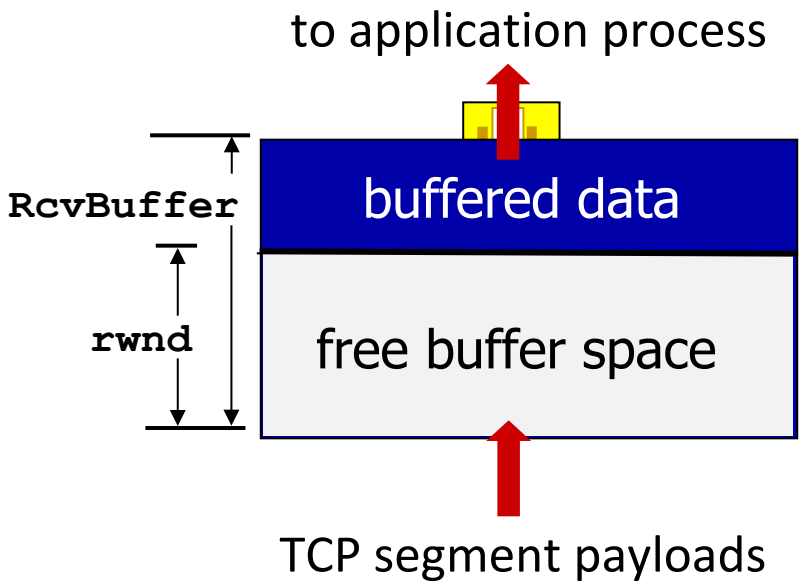
Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer: Go-Back-N, SR
  - reliable data transfer: TCP
  - TCP connection management
  - TCP flow control
  - TCP congestion control

Recap: TCP flow control

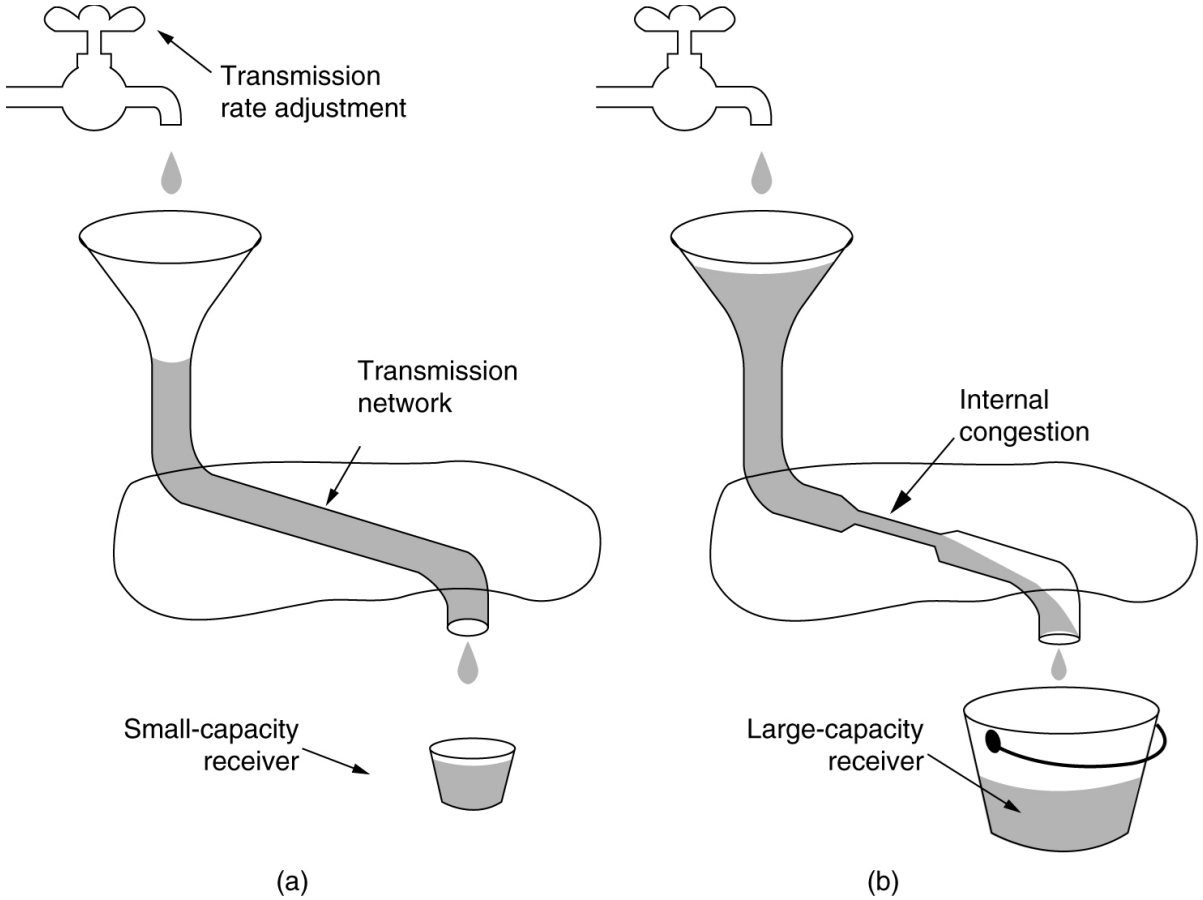
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
- ***sender*** limits amount of unACKed ("in-flight") data to received **`rwnd`**

