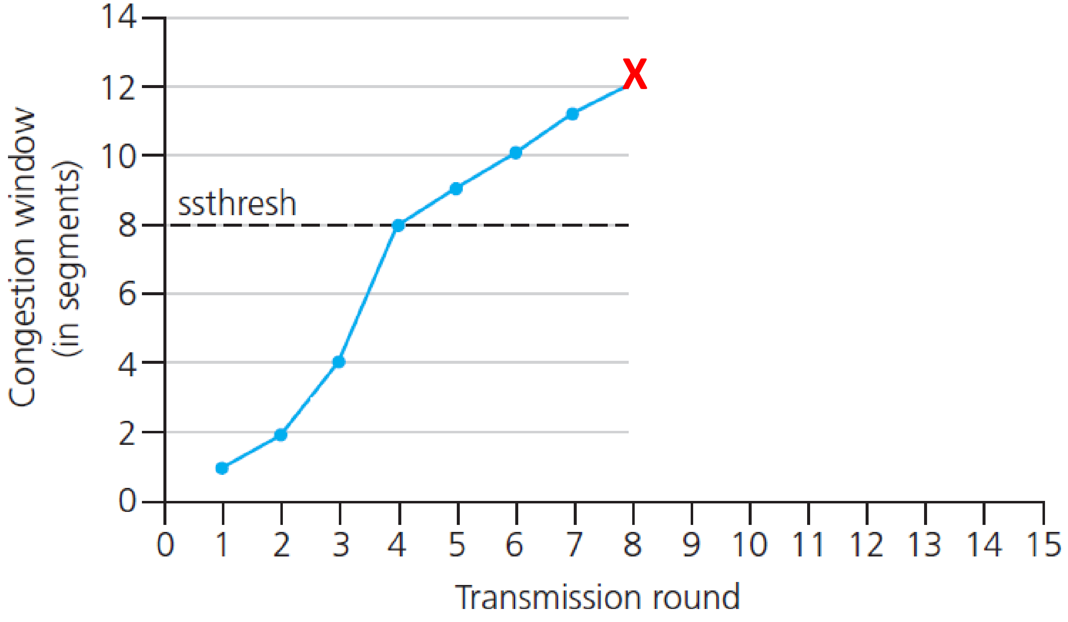
TCP: from slow start to congestion avoidance

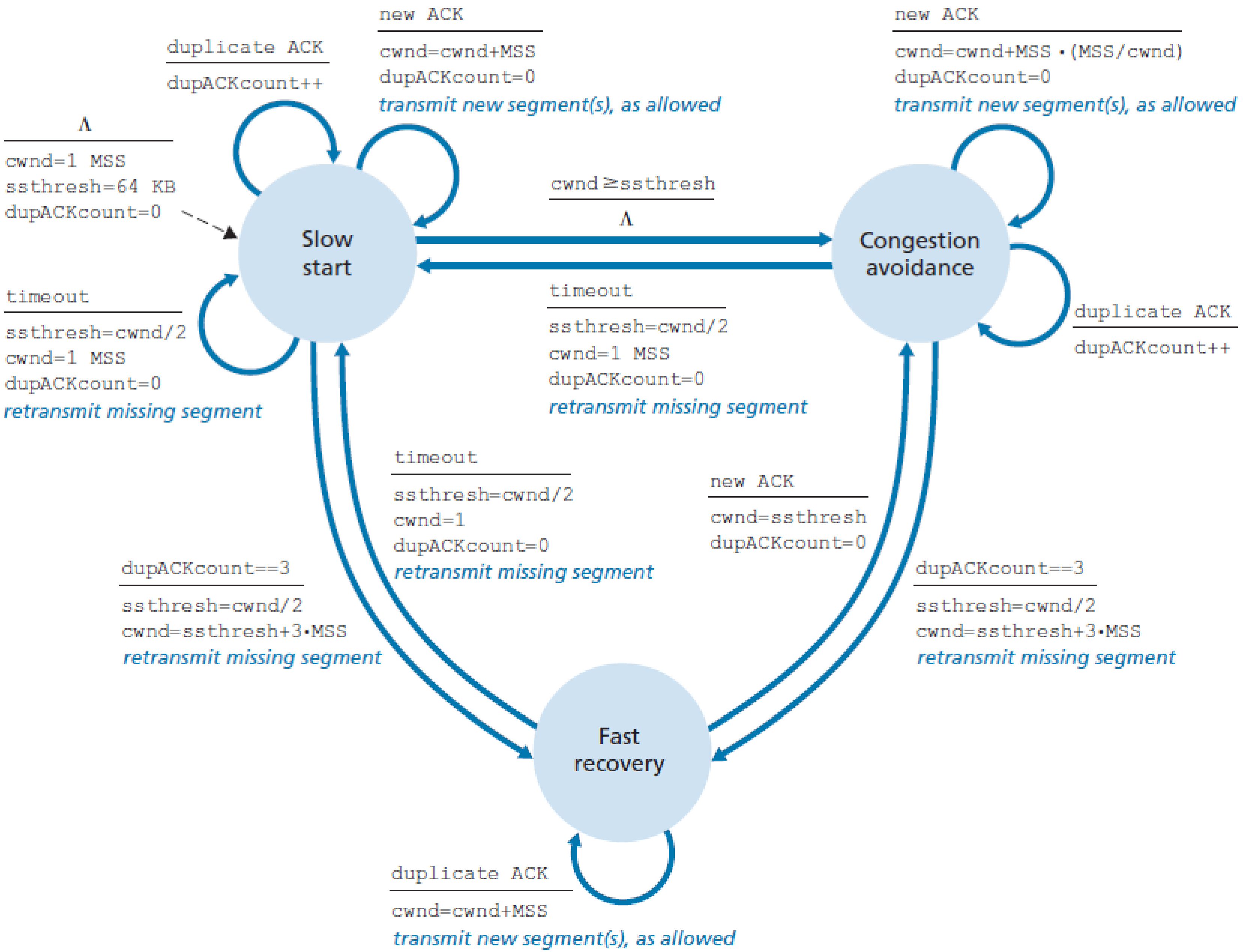
Q: when should the exponential increase switch to linear?

