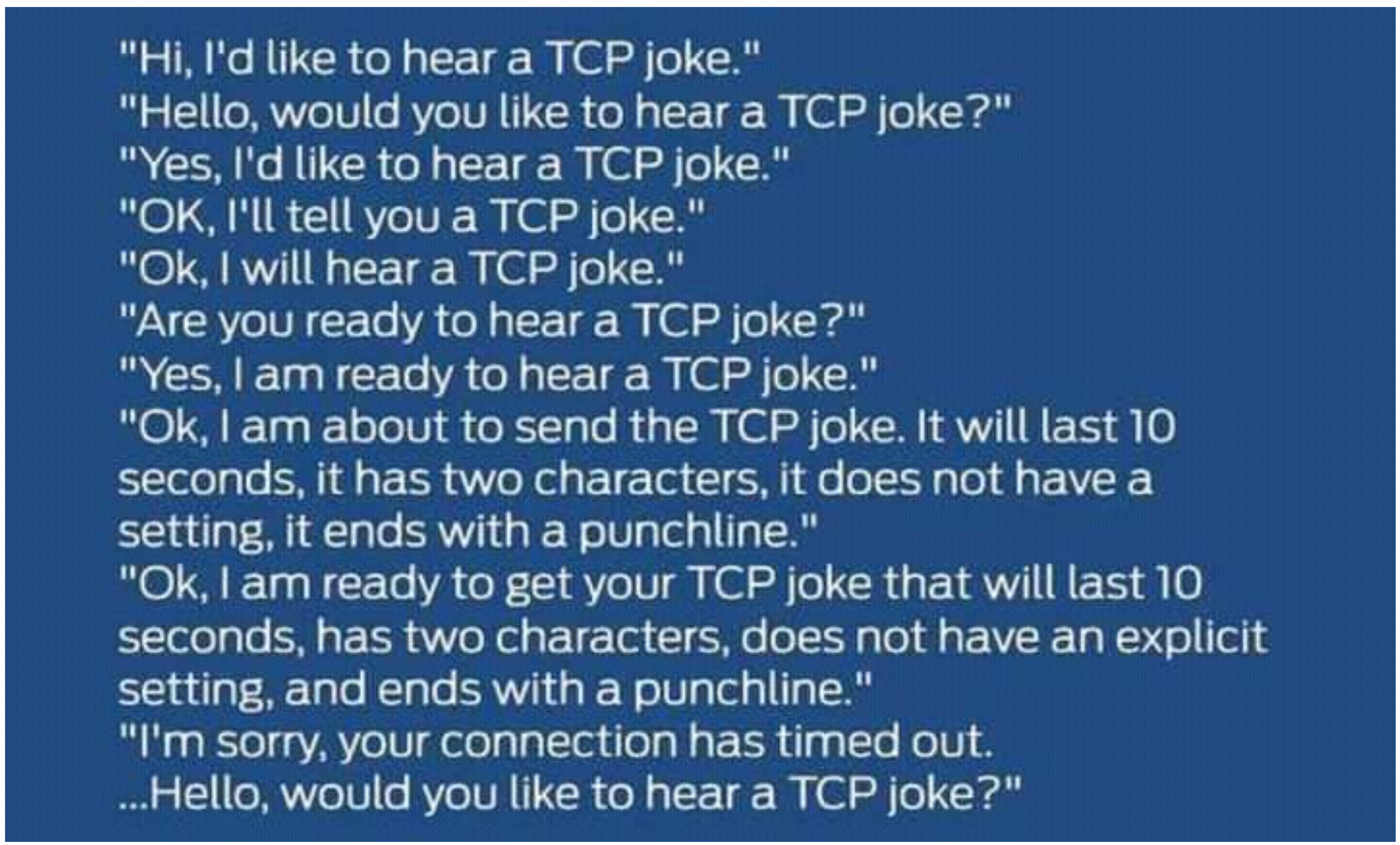
CS118 Discussion 1B, Week 5

Boyan Ding

Outline

- Lecture review: TCP
  - Connection management; flow control; congestion control



Selective repeat/TCP clarification

Since TCP adopts the Selective Repeat, we will be using the selective repeat implementation by TCP throughout the course, that is:

(a) ACK and SEQ numbers are counted in *bytes*

(b) ACK numbers are sent in cumulative fashion

(c) ACK number = next expected byte number

TCP: connection setup

- Connection setup: three-way handshaking
  - 1st round: SYN+initial sequence number (ISN)
  - 2nd round: SYN+SYNACK (client's ISN+1)+server's ISN
  - 3rd round: SYNACK (server's ISN+1) +(optional) data

