Lesson 05-02:
Principles of Reliable Data Transfer

CS 356 Computer Networks

Mikyung Han

mhan@cs.utexas.edu

Internet Reference Model

Example Protocols		Responsible for
FTP, HTTP, SMTP	Application	application specific needs
TCP, UDP	Transport	process to process data transfer
IP	Network	host to host data transfer across different network
Ethernet, WiFi	Link	data transfer between physically adjacent nodes
802.3 PHY	Physical	bit-by-bit or symbol-by-symbol delivery

2

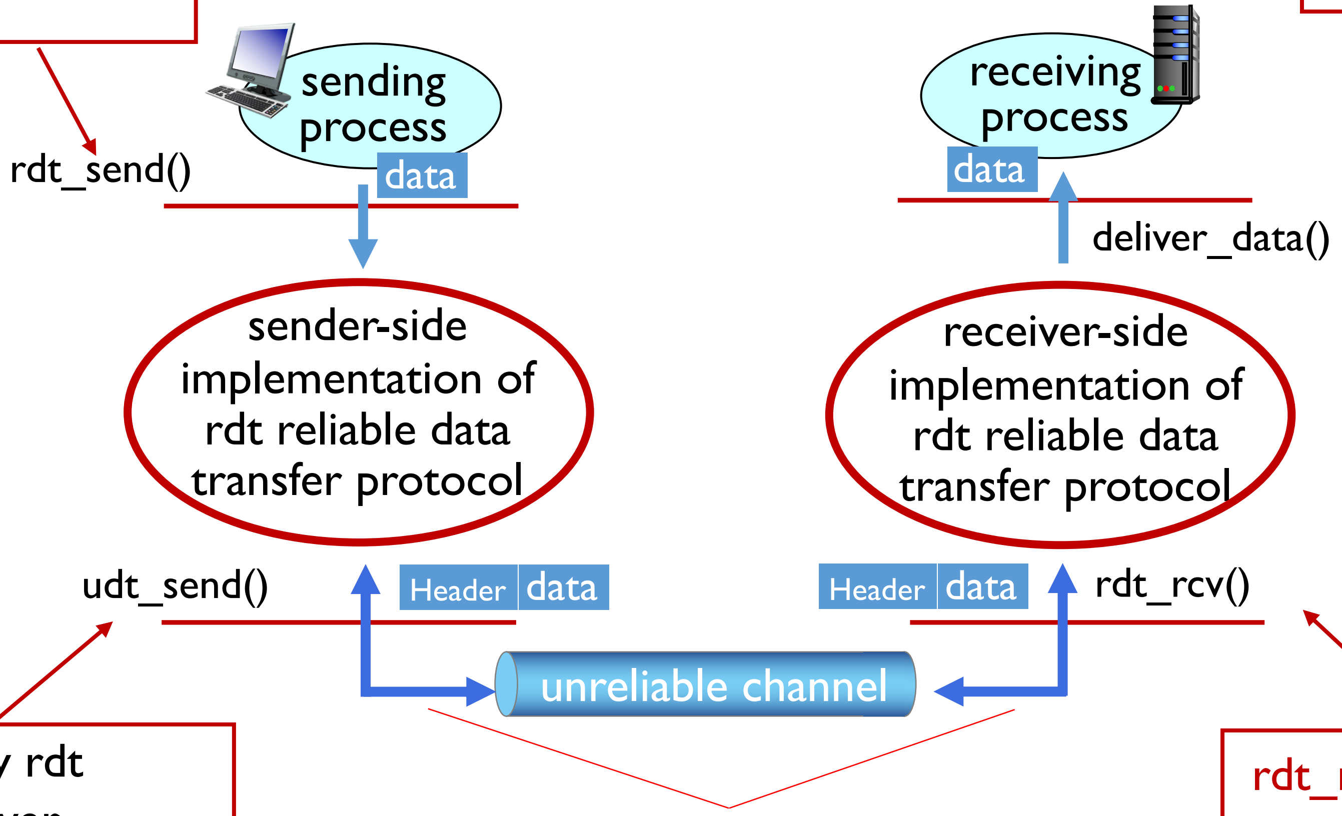
Outline

1. Channel with bit errors: rdt 2.0

Reliable data transfer protocol (rdt): interfaces

rdt_send(): called from app layer. Passes data to deliver to receiver upper layer

deliver_data(): called by rdt to deliver data to upper layer



Bi-directional communication over unreliable channel

udt_send(): called by rdt to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on receiver side of channel

rdt2.0: channel with bit errors

- How to detect bit errors?

- How to recover from errors?
  - ACKs: receiver explicitly tells sender that pkt received OK
  - NAKs: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK

stop and wait

sender sends one packet, then waits for receiver response

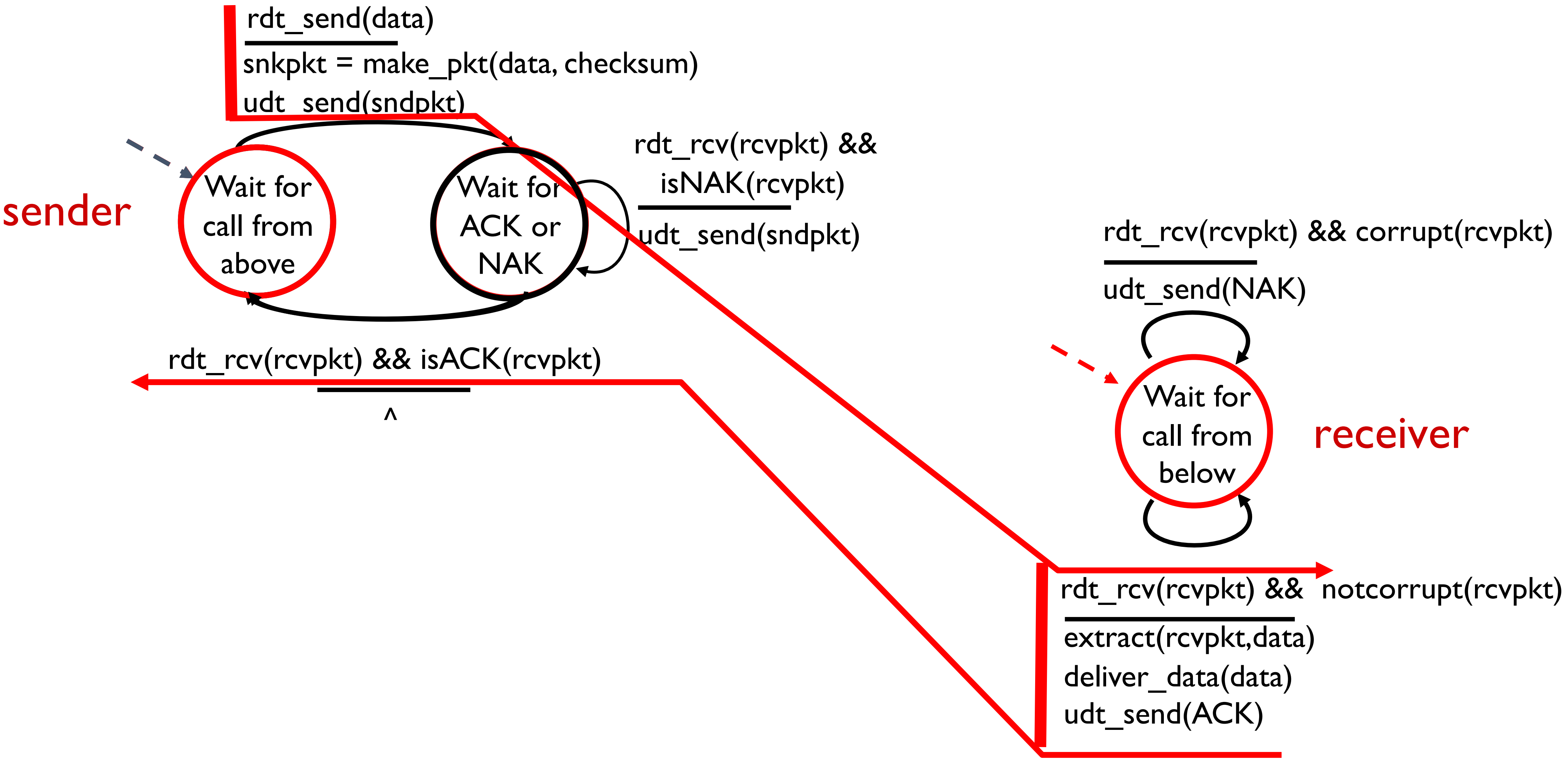
Recap: checksum can detect bit errors

example: add two 16-bit integers

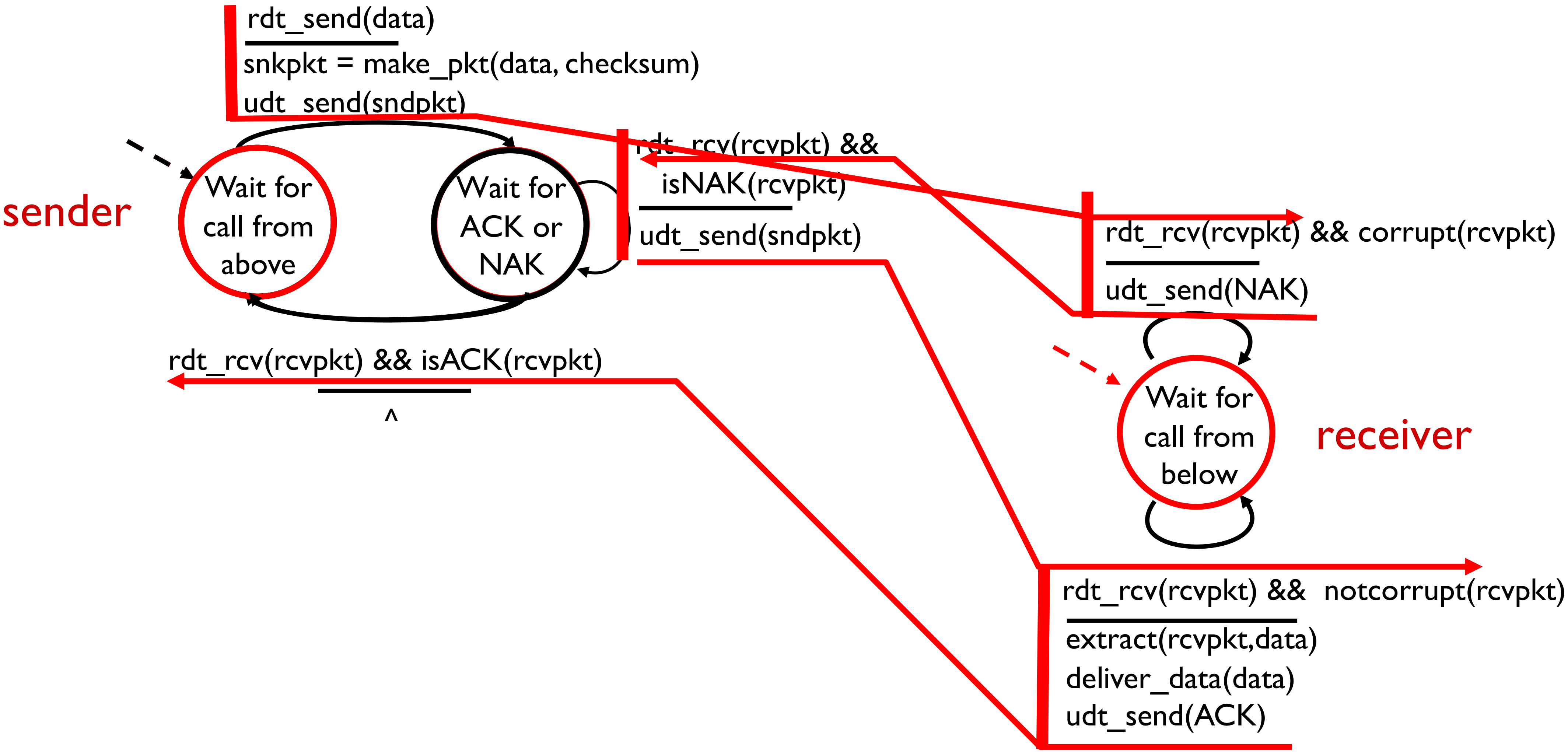
	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

When does checksum NOT work?

rdt2.0: operation with no errors



rdt2.0: operation with errors



What is the fatal flaw of rdt 2.0?

What if ACK/NAKs get corrupted?

- (T/F) Sender can find out if the packet received was corrupted
- (T/F) Sender knows if the corrupted packet was an ACK or NACK
- (T/F) Sender should always retransmit when receiving corrupted pkt
- What happens when sender retransmit for a corrupted ACK?
- What can we do?

What if ACK/NAKs get corrupted?

- Sender can find out if the packet received was corrupted
- Sender doesn't know if the corrupted packet was an ACK or NACK
- Sender should always retransmit when receiving corrupted pkt
- Duplicates happen when sender retransmit for a corrupted ACK

- Sender adds sequence number to each pkt
- Receiver discards (doesn't deliver up) duplicate pkt
  - a packet with previously seen sequence number

How many bits should be used for seq no?

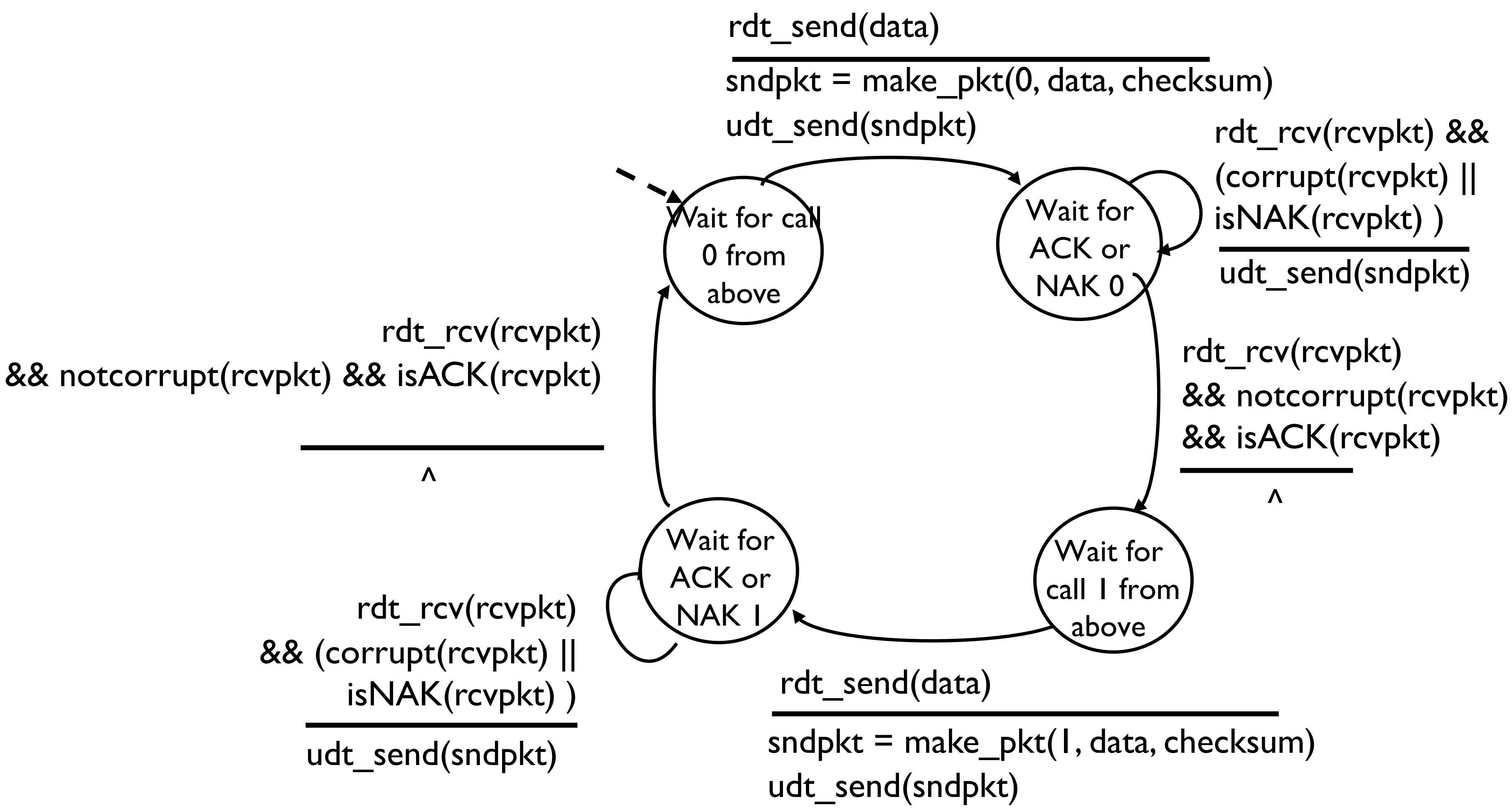
- We want to use a little space as possible
- How many packets do we want to distinguish?
- Note: link is never lossy but only bit error happens

