# Lecture 11: Transport Layer
# Reliable Data Transfer and Seq #s

COMP 411, Fall 2022

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 5 due Friday at 5p

## Recap

- reliable data transport over channels with errors and loss

## Pipelined protocols

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

# Reliable Data Transport

## CHANNELS WITH ERROR AND LOSS

# rdt3.0: channels with errors and loss

Problems

- underlying channel may flip bits in packet
  - both data and ACKs may be garbled
- underlying channel can also lose packets
  - both data and ACKs
- checksum, seq. #, ACKs, retransmissions will be of help
  - … but not enough

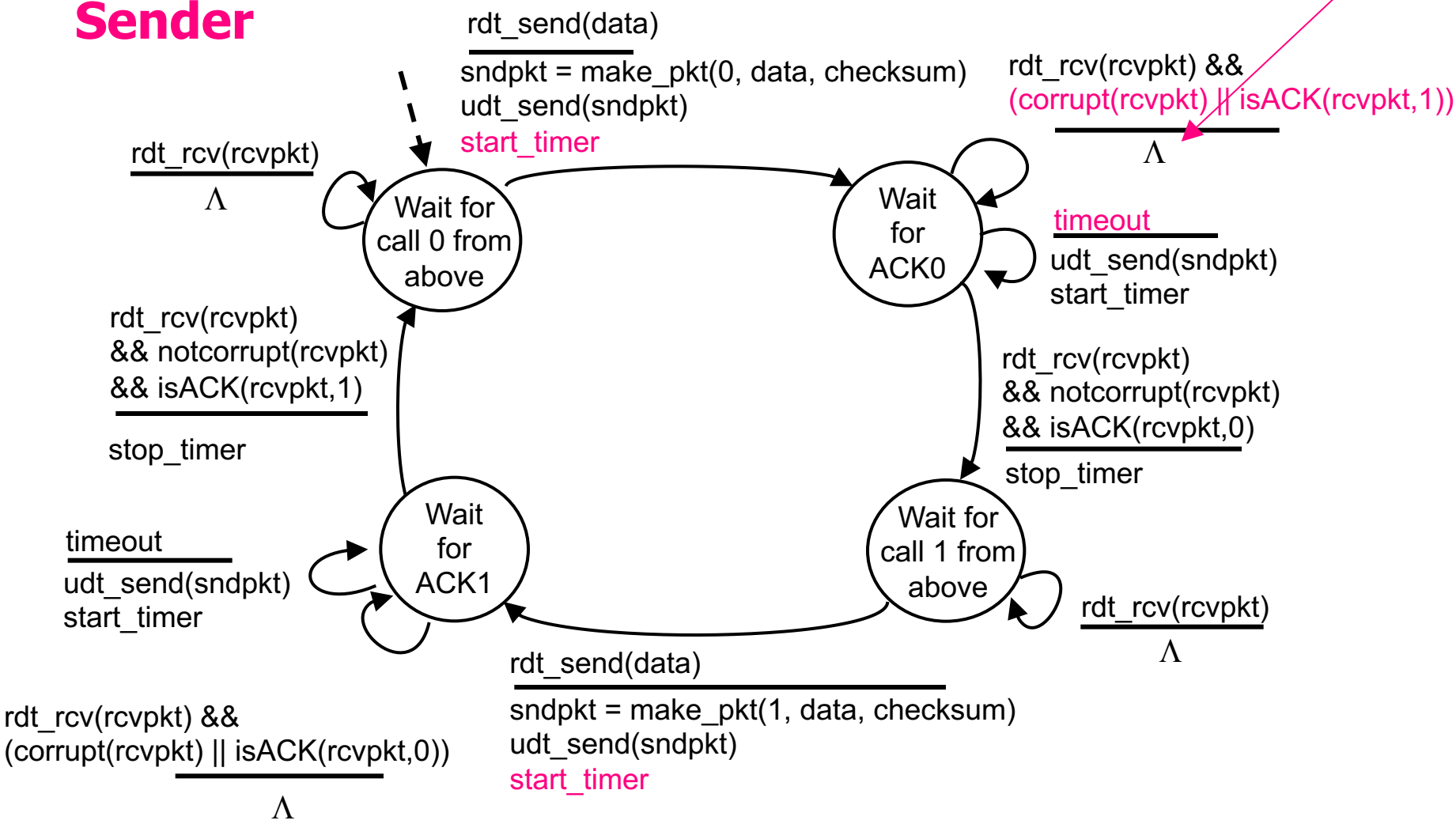
Solution: add countdown timer

- sender waits "reasonable" amount of time for ACK
  - retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost)
  - retransmission will be duplicate, but seq #'s already handles this
- receiver must specify seq # of pkt being ACKed

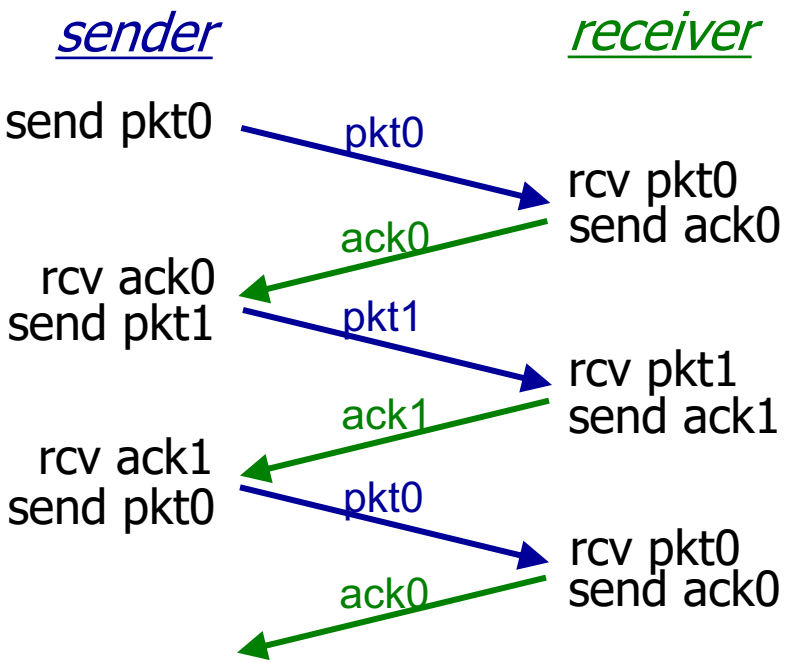
# rdt3.0 sender

Why do nothing ? Why not resend pkt0? Because sender doesn't know whether ack1 means pkt 0 garbled or pkt 1 duplicate received By not resending pkt 0, sender doesn't introduce potentially unnecessary (even if valid) traffic: saves bandwidth

