Lecture 7.
TCP mechanisms for:

- ➔ **data transfer control / flow control**
- ➔ **error control**
- ➔ **congestion control**

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

G.Bianchi, G.Neglia, V.Mancuso

Università degli Studi di Palermo

Data transfer control over TCP

a double-face issue:

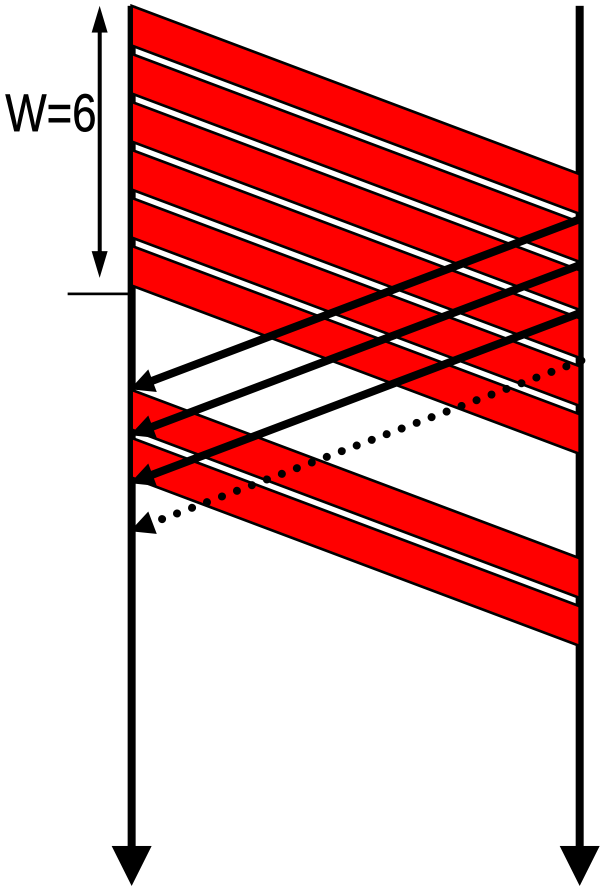
➔Bulk data transfer

- ⇨HTTP, FTP, …
- ⇨goal: attempt to send data as fast as possible
- ⇨problems: sender may transmit faster than receiver
  - →Flow control

➔Interactive

- ⇨TELNET, RLOGIN, ...
- ⇨goal: attempt to send data as soon as possible
- ⇨Problem: efficiency - interactivity trade-off
  - →The tinygrams issue! (1 byte payload / segment - 20 TCP + 20 IP header)

TCP pipelining

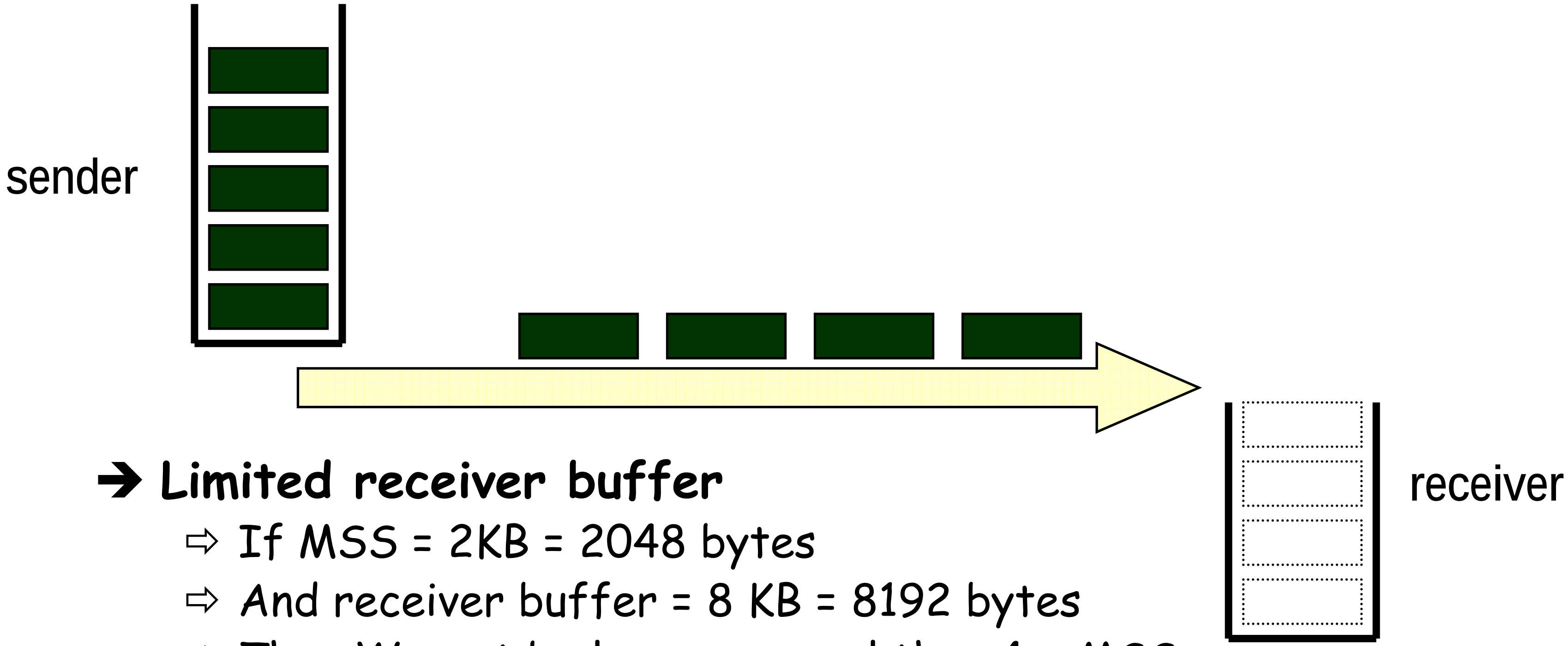


- ➔ More than 1 segment "flying" in the network
- ➔ Transfer efficiency increases with W

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

- ➔ So, why an upper limit on W?

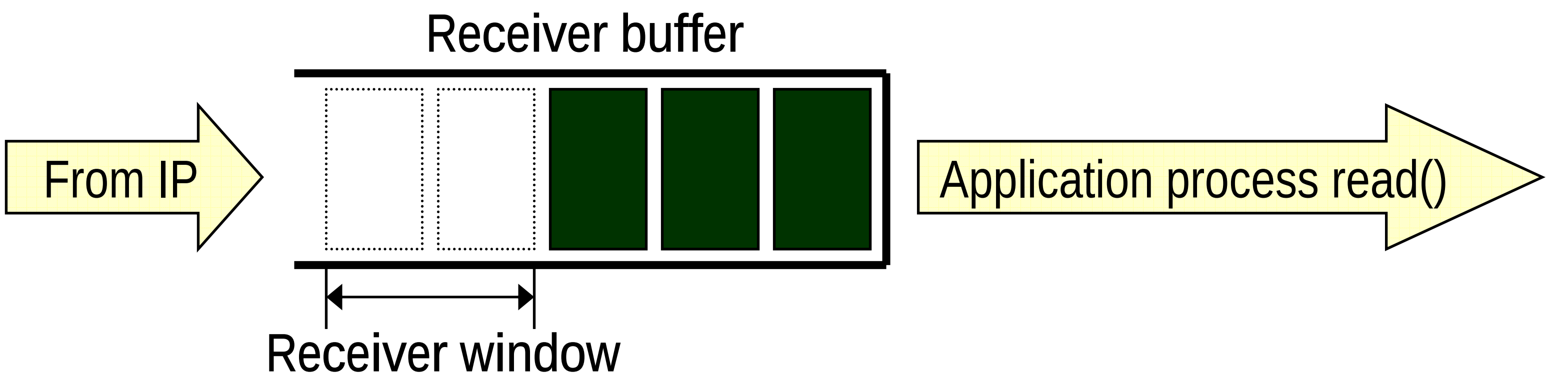
Why flow control?



