NANYANG TECHNOLOGICAL UNIVERSITY

CE3005: Computer Networks

Module 2-4:
Transport Layer – UDP and TCP

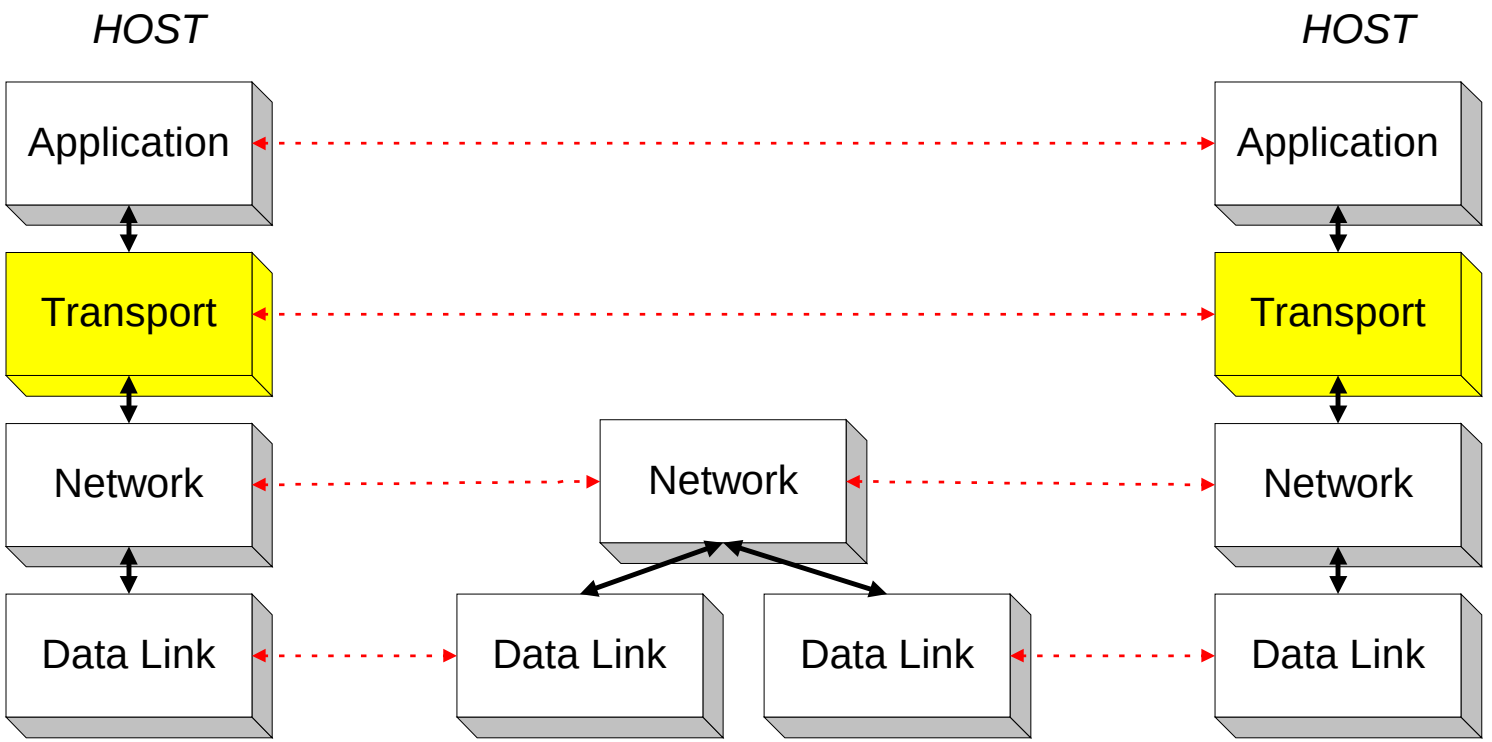
Semester 1 2016-2017

School of Computer Engineering

Contents

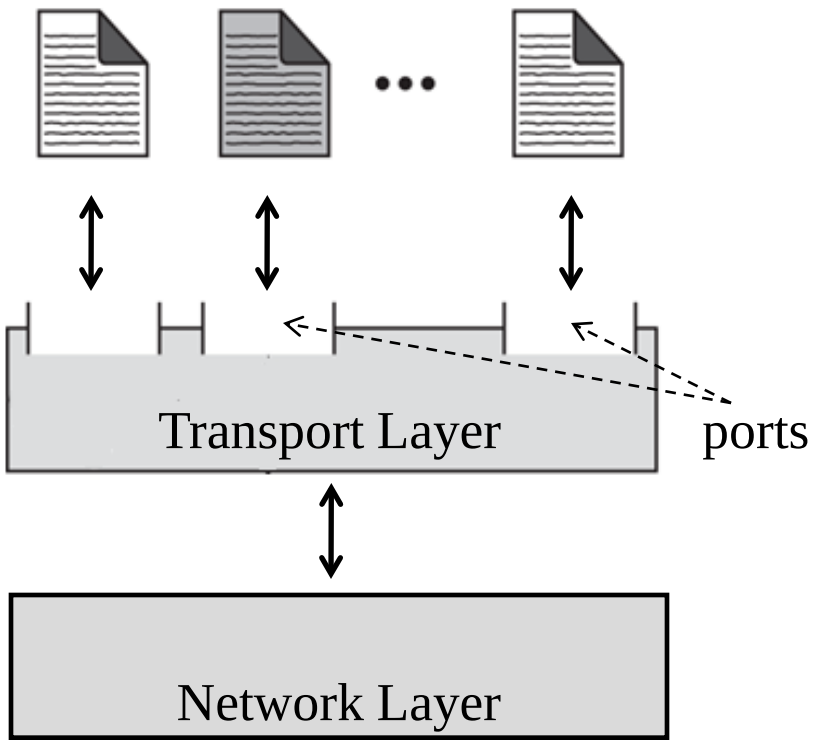
Transport Layer

- **Transport layer provides end-to-end service for transferring data between processes (process-to-process communication).**
- **Only implemented at the end hosts.**



Transport Layer - Ports

A single transport layer is used to support multiple application processes through the use of ports.



Hence, transport layer is also said to perform multiplexing/ de-multiplexing:

- multiplexing: gathering data from multiple processes and passing it to a single network layer
- de-multiplexing: delivering of data from single network layer to different processes correctly

Transport Protocols in the Internet

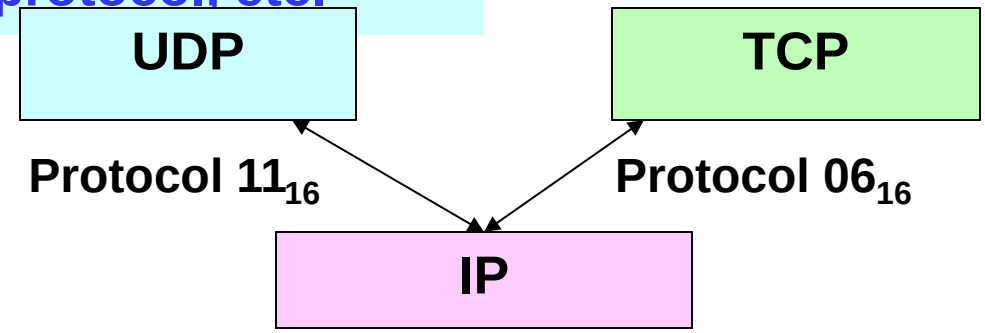
The Internet supports 2 transport protocols:

UDP - User Datagram Protocol

- unreliable, connectionless
- datagram oriented
- simple
- example applications:
  - routing (RIP), domain name service (DNS), DHCP, quote of the day service, real-time protocol, etc.

TCP - Transmission Control Protocol

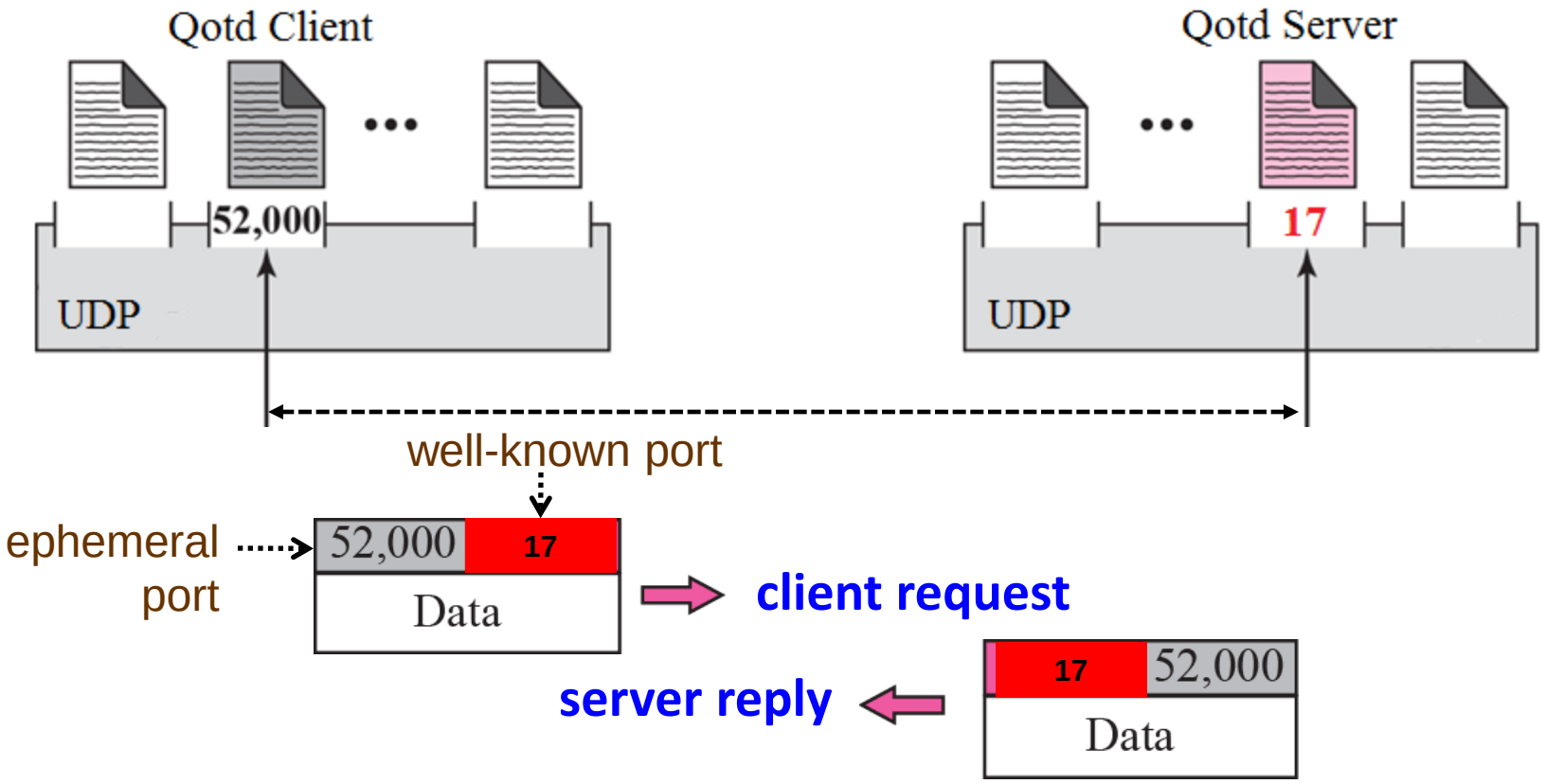
- reliable, connection-oriented
- stream oriented
- complex
- example applications:
  - web (http), email (smtp), file transfer (ftp), quote of the day, etc.



UDP – Datagram Service

Application layer is aware that UDP sends each message as a datagram.

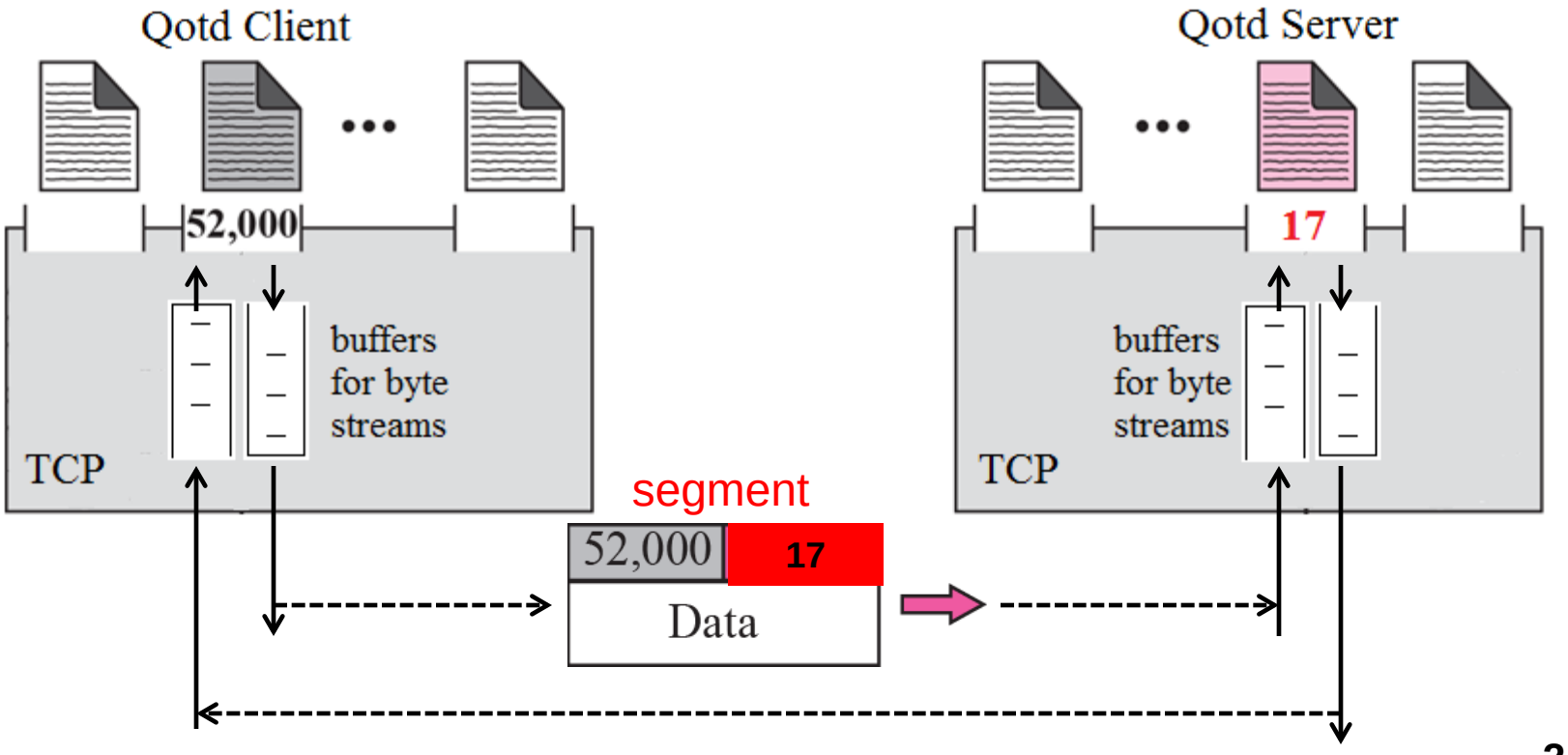
e.g. in Java, `request = new DatagramPacket();`



TCP – Byte Stream Service

Application layer views TCP as a channel for sending stream of bytes, and is NOT aware that bytes are sent in blocks called segments.

e.g. in Java, `outStream = socket.getOutputStream();`
`request = outStream.write(message);`



User Datagram Protocol (UDP)

Q: If UDP provides non- reliable communications, then why UDP?

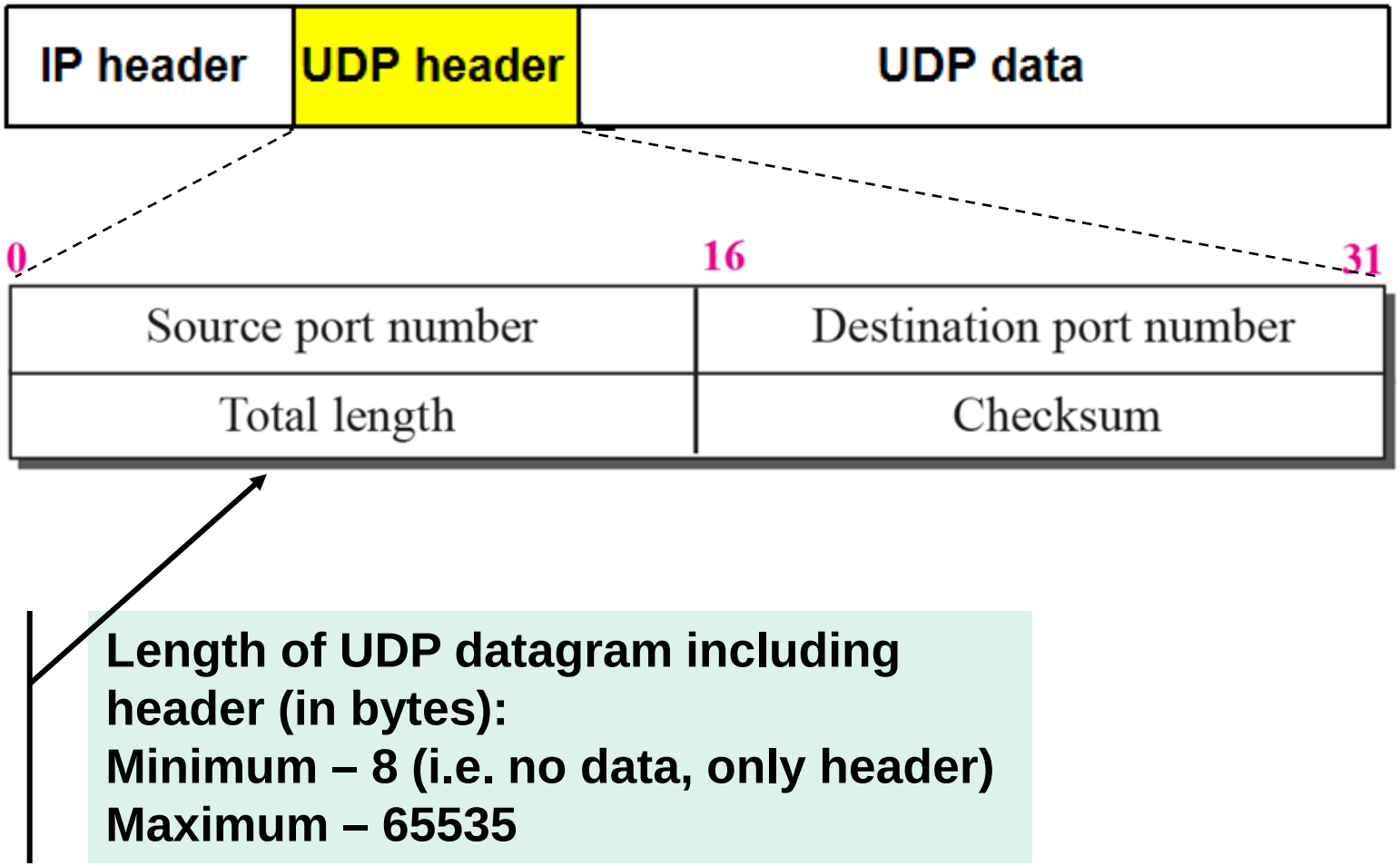
A: Some applications do not need reliable communications. For example:

- **Broadcasting, advertising messages to users.**
- **Sending live video streams over the Internet (loss tolerant, rate sensitive).**

Q: So, what does UDP do?

