Introduction to Networking

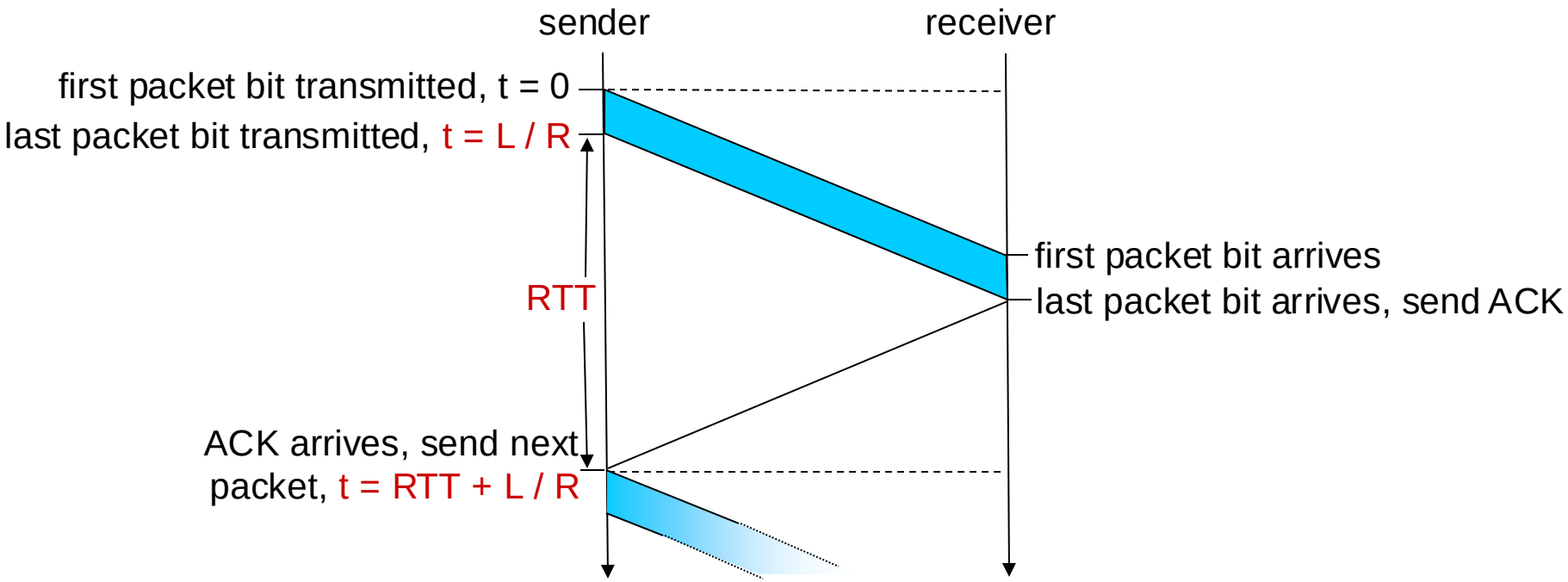
CAN201 – Week 5

Module Leader: Dr. Wenjun Fan & Dr. Fei Cheng

Lecture 5 – Transport Layer (2)

- **Roadmap**
  1. Pipelined communication
  2. TCP: connection-oriented transport
  3. Principles of congestion control

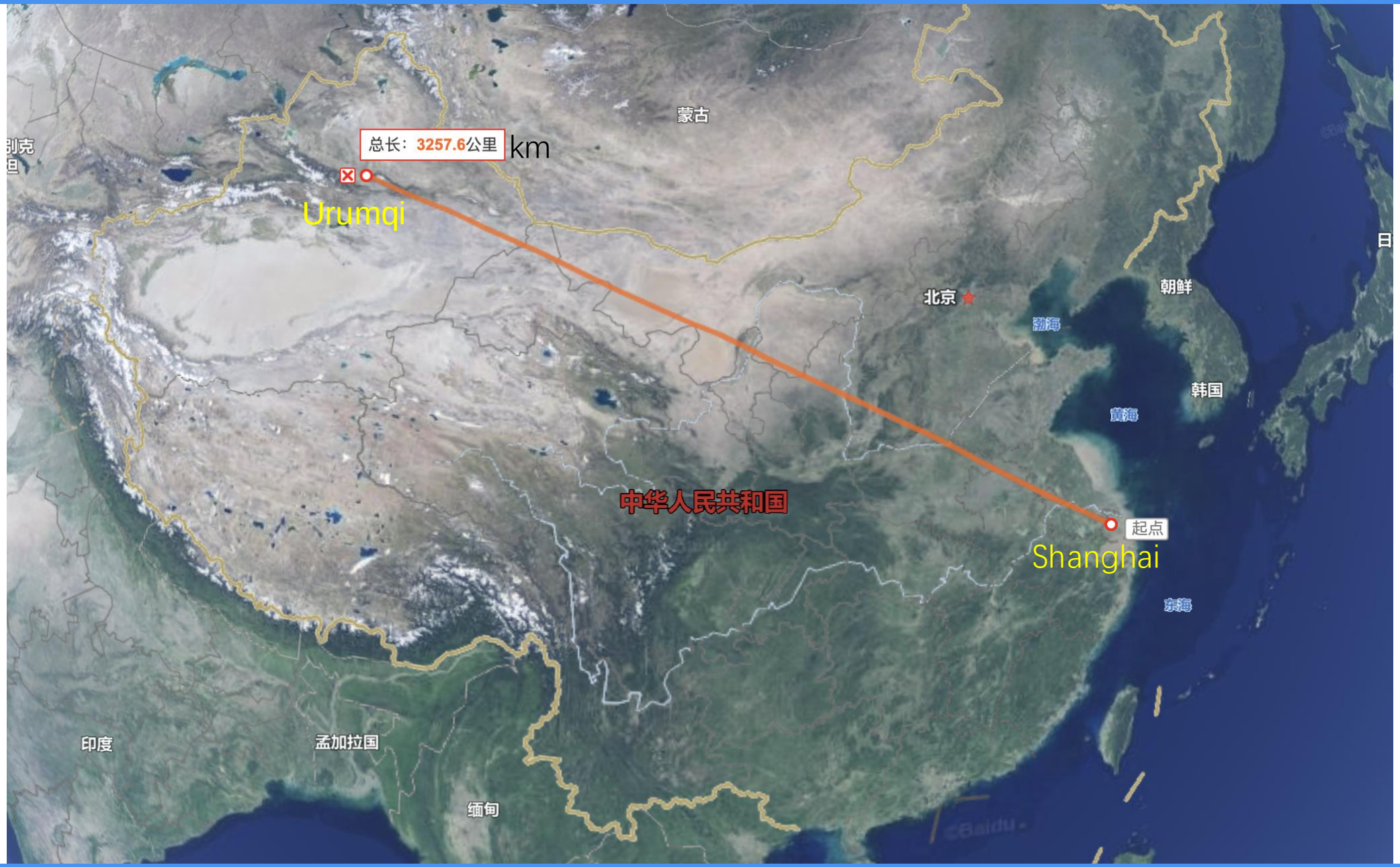
rdt3.0: stop-and-wait operation



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

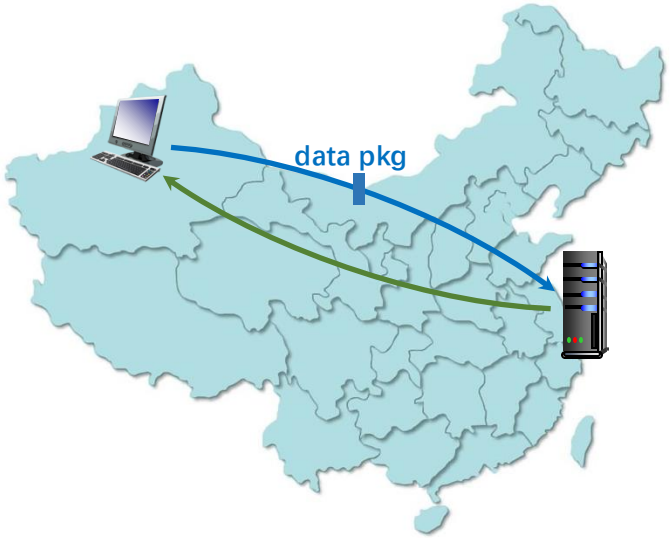
Question

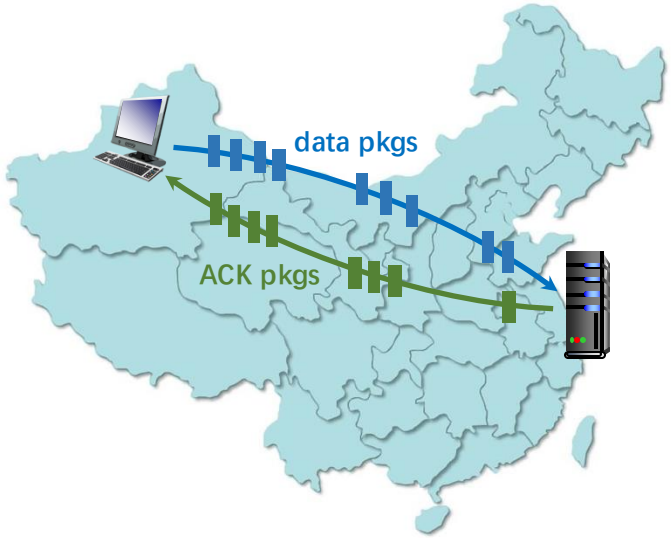
- How to increase utilization?



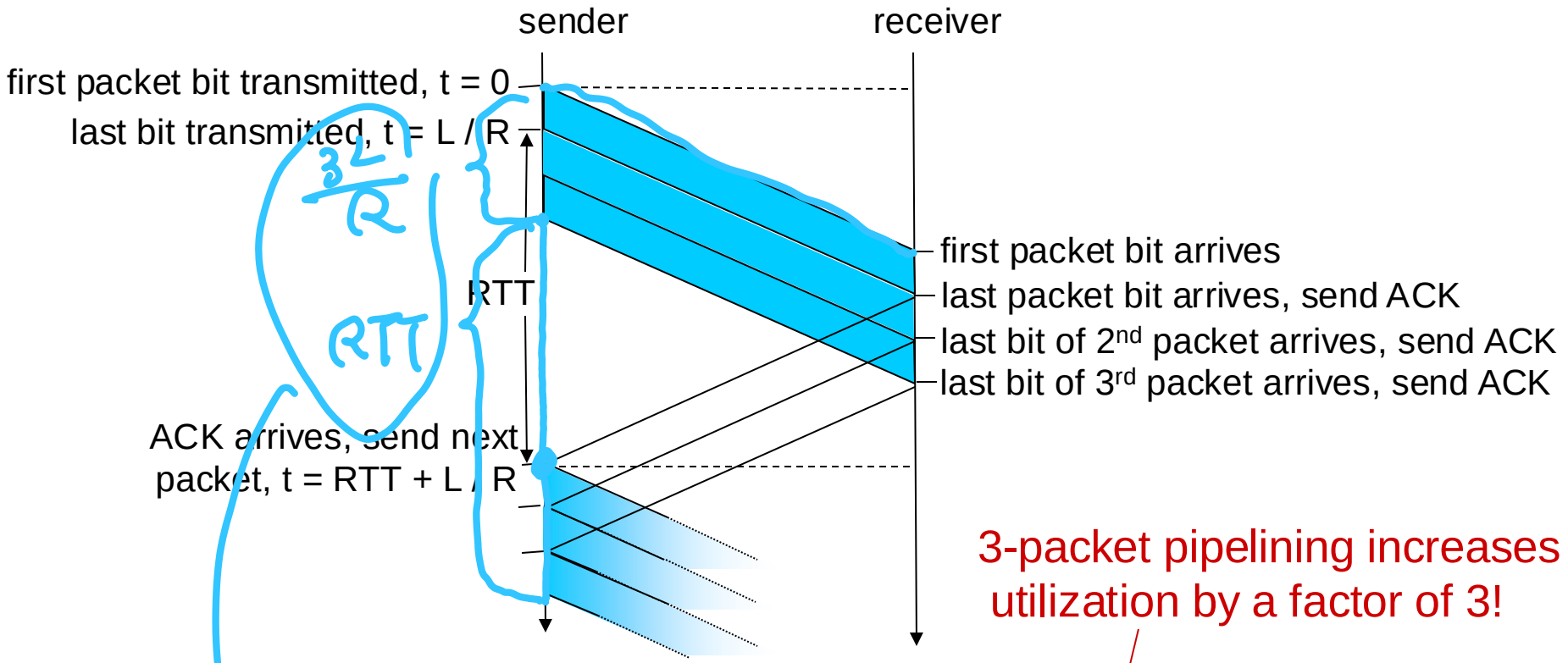
Pipelined protocols 流水线协议

- **Pipelining:** sender allows multiple, "in-flight", yet-to-be-acknowledged pkts 正在进行. 尚未确认的包
  - Range of sequence numbers must be increased
  - Buffering at sender and/or receiver





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

0 ✓
1 ✓
2 ✗

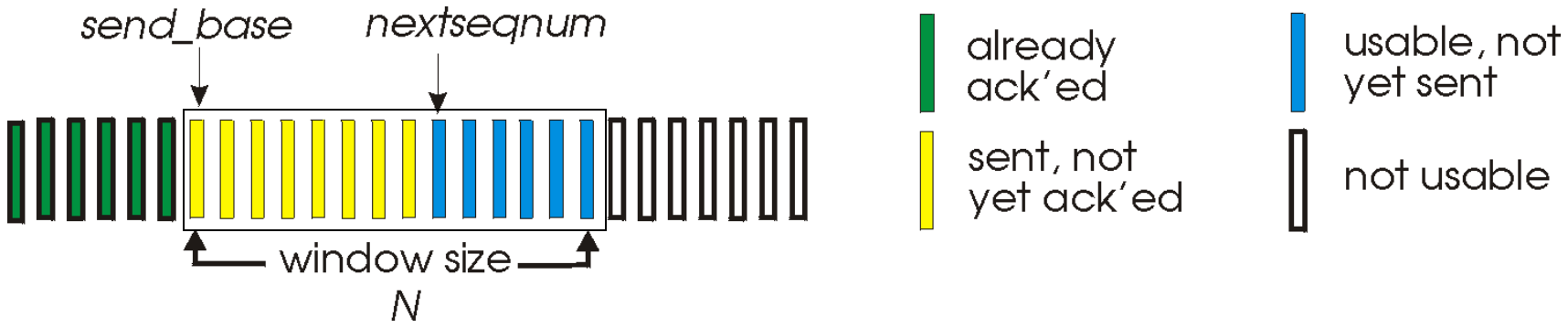
必须接收顺序 send没收到 2 opt, receiver一直

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

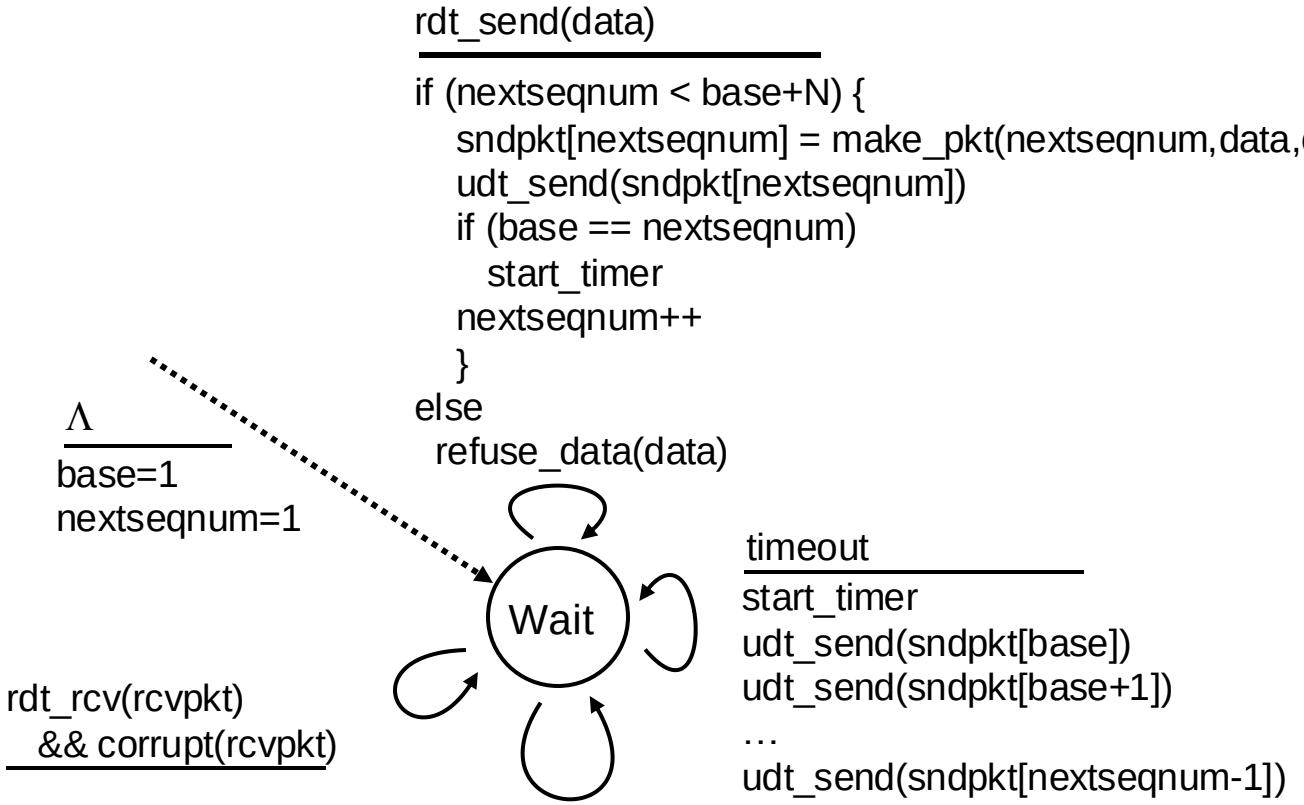
Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unacked pkts allowed

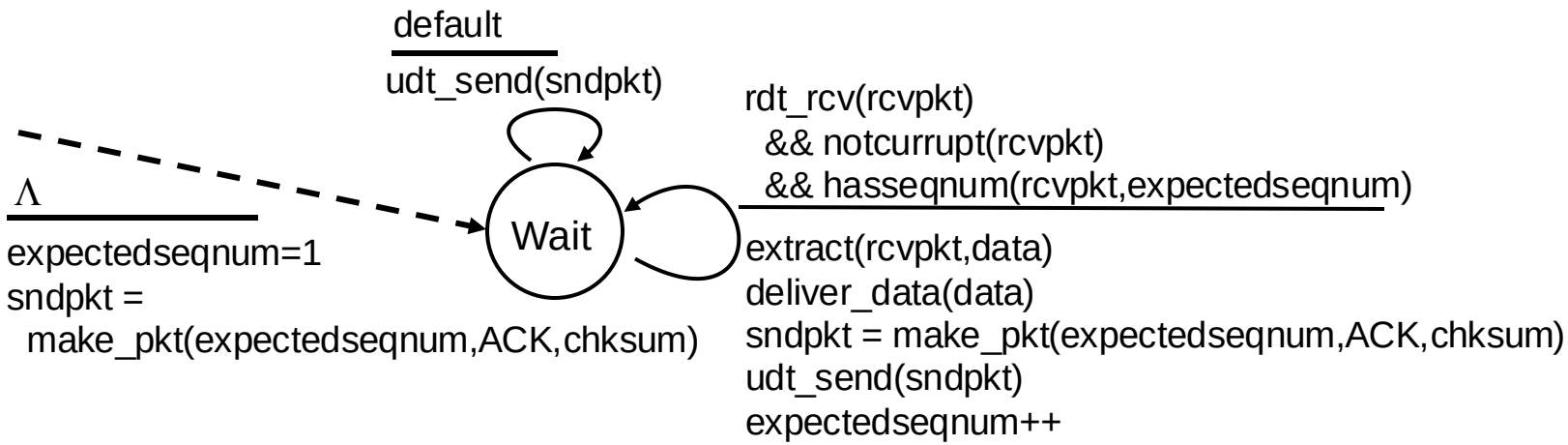


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- Timer for oldest in-flight pkt
- Timeout(n): retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