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





Slow start
Congestion avoidance
Fast recovery
duplicate ACK
dupACKcount++
new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
cwnd ≥ ssthresh
Λ
new ACK
cwnd=cwnd+MSS • (MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed
duplicate ACK
dupACKcount++
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
duplicate ACK
cwnd=cwnd+MSS
transmit new segment(s), as allowed

TCP congestion control: AIMD

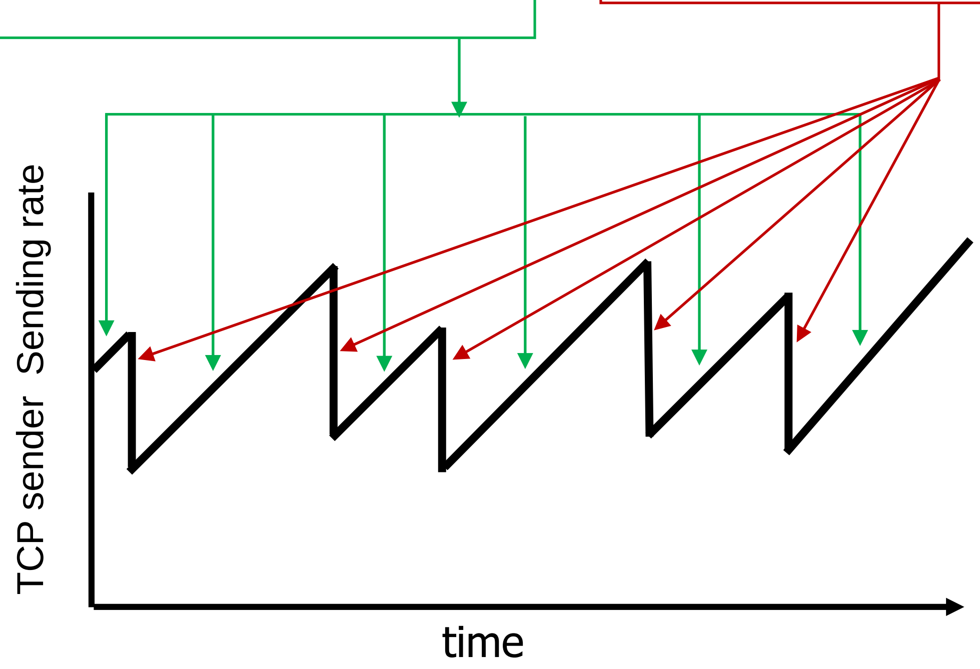
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

