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"

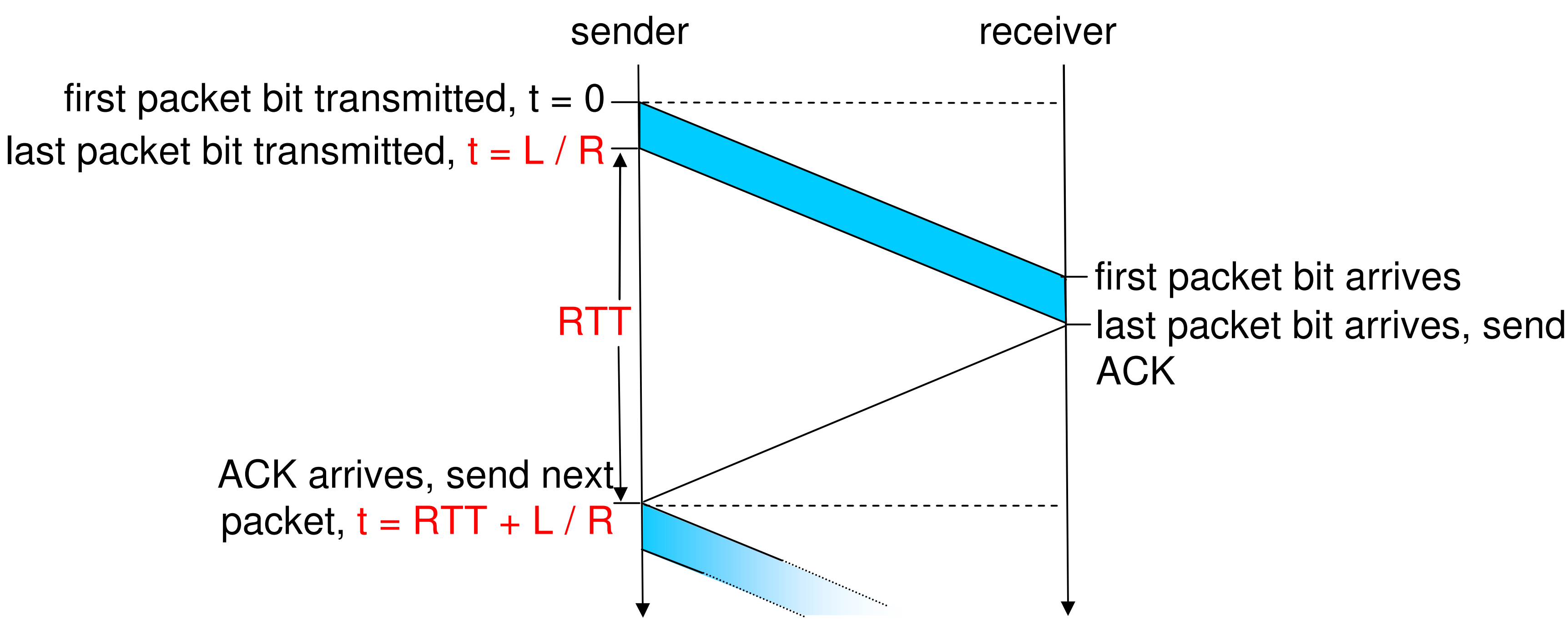
Canale A-L

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

rdt3.0: stop-and-wait operation

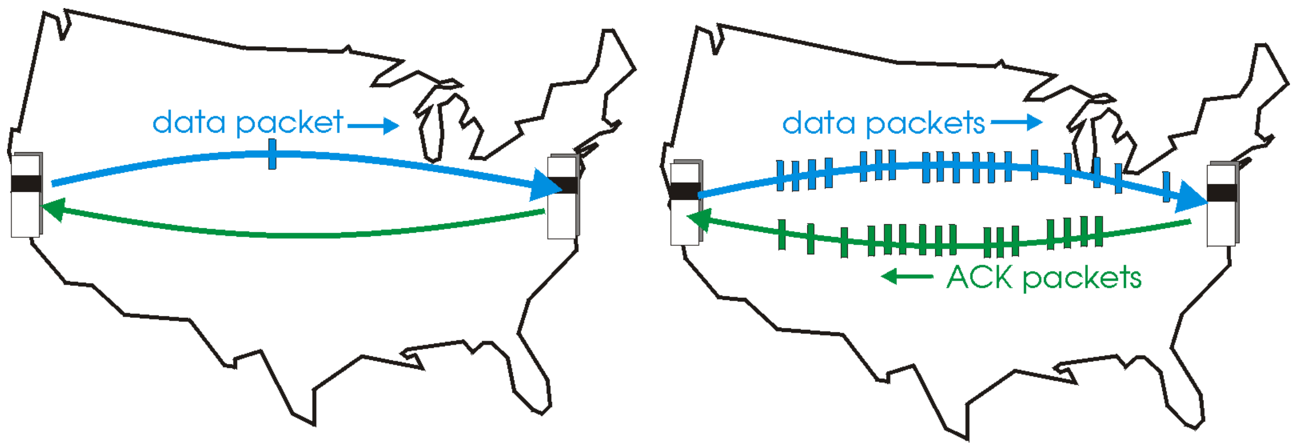


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

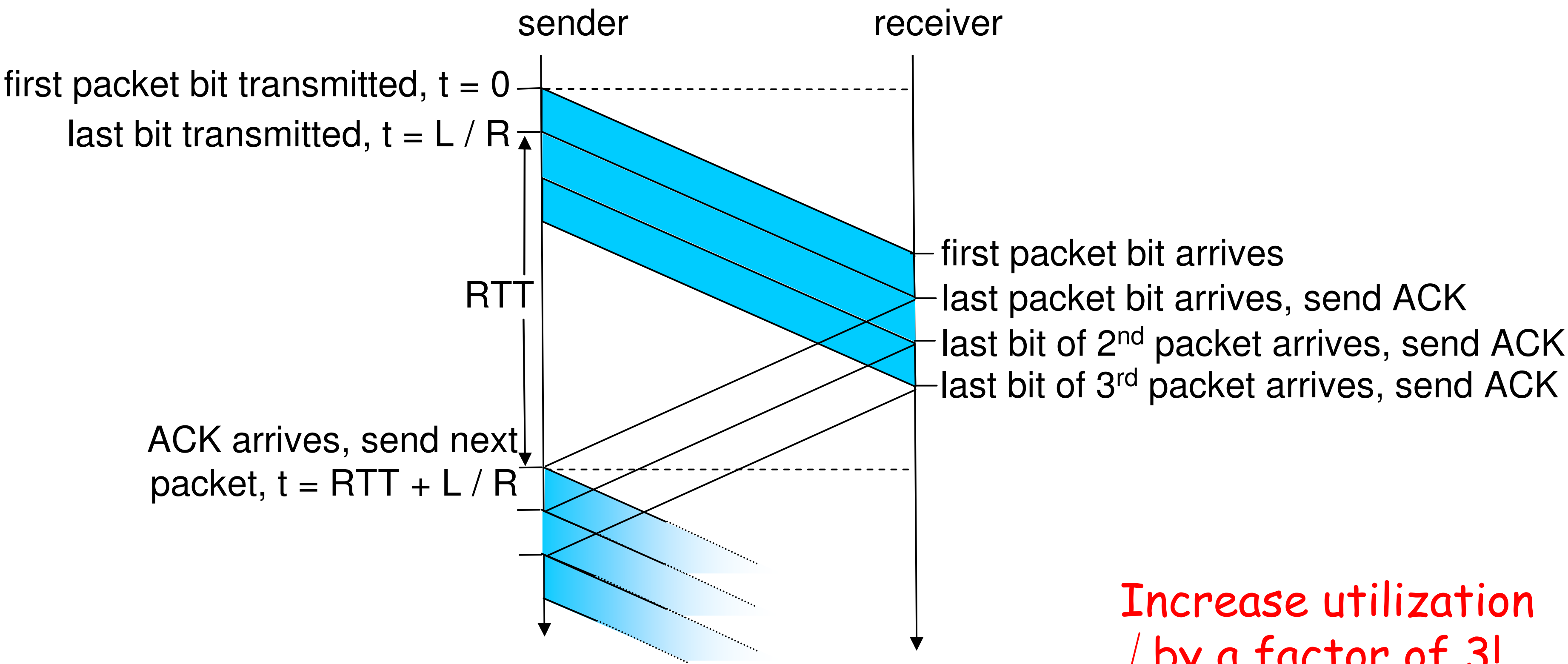
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



Increase utilization by a factor of 3!

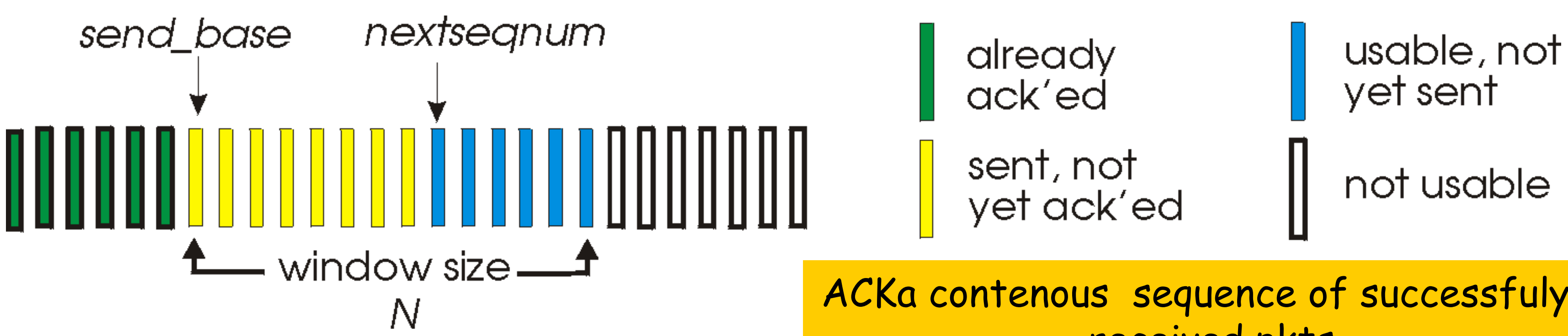
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

Diverso rispetto a Stop and Wait
Q: Perché?