A: Provide process-to-process communication service for applications to use.

UDP Header Format

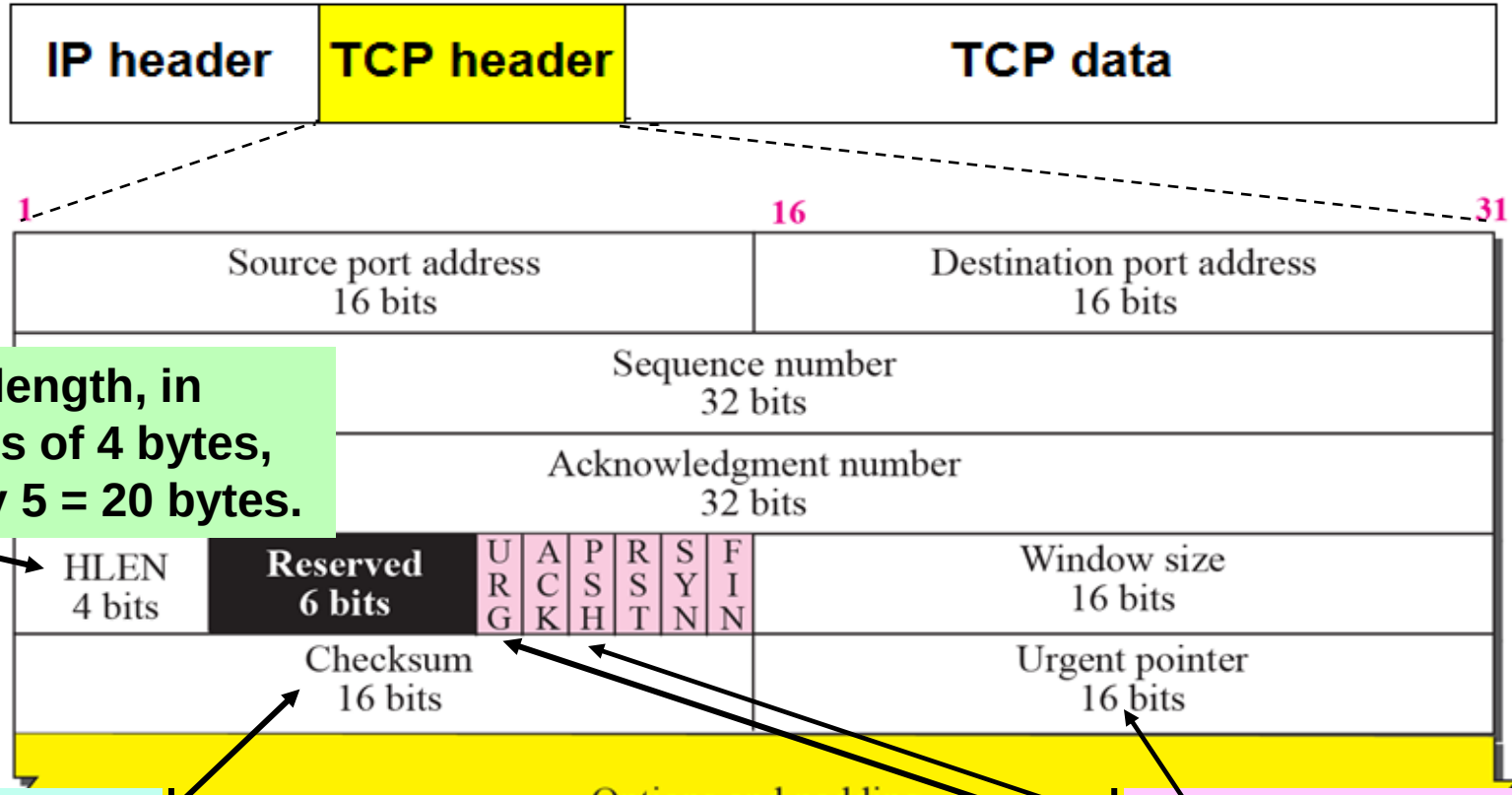


Length of UDP datagram including header (in bytes):
Minimum – 8 (i.e. no data, only header)
Maximum – 65535

Transmission Control Protocol

- **To support applications requiring reliable communications, TCP adds reliability over unreliable IP.**
- **Essentially, TCP features:**
  - **Connection Management: A connection must be setup before data exchange can be performed.**
  - **Flow Control: Sender will not overwhelm receiver.**
  - **Error Control: Receiver detects errors, sender retransmits error packets.**
  - **Congestion Control: During transmission, sender detects network usage(congestion) and adjust transmission rate.**

TCP Header Format



Header length, in multiples of 4 bytes, typically 5 = 20 bytes.

For error detection, ignore.

Typically not used, ignore.

For urgent data, typically not used - ignore.

TCP Header Format

- Port addresses

Port address	Type	Description of use
0-1023	Well Known port	Used by system processes to provide network services
1024-49151	Registered port	Assigned by IANA upon request by entities. Can also be use by user.
49152-65535	Ephemeral port	Dynamic or private ports that cannot be registered by IANA

TCP Header Format

- Sequence Number (SN):
  - Each TCP connection will start with a different SN called Initial Sequence Number (ISN)
  - The position of each data byte in the byte stream is labelled from ISN+1, and cycle back to 0 once reaching 2^{32} -1;
    i.e. 1st byte (ISN+1) mod 2^{32}, 2nd byte (ISN+2) mod 2^{32}, …
  - SN indicates the position of the 1st byte in each segment
- Acknowledgement Number (AN):
  - AN of the next data byte expected from sender
  - Also imply all bytes up to AN-1 have been received correctly
- Window Size (W):
  - Indicate the number of bytes (also called credits) counting from AN that the receiver is ready to accept

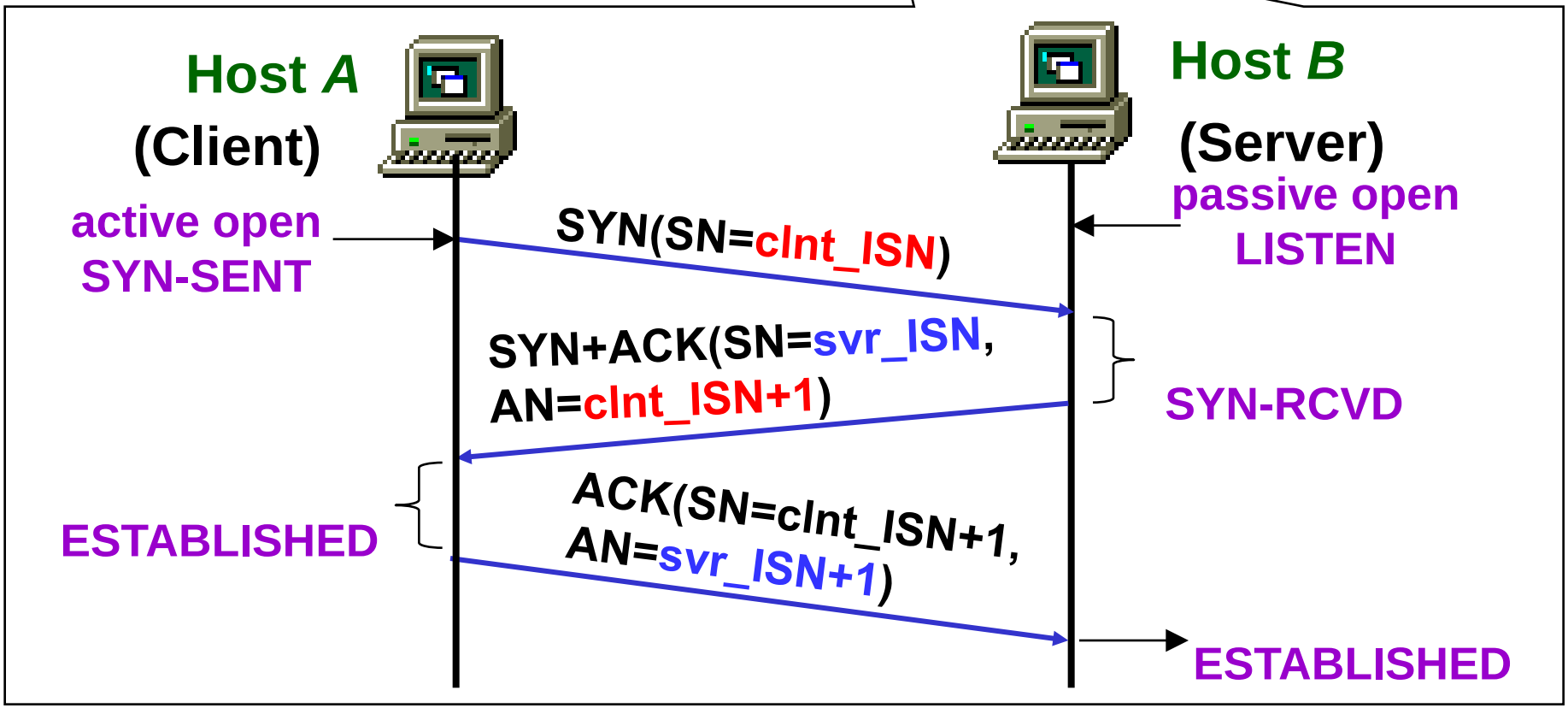
Note: Other fields will be discussed in relevant slides later.

TCP Connection Management

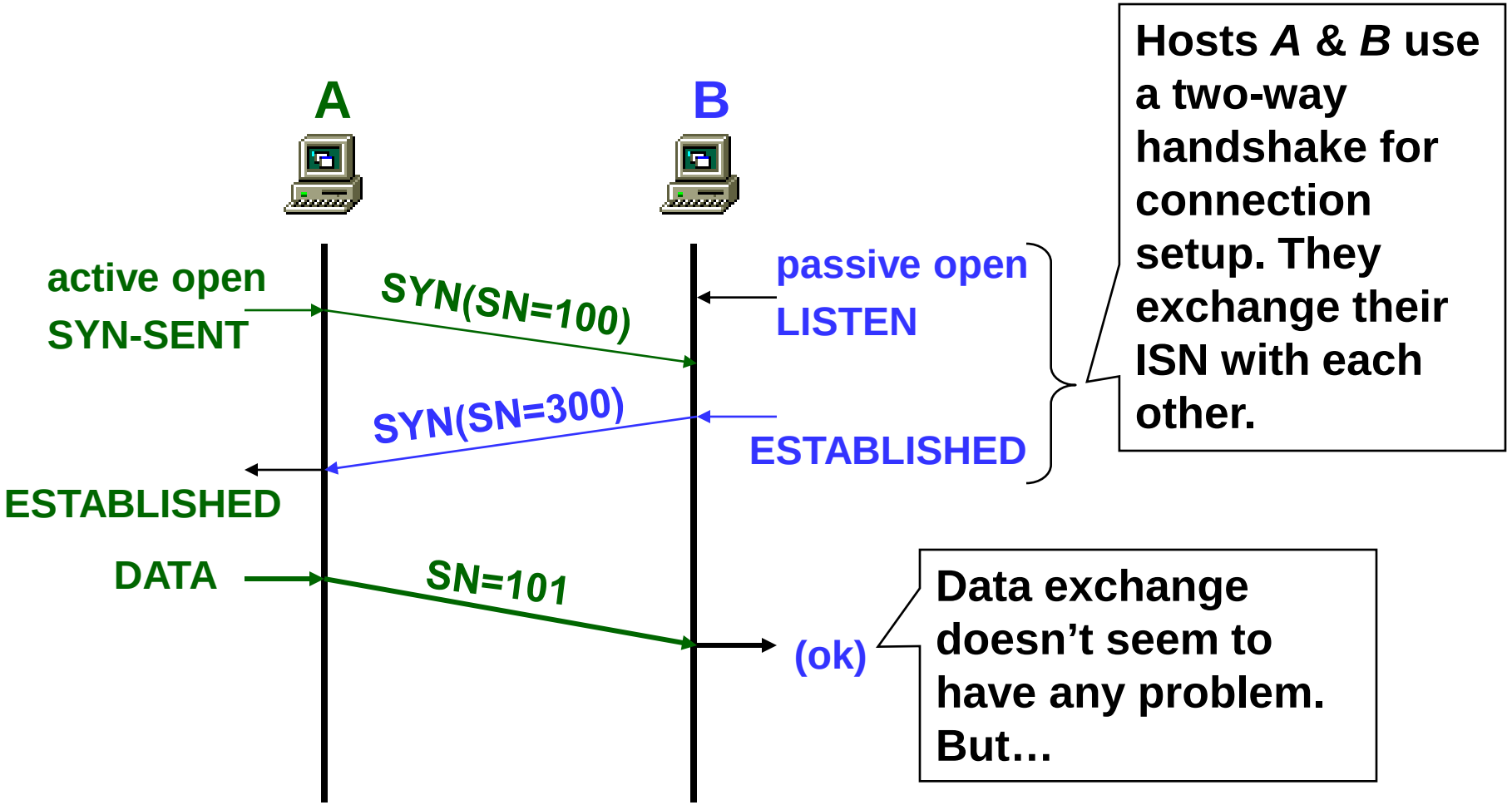
- **Connection establishment: serves the following purposes**
  - **ensure both ends are ready to communicate**
  - **establish initial sequence number (ISN)**
  - **exchange parameter, e.g. window size (in bytes)**
  - **allocate resources, eg. buffer space, etc. to support the connection**
- **Connection establishment starts with a synchronization (SYN) request.**

TCP: Connection Establishment

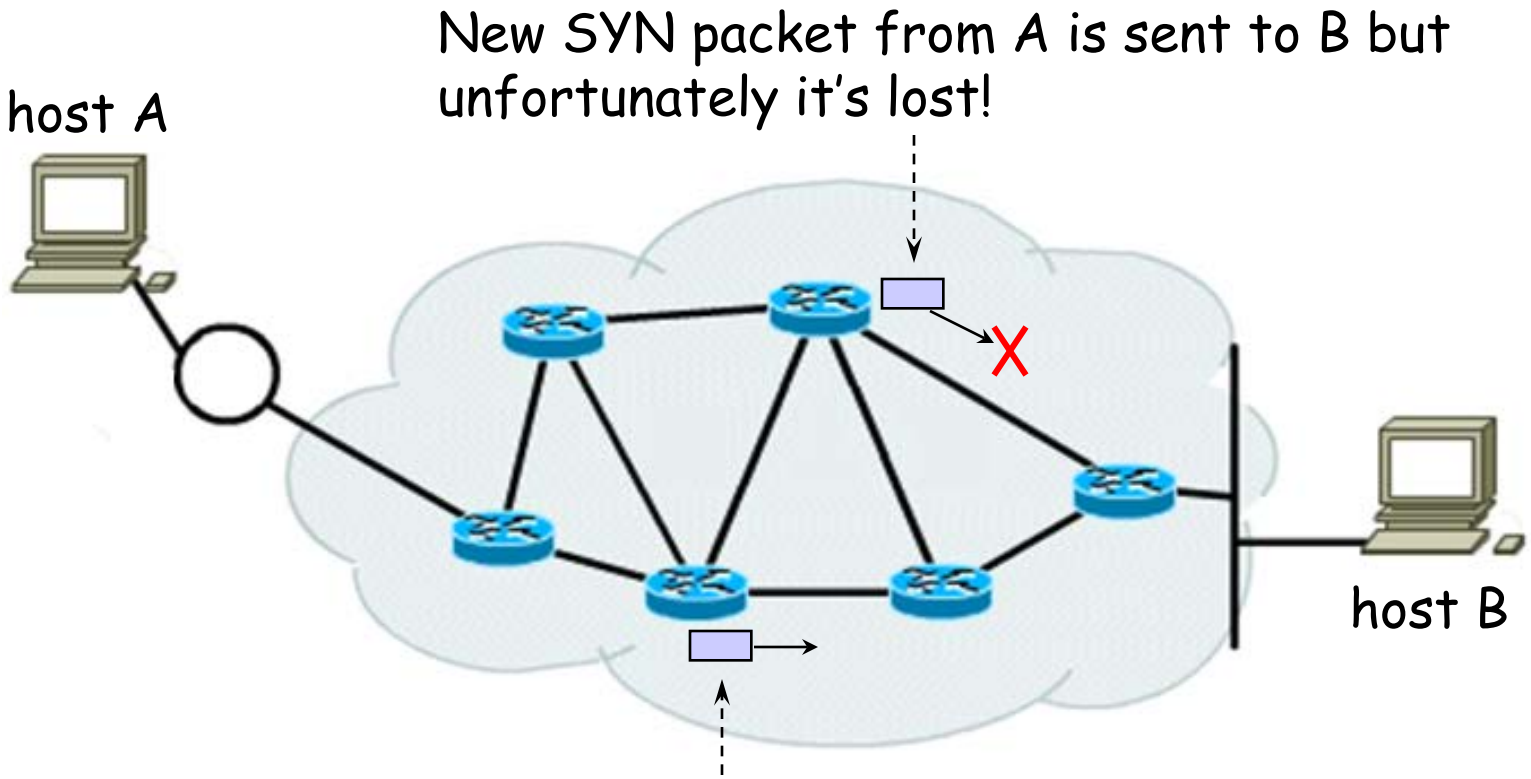
- **TCP uses 3-way handshake approach with positive acknowledgements (why not just 2-way?)**



