

Lesson 05-02: Principles of Reliable Data Transfer

CS 326E Elements of Networking

Mikyung Han

mhan@cs.utexas.edu

Example Protocols

FTP, HTTP, SMTP

TCP, UDP

IP

Ethernet, WiFi

802.3 PHY

Application

Transport

Network

Link

Physical

Responsible for

application specific needs

process to process data transfer

host to host data transfer across different network

data transfer between physically adjacent nodes

bit-by-bit or symbol-by-symbol delivery

Internet Reference Model



Outline

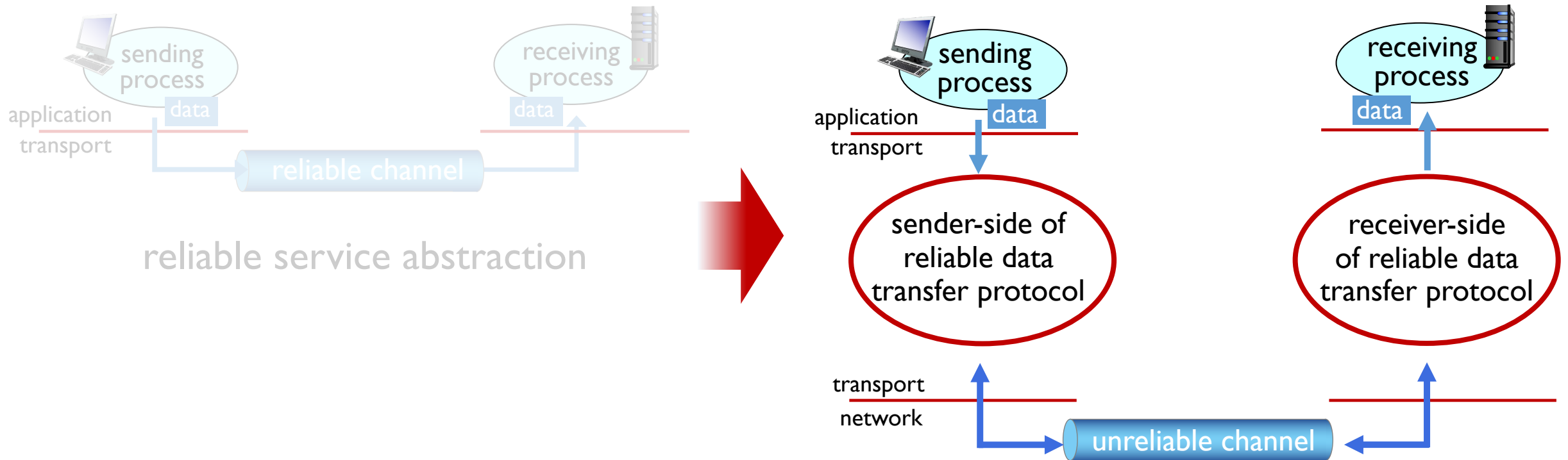
👉 I. Channel with bit errors: rdt 2.0

Principles of reliable data transfer (rdt)



reliable service **abstraction**

Reliable data transfer has both sender-side and receiver-side implementation



Communication is bi-directional!
The receiving end has to also send control info such as ack

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

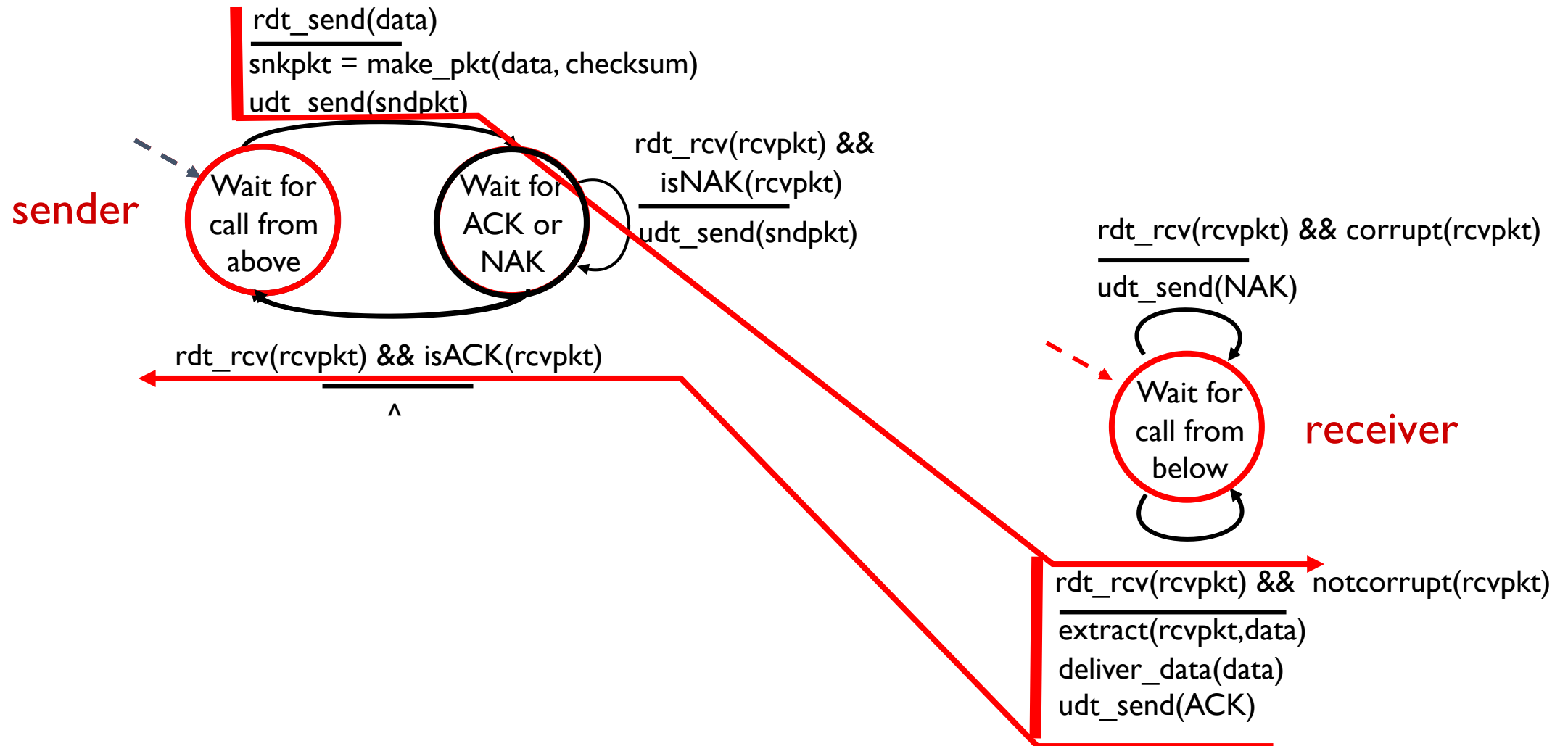
Recap: **checksum** can detect bit errors

example: add two 16-bit integers

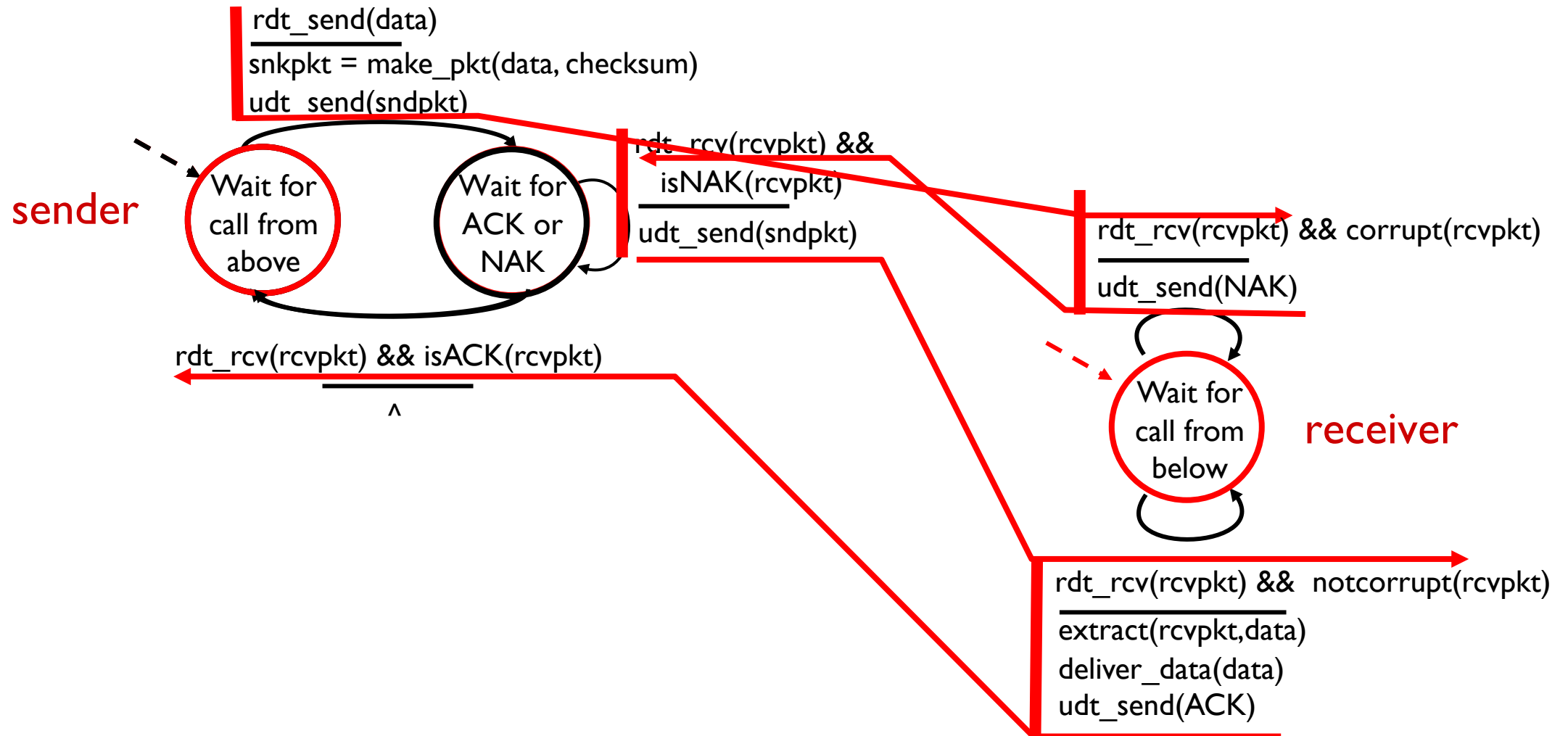
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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?

Kahoot ☺

True or False?

- (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?
- Possible solution?

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

Do we need to specify sequence number in ACK/NAKs?

- To specify which seq no it is acknowledging the receipt?
- aka ACK number

Why or why not?

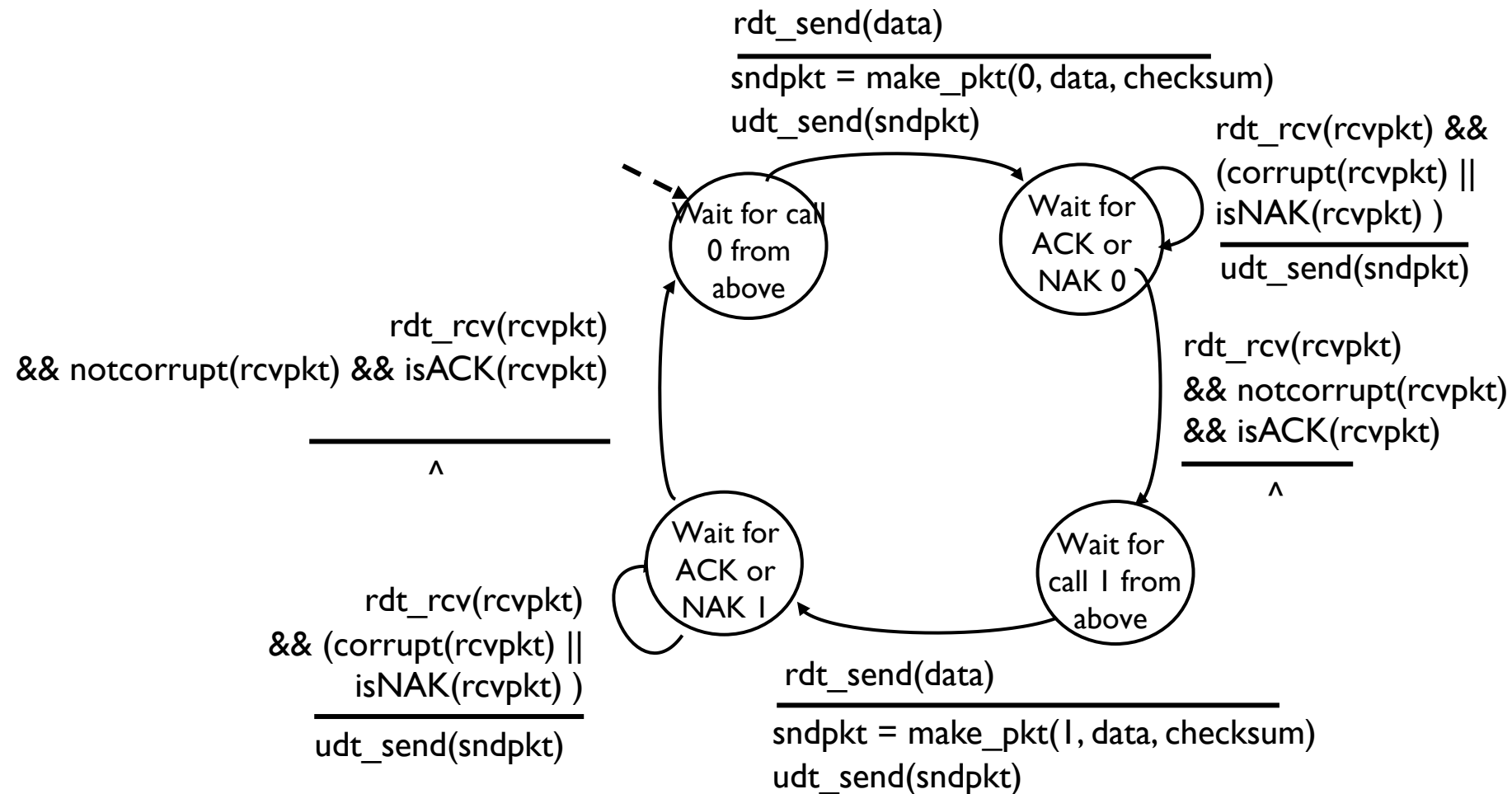
Example sequence

Outline

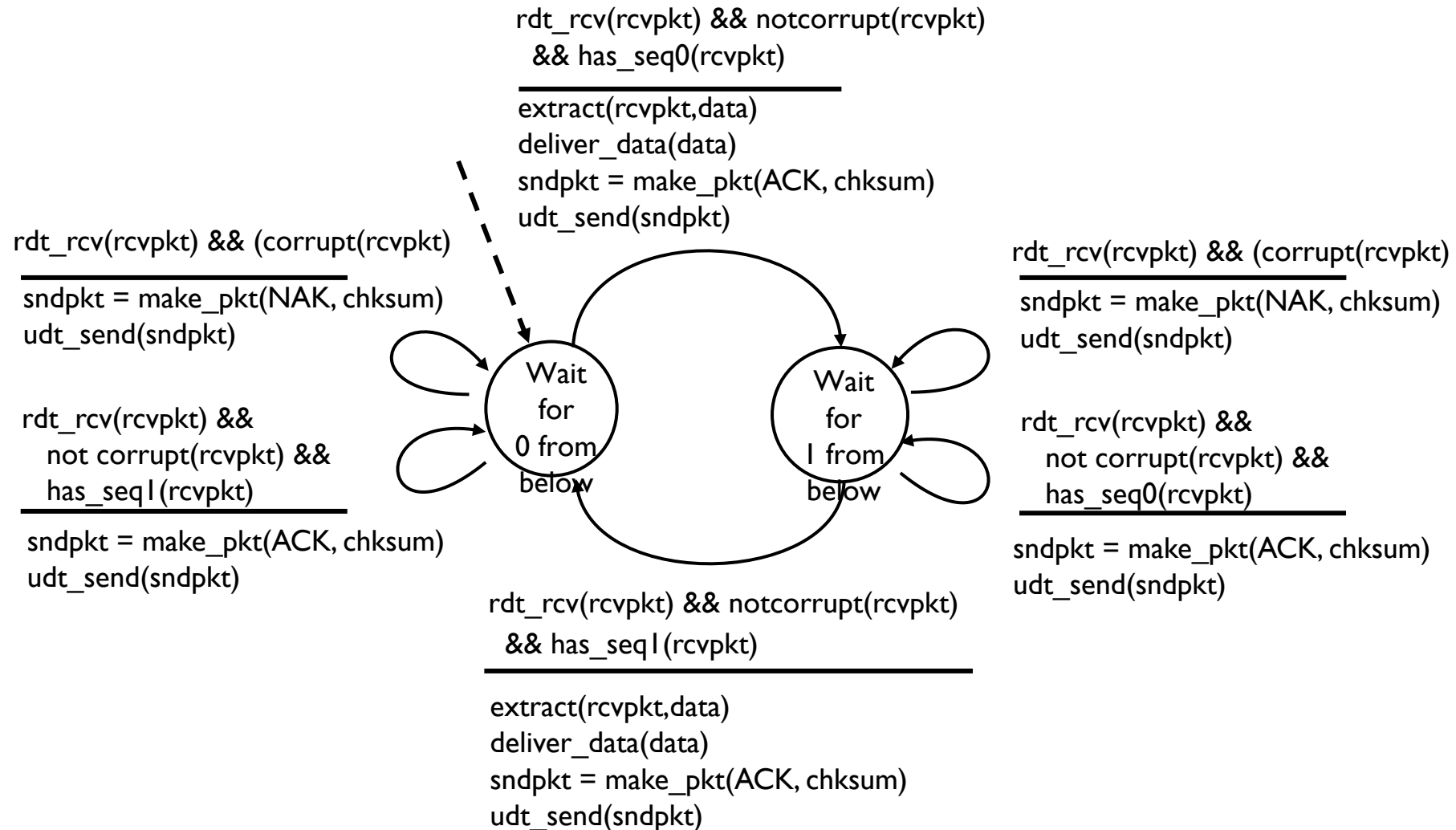
1. rdt 2.0

 2. rdt 2.1 and rdt 2.2

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- How to “simulate” NAK?