Sender:

- **k-bit** seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



ACKa contenous sequence of successfulyy received pkts

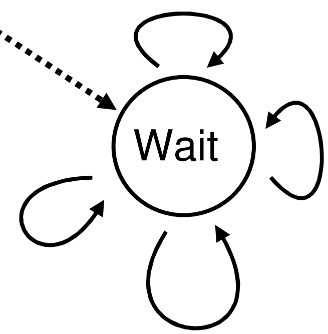
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM

```
rdt_send(data)
______________________________
if (nextseqnum < base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum)
     start_timer
   nextseqnum++
   }
else
  refuse_data(data)
```

Un solo timer in questa implementazione

```
Λ
______________
base=1
nextseqnum=1
```




```
timeout
______________________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

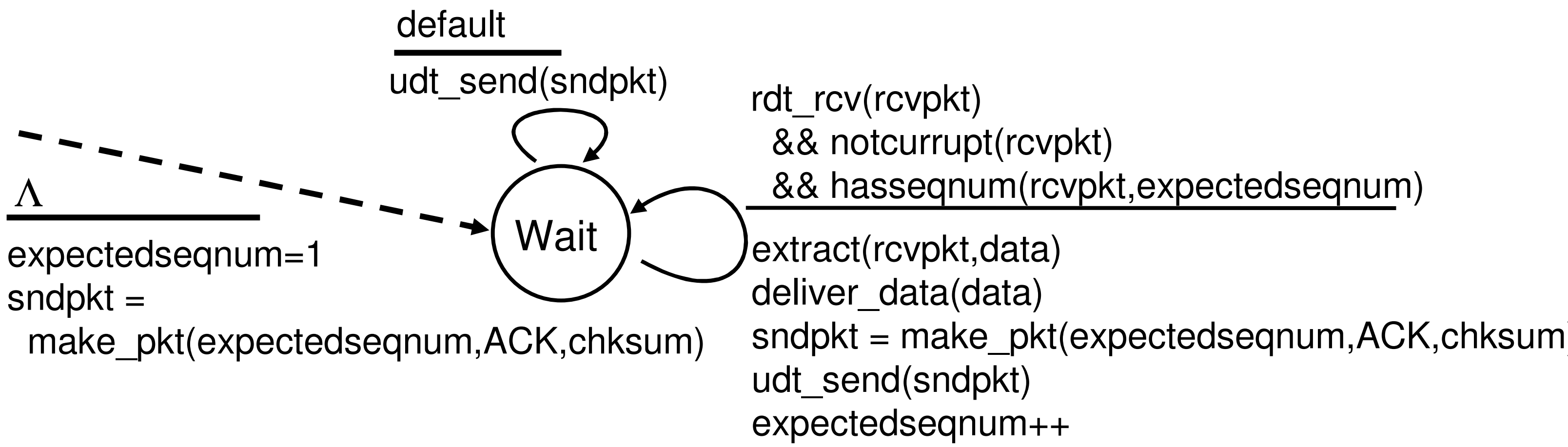
All retransmitted

```
rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)
______________________________
```

