



Introduction to Networking

CAN201 – Week 5
Module Leader: Dr. Wenjun Fan & Dr. Fei Cheng

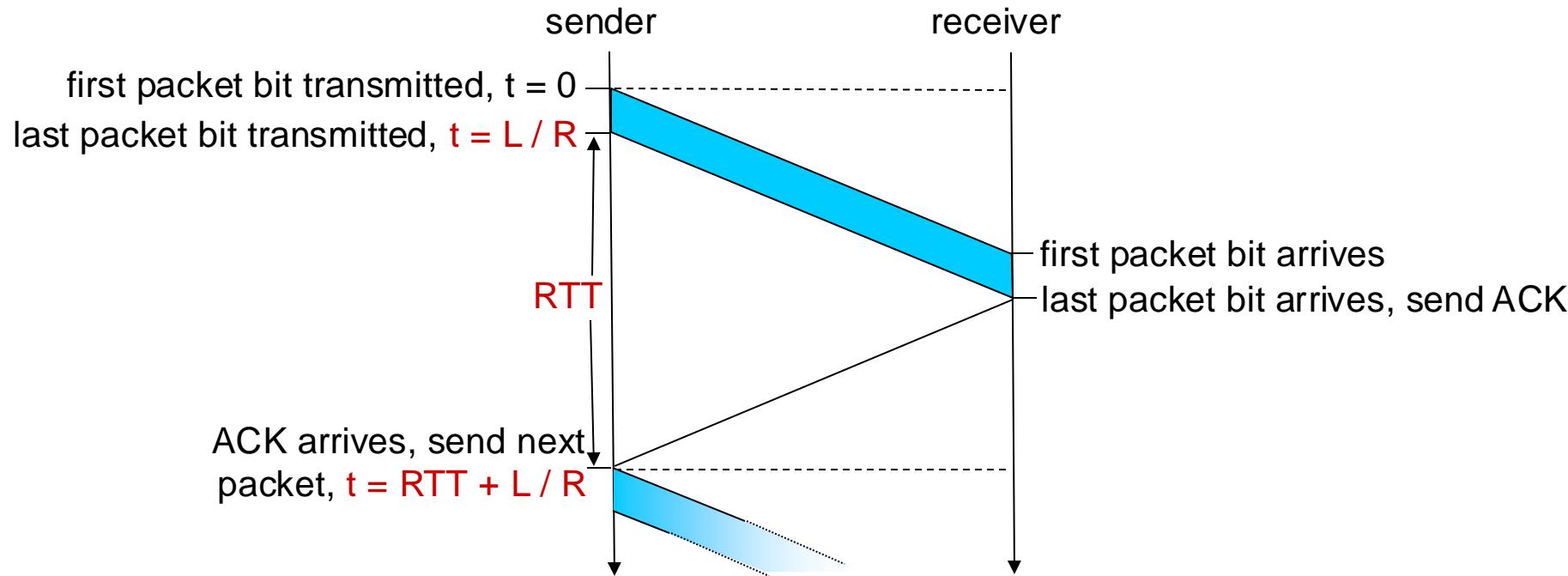
Lecture 5 – Transport Layer (2)

- **Roadmap**

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



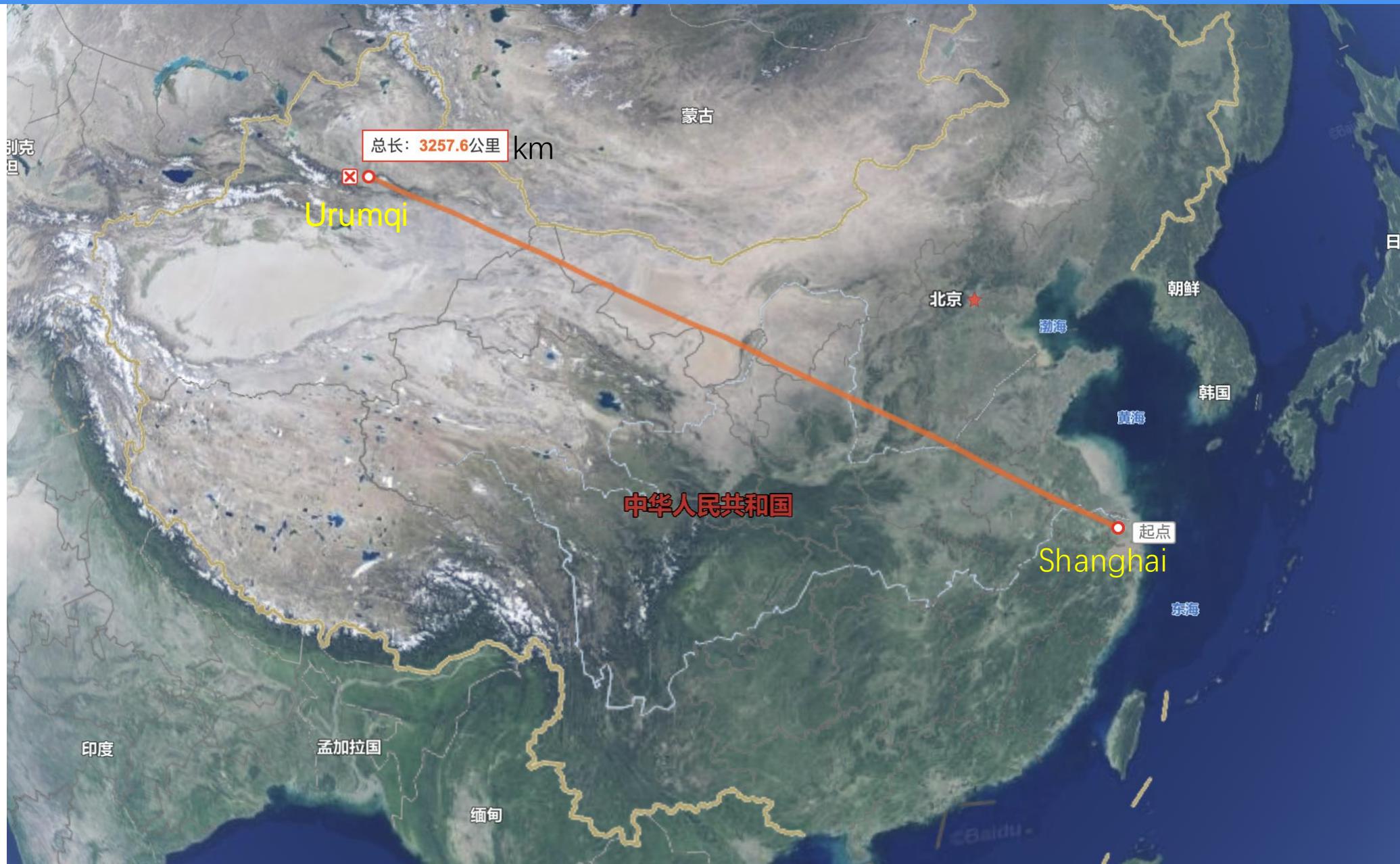
rdt3.0: stop-and-wait operation



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

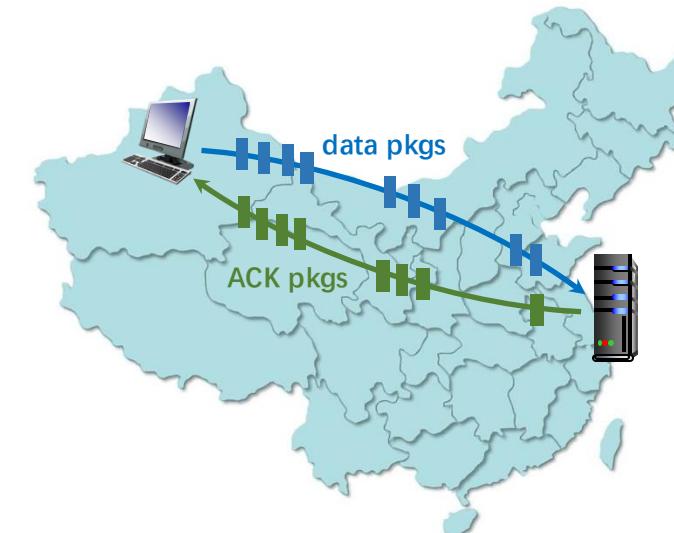
Question

- How to increase utilization?

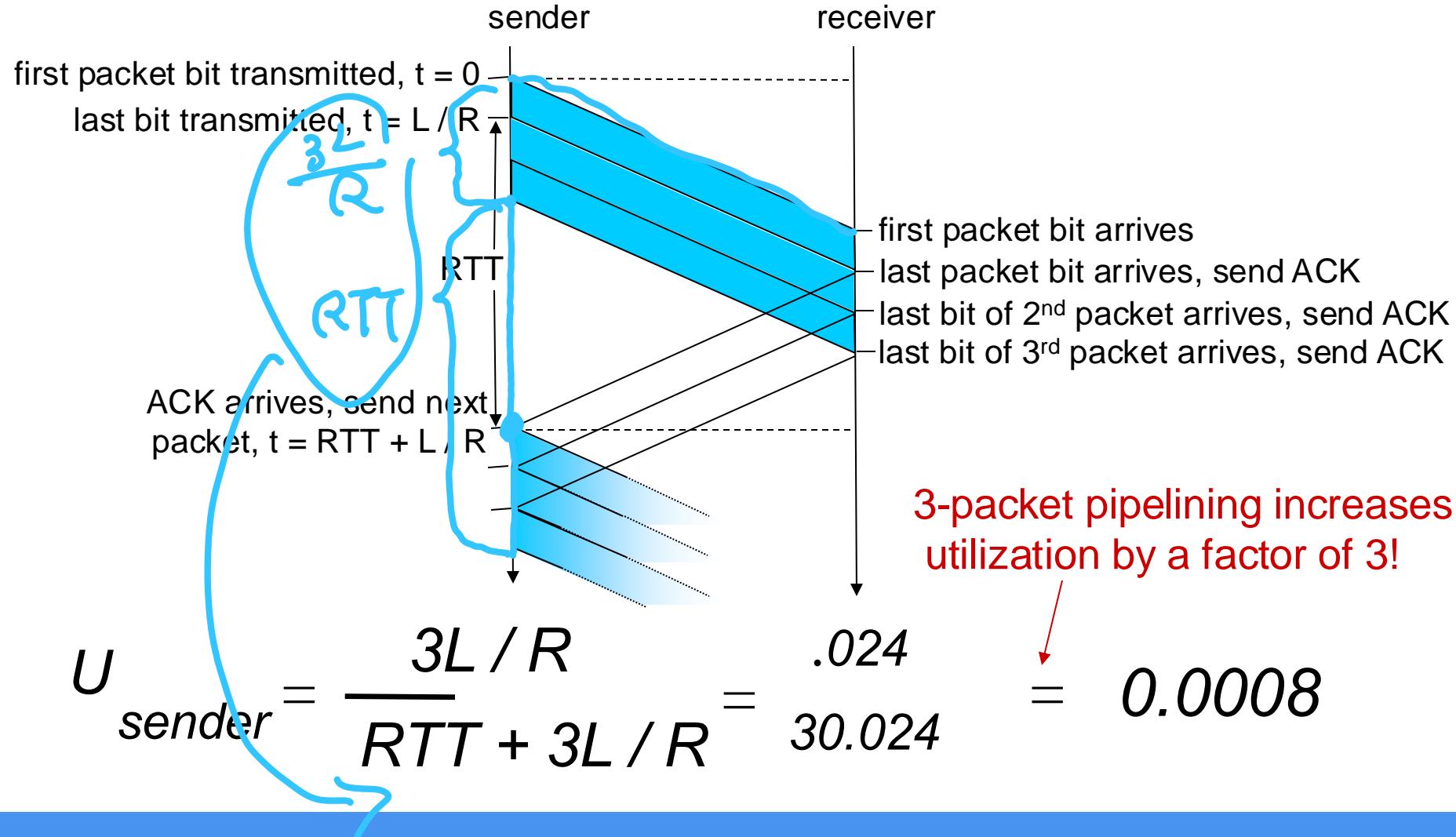


Pipelined protocols 流水线协议

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



Pipelining: increased utilization



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 ✗

发送报文序号，发送到2报文，接收器一直
接收并确认，直到接收到2报文

Selective Repeat (SR):

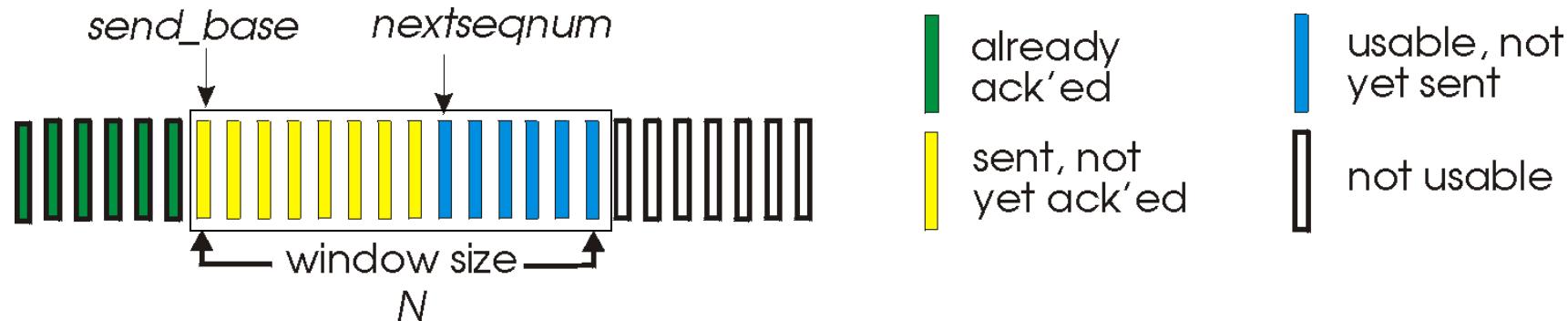
- Sender can have up to N unacked packets in pipeline
- Rcvr sends **individual ack** for each packet

- Sender maintains timer for each unacked packet
 - When timer expires, retransmit only that unacked packet

Go-Back-N: sender

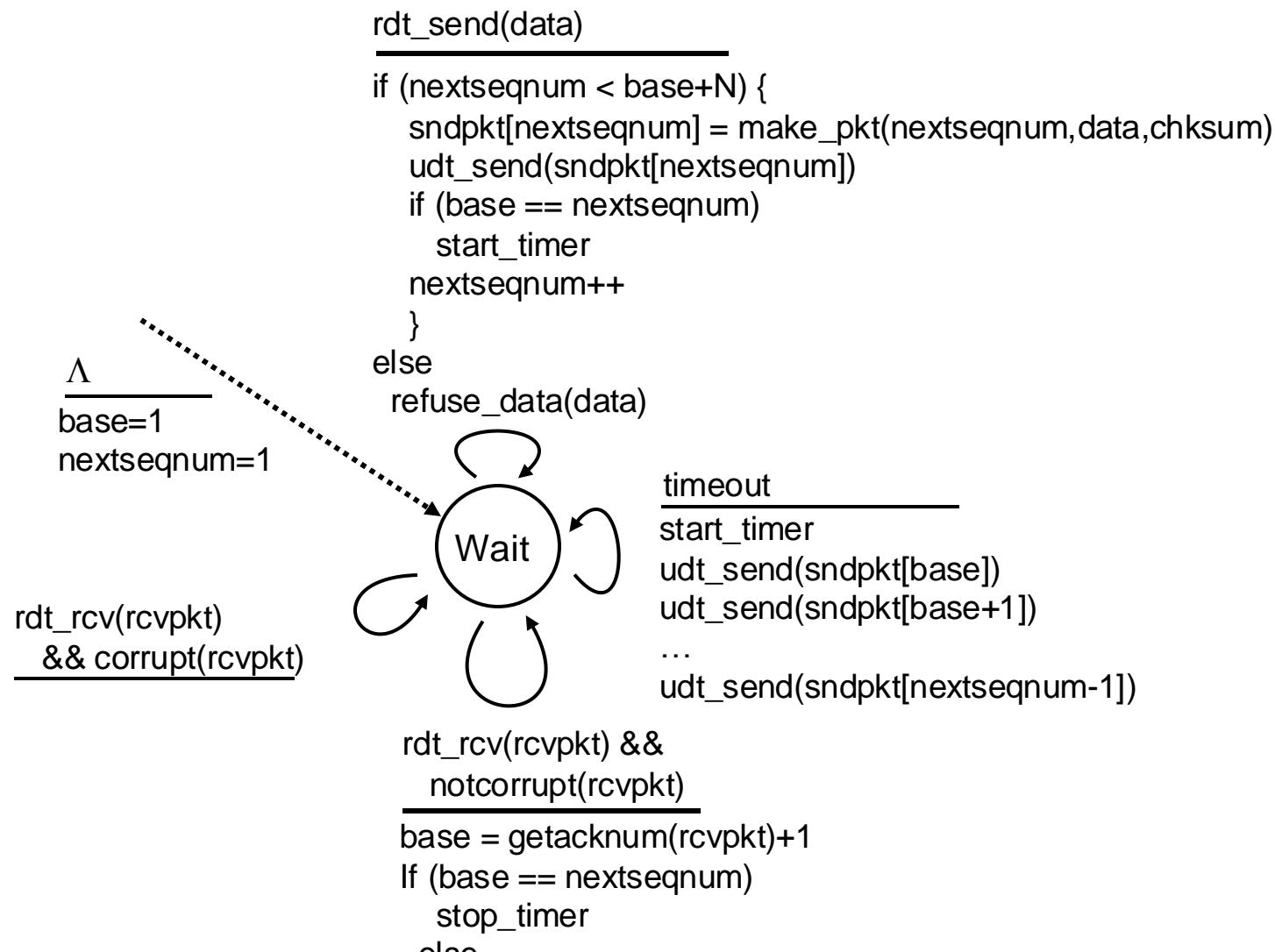


- k-bit seq # in pkt header
- “window” of up to N, consecutive unacked pkts allowed



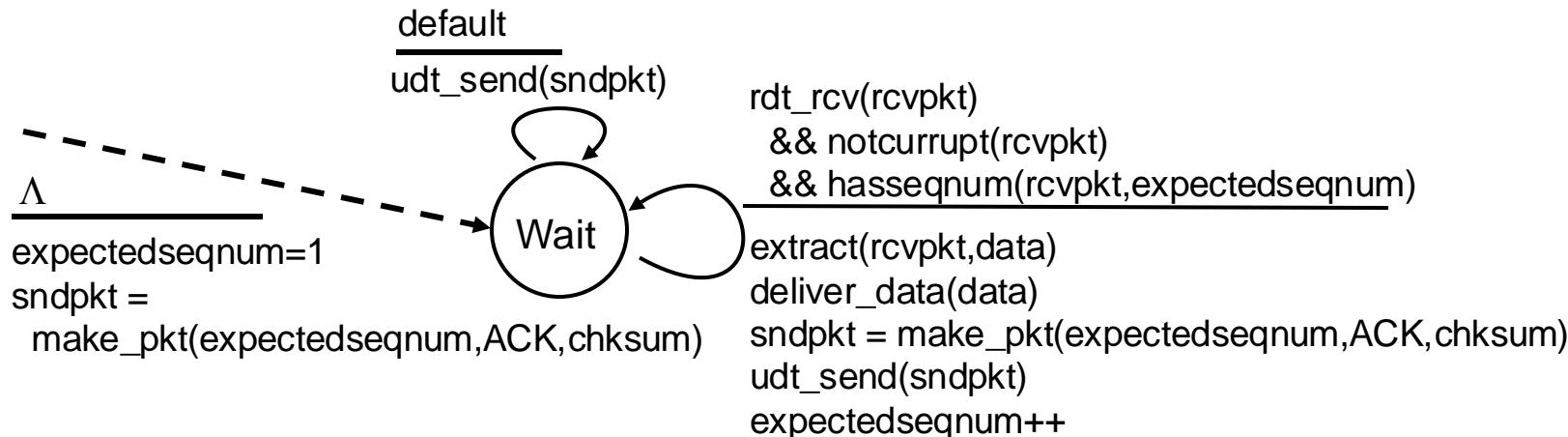
- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
 - may receive duplicate ACKs (see receiver)
- Timer for oldest in-flight pkt
- Timeout(n): retransmit packet n and all higher seq # pkts in window

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

sender window (N=4)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

send pkt2
send pkt3
send pkt4
send pkt5

receiver

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, **discard**, **丢弃**
(re)send ack1