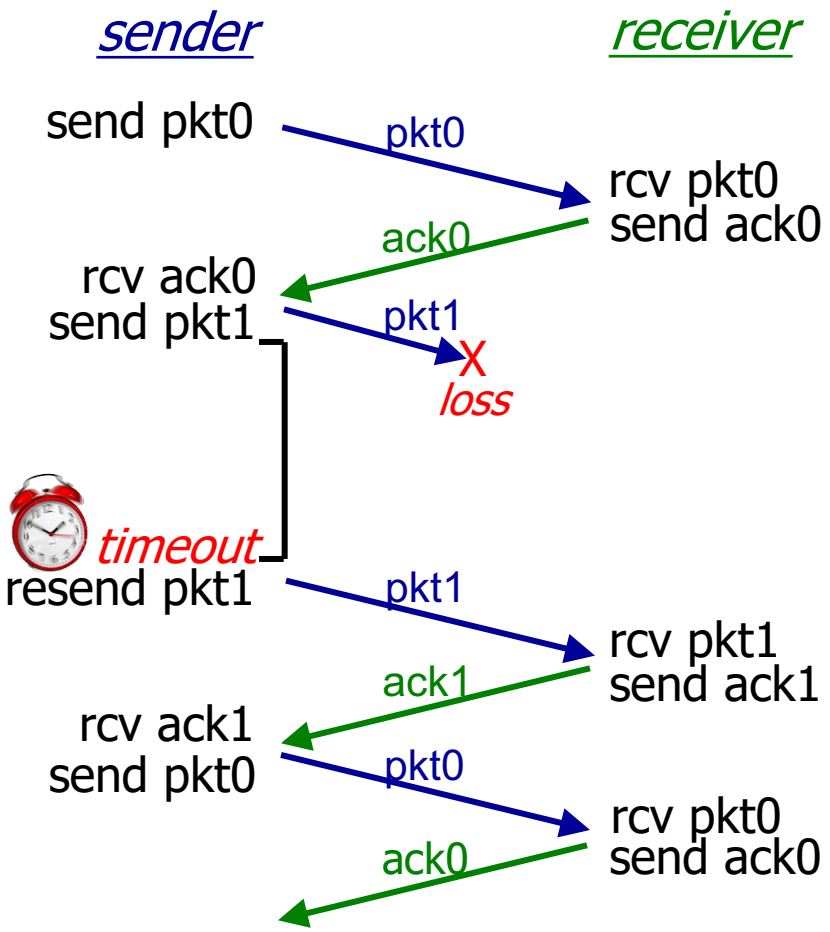
## Sender



# rdt3.0 in action

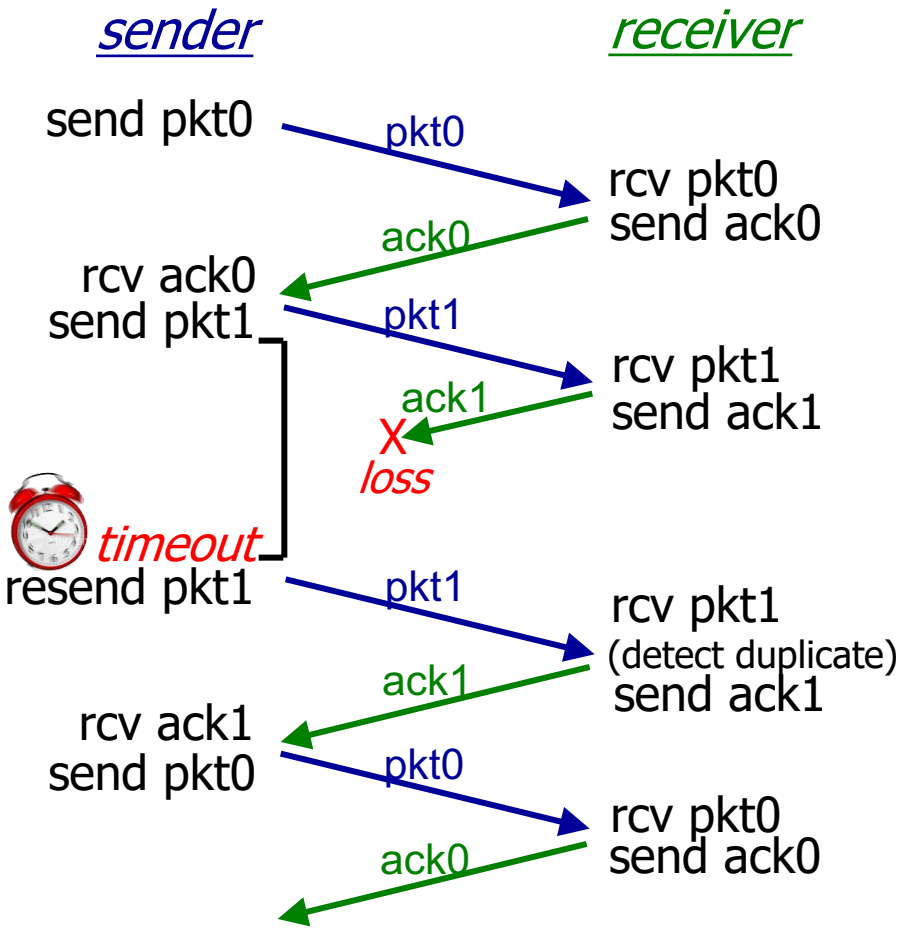



(a) no loss

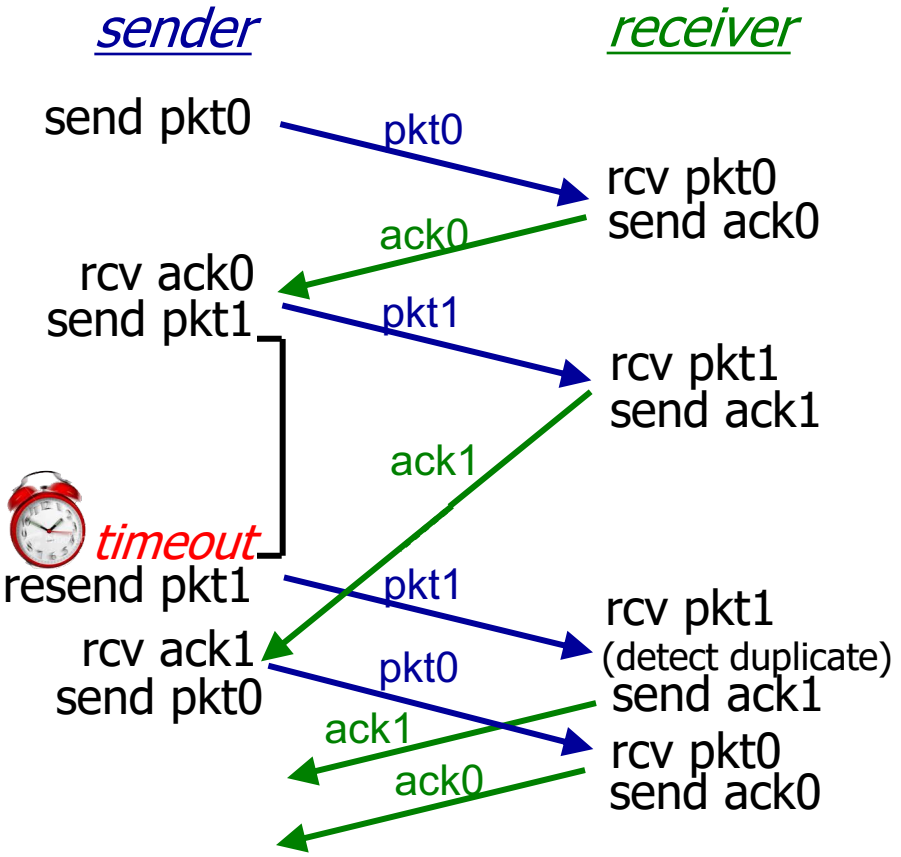



(b) packet loss

# rdt3.0 in action




(c) ACK loss

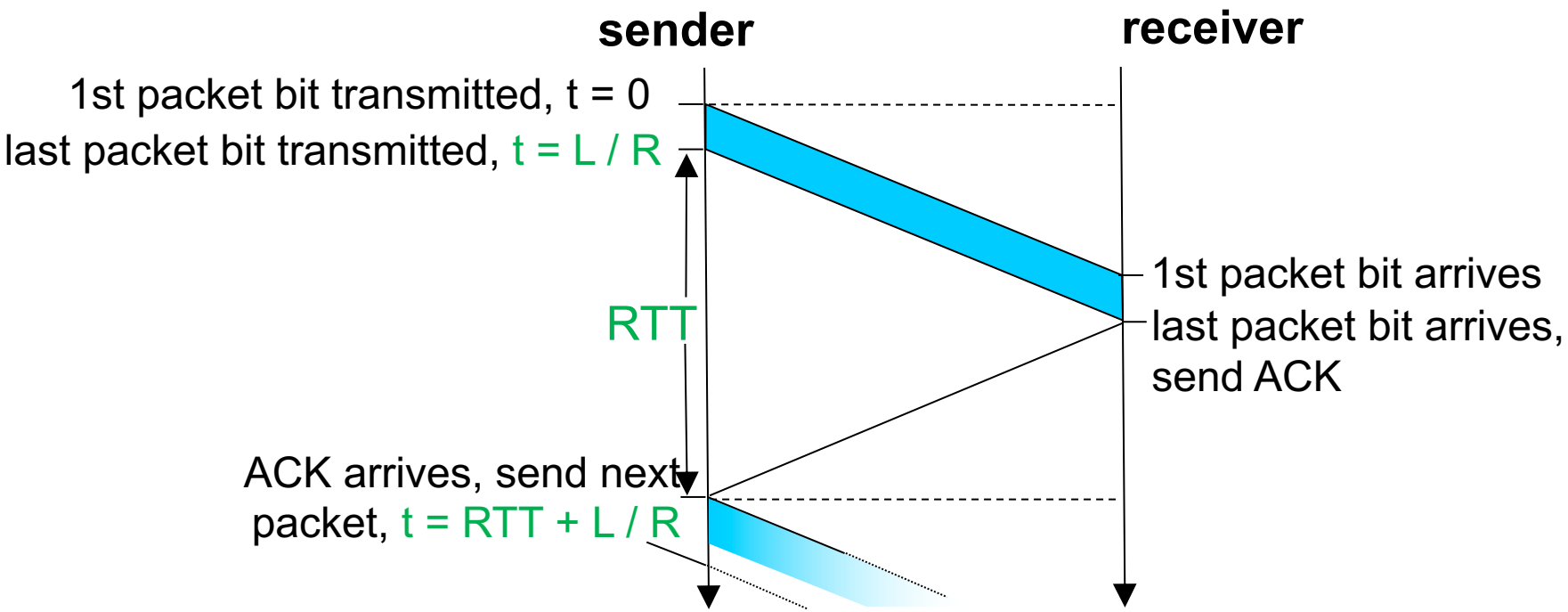



(d) premature timeout/ delayed ACK

