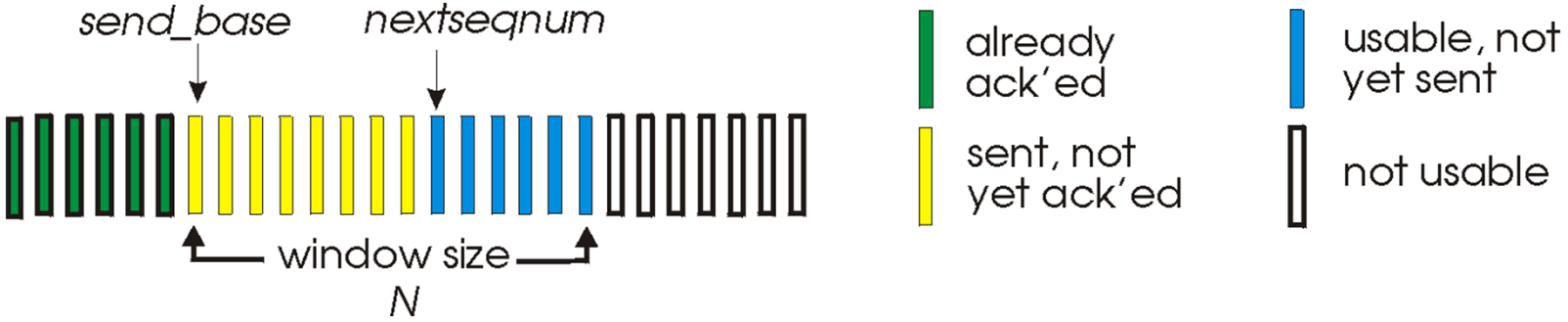
Go-Back-N: sender side

- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
- sender: "sliding window" of up to *N* (consecutive sent but unACKed) pkts
  - k-bit seq # in pkt header

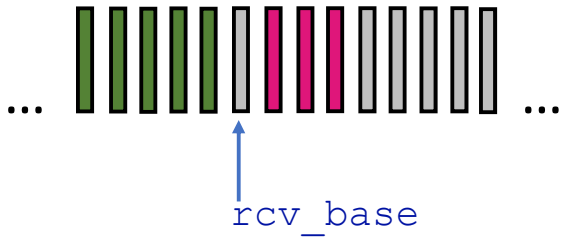


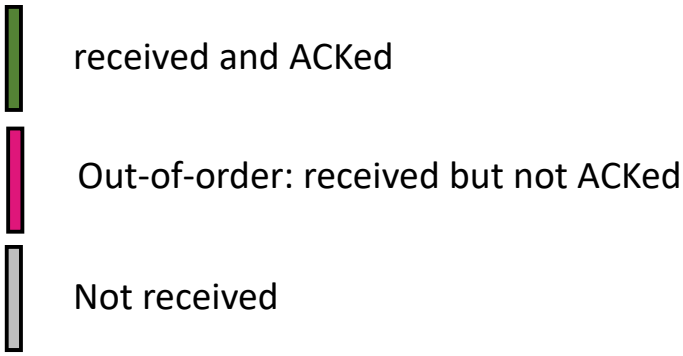
- on receiving ACK(n): move window forward to begin at *n*+1
- timer for oldest unACKed packet.
  - timer starts if either (non-empty) window moves forward or app-layer sends a pkt to empty window
- *timeout(n):* retransmit packet *n* and all (available) packets in window