- **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>

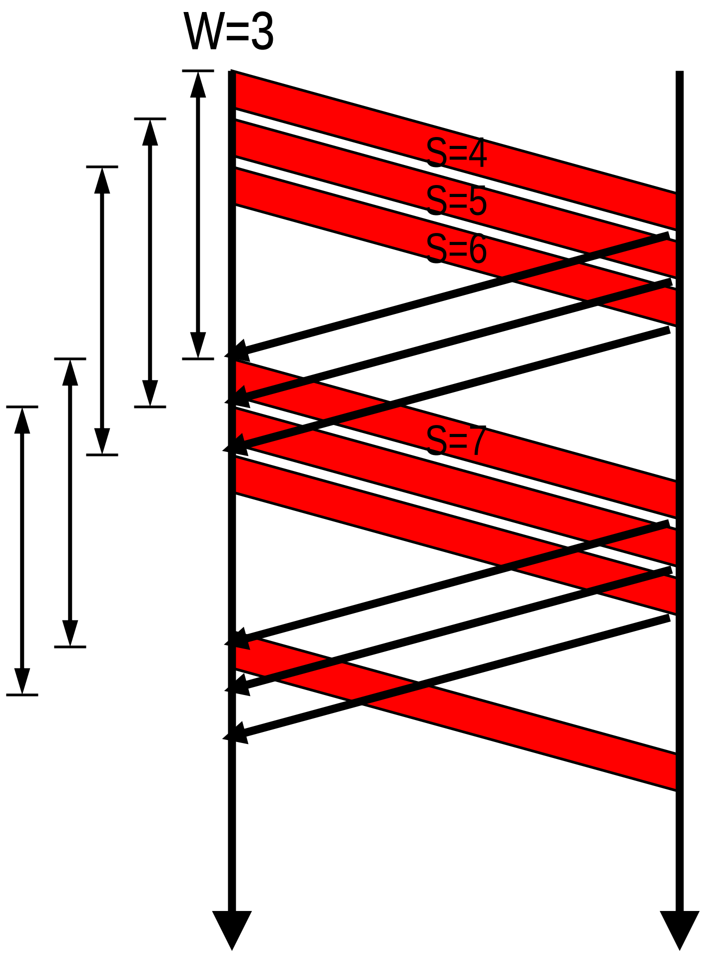
➔ **Window size field: used to advertise receiver's remaining storage capabilities**

- ⇨ 16 bit field, on <u>every</u> 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)

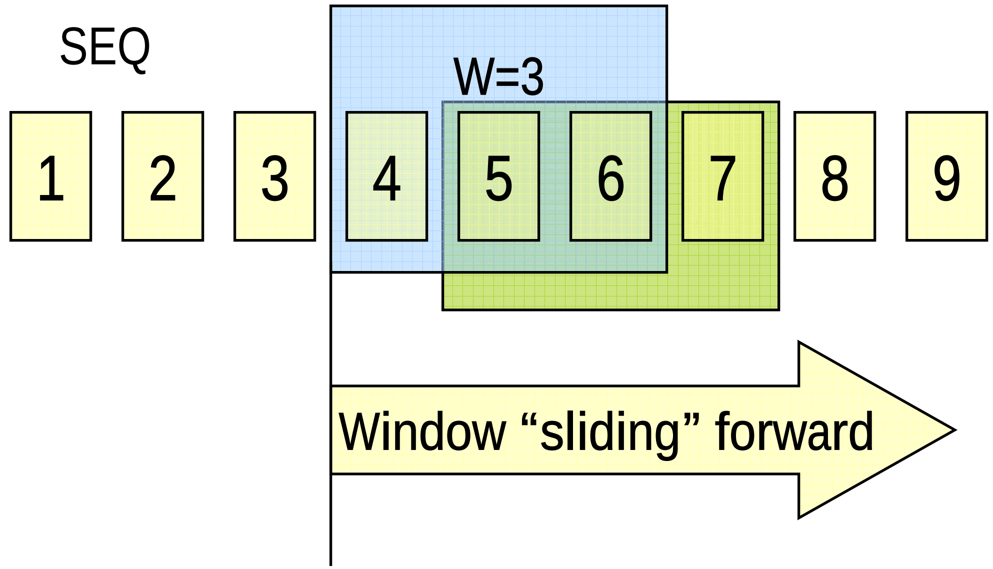
What is flow control needed for?

- ➔ **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**
  - ⇨ in average!
  - ⇨ Note that instantaneous transmission rate is arbitrary…
  - ⇨ as well as receiver rate is discretized (application reads)