```
rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
______________________________
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
   stop_timer
 else
   start_timer
```

Gestione timer

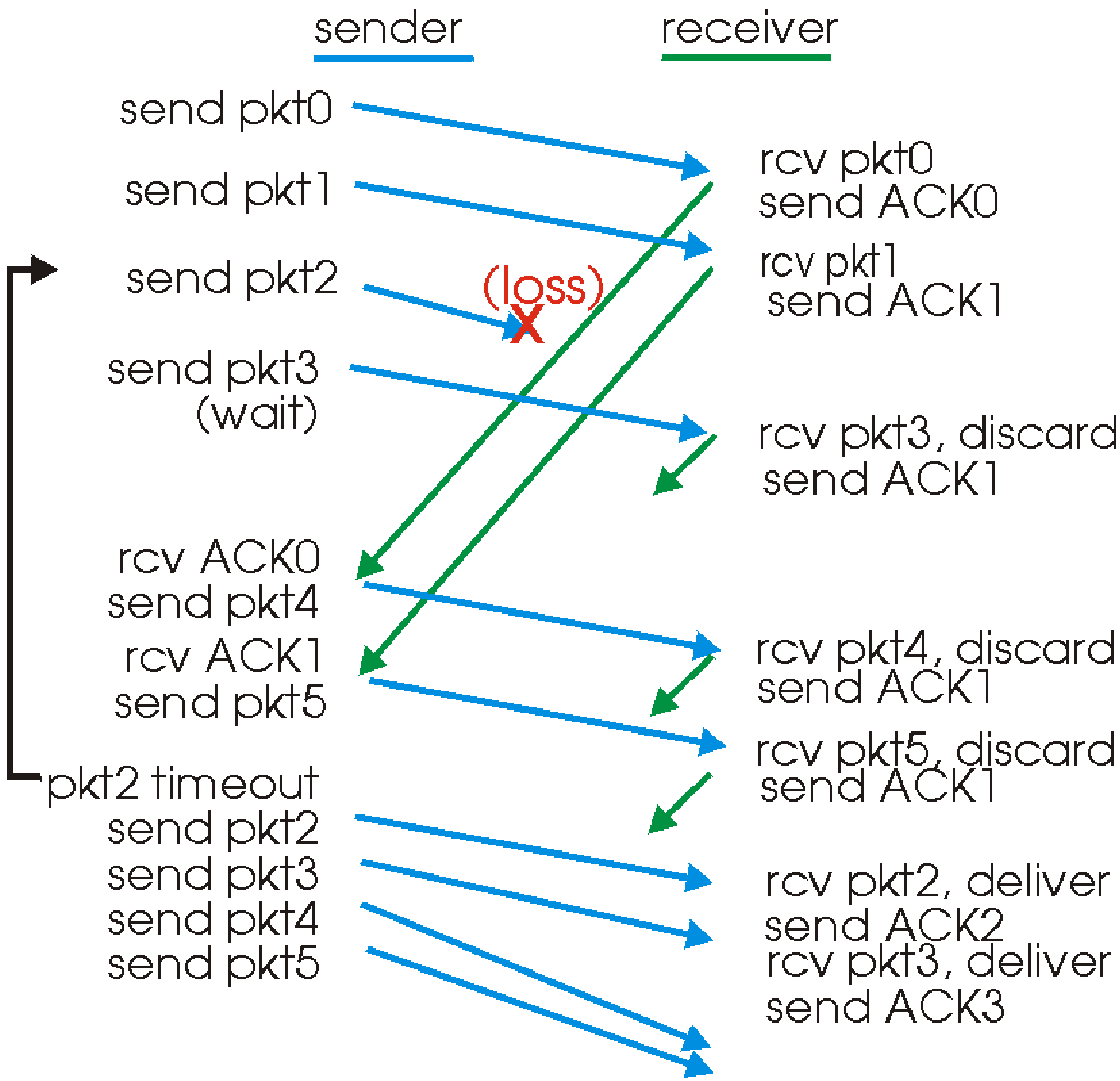
GBN: receiver extended FSM



ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

GBN in action



WINDOW SIZE ==4

A few questions…

- Why limiting the window size?
  - max window size to improve performance related to RTT
  - window size powerful tool to control data rate (important for flow control, con gestion control)
  - related to window size field length

ADD

Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

Each packet has one Logical timer

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]

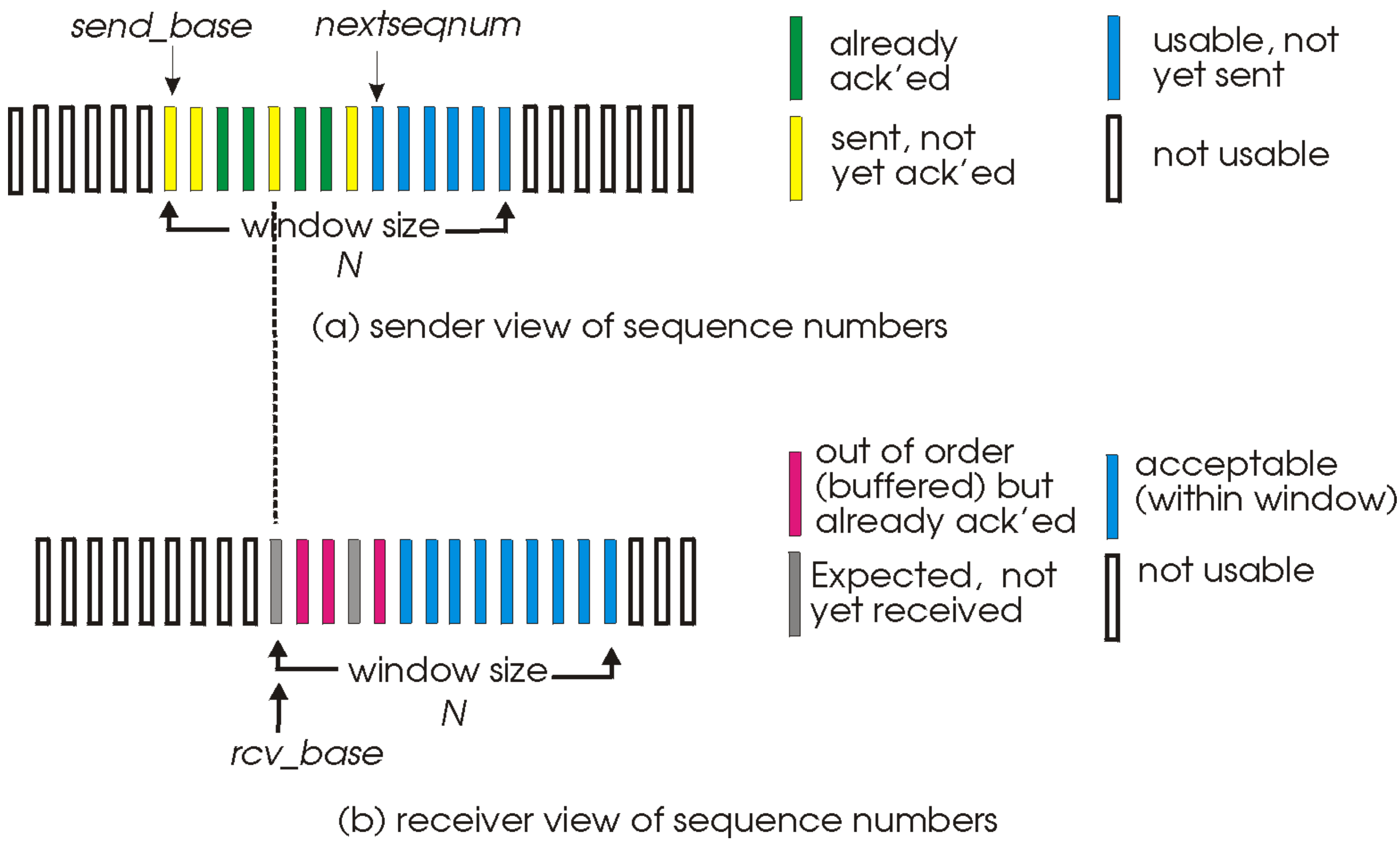
- ACK(n)

otherwise:

- ignore

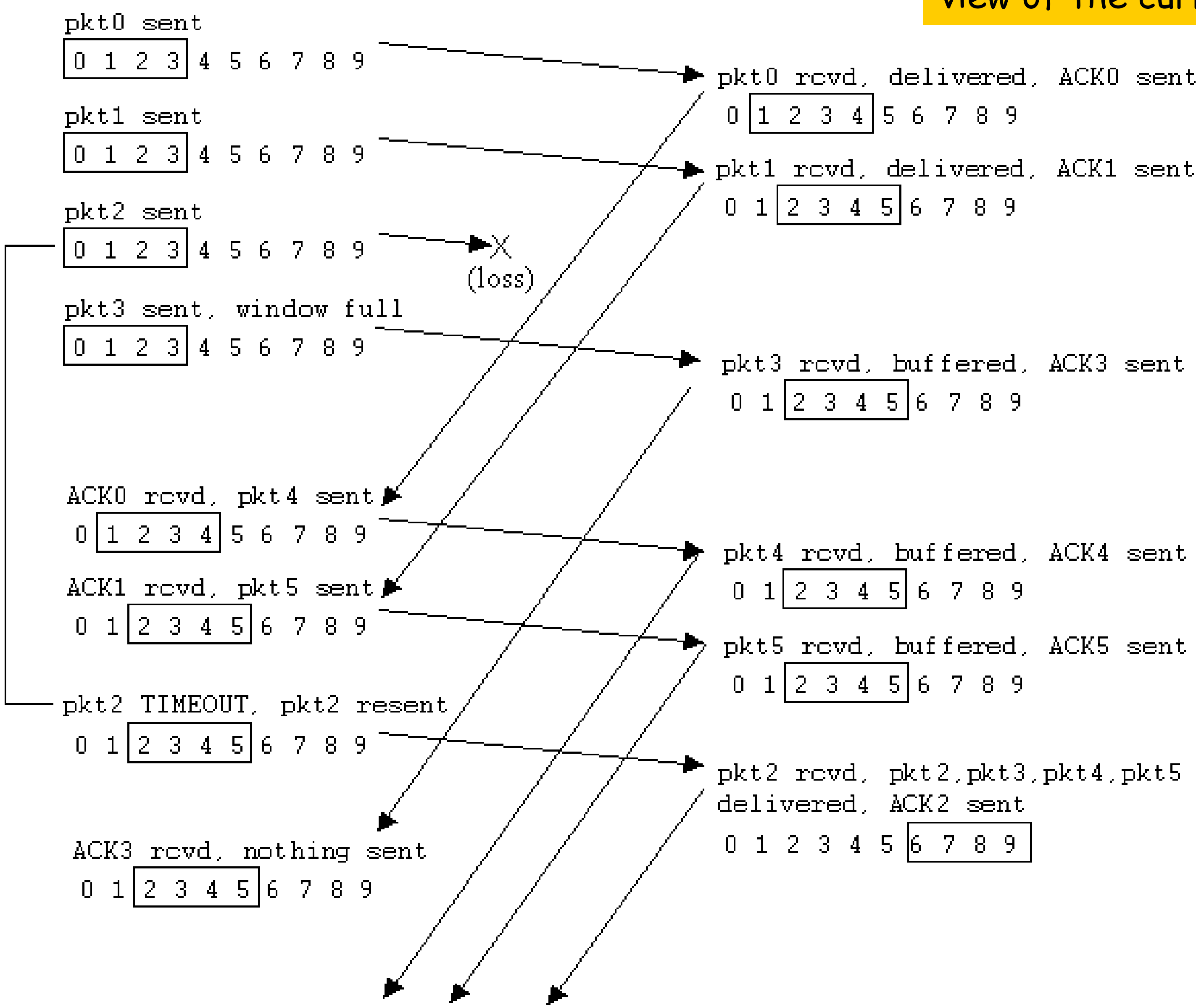
Important!! Sender and receiver may have different views!!

Selective repeat: sender, receiver windows



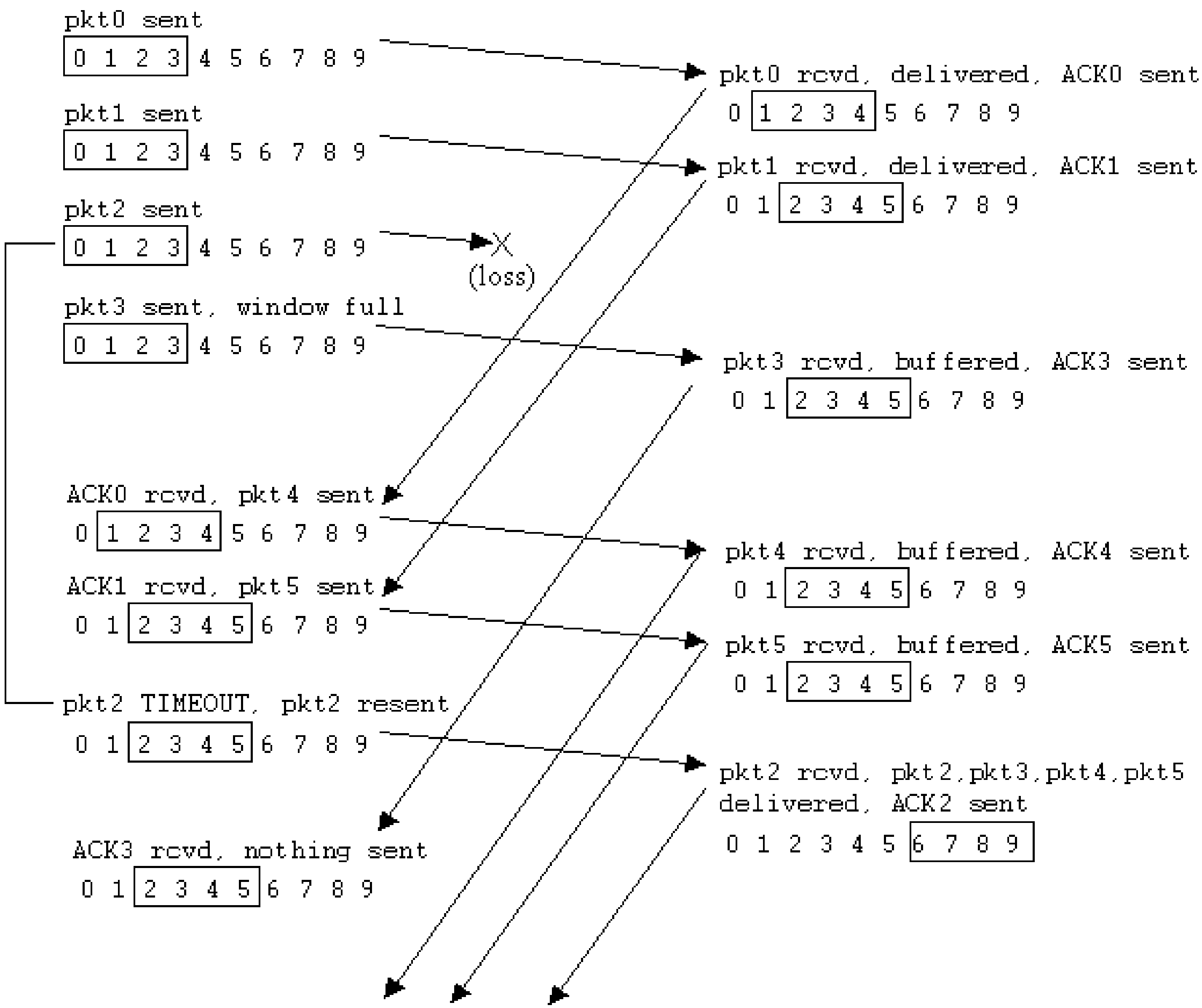
Selective repeat in action

Transmitter and receiver can have different view of the current windoow



Selective repeat in action

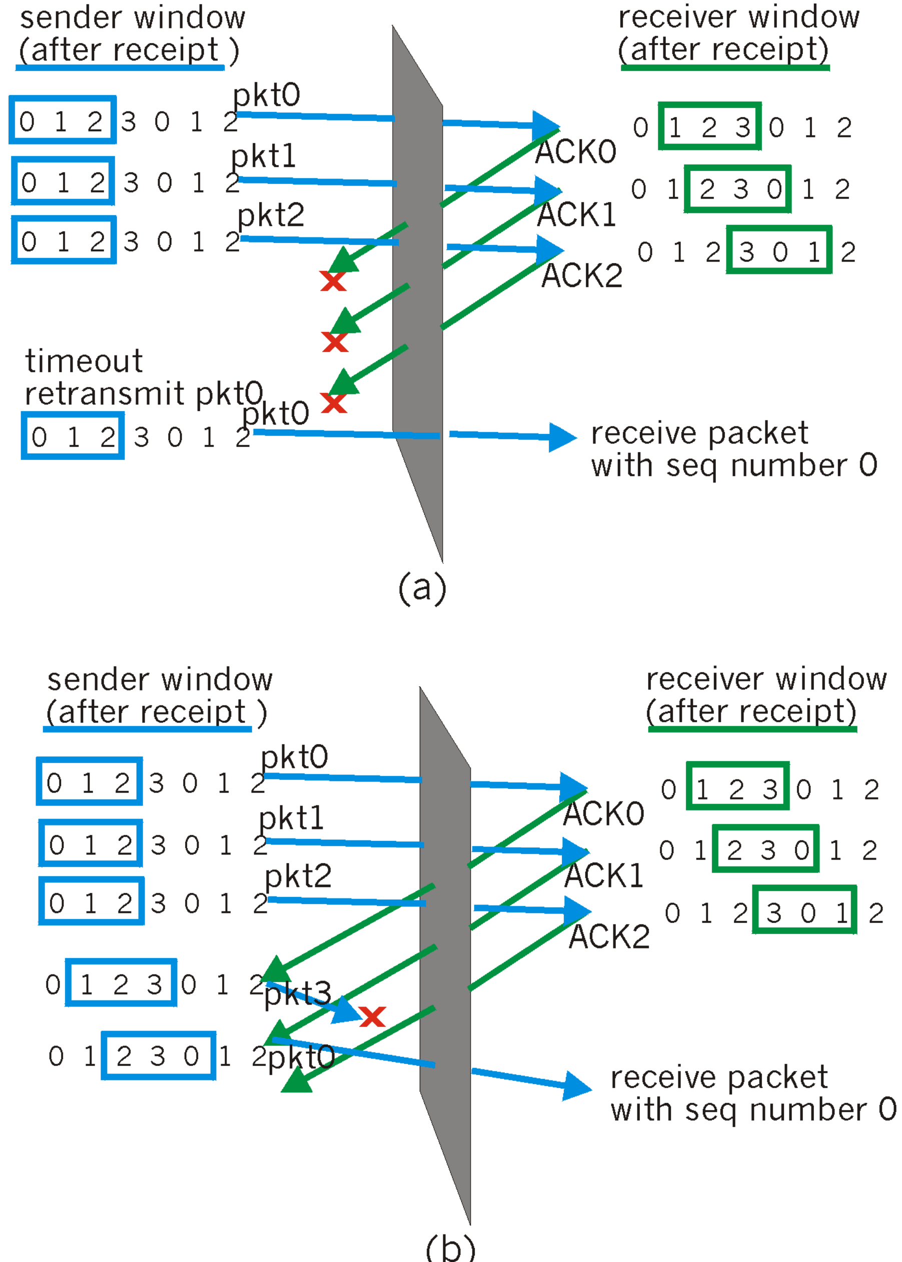
1) Delays in receiving ACKs
2) lost ACKs



Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3



- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

Clearly at least the window must be small enough so that there is not ambiguity on sequence numbers!!! Is it enough in Selective Repeat??

Answer to the dilemma

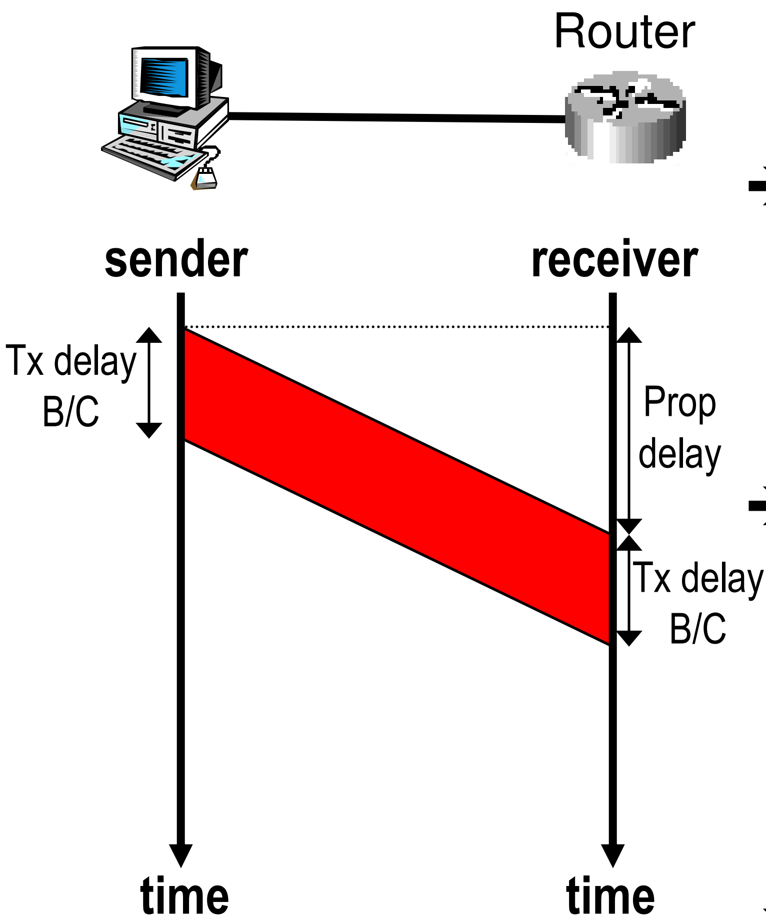
