# Lecture 10: Transport Layer
# Reliable Data Transfer and Seq #s

COMP 332, Spring 2024

Victoria Manfredi

WESLEYAN
UNIVERSITY

**Acknowledgements**: materials adapted from Computer Networking: A Top Down Approach 7th edition: ©1996-2016, J.F Kurose and K.W. Ross, All Rights Reserved as well as from slides by Abraham Matta at Boston University, and some material from Computer Networks by Tannenbaum and Wetherall.

# Today

## Announcements

- homework 4 due Thursday, 11:59p

## Recap

- reliable data transport over channels with errors and loss

## Pipelined protocols

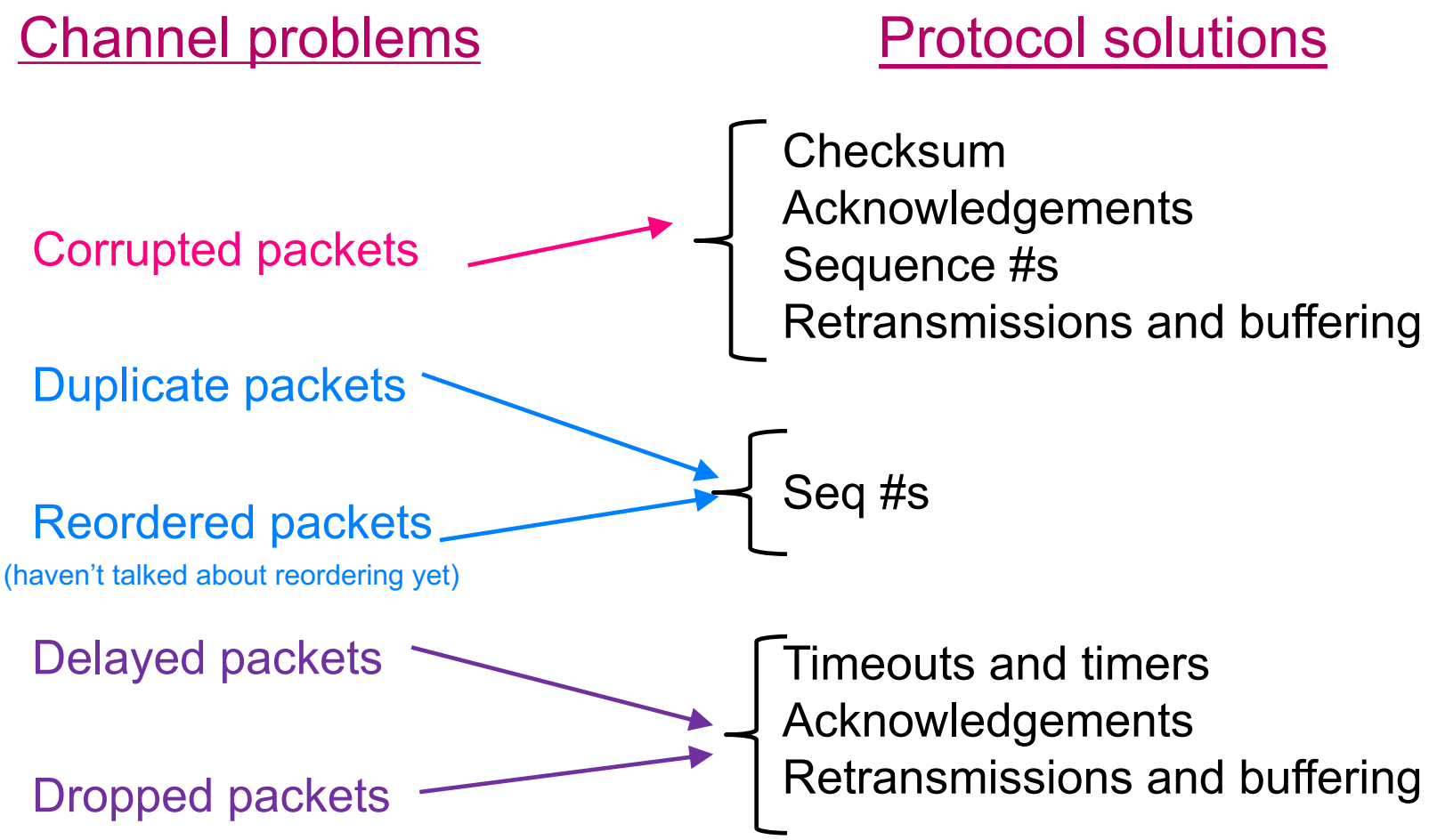
- go-back-N
- selective repeat
- sequence numbers in practice

## TCP overview

# Reliable Data Transport

## REVIEW

# Summary of techniques and uses




Will see: # of seq #s must be > 2x window size if reordering

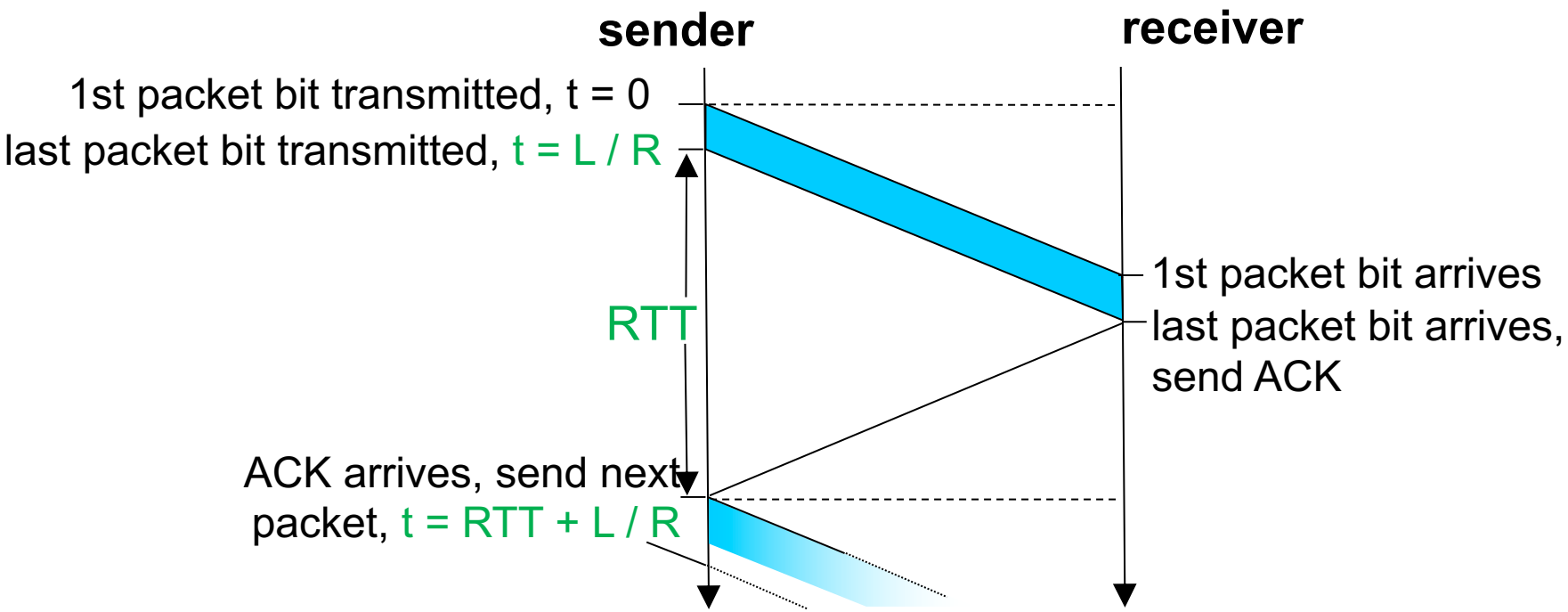
# Are we done?

Have we solved reliable communication over an unreliable channel?