We only need to distinguish the new packet
from previously already seen packet

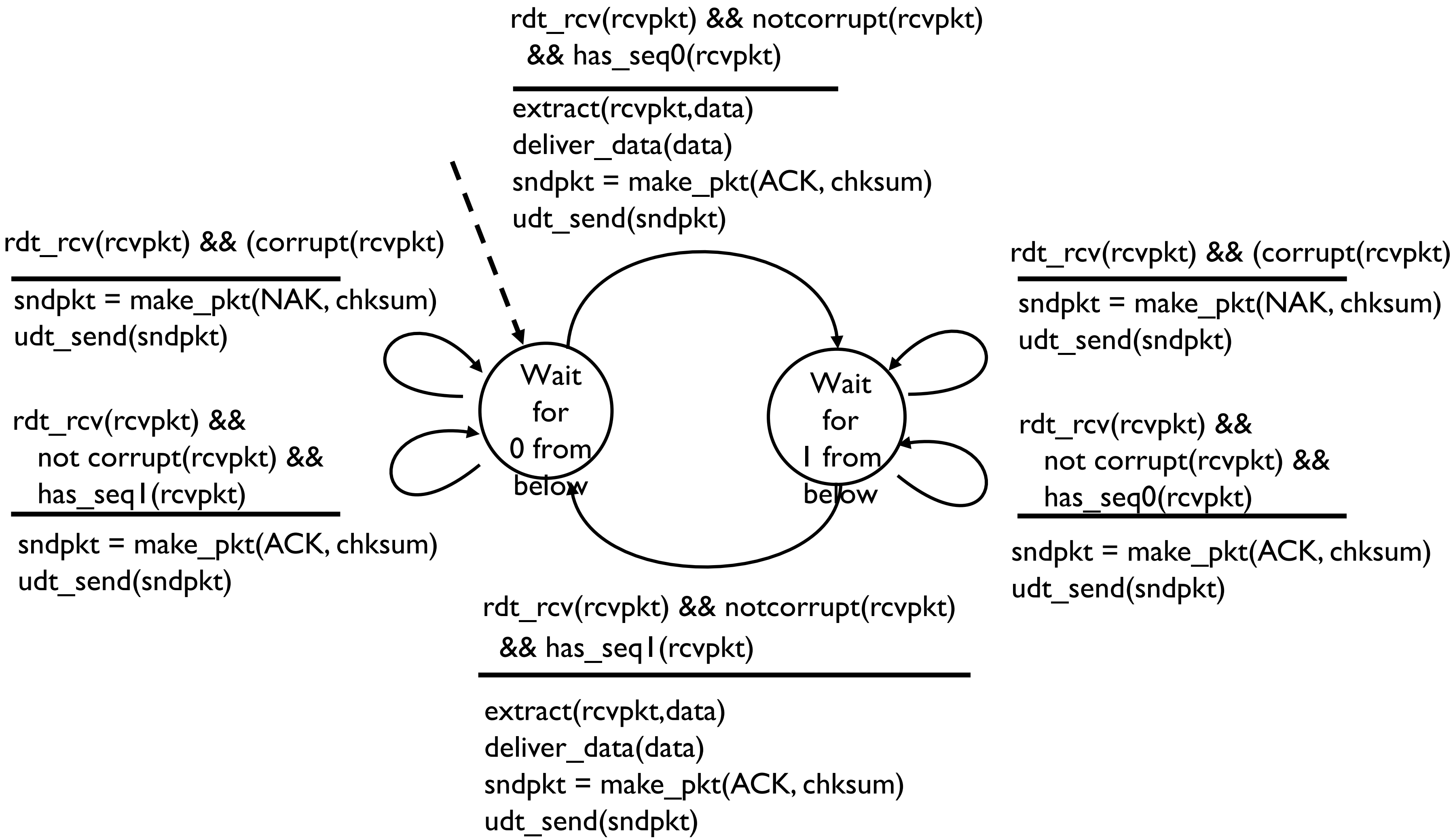
Outline

1. rdt 2.0
2. rdt 2.1 and rdt 2.2

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



rdt2.1: discussion

<u>sender:</u>

- 1 bit seq # added to pkt: 0 or 1
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

<u>receiver:</u>

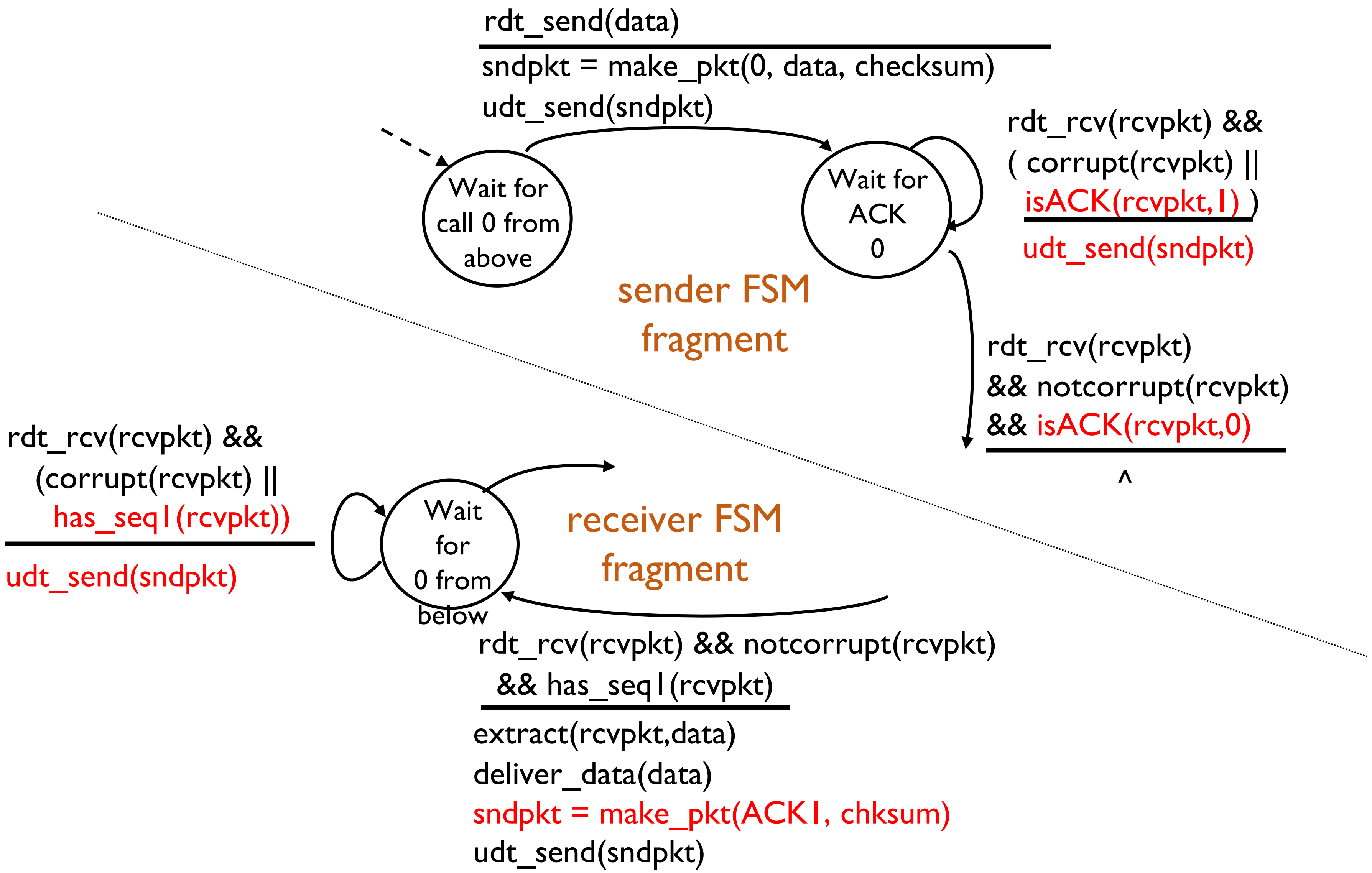
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- Can receiver know if its last ACK/NAK received OK at sender?

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- duplicate ACK has the same action as NAK
  - Sender retransmits

ACK for seq 0 and seq 1 needs to be distinguished

rdt2.2: sender, receiver fragments



Outline

1. rdt 2.0
2. rdt 2.1 and rdt 2.2
3. Channels with errors and losses: rdt 3.0

rdt3.0: channels with errors and loss

Loss can happen for both DATA and ACKs

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

If receiver never gets DATA what happens?

If receiver got DATA but ACK is lost what happens?

Channel loss introduces the need for timeout

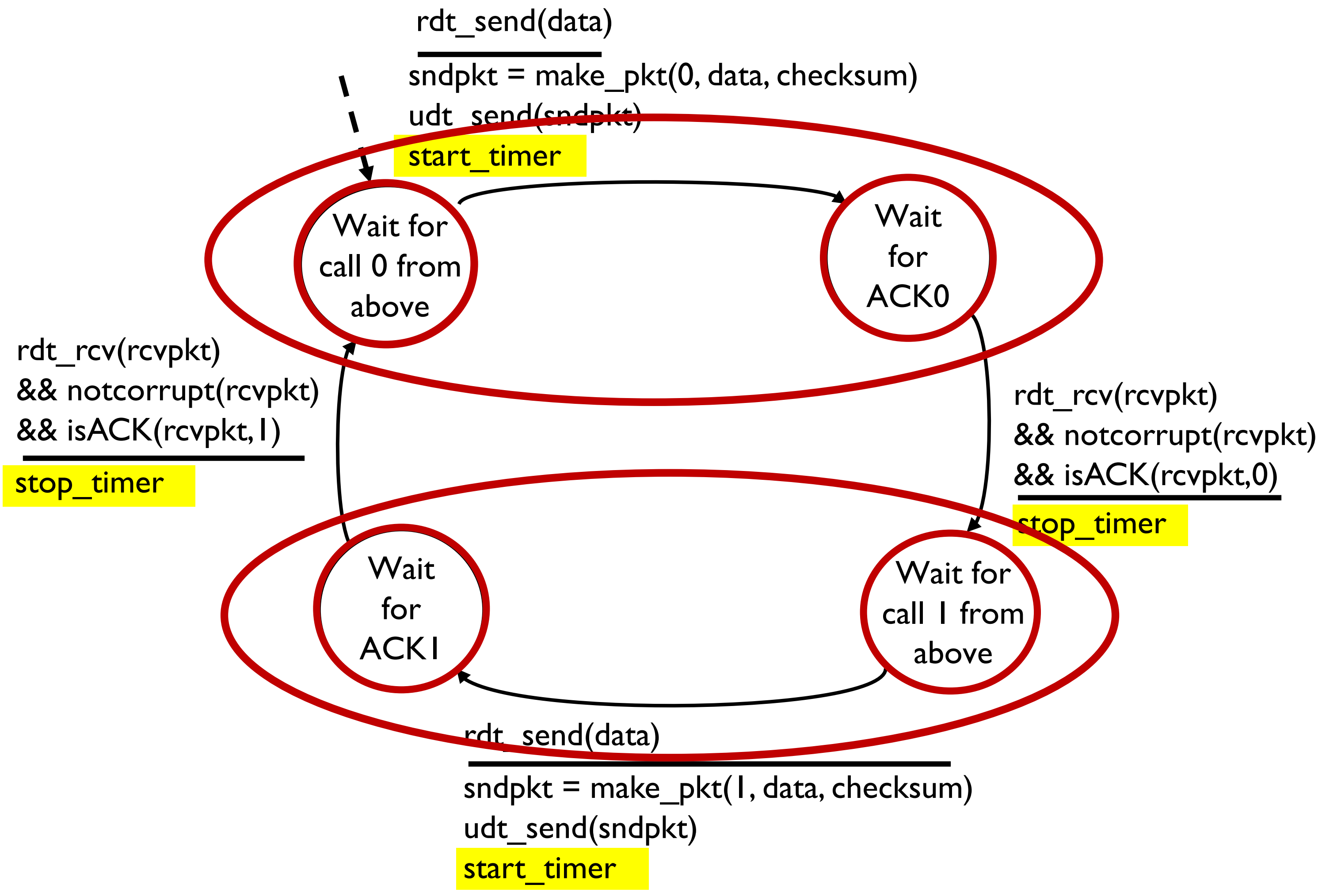
Approach: sender waits "reasonable" amount of time for ACK