- The window size must be less than or equal to half the size of the sequence number space for SR protocols

Another issue

- When ARQ solutions are applied at transport layer packets traverse not only one link but a path. Packets may arrive not in order, old packets may arrive with a long delay→in that case the answer is more involved. We cannot reuse a sequence number unless we are sure that old packets carrying that sequence number are out of the network (limit on the packet lifetime).

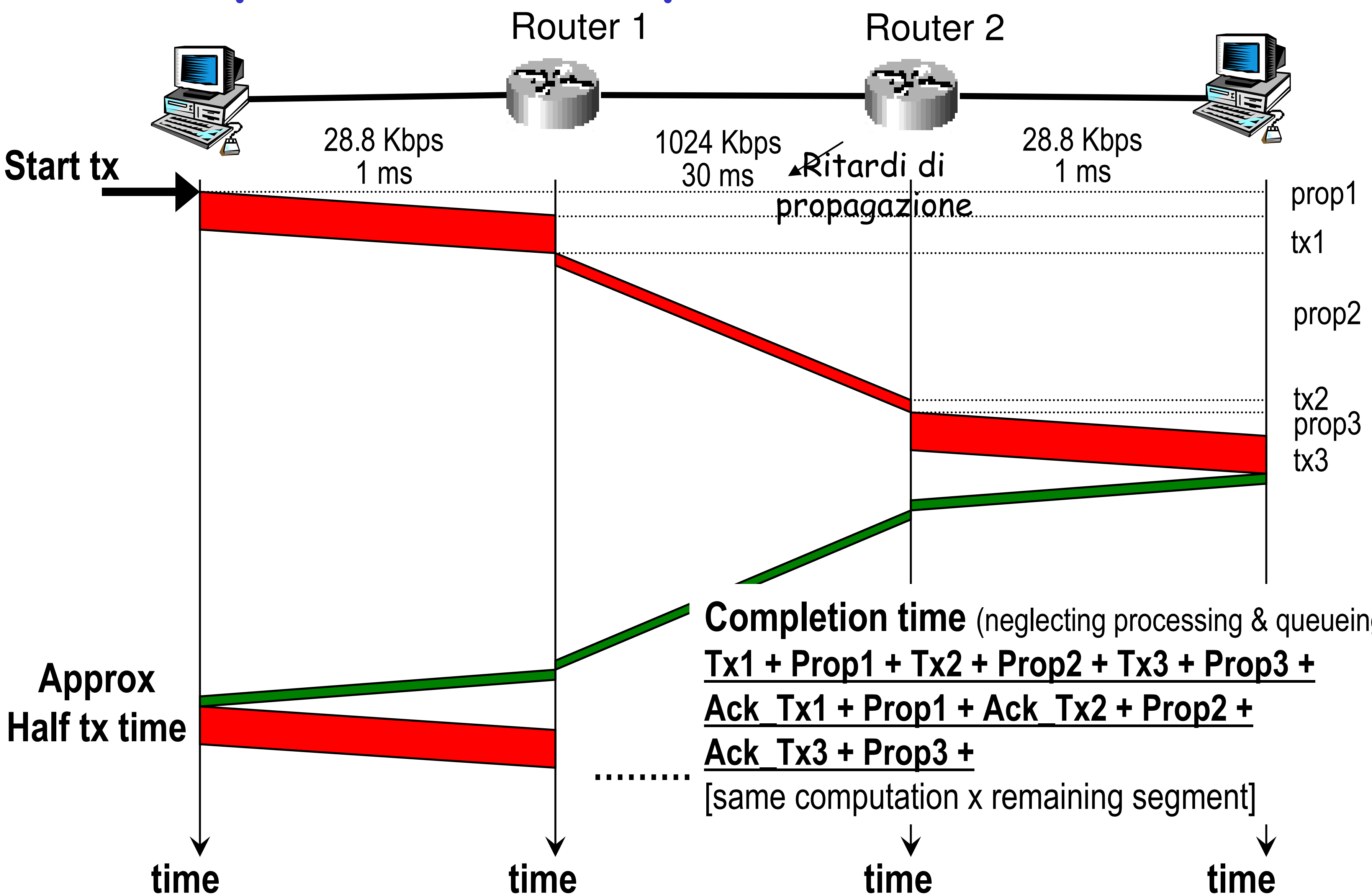
Performance issues with/without pipelining

Link delay computation



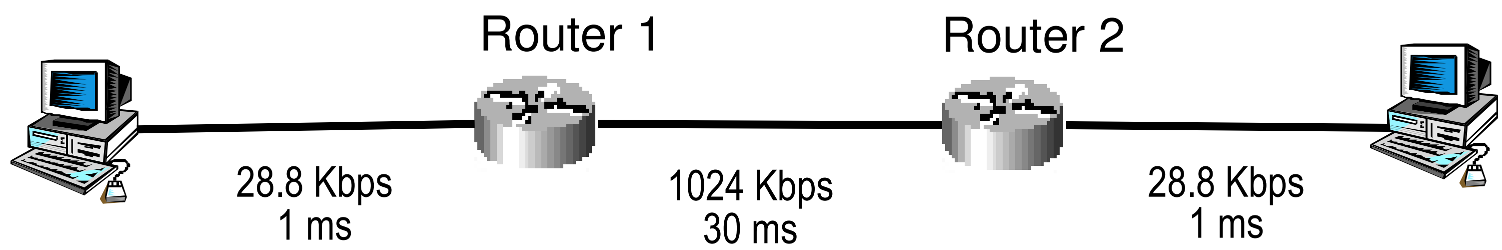
- Transmission delay:
  - C [bit/s] = link rate
  - B [bit] = packet size
  - transmission delay = B/C [sec]
- Example:
  - 512 bytes packet
  - 64 kbps link
  - transmission delay = 512*8/64000 = 64ms
- Propagation delay – constant depending on
  - Link length
  - Electromagnetig waves propagation speed in considered media
    - 200 km/s for copper links
    - 300 km/s in air
- other delays neglected
  - Queueing
  - processing

Stop-and-wait performance



Completion time (neglecting processing & queueing) =
Tx1 + Prop1 + Tx2 + Prop2 + Tx3 + Prop3 +
Ack_Tx1 + Prop1 + Ack_Tx2 + Prop2 +
Ack_Tx3 + Prop3 +
[same computation x remaining segment]

Stop-and-wait performance
Numerical example



- Message:
  - 1024 bytes;
  - 2 segments: 536+488 bytes
  - Overhead: 20 bytes TCP + 20 bytes IP
  - ACK = 40 bytes (header only)

Lower layer headers not considered

➔ **Segment 1:**