  `LastByteSent- LastByteAcked ≤ rwnd`
- guarantees receive buffer will not overflow

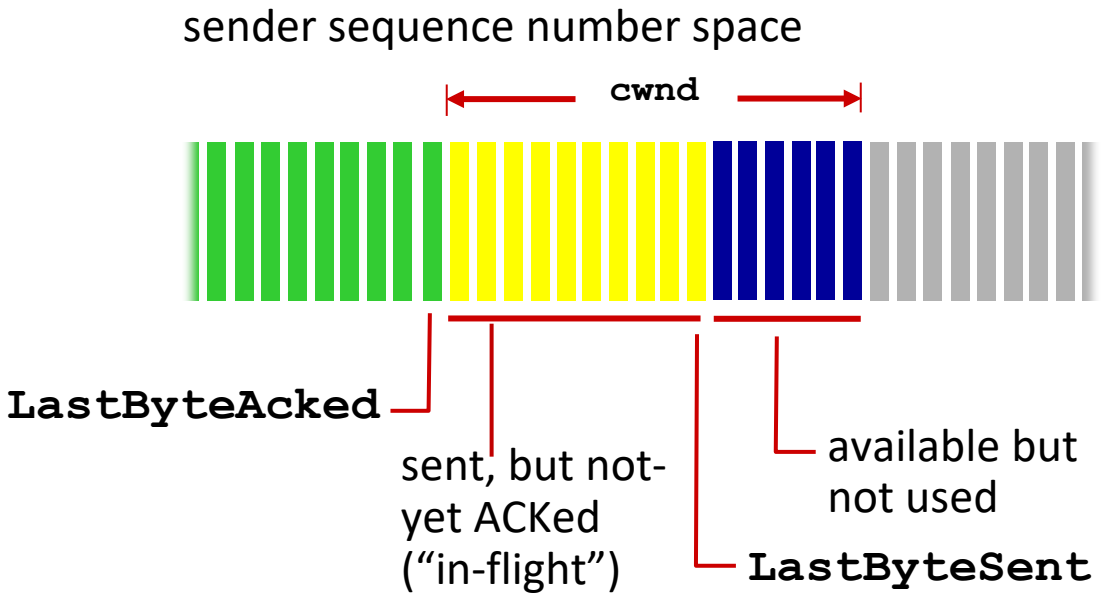


TCP receiver-side buffering

Flow vs congestion control



TCP congestion control



TCP sending behavior:

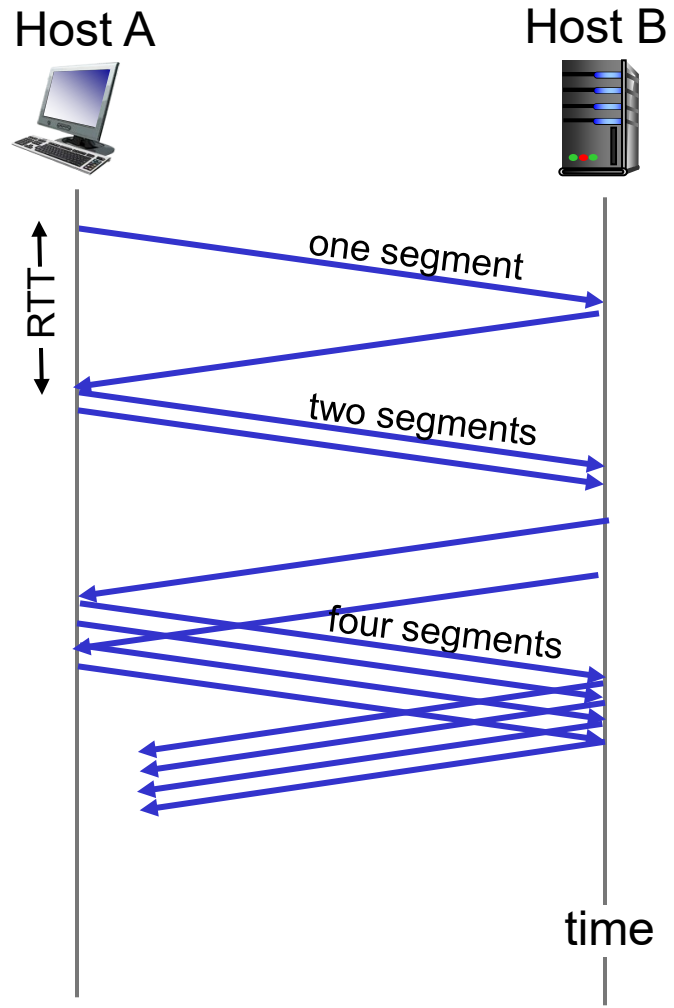
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP ***sender*** limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion
  - initially, cwnd = 1 MSS (maximal segment size)

Slow start

- when connection begins, increase rate exponentially until first loss event
  - on each new ack
    - cwnd += 1 MSS
    - effectively, doubling cwnd every RTT
  - “start small, but grow really fast”
    - Q: why?



Congestion avoidance

- Congestion avoidance
  - Slow start threshold: *ssthresh*
  - when cwnd > ssthresh
  - on each new ack
    - cwnd += MSS^2/cwnd
    - effectively, cwnd += 1 MSS every RTT
    - linear increment

Q: why increase cwnd to “avoid” congestion?

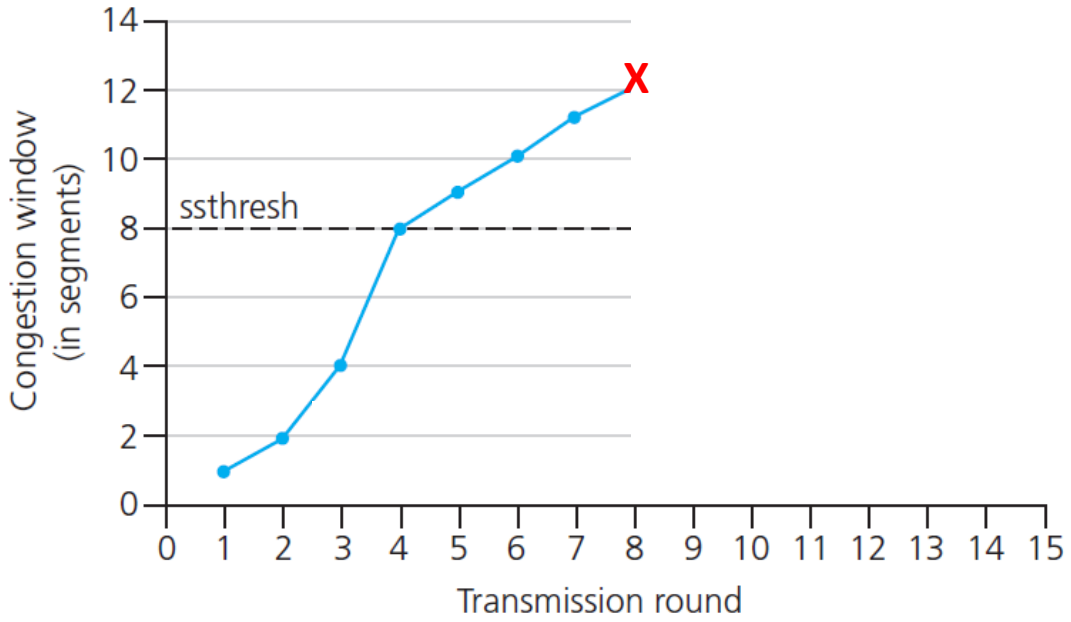
TCP: from slow start to congestion avoidance