- 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 packet being ACKed

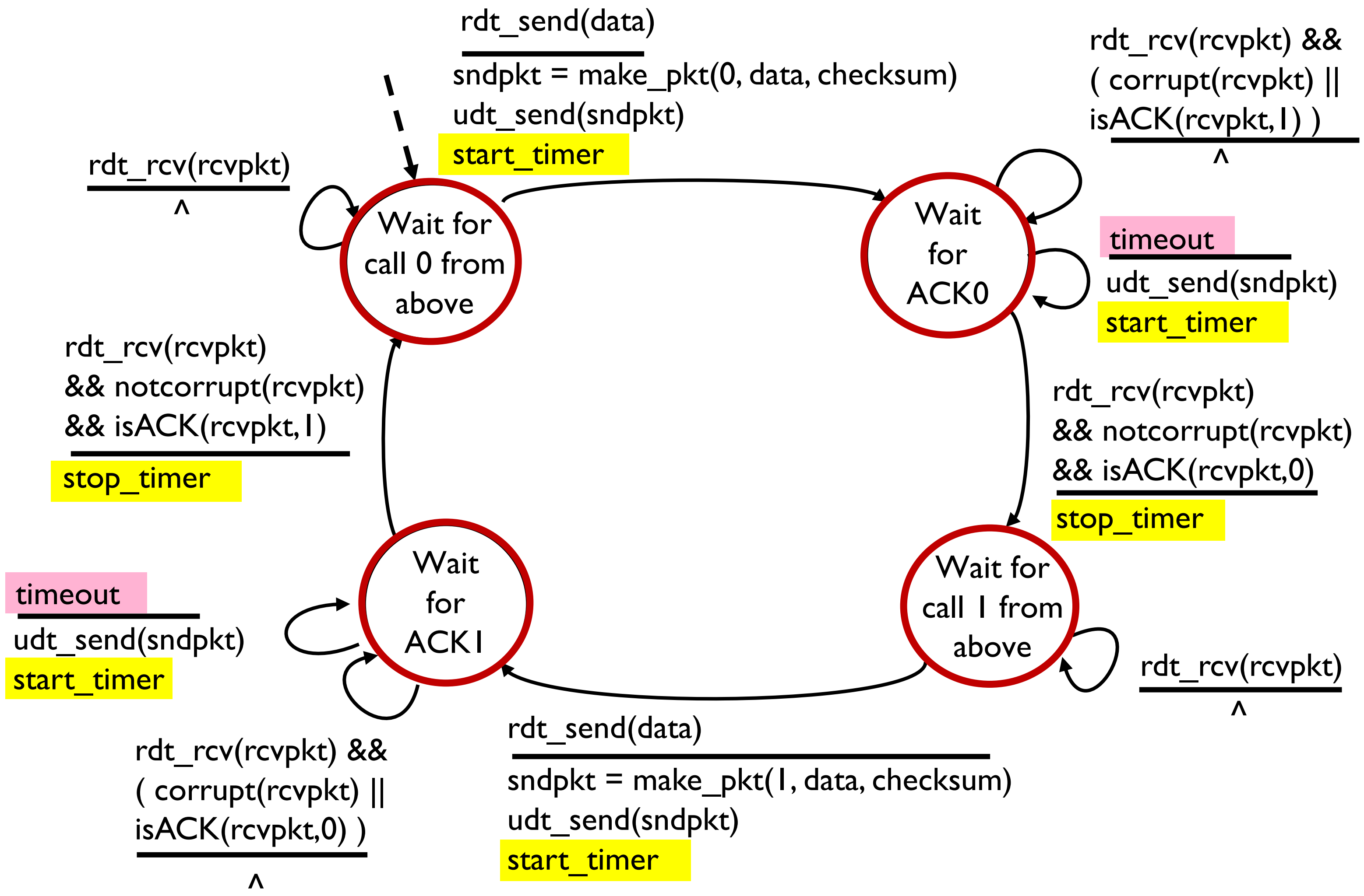
timeout

What is the "reasonable" time?