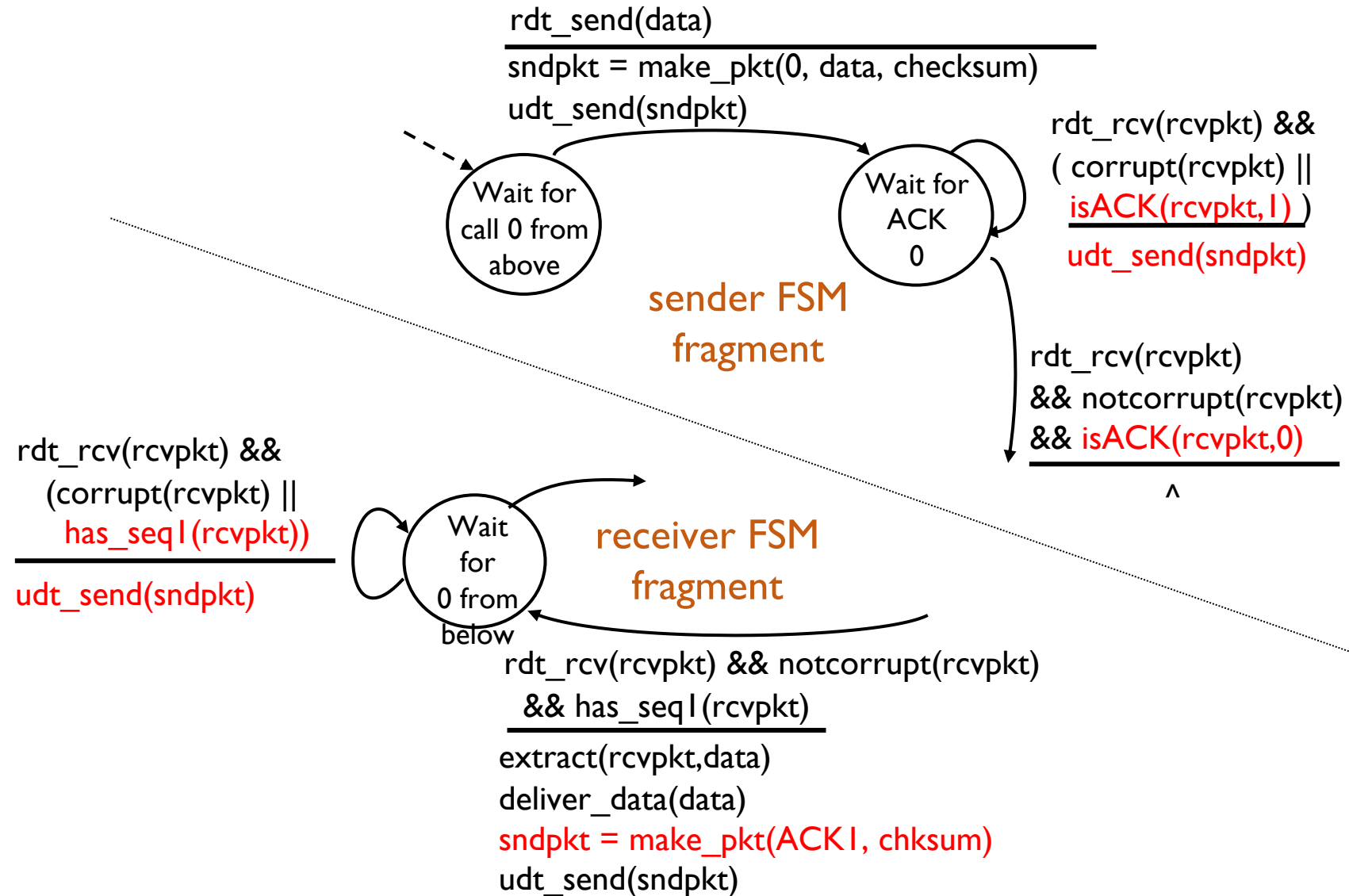
How to simulate NAK with just ACKs?

rdt2.2: a NAK-free protocol

- Do we need a sequence number for ACK?
 - Sender sends DATA 1
 - Receiver sends ACK without sequence number
 - In this case, does sender know if it is really ACKing DATA 1 or dupe ACKing DATA 0?

No. Receiver needs to explicitly specify:
ACK for seq 0 and seq 1 must be distinguished specified
by the receiver

rdt2.2: sender, receiver fragments



rdt2.2x: an **NAK-only** protocol

- How to “simulate” ACK with just NAKs?

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**

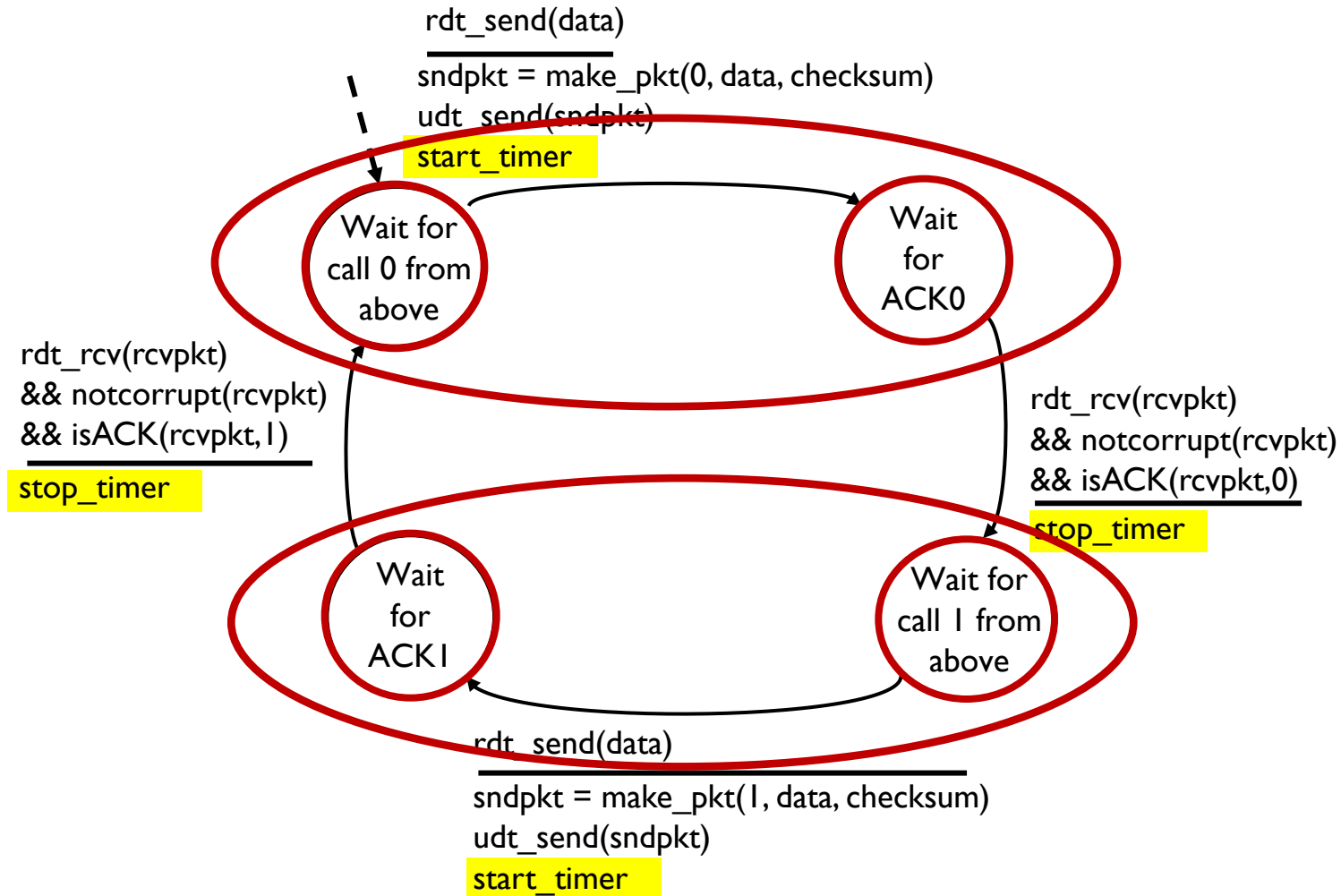
Approach: sender waits “reasonable” amount of time for ACK