# Reliable Data Transport

# PIPELINED PROTOCOLS

# rdt3.0: stop-and-wait operation




Time spent sending stuff

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

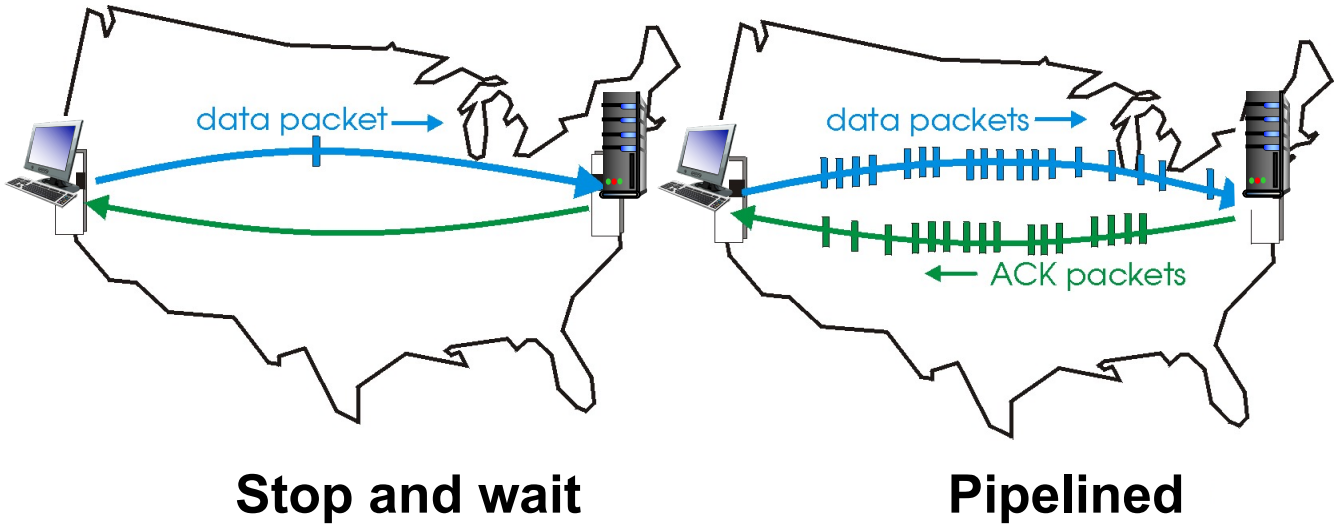
Total time

Problem: how to maintain high link utilization?

# Get rid of stop-and wait

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

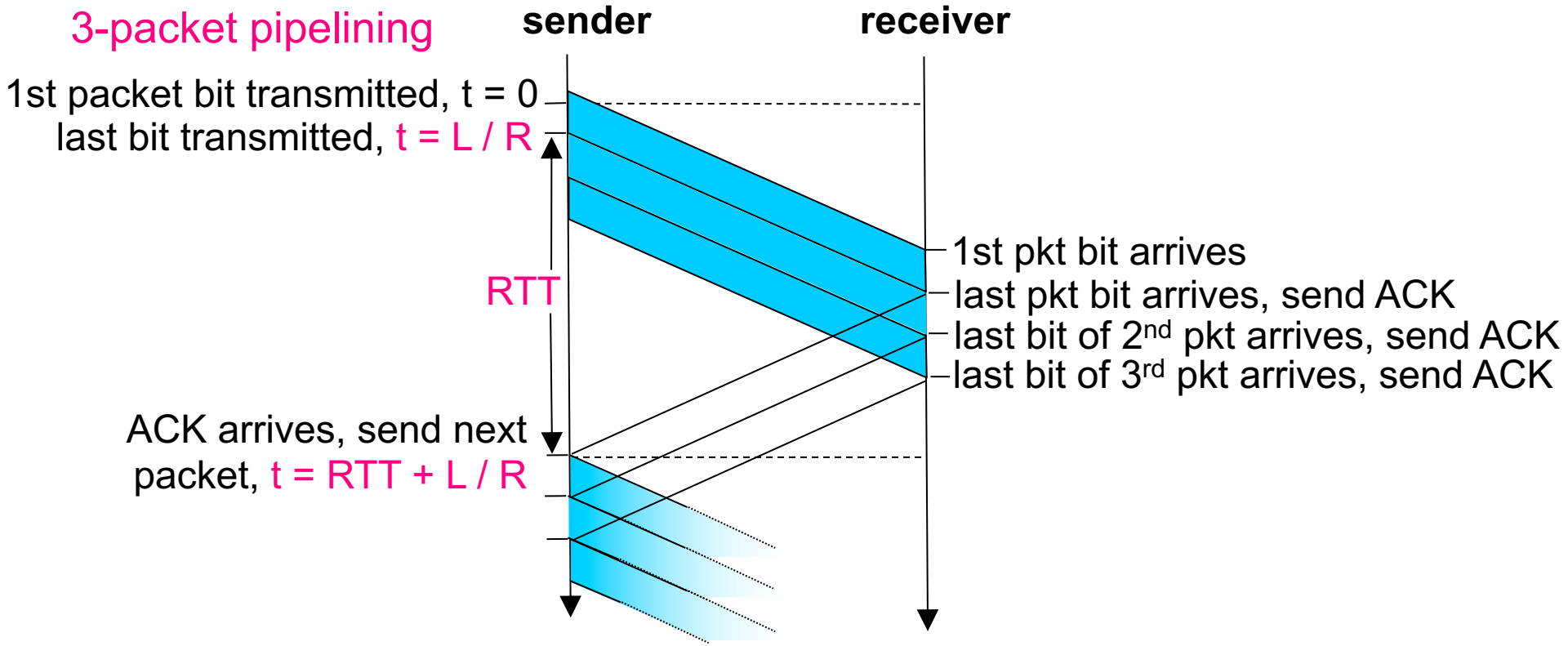
