Overview of last class

rdt2.2: a NAK-free protocol

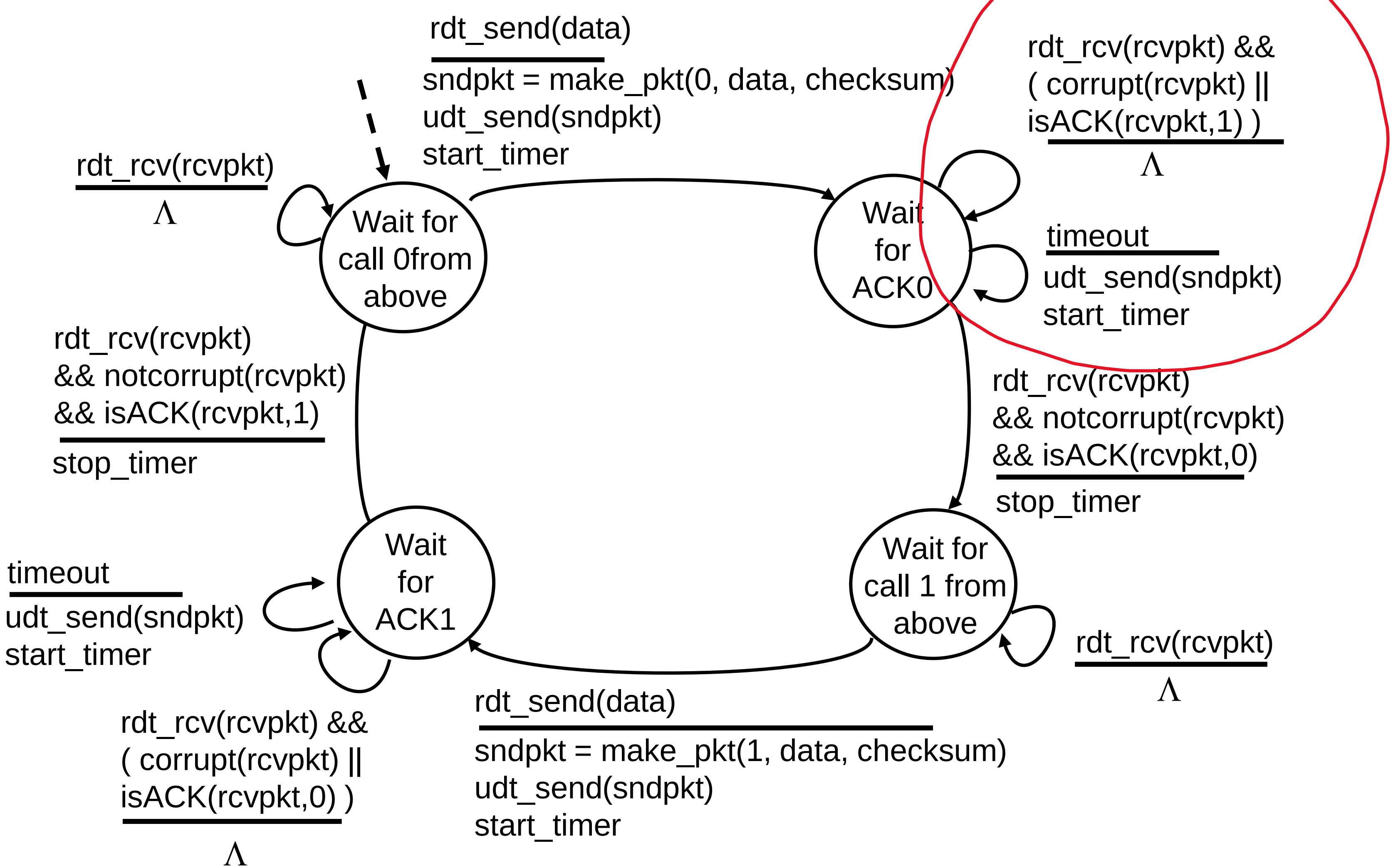
- Receiver *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

- No need to define two types of message; support general scenario
- ACK/NAK handles stop-and-wait is fine, difficult for pipeline case
  - Multiple new packets in transmission
  - Which one to be acknowledged?

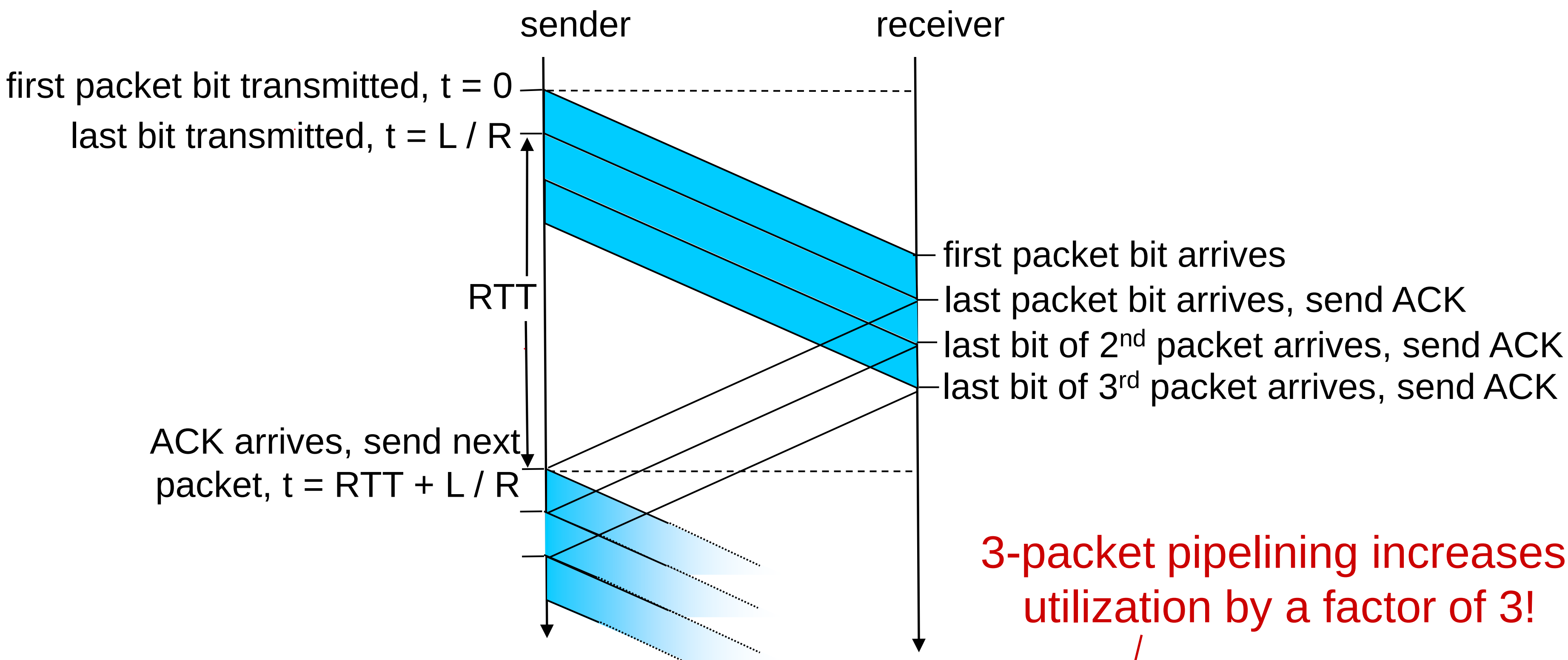
mechanisms for reliable commun. protocol design

- checksum
- feedback : (Ack + seq. number)
- retransmission
- seq. number
- Timeout

rdt3.0 sender



Pipelining: increased utilization

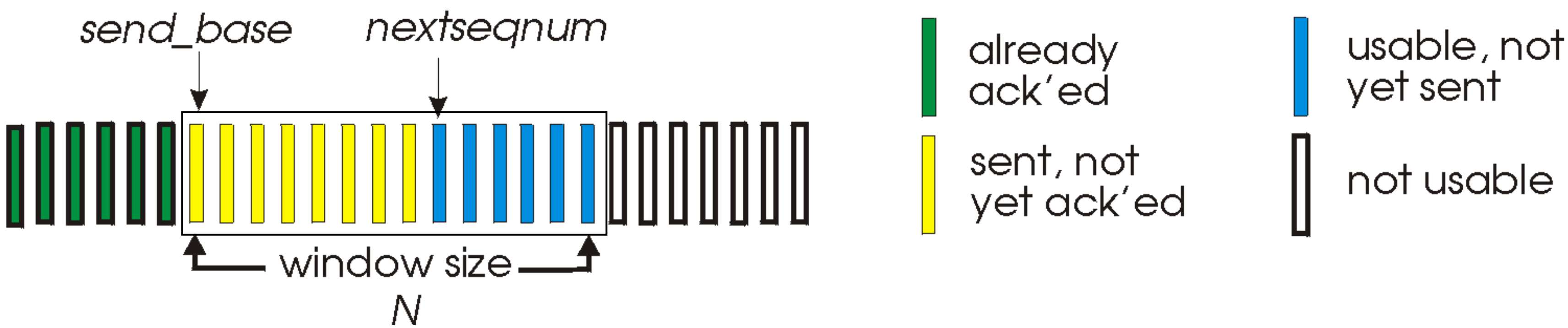


3-packet pipelining increases utilization by a factor of 3!

