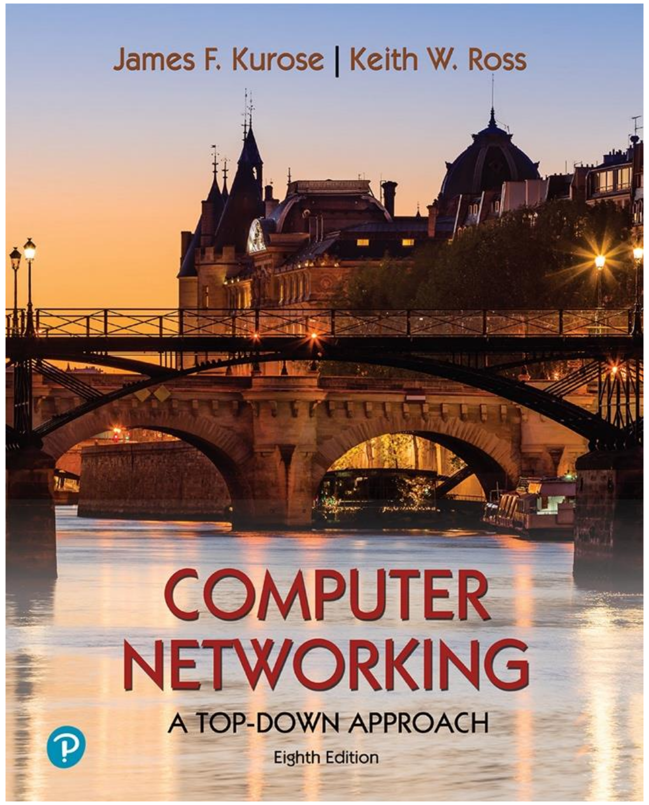
Chapter 3
Transport Layer

Yaxiong Xie

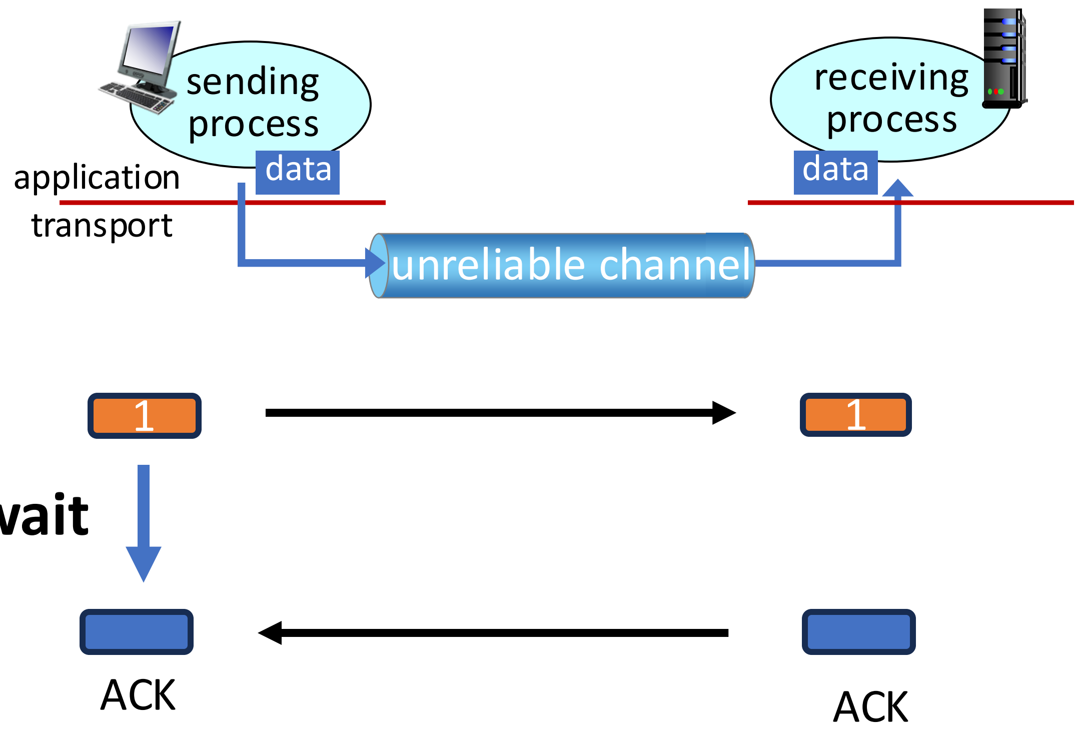
Department of Computer Science and Engineering
University at Buffalo, SUNY

Adapted from the slides of the book's authors



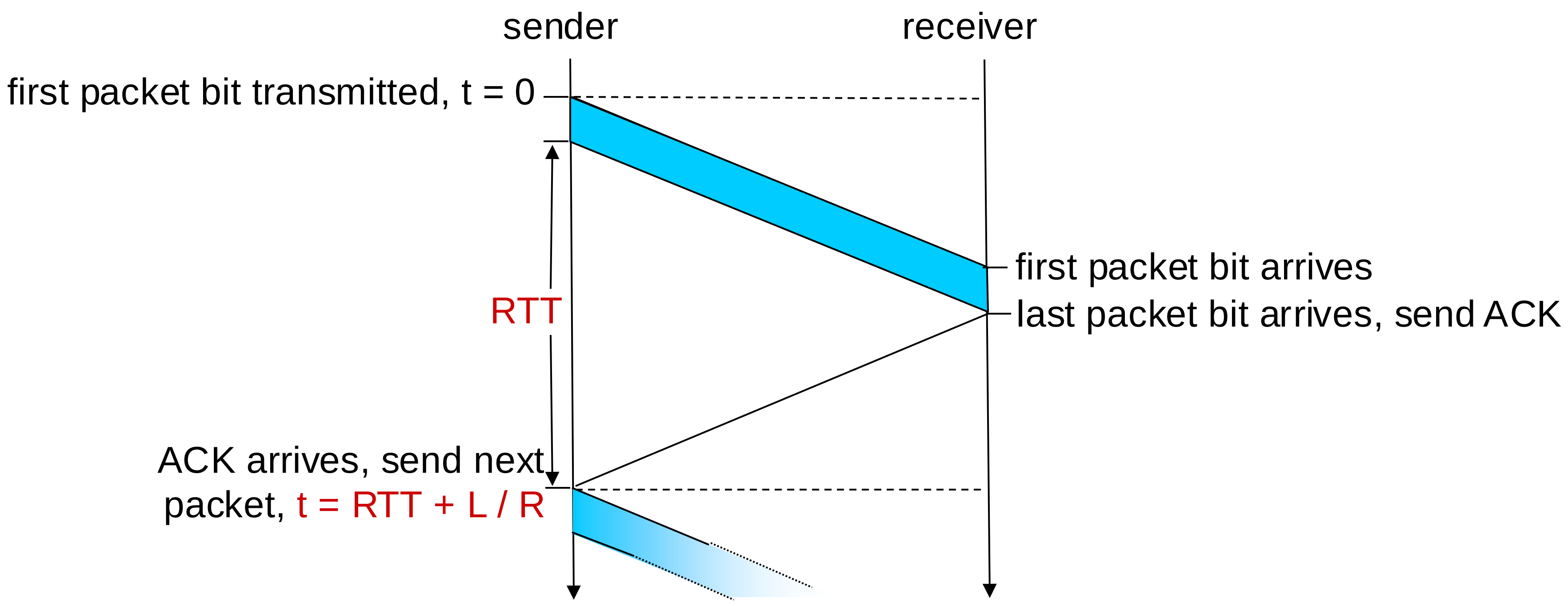
Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Stop-and-wait operation



Stop-and-wait

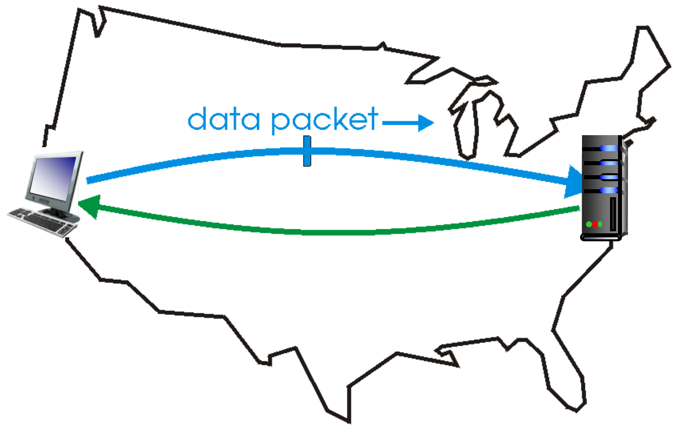
Stop-and-wait operation


sender
receiver
first packet bit transmitted, t = 0
first packet bit arrives
last packet bit arrives, send ACK
RTT
ACK arrives, send next packet, t = RTT + L / R

Pipelined protocols operation

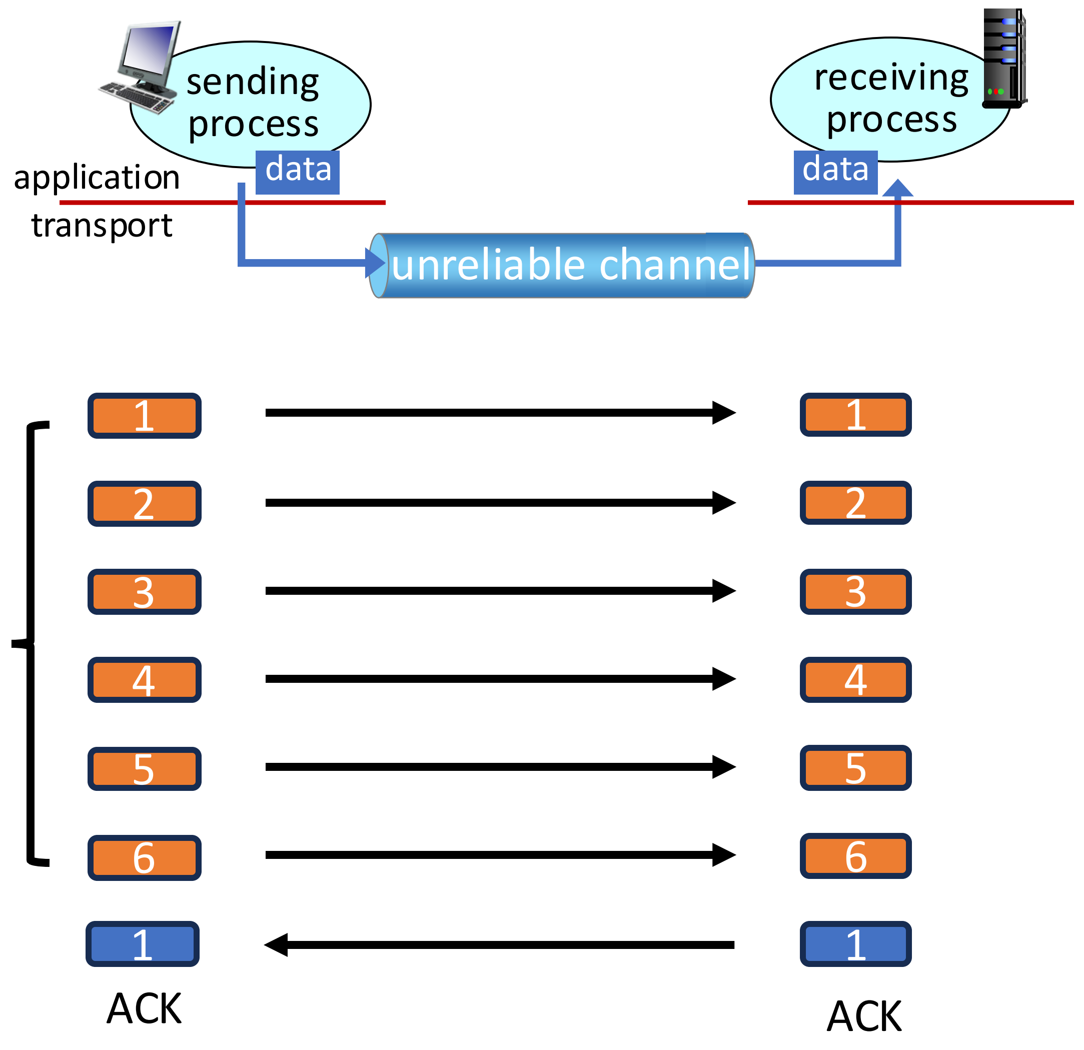
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



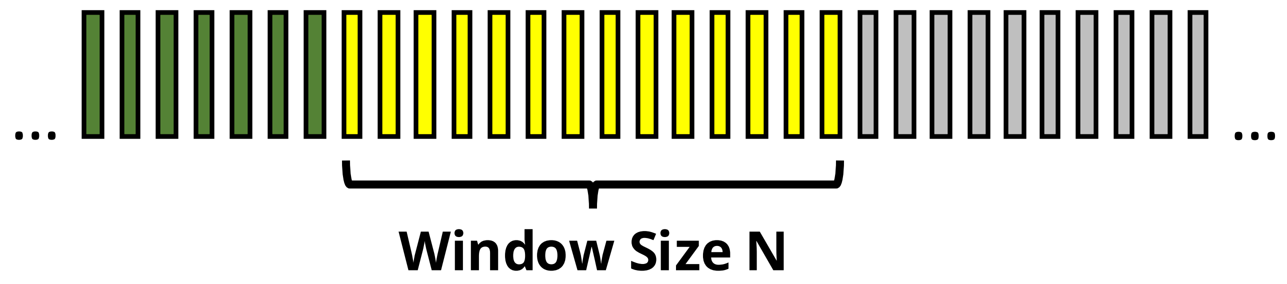
(a) a stop-and-wait protocol in operation

Pipelining



How many packets shall we sent before receiving the ACK?

