

Chapter 3

Transport Layer

Reti di Elaboratori

Corso di Laurea in Informatica

Università degli Studi di Roma "La Sapienza"

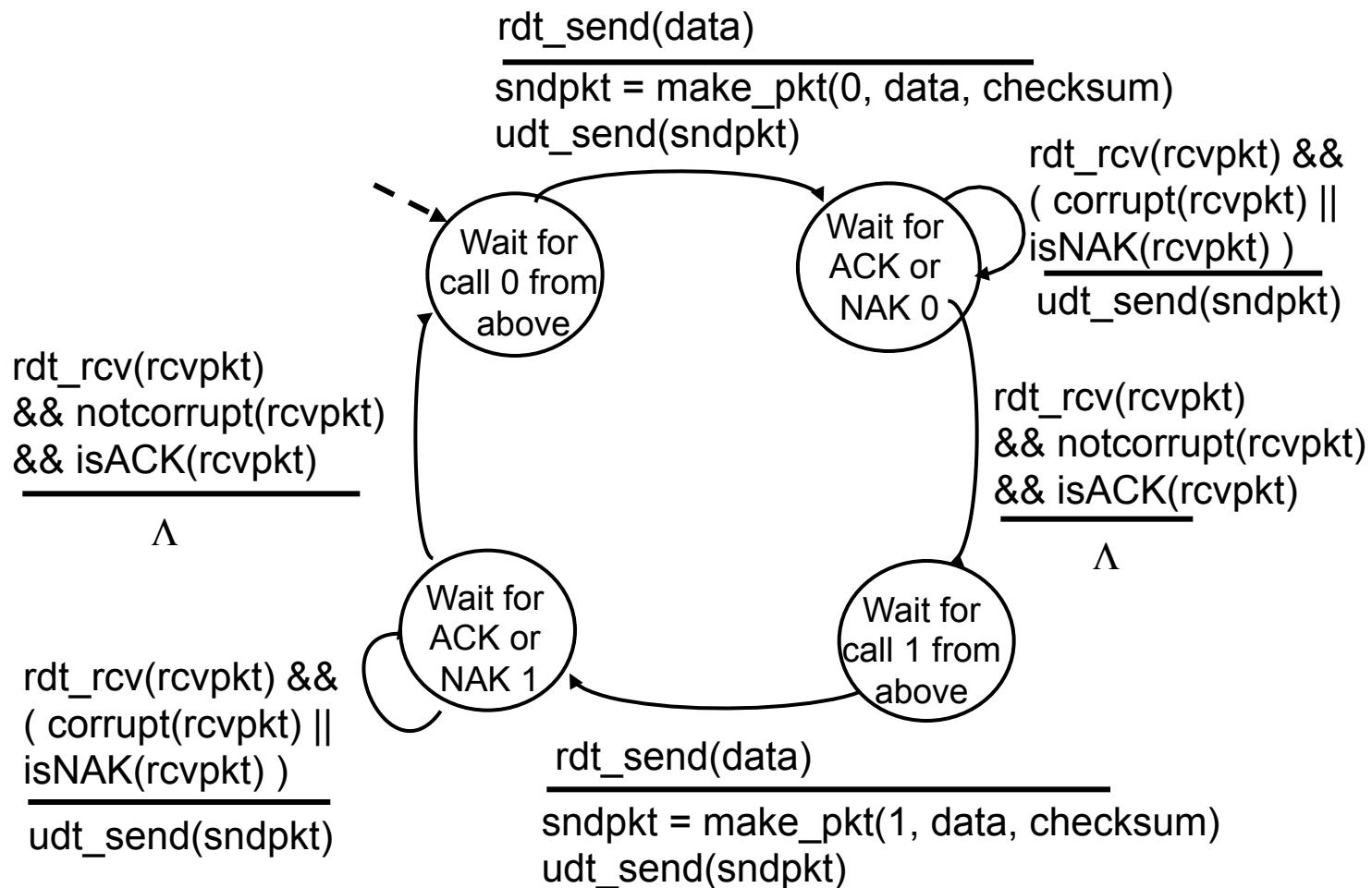
Prof.ssa Chiara Petrioli

Parte di queste slide sono state prese dal materiale associato al libro
Computer Networking: A Top Down Approach, 5th edition.
All material copyright 1996-2009

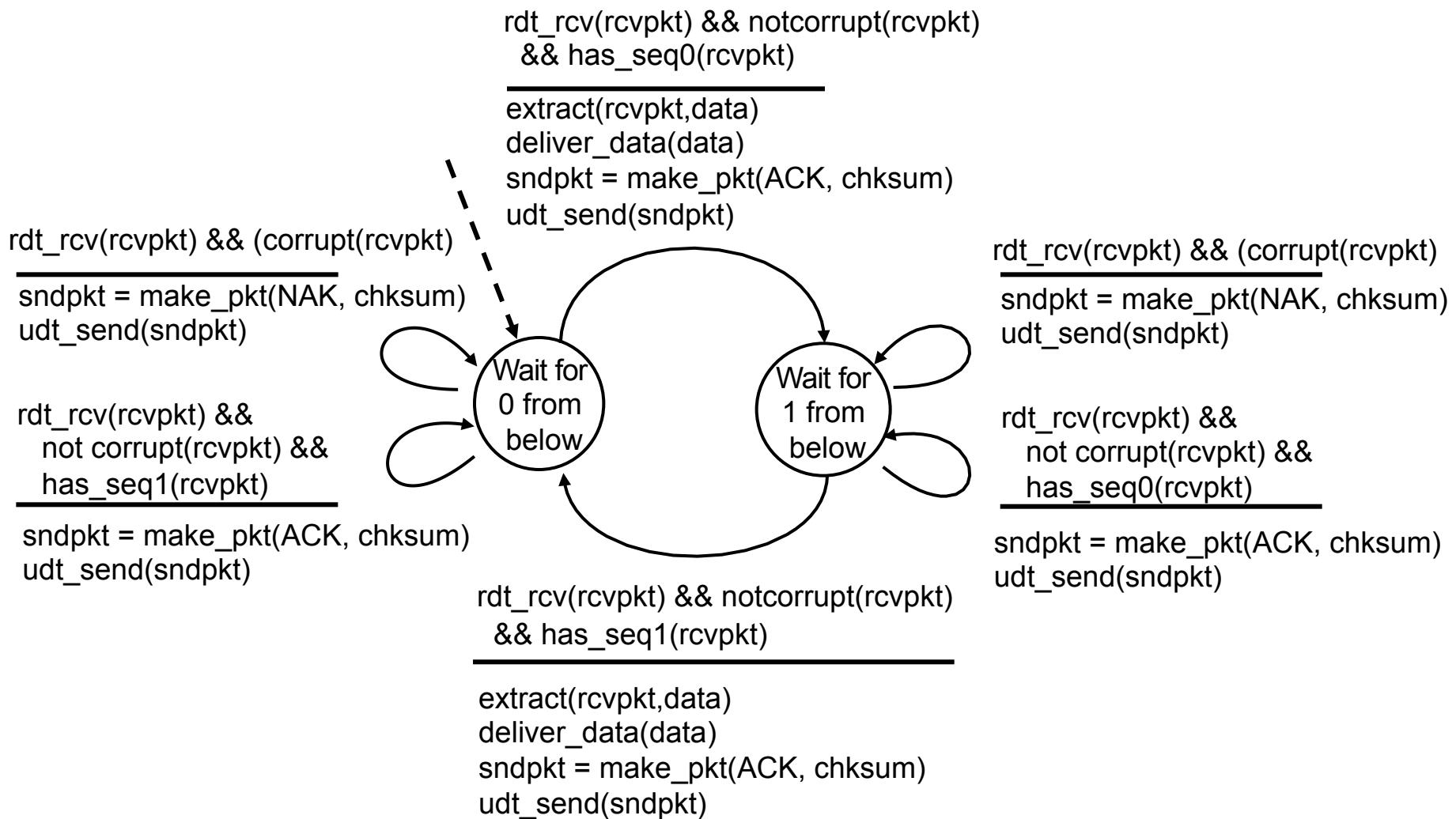
J.F Kurose and K.W. Ross, All Rights Reserved

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

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
 - state must “remember” whether “current” pkt has 0 or 1 seq. #

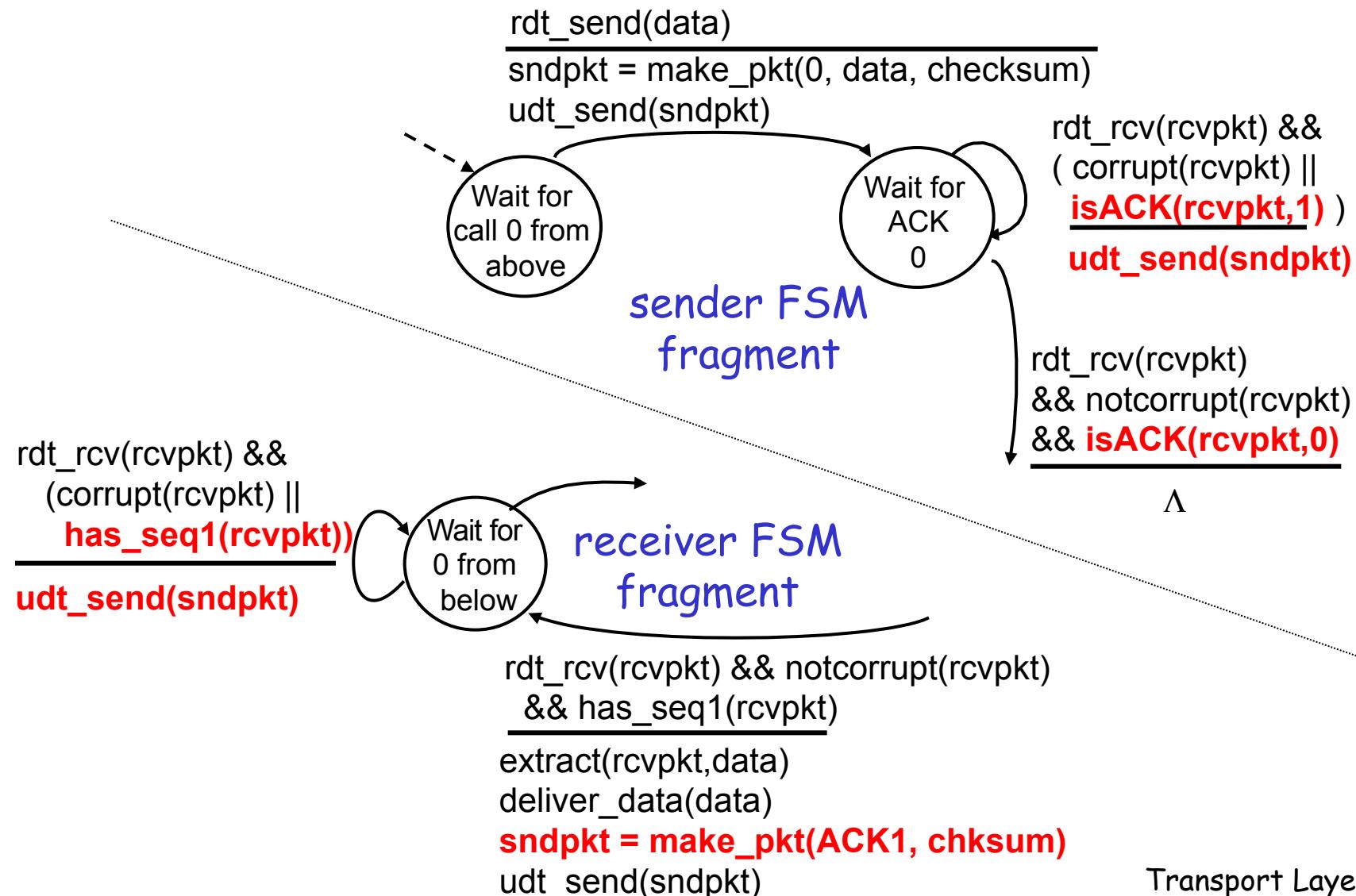
Receiver:

- must check if received packet is duplicate
 - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using NAKs only
- instead of NAK, receiver sends ACK for last pkt received OK
 - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors and loss

New assumption:

underlying channel can also loose packets (data or ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: how to deal with loss?

