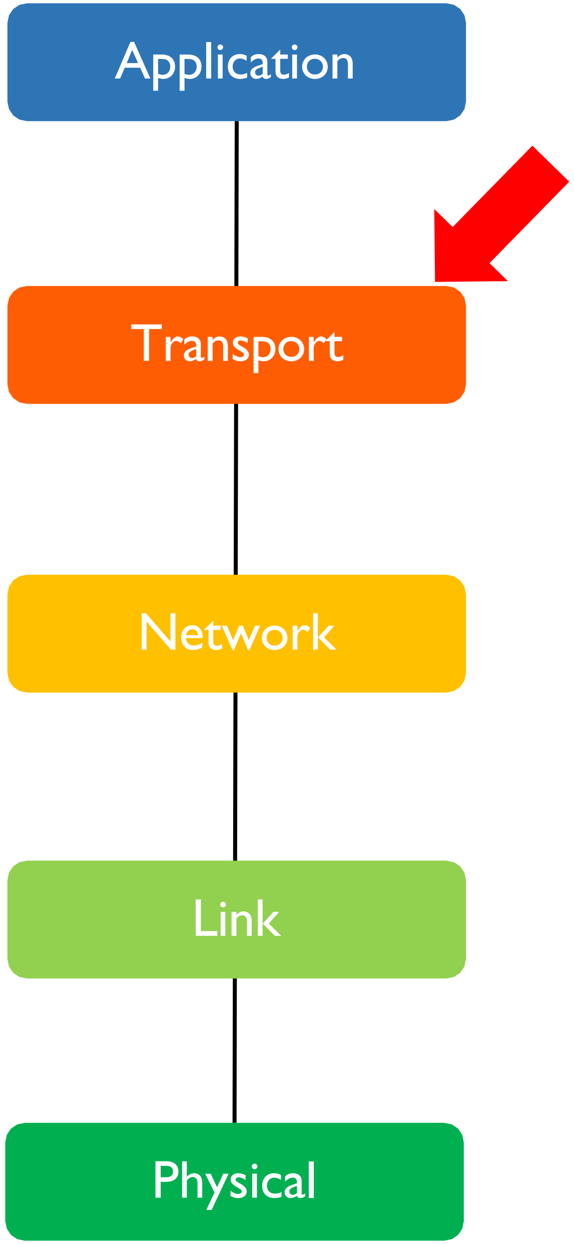
Lesson 05-02:
Principles of Reliable Data Transfer

CS 326E Elements of Networking

Mikyung Han

mhan@cs.utexas.edu

Internet Reference Model

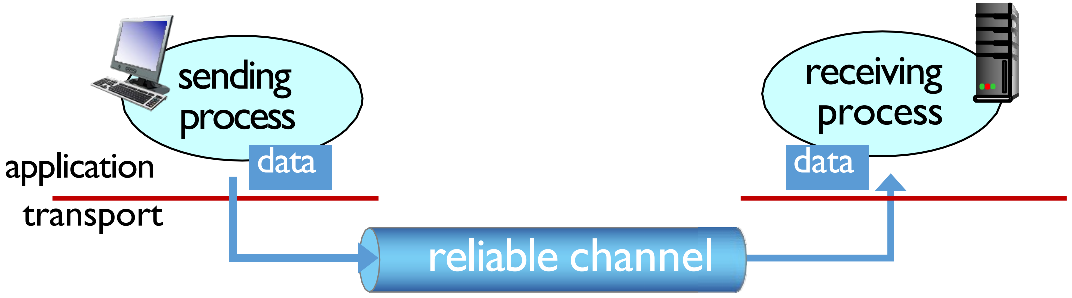


Example Protocols		Responsible for
FTP, HTTP, SMTP	Application	application specific needs
TCP, UDP	Transport	process to process data transfer
IP	Network	host to host data transfer across different network
Ethernet,WiFi	Link	data transfer between physically adjacent nodes
802.3 PHY	Physical	bit-by-bit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

Outline

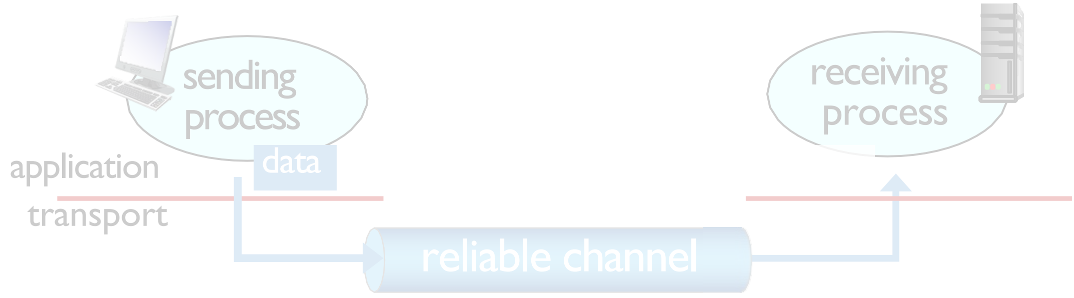
🤘 0. What is reliable data transfer?

Principles of reliable data transfer

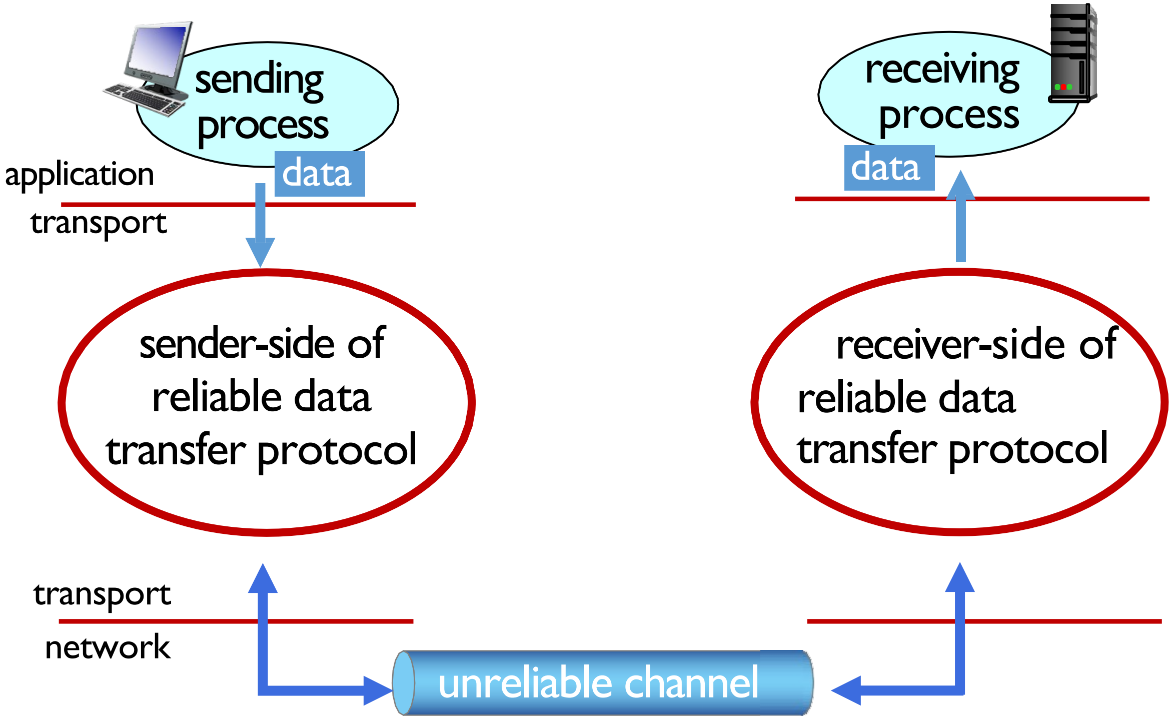


reliable service abstraction

Principles of reliable data transfer



reliable service abstraction

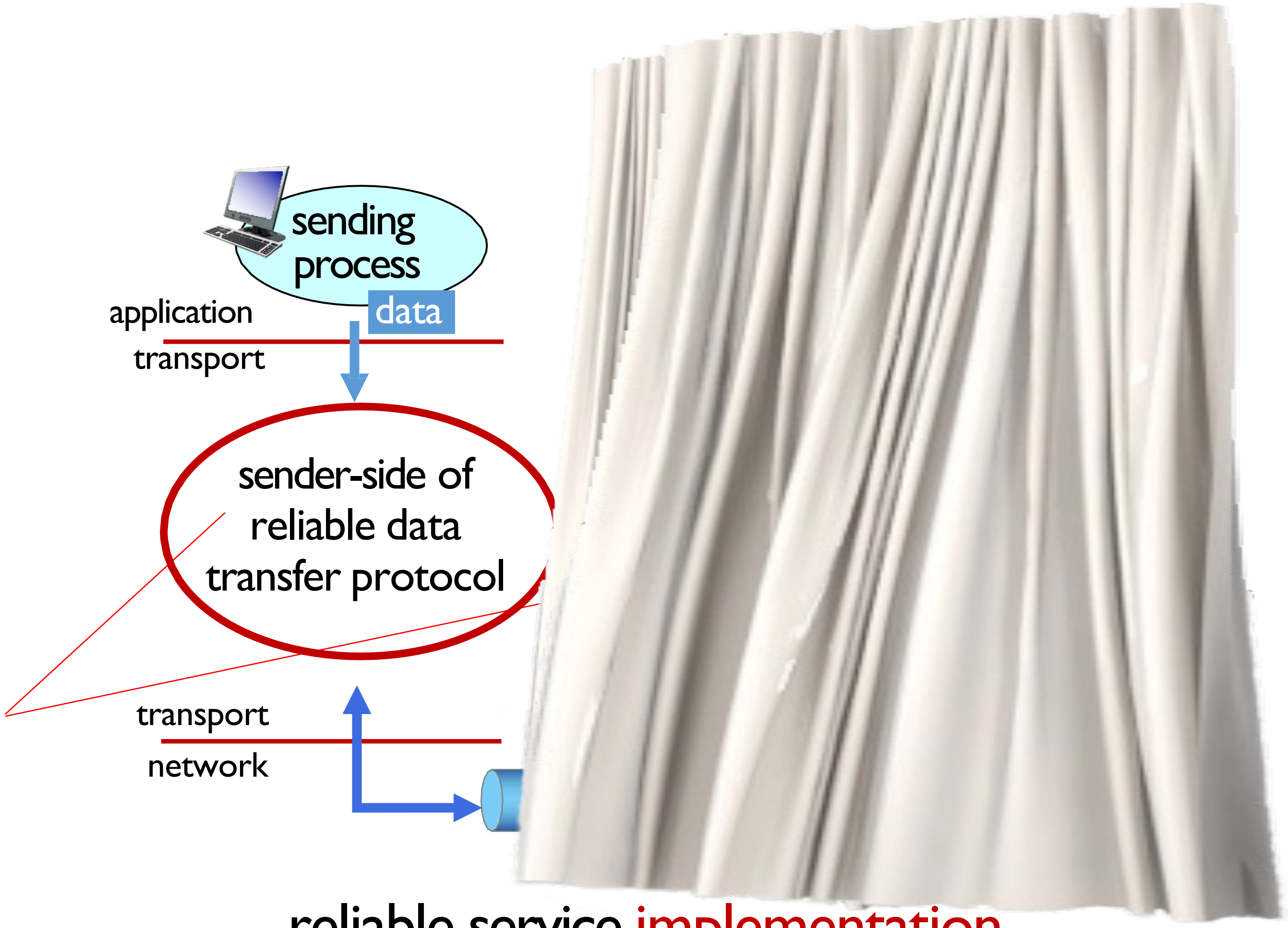

sending process
receiving process
application
transport
data
data
sender-side of reliable data transfer protocol
receiver-side of reliable data transfer protocol
transport
network
unreliable channel