Sliding Window

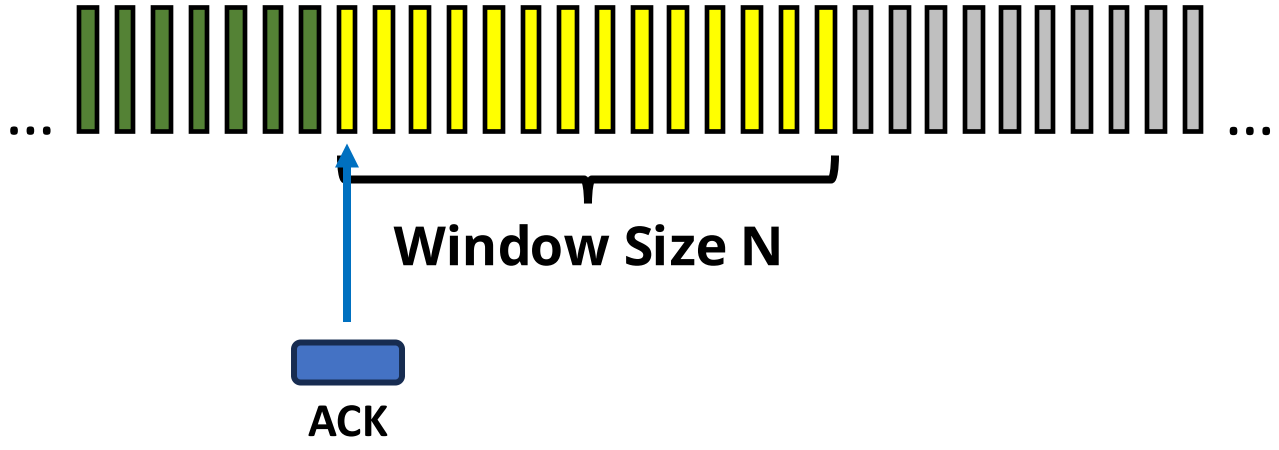


🟩 sent and ACKed

🟨 sent, not yet ACKed

⬜ Not yet sent

Sliding Window

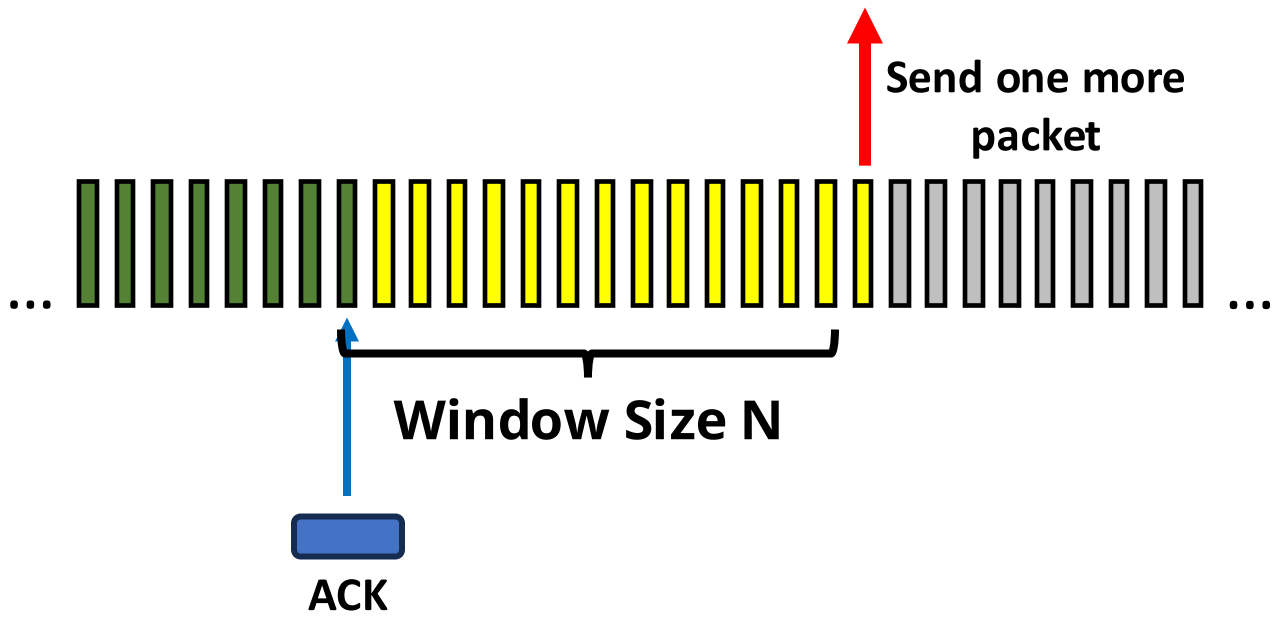


🟩 sent and ACKed

▯ Not yet sent

🟨 sent, not yet ACKed

Sliding Window

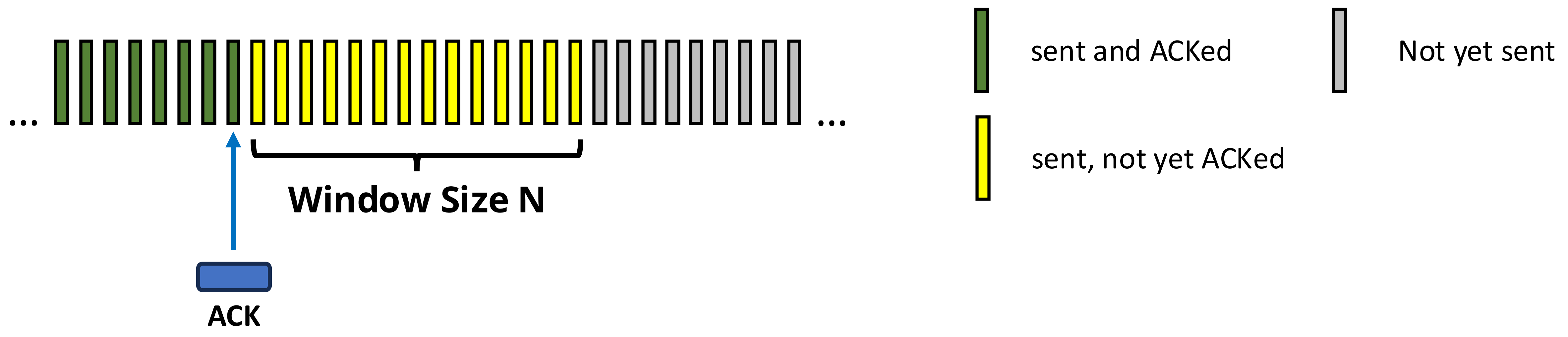


🟩 sent and ACKed

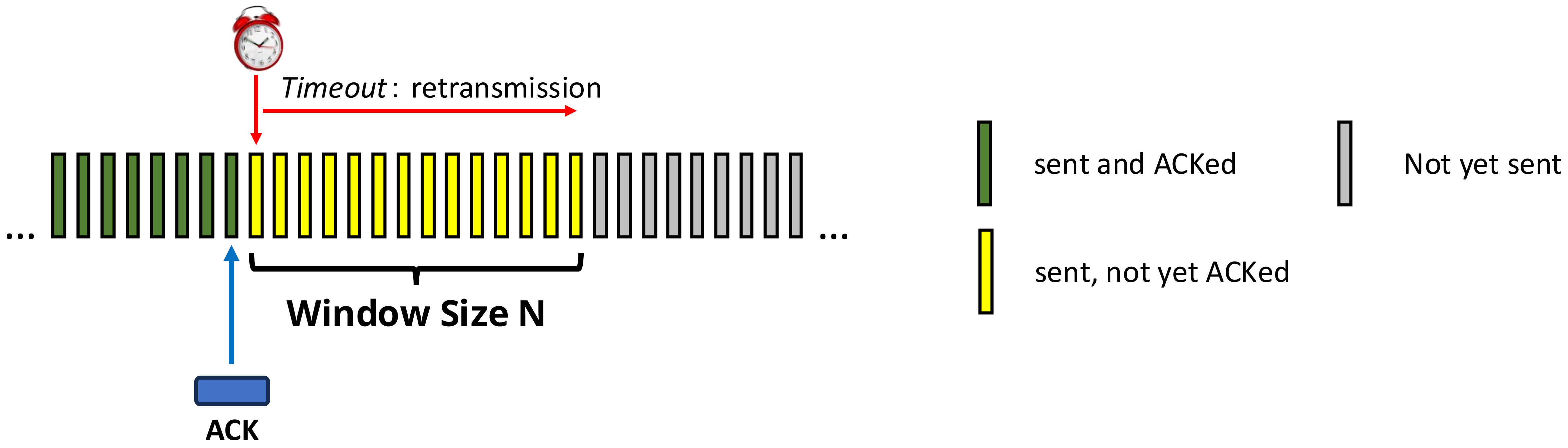
⬜ Not yet sent

🟨 sent, not yet ACKed

Sliding Window

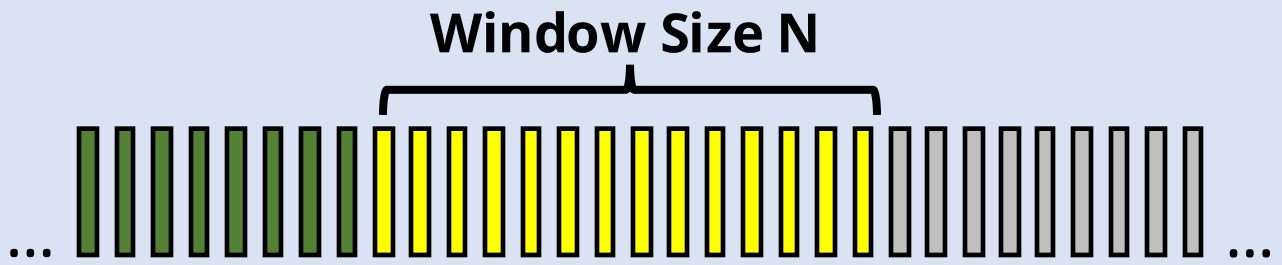


Sliding Window



- Timer for oldest in-flight packet
- *Timeout(n):* retransmit packet n and all higher seq # packets in window