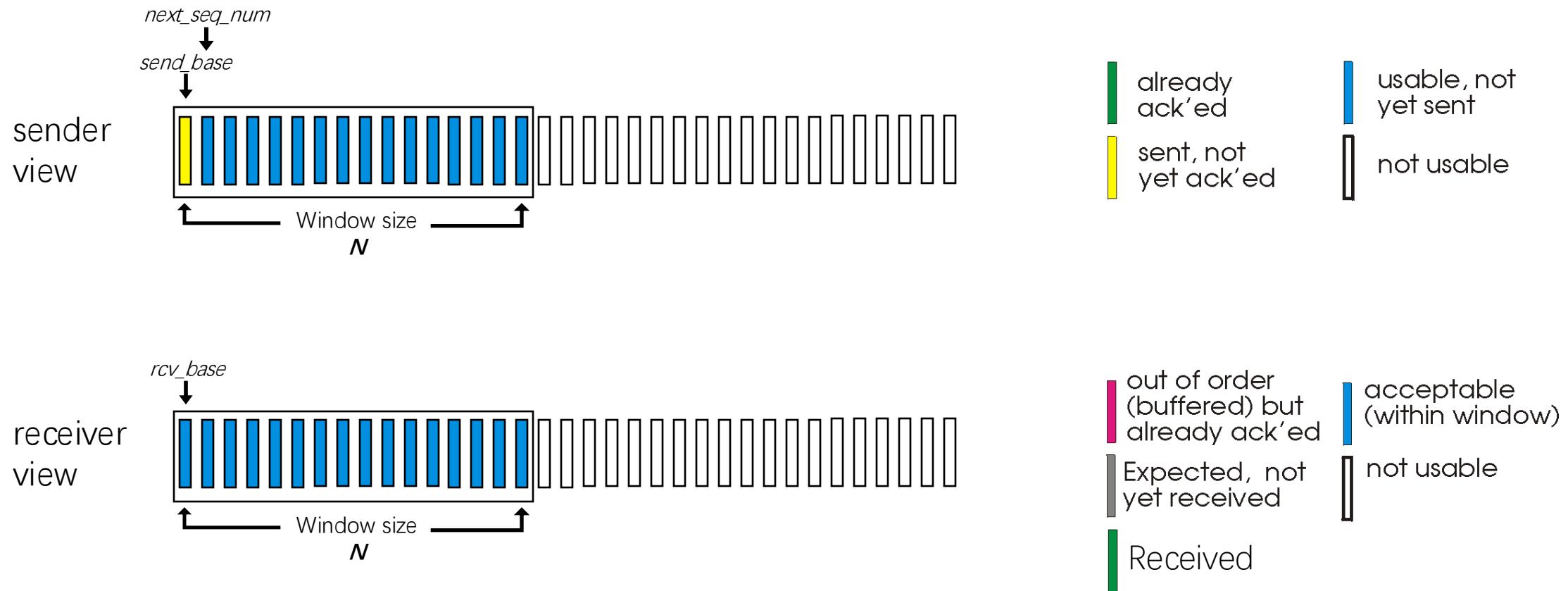
receive pkt4, **discard**,
(re)send ack1
receive pkt5, **discard**,
(re)send ack1

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

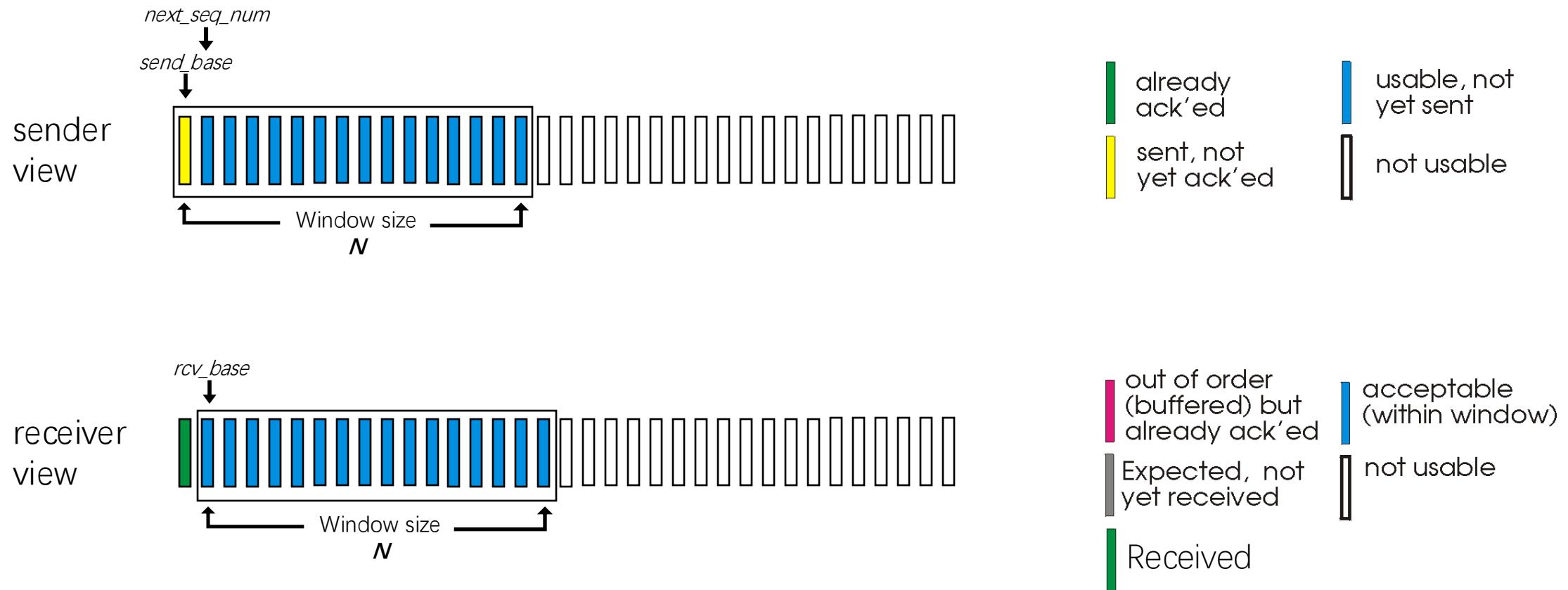
Selective repeat

- 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

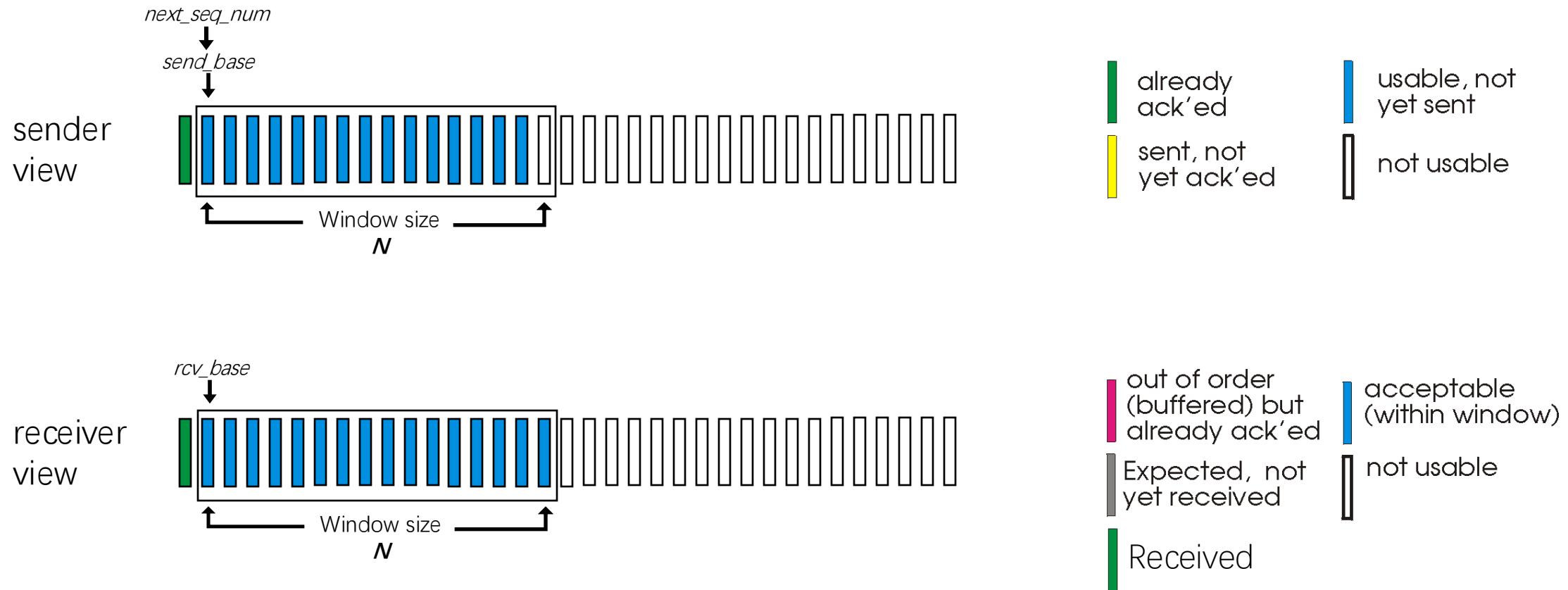
Selective repeat: sender, receiver windows



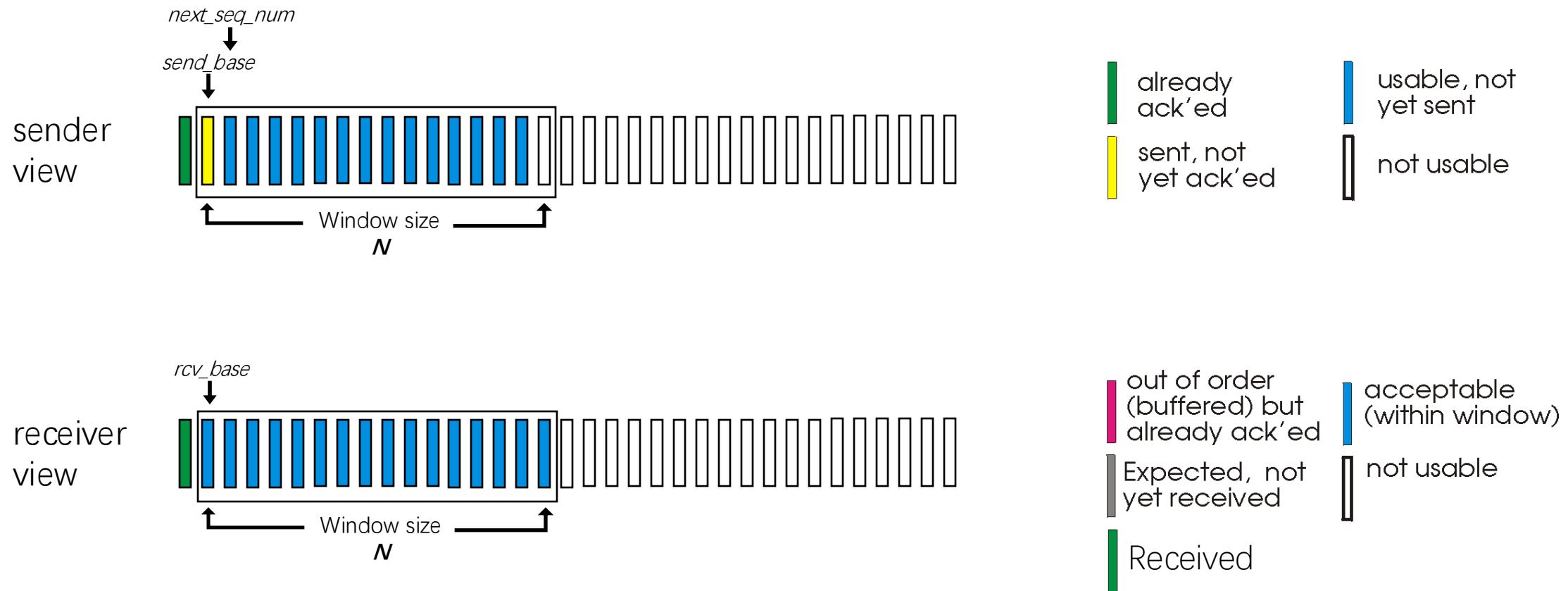
Selective repeat: sender, receiver windows



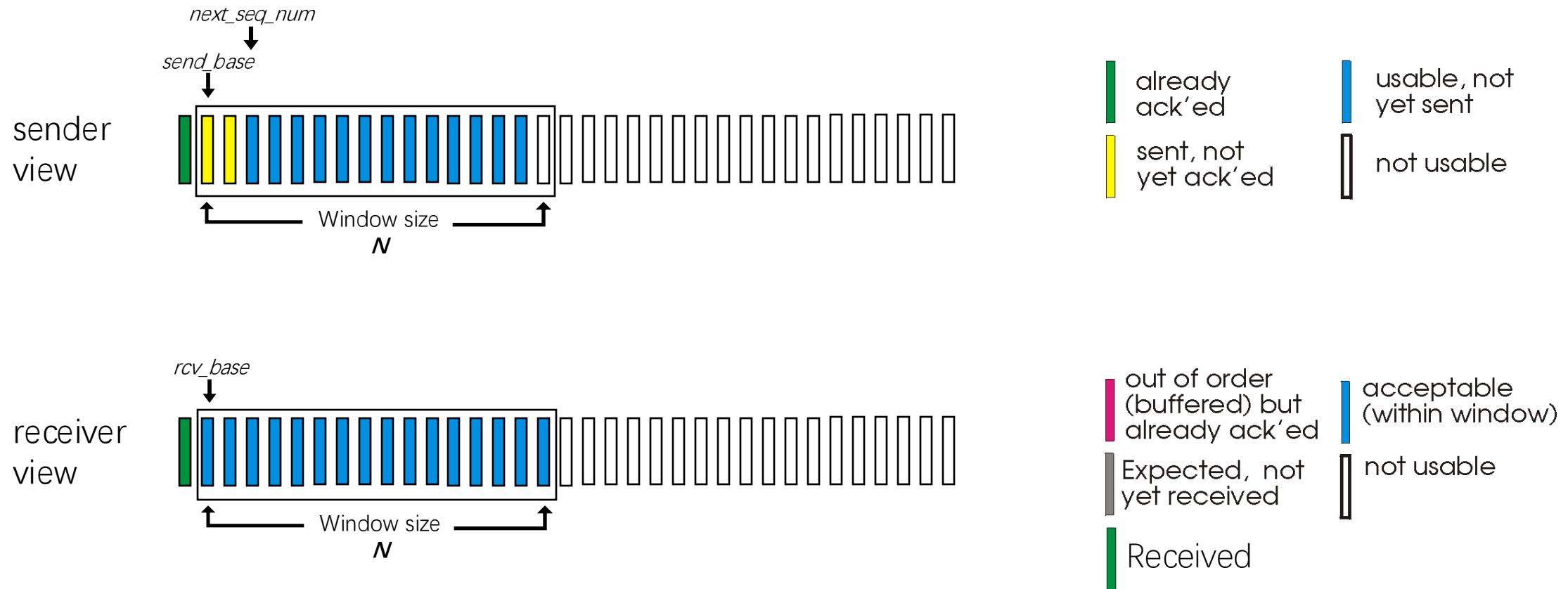
Selective repeat: sender, receiver windows



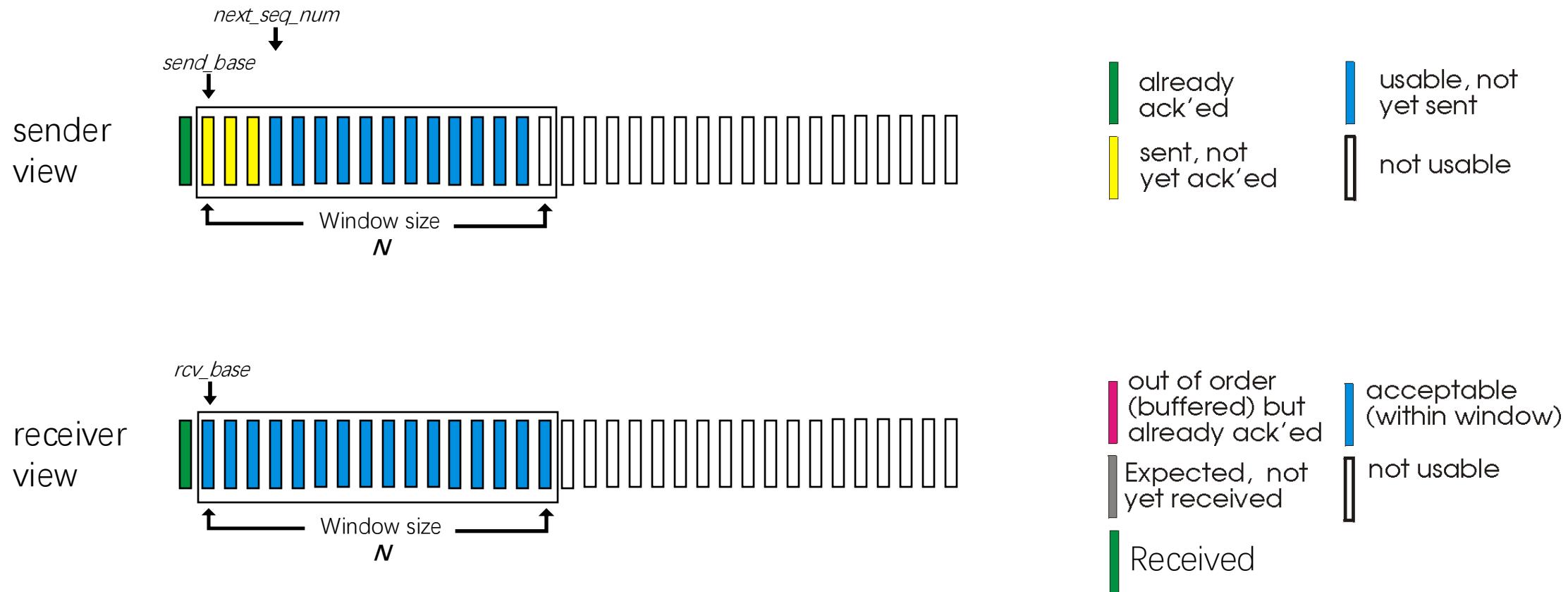
Selective repeat: sender, receiver windows



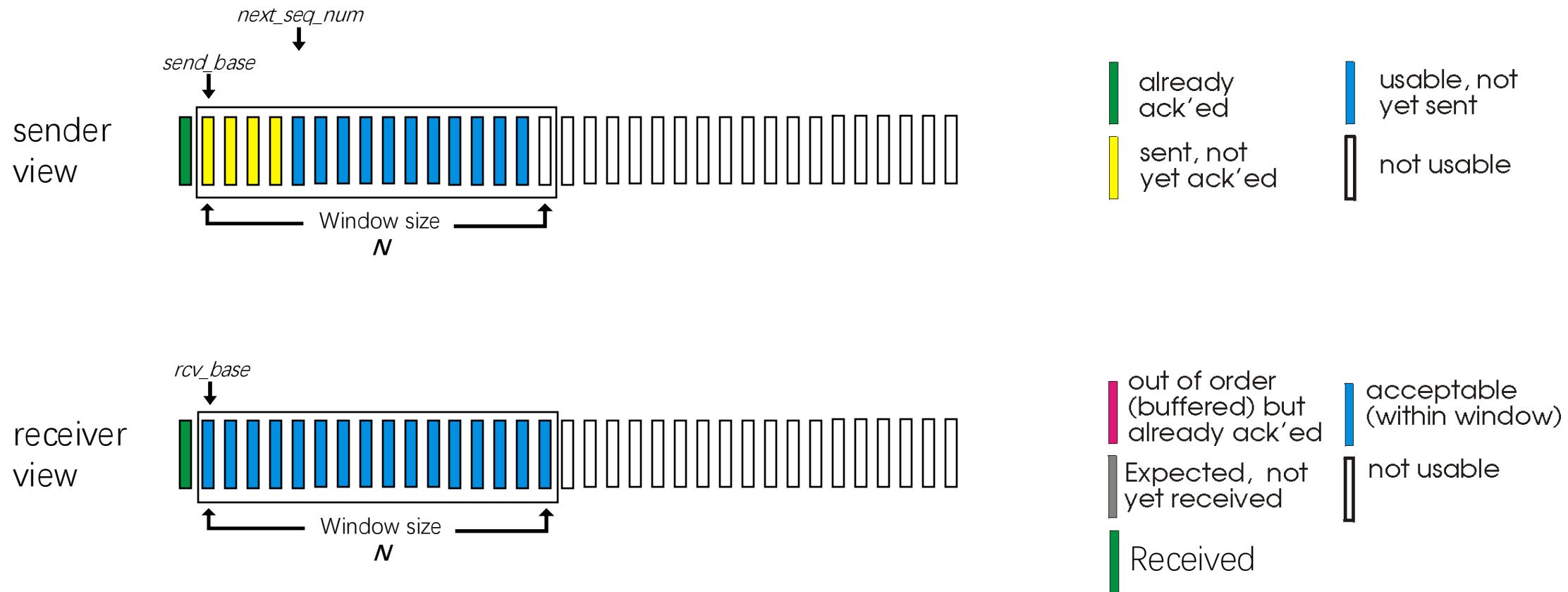
Selective repeat: sender, receiver windows