reliable service implementation

Principles of reliable data transfer

Sender, receiver do not know the "state" of each other, e.g., was a message received?

- unless communicated via a message



reliable service implementation

Let’s start with perfect condition: rdt 1.0

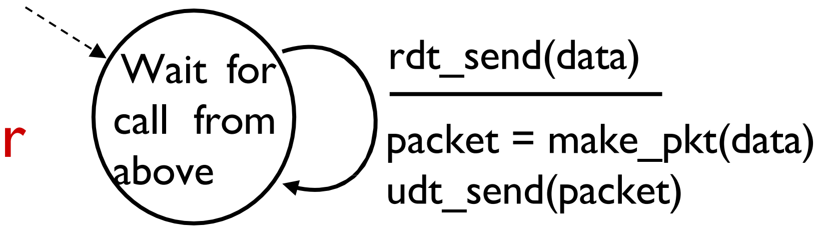
- No packet loss

- No bit errors

rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

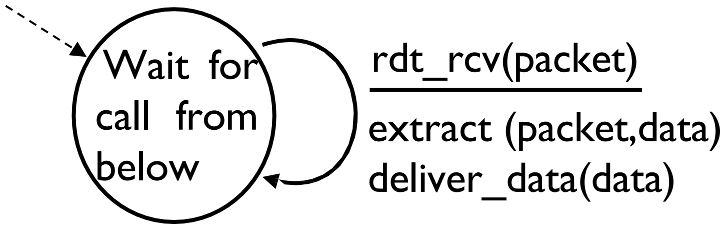


Wait for call from above

rdt_send(data)