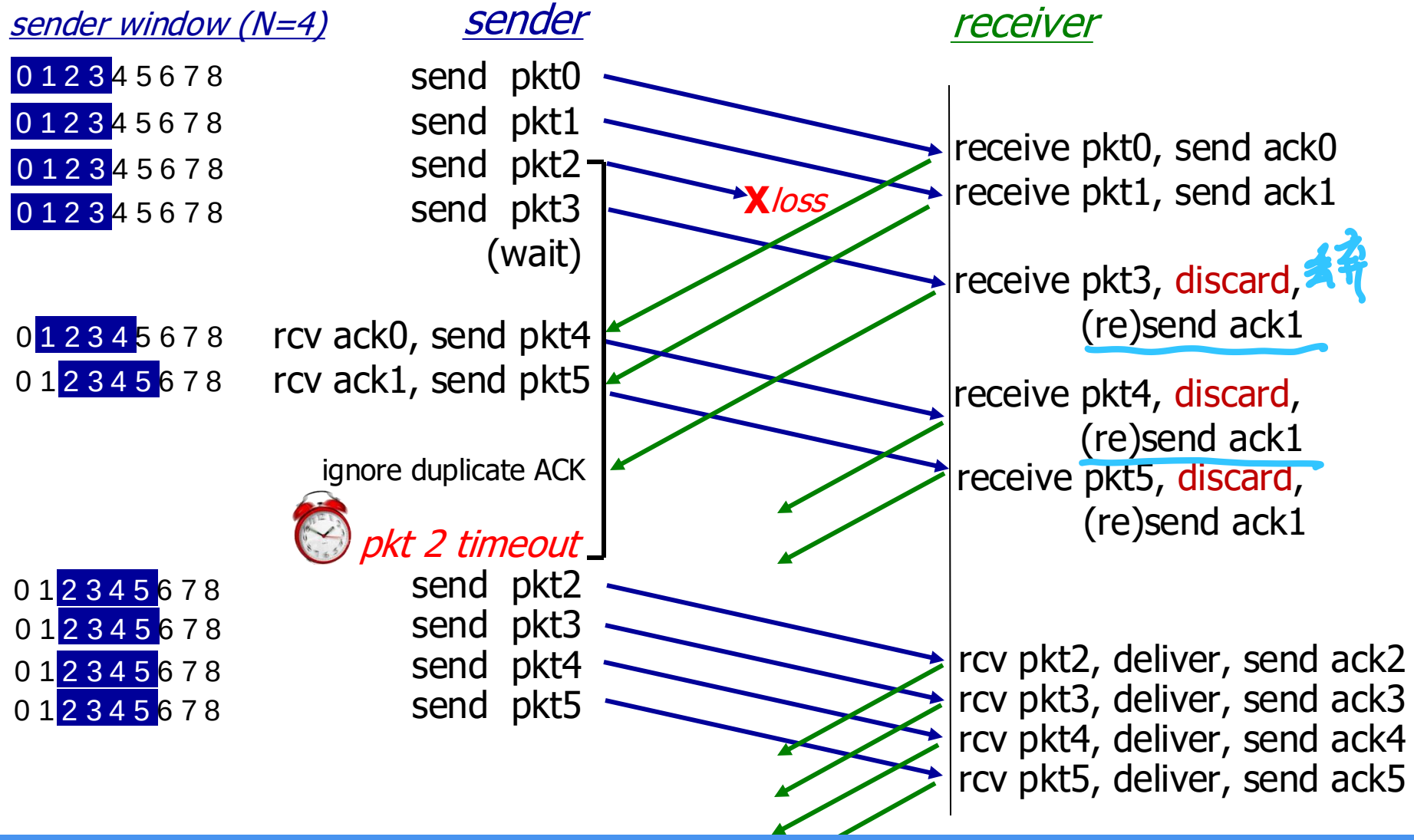


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

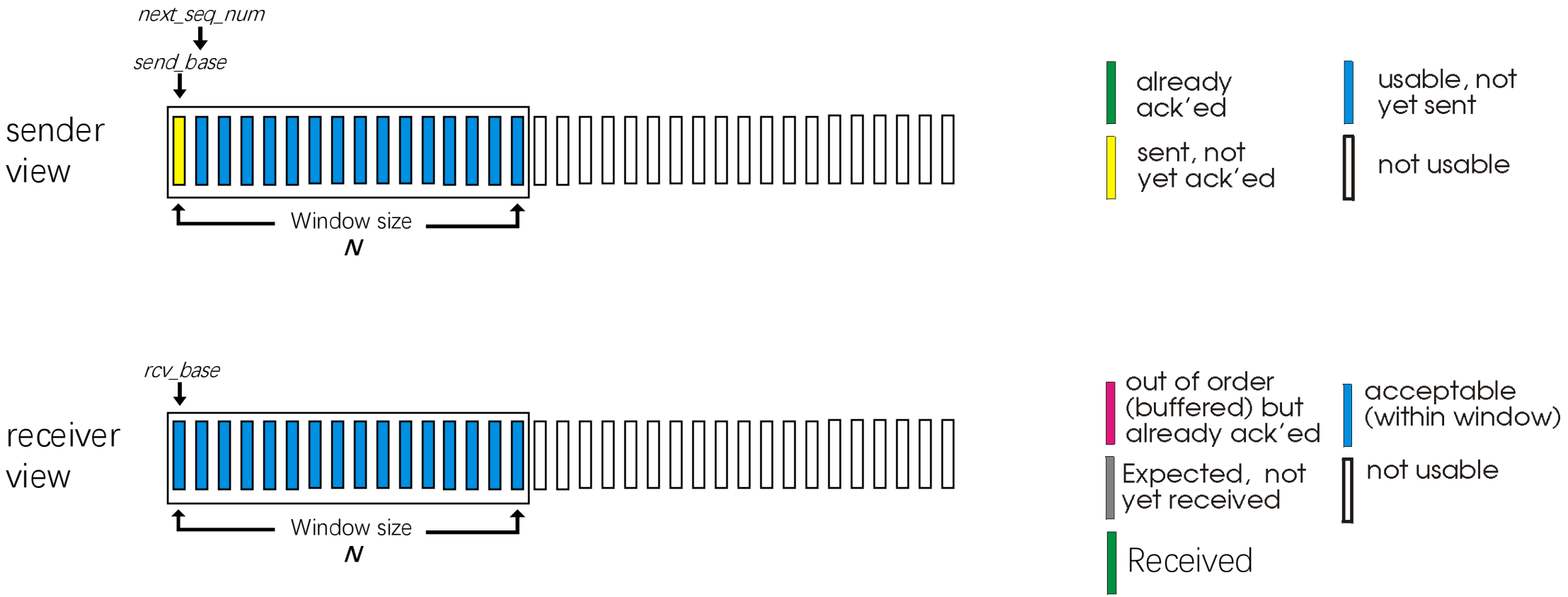
GBN in action



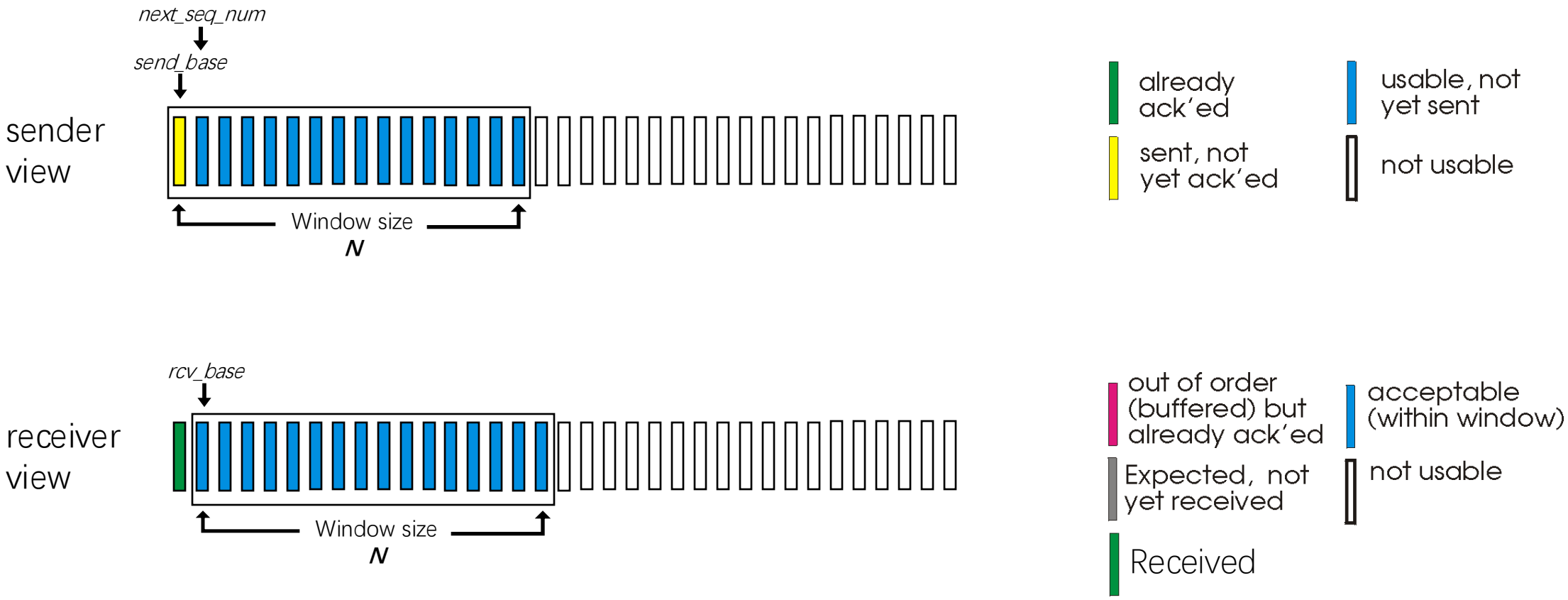
Selective repeat

- **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

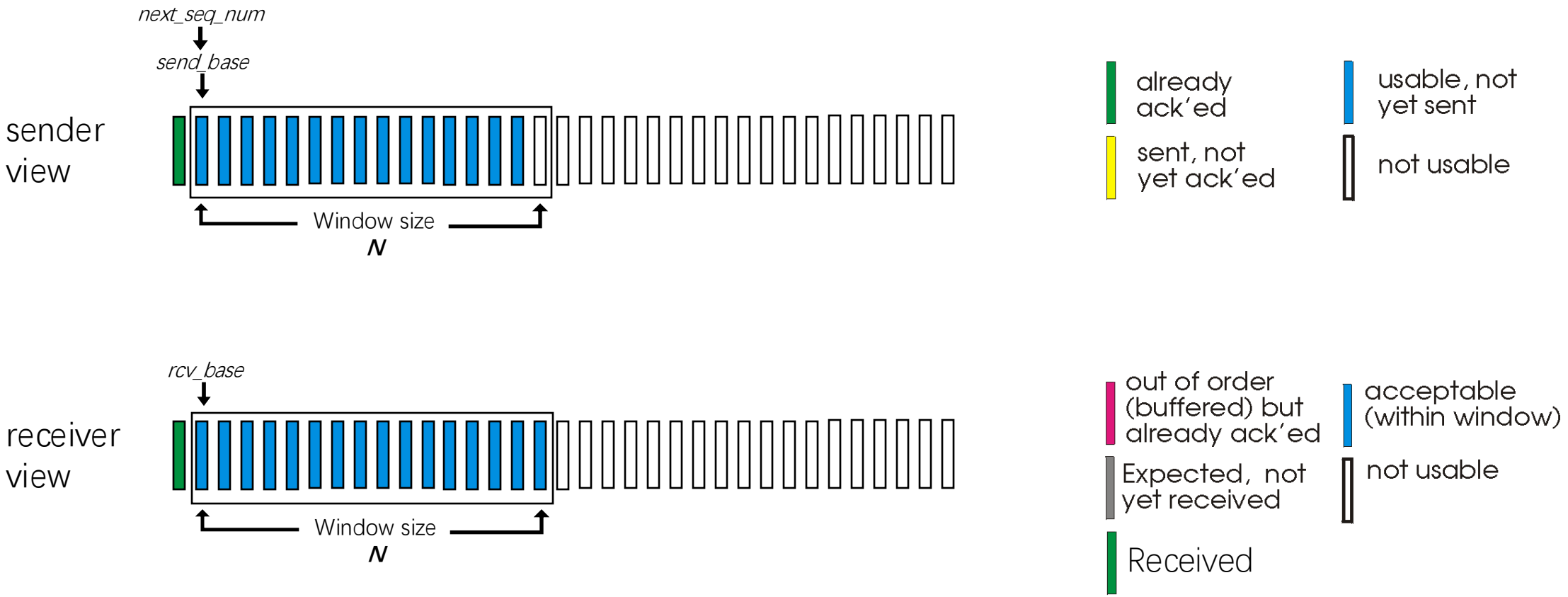
Selective repeat: sender, receiver windows



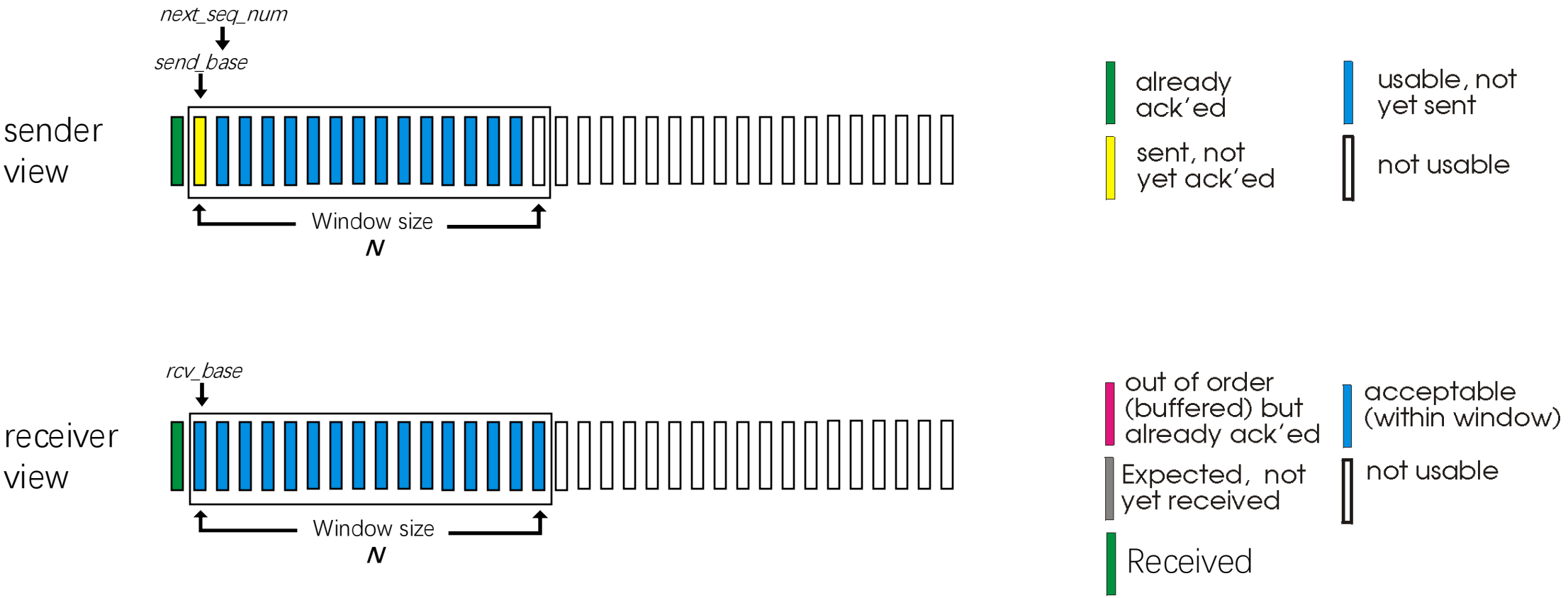
Selective repeat: sender, receiver windows



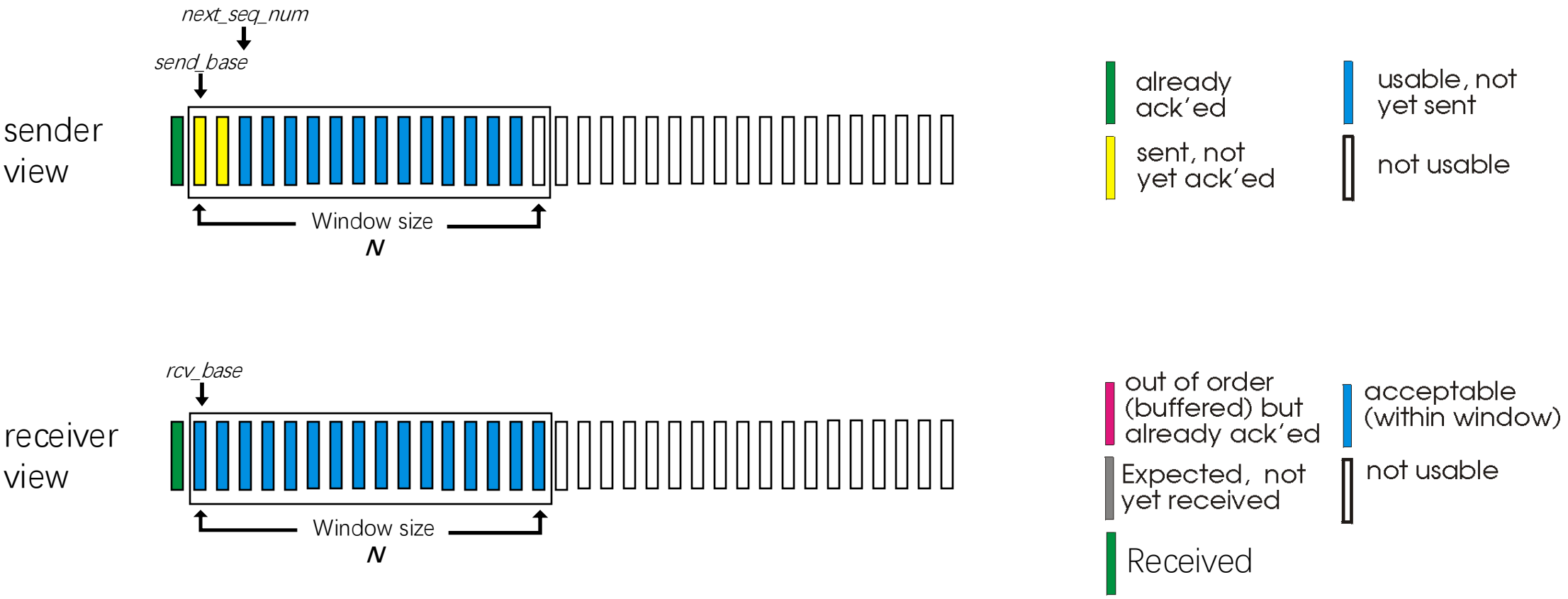
Selective repeat: sender, receiver windows



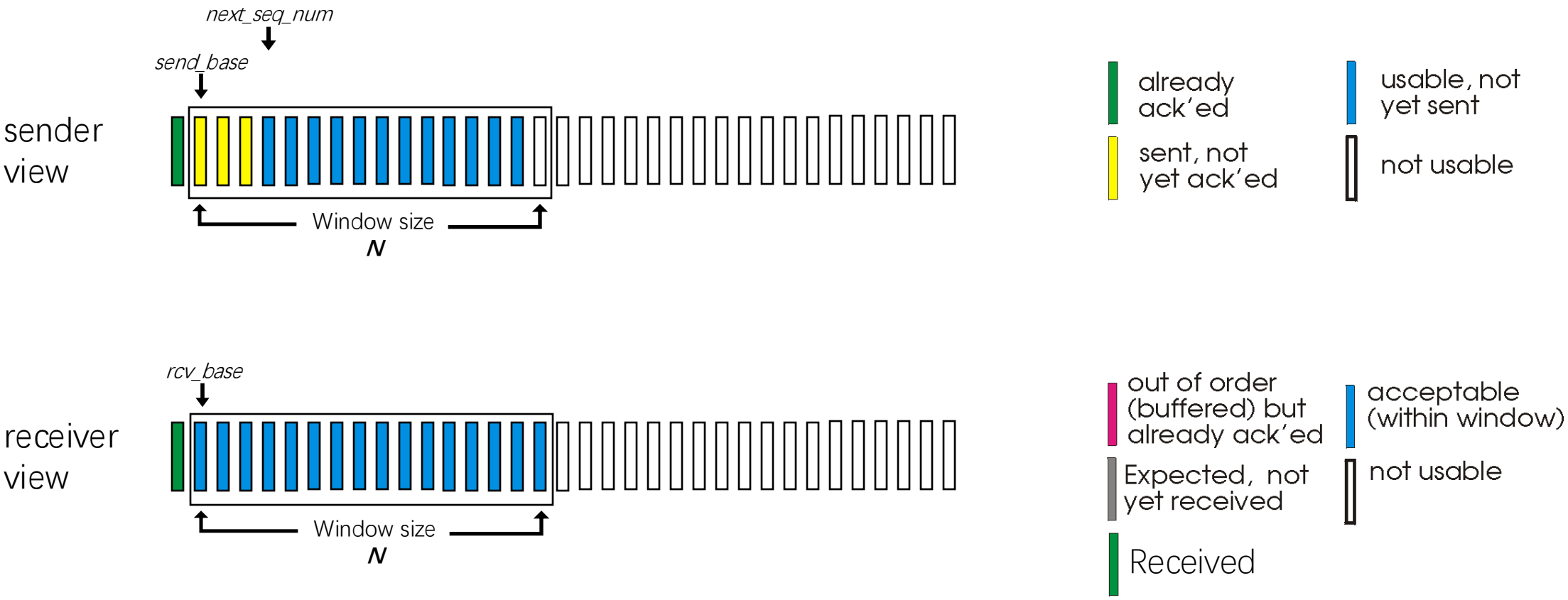
Selective repeat: sender, receiver windows



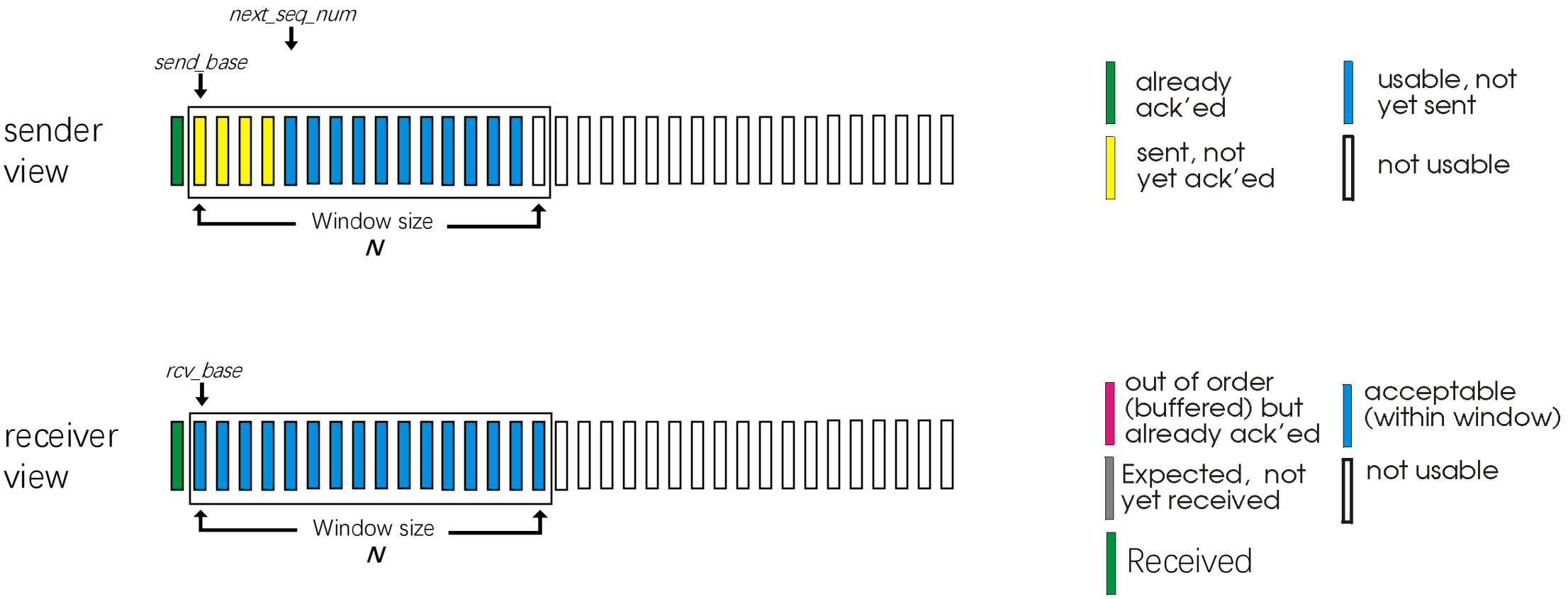
Selective repeat: sender, receiver windows