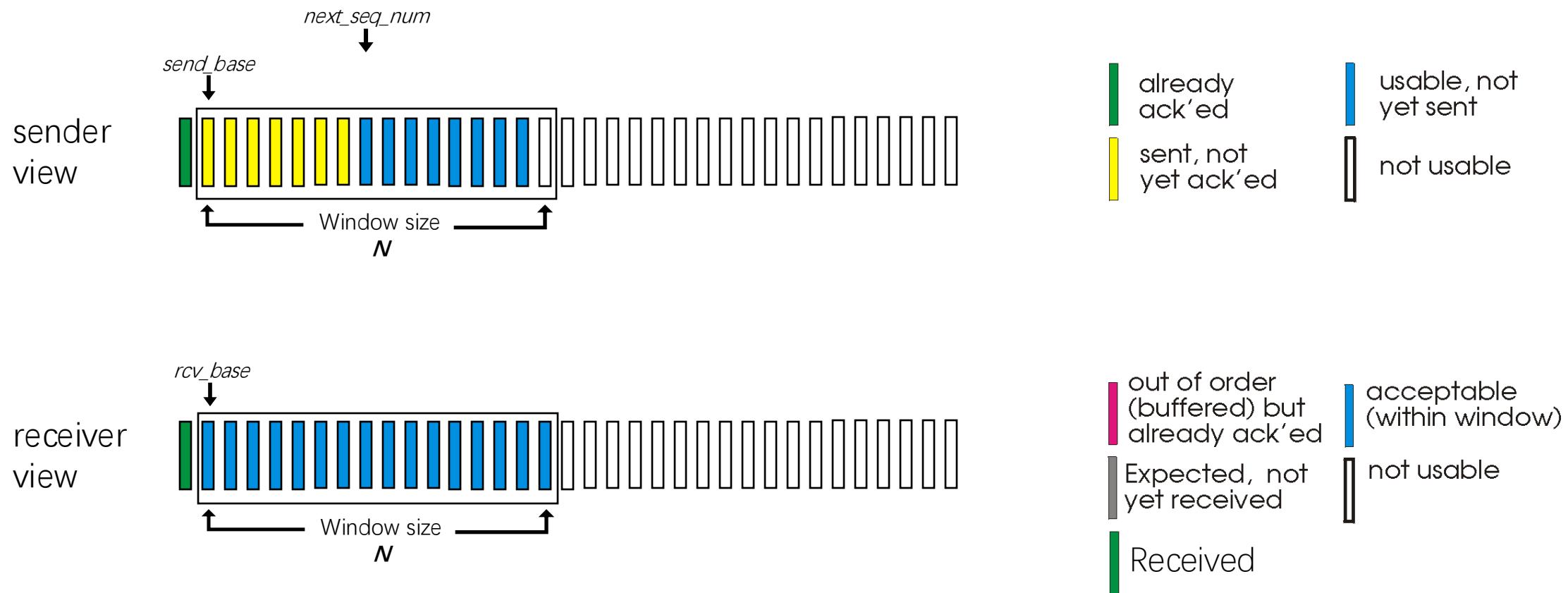
Selective repeat: sender, receiver windows



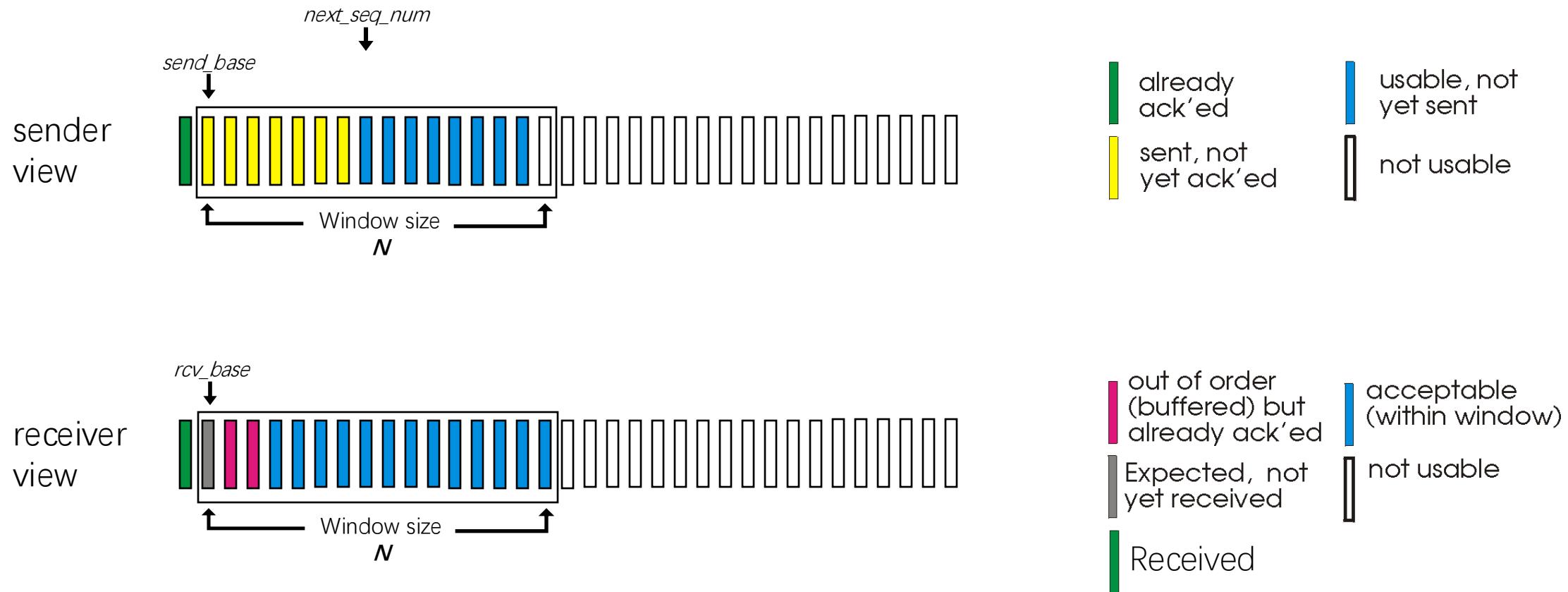
Selective repeat: sender, receiver windows



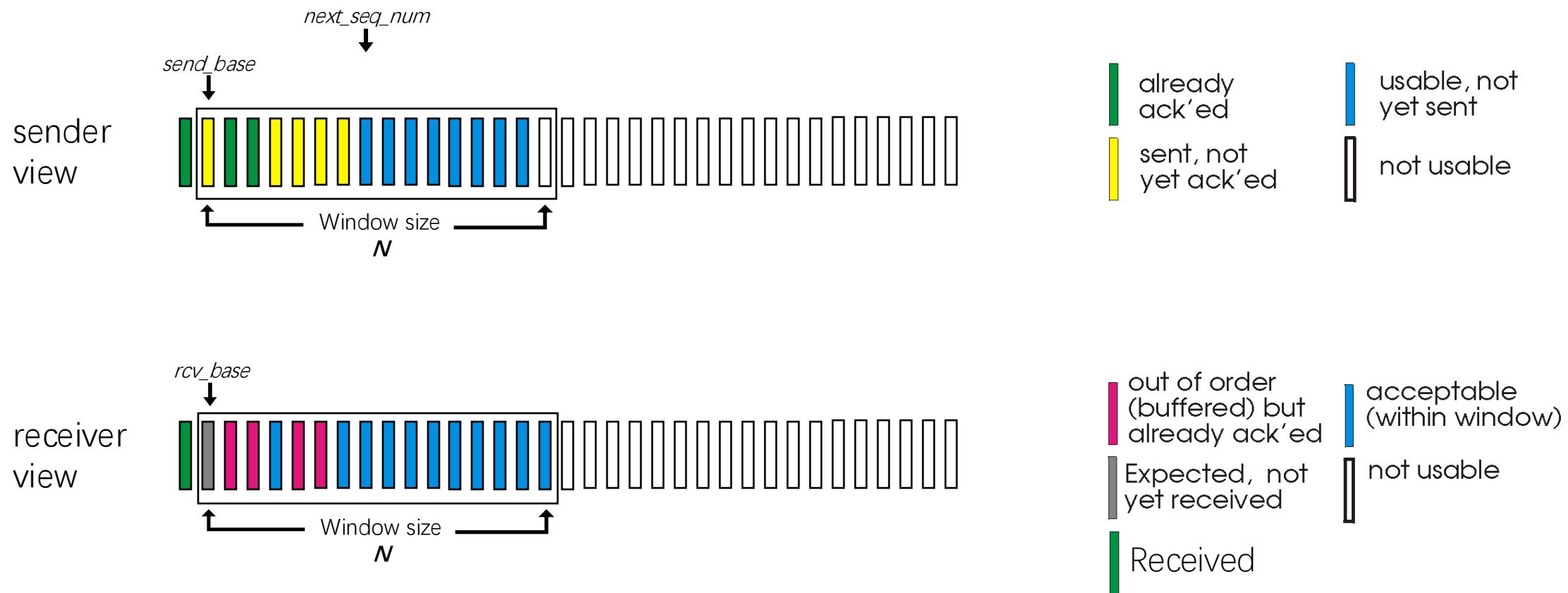
Selective repeat: sender, receiver windows



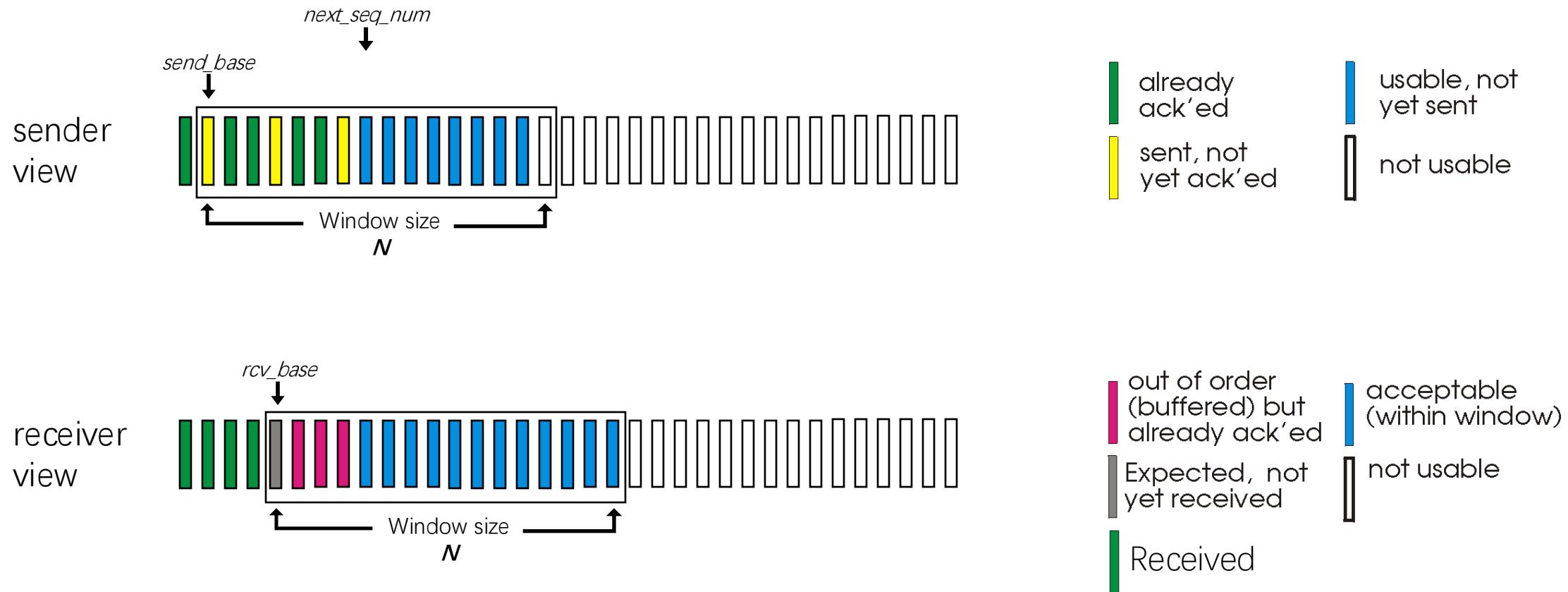
Selective repeat: sender, receiver windows



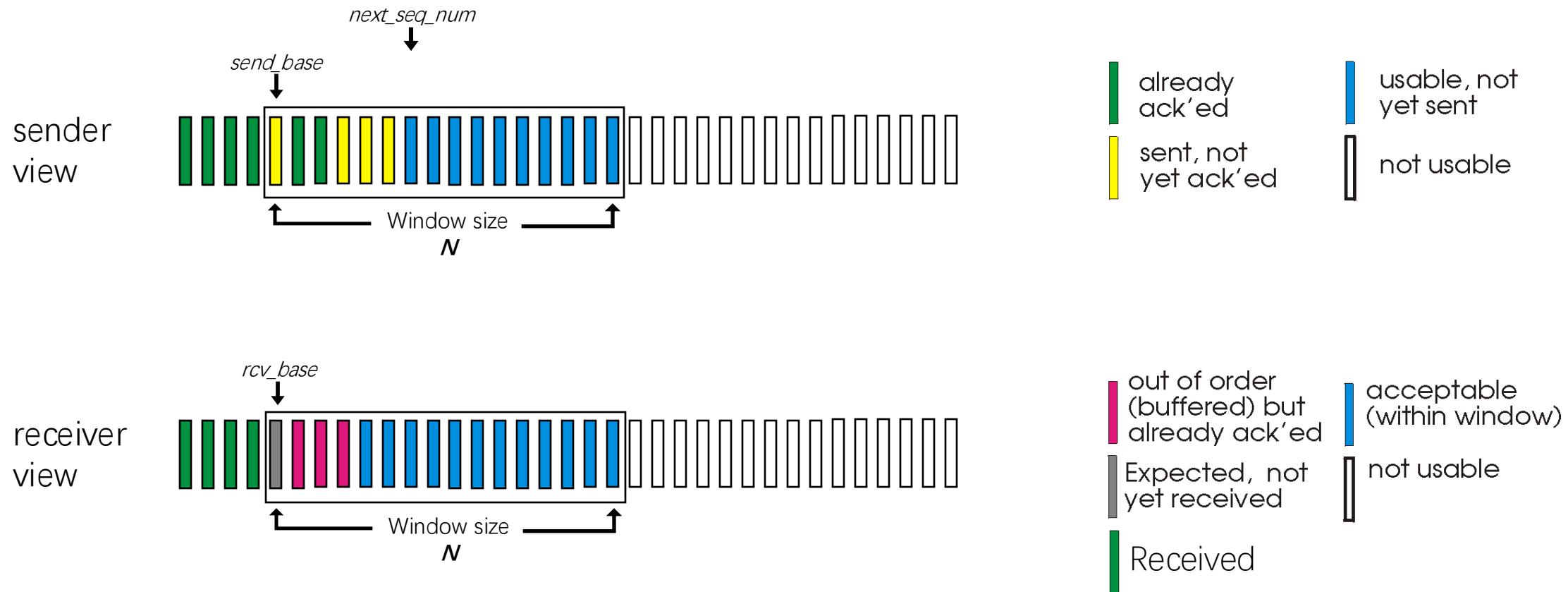
Selective repeat: sender, receiver windows



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

sender window (N=4)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived

 **pkt 2 timeout**
send pkt2

record ack4 arrived
record ack5 arrived

Q: what happens when ack2 arrives?

receiver

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4

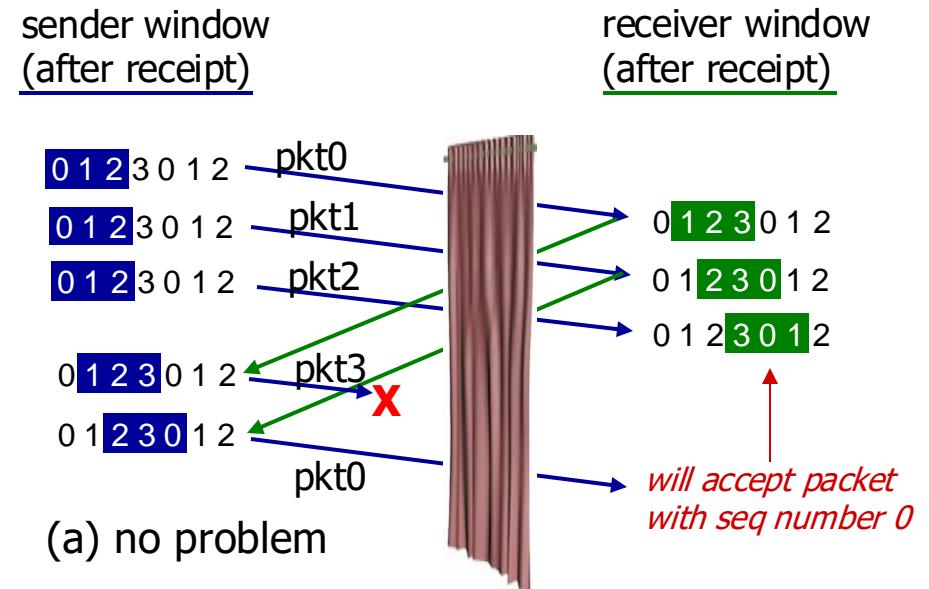
receive pkt5, buffer,
send ack5

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

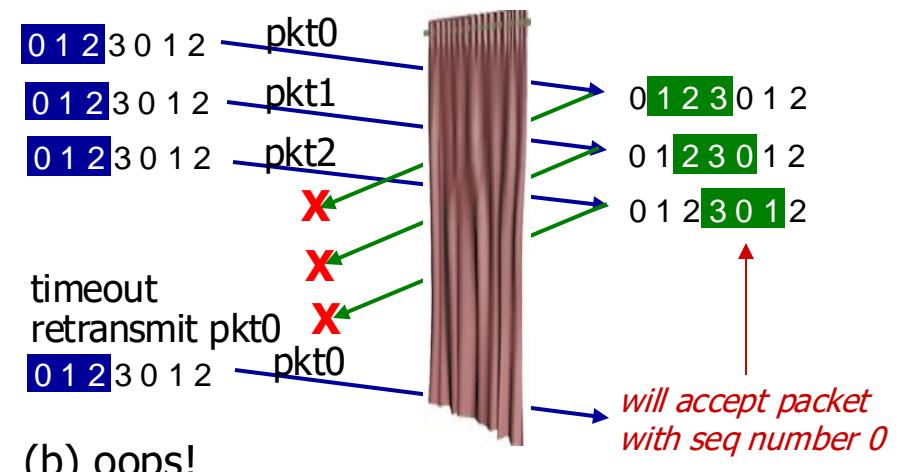
Selective repeat: dilemma

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**

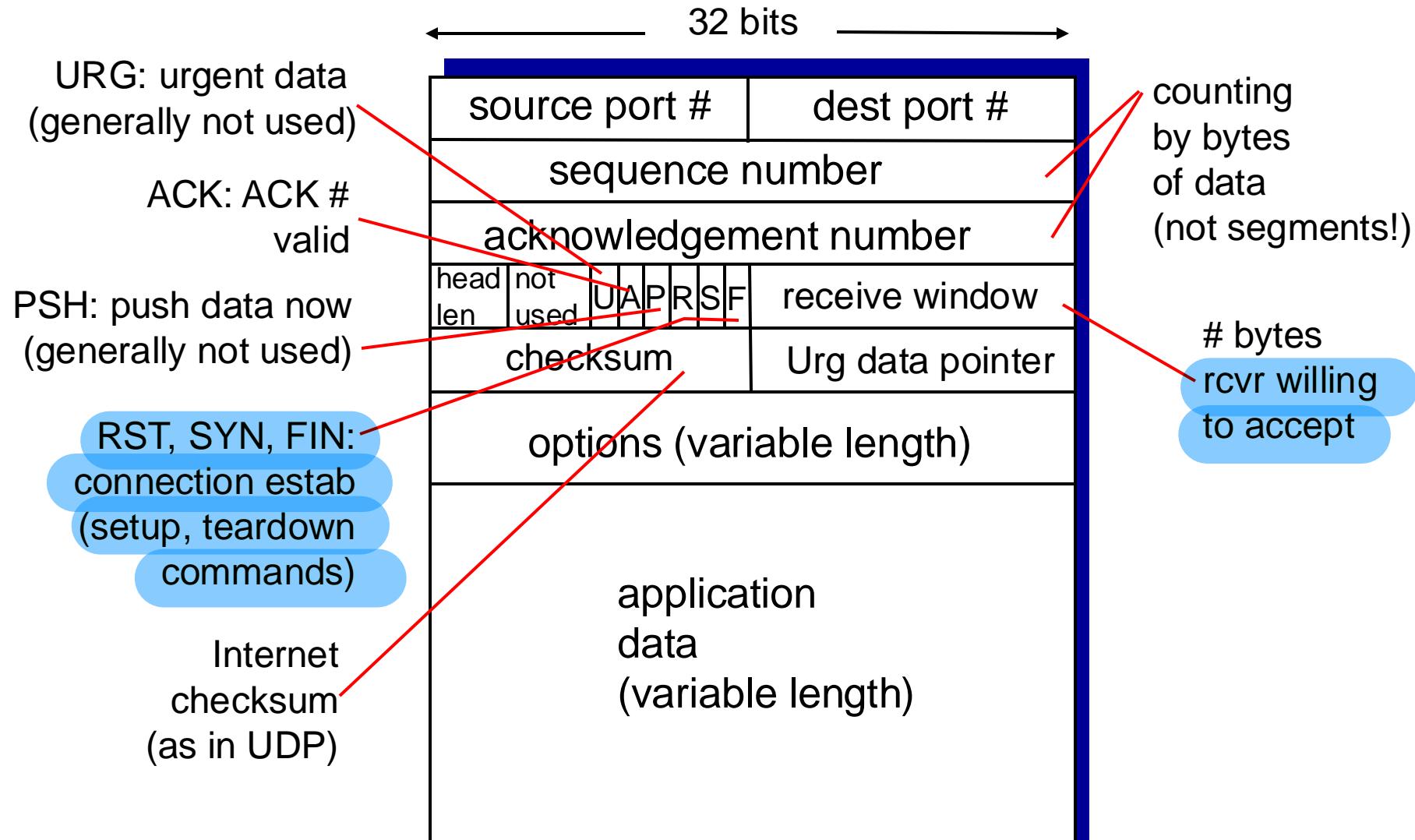
1. Pipelined communication
2. TCP: connection-oriented transport
3. Principles of congestion control