# Reliable Data Transport

# PIPELINED PROTOCOLS

# rdt3.0: stop-and-wait operation




Time spent sending stuff

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

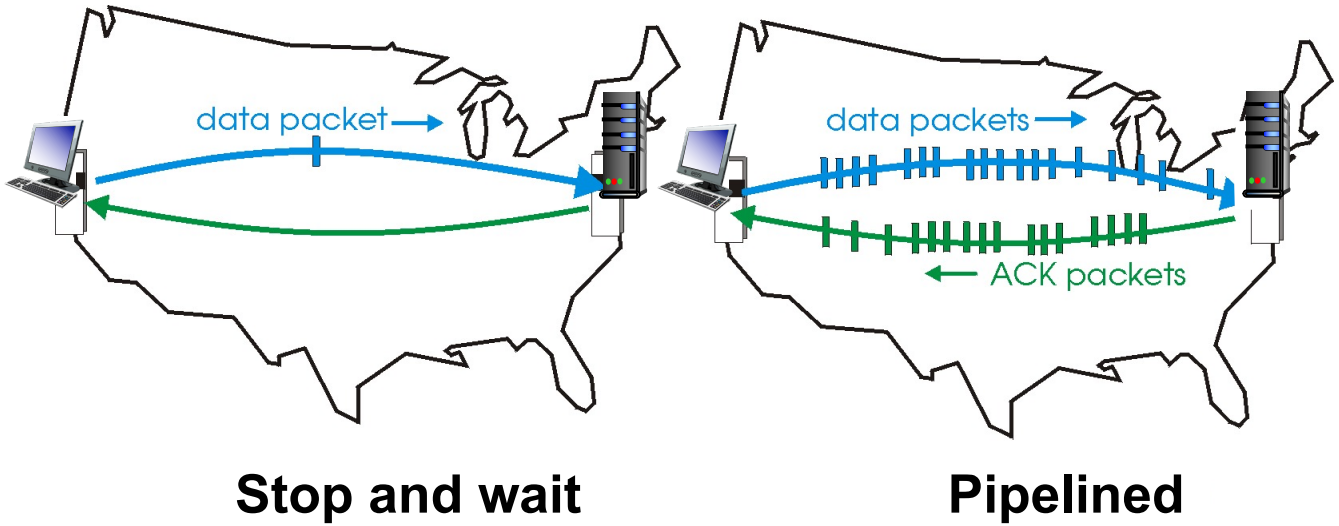
Total time

Problem: how to maintain high link utilization?

# Get rid of stop-and wait

Use pipelining (aka sliding-window protocols), like in HTTP

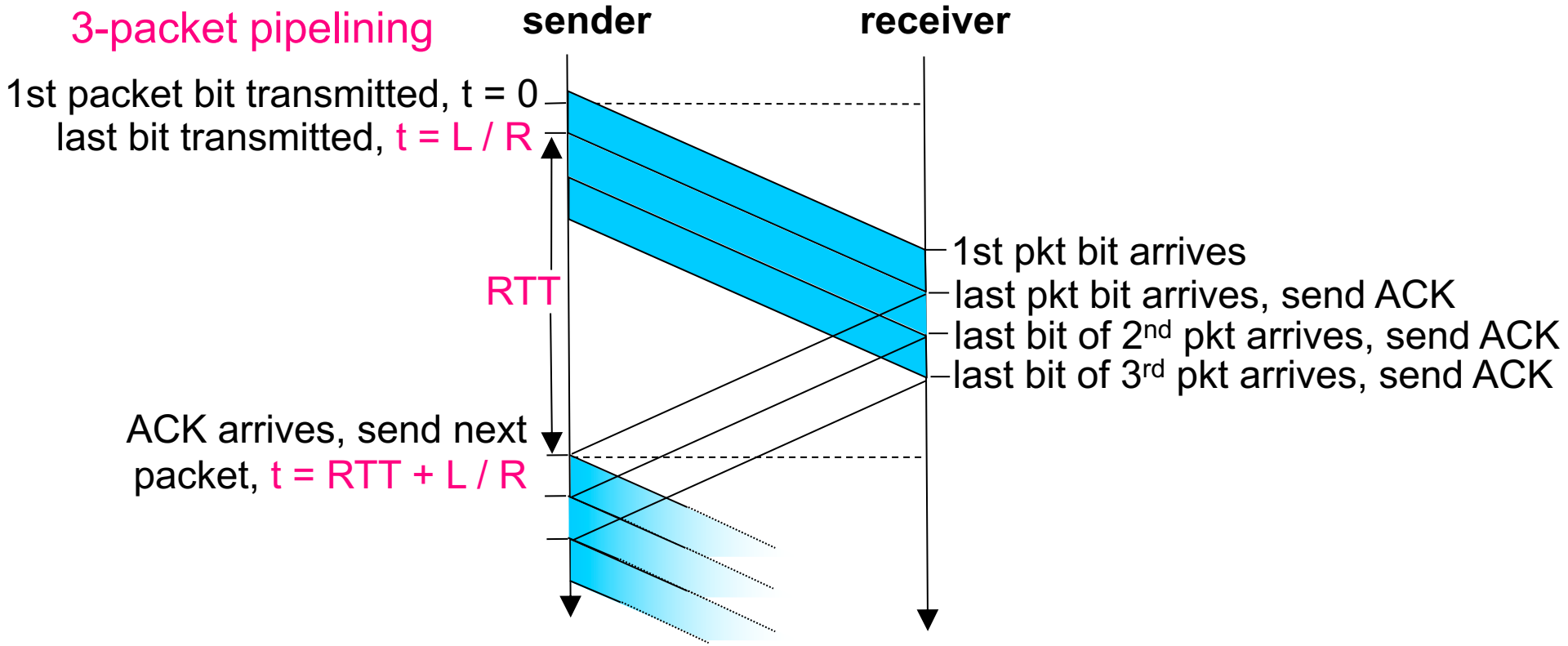
- sender allows multiple, in-flight, yet-to-be-acknowledged pkts
  - send up to N packets at a time, unacked
  - range of seq #s must be increased
  - sender needs more memory to buffer outstanding unacked packets




Achieves higher link utilization than stop-and-wait!

# Increased utilization with pipelining

3-packet pipelining




$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Time spent sending stuff

Total time

3-packet pipelining increases utilization by factor of 3!

# Pipelined protocols

Send N packets without receiving ACKs. How to ACK now?

Cumulative ACKs: Go-Back-N protocol

- **sender**
  - has timer for oldest unacked pkt
  - when timer expires: retransmit all unacked pkts
  - pkts received correctly may be retransmitted
- **receiver** only sends cumulative ack, doesn't ack pkt if gap

Selective ACKs: Selective Repeat protocol

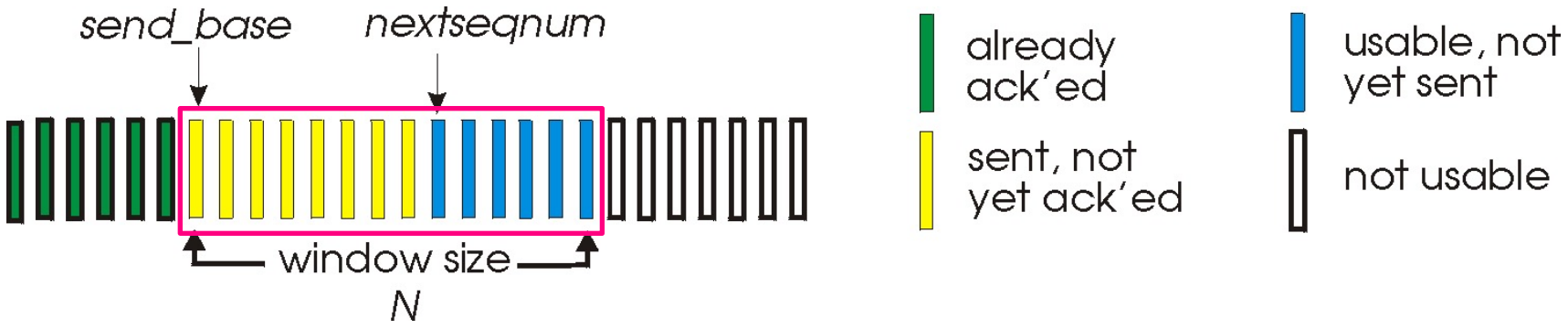
- **sender**
  - has timer for each unacked pkt
  - when timer expires, retransmit only unacked pkt
  - only corrupted/lost pkts are retransmitted
- **receiver** sends individual ack for each pkt

# How pipelining protocols work

What is window size on stop and wait protocol?