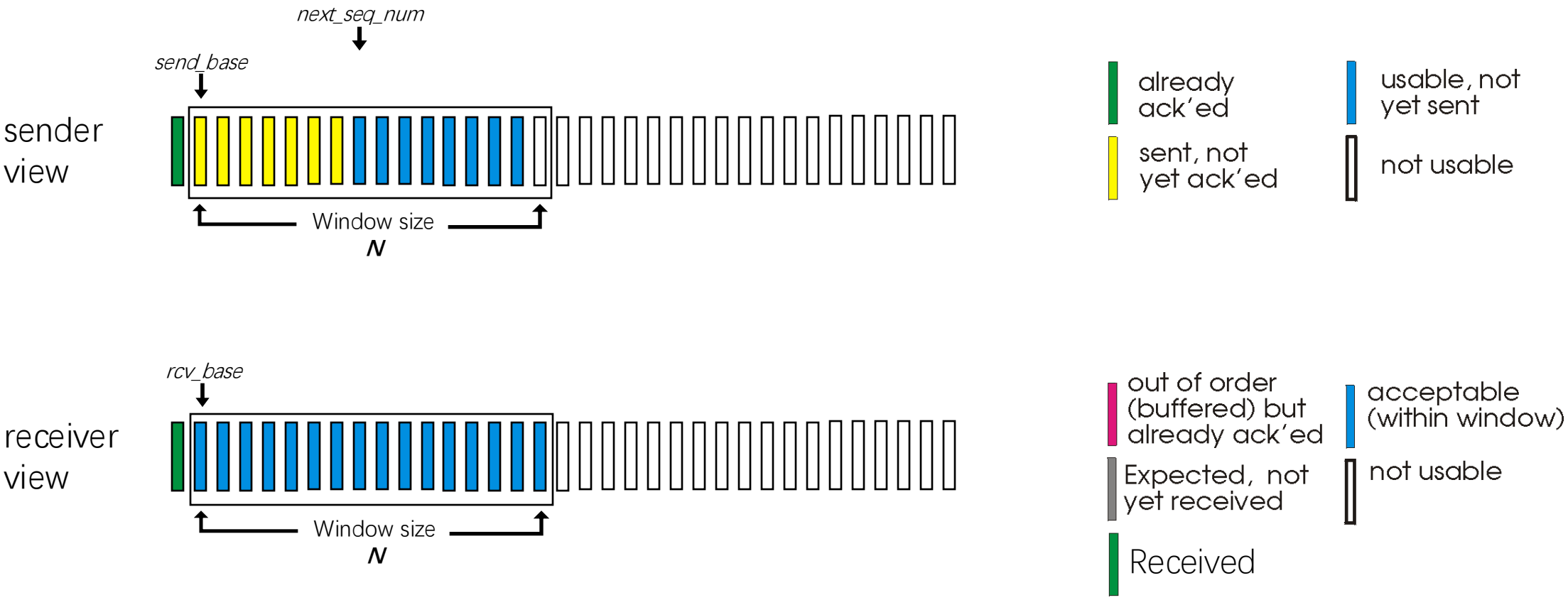
Selective repeat: sender, receiver windows



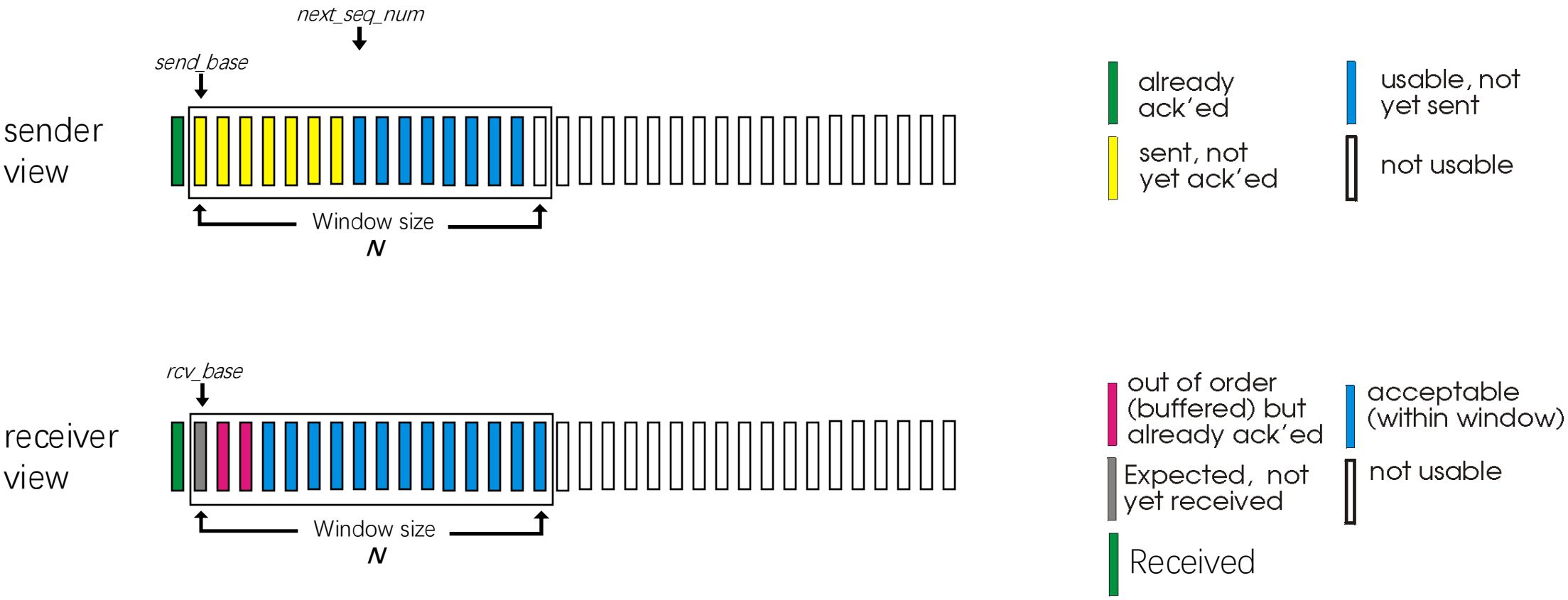
Selective repeat: sender, receiver windows



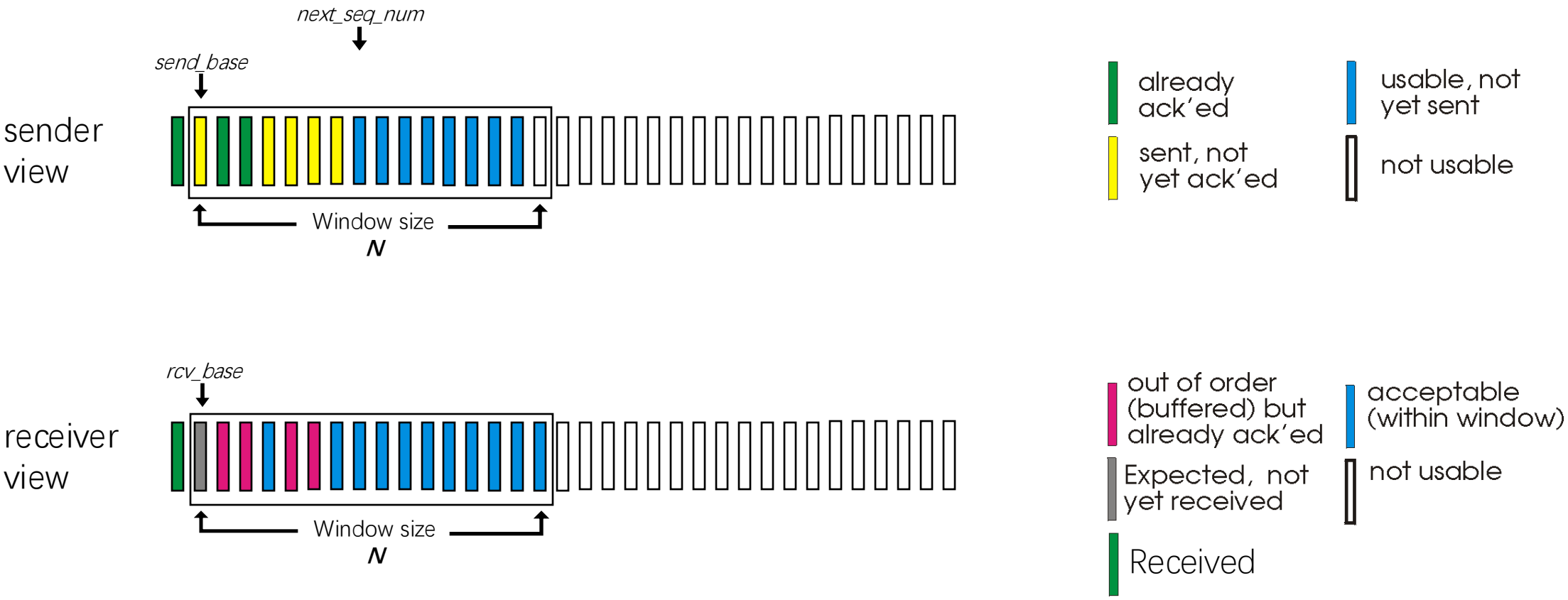
Selective repeat: sender, receiver windows



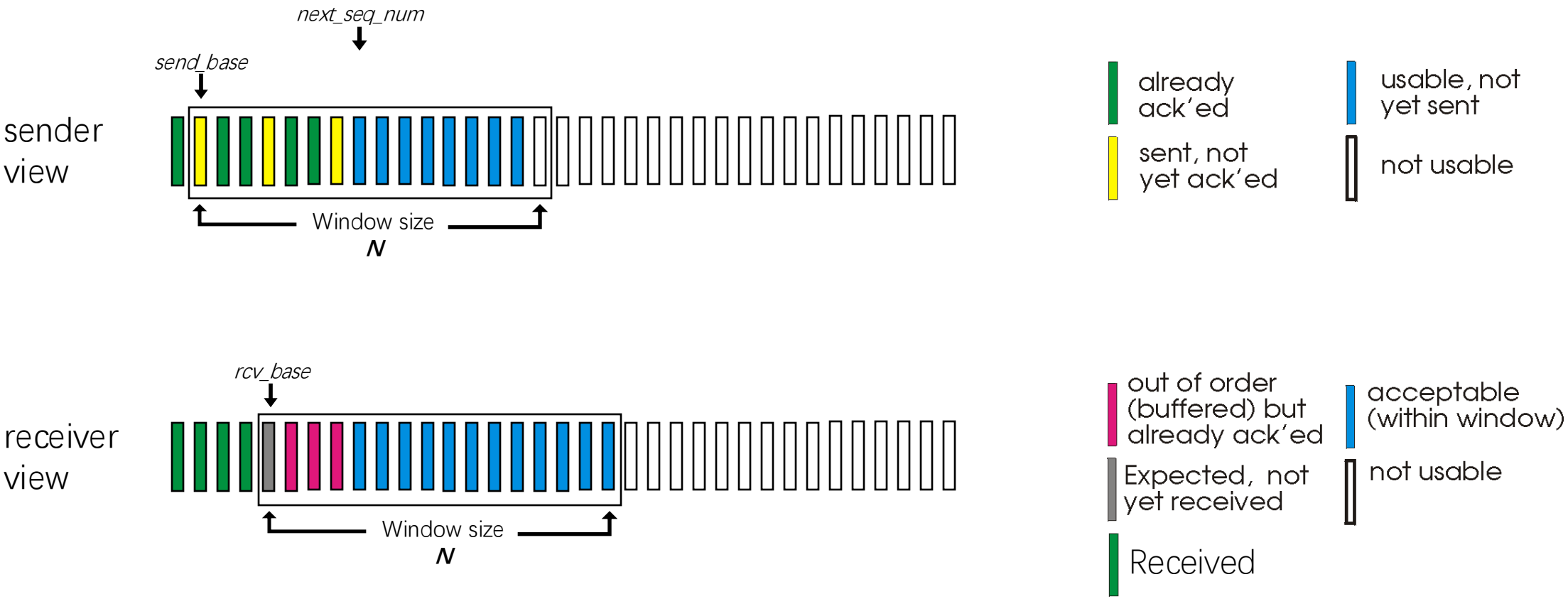
Selective repeat: sender, receiver windows



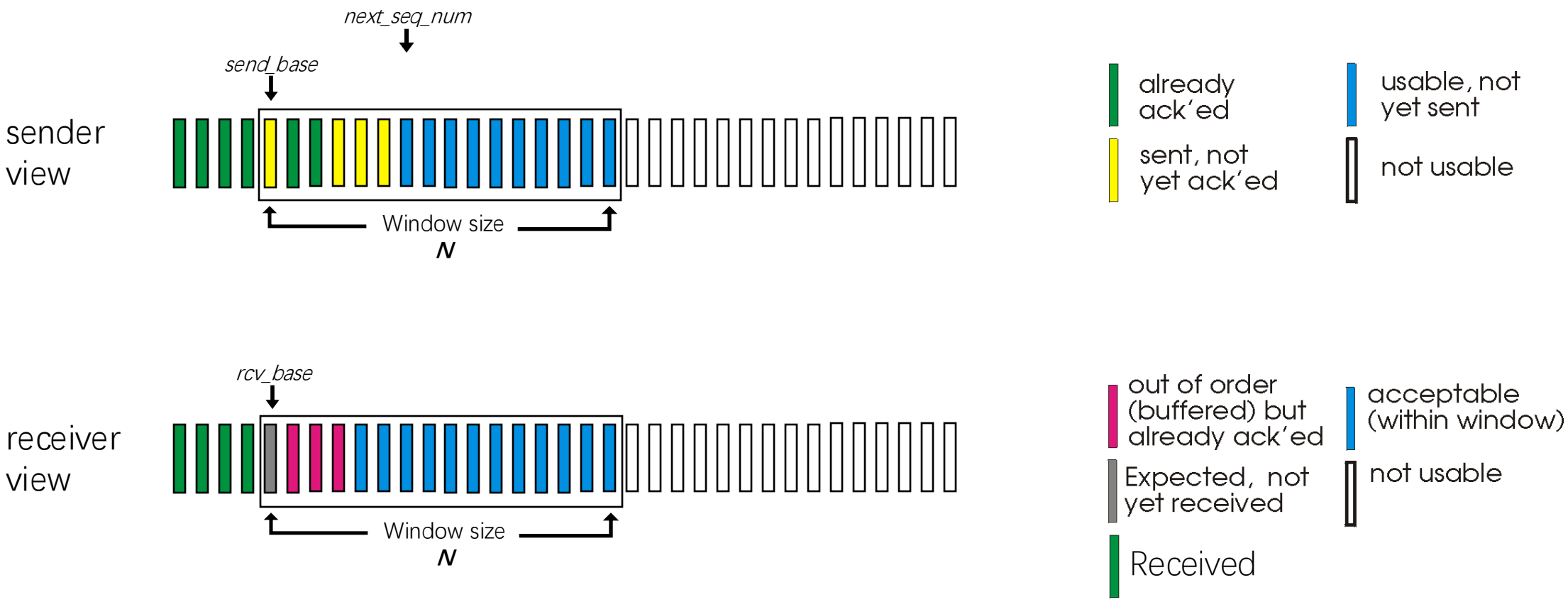
Selective repeat: sender, receiver windows



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]:

- 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

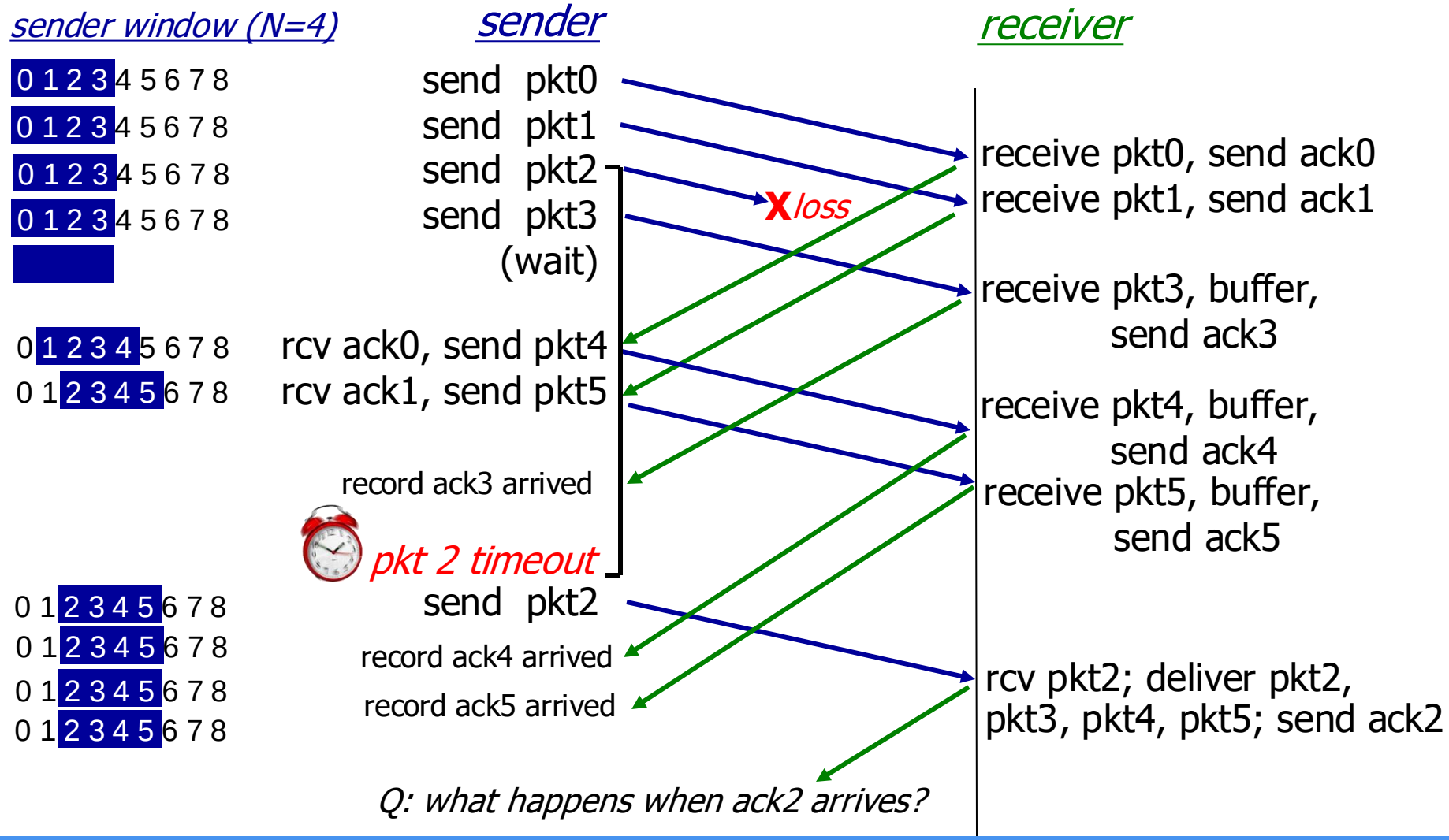
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- Ignore

Selective repeat in action

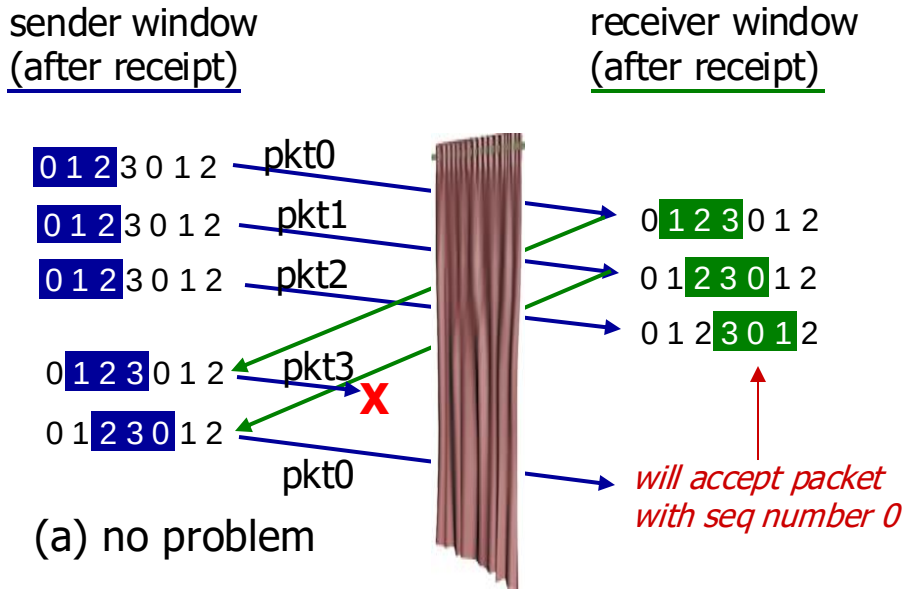


Selective repeat: dilemma

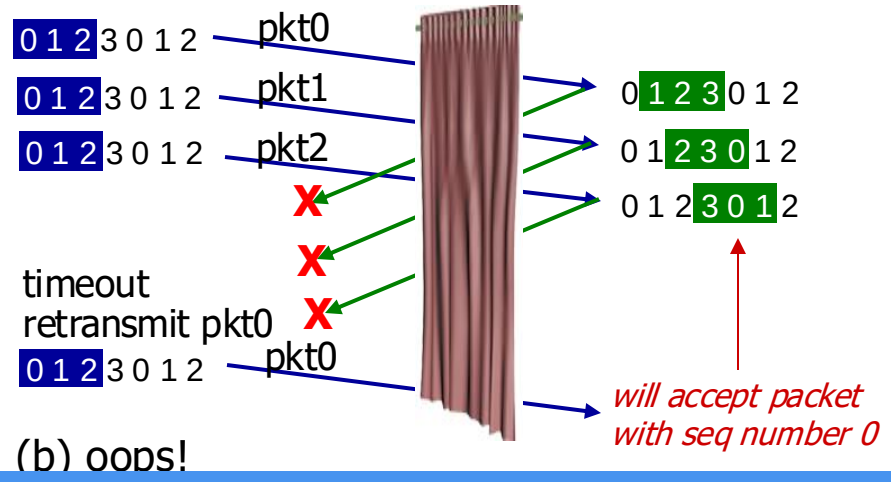
Example:

- 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!