TCP: connection setup

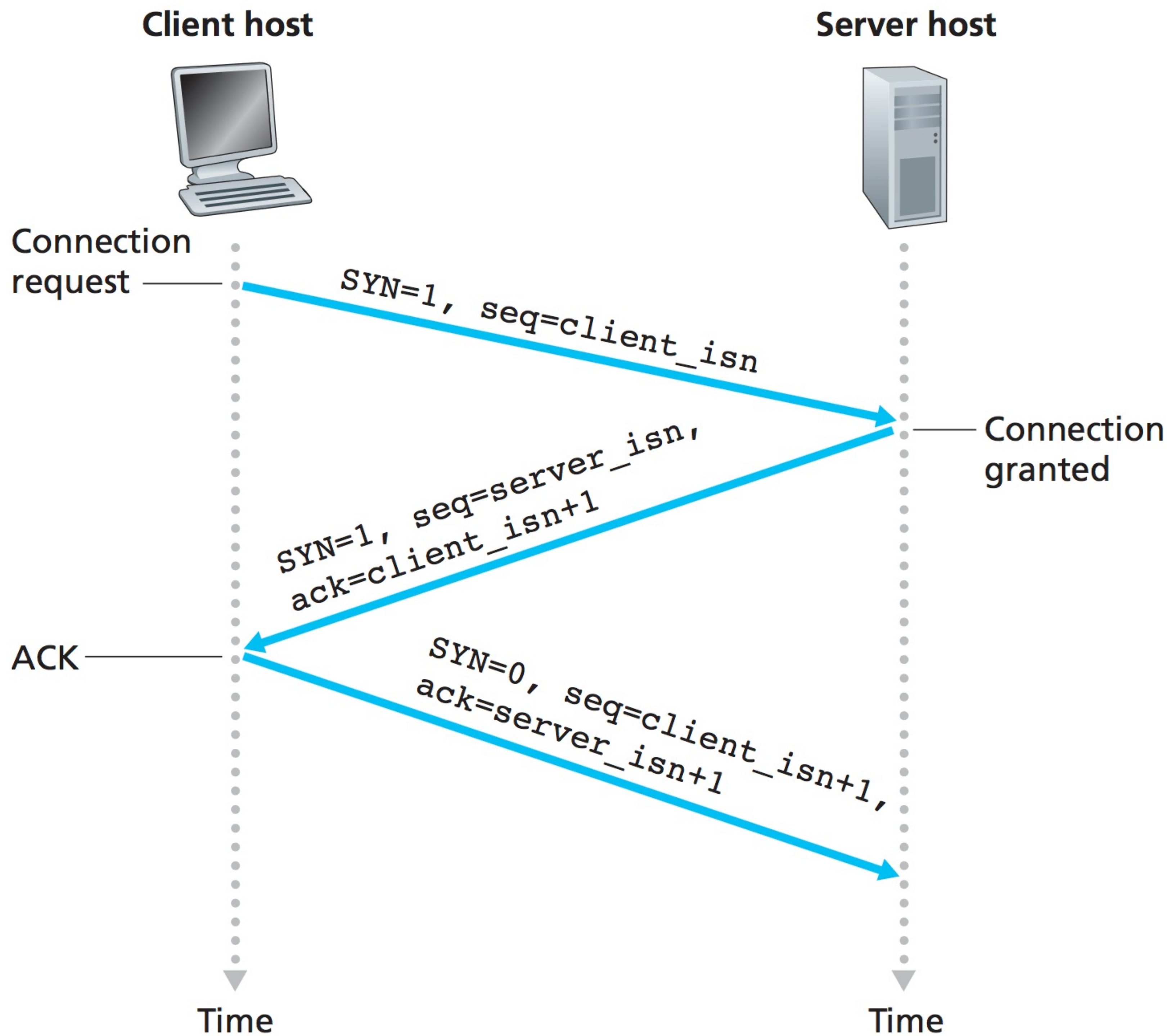
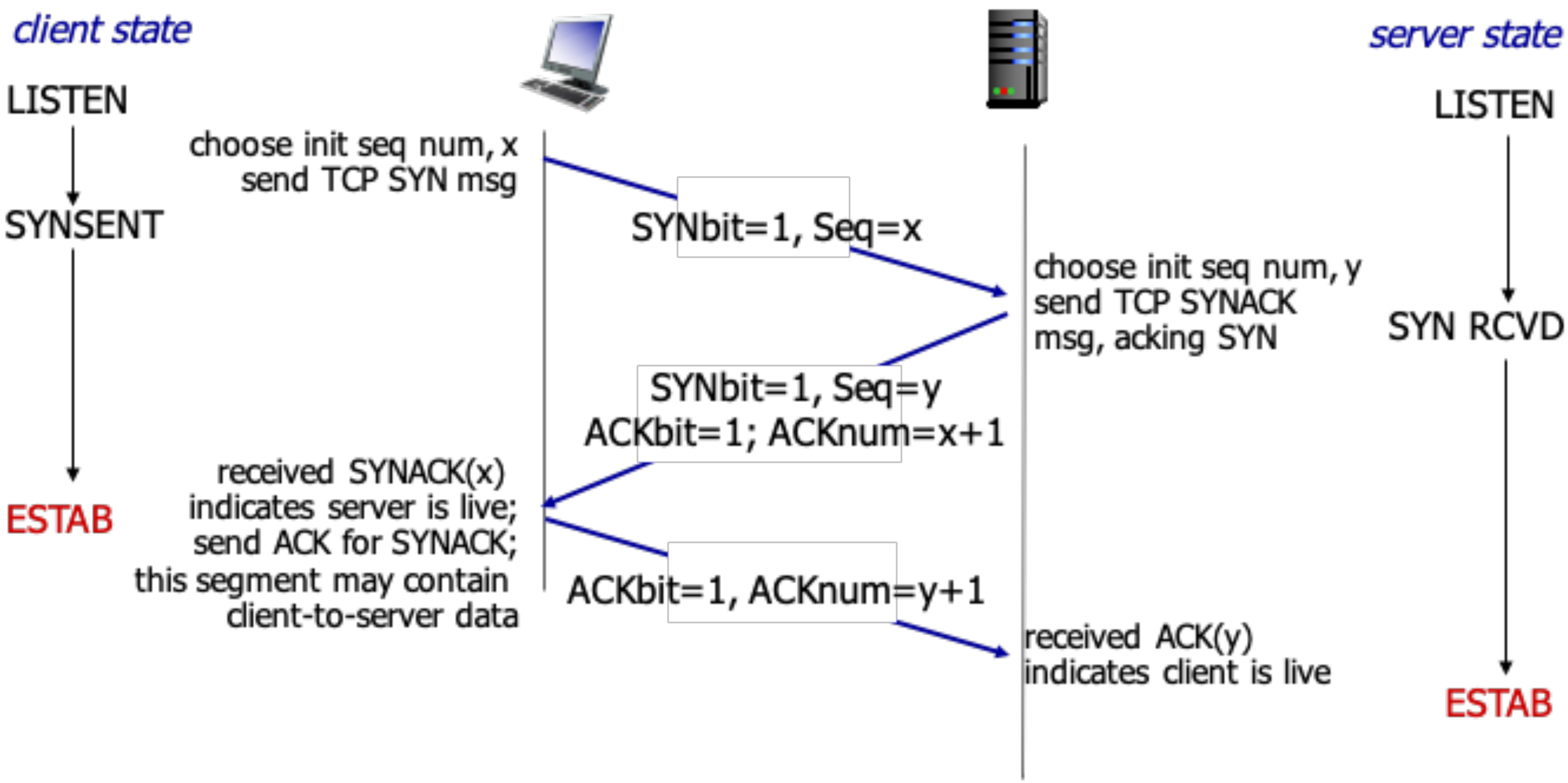


Figure 3.39 ♦ TCP three-way handshake: segment exchange

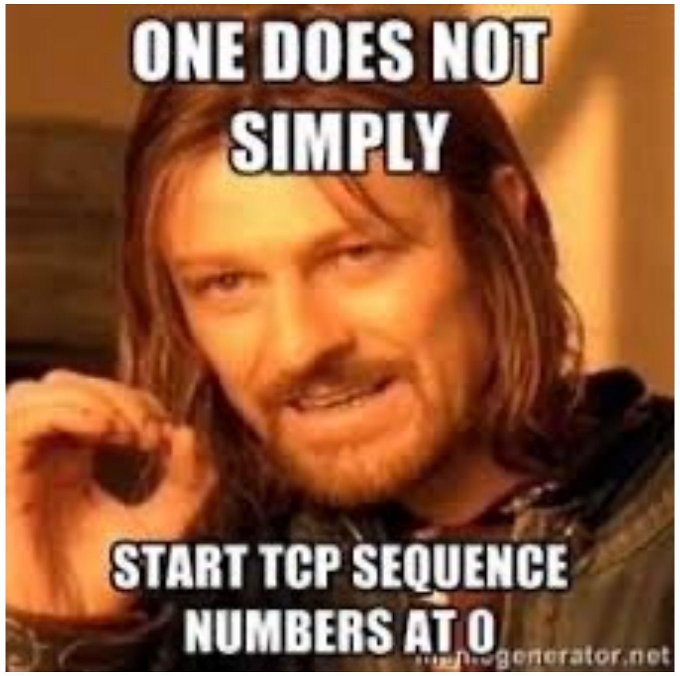
TCP: connection setup (cont'd)