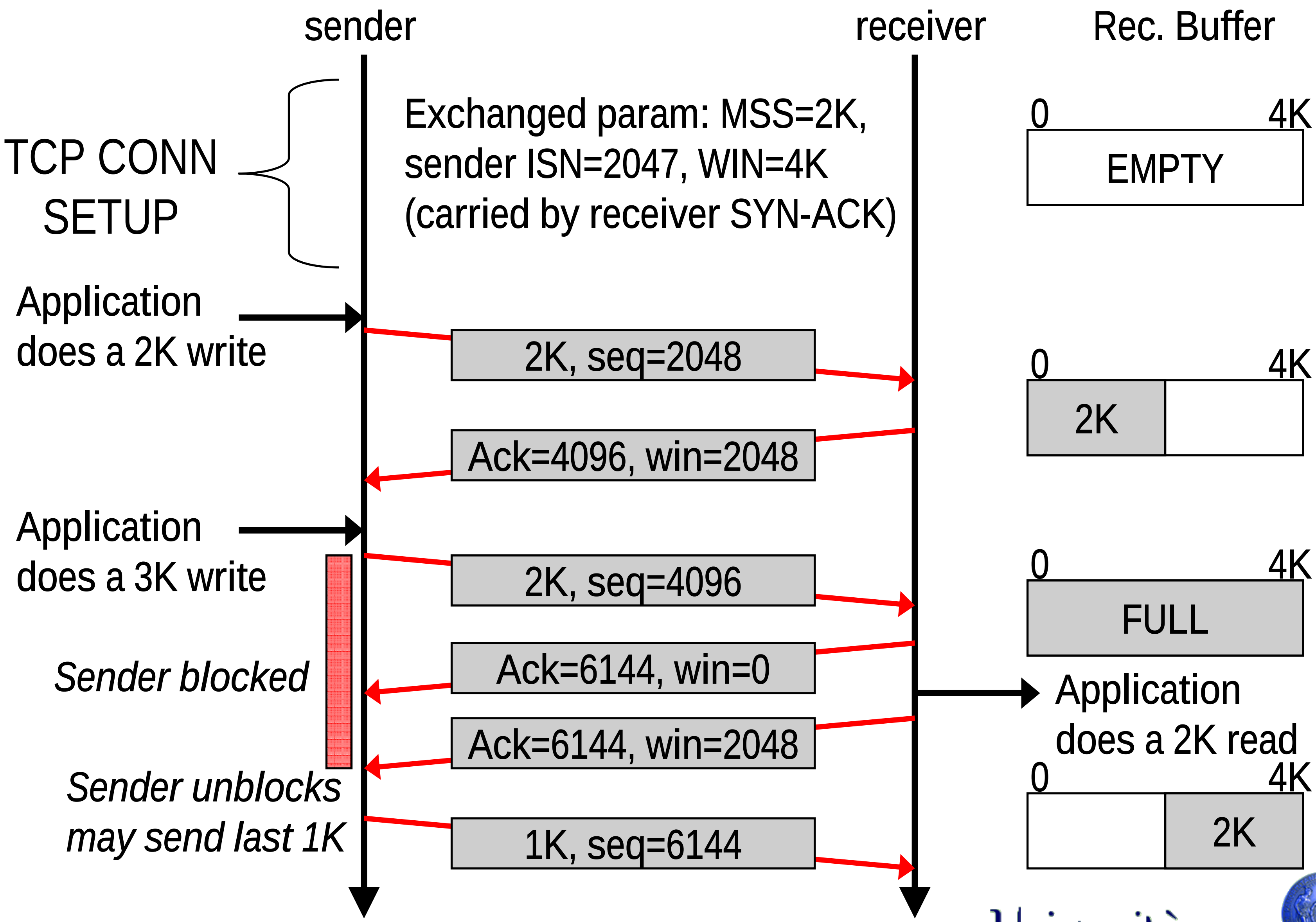
Sliding window



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



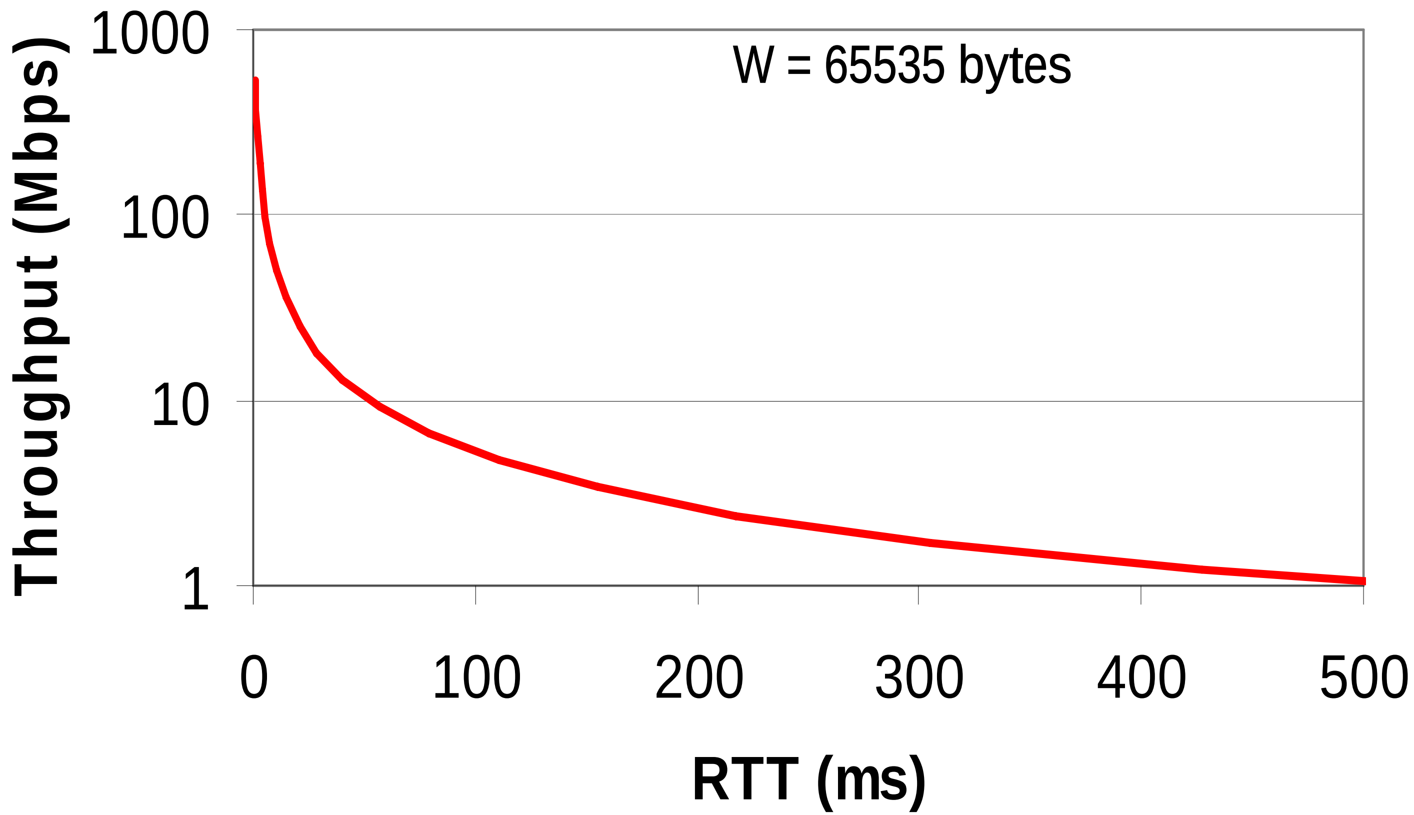
Dynamic window - example



Performance: bounded by receiver buffer size

- ➔ **Up to 1992, common operating systems had transmitter & receiver buffer defaulted at 4096**
  - ⇨ e.g. SunOS 4.1.3
- ➔ **way suboptimal over Ethernet LANs**
  - ⇨ raising buffer to 16384 = 40% throughput increase (Papadopulos & Parulkar, 1993)
  - ⇨ e.g. Solaris 2.2 default
- ➔ **most socket APIs allow apps to set (increase) socket buffer sizes**
  - ⇨ But theoretical maximum remains W=65535 bytes…

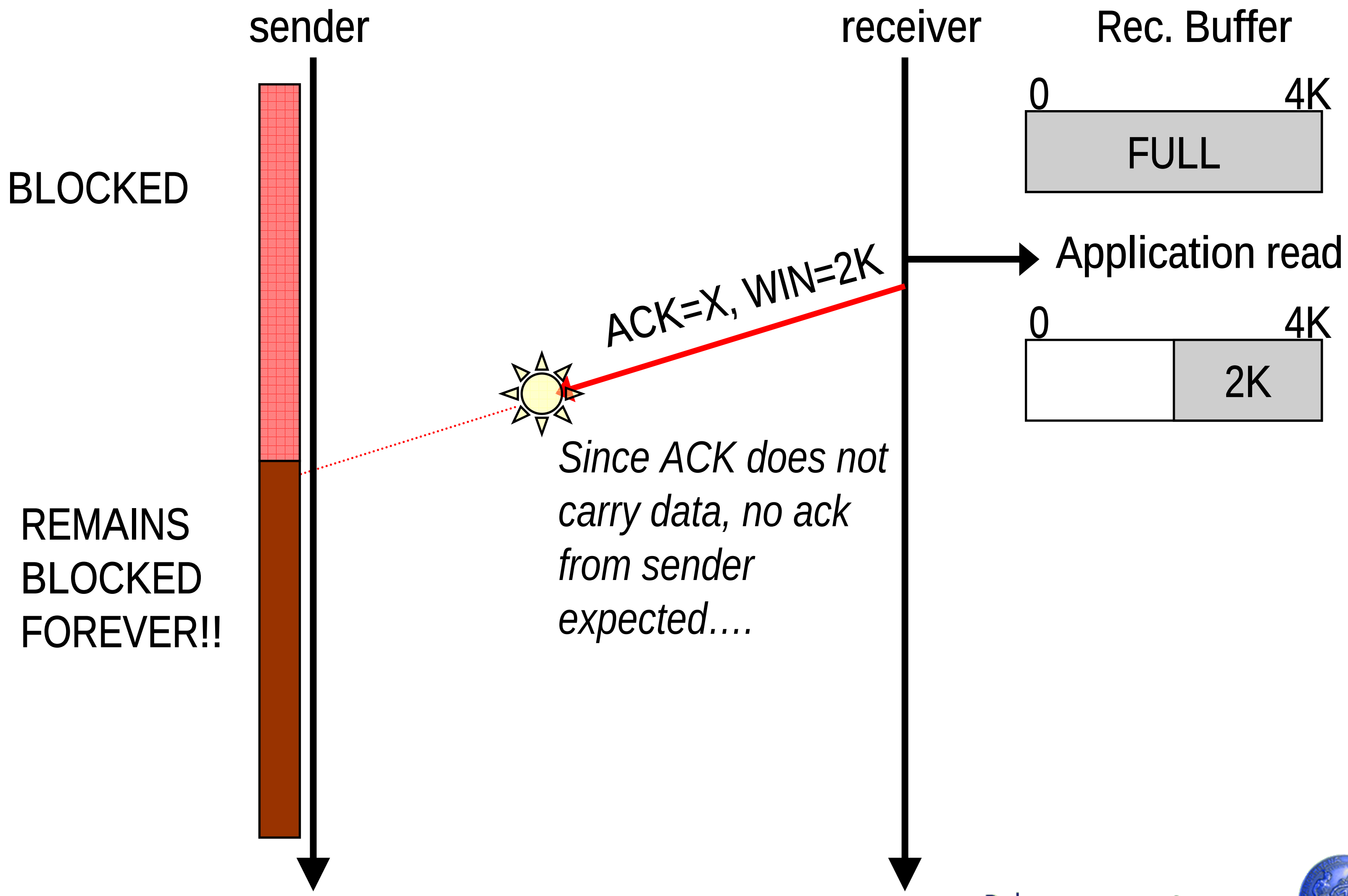
Maximum achievable throughput
(assuming infinite speed line…)



Window Scale Option

- **Appears in SYN segment**
  - operates only if both peers understand option
- **allows client & server to agree on a different W scale**
  - specified in terms of bit shift (from 1 to 14)
  - maximum window: $65535 * 2^b$
  - b=14 means max W = 1.073.725.440 bytes!!