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



https://media.pearsoncmg.com/ph/esm/ecs_kurose_compnetwork_8/cw/content/interactiveanimations/tcp-congestion/index.html

Network congestion

- cwnd is always increased in slow-start and congestion avoidance
  - network congestion is inevitable
- Network congestion indicator
  - TCP treats packet loss as network congestion
- Packet loss indicators
  - acknowledgment timeout
  - 3 duplicate acknowledgments

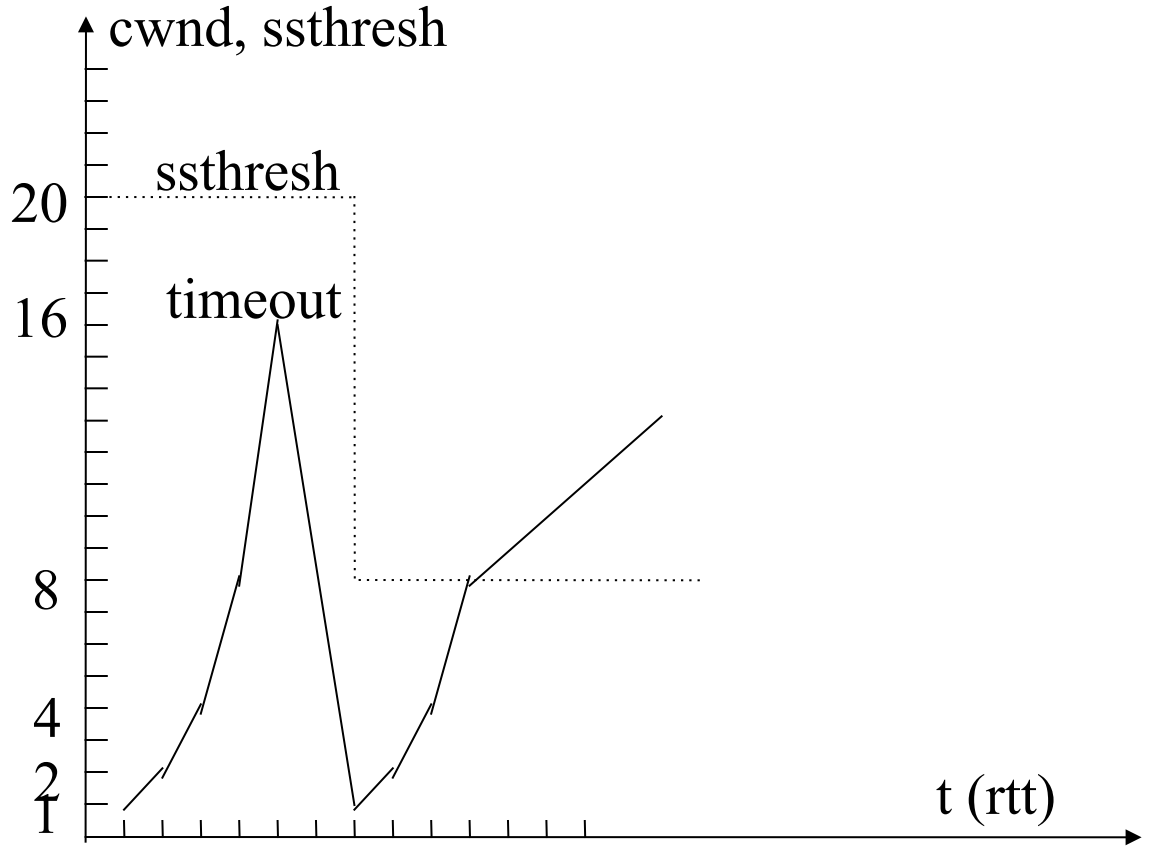
Q: packet loss == network congestion?