- 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

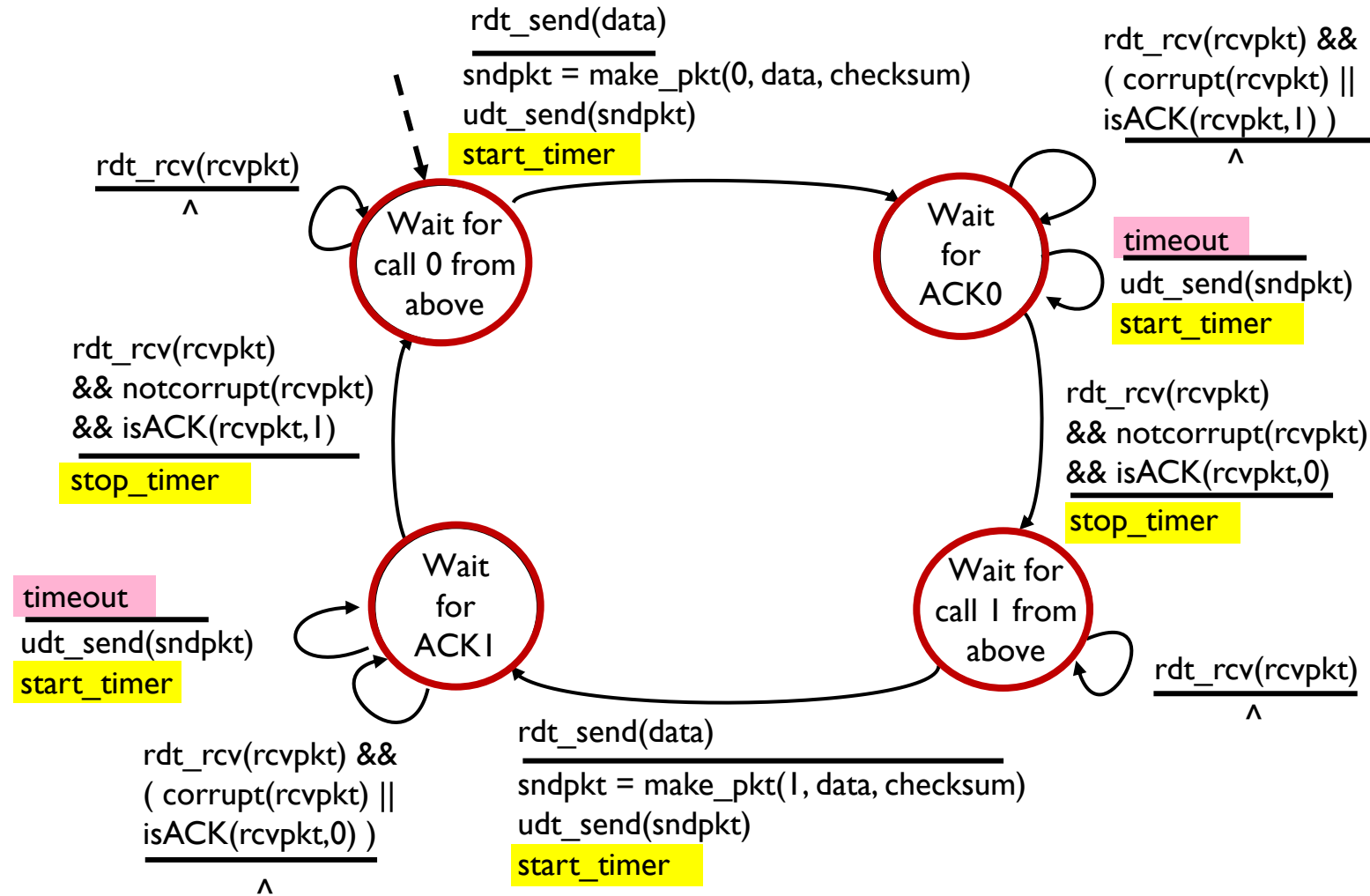


What is the “reasonable” time?