## Use sliding window

- how sender keeps track of what it can send
- window: set of N adjacent seq #s
  - only send packets in window




If window large enough, will fully utilize link

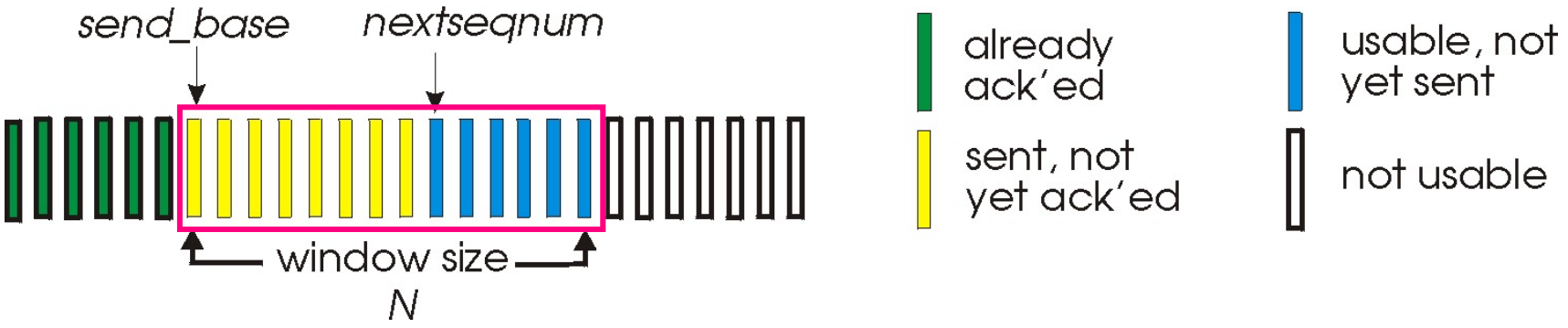
# Pipelined Protocols

## GO-BACK-N

# Go-Back-N: sender

Window of up to N consecutive unacked pkts allowed

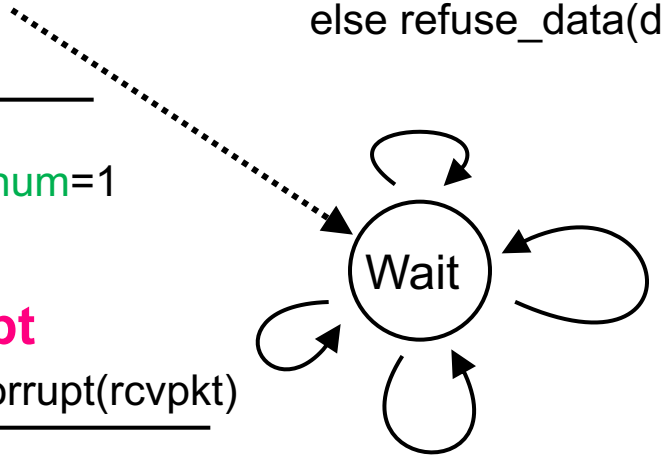
- ACK(n) is cumulative ACK
  - ACKs all pkts up to, including seq # n
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
  - timeout(n): retransmit packet n and all higher seq # pkts in window



# Go-Back-N: sender FSM

**Send as long as pkt within window**

```
rdt_send(data)
if (nextseqnum < base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum)
      start_timer
   nextseqnum++
}
else refuse_data(data)
```




**Resend up to nextseqnum on timeout**

```
timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

**Ignore corrupt**

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
Λ
```

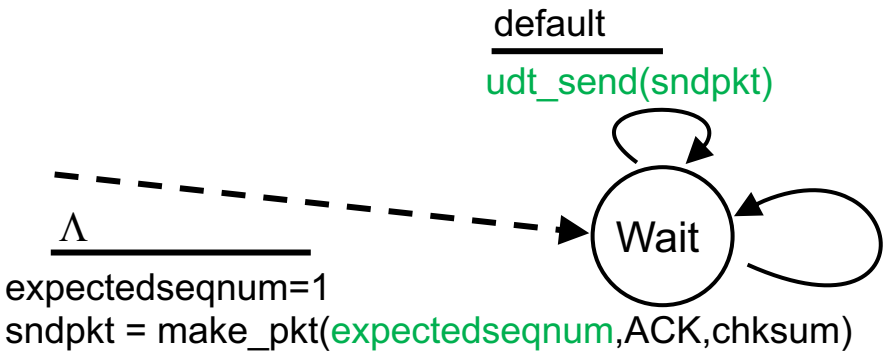
```
rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
   stop_timer
  else
   start_timer
```

**Cumulative ack: move base to ack# + 1**

# Go-Back-N: receiver FSM

**Out-of-order pkt and all other cases**
- discard: no receiver buffering!
- re-ACK pkt with highest in-order seq #

**Correct pkt with highest in-order seq #**
- send ACK, may be duplicate ACK
- need only remember expectedseqnum



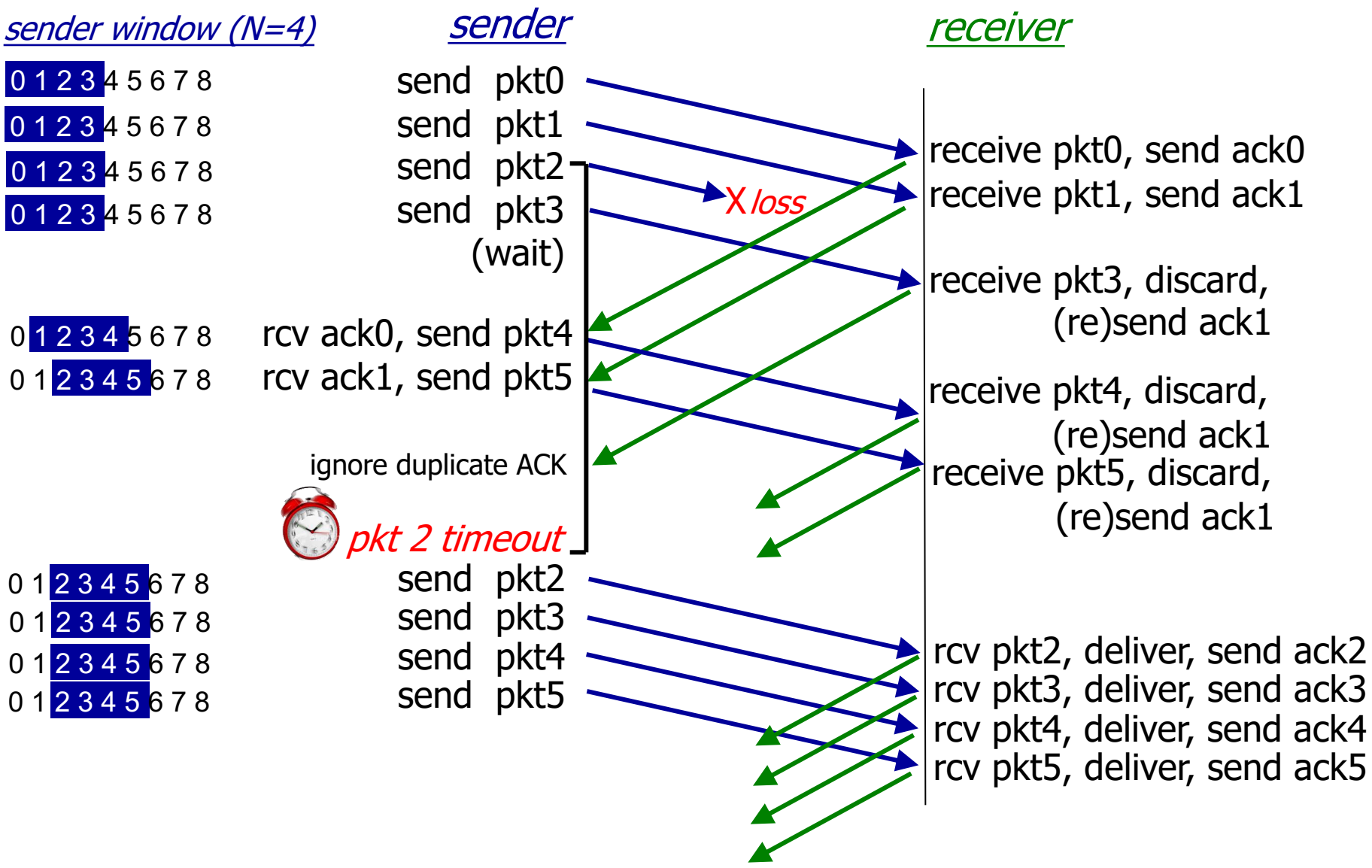

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++

Retransmit windowsize worth of packets for 1 error

large window size ⇒ large delays

# Go-Back-N in action



# Go-Back-N summary

Pros

- no receiver buffering
  - saves resources by requiring packets to arrive in-order
  - avoids large bursts of packet delivery to higher layers
- simpler buffering & protocol processing
  - can easily detect duplicates if out-of-sequence packet is received