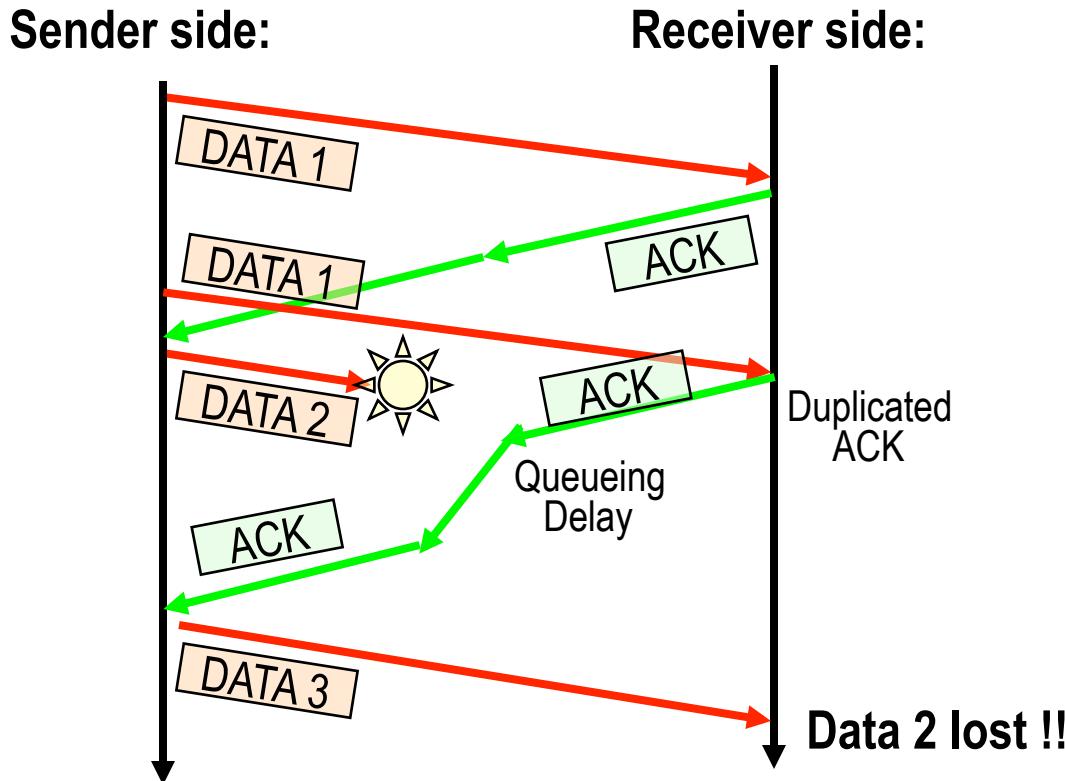
- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

Approach: sender waits

“reasonable” amount of time for ACK

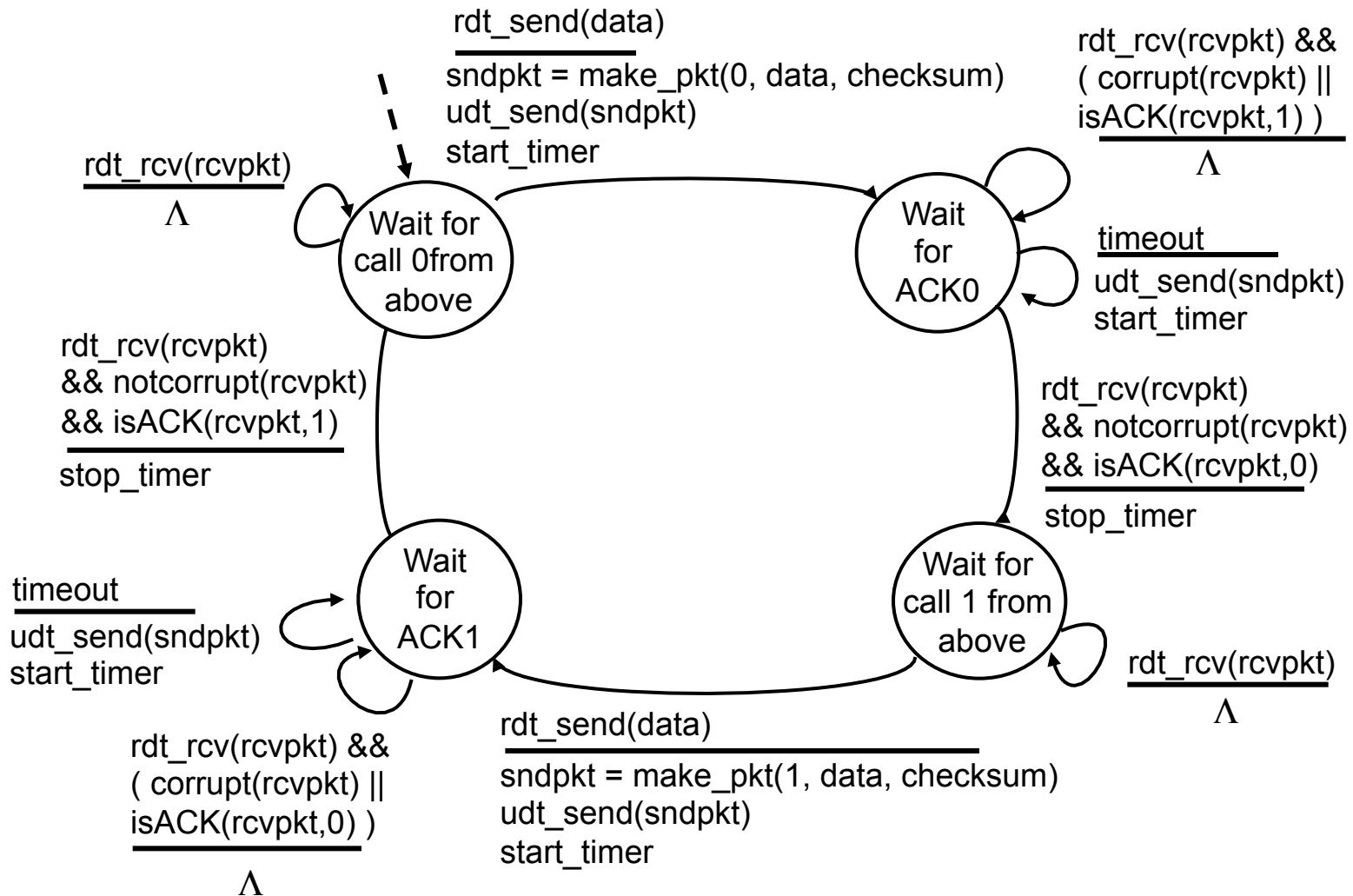
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but use of seq. #'s already handles this
 - receiver must specify seq # of pkt being ACKed
- requires countdown timer

Why sequence numbers? (on ack)

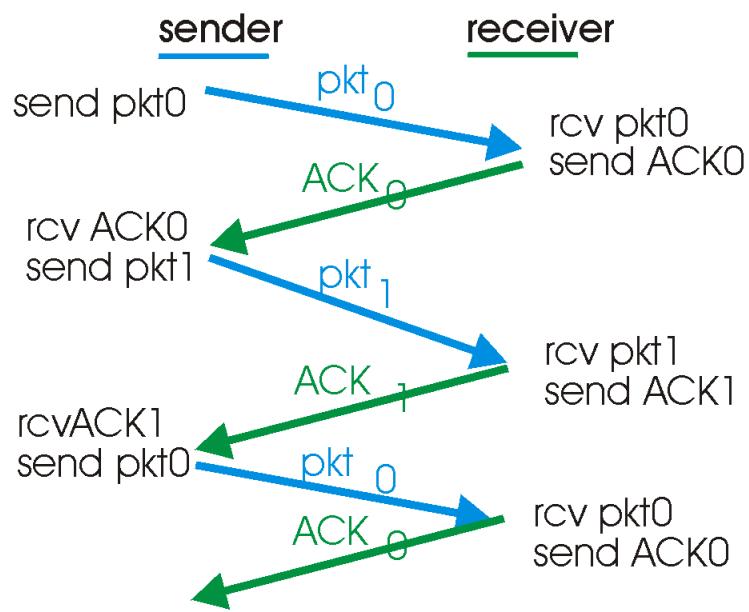


*With pathologically critical network (as the Internet!)
also need to univocally “label” all acks circulating
in the network between two end points.
1 bit (0-1) enough for Stop-and-wait ?*