On TCP handshake sequence numbers

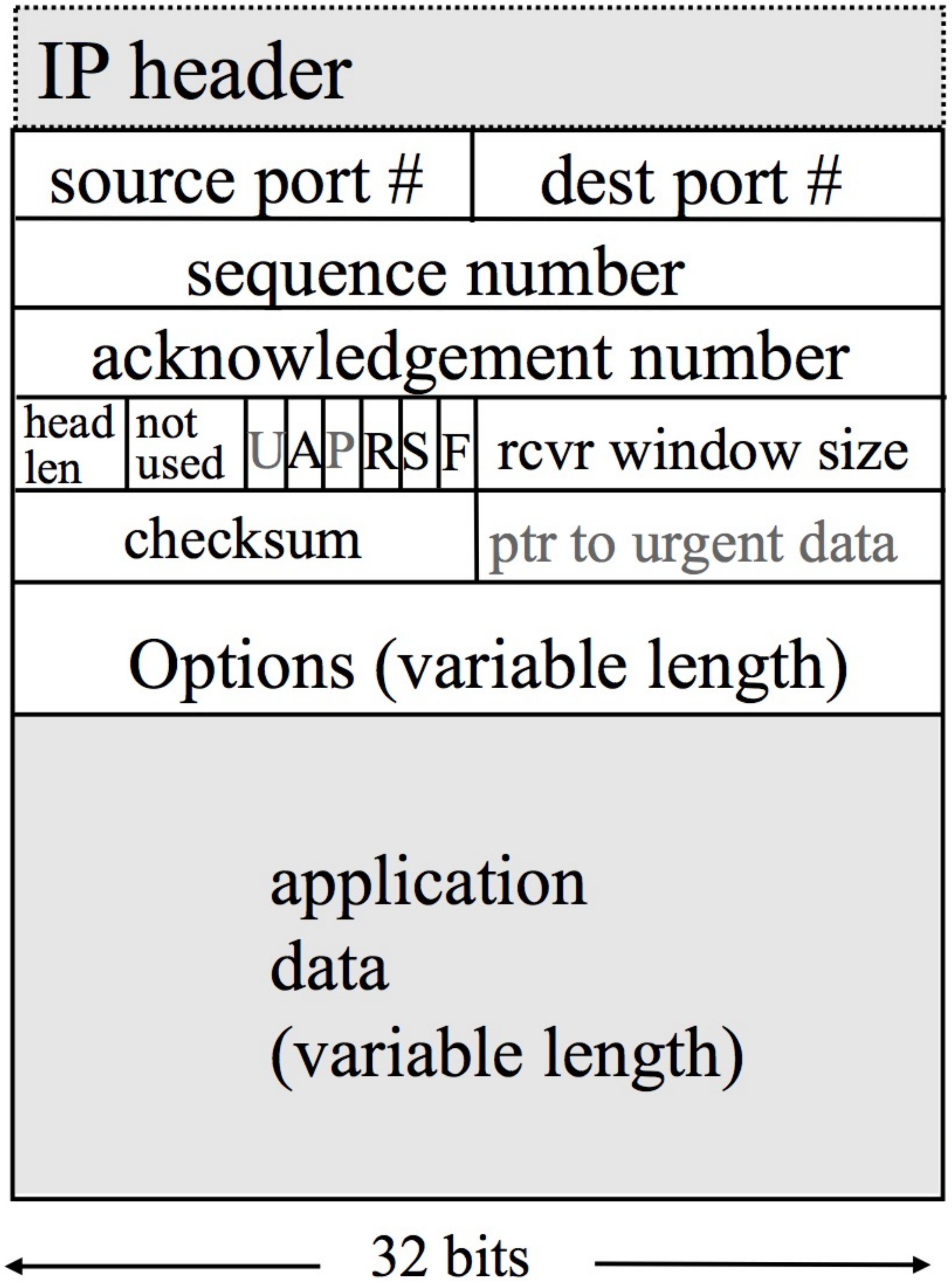
```
1) A --> B  SYN my sequence number is X
2) A <-- B  ACK your sequence number is X
3) A <-- B  SYN my sequence number is Y
4) A --> B  ACK your sequence number is Y
```