Blocked sender deadlock problem



Solution: Persist timer

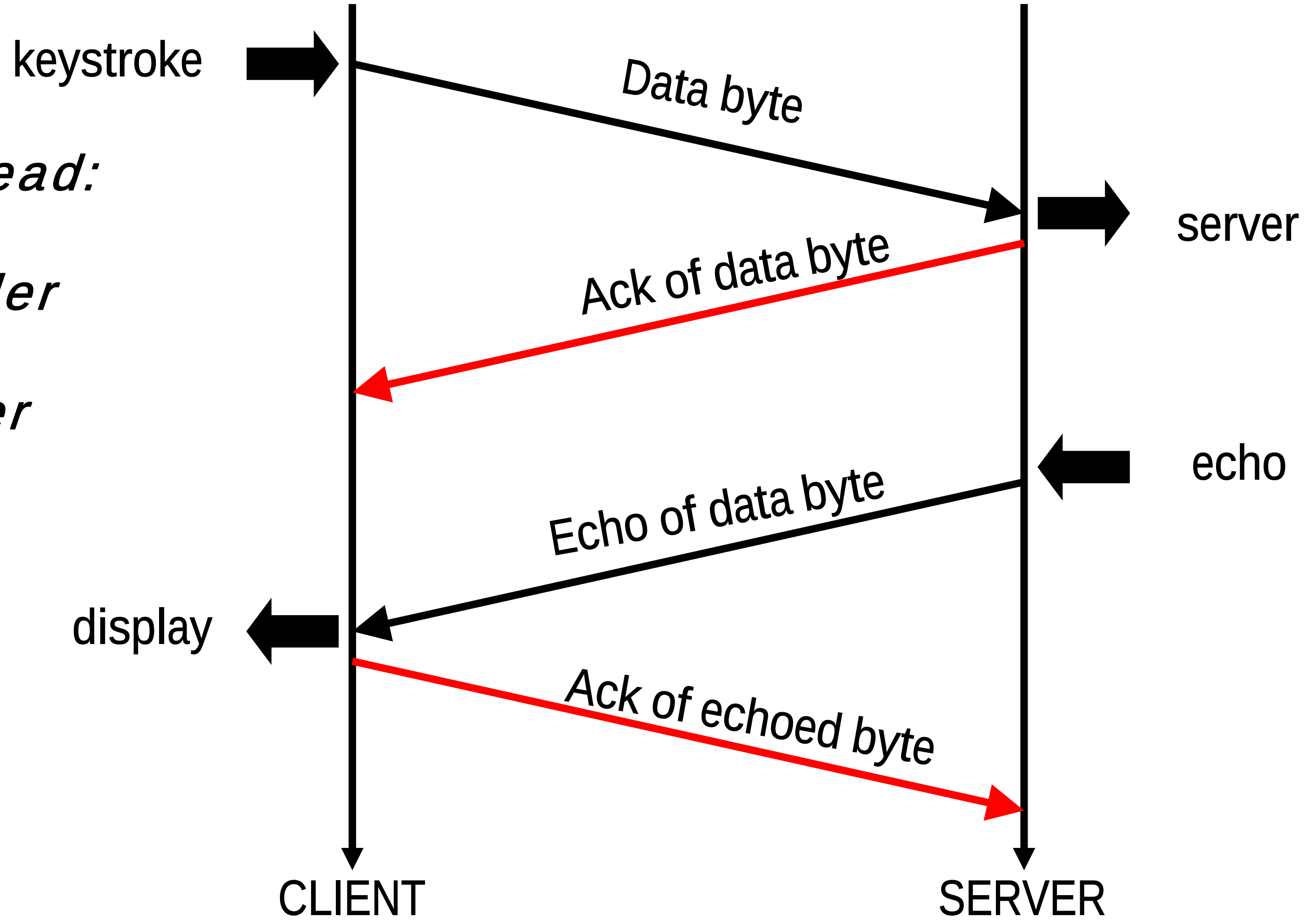
- ➔ **When win=0 (blocked sender), sender starts a "persist" timer**
  - →Initially 500ms (but depends on implementation)
- ➔ **When persist timer elapses AND no segment received during this time, sender transmits "probe"**
  - ⇨ Probe = 1byte segment; makes receiver reannounce next byte expected and 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

Interactive applications

ideal rlogin operation: 4 transmitted segments per 1 byte!!!!!

Header overhead:

20 TCP header
+
20 IP header
+
1 data

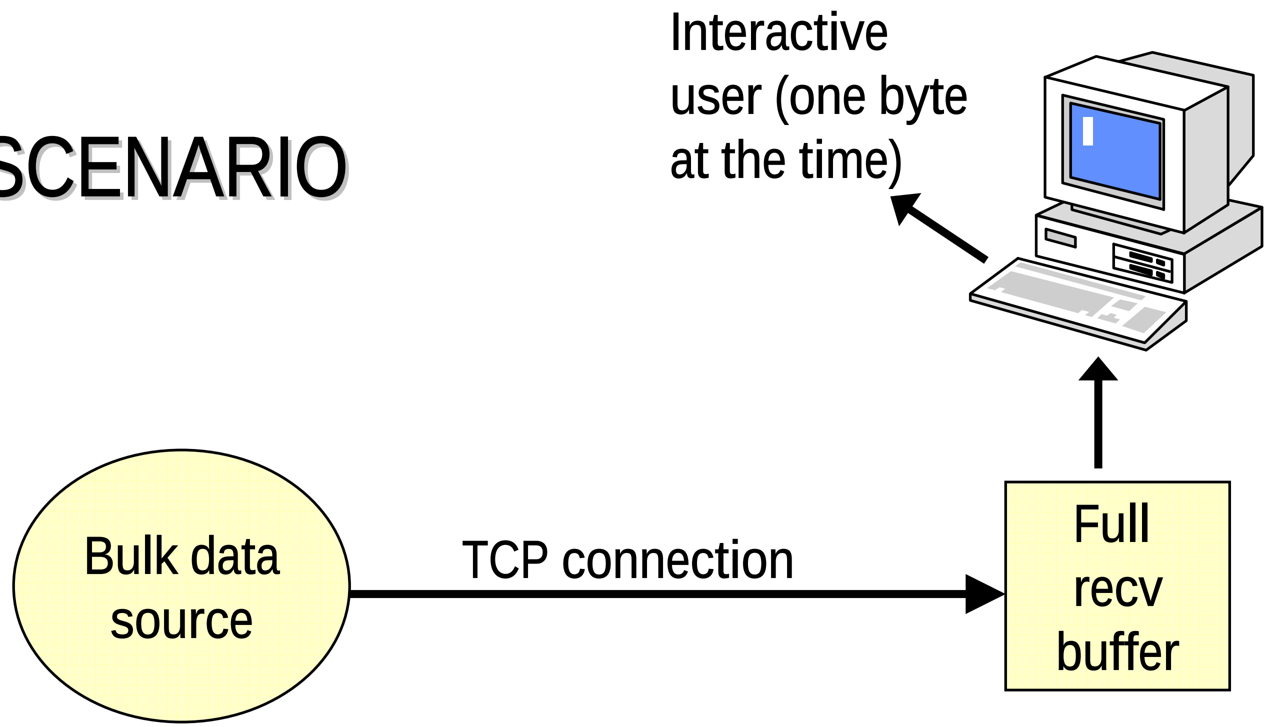


Interactive apps: create some tricky situations....