Cons

- wastes capacity
  - on timeout for packet N sender retransmits from N all over again (all outstanding packets) including potentially correctly received packets

Tradeoff: buffering/processing complexity vs. capacity (time vs. space)

# Pipelined Protocols

## SELECTIVE REPEAT

# Selective repeat

Rather than ACK cumulatively, ACKs selectively

Receiver

- individually ACKs all correctly received pkts
- buffers pkts, as needed, for eventual in-order delivery to upper layer

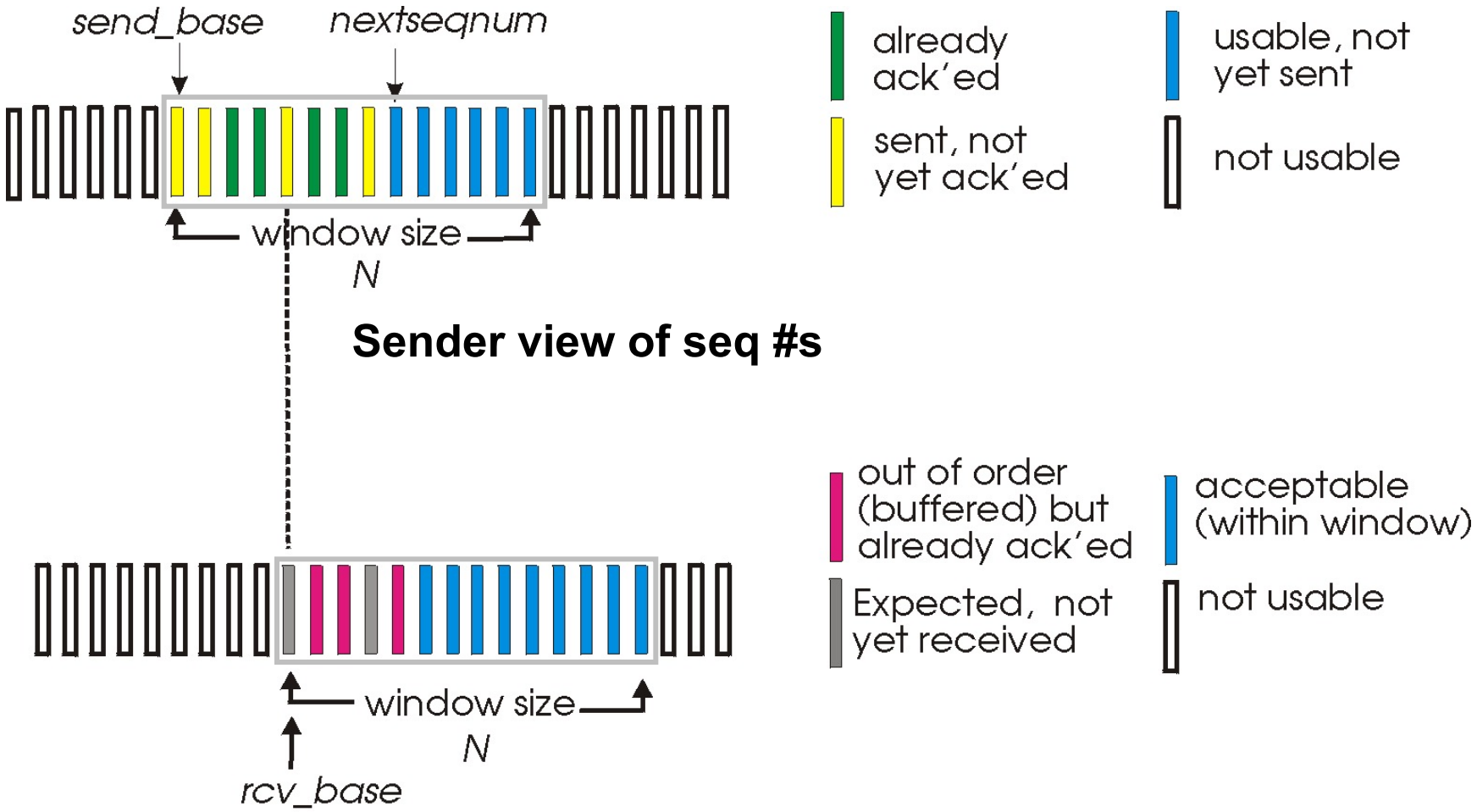
Sender

- 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

**Event:** data from above

- **action:** if has next available seq # in window, send packet, start timer

**Event:** timeout(n)

- **action:** resend packet n, restart timer

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

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

# Selective repeat receiver

**Event:** pkt n in [rcvbase, rcvbase+N-1]

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

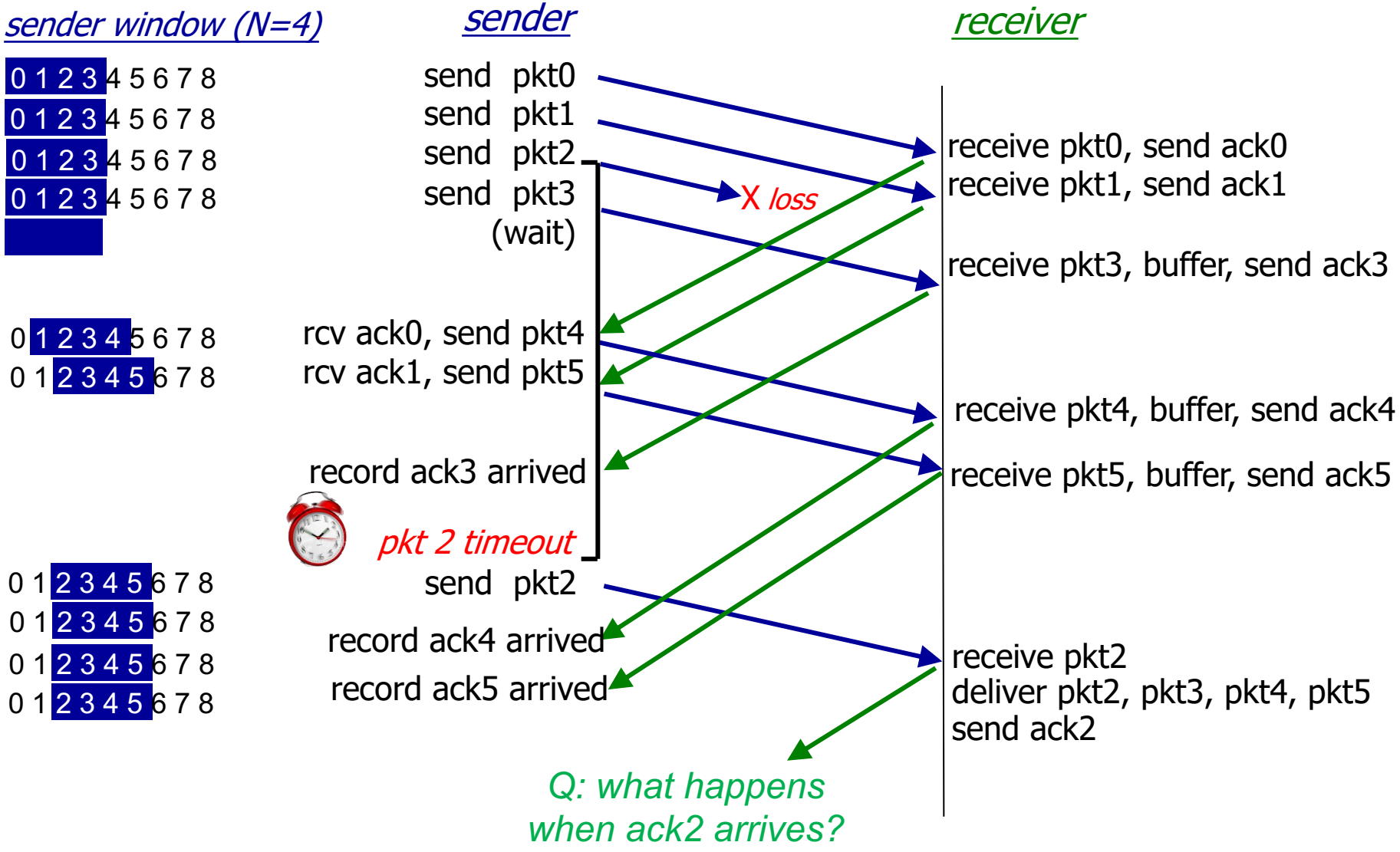
**Event:** pkt n in [rcvbase-N, rcvbase-1]