Sliding Window + cumulative ACK



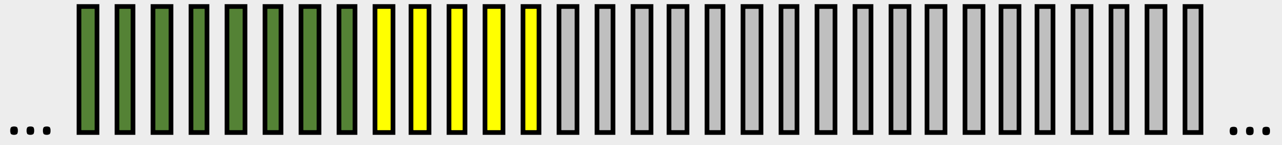
🟩 sent and ACKed

▯ Not yet sent

🟨 sent, not yet ACKed

Sender

Receiver

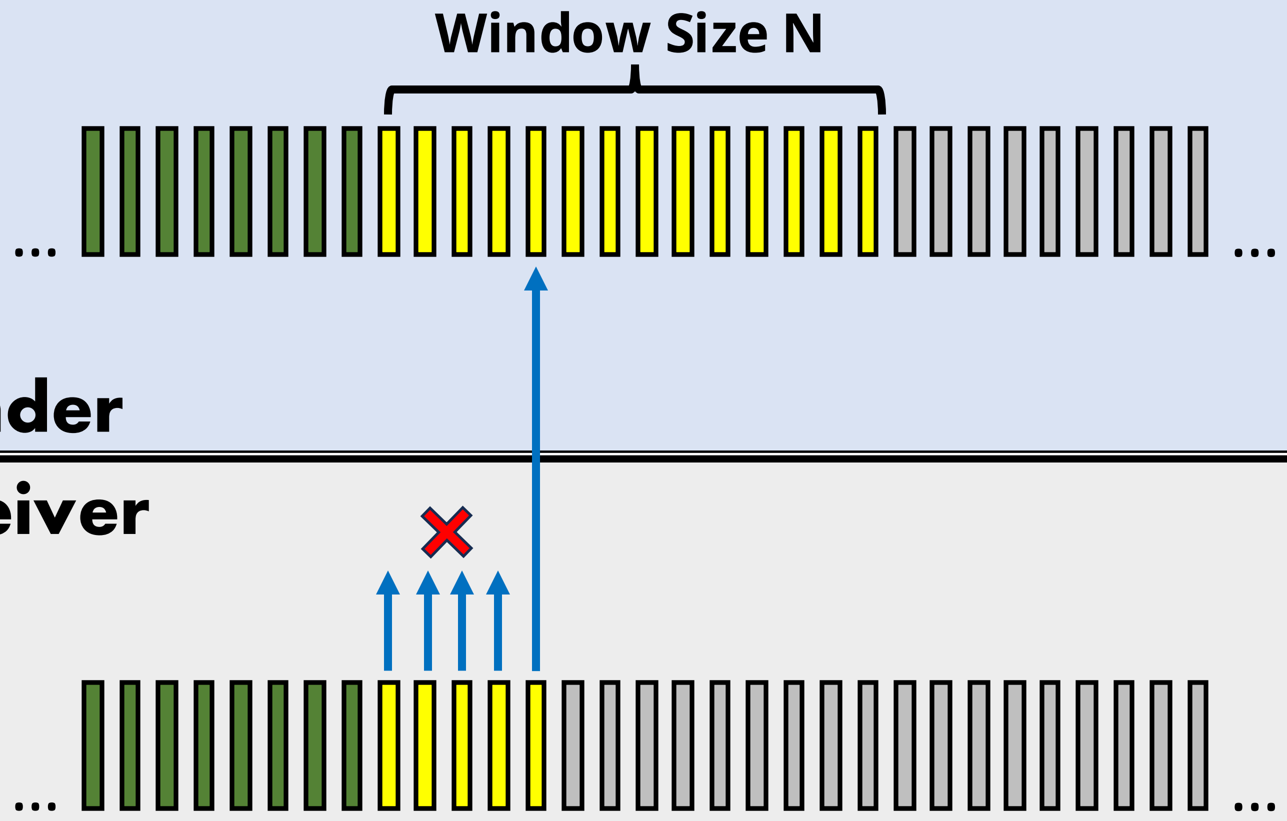


🟩 received and ACKed

▯ Not yet received

🟨 received, not yet ACKed

Sliding Window + cumulative ACK



🟩 sent and ACKed

▯ Not yet sent

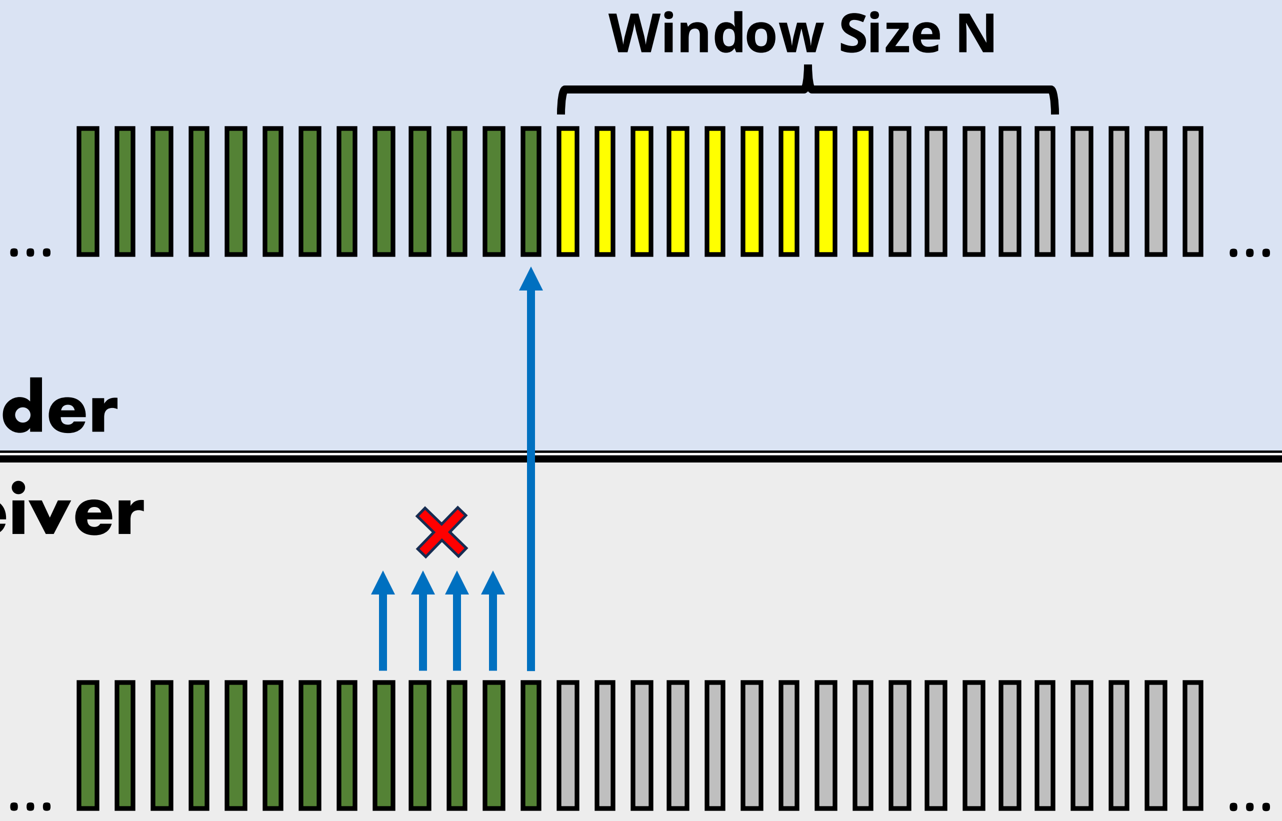
🟨 sent, not yet ACKed

🟩 received and ACKed

▯ Not yet received

🟨 received, not yet ACKed

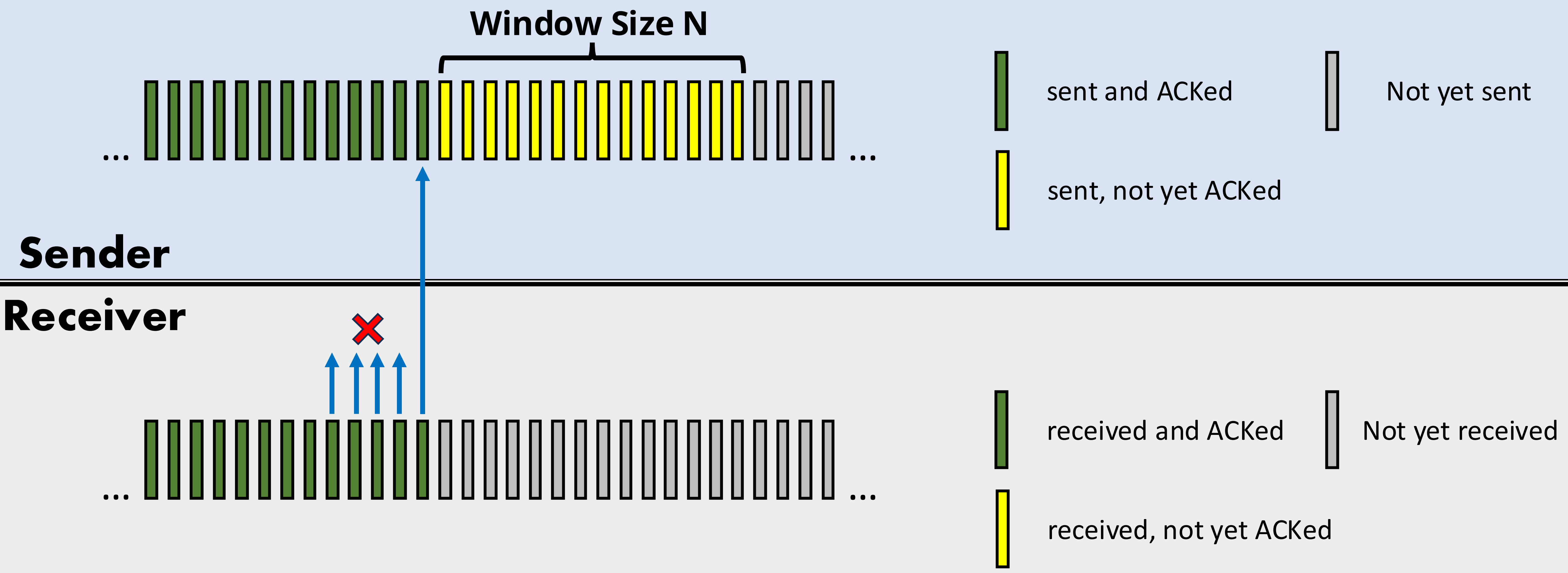
Sliding Window + cumulative ACK



🟩	sent and ACKed	▯	Not yet sent
🟨	sent, not yet ACKed		

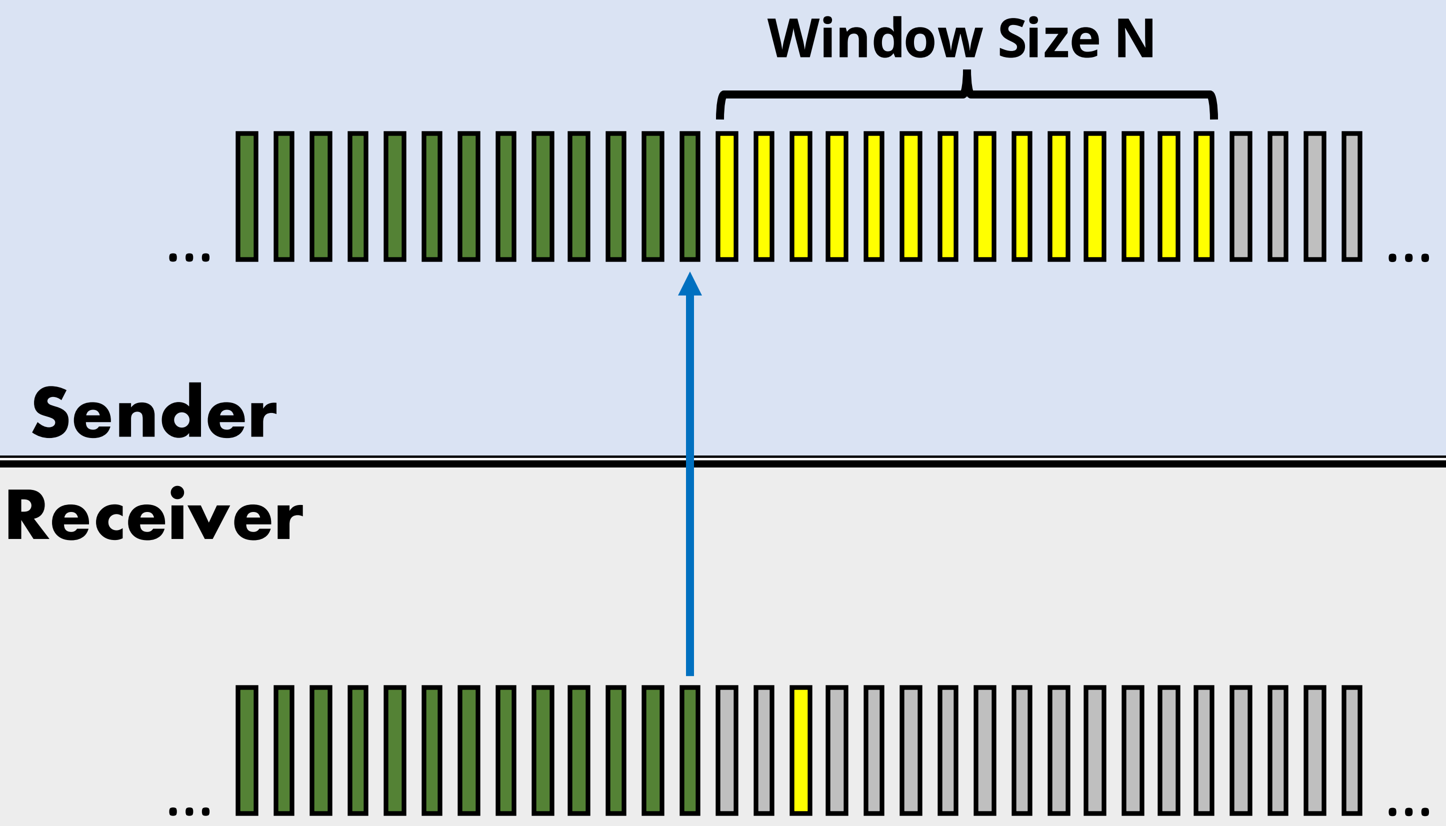
🟩	received and ACKed	▯	Not yet received
🟨	received, not yet ACKed		

Sliding Window + cumulative ACK



- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*

Sliding Window + cumulative ACK: out-of-order



🟩 sent and ACKed

▯ Not yet sent

🟨 sent, not yet ACKed

🟩 received and ACKed

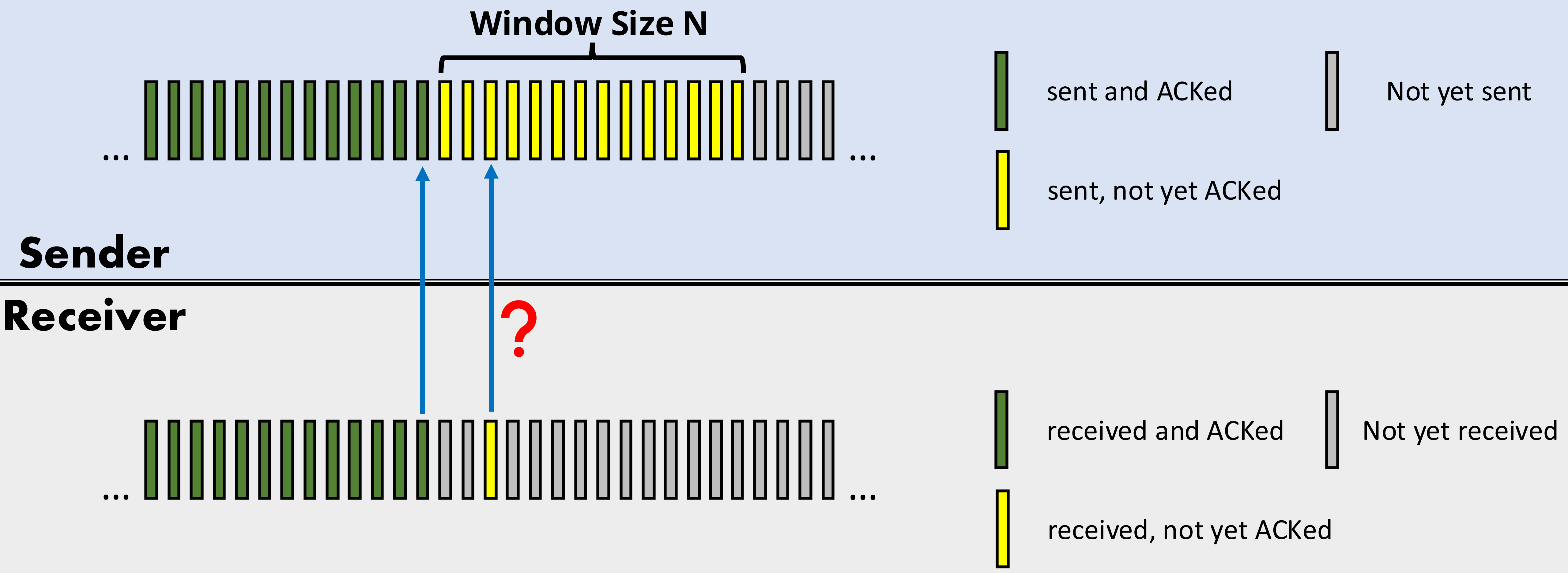
▯ Not yet received

🟨 received, not yet ACKed

Two Question:

1. do we need to send ACK when we receive an out-of-order packet? If yes, what kind of ACK?

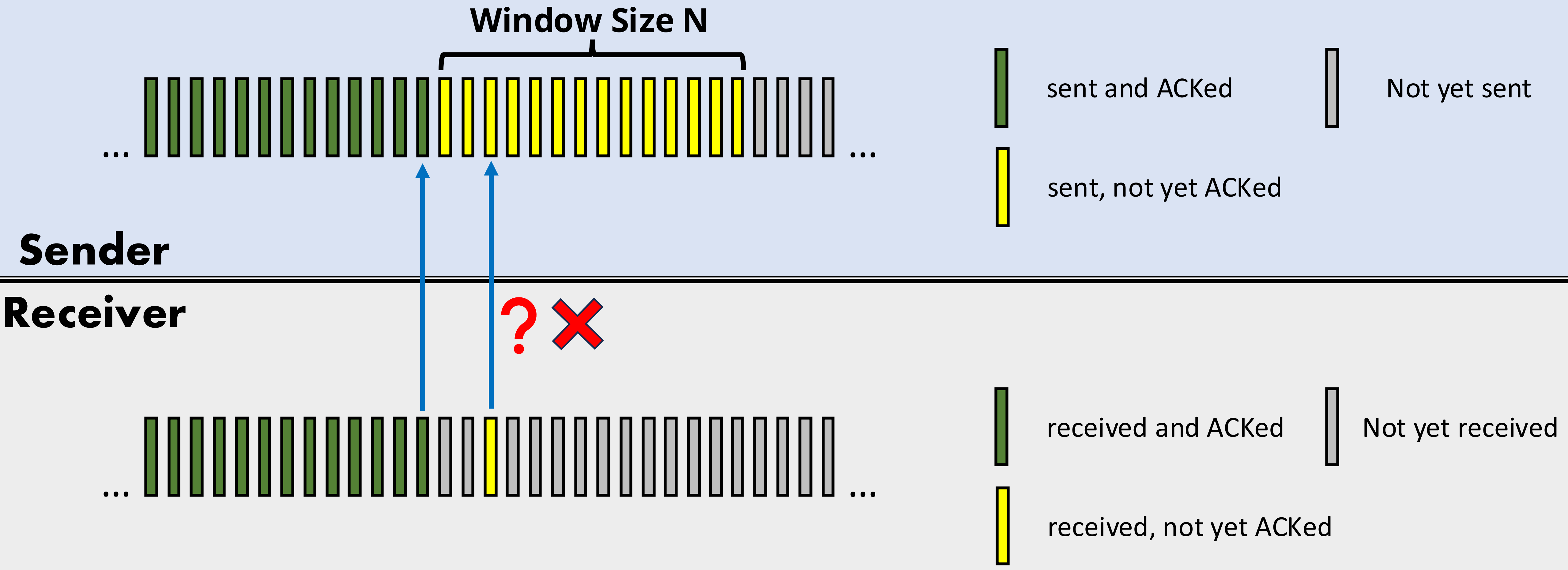
Sliding Window + cumulative ACK: out-of-order



Two Question:

1. do we need to send ACK when we receive an out-of-order packet? If yes, what kind of ACK?
2. how shall we deal with the out-of-order packets we received?

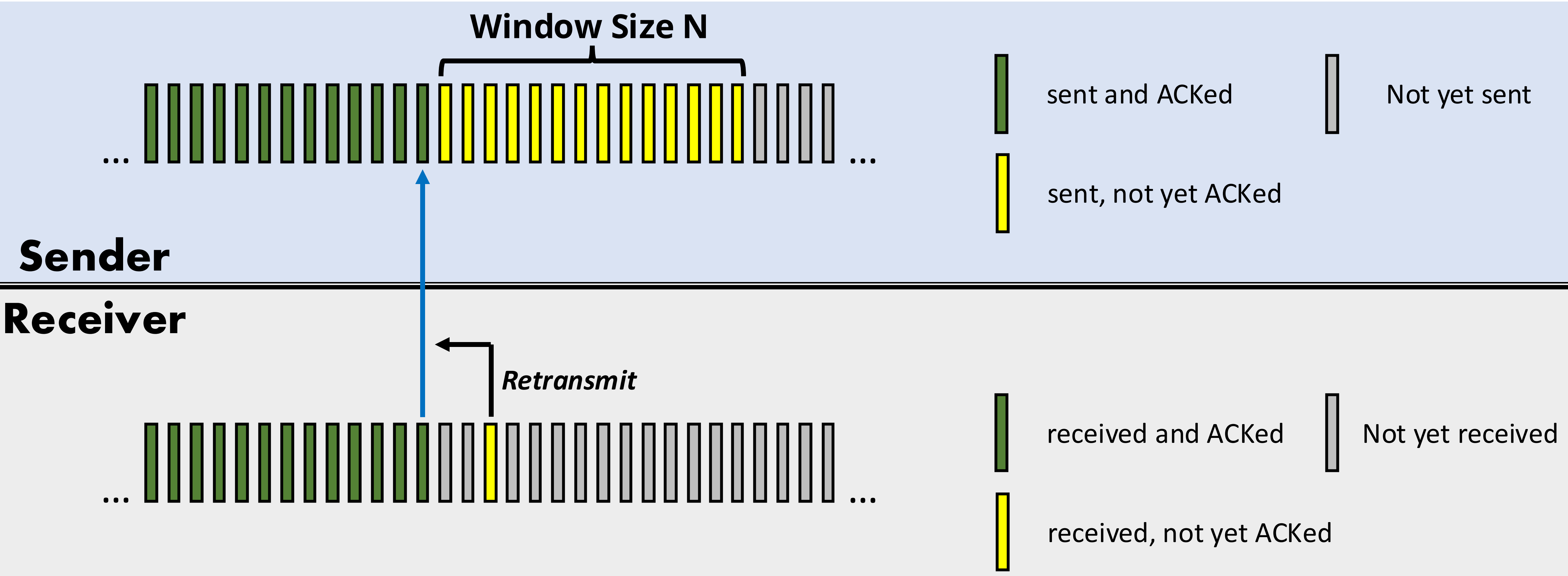
Sliding Window + cumulative ACK: out-of-order



Two Question:

1. do we need to send ACK when we receive an out-of-order packet? If yes, what kind of ACK?
2. how shall we deal with the out-of-order packets we received?

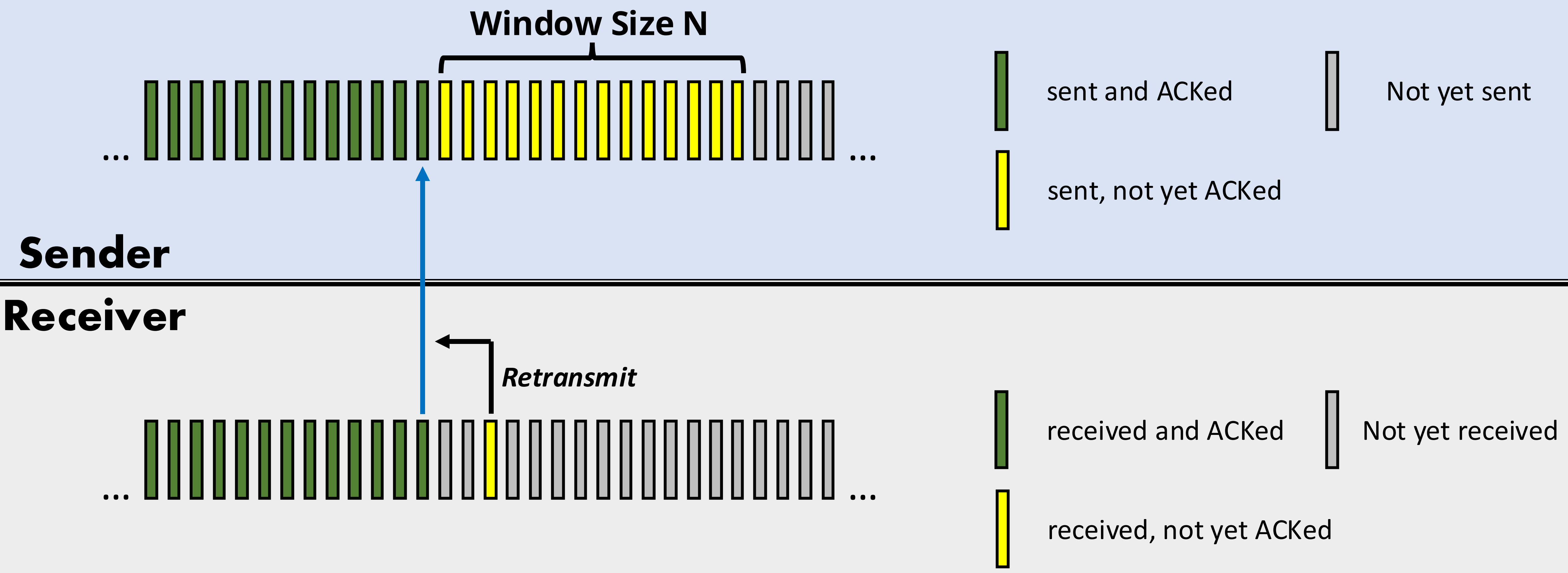
Sliding Window + cumulative ACK: out-of-order



🟩	sent and ACKed	▯	Not yet sent
🟨	sent, not yet ACKed		
🟩	received and ACKed	▯	Not yet received
🟨	received, not yet ACKed		