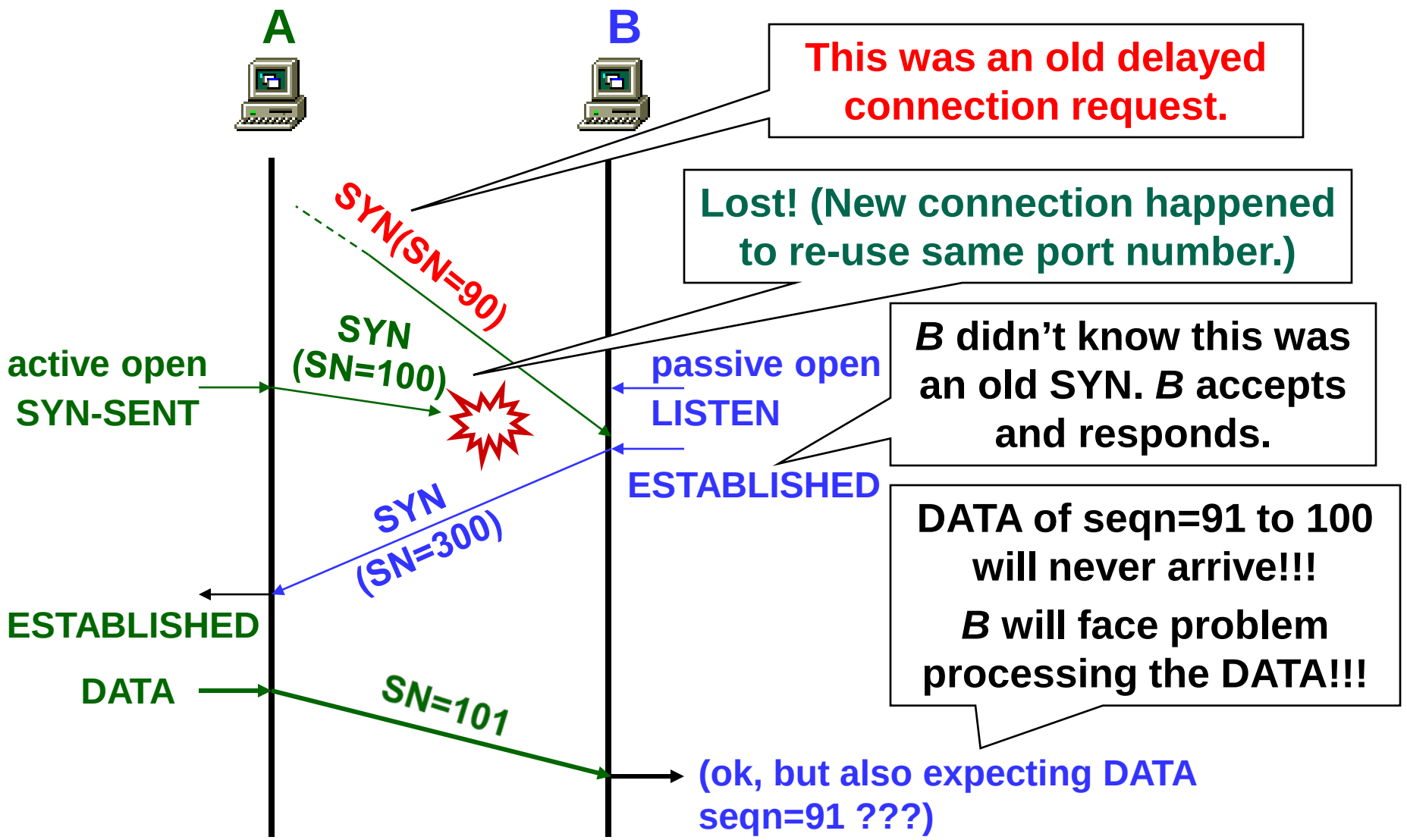
Two-way Handshake without positive acknowledgement



Recall that IP is unreliable. It is possible for packets to be delayed, lost, or duplicated due to timeout re-sent by TCP.

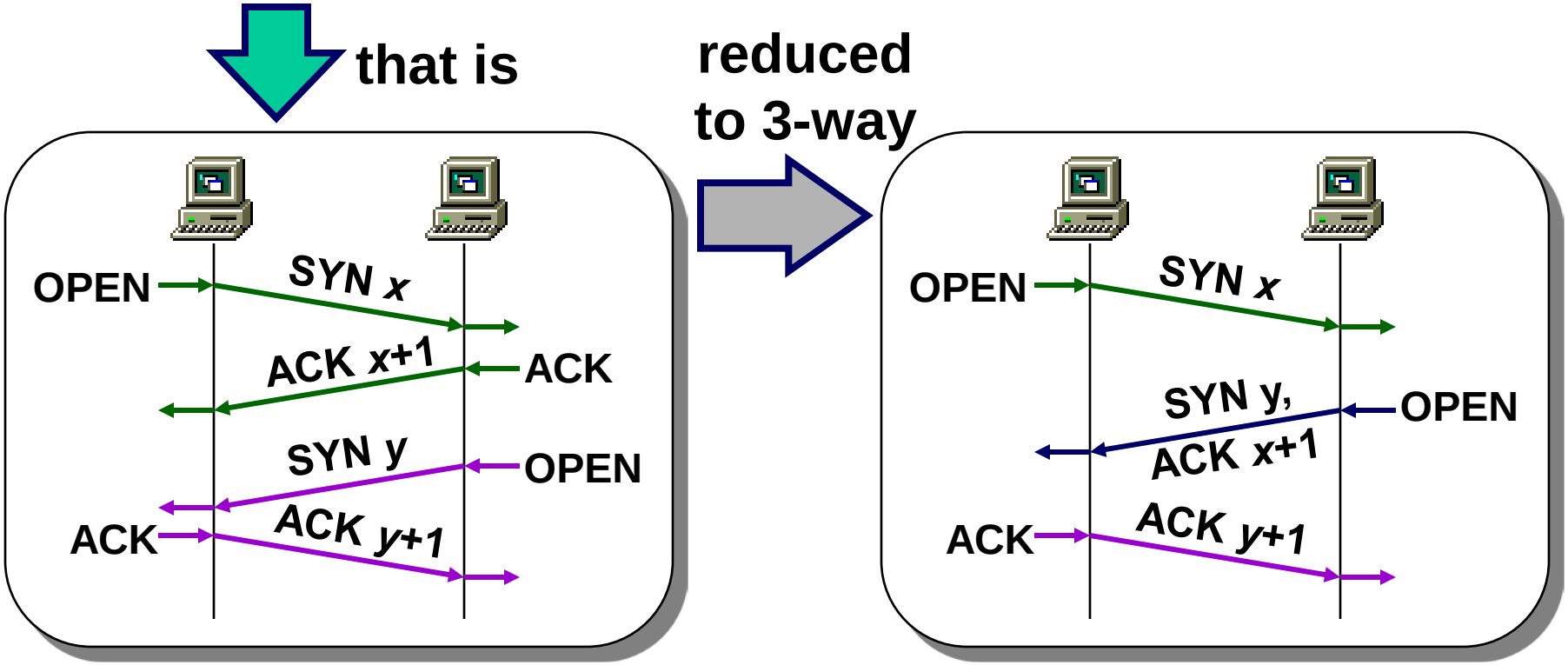


Two-way Handshake Problem

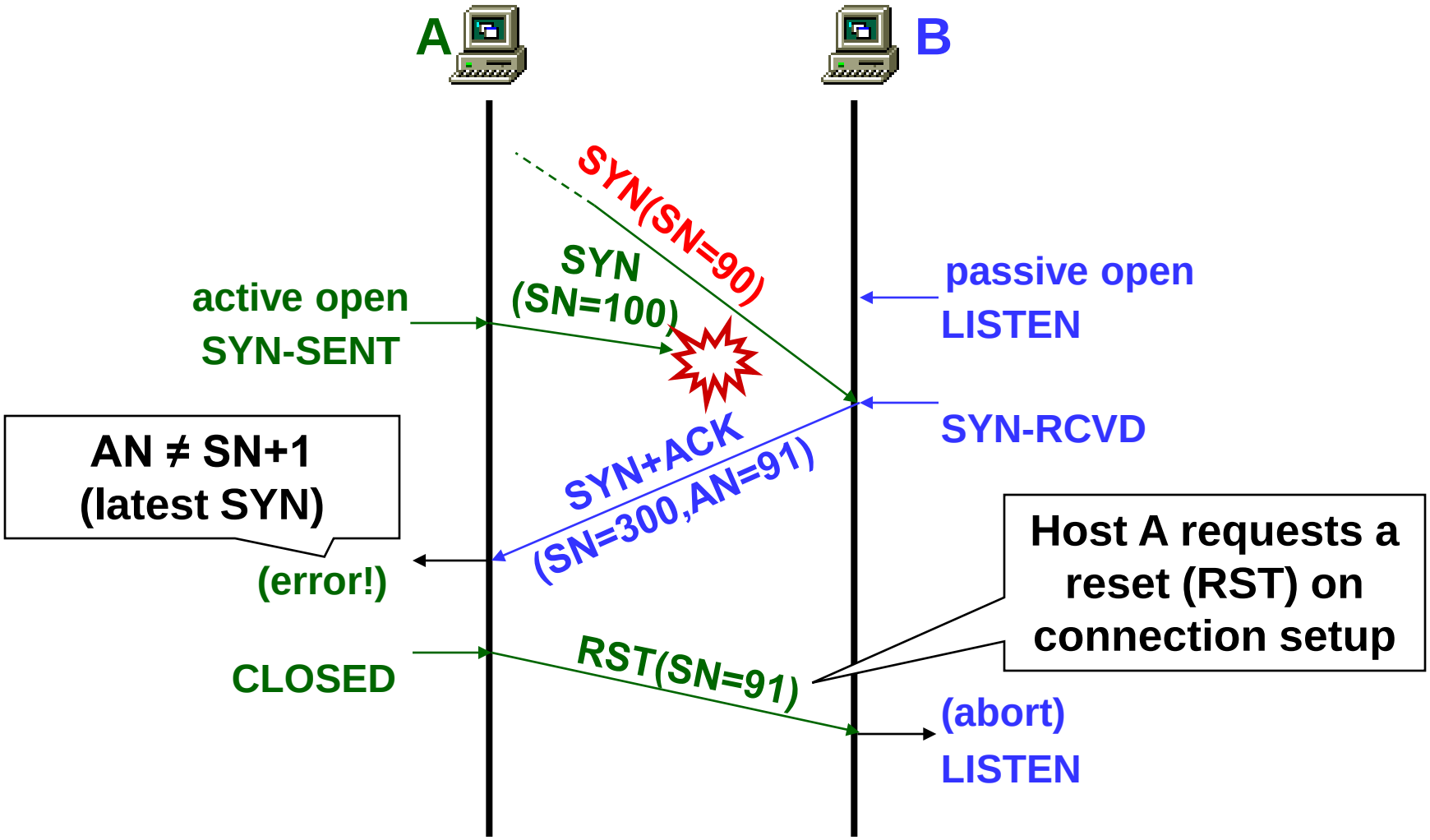


Solution: Three-way Handshake

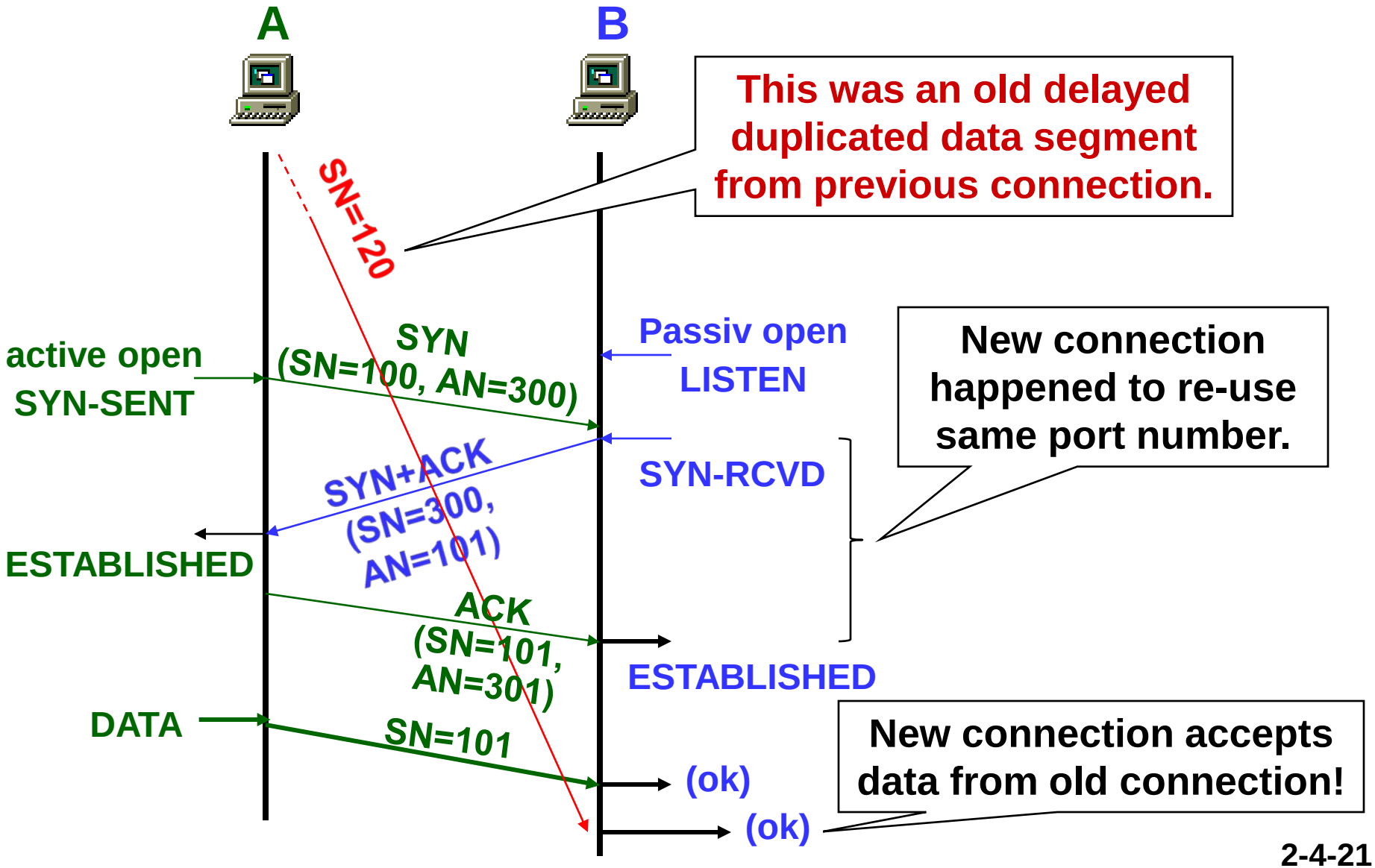
Synchronization can be made reliable if each connection request is positively acknowledged.



Three-way Handshake (Put to test)



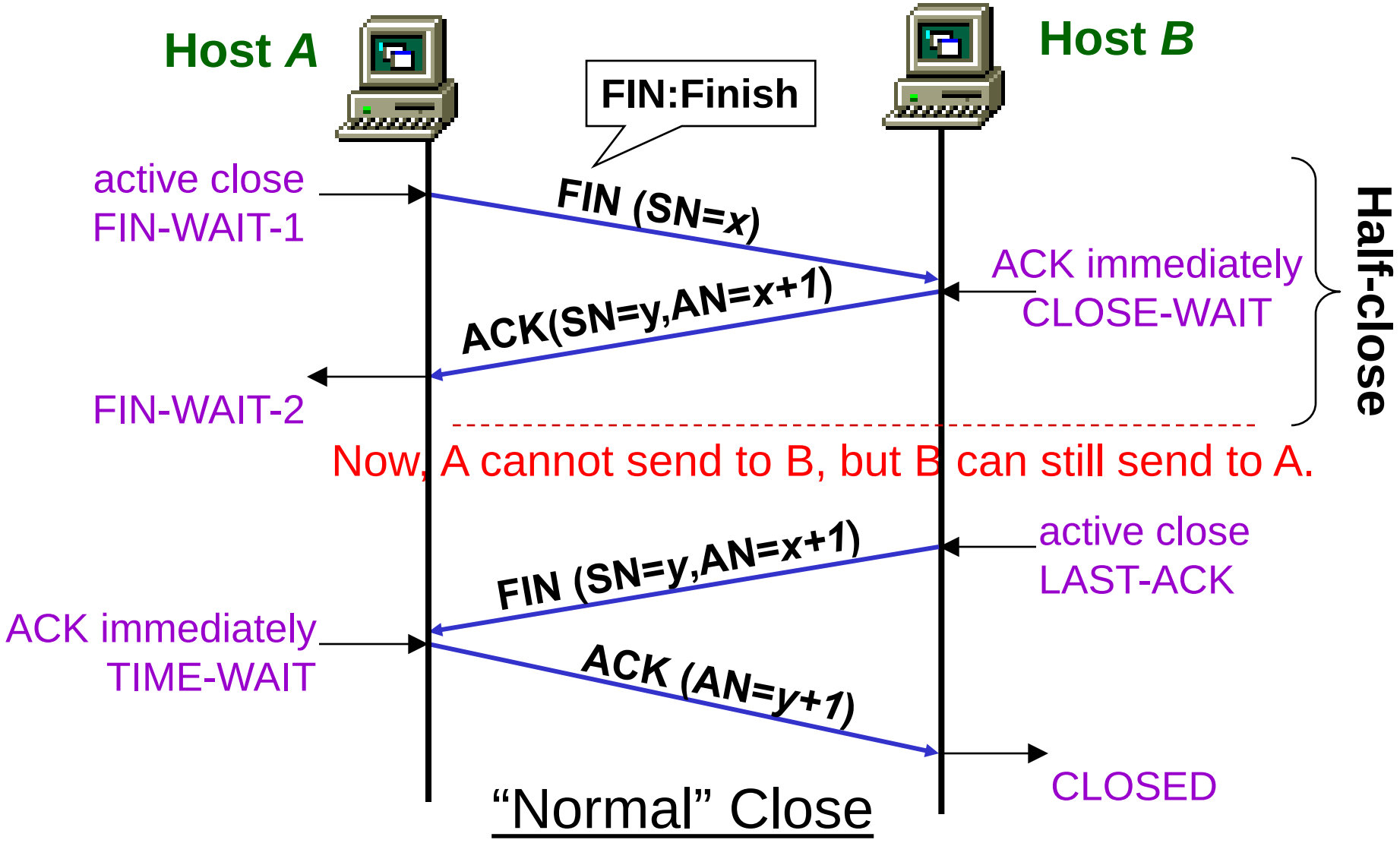
When new connection ISN is very close to old connection ISN …



