DIPARTIMENTO DI INFORMATICA
Via Salaria 113 - 00198, Roma

SAPIENZA UNIVERSITÀ DI ROMA

Chapter 3
Transport Layer

Reti di Elaboratori

Corso di Laurea in Informatica

Università degli Studi di Roma "La Sapienza"

Prof.ssa Chiara Petrioli

Parte di queste slide sono state prese dal materiale associato al libro *Computer Networking: A Top Down Approach* , 5th edition.

Thanks also to Antonio Capone, Politecnico di Milano, Giuseppe Bianchi and Francesco LoPresti, Un. di Roma Tor Vergata

TCP data transfer management

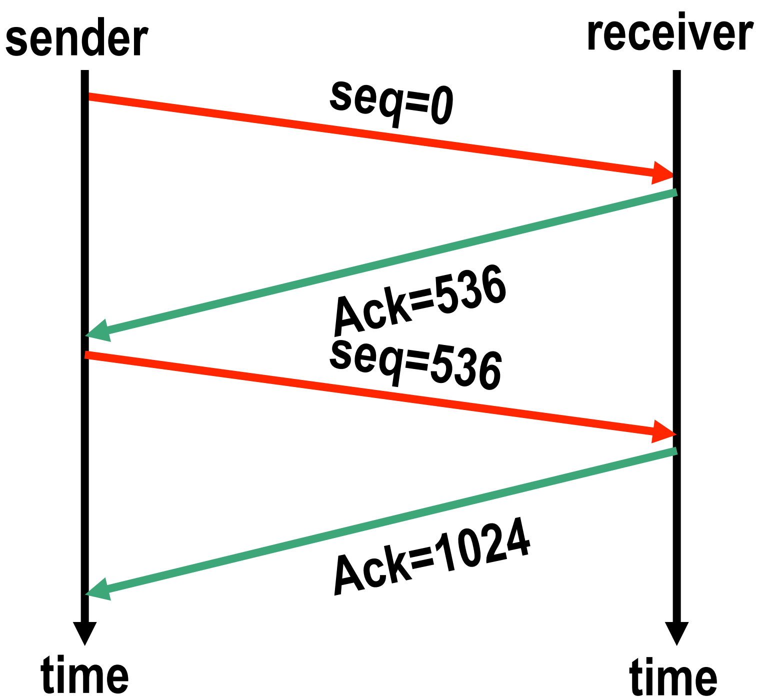
- Full duplex connection
  - data flows in both directions, independently
  - To the application program these appear as two unrelated data streams
- each end point maintains a sequence number
  - Independent sequence numbers at both ends
  - Measured in bytes
- acks often carried on top of reverse flow data segments (piggybacking)
  - But ack packets alone are possible

Byte-oriented

Example: 1 Kbyte message – 1024 bytes

0	1	…	…	100	…	…	535	…	…	…	1023

Example: segment size = 536 bytes → 2 segments: 0-535; 536-1023

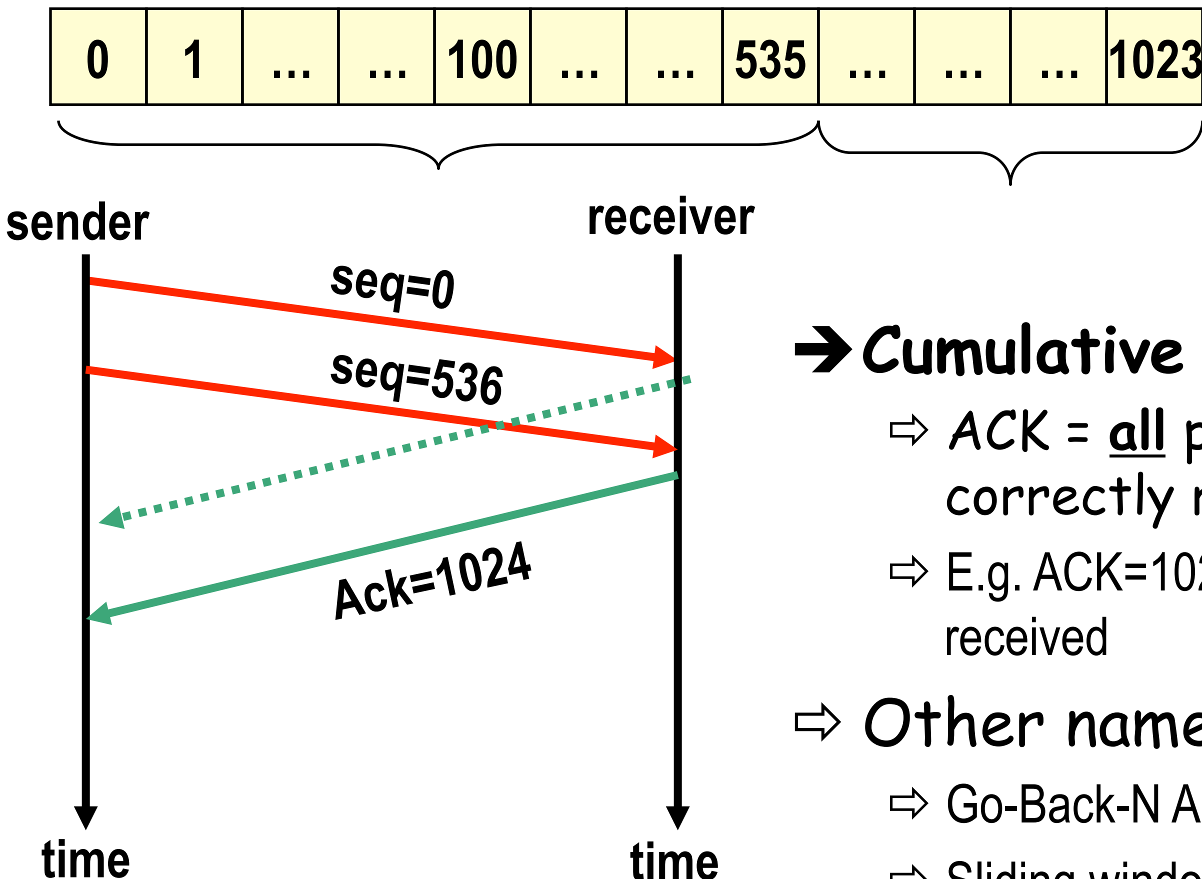


➔ **No explicit segment size indication**

- Seq = first byte number
- Returning Ack = last byte number + 1
- Segment size = Ack-seq#