- **action:** send ACK(n)

**Event:** otherwise

- **action:** ignore

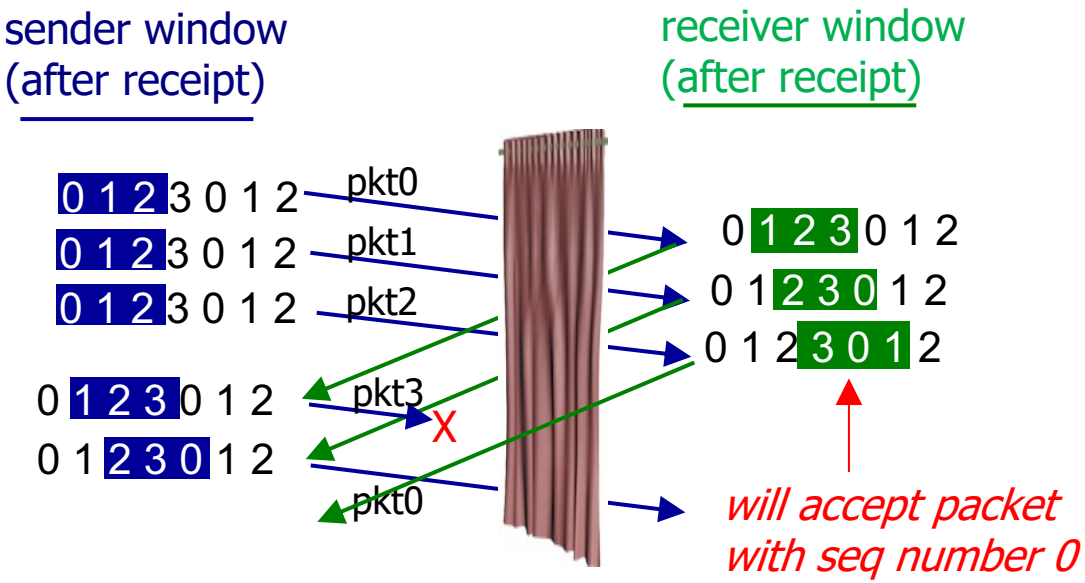
# Selective repeat in action



# Selective repeat: dilemma

## Example

– seq #'s: 0, 1, 2, 3 and window size=3

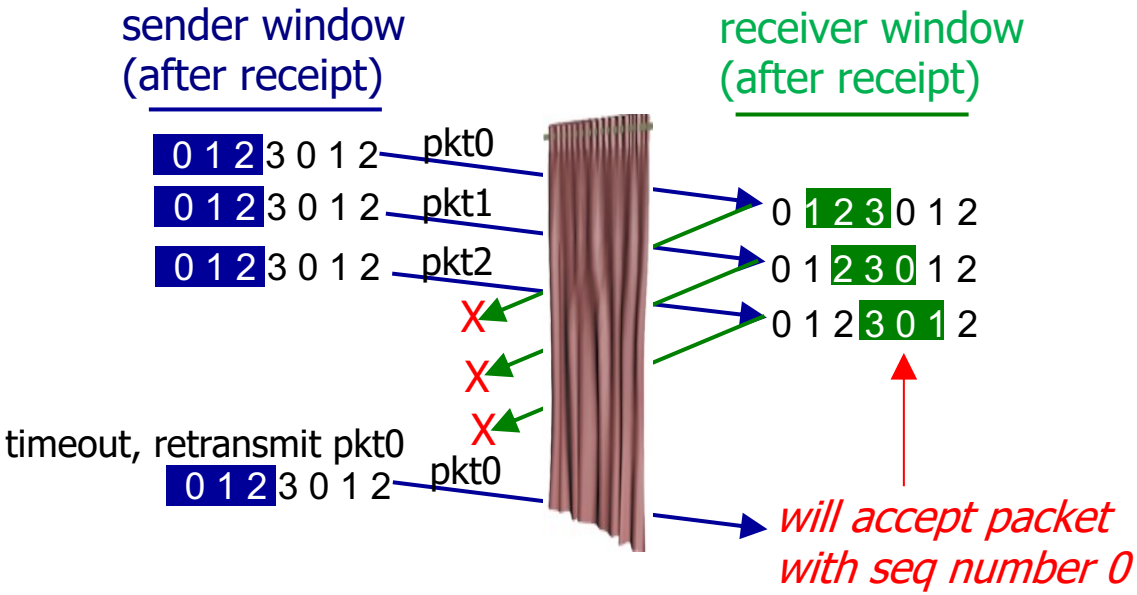



No problem...

# Selective repeat: dilemma

## Example

– seq #'s: 0, 1, 2, 3 and window size=3




Problem: duplicate data accepted as new:

receiver sees no difference in two scenarios!

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

# Selective repeat summary

Q: When is selective repeat useful?

When channel generates errors frequently

Pros

- more efficient capacity use
  - only retransmit missing packets

Cons

- receiver buffering
  - to store out-of-order packets
- more complicated buffering & protocol processing
  - to keep track of missing out-of-order packets

Tradeoff again between buffering/processing complexity and capacity

# Sequence numbers

## HOW USED IN PRACTICE

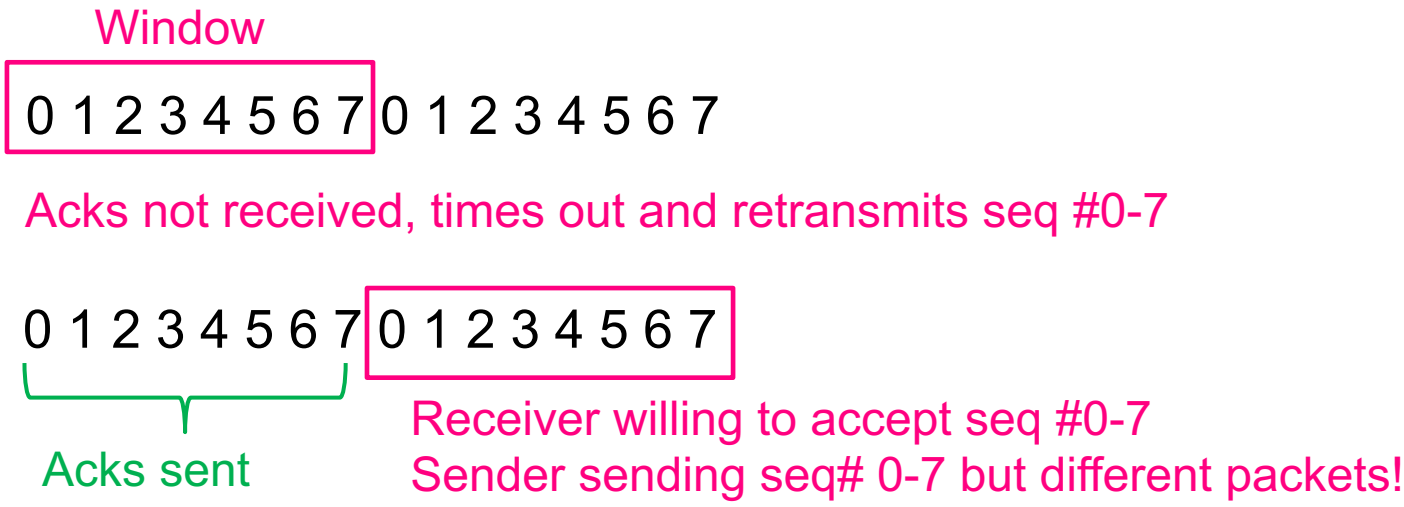
# Sequence #s in practice

## How large must seq # space be?

- depends on window size

## Example

- seq # space = [0, $2^4$-1]
- window size = 8




**Solution:** seq # space must be large enough to cover both sender + receiver windows. I.e., >= 2x window size

# Sequence #s in practice

## What are they counting?

- bytes, not packets
  - sending packets but counting bytes
  - so seq #s do not increase incrementally