- ⇨ Tx1 = 576*8/28,8 = 160ms
- ⇨ Tx3 = Tx1
- ⇨ Tx2 = 576*8/1024 = 4,5 ms

➔ **Segment 2:**

- ⇨ Tx1 = 528*8/28,8 = 146,7ms
- ⇨ Tx3 = Tx1
- ⇨ Tx2 = 528*8/1024 = 4,1 ms

➔ **Acks:**

- ⇨ Tx1 = Tx3 = 40*8/28,8 = 11,1ms
- ⇨ Tx2 = 40*8/1024 = 0,3 ms

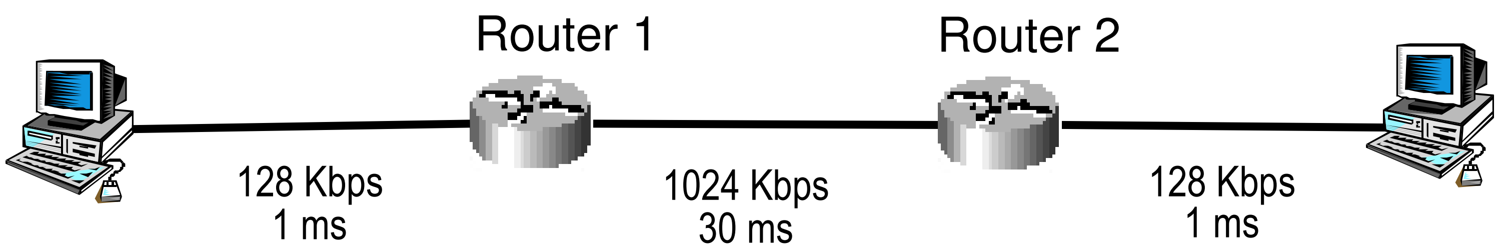
RESULT:

D = 667 (tx total) + 2*RTT = = 795 ms

THR = 1024*8/795 = = 10,3 kbps

Stop-and-wait performance
Numerical example

With ISDN?



- **Segment 1:**
  - Tx1 = Tx3 = 576*8/128 = 36ms
  - Tx2 = 576*8/1024 = 4,5 ms
- **Segment 2:**
  - Tx1 = Tx3 = 528*8/128 = 33 ms
  - Tx2 = 528*8/1024 = 4,1 ms
- **Acks:**
  - Tx1 = Tx3 = 40*8/128 = 2,5 ms
  - Tx2 = 40*8/1024 = 0,3 ms

RESULT:

D = 151,9 (tx total) + 2*RTT = = 279,9 ms

THR = 1024*8/279,9 = = 29,3 kbps

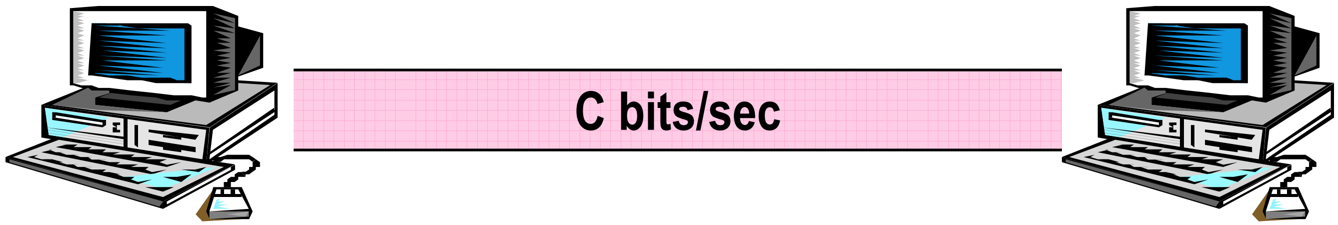
on Gbps fiber optics?

D = negligible + 2*RTT = = 128 ms

THR = 1024*8/128 = = 64 kbps

MA E' VERAMENTE MEGLIO AD ALTO DATA RATE? NO —DI QUI A POCO…

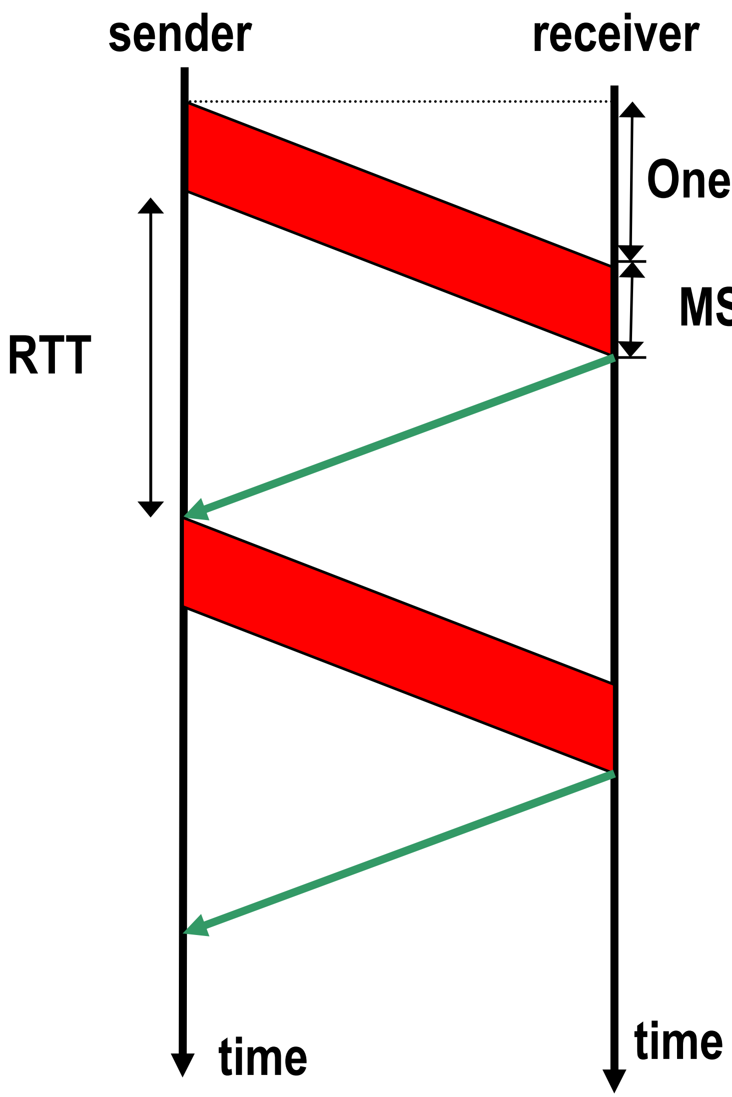
Simplified performance model



Approximate analysis, much simpler than multi-hop
Typically, C = bottleneck link rate

MSS = segment size (ev. ignore overhead)
MSIZE = message size
Ignore ACK transmission time
No loss of segments
W = number of outstanding segments
- W=1: stop-and-wait
- W>1: go-back-N (sliding window)
- *This is a highly dynamic parameter in TCP!!*
- *For now, consider W fixed*

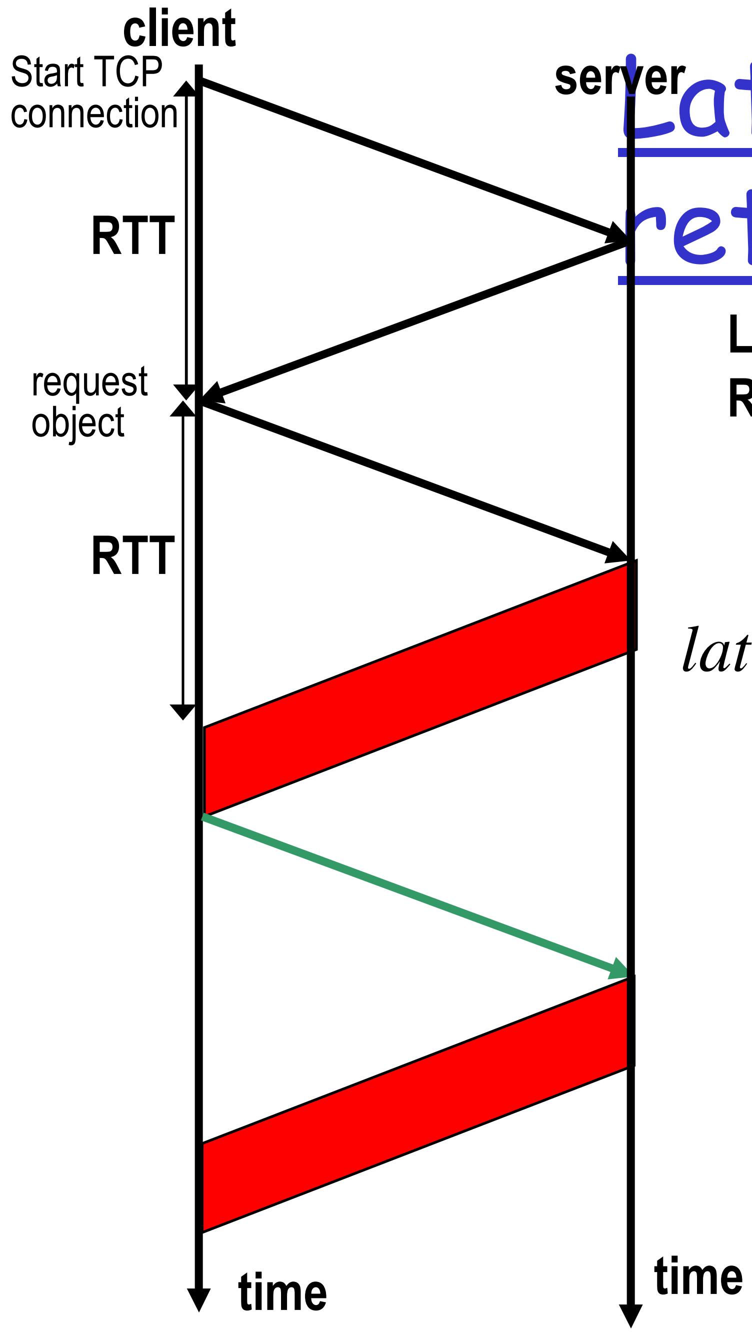
W=1 case (stop-and-wait)



$$throughput = \frac{MSS}{RTT + MSS / C}$$

REMARK: throughput always lower than Available link rate!