Pipelining - cumulative ack

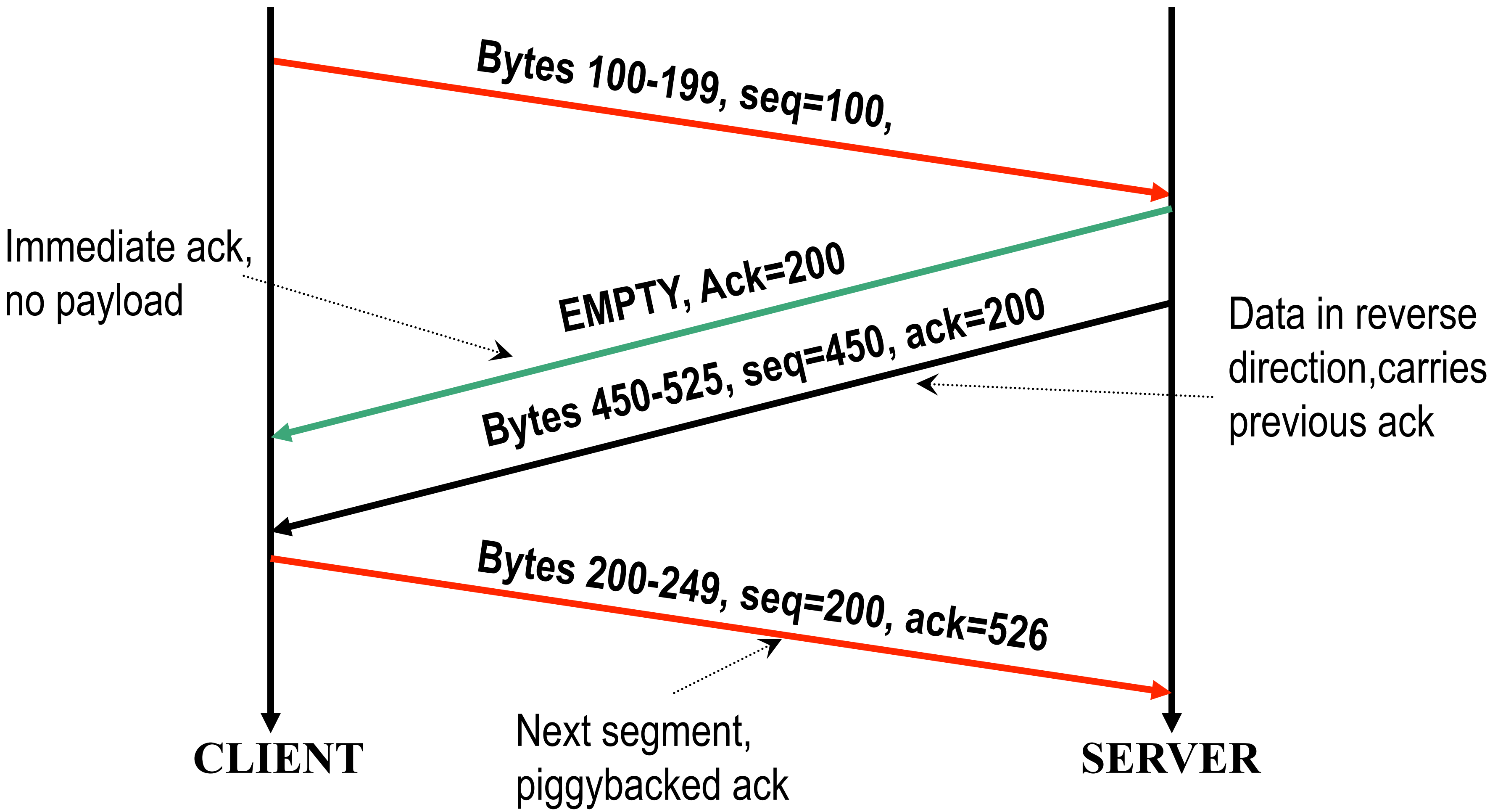
Example: 1024 bytes msg; seg_size = 536 bytes → 2 segments: 0-535; 536-1023



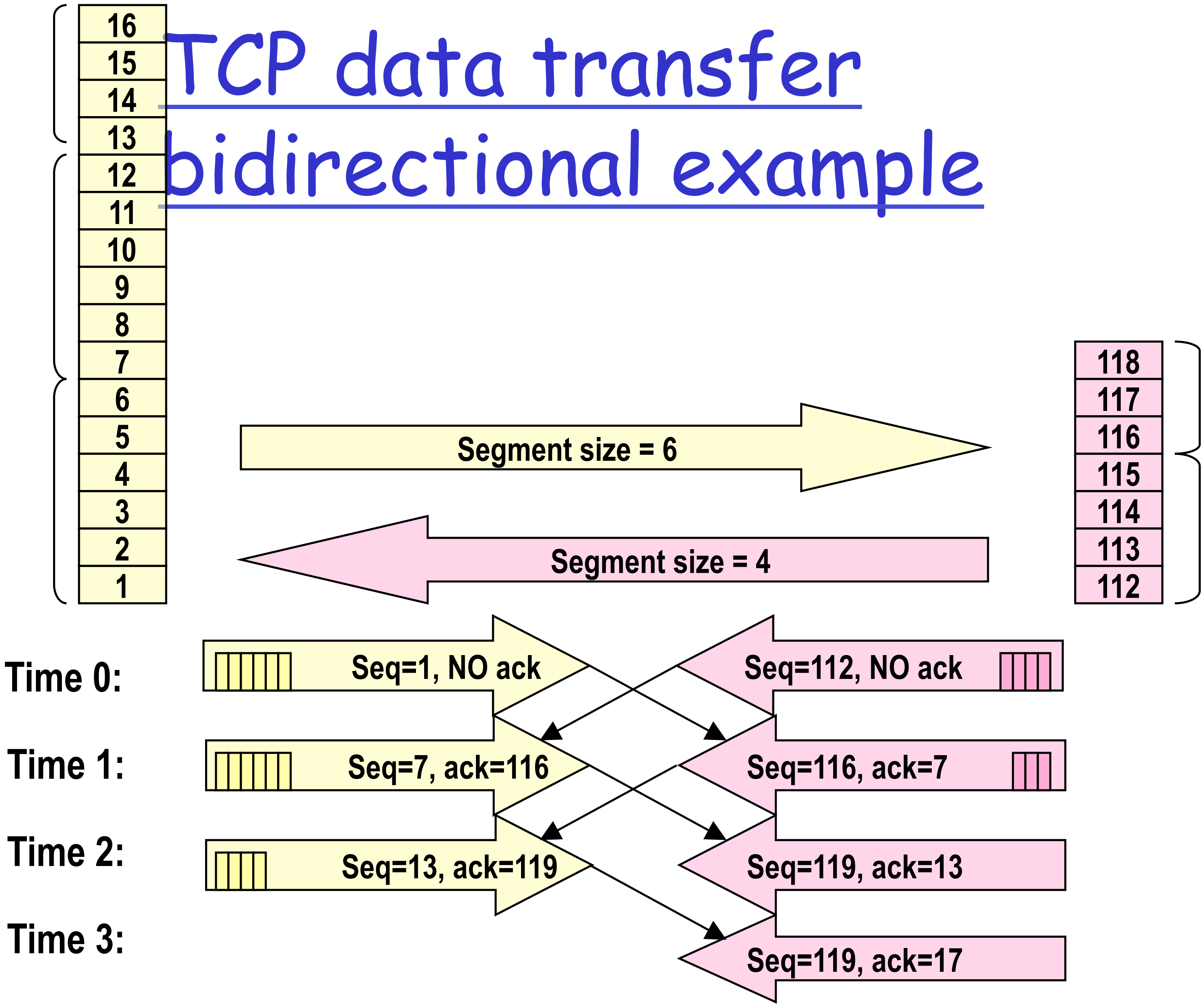
- **Cumulative ack**
  - ACK = <u>all</u> previous bytes correctly received!
  - E.g. ACK=1024: all bytes 0-1023 received
- Other names of pipelining:
  - Go-Back-N ARQ mechanisms
  - Sliding window mechanisms

Why pipelining? Dramatic improvement in efficiency!