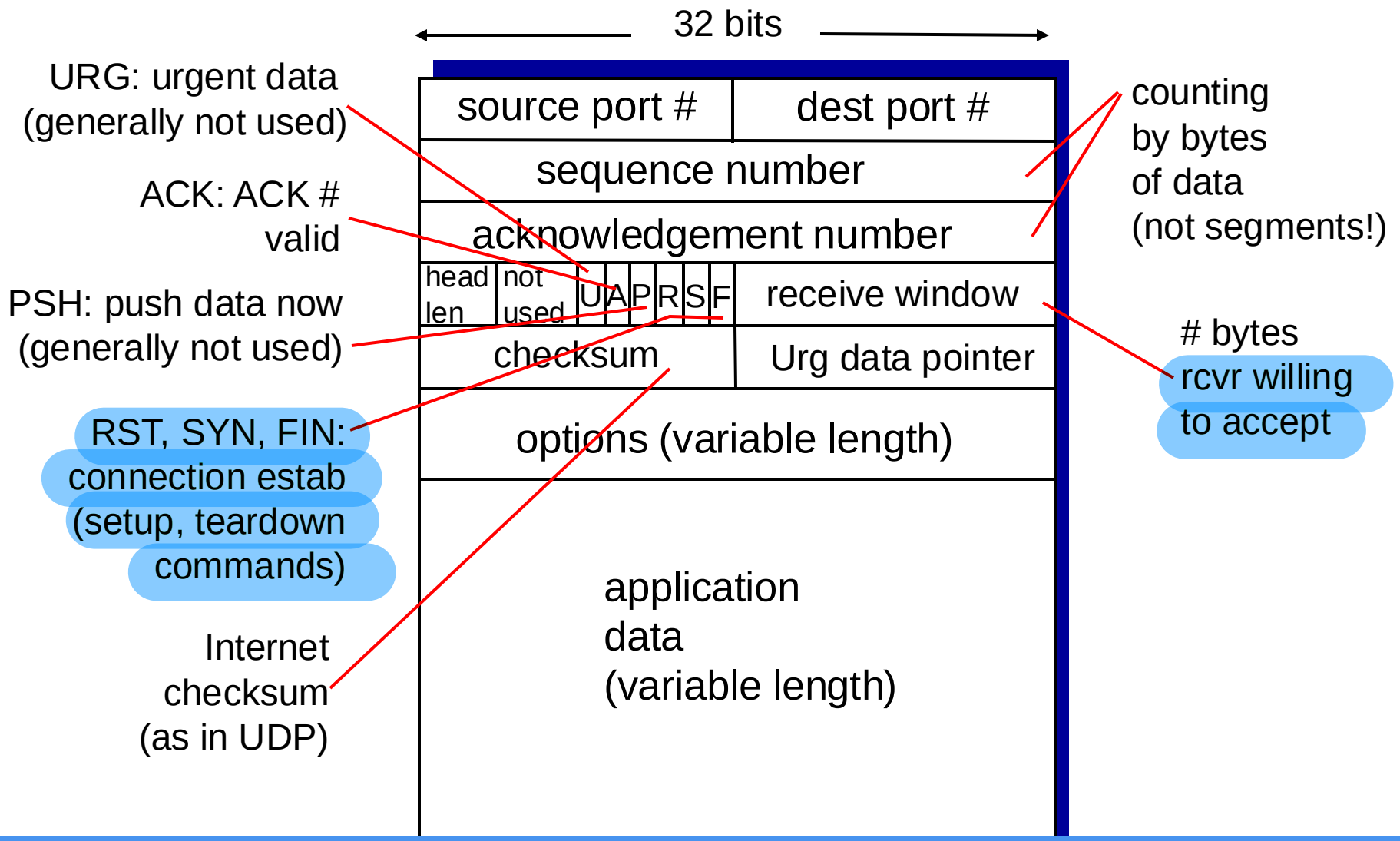
Lecture 5 – Transport Layer (1)

- **Roadmap**
  1. Pipelined communication
  2. TCP: connection-oriented transport
  3. Principles of congestion control

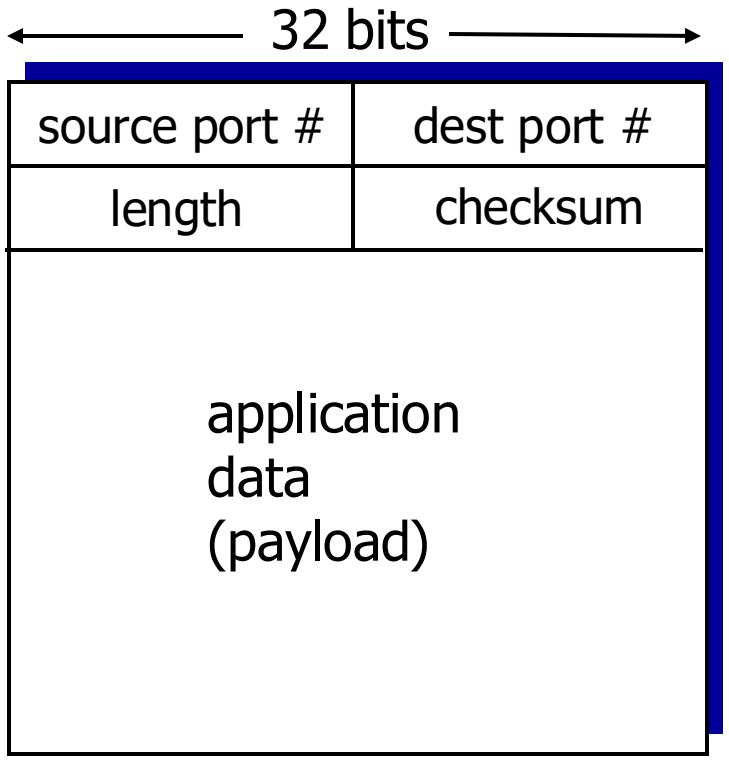
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- **Point-to-point:**
  - one sender, one receiver
- **Reliable, in-order *byte stream* :**
  - no "message boundaries"
- **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

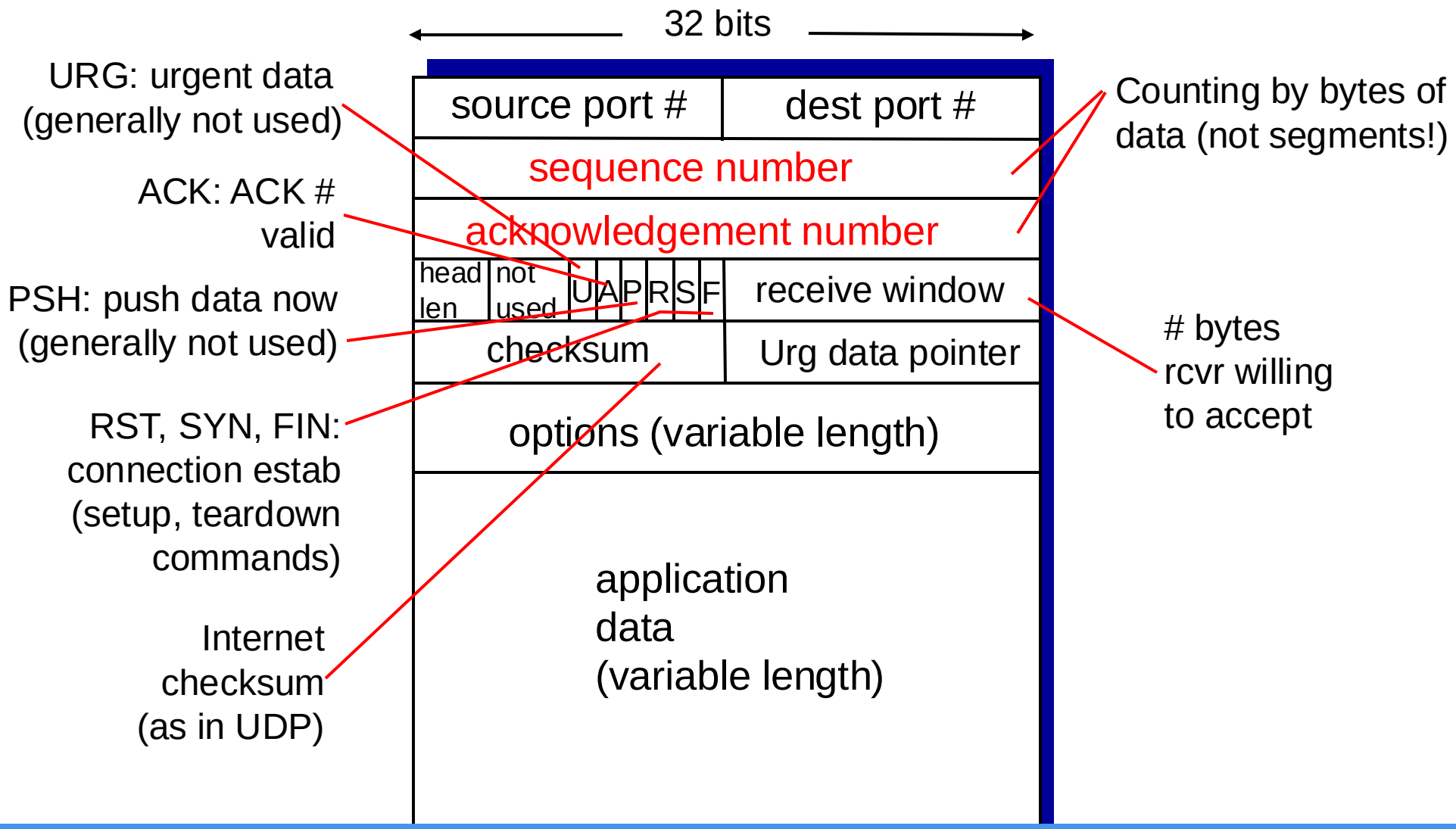


vs UDP structure



UDP segment format

TCP segment structure



TCP seq. numbers, ACKs

Sequence numbers: 序列号

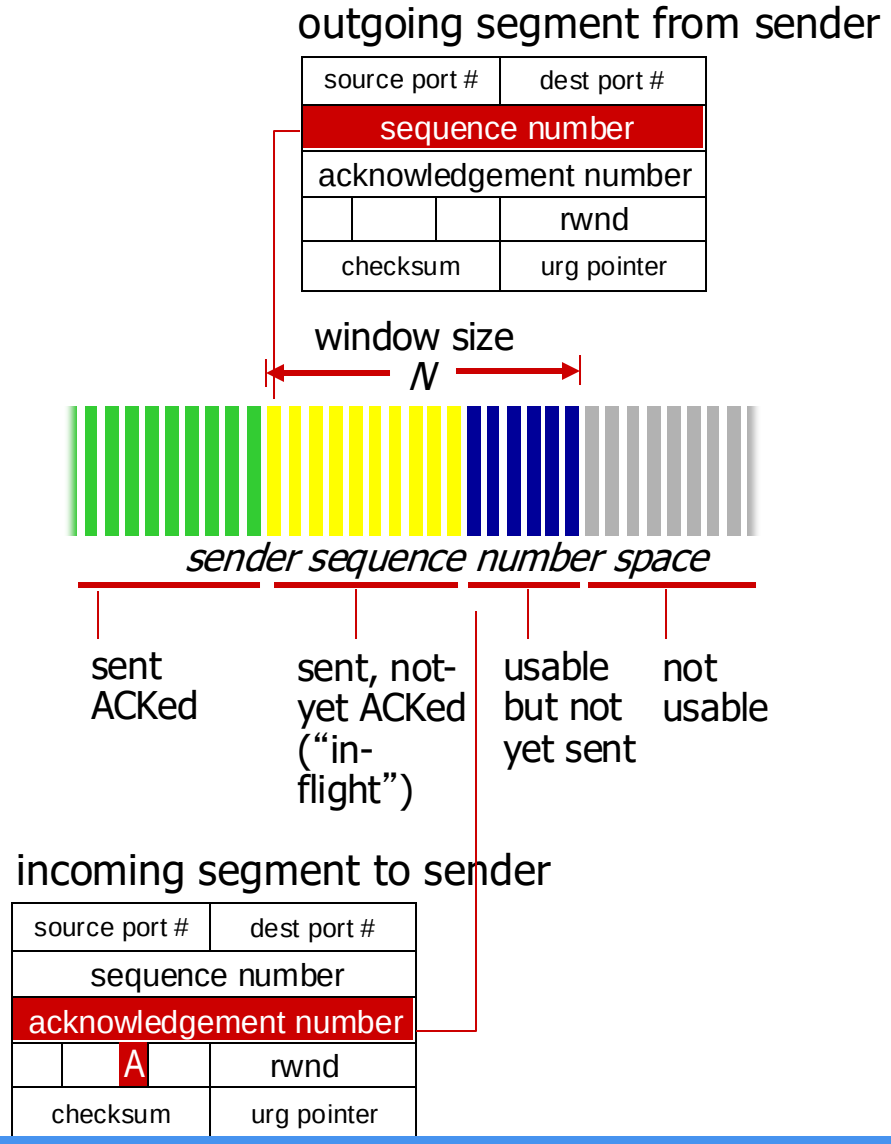
- Byte stream "number" of first byte in segment's data

Acknowledgements:

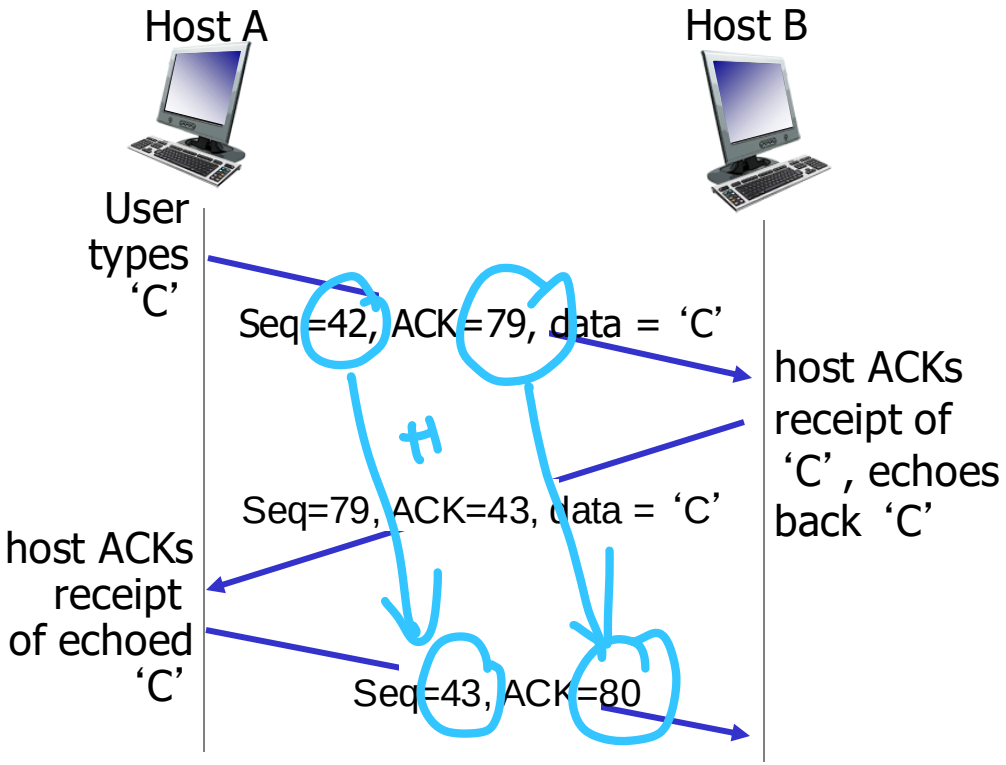
- Seq # of next byte expected from other side
- Cumulative ACK

Q: How receiver handles out-of-order segments

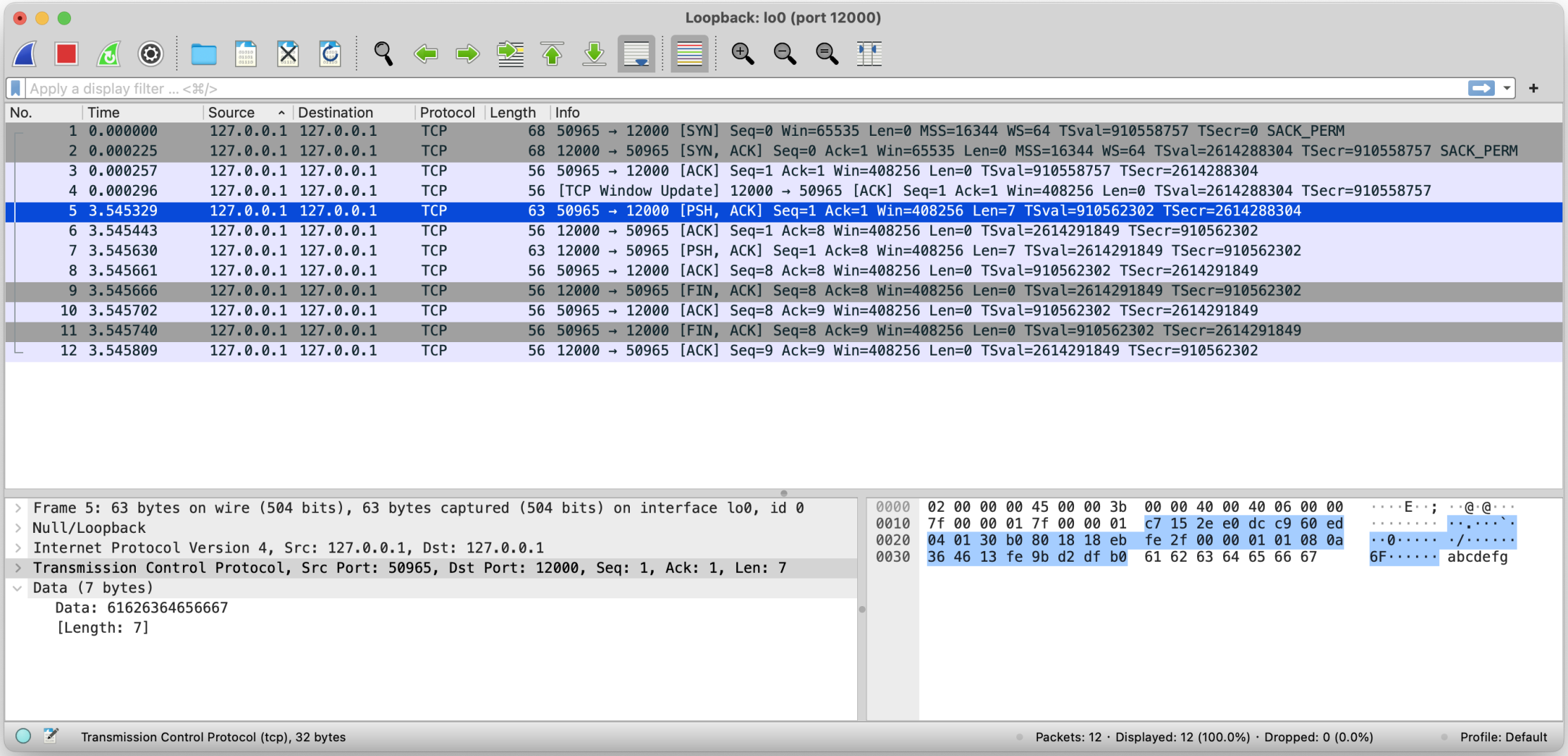
- A: TCP spec doesn't say, up to implementor. (Buffer…)