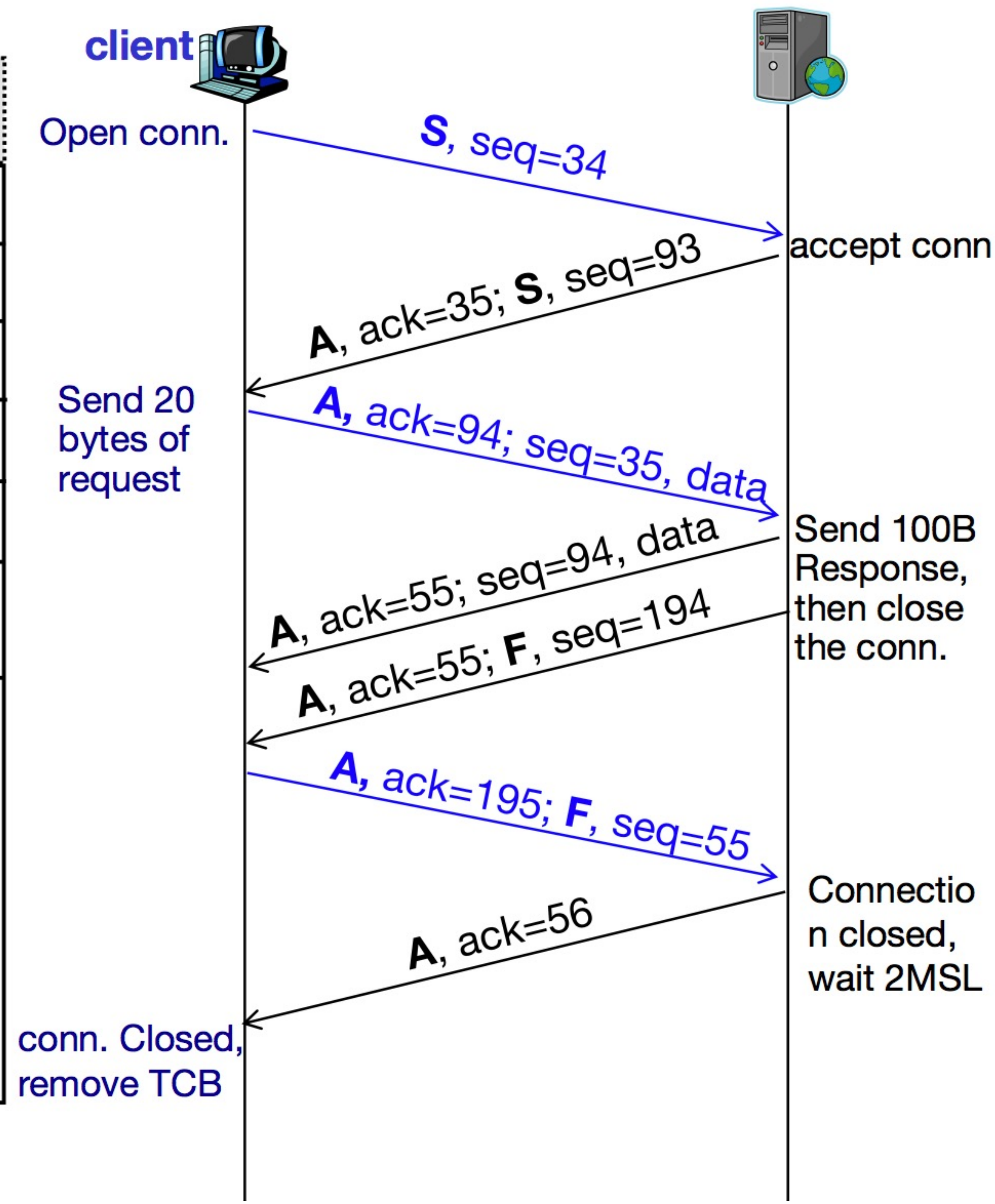
- Because steps 2 and 3 can be combined in a single message this is called the **three way (or three message) handshake**.
- How X and Y are chosen? Not specified; could be random numbers (using clock), as this is more secure. [RFC 793]



https://tools.ietf.org/html/rfc793#section-3.3
https://support.microsoft.com/en-us/help/172983/

An HTTP 1.0 connection example





TCP: connection teardown

- Normal termination
  - allow unilateral close
  - avoid sequence number overlapping
- TCP must continue to receive data even after closing
  - "Half-open/close" connection: cannot close connection immediately: what if a new connection restarts and uses same sequence number?

TCP: connection teardown

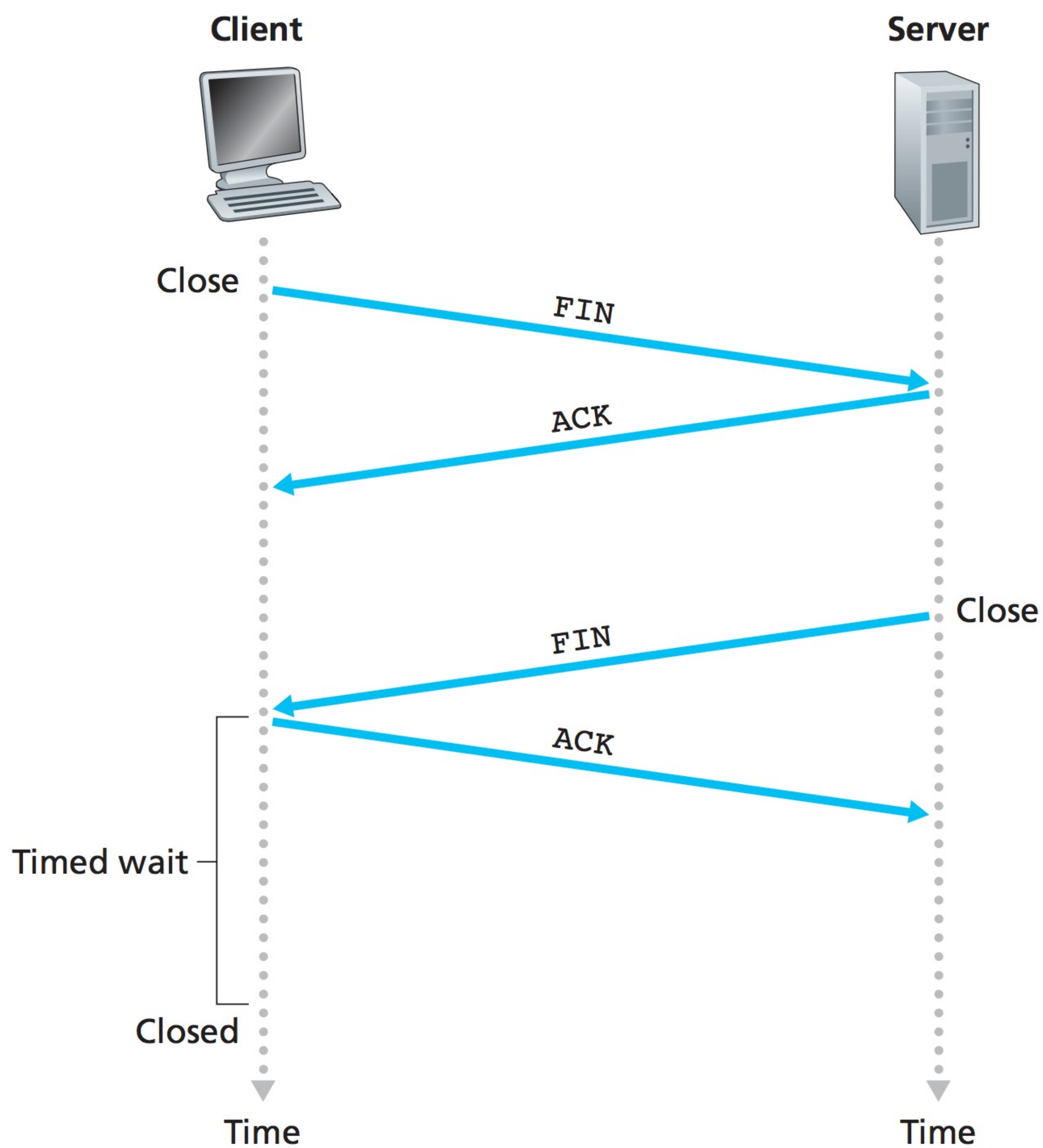


Figure 3.40 ♦ Closing a TCP connection

TCP: flow control

- Limits the rate a sender transfers data
- Avoid having the sender send data too fast
- Avoid exceeding the capacity of the receiver to process data
- Receiver specify the receive window
- The window size announce the number of bytes still free in the receiver buffer

TCP: timeout

- Why is the relationship of TCP timeout with RTT? Why?
  - EstimatedRTT = (1 - α) × EstimatedRTT + α × SampleRTT
  - DevRTT = (1- β) x DevRTT + β × |SampleRTT - Estimated RTT|
  - Retransmission Timer (RTO) = Estimated RTT + 4 × DevRTT

Note: First calculate the Estimated RTT and use the new Estimated RTT for dev RTT calculation.