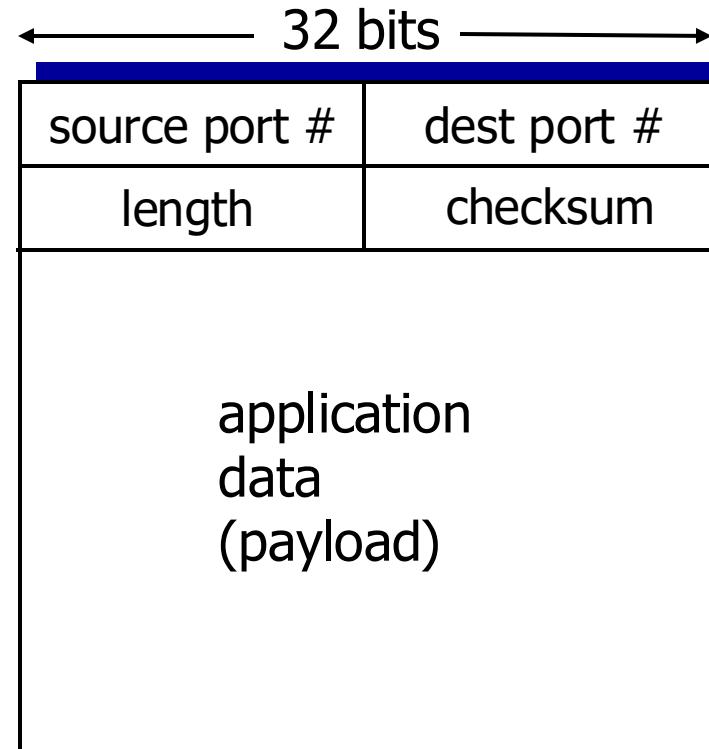
TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
 - one sender, one receiver
可靠, 有序字节流
- Reliable, in-order byte stream :
 - no “message boundaries”
- Pipelined:
 - TCP congestion and flow control
set window size
- Full duplex data:
双向数据流
 - 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

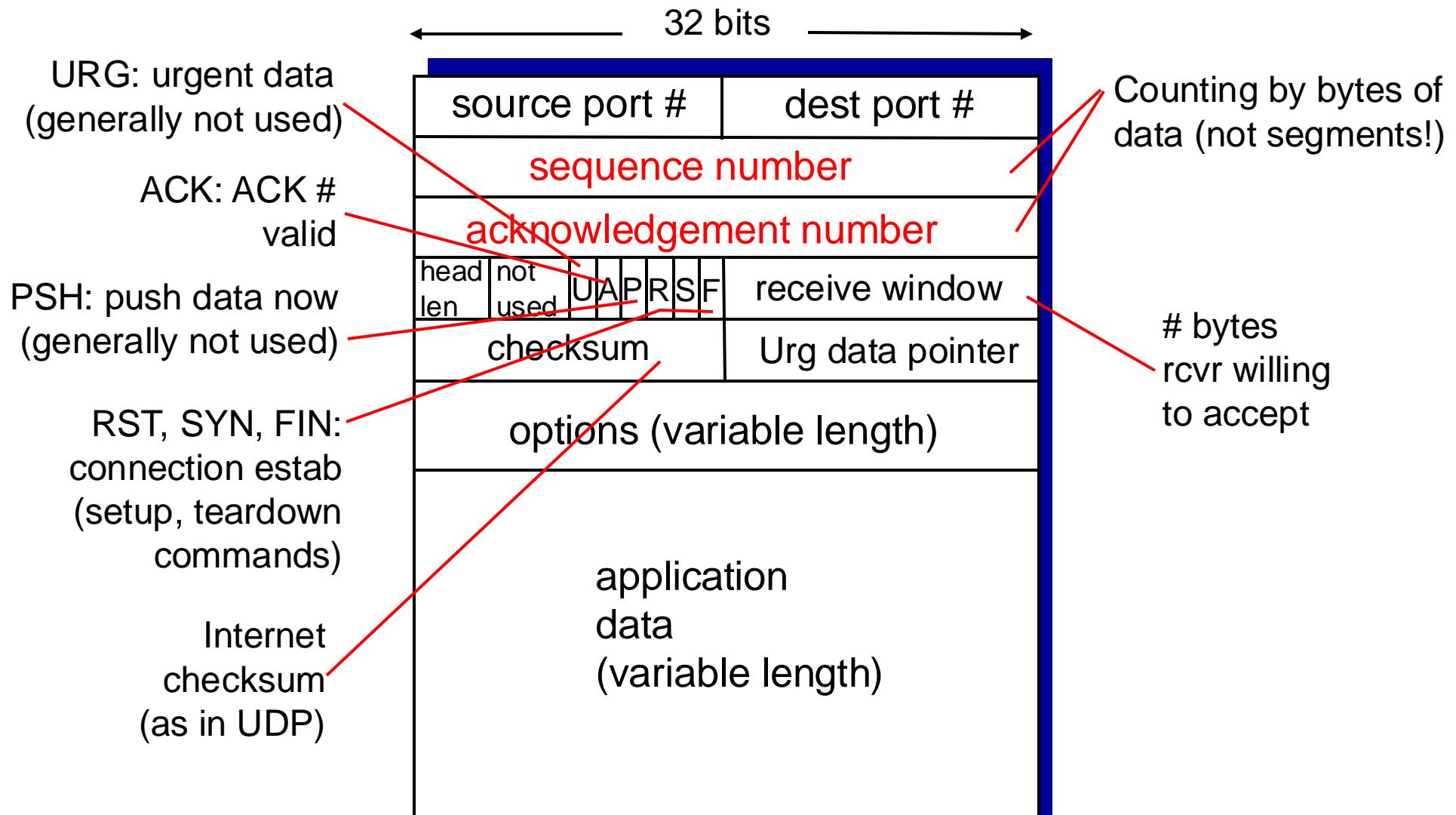


vs UDP structure



UDP segment format

TCP segment structure



TCP seq. numbers, ACKs

Sequence numbers: 序列号

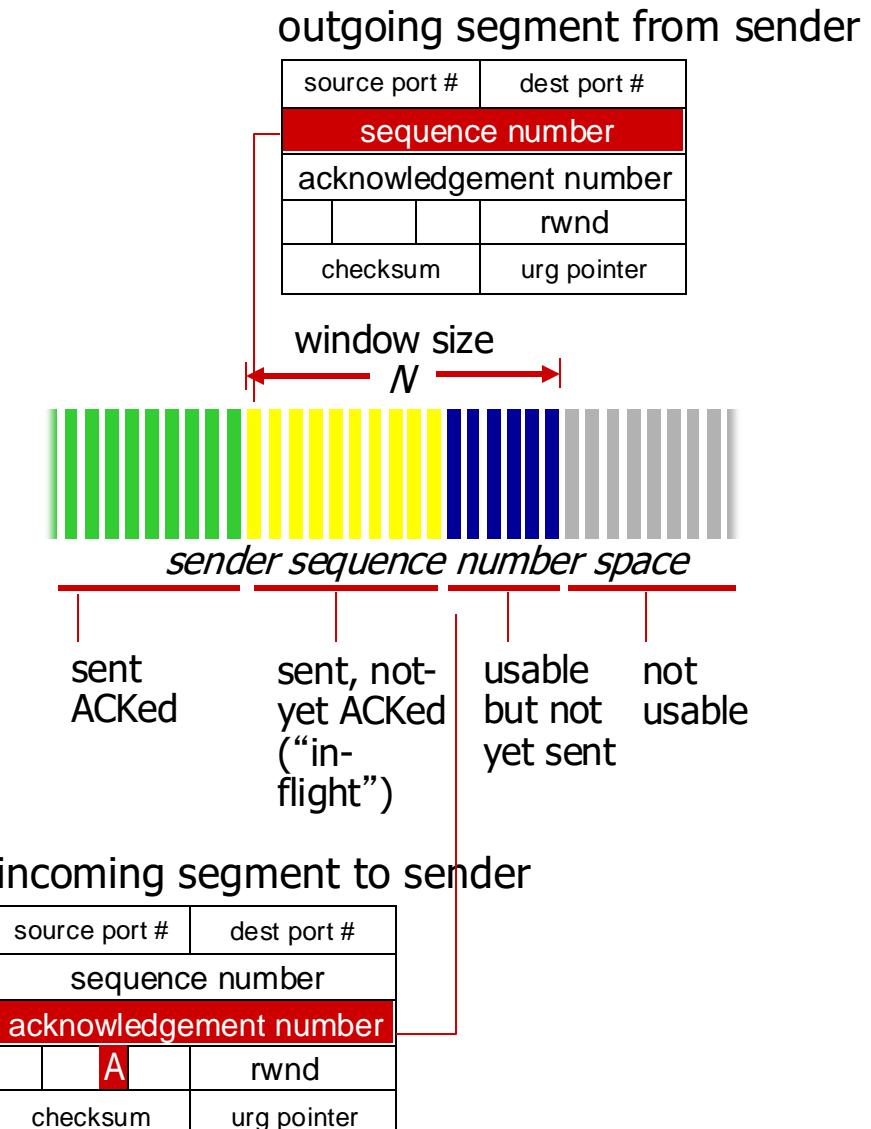
- Byte stream “number” of first byte in segment’s data

Acknowledgements:

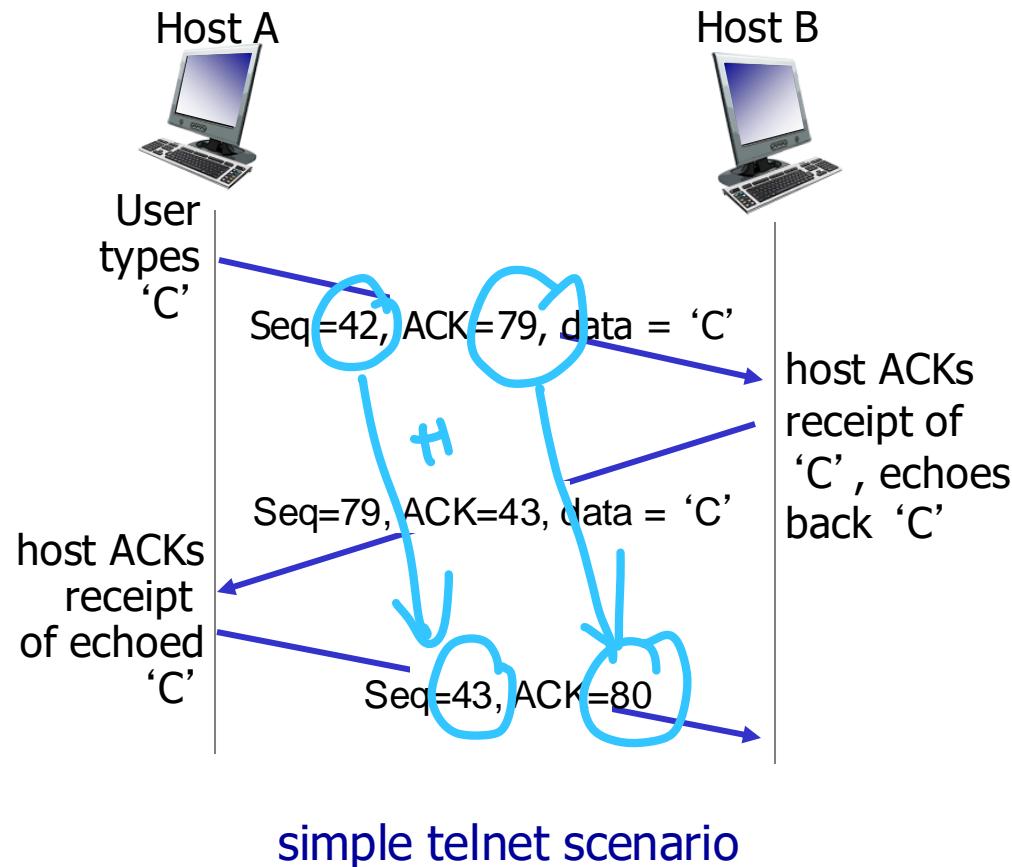
- Seq # of next byte expected from other side
- Cumulative ACK

Q: How receiver handles out-of-order segments

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



TCP seq. numbers, ACKs



Loopback: lo0 (port 12000)

Apply a display filter ... <?>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	68	50965 → 12000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=910558757 TSecr=0 SACK_PERM
2	0.000225	127.0.0.1	127.0.0.1	TCP	68	12000 → 50965 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2614288304 TSecr=910558757 SACK_PERM
3	0.000257	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=910558757 TSecr=2614288304
4	0.000296	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 12000 → 50965 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=2614288304 TSecr=910558757
5	3.545329	127.0.0.1	127.0.0.1	TCP	63	50965 → 12000 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=7 TSval=910562302 TSecr=2614288304
6	3.545443	127.0.0.1	127.0.0.1	TCP	56	12000 → 50965 [ACK] Seq=1 Ack=8 Win=408256 Len=0 TSval=2614291849 TSecr=910562302
7	3.545630	127.0.0.1	127.0.0.1	TCP	63	12000 → 50965 [PSH, ACK] Seq=1 Ack=8 Win=408256 Len=7 TSval=2614291849 TSecr=910562302
8	3.545661	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [ACK] Seq=8 Ack=8 Win=408256 Len=0 TSval=910562302 TSecr=2614291849
9	3.545666	127.0.0.1	127.0.0.1	TCP	56	12000 → 50965 [FIN, ACK] Seq=8 Ack=8 Win=408256 Len=0 TSval=2614291849 TSecr=910562302
10	3.545702	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [ACK] Seq=8 Ack=9 Win=408256 Len=0 TSval=910562302 TSecr=2614291849
11	3.545740	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [FIN, ACK] Seq=8 Ack=9 Win=408256 Len=0 TSval=910562302 TSecr=2614291849
12	3.545809	127.0.0.1	127.0.0.1	TCP	56	12000 → 50965 [ACK] Seq=9 Ack=9 Win=408256 Len=0 TSval=2614291849 TSecr=910562302

```
> Frame 5: 63 bytes on wire (504 bits), 63 bytes captured (504 bits) on interface lo0, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 50965, Dst Port: 12000, Seq: 1, Ack: 1, Len: 7
  Data (7 bytes)
    Data: 61626364656667
    [Length: 7]
```

0000	02 00 00 00 45 00 00 3b	00 00 40 00 40 06 00 00	...E.. ; ..@ @ ..
0010	7f 00 00 01 7f 00 00 01	c7 15 2e e0 dc c9 60 ed
0020	04 01 30 b0 80 18 18 eb	fe 2f 00 00 01 01 08 0a	...0..... /.....
0030	36 46 13 fe 9b d2 df b0	61 62 63 64 65 66 67	6F..... abcdefg

Packets: 12 · Displayed: 12 (100.0%) · Dropped: 0 (0.0%) · Profile: Default

Send from client

Loopback: lo0 (port 12000)

Apply a display filter ... <?>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	68	50965 → 12000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=910558757 TSecr=0 SACK_PERM
2	0.000225	127.0.0.1	127.0.0.1	TCP	68	12000 → 50965 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2614288304 TSecr=910558757 SACK_PERM
3	0.000257	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [ACK] Seq=1 Ack=1 Win=0 TSval=910558757 TSecr=2614288304
4	0.000296	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 12000 → 50965 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=2614288304 TSecr=910558757
5	3.545329	127.0.0.1	127.0.0.1	TCP	63	50965 → 12000 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=7 TSval=910562302 TSecr=2614288304
6	3.545443	127.0.0.1	127.0.0.1	TCP	56	12000 → 50965 [ACK] Seq=1 Ack=8 Win=408256 Len=0 TSval=2614291849 TSecr=910562302
7	3.545630	127.0.0.1	127.0.0.1	TCP	63	12000 → 50965 [PSH, ACK] Seq=1 Ack=8 Win=408256 Len=7 TSval=2614291849 TSecr=910562302
8	3.545661	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [ACK] Seq=8 Ack=8 Win=408256 Len=0 TSval=910562302 TSecr=2614291849
9	3.545666	127.0.0.1	127.0.0.1	TCP	56	12000 → 50965 [FIN, ACK] Seq=8 Ack=8 Win=408256 Len=0 TSval=2614291849 TSecr=910562302
10	3.545702	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [ACK] Seq=8 Ack=9 Win=408256 Len=0 TSval=910562302 TSecr=2614291849
11	3.545740	127.0.0.1	127.0.0.1	TCP	56	50965 → 12000 [FIN, ACK] Seq=8 Ack=9 Win=408256 Len=0 TSval=910562302 TSecr=2614291849
12	3.545809	127.0.0.1	127.0.0.1	TCP	56	12000 → 50965 [ACK] Seq=9 Ack=9 Win=408256 Len=0 TSval=2614291849 TSecr=910562302

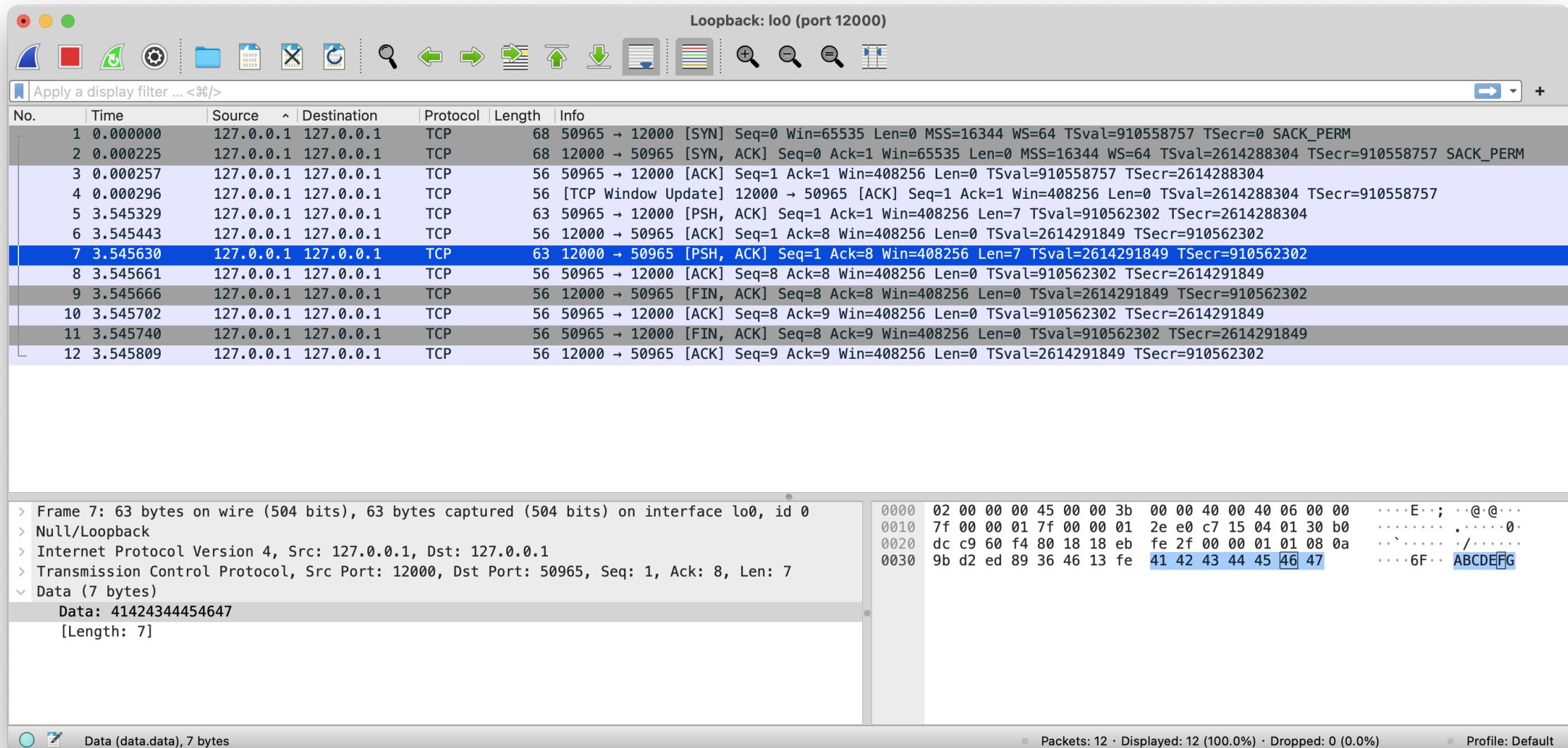
Acknowledgment Number: 8 (relative ack number)
 Acknowledgment number (raw): 3704185076
 1000 = Header Length: 32 bytes (8)
 Flags: 0x010 (ACK)
 000. = Reserved: Not set
 ...0 = Accurate ECN: Not set
 0.... = Congestion Window Reduced: Not set
0.... = ECN-Echo: Not set
0.... = Urgent: Not set
1.... = Acknowledgment: Set
 0.... = Push: Not set
0.... = Reset: Not set
0.... = Syn: Not set
0.... = Fin: Not set
 [TCP Flags:A.....]
 Window: 6379

Flags (12 bits) (tcp.flags), 2 bytes

Packets: 12 · Displayed: 12 (100.0%) · Dropped: 0 (0.0%)

Profile: Default

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

RTT

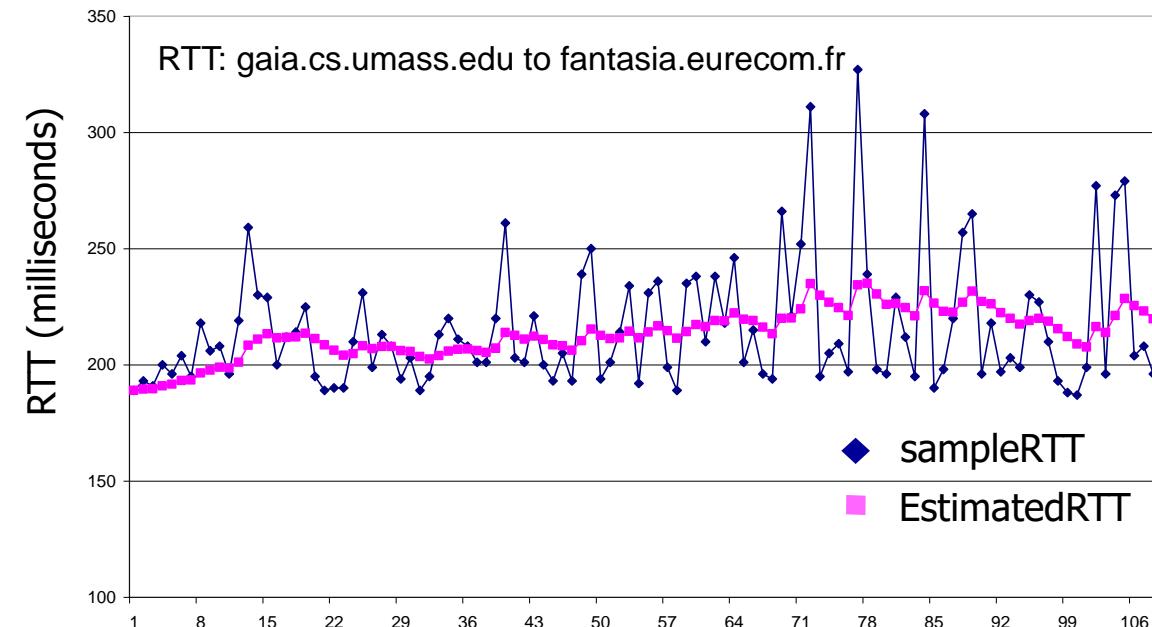
Q: How to estimate RTT?