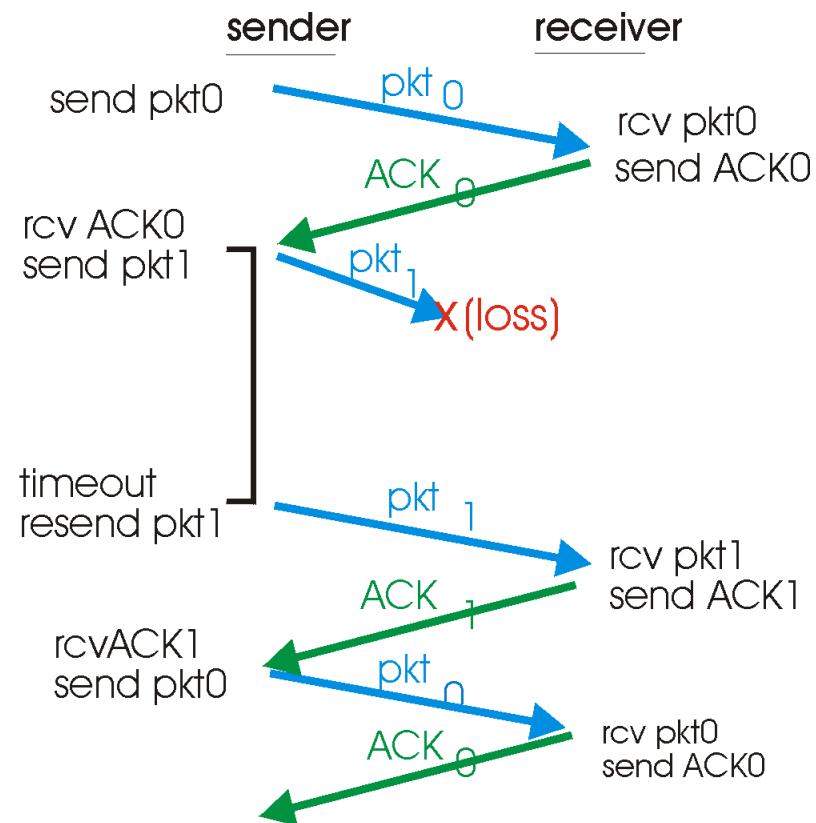
rdt3.0 sender



rdt3.0 in action

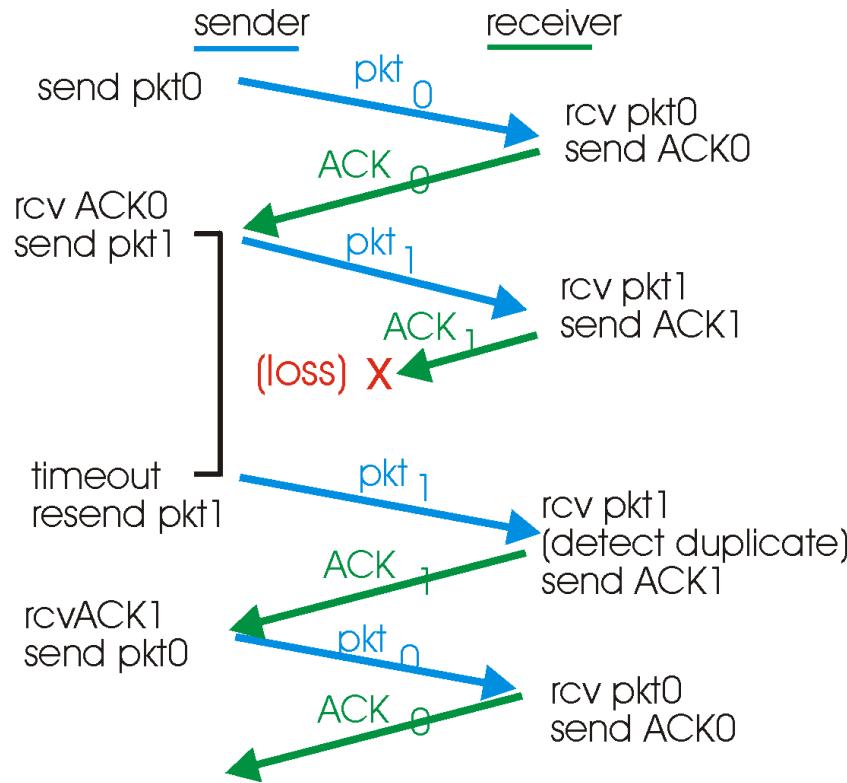


(a) operation with no loss

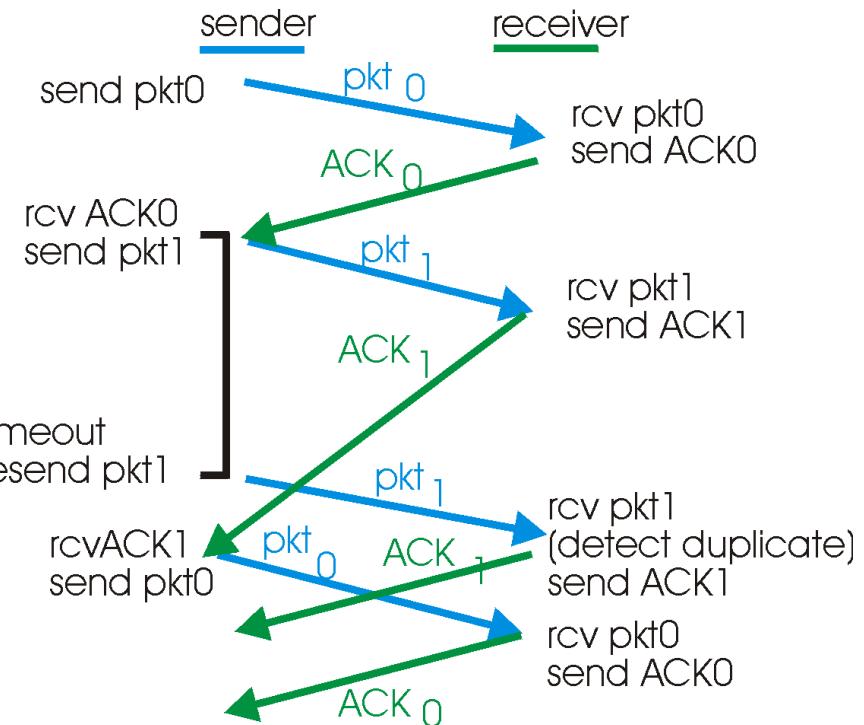


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

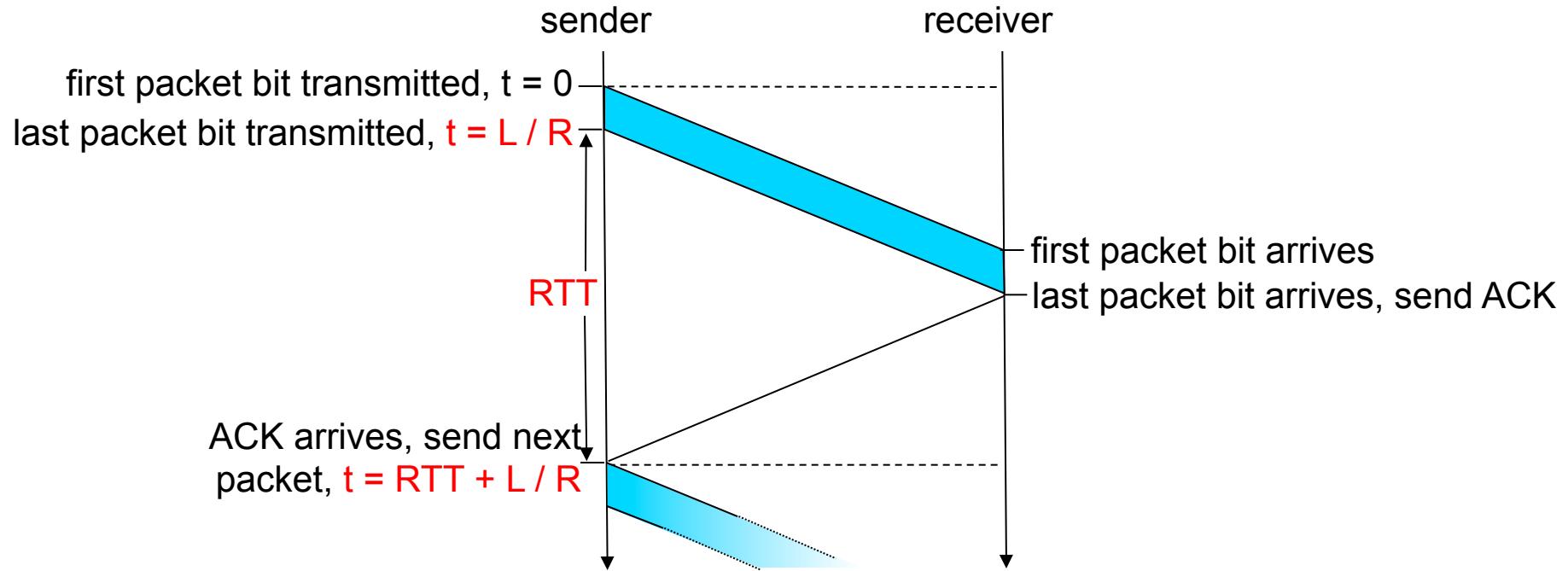
- ❑ rdt3.0 works, but performance stinks
- ❑ example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{10} \text{ b/sec}} = 8 \text{ microsec}$$

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

- U_{sender} : **utilization** - fraction of time sender busy sending
- 1KB pkt every 30 msec \rightarrow 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

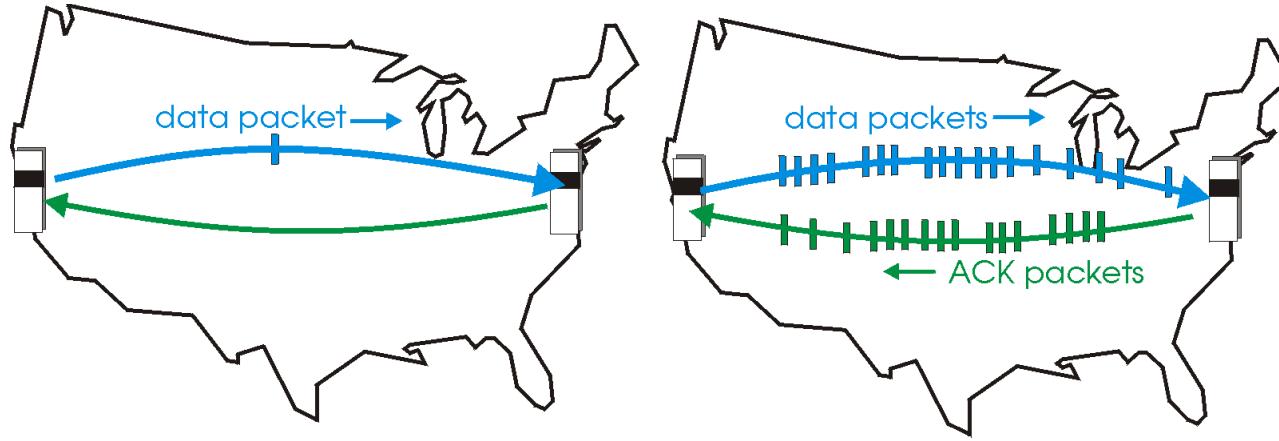


$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

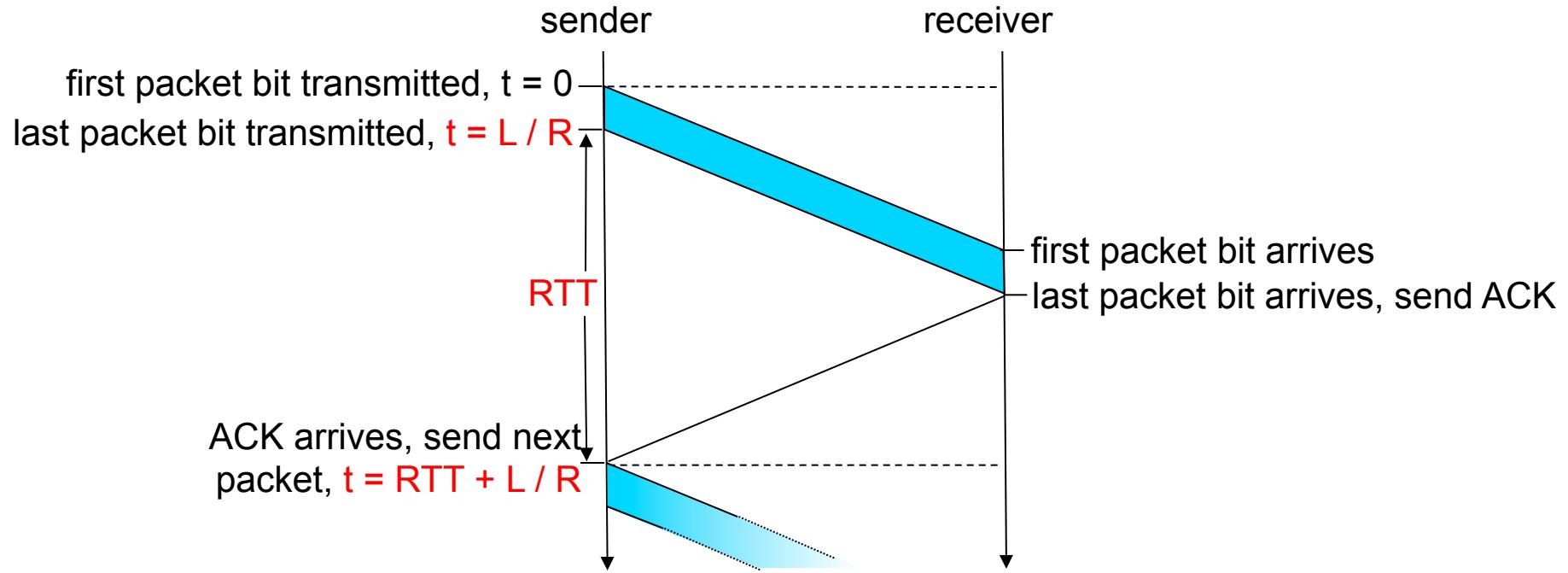
Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

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



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

rdt3.0: stop-and-wait operation

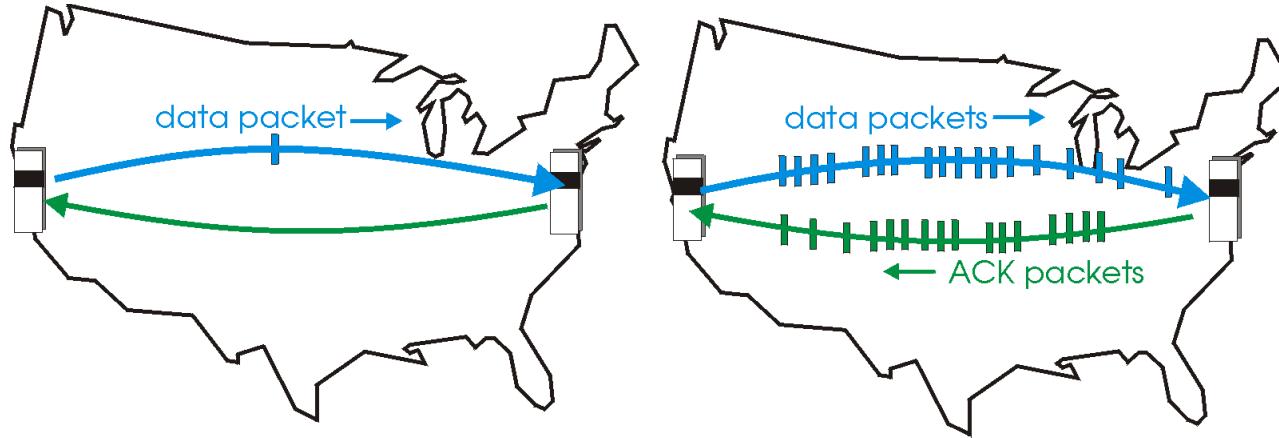


$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

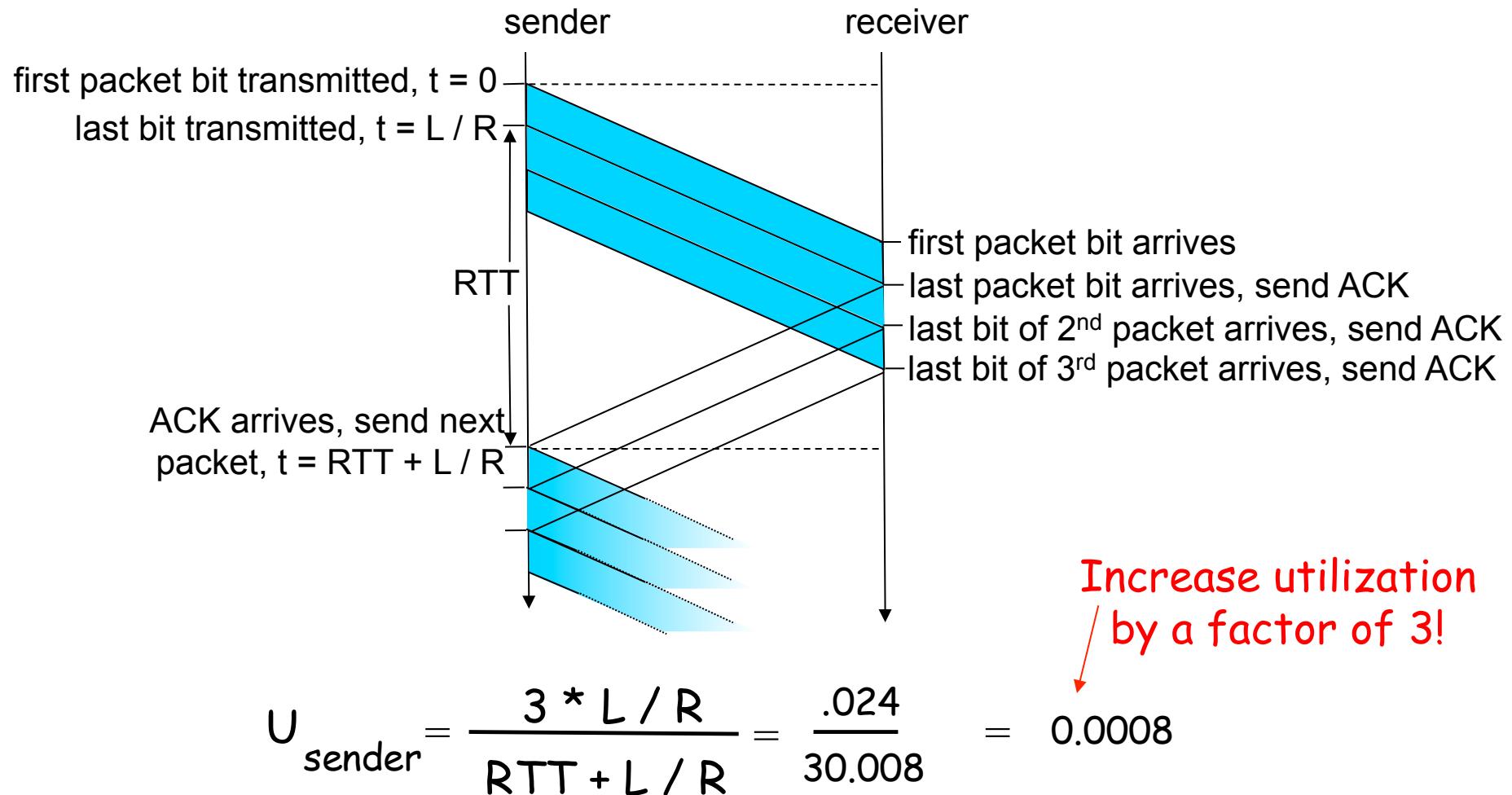
Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