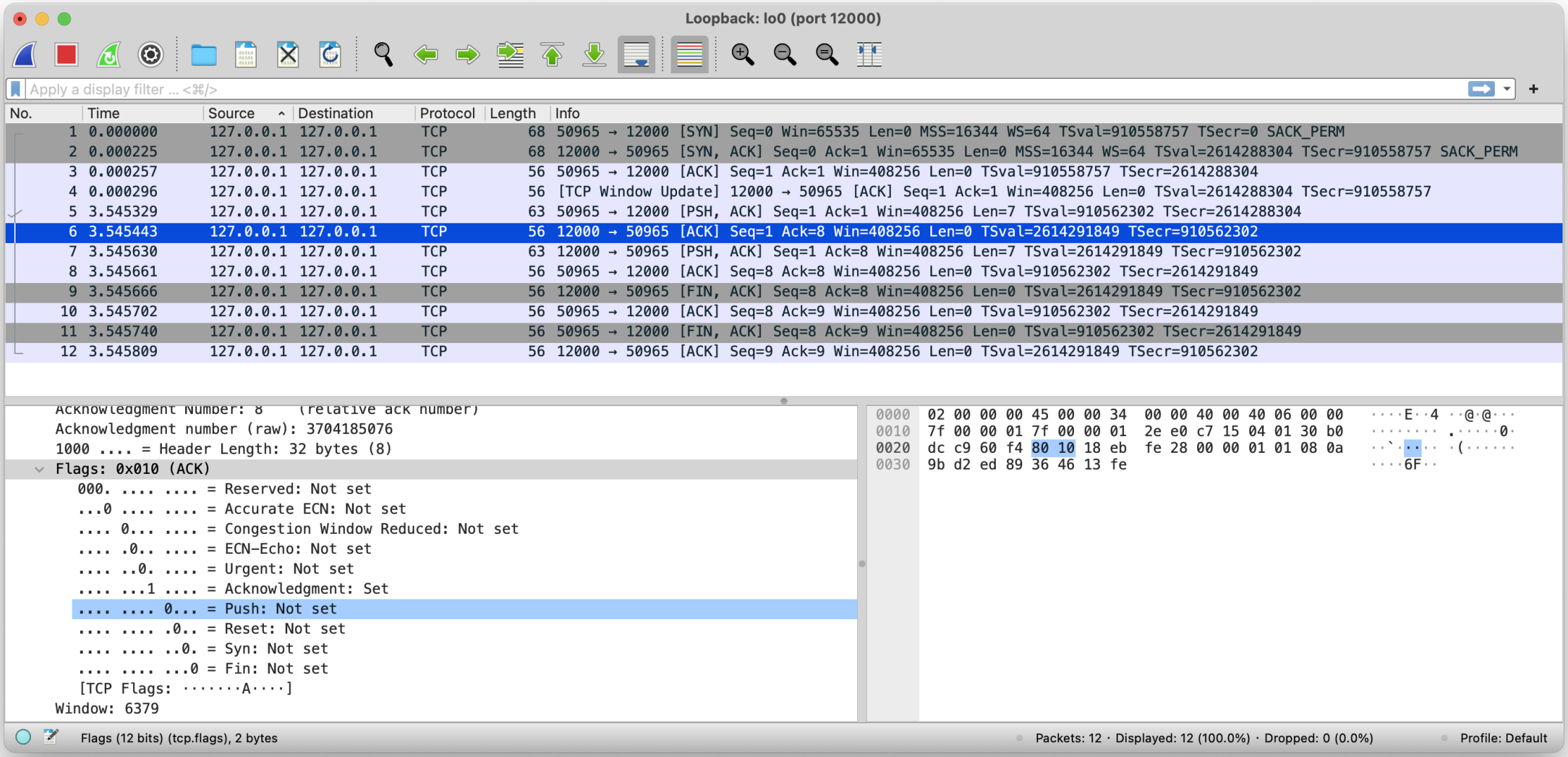
TCP seq. numbers, ACKs



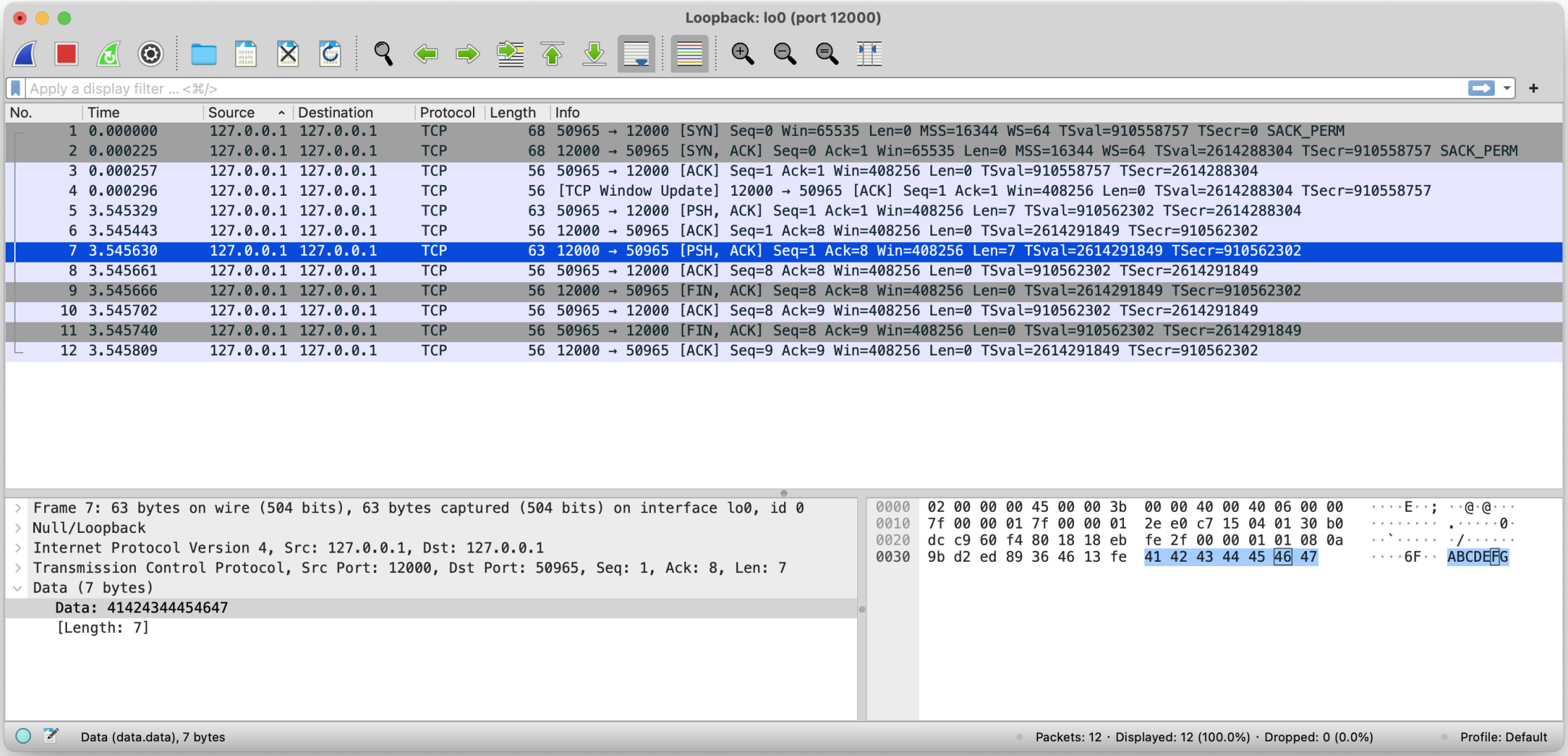
simple telnet scenario



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**

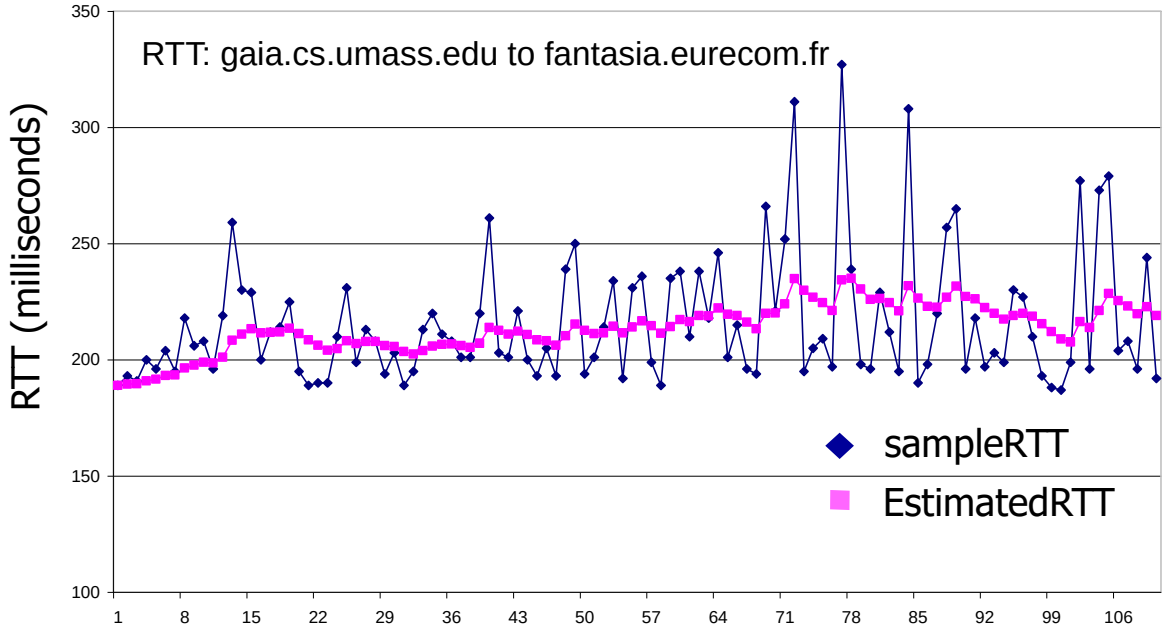
Q: How to estimate RTT?

- **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

$$EstimatedRTT_n = (1- \alpha)*EstimatedRTT_{n-1} + \alpha*SampleRTT$$

- 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-β)*DevRTT +
         β*|SampleRTT-EstimatedRTT|
(typically, β = 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 开始计时器
  - Think of timer as for oldest unacked segment
  - Expiration interval: `TimeOutInterval`

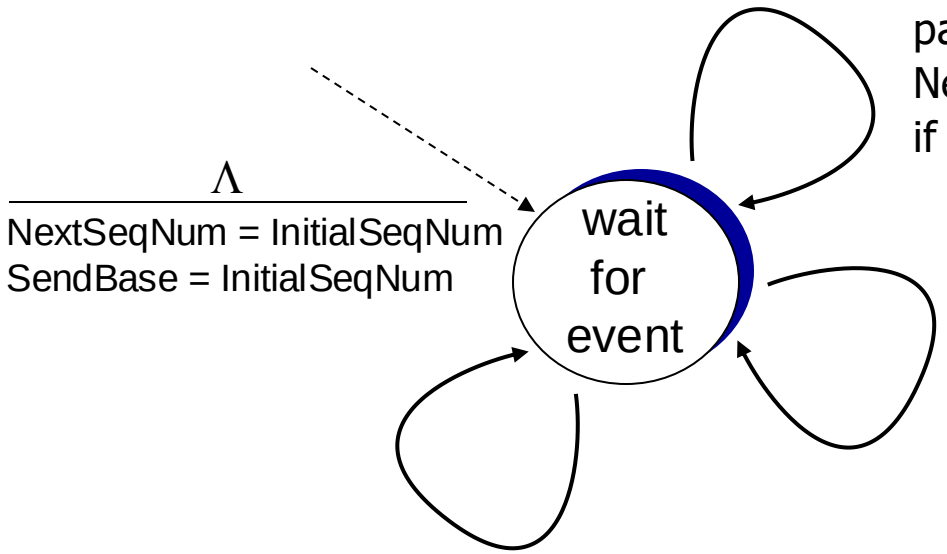
Timeout:

- 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 sender (simplified)