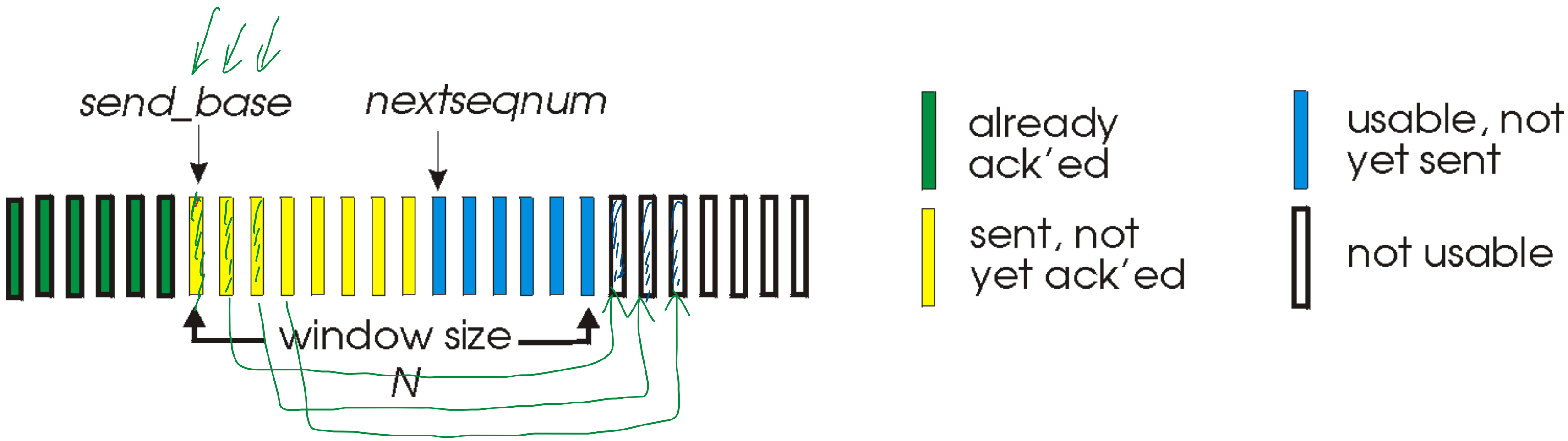
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

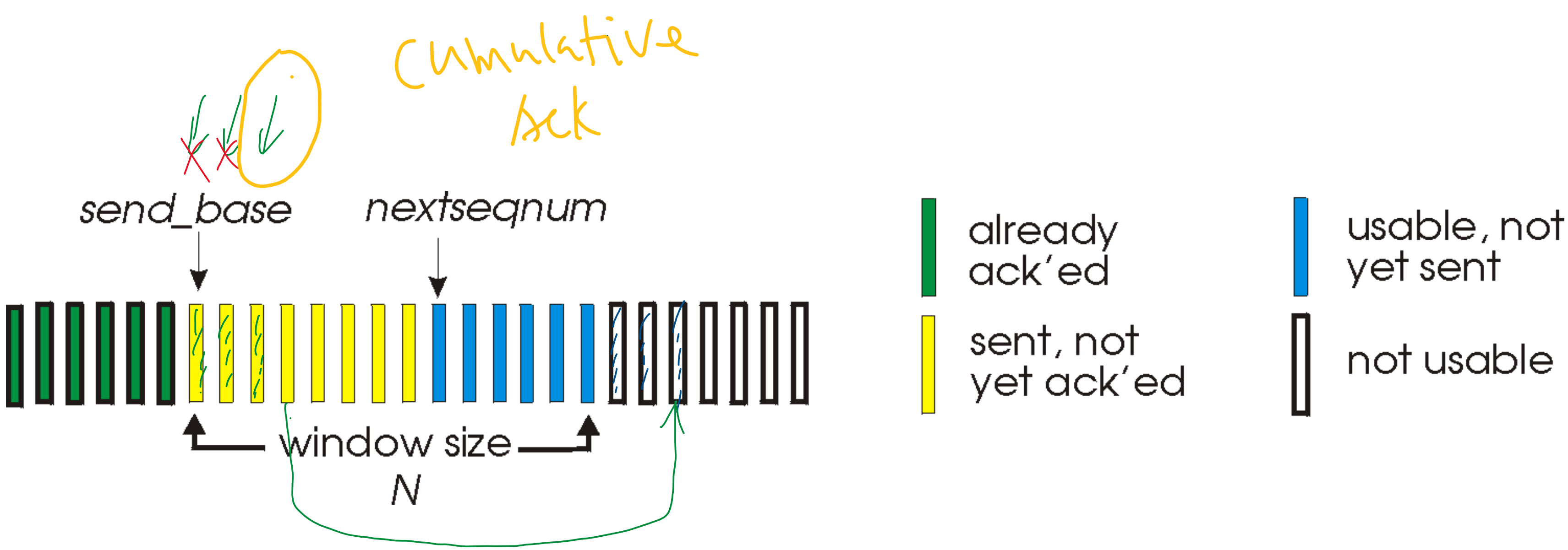
Go-Back-N: sender

- 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 oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window


send_base
nextseqnum
window size
N
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable


Cumulative Ack
send_base
nextseqnum
window size
N
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable

GBN: sender extended FSM

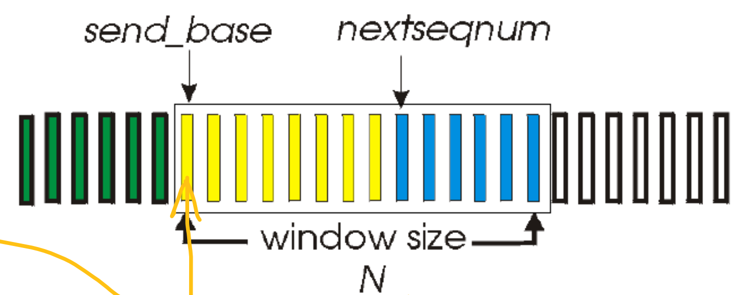
Λ

base=1
nextseqnum=1

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)
```

Wait



Timeout timer

timeout

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

All yellow ones

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

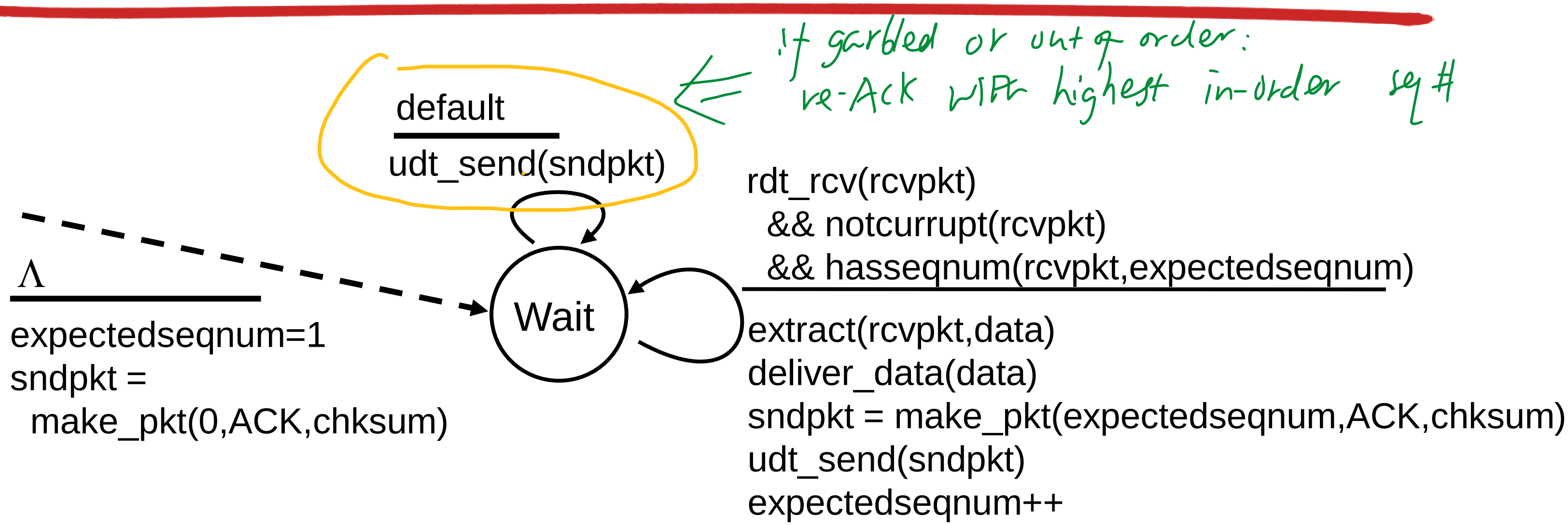
Retransmission by timeout

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

if base ≤ getacknum(rcvpkt)

GBN: receiver extended FSM

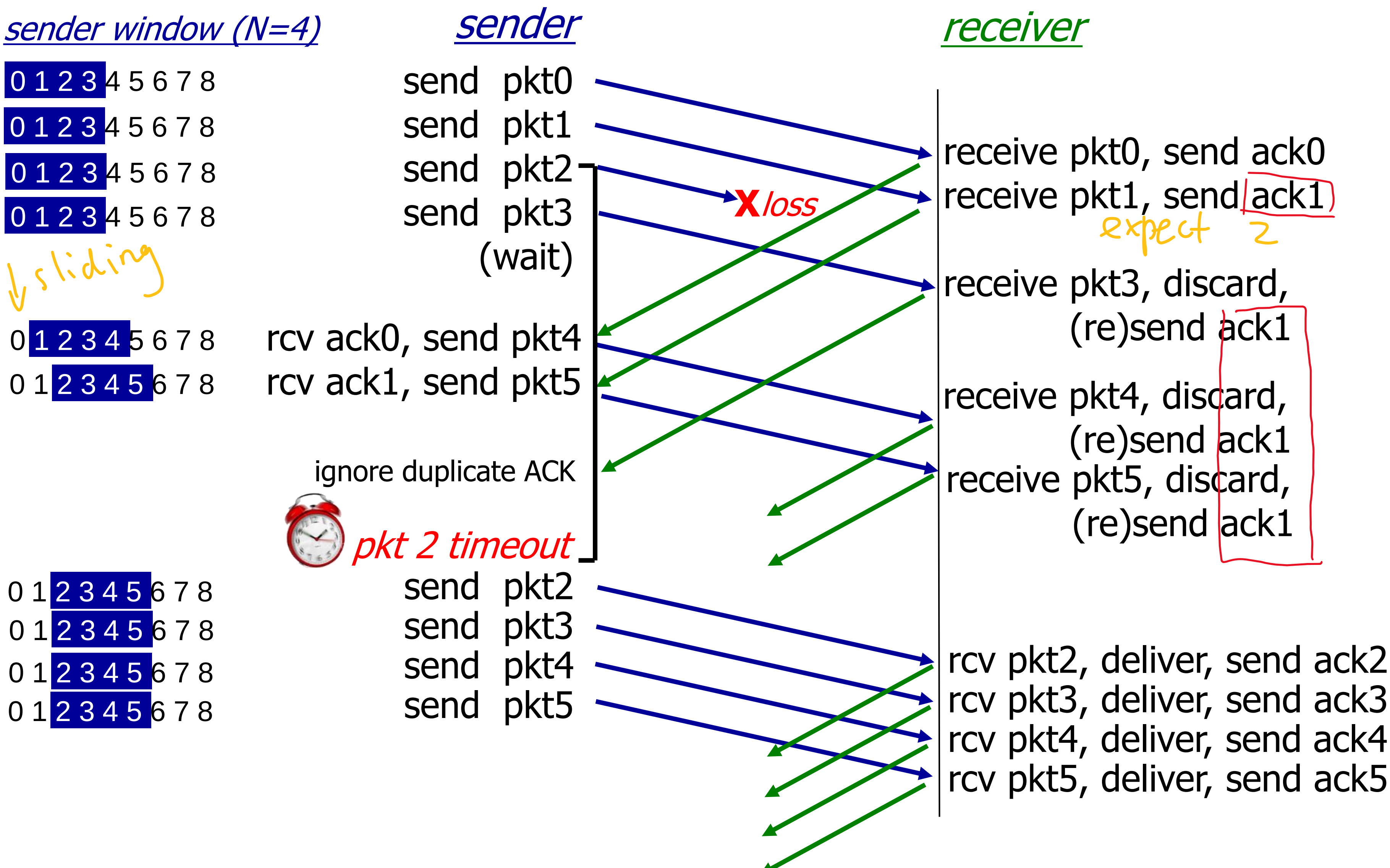


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

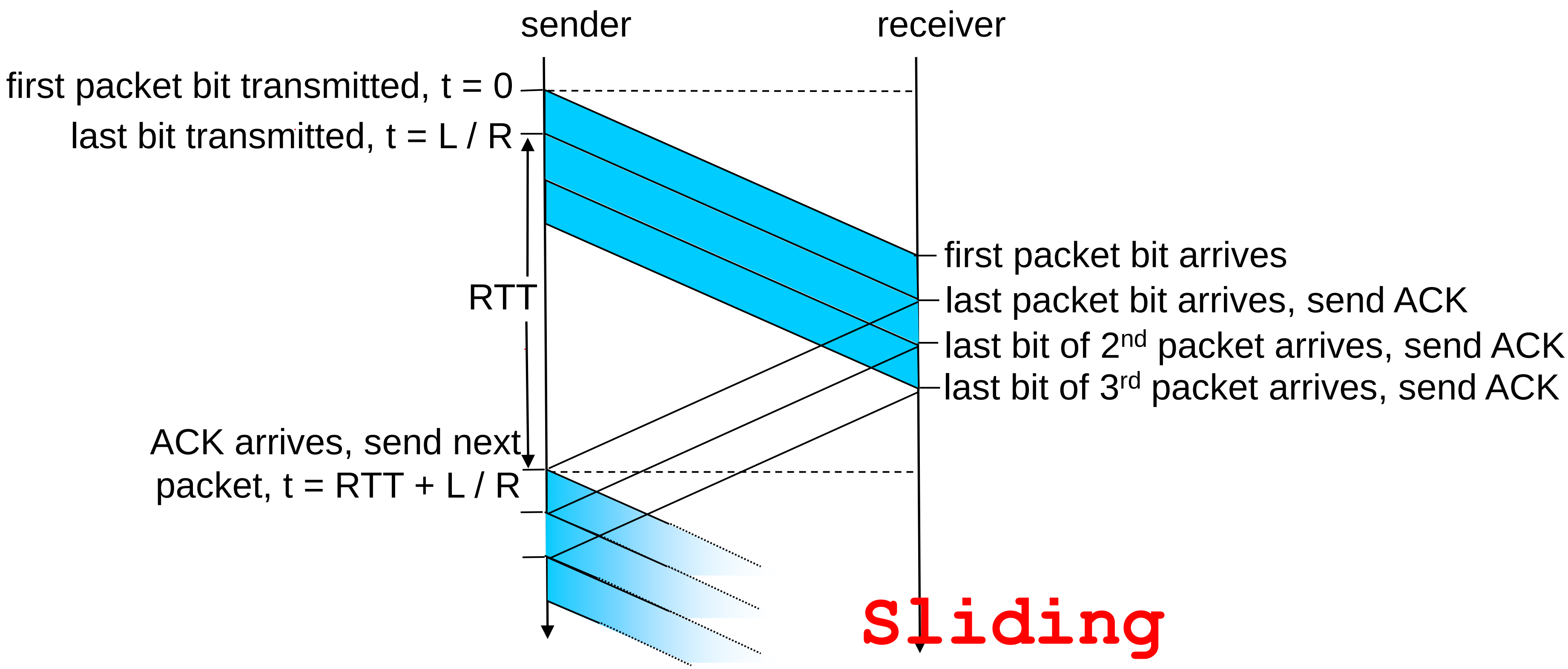
- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

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

GBN in action



Window size: sending rate control



$$U_{sender} = \frac{NL / R}{RTT + L / R}$$

Sliding

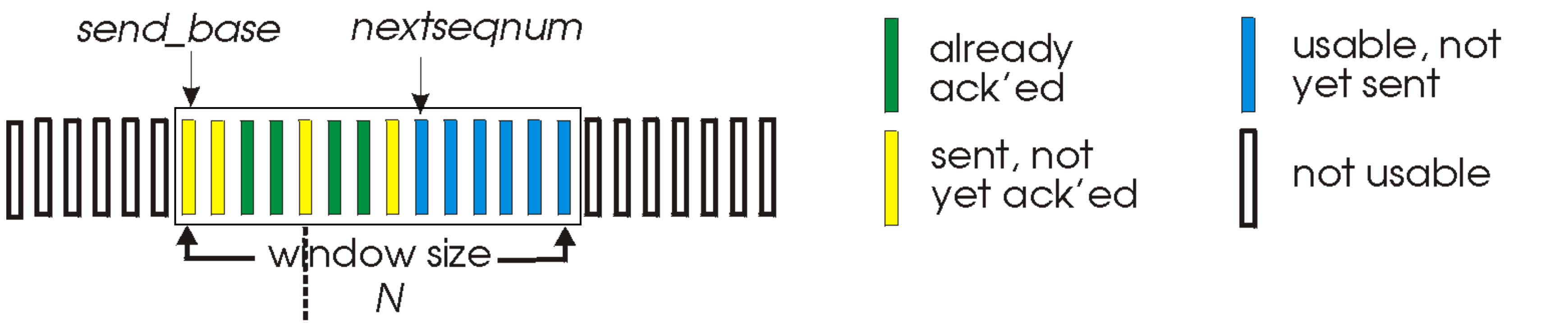