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



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

Pipelining: increased utilization

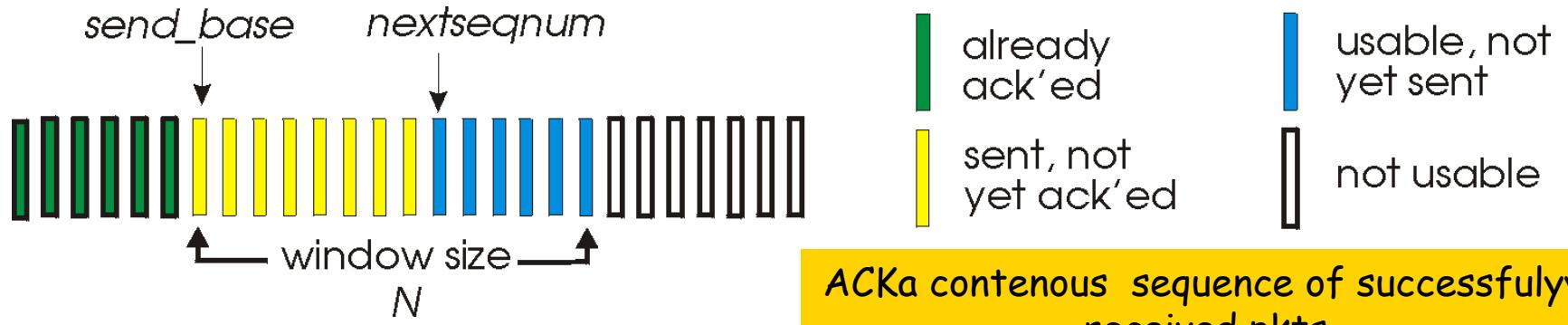


Go-Back-N

Diverso rispetto a Stop and Wait
Q: Perché?

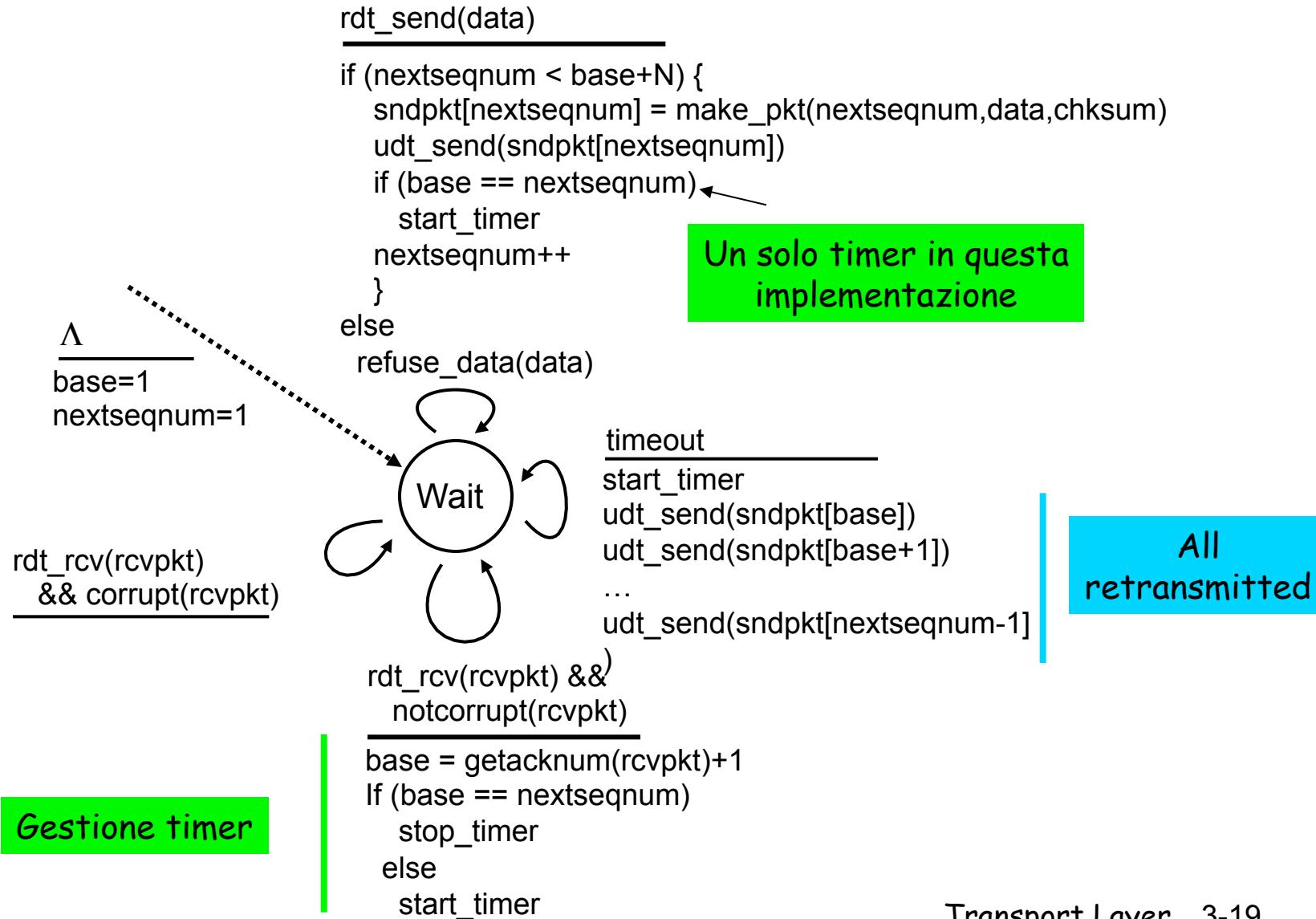
Sender:

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ ed pkts allowed

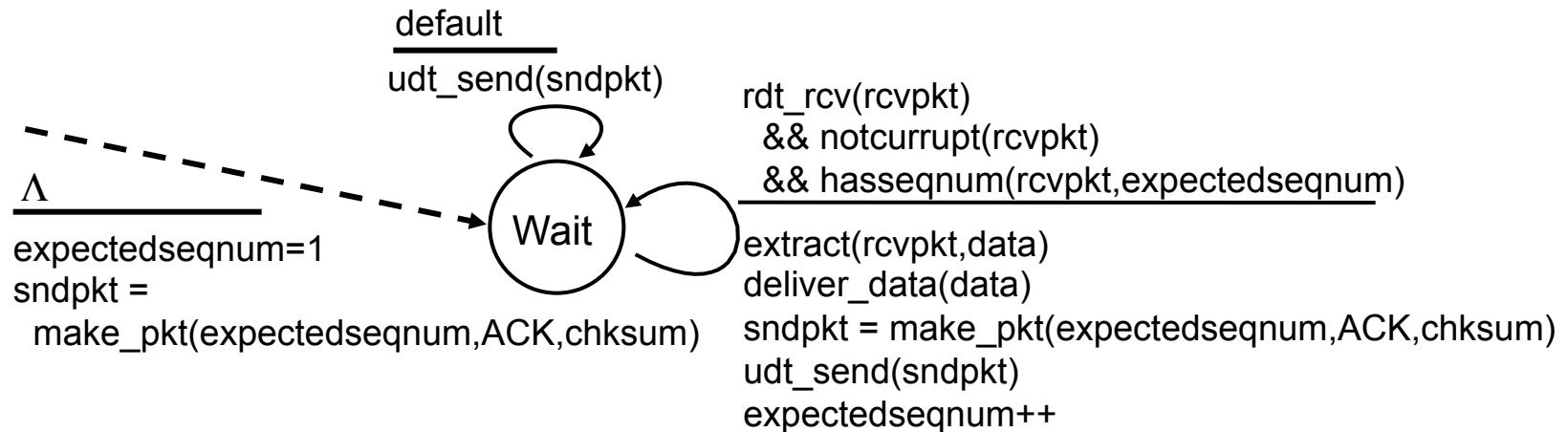


- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
 - may receive duplicate ACKs (see receiver)
- timer for in-flight pkt
- $\text{timeout}(n)$: retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



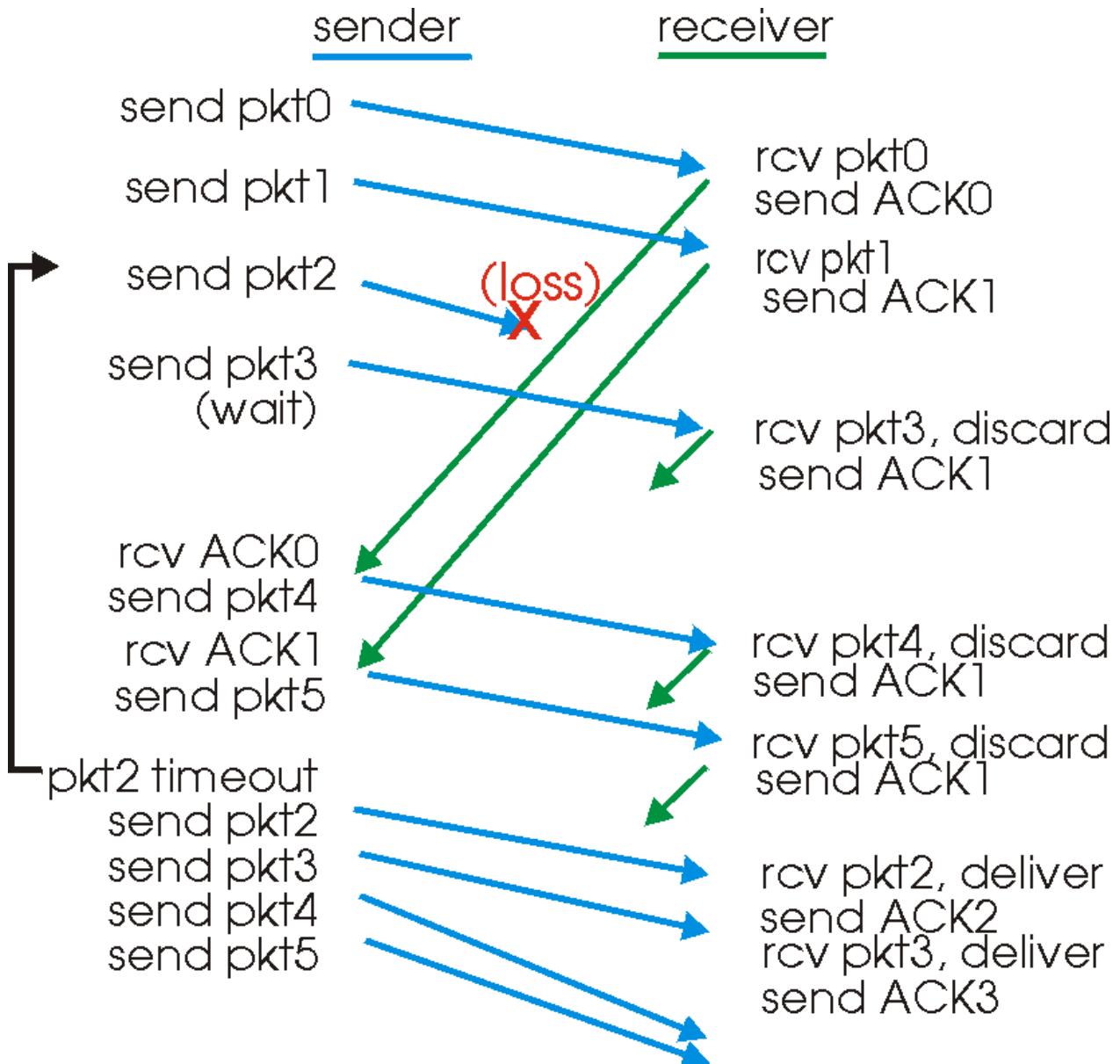
GBN: receiver extended FSM



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, congestion control)
 - related to window size field length

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]

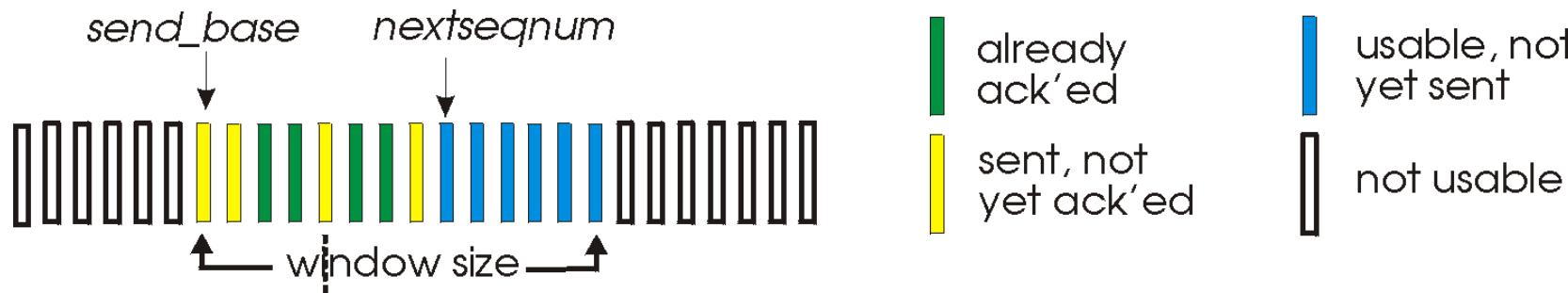
- ACK(n)

otherwise:

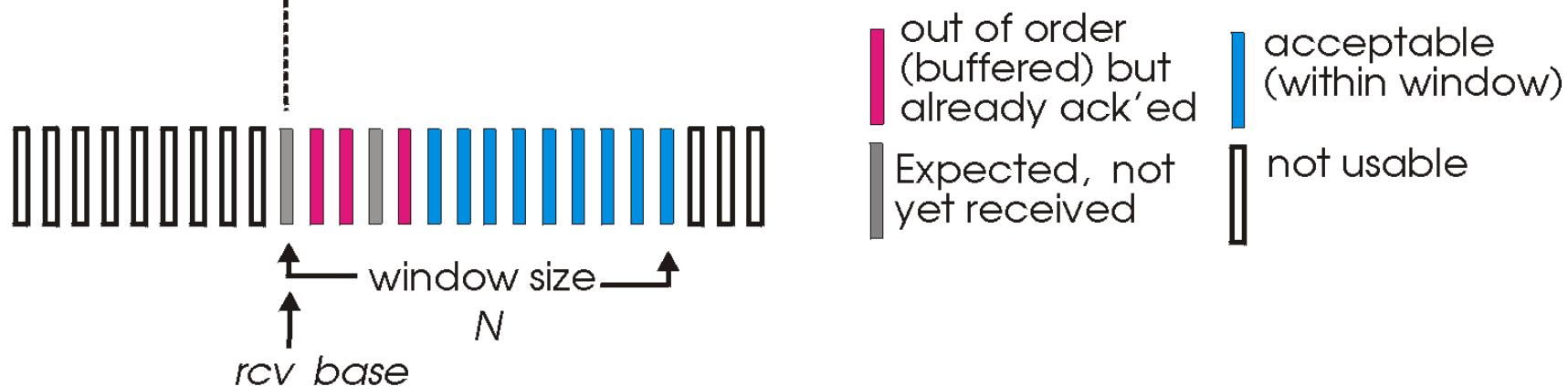
- ignore

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

Selective repeat: sender, receiver windows



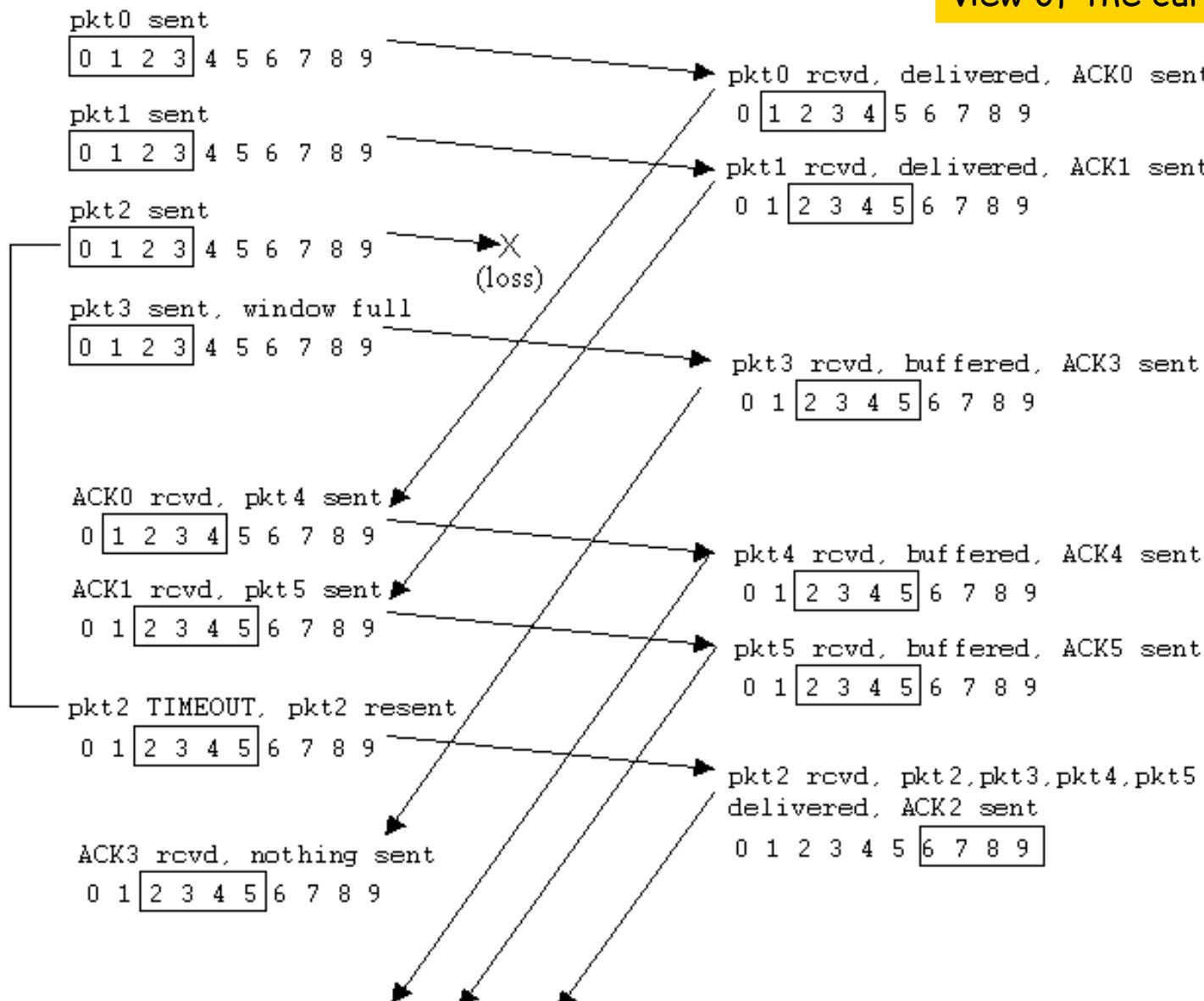
(a) sender view of sequence numbers



(b) receiver view of sequence numbers

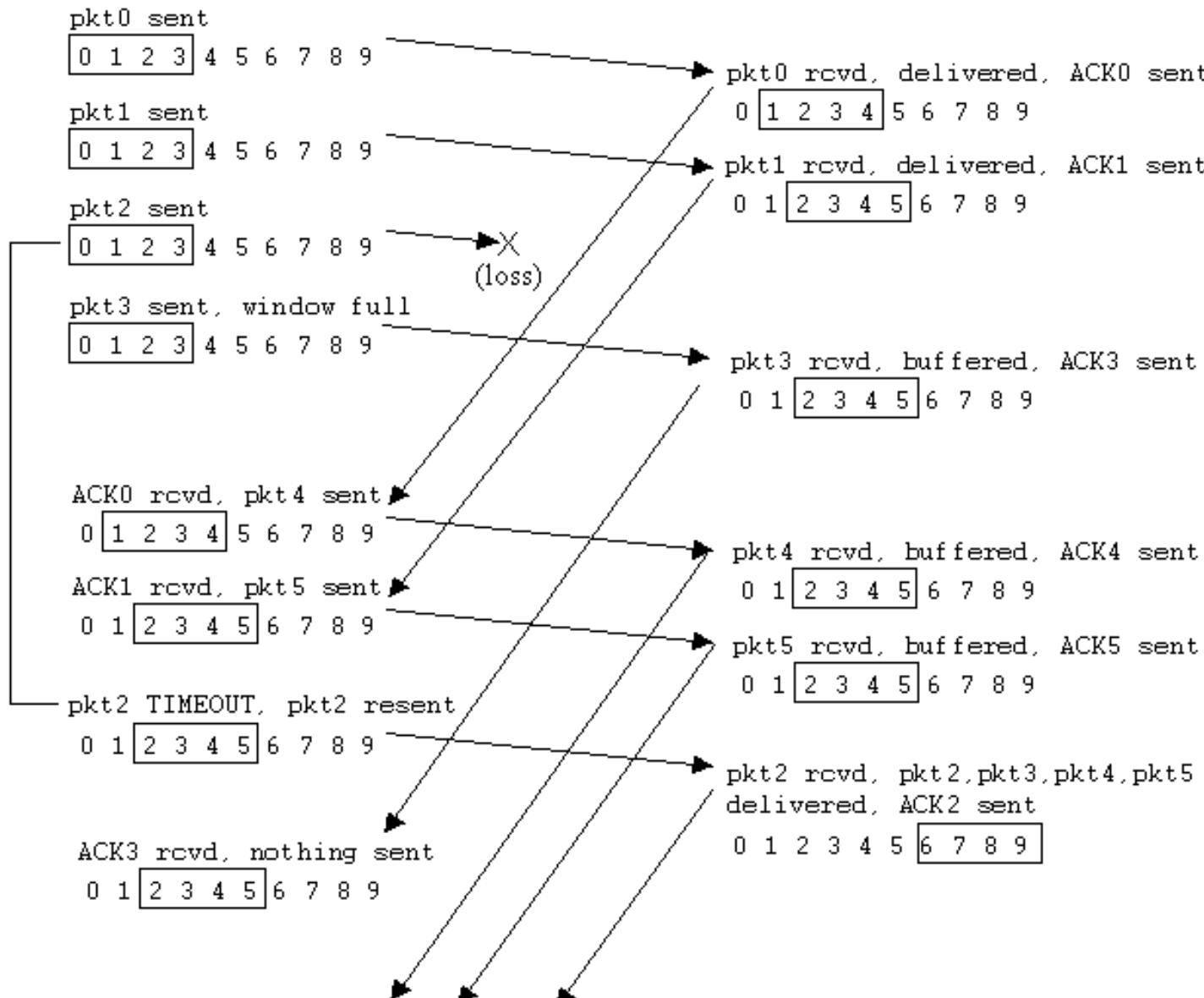
Selective repeat in action

Transmitter and receiver
can have different
view of the current window



Selective repeat in action

- 1) Delays in receiving ACKs
- 2) lost ACKs

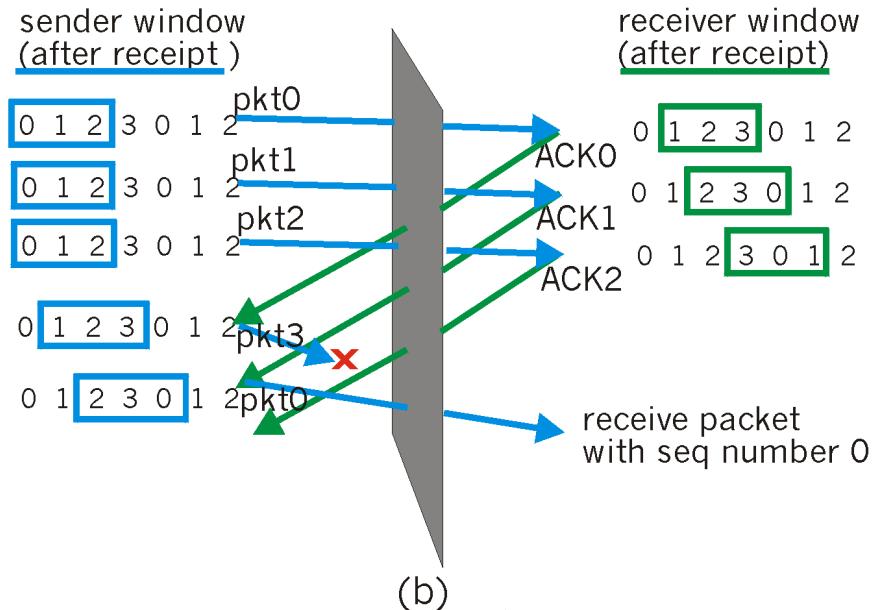
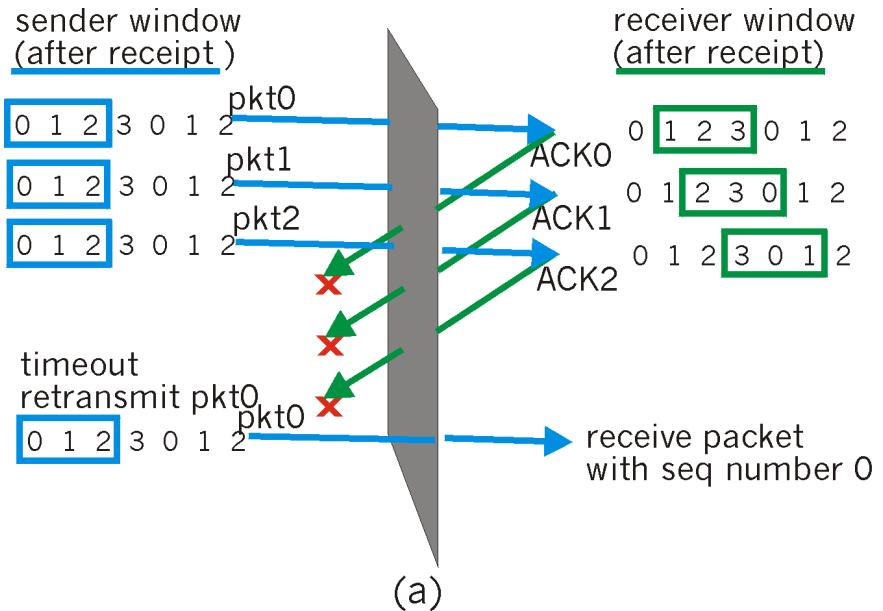