## Sequence # space

- finite
  - e.g., 32 bits so 0 to $2^{32}$-1 values
  - must wrap around to 0 when hit max seq #
- TCP initial seq # is randomly chosen from space of values
  - security (harder to spoof)
  - to prevent confusing segments from different connections
  - different operating systems set differently: can fingerprint machines

# TCP

# OVERVIEW

# Transmission Control Protocol (TCP)

**RFCs: 793,1122,1323, 2018, 2581**

Main transport protocol used in Internet, provides

- mux/dmux: which packets go where
- connection-oriented, point-to-point
  - 2 hosts set up connection before exchanging data, tear down after
  - bidirectional data flow (full duplex)
- flow control: don't overwhelm receiver
- congestion control: don't overwhelm network
- reliable: resends lost packets, checks for and corrects errors
- in-order: buffers data until sequential chunk to pass up
- byte stream: no msg boundaries, data treated as stream

# How does TCP provide these services?

Using many techniques we already talked about

## Sliding window

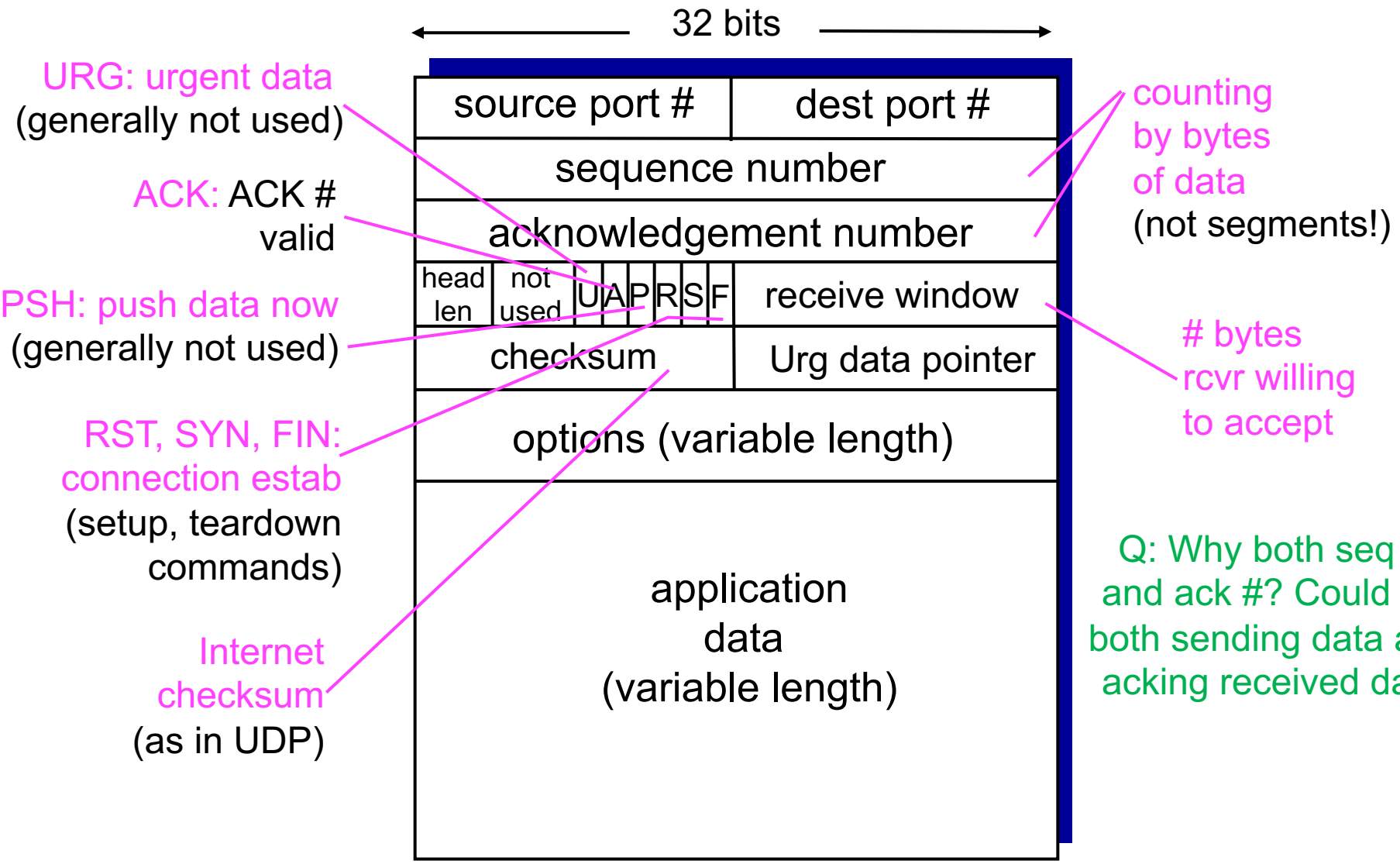
- congestion and flow control determine window size
- seq #s are byte offsets

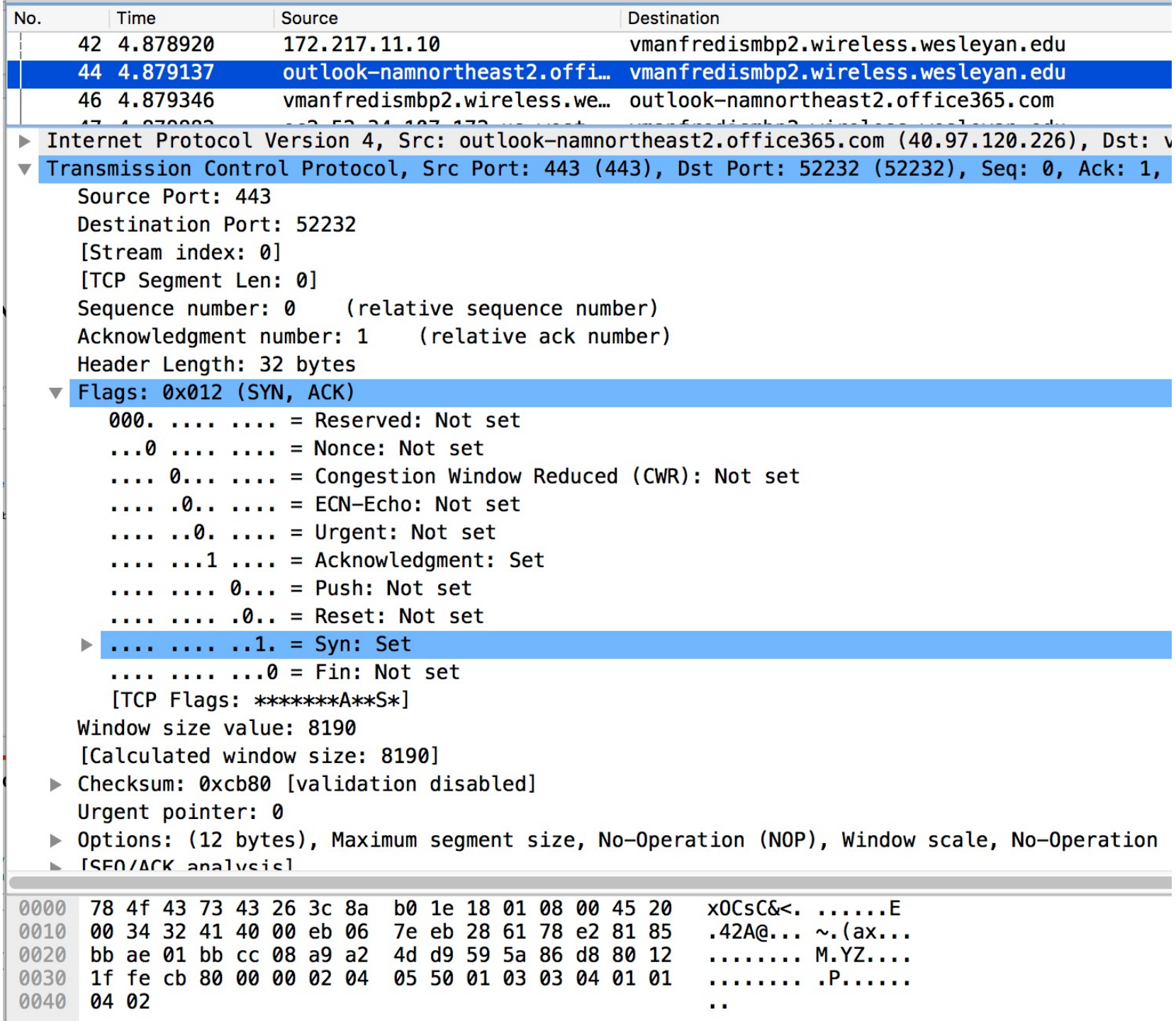
## Cumulative ACKs but does not drop out-of-order packets