- Window of size N: up to N packets in air; states to be determined
- For the data well received in sequence, considered as processed, so move on

Class Today

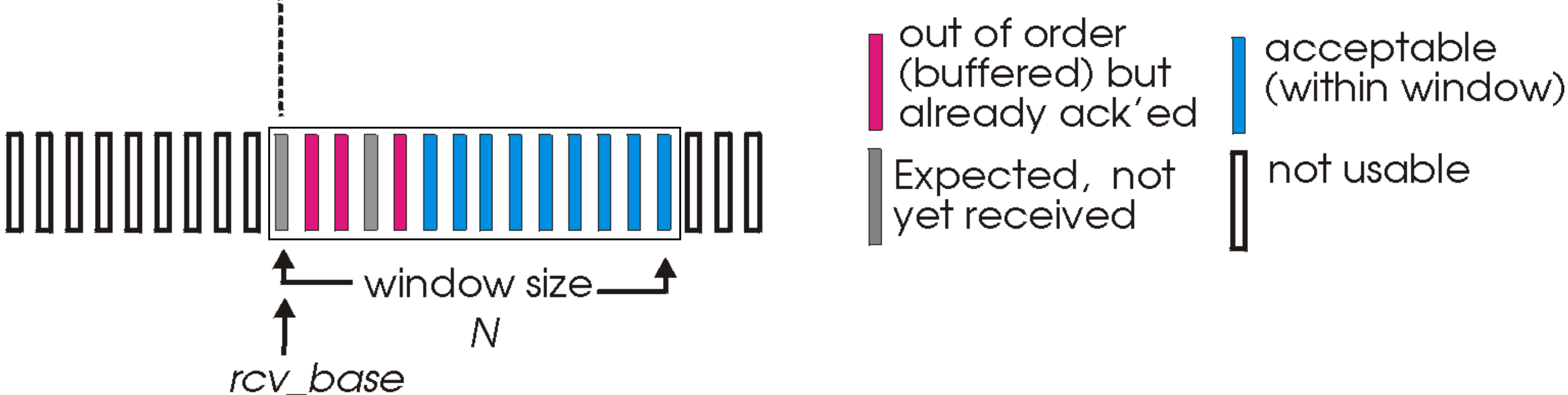
Selective repeat

- 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
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows

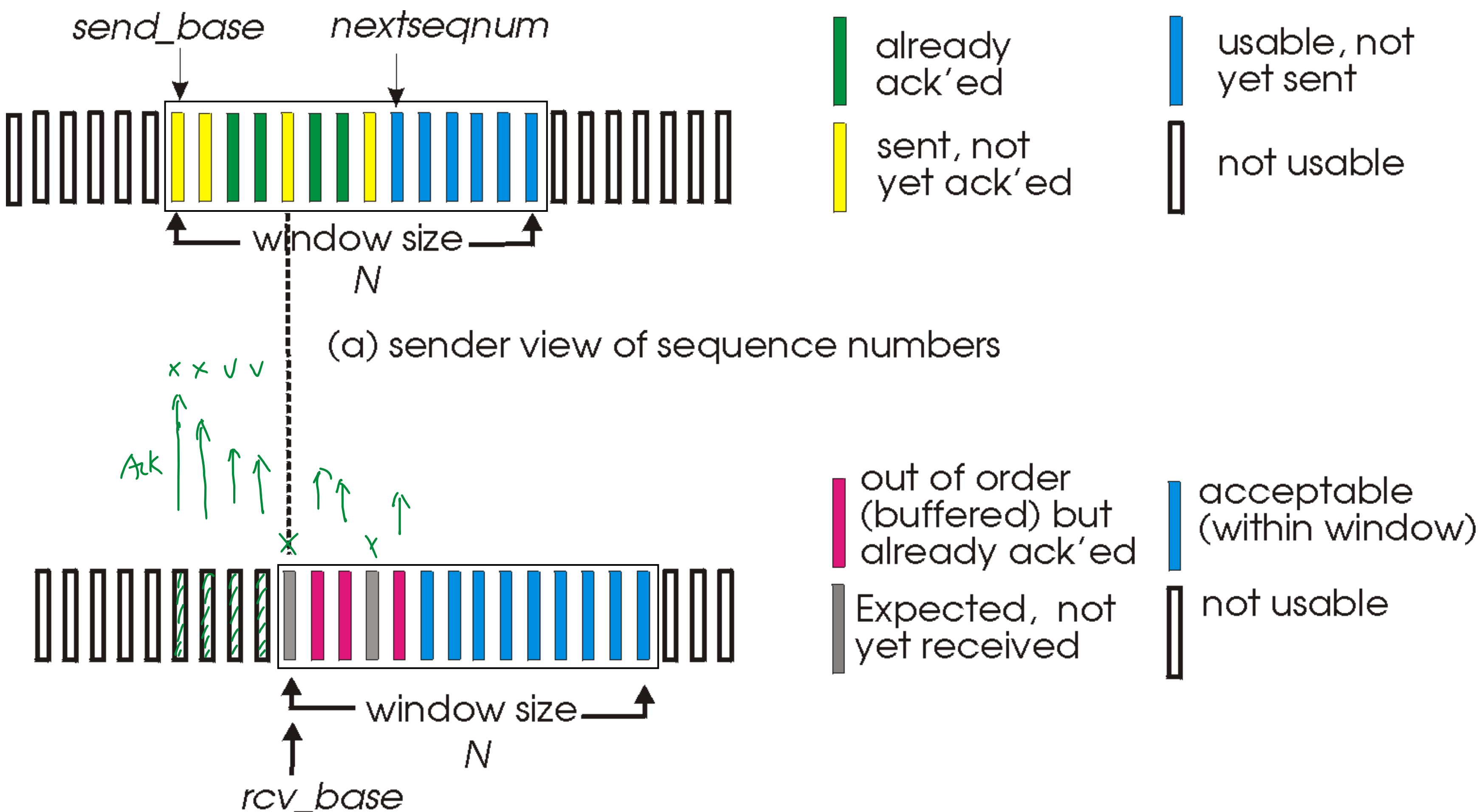


(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective repeat: sender, receiver windows



(b) receiver view of sequence numbers

Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N-1]:

- 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

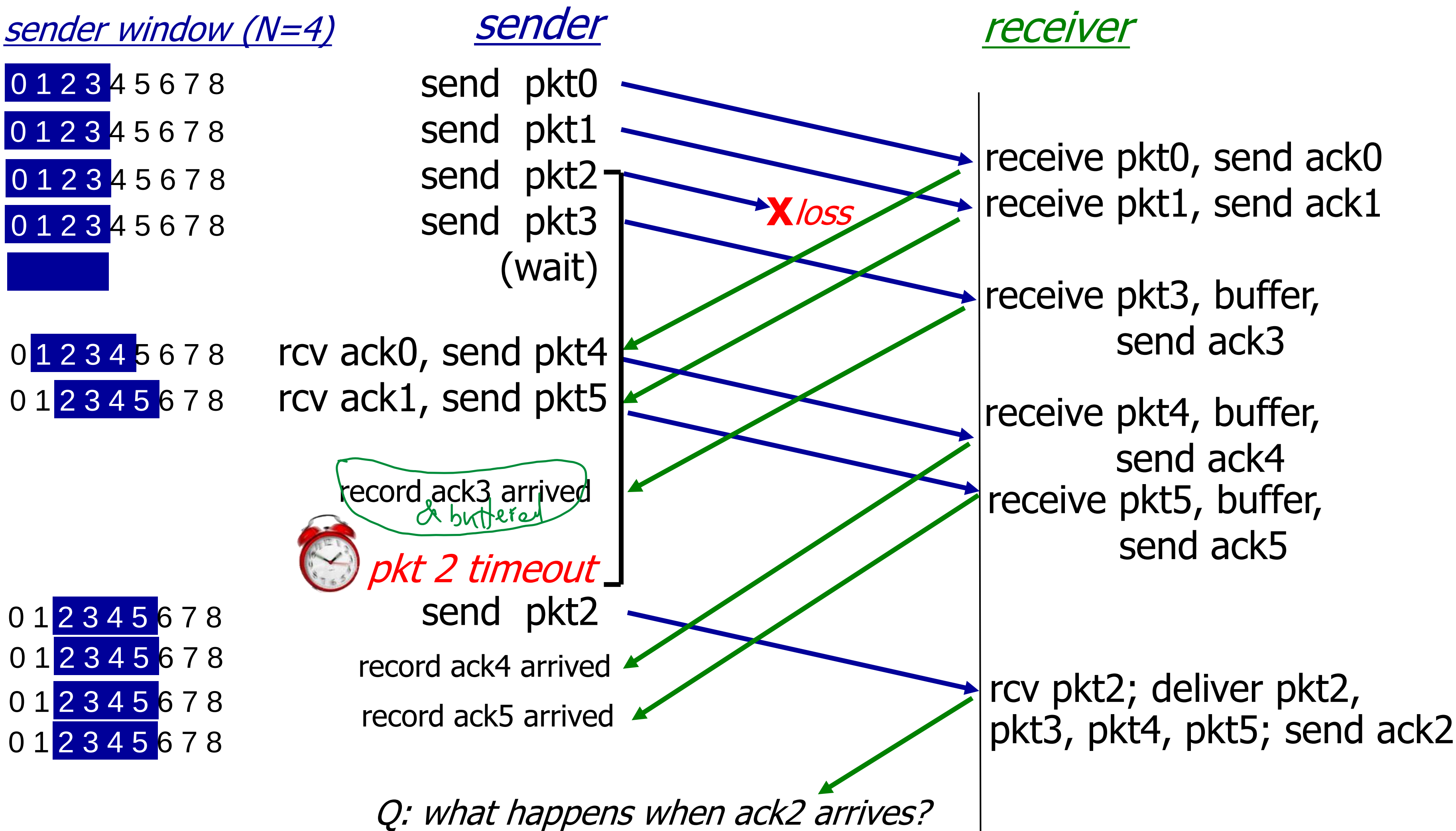
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



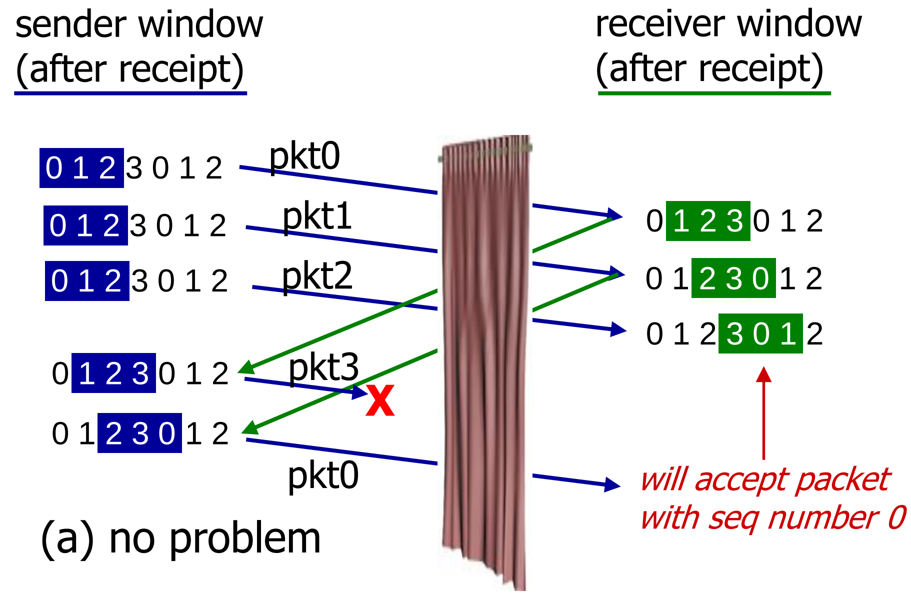
Selective repeat: dilemma

00, 01, 10, 11 | 00, 01, 10, 11

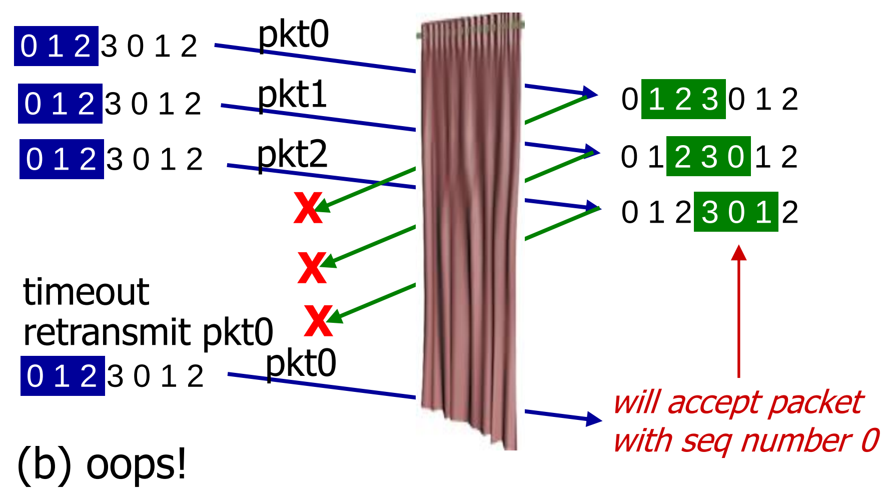