packet = make_pkt(data)
udt_send(packet)

receiver



Wait for call from below

rdt_rcv(packet)

extract (packet,data)
deliver_data(data)

Outline

1. Channel with bit errors: rdt 2.0

rdt2.0: channel with bit errors

- How to detect bit errors?

- How to recover from errors?
  - Sender retransmits upon the receipt of NAK
  - NAKs: receiver explicitly tells sender that pkt had errors

stop and wait

sender sends one packet, then waits for receiver response

What is the fatal flaw of rdt 2.0?

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

Say, sender retransmits upon receiving ACK which was corrupted.

- (T/F) Receiver knows the retransmit pkt is a duplicate

How to tell if the pkt received is a new packet or a duplicate?

Sequence number distinguishes a new packet from a duplicate

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 (duplicate)

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

(RDT 2.1) So far, we have

- ✓ Checksum
- ✓ DATA + Sequence number
- ✓ ACK or NAK
- ✓ Retransmission of DATA

Outline

1. rdt 2.0
2. rdt 2.1 and rdt 2.2

rdt2.1: DATA has sequence no + ACK/NAK

How about having just ACK pkts (no NAKs)? Any potential benefits?

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- How to say NAK with just ACKs?
- Consider below scenario
  - Sender sends DATA 1 but it got corrupted
  - Receiver sends ACK?!
  - What additional info should this ACK contain?

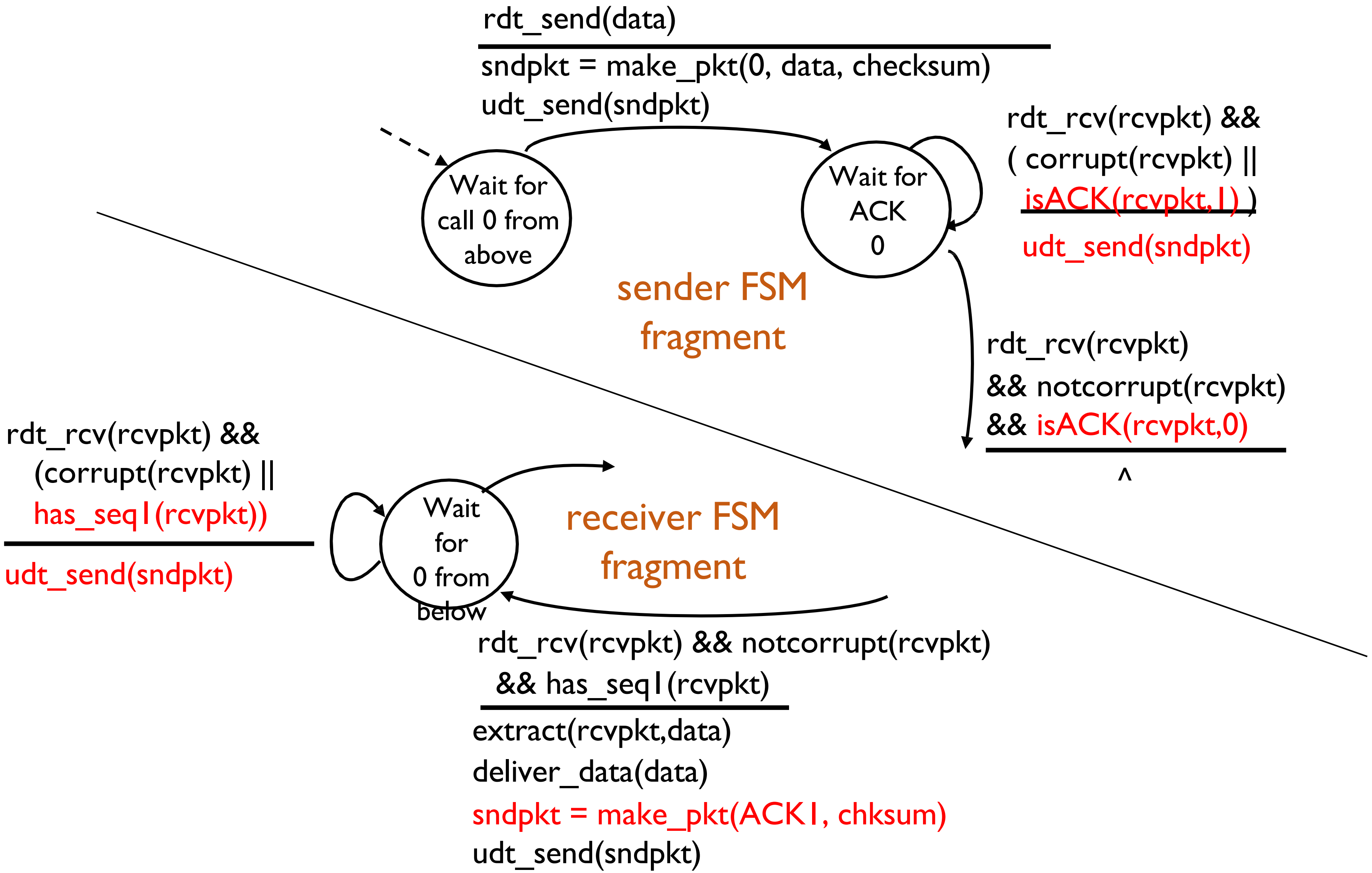
We need an ACK number (seq no for ACKs)!

