Introduction to Networking

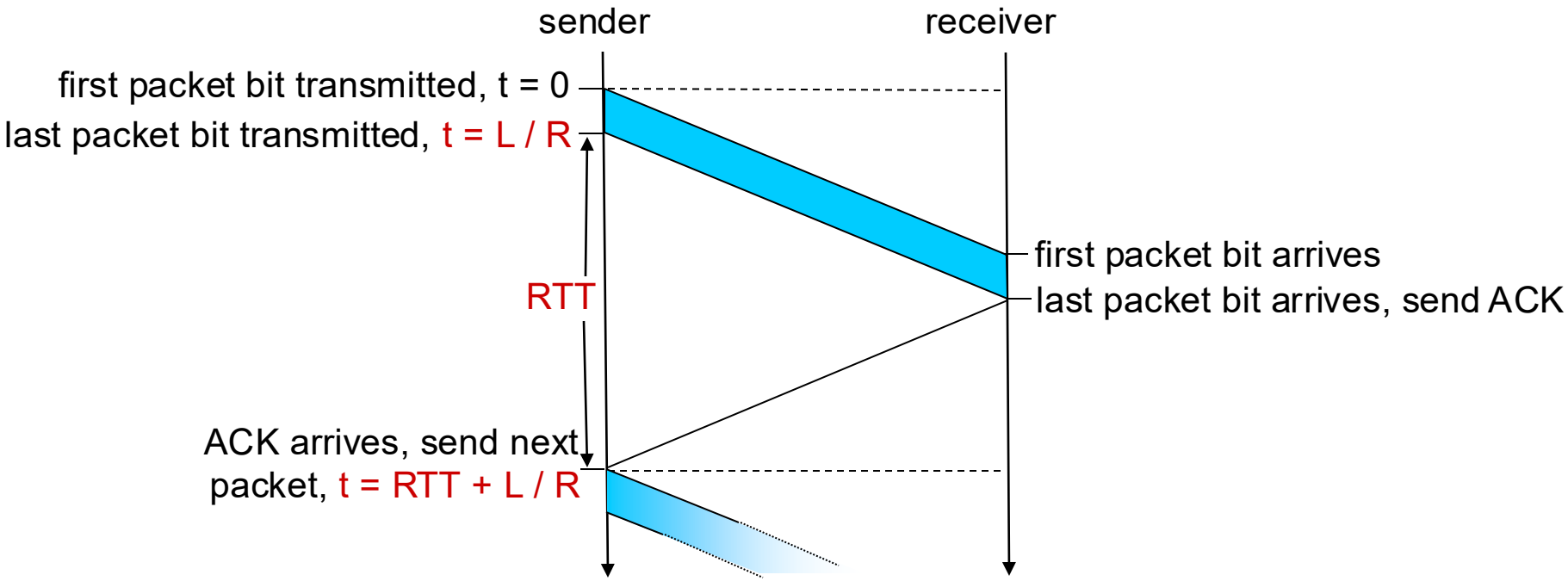
CAN201 – Week 5

Module Leader: Dr. Fei Cheng & Dr. Gordon Boateng

Lecture 5 – Transport Layer (2)

- **Roadmap**
  1. Pipelined communication
  2. TCP: connection-oriented transport

rdt3.0: stop-and-wait operation



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

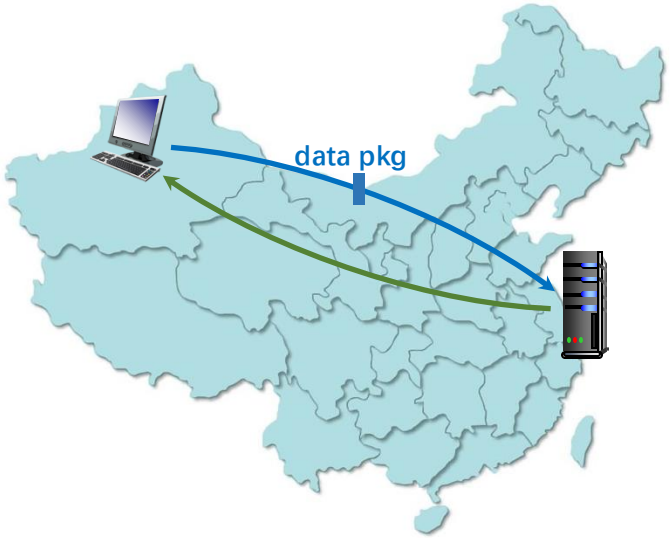
Question

- How to increase utilization?

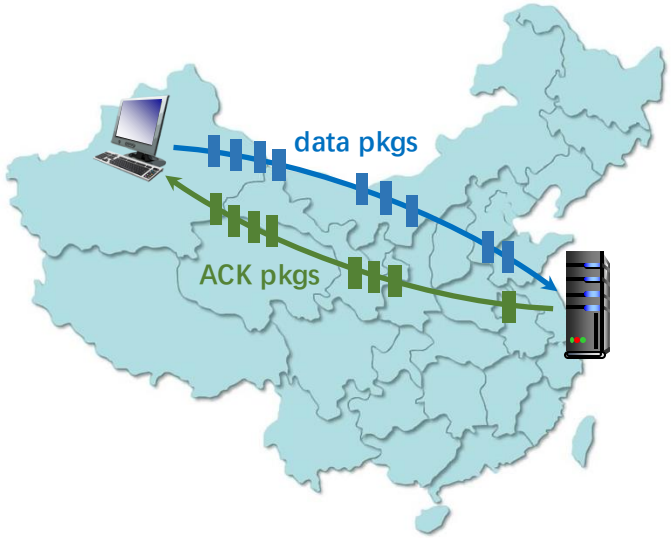

哈萨克斯坦
蒙古
总长：3257.6公里
km
Urumqi
北京
朝鲜
渤海
韩国
黄海
中华人民共和国
起点
Shanghai
东海
印度
孟加拉国
缅甸
老挝

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

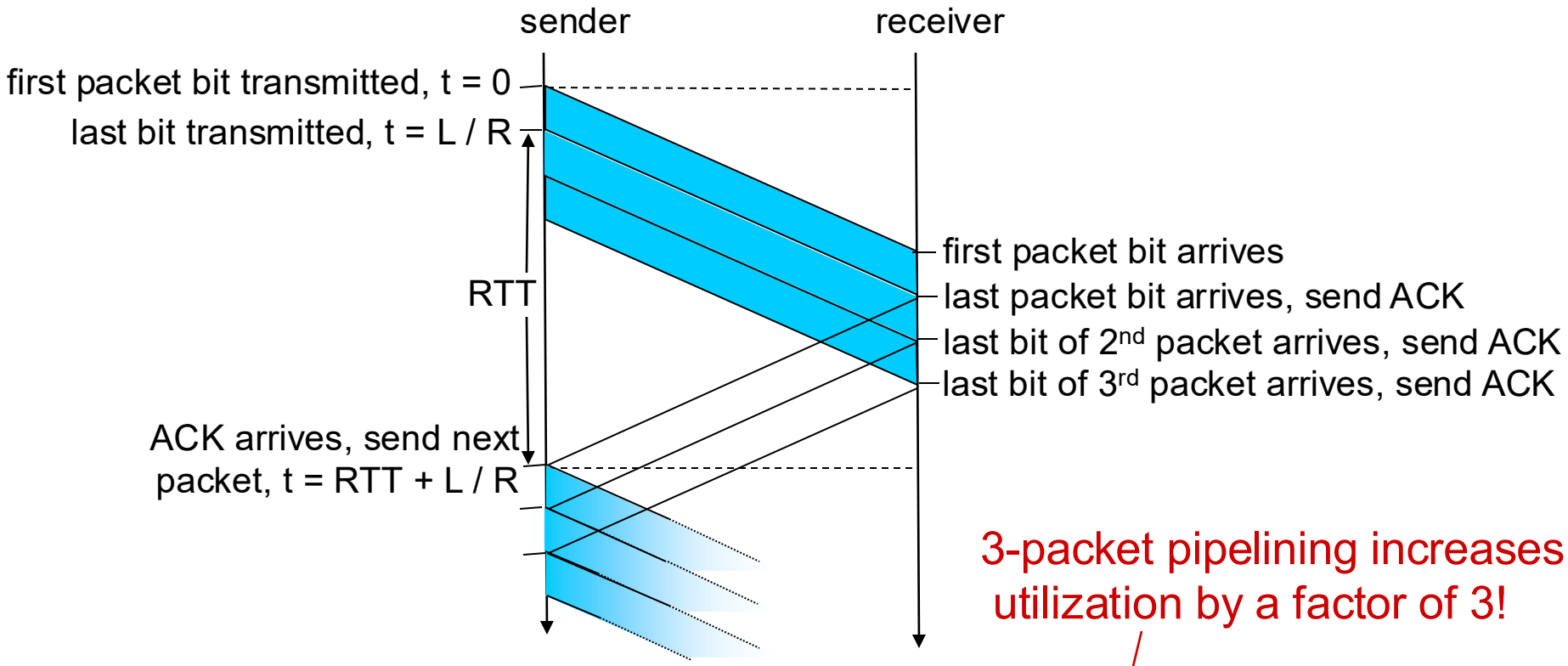


a stop-and-wait protocol in operation



a pipelined protocol in operation

Pipelining: increased utilization



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

$$U_{sender} = \frac{3L / R}{RTT + 3L / R} = \frac{.024}{30.024} = 0.0008$$

Two forms: Go-Back-N and Selective repeat

Go-back-N (GBN):

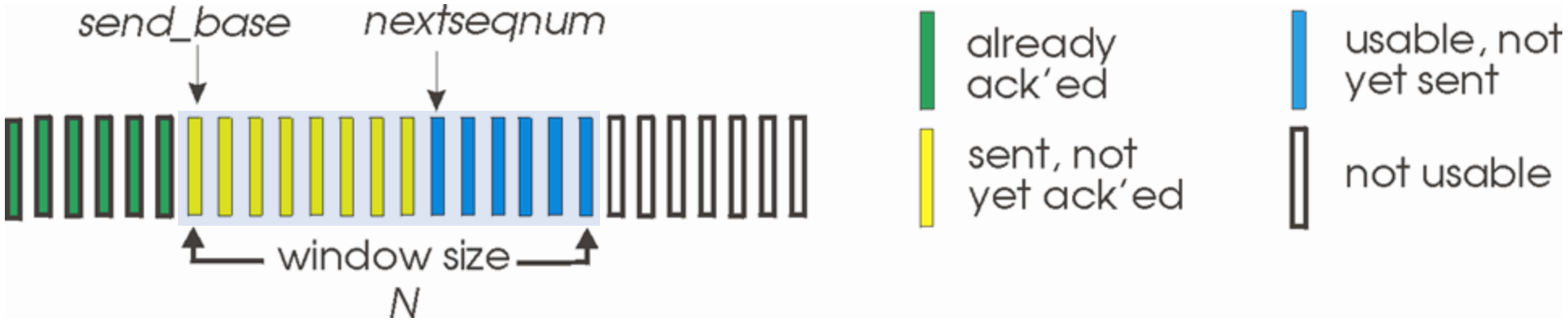
- **Sender can have up to N unacked packets in pipeline**
- **Receiver only sends *cumulative ACK***
  - Doesn't ACK packet if there's a gap
- **Sender has timer for oldest unACKed packet**
  - When timer expires, retransmit *all* unacked packets

Selective Repeat (SR):

- **Sender can have up to N unacked packets in pipeline**
- **Rcvr sends individual ack for each packet**
- **Sender maintains timer for each unacked packet**
  - When timer expires, retransmit only that unacked packet

Go-Back-N: sender

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

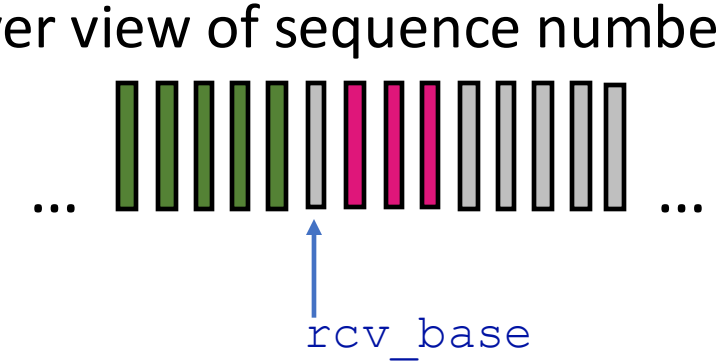


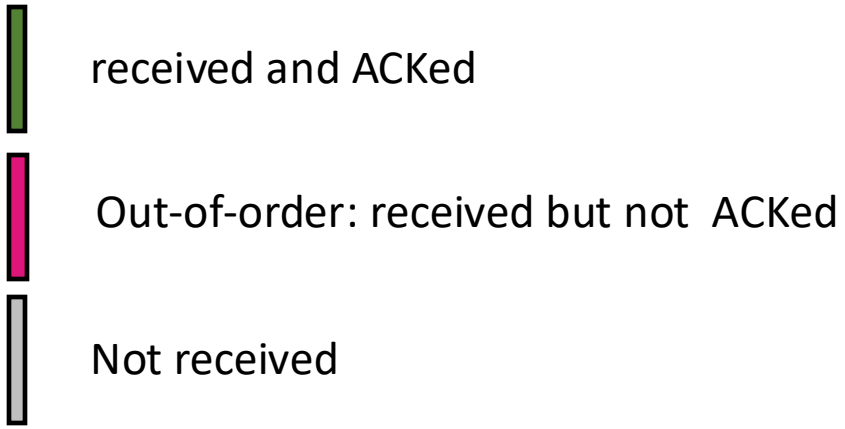
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

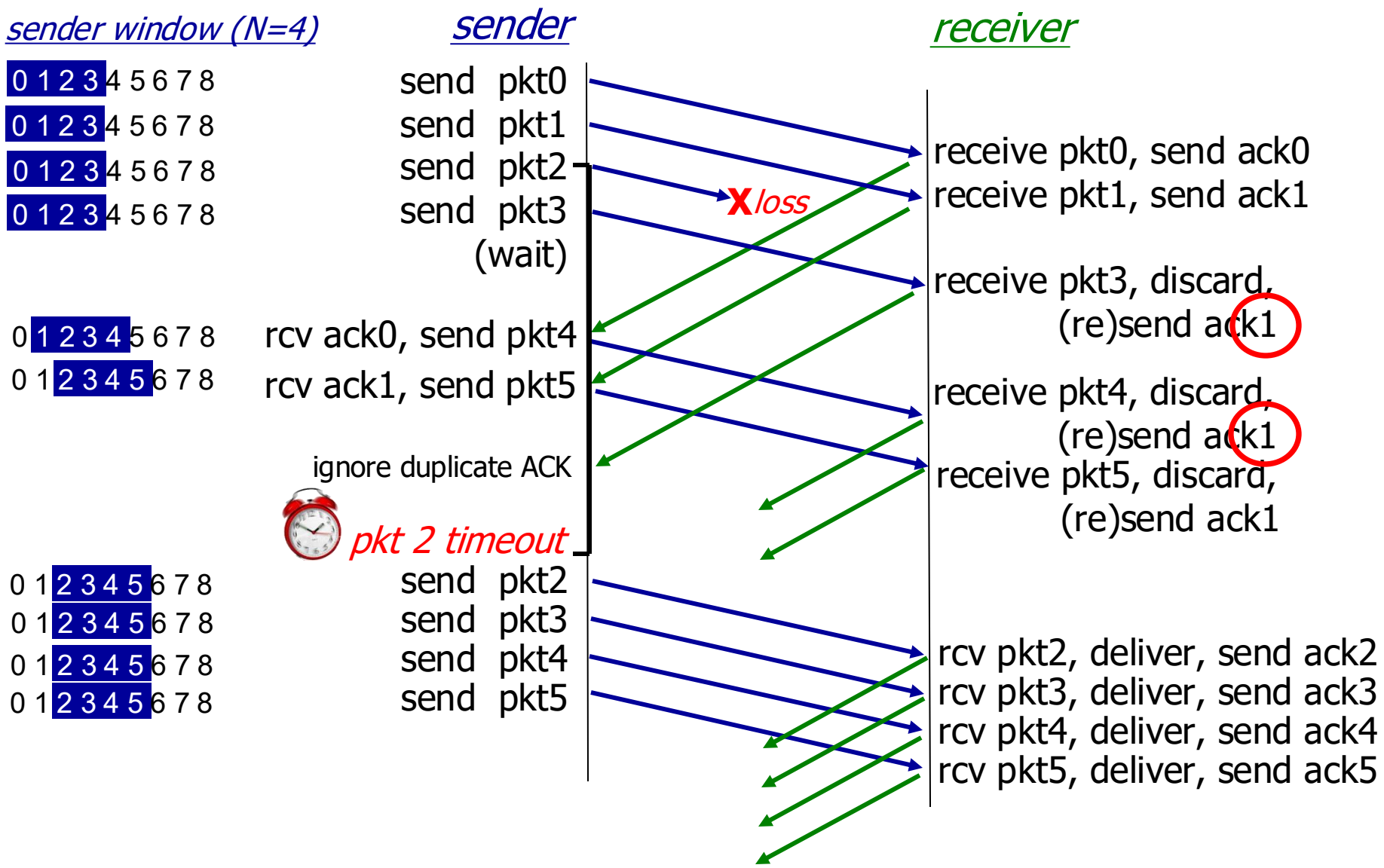
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order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

Receiver view of sequence number space:





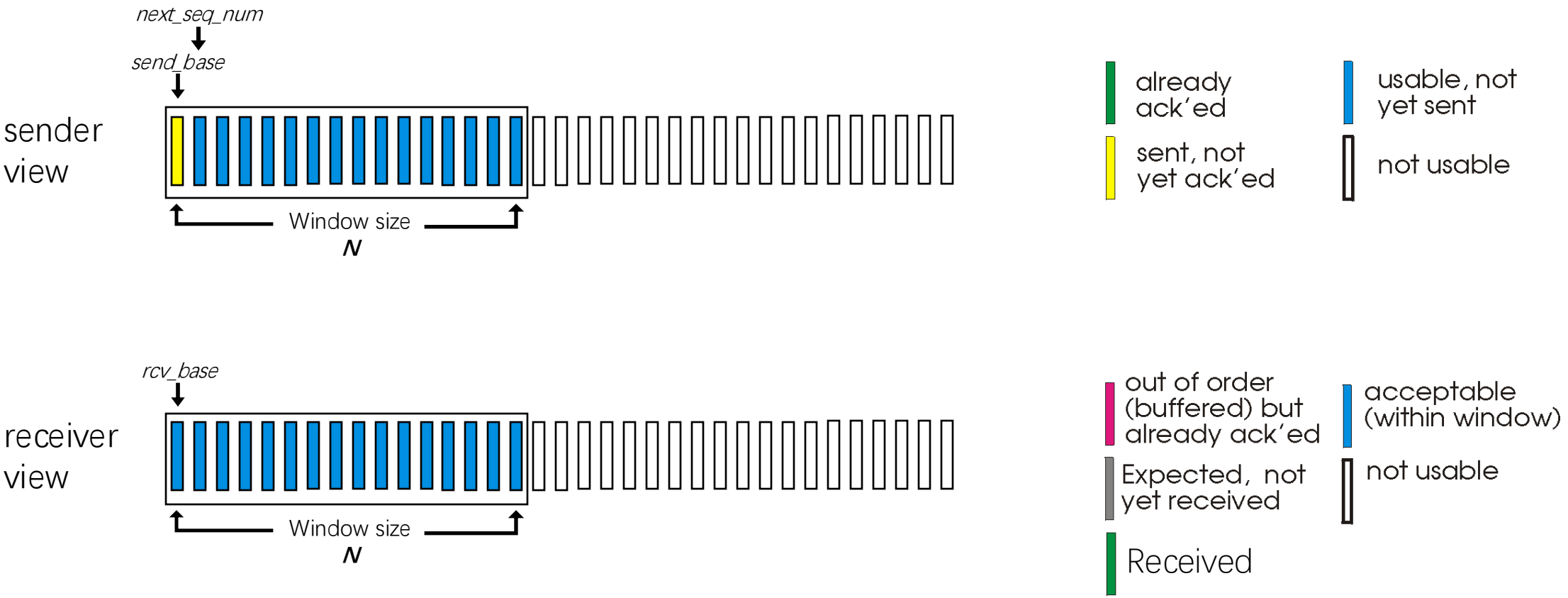
Go-Back-N in action



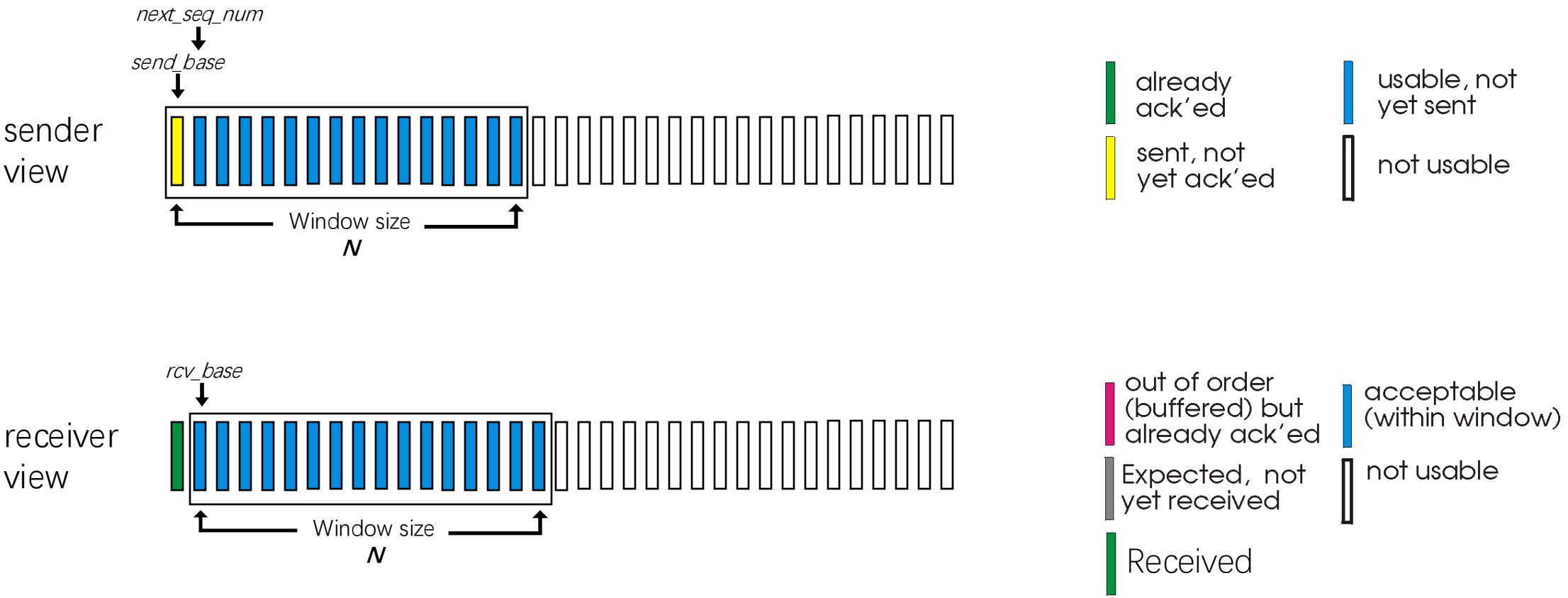
Selective repeat: the approach

- *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

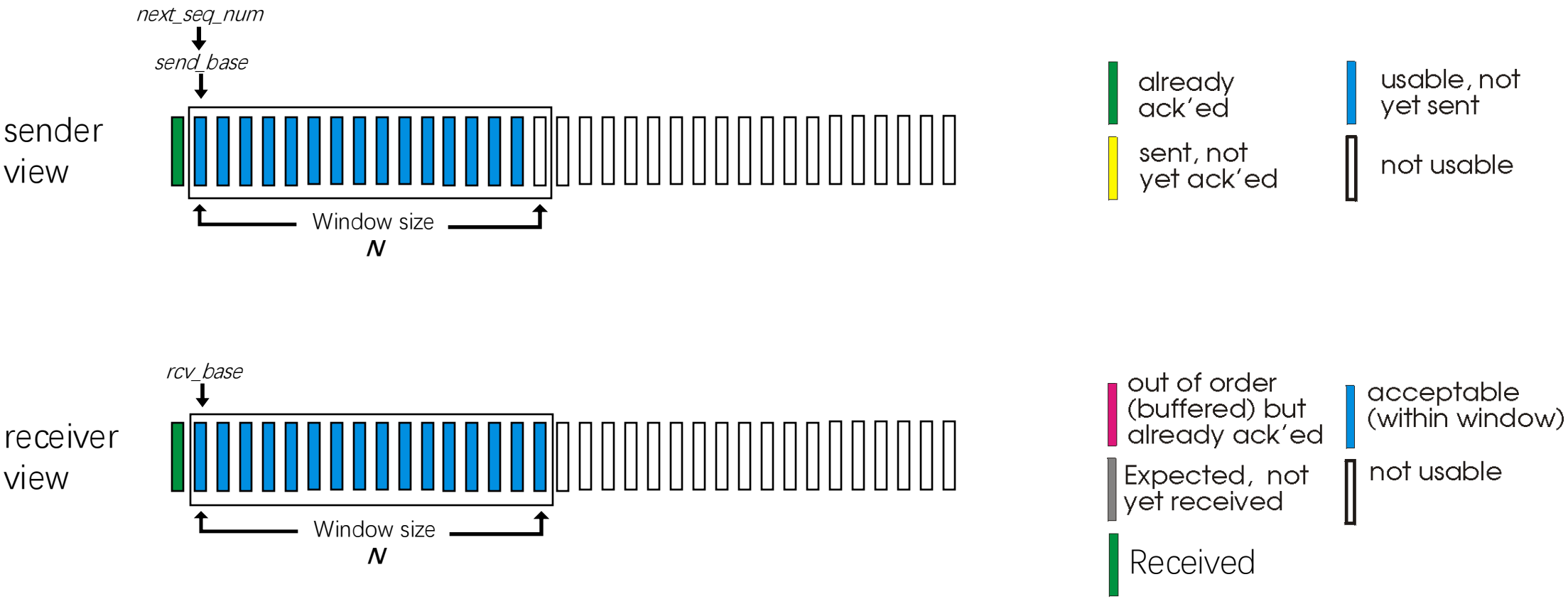
Selective repeat: sender, receiver windows



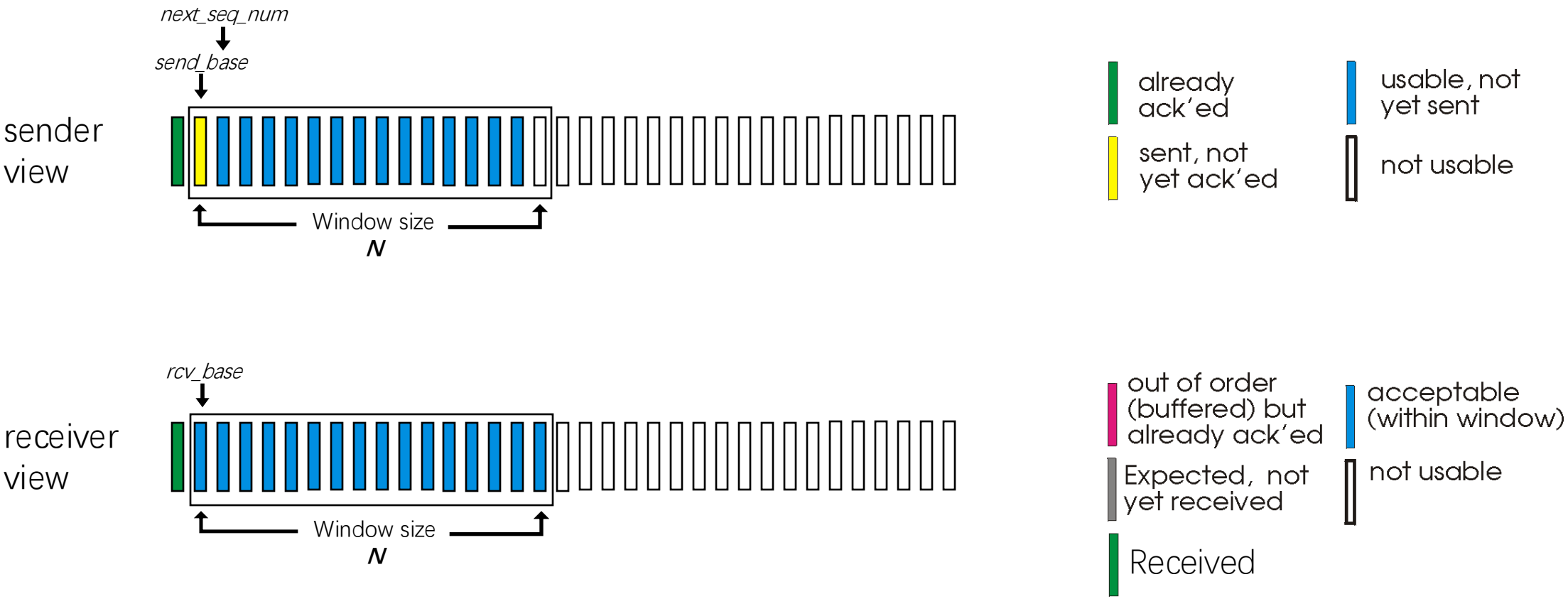
Selective repeat: sender, receiver windows



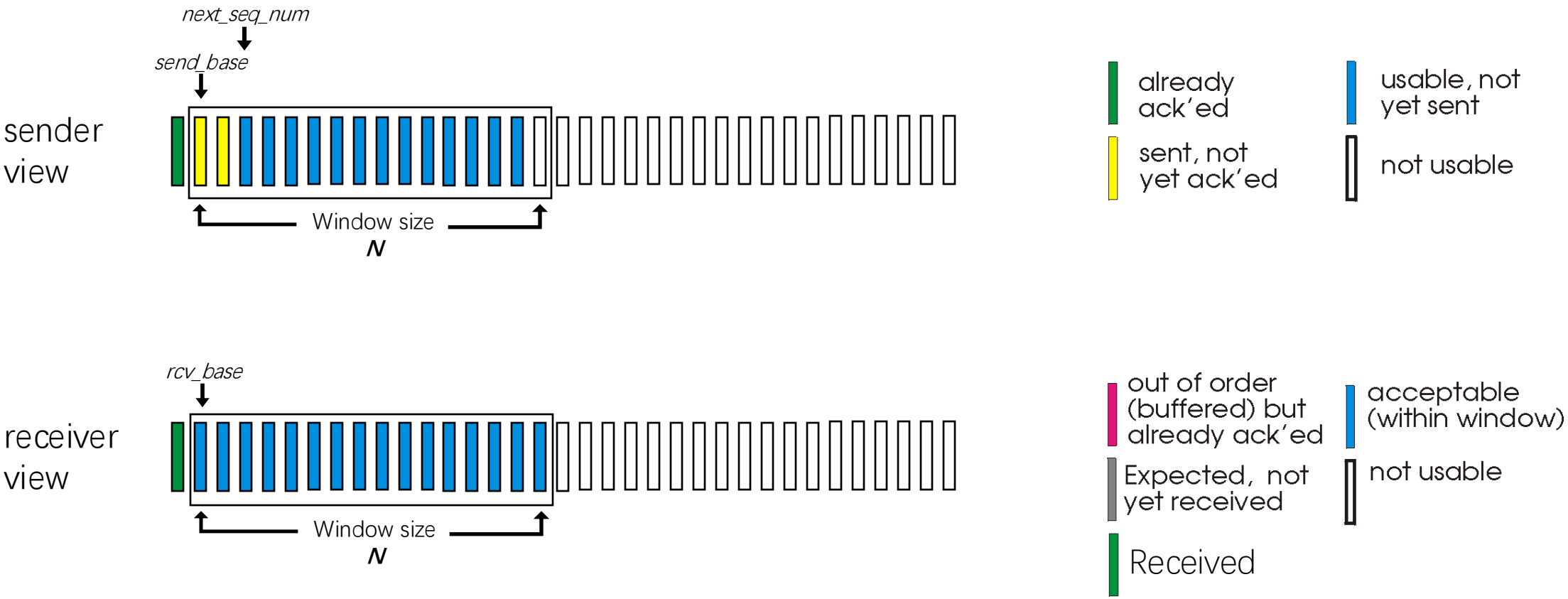
Selective repeat: sender, receiver windows



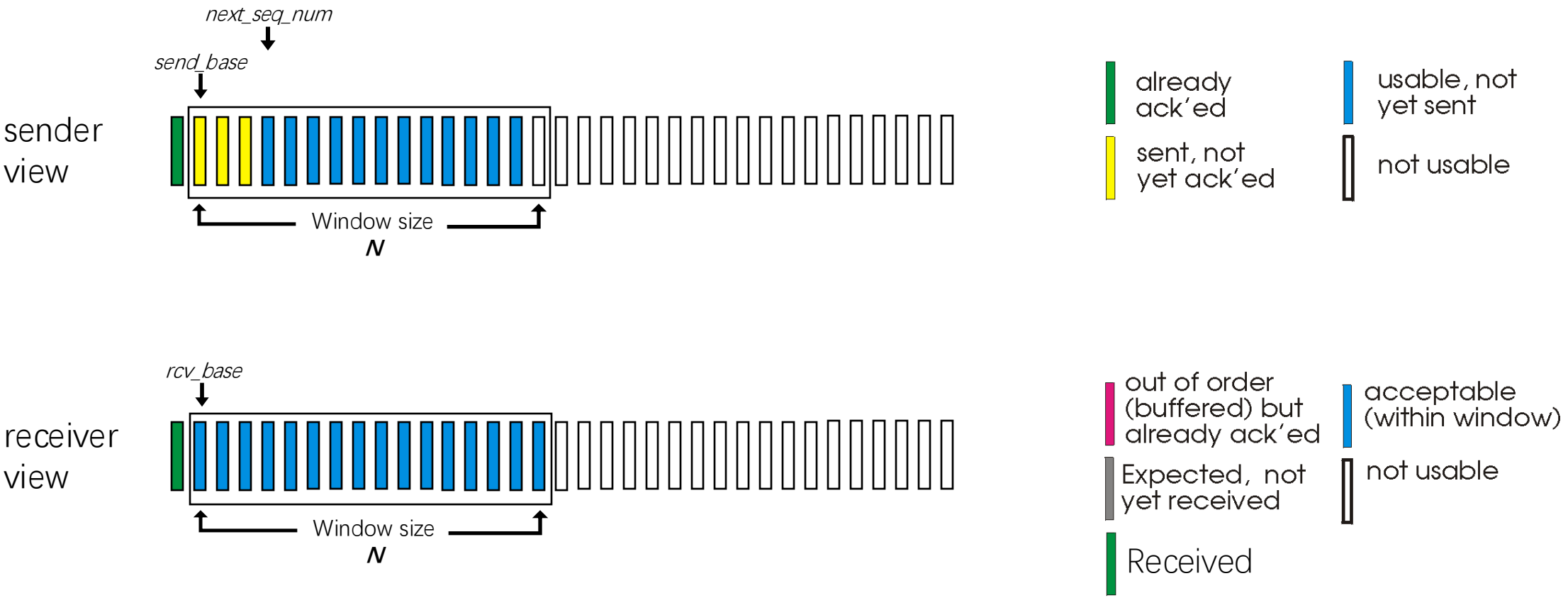
Selective repeat: sender, receiver windows



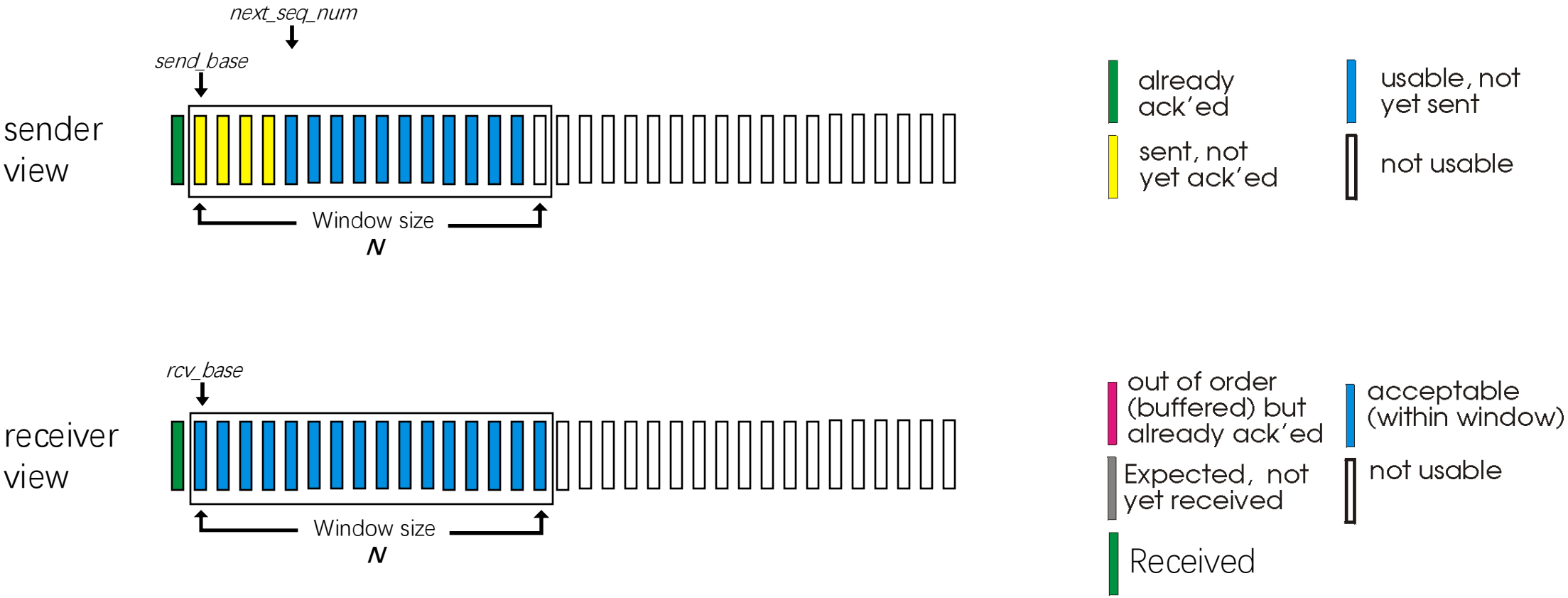
Selective repeat: sender, receiver windows



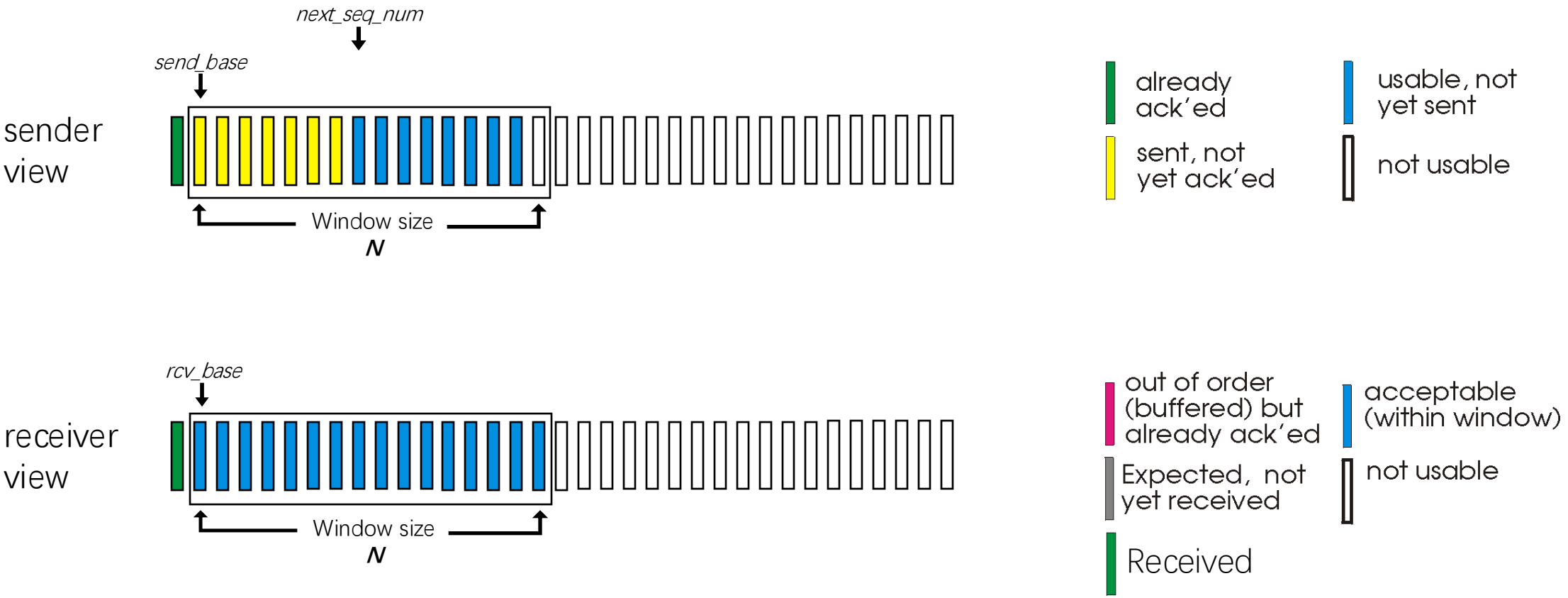
Selective repeat: sender, receiver windows



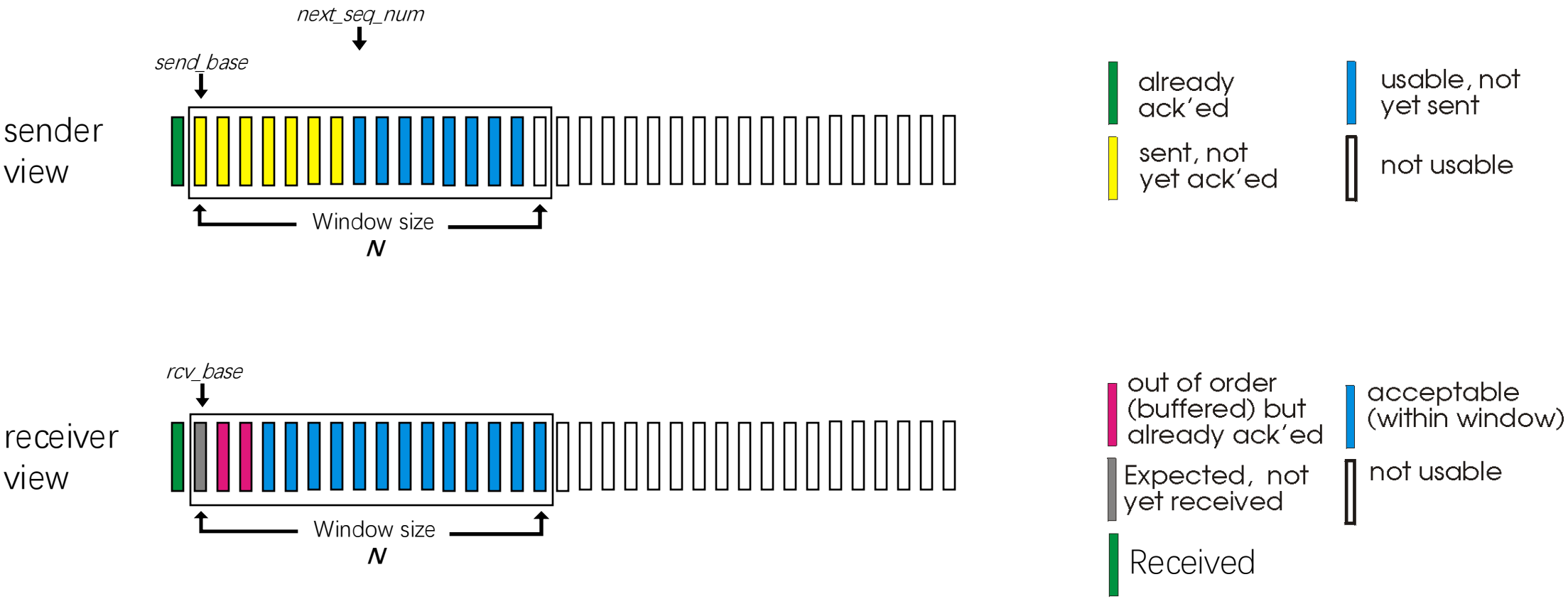
Selective repeat: sender, receiver windows



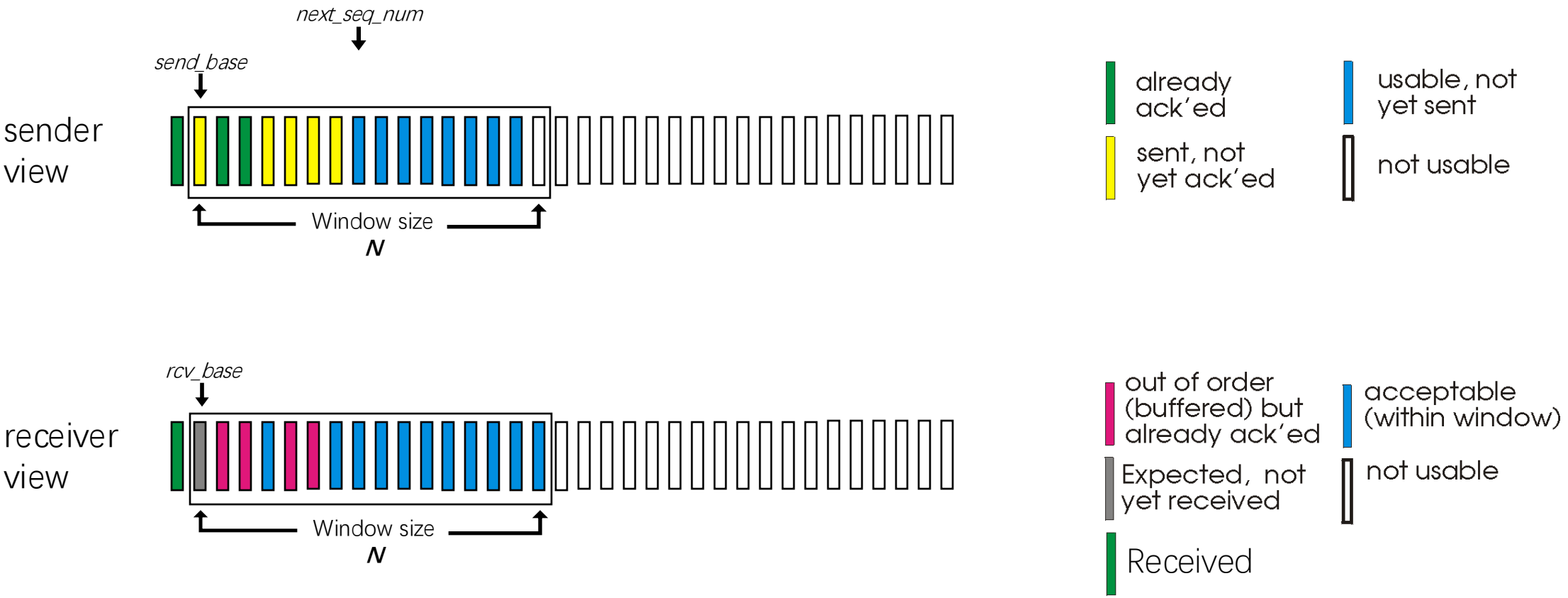
Selective repeat: sender, receiver windows



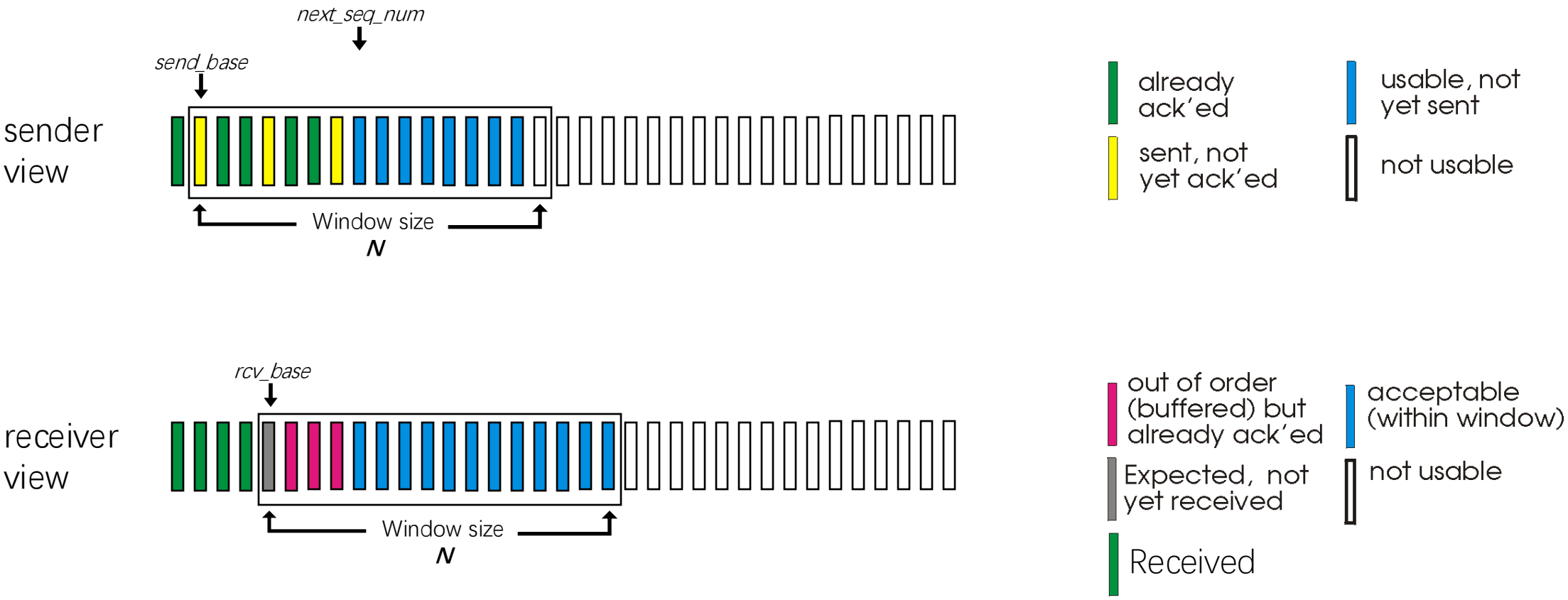
Selective repeat: sender, receiver windows



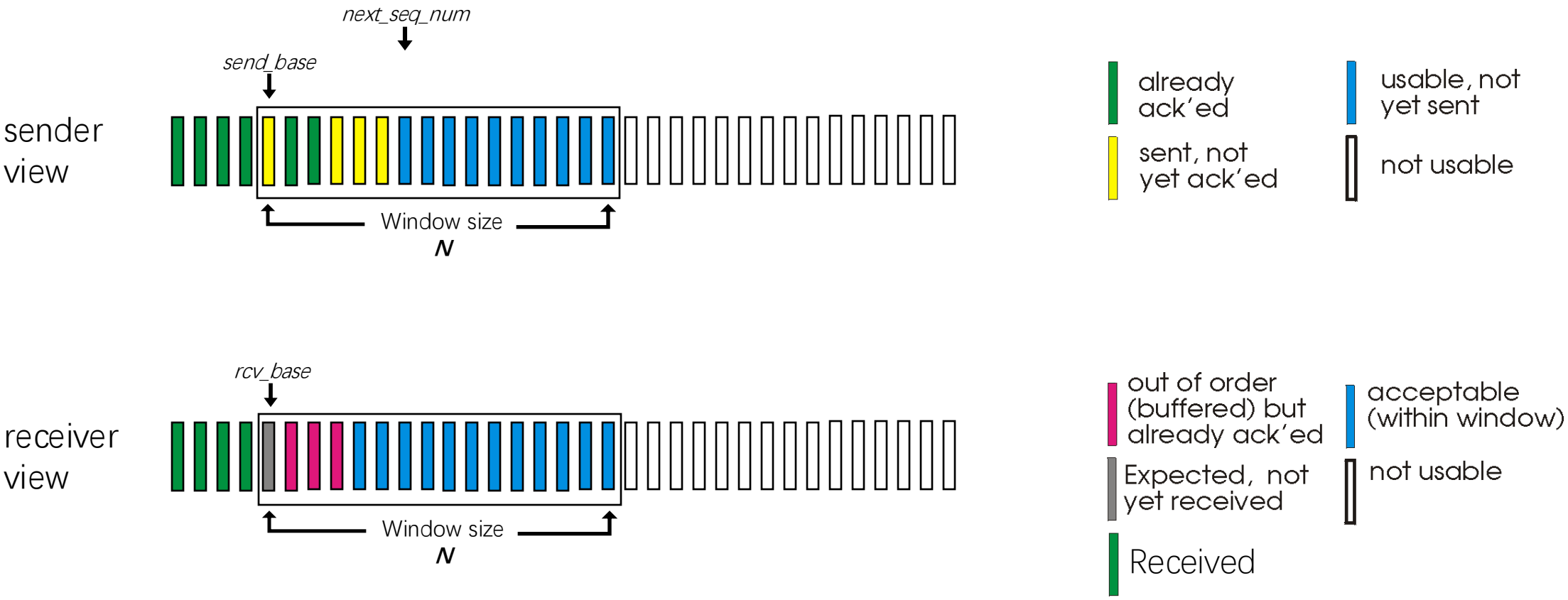
Selective repeat: sender, receiver windows



Selective repeat: sender, receiver windows



Selective repeat: sender, receiver windows



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

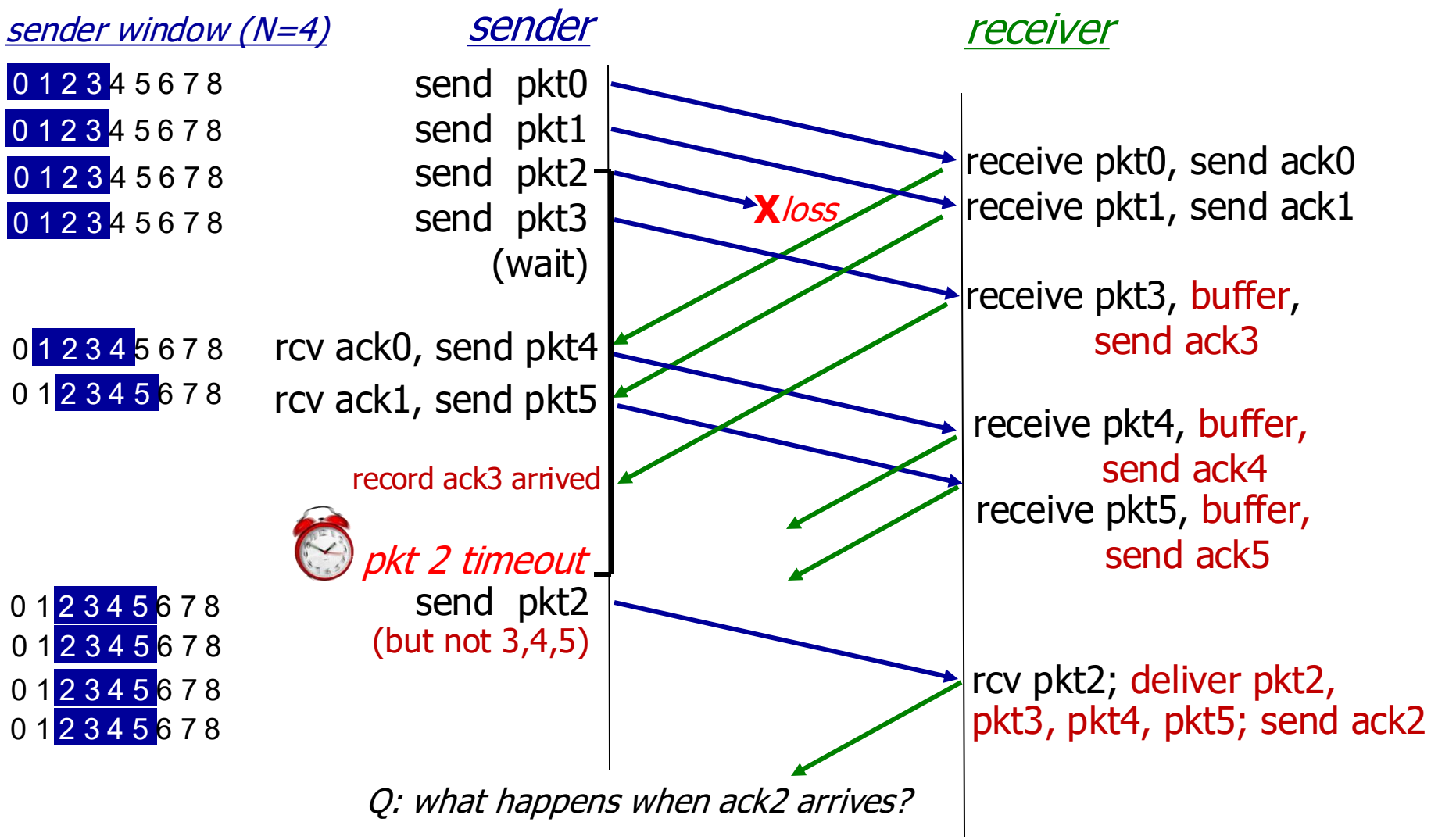
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- Ignore

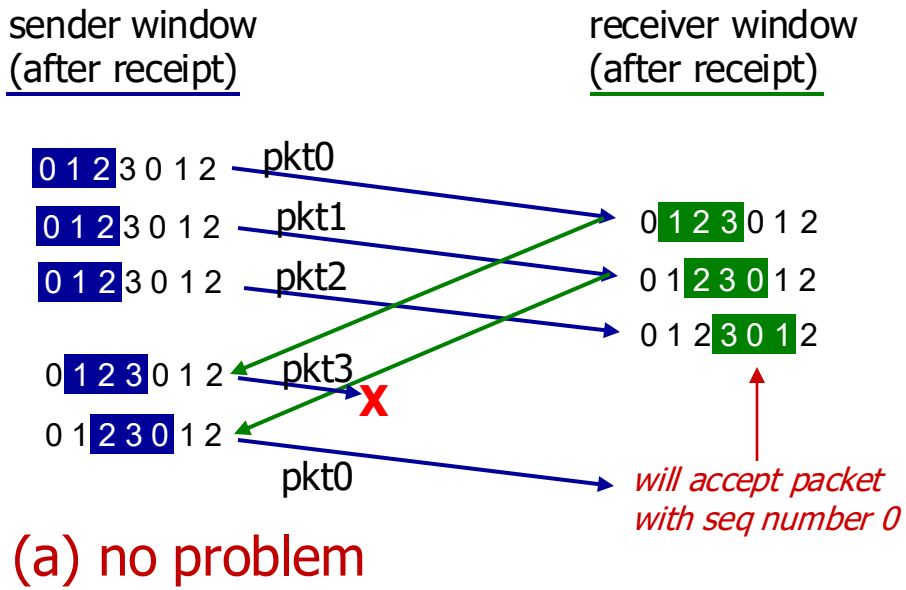
Selective Repeat in action



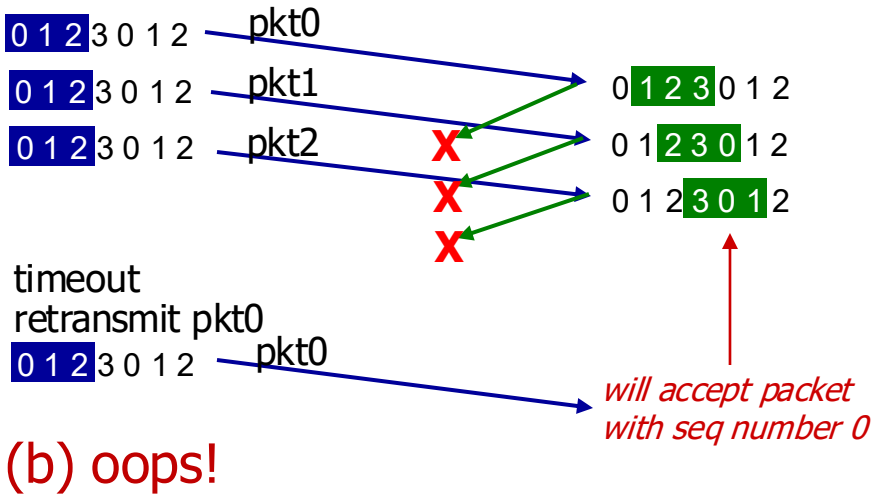
Selective repeat: a dilemma!