example:

- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

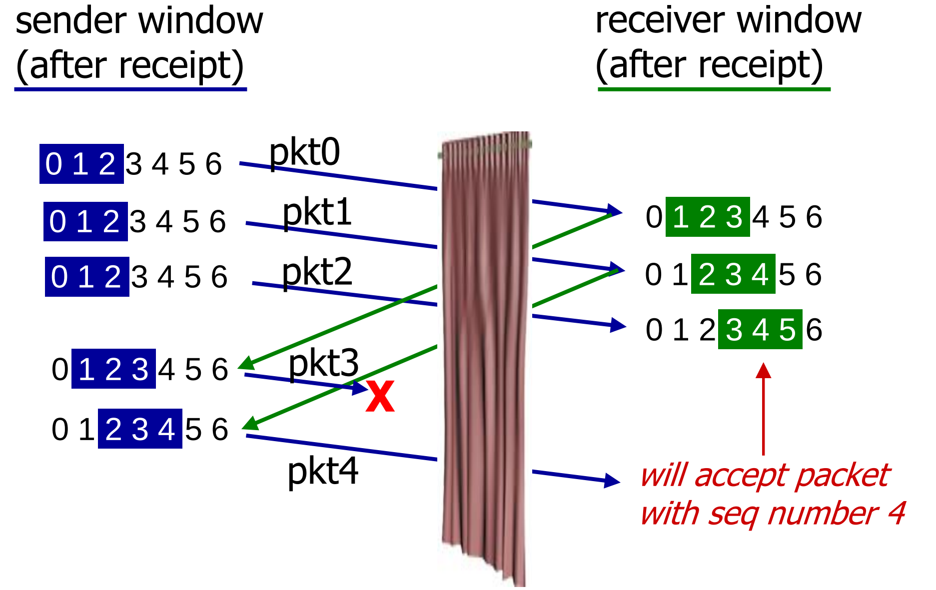
Q: what relationship between seq # size and window size to avoid problem in (b)?



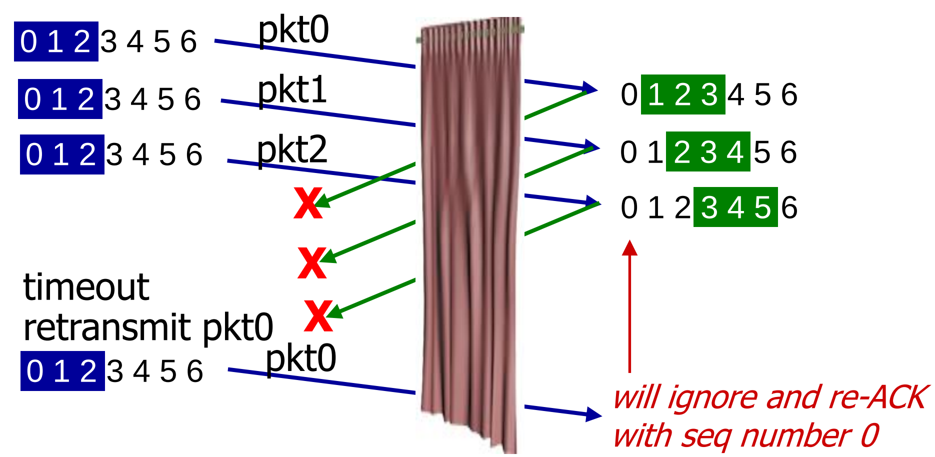
receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



Address the dilemma: with sequence number



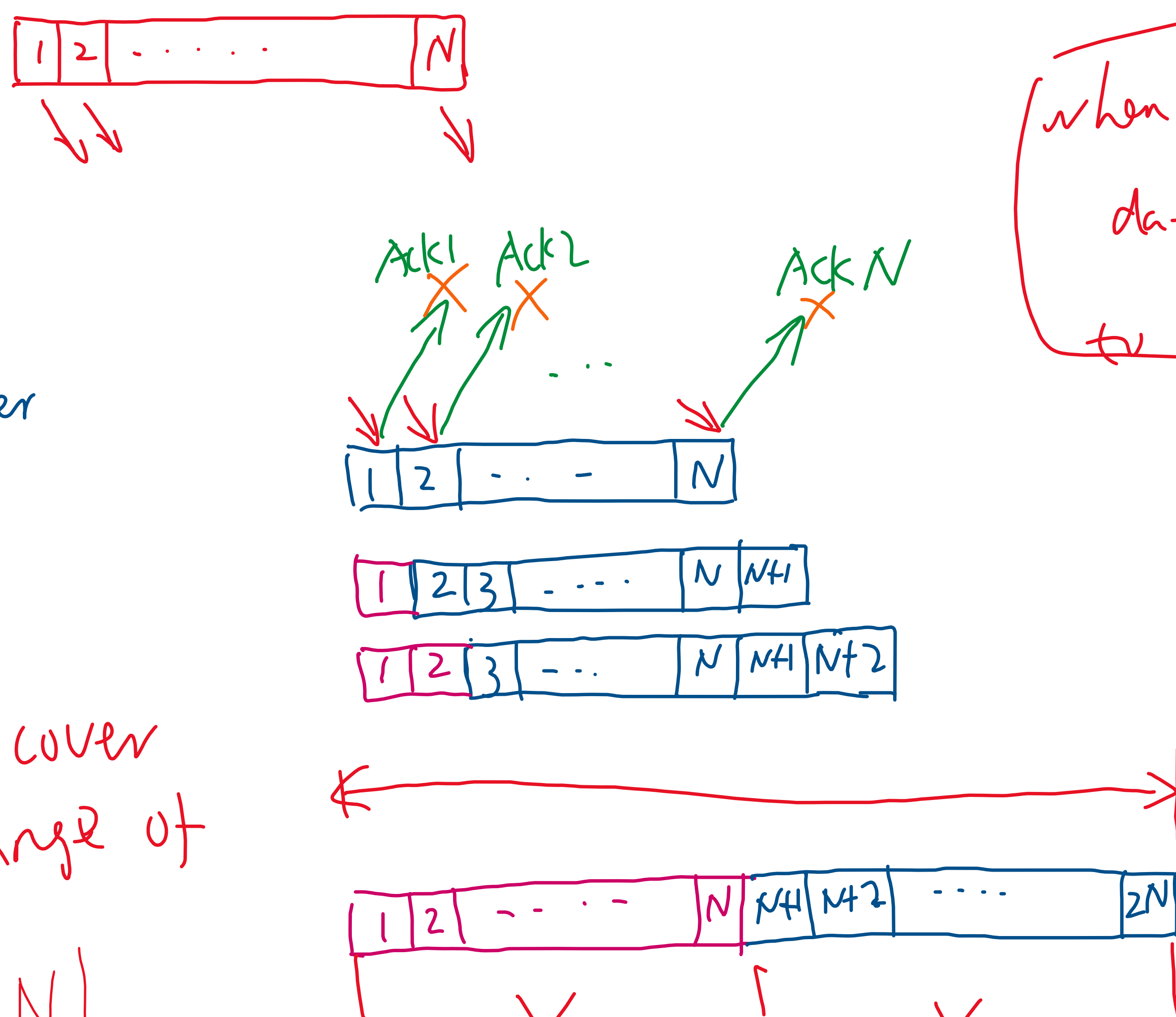
(a) Receive the new pkt4



(b) Receive the retransmitted pkt 0

Sequence number size Analysis

sender



when out of order data needs to be buffered

Receiver

seq # cover range of

$\log_2(2N)$

$\Rightarrow$ # of bits for seq #.

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

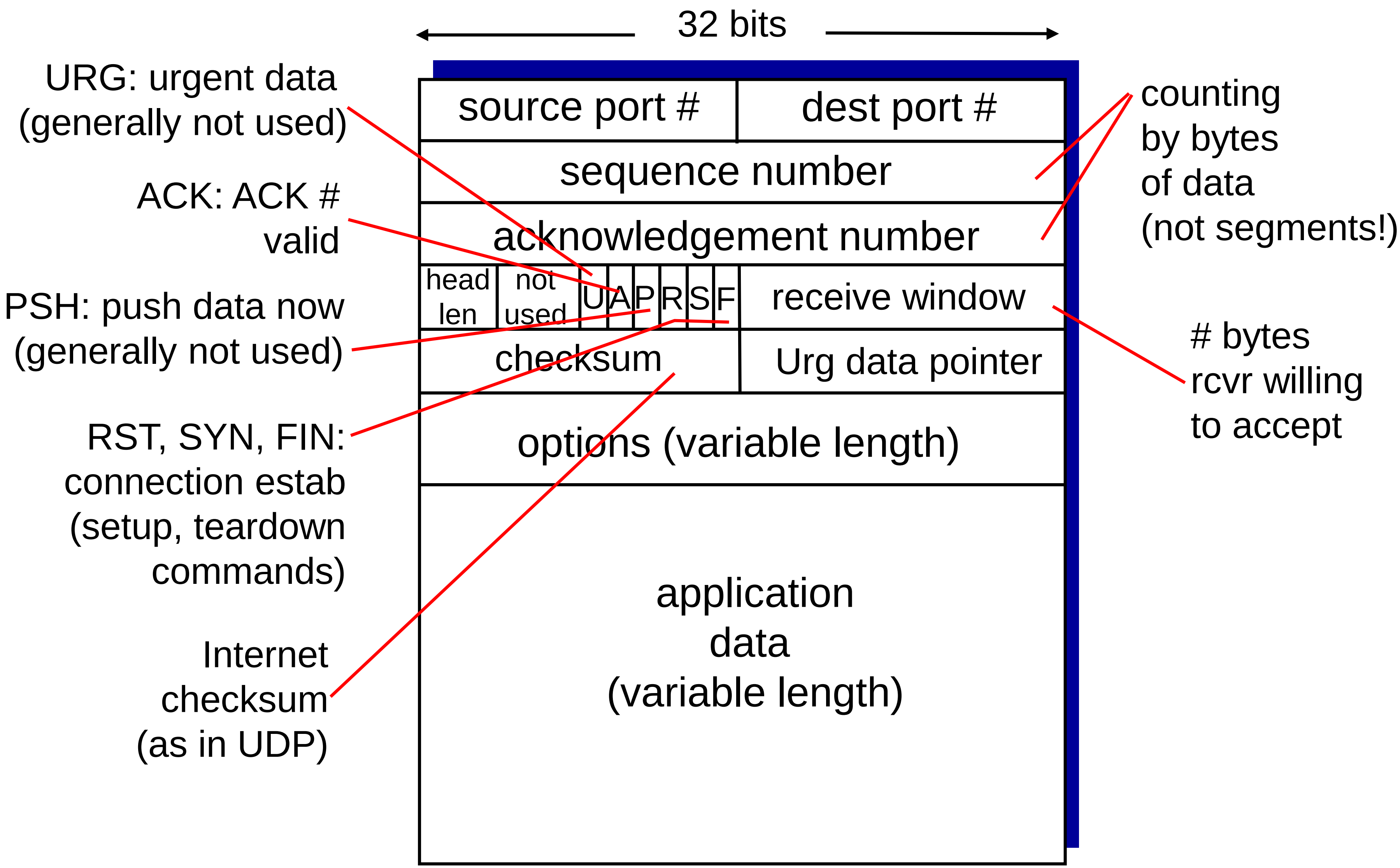
3.6 principles of congestion control

3.7 TCP congestion control

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