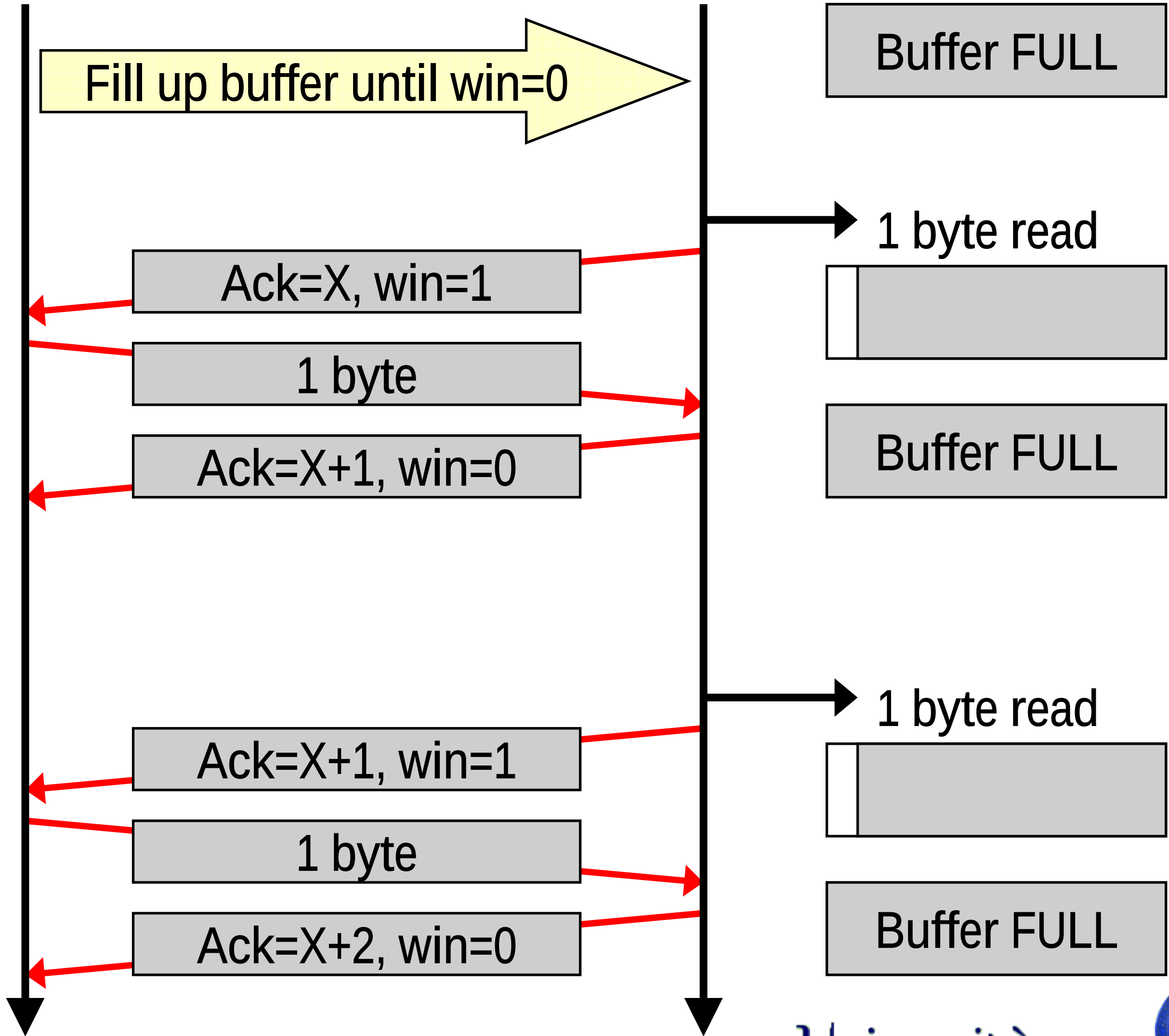
Università degli Studi di Palermo

The silly window syndrome

SCENARIO



The silly window syndrome



Network loaded with tinygrams (40bytes header + 1 payload!!)

Forever!

Silly window solution

- → **Problem discovered by David Clark (MIT), 1982**
- → **easily solved, by preventing receiver to send a window update for 1 byte**
- → **rule: send window update when:**
  - → receiver buffer can handle a whole MSS

    or

  - → half received buffer has emptied (if smaller than MSS)
- → **sender also may apply rule**
  - → by waiting for sending data when win low

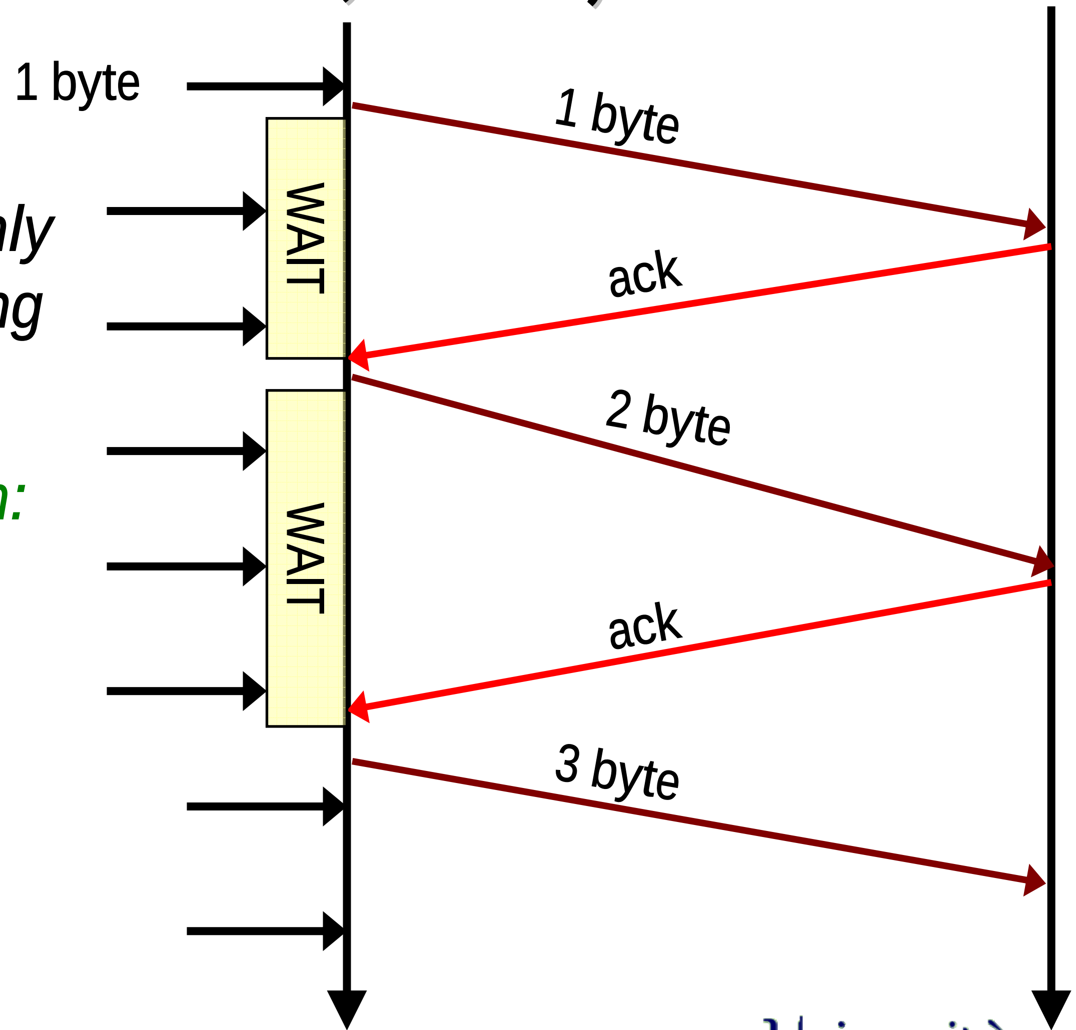
Nagle's algorithm
(RFC 896, 1984)

NAGLE RULE: a TCP connection can have only ONE SMALL outstanding segment

self-clocking algorithm:

on LANs, plenty of tynigrams

on slow WANs, data aggregation



Comments about Nagle's algo

- ➔ **Over ethernet:**
  - ⇨ about 16 ms round trip time
  - ⇨ Nagle algo starts operating when user digits more than 60 characters per second (!!!)
- ➔ **disabling Nagle's algorithm**
  - ⇨ a feature offered by some TCP APIs
    - → set TCP_NODELAY
  - ⇨ example: mouse movement over X-windows terminal

PUSH flag

<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>

- ➔ **Used to notify**
  - ⇨ TCP sender to send data
    - → but for this an header flag NOT needed! Sufficient a “push” type indication in the TCP sender API
  - ⇨ TCP receiver to pass received data to the application

Università degli Studi di Palermo

Urgent data

<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>

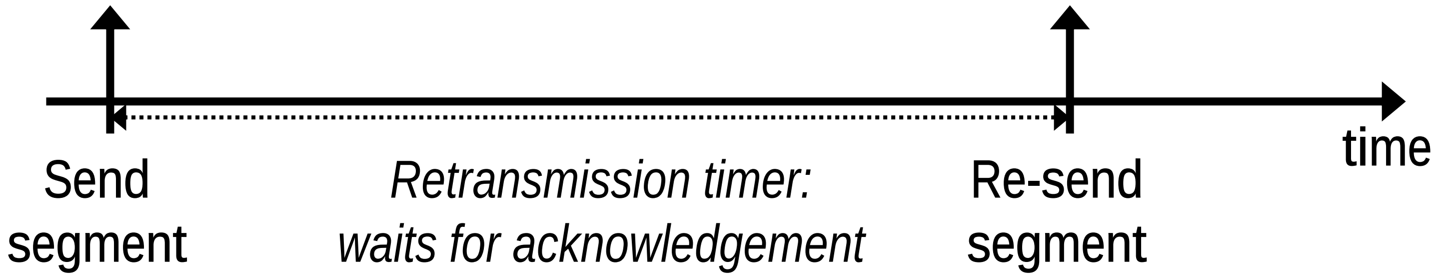
- ➔ **URG on: notifies rx that "urgent" data placed in segment.**
- ➔ **When URG on, *urgent pointer* contains position of *last byte* of urgent data**
  - → *or the one after the last, as some bugged implementations do??*
  - → *and the first? No way to specify it!*
- ➔ **receiver is expected to pass all data up to urgent ptr to app**
  - → interpretation of urgent data is left to the app
- ➔ **typical usage: ctrlC (interrupt) in rlogin & telnet; abort in FTP**
- ➔ **urgent data is a *<u>second</u>* exception to blocked sender**