rdt2.2: a NAK-free protocol

- ACK with ACK no in action
  - Sender sends DATA 1 but it got corrupted
  - Receiver sends… ACK 0 or ACK 1?
  - Depends on the protocol definition of ACK!
  - ACK0 could mean either
    - DATA0 was successful, so send me DATA1 (RDT way)
    - Or, DATA0 was unsuccessful, so send me DATA0 again! (TCP way)

RDT 2.2: Having ACK # allows us to be NAK-free!

rdt2.2: sender, receiver fragments



rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for call 0 from above

Wait for ACK 0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

udt_send(sndpkt)

sender FSM fragment

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)

Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))

udt_send(sndpkt)

Wait for 0 from below

receiver FSM fragment

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

(RDT 2.2) So far, we have

- ✓ Checksum
- ✓ DATA + Sequence number
- ✓ ACK only + ACK number
- ✓ Retransmission of DATA

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 #, ACK #, 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

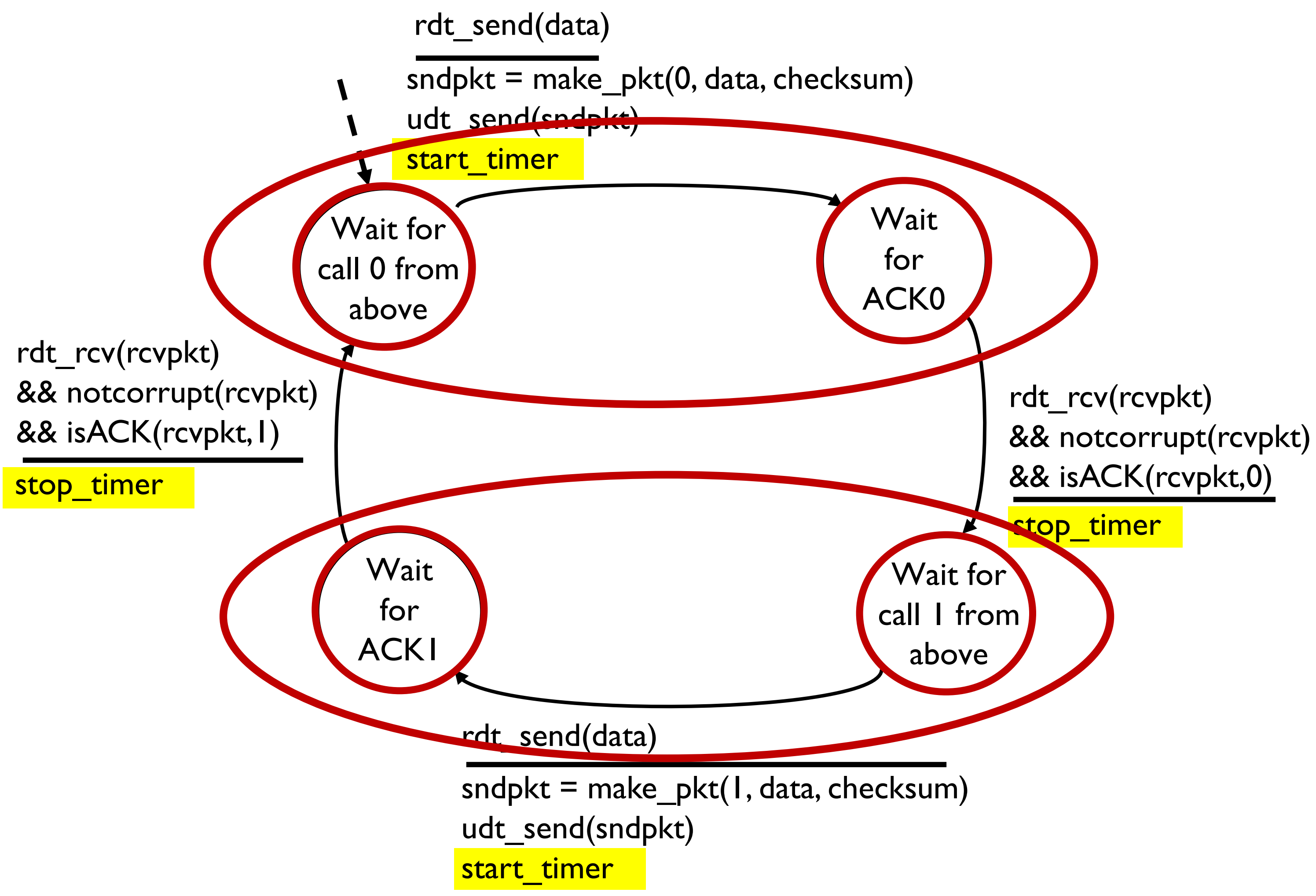
Approach: sender waits "reasonable" amount of time for ACK