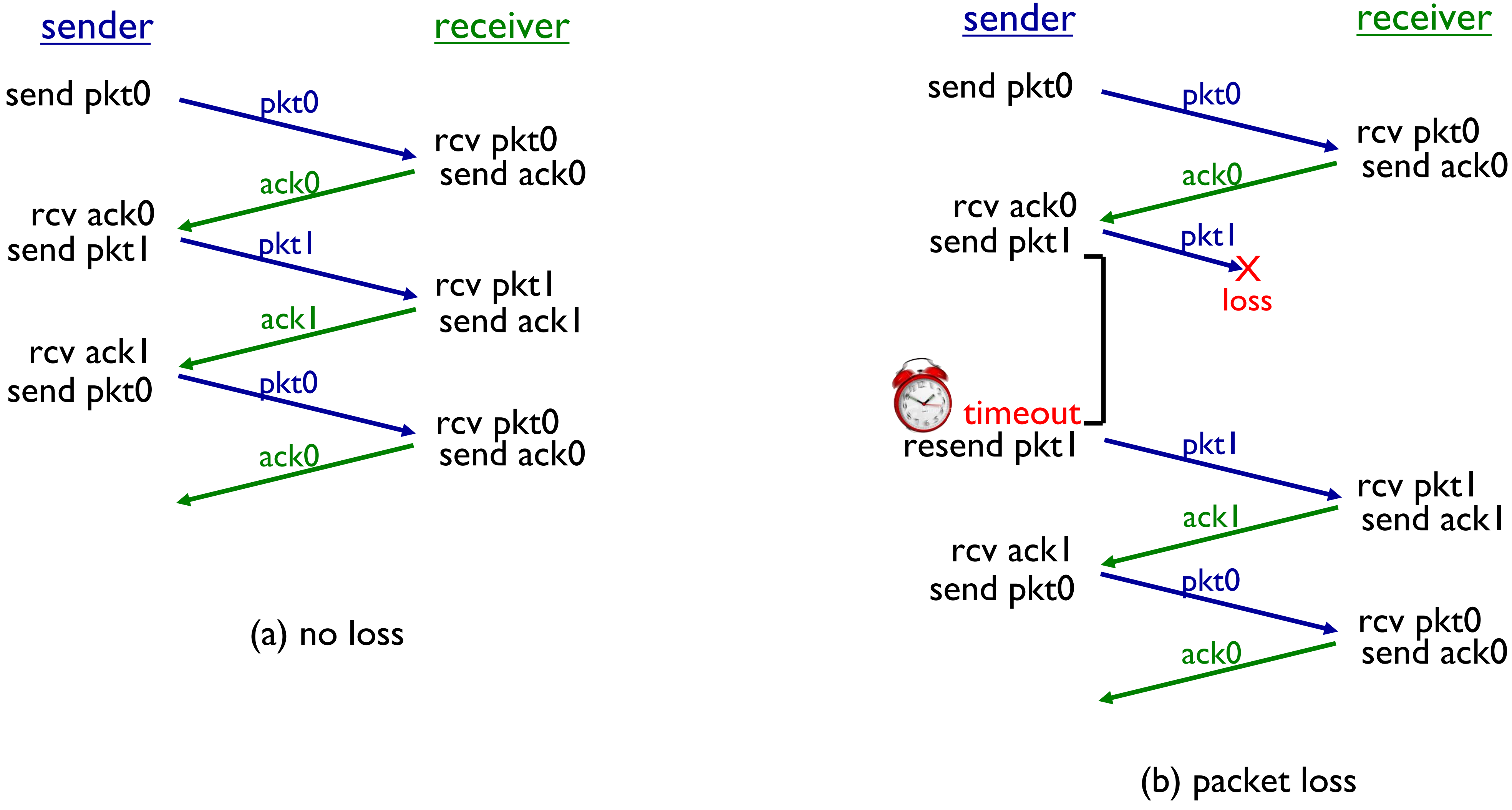
rdt3.0 sender



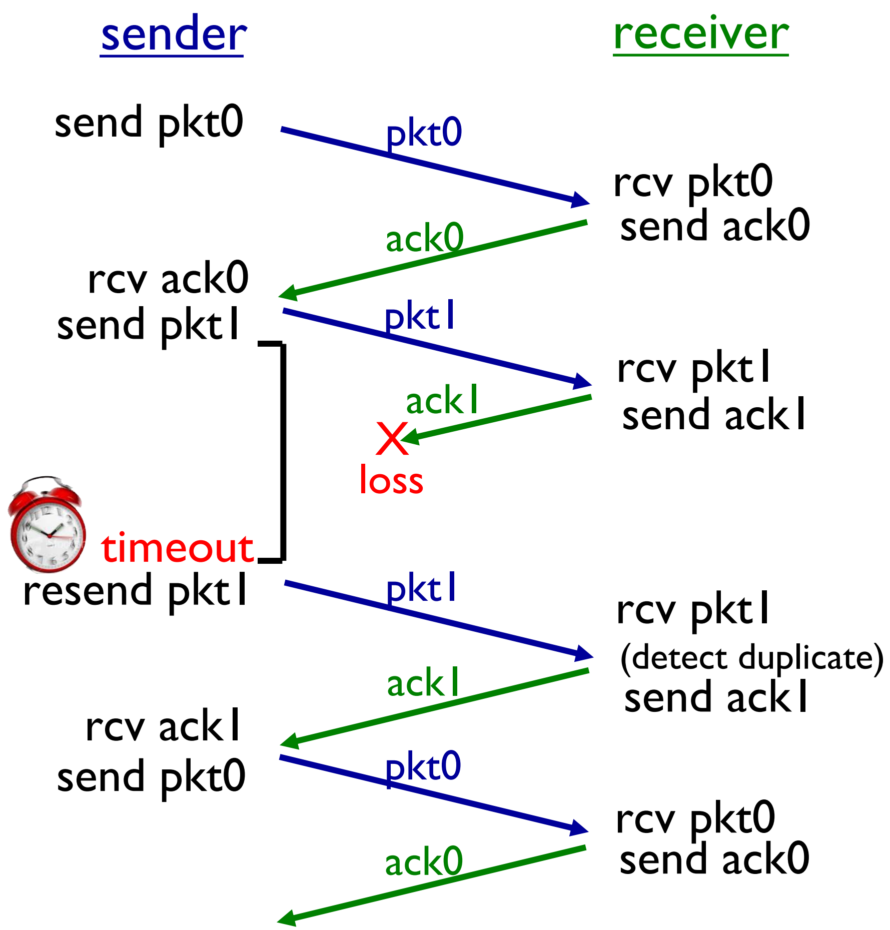
rdt3.0 sender



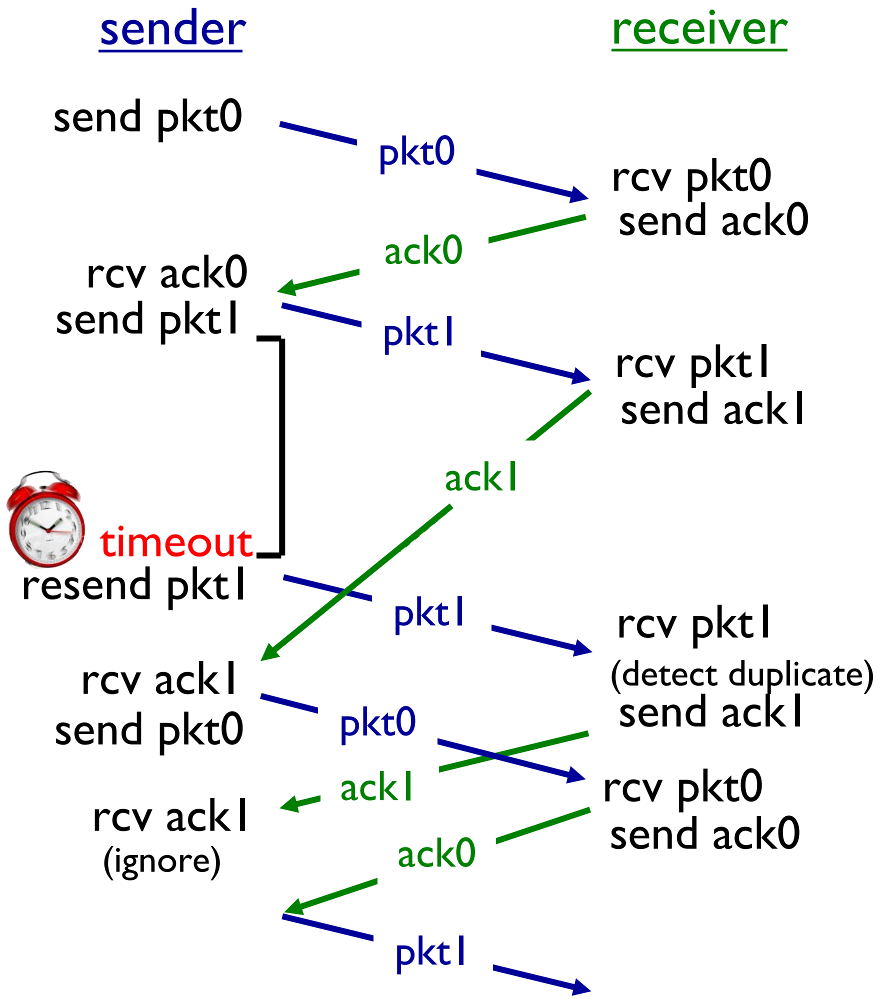
rdt3.0 in action



rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Suppose RTT between sender and receiver is constant and known to sender

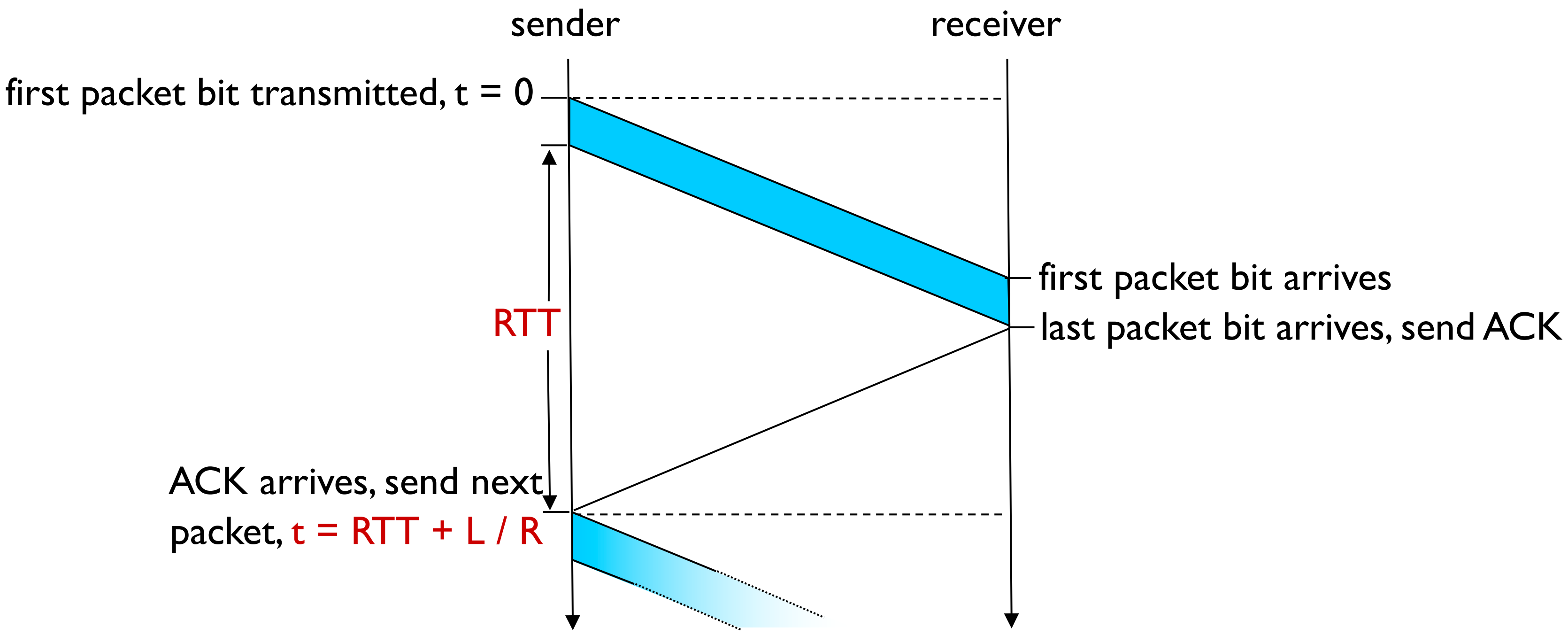
True or false?