- 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: 偏差
估计

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

(typically, $\beta = 0.25$)



$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



(Dev = Deviations)

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:

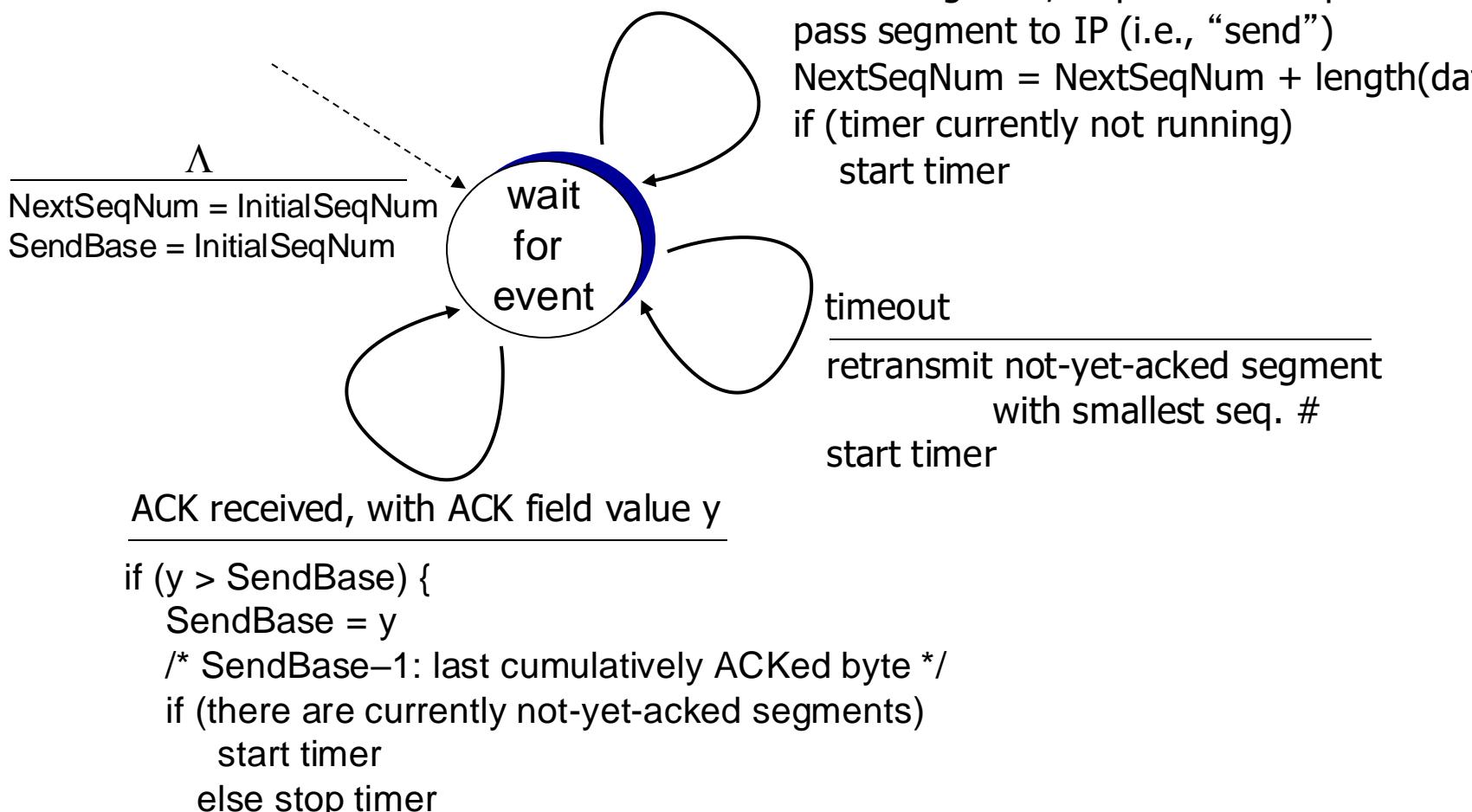
Data rcvd from app:

- 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:

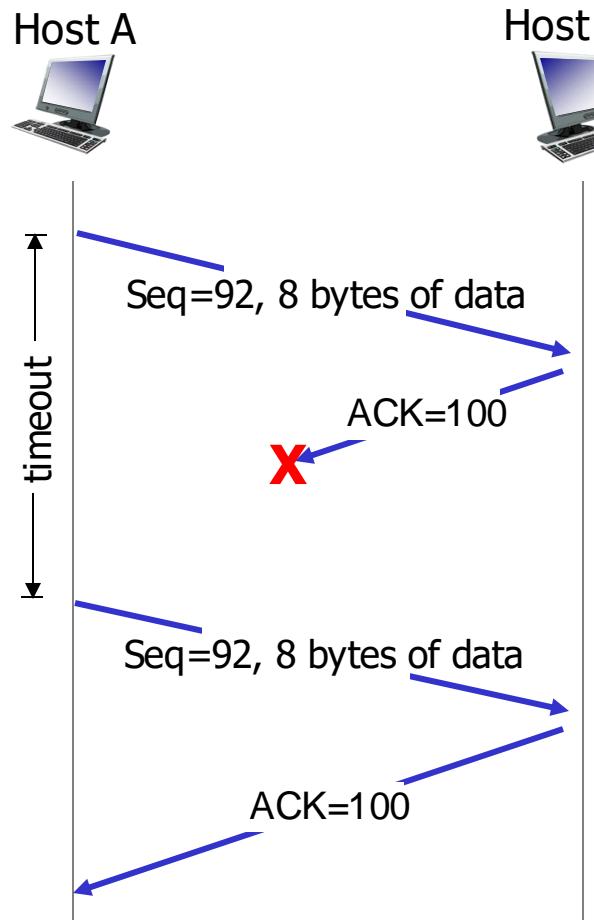
- Retransmit segment that caused timeout
 - Restart timer
- ## *Ack rcvd:*
- If ack acknowledges previously unacked segments
 - Update what is known to be ACKed
 - Start timer if there are still unacked segments

TCP sender (simplified)

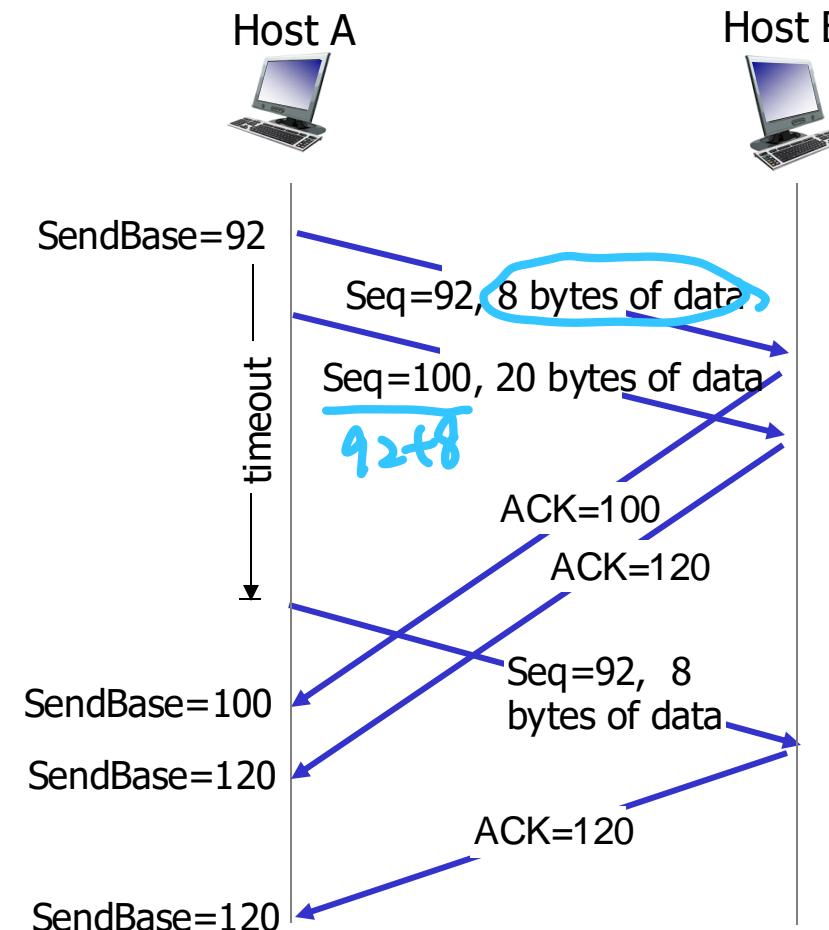




TCP: retransmission scenarios

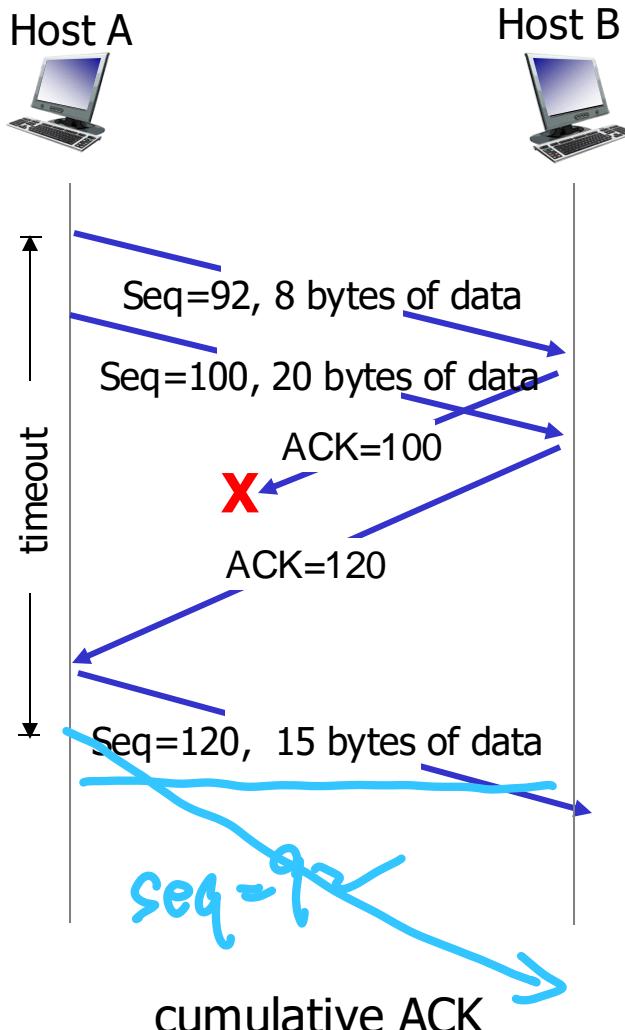
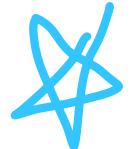


lost ACK scenario



premature timeout

TCP: retransmission scenarios



没到 timeout 发下一个

TCP ACK generation [RFC 1122, RFC 2581]

<i>Event at receiver</i>	<i>TCP receiver action</i>
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
<u>arrival of out-of-order segment</u> higher-than-expect seq. # . Gap detected	<u>immediately send <i>duplicate ACK</i>,</u> 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

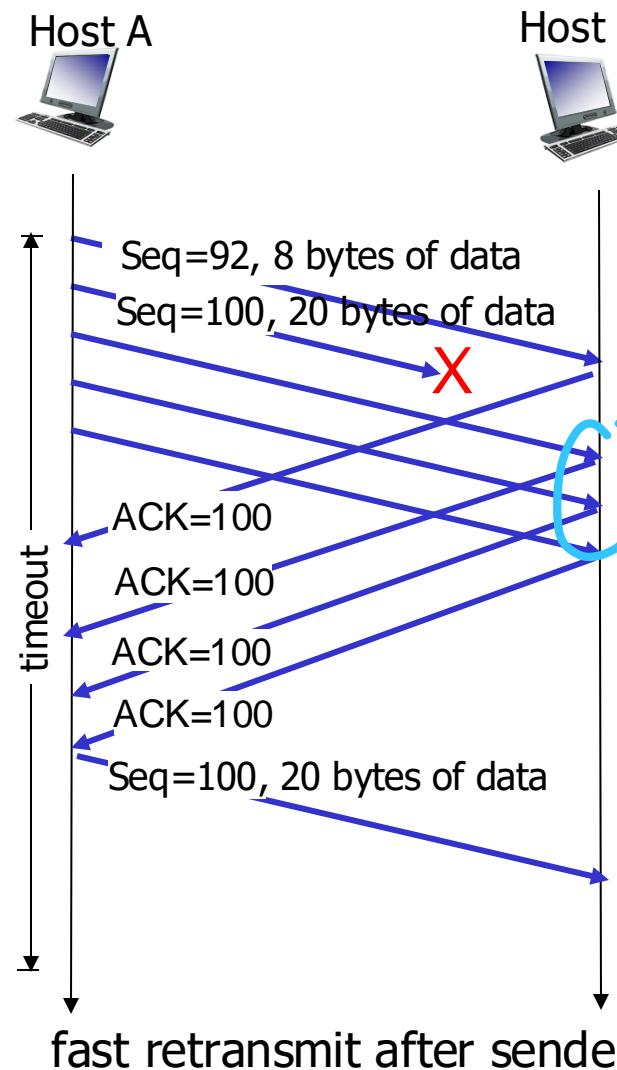
- Time-out period often relatively long:
 - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
 - Sender often sends many segments back-to-back
 - If segment is lost, there will likely be many duplicate ACKs.

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



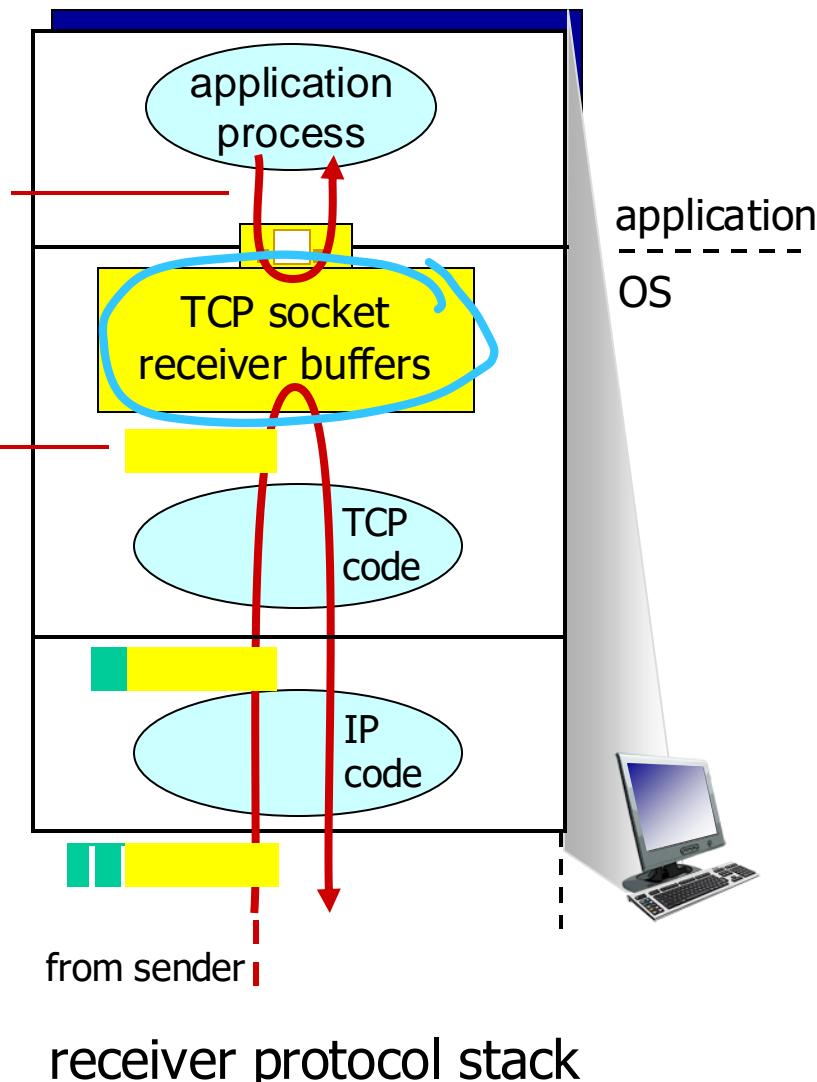
希望序列是 100 結果不是。
返回 ACK 100.