- 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

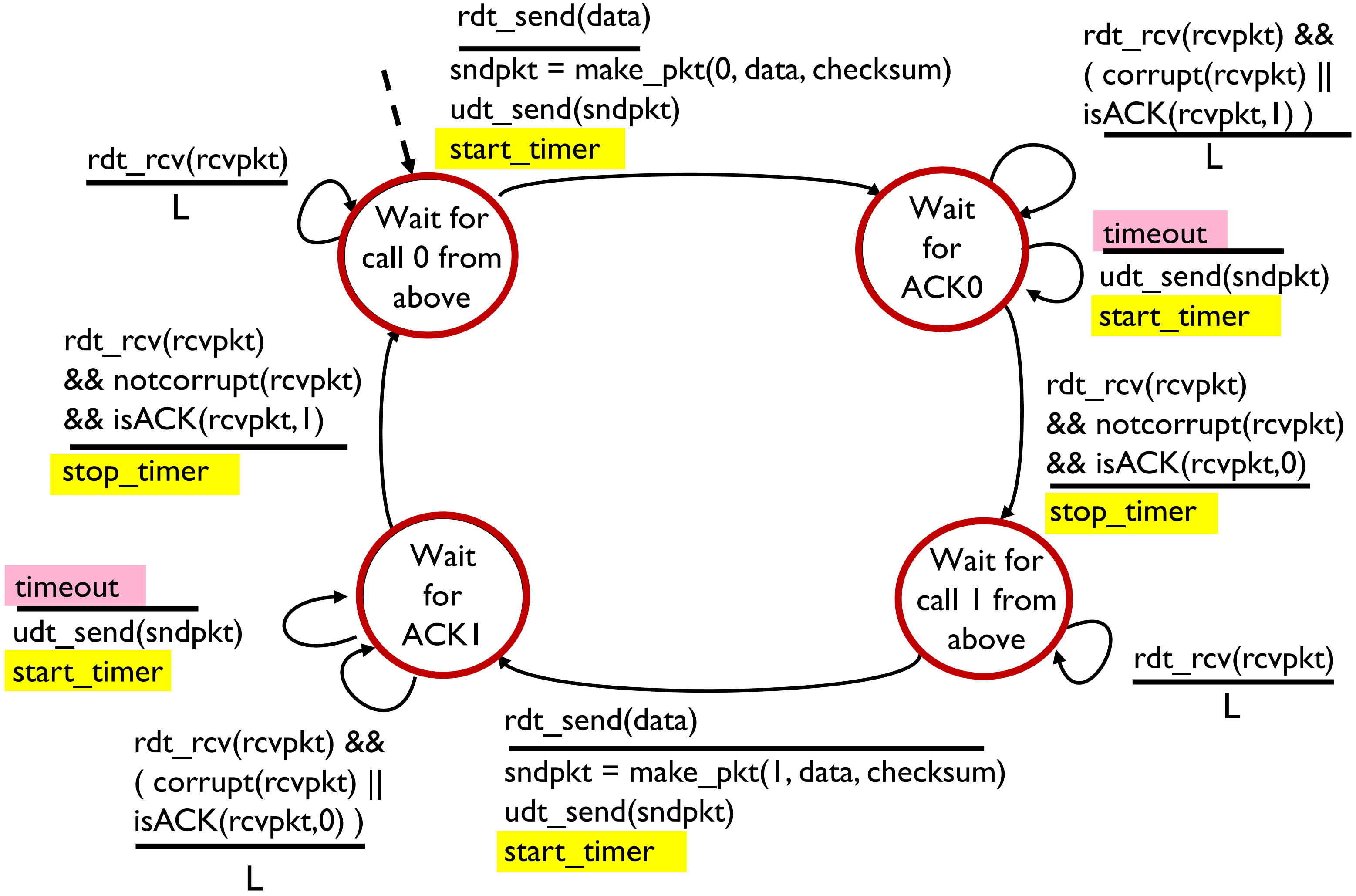
timeout

What is the "reasonable" time?