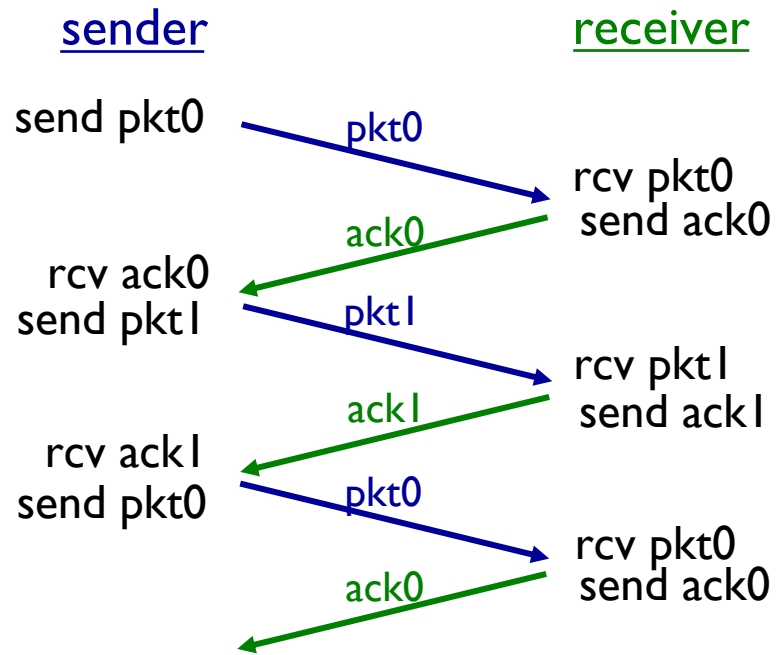
rdt3.0 sender



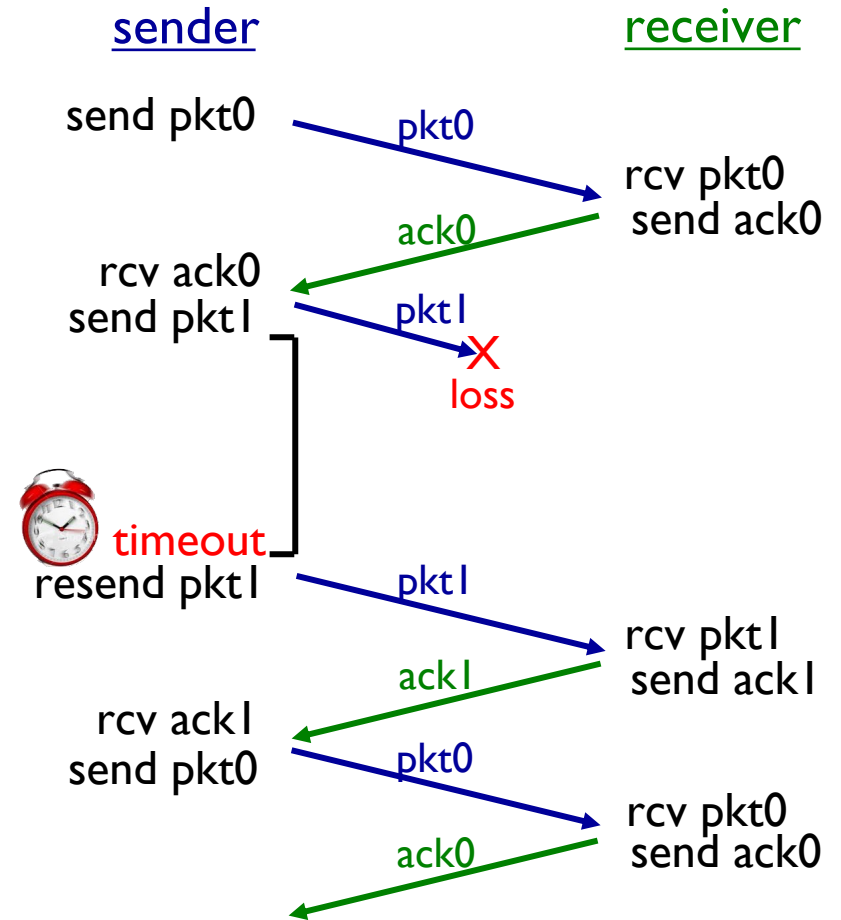
rdt3.0 sender



rdt3.0 in action

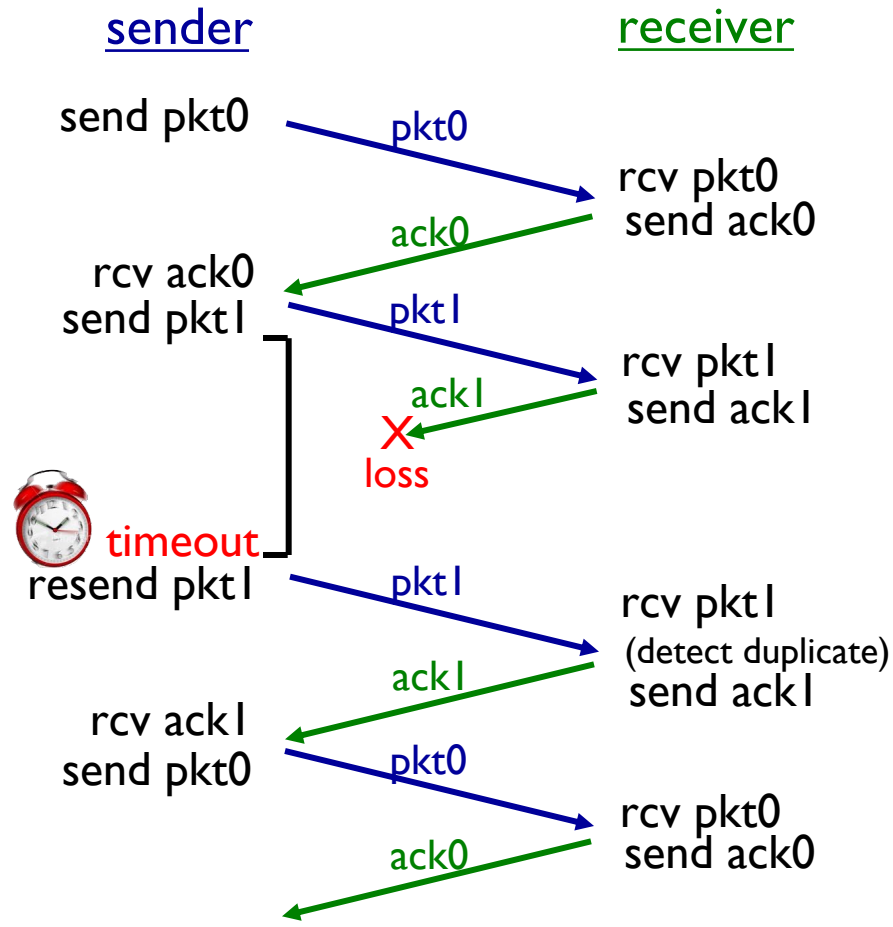


(a) no loss

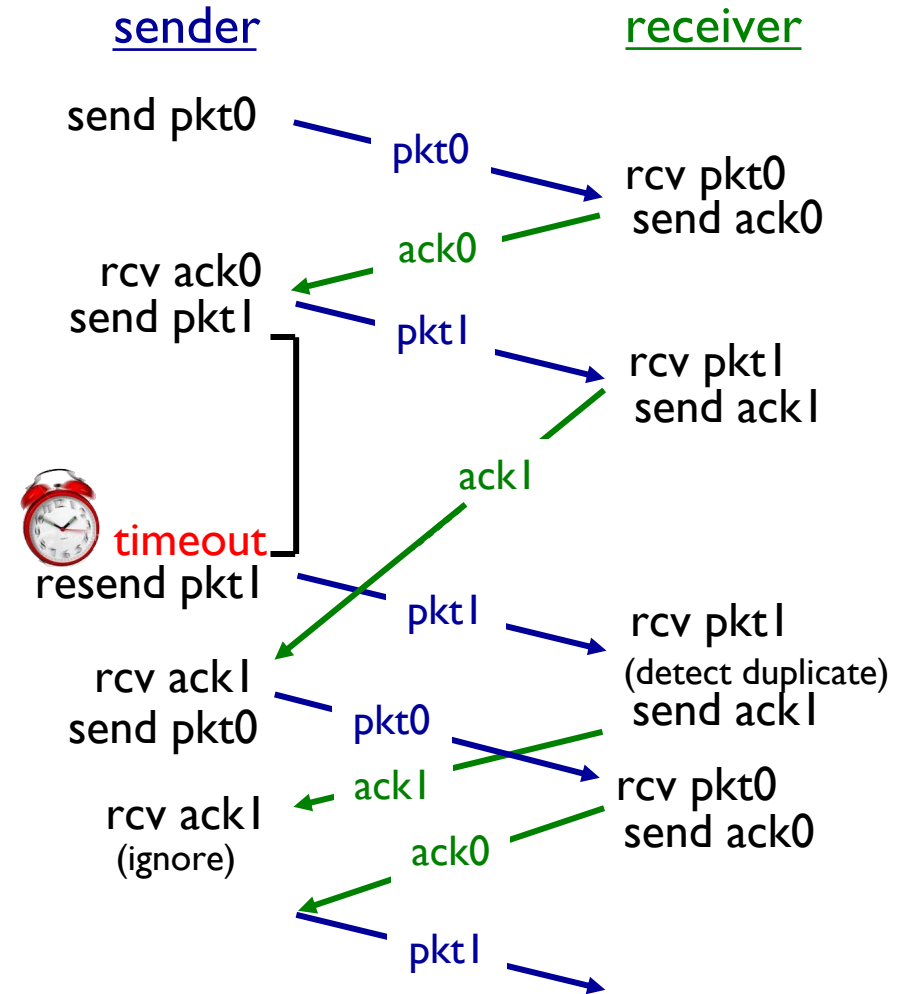


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

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

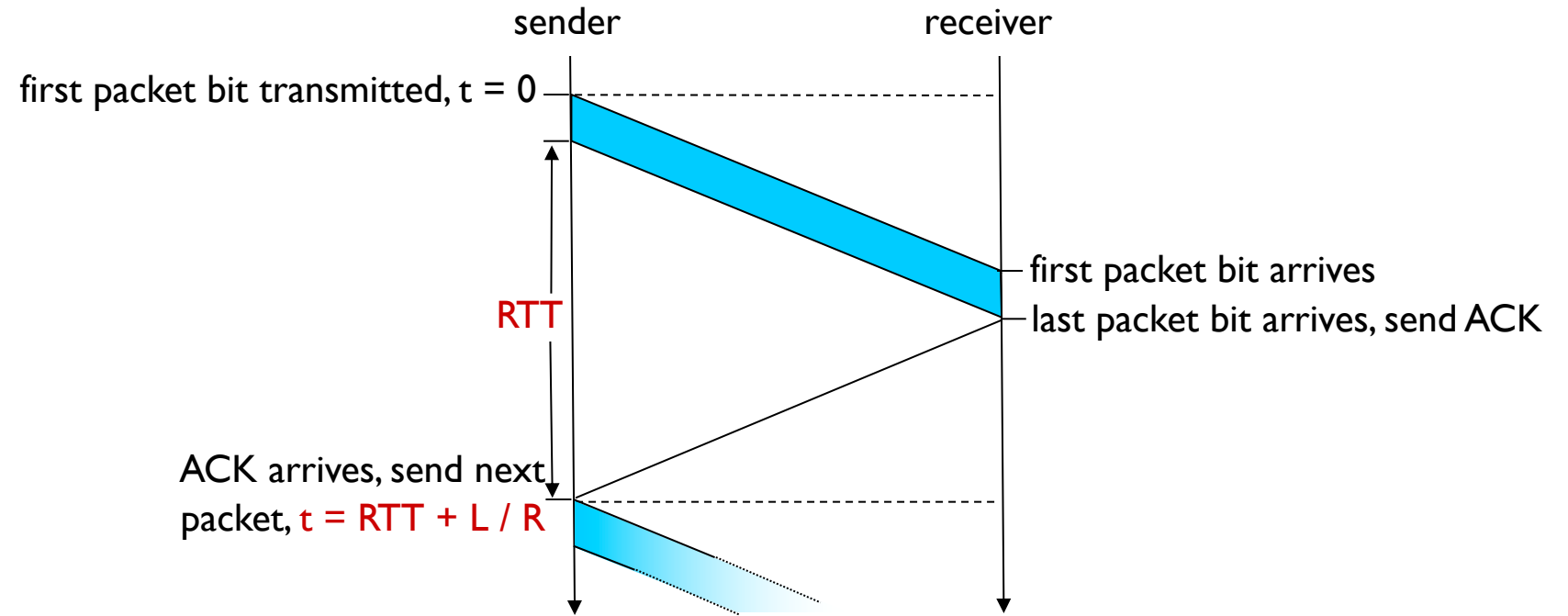
True or false?

- Sender knows whether DATA is correctly 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

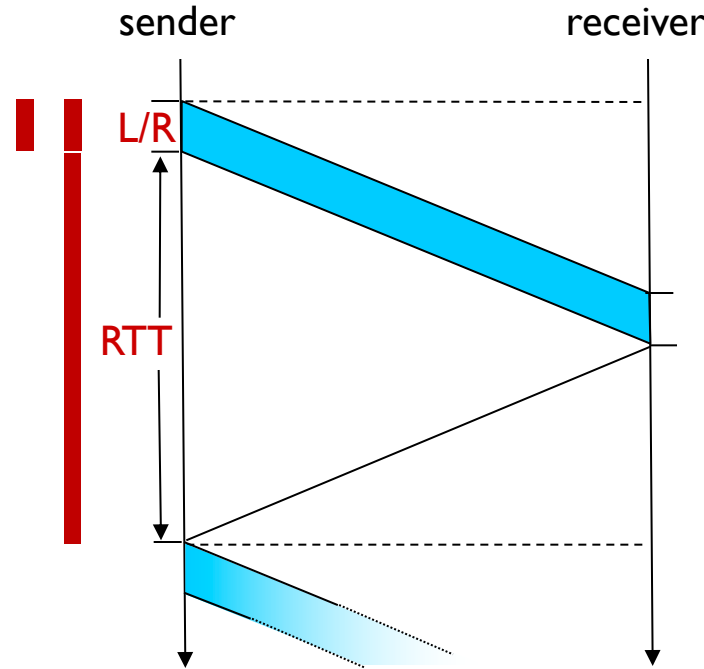


Performance of stop-and wait

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

stop-and-wait suffers from very low link utilization

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

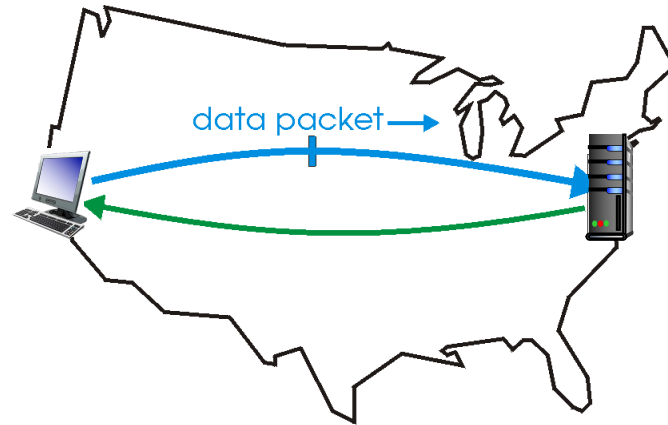


What is the root cause of this low link utilization?