Multiple acks; Piggybacking



TCP data transfer bidirectional example



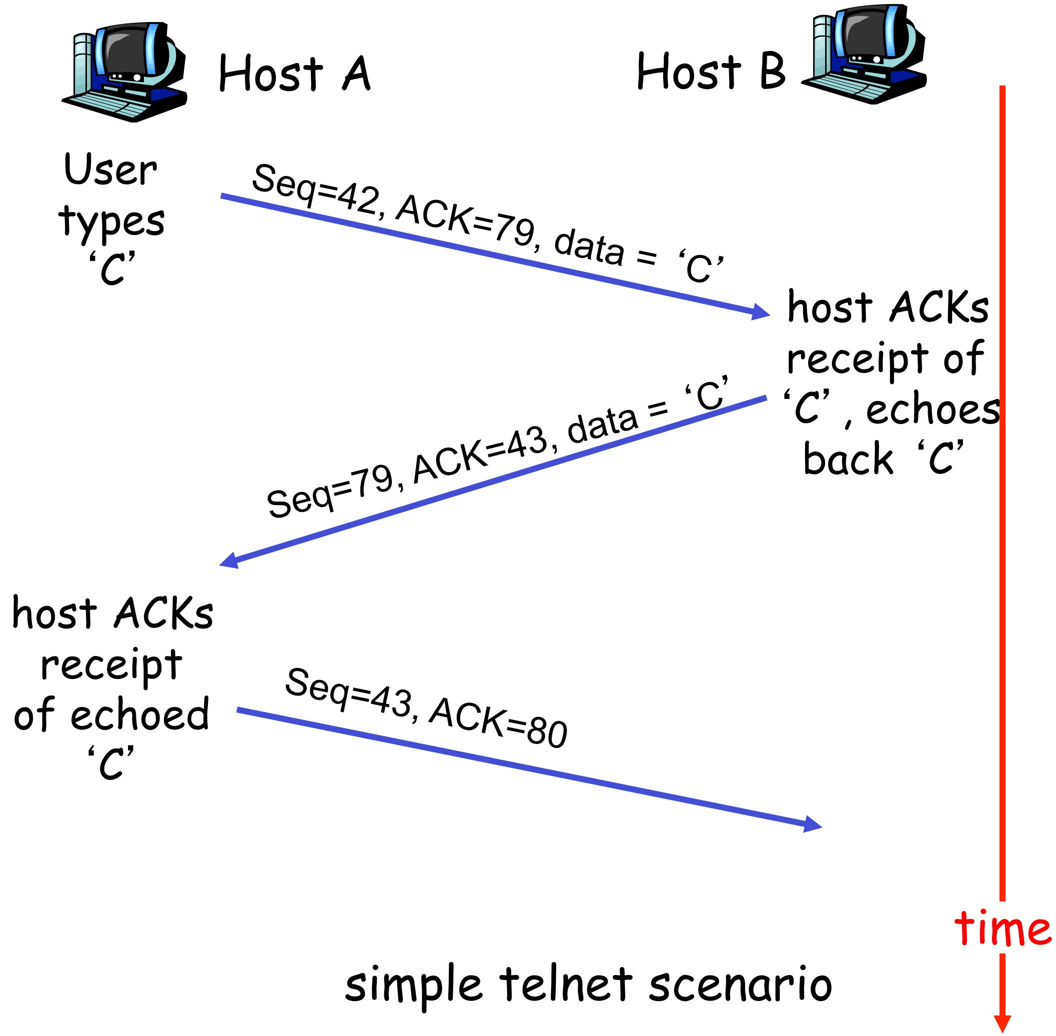
TCP seq. #'s and ACKs

TCP Solution: Go Back N like

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value? (not trivial, highly varying, it is a RTT over a network path)

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions

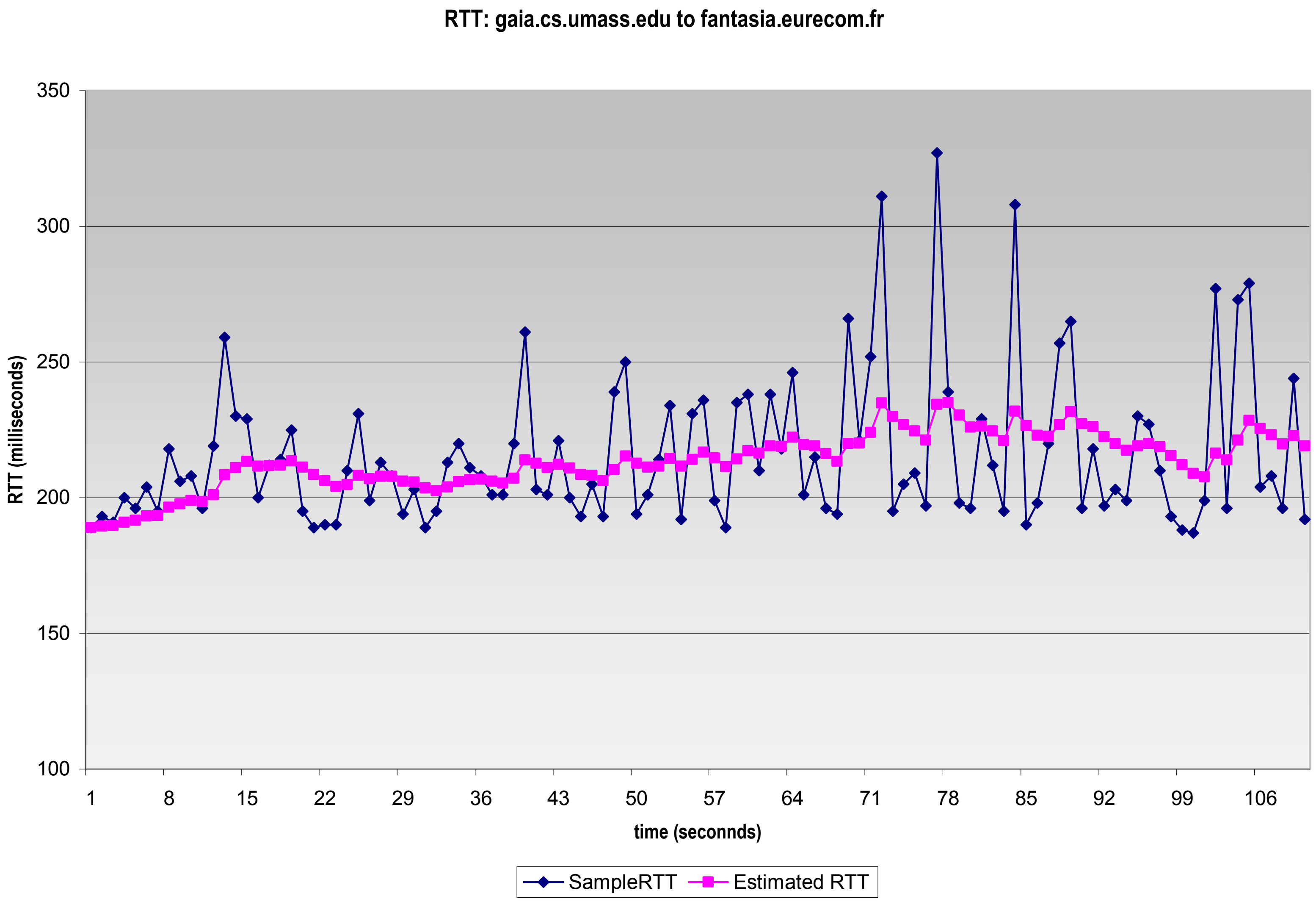
  Why??
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