Go-Back-N: receiver side

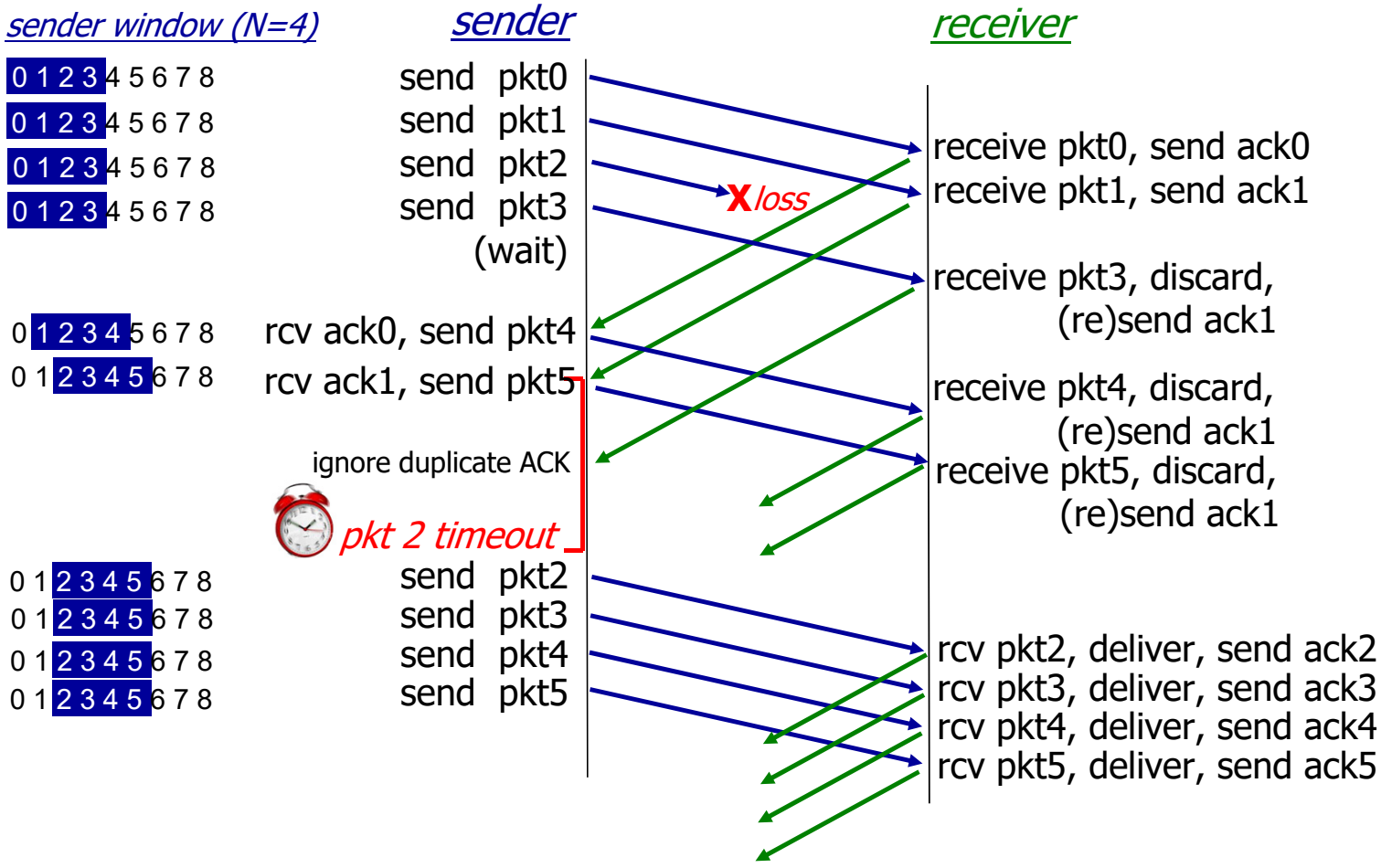
- Always send ACK for correctly-received packet, with highest *in-order* seq #
  - need only remember `rcv_base`
  - might generate duplicate ACKs
- on receipt of in-order packet:
  - update `rcv_base`
  - send ACK with the (in-order) seq #
- on receipt of out-of-order packet:
  - discard (don't buffer) or buffer the packet, which is an implementation decision
  - re-ACK with highest in-order seq #

Receiver view of sequence number space:





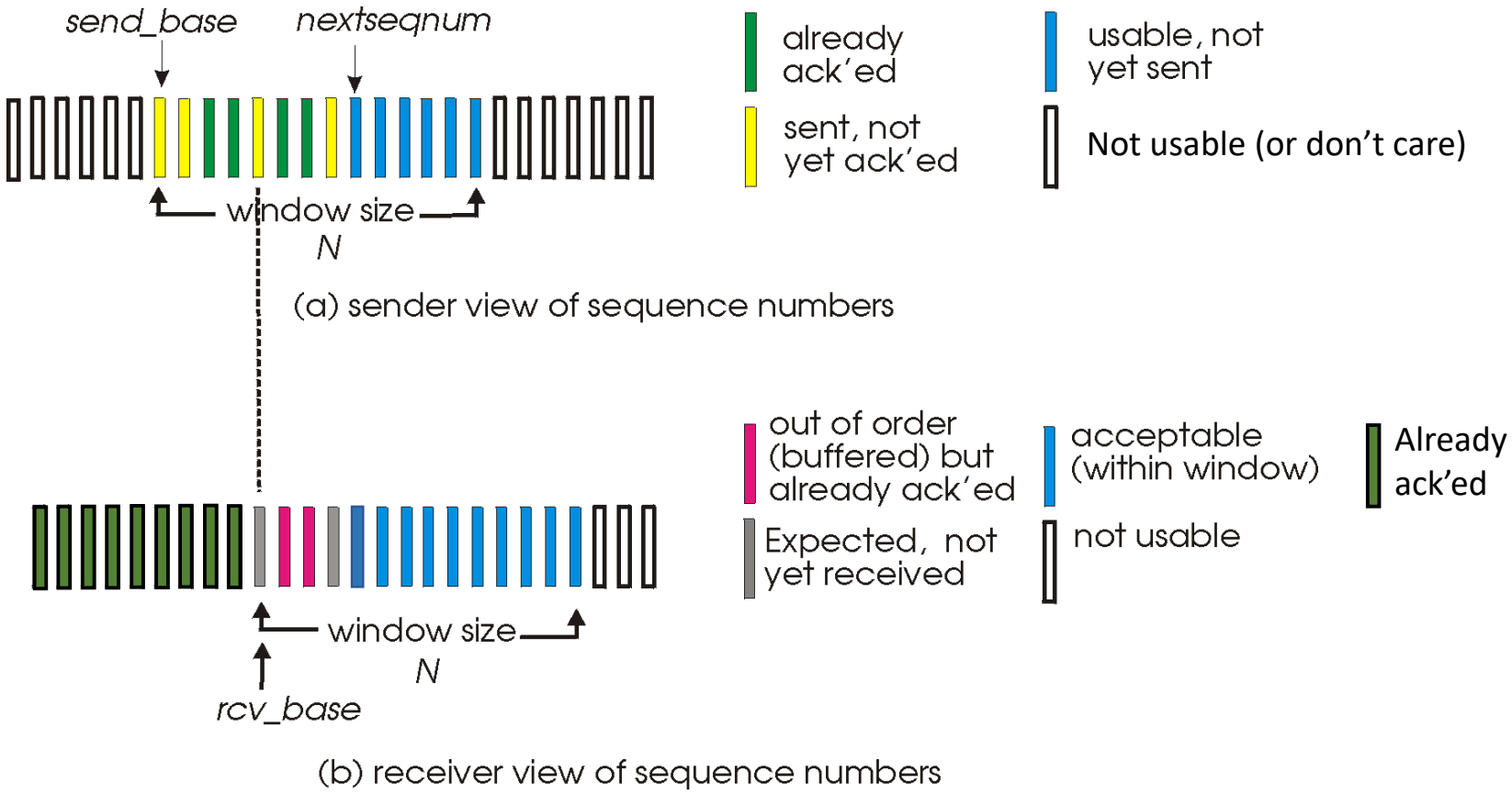
Go-Back-N in action



Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - *N* consecutive seq #
  - limits # of sent and still unACKed packets

Selective repeat: sender, receiver windows



Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

- mark packet *n* as received
- if *n* is oldest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase+N-1]

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

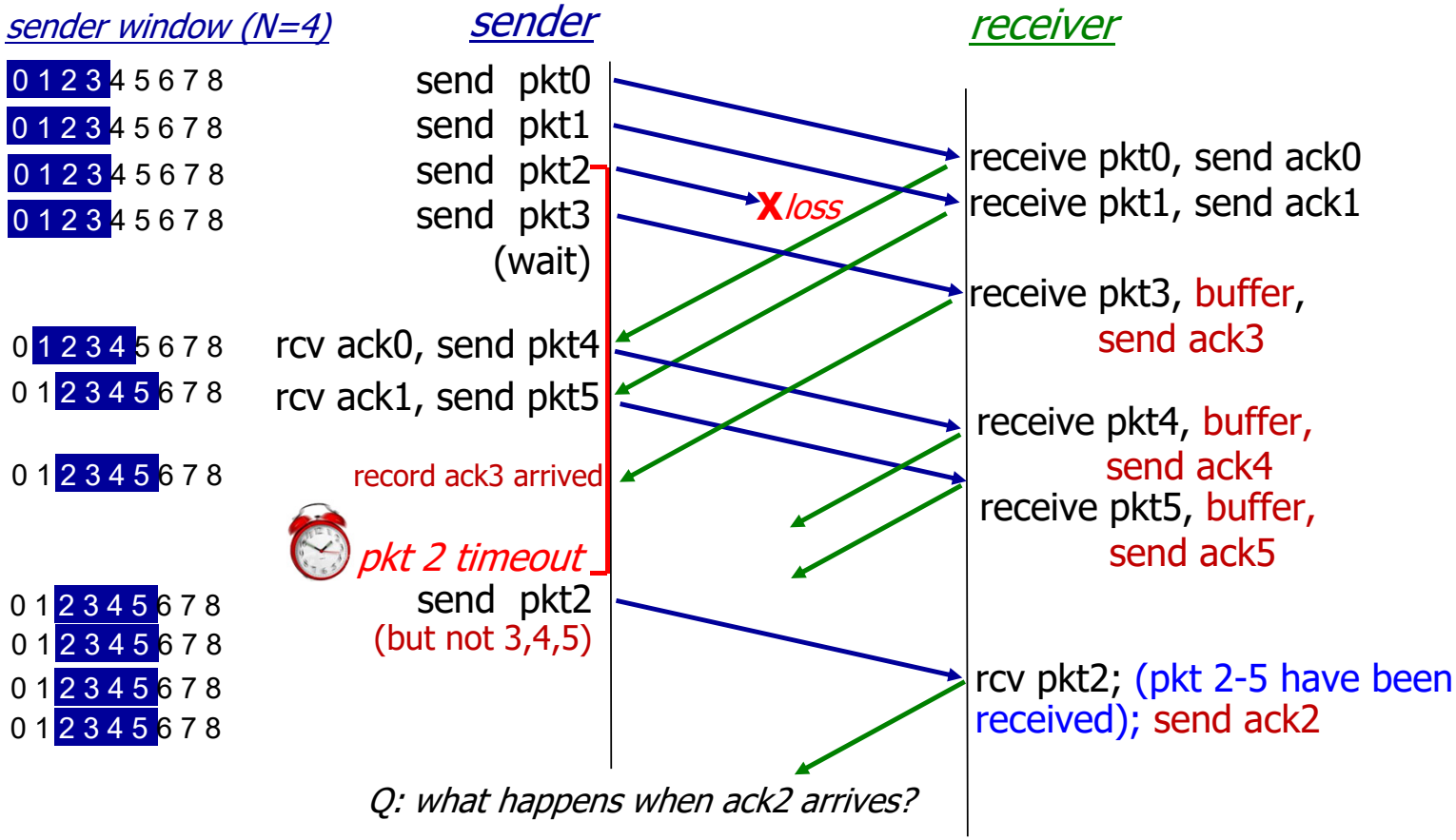
packet *n* in [rcvbase-N, rcvbase-1]