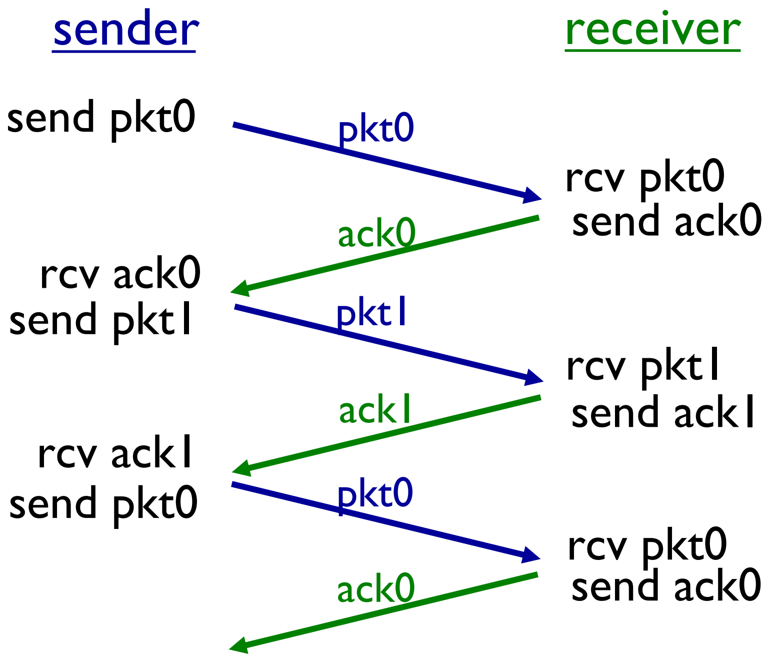
rdt3.0 sender



rdt3.0 sender



rdt3.0 in action



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

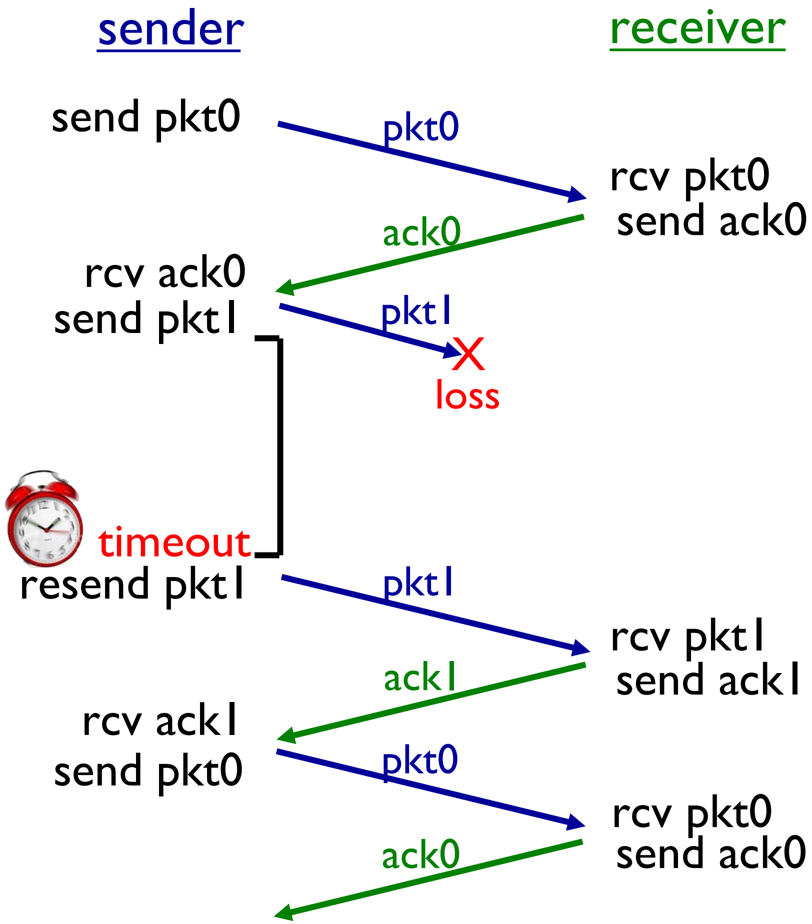
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(a) no loss