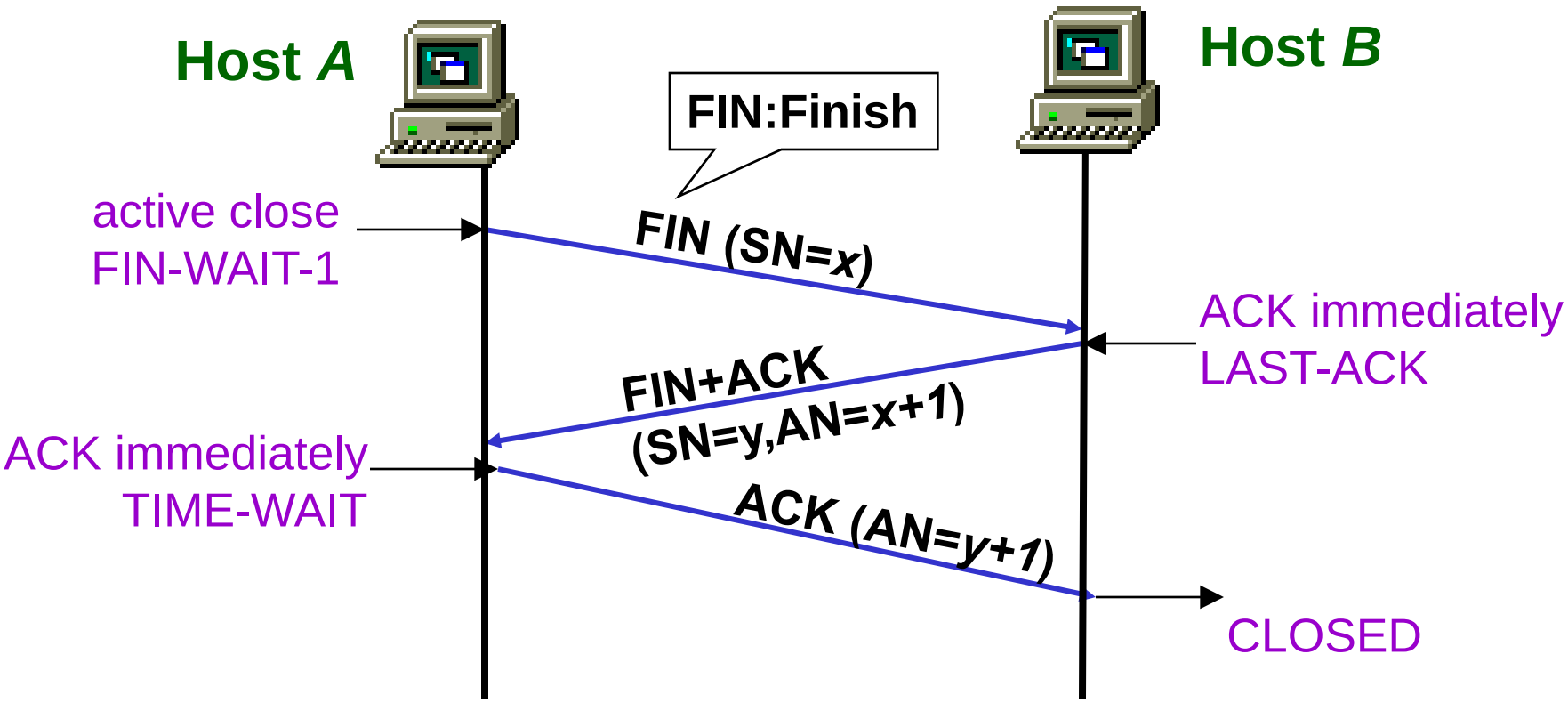
ISN (in RFC 793)

- **Increment ISN by 1 every 4μs (microsec)**
  - **One second of separation between two connections will have a gap of 250,000 in SNs! (hence segments from old connection will most likely not be mistaken as segments for new connection).**
  - **With a 32-bit SN, it takes about 4.55 hours for the same SN to cycle back (most likely the previously delayed segments with similar SN will no longer exist in the network anymore).**

TCP: Connection Termination

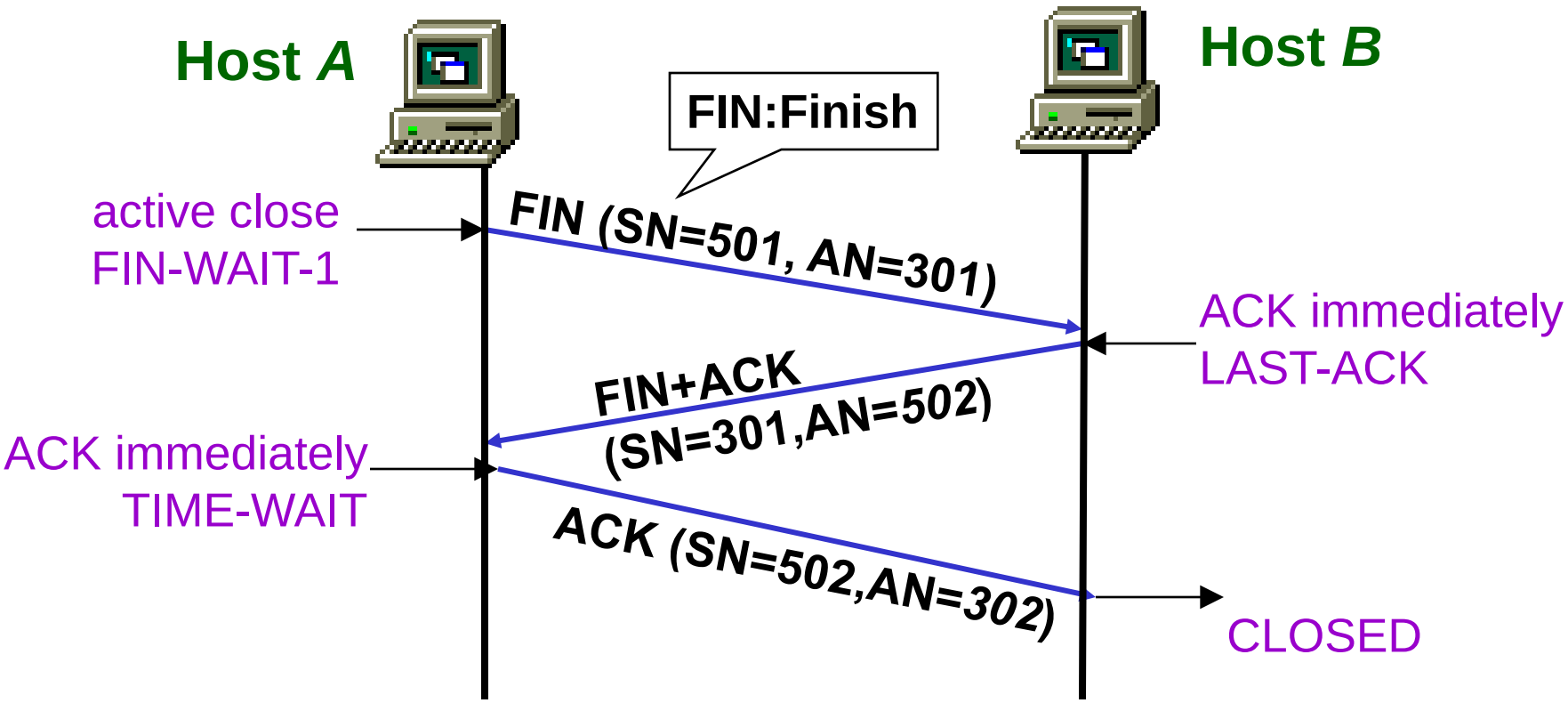


TCP: Connection Termination (another scenario)

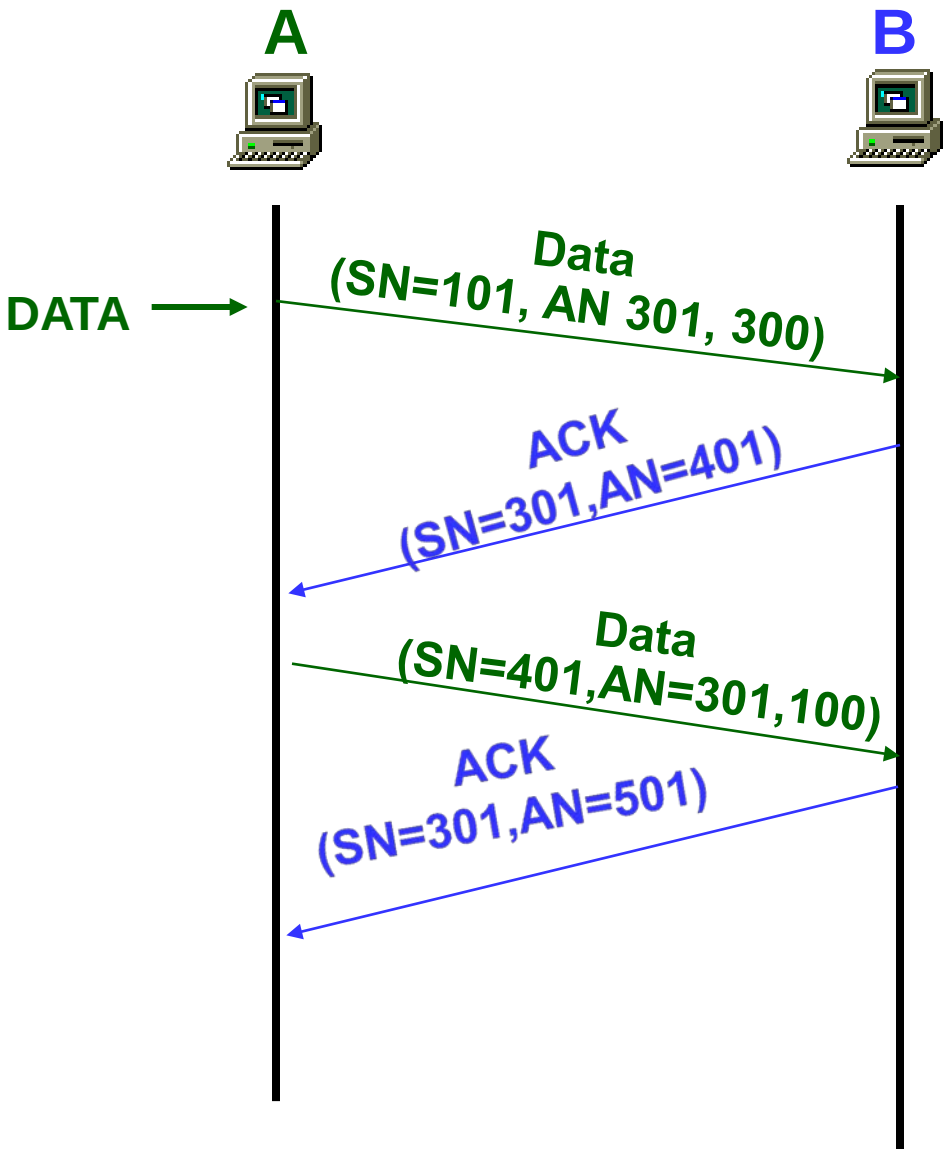


When B does not have outstanding data to send to A when receiving FIN from A.

TCP: Connection Termination (Numerical example)



Data Transfer… (A -→ B)



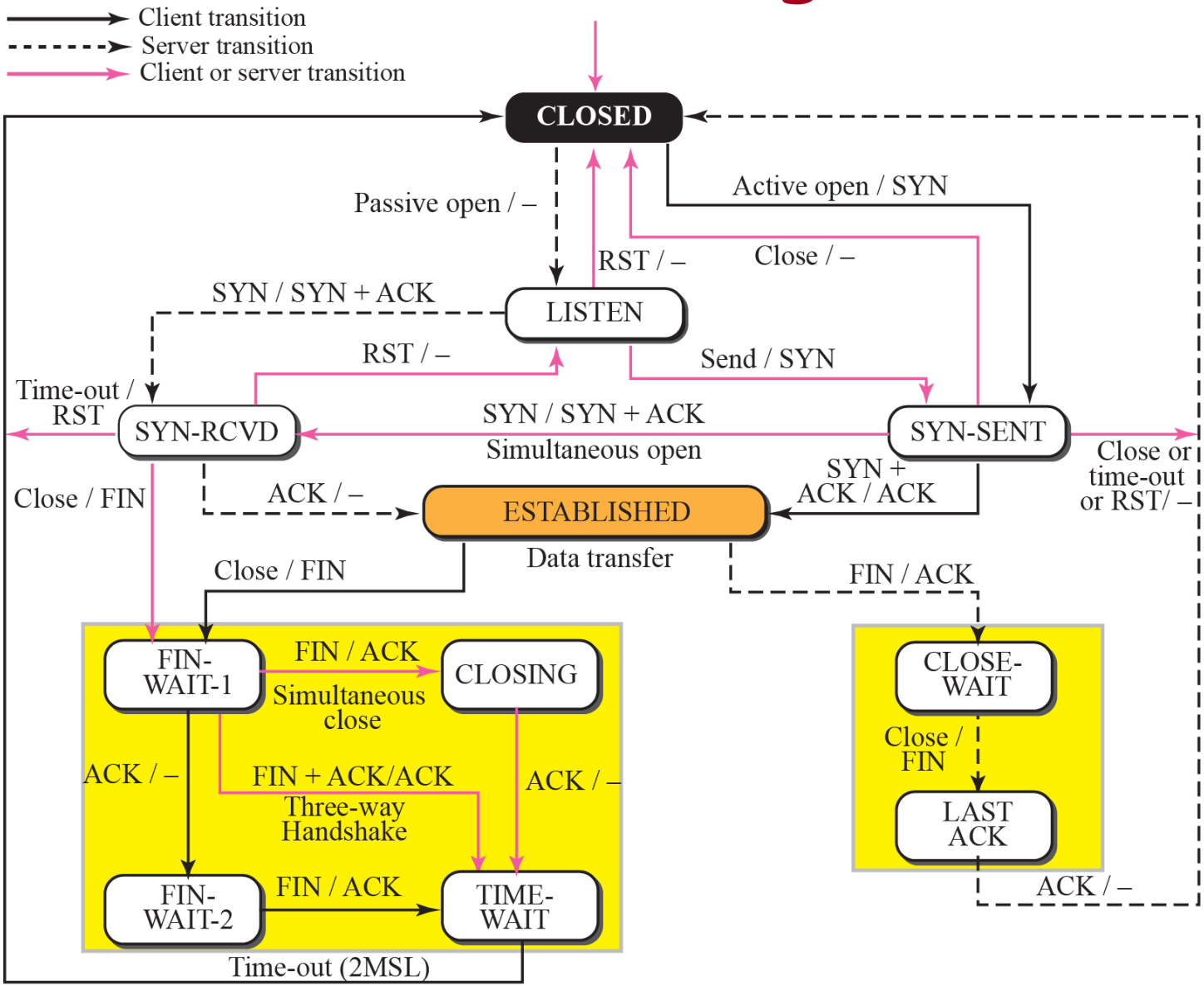
Segment format

(SN, AN, Data size)

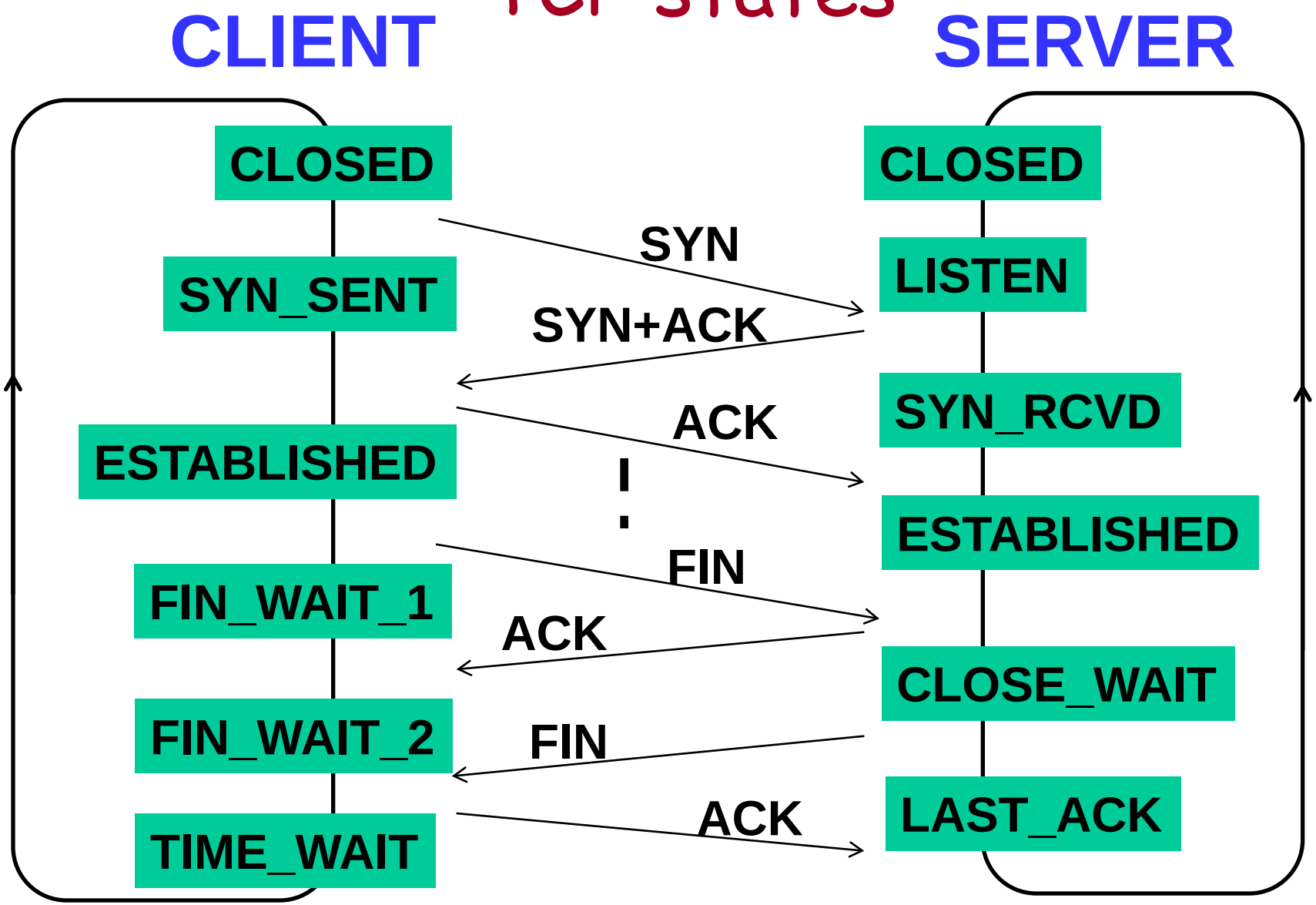
SN

- **Note:**
  - **A SYN and FIN segment does not carry data but consumes one sequence number**
  - **A SYN +ACK, FIN+ ACK does not carry data but consumes one sequence number**
  - **ACK segment, carrying no data, does not consume any sequence number.**