- send ACK(*n*) back to the sender

otherwise:

- ignore

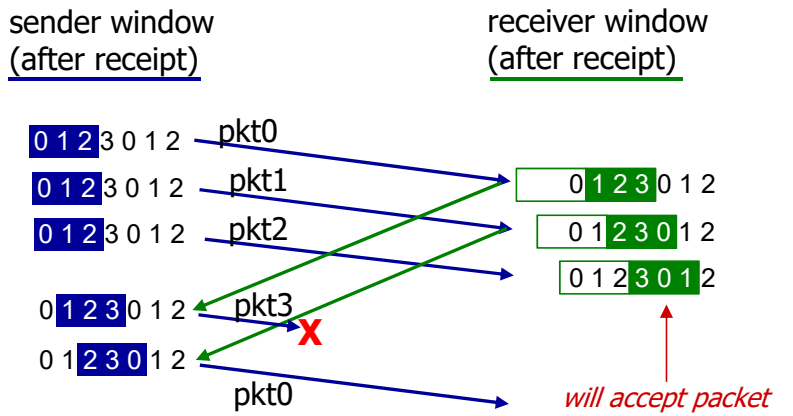
Selective Repeat in action



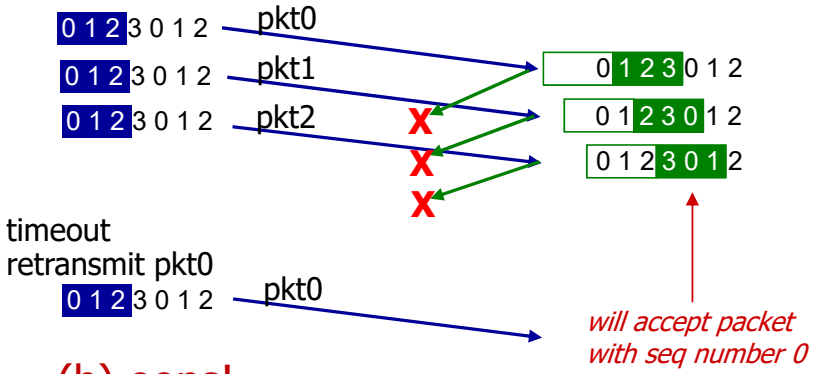
Selective repeat: a dilemma!

example:

- use 2-bit seq # : {0, 1, 2, 3}
- window size = 3



(a) no problem



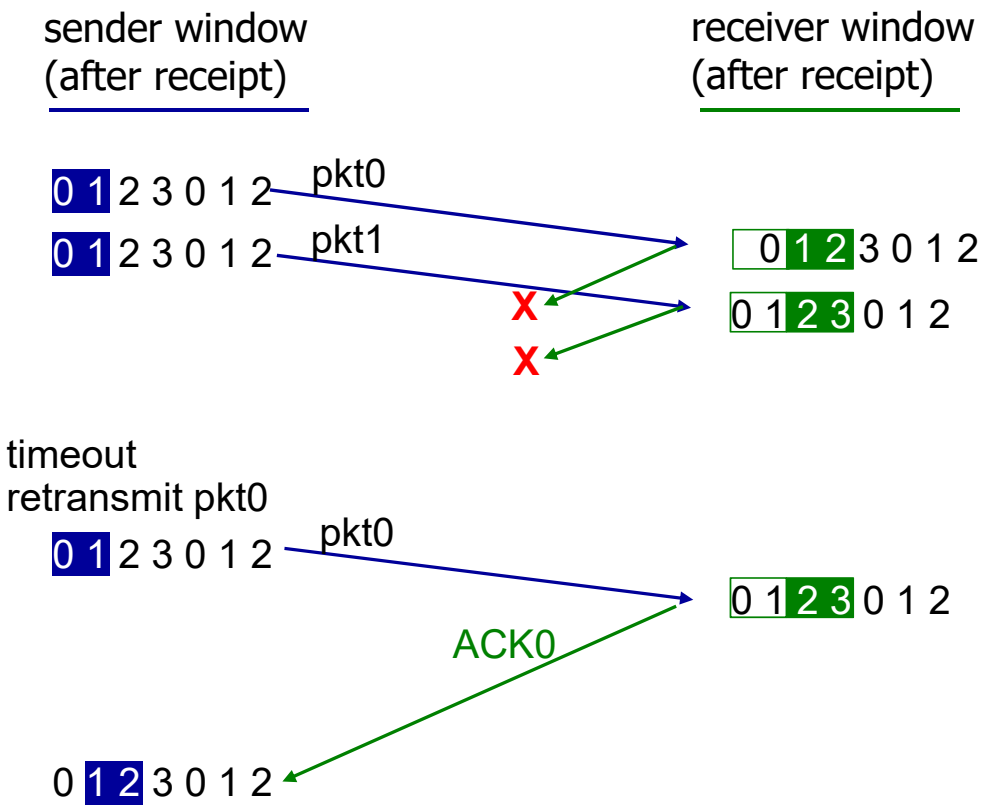
(b) oops!

Selective repeat: a dilemma!

example:

- use 2-bit seq # : {0, 1, 2, 3}
- window size = 2

Q: what relationship is needed between sequence # size and window size to avoid the problem in scenario (b)?



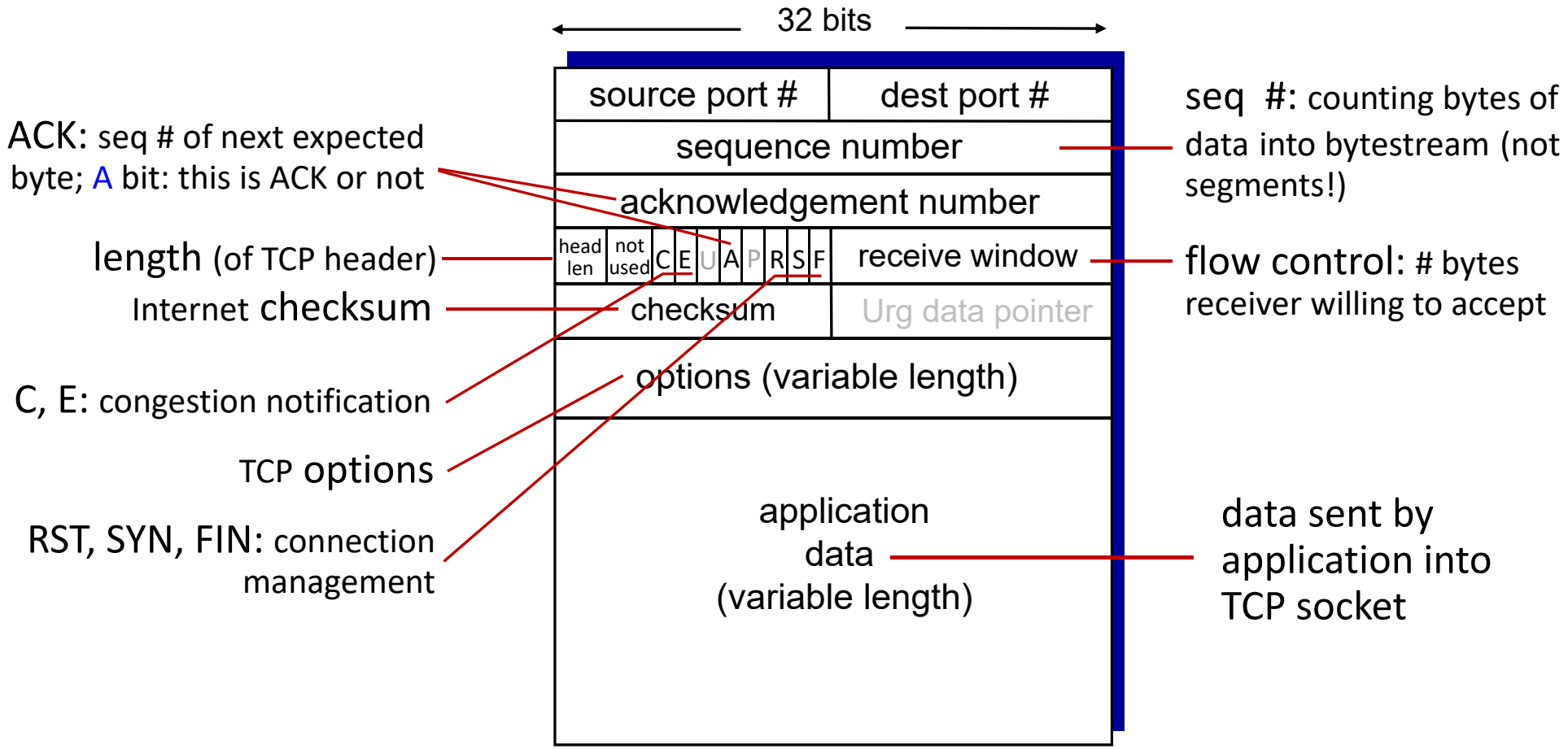
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no “message boundaries“
  - MSS: maximum segment size
- full duplex data:
  - bi-directional data flow in same (TCP) connection
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP sequence numbers and ACKs