Question

P31. Suppose that the five measured `SampleRTT` values (see **Section 3.5.3**) are 106 ms, 120 ms, 140 ms, 90 ms, and 115 ms. Compute the `EstimatedRTT` after each of these SampleRTT values is obtained, using a value of α=0.125 and assuming that the value of `EstimatedRTT` was 100 ms just before the first of these five samples were obtained. Compute also the `DevRTT` after each sample is obtained, assuming a value of β=0.25 and assuming the value of `DevRTT` was 5 ms just before the first of these five samples was obtained. Last, compute the TCP `TimeoutInterval` after each of these samples is obtained.

Question

Calculate the EstimatedRTT after obtaining the first sample RTT=106*ms*,

$EstimatedRTT = \alpha * SampleRTT+(1- \alpha) * EstimatedRTT$

$EstimatedRTT = 0.125 * 106 + (1-0.125) * 100$

$= 0.125 * 106 + 0.875 * 100$

$= 13.25 + 87.5$

$= 100.75 ms$

Calculate the DevRTT after obtaining the first sample RTT:

$DevRTT = \beta * | SampleRTT - EstimatedRTT| + (1- \beta) * DevRTT$

$= 0.25 * |106-100.75| + (1-0.25) * 5$

$= 0.25 * 5.25 + 0.75 * 5$

$= 1.3125 + 3.75$

$= 5.0625 ms$

Calculate the Timeout Interval after obtaining the first sample RTT:

$TimeoutInterval = EstimatedRTT + 4 * DevRTT$

$= 100.75 + 4 * 5.0625$

$= 121 ms$

TCP: timeout

- Karn's algorithm — in case of retransmission
  - do not take the RTT sample (i.e. do not update SRTT or DevRTT)
  - double the retransmission timer value (RTO) after each timeout
  - Take RTT measure again upon next data transmission (that did not get retransmitted)

TCP: congestion control

- Why Congestion Control
  - Oct. 1986, Internet had its first congestion collapse (LBL to UC Berkeley)
  - 400 yards, 3 hops, 32 kbps
  - throughput dropped by a factor of 1000 to 40 bps
- 1988, Van Jacobson proposed TCP congestion control
  - Window based with ACK mechanism
  - End-to-end

TCP: congestion control — window-based

- Limit number of packets in network to window size W
  - Source rate allowed (bps) = W×Message Size/RTT
  - Too small W?
  - Too large W?

TCP: congestion control — effects

- Packet loss
- Retransmission and reduced throughput
- Congestion may continue after the overload

TCP: congestion control — basics

- Goals: achieve high utilization without congestion or unfair sharing
- Receiver control (**`rwnd`**): set by receiver to avoid overloading receiver buffer
- Network control (**`cwnd`**): set by sender to avoid overloading network
  - `W = min(cwnd, rwnd)`
- Congestion window **`cwnd`** usually is the bottleneck

TCP: congestion control — main parts

- Slow start
- Congestion Avoidance
- Fast retransmit
- Fast recovery

https://tools.ietf.org/html/rfc5681

TCP: congestion control — slow start

- Start with cwnd = 1 (MSS: max. segment size; abstract as pkt)
- Exponential growth
  - each RTT:
    - `cwnd ← 2 × cwnd`
  - equivalently, each ACK:
    - `cwnd ← cwnd + 1 (MSS)`
- Enter Congestion Avoidance when **`cwnd > ssthresh`**

TCP: congestion control — congestion avoidance

- Start with `cwnd >= ssthresh`
- Linear growth
  - each RTT:
    - `cwnd ← cwnd + 1 (MSS)`
  - equivalently, each ACK:
    - `cwnd ← cwnd + 1/cwnd (MSS^2/cwnd)`

TCP: congestion control — packet loss

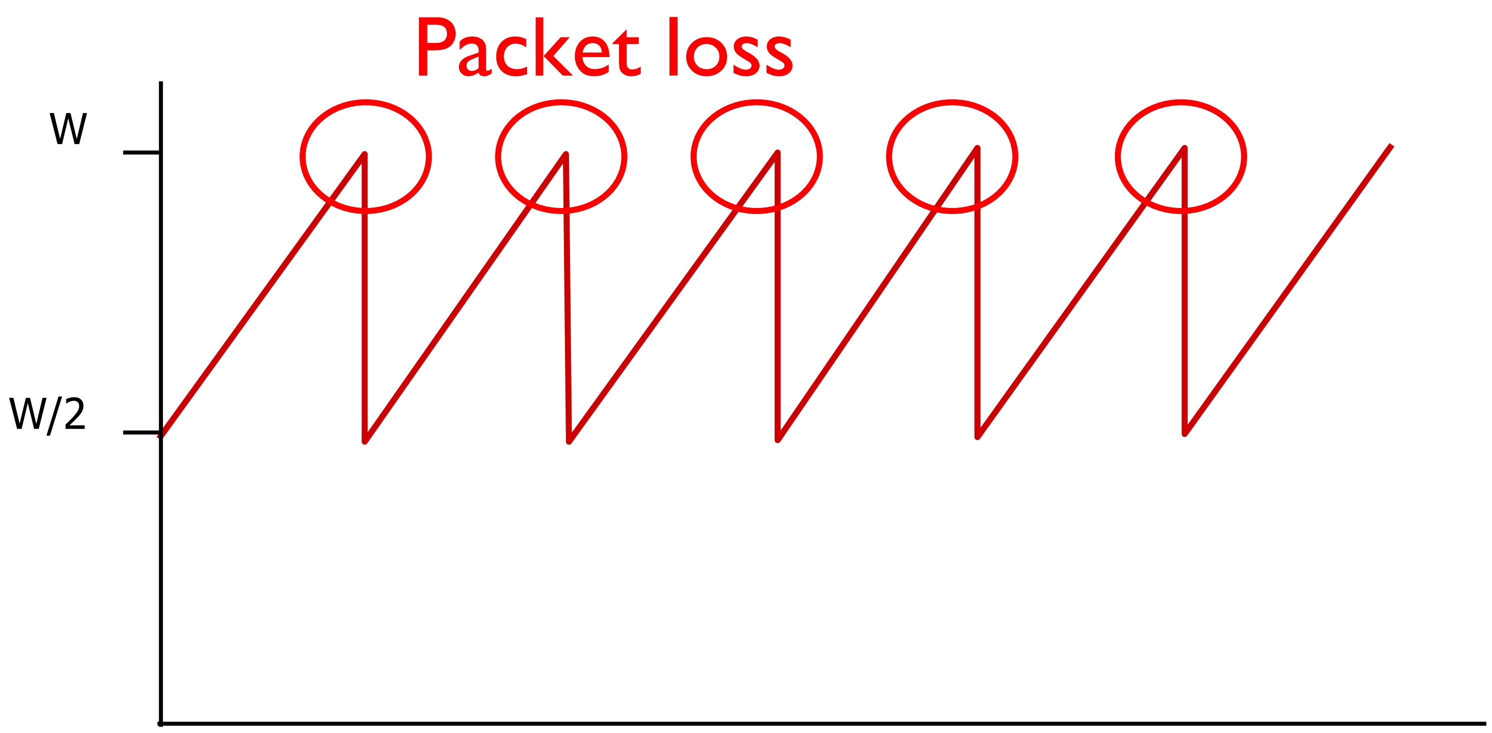
- Assumption: loss indicates congestion
- Packet loss detected by
    - Retransmission Timer Outs (RTO timer)
    - Duplicate ACKs (three)
        - ignore the 1st or 2nd duplicate ACK