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

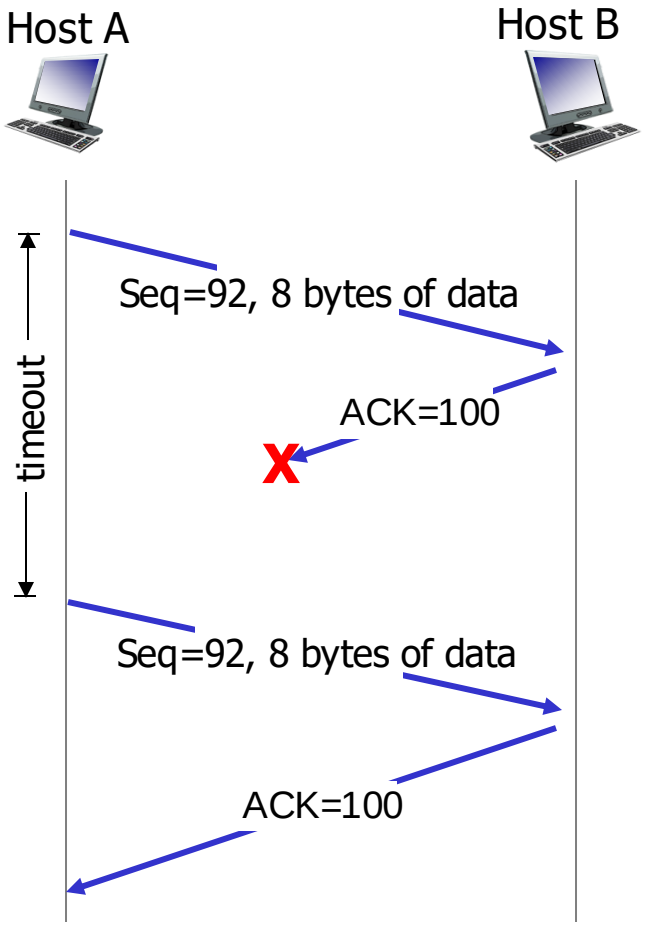
timeout

retransmit not-yet-acked segment
with smallest seq. #
start timer

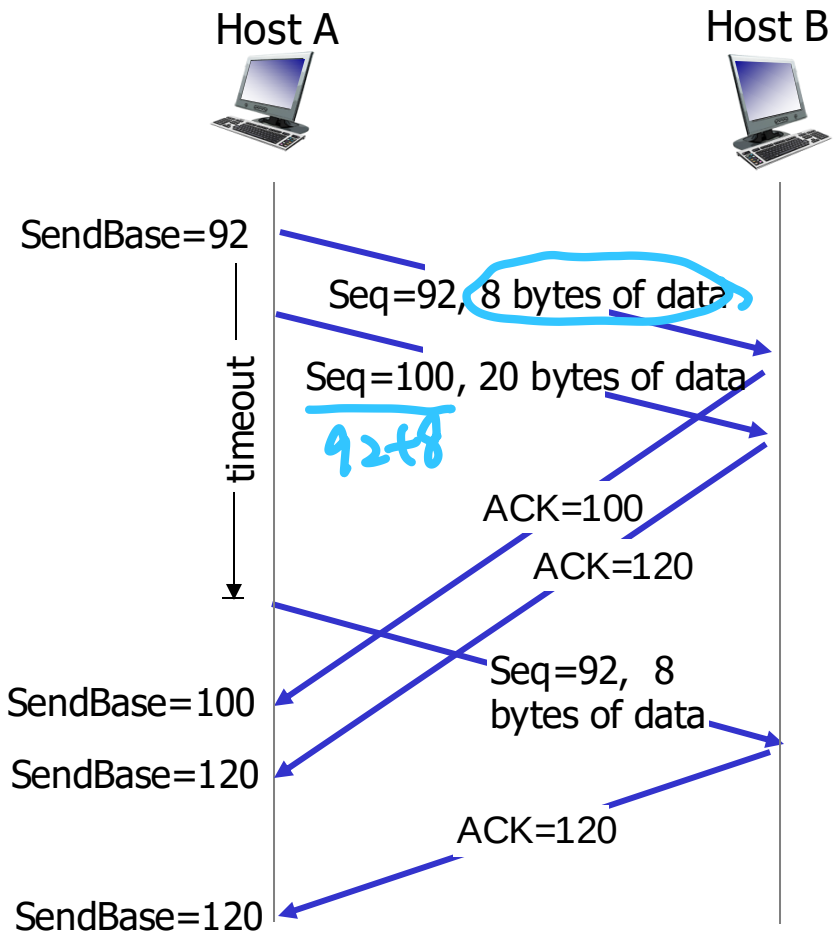
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
    else stop timer
}
```

TCP: retransmission scenarios

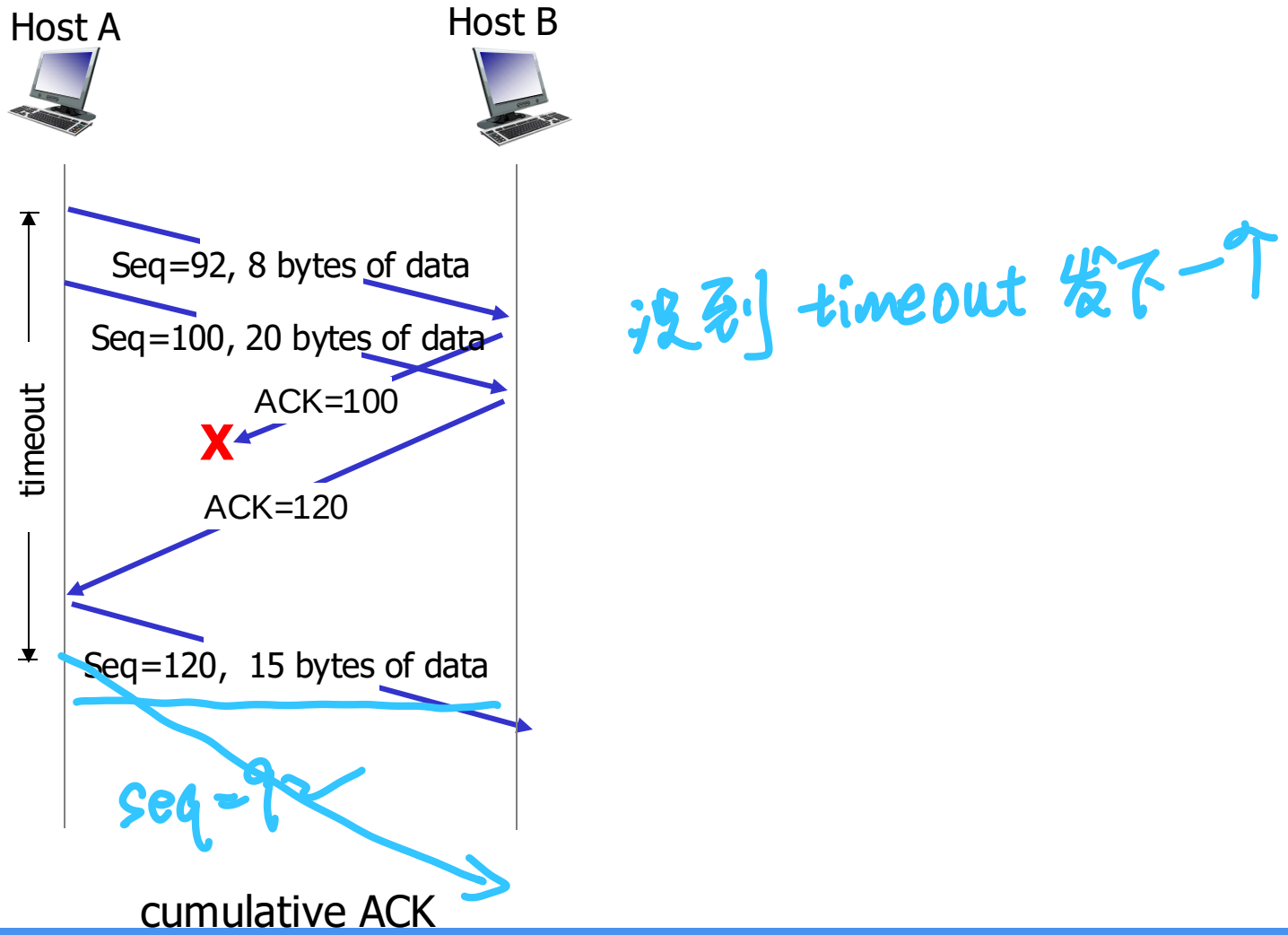


lost ACK scenario



premature timeout

TCP: retransmission scenarios



没到 timeout 发下一个

TCP ACK generation [RFC 1122, RFC 2581]

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 delayed 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 fast retransmit 快速重传

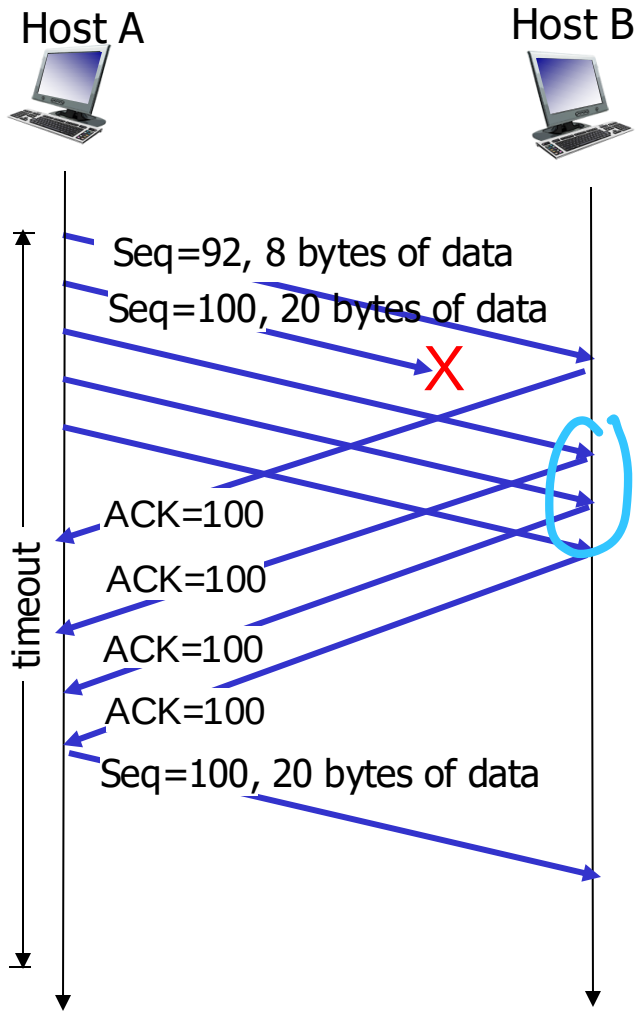
- **Time-out period often relatively long:**
  - Long delay before resending lost packet
- **Detect lost segments via duplicate ACKs.**
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

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

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

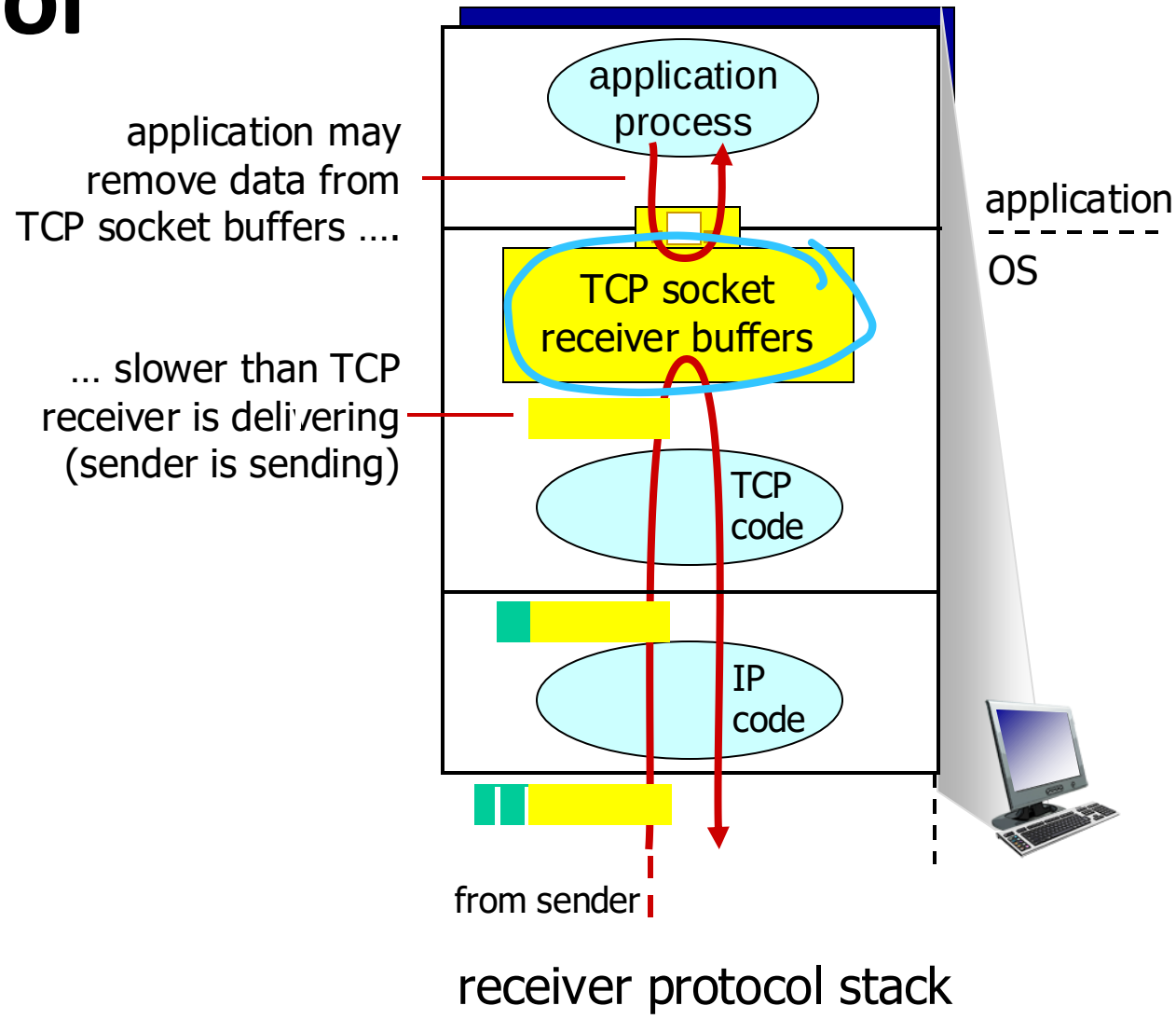
TCP fast retransmit



TCP flow control

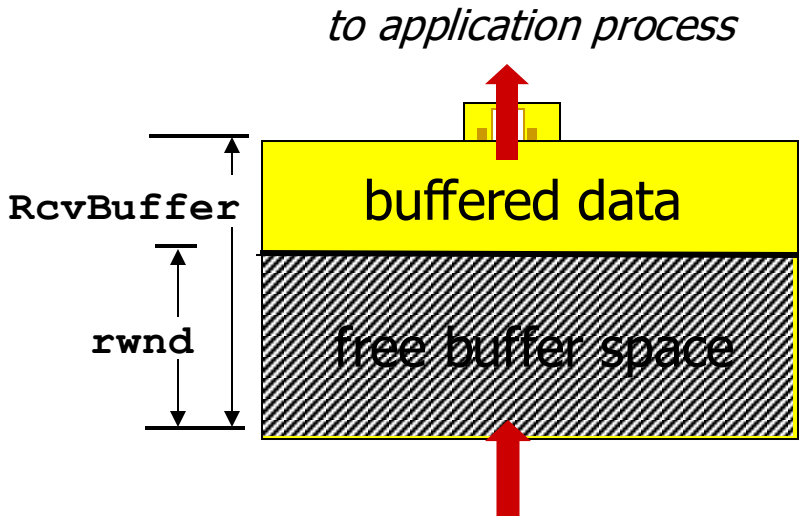
Flow control

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



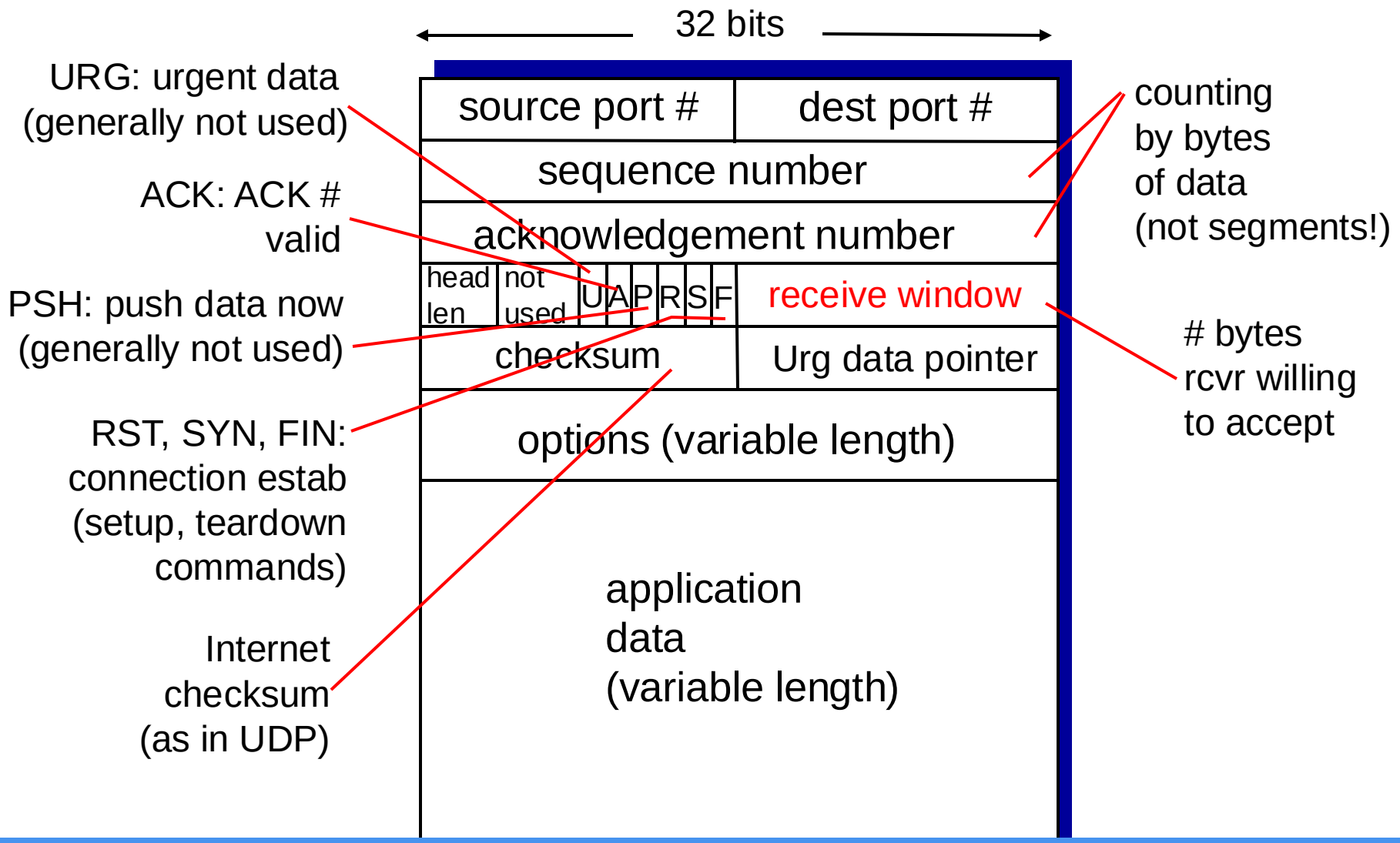
TCP flow control

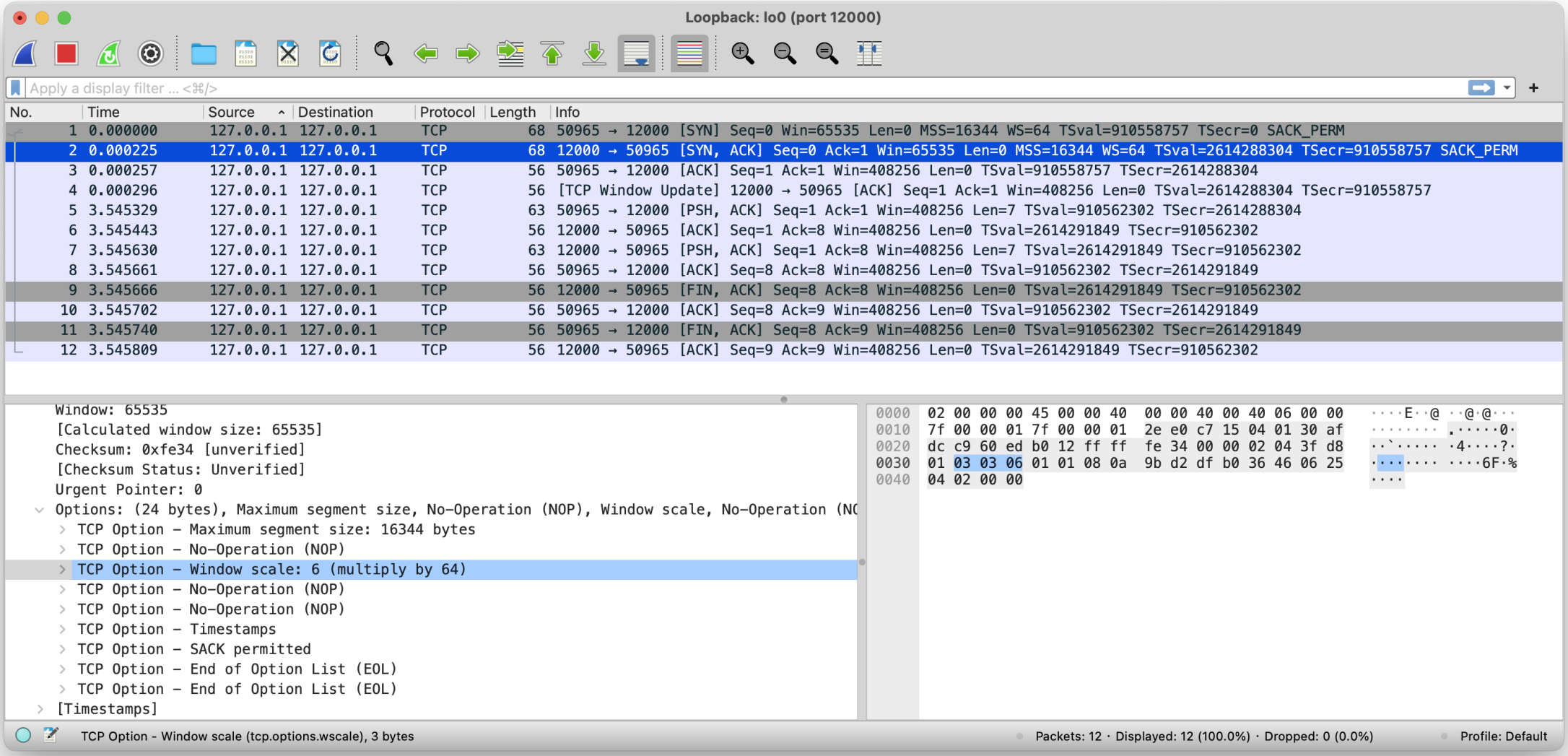
- Receiver "advertises" free buffer space by including `rwnd` value in TCP header of receiver-to-sender segments
  - `RcvBuffer` size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust `RcvBuffer`
- Sender limits amount of unacked ("in-flight") data to receiver's `rwnd` value
- Guarantees receive buffer will not overflow



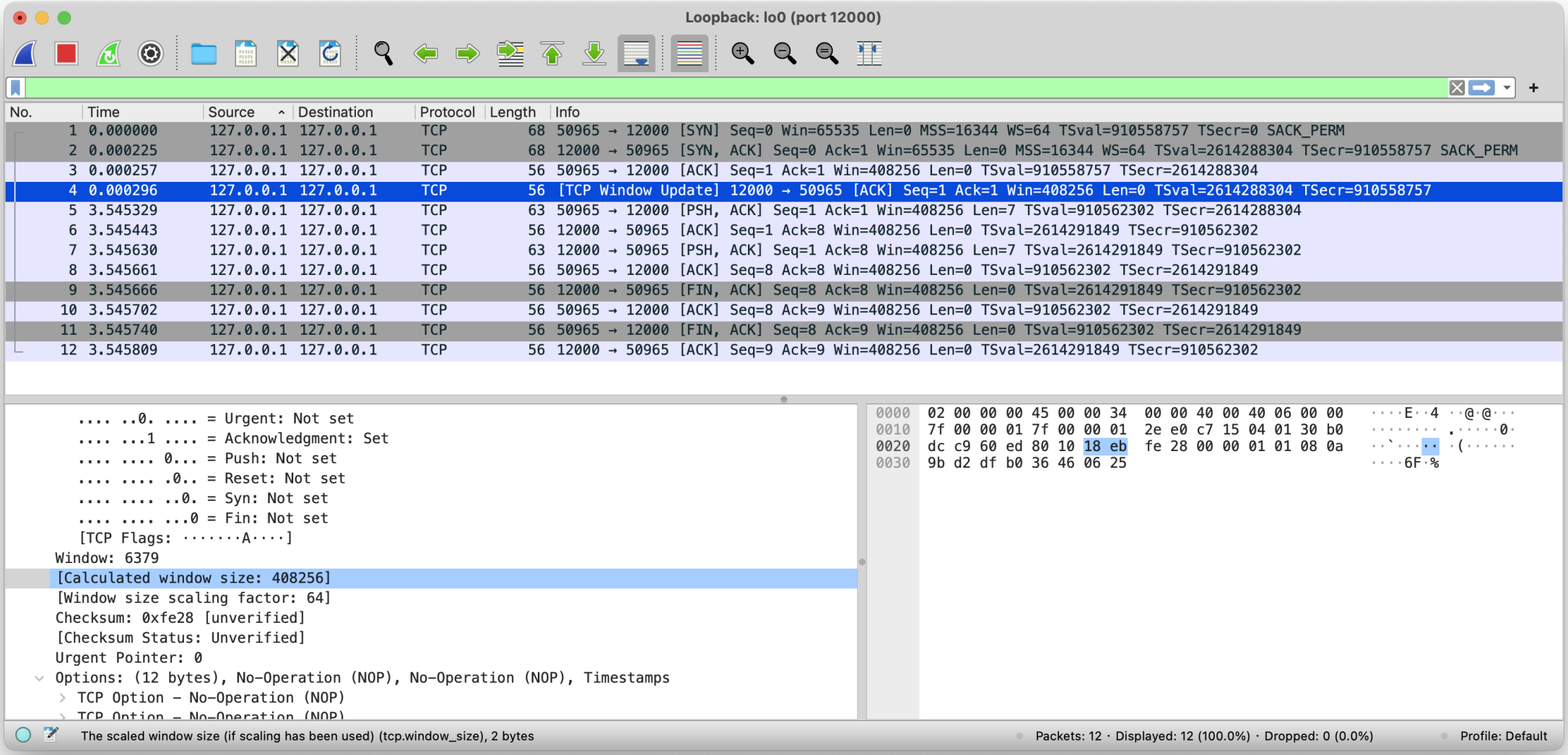
receiver-side buffering

TCP segment structure





SYN: Window Scale factor

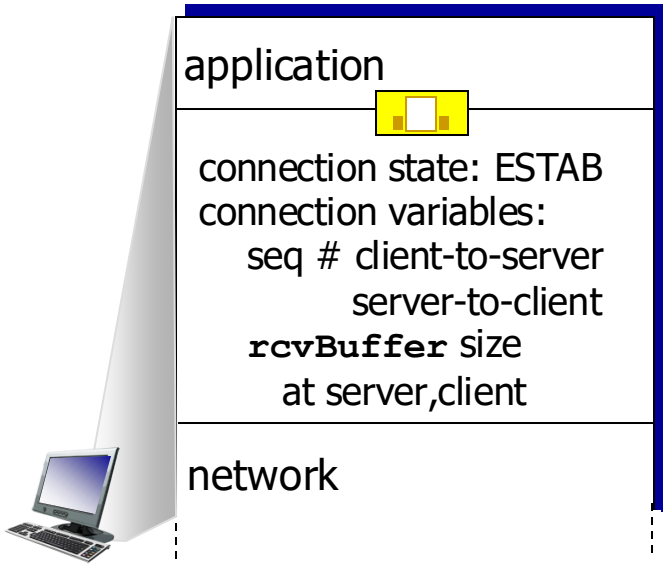


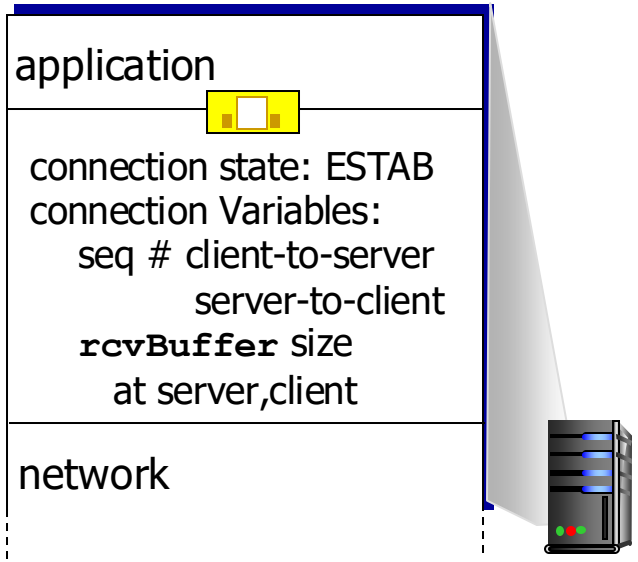
Calculated Window Size = 6379 x 64 = 408256

Connection Management

1 Before exchanging data, sender/receiver "handshake":

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



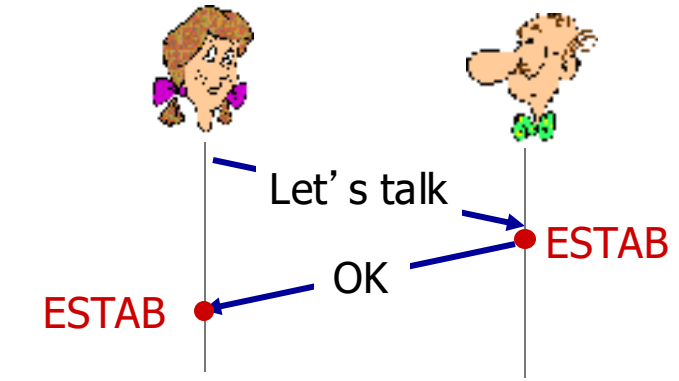


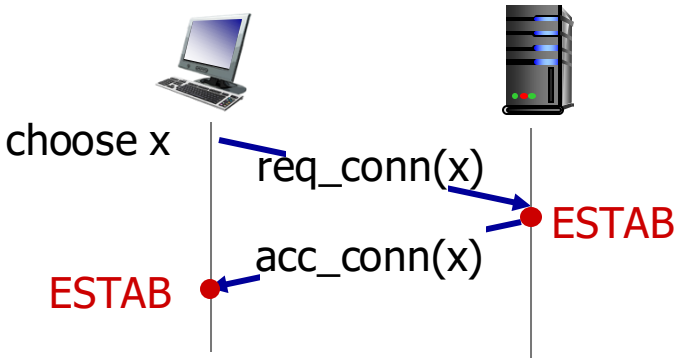

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:





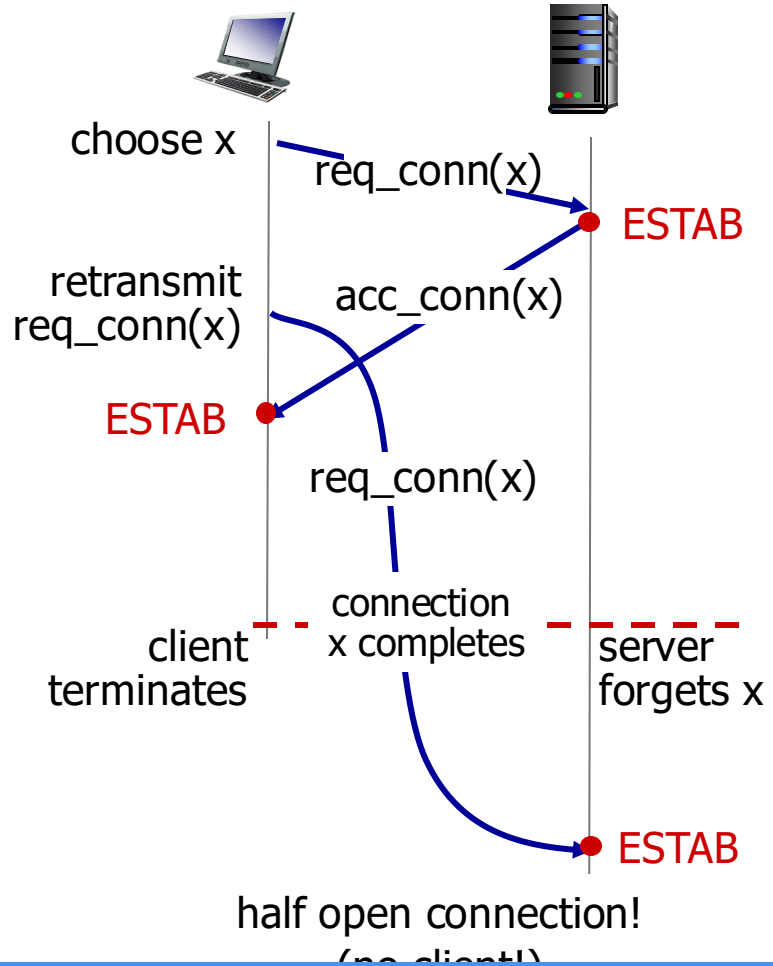
三次握手可能次数.

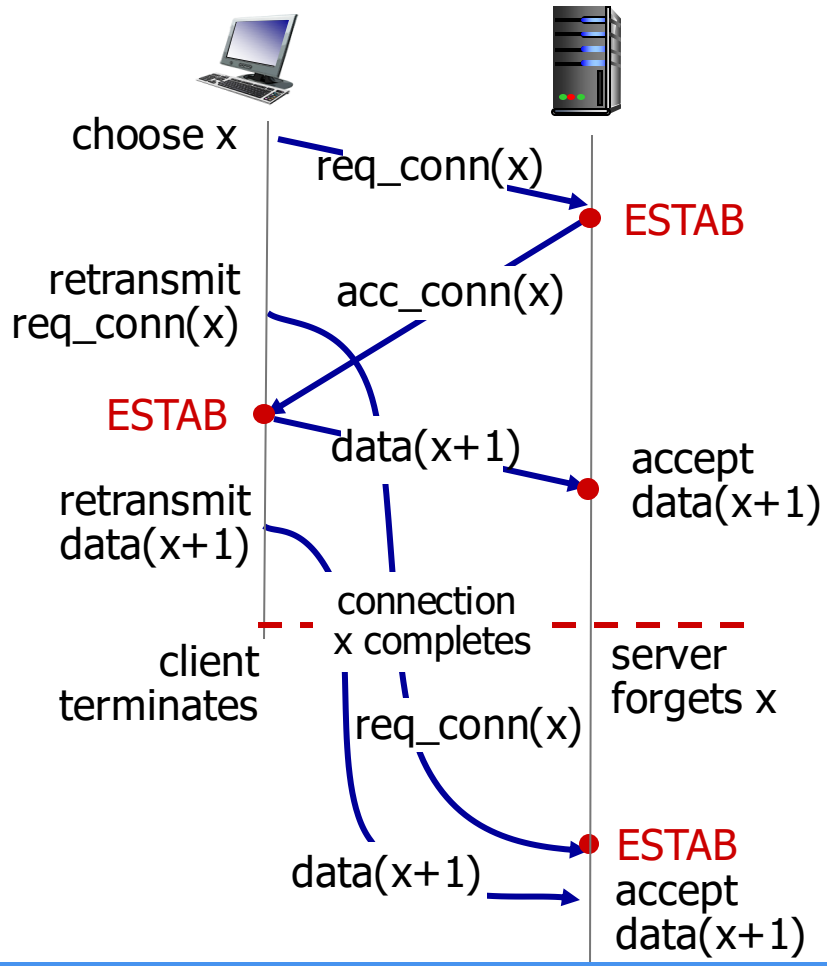
Q: **Will 2-way handshake always work in network?**

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

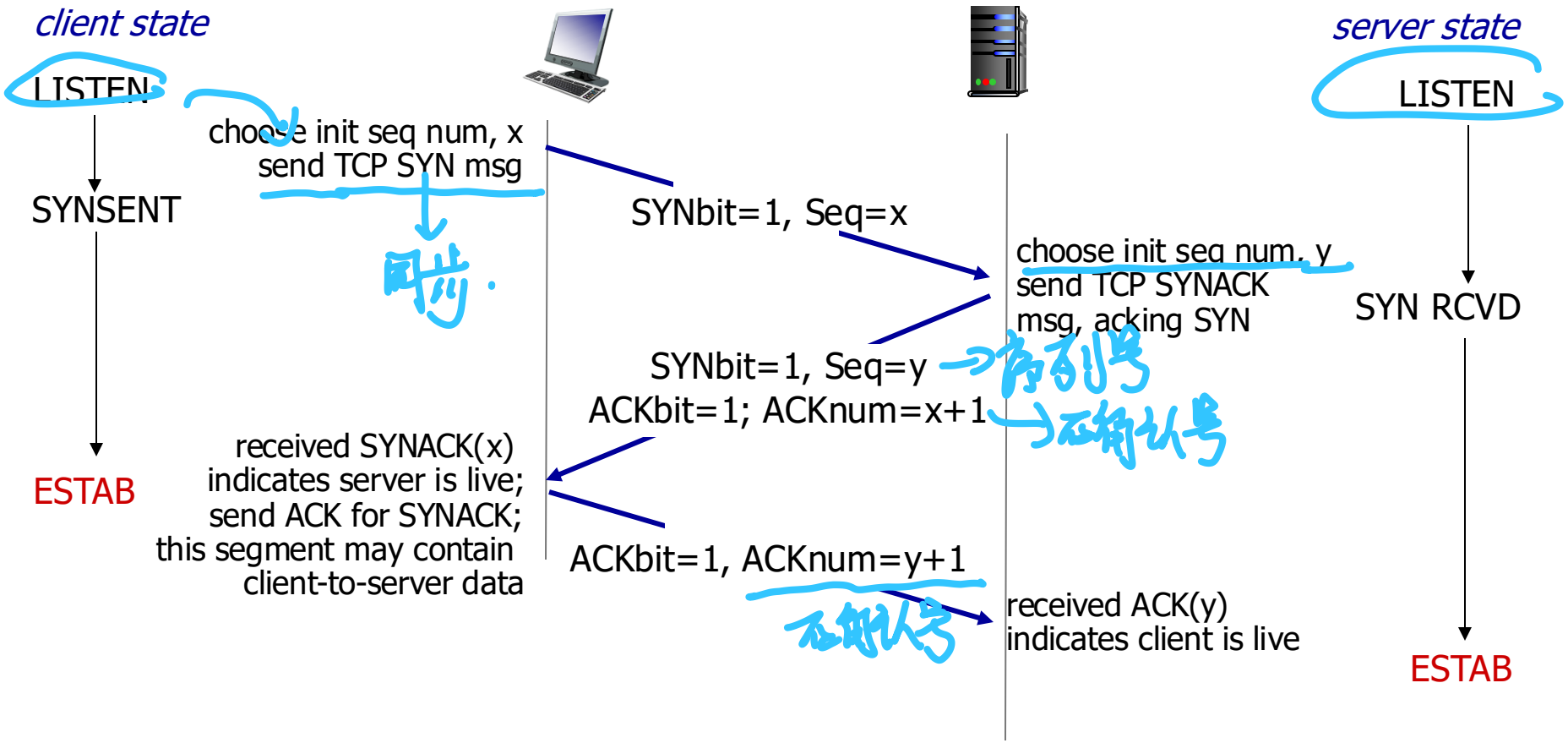
Agreeing to establish a connection

2-way handshake failure scenarios:





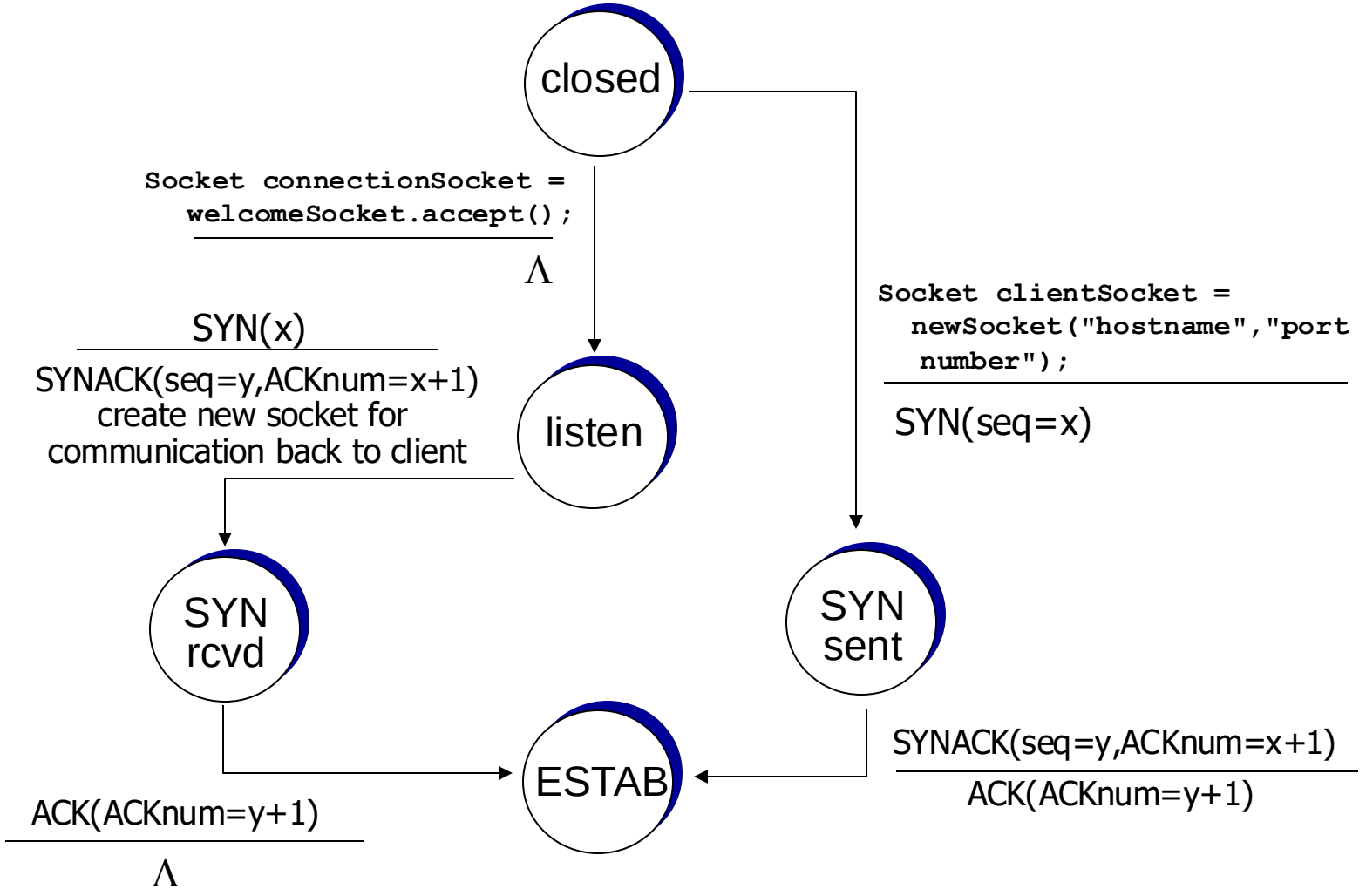
TCP 3-way handshake



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
```