TCP flow control

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

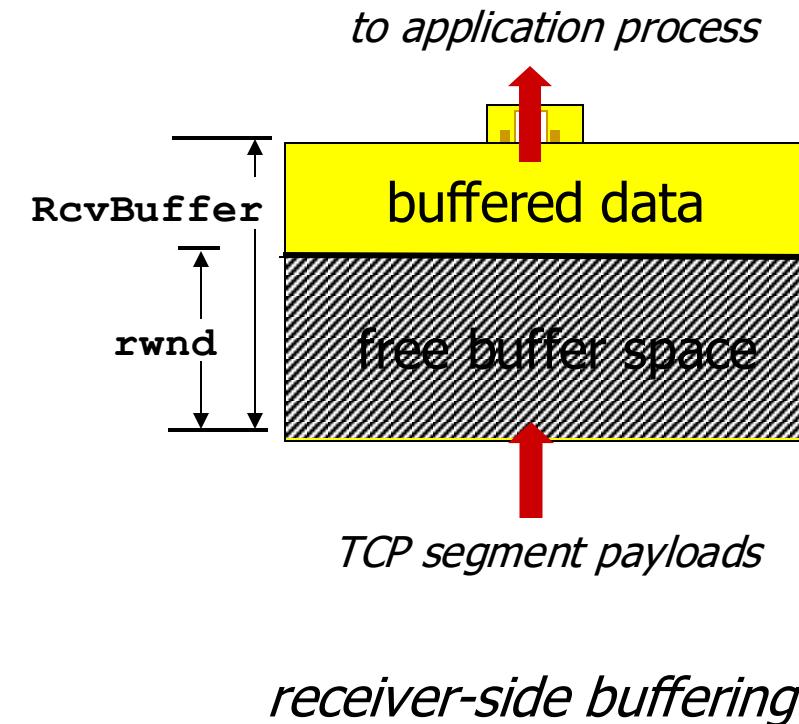
application may
remove data from
TCP socket buffers

... slower than TCP
receiver is delivering
(sender is sending)

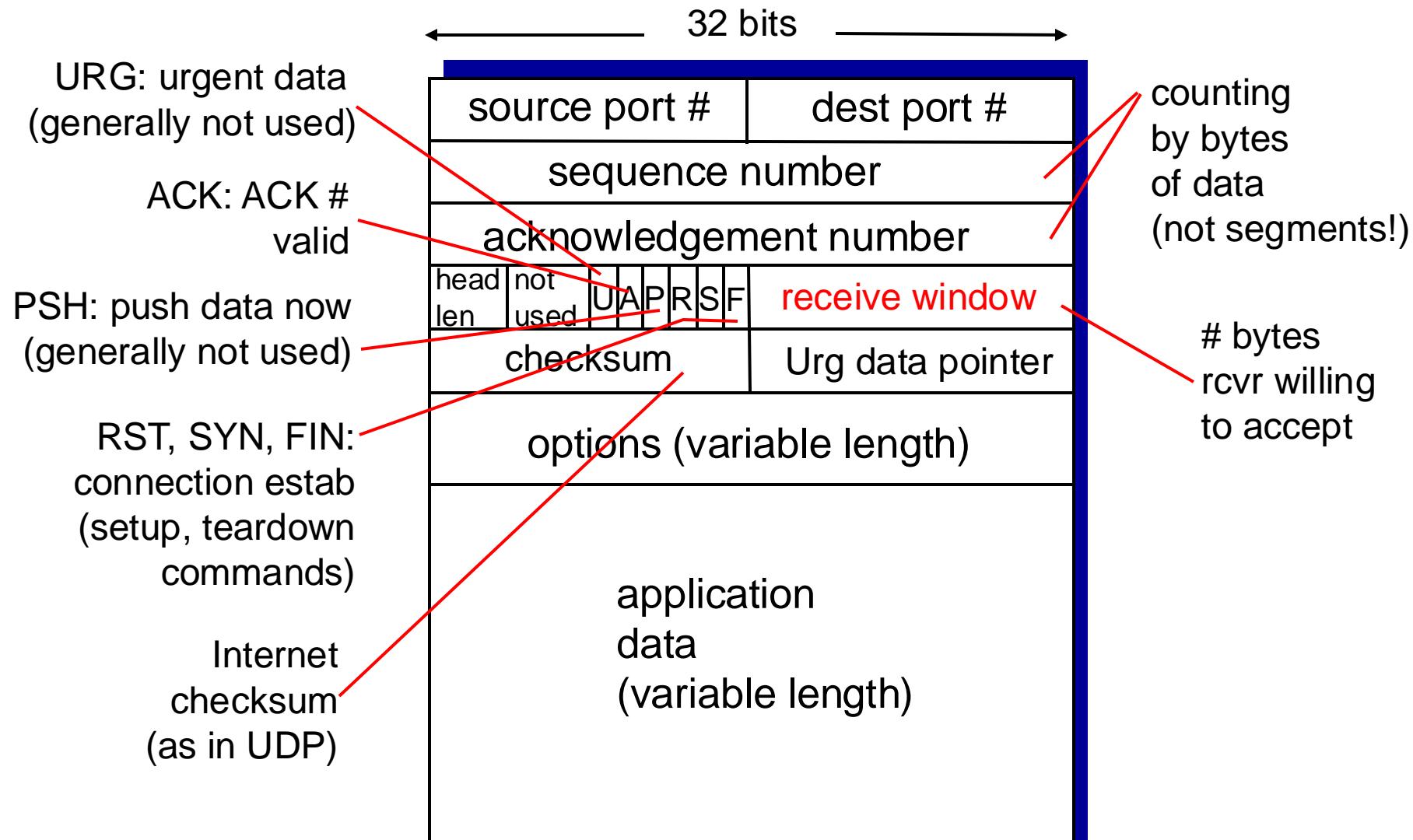


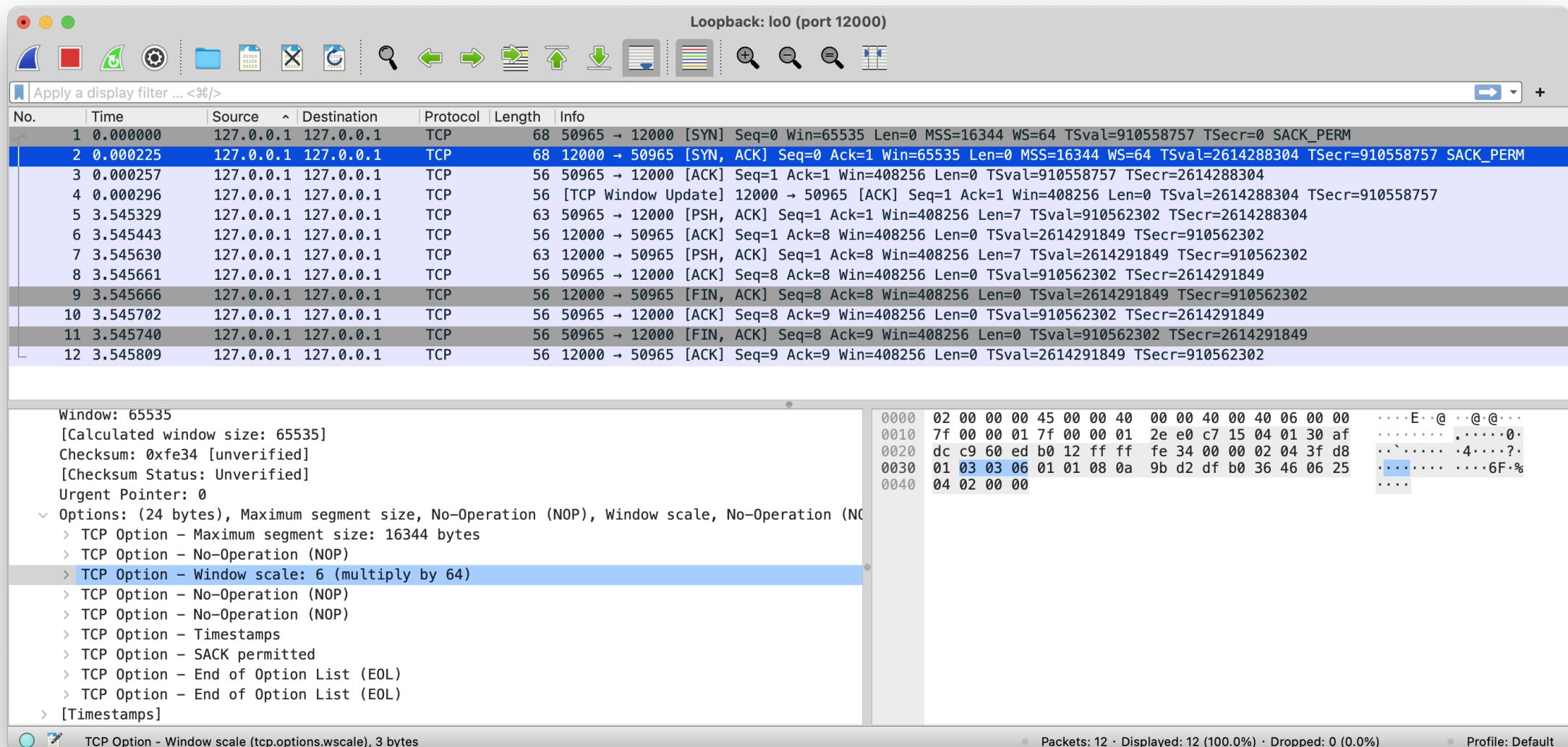
TCP flow control

- 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

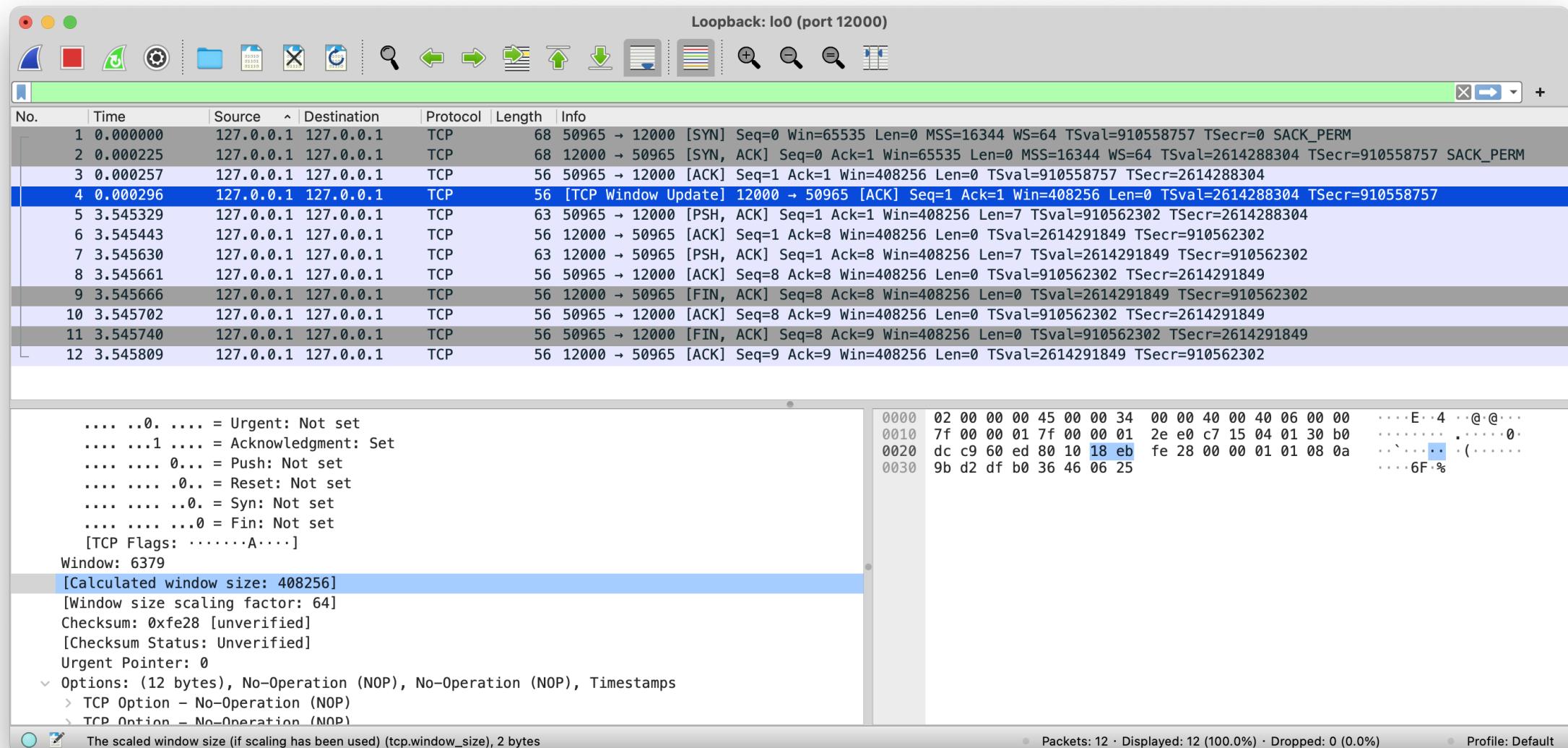


TCP segment structure





SYN: Window Scale factor

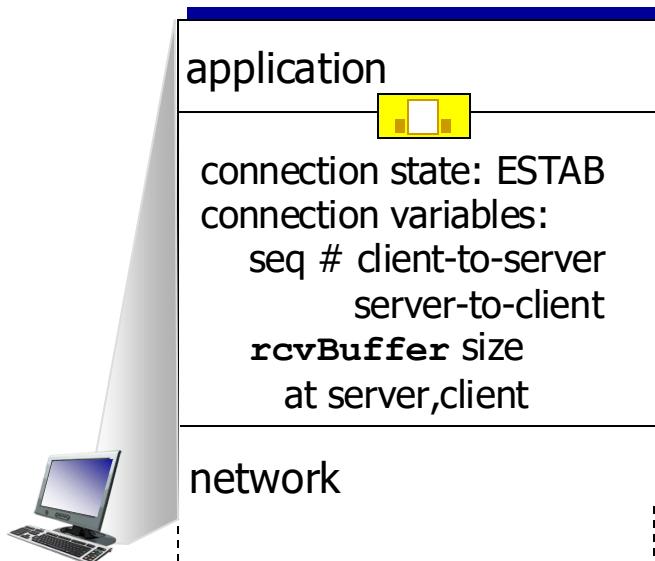


$$\text{Calculated Window Size} = 6379 \times 64 = 408256$$

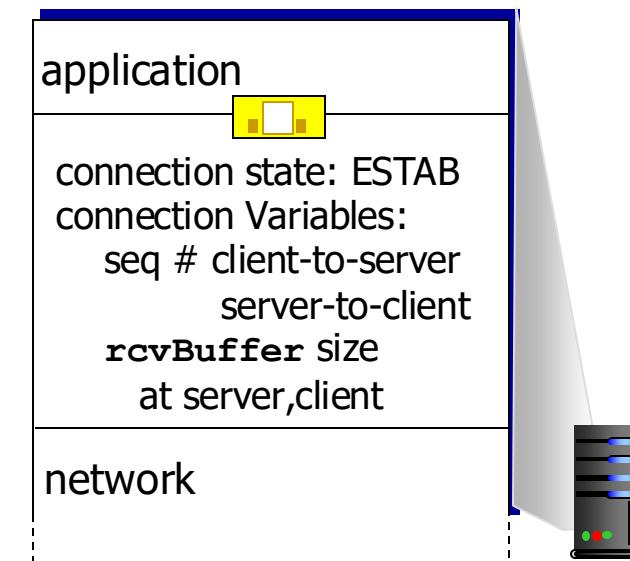
Connection Management

 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");
```

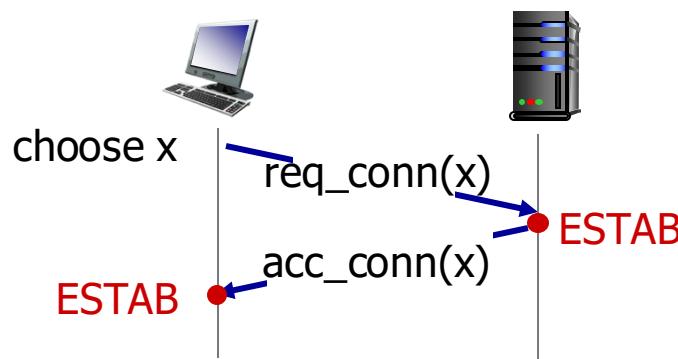
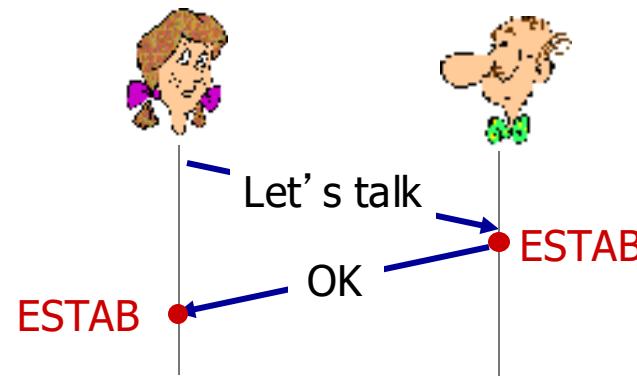