Selective repeat: dilemma

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

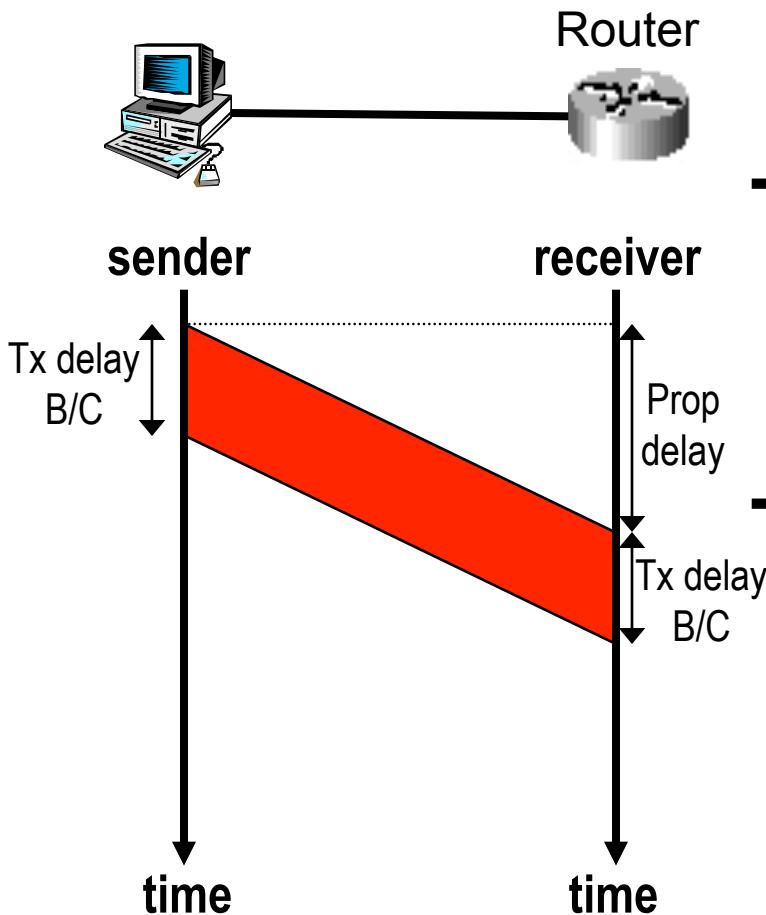
- The window size must be less than or equal to half the size of the sequence number space for SR protocols

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 = $512*8/64000 = 64\text{ms}$

→ Propagation delay - constant depending on

→ Link length

→ Electromagnetic waves propagation speed in considered media

→ 200 km/s for copper links

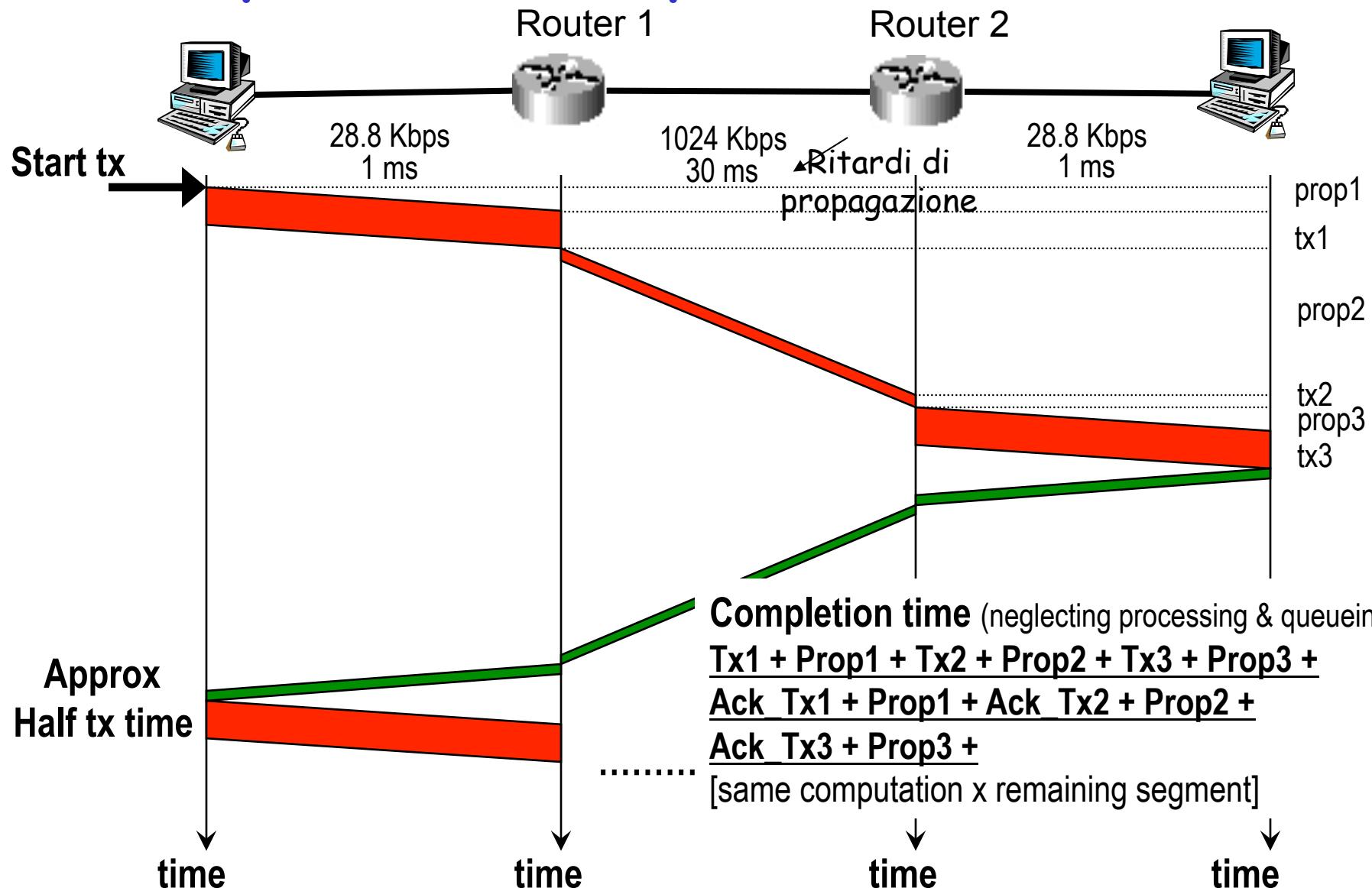
→ 300 km/s in air

→ other delays neglected

→ Queueing

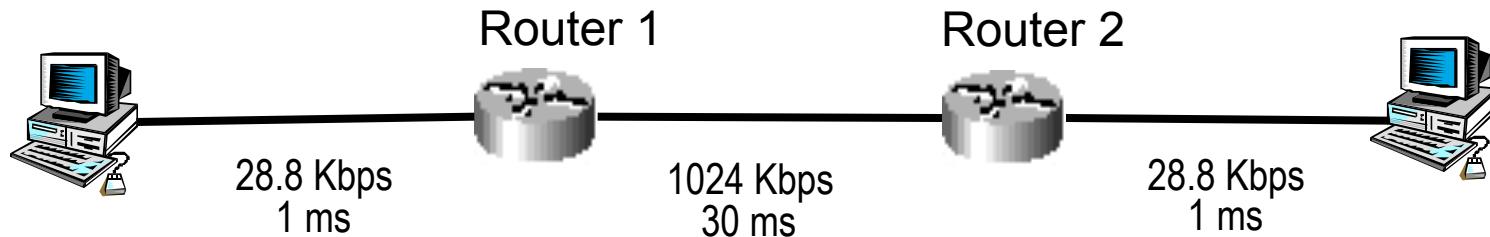
→ processing

Stop-and-wait performance



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:

$$\Rightarrow Tx1 = 576 \cdot 8 / 28,8 = 160\text{ms}$$
$$\Rightarrow Tx3 = Tx1$$
$$\Rightarrow Tx2 = 576 \cdot 8 / 1024 = 4,5\text{ ms}$$

→ Segment 2:

$$\Rightarrow Tx1 = 528 \cdot 8 / 28,8 = 146,7\text{ms}$$
$$\Rightarrow Tx3 = Tx1$$
$$\Rightarrow Tx2 = 528 \cdot 8 / 1024 = 4,1\text{ ms}$$

→ Acks:

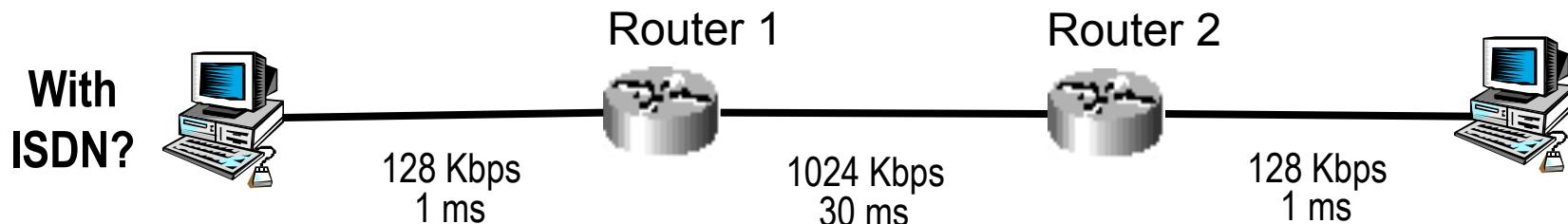
$$\Rightarrow Tx1 = Tx3 = 40 \cdot 8 / 28,8 = 11,1\text{ms}$$
$$\Rightarrow Tx2 = 40 \cdot 8 / 1024 = 0,3\text{ ms}$$

RESULT:

$$D = 667 (\text{tx total}) + 2 * \text{RTT} = 795\text{ ms}$$
$$\text{THR} = 1024 \cdot 8 / 795 = 10,3\text{ kbps}$$

Stop-and-wait performance

Numerical example



→ Segment 1:

$$\begin{aligned}\Rightarrow Tx_1 &= Tx_3 = \\ 576 \cdot 8 / 128 &= 36 \text{ ms} \\ \Rightarrow Tx_2 &= 576 \cdot 8 / 1024 = \\ &4,5 \text{ ms}\end{aligned}$$

→ Acks:

$$\begin{aligned}\Rightarrow Tx_1 &= Tx_3 = \\ 40 \cdot 8 / 128 &= 2,5 \text{ ms} \\ \Rightarrow Tx_2 &= 40 \cdot 8 / 1024 = \\ &0,3 \text{ ms}\end{aligned}$$

→ Segment 2:

$$\begin{aligned}\Rightarrow Tx_1 &= Tx_3 = \\ 528 \cdot 8 / 128 &= 33 \text{ ms} \\ \Rightarrow Tx_2 &= 528 \cdot 8 / 1024 = \\ &4,1 \text{ ms}\end{aligned}$$

RESULT:

$$D = 151,9 \text{ (tx total)} + 2 \cdot RTT = \\ = 279,9 \text{ ms}$$

$$THR = 1024 \cdot 8 / 279,9 = \\ = 29,3 \text{ kbps}$$

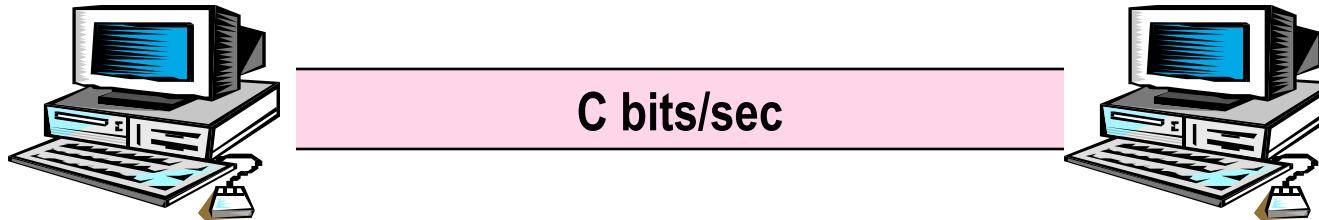
on Gbps fiber optics?