Timeout retransmit

- Congestion control
  - ssthresh = cwnd / 2
  - cwnd = 1 MSS
  - followed by slow-start
- Error control
  - retransmit packet
  - restart timer

Congestion window

- Slow-start
- Congestion avoidance
- Timeout retransmission
  - TCP timeout is quite conservative
  - pay attention to how ssthresh is adjusted!

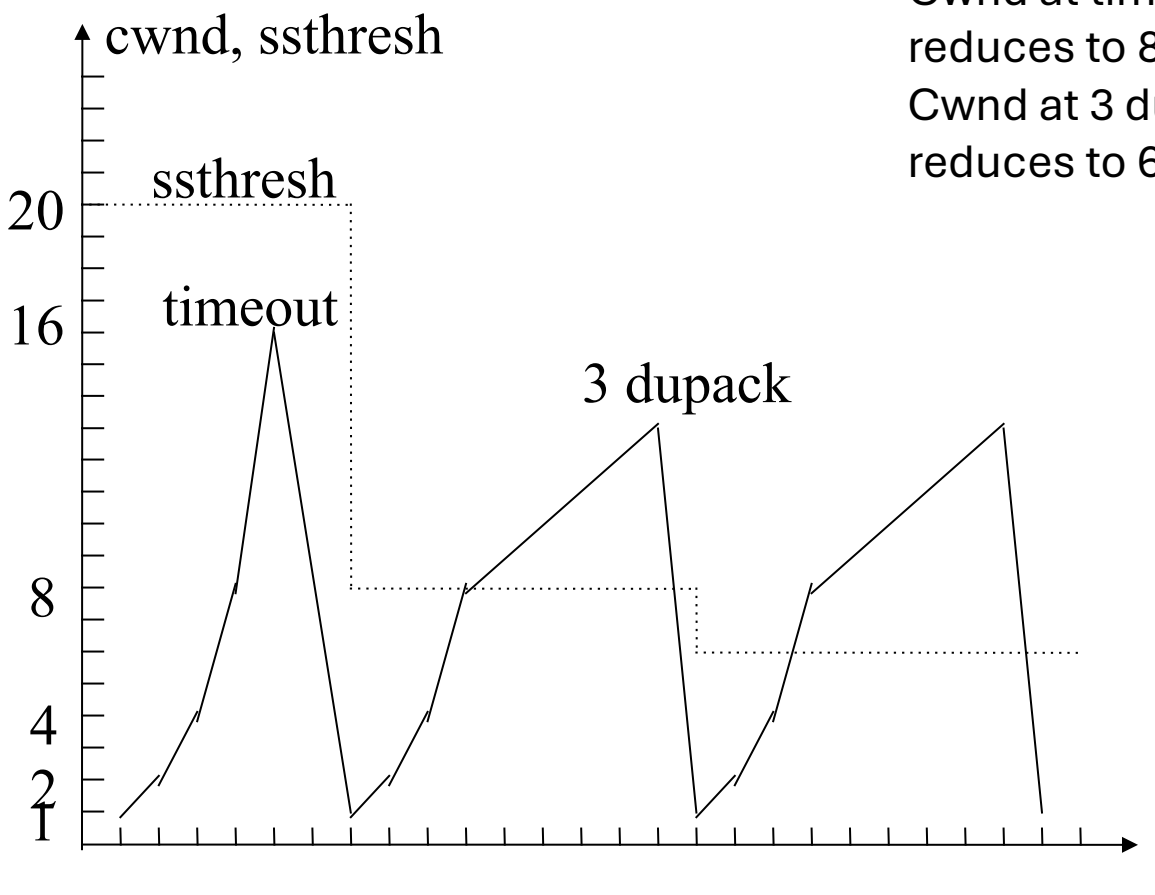


Fast retransmit

- Duplicate acknowledgment
  - example