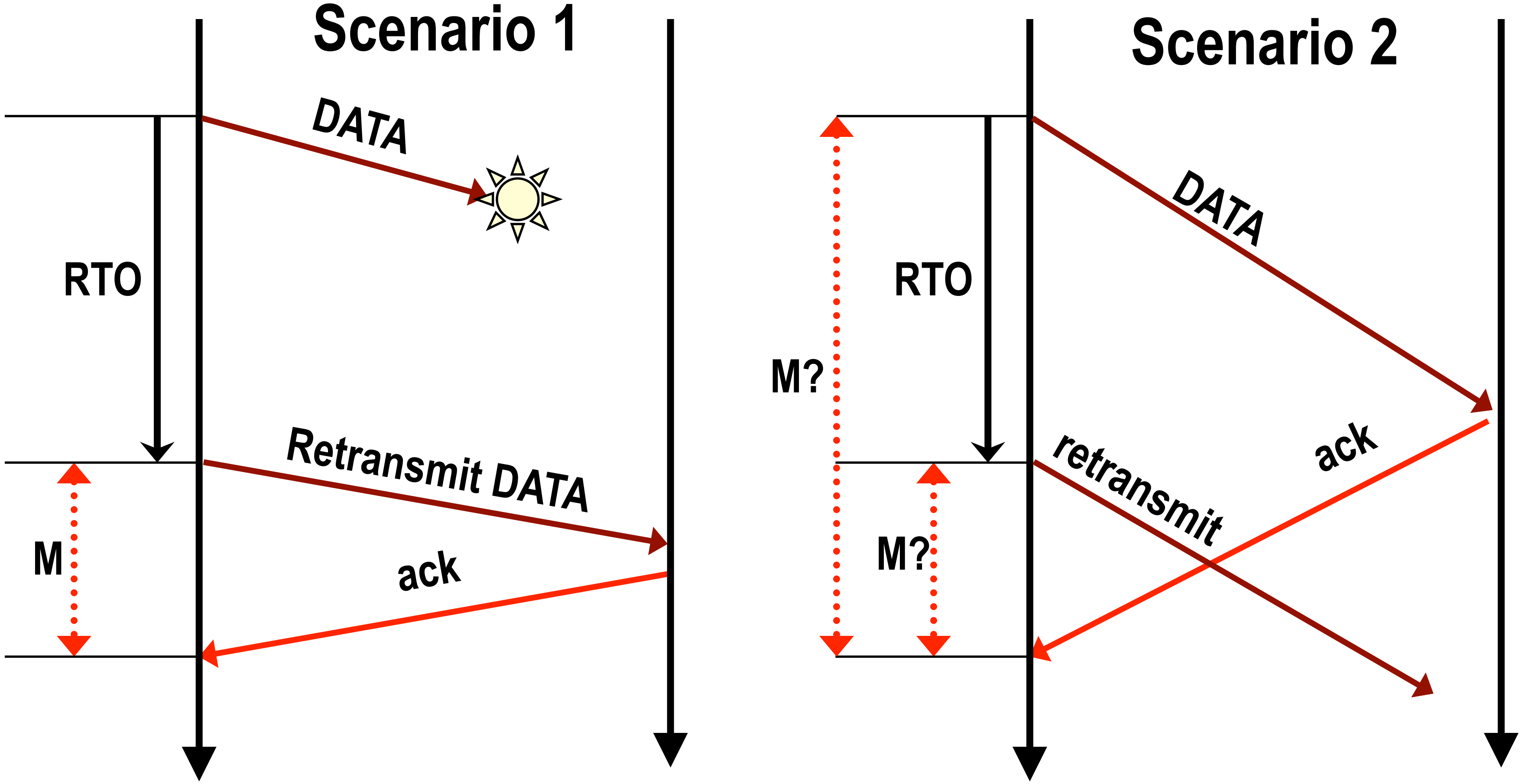
```
DevRTT = (1-β)*DevRTT +
               β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Guessing right?

Karn's problem



How can we distinguish among an ACK to the original segment and to a duplicate?

Solution to Karn's problem

- **Very simple: DO NOT update RTT when a segment has been retransmitted because of RTO expiration!**
- **Instead, use Exponential backoff**
  - *double RTO for every subsequent expiration of same segment*
    - When at 64 secs, stay
    - persist up to 9 minutes, then reset

TCP reliable data transfer (more in detail)

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP sender (simplified)

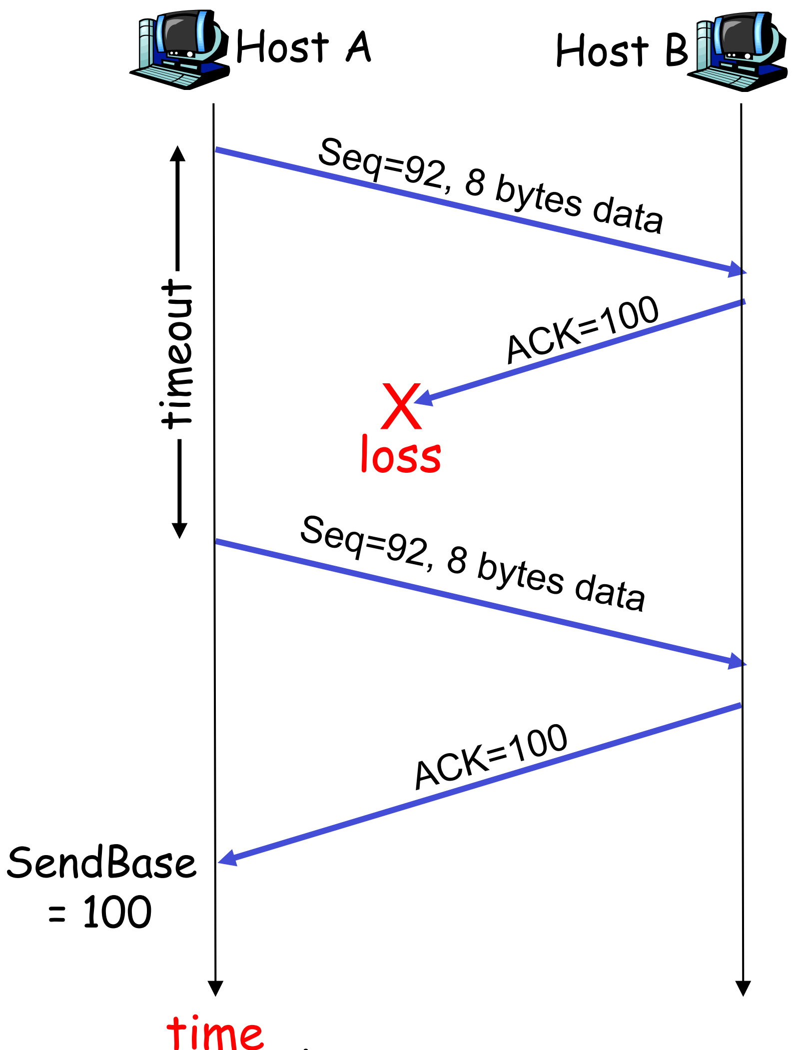
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
          }

} /* end of loop forever */
```

Purche' non si ecceda la finestra

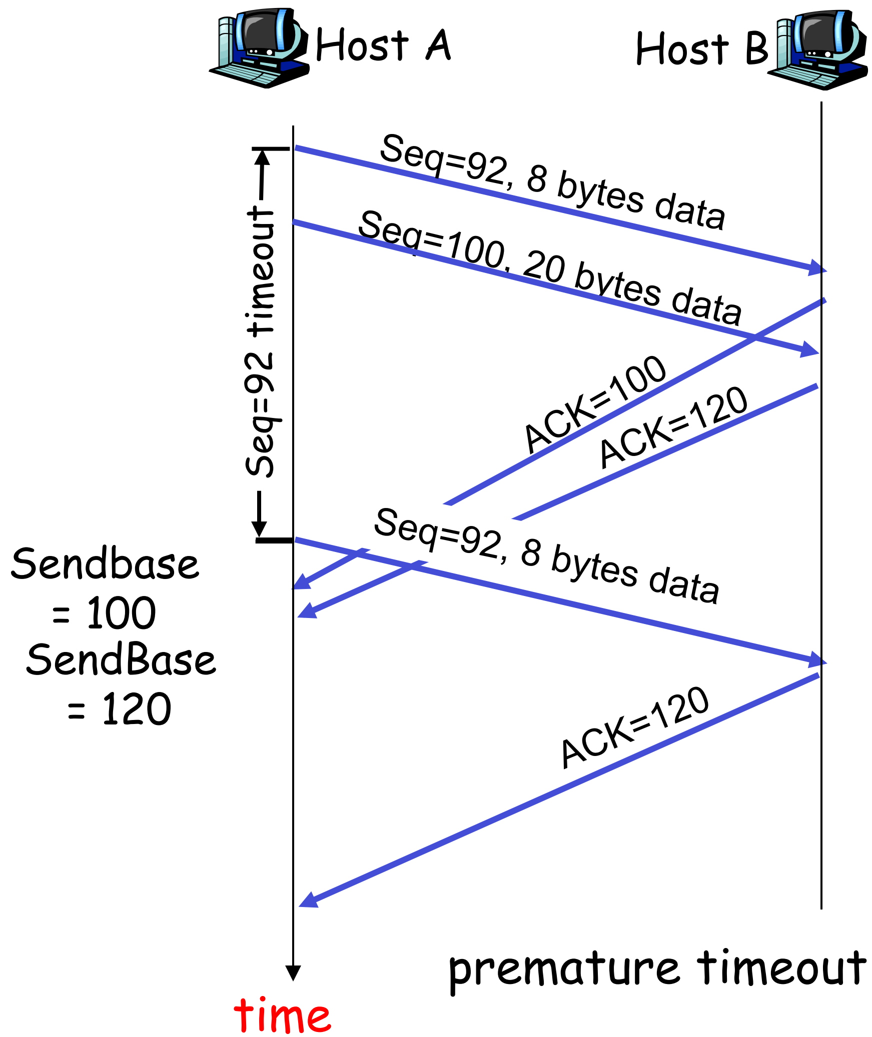
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: retransmission scenarios