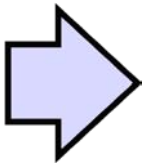
TCP State Diagram



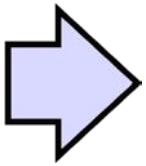
TCP states


CLIENT
SERVER
CLOSED
SYN_SENT
ESTABLISHED
FIN_WAIT_1
FIN_WAIT_2
TIME_WAIT
CLOSED
LISTEN
SYN_RCVD
ESTABLISHED
CLOSE_WAIT
LAST_ACK
SYN
SYN+ACK
ACK
FIN
ACK
FIN
ACK

TCP Flow Control

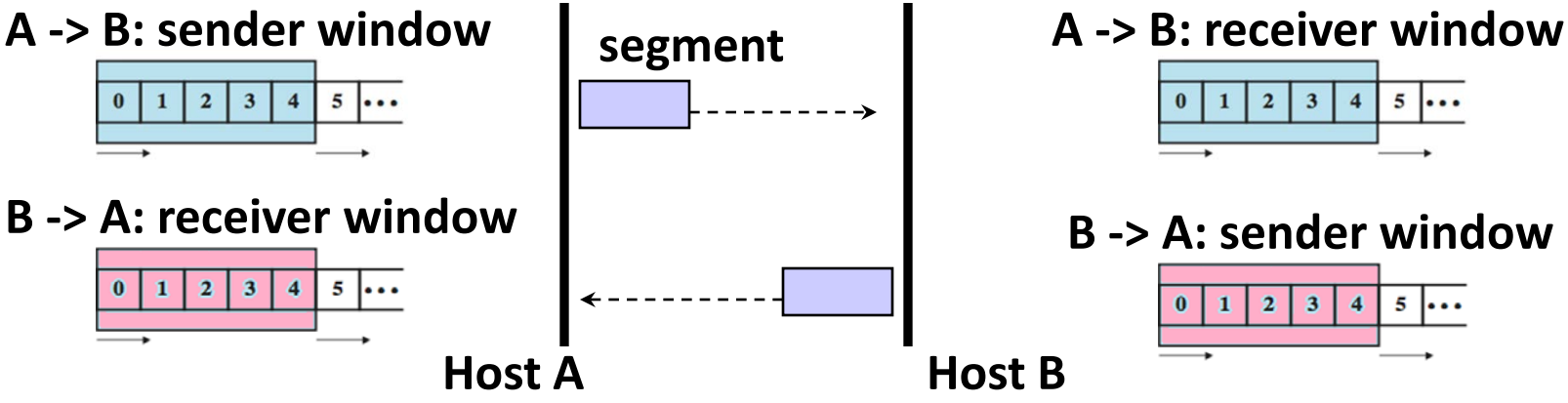
WHY? 

So that sender won't overrun receiver's buffers by transmitting too much, too fast.

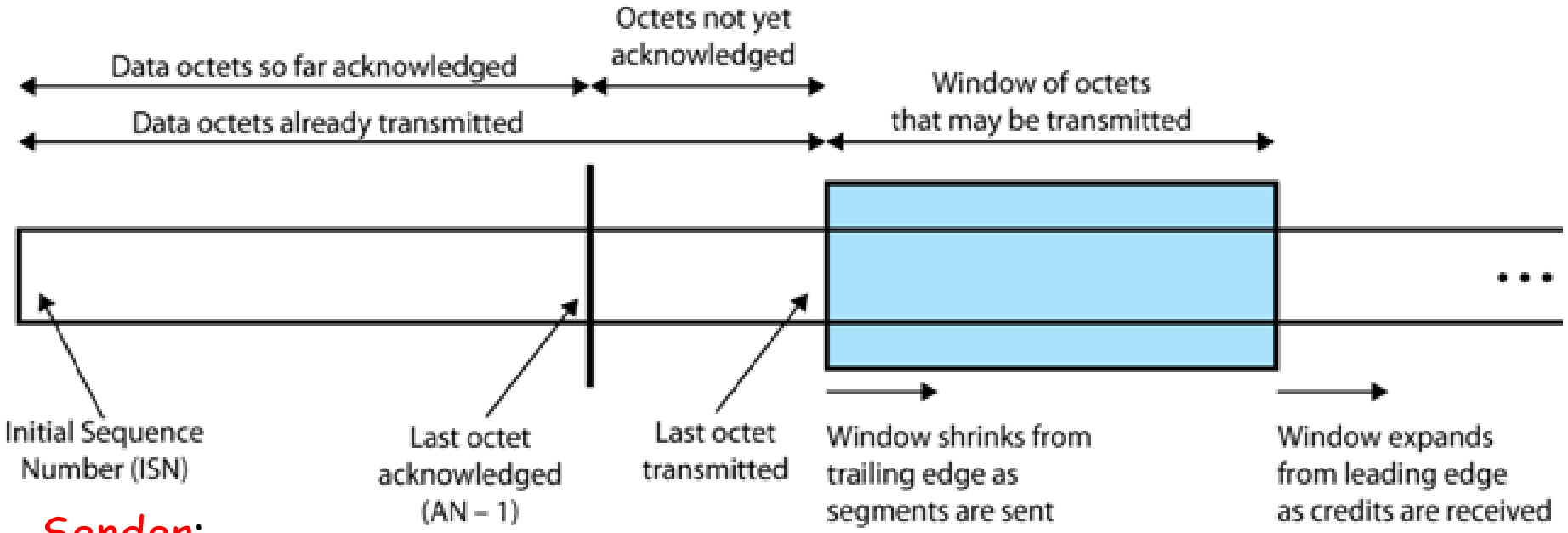
HOW? 

Similar to sliding-window flow control in datalink layer, although some details are different.

To support bi-directional data transfer, 2 pairs of windows are used:



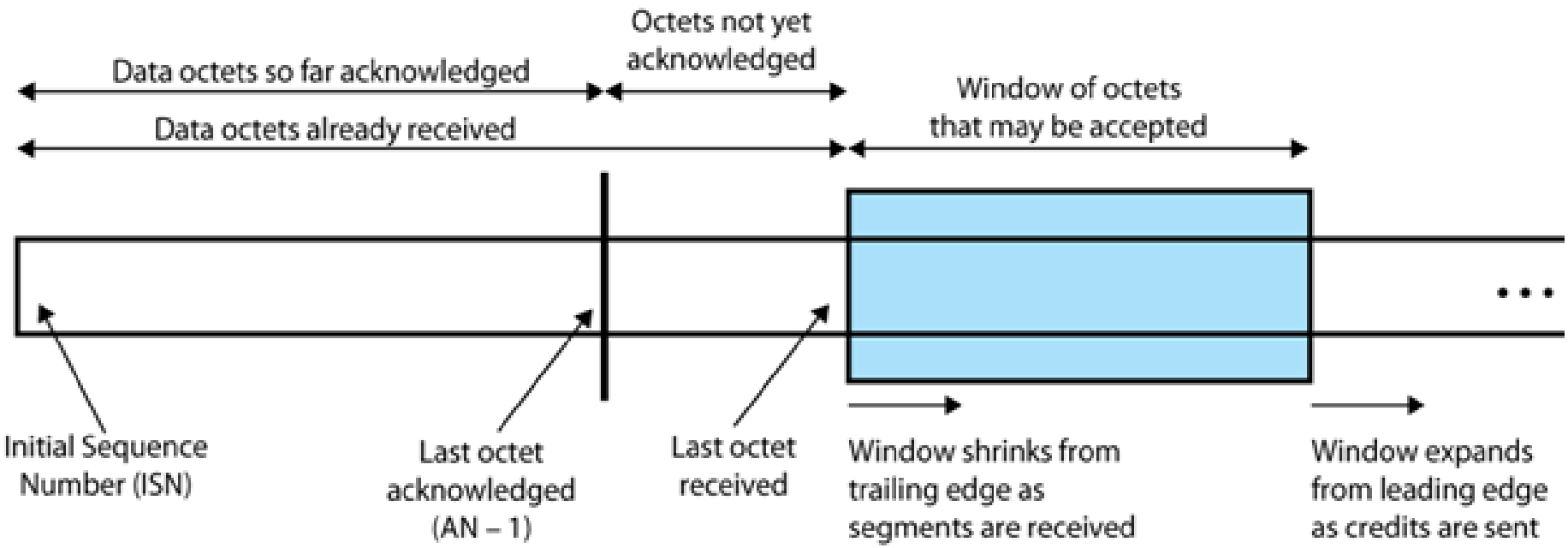
In sliding-window flow control, multiple segments are allowed to be in transit at the same time. The behaviour of sender is illustrated below:



Sender:

- Maintain a blue window representing bytes that can be transmitted without ACK
- When segment is sent, shrink blue window from trailing edge
- Stop sending when blue window size = 0
- When ACK is received, new blue window size = W bytes starting from AN.

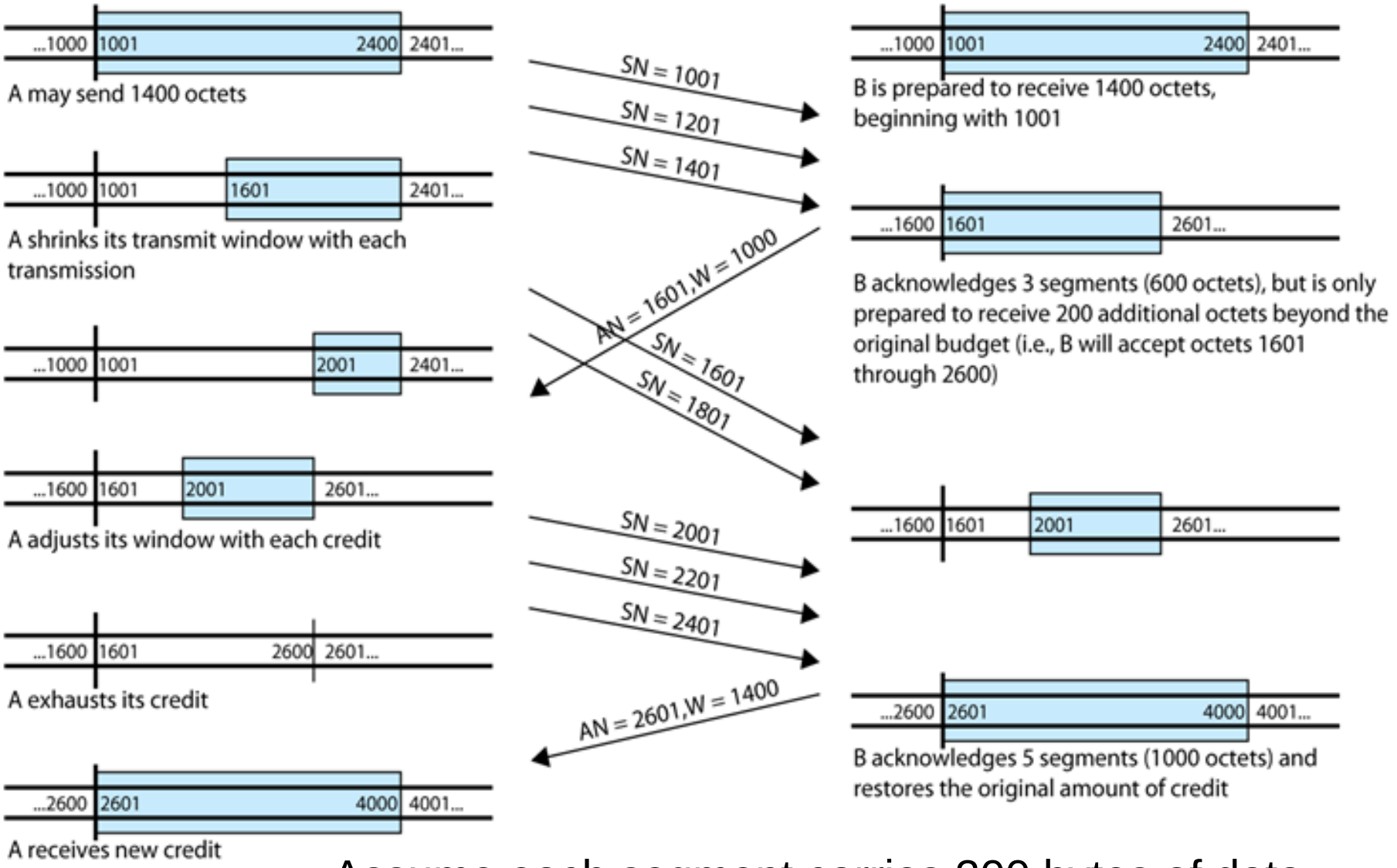
The corresponding behaviour of receiver in the sliding-window flow control is illustrated below:



Receiver:

- Maintain a blue window representing bytes ready to accept
- When segment is received, shrink blue window from trailing edge
- If NOT ready to accept more segments, send ACK with credit W = remaining blue window size
- If ready to accept more segments, send ACK with W > remaining blue window size, and expand blue window from leading edge

Here is an overall picture of how sliding-window flow control works:

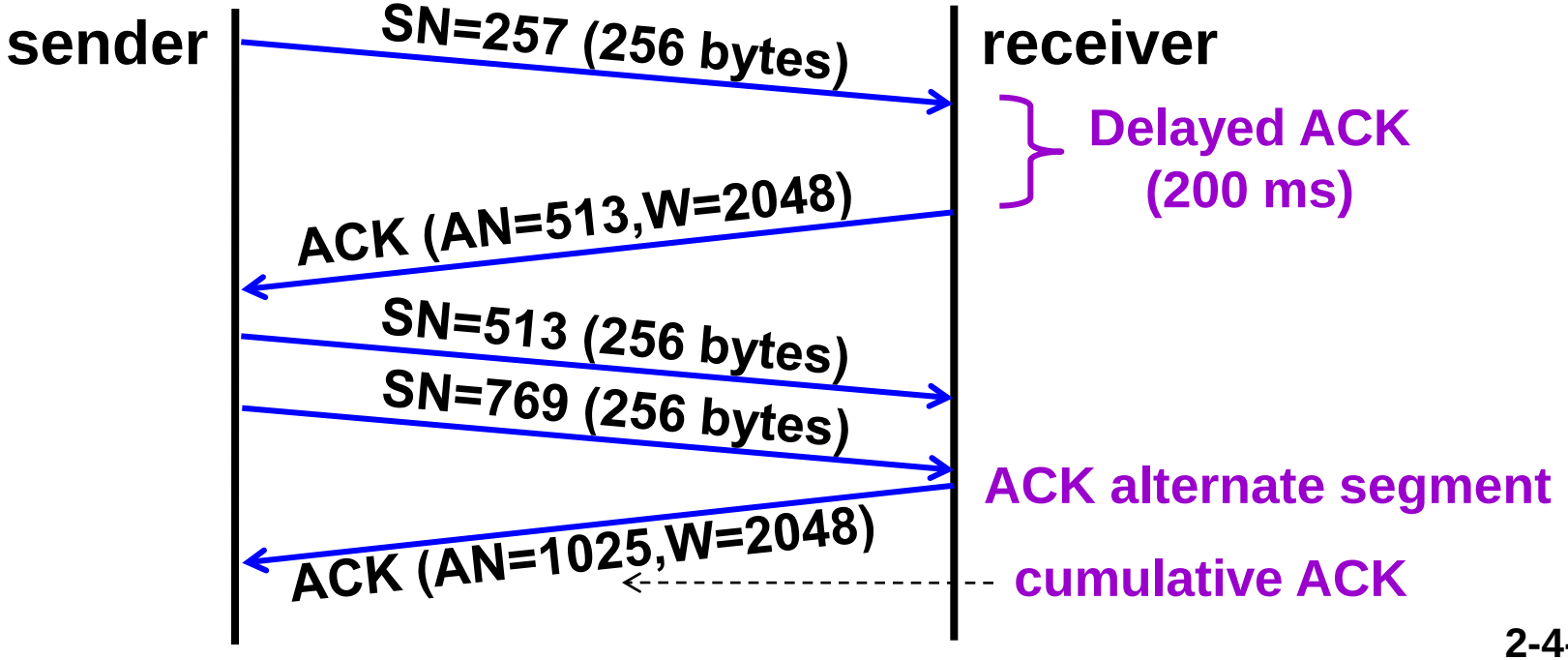


Assume each segment carries 200 bytes of data.