example:

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



(a) no problem



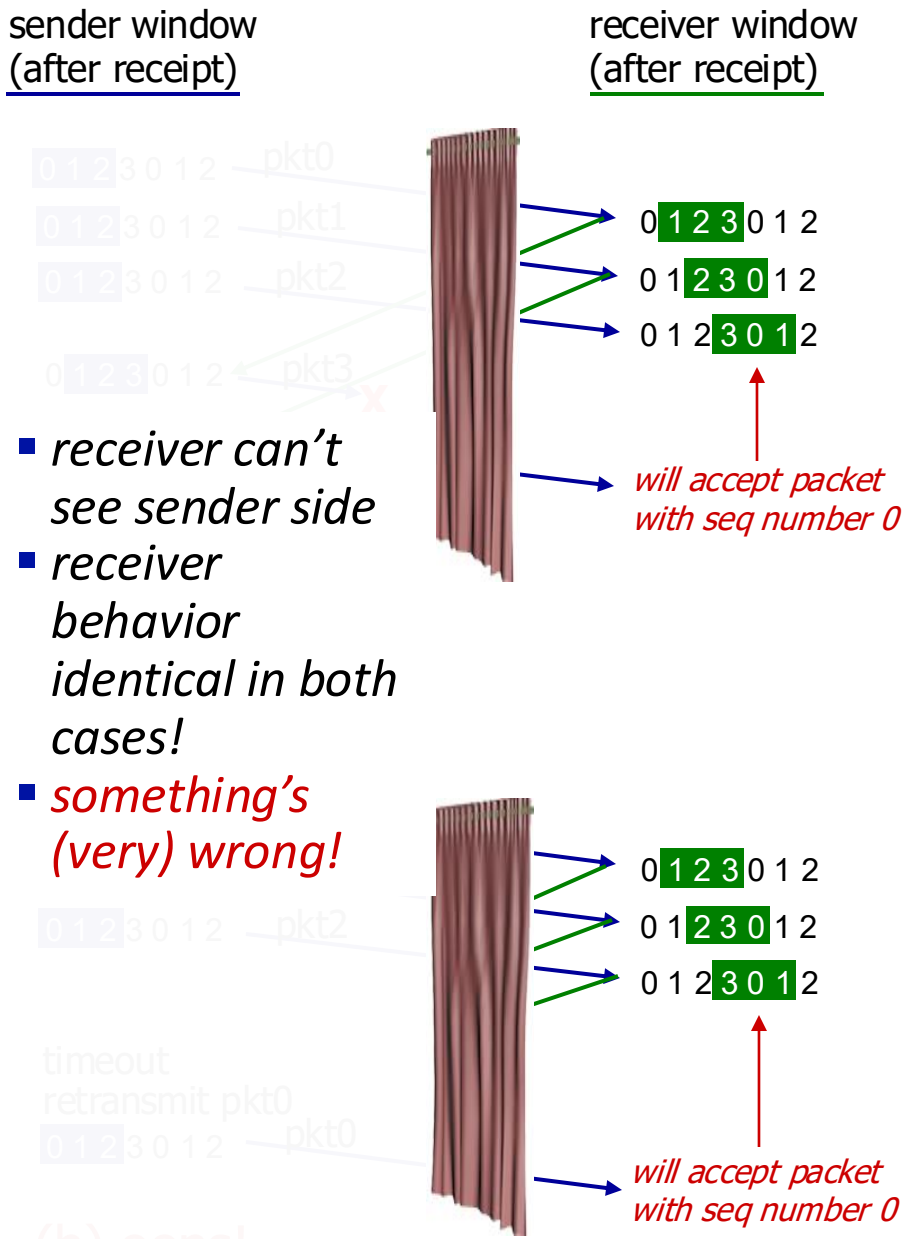
(b) oops!

Selective repeat: a dilemma!

example:

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

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



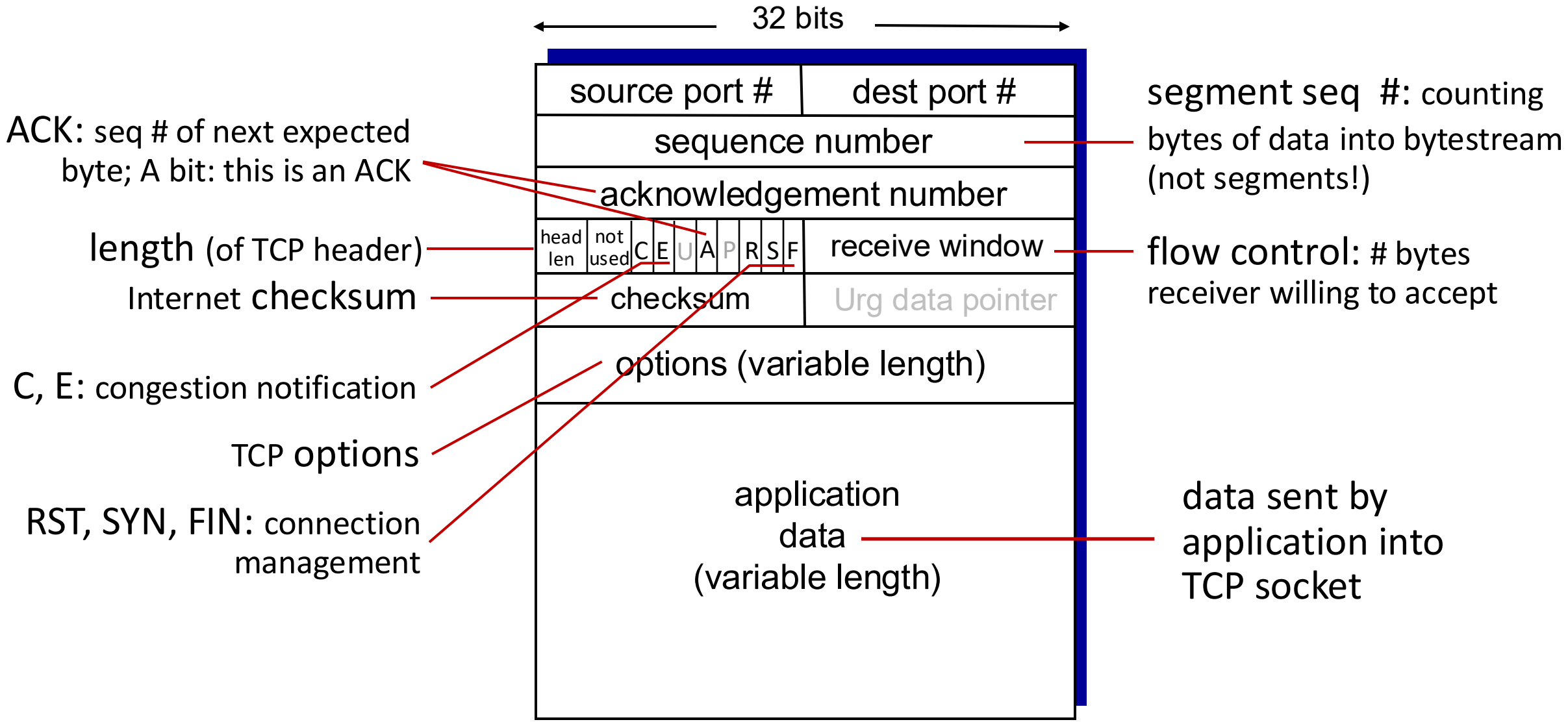
Lecture 5 – Transport Layer (2)

- **Roadmap**
  1. Pipelined communication
  2. TCP: connection-oriented transport

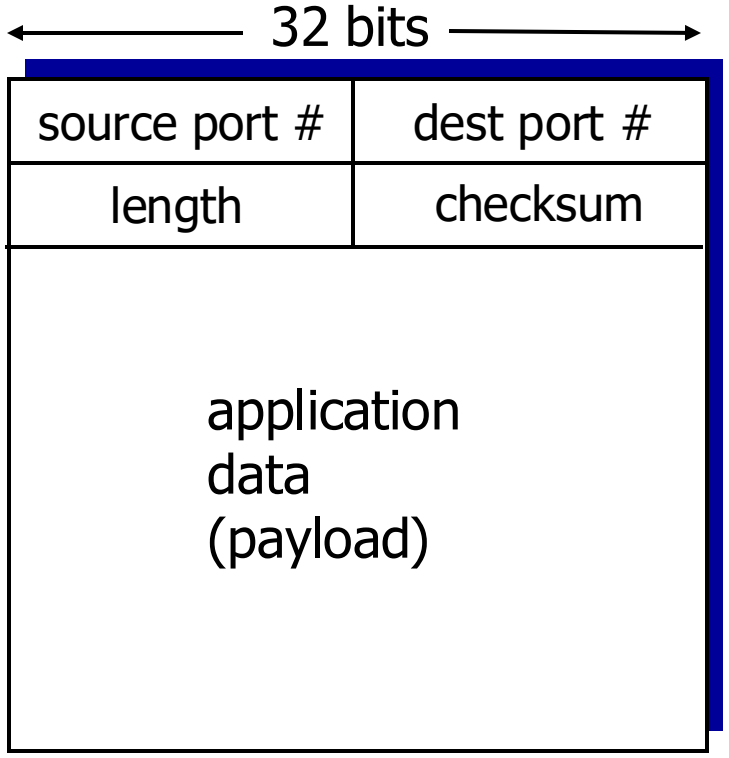
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- **Point-to-point:**
  - one sender, one receiver
- **Reliable, in-order *byte stream* :**
  - no "message boundaries"
- **Pipelined:**
  - TCP congestion and flow control set window size
- **Full duplex data:**
  - bi-directional data flow in same connection
- **Connection-oriented:**
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- **Flow controlled:**
  - sender will not overwhelm receiver

TCP segment structure

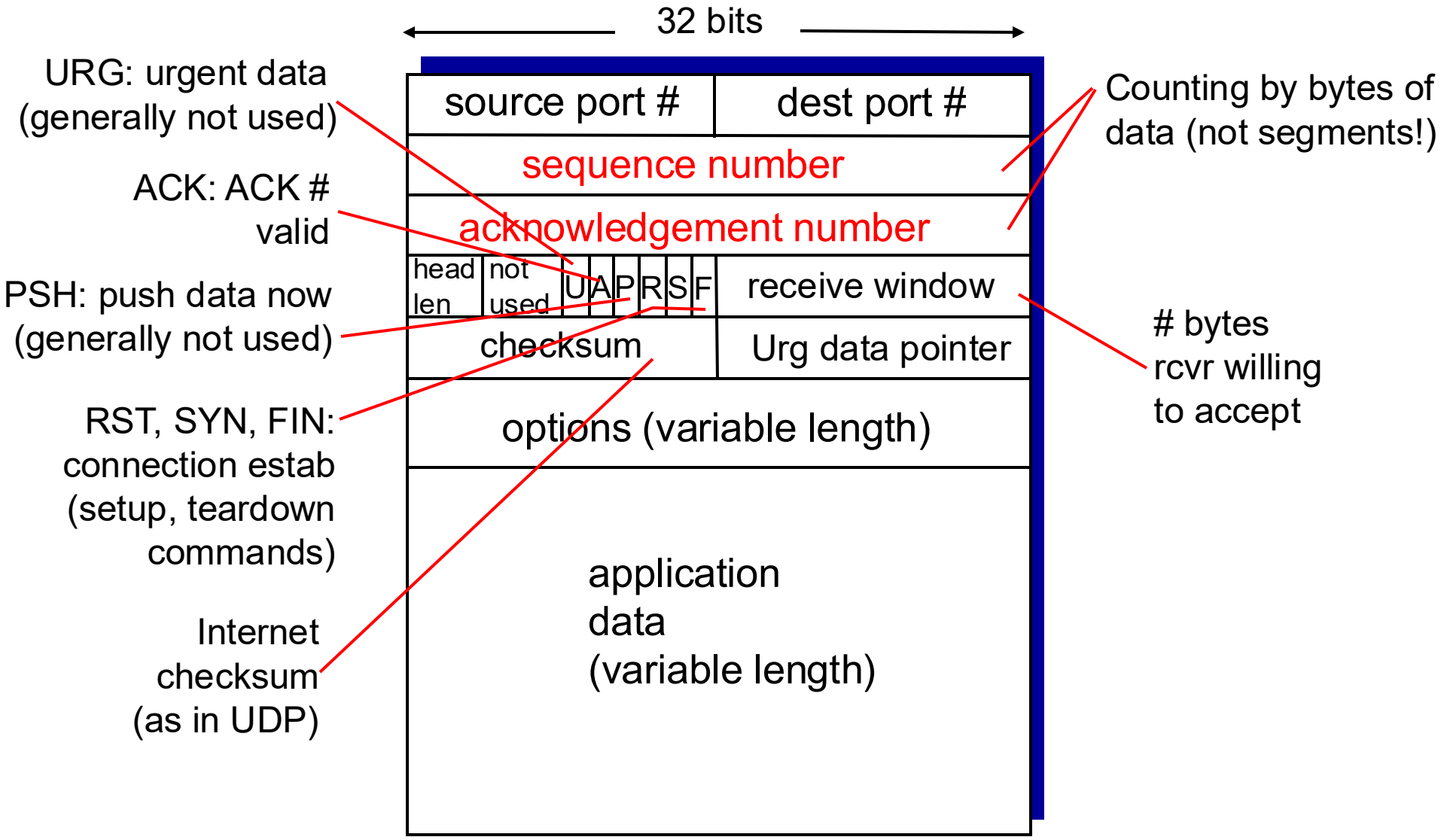


vs UDP structure



UDP segment format

TCP segment structure



TCP sequence numbers, ACKs

Sequence numbers:

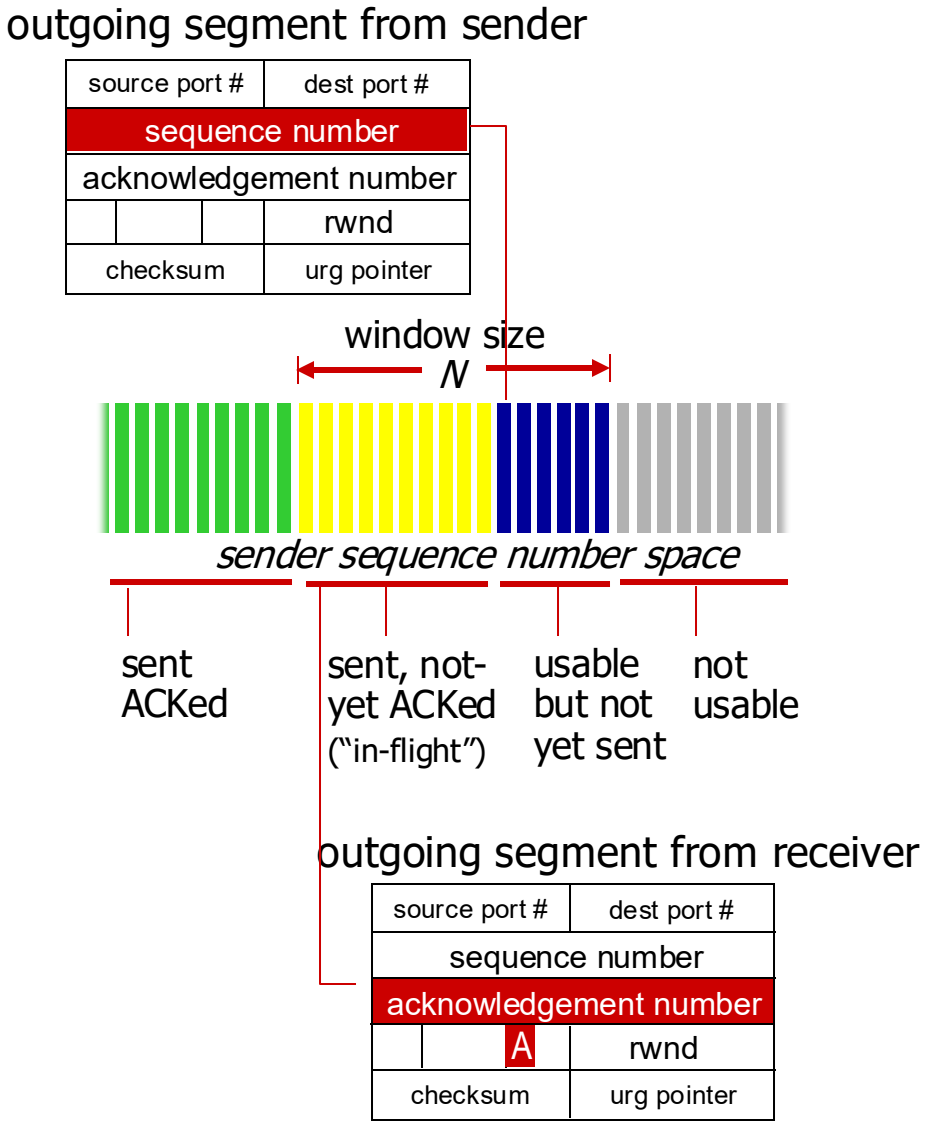
- byte stream "number" of first byte in segment's data

Acknowledgements:

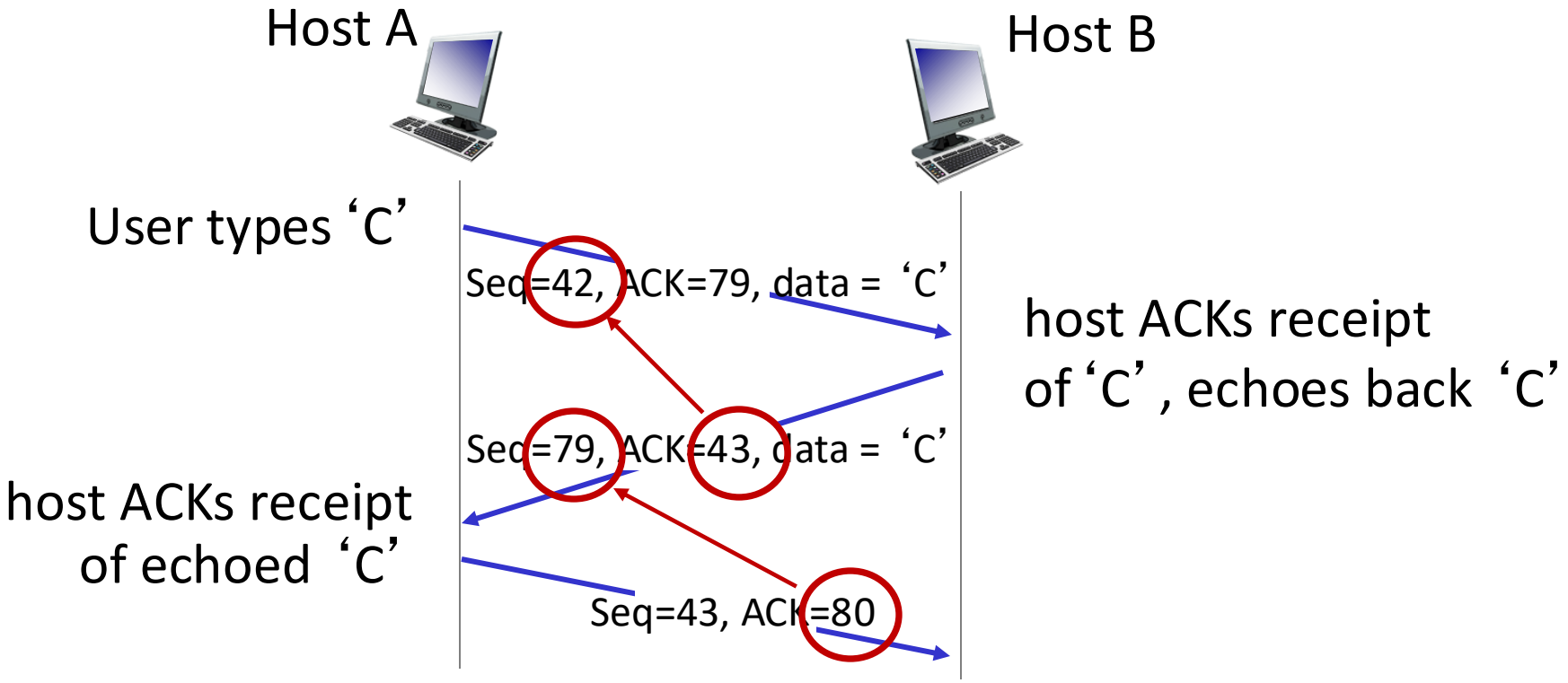
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