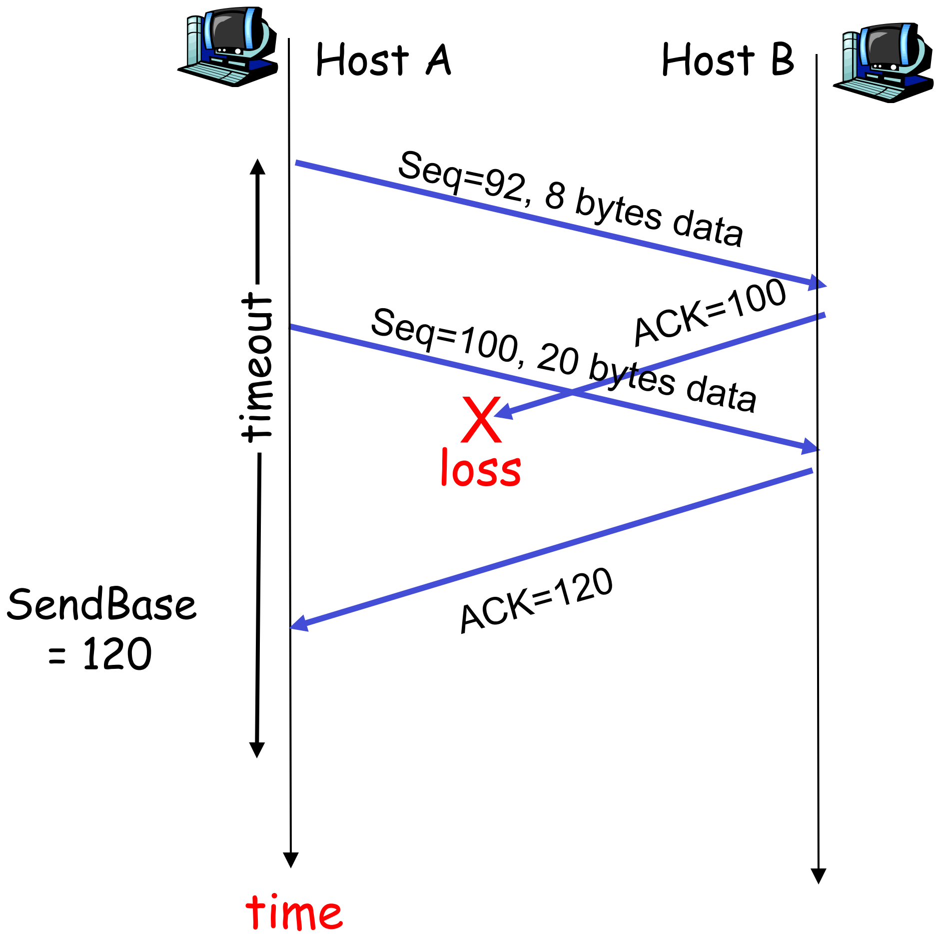


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Main motivation: performance

Favor piggybacking

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	**Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK**
Arrival of in-order segment with expected seq #. One other segment has ACK pending	**Immediately send single cumulative ACK, ACKing both in-order segments**
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	**Immediately send duplicate ACK, indicating seq. # of next expected byte**
Arrival of segment that partially or completely fills gap	**Immediate send ACK, provided that segment starts at lower end of gap**

Duplicate ACK important feedback—more later

Can advance source window

So what is the TCP solution

- Go-Back-N??
- Selective Repeat?
- A: An Hybrid solution.
  - Possibility of buffering correctly received packets AND selective retransmission of packets, BUT NOT pure Selective Repeat, cumulative ACK, buffering not required (free implementation choice)
  - Shares some aspects with GBN BUT
    - A single timer for the oldest unacked packet;
    - when the timer experises ONLY that packet is retransmitted

TCP: a reliable transport

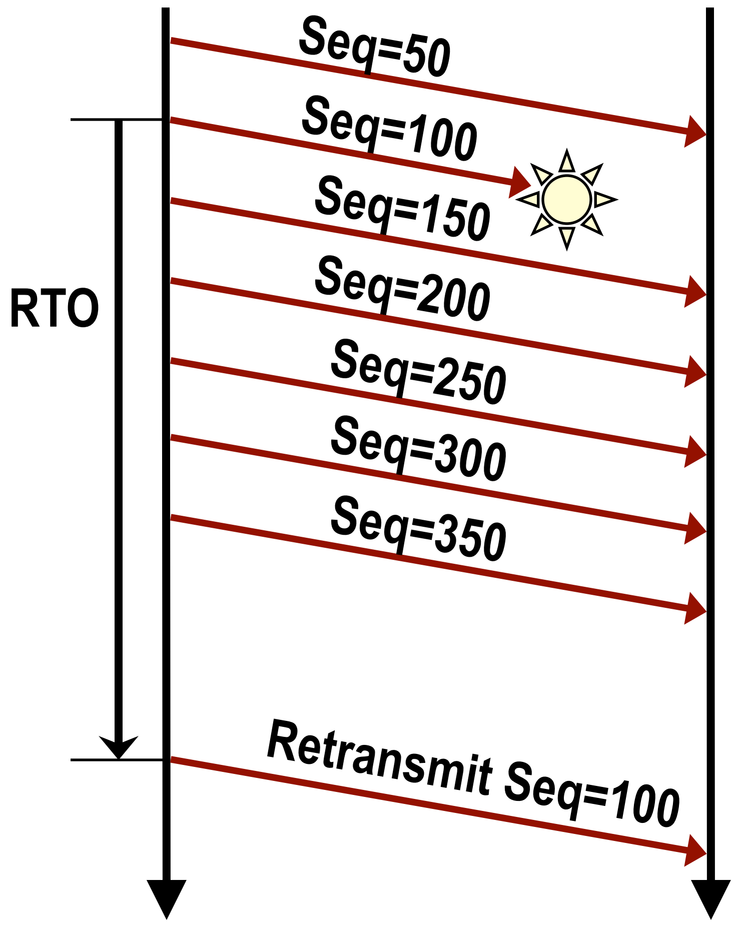
- TCP is a reliable protocol
  - all data sent are guaranteed to be received
  - ***very important feature, as IP is unreliable network layer***
- employs positive acknowledgement
  - cumulative ack
  - selective ack may be activated when both peers implement it (use option) → TCP SACKS
- does not employ negative ack
  - error discovery via timeout (retransmission timer)
  - …But "implicit NACK" is available (more later: fast retransmit)

Need for implicit NACKs

- ➔ TCP does not support negative ACKs
- ➔ This can be a serious drawback
  - ⇨ Especially in the case of single packet loss
- ➔ Necessary RTO expiration to start retransmit lost packet

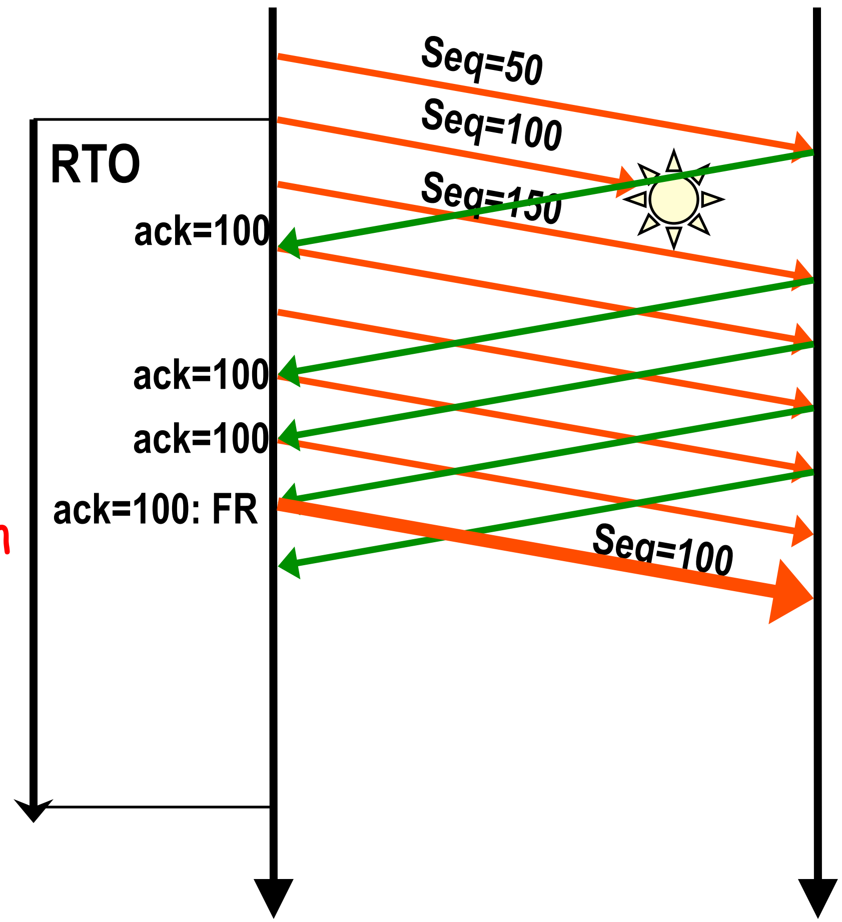
  May take too much time before retransmitting!!!

- ➔ ISSUE: is there a way to have NACKs in an implicit manner????