- Sender knows whether the packet is truly received by the receiver
- Sender knows whether ACK is lost
- Sender still needs a timer

What should be the timeout value in this case?

rdt 3.0 is functionally ok; What about performance?

stop-and-wait only allows 1 unACKed packet



Performance of stop-and wait

- U_{sender}: utilization – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

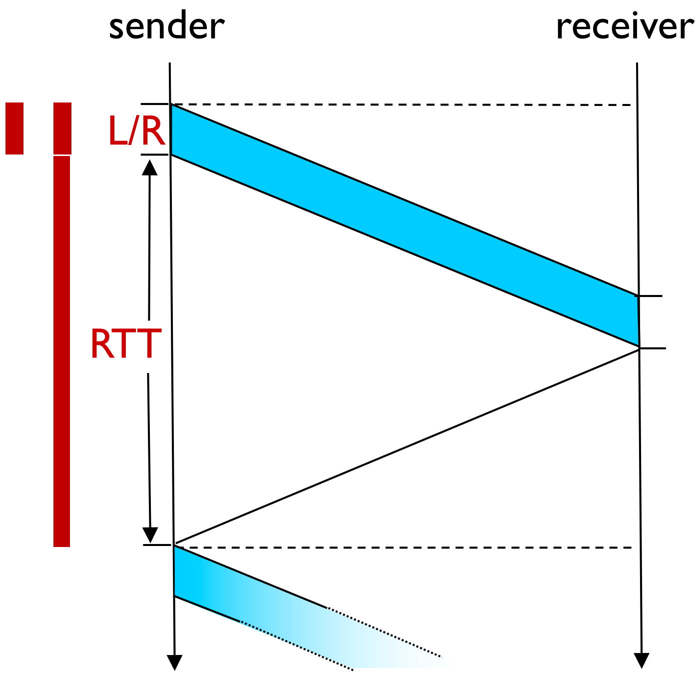
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

stop-and-wait suffers from very low link utilization

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

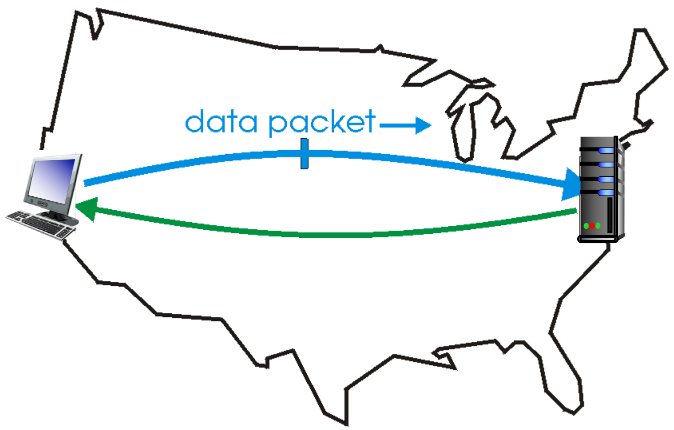


What is the root cause of this low link utilization?

Pipelining allows to send multiple "in-flight" packets

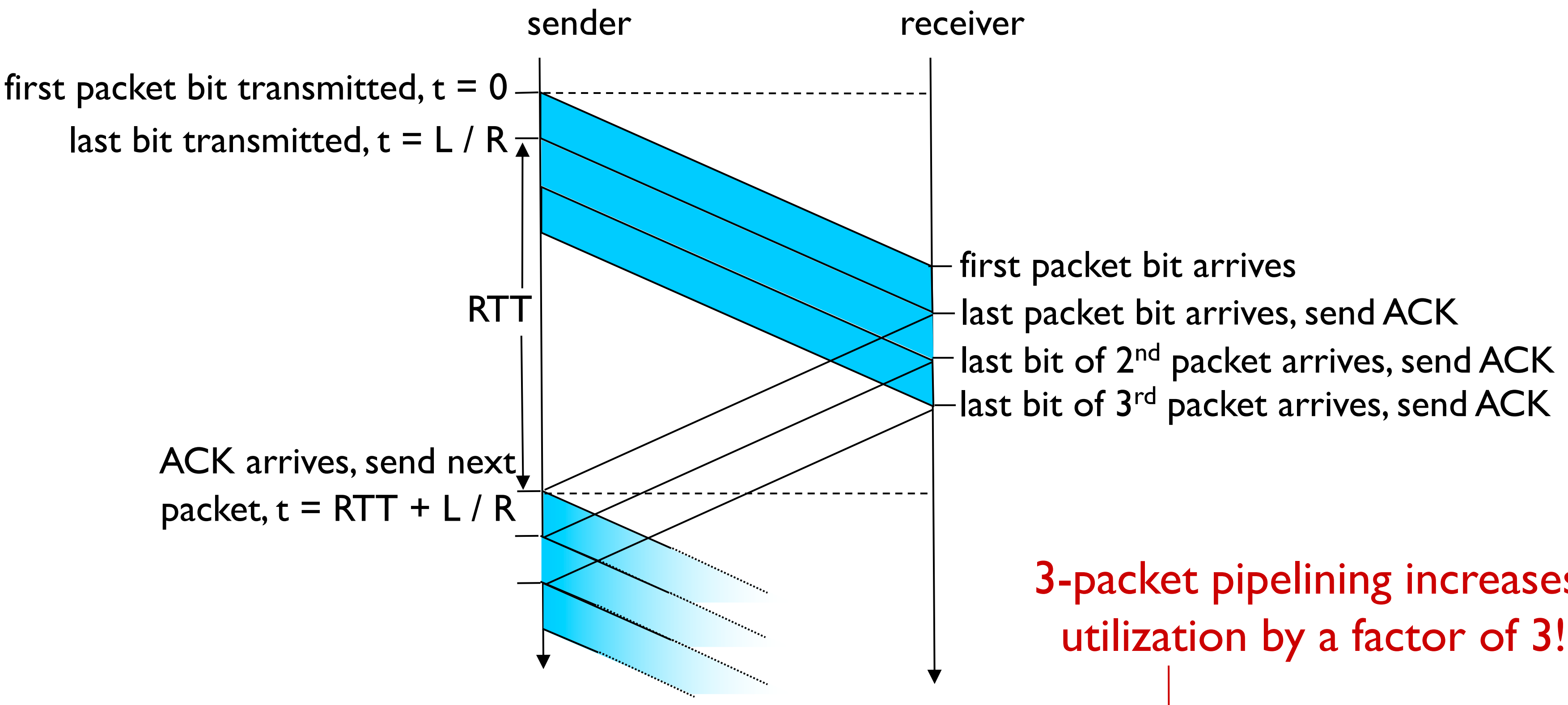
In-flight packets: yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation

Pipelining: increased utilization



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

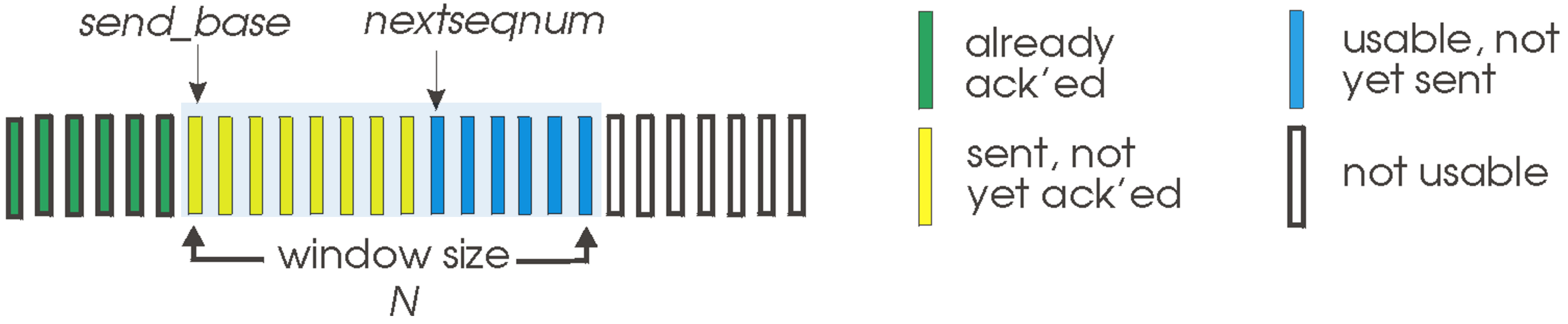
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Outline

1. rdt 2.0
2. rdt 2.1 and rdt 2.2
3. rdt 3.0
4. Go-Back-N

Go-Back-N sends up to N consecutive “in-flight” pkts

- k-bit seq # in pkt header

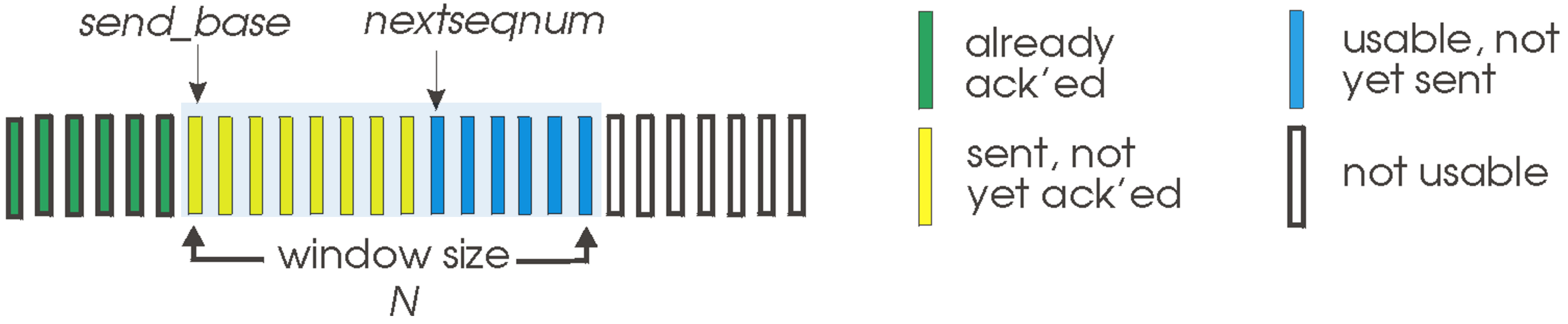


True or false?