TCP Flow Control Enhancement 1 – Delayed ACK (RFC 1122)

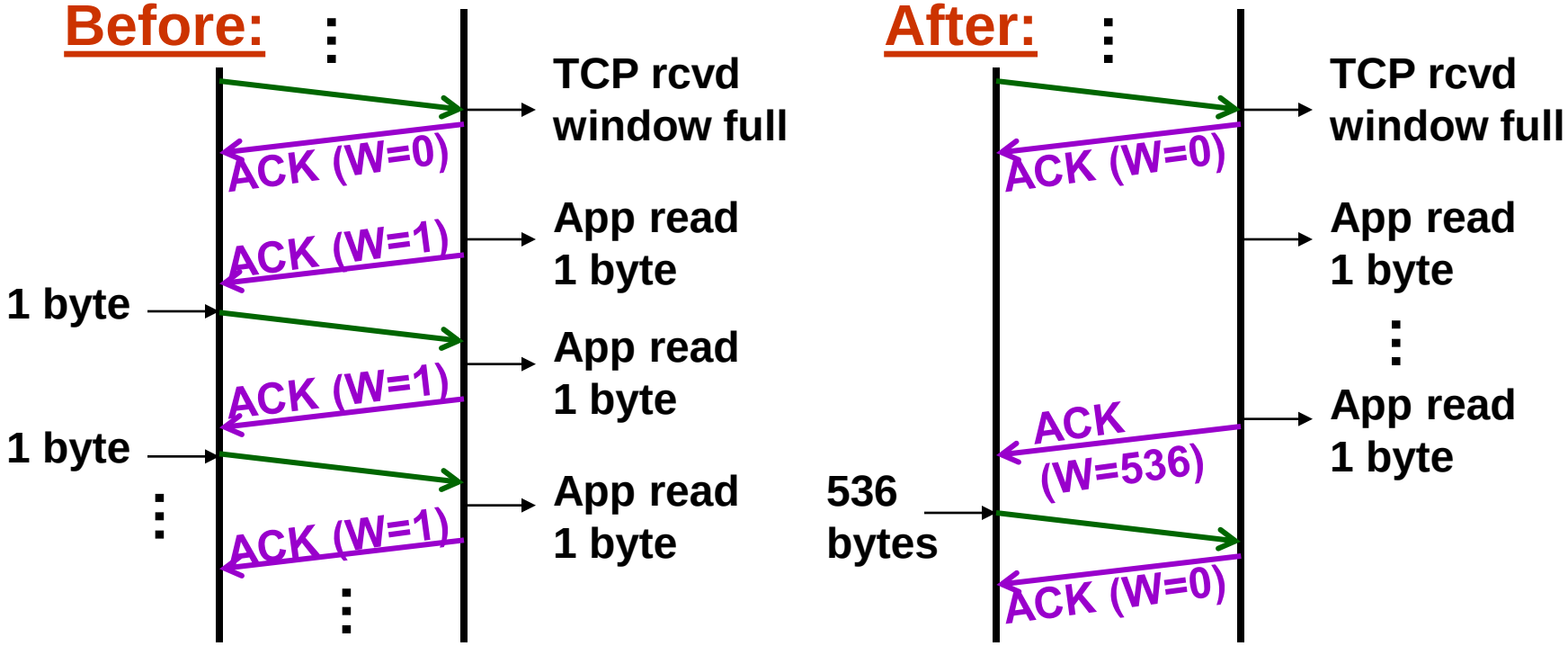
Problem: Wasteful to send ACK only segment (40 bytes TCP+IP headers)

- Maximum < 500 ms, to avoid error-control timeout re-sent
- ACK every alternate segment received
- (Note: Piggy-backed ACK can be sent immediately)



TCP Enhancement 2: avoiding Silly Window Syndrome at Receiver – Clark's solution (RFC 813, RFC 1122)

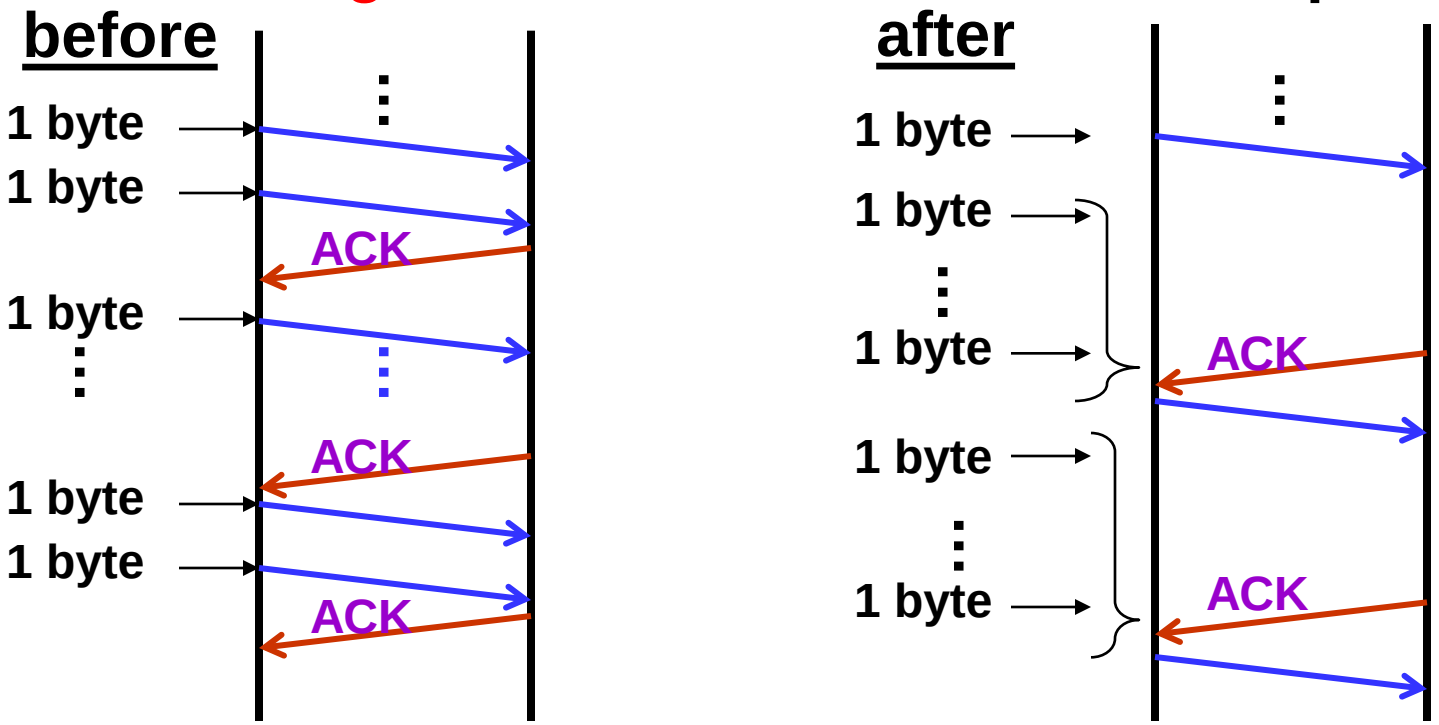
Problem: Wasteful for receiver to keep ACK with small window when sender can send more



Solution: Receiver ACK W=0 instead of small window size until free buffer gets large; e.g. buffer half empty.

TCP Enhancement 3: avoiding Silly Window Syndrome at Sender - Nagle's Algorithm (RFC 896, RFC 1122)

Problem: Wasteful for sender to keep sending small segments when receiver can accept more



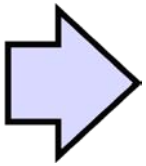
Solution: Send the 1st small segment, buffer the rest and send them together when ACK is returned.

Nagle's Algorithm

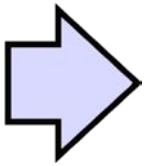
- Nagle's algorithm solves a problem but creates another problem because:
  - Delay transmission of segments is unfriendly to real-time traffic; e.g. interactive Internet games
- Delayed ACK and Nagle's algorithm running together can cause poor performance:
  - Sender waits for ACK but receiver delays ACK
  - Hence, TCP allows applications to enable/disable Nagle's algorithm; e.g. in Java

    ```
    socket.setTcpNoDelay(true/false);
    ```
- Nevertheless, Nagle's algorithm and Clark's solution are both required to solve the silly window syndrome together.

TCP Error Control

WHY? 

So that TCP can guarantee reliable service to application layer even when IP is unreliable.

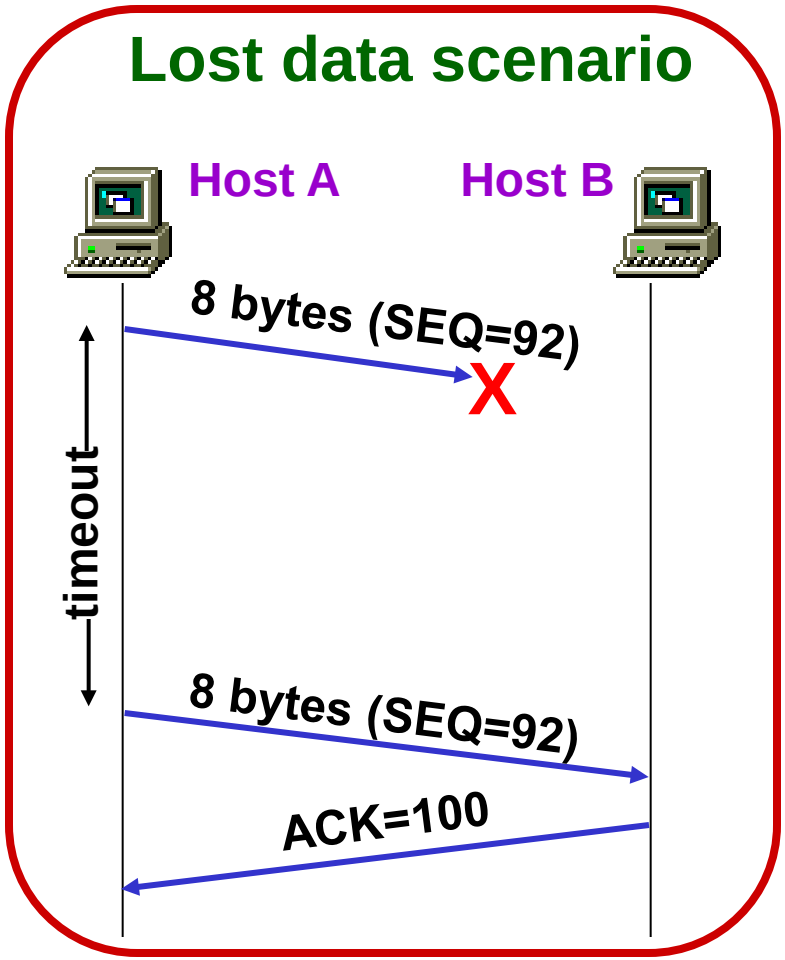
HOW? 

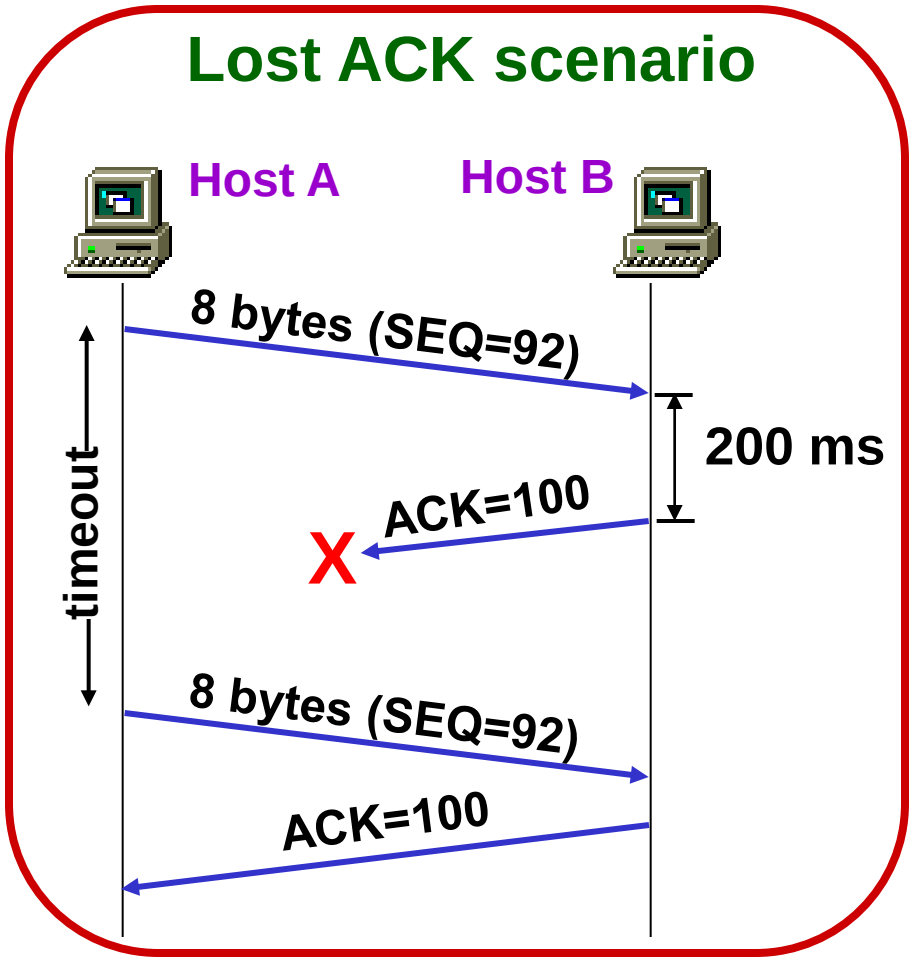
Similar to Selective-Reject in datalink layer, but the details are different.

Error types:

- Segments arriving out-of-order
  - Detected based on SN in TCP header; re-order and ACK
- Segments duplicated
  - Detected based on SN; discard and ACK
- Segments corrupted
  - Detected based on checksum in TCP header
  - Discard and wait for timeout retransmission
- Segments loss
  - Wait for timeout retransmission

Examples of timeout retransmission scenarios





Retransmission Timer

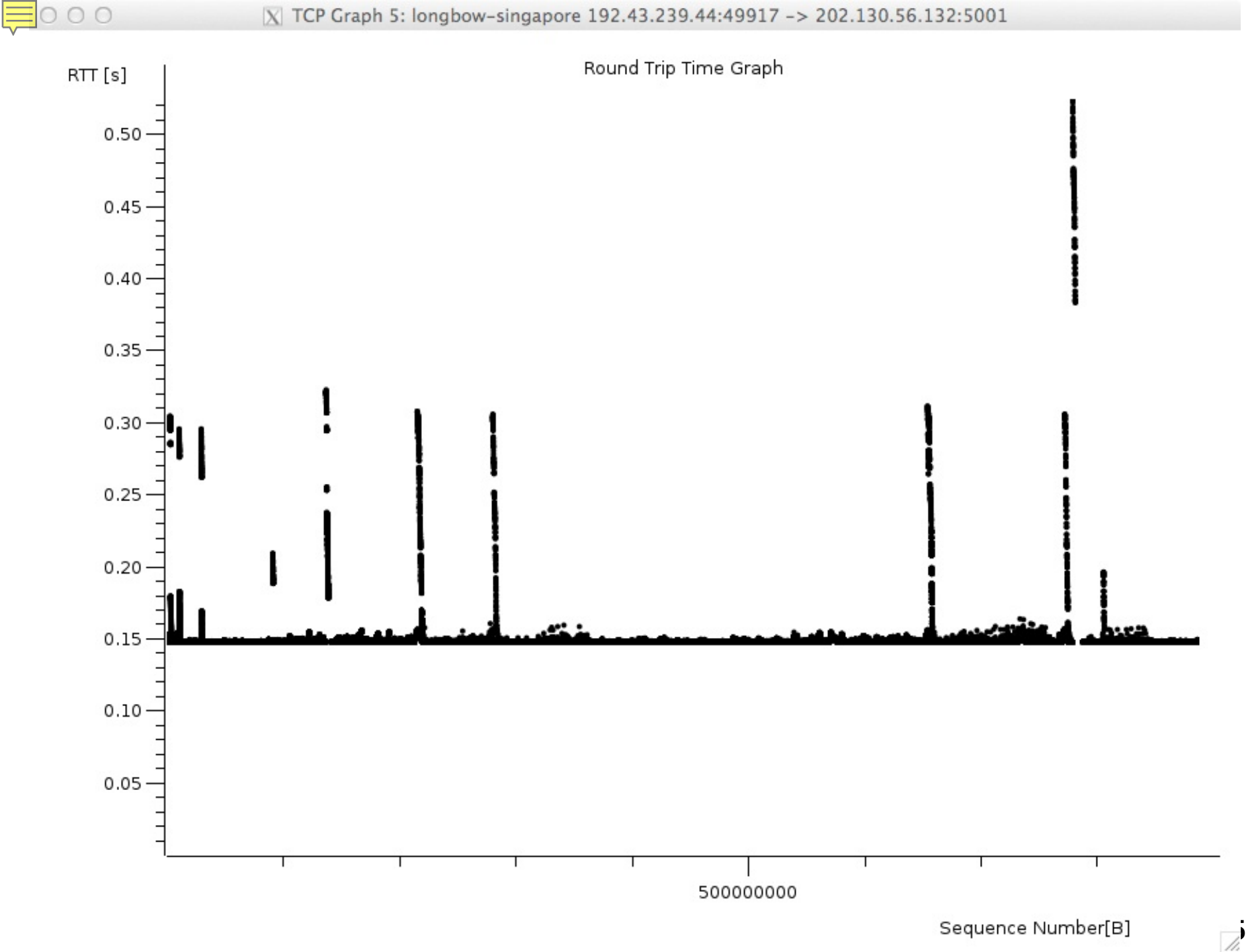
Problem: How long should TCP wait for the ACK before it retransmits that segment?

☹ **Too short** (premature timeout): there will be unnecessary retransmission

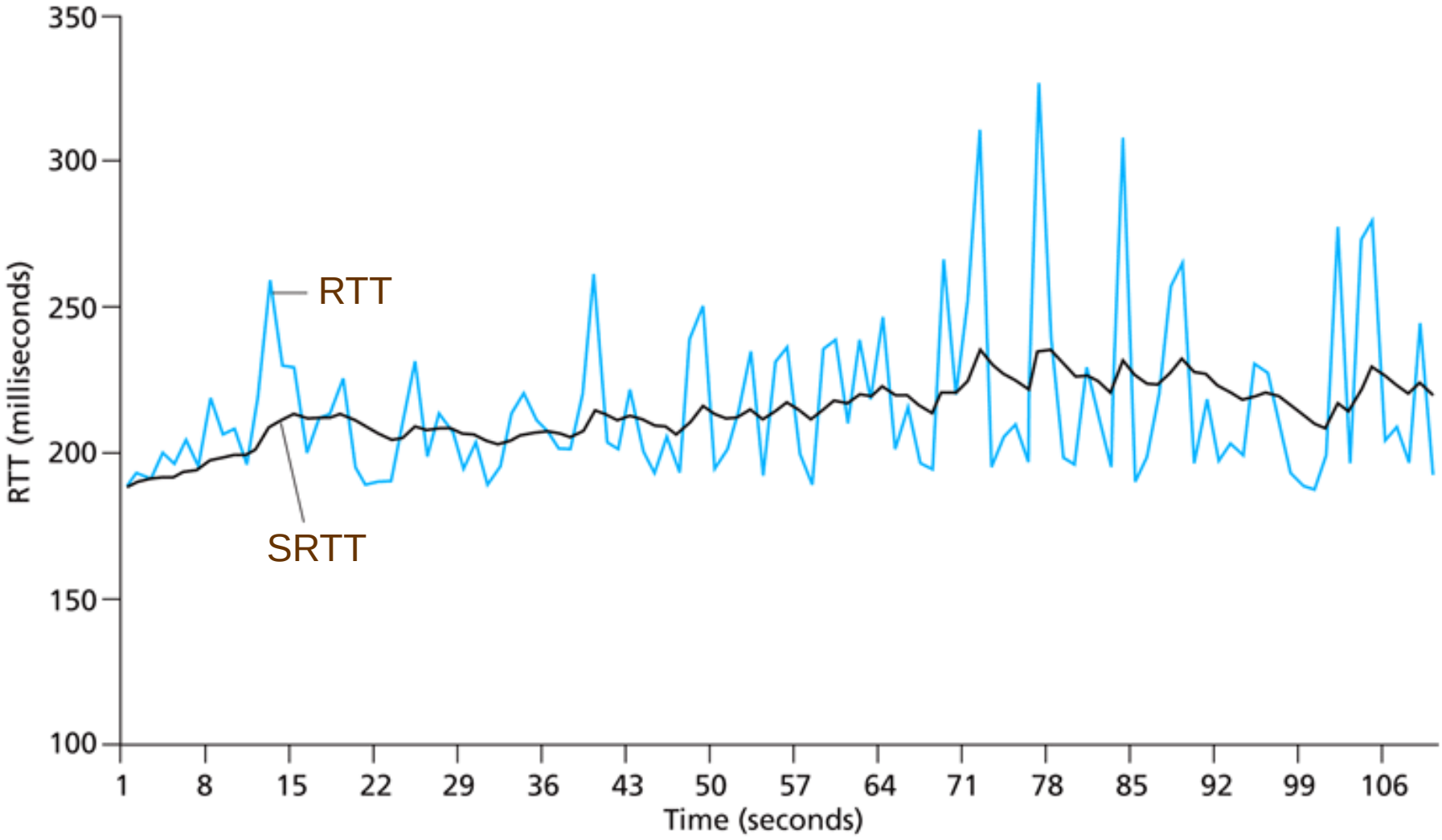
☹ **Too long** (slow reaction to losses): a long period of time is required to discover a lost segment

Note: Delays in network are constantly changing in practice, so timeout must be adaptive.