Sequence numbers:

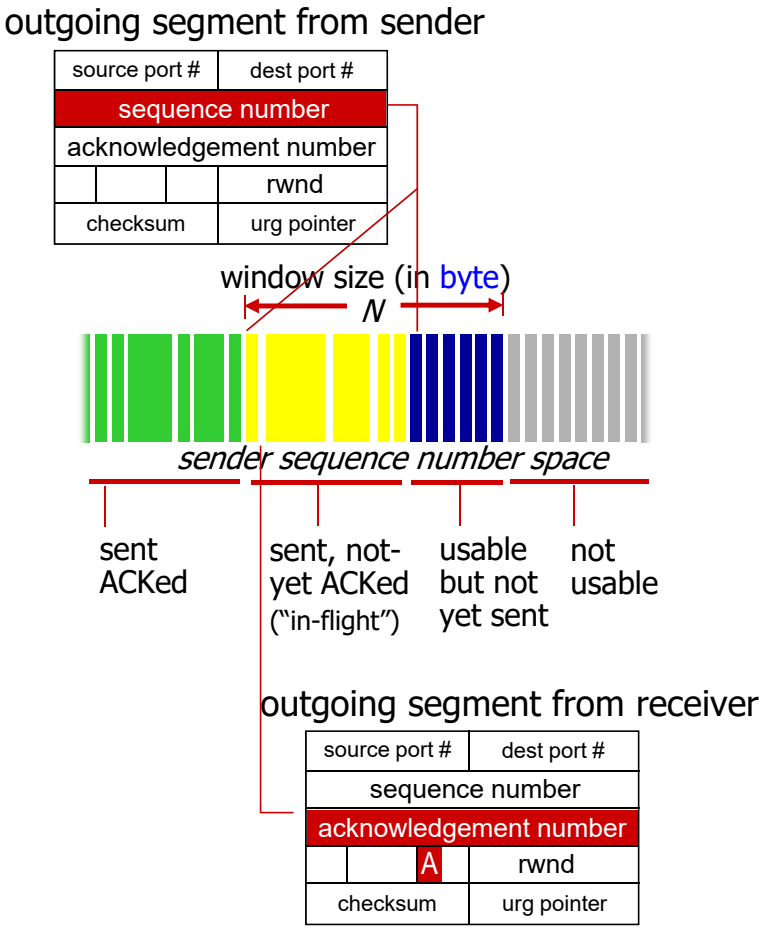
- byte stream "number" of first byte in segment's data

Acknowledgements:

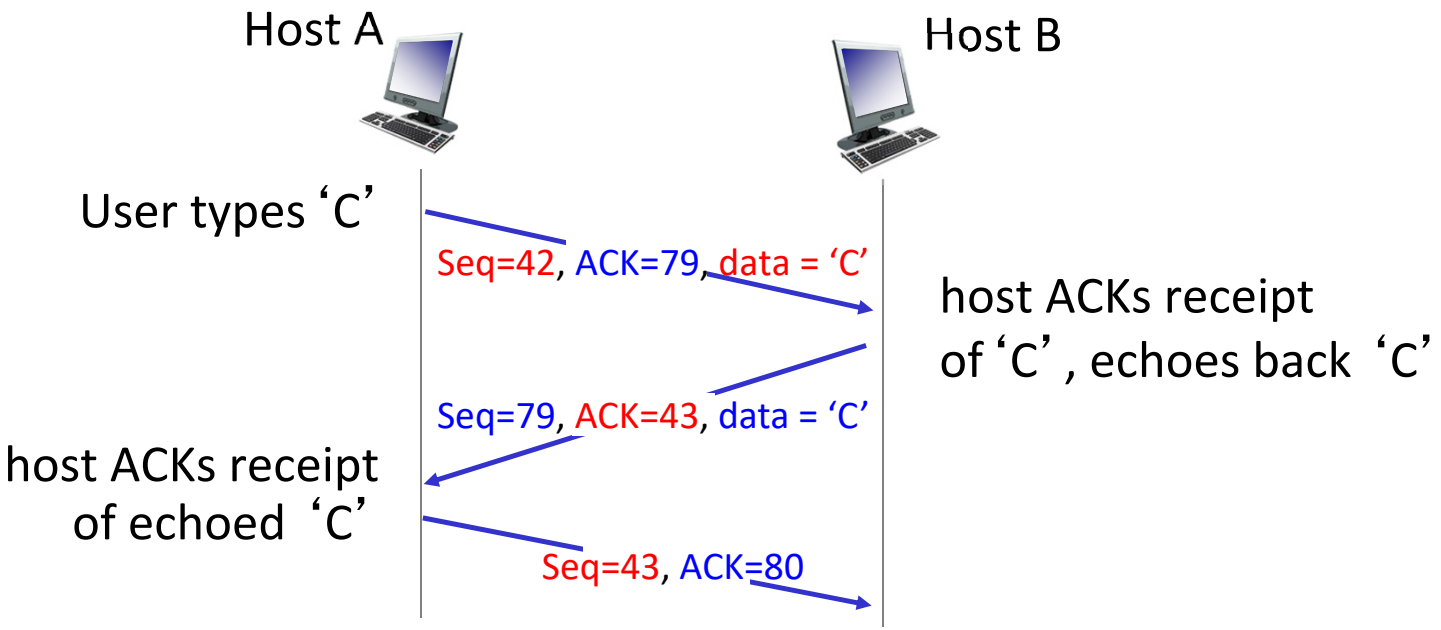
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor



TCP sequence numbers and ACKs



simple telnet scenario

TCP round trip time and 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

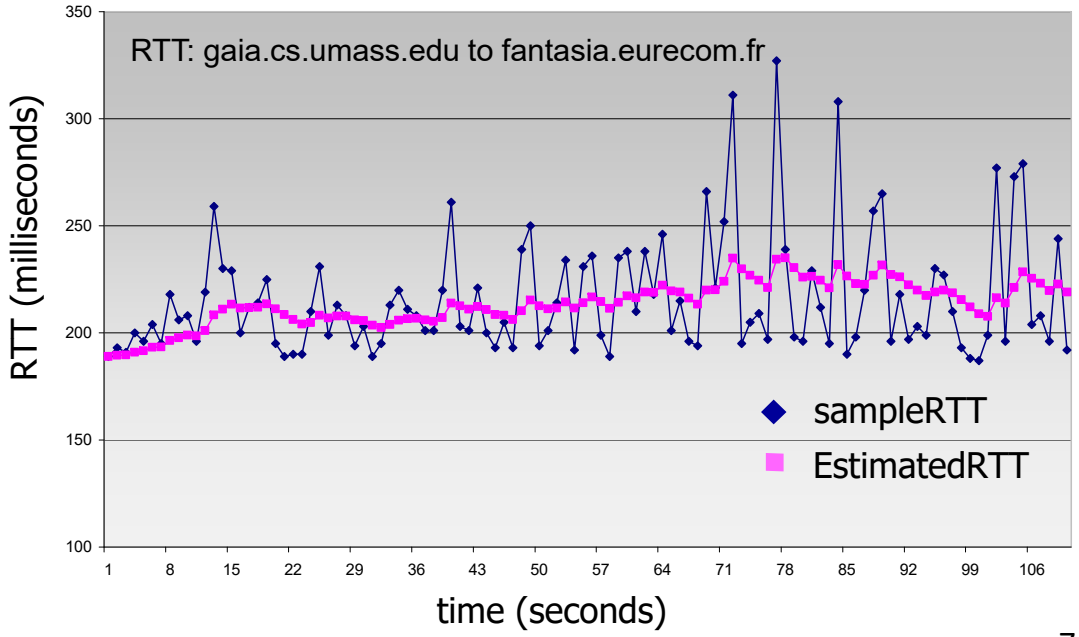
Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary and we 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 (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time and timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT**: want a larger safety margin

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

estimated RTT

"safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

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

(typically, β = 0.25)