Latency in TCP retrieval model



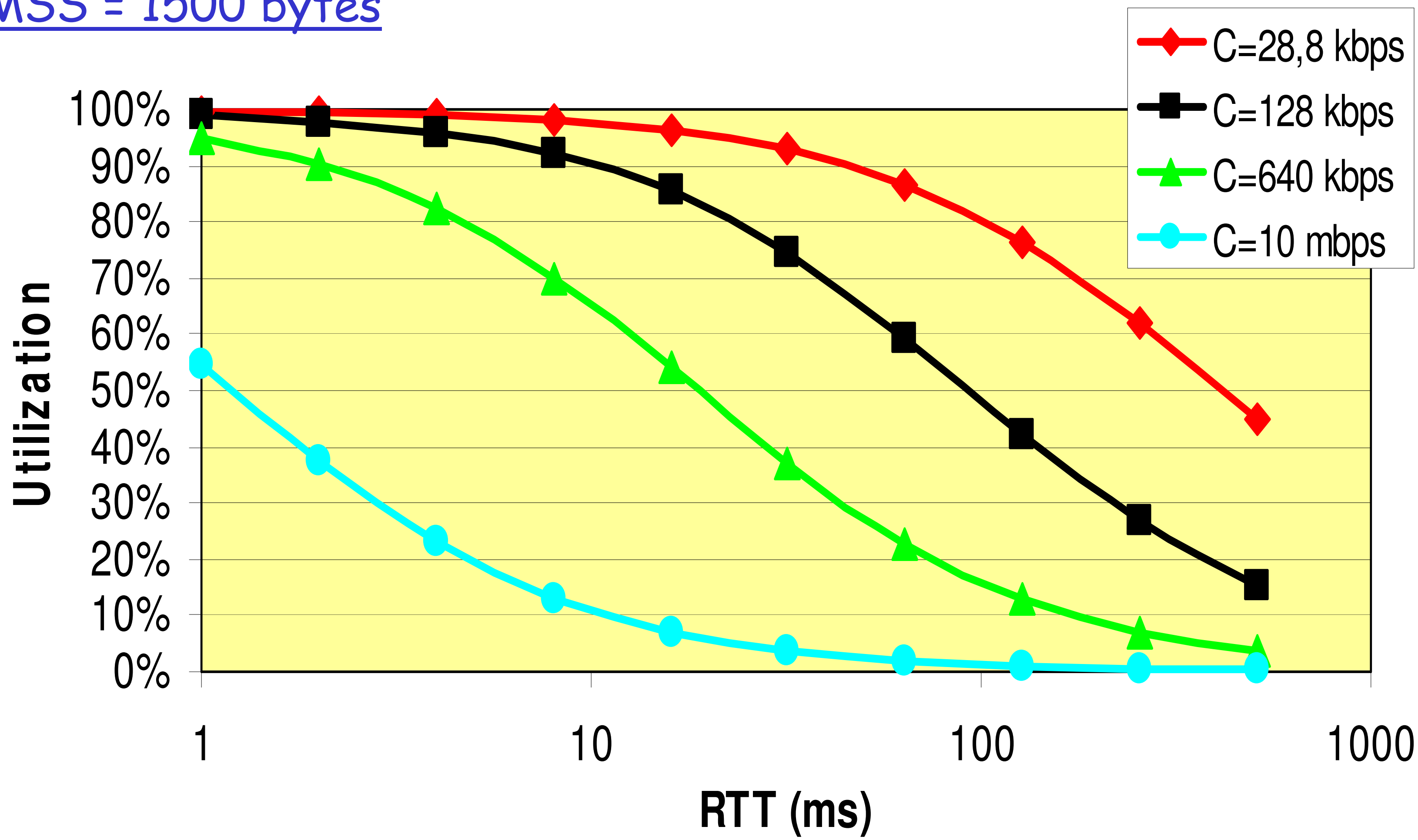
Latency: time elapsing between TCP connection Request, and last bit received at client

$$latency = 2RTT + \frac{MSIZE}{C} + \left\lceil \frac{MSIZE}{MSS} - 1 \right\rceil RTT$$

Number of segments In which message is split

W=1 case (stop-and-wait)

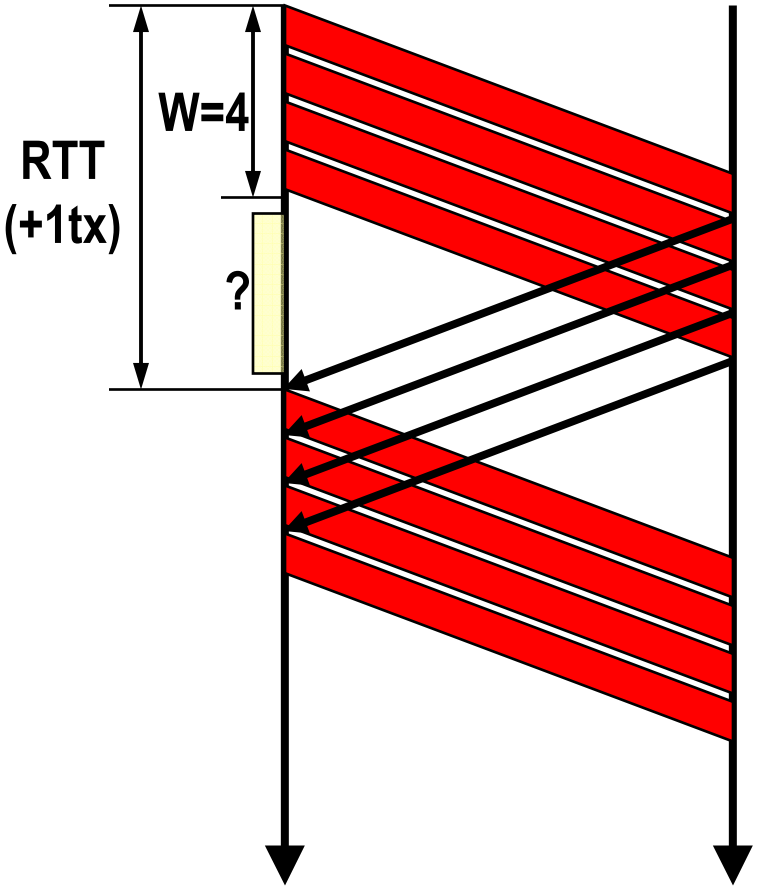
MSS = 1500 bytes



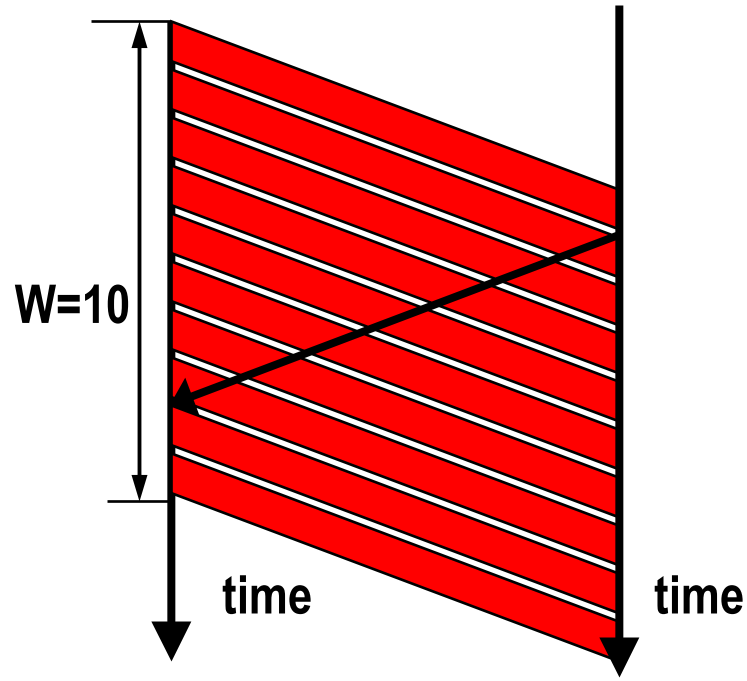
Under-utilization with: 1) high capacity links, 2) large RTT links

Pipelining (W>1) analysis

two cases



UNDER-SIZED WINDOW:
THROUGHPUT INEFFICIENCY



WINDOW SIZING that allows
CONTINUOUS TRANSMISSION

Continuous transmission

Condition in which link rate is fully utilized

$$\underbrace{W \cdot \frac{MSS}{C}}_{\text{Time to transmit W segments}} > \underbrace{RTT + \frac{MSS}{C}}_{\text{Time to receive Ack of first segment}}$$

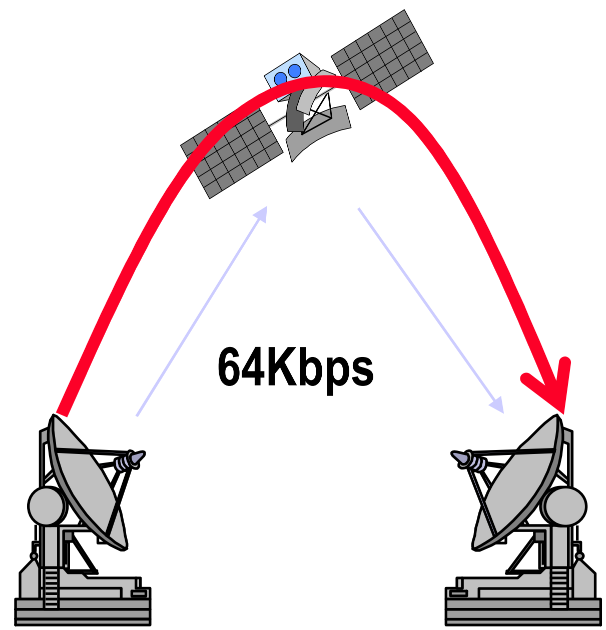
We may elaborate:

$$W \cdot MSS > RTT \cdot C + MSS \approx RTT \cdot C$$

This means that full link utilization is possible when window size (in bits) is Greater than the bandwidth (C bit/s) delay (RTT s) product!

Bandwidth-delay product

- ➔**Network: like a pipe**
- ➔**C [bit/s] x D [s]**
  - ⇨ number of bits “flying” in the network
  - ⇨ number of bits injected in the network by the tx, before that the first bit is rxed



A 15360 (64000x0.240) bits “worm” in the air!!

bandwidth-delay product = no of bytes that saturate network pipe

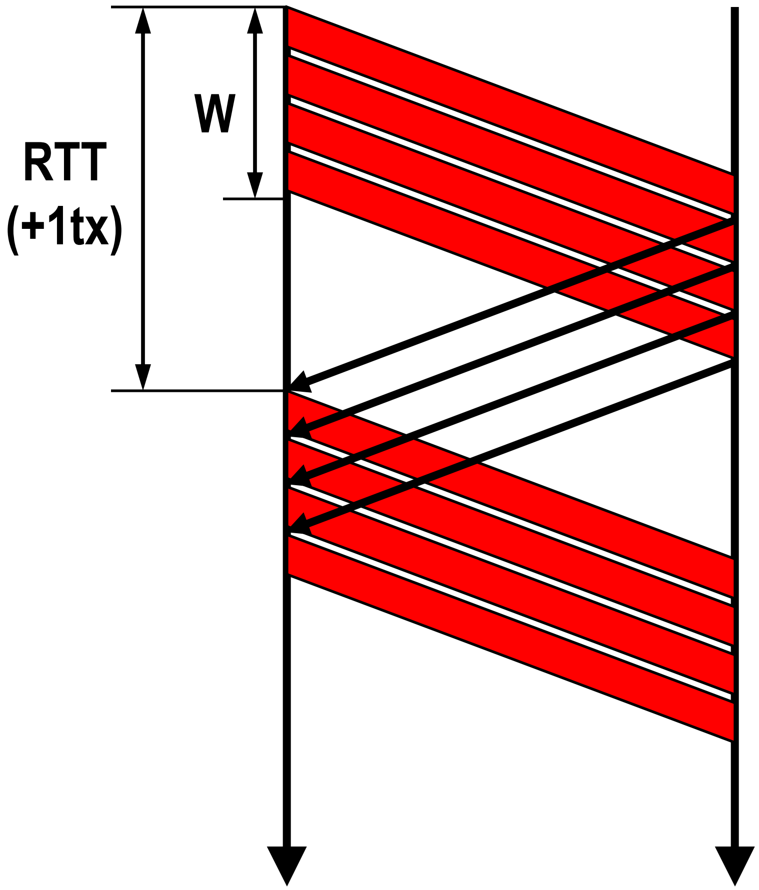
Long Fat Networks

LFNs (el-ef-an(t)s): large bandwidth-delay product