- **only one retransmission timer**
  - intuitively, associate with oldest unACKed packet
- **timeout period**
  - estimated from observations
- **fast retransmit**
  - 3 duplicate ACKs trigger early retransmit

TCP is not perfect but works pretty well!

# TCP segment structure




No. Time Source Destination
42 4.878920 172.217.11.10 vmanfredismbp2.wireless.wesleyan.edu
44 4.879137 outlook-namnortheast2.offi… vmanfredismbp2.wireless.wesleyan.edu
46 4.879346 vmanfredismbp2.wireless.we… outlook-namnortheast2.office365.com
Internet Protocol Version 4, Src: outlook-namnortheast2.office365.com (40.97.120.226), Dst:
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52232 (52232), Seq: 0, Ack: 1,
Source Port: 443
Destination Port: 52232
[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 0 (relative sequence number)
Acknowledgment number: 1 (relative ack number)
Header Length: 32 bytes
Flags: 0x012 (SYN, ACK)
000. .... .... = Reserved: Not set
...0 .... .... = Nonce: Not set
.... 0... .... = Congestion Window Reduced (CWR): Not set
.... .0.. .... = ECN-Echo: Not set
.... ..0. .... = Urgent: Not set
.... ...1 .... = Acknowledgment: Set
.... .... 0... = Push: Not set
.... .... .0.. = Reset: Not set
.... .... ..1. = Syn: Set
.... .... ...0 = Fin: Not set
[TCP Flags: *******A**S*]
Window size value: 8190
[Calculated window size: 8190]
Checksum: 0xcb80 [validation disabled]
Urgent pointer: 0
Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation
0000 78 4f 43 73 43 26 3c 8a b0 1e 18 01 08 00 45 20 xOCsC&<. ......E
0010 00 34 32 41 40 00 eb 06 7e eb 28 61 78 e2 81 85 .42A@... ~.(ax...
0020 bb ae 01 bb cc 08 a9 a2 4d d9 59 5a 86 d8 80 12 ........ M.YZ....
0030 1f fe cb 80 00 00 02 04 05 50 01 03 03 04 01 01 ........ .P......
0040 04 02 ..

# TCP

# SEQ #S AND ACK #S

# TCP seq. numbers, ACKs

## Sequence #s

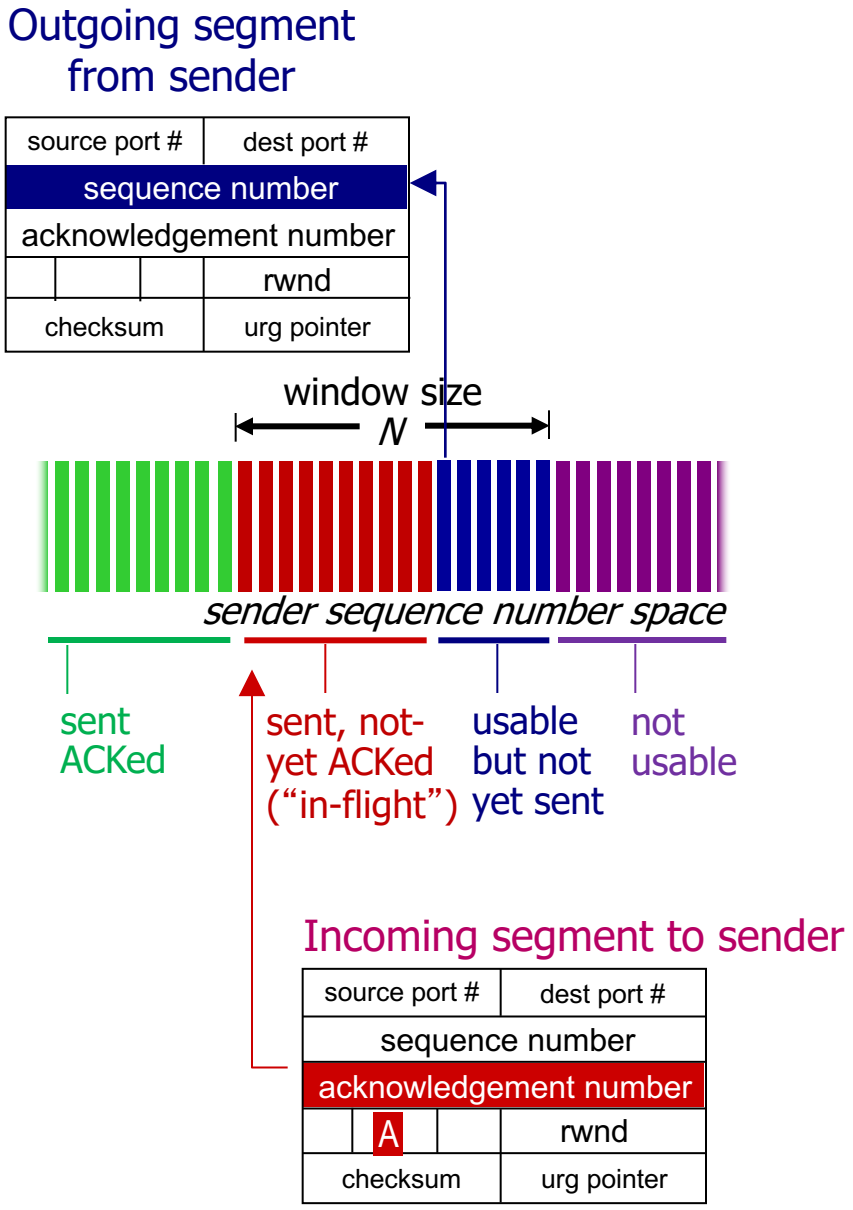
- byte stream # of first byte in segment's data