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

TCP Sender (simplified)

event: data received from application layer

- 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`**

event: timeout

- retransmit the (only one) segment that caused timeout
- restart timer

event: ACK received

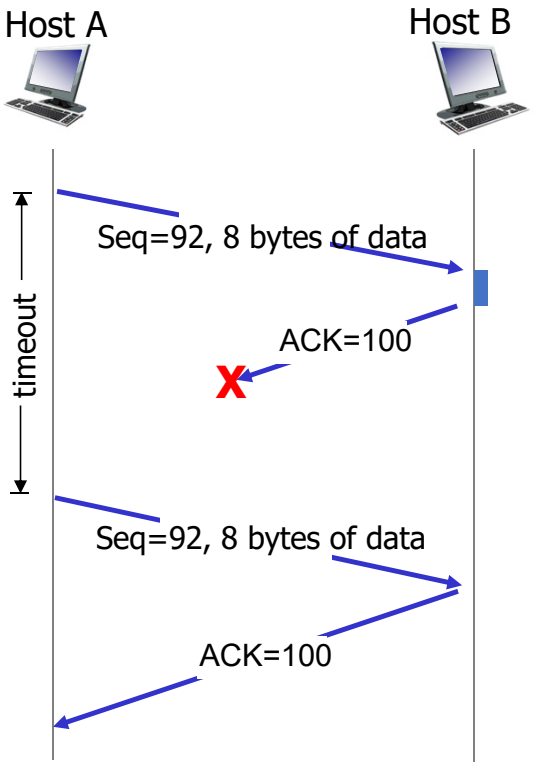
- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed (window moves forward)
  - start timer if there are still unACKed segments

TCP Receiver: ACK generation [RFC 5681]

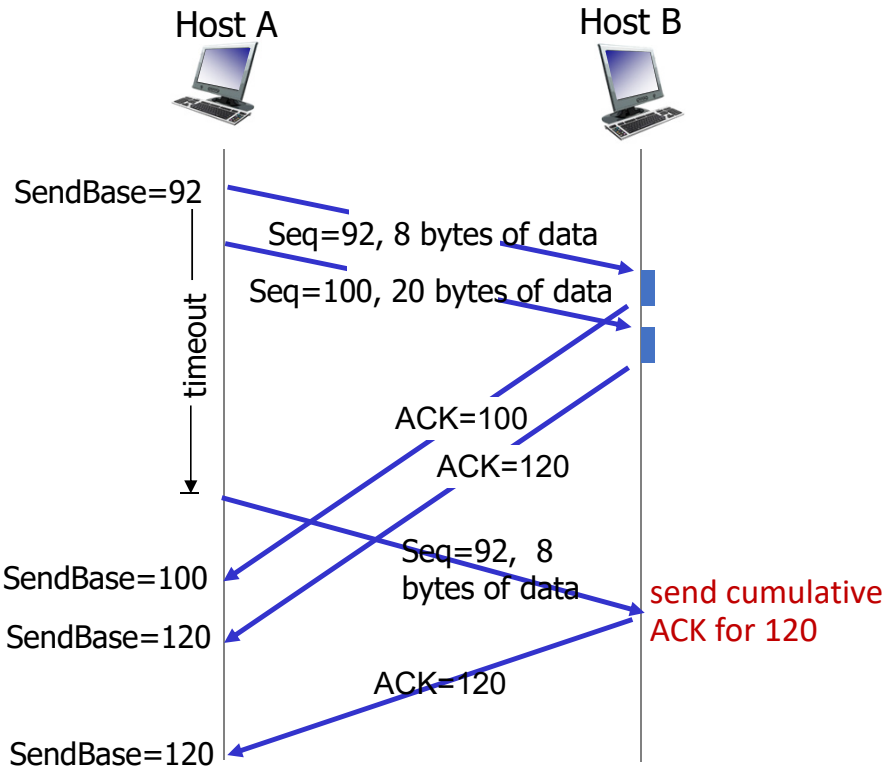
Event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately 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

TCP: retransmission scenarios

lost ACK scenario

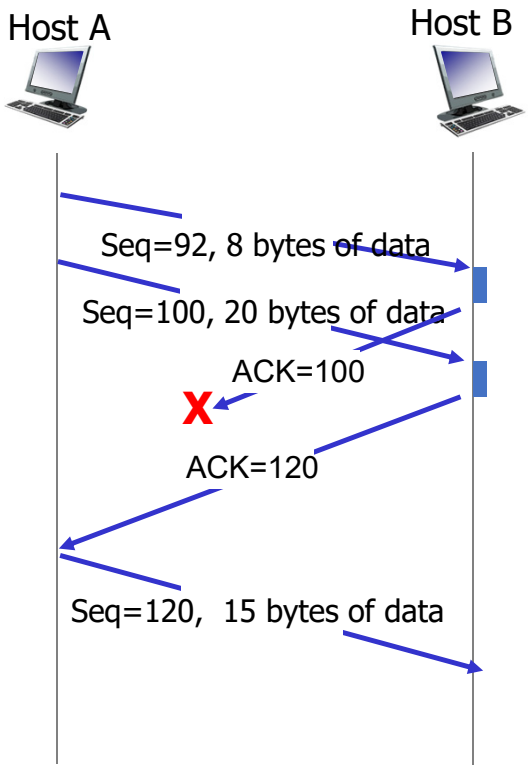


premature timeout



TCP: retransmission scenarios

cumulative ACK covers
for earlier lost ACK



TCP fast retransmit

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!