    - rcv: [0, 499], [500, 999], [1500, 1999], [2000, 2499], [2500, 2999]
    - ack: 500, 1000, 1000, 1000, 1000 (3rd dupack)
- Congestion control (fast retransmit)
  - on 3rd dupack: ssthresh=cwnd/2; cwnd=1 MSS
  - followed by slow start
- Error control
  - retransmit: [1000,1499]

Q: dupack == loss?

Fast retransmit: cwnd



Cwnd at timeout: 16; ssthresh reduces to 8
Cwnd at 3 dupack: 12; ssthresh reduces to 6

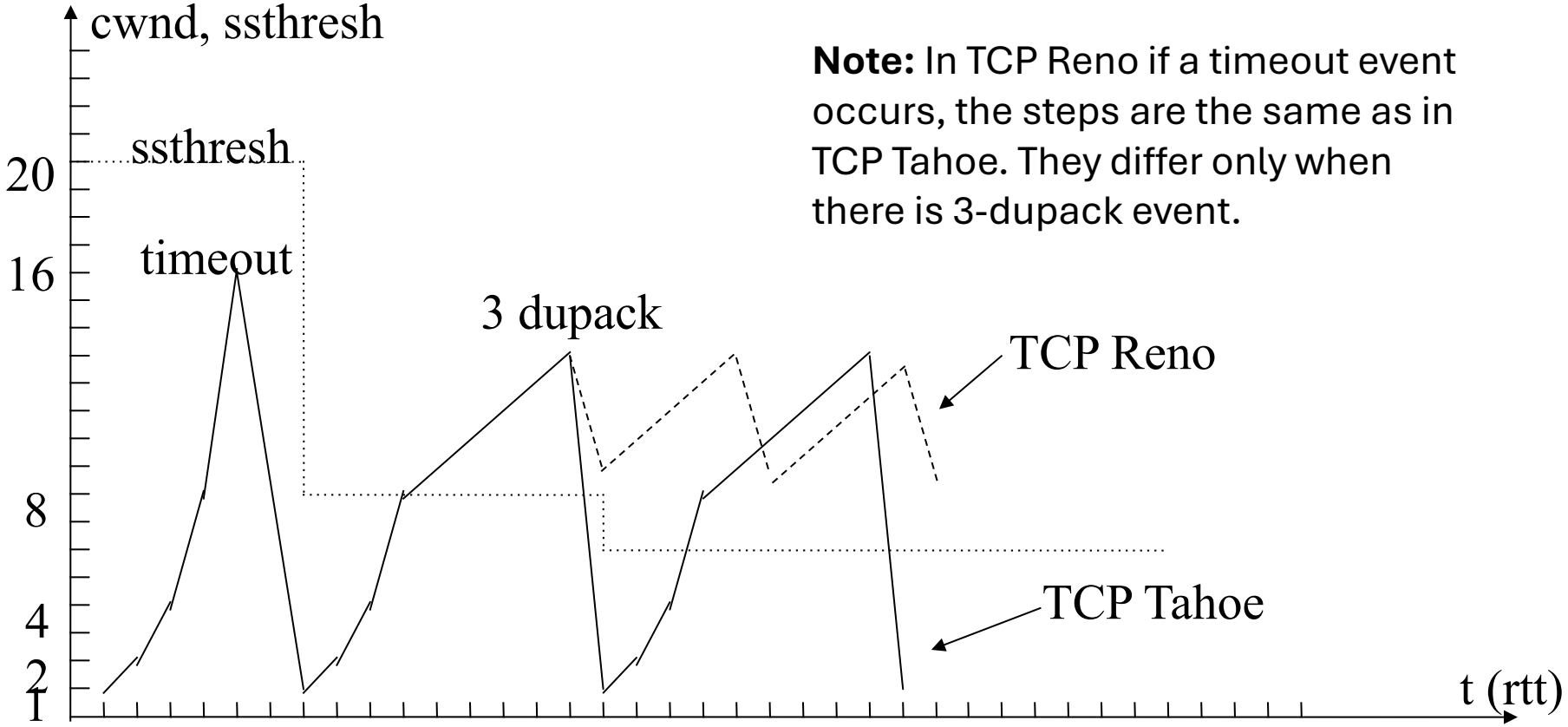
Q: why “fast” retransmit?