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs

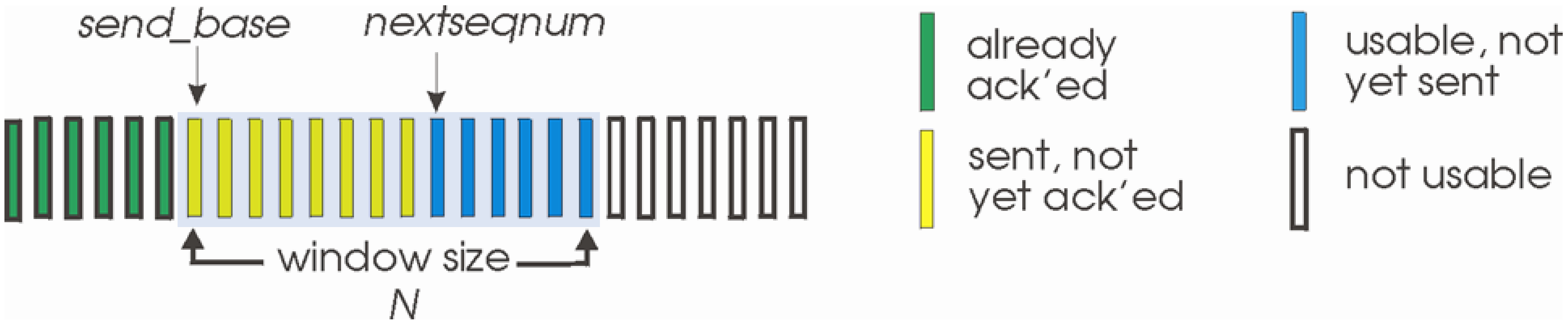
Sliding Window + cumulative ACK: out-of-order



- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

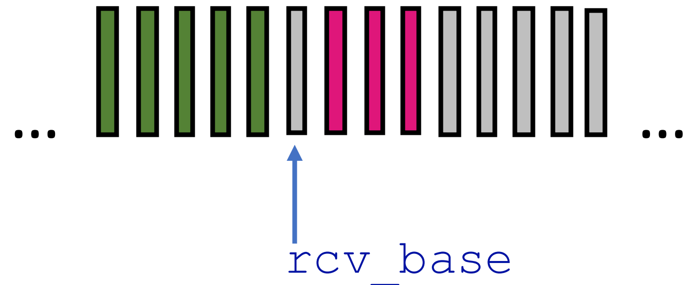


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

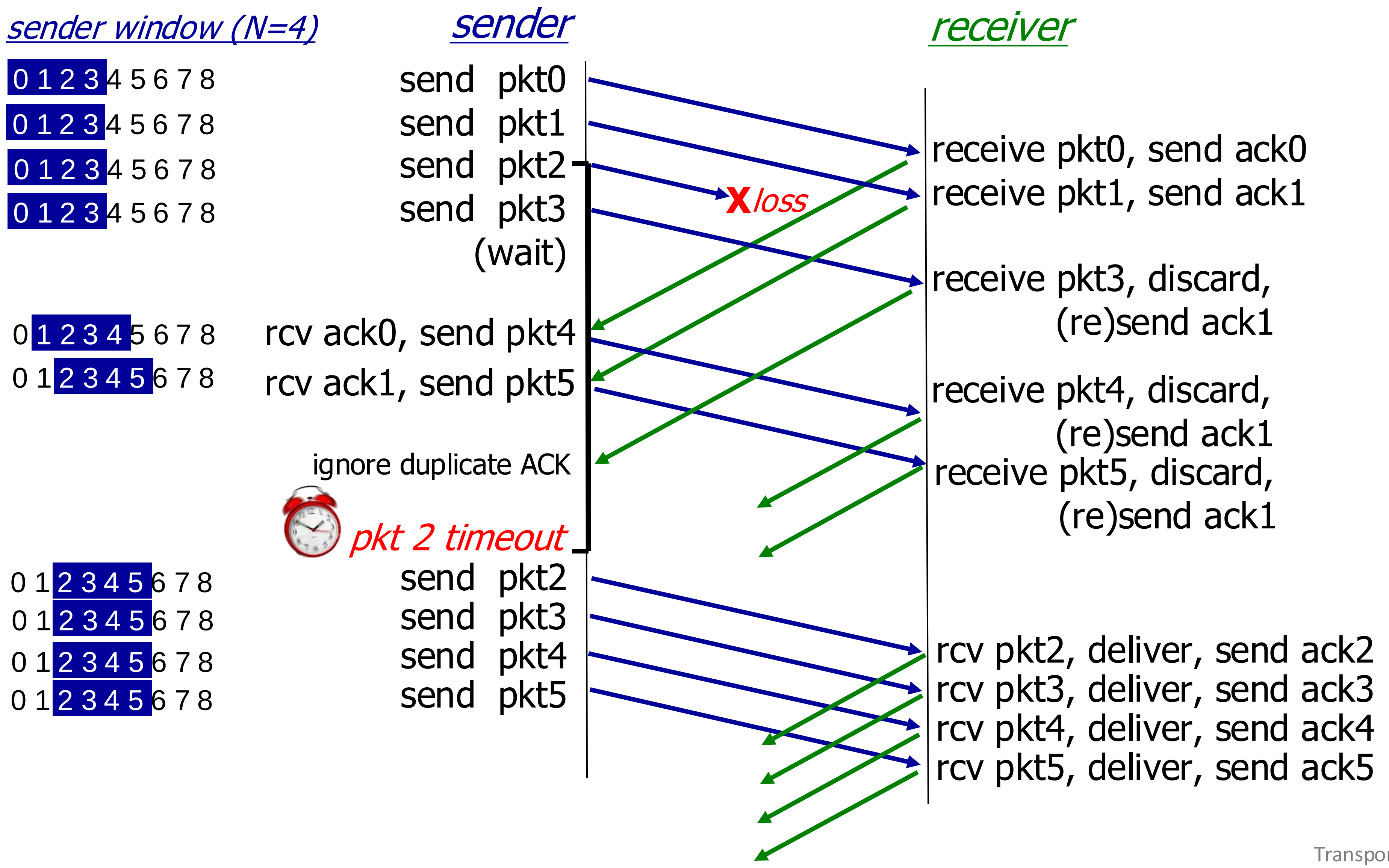


🟩 received and ACKed

🟥 Out-of-order: received but not ACKed

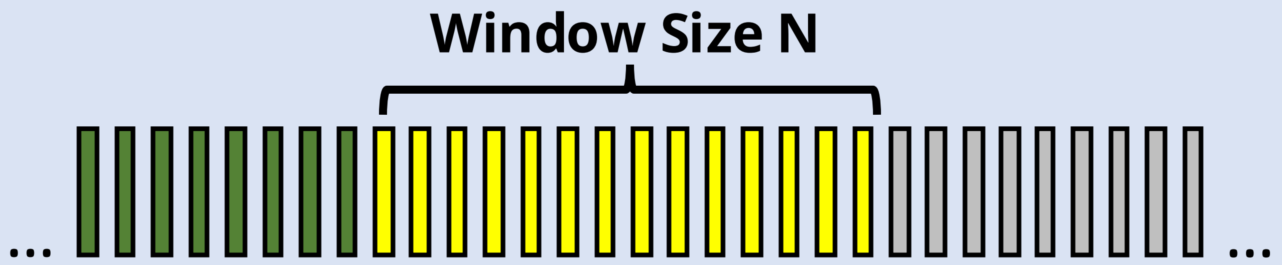
⬜ Not received

Go-Back-N in action



Sliding Window + selective repeat

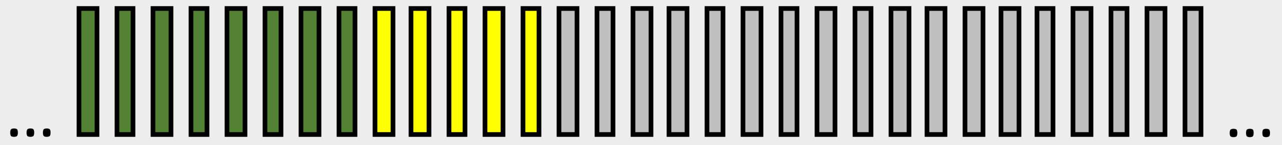
Sliding Window + selective repeat



🟩	sent and ACKed	▯	Not yet sent
🟨	sent, not yet ACKed		

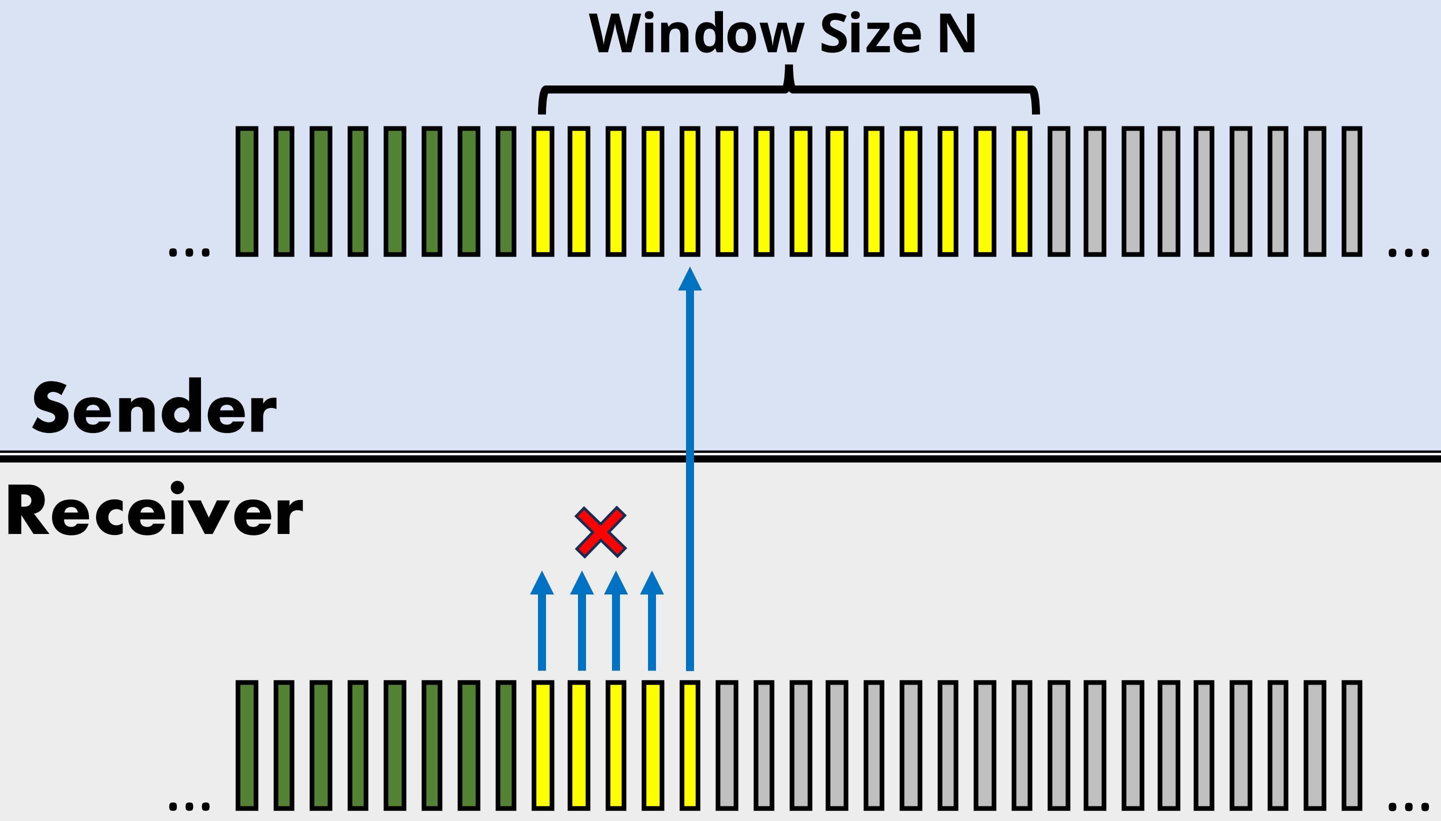
Sender

Receiver



🟩	received and ACKed	▯	Not yet received
🟨	received, not yet ACKed		

Sliding Window + selective repeat



🟩 sent and ACKed

🟨 sent, not yet ACKed

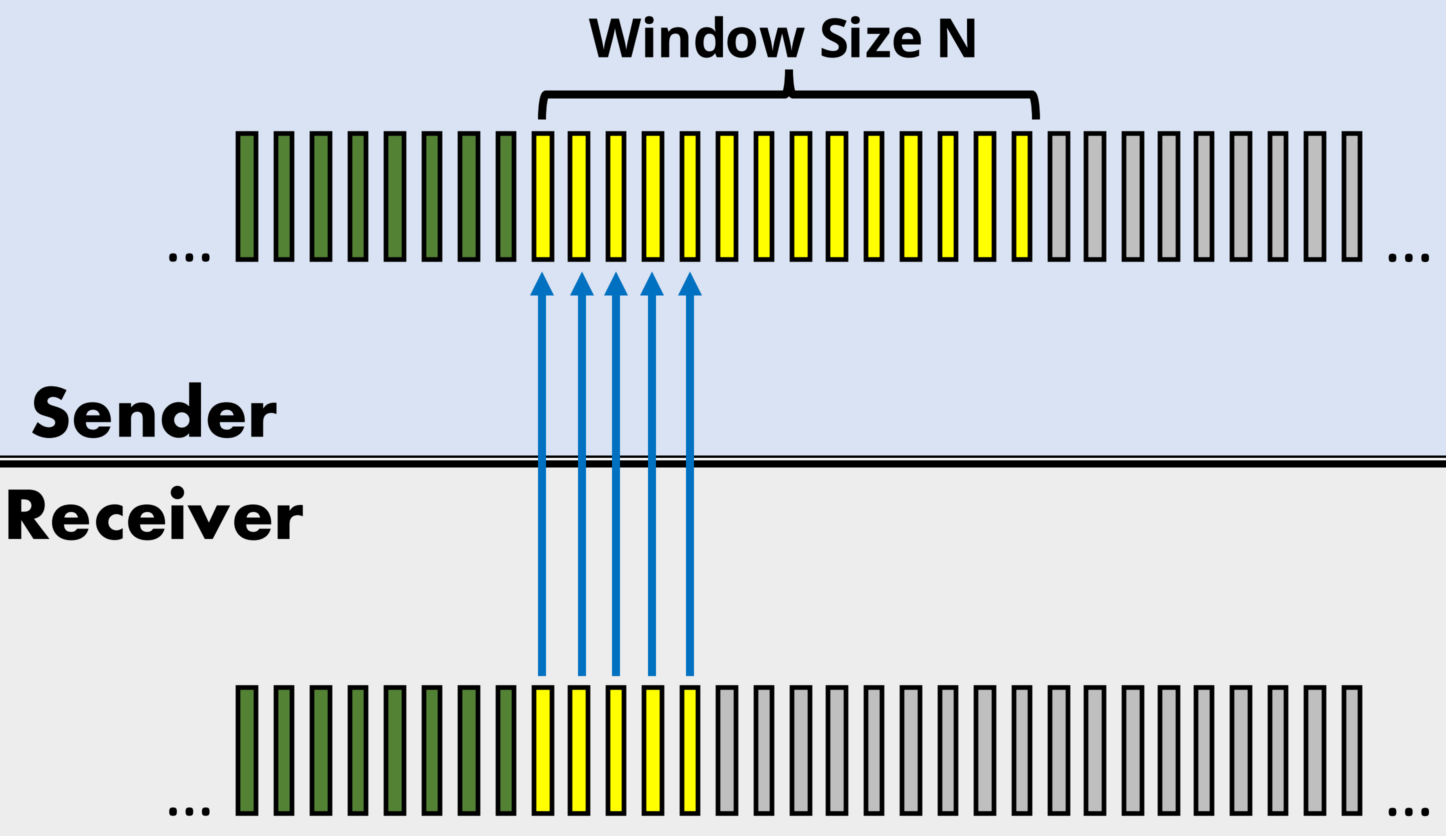
⬜ Not yet sent

🟩 received and ACKed

🟨 received, not yet ACKed

⬜ Not yet received

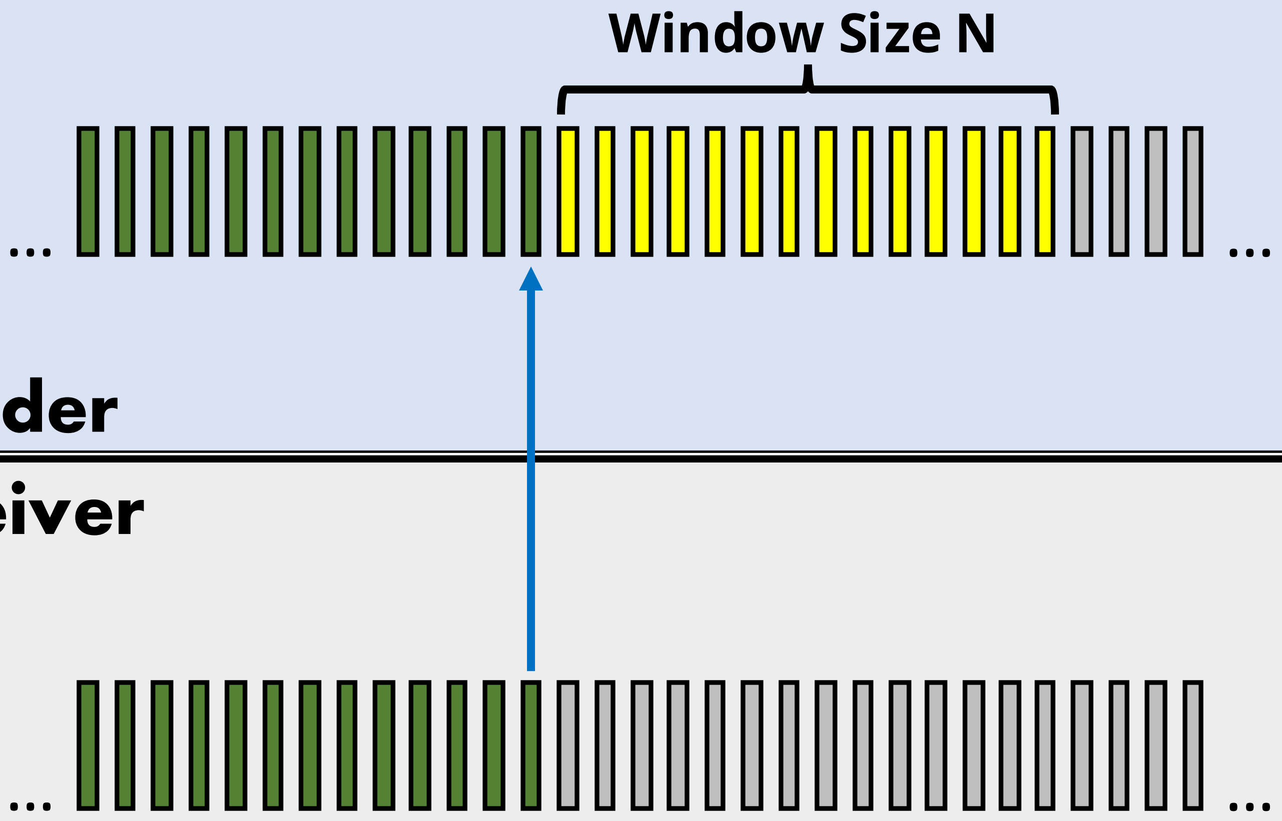
Sliding Window + selective repeat



🟩	sent and ACKed	▯	Not yet sent
🟨	sent, not yet ACKed		

🟩	received and ACKed	▯	Not yet received
🟨	received, not yet ACKed		

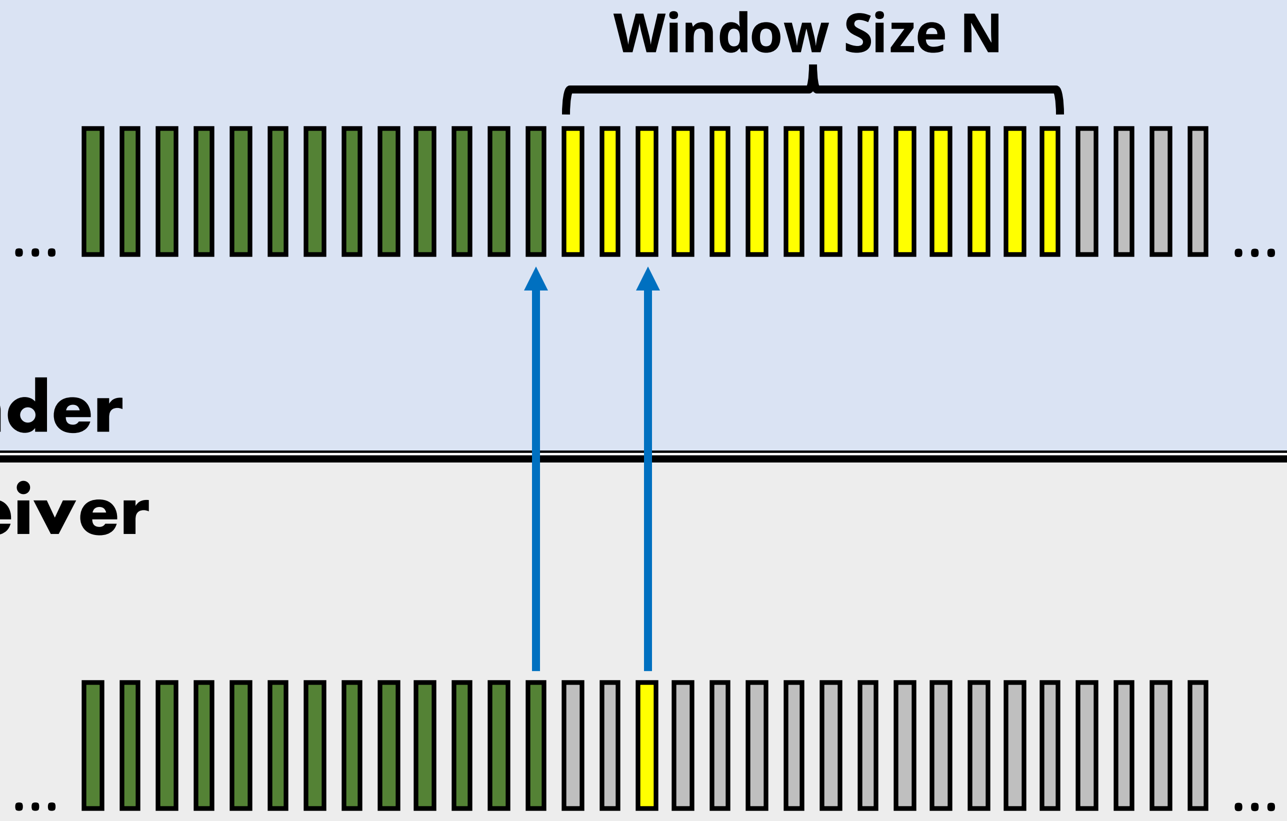
Sliding Window + selective repeat: out-of-order



🟩	sent and ACKed	▯	Not yet sent
🟨	sent, not yet ACKed		

🟩	received and ACKed	▯	Not yet received
🟨	received, not yet ACKed		

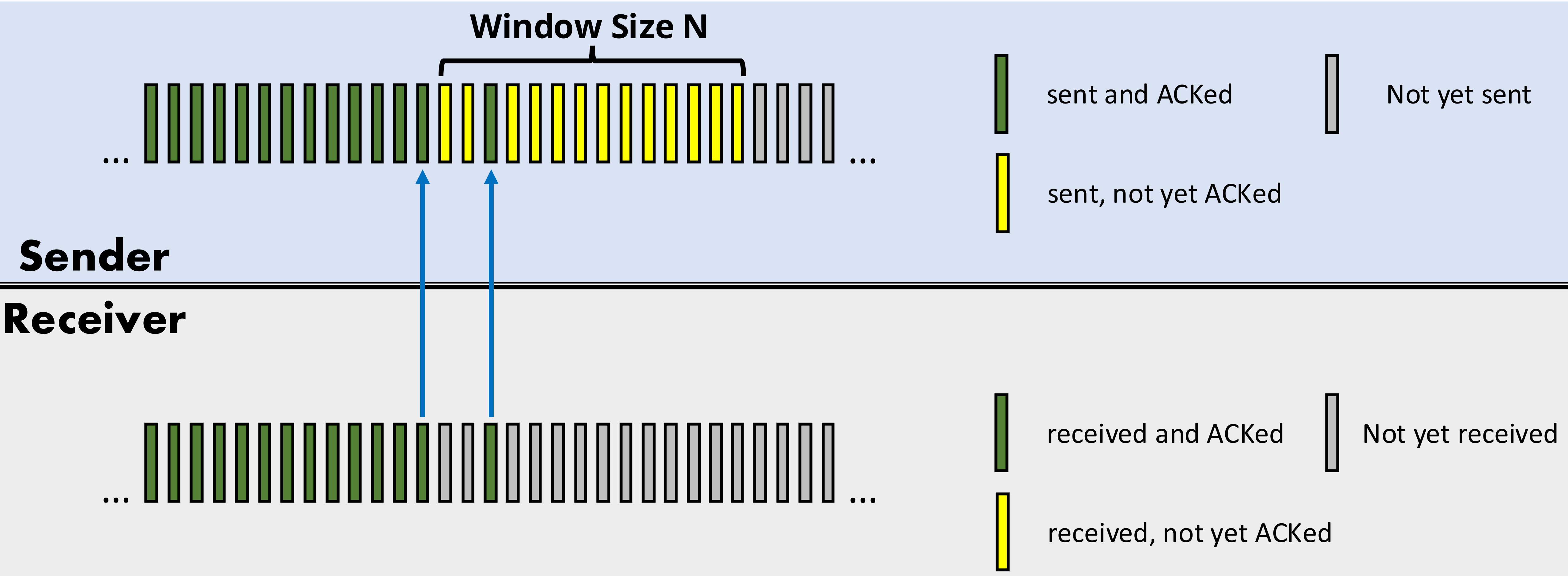
Sliding Window + selective repeat: out-of-order



🟩	sent and ACKed	⬜	Not yet sent
🟨	sent, not yet ACKed		

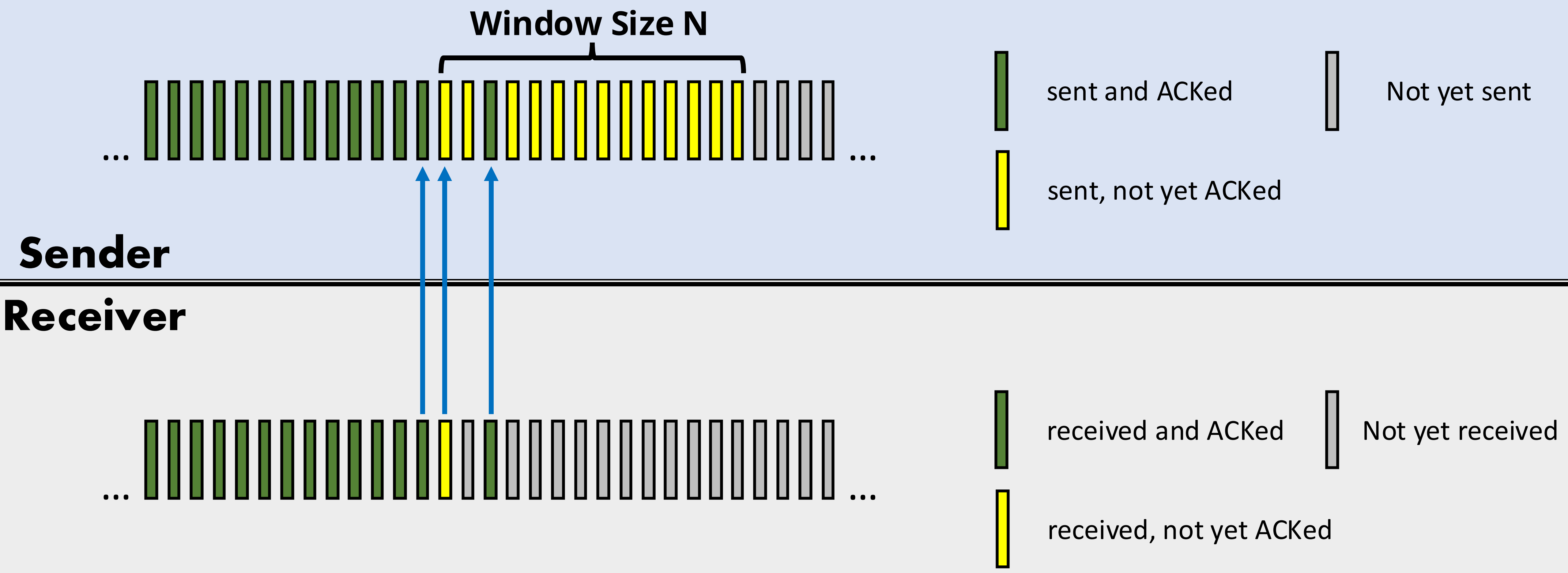
🟩	received and ACKed	⬜	Not yet received
🟨	received, not yet ACKed		

Sliding Window + selective repeat: out-of-order



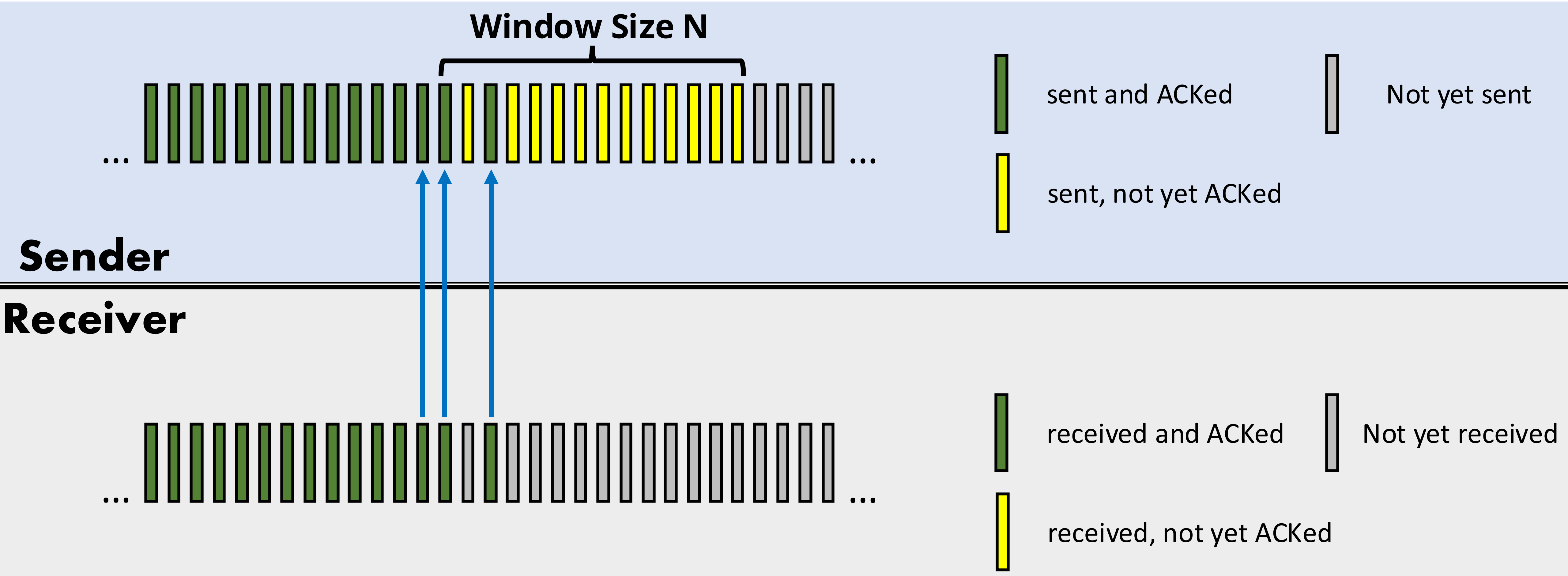
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat: out-of-order



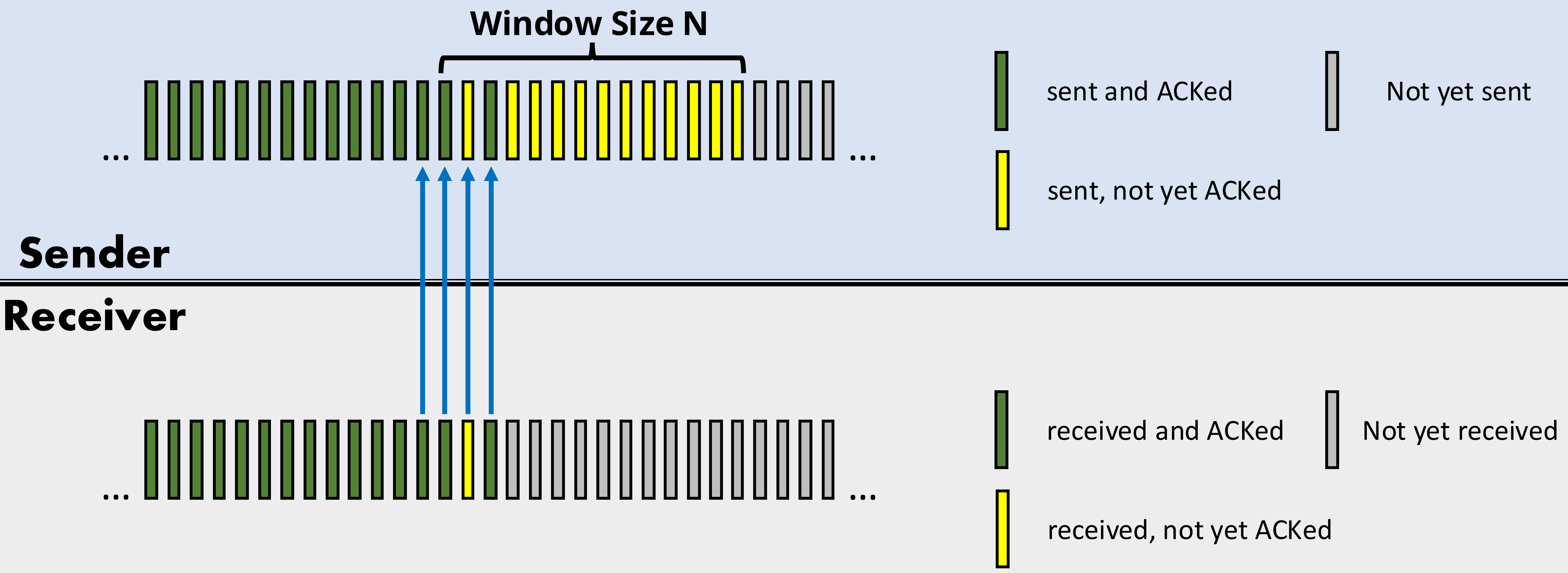
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat: out-of-order