TCP: congestion control — fast retx/recovery

- Upon 3rd duplicate ACK:
    - set `ssthresh ← cwnd/2`
    - set `cwnd ← ssthresh + 3 (MSS)`
    - upon additional dup ACK: grow `cwnd` linearly
    - New ACK: `cwnd ← ssthresh`
- Time Out
    - set `ssthresh ← cwnd/2`
    - set `cwnd ← 1 (MSS)`
    - enter slow start

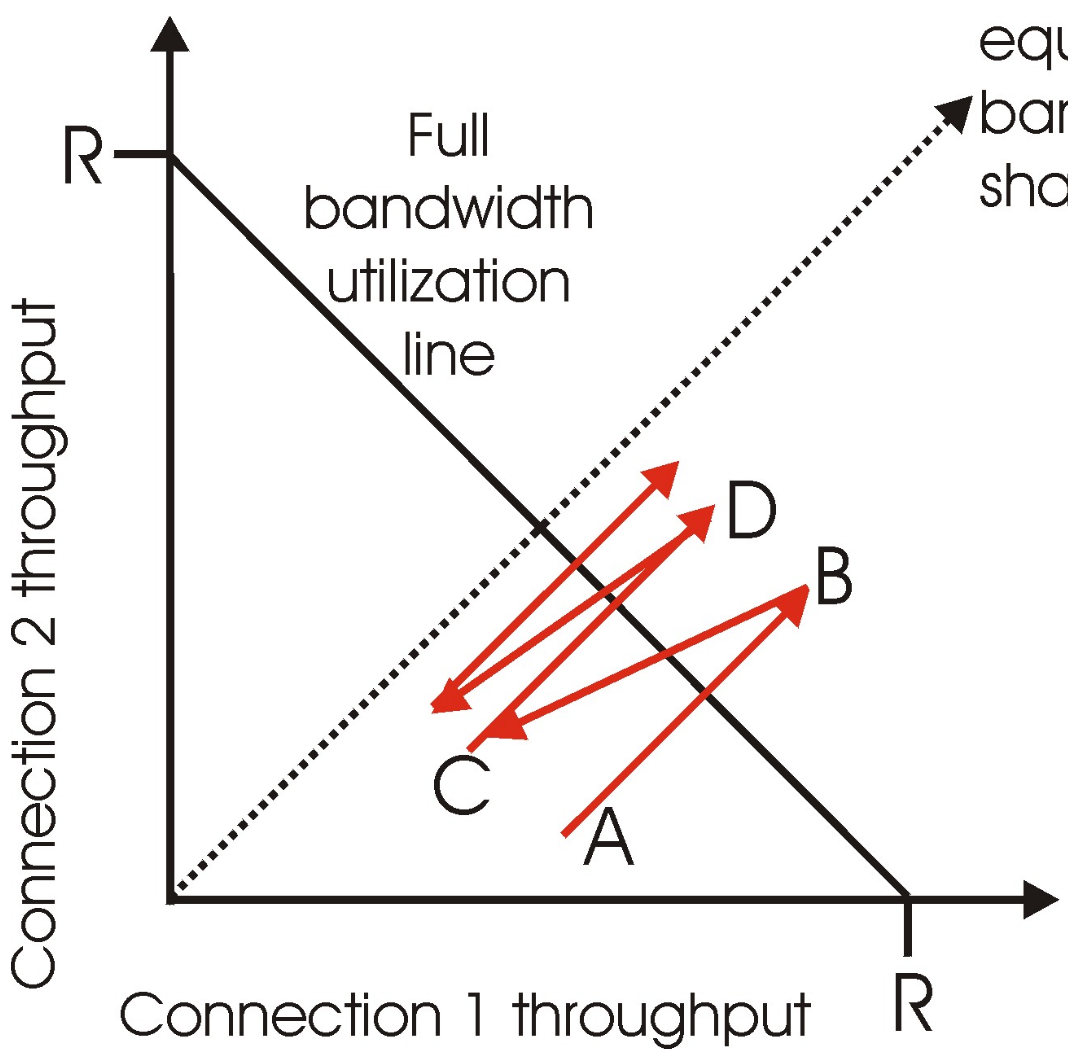
Macroscopic view of TCP throughput

- Additive Increase Multiple Decrease (AIMD)
- Saw tooth behavior: probing for bandwidth



AIMD for fairness

- AIMD-based congestion window adaptation
  - Fairness v.s. efficiency



A->B: Additive increase

B->C: Packet loss, shrink cwnd by half

(Multiple decrease)

C-D: Additive increase

What if AIAD?

TCP: congestion control — summary

- Slow start for fast convergence
- Congestion is indicated by packet loss
  - Timeout or duplicated loss
  - Congestion control is coupled with reliable transfer
- Fast retransmission/recovery based on duplicated ACK
  - `cwnd` $\leftarrow$ `cwnd/2 + 3MSS`

```
The lost segment starting at SND.UNA MUST be retransmitted and
cwnd set to ssthresh plus 3*SMSS.  This artificially "inflates"
the congestion window by the number of segments (three) that have
left the network and which the receiver has buffered.
```

—RFC 5681

Let's try it!