```
Socket connectionSocket =  
    welcomeSocket.accept();
```

Agreeing to establish a connection

二次握手可能失败 ·

2-way handshake:

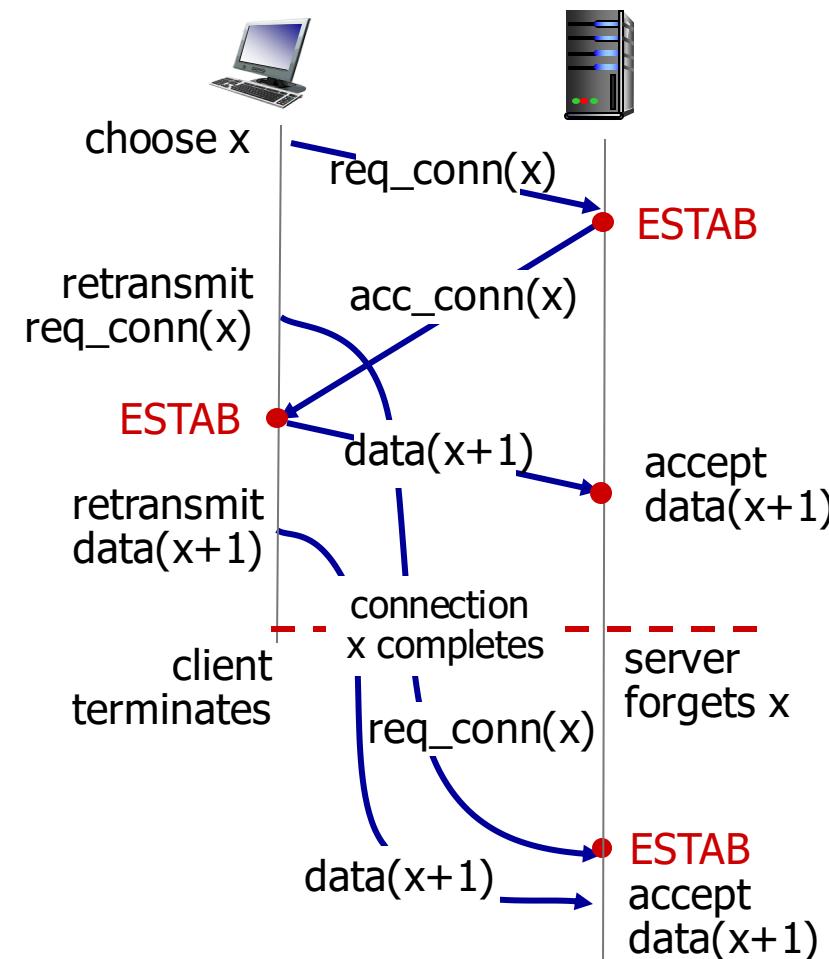
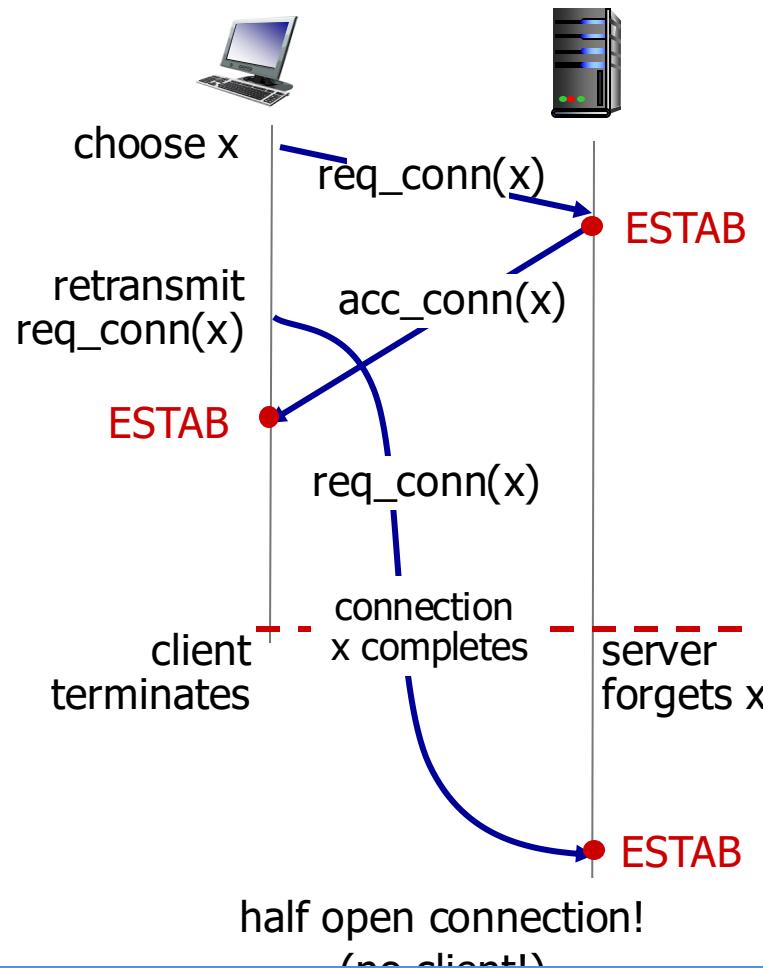


Q: Will 2-way handshake always work in network?

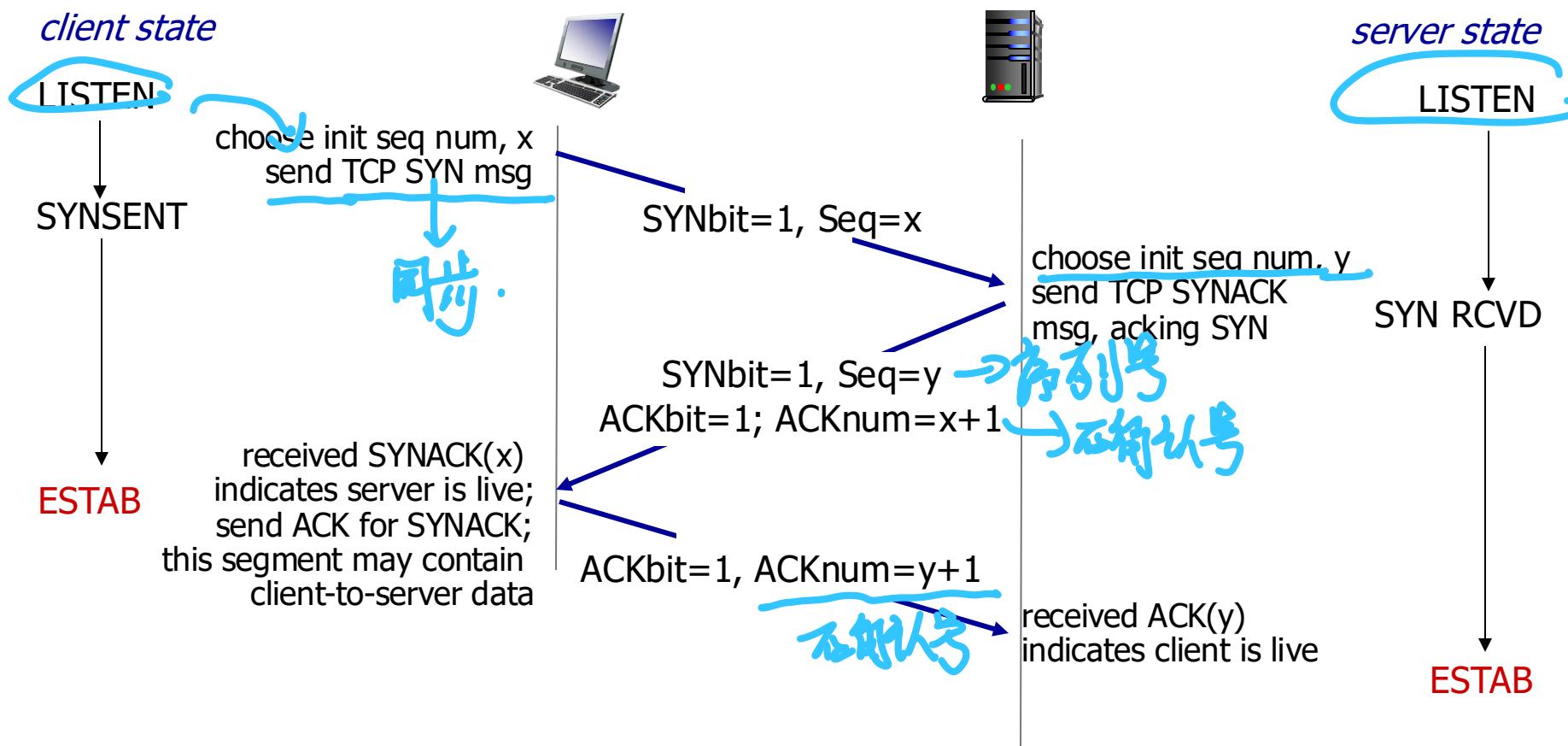
- Variable delays
- Retransmitted messages (e.g. `req_conn(x)`) due to message loss
- Message reordering
- Can't "see" other side

Agreeing to establish a connection

2-way handshake failure scenarios:



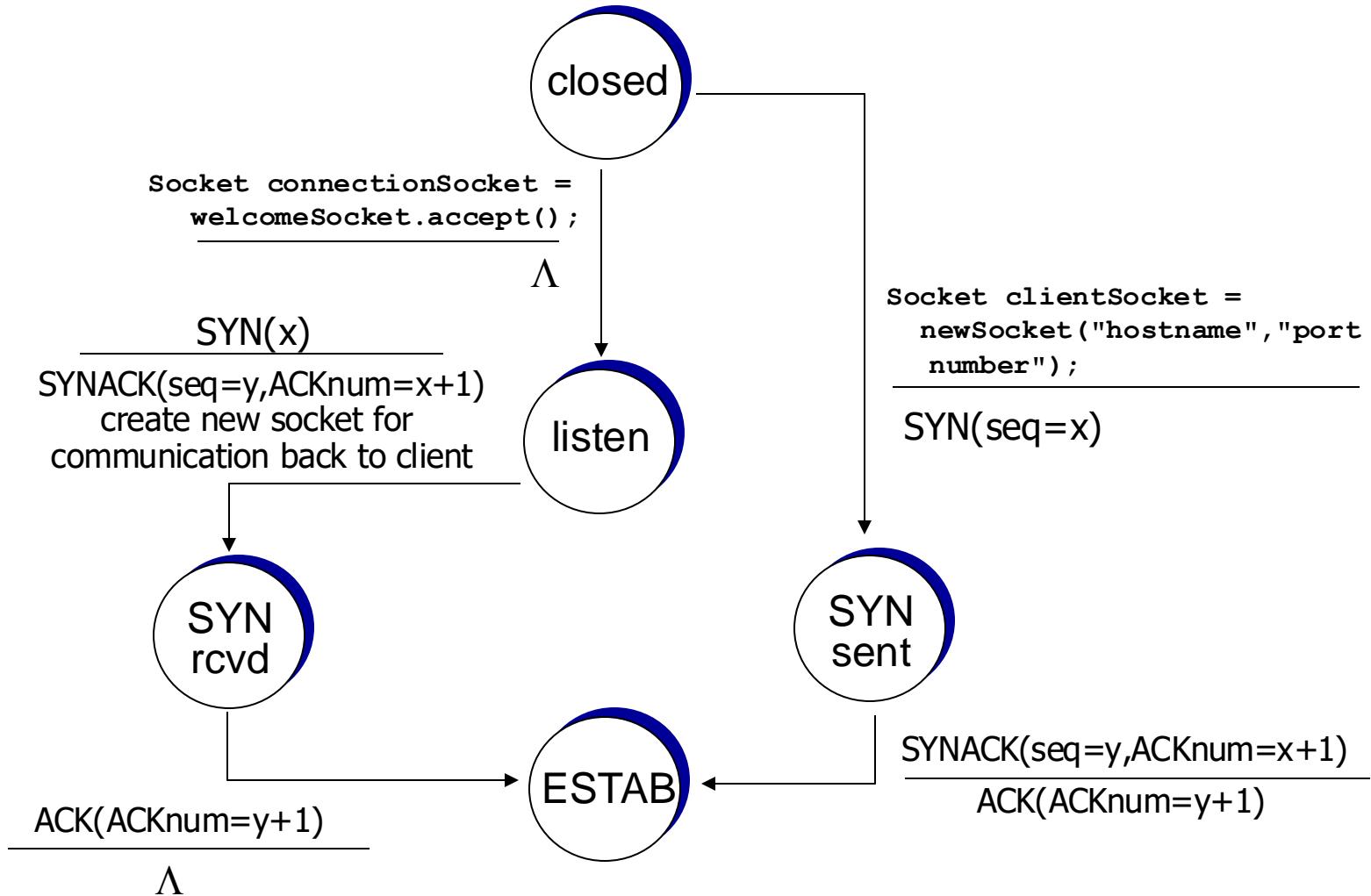
TCP 3-way handshake



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
```

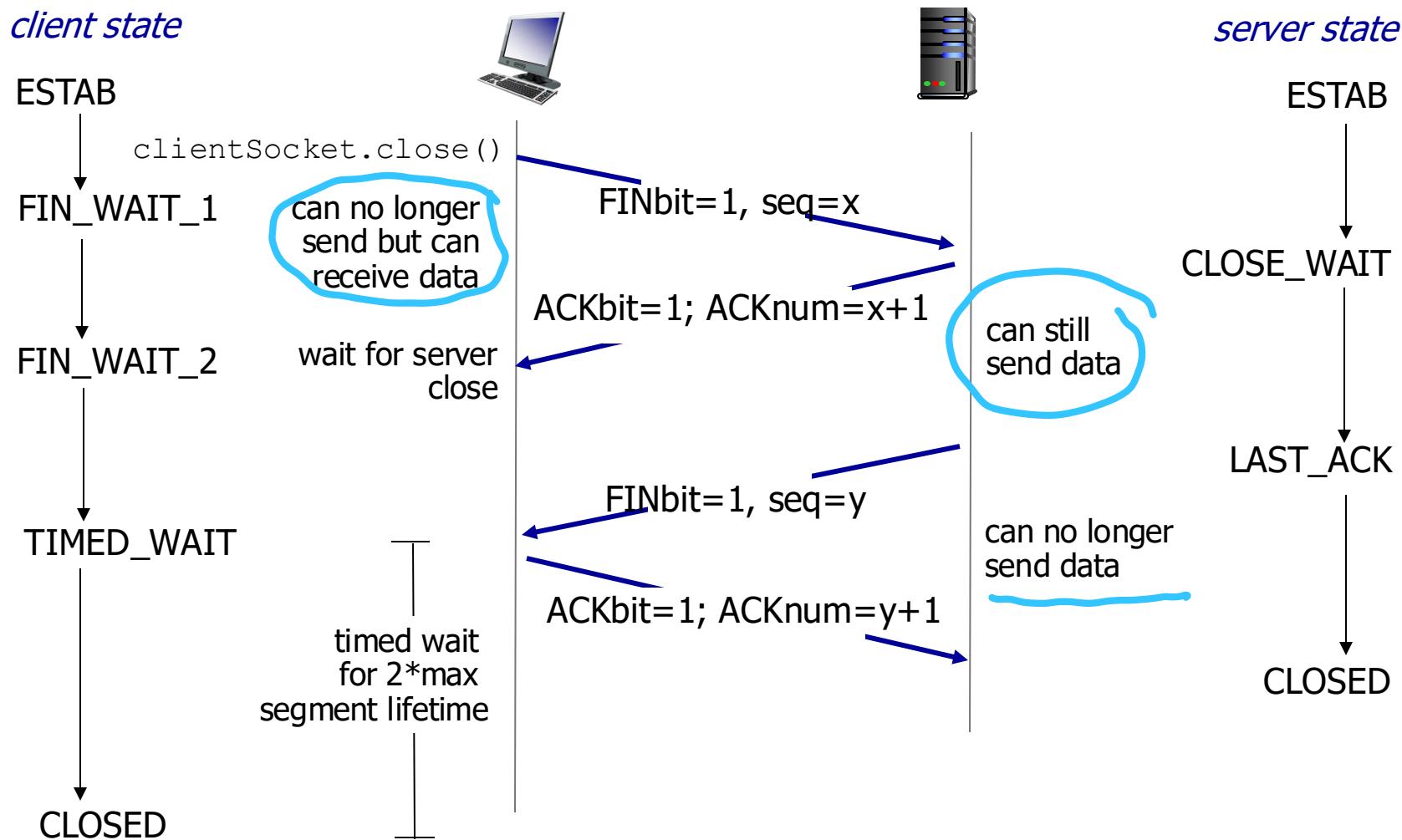
TCP 3-way handshake: FSM



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



Capturing from Loopback 0 (port 0-65535)

Apply a display filter ... <%>

No. Time Source Destination Proto Len Info

1	0.000000	127.0.0.1	127.0.0.1	TCP	68	57315 → 16763 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=870206782 TSecr=0 SACK_PERM...
2	0.000113	127.0.0.1	127.0.0.1	TCP	68	16763 → 57315 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=870206782 TSecr=...
3	0.000120	127.0.0.1	127.0.0.1	TCP	56	57315 → 16763 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=870206782 TSecr=870206782
4	0.000126	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 16763 → 57315 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=870206782 TSecr=...
5	1.504812	127.0.0.1	127.0.0.1	TCP	59	57315 → 16763 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=3 TSval=870208281 TSecr=870206782
6	1.504845	127.0.0.1	127.0.0.1	TCP	56	16763 → 57315 [ACK] Seq=1 Ack=4 Win=408256 Len=0 TSval=870208281 TSecr=870208281
7	1.504903	127.0.0.1	127.0.0.1	TCP	59	16763 → 57315 [PSH, ACK] Seq=1 Ack=4 Win=408256 Len=3 TSval=870208281 TSecr=870208281
8	1.504939	127.0.0.1	127.0.0.1	TCP	56	57315 → 16763 [ACK] Seq=4 Ack=4 Win=408256 Len=0 TSval=870208281 TSecr=870208281
9	1.504989	127.0.0.1	127.0.0.1	TCP	56	57315 → 16763 [FIN, ACK] Seq=4 Ack=4 Win=408256 Len=0 TSval=870208281 TSecr=870208281
10	1.505036	127.0.0.1	127.0.0.1	TCP	56	16763 → 57315 [ACK] Seq=4 Ack=5 Win=408256 Len=0 TSval=870208281 TSecr=870208281
11	2.507319	127.0.0.1	127.0.0.1	TCP	56	16763 → 57315 [FIN, ACK] Seq=4 Ack=5 Win=408256 Len=0 TSval=870209282 TSecr=870208281
12	2.507407	127.0.0.1	127.0.0.1	TCP	56	57315 → 16763 [ACK] Seq=5 Ack=5 Win=408256 Len=0 TSval=870209282 TSecr=870209282

Frame 5: 59 bytes on wire (472 bits), 59 bytes captured (472 bits) on interface 0
Null/Loopback

0000 02 00 00 00 45 00 00 37 00 00 40 00 40 06 00 00 . . . E . 7 . . @ . . .
0010 7f 00 00 01 7f 00 00 01 df e3 41 7b 05 ad 1b 38 A { . . . 8
0020 7c d9 4c 9c 80 18 18 eb fe 2b 00 00 01 01 08 0a | . L . . . +
0030 33 de 53 19 33 de 4d 3e 61 62 63 3 . S . 3 . M > abc

Loopback: lo0: <live capture in progress>

Packets: 12 · Displayed: 12 (100.0%)

Profile: Default

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



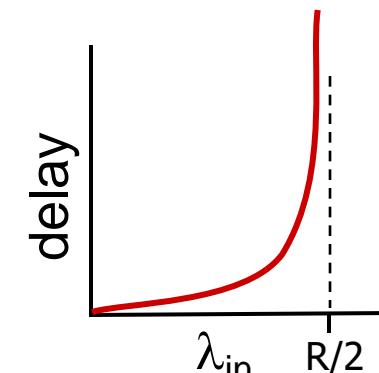
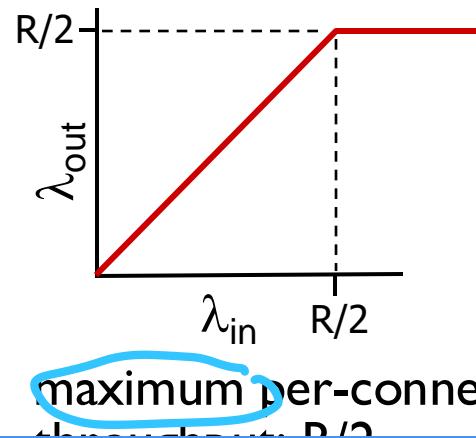
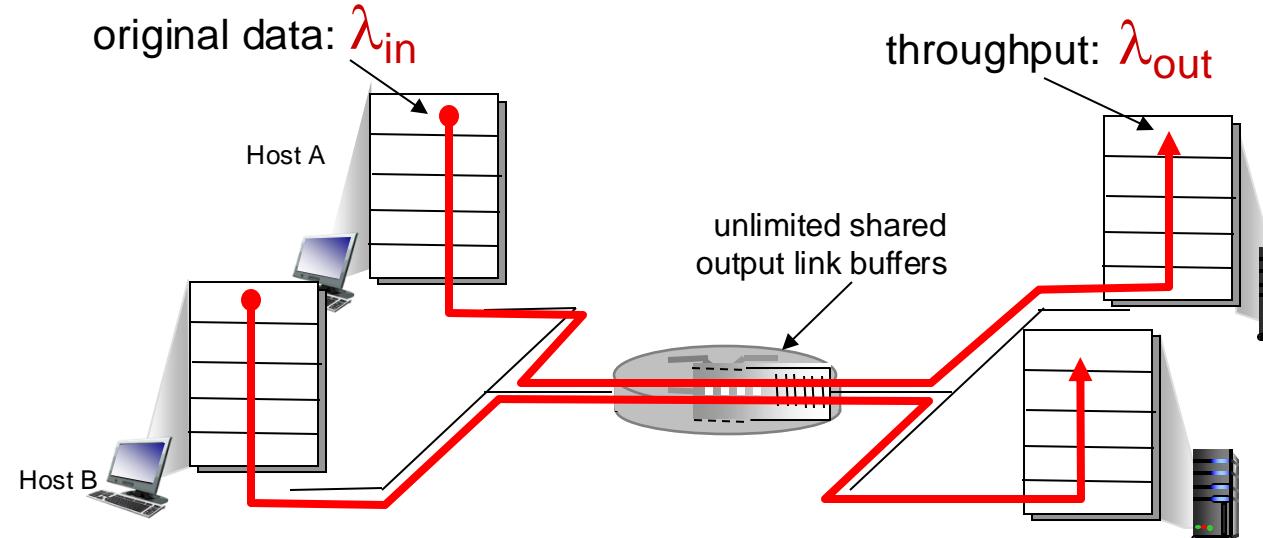
Principles of congestion control

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

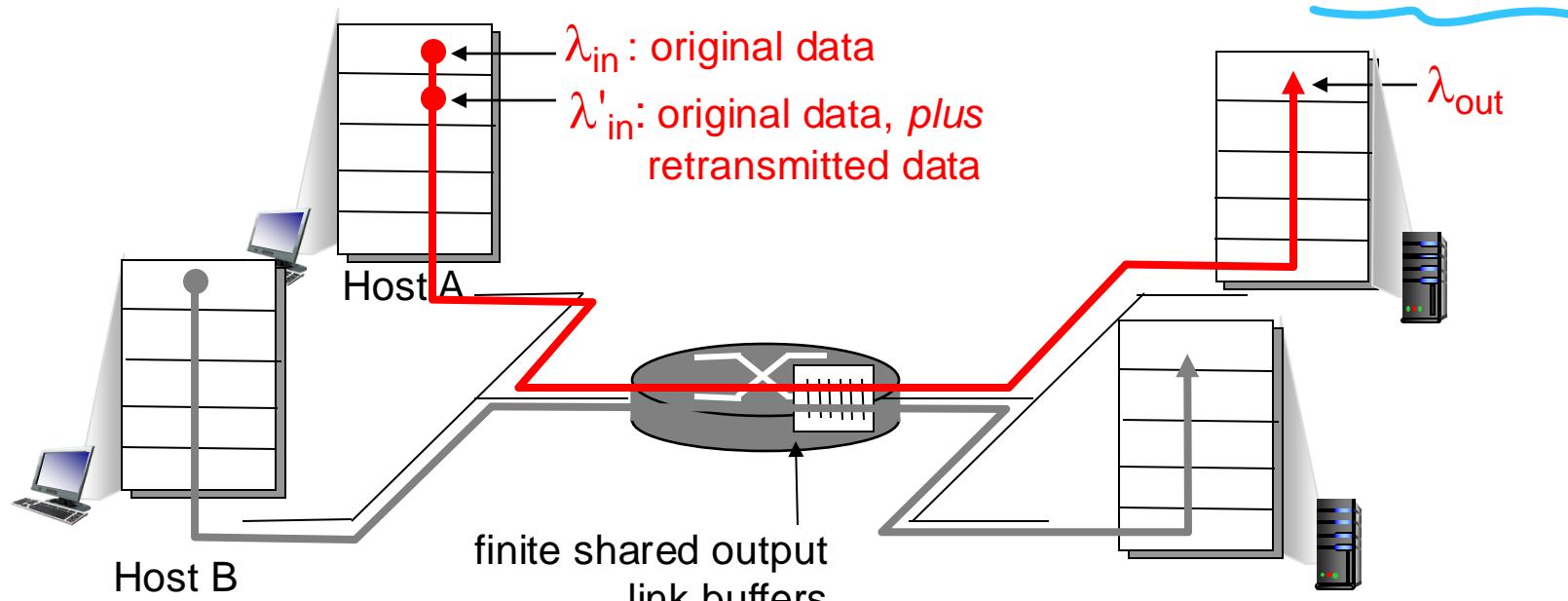


▪ maximum per-connection throughput is $R/2$

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

Causes/costs of congestion: scenario 2

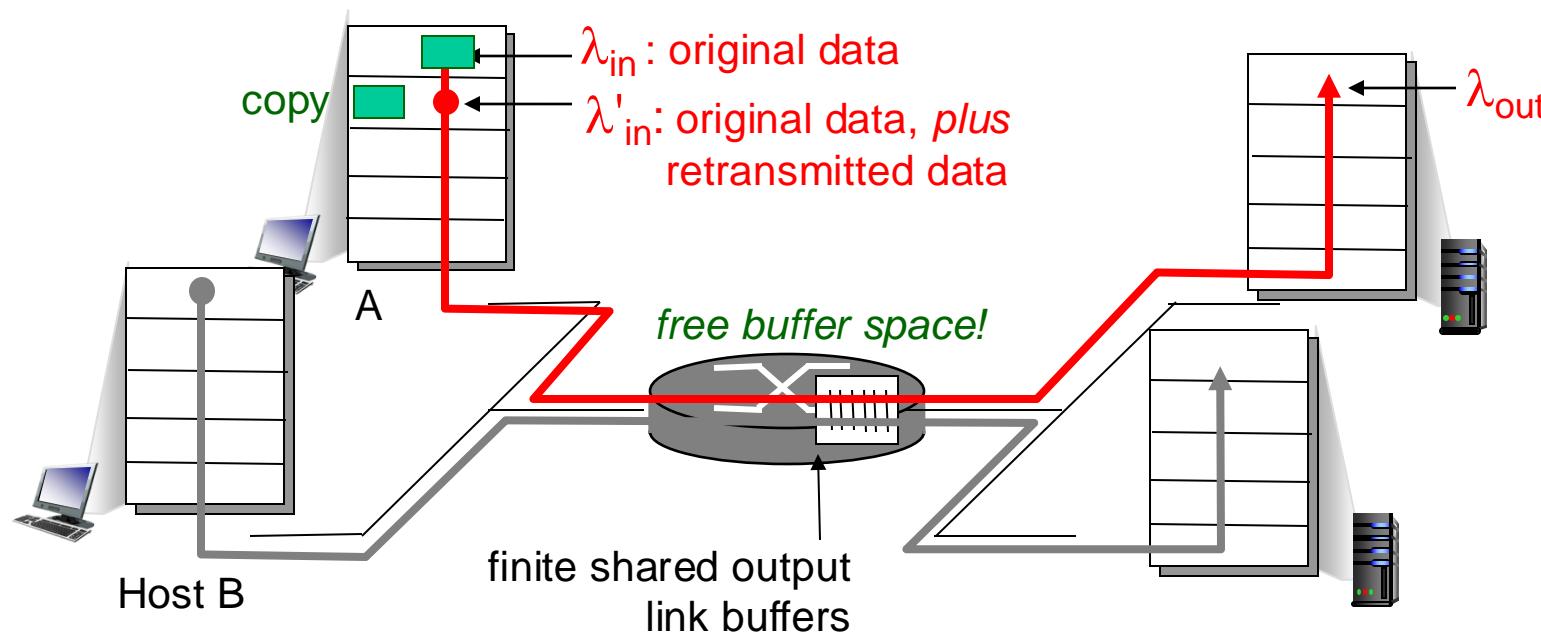
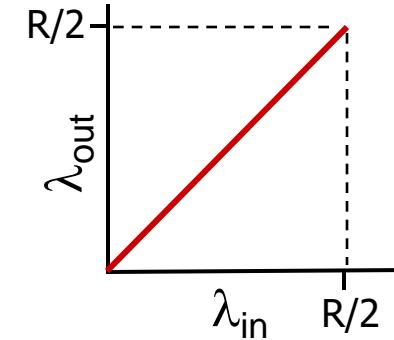
- One router, *finite* buffers
- Sender retransmission of timed-out packet
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- Sender sends only when router buffers available