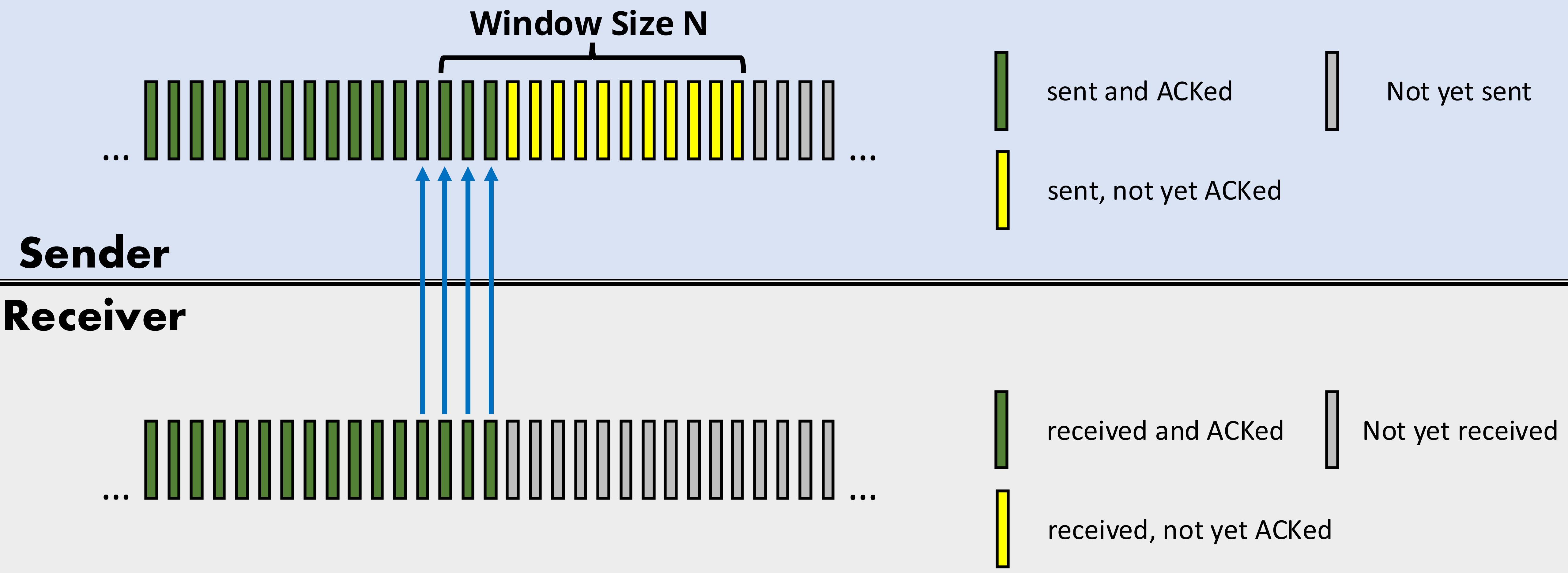
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat: out-of-order



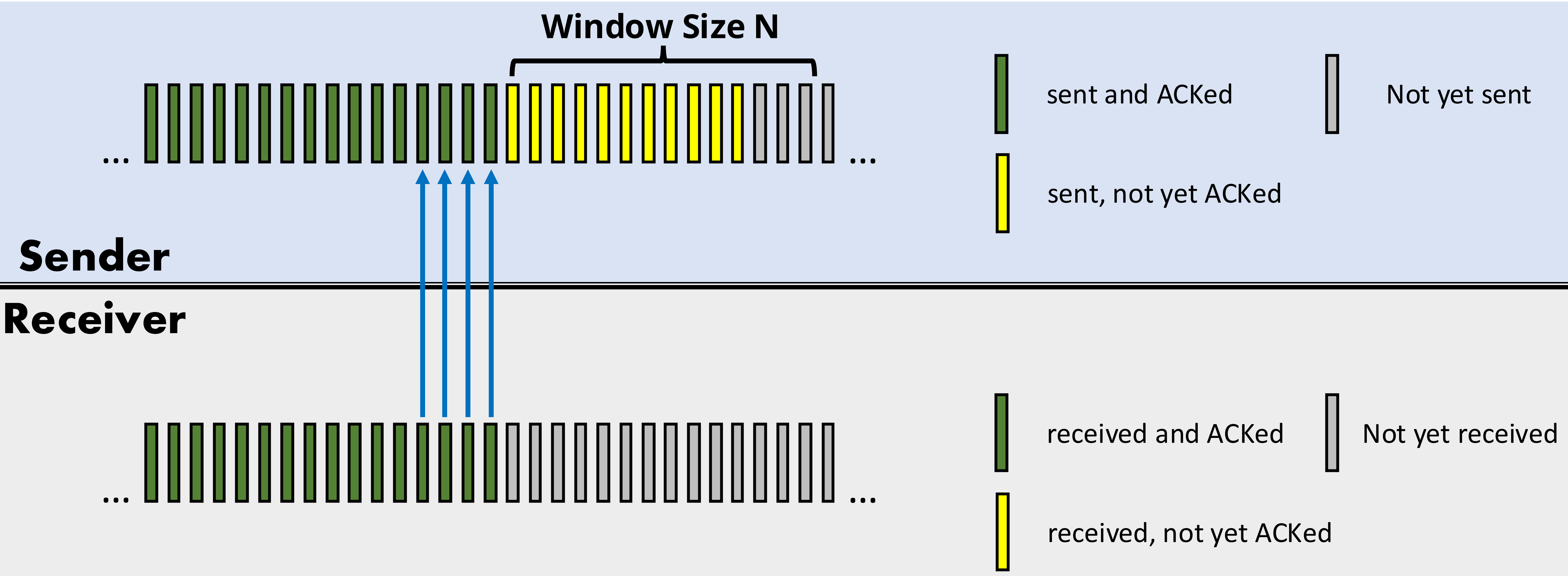
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat: out-of-order



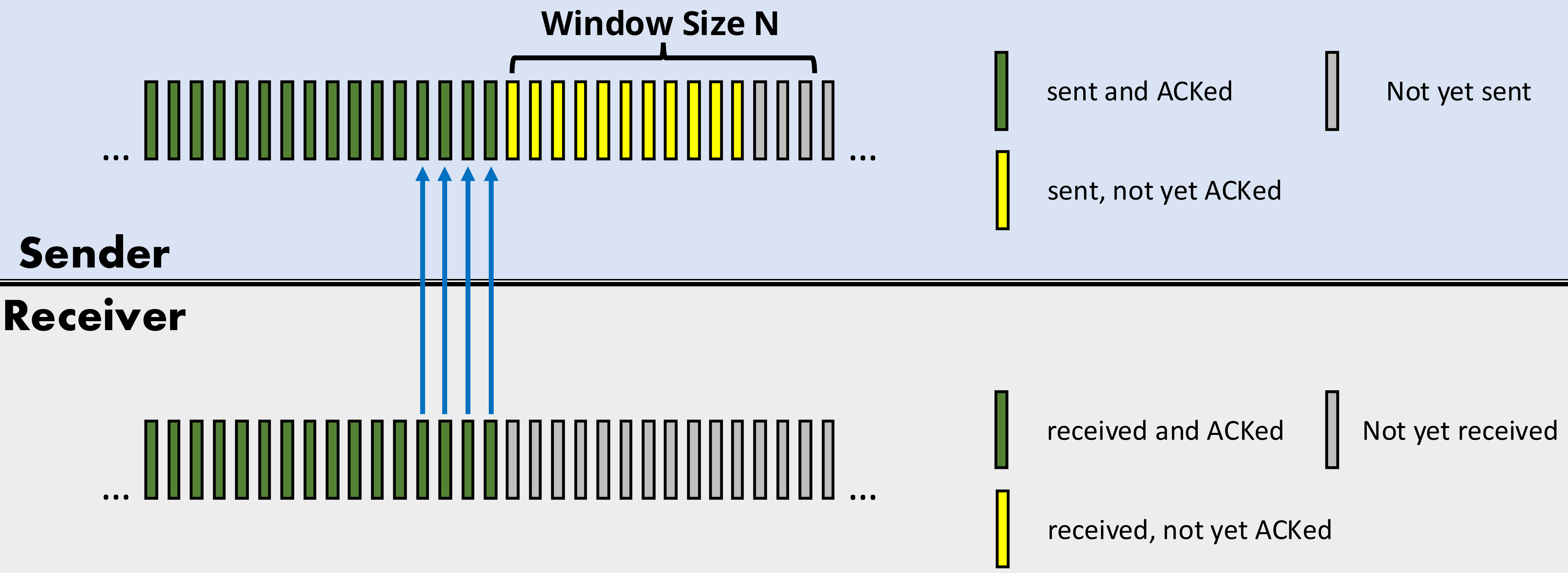
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat: out-of-order



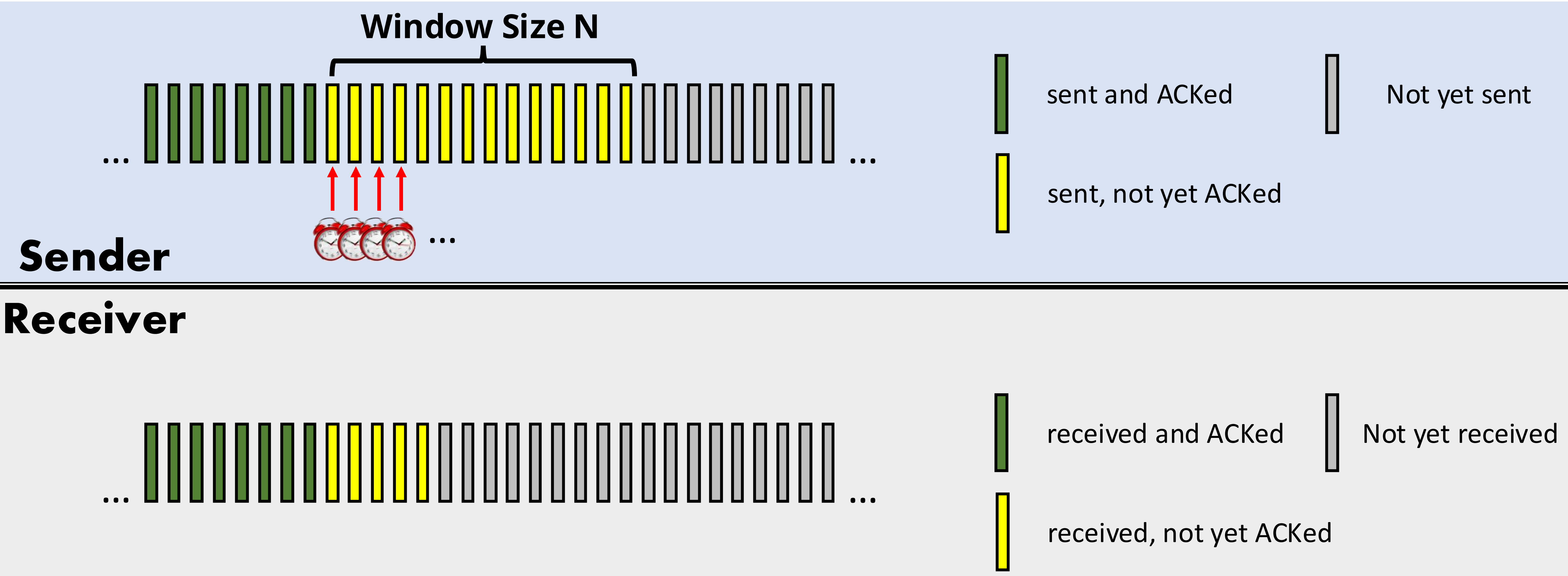
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat: out-of-order



- *receiver individually ACKs* all correctly received packets
    - buffers packets, as needed, for in-order delivery to upper layer

Sliding Window + selective repeat

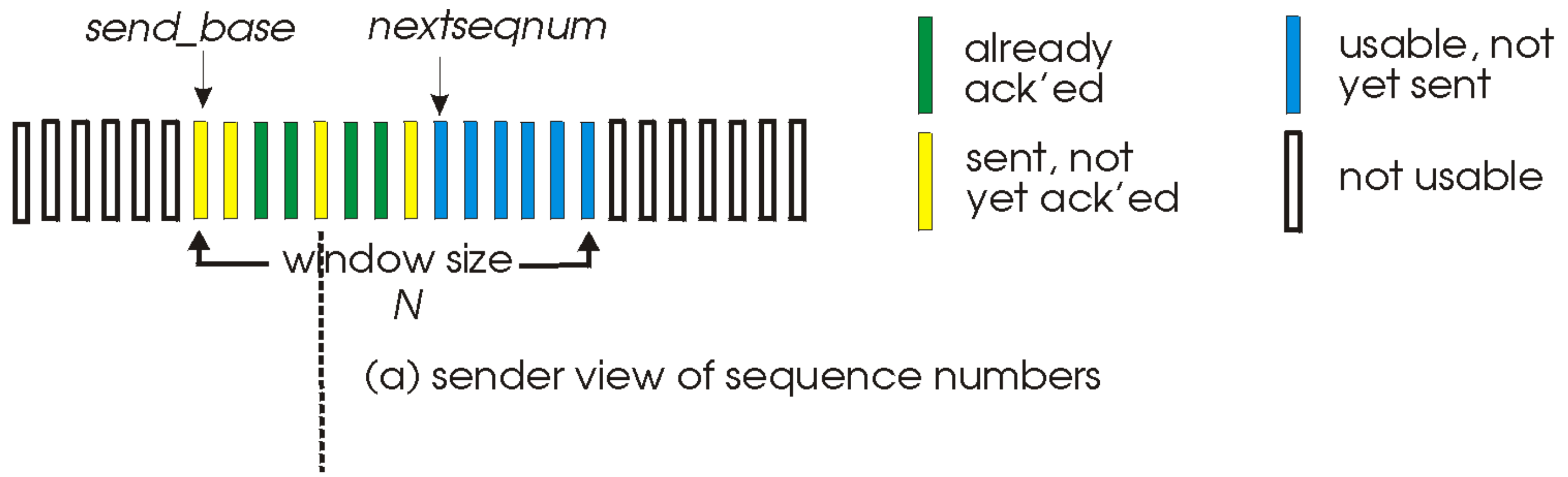


- Maintains (conceptually) a timer for each unACKed pkt
  - timeout: retransmits single unACKed packet associated with timeout

Selective repeat: the approach

- *pipelining*: *multiple* packets in flight
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) "window" over *N* consecutive seq #s
    - limits pipelined, "in flight" packets to be within this window

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

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 smallest 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