TCP Reno: Fast recovery

- Duplicate acknowledgment
  - example
    - rcv: [0, 499], [500, 999], [1500, 1999], [2000, 2499], [2500, 2999]
    - ack: 500, 1000, 1000, 1000, 1000 (3rd dupack)
- Congestion control (fast recovery)
  - on 3rd dupack: ssthresh=cwnd/2, cwnd= ssthresh + 3
  - followed by congestion avoidance
- Error control
  - retransmit: [1000,1499]

Q: fast transmit vs recovery?

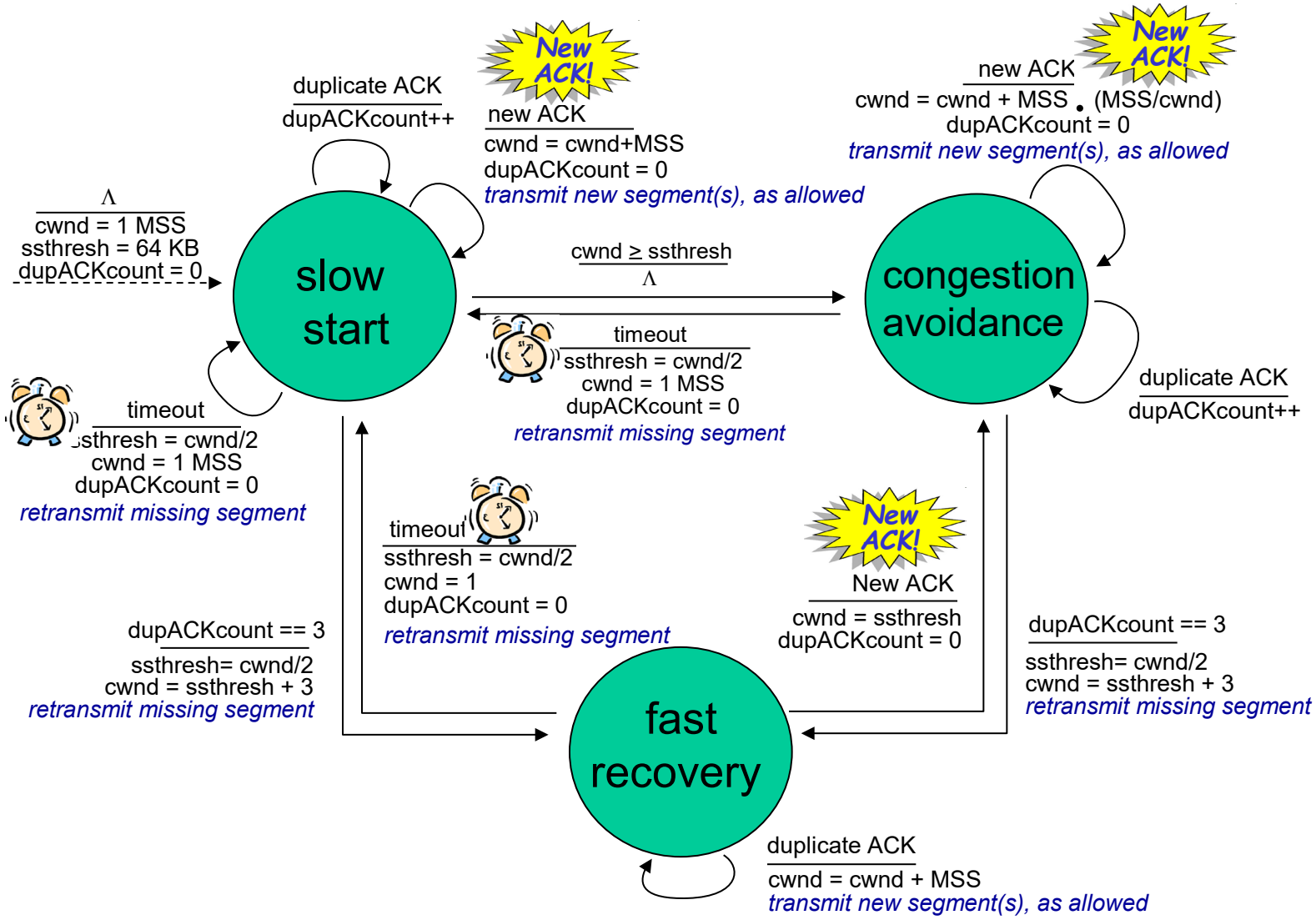
Fast recovery: cwnd



Note: In TCP Reno if a timeout event occurs, the steps are the same as in TCP Tahoe. They differ only when there is 3-dupack event.