- sender allows multiple, in-flight, yet-to-be-acknowledged pkts
  - send up to N packets at a time, unacked
  - range of seq #s must be increased
  - sender needs more memory to buffer outstanding unacked packets




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

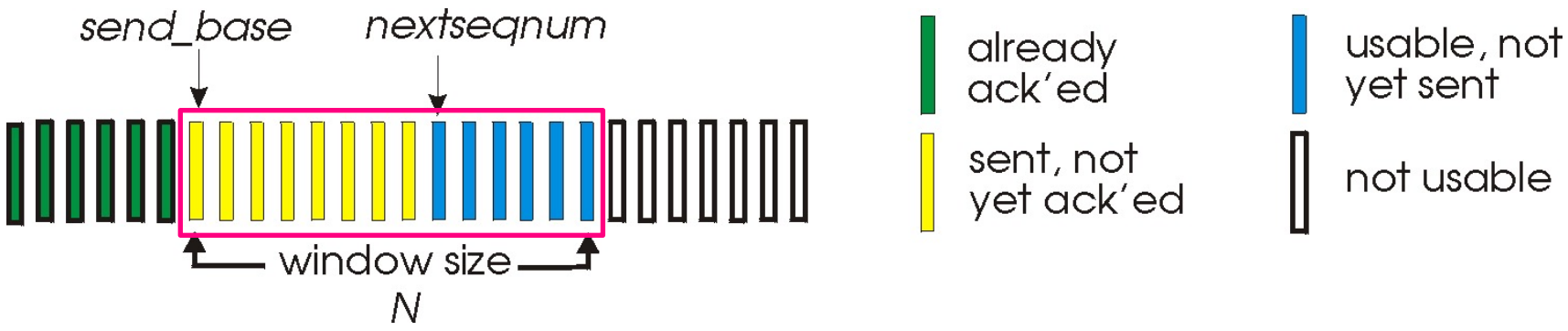
Selective ACKs: Selective Repeat protocol