Pipelining allows to send multiple “in-flight” packets

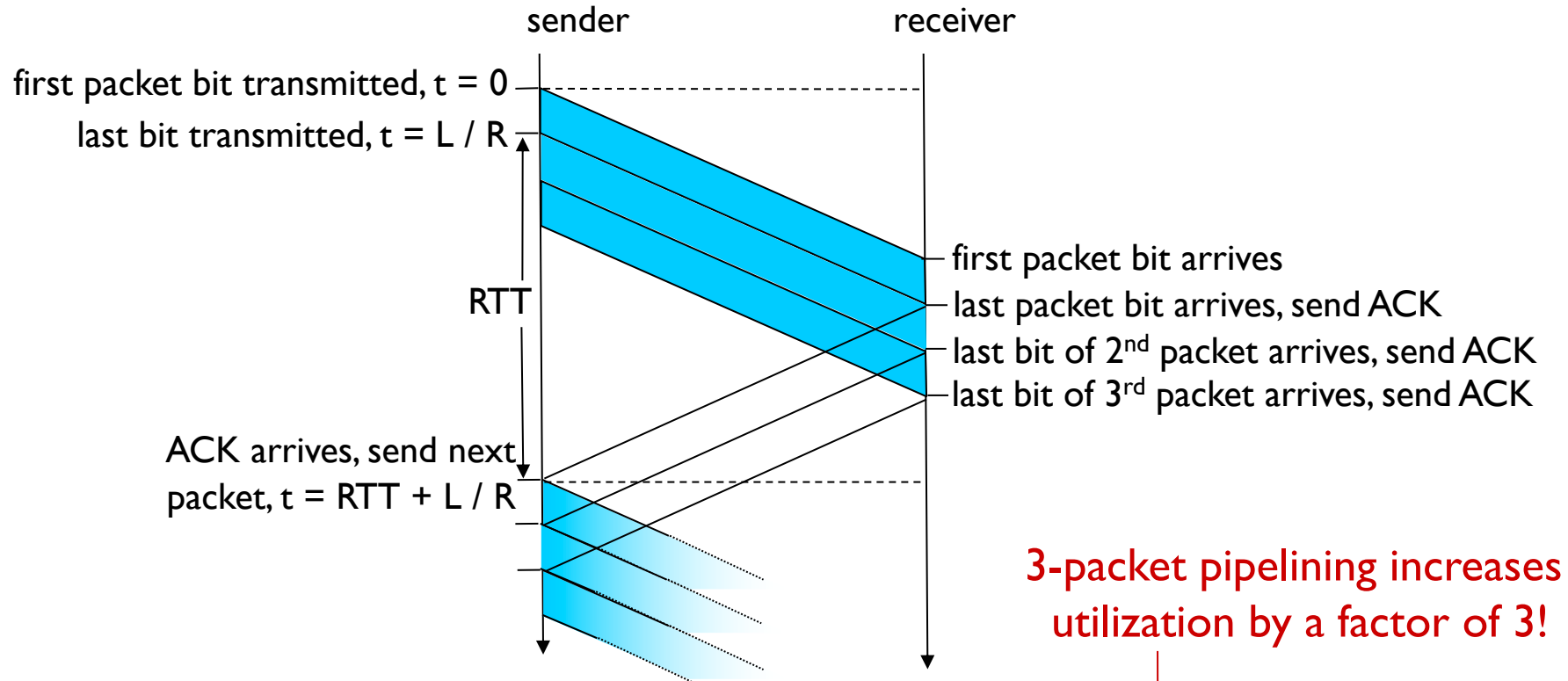
In-flight packets: yet-to-be-acknowledged packets

- 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!

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

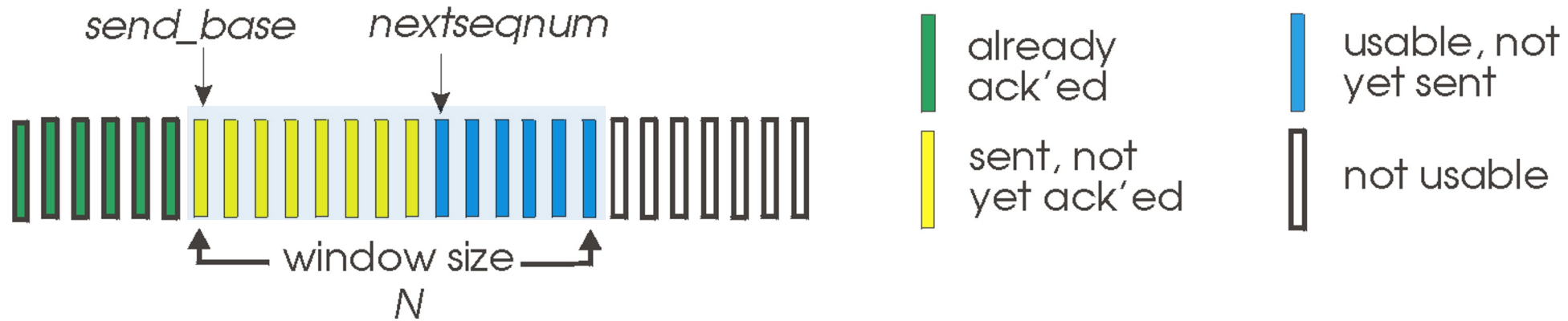
Outline

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

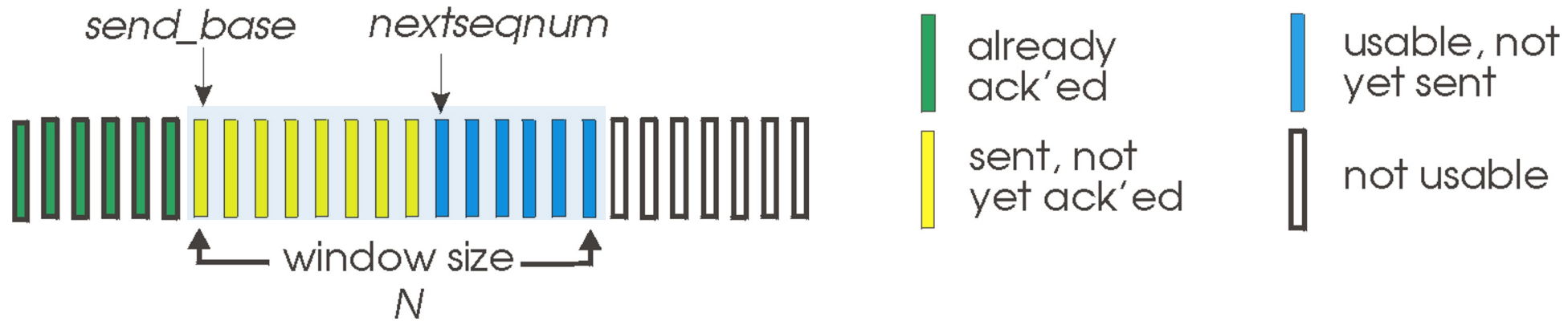


True or false?