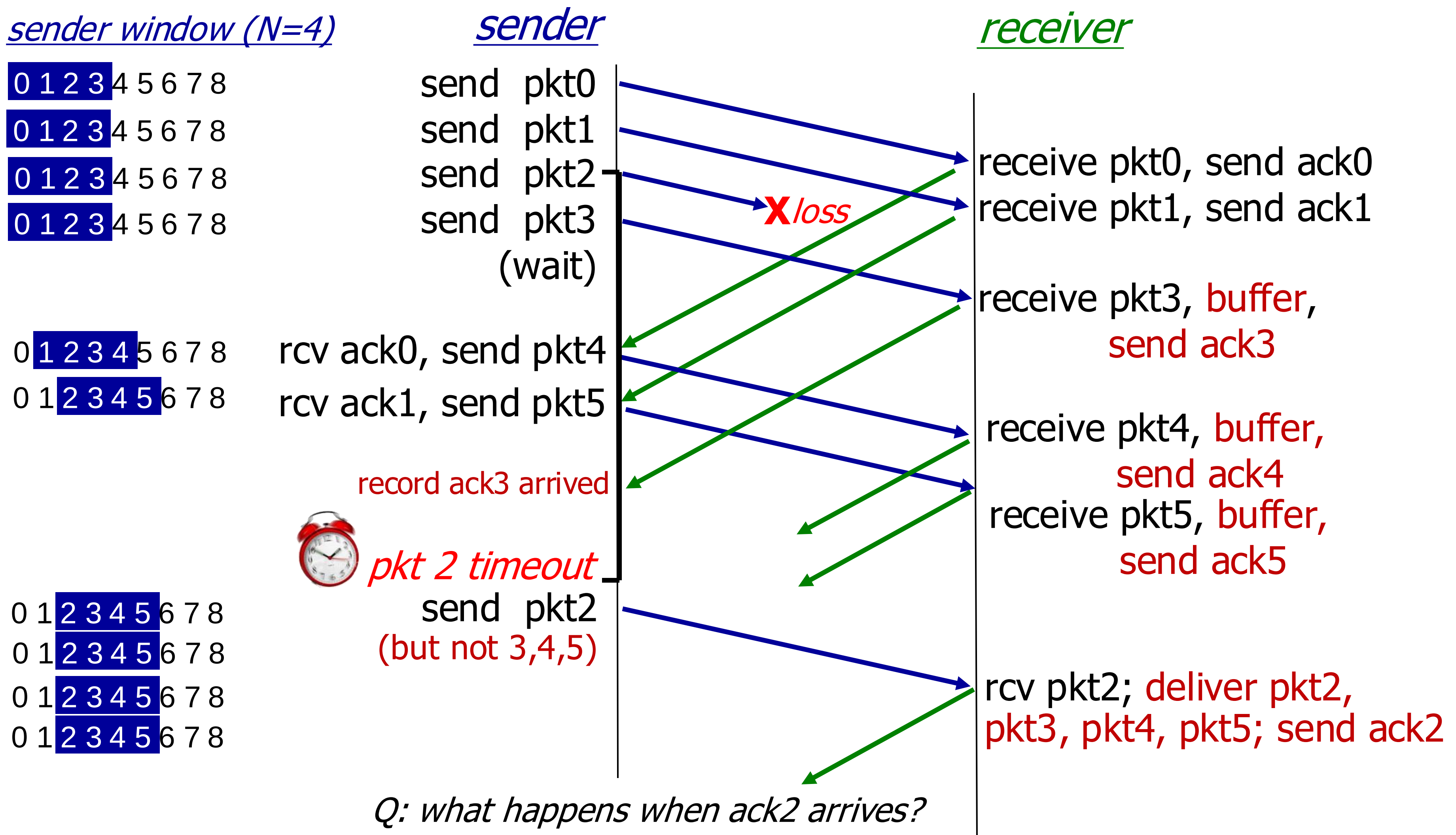
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action

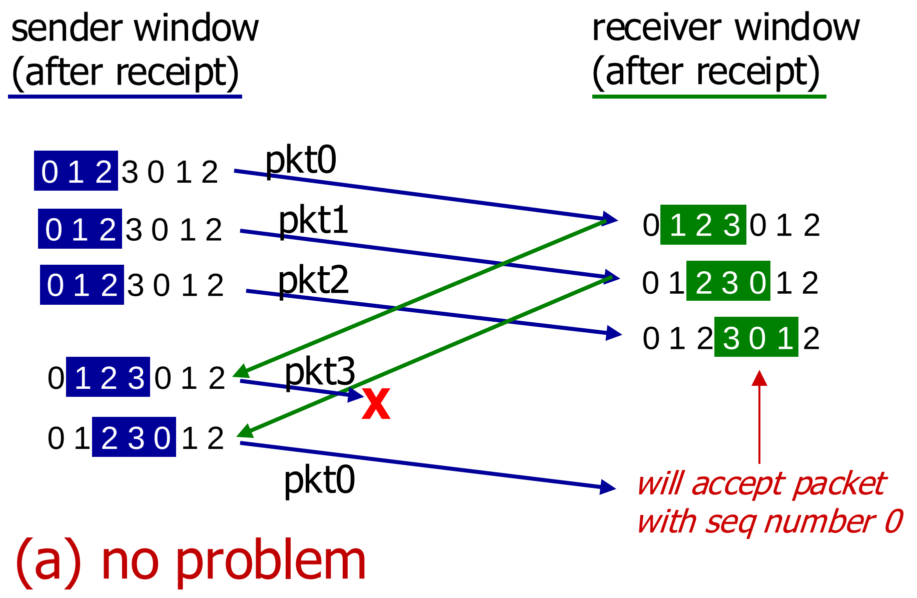


Q: what happens when ack2 arrives?

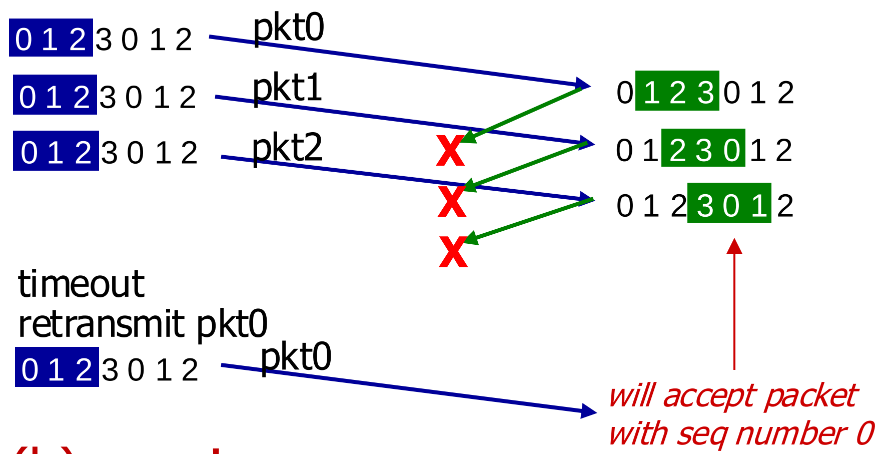
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



(a) no problem

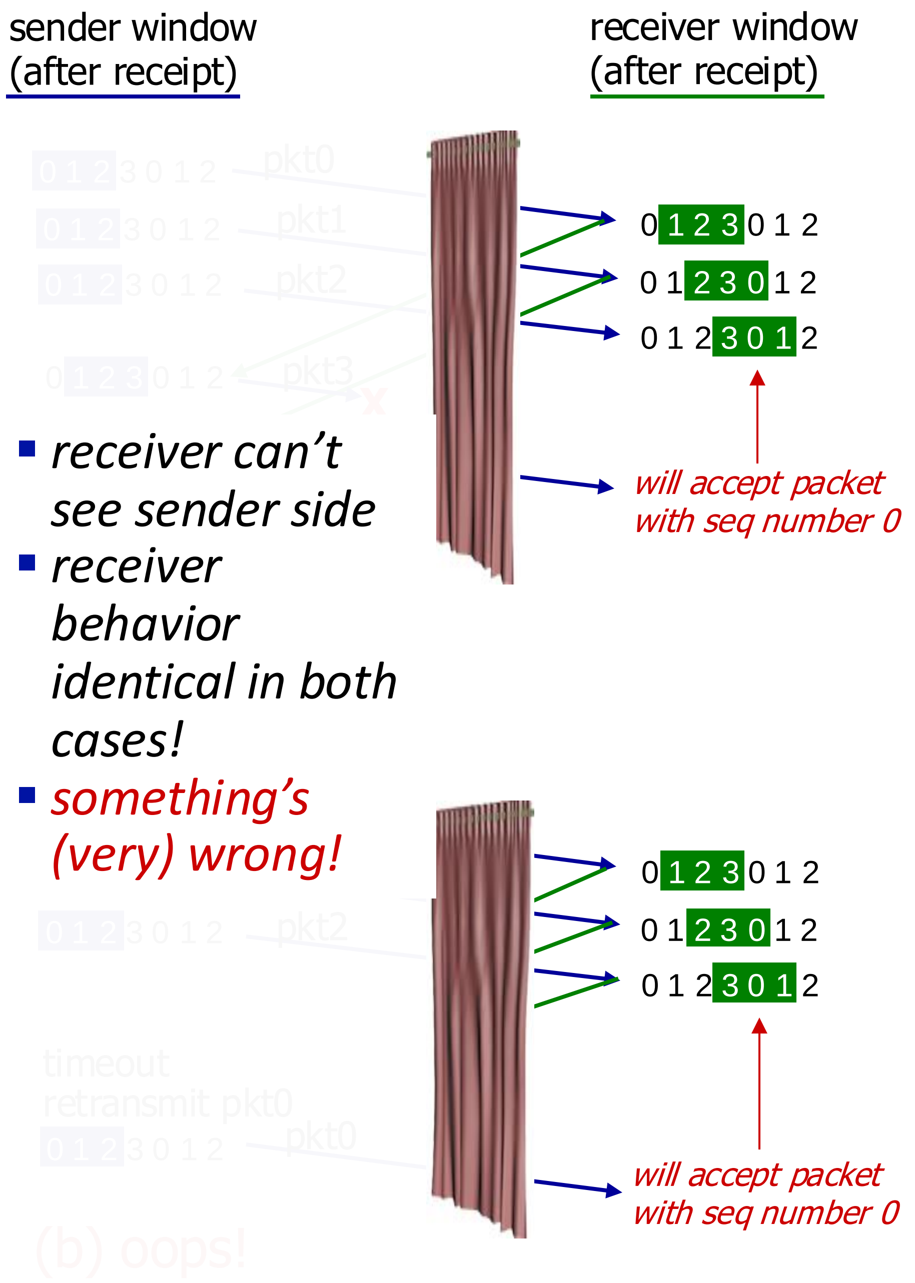


(b) oops!

Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3


sender window (after receipt)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
will accept packet with seq number 0
timeout retransmit pkt0
(b) oops!
will accept packet with seq number 0

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

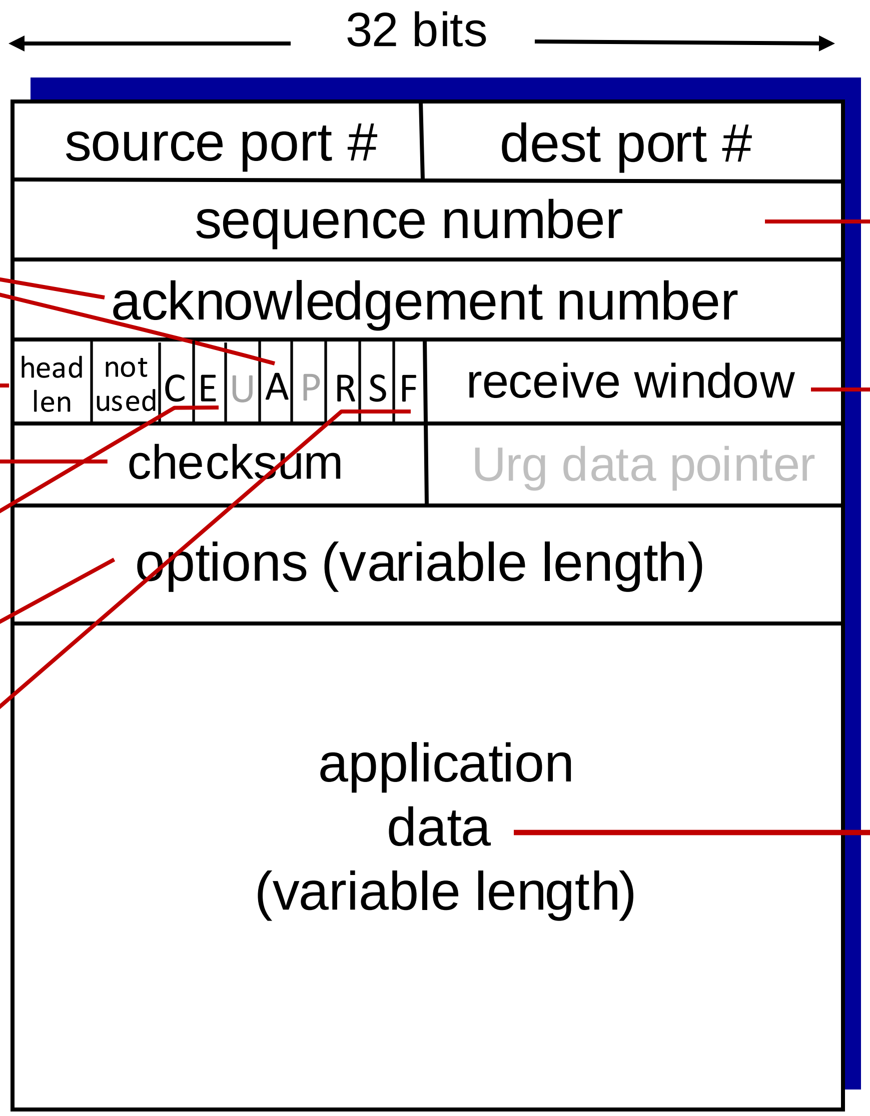
Chapter 3: roadmap

- 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 steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- 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



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

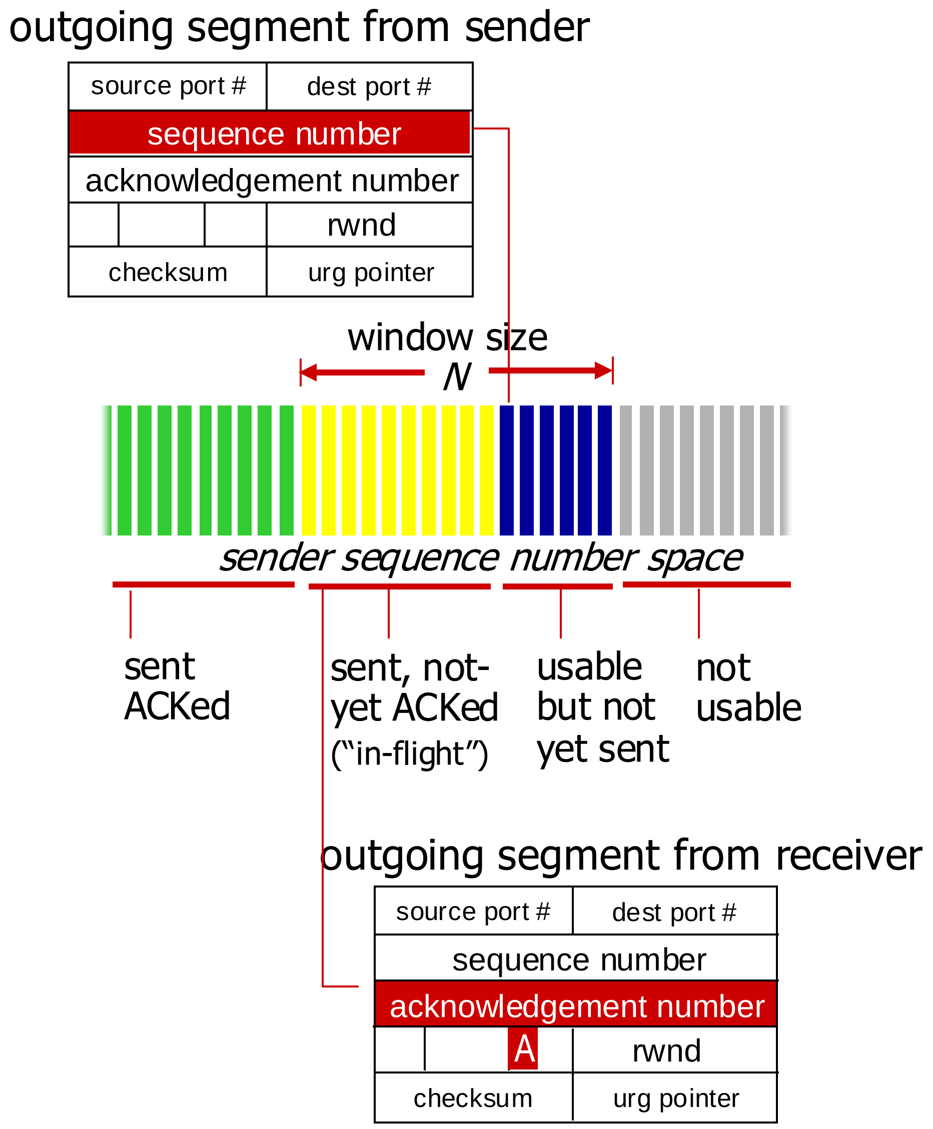
- byte stream "number" of first byte in segment's data

Acknowledgements:

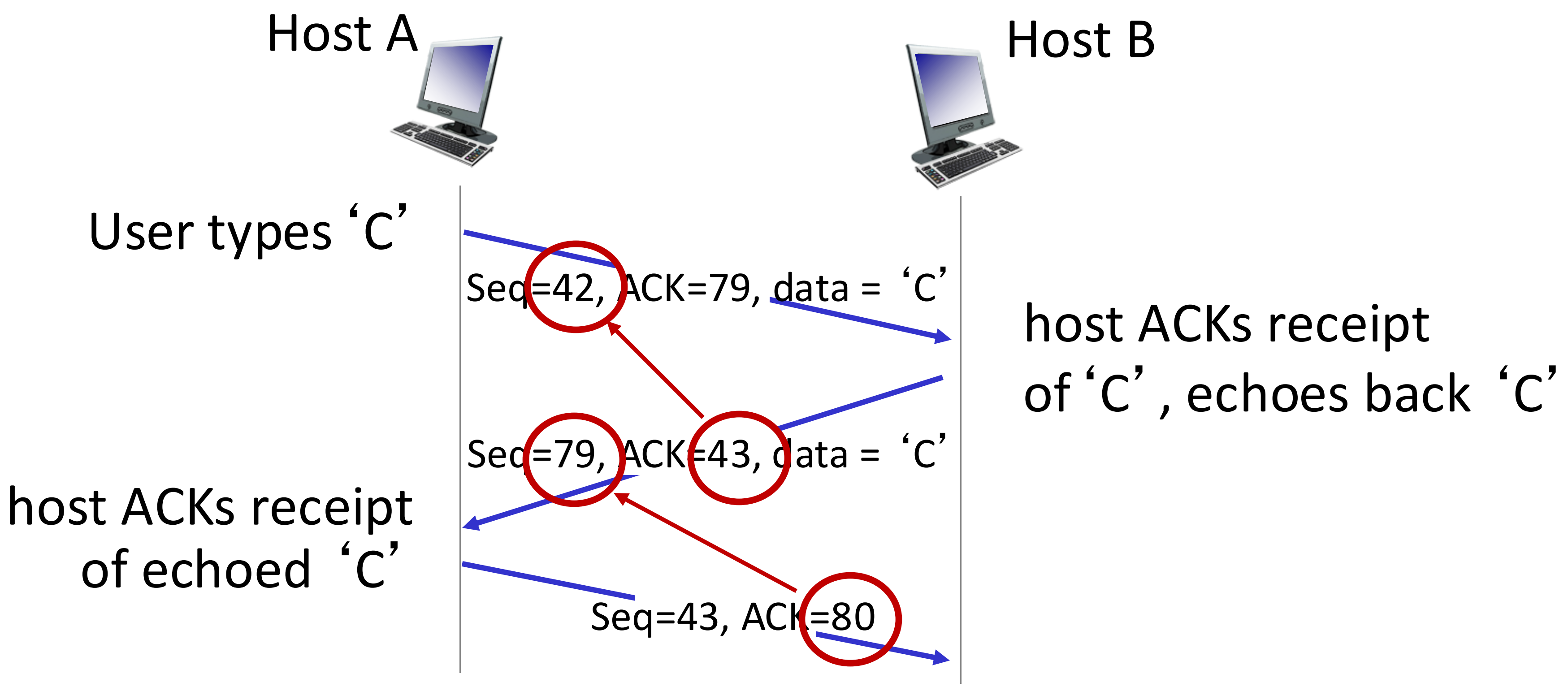
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs


Host A
Host B
User types ‘C’
Seq=42, ACK=79, data = ‘C’
host ACKs receipt of ‘C’, echoes back ‘C’
Seq=79, ACK=43, data = ‘C’
host ACKs receipt of echoed ‘C’
Seq=43, ACK=80

simple telnet scenario

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

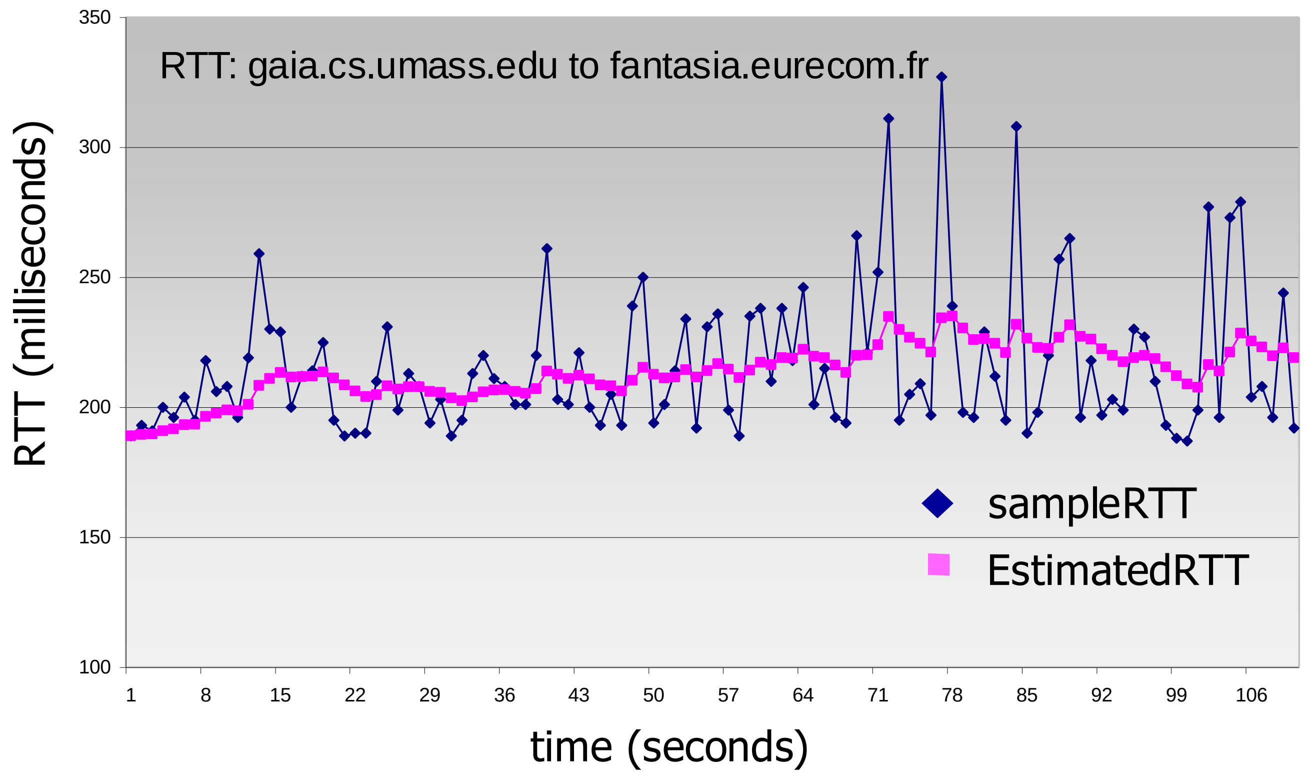
Q: how to estimate RTT?

- `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`

TCP round trip time, 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, timeout

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

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

estimated RTT (↑ `EstimatedRTT`)

"safety margin" (↑ `DevRTT`)

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

$$\text{DevRTT} = (1-\beta)*\text{DevRTT} + \beta*|\text{SampleRTT}-\text{EstimatedRTT}|$$

(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

- 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 segment that caused timeout
- restart timer

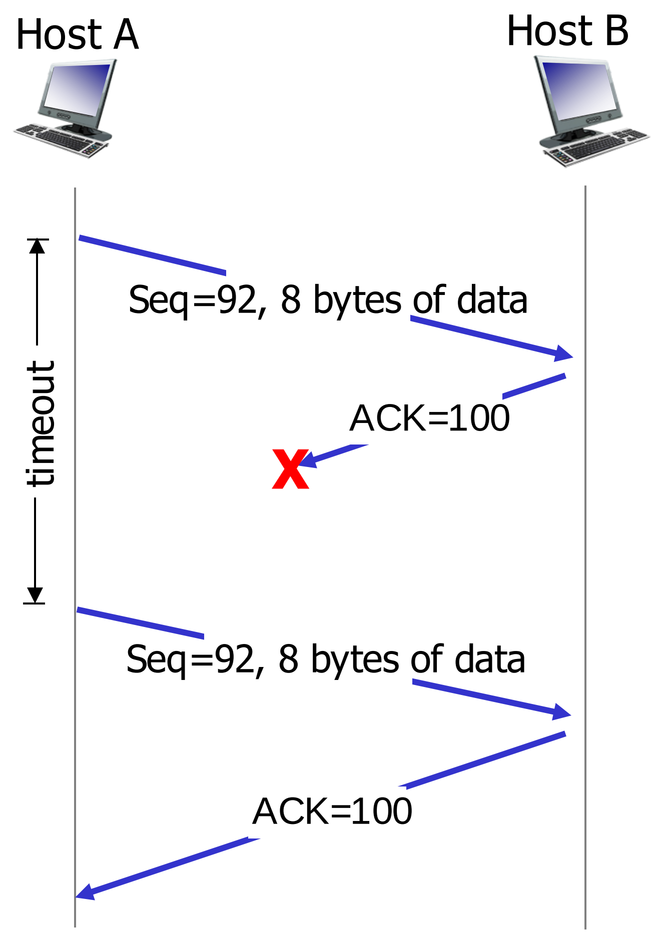
event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

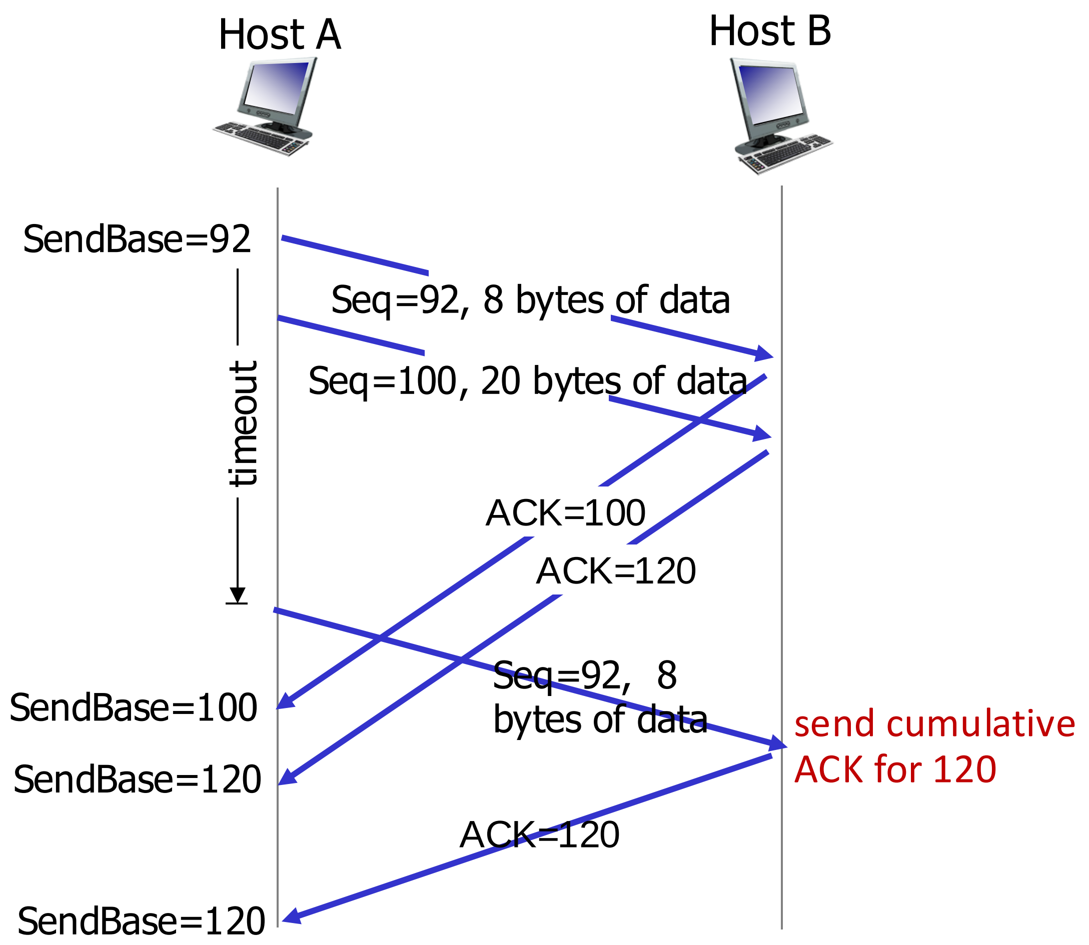
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*

TCP: retransmission scenarios

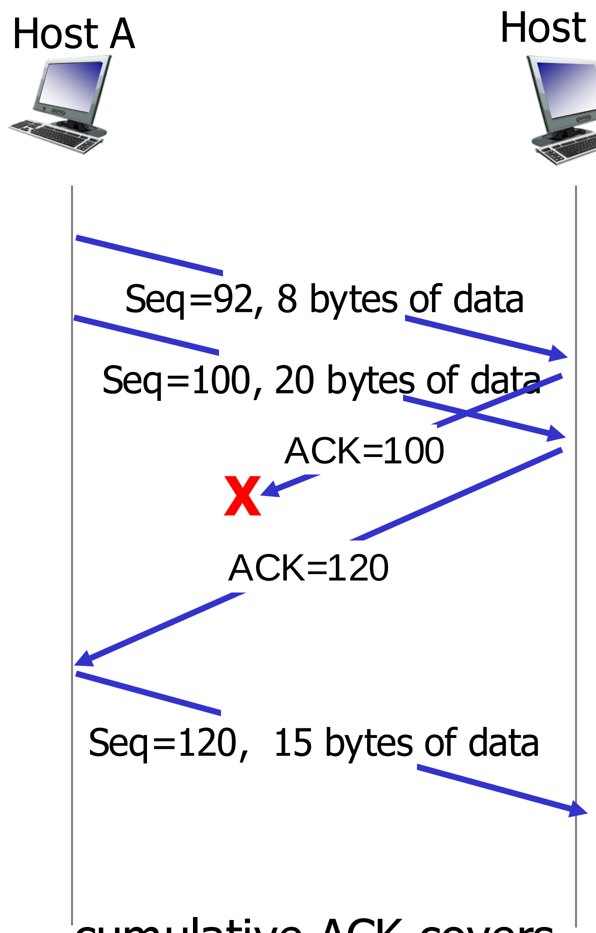


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK covers for earlier lost ACK

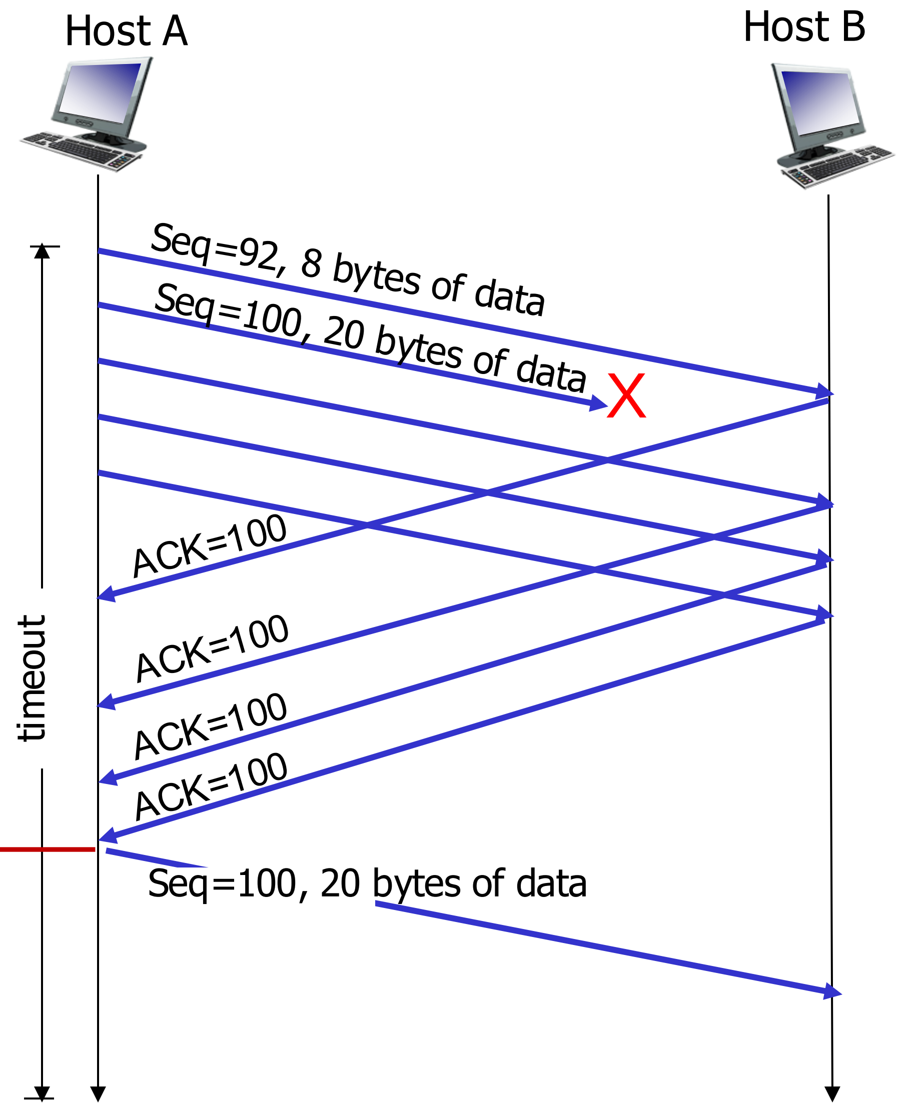
TCP fast retransmit

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!



Host A
Host B
Seq=92, 8 bytes of data
Seq=100, 20 bytes of data
X
ACK=100
ACK=100
ACK=100
ACK=100
timeout
Seq=100, 20 bytes of data