- **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

## 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

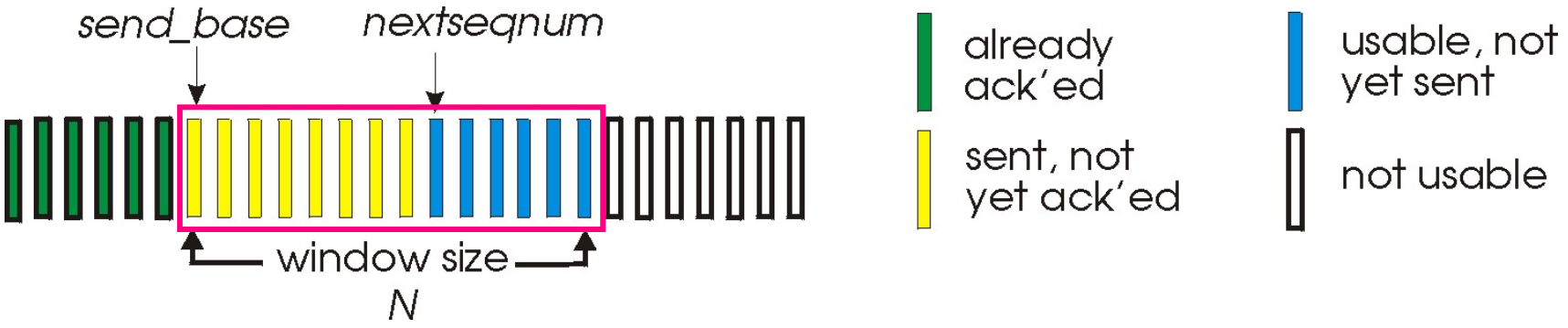
# Pipelined Protocols

## GO-BACK-N

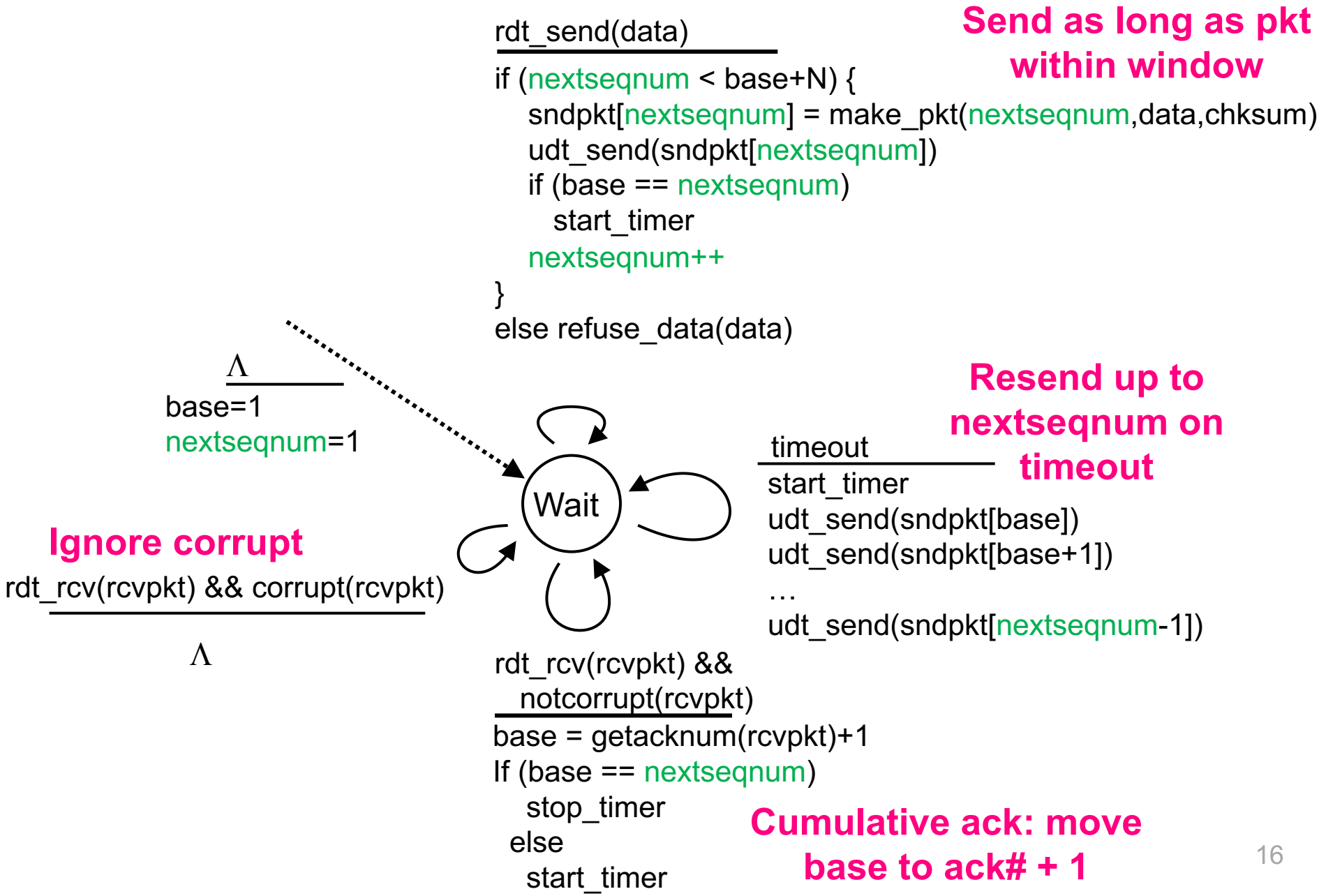