- (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



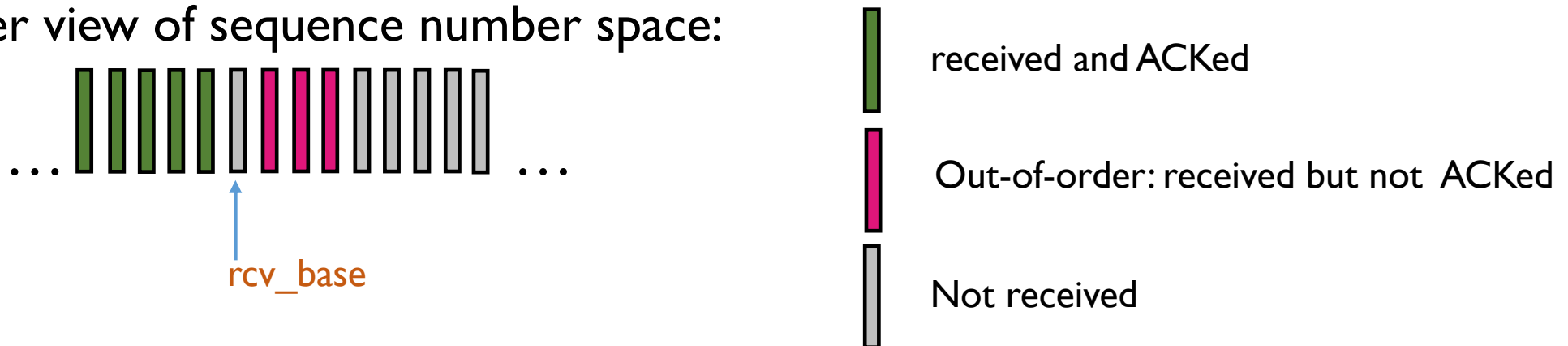
Answer key

- 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**

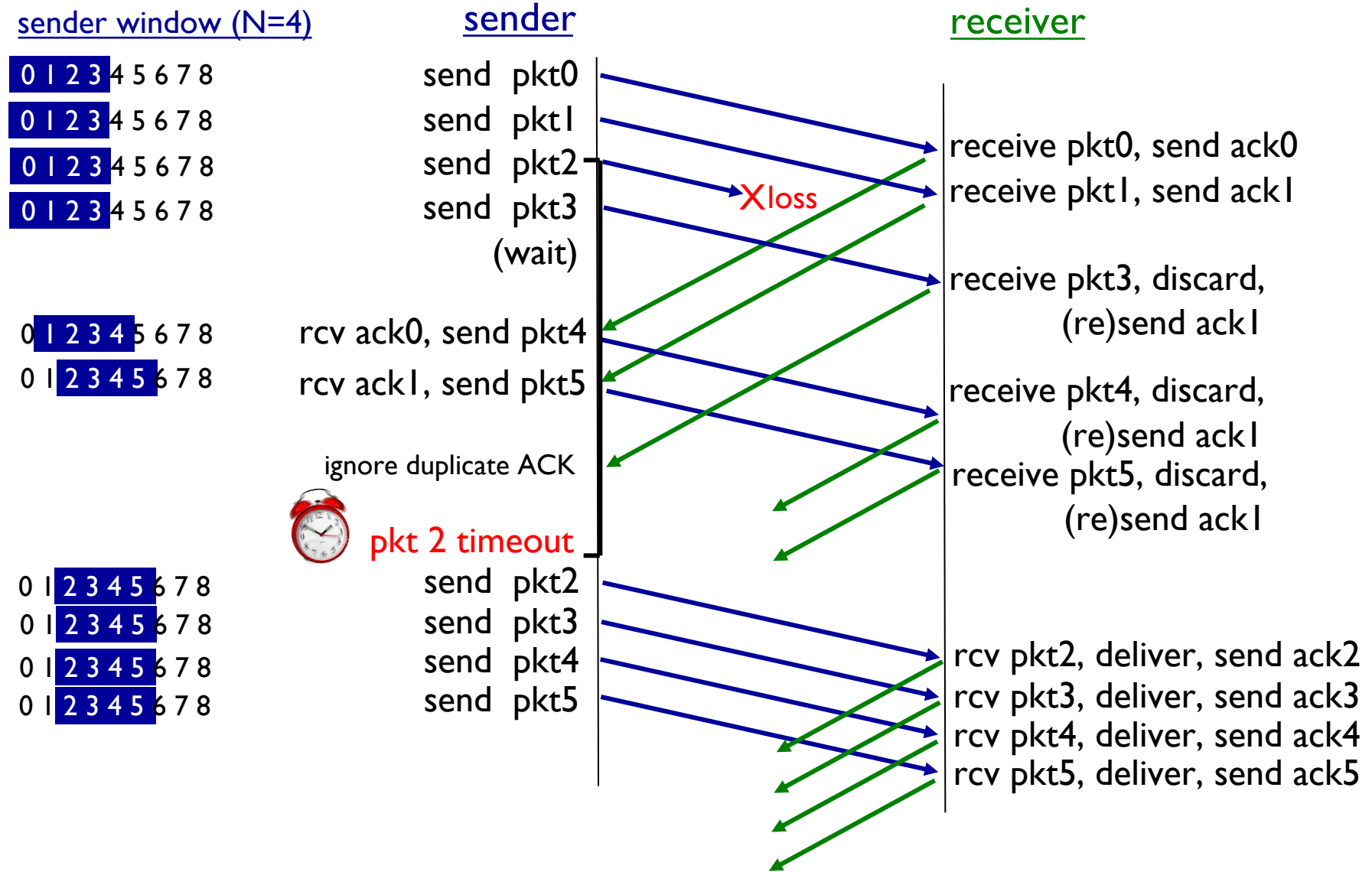
Go-Back-N receiver always send ACK(**n**) where **n** is highest in-order seq # received correctly

- 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

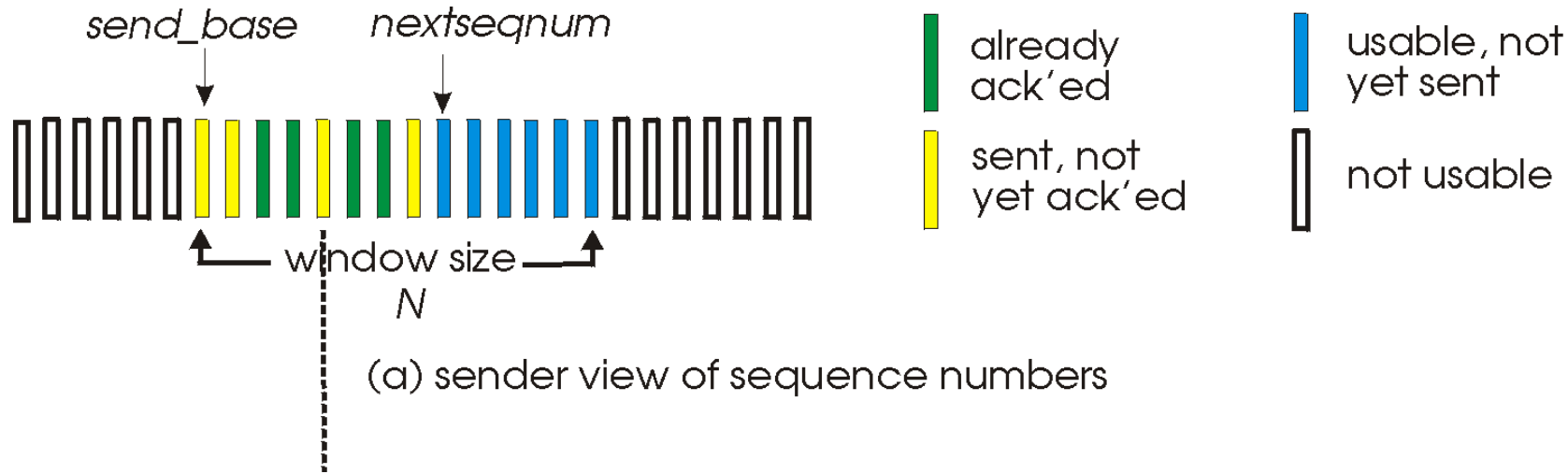
True or false?

- 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 ptkts

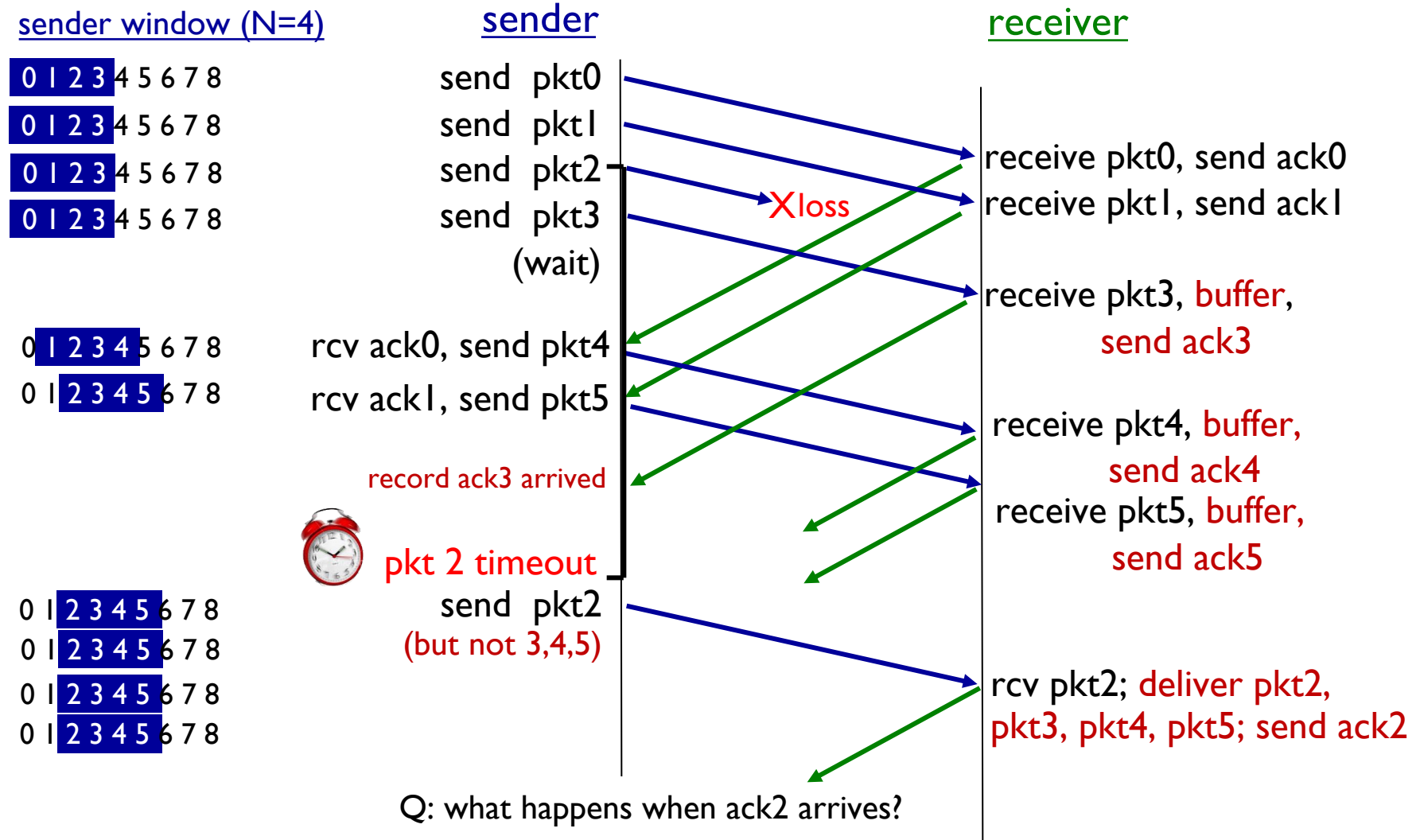
Selective repeat answer key

- 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



Selective Repeat 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
-  6. What should be the proper window size?

Sequence number with 2 bits

0, 1, 2, 3, 0, 1, 2, 3, ...

■ Can we allow window size 5?

• 0, 1, 2, 3, 0, 1, 2, 3, ...

■ How about window size 3?