NETWORK	RTT (ms)	rate (kbps)	BxD (bytes)
Ethernet	3	10.000	3.750
T1, transUS	60	1.544	11.580
T1 satellite	480	1.544	92.640
T3 transUS	60	45.000	337.500
Gigabit transUS	60	1.000.000	7.500.000

The 65535 (16 bit field in TCP header) maximum window size W may be a limiting factor!

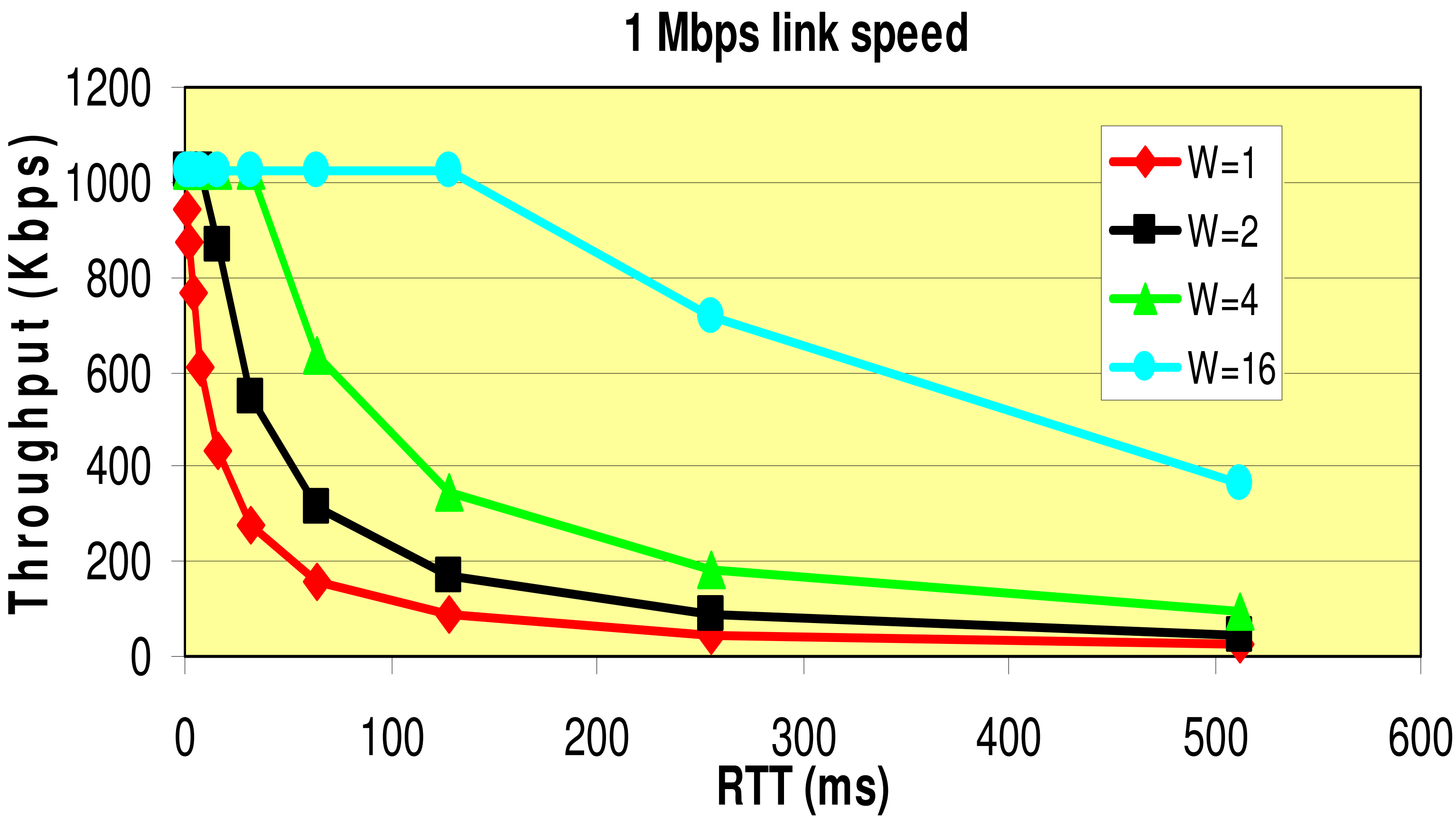
Pipelining (W>1) analysis



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

Throughput for pipelining

MSS = 1500 bytes



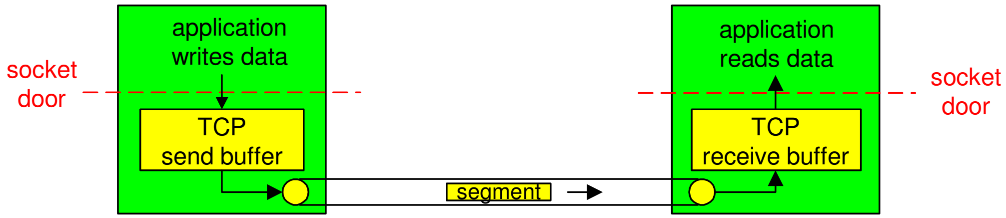
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP: Overview

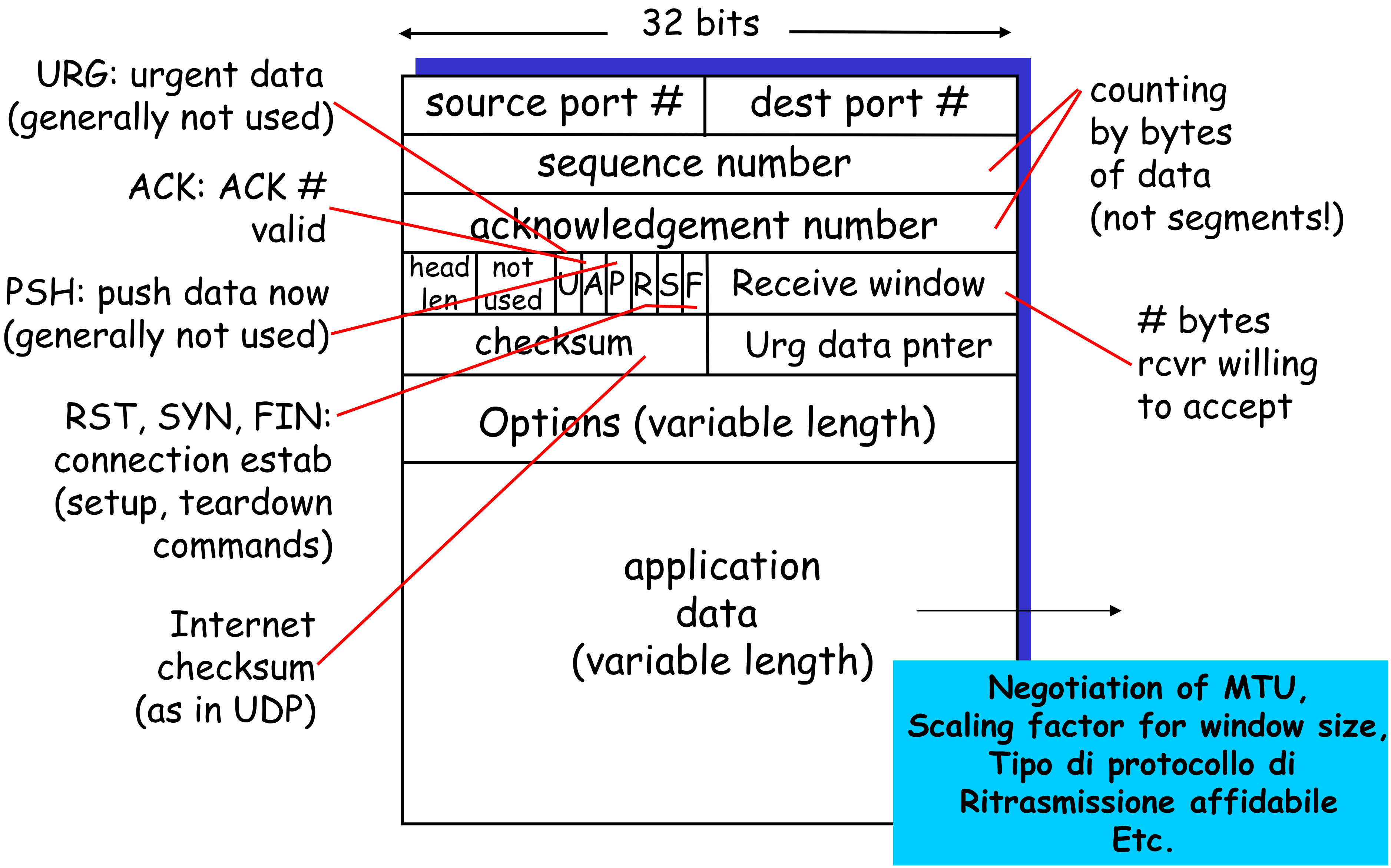
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- **reliable, in-order *byte steam:***
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



- full duplex data:
  - **bi-directional data flow in same connection**
  - **MSS: maximum segment size**
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



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!!

<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>

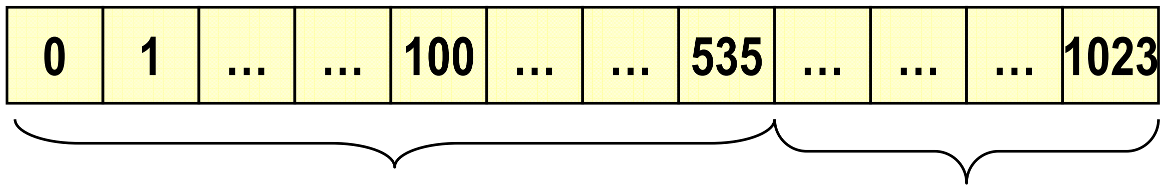
- Sequence number:
  - Sequence number of the ***first*** byte in the segment.
  - When reaches 2^{32}-1, next wraps back to 0
- Acknowledgement number:
  - valid only when ACK flag on
  - Contains the ***next*** byte sequence number that the host ***expects*** to receive (= last successfully received byte of data + 1)
  - grants successful reception for <u>all bytes</u> up to ack# - 1 (cumulative)
- When seq/ack reach 2^{32}-1, next wrap back to 0

TCP data transfer management