TCP
Error control

TCP: a reliable transport

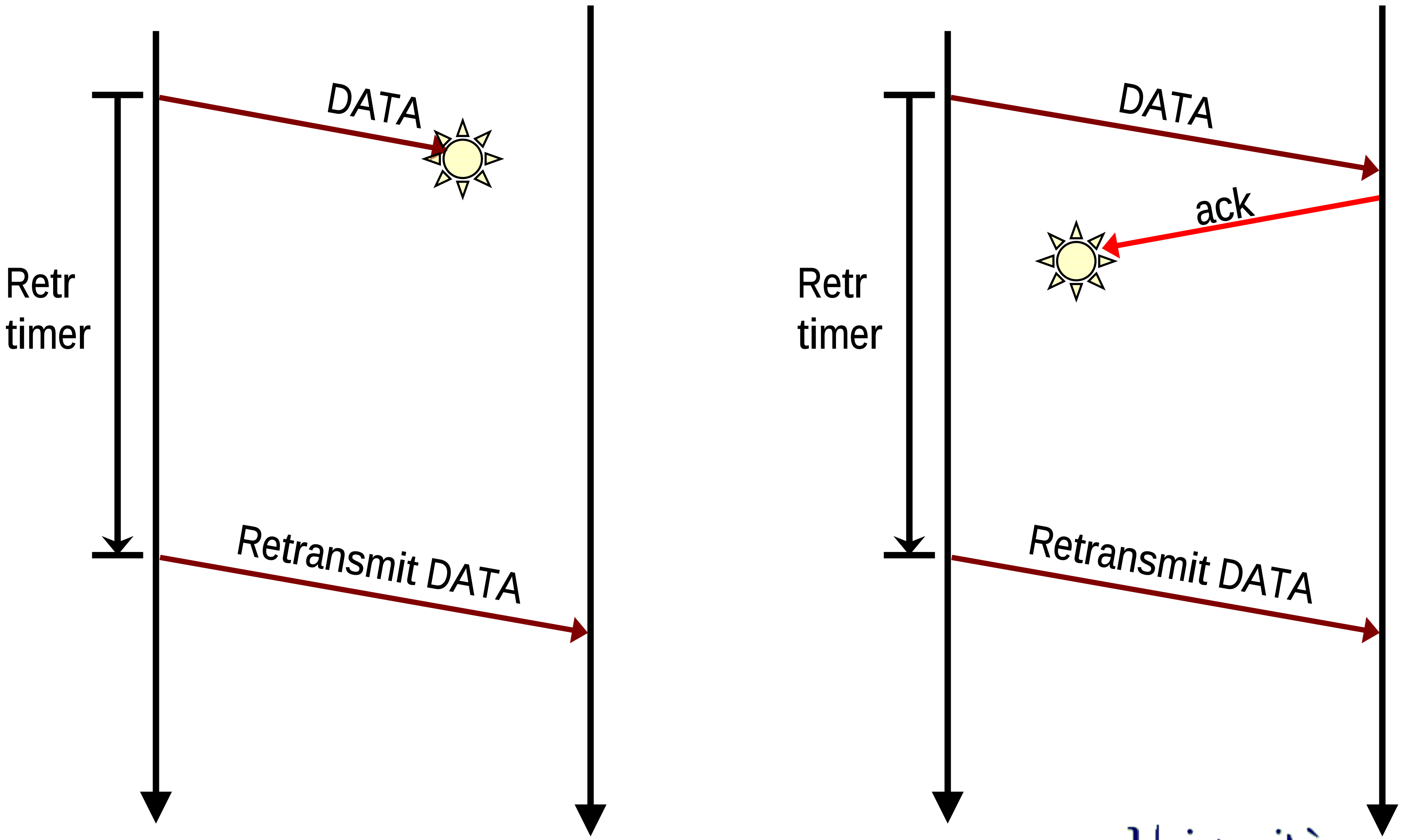
- **TCP is a reliable protocol**
  - all data sent are guaranteed to be received
  - *very important feature, as IP is unreliable network layer*
- **employs positive acknowledgement**
  - cumulative ack
  - selective ack may be activated when both peers implement it (use option)
- **does not employ negative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

Error discovery
via retransmission timer expiration

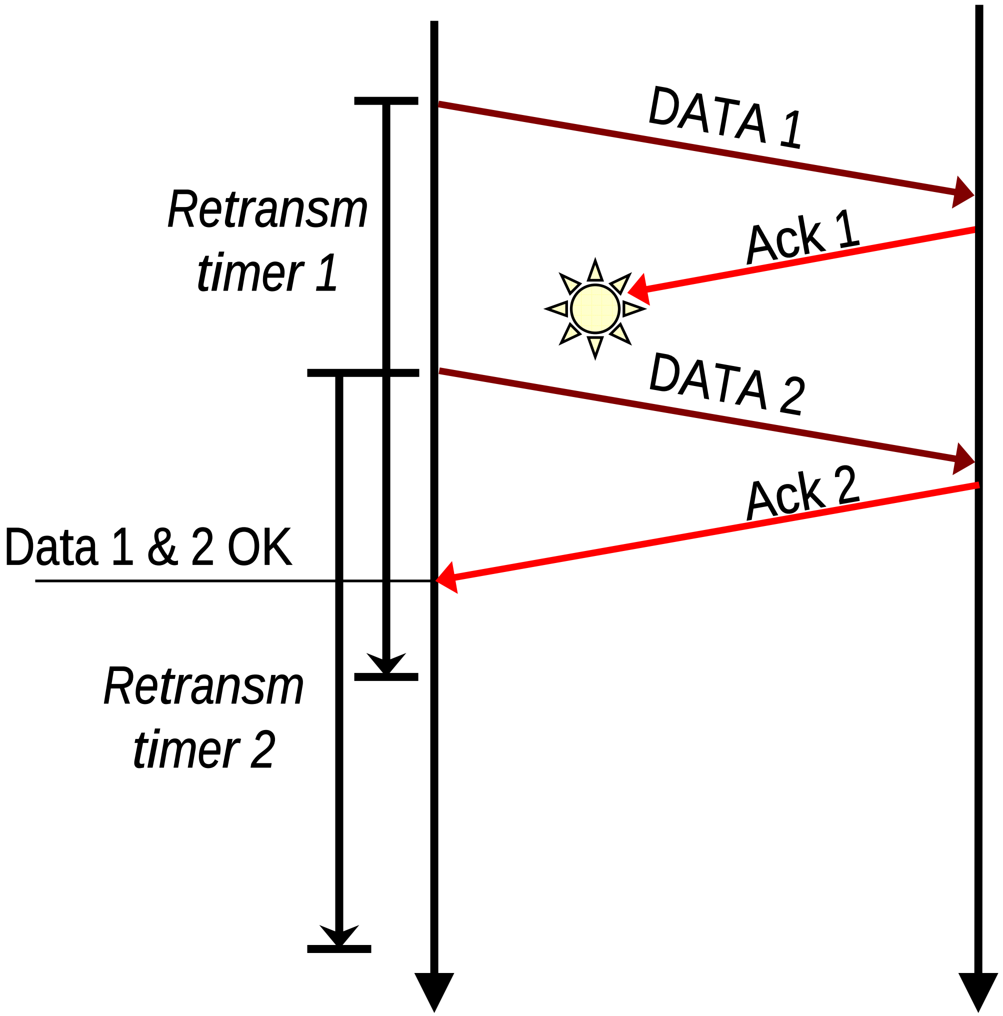


Fundamental problem:
setting the retransmission timer right!