# Go-Back-N: sender

Window of up 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



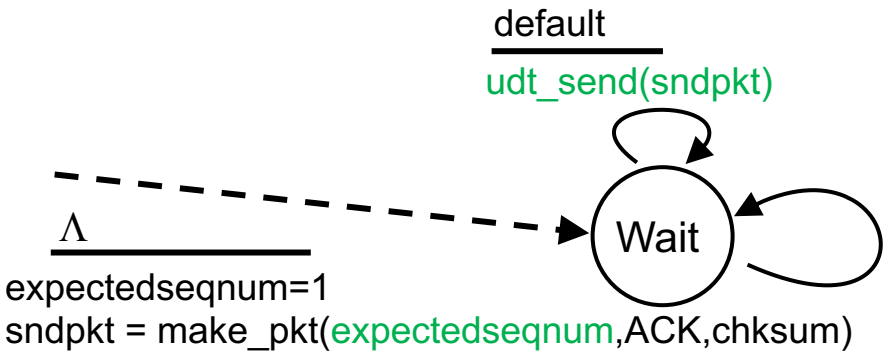
# Go-Back-N: sender FSM



# 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



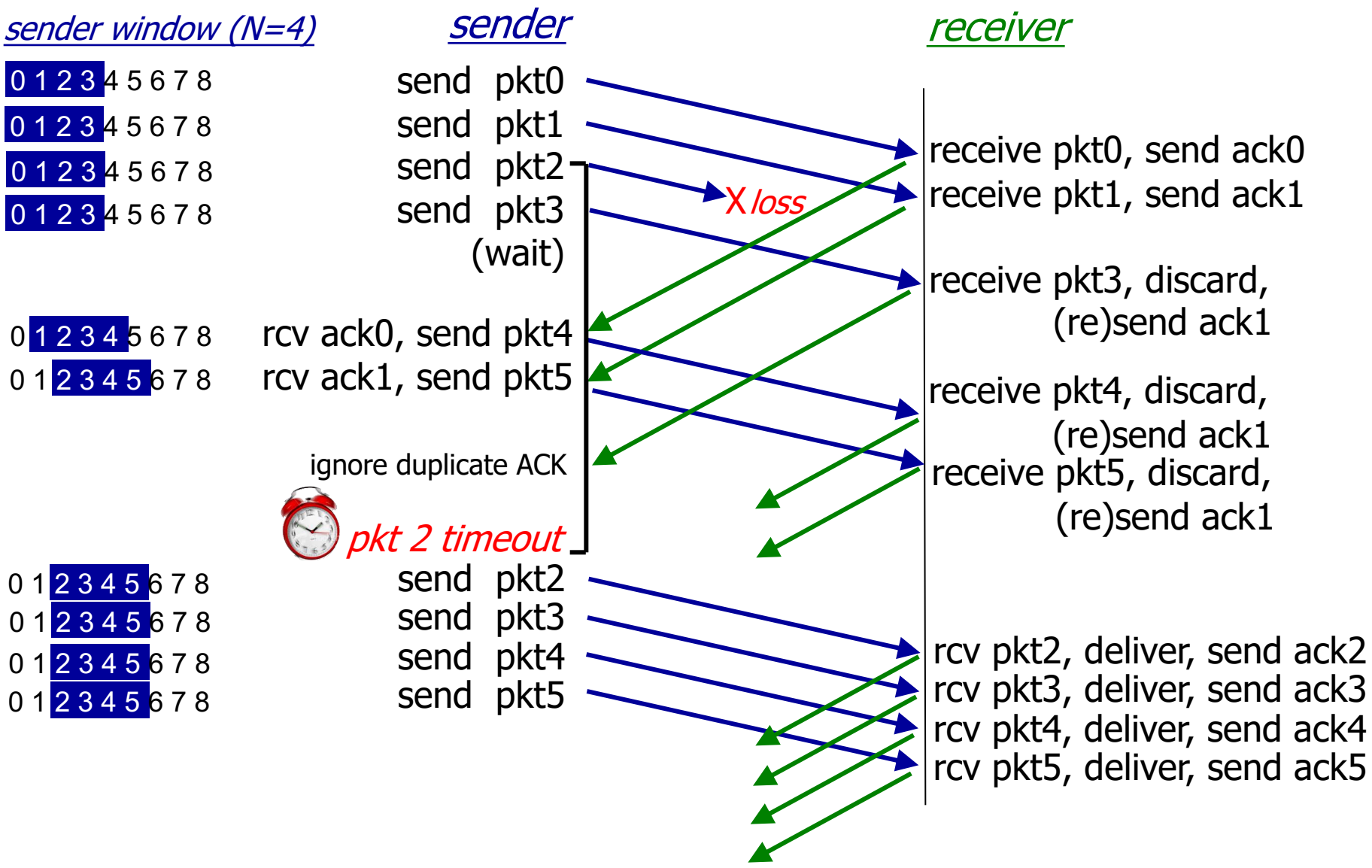

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)