The Fast Retransmit Algorithm

- Idea: use duplicate ACKs!
  - Receiver responds with an ACK every time it receives an out-of-order segment
  - ACK value = last correctly received segment
- FAST RETRANSMIT algorithm:
  - if 3 duplicate acks are received for the same segment, assume that the next segment has been lost. Retransmit it right away.
  - Helps if single packet lost. Not very effective with multiple losses



Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
             }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
                }
```

a duplicate ACK for already ACKed segment

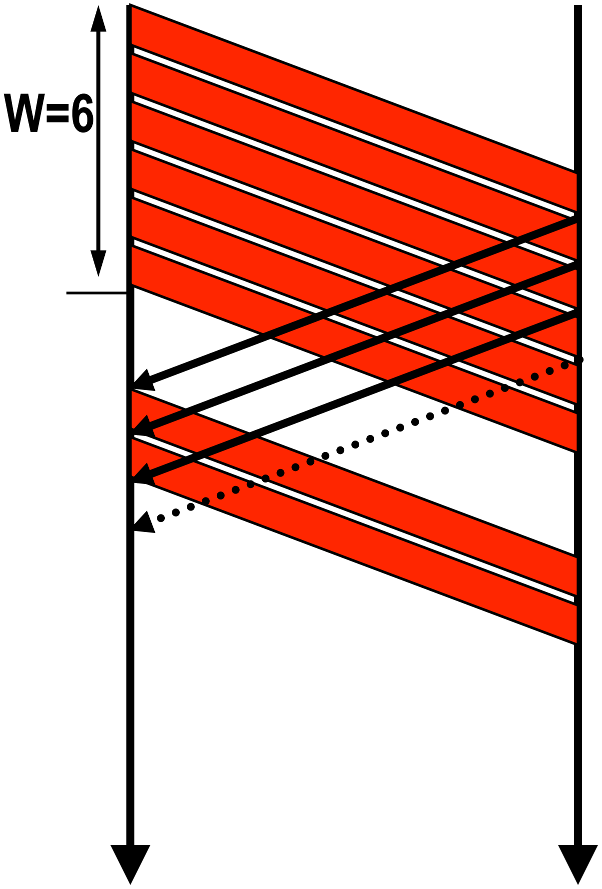
fast retransmit

TCP mechanisms for:

- flow control
- congestion control

Graphical examples (applet java) of several algorithms at:
http://www.ce.chalmers.se/~fcela/tcp-tour.html

TCP pipelining

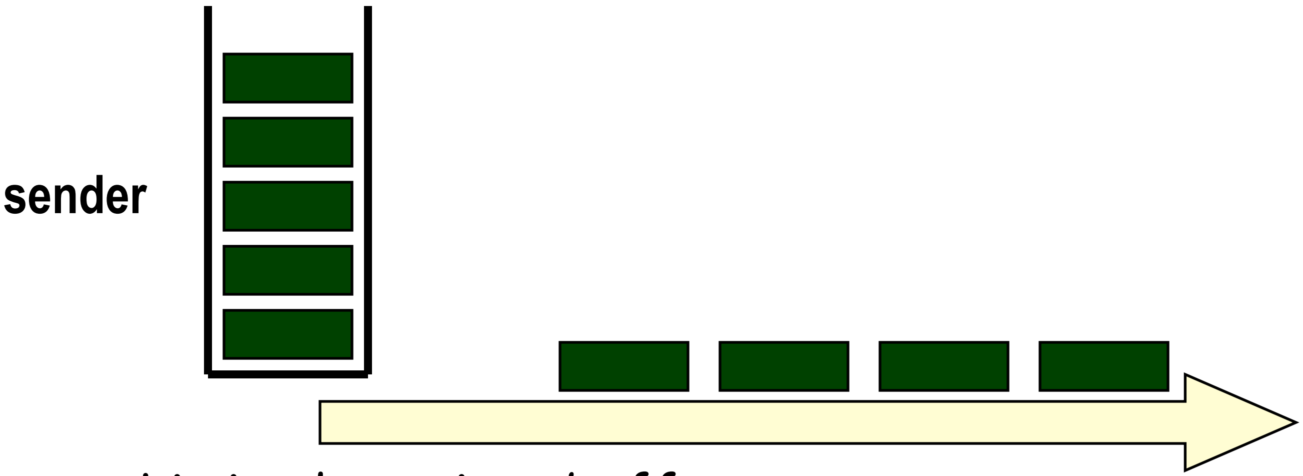


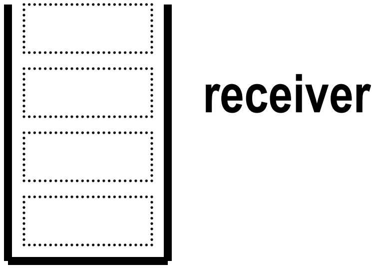
- More than 1 segment "flying" in the network
- Transfer efficiency increases with W

$$thr = \min\left(C, \frac{W \cdot MSS}{RTT + MSS / C} \right)$$

- So, why an upper limit on W?
  - Esempio: flow control

Why flow control?



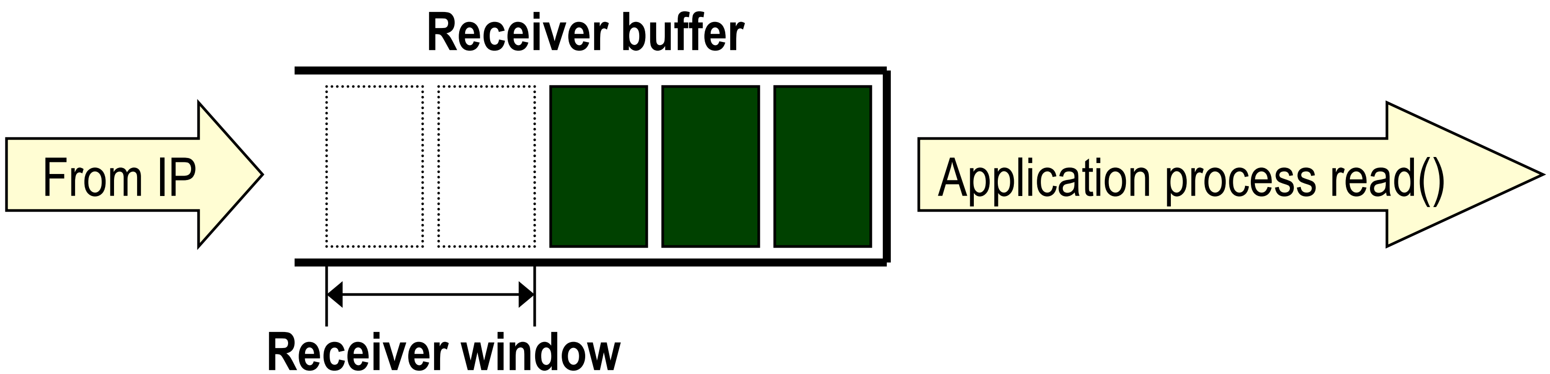


- Limited receiver buffer
  - If MSS = 2KB = 2048 bytes
  - And receiver buffer = 8 KB = 8192 bytes
  - Then W must be lower or equal than 4 x MSS