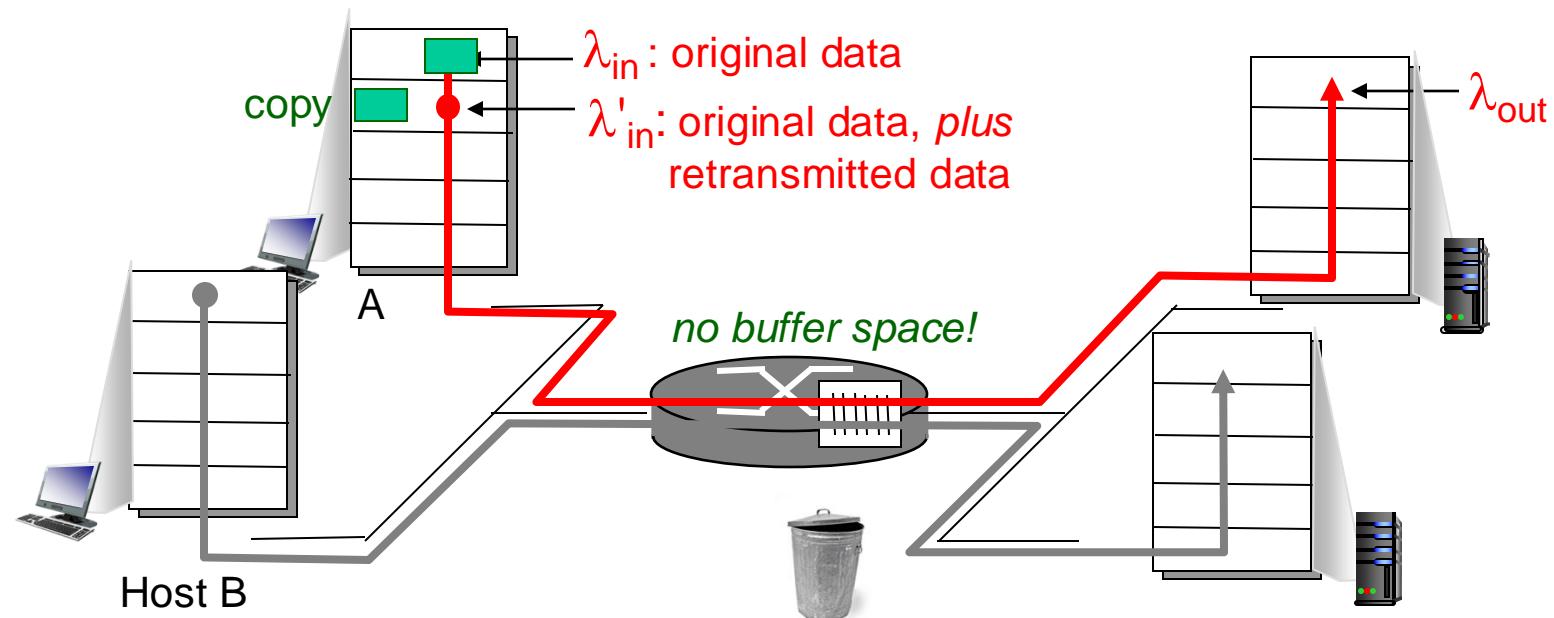


Causes/costs of congestion: scenario 2

Idealization: known loss

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

- Sender only resends if packet known to be lost

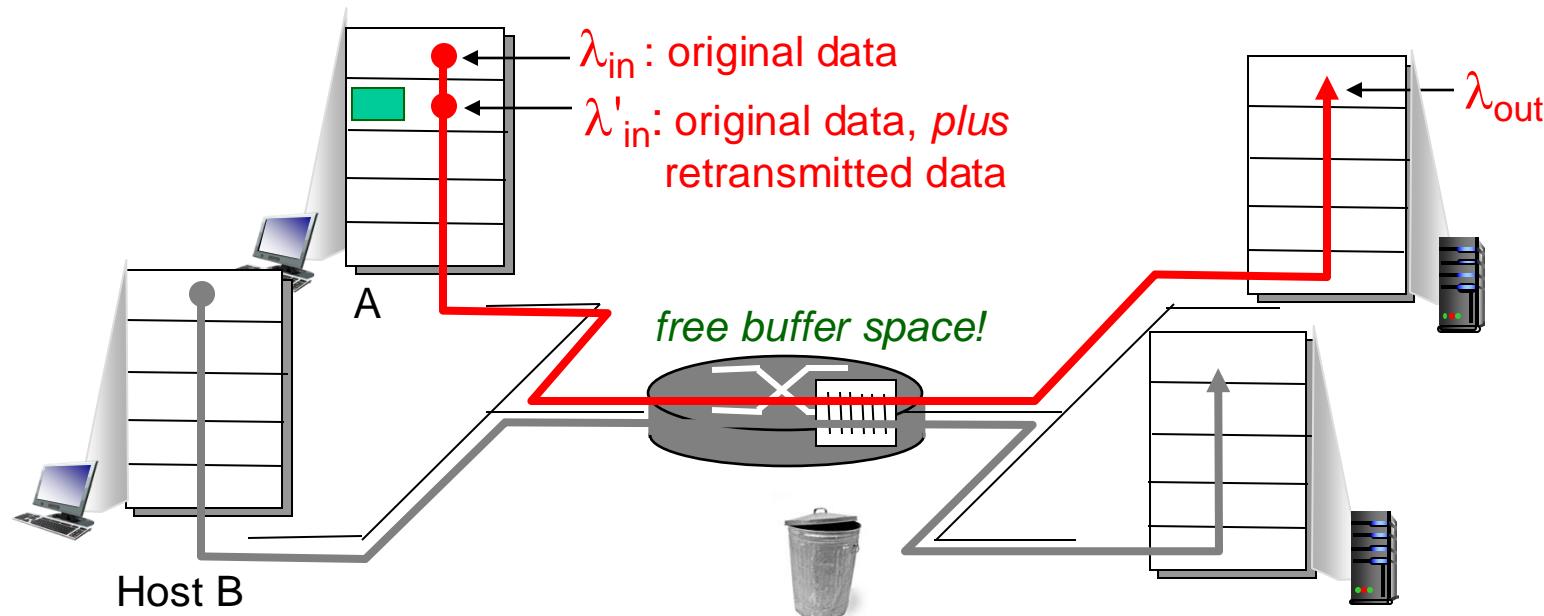
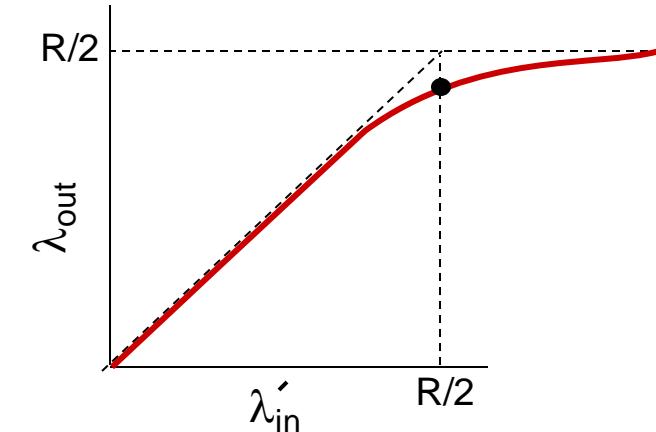


Causes/costs of congestion: scenario 2

Idealization: **known loss**

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

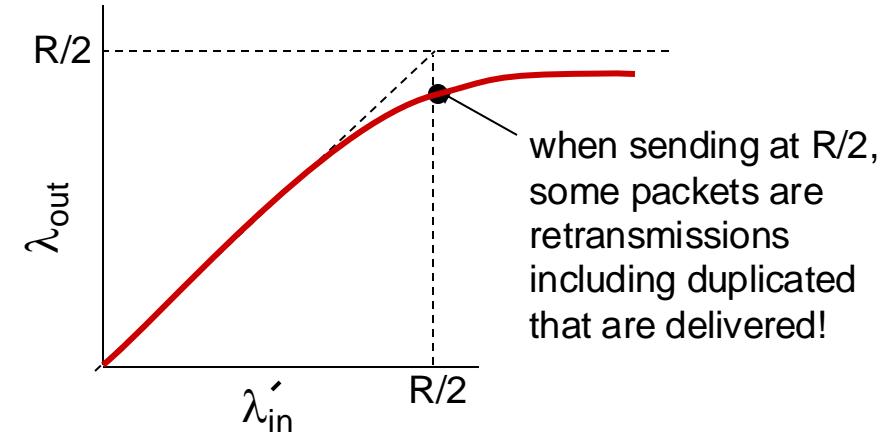
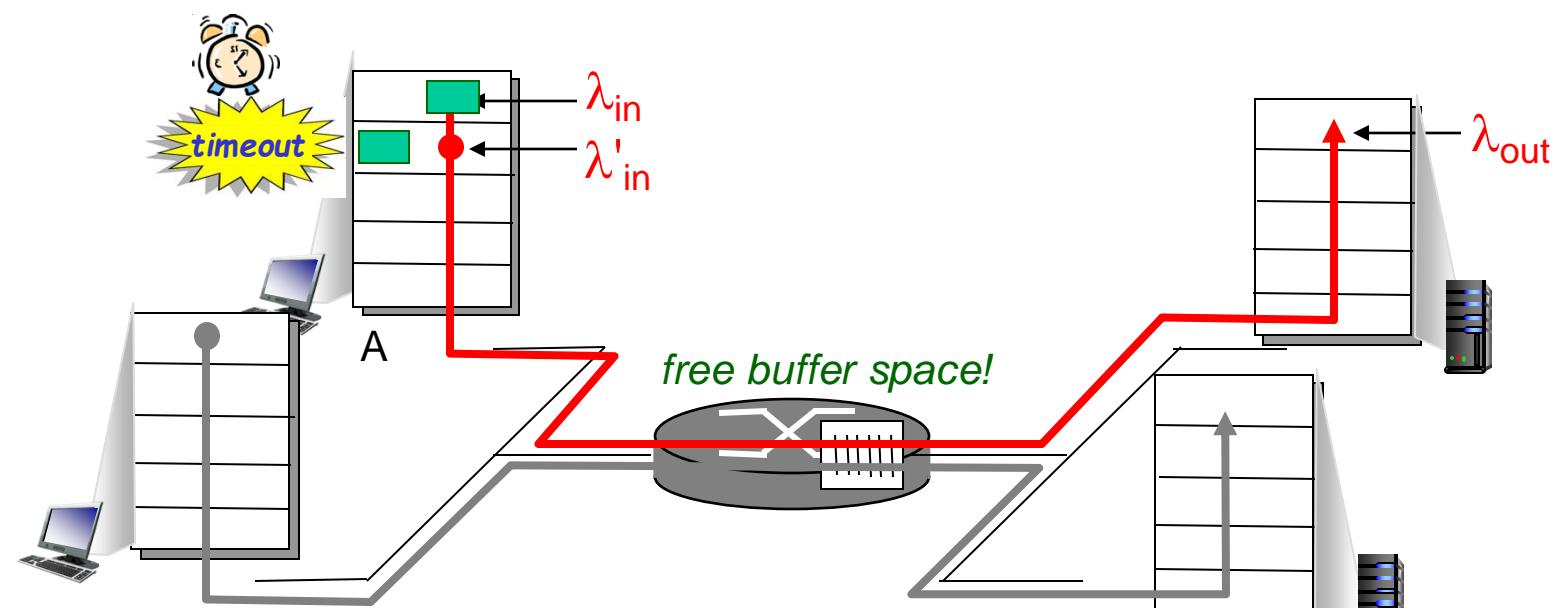
- Sender only resends if packet known to be lost



Causes/costs of congestion: scenario 2

Realistic: duplicates

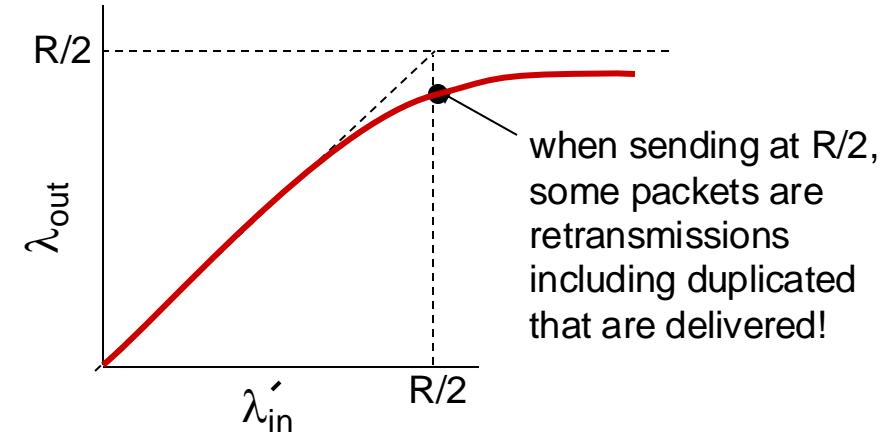
- Packets can be lost, dropped at router due to full buffers
- Sender times out prematurely, sending **two copies**, both of which are delivered



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:

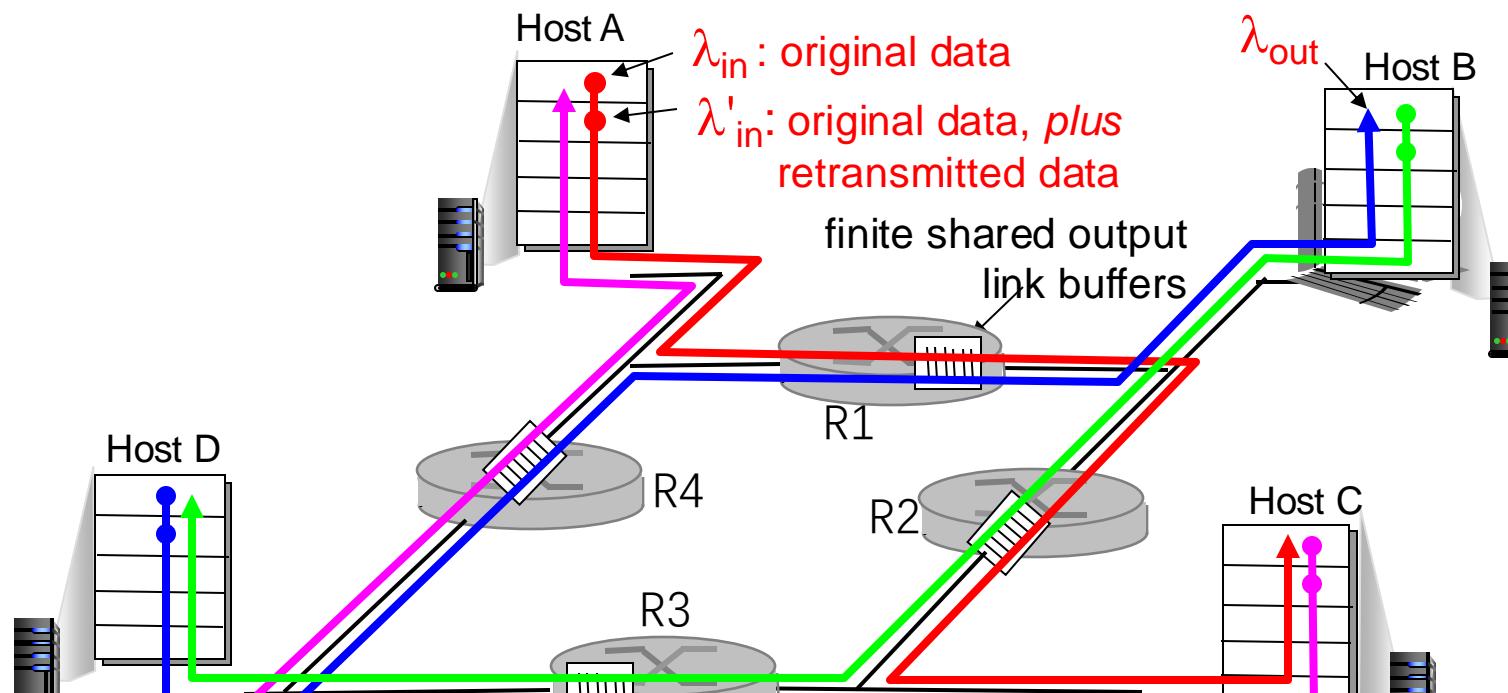
- More work (retrans) for given “goodput”
- Unneeded retransmissions: link carries multiple copies of pkt
 - decreasing goodput

Causes/costs of congestion: scenario 3

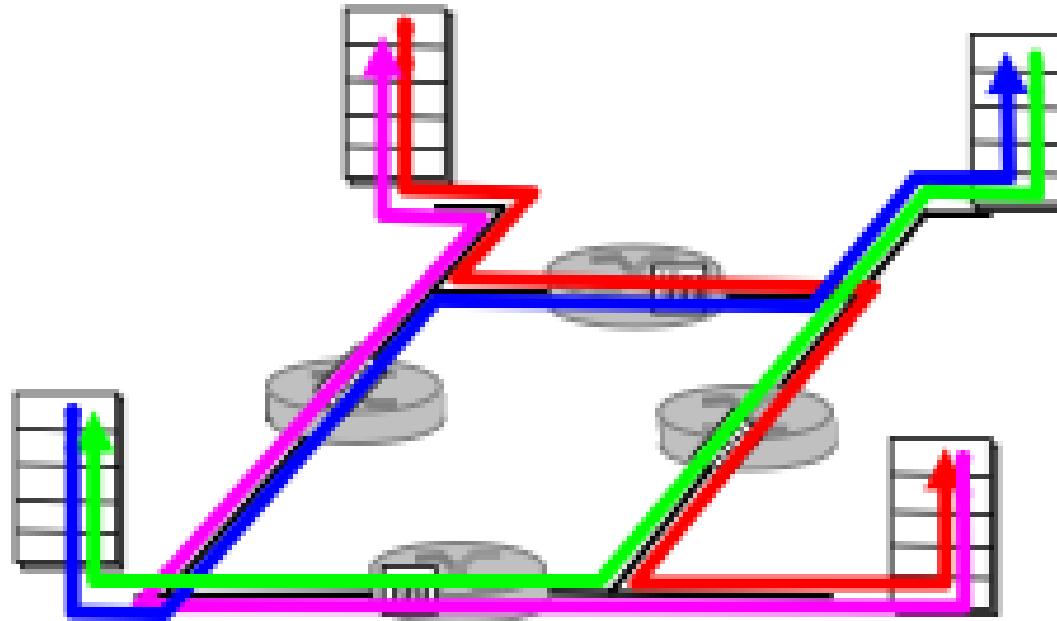
- Four senders
- Multi-hop paths
- Timeout/retransmit

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 $\rightarrow 0$



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.