Q: why fast recovery?

Summary: TCP Reno congestion control



https://media.pearsoncmg.com/ph/esm/ecs_kurose_compnetwork_8/cw/content/interactiveanimations/tcp-congestion/index.html

TCP congestion control: AIMD

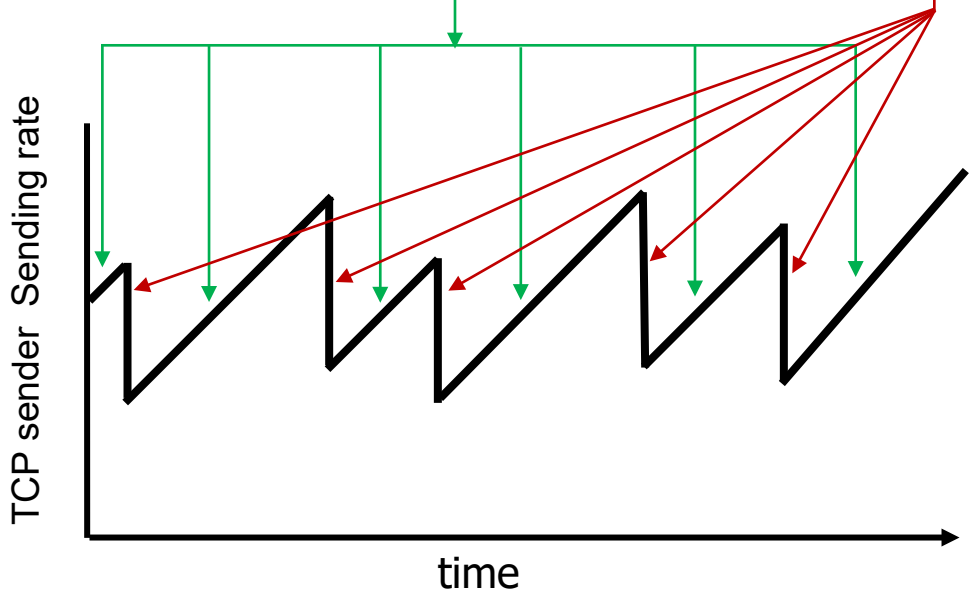
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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 AIMD: more

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

Questions: TCP Reno

- Num timeouts?
- Num 3-duplicate acks?
- Num of congestion avoidance phases?
- Number of slow start phases?
- Verify cwnd values
- sshthresh values?