  - Stream of bytes instead of number of segments facilitates quickly putting data into the right position.
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

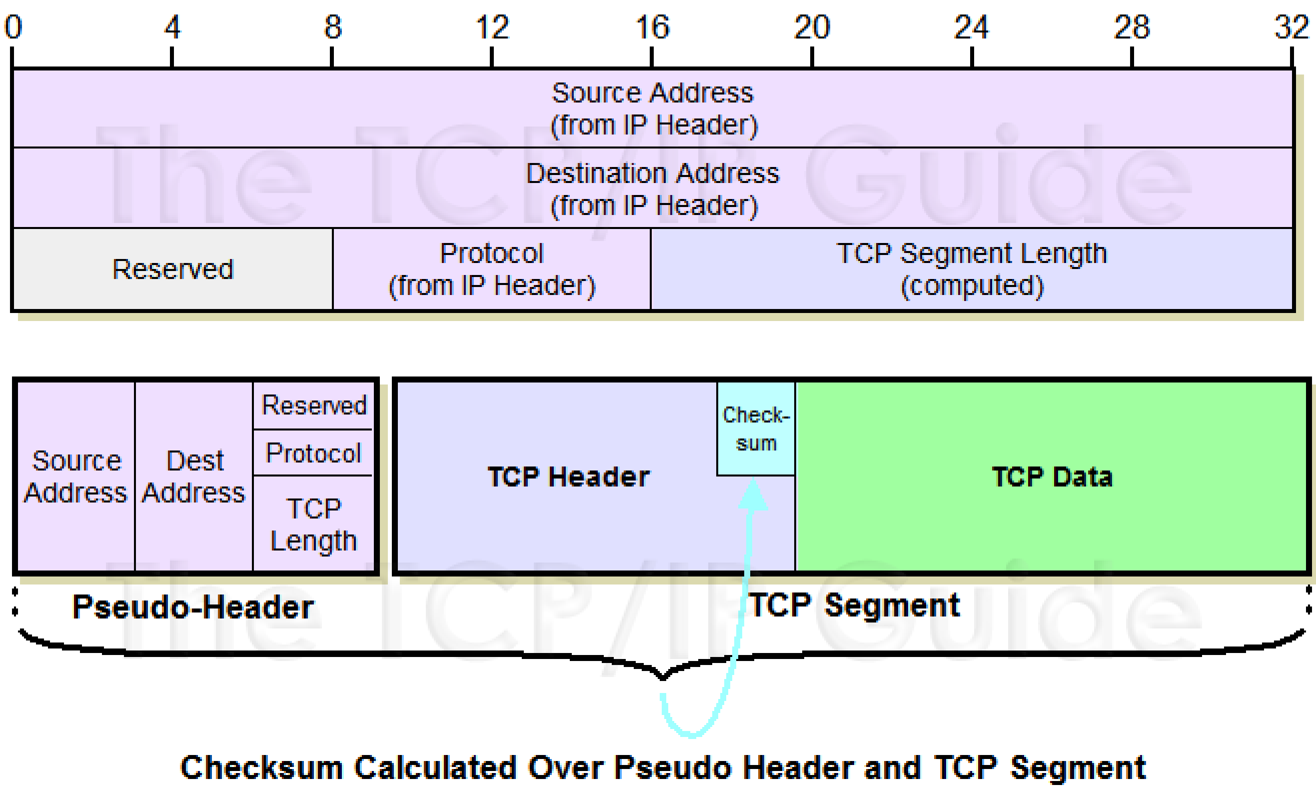


More on TCP segment

- "Options" field:
  - negotiate maximum segment size; scaling factor for high speed network; timestamp
- Flag bits:
  - PSH: indicate receiver to pass data to upper layer immediately
  - URG: Sender upper layer indicate data urgent
  - Urg data pointer: pointer for the urgent data

TCP Checksum

- To calculate the TCP segment header's Checksum field, the TCP pseudo header is first constructed and placed, logically, before the TCP segment. The checksum is then calculated over both the pseudo header and the TCP segment. The pseudo header is then discarded. (http://www.tcpipguide.com/free/t_TCPChecksumCalculationandtheTCPPseudoHeader-2.htm)



TCP seq. numbers, ACKs