Solution: Measure Round Trip Time (RTT) and compute smoothed RTT (SRTT). The Retransmission TimeOut (RTO) is then derived from SRTT.


TCP Graph 5: longbow-singapore 192.43.239.44:49917 -> 202.130.56.132:5001
Round Trip Time Graph
RTT [s]
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
500000000
Sequence Number[B]

Example of RTT and SRTT



Computing Retransmission Timer RTO – Jacobson's Algorithm (RFC 6298)

- Initialization: **RTO = 1** (s)
- After 1st RTT is measured:

(smoothed RTT) **SRTT = RTT**

(RTT variation) **RTTVAR = RTT/2**

RTO = SRTT + 4×RTTVAR

Round	10	11
SRTT	20	((7/8)*20) + ((1/8)*16) = 19.5
RTTVAR	10	((3/4)*10) + ((1/4)*4)= 8.5
RTT		16
RTO	60	53.5

- After each subsequent RTT is measured:

RTTVAR = (1-β)×RTTVAR + β×|SRTT - RTT| , β = 1/4

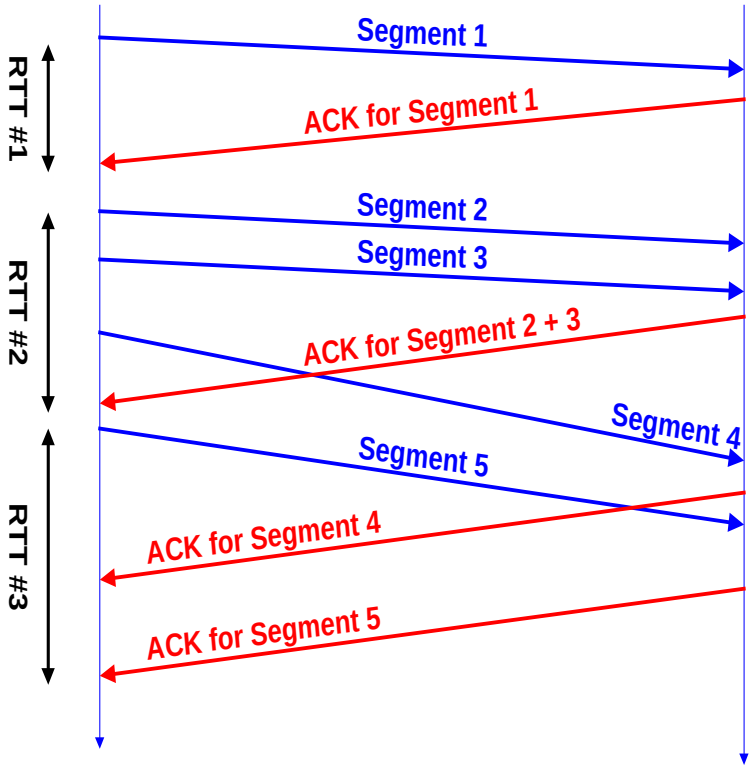
SRTT = (1-α)×SRTT + α×RTT , α = 1/8

RTO = SRTT + 4×RTTVAR

- **(minimum) 1s ≤ RTO ≤ maximum (at least 60s)**

Measuring RTT – Karn's Algorithm (RFC 1122)

- Each TCP connection measures the RTT from sending a segment to receiving its corresponding ACK.
- Typically, there is only one measurement ongoing at any time (i.e. measurements do not overlap).

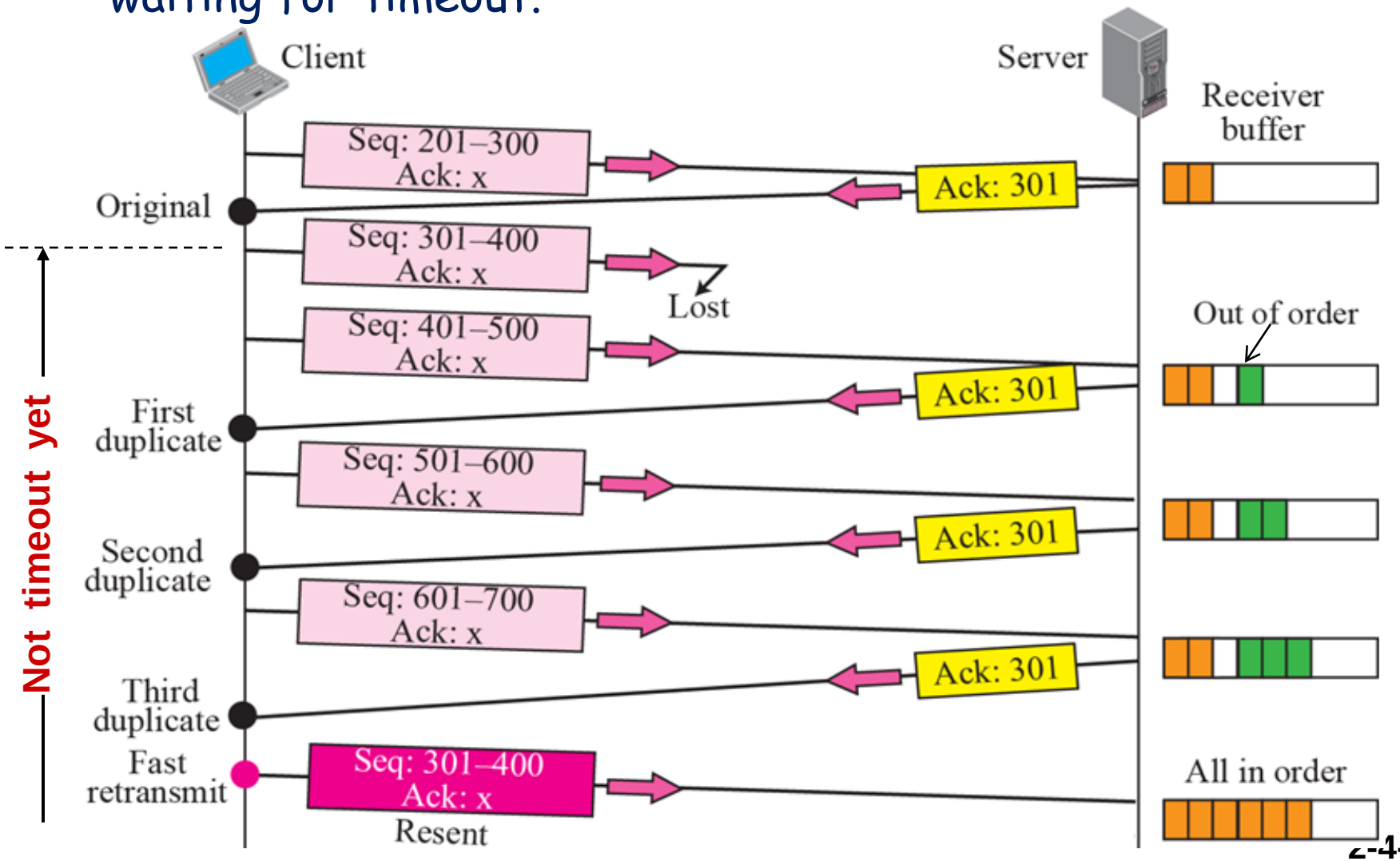


Karn's Algorithm:

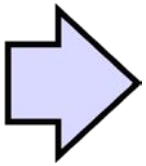
- If a segment is retransmitted due to timeout, ignore its measured RTT because it is ambiguous whether the ACK is for 1st or re-transmission.
- When retransmission occurs, set **RTO = 2 × RTO**

TCP Error Control Enhancement – Fast Retransmit

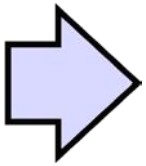
- Fast retransmit if receiving 3 duplicate ACKs instead of waiting for timeout.



Congestion Control

WHY? 

To prevent senders from sending too much traffic to the network such that it becomes overly congested and useless; informally, to be a "considerate user" of the network

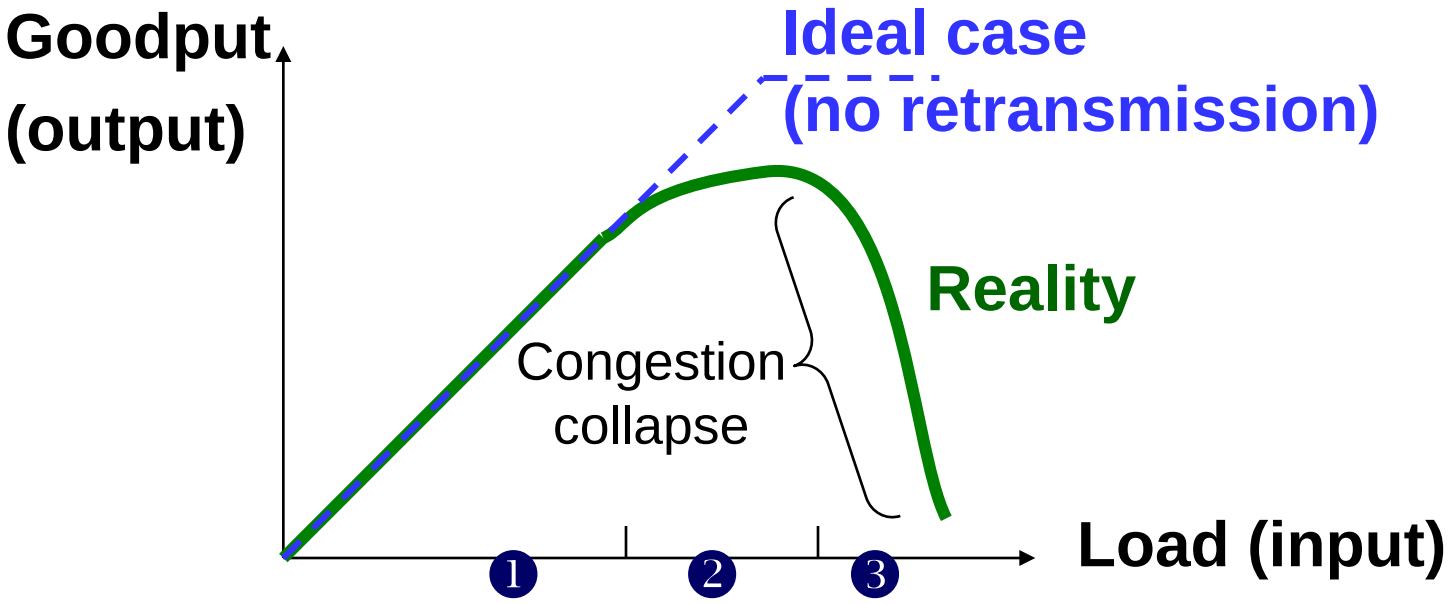
HOW? 

Implement congestion control algorithm with a congestion window which controls the amount of traffic that a connection can send

TCP sender assumes network congestion when loss events occurred:

- **Timeout or receiving duplicate ACKs**
  - **Possibly due to queueing or buffer overflow at routers which are signs of congestion**

What will happen if network congestion is ignored? Congestion Collapse!



Output = Input + Retransmission

1. **Zero retransmission, hence Output = Input**
2. **Several retransmissions, hence output continues to increase slightly as input increases**
3. **Transmissions are dominated by retransmissions**

TCP Congestion Control

- **Two phases**
  - **Slow Start**
  - **Congestion Avoidance**
- **Two parameters:**
  - ***cwnd*: Congestion Window, measured in number of MSS (maximum segment size, typically 536 bytes, but can be changed by using TCP options field).**
  - ***ssthresh*: Slow Start Threshold defines the point to transit from slow start to congestion avoidance phase; in practice, typically set large value for initial *ssthresh (half- maximum number of MSS)***

Note: With congestion control, maximum data bytes that can be sent without ACK = min { W, *cwnd x MSS* }

Window size in bytes

Convert to bytes

TCP Slow Start Phase

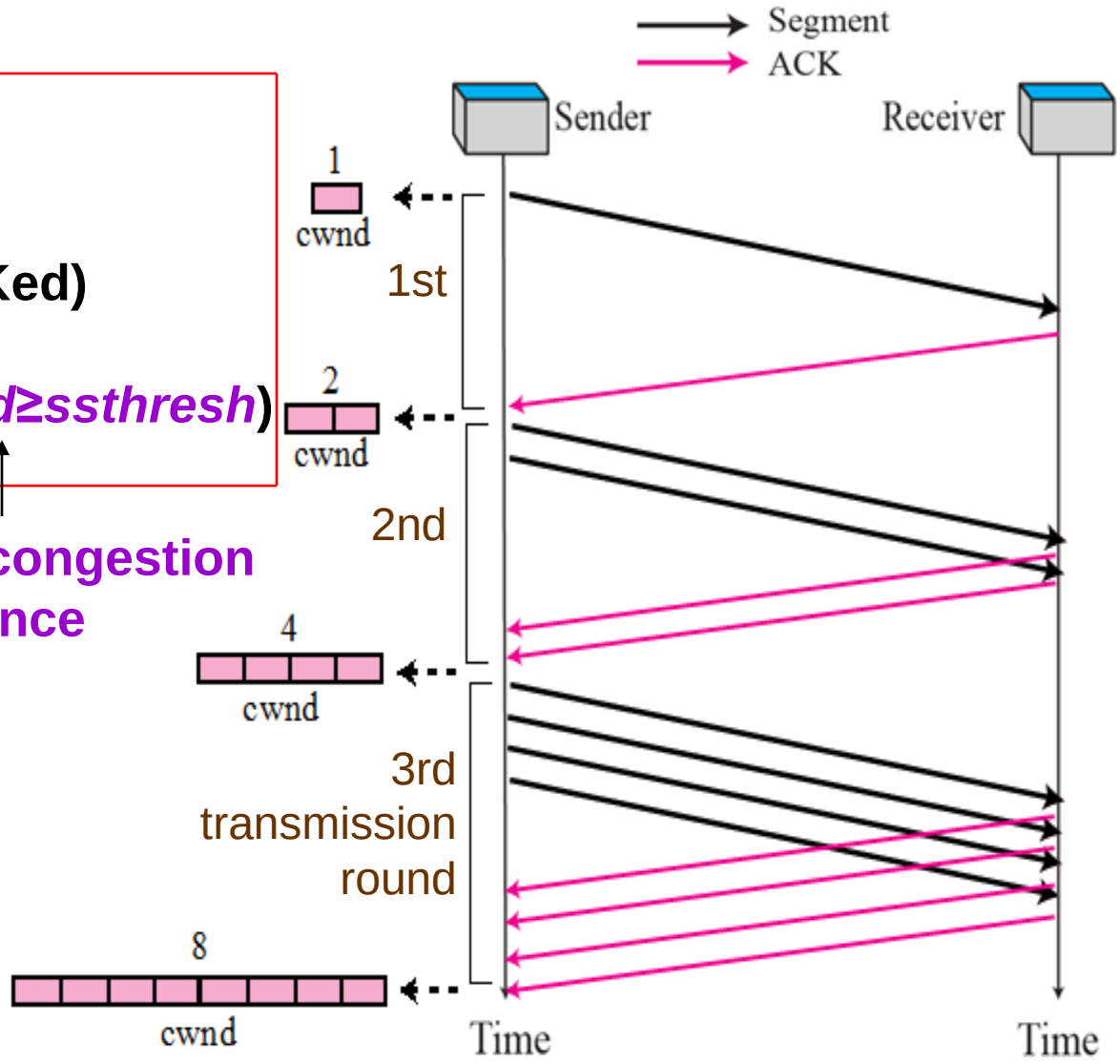
Slow start

initialize: *cwnd* = 1
do
for (each segment ACKed)
cwnd++
until (loss event OR *cwnd*≥*ssthresh*)

Go to recovery

Go to congestion avoidance

- ***cwnd*** increases exponentially at each transmission round (not so slow after all!)