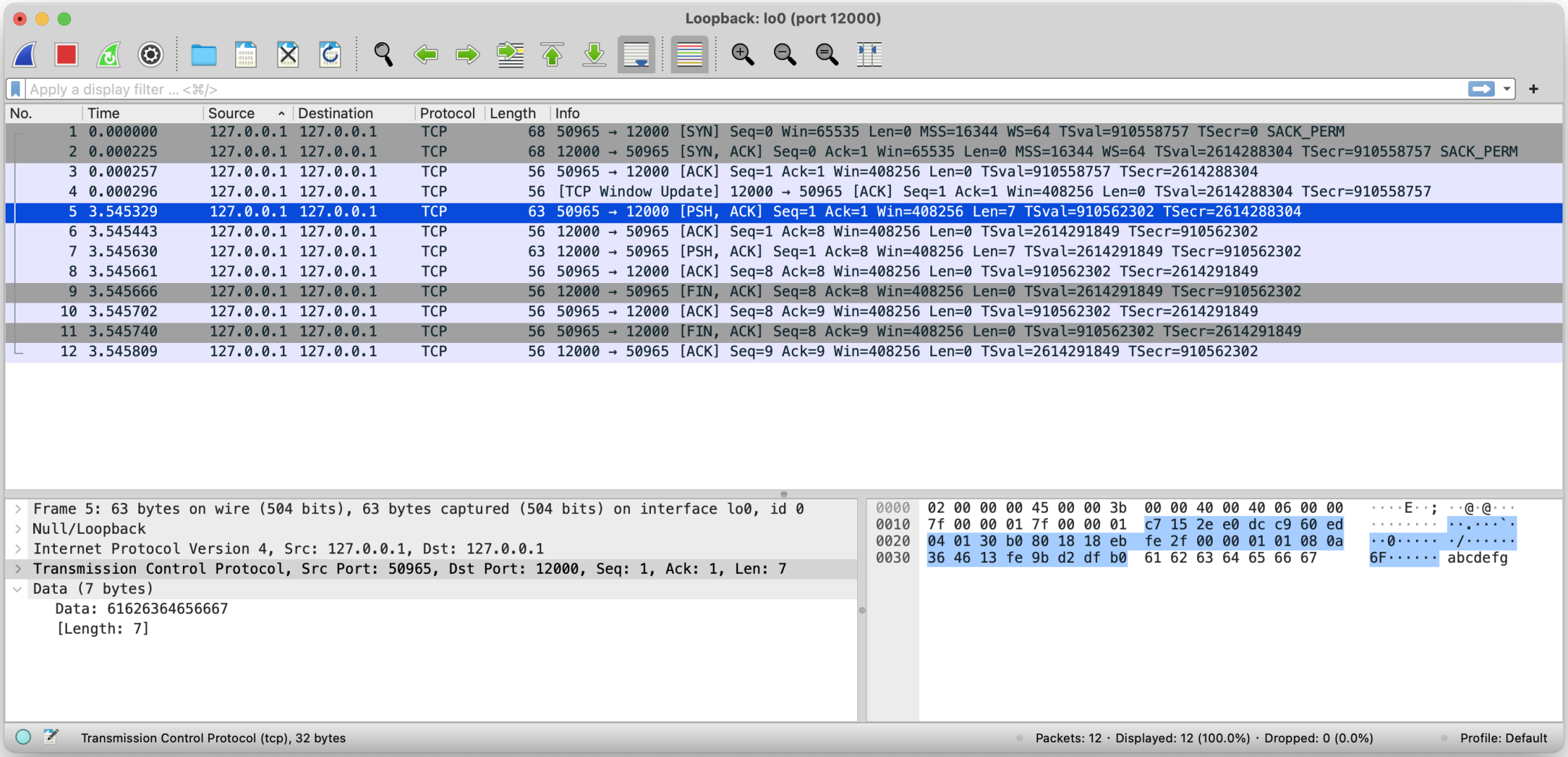
- *A:* TCP spec doesn't say, - up to implementor



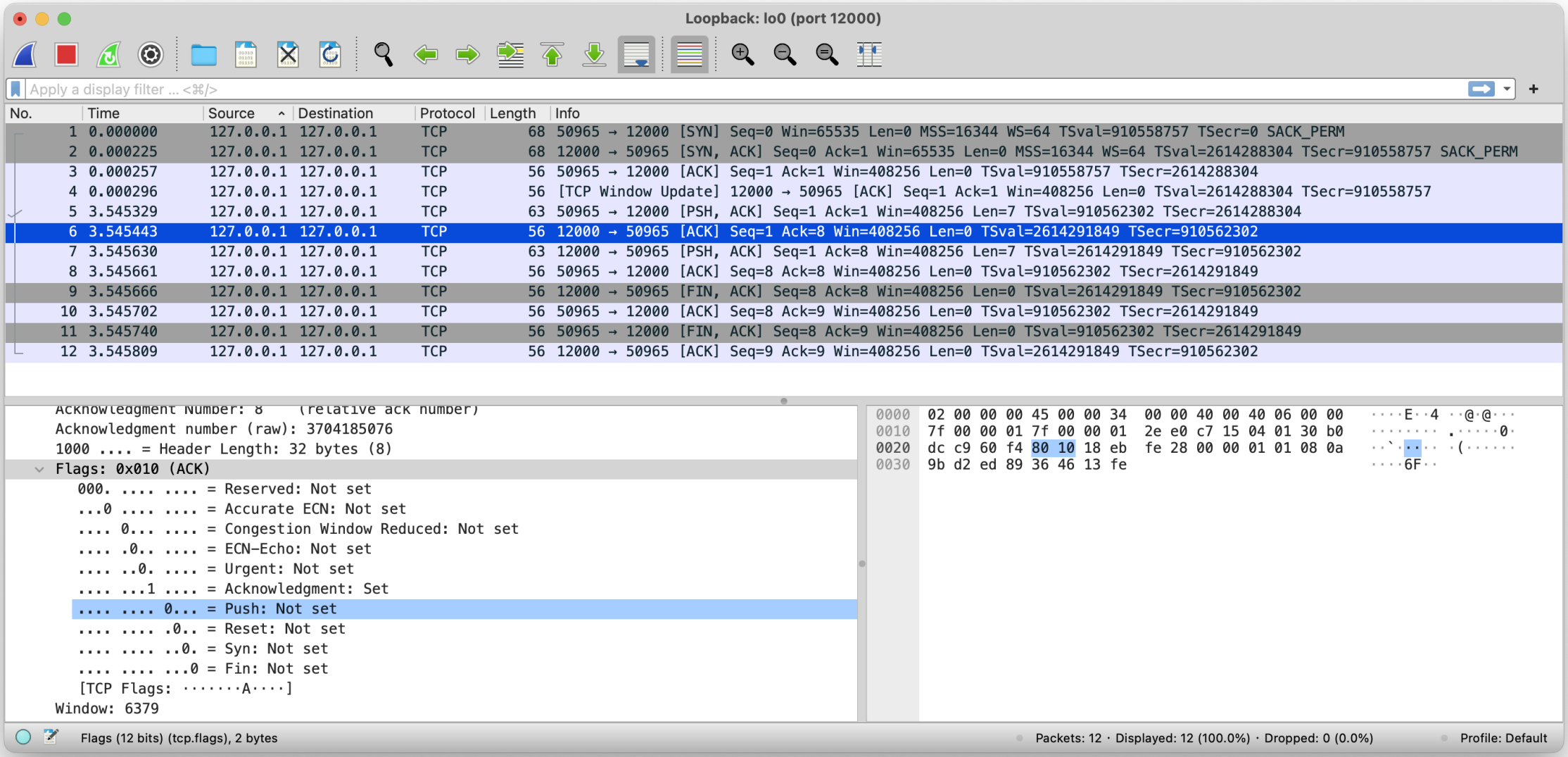
TCP sequence numbers, ACKs



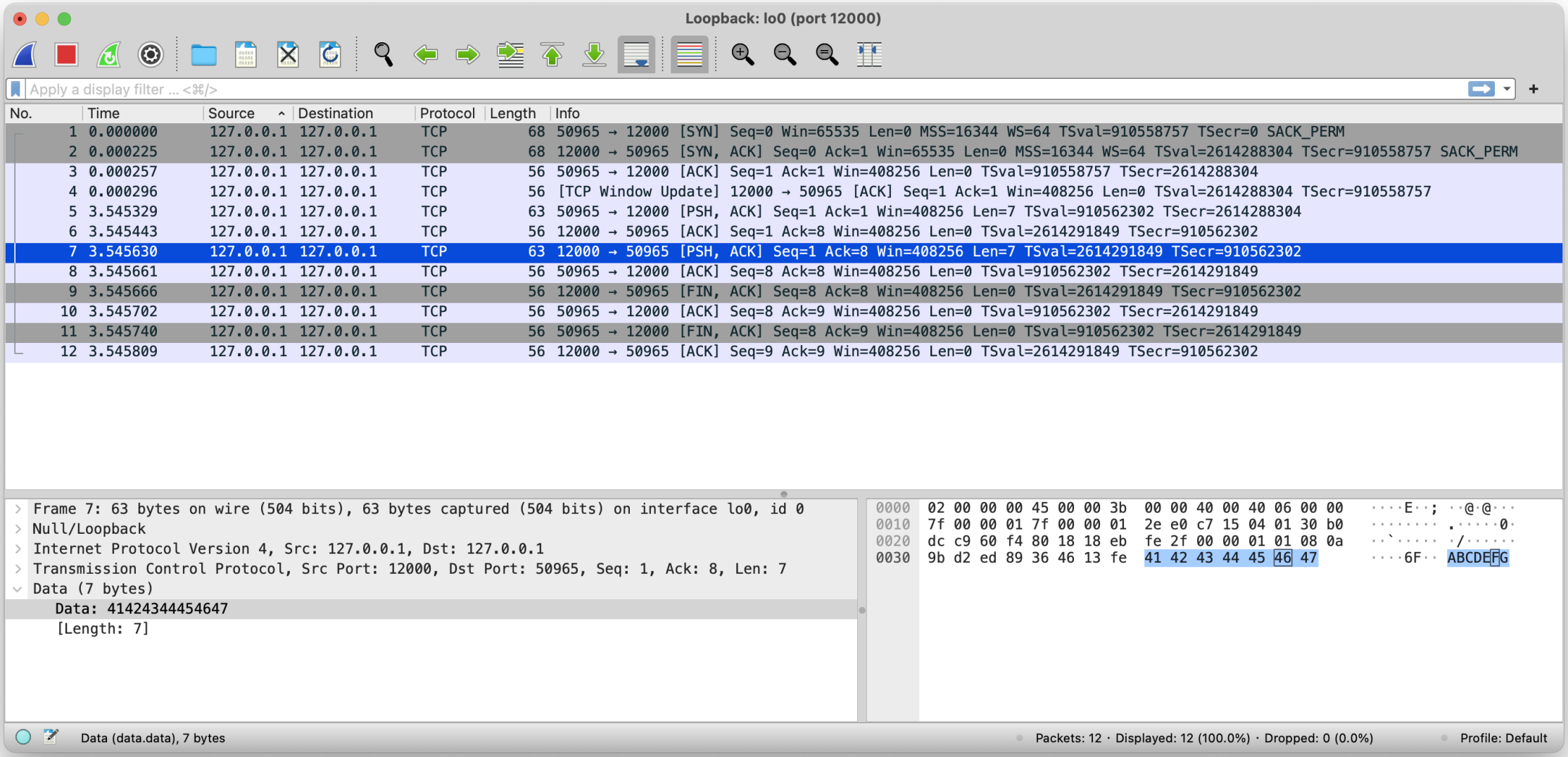
simple telnet scenario



Send from client



ACK from server



Send from server

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**

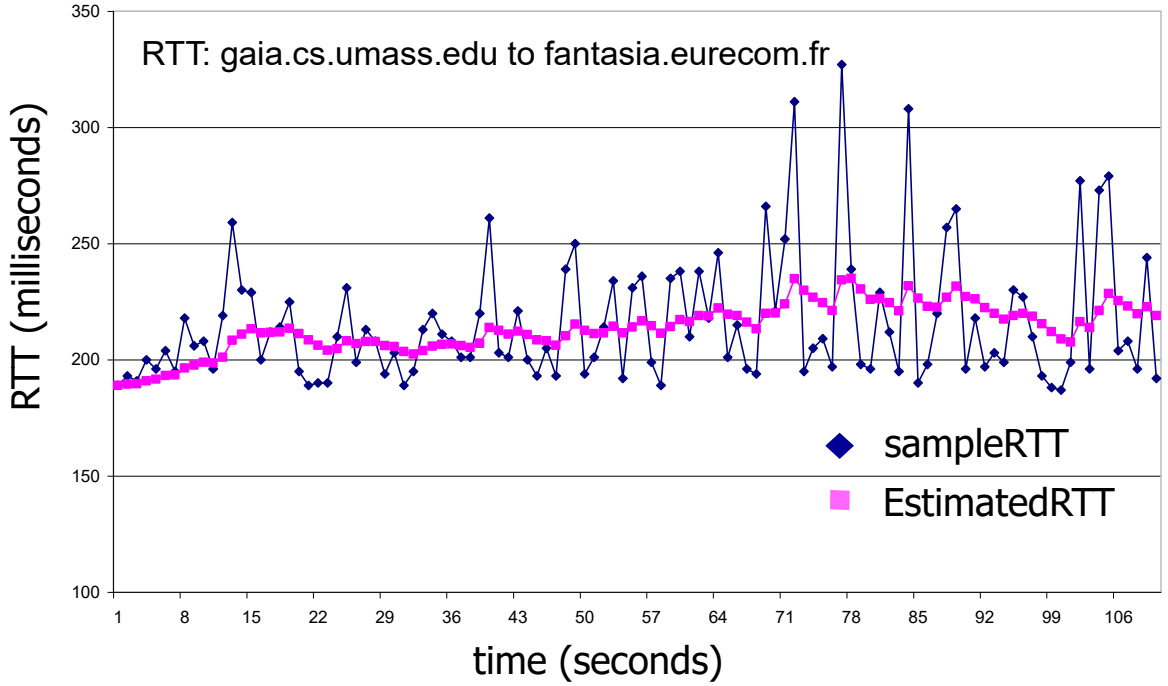
Q: How to estimate RTT?

- **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

$$\text{EstimatedRTT}_n = (1-\alpha)*\text{EstimatedRTT}_{n-1} + \alpha*\text{SampleRTT}$$

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$



TCP round trip time, timeout

- Timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT ->** larger safety margin
- Estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

"safety margin"

(Dev = Deviation)

TCP reliable data transfer

- **TCP creates rdt service on top of IP's unreliable service**
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- **Retransmissions triggered by:**
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

Data rcvd from app:

- **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`

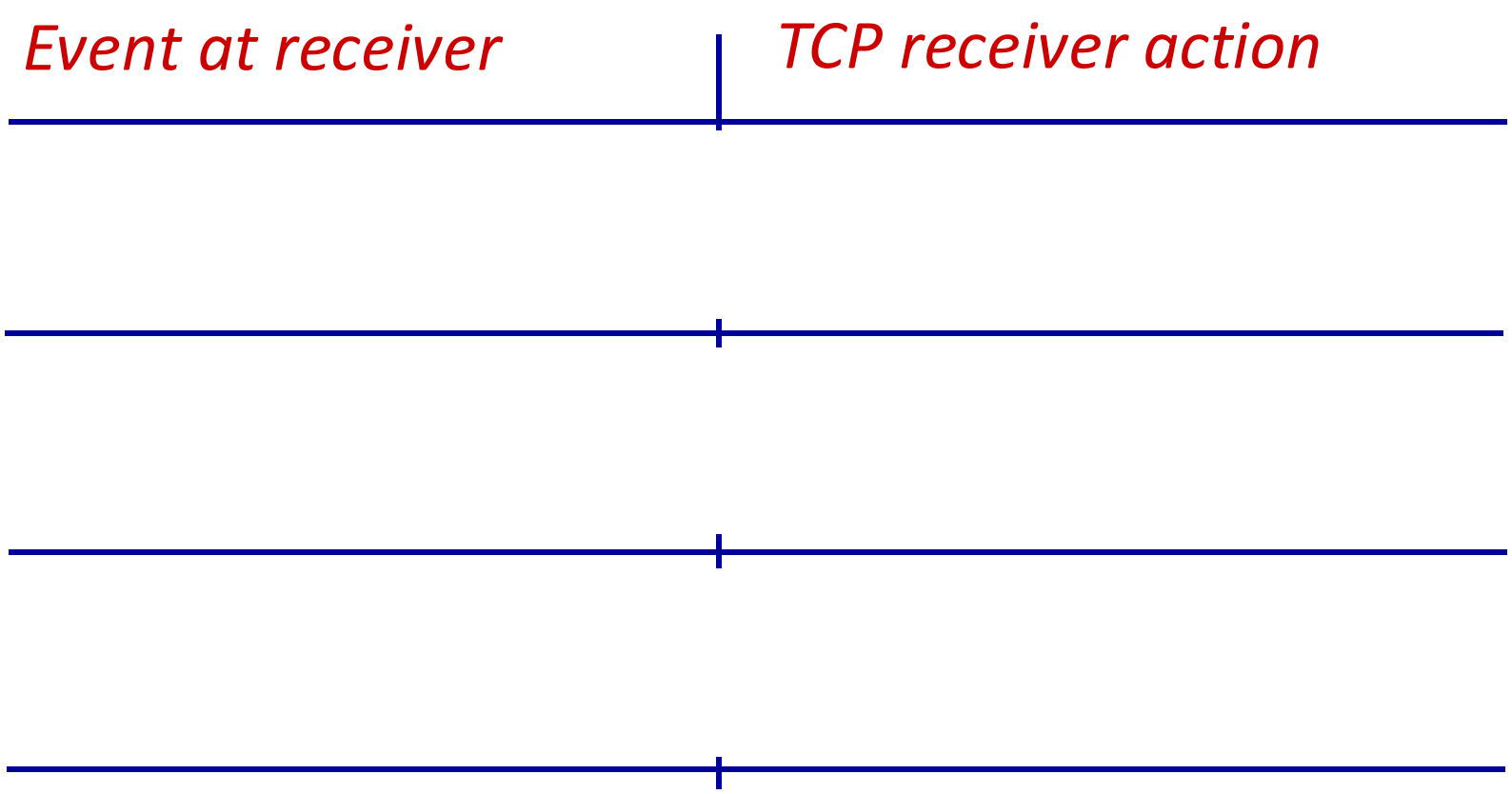
Timeout:

- **Retransmit segment that caused timeout**
- **Restart timer**

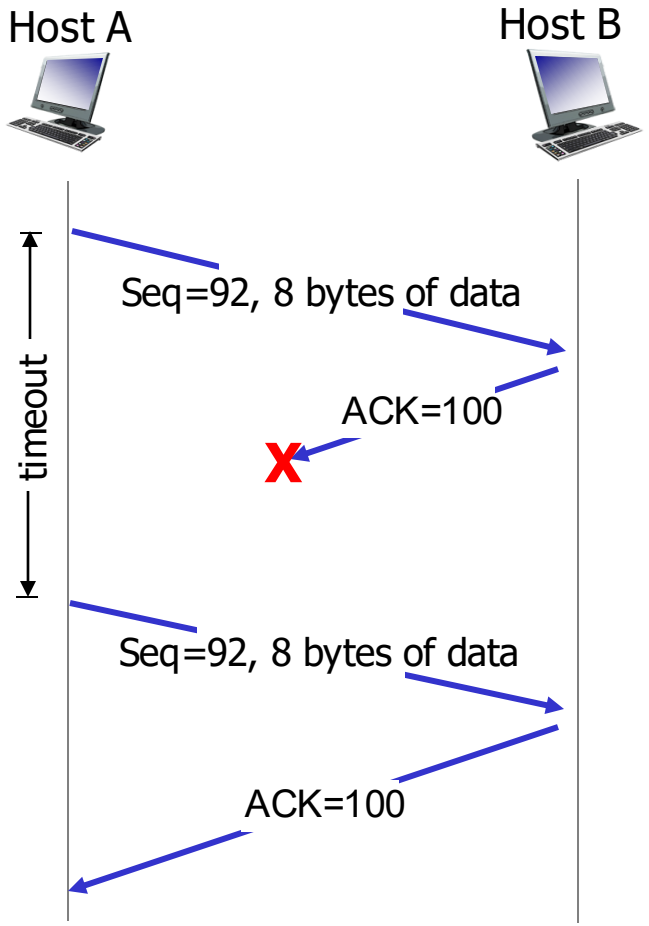
Ack rcvd:

- **If ack acknowledges previously unacked segments**
  - Update what is known to be ACKed
  - Start timer if there are still unacked segments

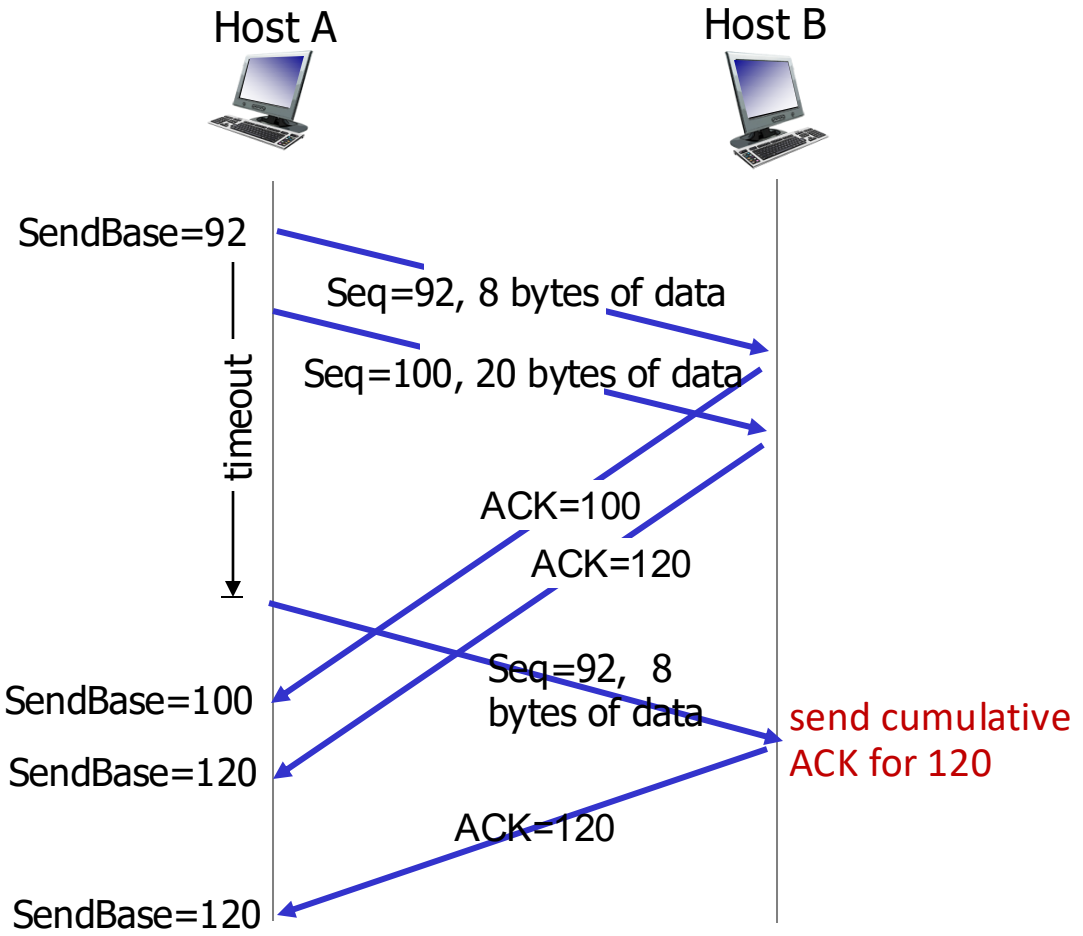
TCP Receiver: ACK generation [RFC 5681]



TCP: retransmission scenarios

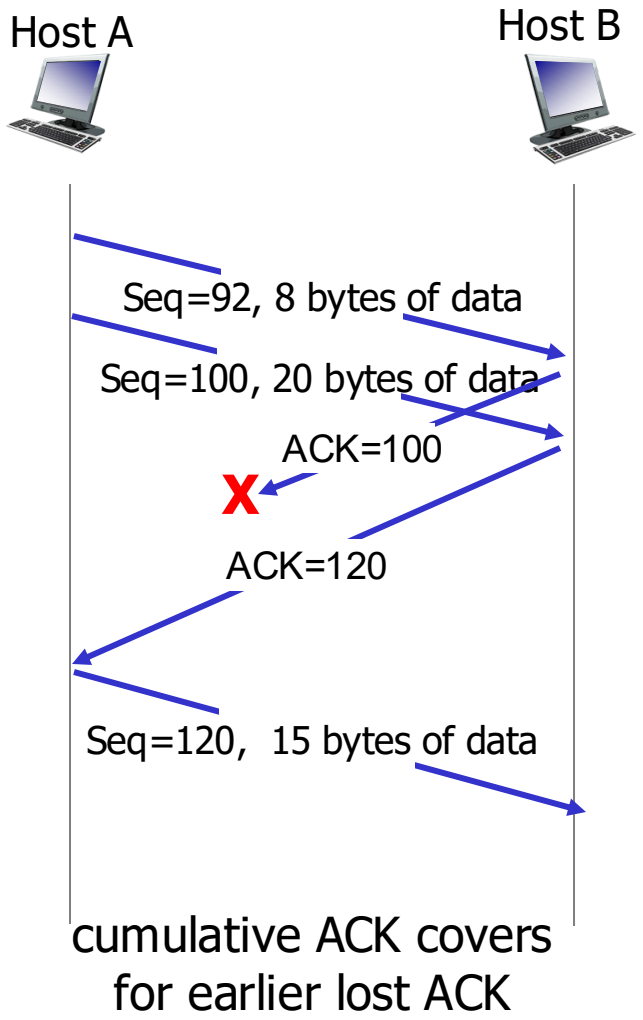


lost ACK scenario



premature timeout

TCP: retransmission scenarios



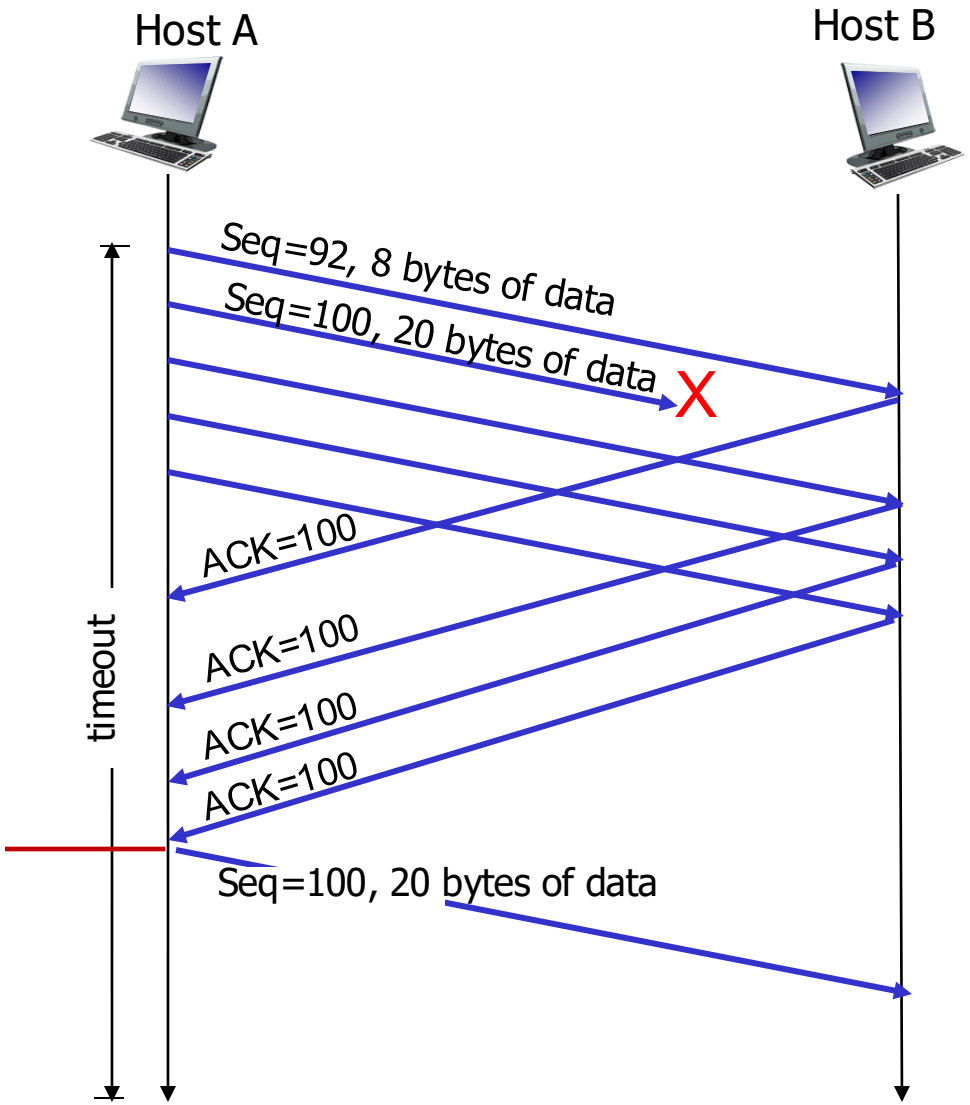
TCP fast retransmit

TCP fast retransmit

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

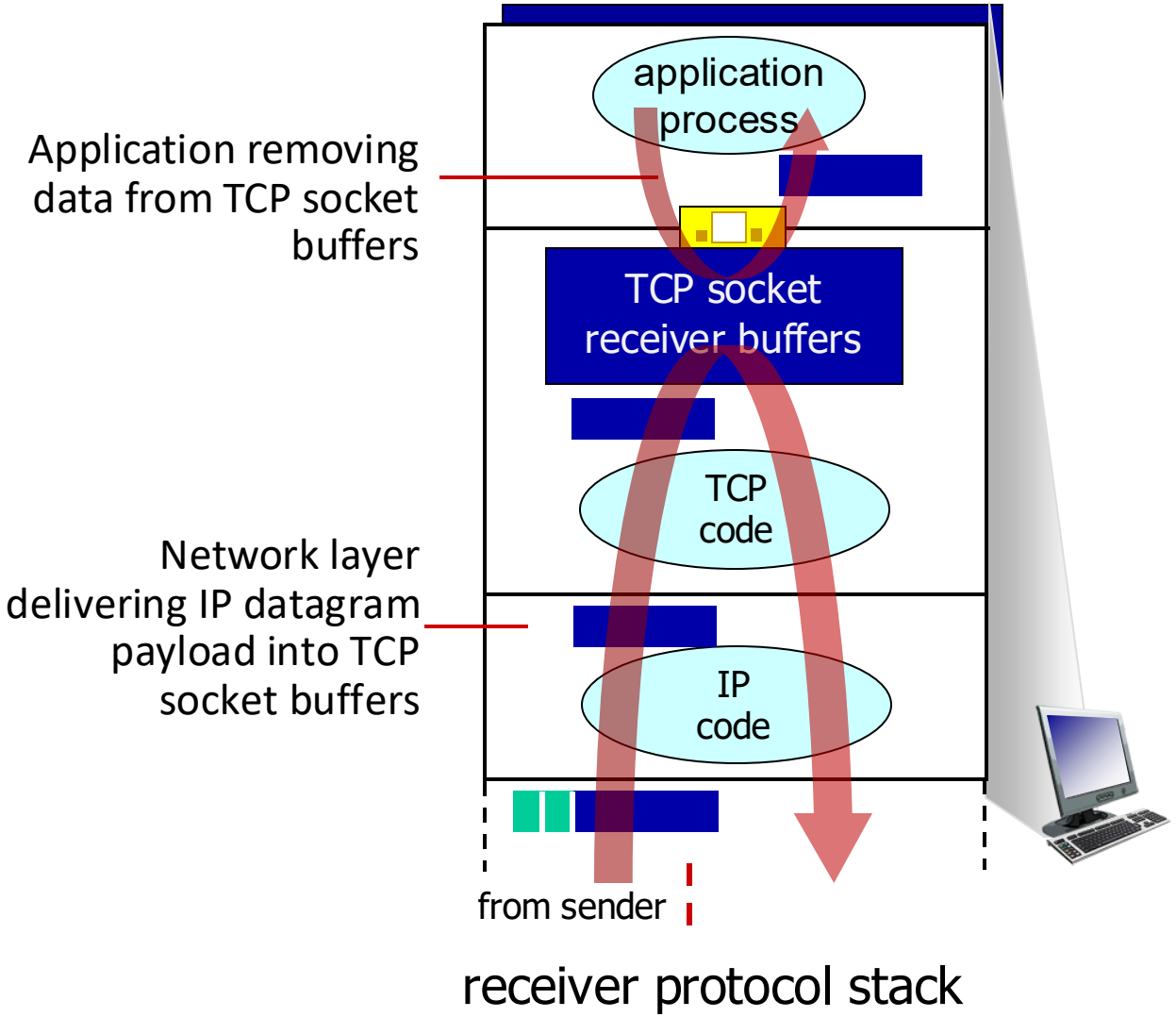
- likely that unACKed segment lost, so don't wait for timeout

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



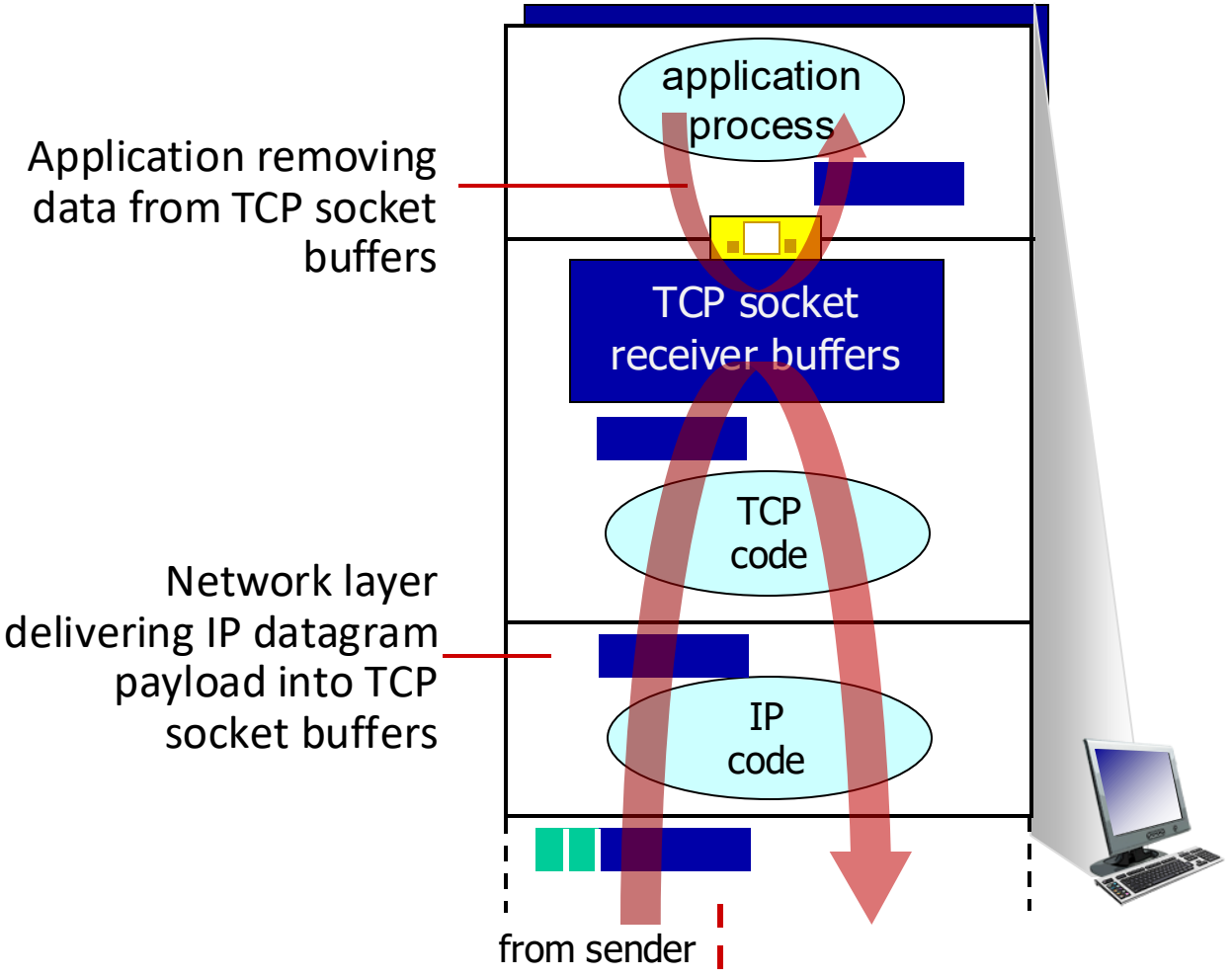
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

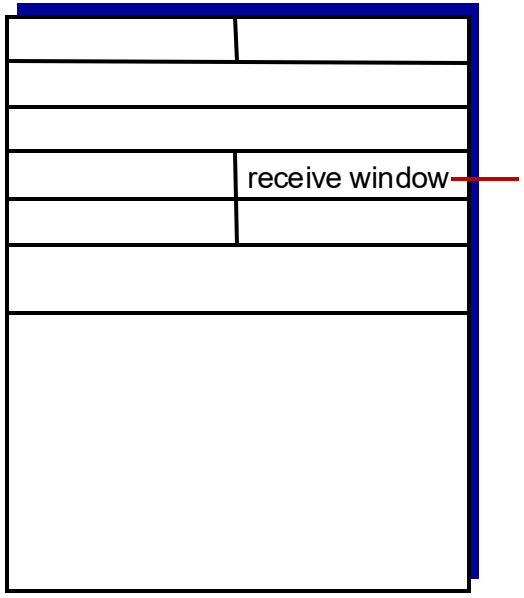
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