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

Multiplicative Decrease

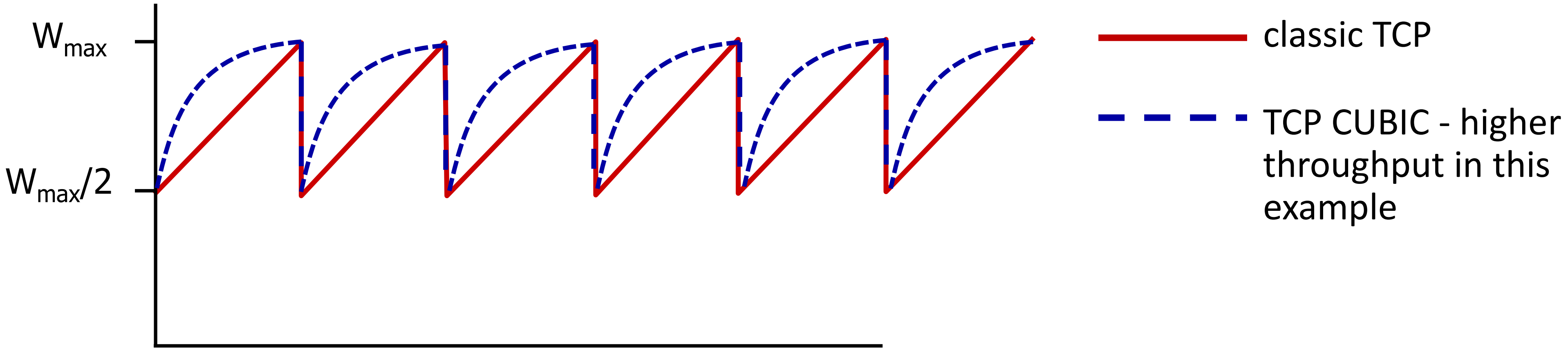
cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP CUBIC

- W_{max}: sending rate at which congestion loss was detected
- K: point in time (future) when TCP window size will reach W_{max}
- t: current time
- after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



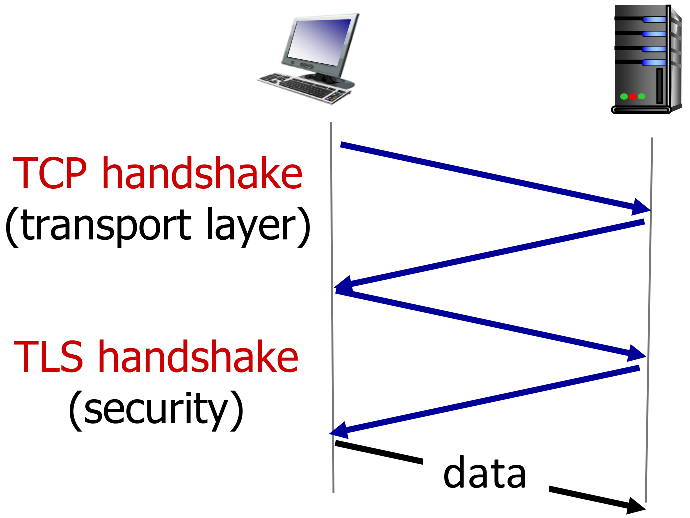
Transport layer: roadmap

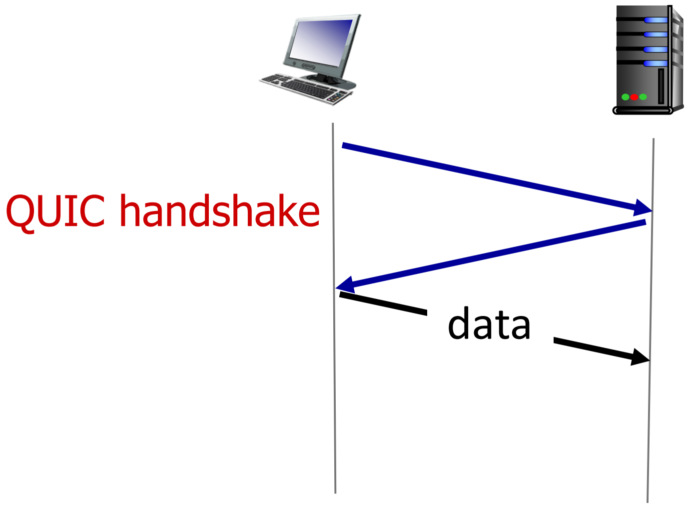
- Transport services and protocols
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control
- Evolution of transport-layer functionality

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)

QUIC: Connection establishment





TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake