暨南大學
JINAN UNIVERSITY

International School
Jinan University

Computer Networks

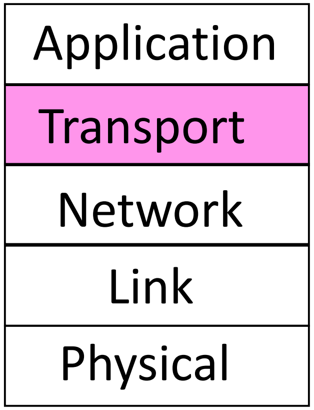
L11 – Transport Layer III

Lecturer: CUI Lin

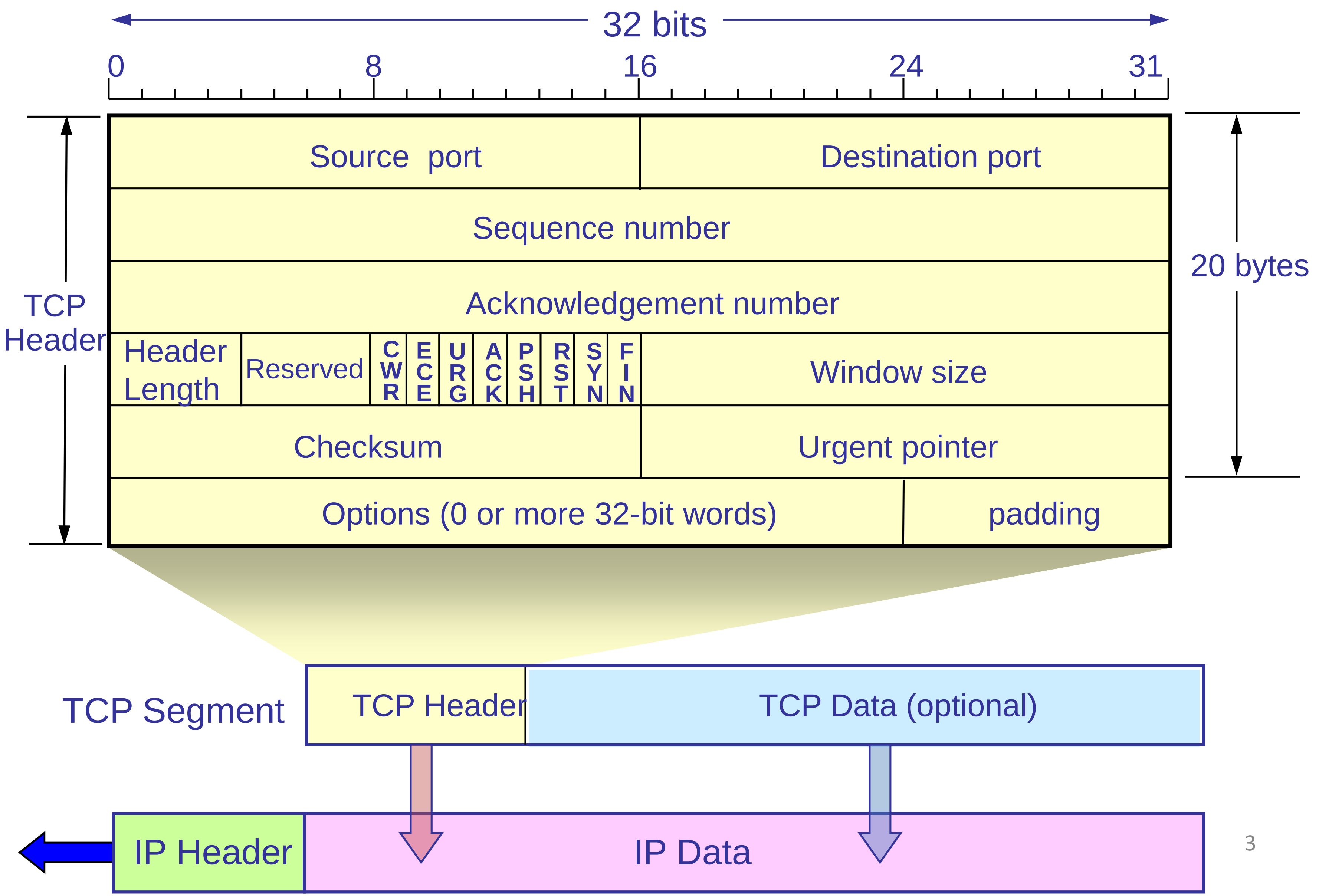
Department of Computer Science
Jinan University

The Transport Layer

- Responsible for delivering data across networks with the desired reliability or quality
  - Flow control
  - Congestion control



TCP Header


32 bits
0
8
16
24
31
TCP Header
20 bytes
Source port
Destination port
Sequence number
Acknowledgement number
Header Length
Reserved
CWR
ECE
URG
ACK
PSH
RST
SYN
FIN
Window size
Checksum
Urgent pointer
Options (0 or more 32-bit words)
padding
TCP Segment
TCP Header
TCP Data (optional)
IP Header
IP Data

TCP Topics

- UDP: User Datagram Protocol
- The TCP service model
- The TCP segment header
- TCP connection establishment
- TCP connection state modeling
- TCP sliding window
- TCP timer management
- TCP congestion control

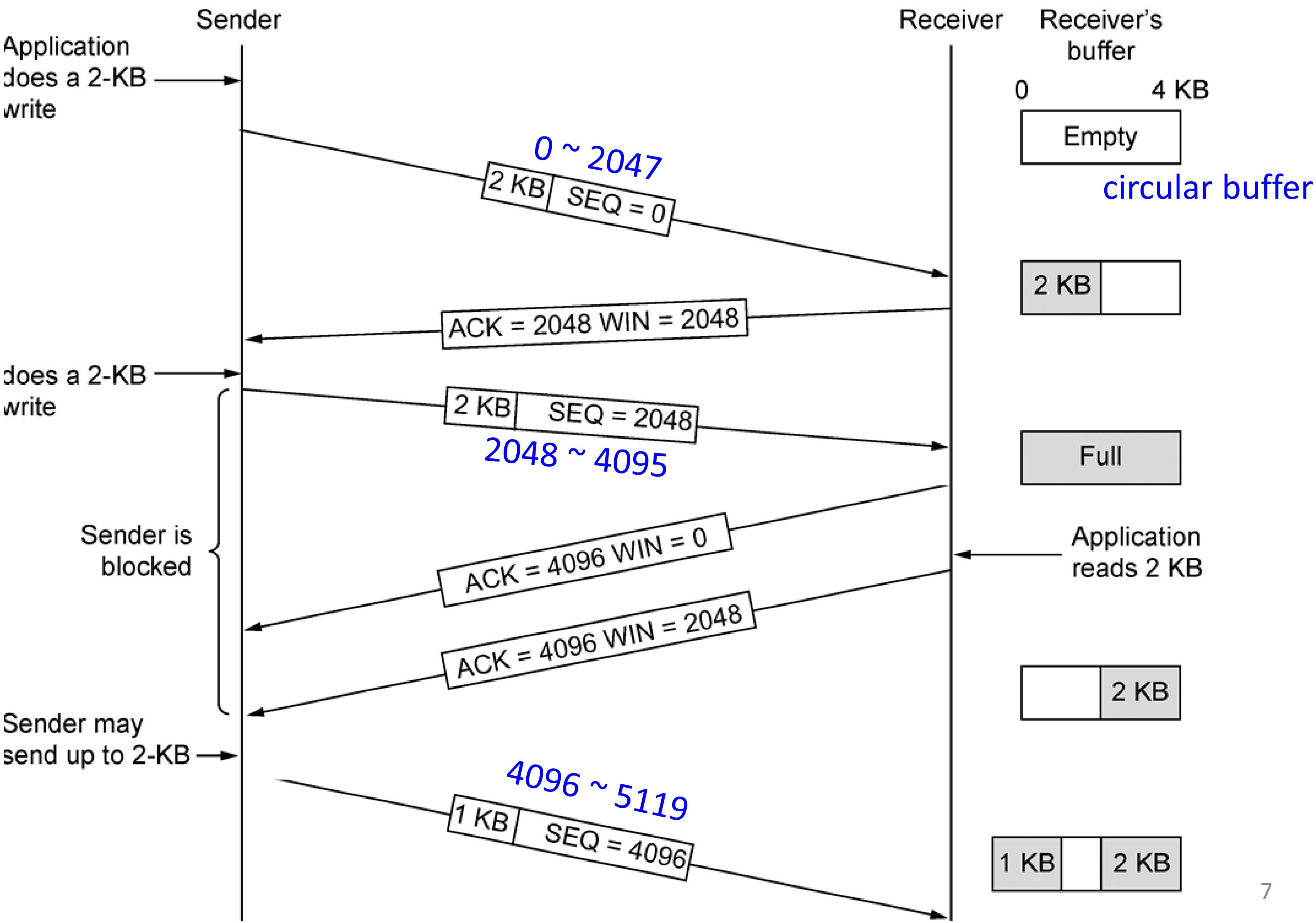
TCP Sliding Window Overview

- Roles of sliding window
  - Ensure reliable delivery of data
    - Unacknowledged data must be buffered by sender
  - Ensure in-order delivery of data
    - Out-of-order data must be buffered by receiver
  - Enforce end-to-end flow control
    - Too fast sender should be blocked

Window Management

- Unlike most data link protocols, in TCP, acknowledgements and permission to send additional data are completely decoupled
  - Window size is advertised to sender independently
  - E.g., a receiver can say: I have received *k* bytes, but I do not want any more now
- This decoupling gives additional flexibility
  - It actually decouples the acknowledgement of correct received segments and receiver buffer allocation.

TCP Window Management



Zero window

- When window is 0, normally, sender is not allowed to send any segments, except:
  - Urgent data: e.g., Ctrl+C command
  - Window probe packet (窗口探测包): let receiver re-announce the next byte expected (下一个期望接收的字节序号) and window size

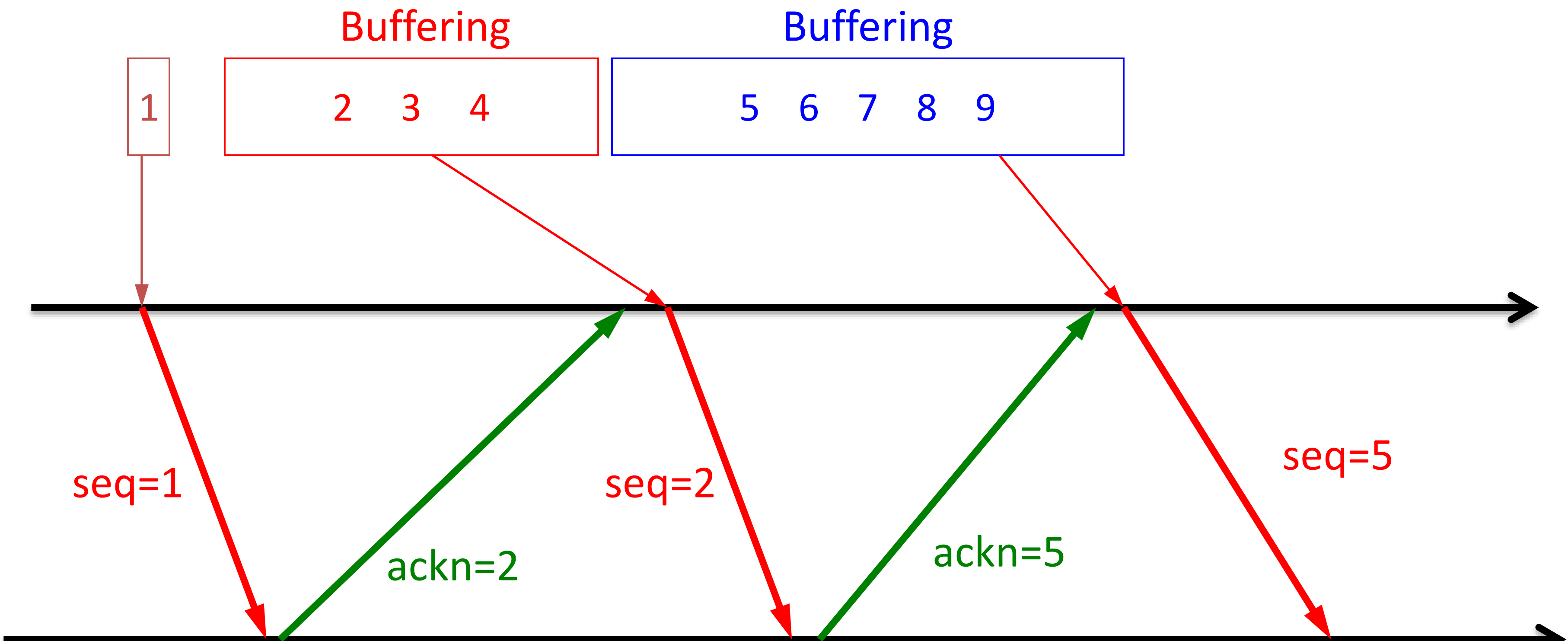
Delayed Acknowledgement

- Sending small segments degrades performance
- Delayed Acknowledgements:
  - delay small segment, including ACKs and window updates, for some amount of time and hope that they can be piggybacked on some data.
- How long should sender delay sending data?
  - too long: hurts interactive applications
  - too short: poor network utilization
  - strategy: timer-based (500 milliseconds)

Sender: Nagle's Algorithm

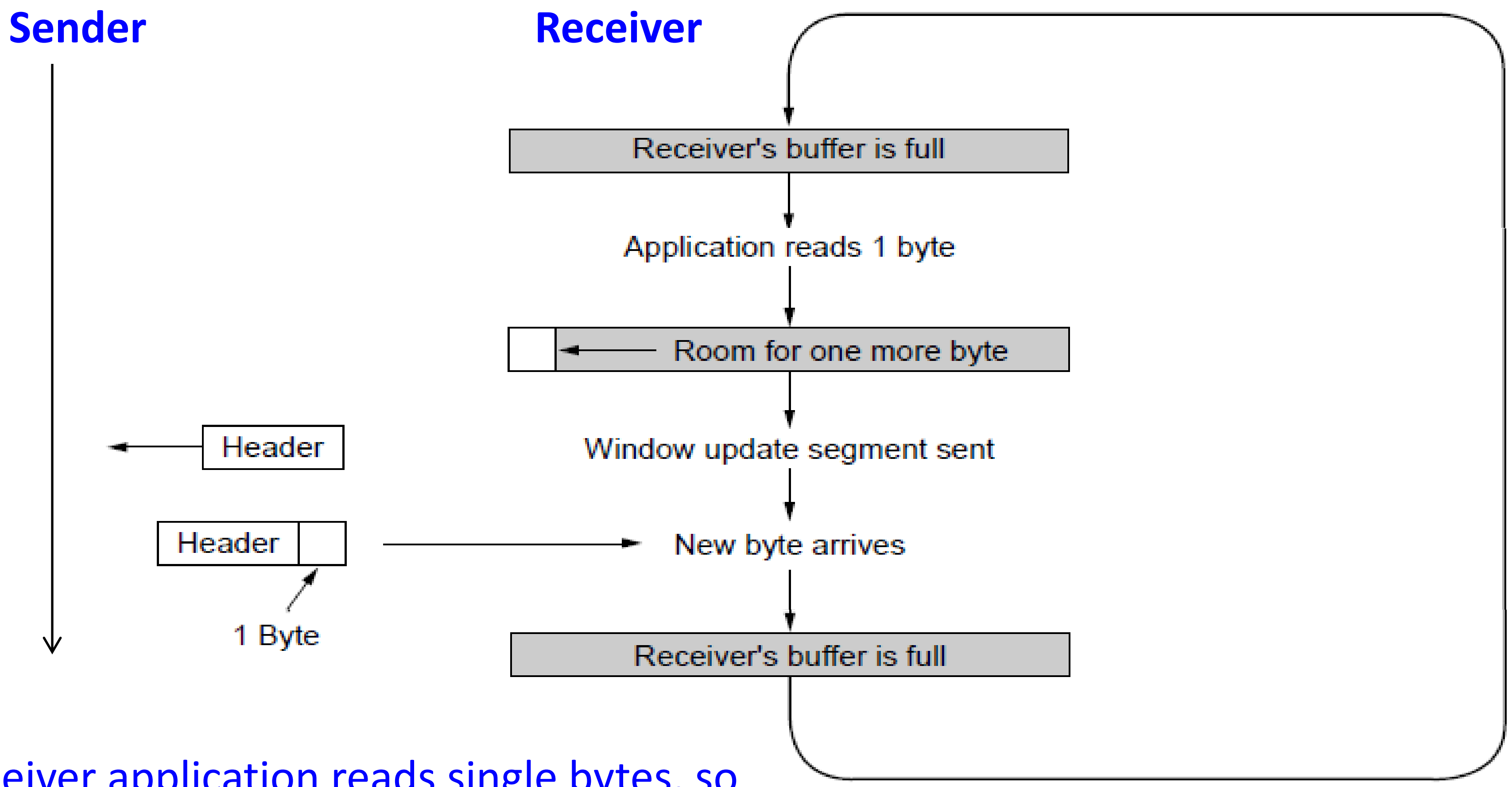
- An improved solution: self-clocking idea
  - Acknowledgment can be treated as a timer firing, triggering transmission of more data
  - Result: one segment per RTT (round-trip time)
- Nagle's Algorithm:
  - When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.
  - Then send all the buffered data in one TCP segment and start buffering again until they are all acknowledged

Nagle's Algorithm



Silly Window Syndrome

- Receiver application may read one byte each time:



Receiver application reads single bytes, so sender always sends one byte segments

Clark's Solution (Clark, 1982)

- Receiver avoids sending a window update for 1 byte. It waits until it has enough available space.
- Specifically, the receiver should not send a window update until it can handle the maximum segment size (MSS), or until its buffer is half empty

TCP Sliding Window

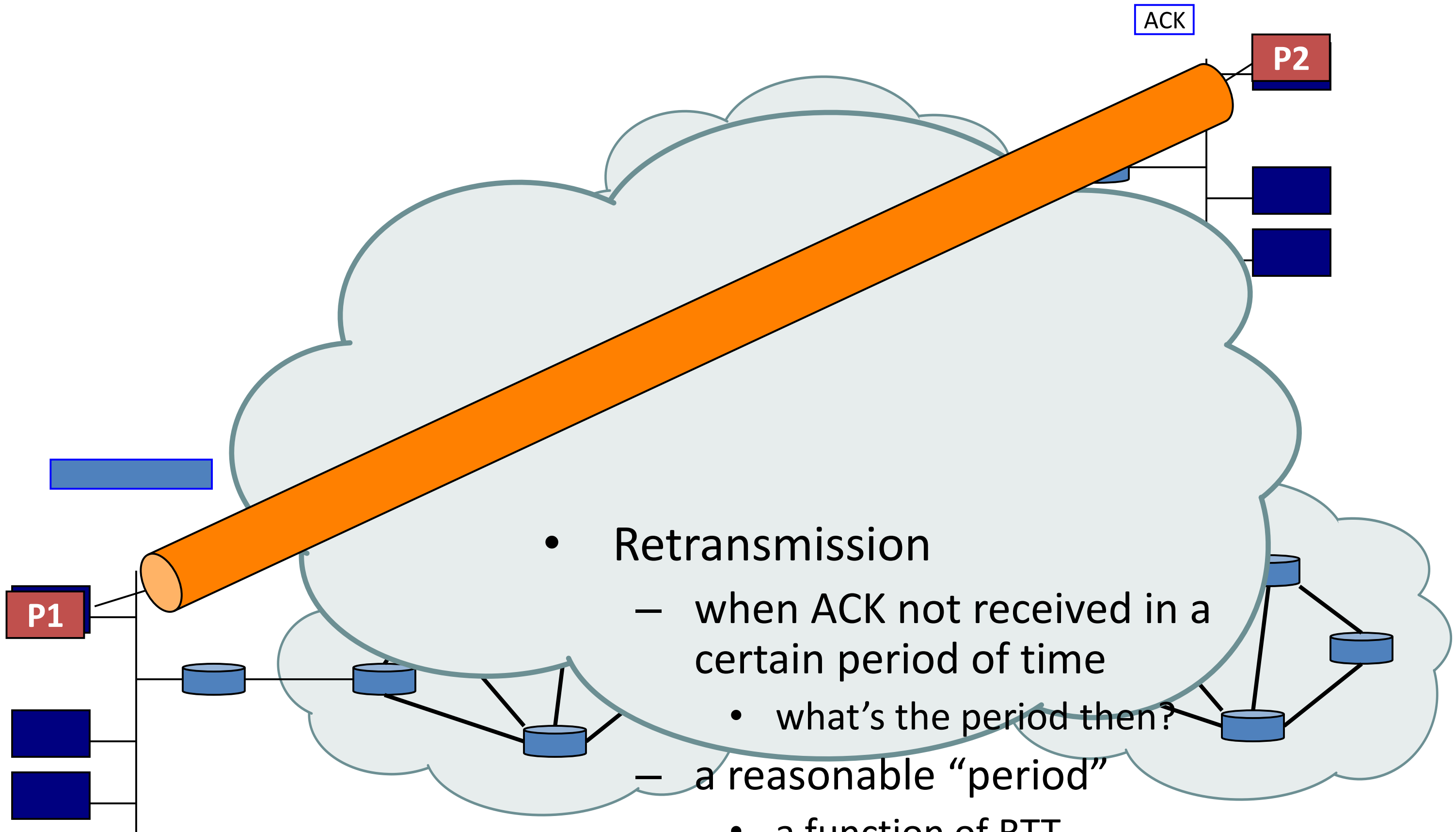
- Nagle's algorithm and Clark's solution are complementary:
  - They can work together
  - Goal is for sender not to send small segments and the receiver not to ask for them
- More improvements:
  - E.g., cumulative acknowledgement

Wireshark TCP Trace

TCP Topics

- UDP: User Datagram Protocol
- The TCP service model
- The TCP segment header
- TCP connection establishment
- TCP connection state modeling
- TCP sliding window
- TCP timer management
- TCP congestion control

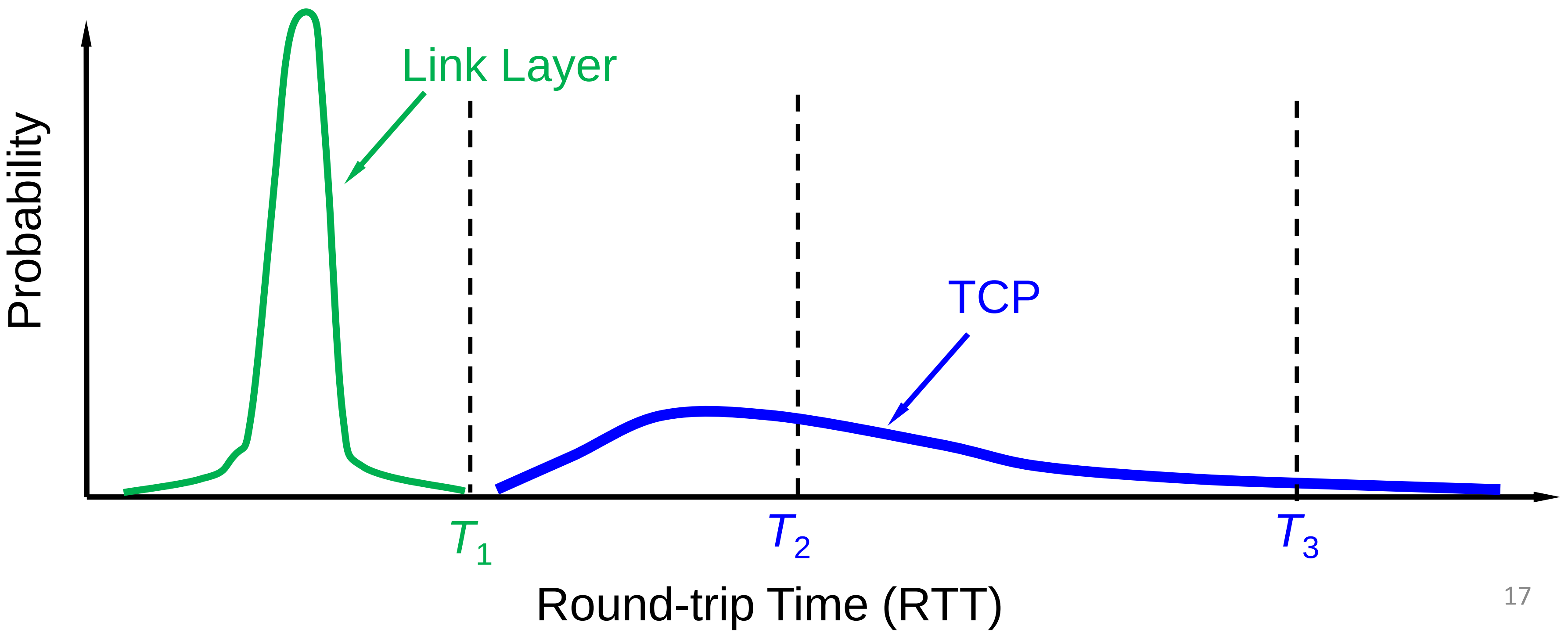
When TCP Segment Seems Lost



- Retransmission
  - when ACK not received in a certain period of time
    - what's the period then?
  - a reasonable "period"
    - a function of RTT
- Challenge:
  - how long the RTT is?
    - Largest range among host pairs
    - variation over time

Round-Trip Time

- Link layer: measured in microseconds, low variance, regular and highly predictable RTT
- TCP: measured in milliseconds, larger & Variable RTT



SRTT: Smoothed Round-Trip Time

- Each TCP connection maintains a variable *SRTT*
- Measure *SampleRTT* for each segment/ACK pair, and update *SRTT:*

$$SRTT_i = \alpha \cdot SRTT_{i-1} + (1 - \alpha) \cdot SampleRTT_i$$

- α is a smoothing factor ($0 \le \alpha < 1$), determining how quickly old values are forgotten
- RFC2988: $\alpha = 7/8$
- Still difficult to determine retransmitted timeout due to large variance of RTT

Round-Trip Time Variation

- Calculate both the mean of RTT and the variance of that mean
- *RTTVAR*: Round-Trip Time Variation

$$RTTVAR_i = \beta \cdot RTTVAR_{i-1} + (1 - \beta) \cdot |SRTT_i - SampleRTT_i|$$

- $0 \leq \beta < 1$, $\beta = 3/4$ in RFC 2988

RTO: Retransmission Timeout

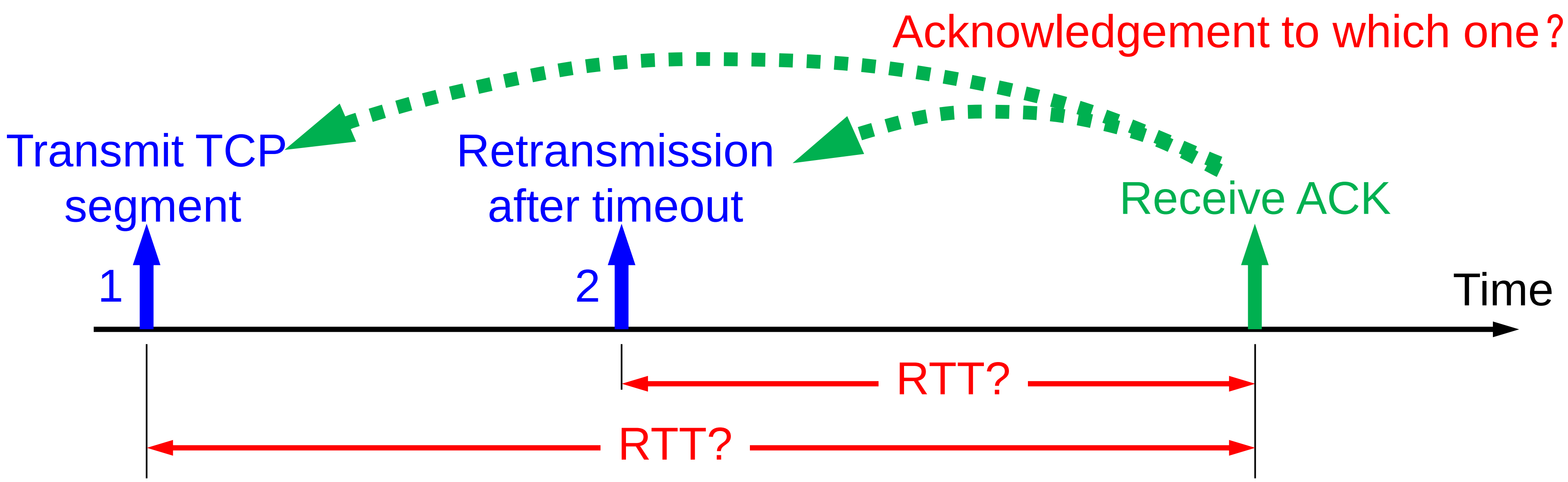
- RFC 2988 defines RTO:

$$\mathrm{RTO} = \mathit{SRTT} + 4 \times \mathit{RTTVAR}$$

- Factor 4 is somewhat arbitrary, but can be computed using *shift*
- Less than 1% packets come later than above RTO
- RTO has a minimum of 1 second, regardless of estimates

Flaw of ACK-based RTT Sampling

- Solution: Karn's Algorithm
  - Don't update estimates on any retransmitted segments
  - Double RTO after each retransmission until there is no retransmission



TCP Topics

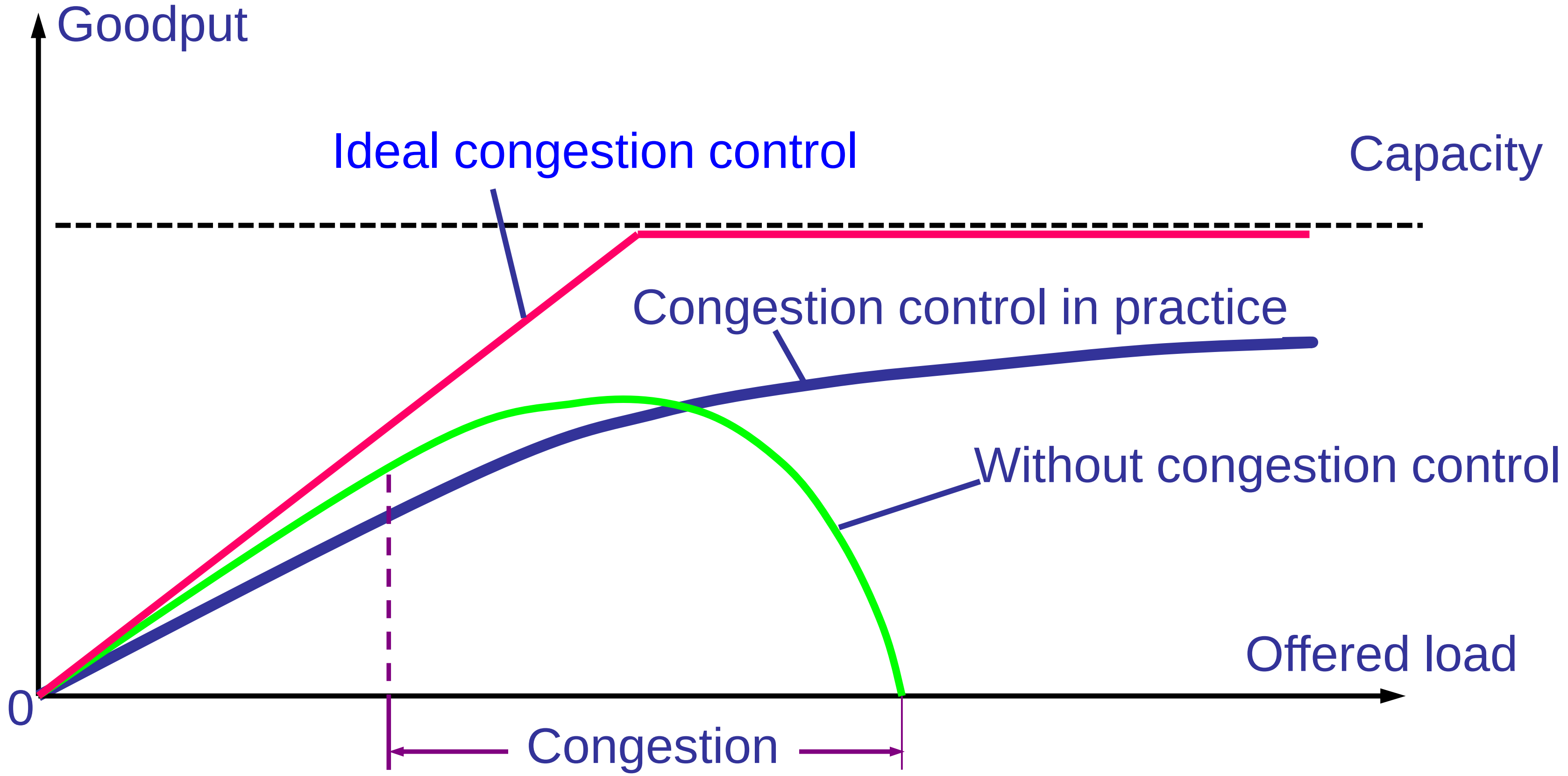
- UDP: User Datagram Protocol
- The TCP service model
- The TCP segment header
- TCP connection establishment
- TCP connection state modeling
- TCP sliding window
- TCP timer management
- TCP congestion control

Listen carefully

TCP Congestion Control

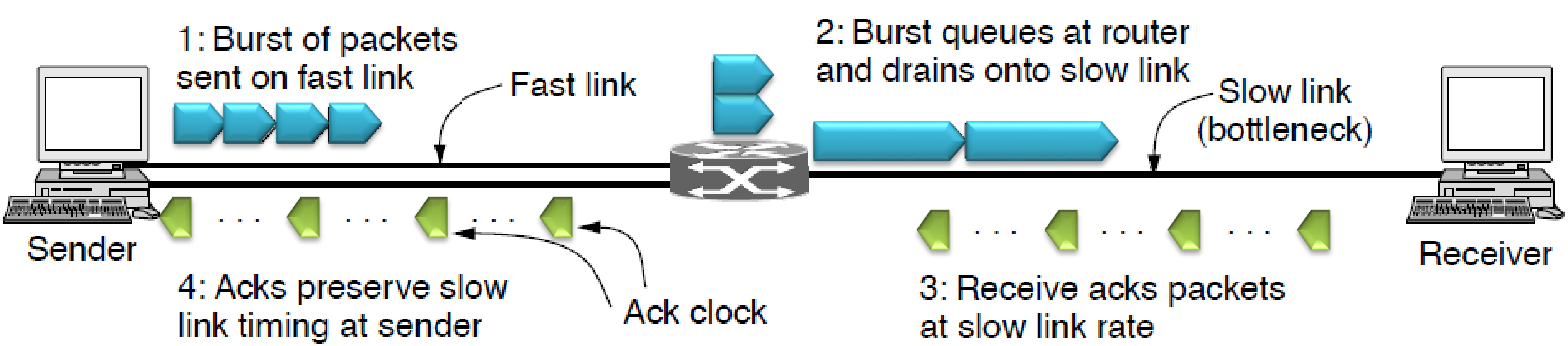
- Congestion (擁塞): load offered to network is more than it can handle
- Degrading network performance
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
- Solution: let sender stop or sending slowly

TCP Congestion Control



ACK Clock

- ACKs return to the sender at about the rate that packets can be sent over the slowest link in the path.
- If sender injects new packets into the network at this rate (i.e., ACK clock),
  - Packets will be sent as fast as the slow link permits.
  - They will not queue up and congest any router along the path.



Congestion Window

- Two related windows:
  - Receiver's window (*rwnd*): flow control window, determined by receiver's buffer
  - Congestion window (*cwnd*): determined by network capacity
- Both window are tracked in parallel
- The number of bytes that can be sent, i.e., sender's window, is the minimum of the two windows, i.e., *min(rwnd, cwnd)*.

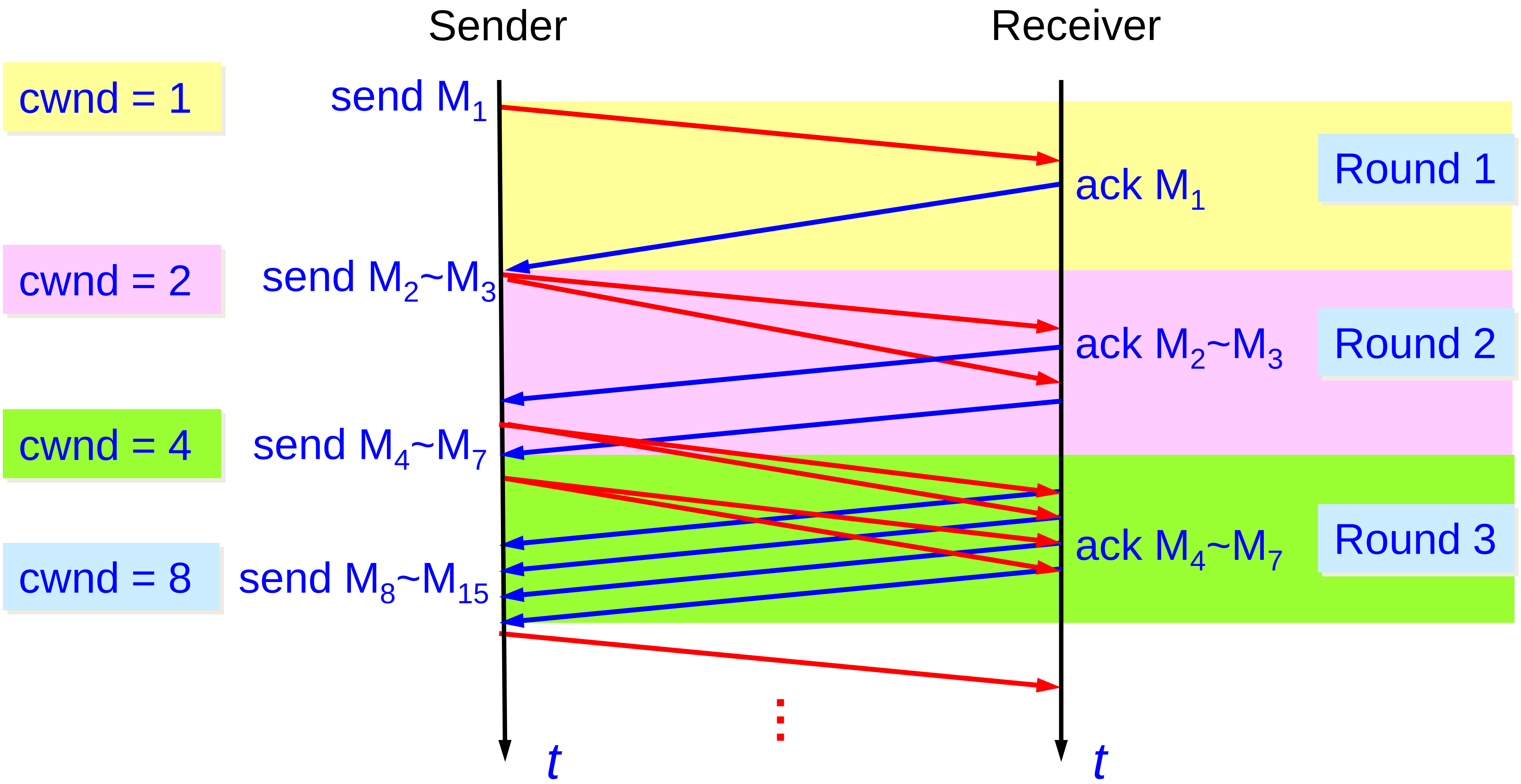
The unit of window is byte.

Sometimes we may use number of segments for simplicity.

Slow Start (慢开始)

- Initially, *cwnd* = 1, i.e., the size of one MSS
- Each time sender receives an ACK, it increases *cwnd* by one MSS
- Until a timeout occurs (congestion), or *cwnd* reaches *slow start threshold* (ssthresh)
- Slow start window threshold is initialized to be size of flow control window
- "Slow":
  - It is slower than sending entire flow control window at once

Slow Start

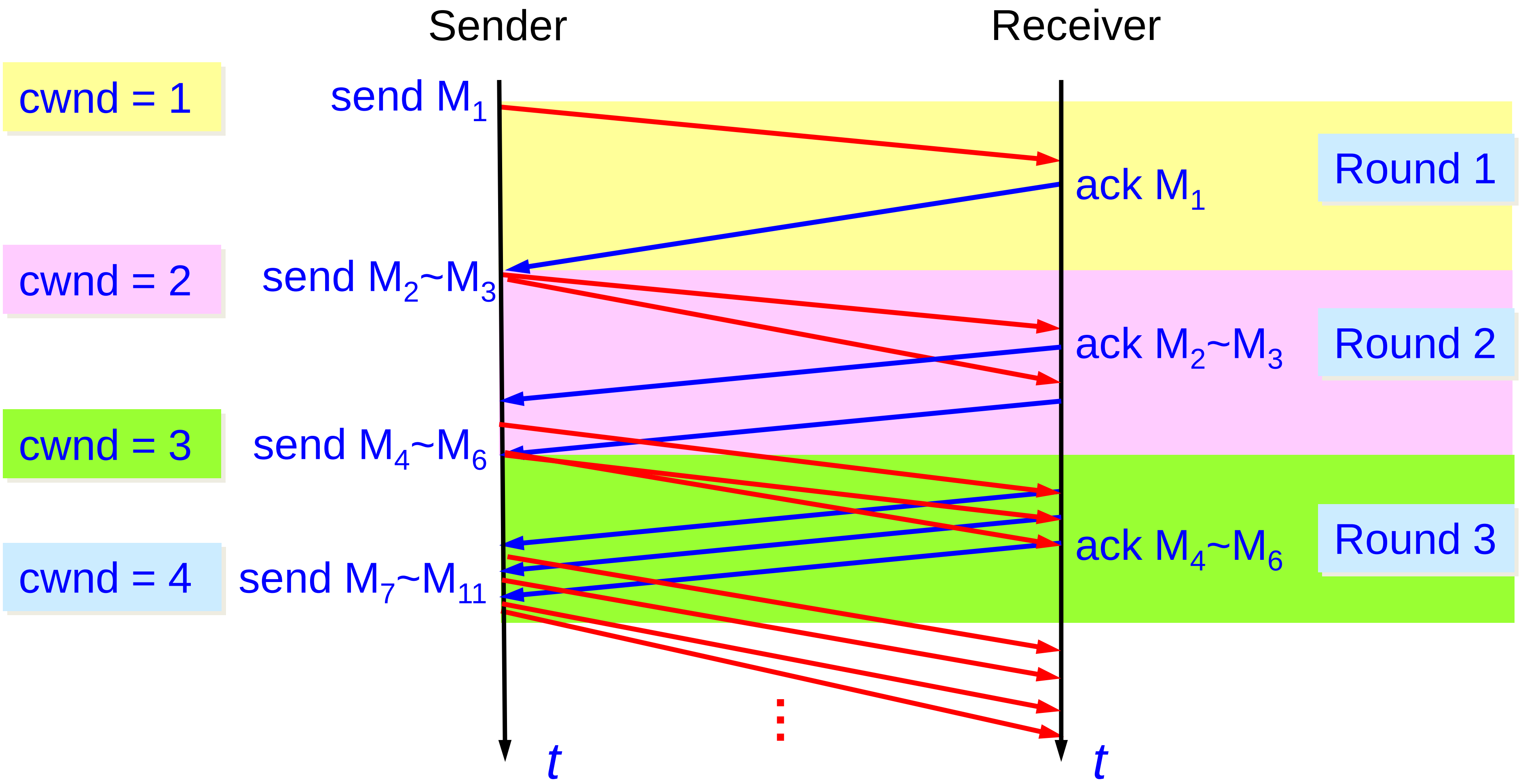


Increase cwnd by 1 MSS for each ACK
Each transmission round is a RTT. *cwnd* is doubled each round

Congestion Avoidance(拥塞避免)

- When *cwnd>ssthresh*, TCP enters the state of congestion avoidance
- TCP slows down the increase of *cwnd*, switching to additive increase (加法增加):
  - *cwnd* is increased by one MSS every RTT (or round), rather than doubled
- Idea is to let TCP connection spend a lot of time with its congestion window close to the optimum value

Additive Increase



Increase *cwnd* by 1 MSS each RTT

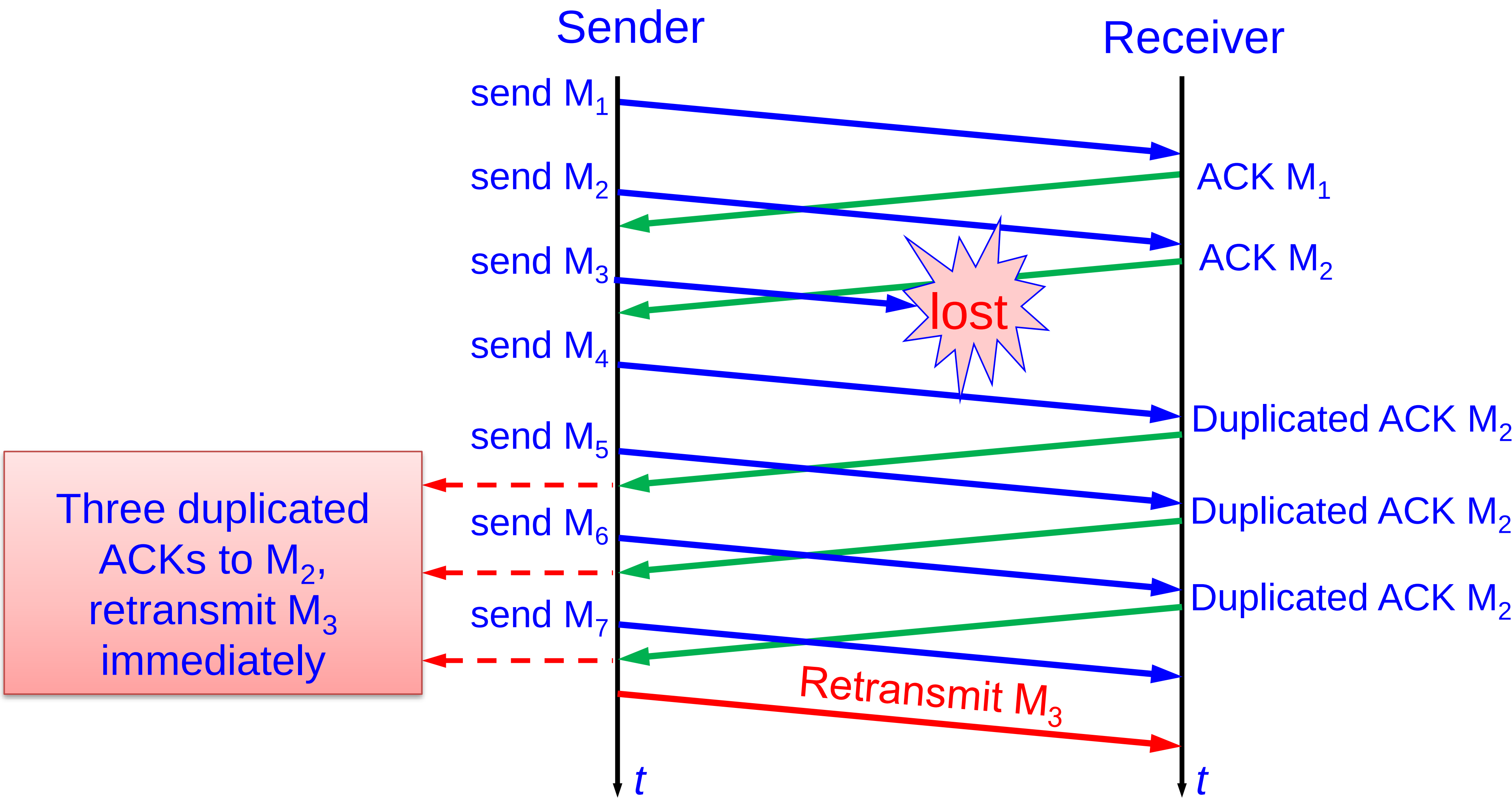
Congestion

- When packet loss is detected:
  - by a timeout
  - Up to 3 duplicated ACKs: fast retransmission
- *ssthresh* is set to be half of *cwnd*, then set *cwnd=1*

$$ssthresh = max\ (cwnd/2,\ 2)$$

- Idea:
  - Current window is too large and cause congestion
  - Half of the window, which was used successfully earlier, is probably a better estimate for *cwnd*

Fast Retransmission(快速重传)

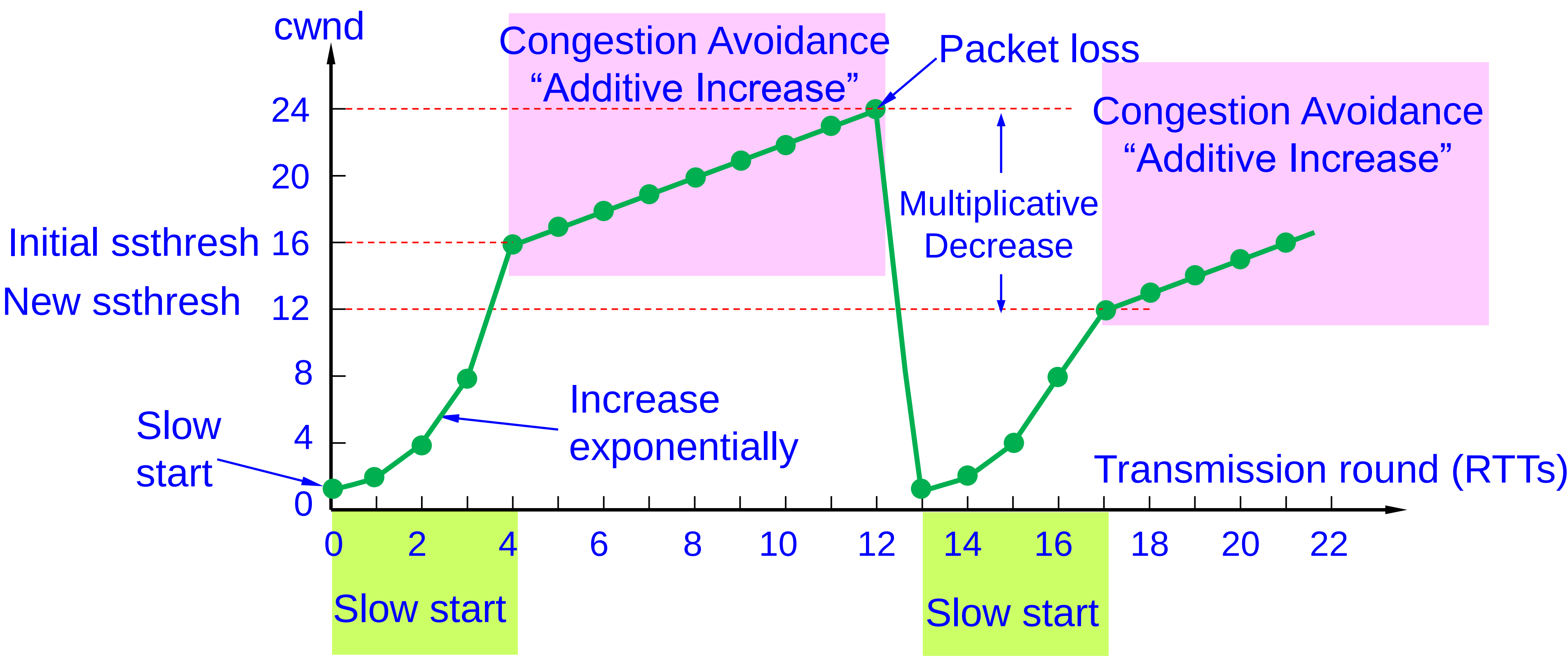


Retransmission starts early before timeout fires.

TCP Tahoe

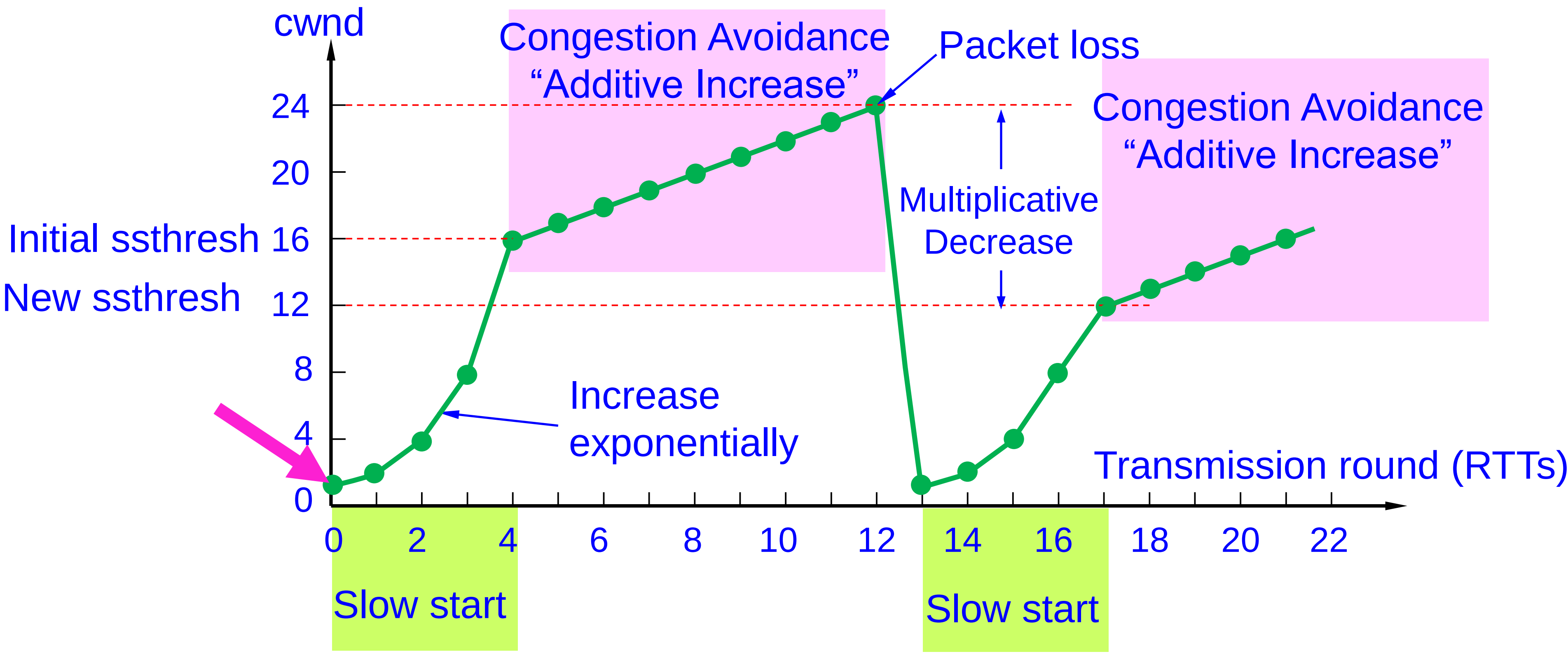
- TCP Tahoe: all schemes so far
  - Slow start
  - Congestion avoidance: Additive increase
  - Fast retransmission
  - Reset to slow start upon packet loss or up to 3 duplicated ACKs: *cwnd=1*

TCP Tahoe Example



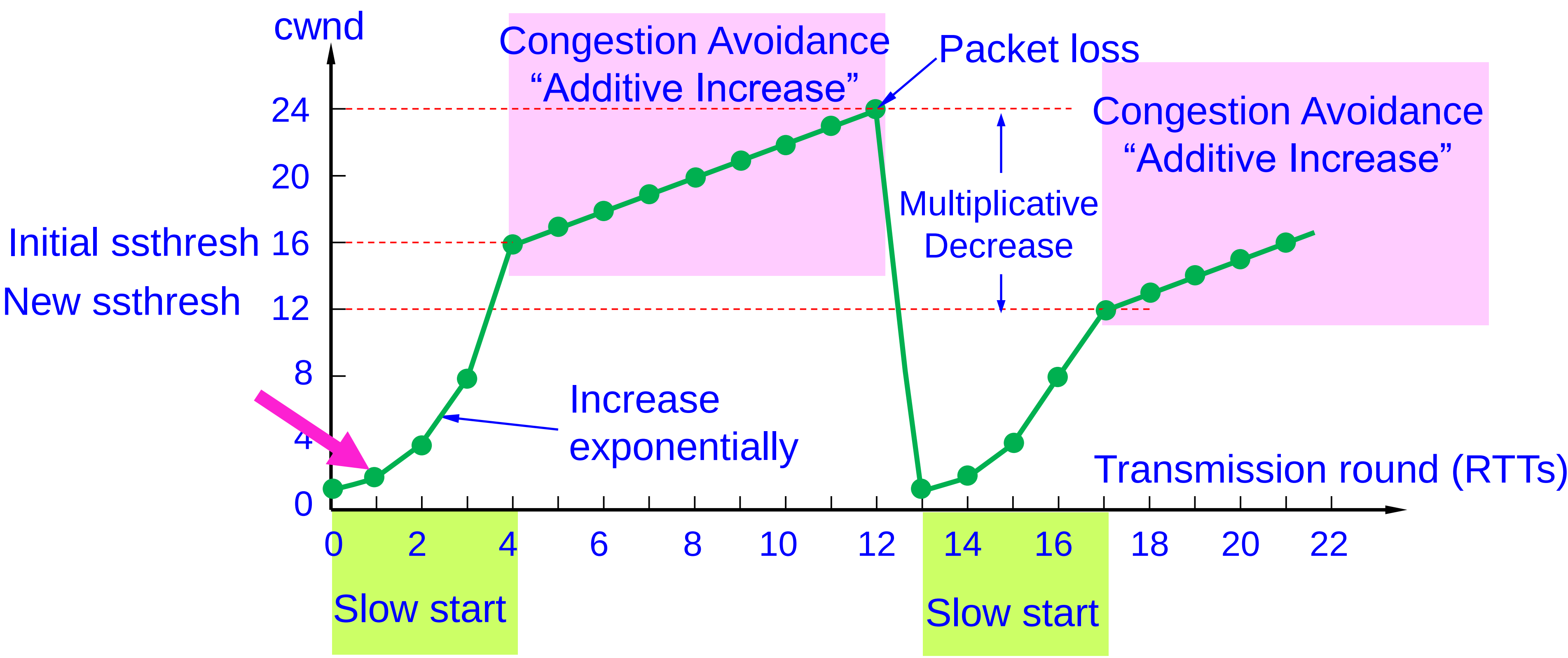
In this example, instead of bytes, we use number of segments for simplicity. Initially, *cwnd=1*, i.e., size of 1 MSS. *ssthresh* = 16 MSS. Sender’s window is min(*rwnd*, *cwnd*). We assume receiver’s window is large enough. So sender’s window is *cwnd*.

TCP Tahoe Example



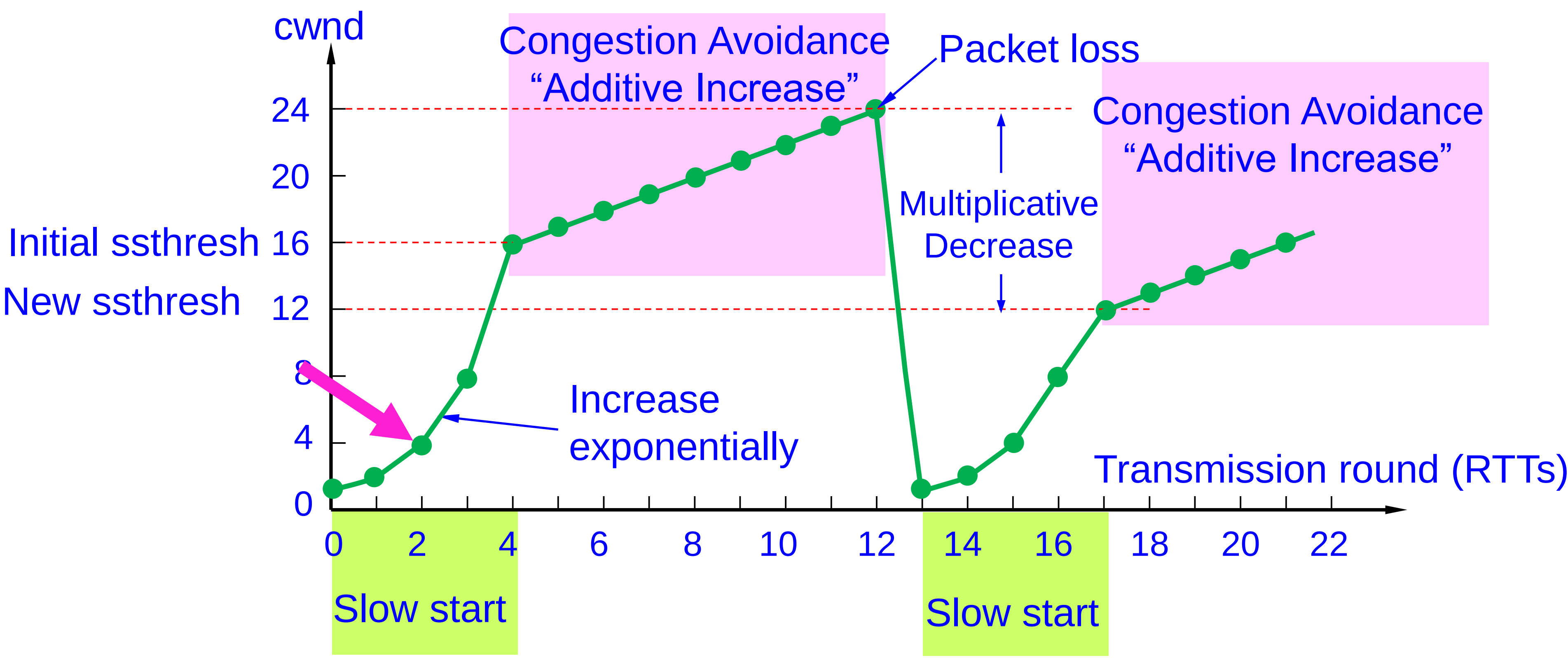
During slow start, *cwnd*=1, sending one segment.

TCP Tahoe Example



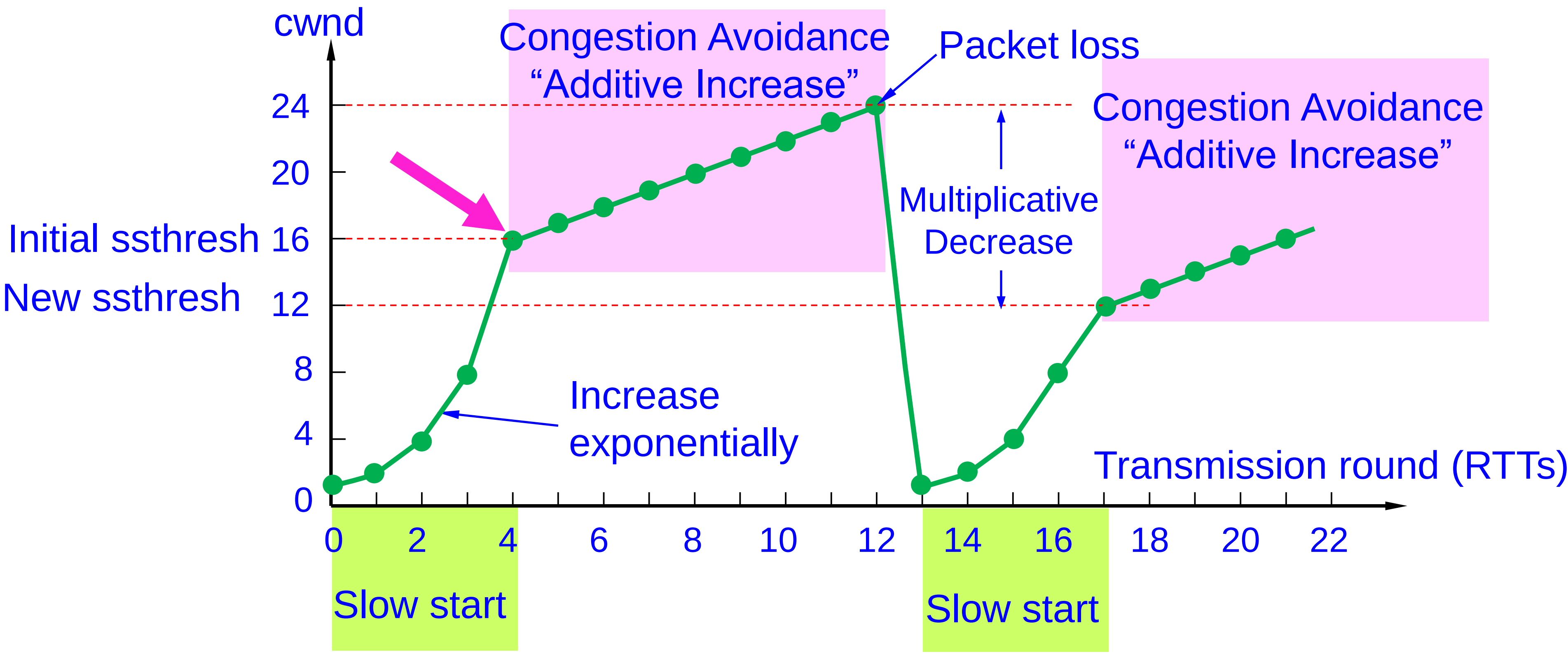
After one RTT, sender receives the first ACK.
cwnd is increased by 1 upon receiving each ACK during slow start.
Now, *cwnd*=2. Sender sends out two segments.

TCP Tahoe Example



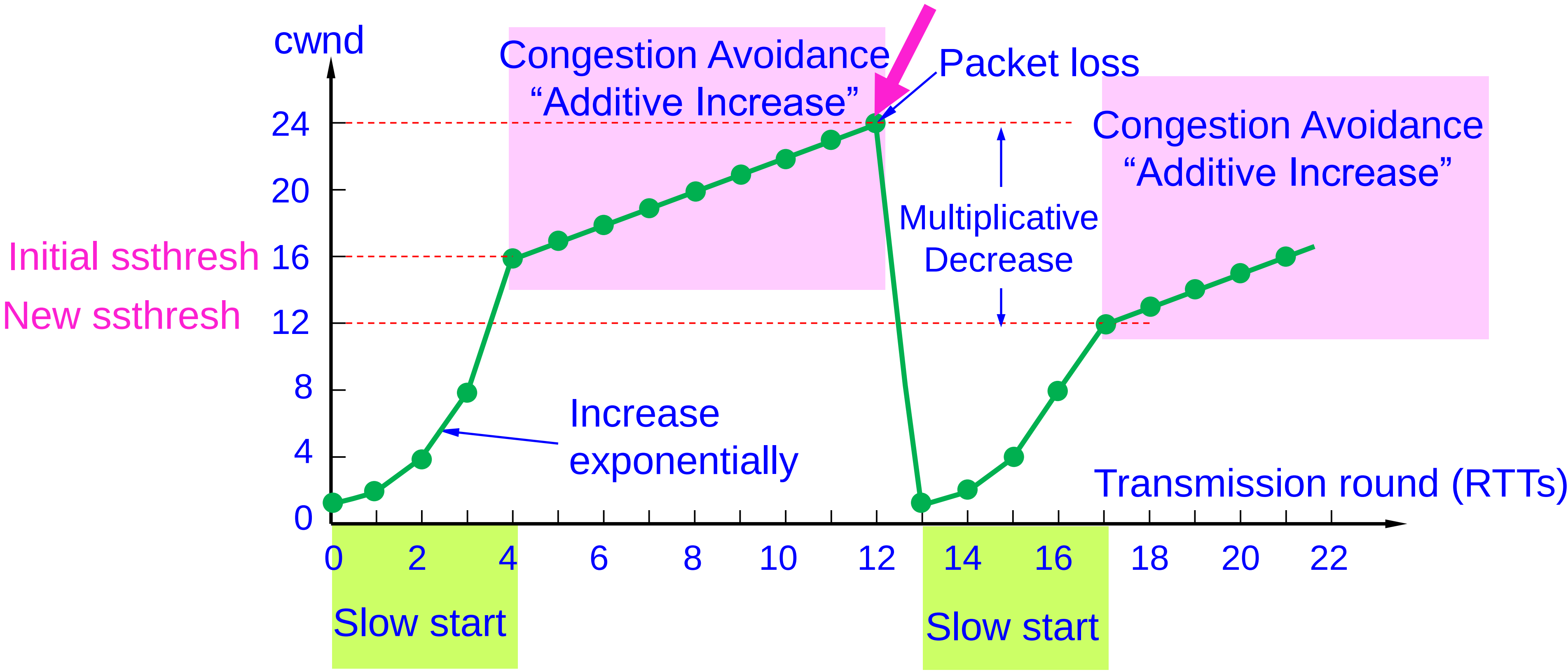
Then sender receives two ACKs.
Now, *cwnd*=4. Sender sends out four segments.
Congestion window is increased exponentially.

TCP Tahoe Example



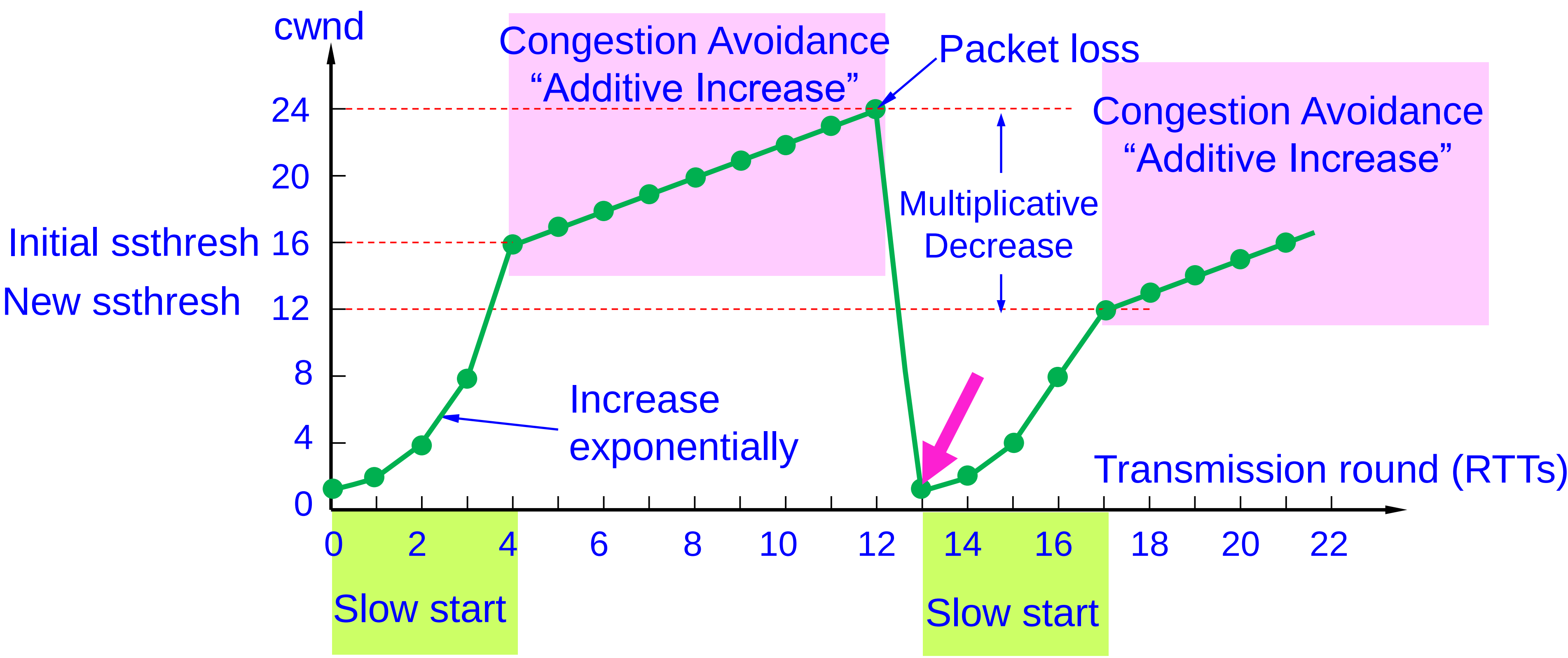
When *cwnd* reaches the slow start threshold (*ssthresh*=16), it enters state of congestion avoidance, starting “additive increase”.
cwnd is increased by 1 every RTT.

TCP Tahoe Example



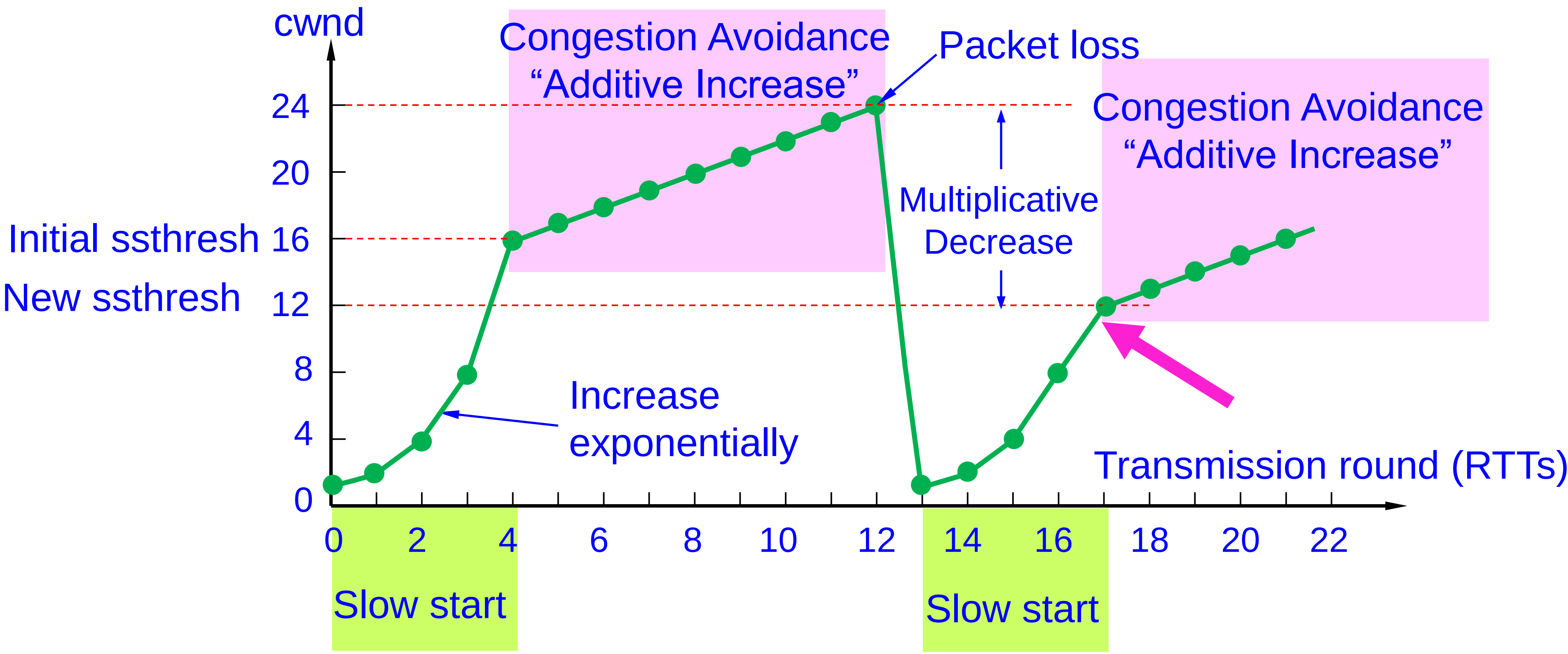
Assume at *cwnd*=24, packet loss is detected. Network congestion occurs.

TCP Tahoe Example



ssthresh is halved to 12.
Slow start is restarted by setting *cwnd*=1.

TCP Tahoe Example



When cwnd reaches the new *ssthresh*=12. It starts additive increase.
The whole process will be repeated.

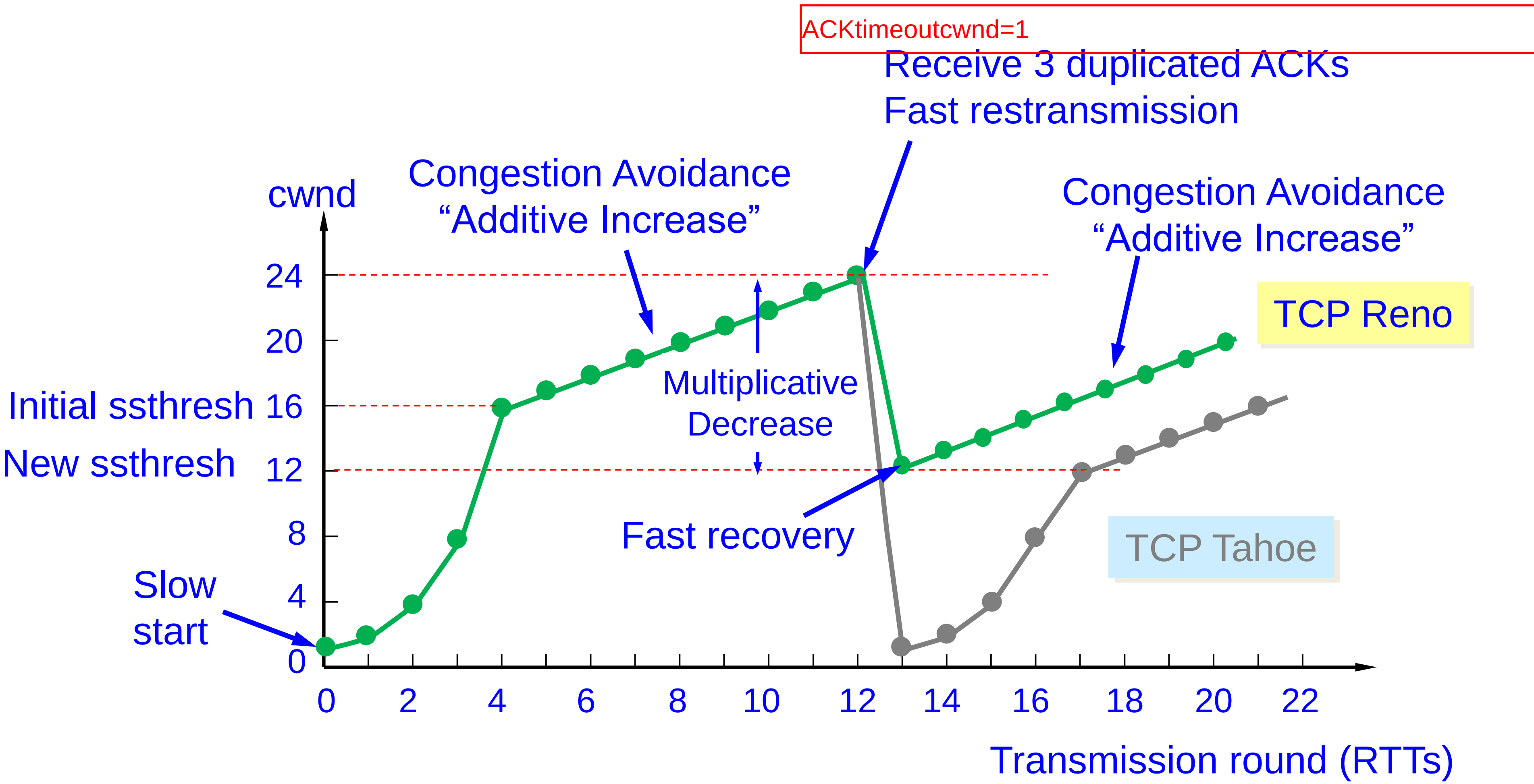
Fast Recovery (快速恢复)

- Observation:
  - Upon receiving 3 duplicated ACKs, *cwnd* is too large, but the “ACK clock” is still working
  - Each time a duplicated ACK arrives, it is likely another packet has left the network
- Improvement: Fast recovery
  - Upon receiving 3 duplicate ACKs, retransmit the presumed lost segment (“fast retransmission”)
  - Halve the *cwnd*, rather than set *cwnd*=1

TCP Reno

- TCP Tahoe plus fast recovery
- Instead of repeat slow starts, congestion window follows a *sawtooth* pattern (锯齿状):
  - Additive Increase: increase cwnd by 1 MSS every RTT
  - Multiplicative Decrease: Halve cwnd in one RTT
- This is also called AIMD rule (加法增加、乘法减少策略)

TCP Reno Example



Note: The slow start algorithm is executed when a new TCP connection is created or when a loss has been detected due to a retransmission timeout (RTO).

TCP Congestion Control

- TCP self regulating:
  - Sender enforced congestion control
  - Receiver enforced flow control
- Three windows:
  - *rwnd* = receiver window size (flow control)
  - *cwnd* = congestion window size (congestion control)
  - Effective sender window size: *swnd = min(rwnd, cwnd)*

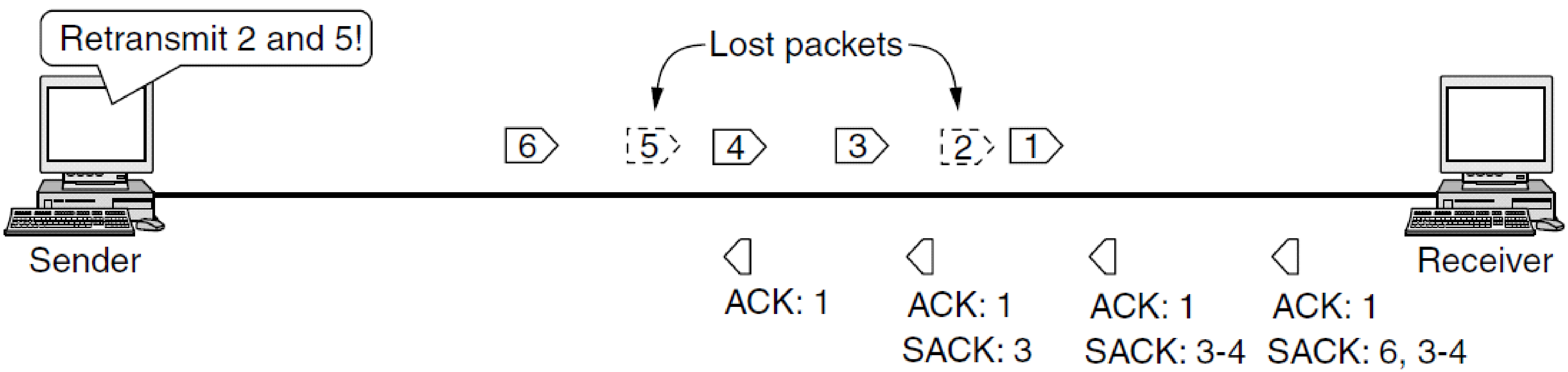
Question: what’s the difference the cases: *rwnd<cwnd*, and *cwnd<rwnd*? What’s the bottleneck?

TCP SACK (选择重传)

- SACK (Selective ACKs) extend ACKs with a vector to describe received segments and hence losses
  - Sender can more directly decide what packets to retransmit
  - Can list up to three ranges of bytes that have been received
- Negotiate when setting up connection:
  - Using SACK permitted TCP option

TCP SACK Example

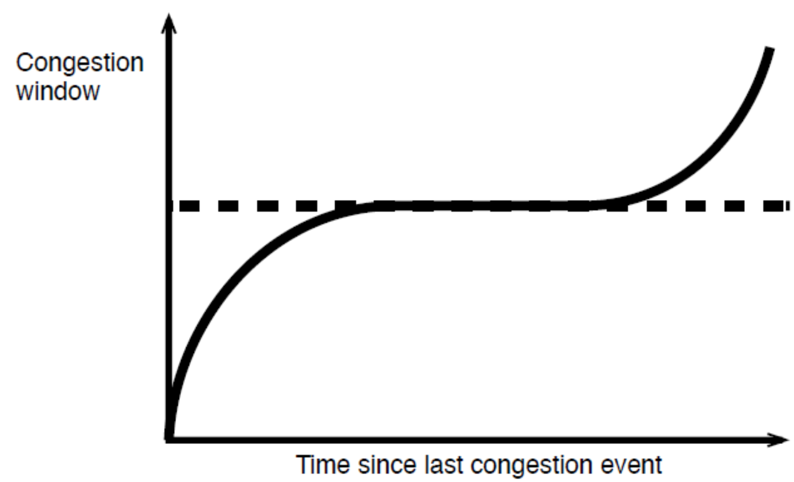
- SACK allows more accurate retransmissions /recovery



No way for us to know that 2 and 5 were lost with only ACKs

TCP Congestion Control of Today's Internet

- There are many variations of Congestion Control algorithms
- CUBIC
  - CUBIC adjusts its congestion window as a function of time
  - Congestion window increases according to a cubic function



Evolution of TCP CUBIC Congestion Window.

TCP Congestion Control of Today's Internet

Don't need to know the detail but name.

- Default congestion control algorithm
  - Linux: CUBIC (since 2006, kernel 2.6.19)
  - Windows: Compound TCP (CTCP), CUBIC (since 2017, Windows 10.1709)
  - MacOS: CUBIC (since 2014, OS X Yosemite)
- BBR
  - BBR: Bottleneck Bandwidth and Round-trip propagation time
  - Developed by Google in 2016
  - When implemented within YouTube, BBR yielded an average of 4% higher network throughput and up to 14% in some countries
  - BBR is also available for Linux >4.9

QUIC: Quick UDP Internet Connection

- QUIC is a transport protocol that runs on top of UDP
- Main features:
  - Runs in application layer
  - Flow control, congestion control, error control
  - Supports multiple data streams
  - Security (authentication, encryption) of headers and payload
  - Fast connection setup
- Some facts:
  - Proposed by Google.
  - HTTP/3 adopts QUIC
  - Supported by many servers and browsers (Chrome, Firefox, Safri)
  - As of 2023, over 7% of all Internet websites use QUIC

We will go back QUIC in lecture of Application layer.

Thank you!

Q & A