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

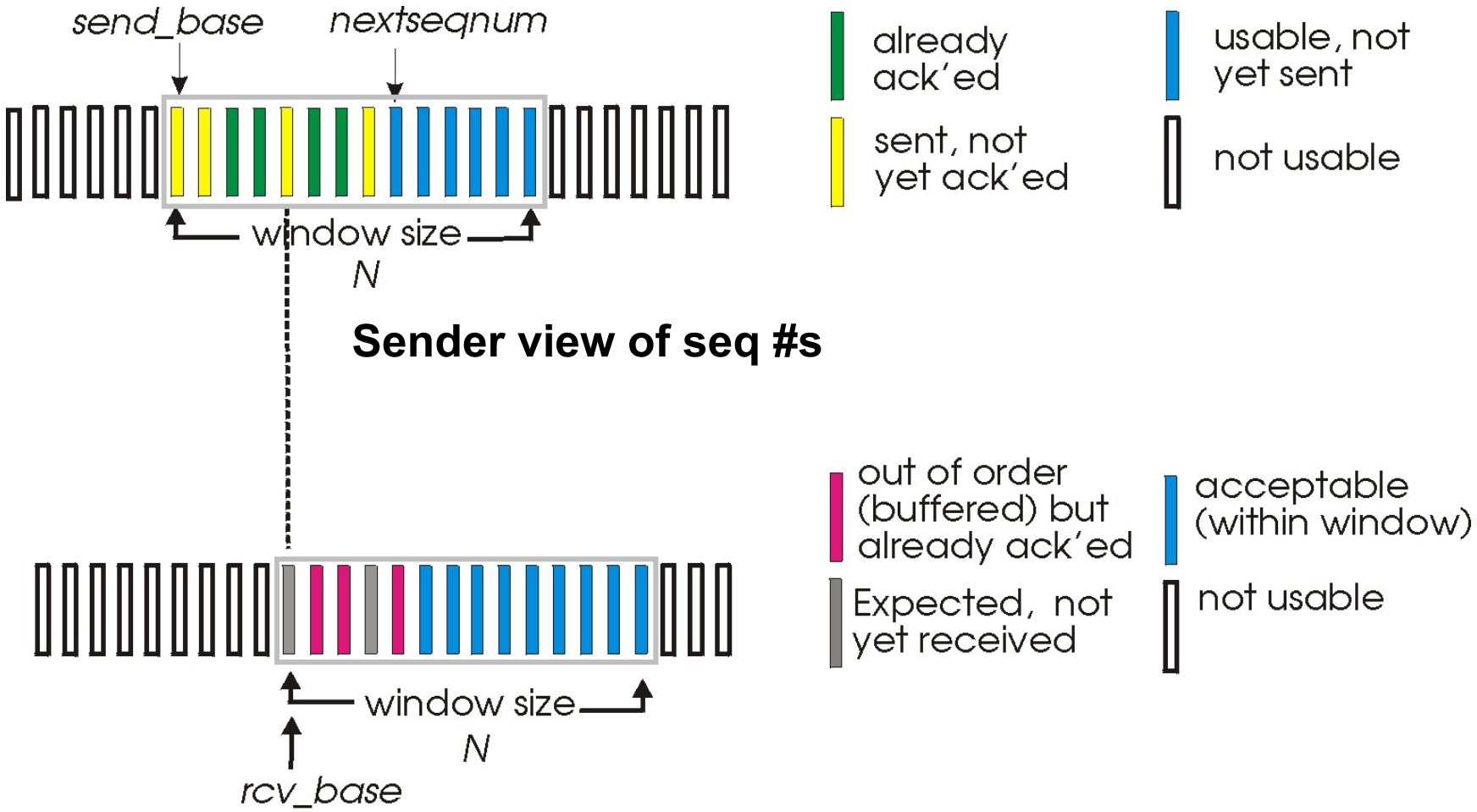
Sender

- 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

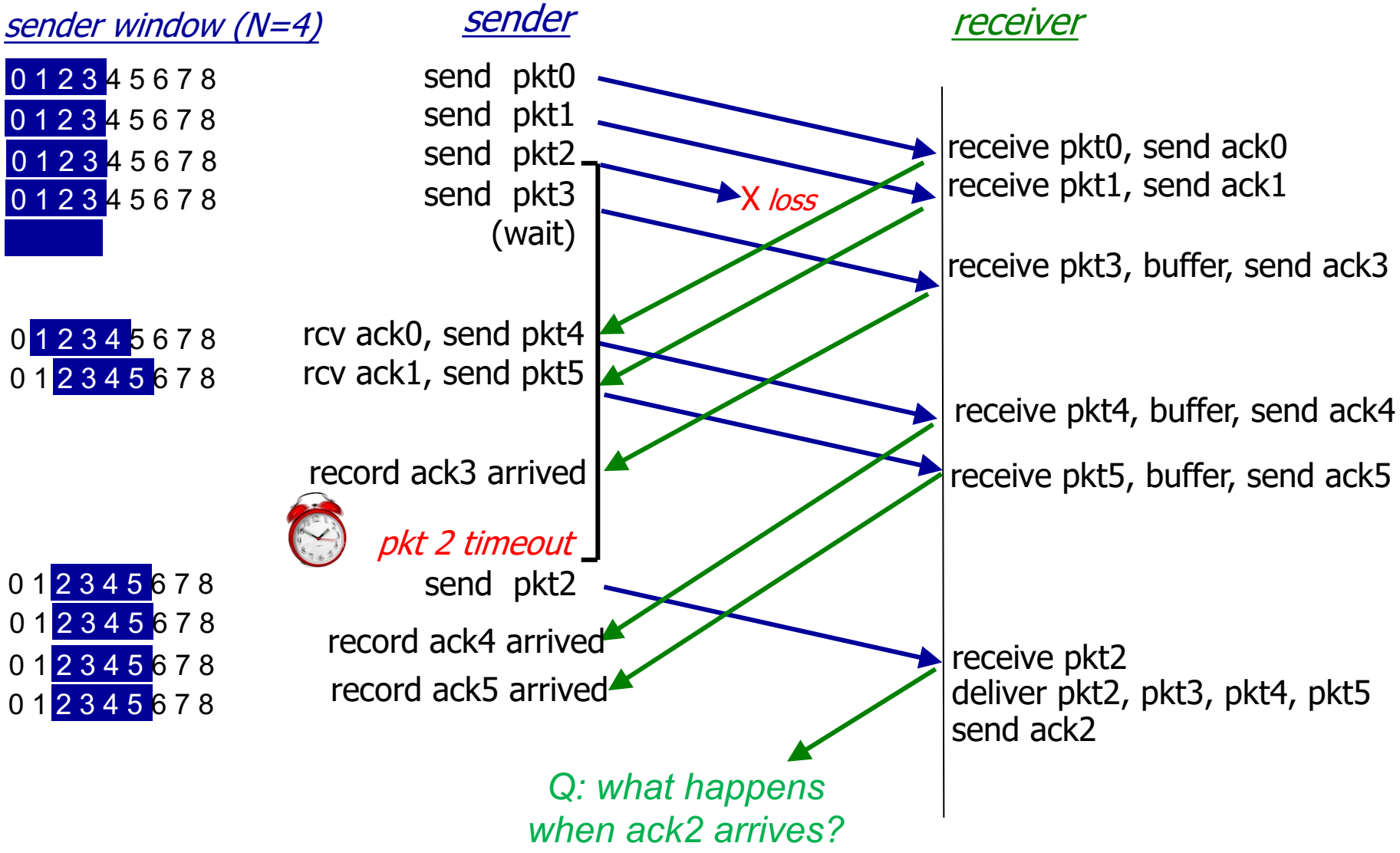
**Event:** pkt n in [rcvbase-N, rcvbase-1]