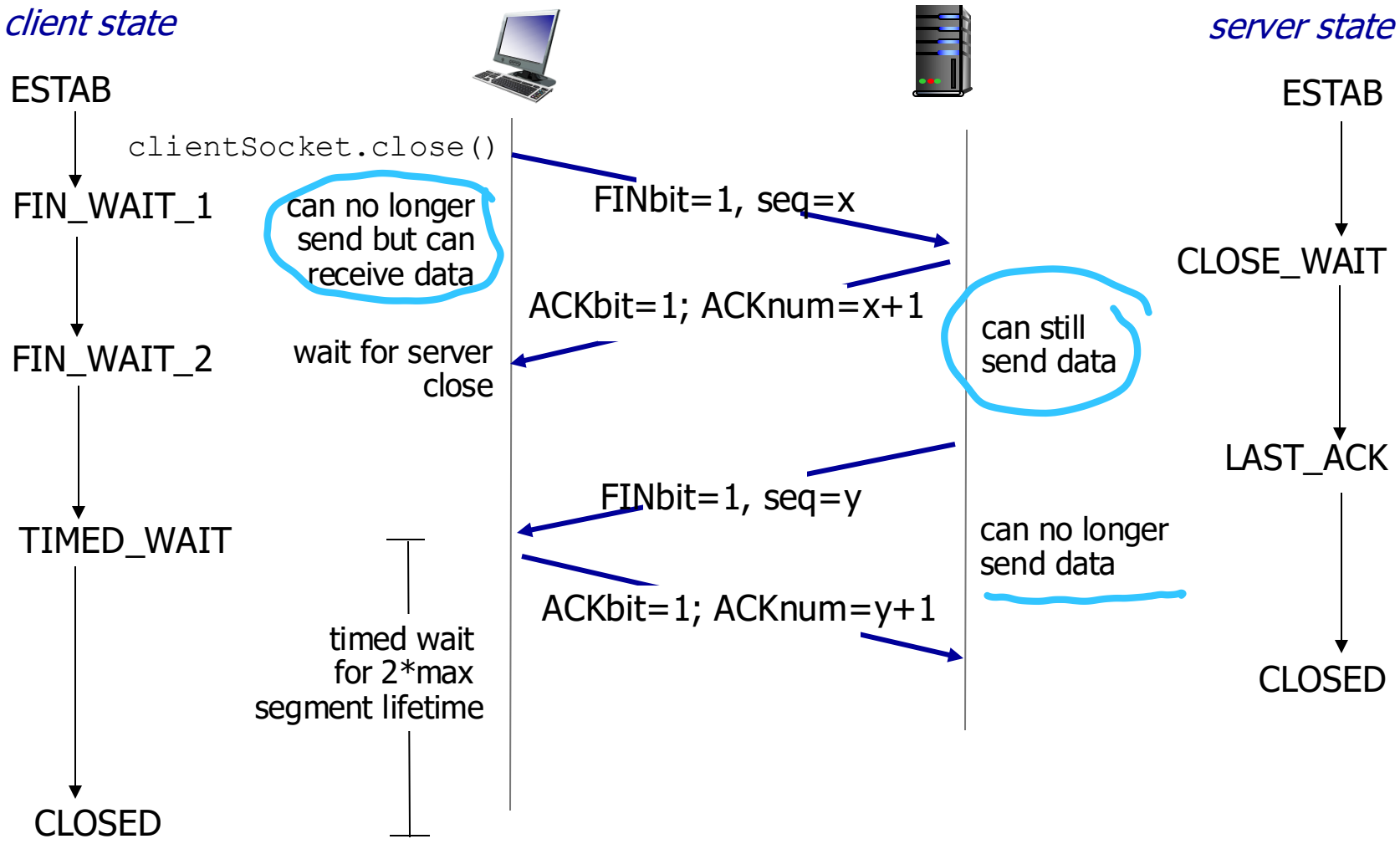
TCP 3-way handshake: FSM

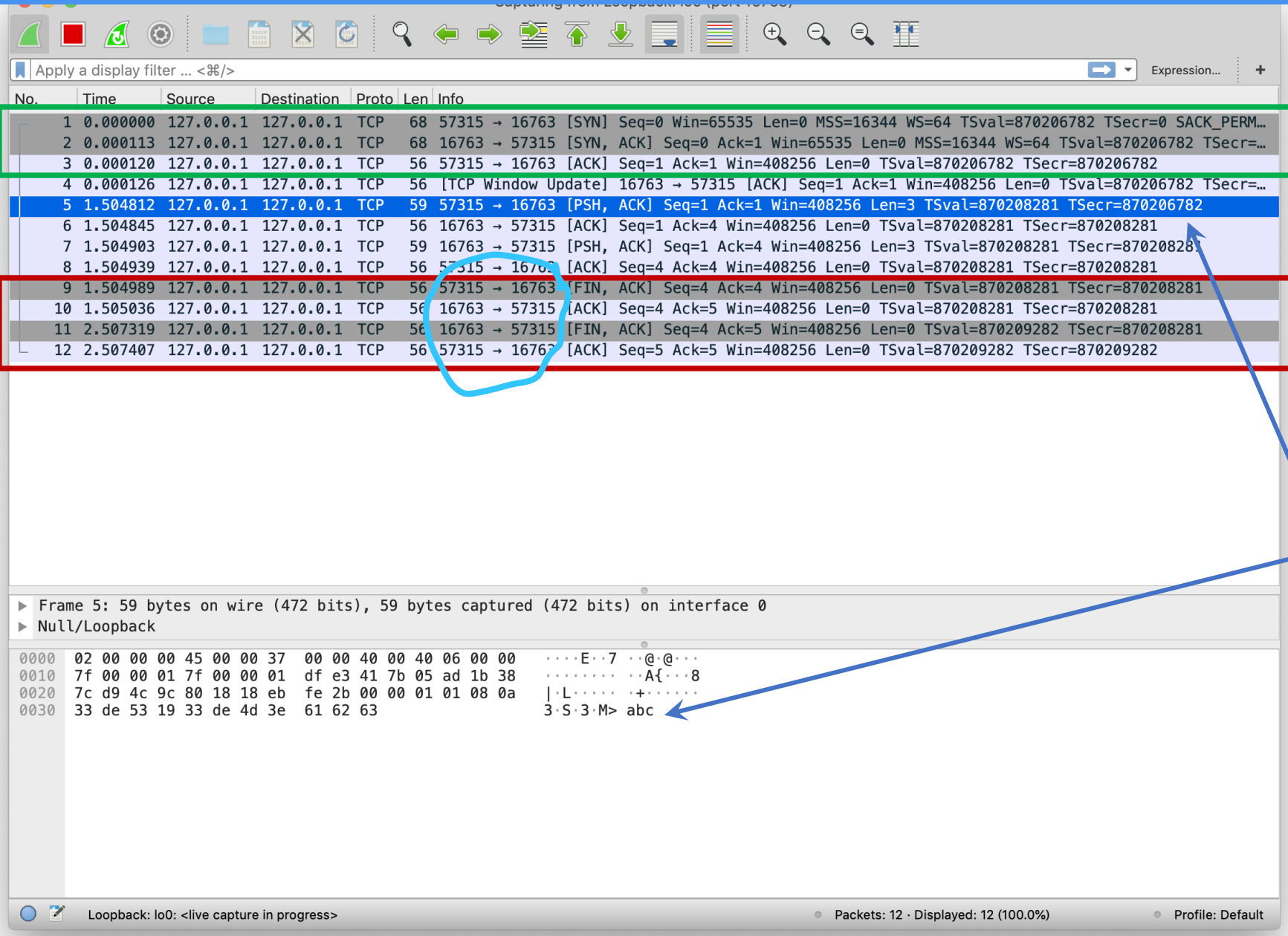


TCP: closing a connection

- **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

Lecture 5 – Transport Layer (1)

- **Roadmap**
  1. Pipelined communication
  2. TCP: connection-oriented transport
  3. Principles of congestion control

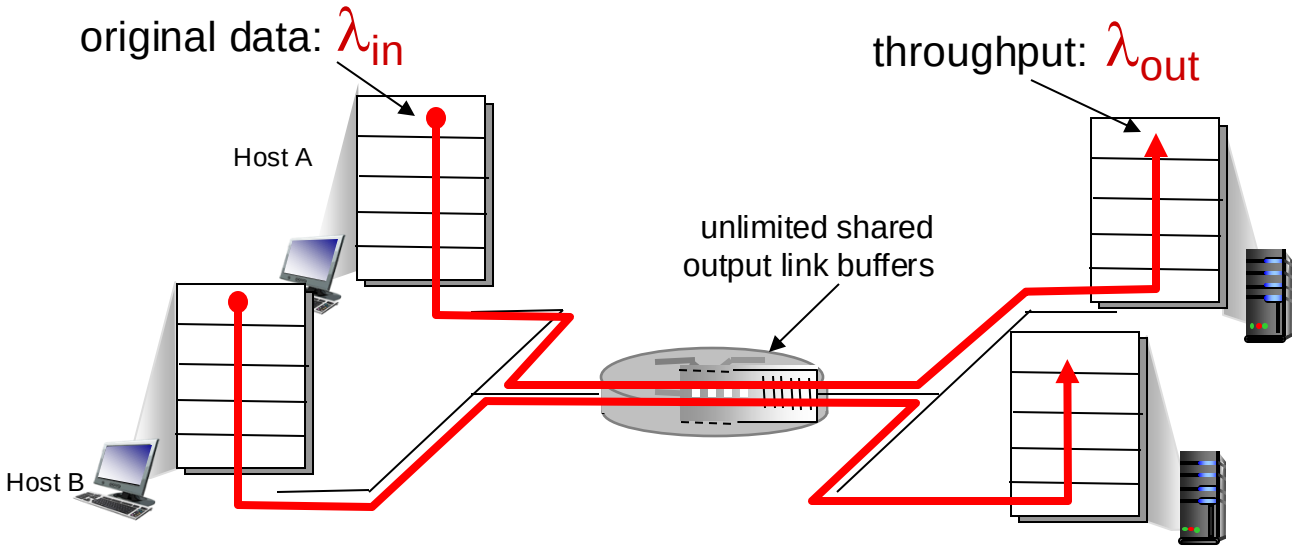
Principles of congestion control

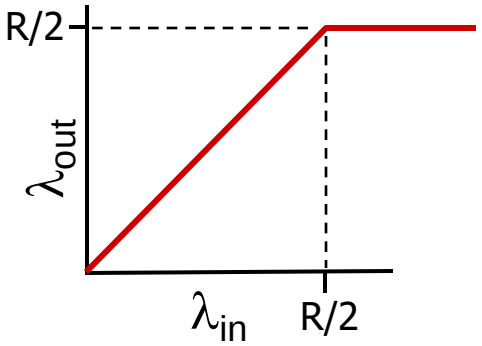
Congestion:

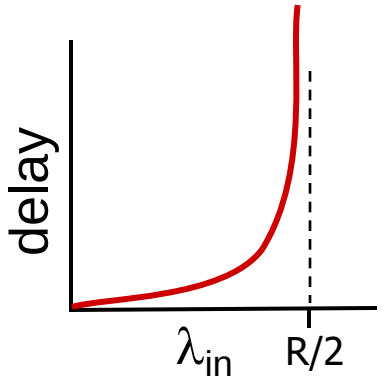
- **Informally: "too many sources sending too much data too fast for *network* to handle"**
- **Different from flow control!**
- **Manifestations:**
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- **A top-10 problem!**

Causes/costs of congestion: scenario 1

- **Two senders, two receivers**
- **One router, infinite buffers**
- **Output link capacity: R**
- **no retransmission**





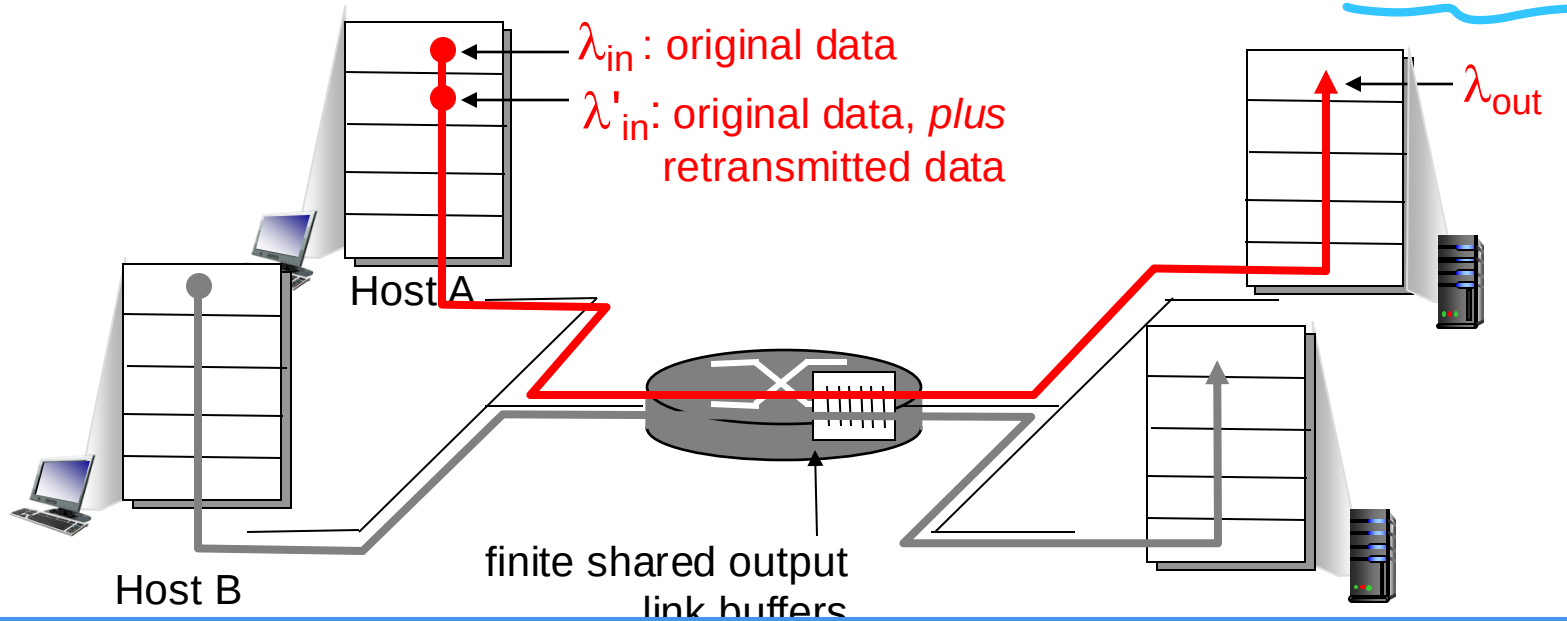


- maximum per-connection throughput: R/2

- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

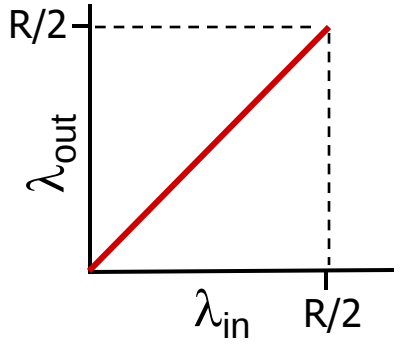
- One router, ***finite*** buffers
- Sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