- A possible implementation:
  - During connection setup, exchange W value.
  - *DOES NOT WORK. WHY?*

Window-based flow control

➔ **receiver buffer capacity varies with time!**

⇨ Upon application process read() [asynchronous, not depending on OS, not predictable]



- ➔ MSS = 2KB = 2048 bytes
- ➔ Receiver Buffer capacity = 10 KB = 10240 bytes
- ➔ TCP data stored in buffer: 3 segments
- ➔ Receiver window = Spare room: 10-6 = 4KB = 4096 bytes
  - ⇨ Then, at this time, W must be lower or equal than 2 x MSS

<table>
<tr><td colspan="8">Source port</td><td>Destination port</td></tr>
<tr><td colspan="9">32 bit Sequence number</td></tr>
<tr><td colspan="9">32 bit acknowledgement number</td></tr>
<tr><td>Header length</td><td>6 bit Reserved</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Window size</td></tr>
<tr><td colspan="8">checksum</td><td>Urgent pointer</td></tr>
</table>

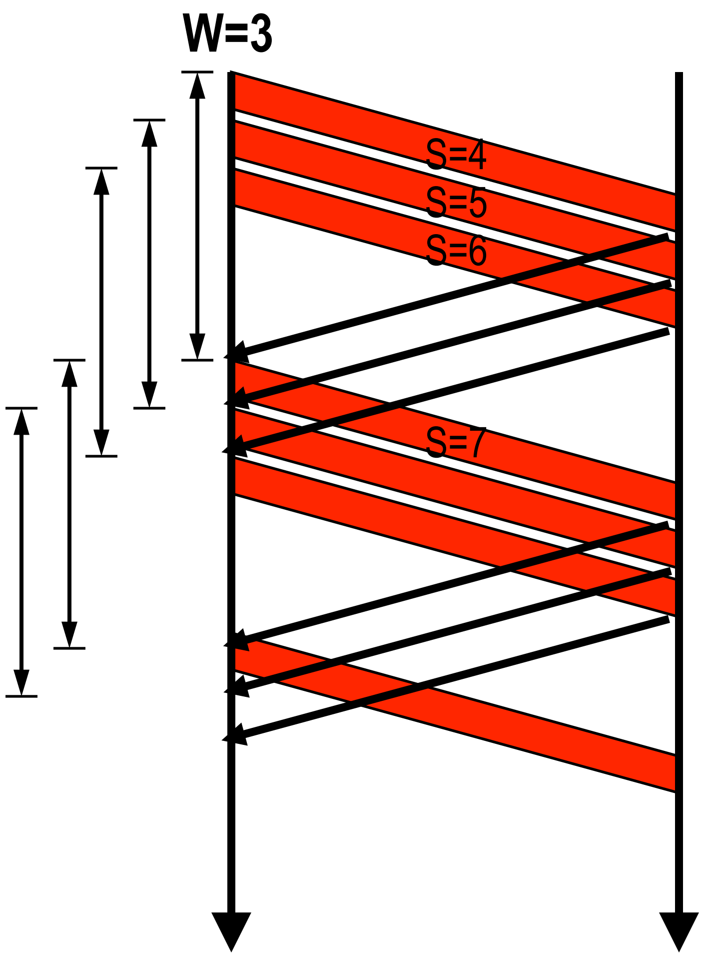
- Window size field: used to advertise receiver's remaining storage capabilities
  - 16 bit field, on every packet
  - Measure unit: bytes, from 0 (included) to 65535
  - Sender rule:

    `LastByteSent - LastByteAcked <= RcvWindow.`
  - W=2048 means:
    - I can accept other 2048 bytes since ack, i.e. bytes [ack, ack+W-1]
    - also means: sender may have 2048 bytes outstanding (in multiple segments)