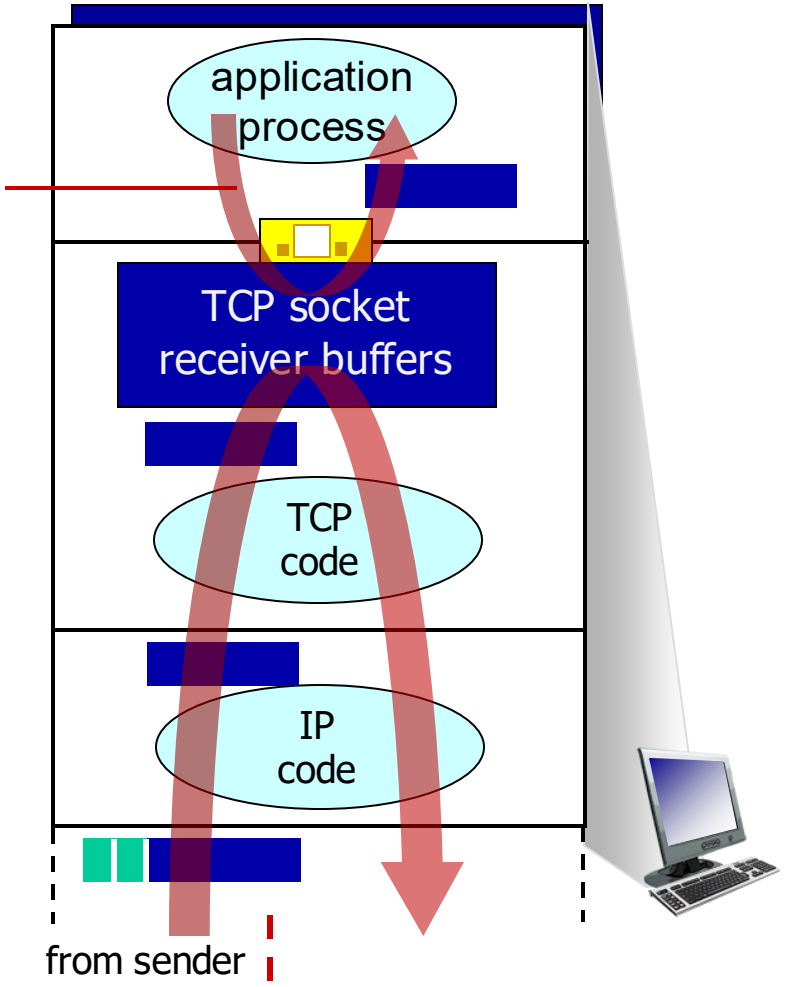
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



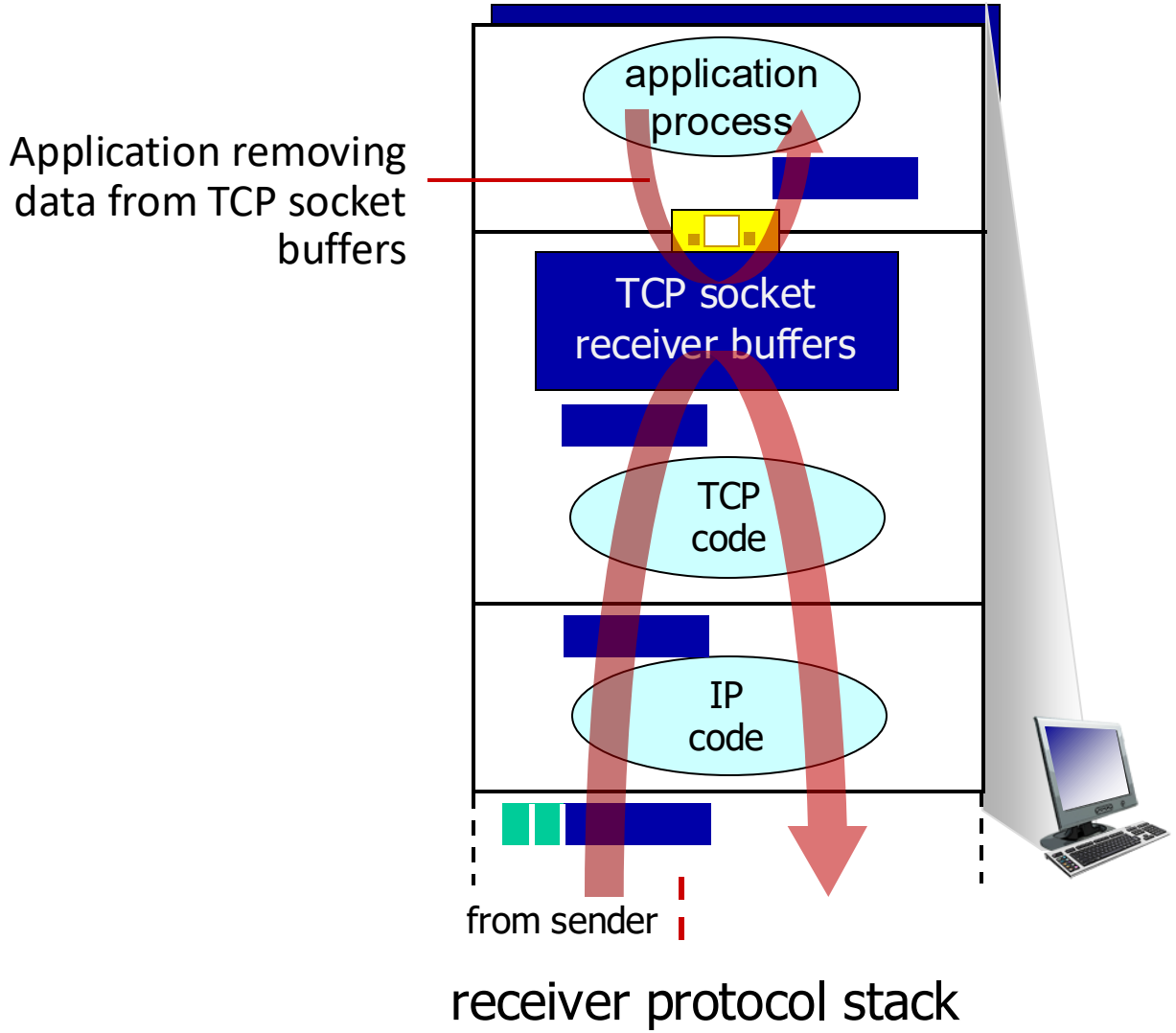
receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

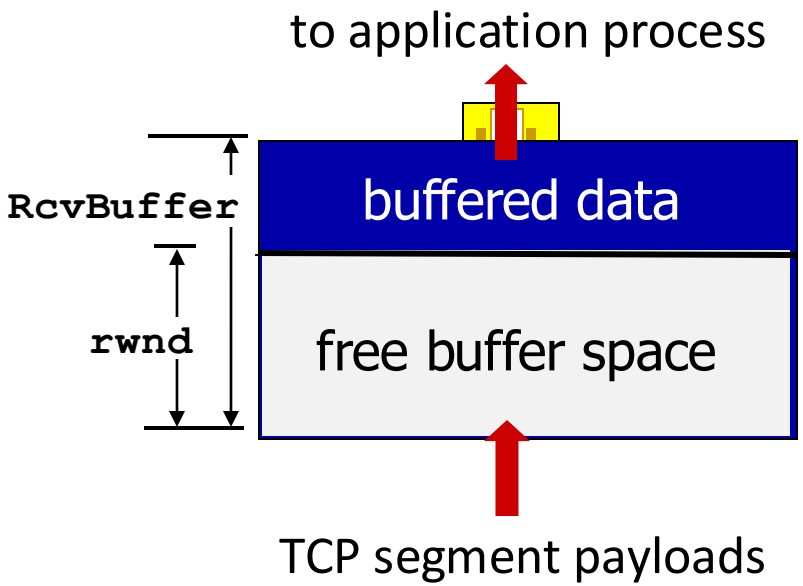
flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



TCP flow control

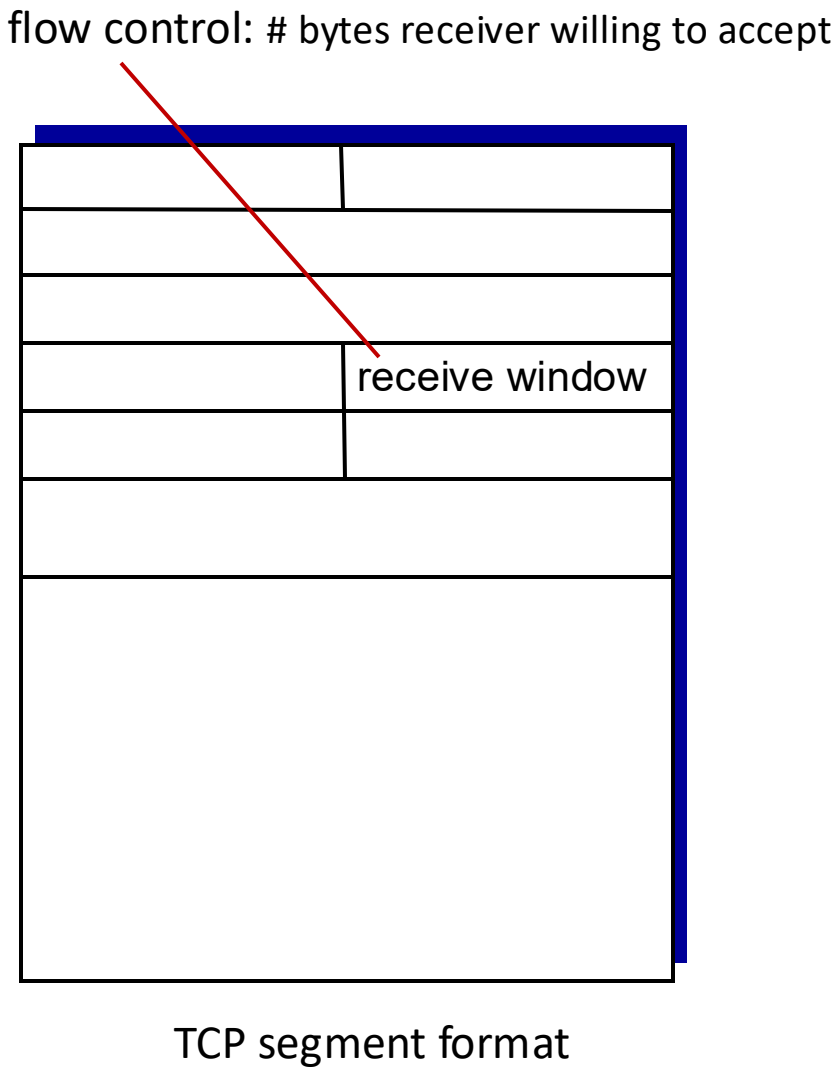
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow

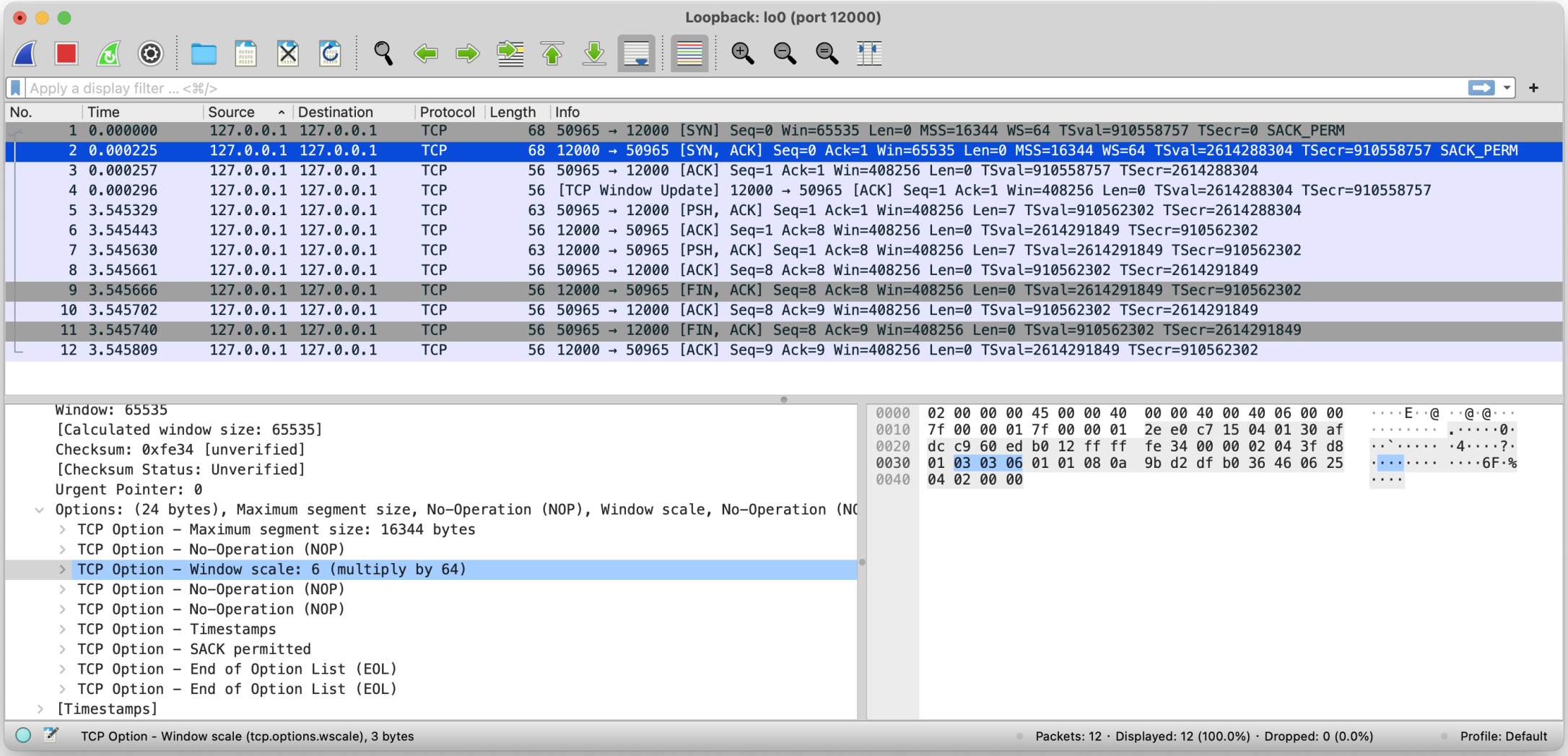


TCP receiver-side buffering

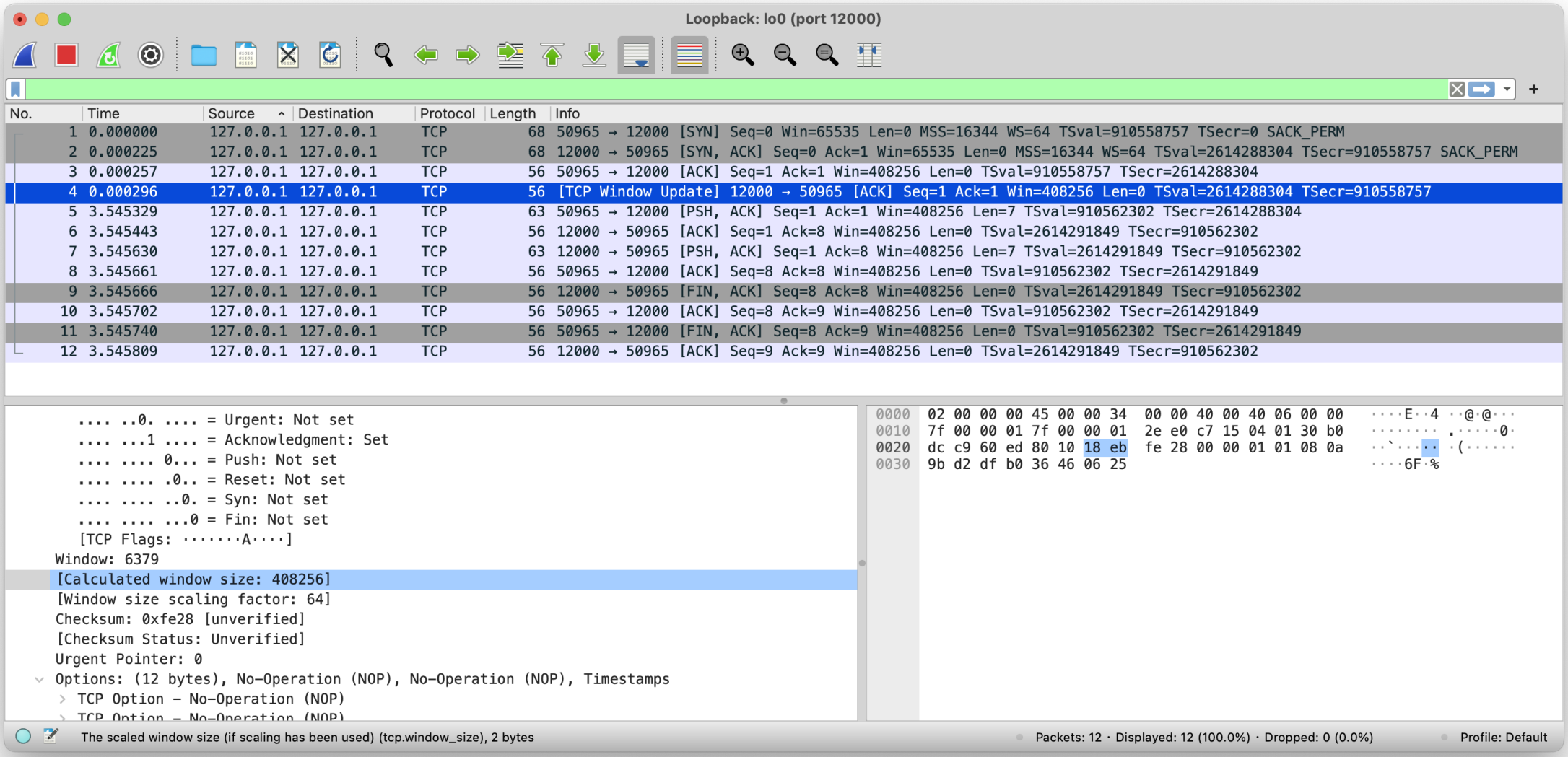
TCP flow control

- TCP receiver “advertises” free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed (“in-flight”) data to received **`rwnd`**
- guarantees receive buffer will not overflow





SYN: Window Scale factor

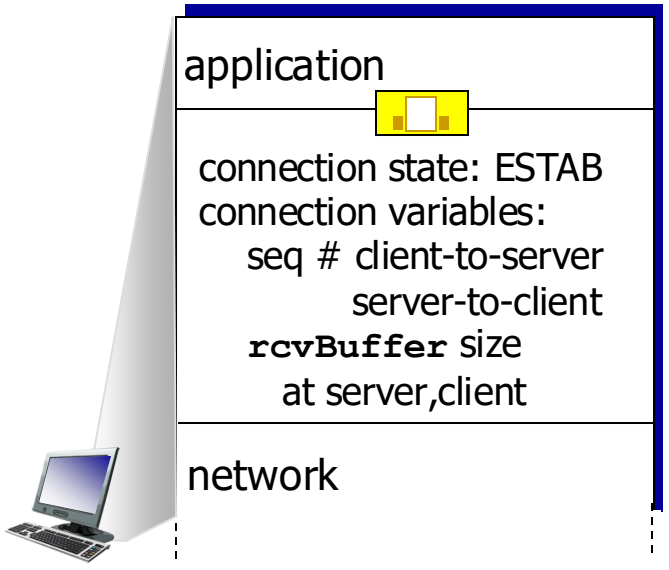


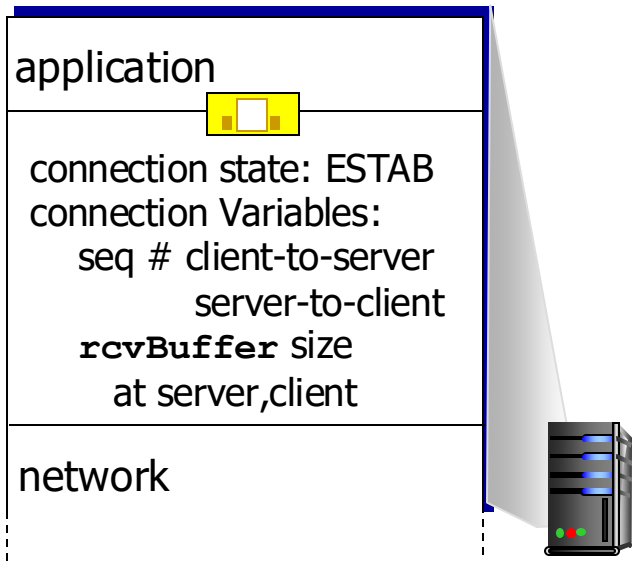
Calculated Window Size = 6379 x 64 = 408256

Connection Management

Before exchanging data, sender/receiver "handshake":

- Agree to establish connection (each knowing the other willing to establish connection) Agree on connection parameters