$$D = \text{negligible} + 2 \cdot RTT = \\ = 128 \text{ ms}$$

$$THR = 1024 \cdot 8 / 128 = \\ = 64 \text{ kbps}$$

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

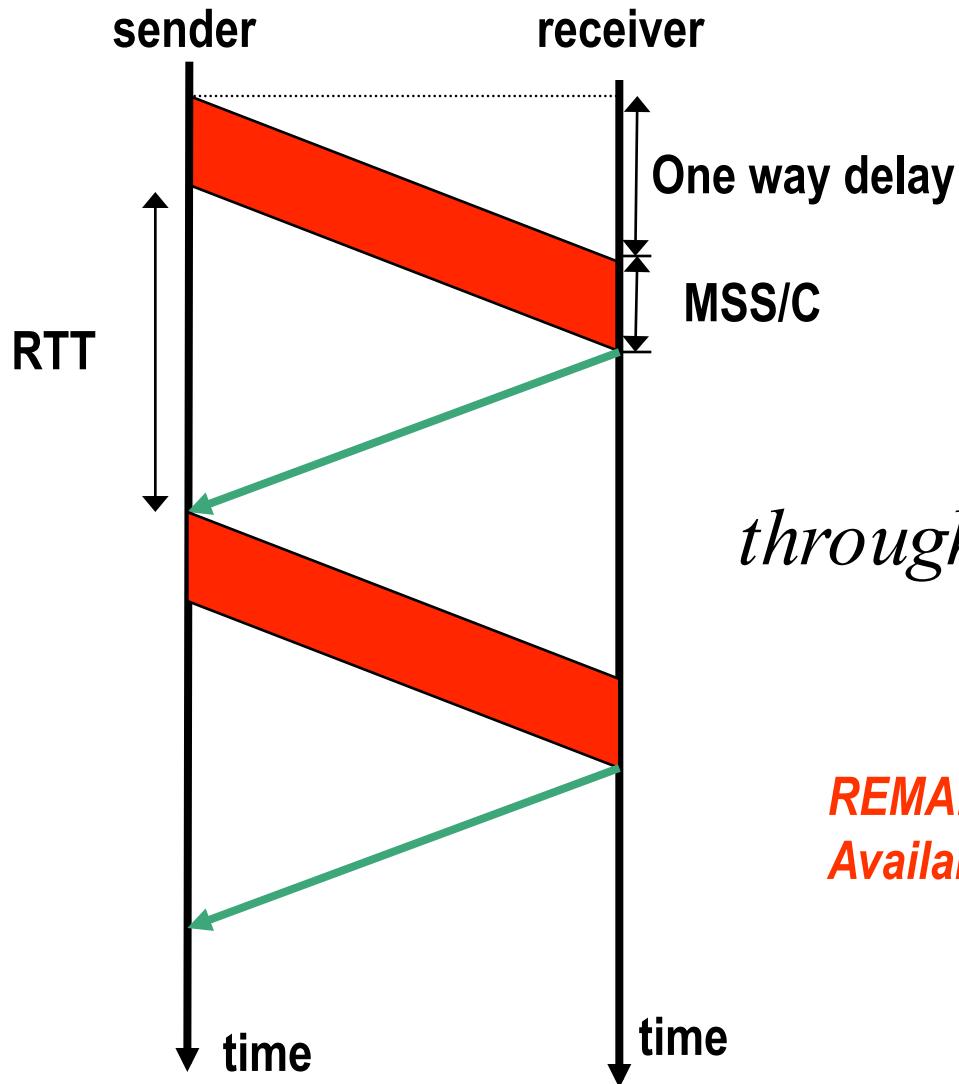
$W=1$: stop-and-wait

$W>1$: go-back-N (sliding window)

This is a highly dynamic parameter in TCP!!

For now, consider W fixed

W=1 case (stop-and-wait)



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

REMARK: throughput always lower than Available link rate!

Latency in TCP

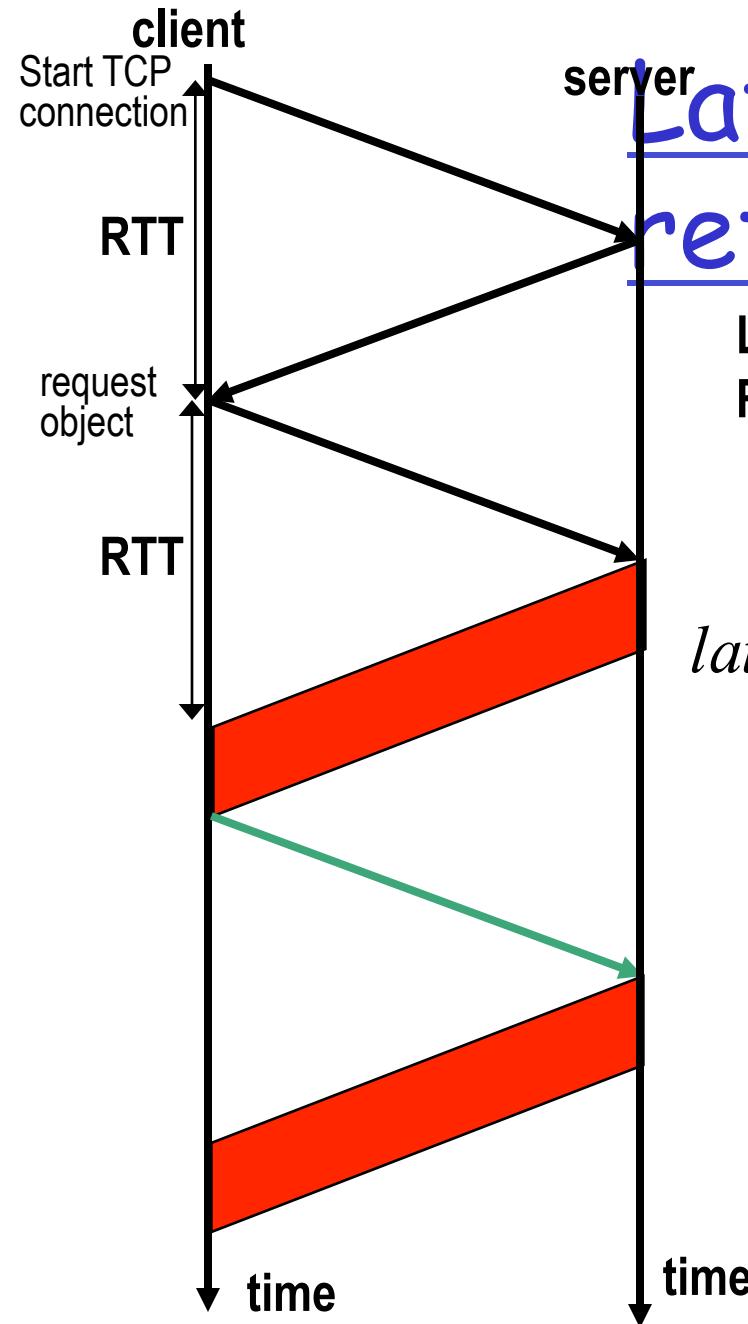
retrieval model

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

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

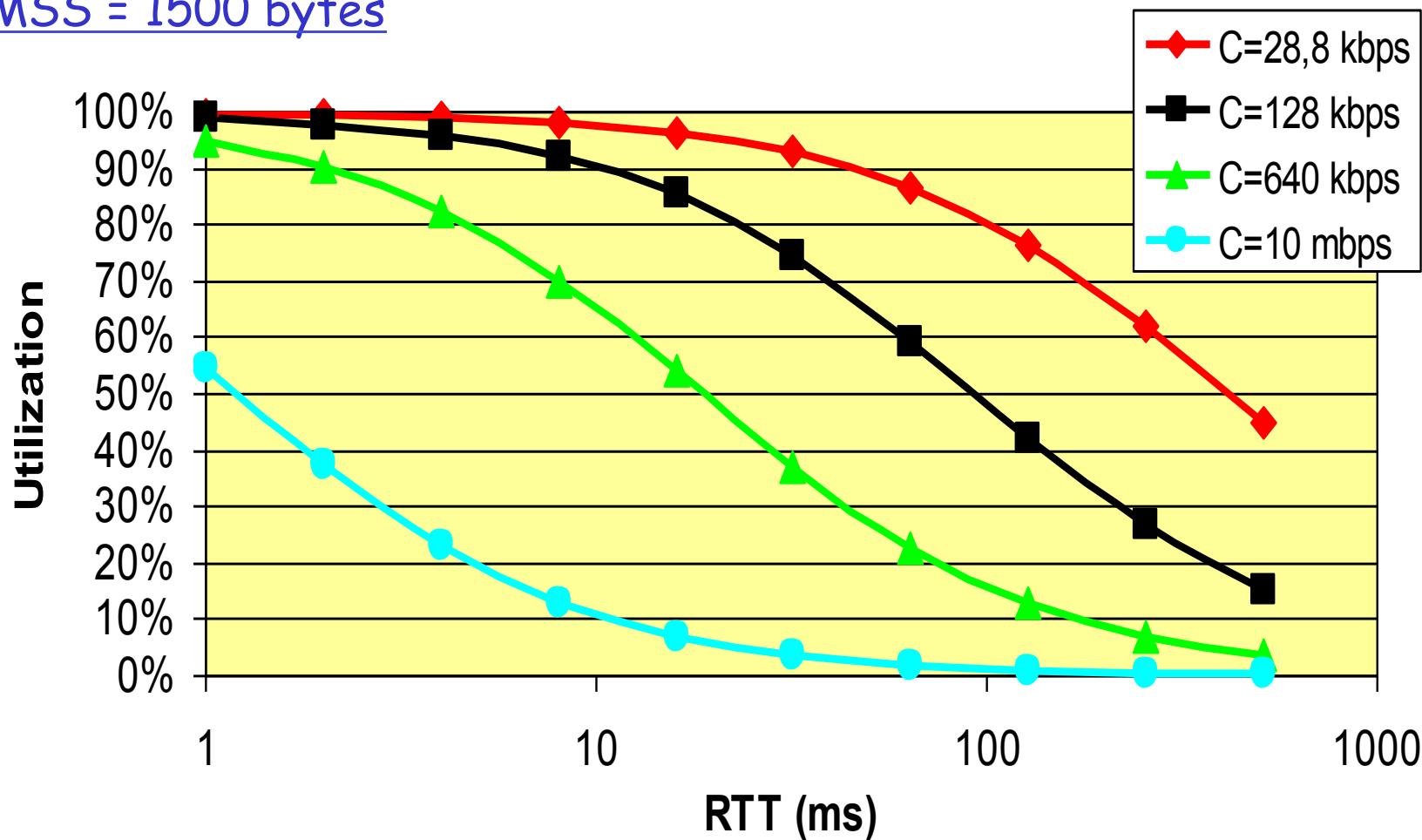


Number of segments
In which message
is split



$W=1$ case (stop-and-wait)

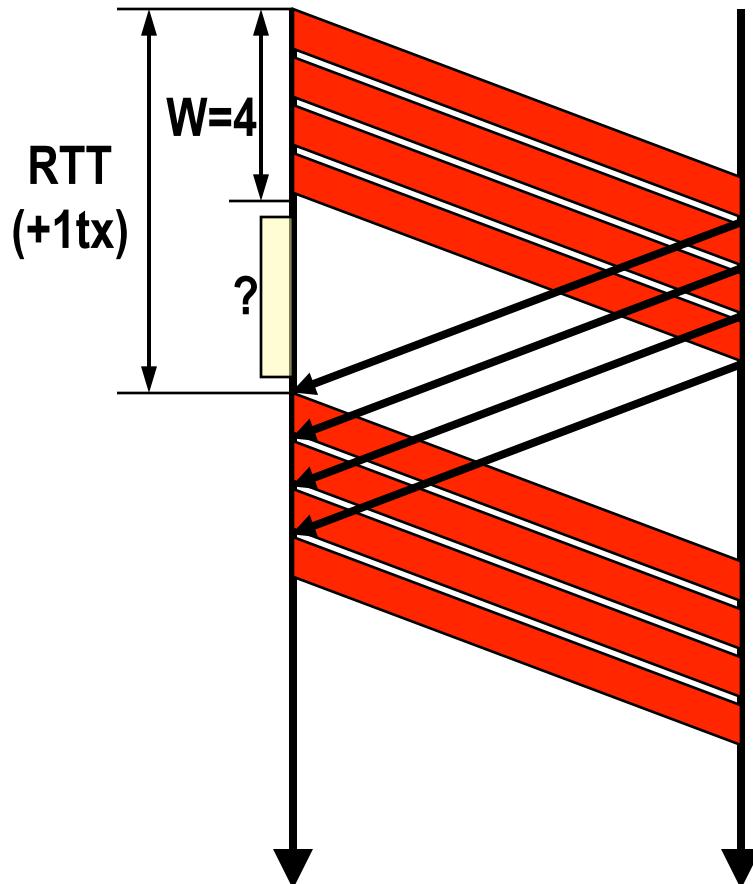
MSS = 1500 bytes



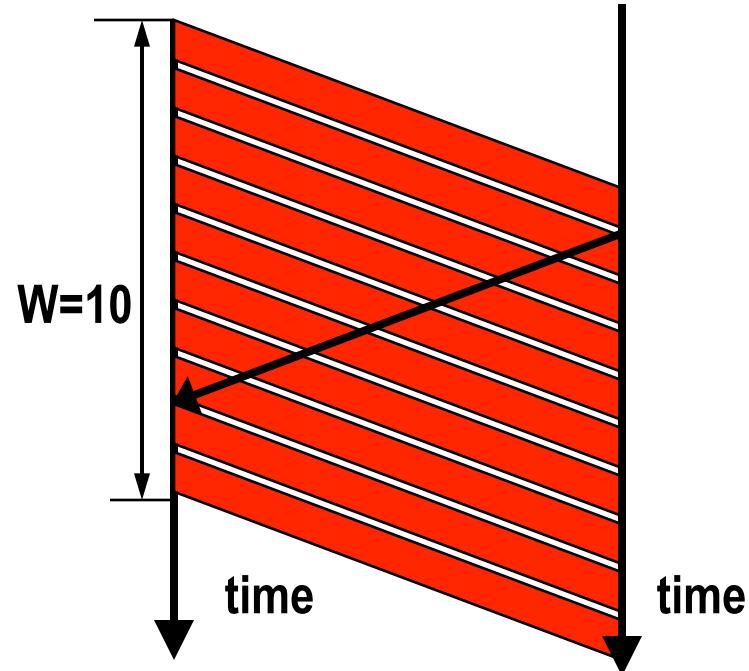
Under-utilization with: 1) high capacity links, 2) large RTT links

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

$$W \cdot \frac{MSS}{C} > RTT + \frac{MSS}{C}$$

Time to transmit
W segments Time to receive
Ack of first segment

We may elaborate:

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

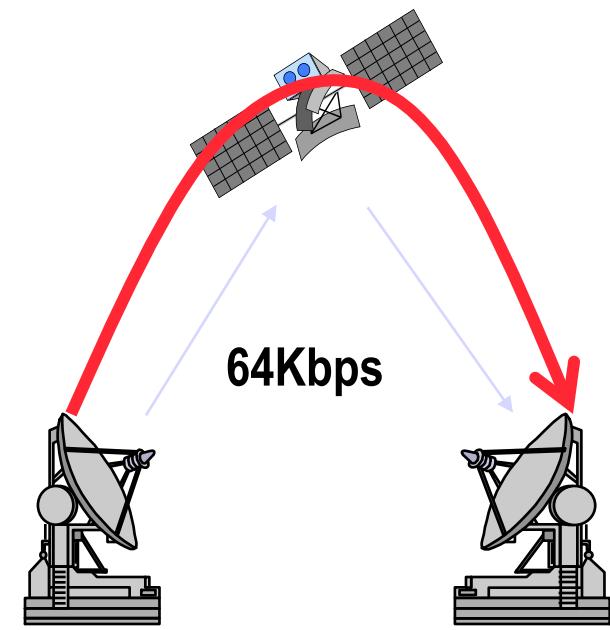
This means that full link utilization is possible when window size (in bits) is Greater than the bandwidth (C bit/s) delay (RTT s) product!