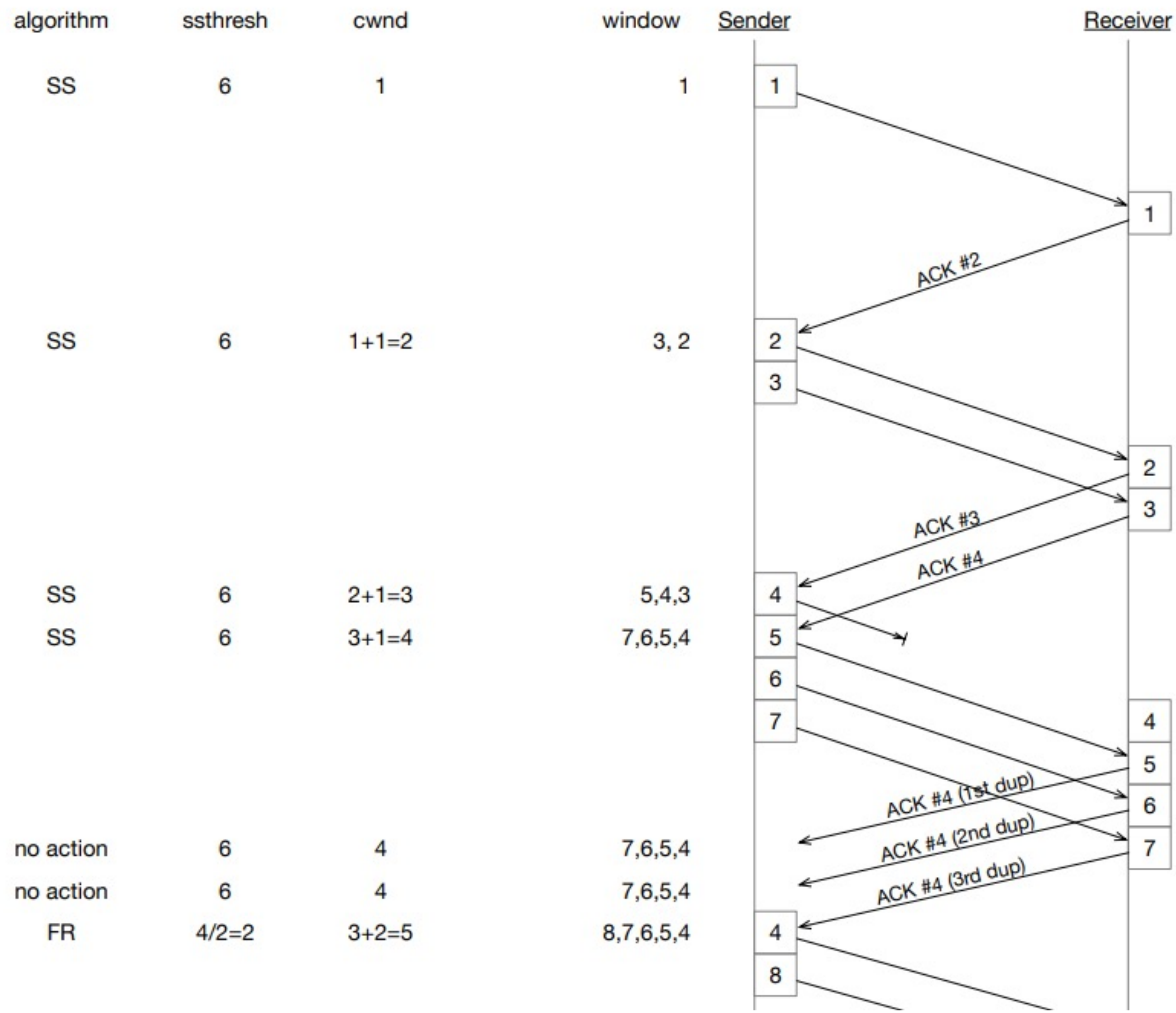
Consider the evolution of a TCP connection with the following characteristics. Assume that all the following algorithms are implemented in TCP congestion control: slow start, congestions avoidance, fast retransmit and fast recovery, and retransmission upon timeout.

- The receiver acknowledges every segment, and the sender always has data available for transmission.
- Initially `ssthresh` at the sender is set to 6. Assume `cwnd` and `ssthresh` are measured in segments, and the transmission time for each segment is negligible. Retransmission timeout (RTO) is initially set to 500ms at the sender and is **unchanged** during the connection lifetime. The RTT is 100ms for all transmissions.
- The connection starts to transmit data at time t = 0, and the initial sequence number starts from 1. **Segment with sequence number 4 is lost once**. No other segments are lost.

How long does it take, in milliseconds, for the sender to receive the ACK for the segment with the sequence number 12? show your intermediate steps or your diagram.

If **`ssthresh`** equals to **`cwnd`**, use the slow start algorithm in your calculation.