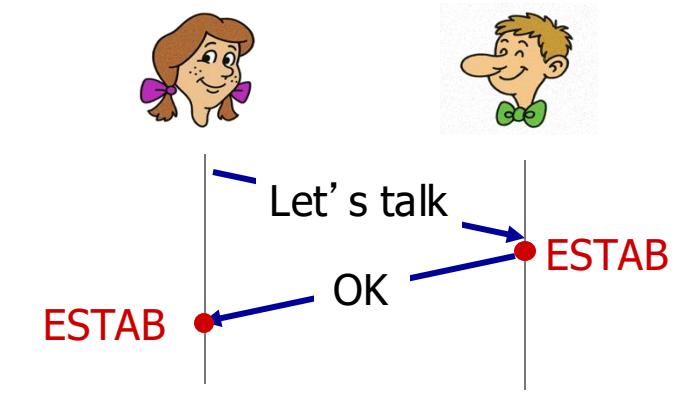


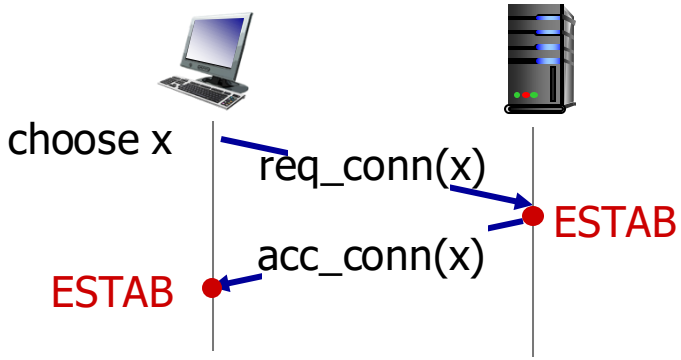

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:

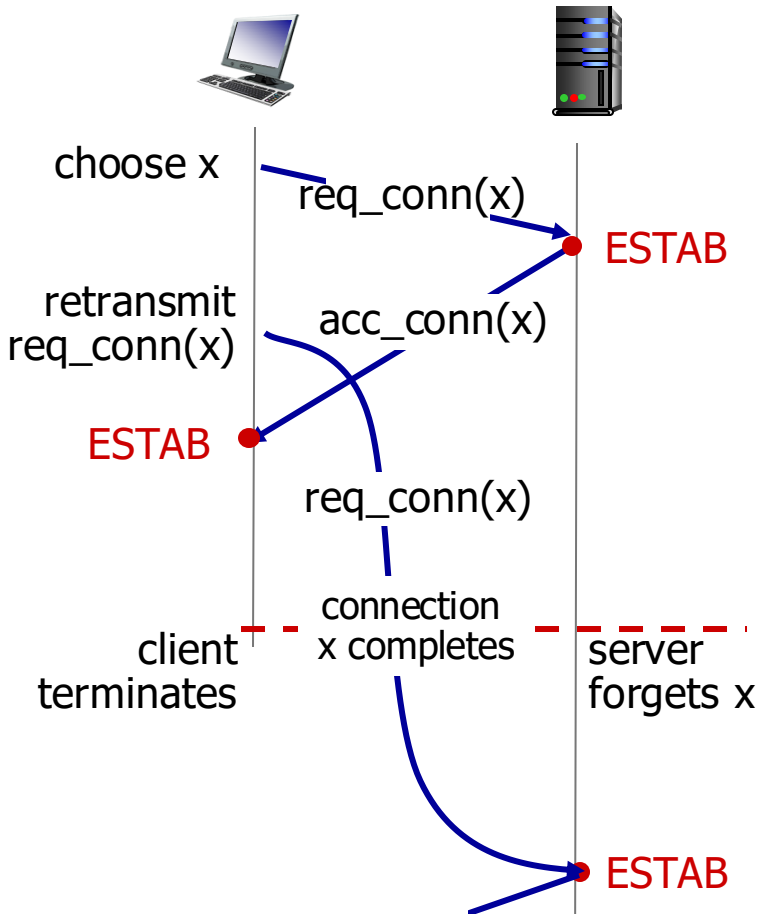




Q: **Will 2-way handshake always work in network?**

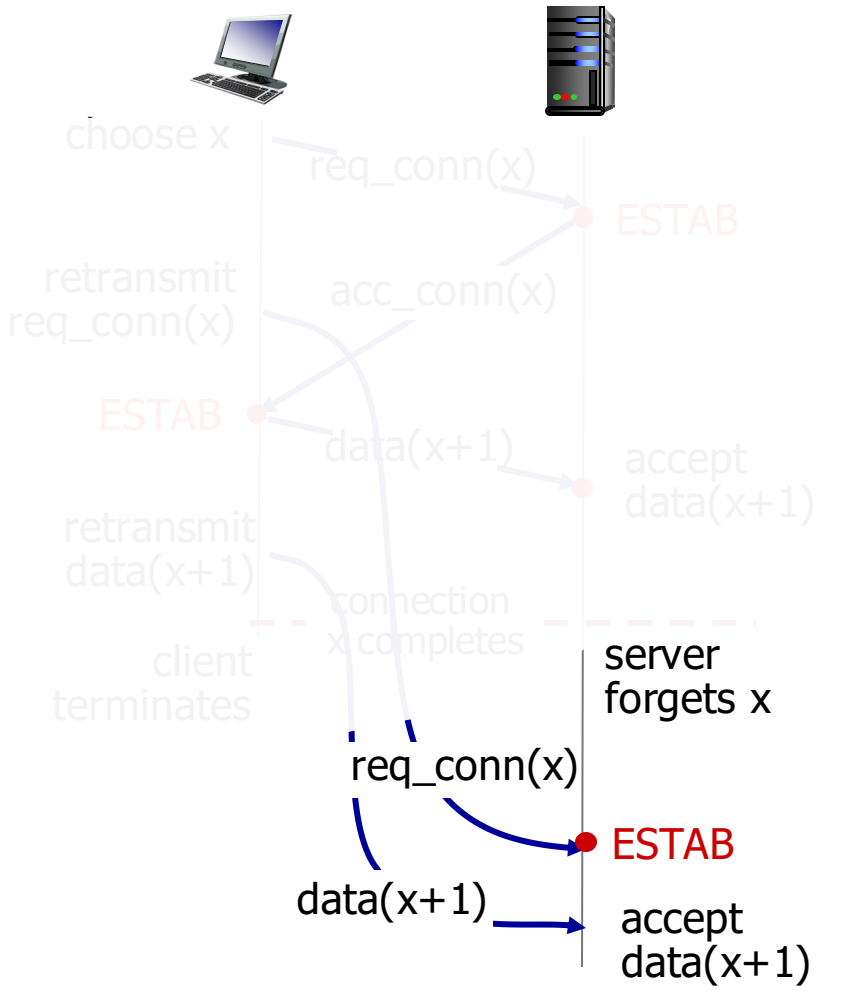
- Variable delays
- Retransmitted messages (e.g. req_conn(x)) due to message loss
- Message reordering
- Can't "see" other side

2-way handshake scenarios



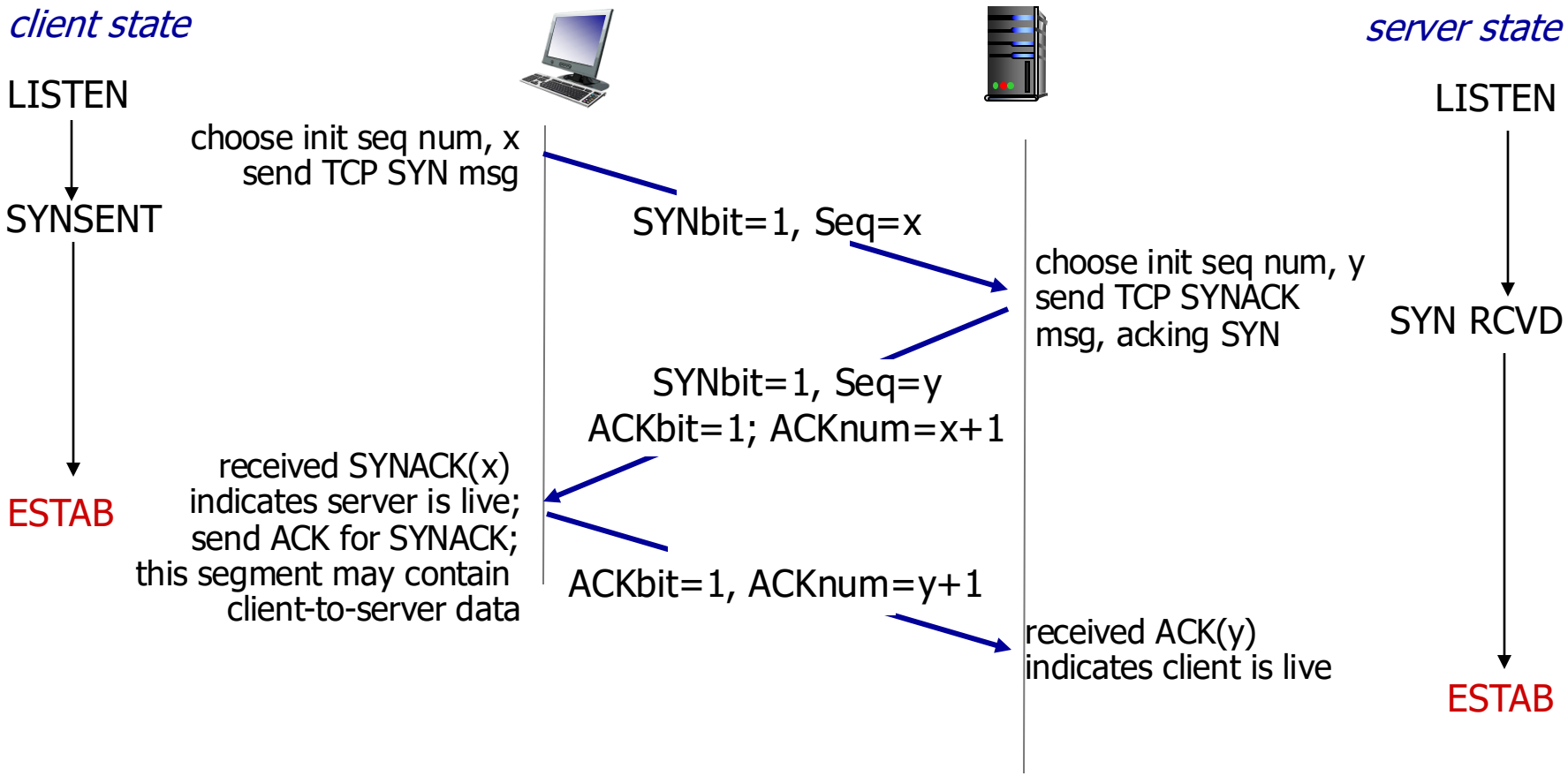
Problem: half open connection! (no client)

2-way handshake scenarios



Problem: dup data accepted!

TCP 3-way handshake



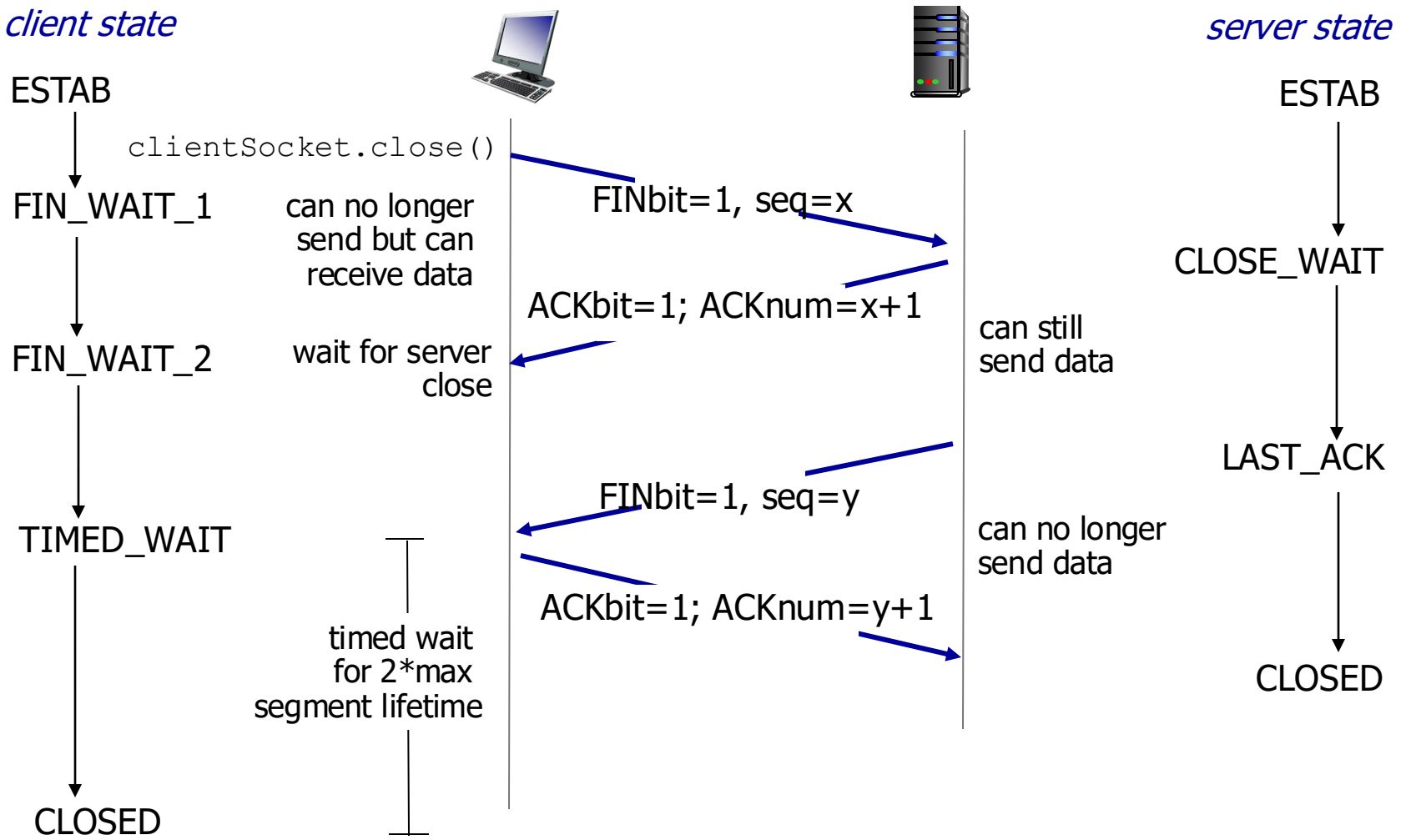
3-way handshake

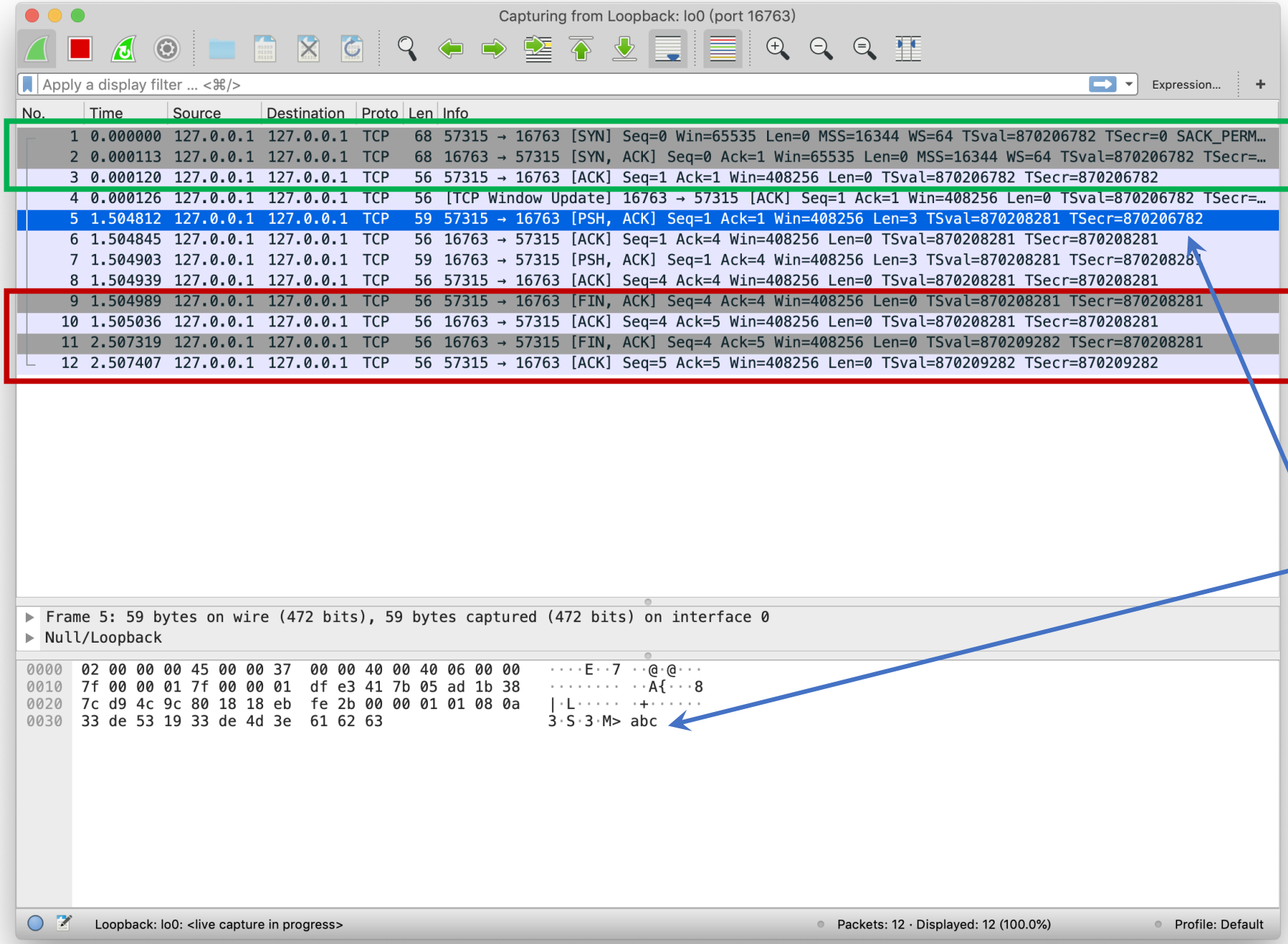
```
52859 → 12000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1392544355 TSecr=0 SACK_PERM
12000 → 52859 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=3908359660 TSecr=1392544355 SACK_PERM
52859 → 12000 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1392544355 TSecr=3908359660
[TCP Window Update] 12000 → 52859 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=3908359660 TSecr=1392544355
52859 → 12000 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=3 TSval=1392553900 TSecr=3908359660
12000 → 52859 [ACK] Seq=1 Ack=4 Win=408256 Len=0 TSval=3908369205 TSecr=1392553900
12000 → 52859 [PSH, ACK] Seq=1 Ack=4 Win=408256 Len=13 TSval=3908369205 TSecr=1392553900
52859 → 12000 [ACK] Seq=4 Ack=14 Win=408256 Len=0 TSval=1392553900 TSecr=3908369205
12000 → 52859 [FIN, ACK] Seq=14 Ack=4 Win=408256 Len=0 TSval=3908369205 TSecr=1392553900
52859 → 12000 [ACK] Seq=4 Ack=15 Win=408256 Len=0 TSval=1392553900 TSecr=3908369205
52859 → 12000 [FIN, ACK] Seq=4 Ack=15 Win=408256 Len=0 TSval=1392553900 TSecr=3908369205
12000 → 52859 [ACK] Seq=15 Ack=5 Win=408256 Len=0 TSval=3908369205 TSecr=1392553900
```

TCP: closing a connection

- **Client, server each close their side of connection**
  - Send TCP segment with FIN bit = 1
- **Respond to received FIN with ACK**
  - On receiving FIN, ACK can be combined with own FIN
- **Simultaneous FIN exchanges can be handled**

TCP: closing a connection





3-way for connection

4-way for closing a connection

Data from client to server

Thanks.