- (T/F) cumulative ACK(n): ACKs all packets up to, excluding seq # n
- (T/F) on receiving ACK(n): reset send_base to n+1
- (T/F) timer for newest in-flight packet
- (T/F) timeout(n): retransmit just packet n

Go-Back-N sends up to N consecutive "in-flight" pkts

- k-bit seq # in pkt header



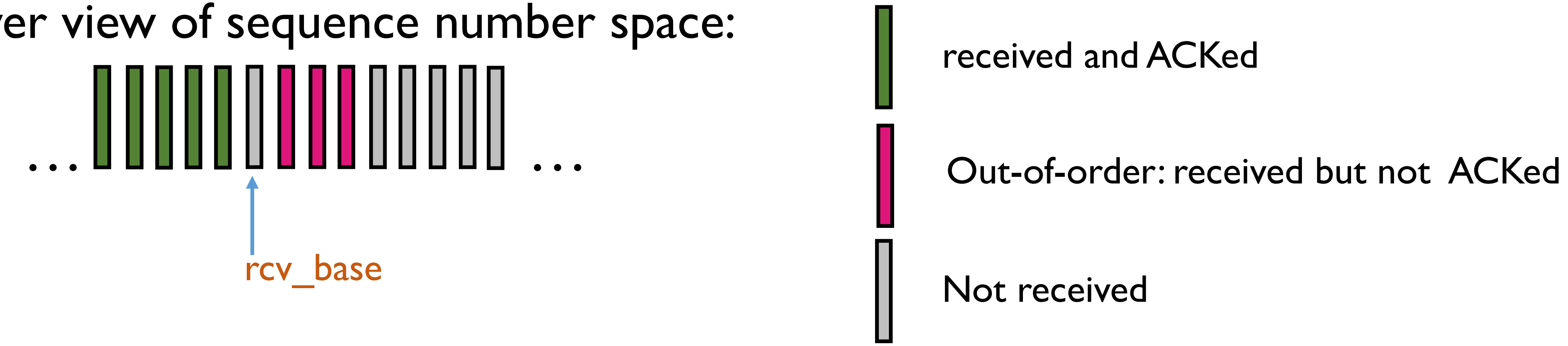
Answer key

- cumulative ACK(n): ACKs all packets up to, including seq # n
- on receiving ACK(n): reset send_base to n+1 (advances the window forward)
- timer for oldest in-flight packet
- timeout(n): retransmit packet n and all higher seq # pks in the window

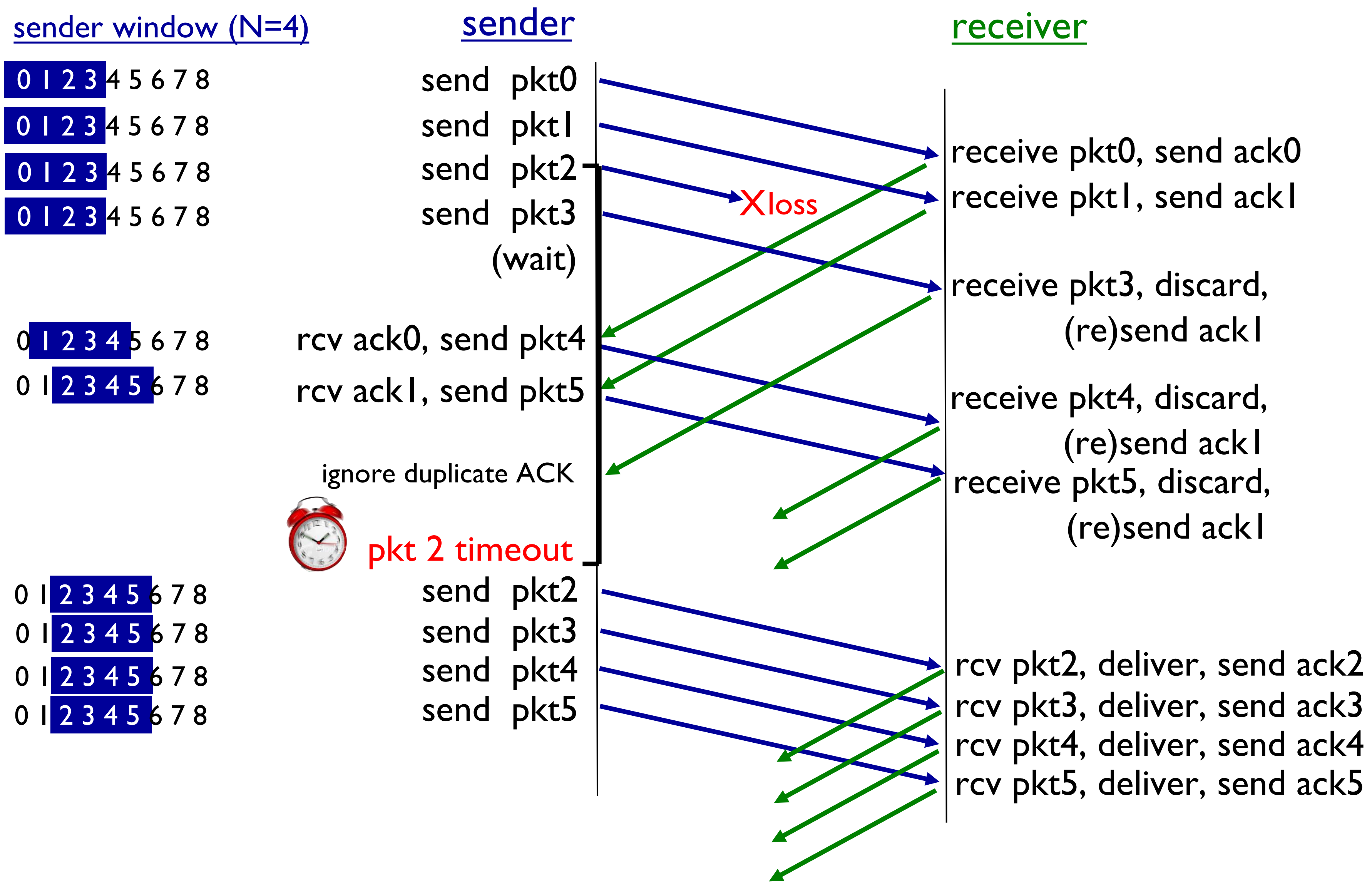
Go-Back-N receiver always send ACK(n) where n is highest in-order seq # received correctly

- May generate duplicate ACKs
- Need to only remember rcv_base
  - What is the relationship between n and rcv_base?
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



Go-Back-N in action



Outline

1. rdt 2.0
2. rdt 2.1 and rdt 2.2
3. rdt 3.0
4. Go-Back-N
5. Selective Repeat

In selective repeat receiver individually ACKs all correctly received pks

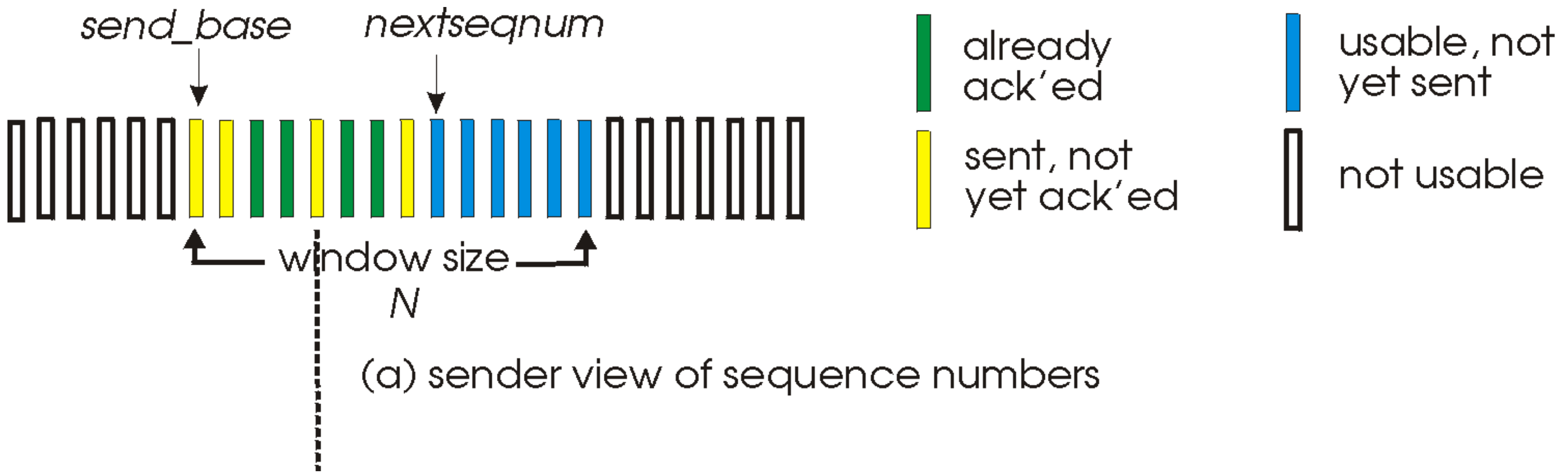
True/False

- Receiver does not need to buffer pkts
- Sender has a timeout for the oldest in-flight packet
- Upon timeout sender sends out just 1 packet
- Sender window consists of N consecutive seq #s
- Sender window limits the number of in-flight ptks