Solution

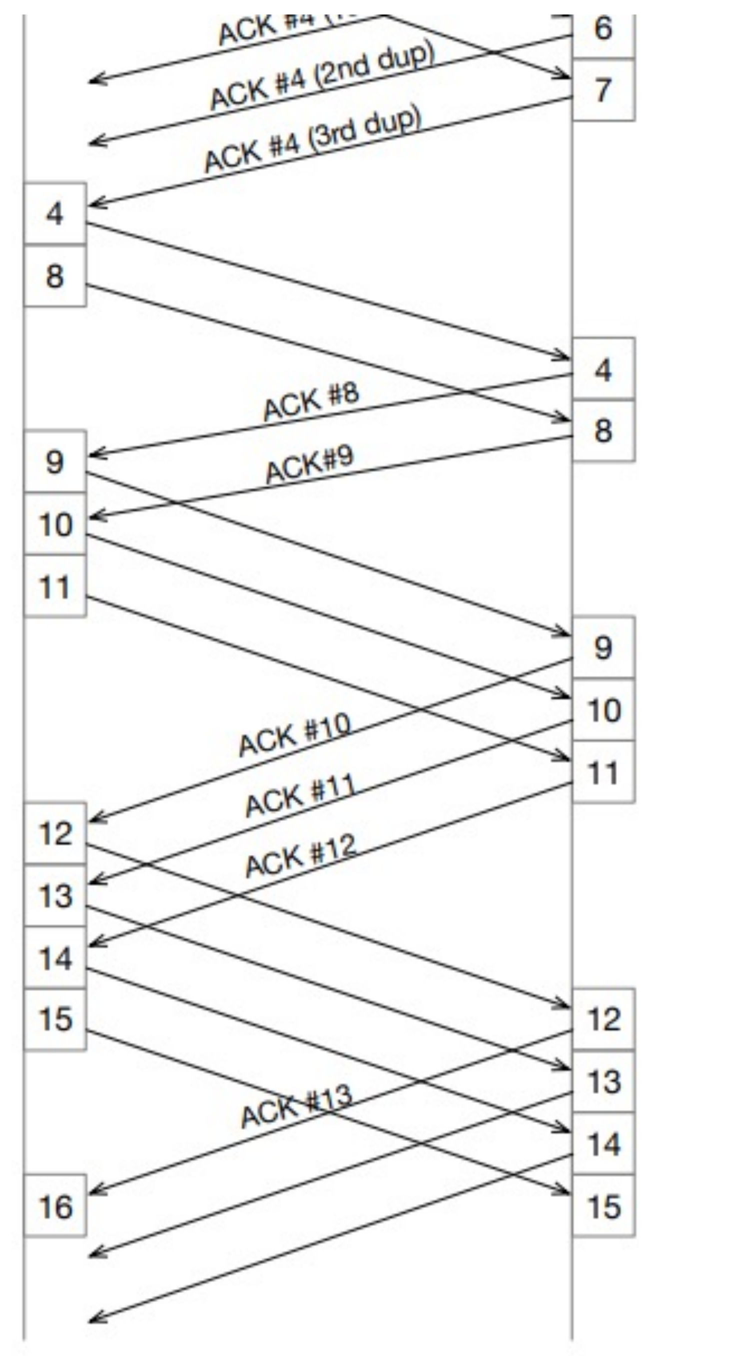


algorithm
ssthresh
cwnd
window
Sender
Receiver
SS
6
1
1
SS
6
1+1=2
3, 2
SS
6
2+1=3
5,4,3
SS
6
3+1=4
7,6,5,4
no action
6
4
7,6,5,4
no action
6
4
7,6,5,4
FR
4/2=2
3+2=5
8,7,6,5,4
1
2
3
4
5
6
7
4
8
ACK #2
ACK #3
ACK #4
ACK #4 (1st dup)
ACK #4 (2nd dup)
ACK #4 (3rd dup)

Solution

- If `ssthresh` equals to `cwnd`, use the congestion avoidance algorithm in your calculation.

no action	6	4	7,6,5,4
no action	6	4	7,6,5,4
FR	4/2=2	3+2=5	8,7,6,5,4
FR->CA	2	2+1/2=2	9,8
CA	2	2 1/2+1/2=3	11,10,9
CA	2	3+1/3=3	12,11,10
CA	2	3 1/3+1/3=3	13,12,11
CA	2	3 2/3+1/3=4	15,14,13,12
CA	2	4+1/4=4	16,15,14,13

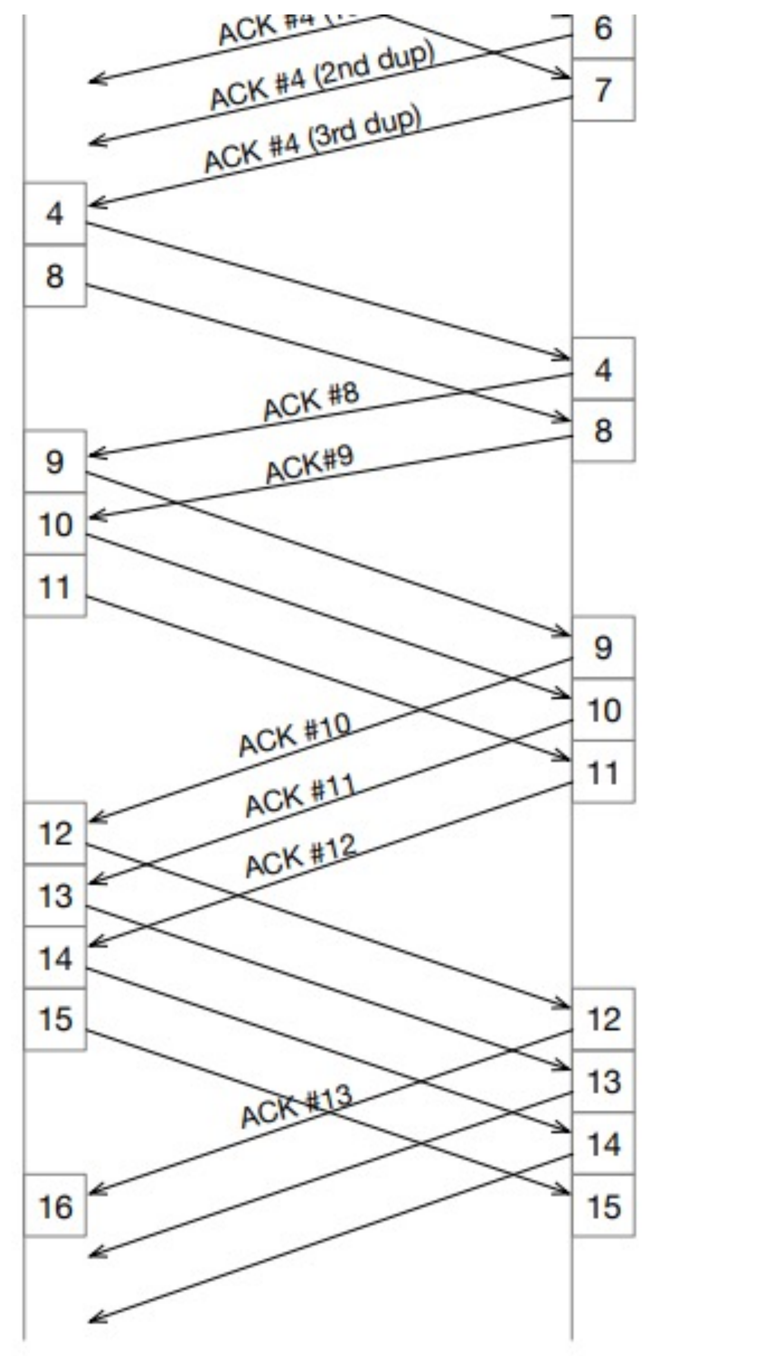


6 RTTs

Solution

- If **`ssthresh`** equals to **`cwnd`**, use the slow start algorithm in your calculation.

no action	6	4	7,6,5,4
no action	6	4	7,6,5,4
FR	4/2=2	3+2=5	8,7,6,5,4
FR->SS	2	2+1=3	10,9,8
CA	2	[3+1/3]=3	11,10,9
CA	2	[3 1/3 + 1/3] = 3	12,11,10
CA	2	[3 2/3 + 1/3] = 4	14,13,12,11
CA	2	[4 + 1/4] = 4	15,14,13,12
CA	2	[4 1/4 + 1/4] = 4	16,15,14,13



6 RTTs