Lost data == lost ack



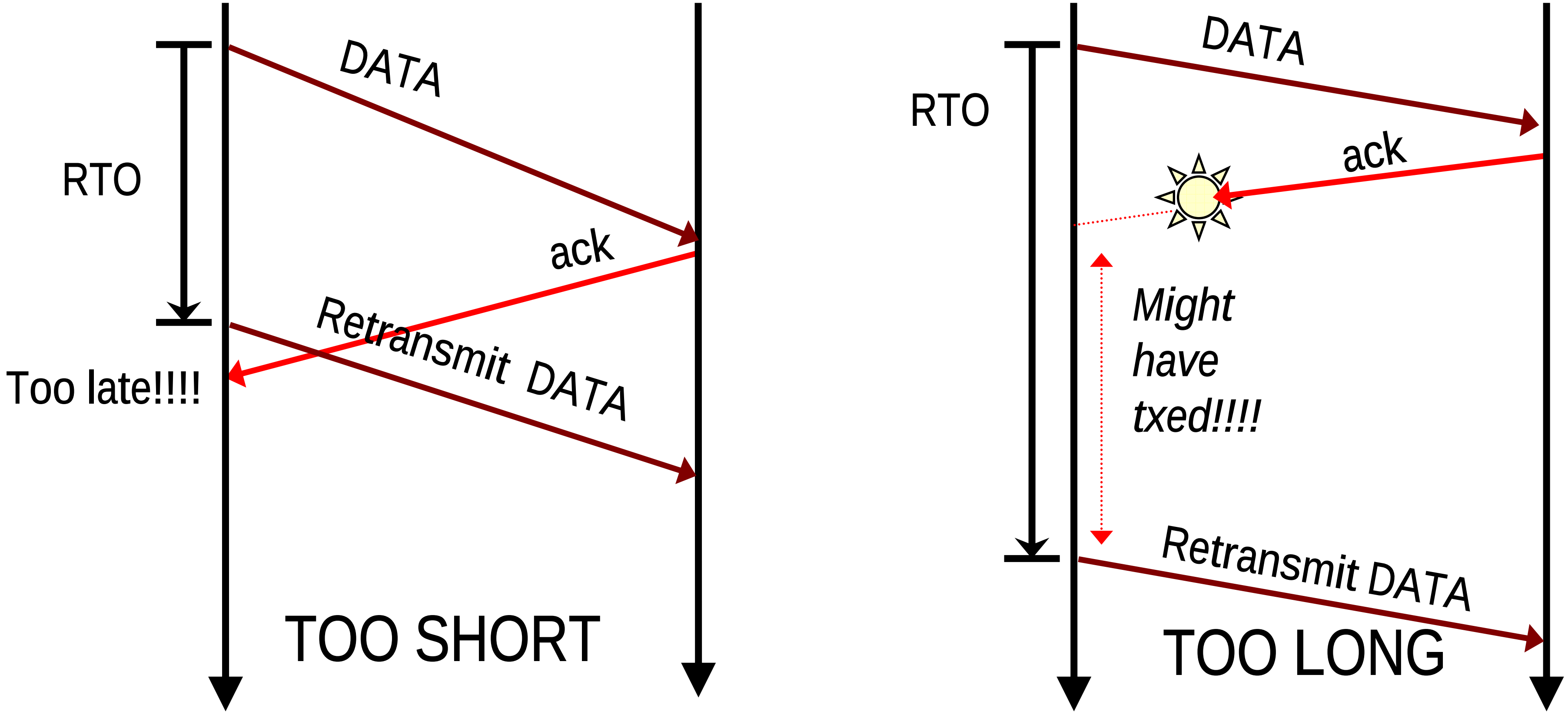
although lost ack may be discovered via subsequent acks



Università degli Studi di Palermo

Retransmission timer setting

RTO = retransmission TimeOut



- **TOO SHORT: unnecessary retransmission occurs, loading the Internet with unnecessary packets**
- **TOO LONG: throughput impairment when packets lost**

Retransmission timer setting

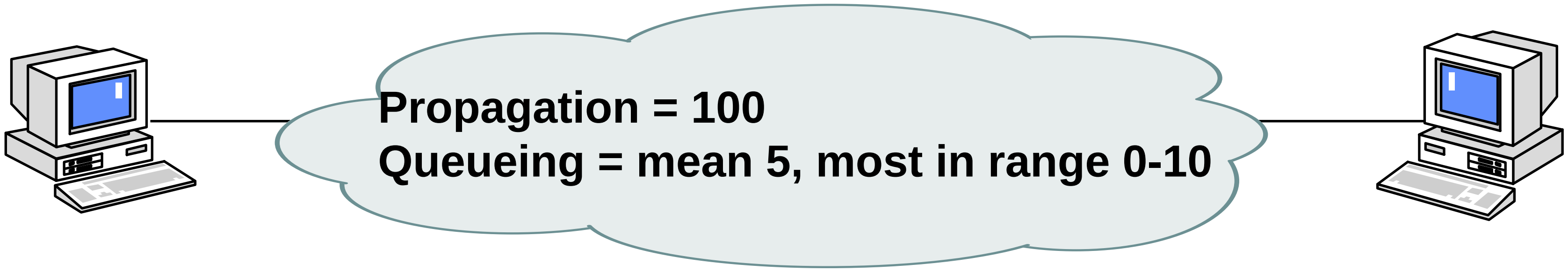
- ➔**Cannot be fixed by protocol! Two reasons:**
  - ⇨different network scenarios have very different performance
    - ➔LANs (short RTTs)
    - ➔WANs (long RTTs)
  - ⇨same network has time-varying performance (very fast time scale)
    - ➔when congestion occurs (RTT grows) and disappears (RTT drops)

Adaptive RTT setting

- ➔ **Proposed in RFC 793**
- ➔ **based on dynamic RTT estimation**
  - ⇨ sender samples time between sending SEQ and receiving ACK (M)
  - ⇨ estimates RTT (R) by low pass filtering M (autoregressive, 1 pole)
    - ➔ $\mathbf{R = \alpha R + (1-\alpha) M}$ $\qquad \mathbf{\alpha = 0.9}$
  - ⇨ sets RTO = R β $\qquad \beta = 2$ (recommended)

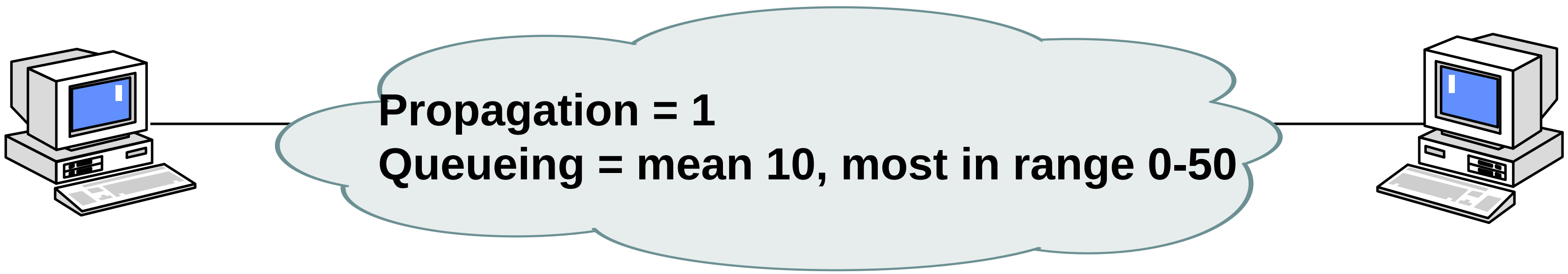
Problem: constant value β=2

SCENARIO 1: lightly loaded long-distance communication



RTO = 2 x measured_RTT ~ 2 x 105 = 210 TOO LARGE!

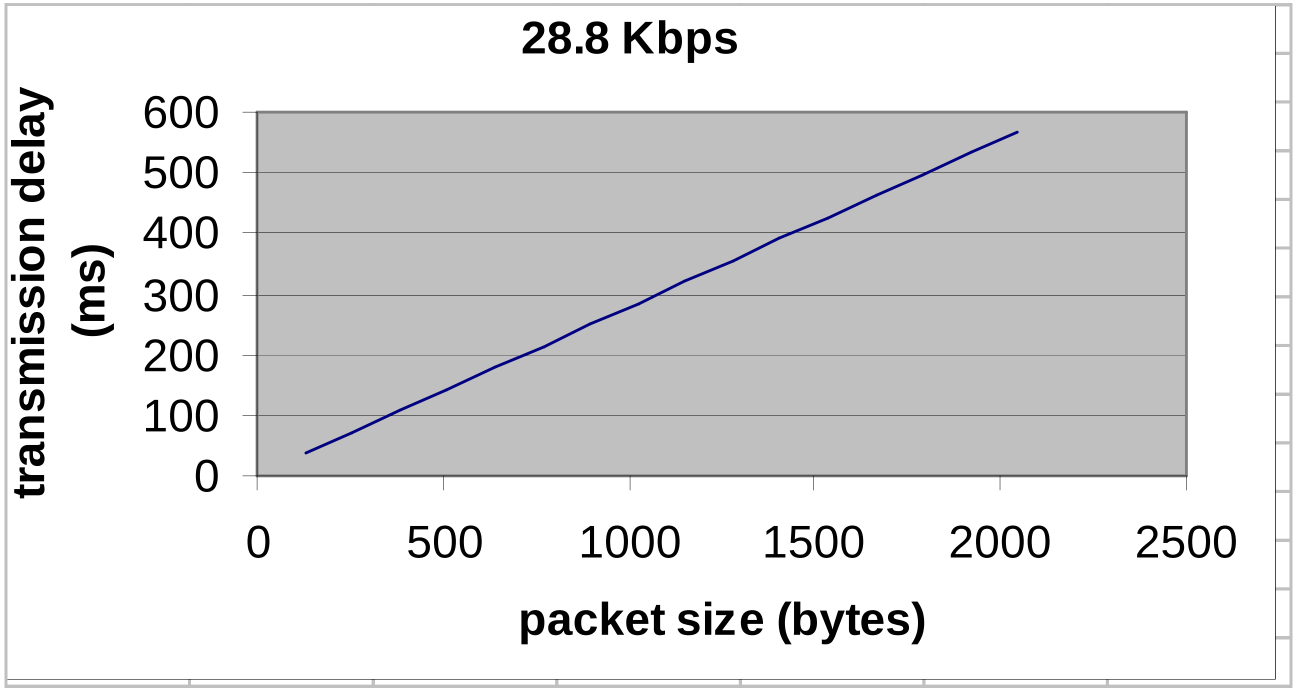
SCENARIO 2: mildly loaded short-distance communication