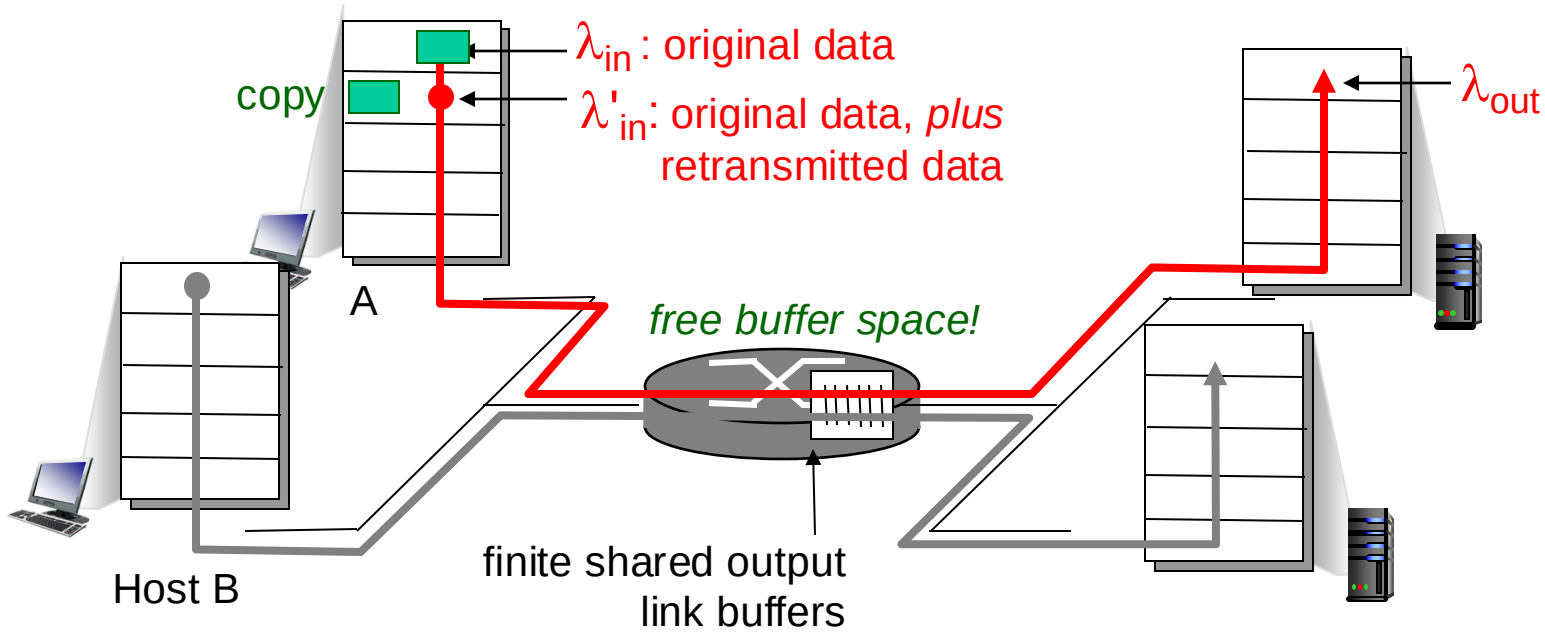


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- Sender sends only when router buffers available



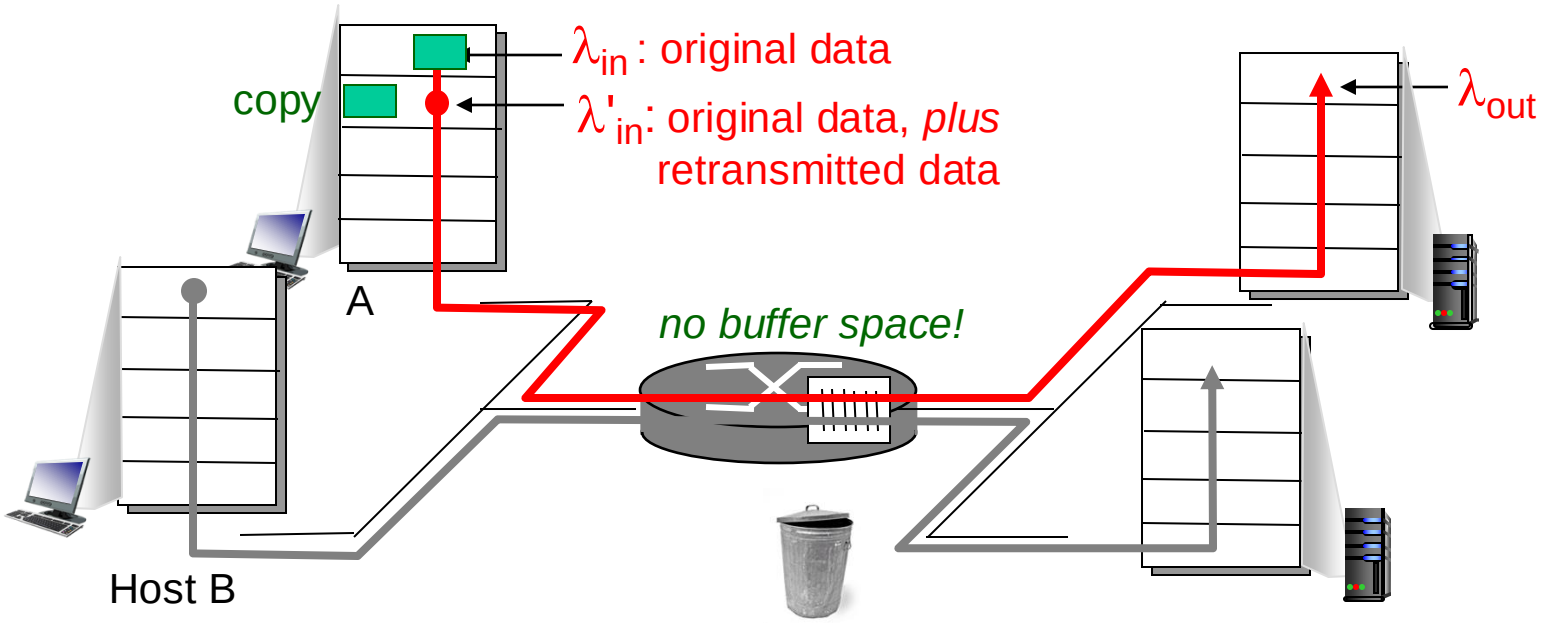


Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- Sender only resends if packet known to be lost

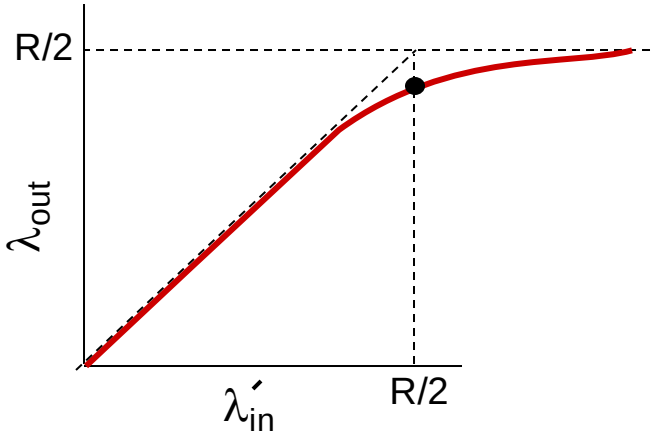


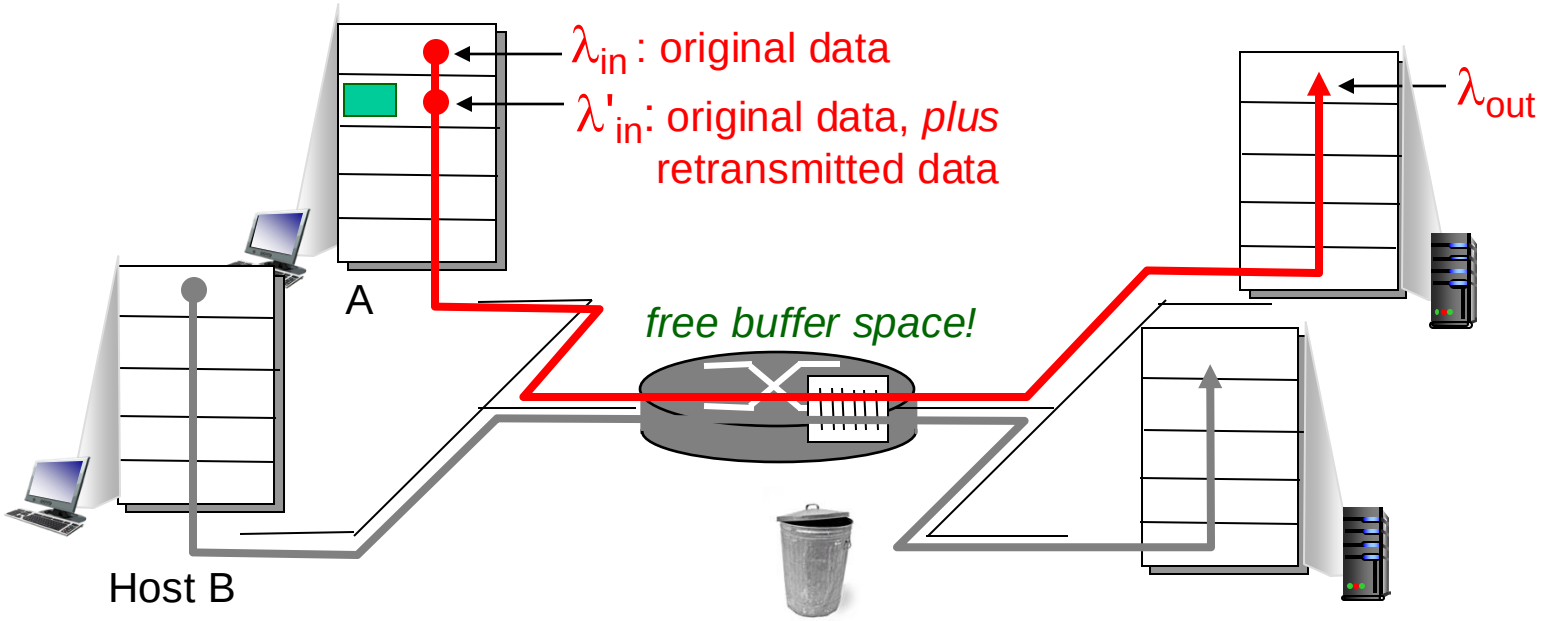
Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- Sender only resends if packet known to be lost

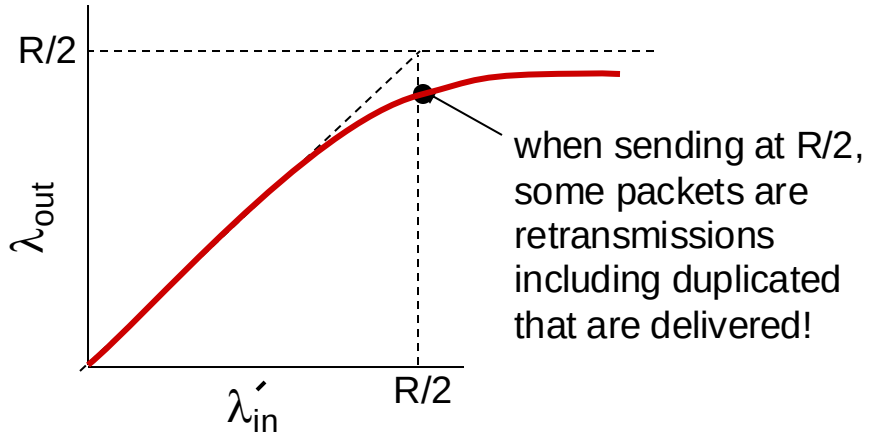


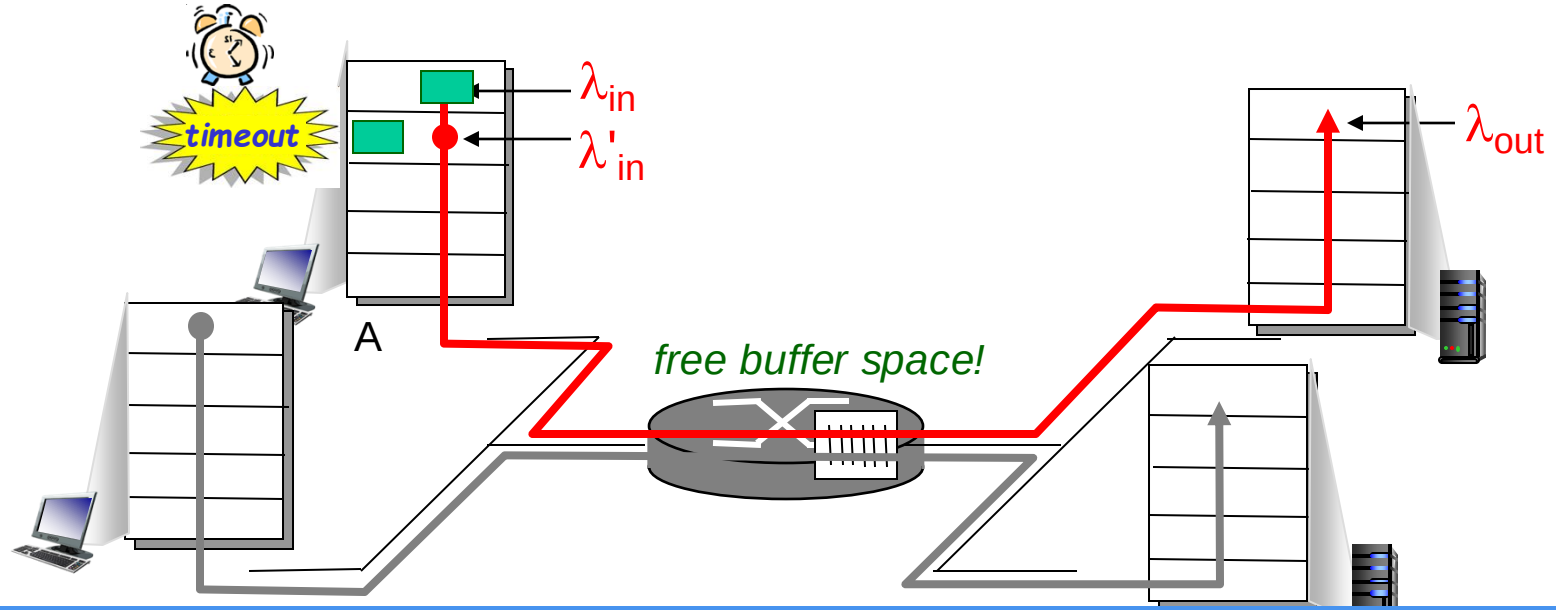


Causes/costs of congestion: scenario 2

Realistic: duplicates

- Packets can be lost, dropped at router due to full buffers
- Sender times out prematurely, sending **two** copies, both of which are delivered

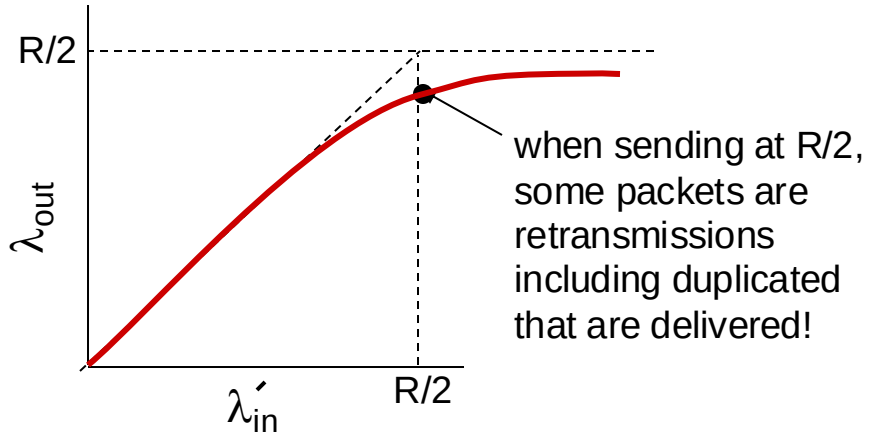




Causes/costs of congestion: scenario 2

Realistic: duplicates

- Packets can be lost, dropped at router due to full buffers
- Sender times out prematurely, sending **two** copies, both of which are delivered



"costs" of congestion:

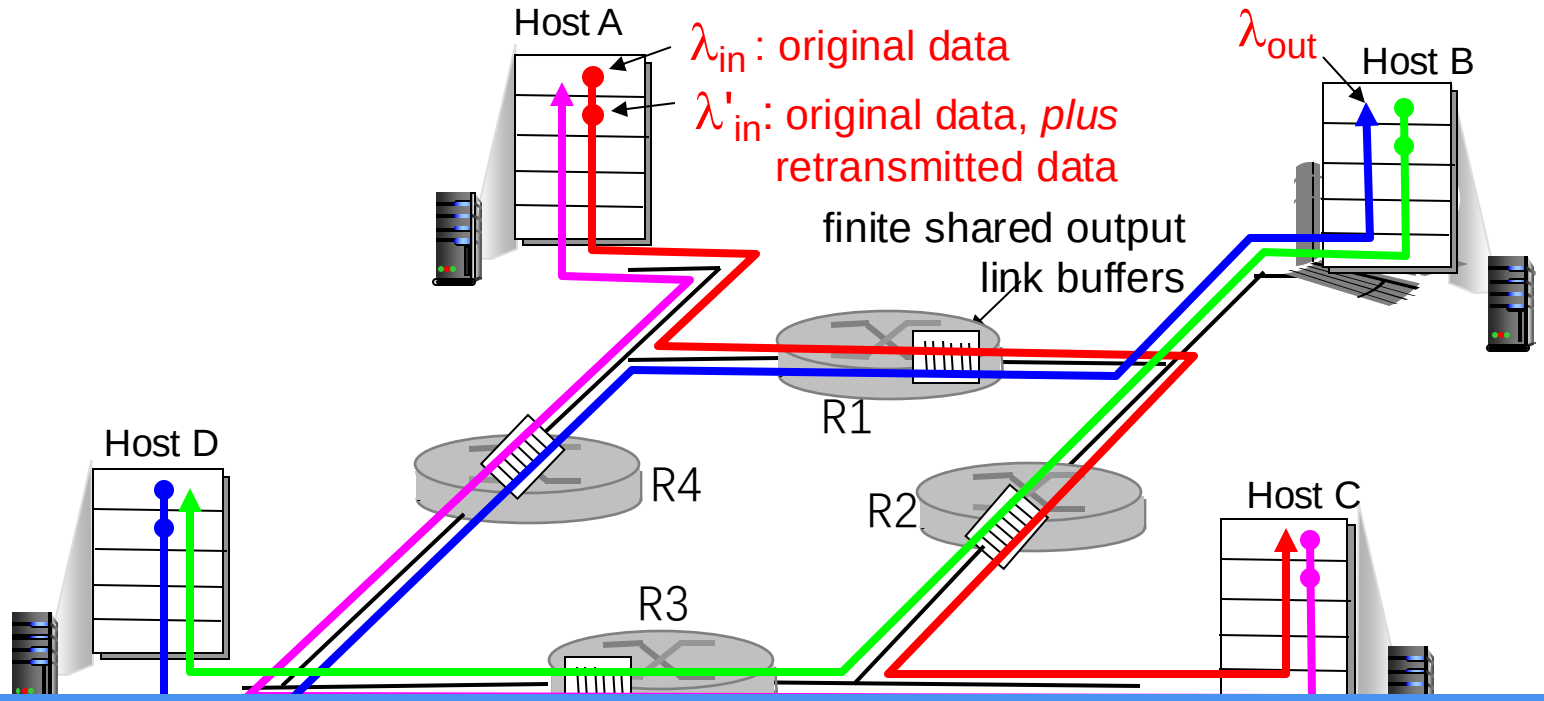
- More work (retrans) for given "goodput"
- Unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

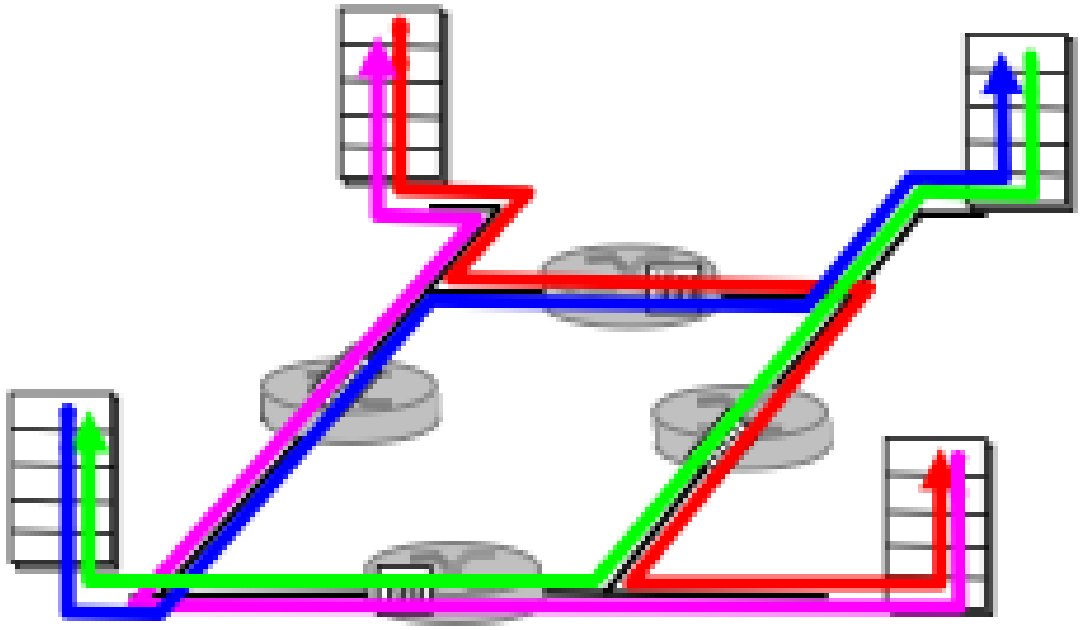
- Four senders
- Multi-hop paths
- Timeout/retransmit

Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput -> 0



Causes/costs of congestion: scenario 3



Another “cost” of congestion:

- When packet dropped, any “upstream transmission capacity used for that packet was wasted!

Approaches to Congestion Control

- **End-to-end congestion control**
  - Network layer does not provide support for congestion control
  - Trans layer has to infer from network behavior
  - TCP will control the size of window
- **Network-assisted congestion control**
  - Routers provide feedback to sender and/or receiver (a single one bit)

Thanks.