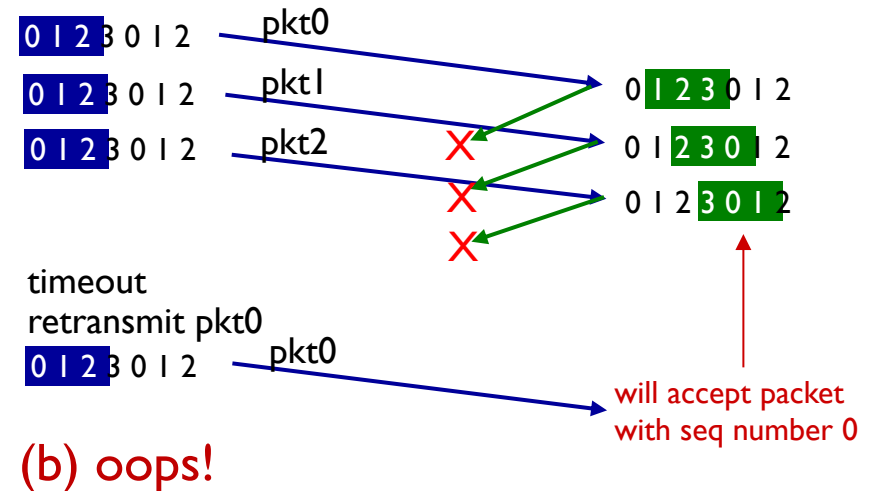
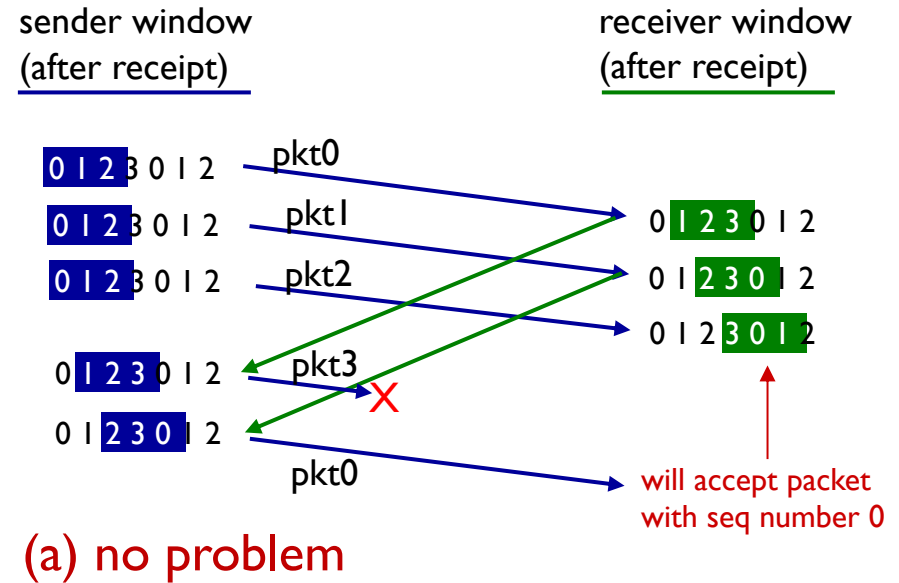
Receiver cannot distinguish 1st and 5th segment
because they have the same seq no of 0

Seq no and window size

example:

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

Why is this happening?



Seq no and window size

example:

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

WHY is this happening?

sender window
(after receipt)

0 1 2 3 0 1 2
pkt0
0 1 2 3 0 1 2
pkt1
0 1 2 3 0 1 2
pkt2
0 1 2 3 0 1 2
pkt3

receiver window
(after receipt)

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

will accept packet
with seq number 0

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

0 1 2 3 0 1 2
pkt2
timeout
retransmit pkt0
0 1 2 3 0 1 2
pkt0

will accept packet
with seq number 0

(b) oops!

Sequence number with 2 bits

0, 1, 2, 3, 0, 1, 2, 3, ...

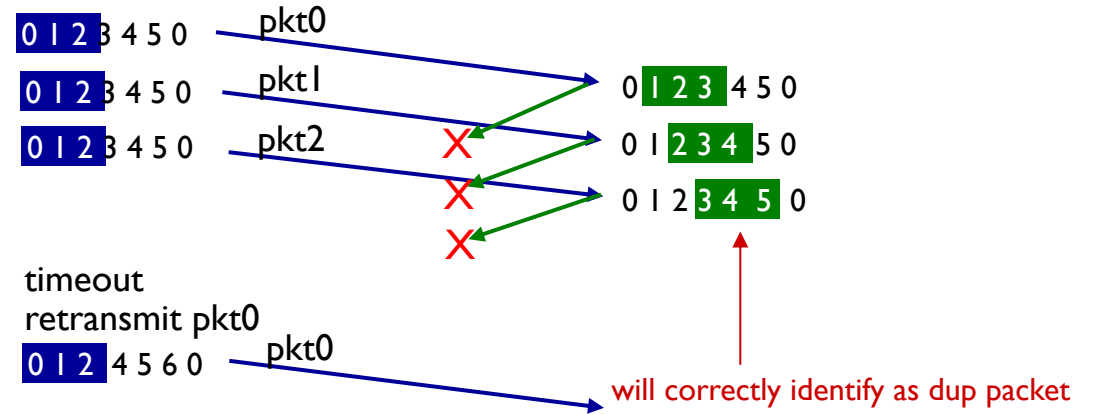
- Sender's retransmission of 1st segment falls into receiver's window of 5th segment
 - If seq no space is infinite would this ever happen?
- The "highest" seq no in receiver window should NOT overlap with the "lowest" seq no in sender window

Sequence no space should fit entire sender window
and receiver window WITHOUT overlap!

Seq no $\geq 2 \times$ window size

example:

- seq #s: 0, 1, 2, 3, 4, 5
- window size=3



With sufficiently large seq number space,
sender's window does NOT overlap with receiver's window

Backup Slides

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

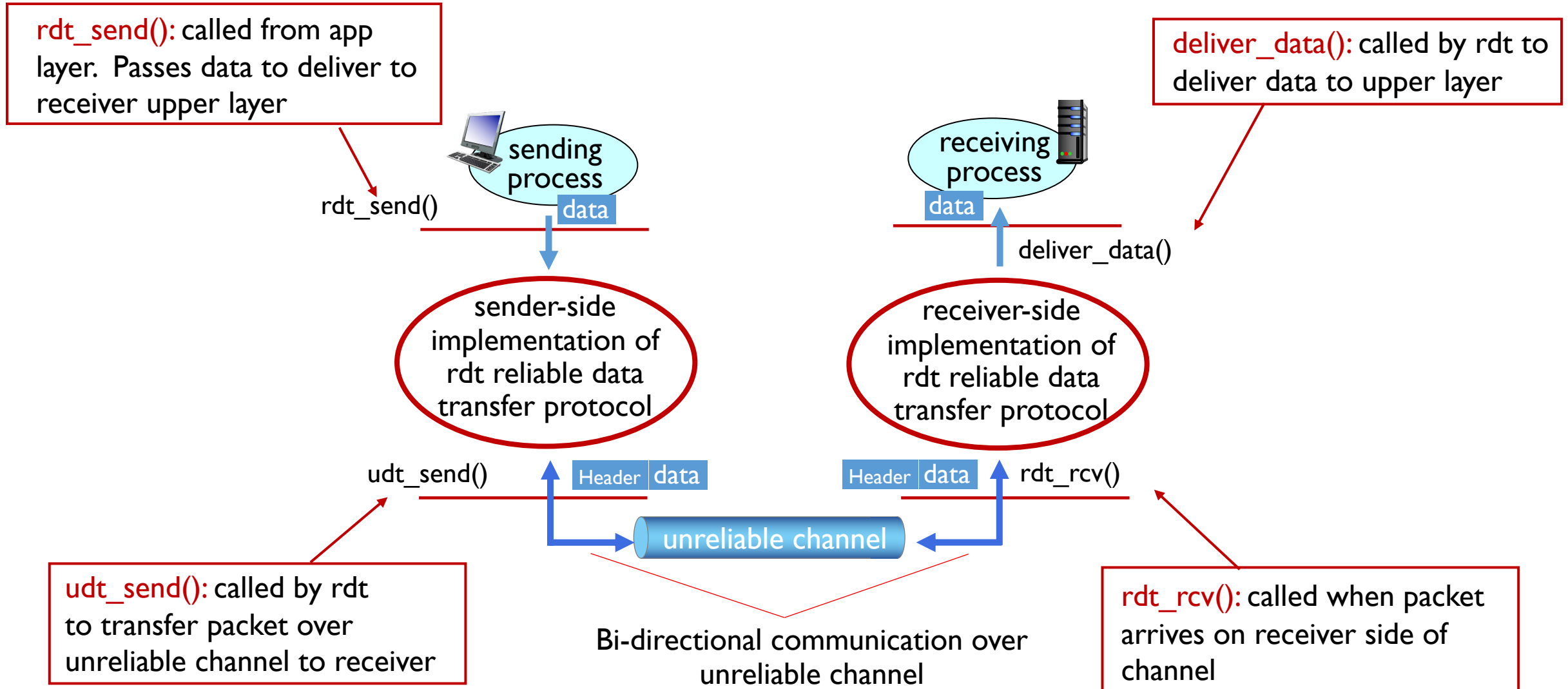
packet n in [rcvbase-N, rcvbase-1]

- ACK(n)

otherwise:

- ignore

Reliable data transfer protocol (rdt): interfaces



Acknowledgements

Slides are adopted from Kurose' Computer Networking Slides

Backup Slides

What if ACK/NAKs get 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 should add **sequence number** to each pkt to inform Receiver
- Receiver discards (doesn't deliver up) duplicate pkt
 - a packet with previously seen sequence number

rdt2.1: discussion

sender:

- 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

receiver:

- 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?