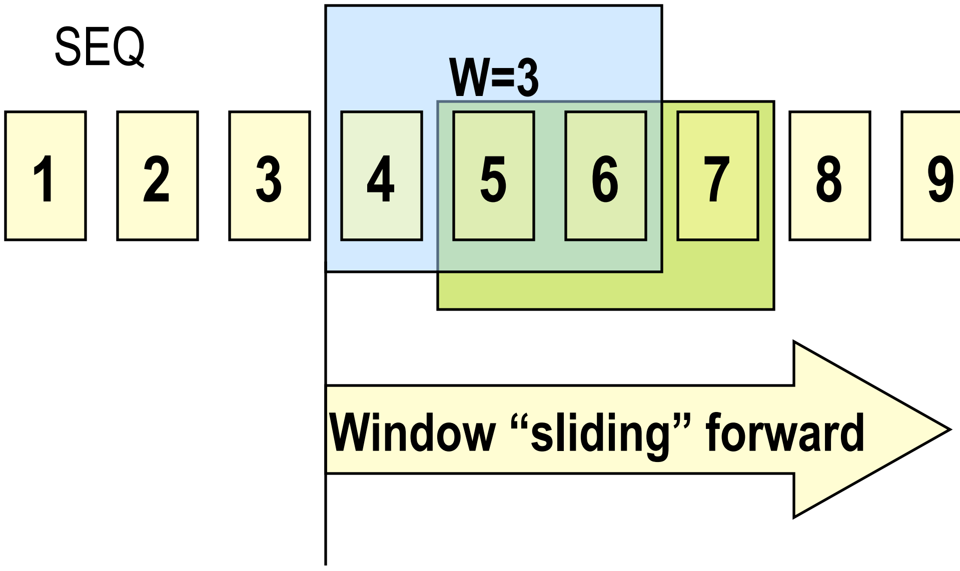
What is flow control needed for?

- Window flow control guarantees receiver buffer to be able to accept outstanding segments.
- When receiver buffer full, just send back win=0
- in essence, flow control guarantees that transmission bit rate never exceed receiver rate

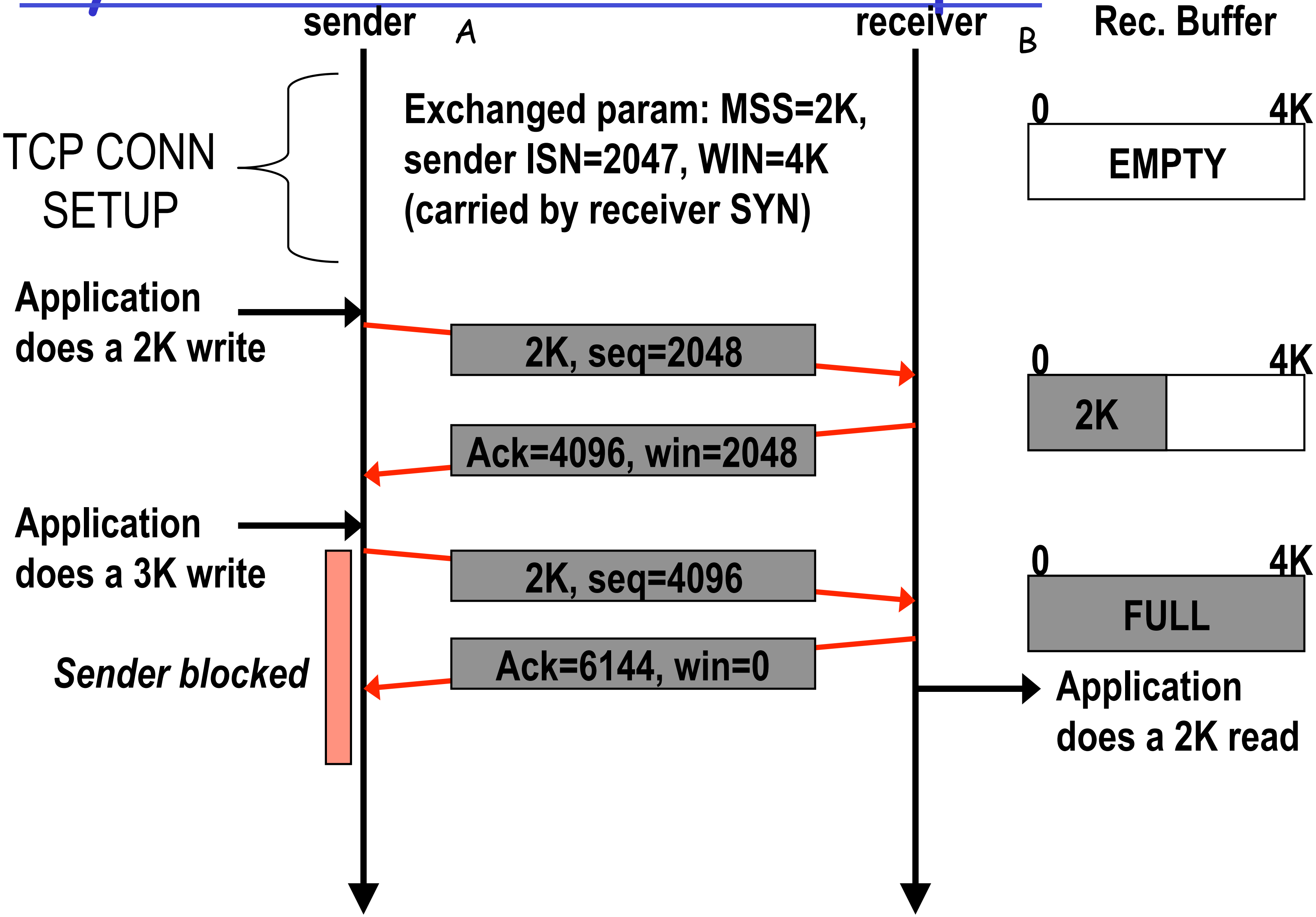
Sliding window



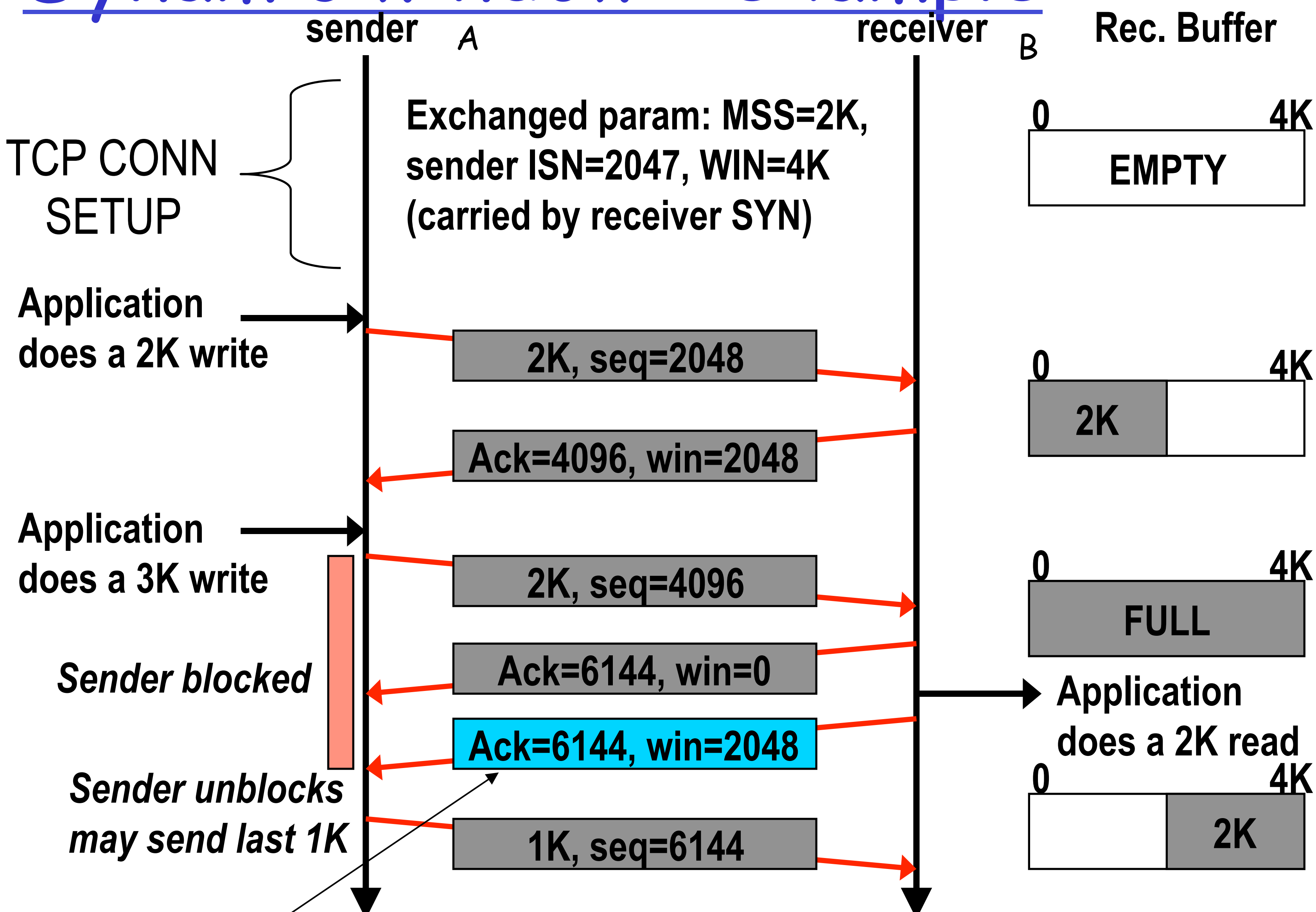
Dynamic window based reduces to pure sliding window when receiver app is very fast in reading data…



Dynamic window - example



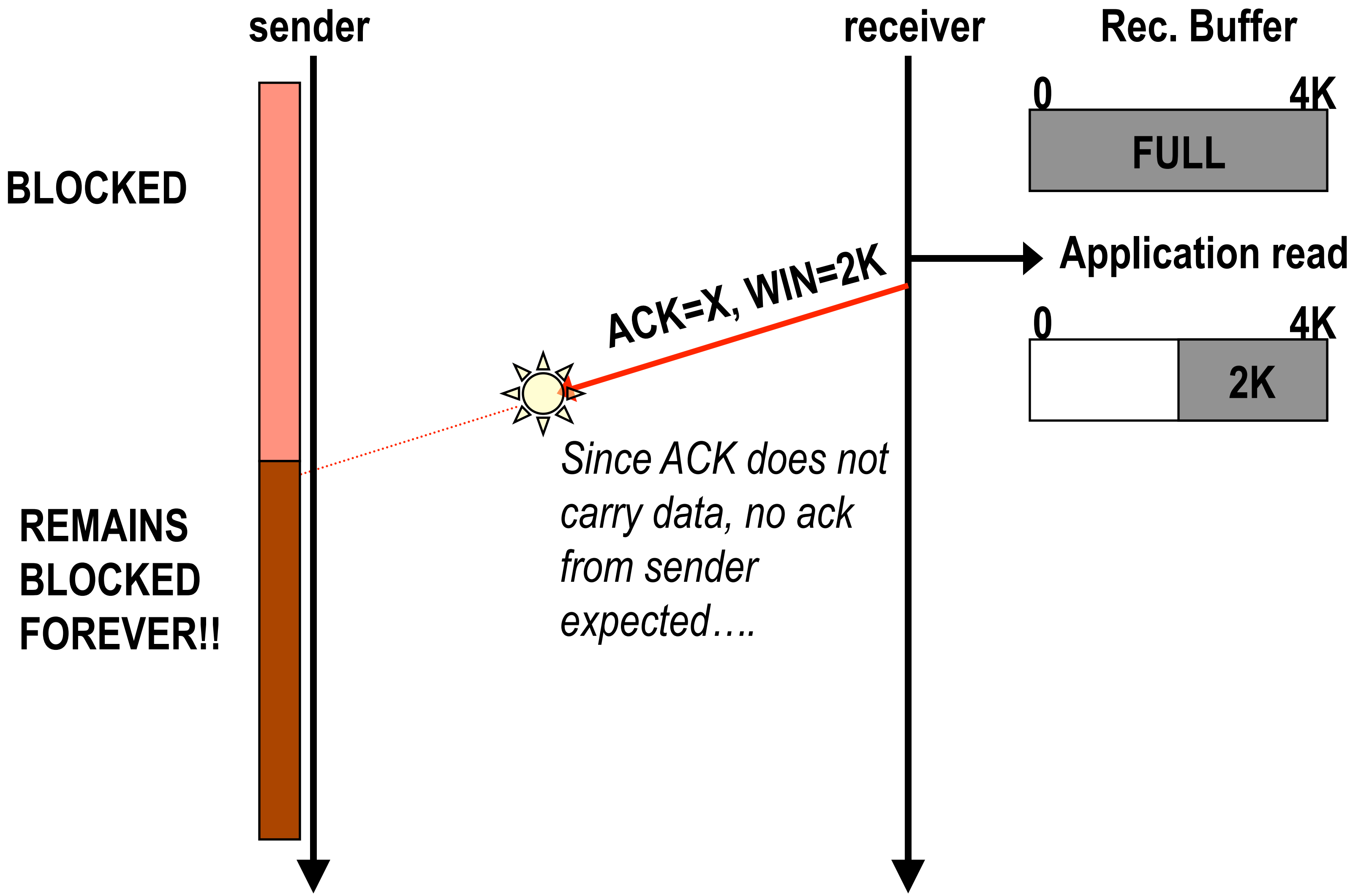
Dynamic window - example



Piggybacked in a packet sent from B to A

Window thus source rate limited by reading speed and buffer size at the receiver

Blocked sender deadlock problem



Solution: Persist timer

- When win=0 (blocked sender), sender starts a “persist” timer
    - Initially 500ms (but depends on implementation)
- When persist timer elapses AND no segment received during this time, sender transmits “probe”
    - Probe = 1byte segment; makes receiver reannounce next byte expected and window size
        - this feature necessary to break deadlock
        - if receiver was still full, rejects byte
        - otherwise acks byte and sends back actual win
- Persist time management (exponential backoff):
    - Doubles every time no response is received
    - Maximum = 60s