Selective repeat answer key

- Receiver should buffer packets for in-order delivery to app. layer
- Sender maintains timer for each in-flight pkt
  - Upon timeout sender retransmits that unACKed packet
- Sender window
  - N consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(n):

- resend packet n, restart timer

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

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

receiver

packet n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

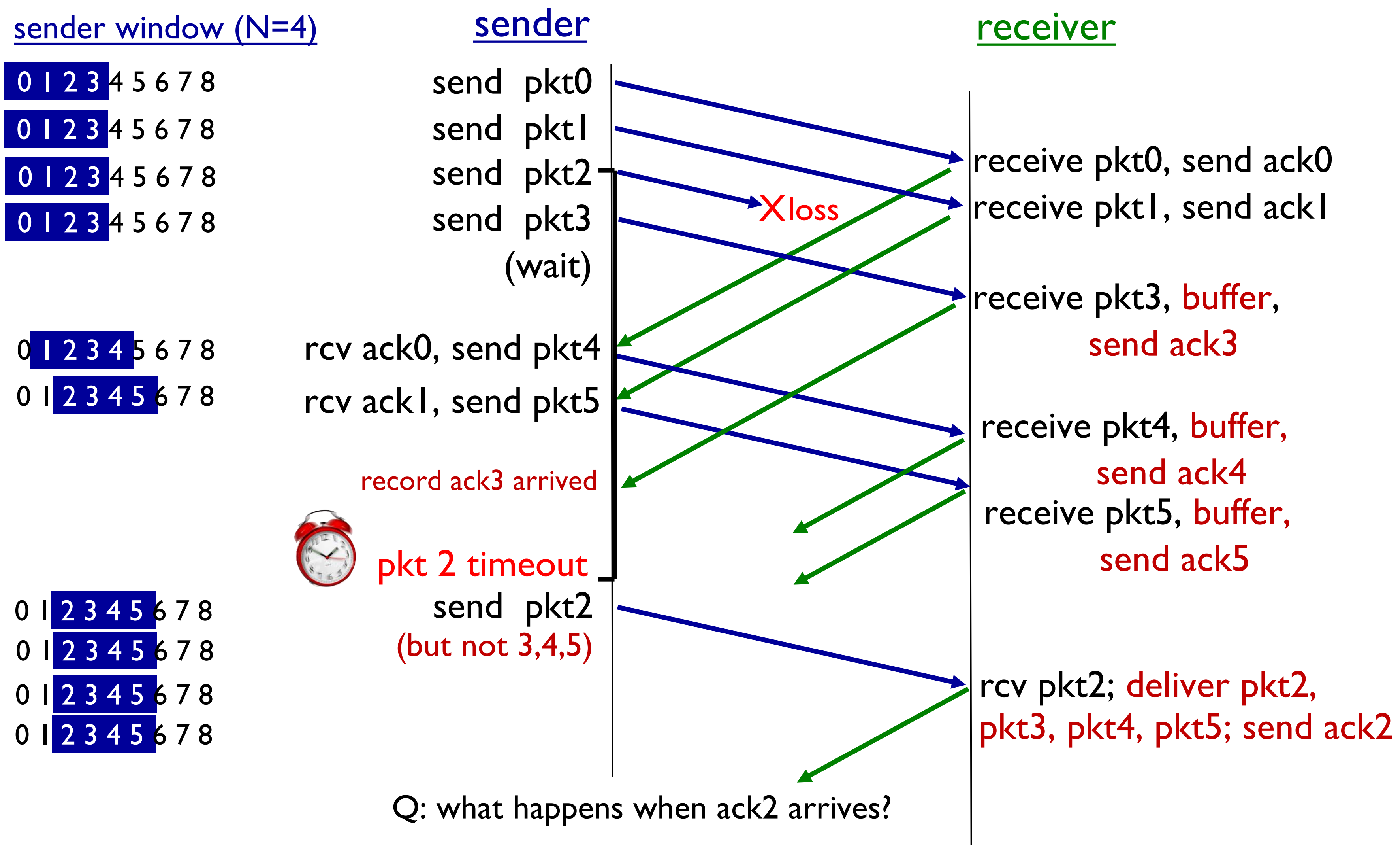
packet n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective Repeat in action

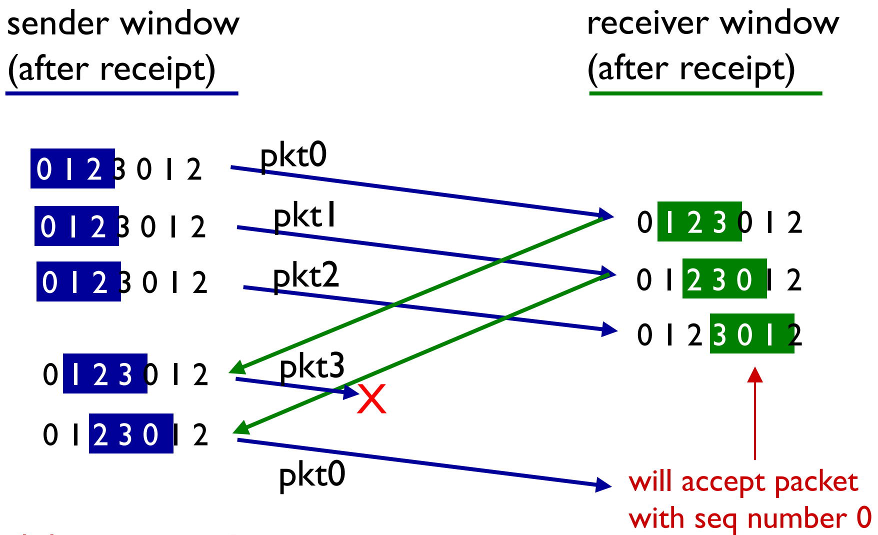


Q: what happens when ack2 arrives?

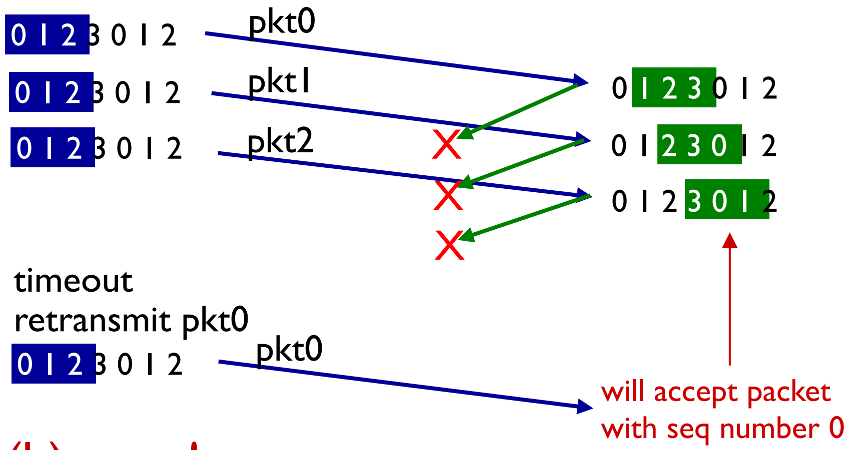
Selective repeat: a dilemma!

example:

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



(a) no problem



(b) oops!

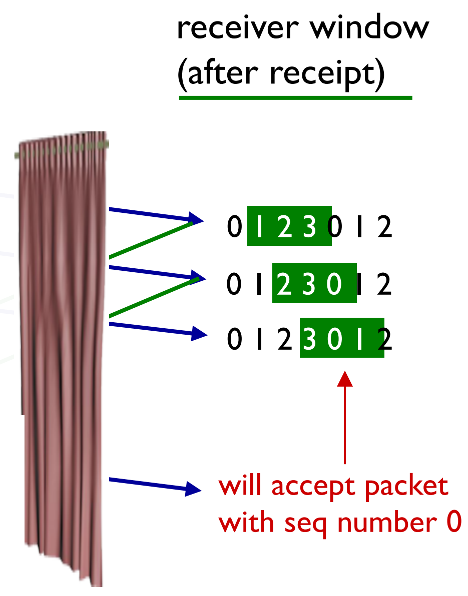
Selective repeat: a dilemma!

example:

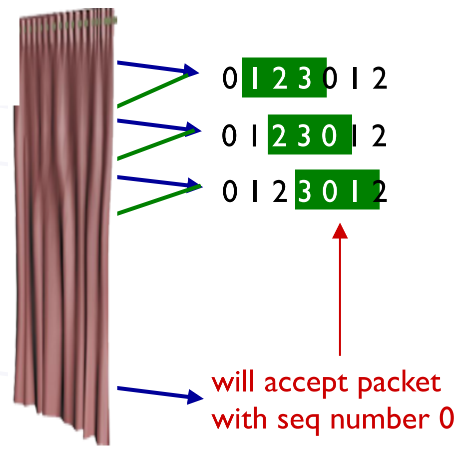
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

What should be the relationship btw seq # size and window size?

sender window (after receipt)



- receiver can't see sender side
- receiver behavior identical in both cases!
- something's (very) wrong!



Acknowledgements

Slides are adopted from Kurose' Computer Networking Slides