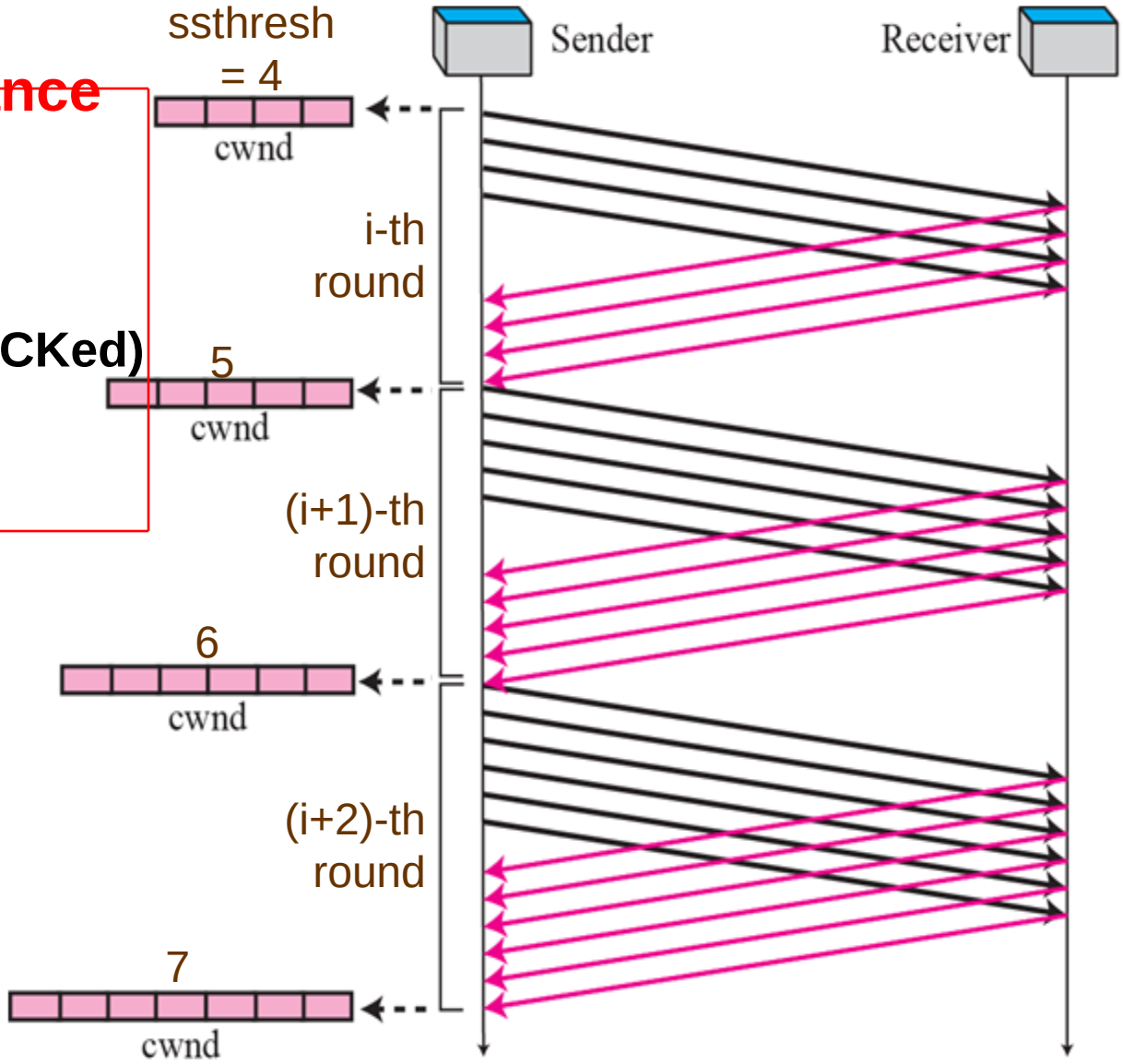
TCP Congestion Avoidance Phase

Congestion avoidance

```
/* slow start is over   */
/* cwnd ≥ ssthresh      */
do
    for (every segment ACKed)
        cwnd += 1/⌊cwnd⌋
until (loss event)
```

Go to recovery

- In contrast, ***cwnd*** only increases linearly at each transmission round.



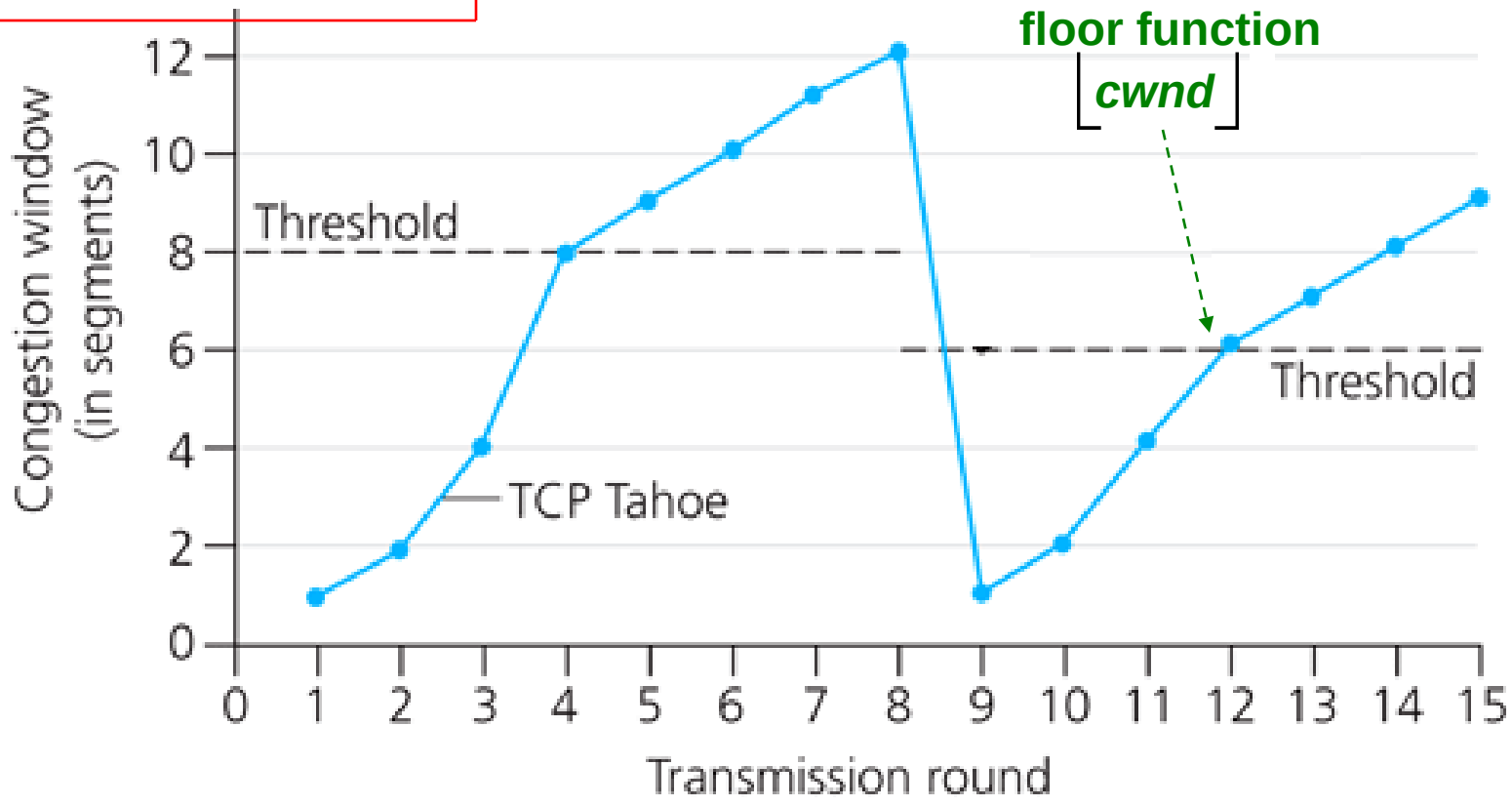
TCP Congestion Control – Tahoe Algorithm

Recovery

ssthresh = $\lfloor$ ***cwnd*/2** $\rfloor$

cwnd **= 1**

go back to slow start

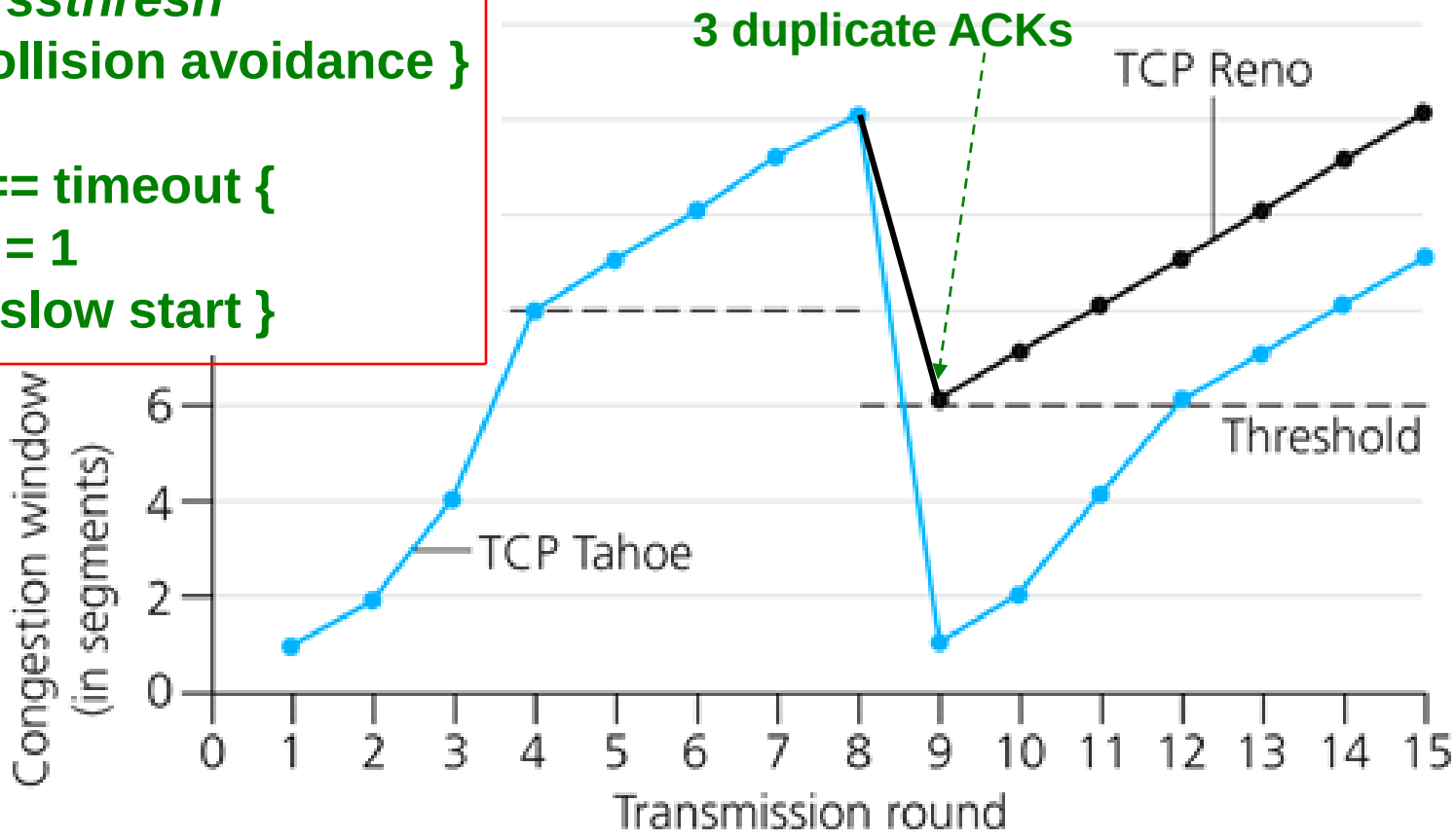


TCP Congestion Control – Reno Algorithm: Implement Fast Recovery

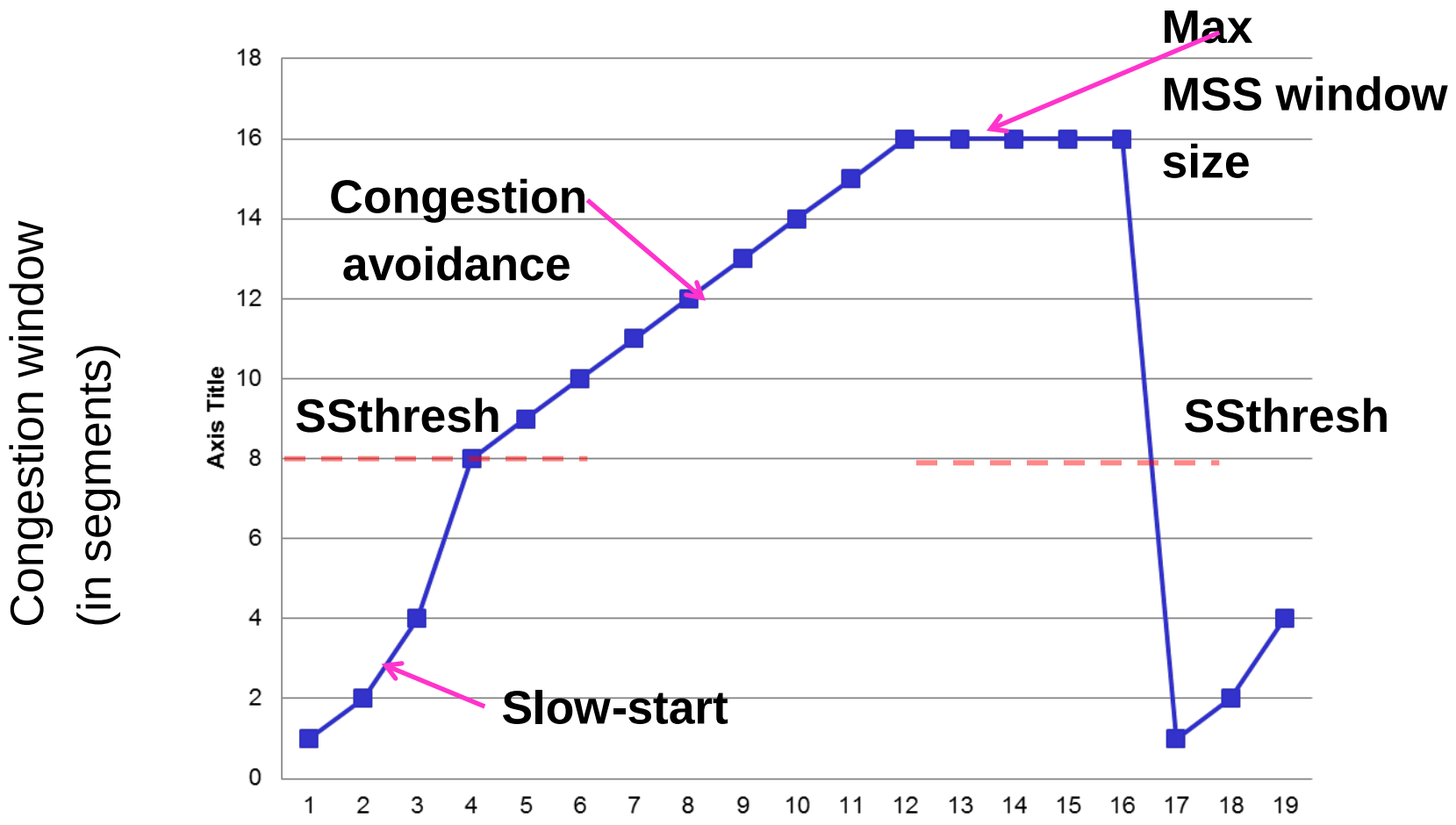
Recovery

ssthresh = ⌊***cwnd***/2⌋
if loss == 3 duplicate ACKs {
  cwnd **= *ssthresh***
  go to collision avoidance }
else
  if loss == timeout {
    cwnd **= 1**
    go to slow start }

Rationale: Network is not too congested if other segments are getting through.



TCP Congestion Control



Assume:

Maximum MSS window size is 16

Calculation of TCP throughput

- **TCP Throughput**
  - **(cwnd * MSS) / RTT = throughput**
    - **Assume ((cwnd *MSS* 8)/ Transmission rate) <<< RTT**
    - **Assume no buffer constraint.**

MSS = 1000 B

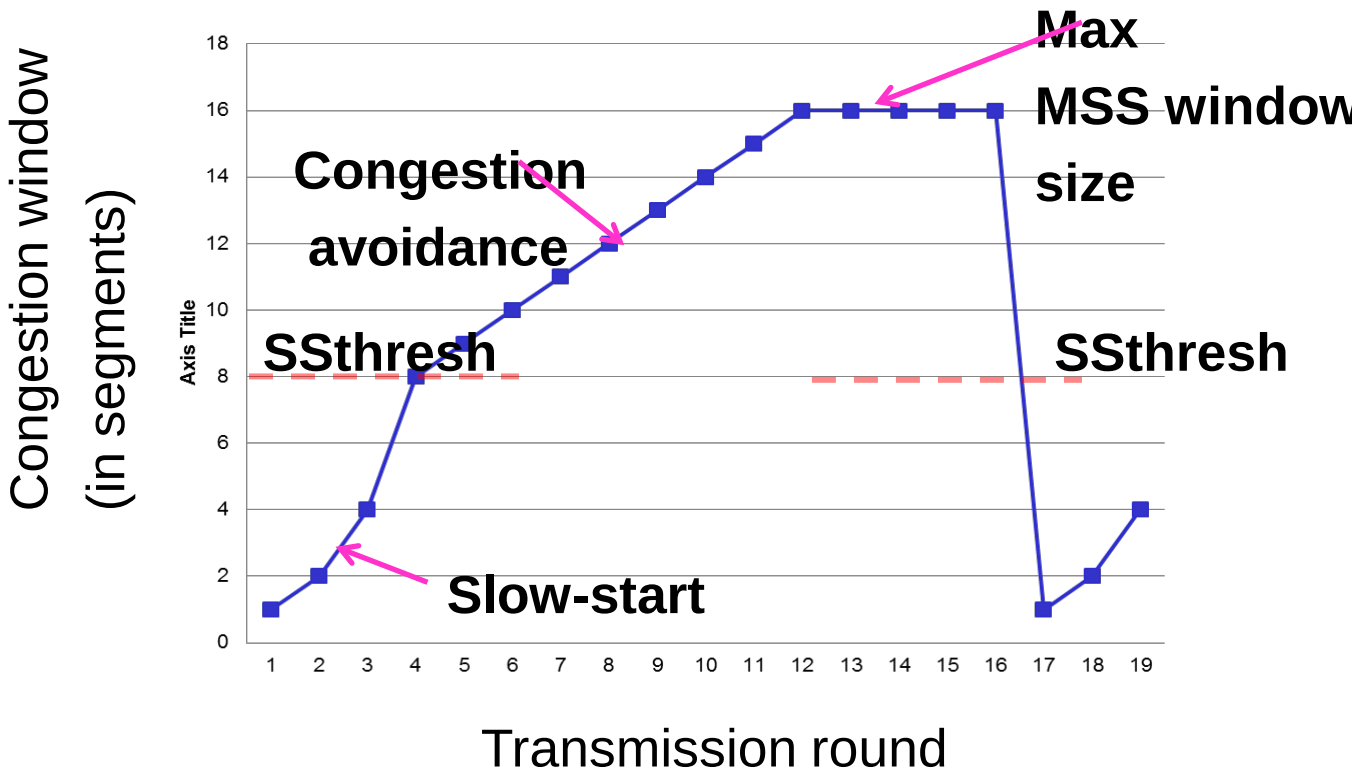
RTT = 1 second

Max throughput

= 16 x MSS/1

= 16KB per sec.

= 128 Kbps



Summary of Transport Layer