## Acknowledgements

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

## Q: how does receiver handle out-of-order segments?

- TCP spec doesn't say
- up to implementer

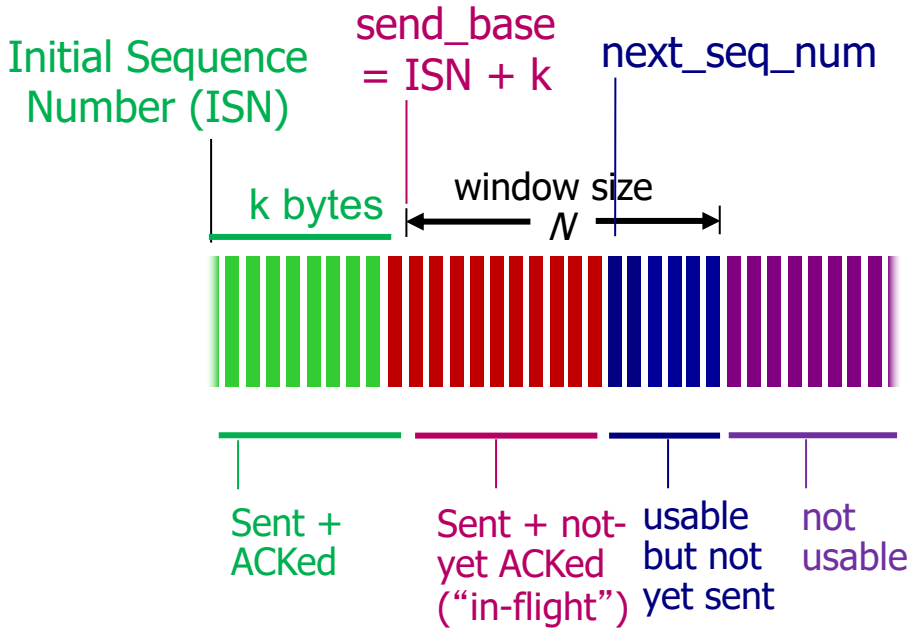


# TCP ACKs

## Cumulative ACKs (but different than in Go-Back-N)

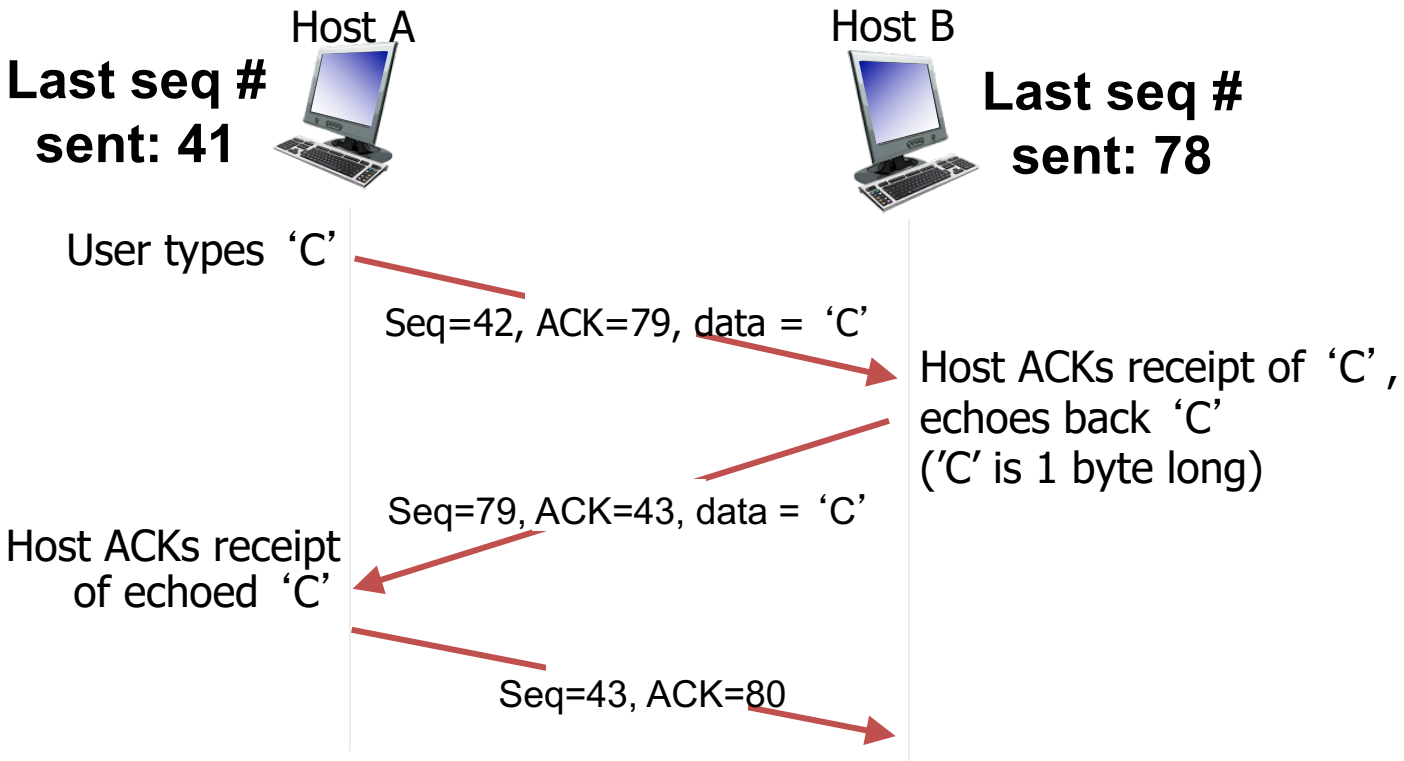
- ACKs what receiver expects next, not last packet received
  - implicitly also ACKs everything up to sequence number received
- only 1 retransmission timer (for first pkt in window)
  - sender retransmits only first pkt in window if no ack when timer expires

## Sequence #s are not sequential: counting bytes not packets



# TCP seq. numbers, ACKs

**Sequence numbers are synchronized during connection set-up**




Simple nc scenario

## Host 1

```
Transmission Control Protocol,
    Source Port: 54573
    Destination Port: 443
    [Stream index: 2]
    [TCP Segment Len: 0]
    Sequence number: 59452065
    Acknowledgment number: 0
    Header Length: 44 bytes
▶   Flags: 0x002 (SYN)
    Window size value: 65535
```

Handshake: Synchronize ISNs

## Host 2

```
Transmission Control Protocol, Src
    Source Port: 443
    Destination Port: 54573
    [Stream index: 2]
    [TCP Segment Len: 0]
    Sequence number: 3712814908
    Acknowledgment number: 59452066
    Header Length: 40 bytes
▶   Flags: 0x012 (SYN, ACK)
    Window size value: 14480
```

Convention: SYN and FIN take 1 byte of seq # space

```
Transmission Control Protocol, Src Po
    Source Port: 54573
    Destination Port: 443
    [Stream index: 2]
    [TCP Segment Len: 212]
    Sequence number: 59452066
    [Next sequence number: 59452278]
    Acknowledgment number: 3712814909
    Header Length: 32 bytes
▶   Flags: 0x018 (PSH, ACK)
    Window size value: 4122
```

Data exchange

```
Transmission Control Protocol, Src Pc
    Source Port: 443
    Destination Port: 54573
    [Stream index: 2]
    [TCP Segment Len: 0]
    Sequence number: 3712814909
    Acknowledgment number: 59452278
    Header Length: 32 bytes
▶   Flags: 0x010 (ACK)
    Window size value: 122
    [Calculated window size: 15616]
    [Window size scaling factor: 128]
```

What are seq and ack #s in next segment from receiver?

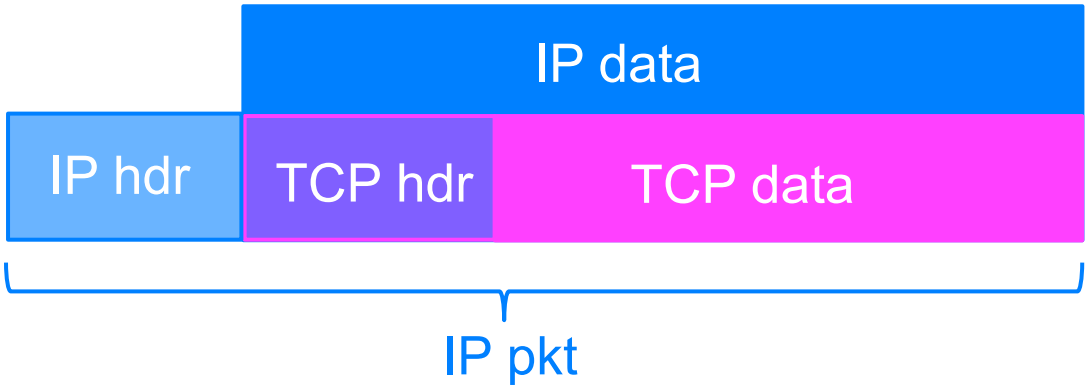
# Segment size

## Max length of IP packet in bytes

- MTU: Maximum Transmission Unit
- 1500 bytes if Ethernet used as link layer protocol

## Max length of TCP data in bytes

- MSS: Maximum Segment Size
- MSS = MTU – IP hdr – TCP hdr
  - TCP header >= 20bytes




TCP segment sent when either it is full (meets MSS) or not full but timeout occurs