RTO = 2 x measured_RTT ~ 2 x 11 = 22 WAY TOO SMALL!

Problem: constant value β=2

SCENARIO 3:
slow speed links



Natural variation of packet sizes causes a large variation in RTT!

(from RFC 1122: utilization on 9.6 kbps link can improve from 10% up to 90%
With the adoption of Jacobson algorithm)

Jacobson RTO (1988)

idea: make it depend on measured variance!

$$Err = M - A$$

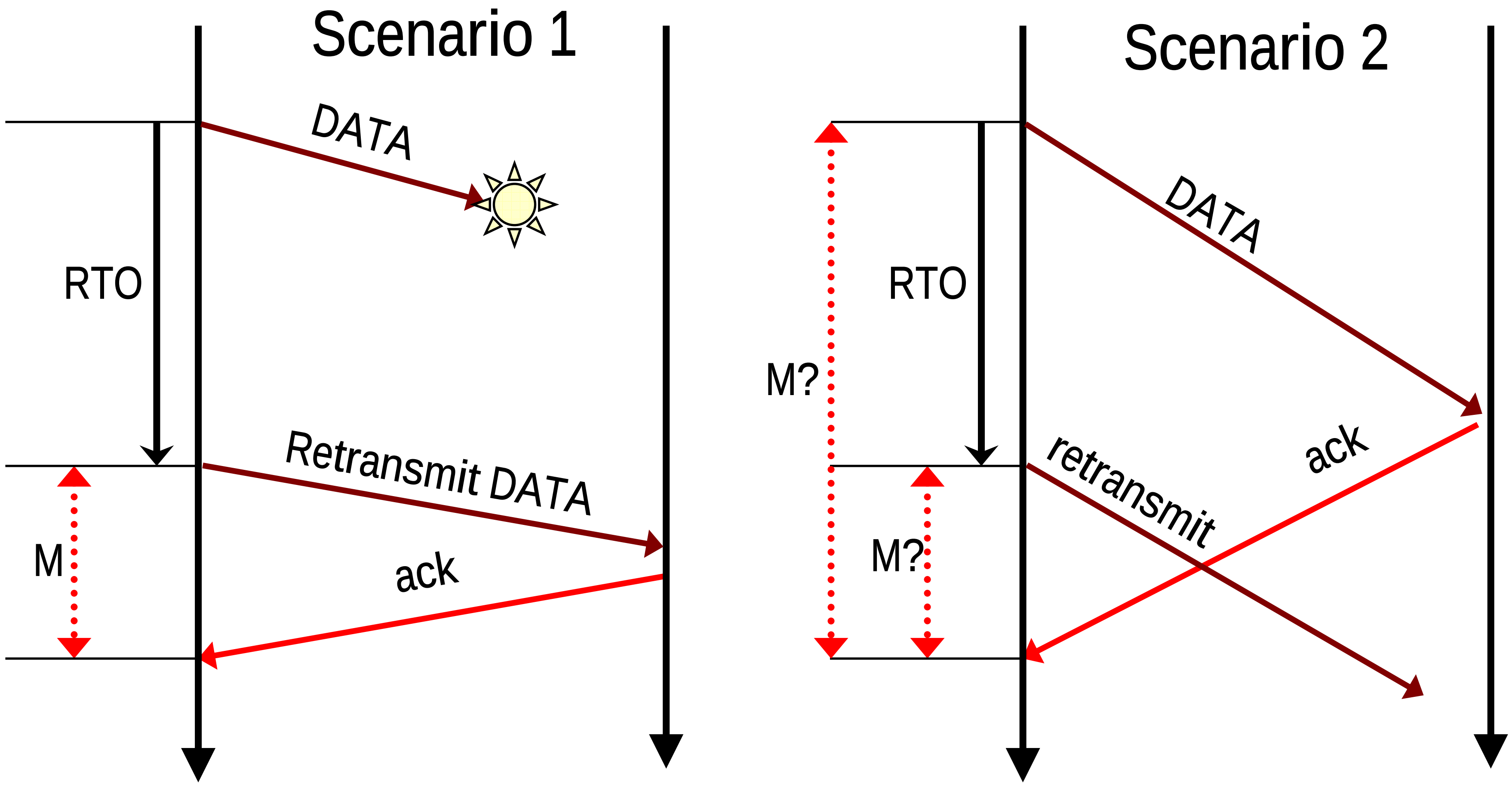
$$A := A + g\ Err$$

$$D := D + h\ (|Err| - D)$$

$$RTO = A + 4D$$

- ➔ **g = gain (1/8)**
  - ⇨ conceptually equivalent to 1-α, but set to slightly different value
- ➔ **D = mean deviation**
  - ⇨ conceptually similar to standard deviation, but cheaper (does not require a square root computation)
  - ⇨ h = 1/4
- ➔ **Jacobson's implementation: based on integer arithmetic (very efficient)**

Guessing right?
Karn's problem



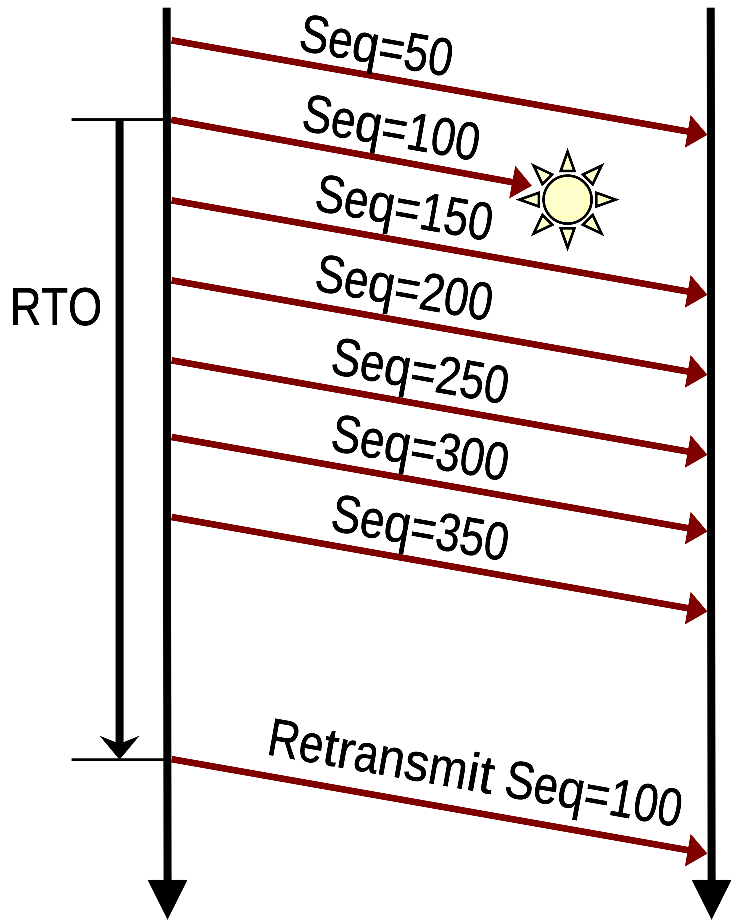
Solution to Karn's problem

- ➔ 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

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
  - ⇨ As well as following ones!!

- ➔ **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

➔ And then? A congestion control issue…