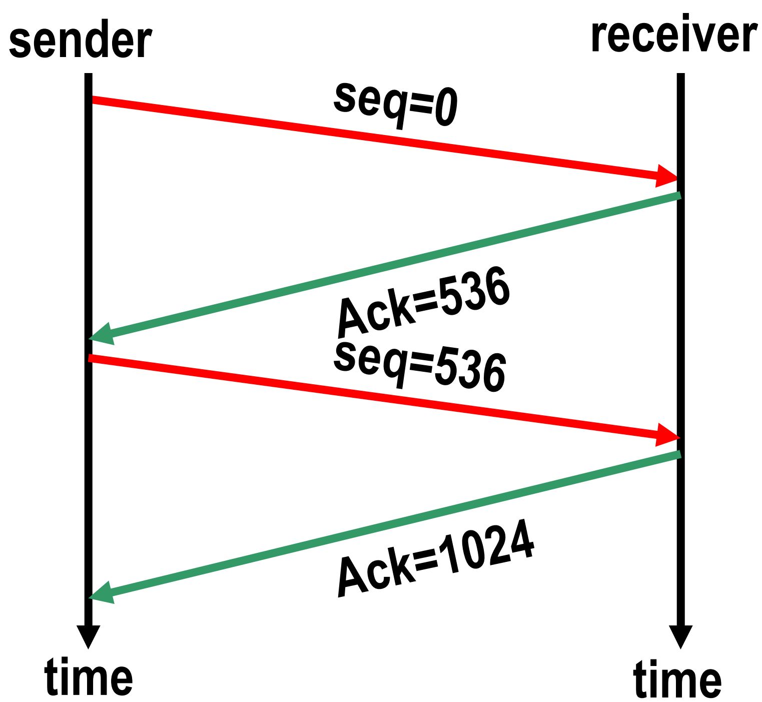
- Full duplex connection
  - data flows in both directions, independently
  - To the application program these appear as two unrelated data streams
- each end point maintains a sequence number
  - Independent sequence numbers at both ends
  - Measured in bytes
- acks often carried on top of reverse flow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

Byte-oriented

Example: 1 Kbyte message – 1024 bytes



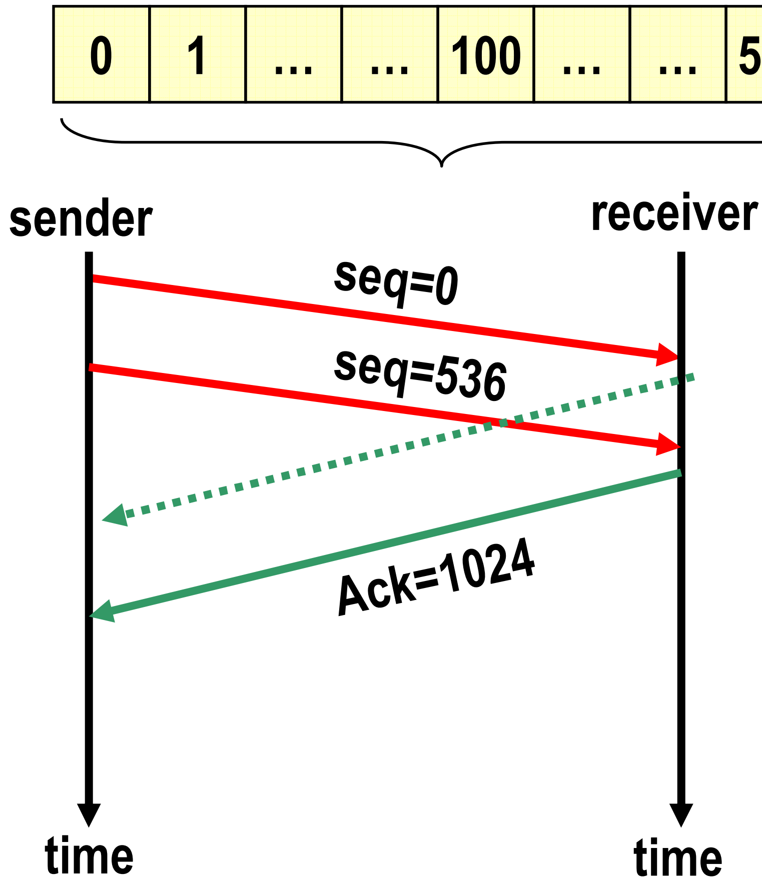
Example: segment size = 536 bytes → 2 segments: 0-535; 536-1023



- ➔ **No explicit segment size indication**
  - ⇨ Seq = first byte number
  - ⇨ Returning Ack = last byte number + 1
  - ⇨ Segment size = Ack-seq#

Pipelining - cumulative ack

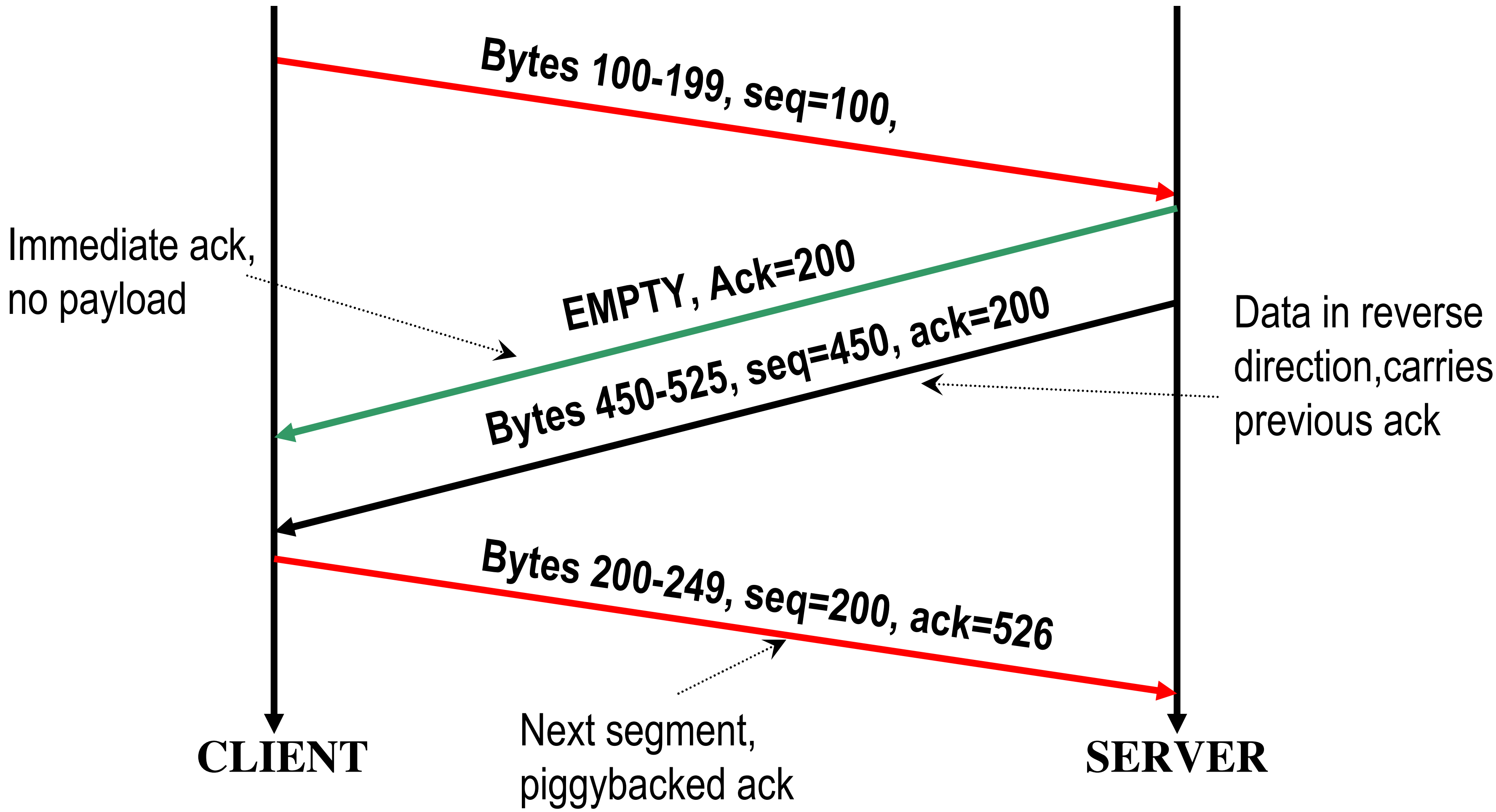
Example: 1024 bytes msg; seg_size = 536 bytes → 2 segments: 0-535; 536-1023



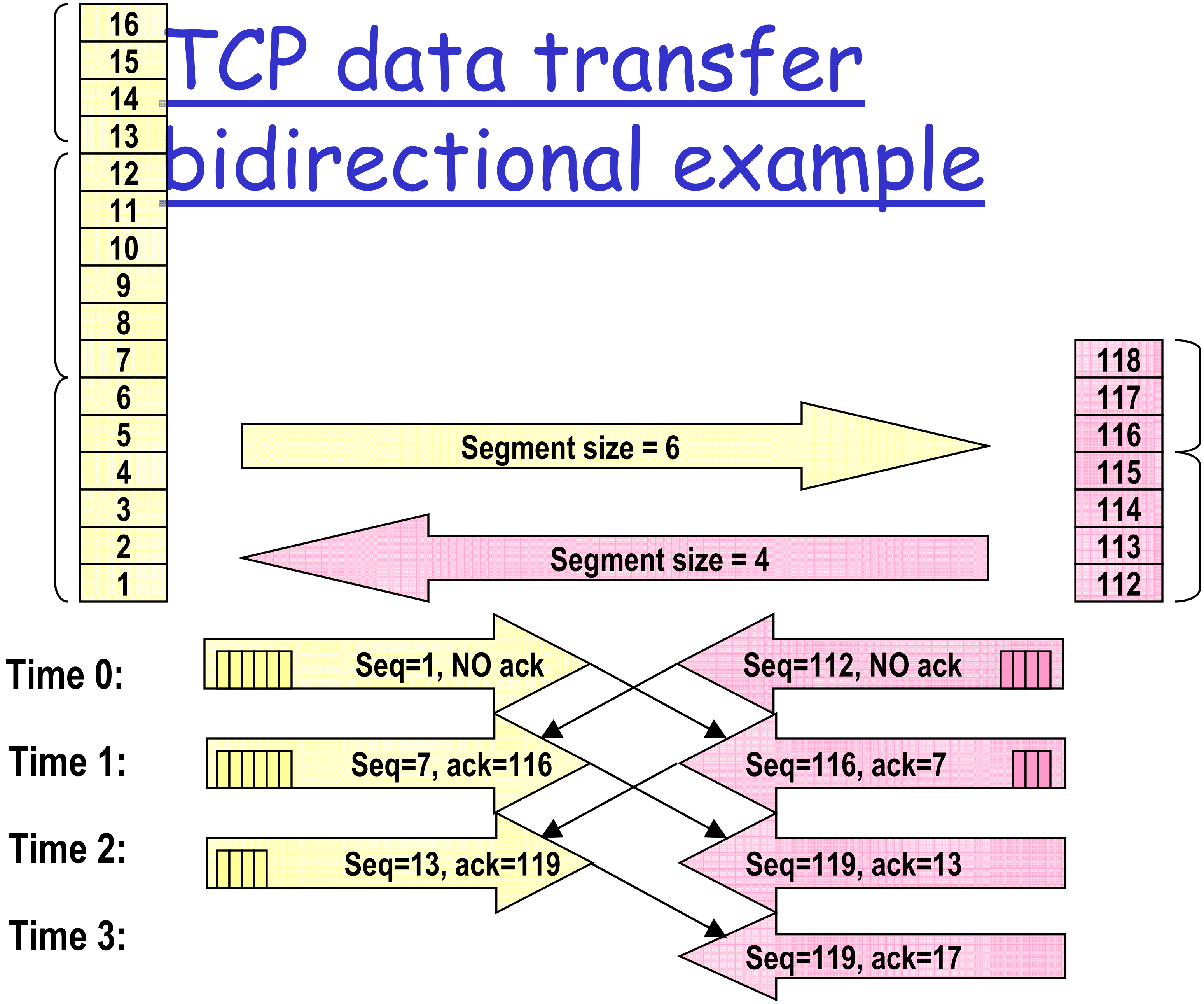
- **Cumulative ack**
  - ACK = **all** previous bytes correctly received!
  - E.g. ACK=1024: all bytes 0-1023 received
- Other names of pipelining:
  - Go-Back-N ARQ mechanisms
  - Sliding window mechanisms

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:

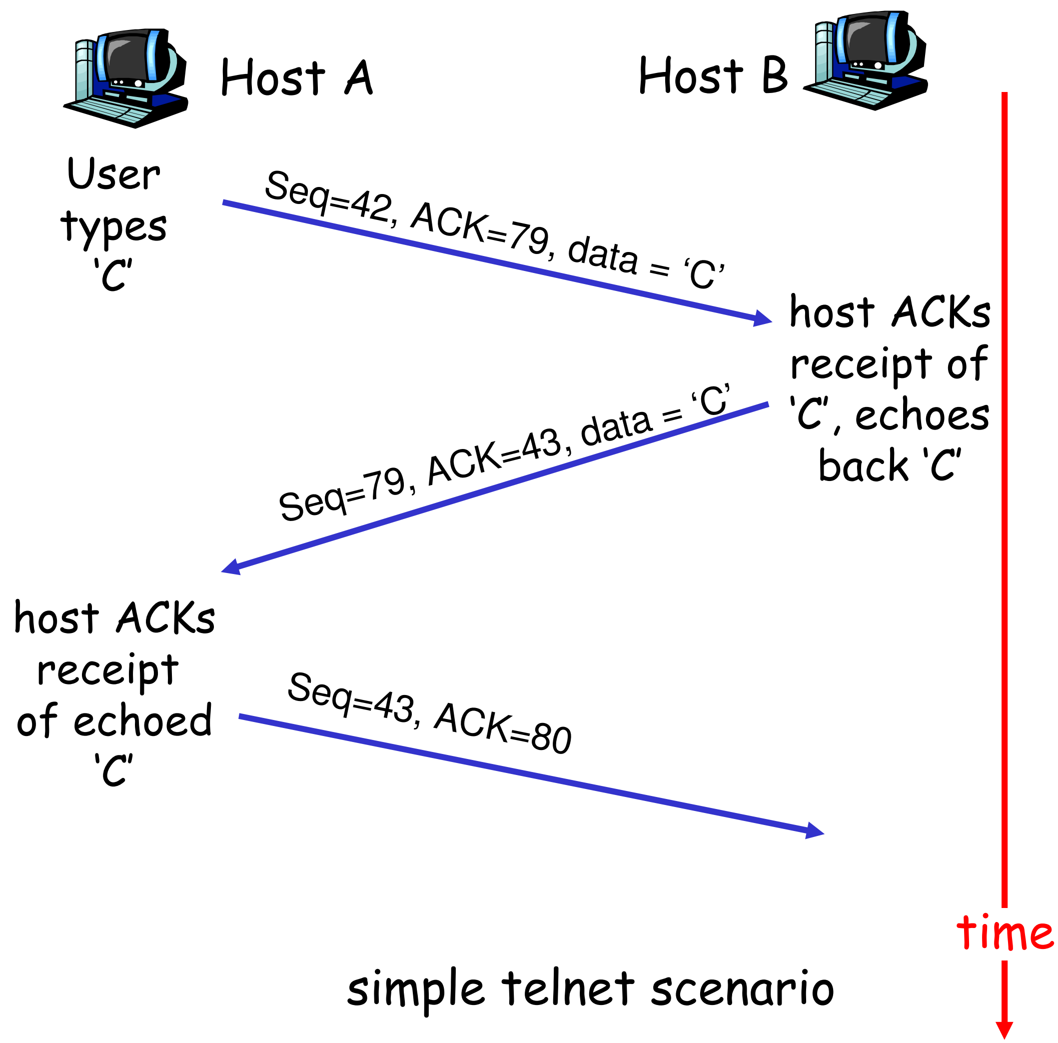
- byte stream "number" of first byte in segment's data

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

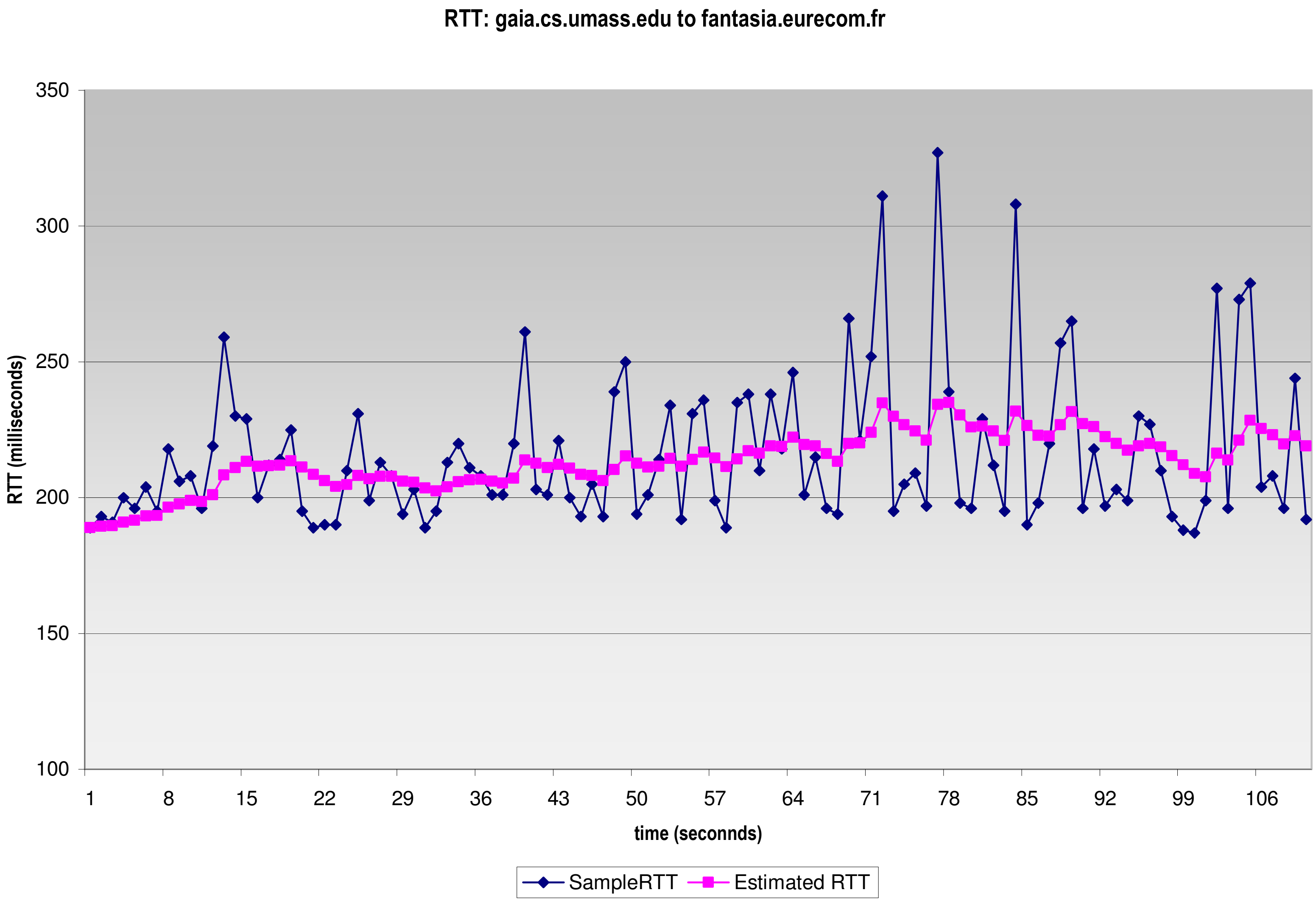
  Why??
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

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: α = 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
```