Bandwidth-delay product



→ Network: like a pipe
→ C [bit/s] $\times D$ [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



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

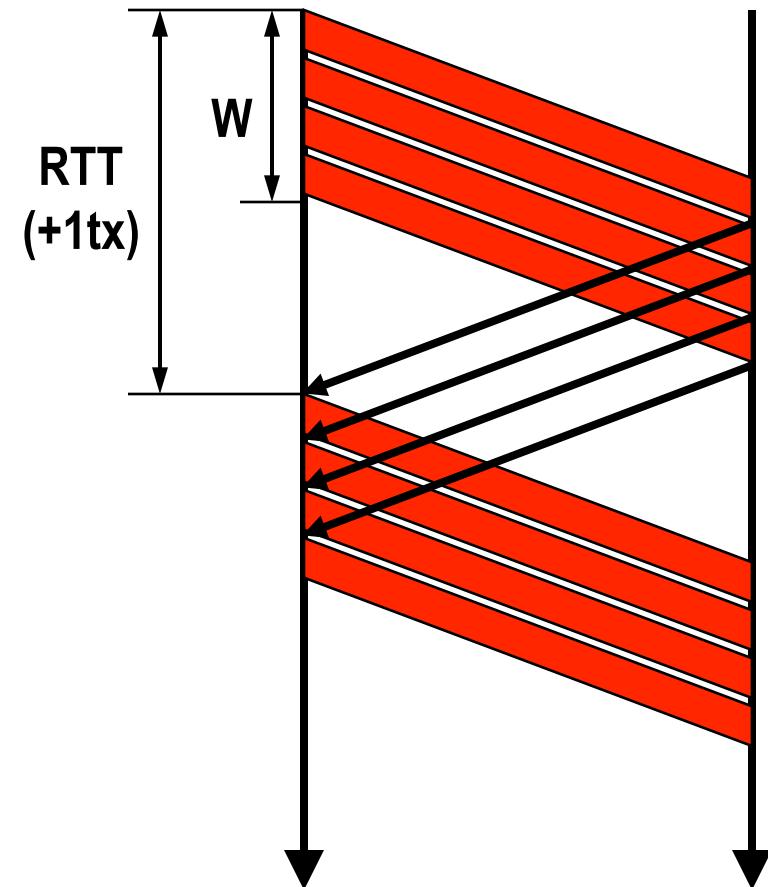
Long Fat Networks

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!

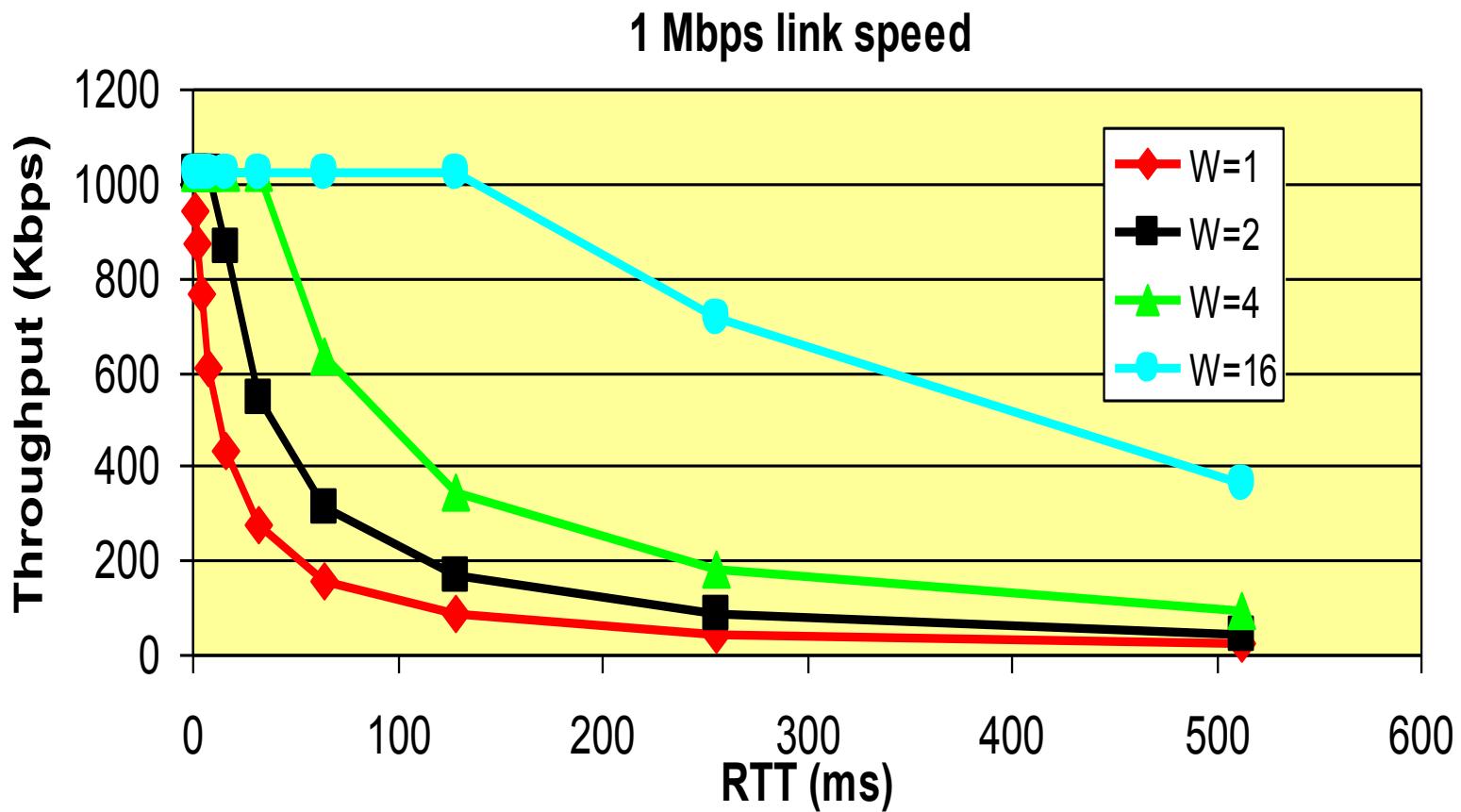
Pipelining ($W > 1$) analysis



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

Throughput for pipelining

MSS = 1500 bytes



Chapter 3 outline

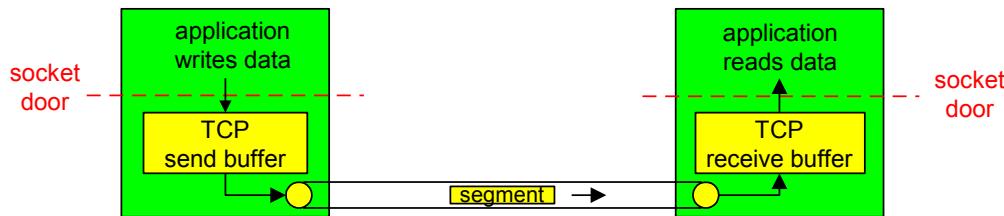
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion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

TCP: Overview

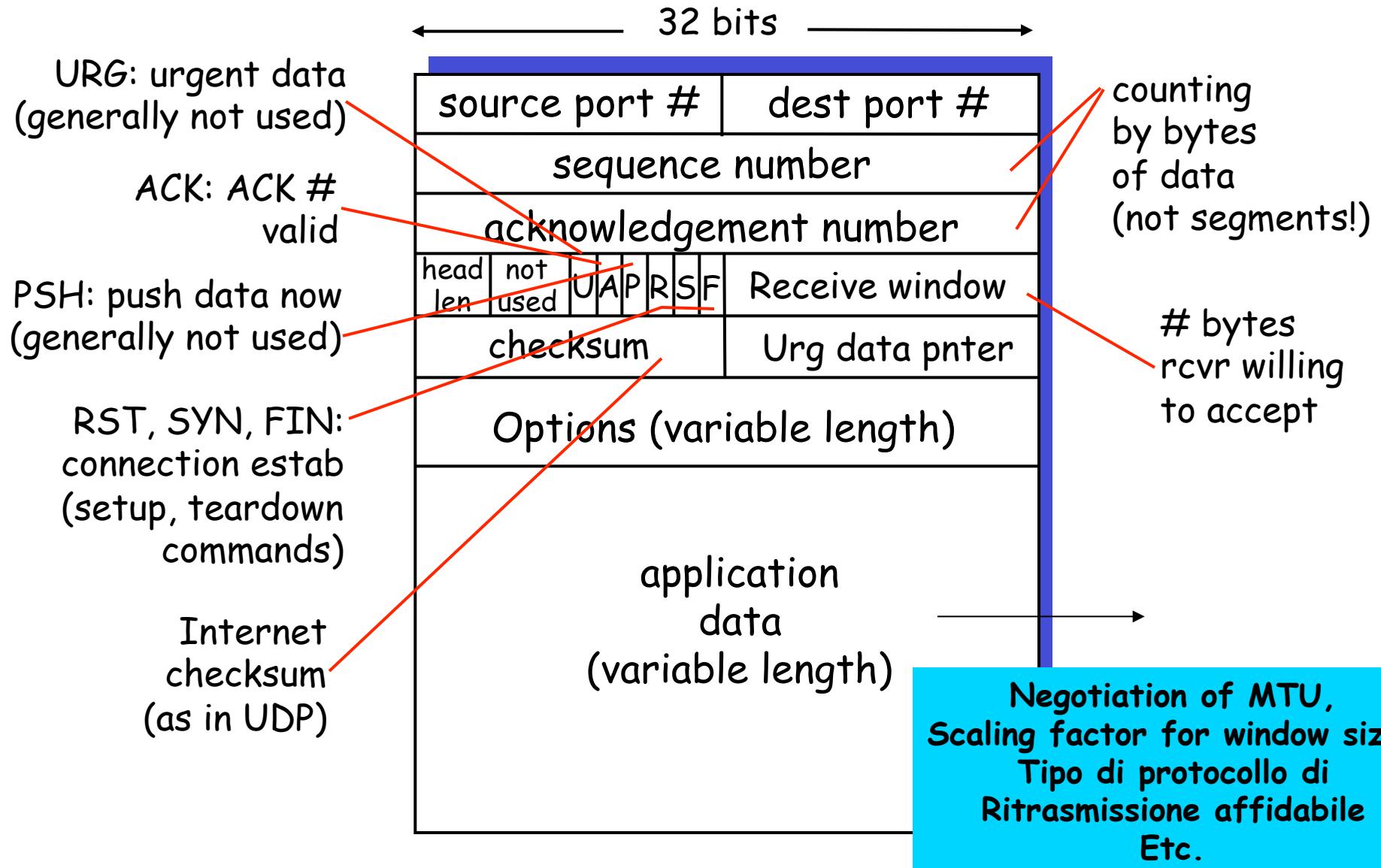
RFCs: 793, 1122, 1323, 2018, 2581

- 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!!

Source port	Destination port																					
32 bit Sequence number																						
32 bit acknowledgement number																						
Header length	6 bit Reserved																					
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>U</td><td>A</td><td>P</td><td>R</td><td>S</td><td>Y</td><td>F</td></tr> <tr> <td>R</td><td>C</td><td>S</td><td>S</td><td>Y</td><td>I</td><td>I</td></tr> <tr> <td>G</td><td>K</td><td>H</td><td>T</td><td>N</td><td>N</td><td>N</td></tr> </table>	U	A	P	R	S	Y	F	R	C	S	S	Y	I	I	G	K	H	T	N	N	N
U	A	P	R	S	Y	F																
R	C	S	S	Y	I	I																
G	K	H	T	N	N	N																
checksum																						
Window size																						
Urgent pointer																						

- 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 all bytes up to ack# - 1 (cumulative)
- When seq/ack reach $2^{32}-1$, next wrap back to 0