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

**Event:** otherwise

- **action:** ignore

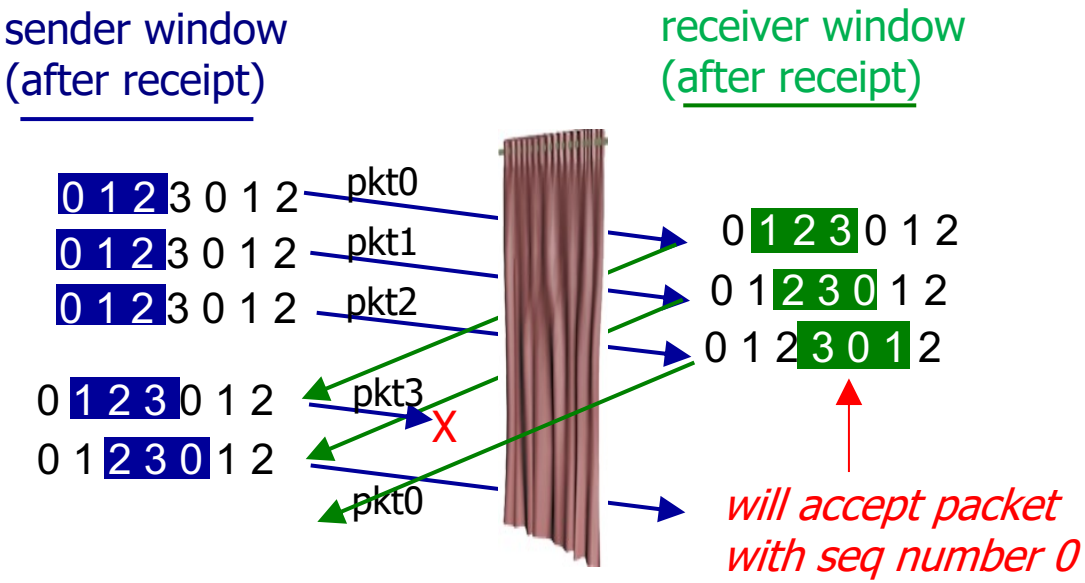
# Selective repeat in action



# Selective repeat: dilemma

## Example

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

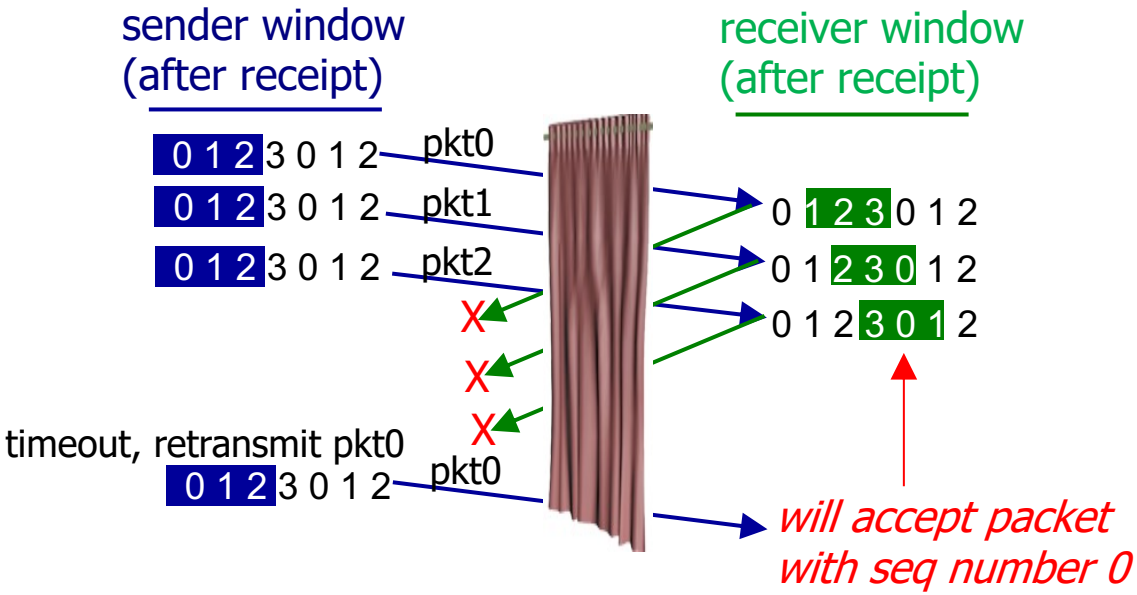



No problem...

# 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

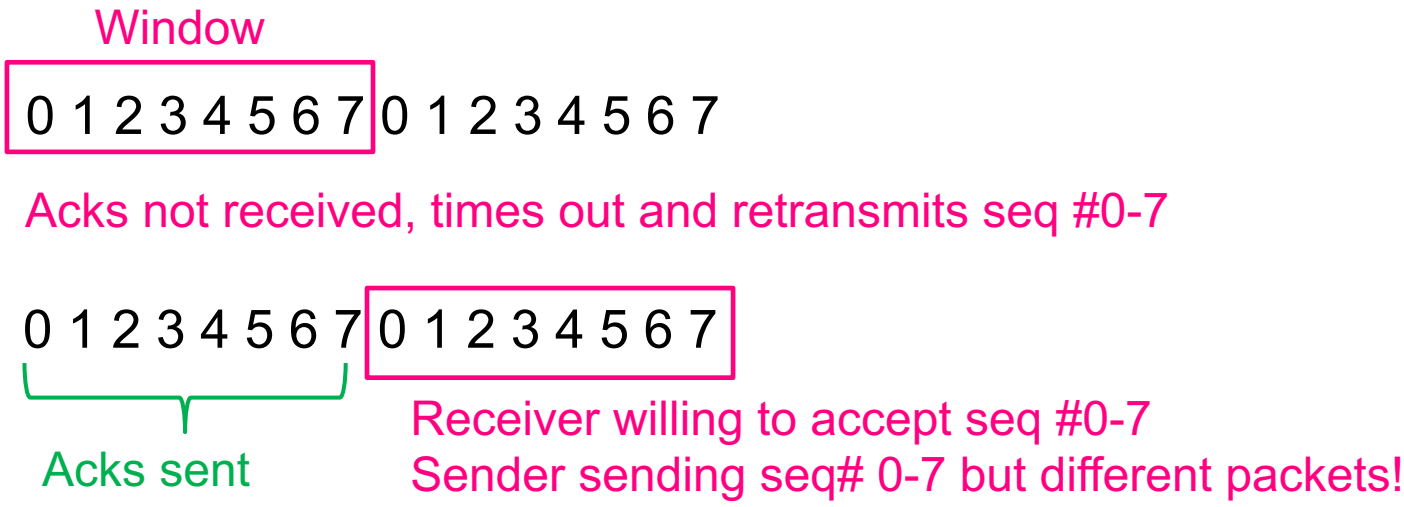
# Sequence #s in practice

## 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