sequence numbers:

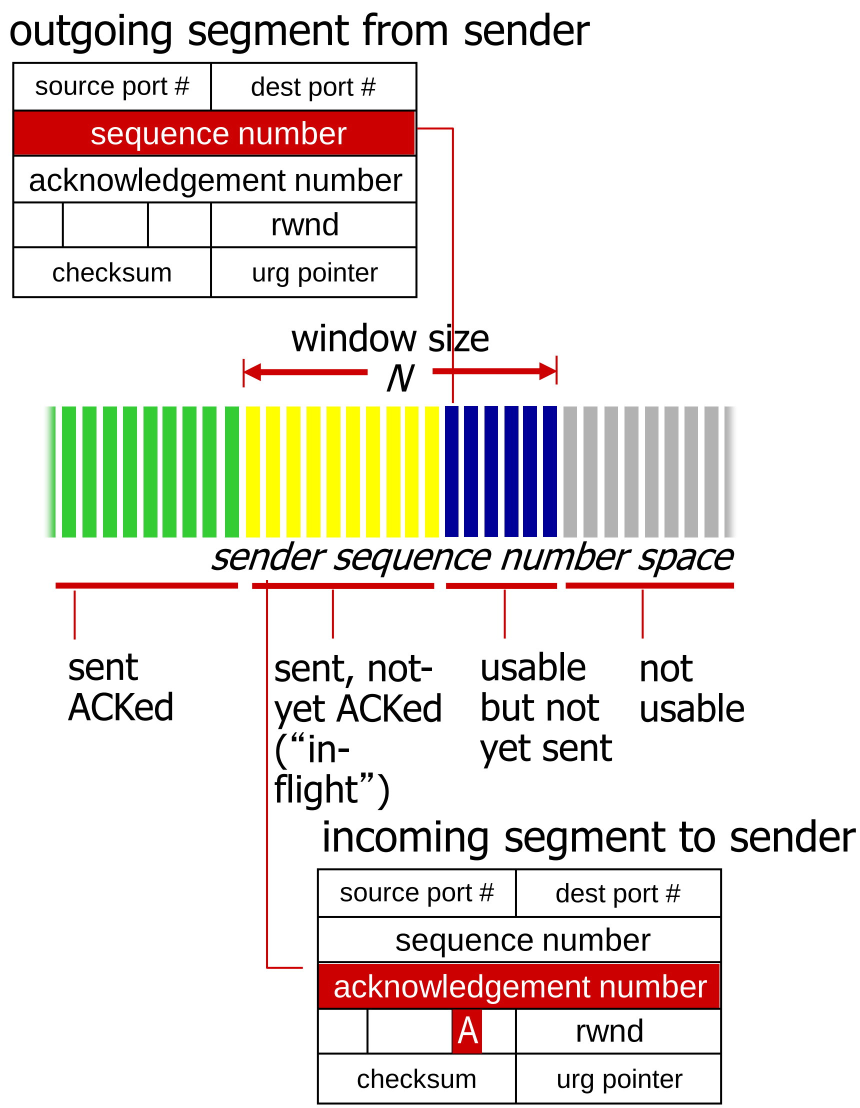
- byte stream "number" of first byte in segment's data

acknowledgements:

- 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

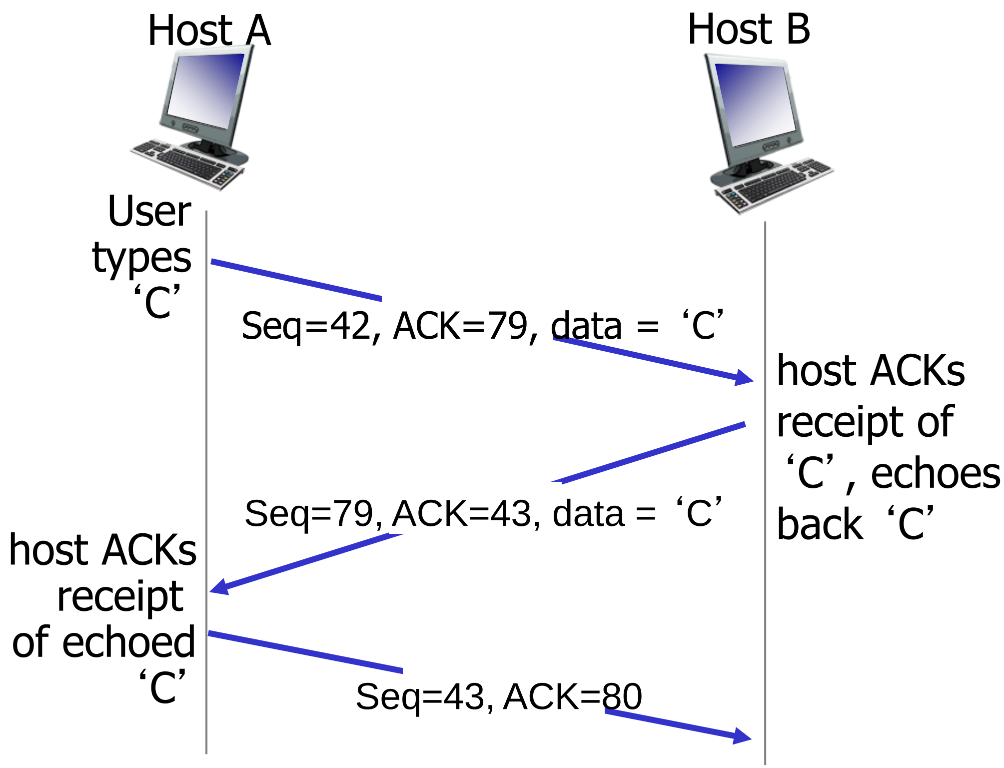


TCP seq. numbers, ACKs

seq. number

Ack. number

- - - - -



seq. number

Ack. number

- - -

A → B

B → A

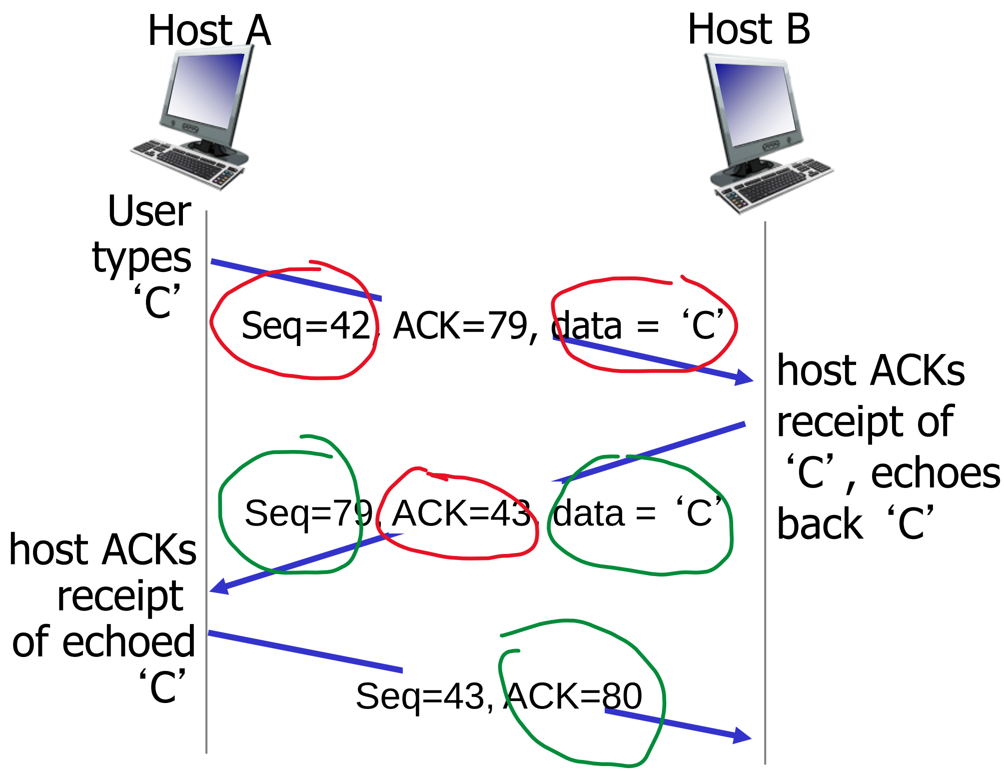
simple telnet scenario

TCP seq. numbers, ACKs

seq. number
Ack. number

A → B

B → A



simple telnet scenario

Seq. number
Ack. number

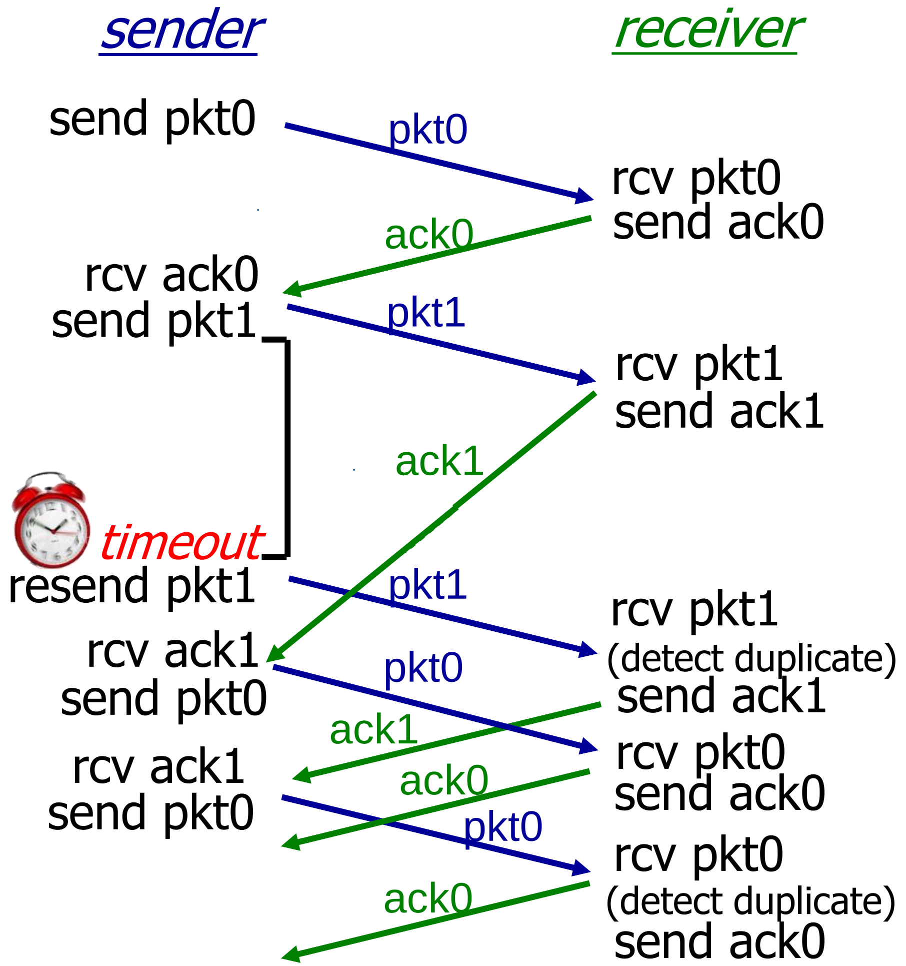
TCP round trip time, timeout

Q: how to set TCP timeout value?

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

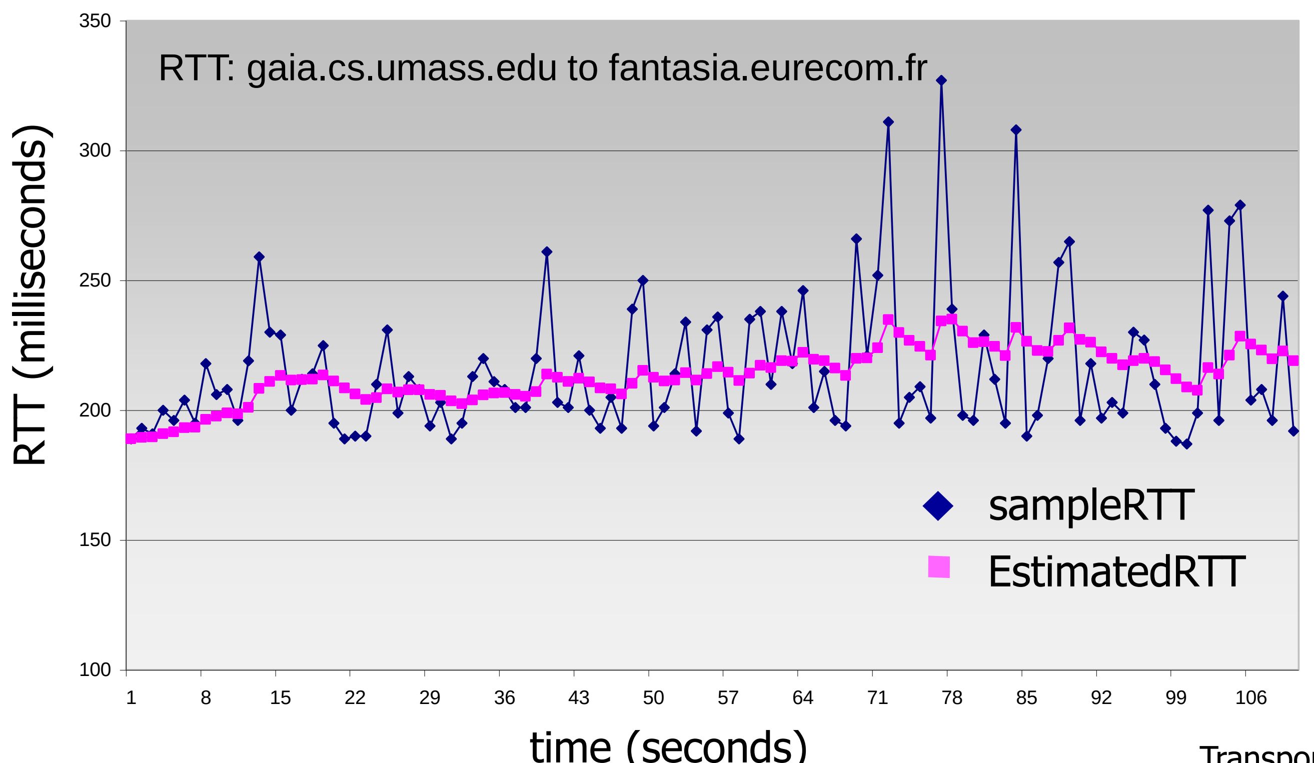


(d) premature timeout/ delayed ACK

TCP round trip time, timeout

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

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



Exponential Weighted Moving Everage

EstimateRTT = SampleRTT$_1$

i=1

While (1)

i++;

$EstimateRTT = (1-\alpha)\,\underline{EstimateRTT} + \alpha\,SampleRTT_i$

end

$EstimateRTT = SampleRTT_1$

$EstimateRTT = (1-\alpha)\,SampleRTT_1 + \alpha\,SampleRTT_2$

$EstimateRTT = (1-\alpha)\left[(1-\alpha)\,SampleRTT_1 + \alpha\,SampleRTT_2\right] + \alpha\,SampleRTT_3$

$= (1-\alpha)^2\,SampleRTT_1 + (1-\alpha)\alpha\,SampleRTT_2 + \alpha\,SampleRTT_3$

$- \cdots -$

$= (1-\alpha)^{n-1}\,SampleRTT_1 + (1-\alpha)^{n-2}\alpha\,SampleRTT_2 + \cdots + (1-\alpha)\alpha\,SampleRTT_{n-1} + \alpha\,SampleRTT_n$

TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

               (typically, β = 0.25)
```

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

estimated RTT

"safety margin"

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/