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

loss

timeout
resend pkt1

pkt1

rcv pkt1
send ack1

ack1

rcv ack1
send pkt0

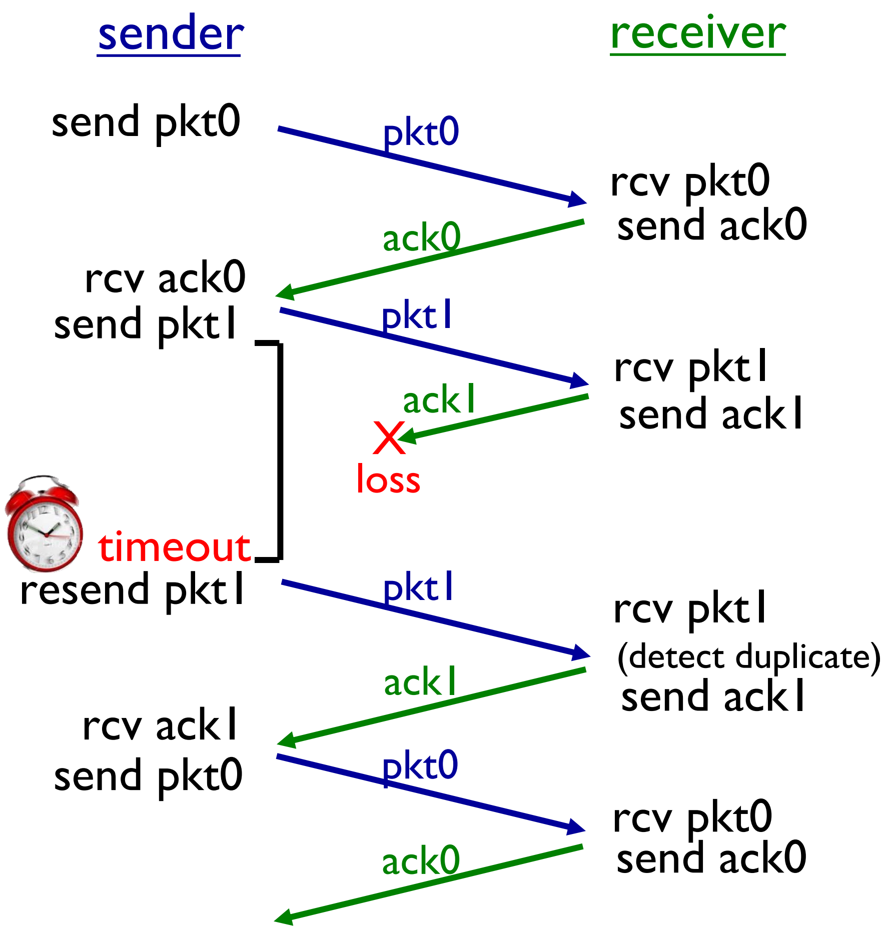
pkt0

rcv pkt0
send ack0

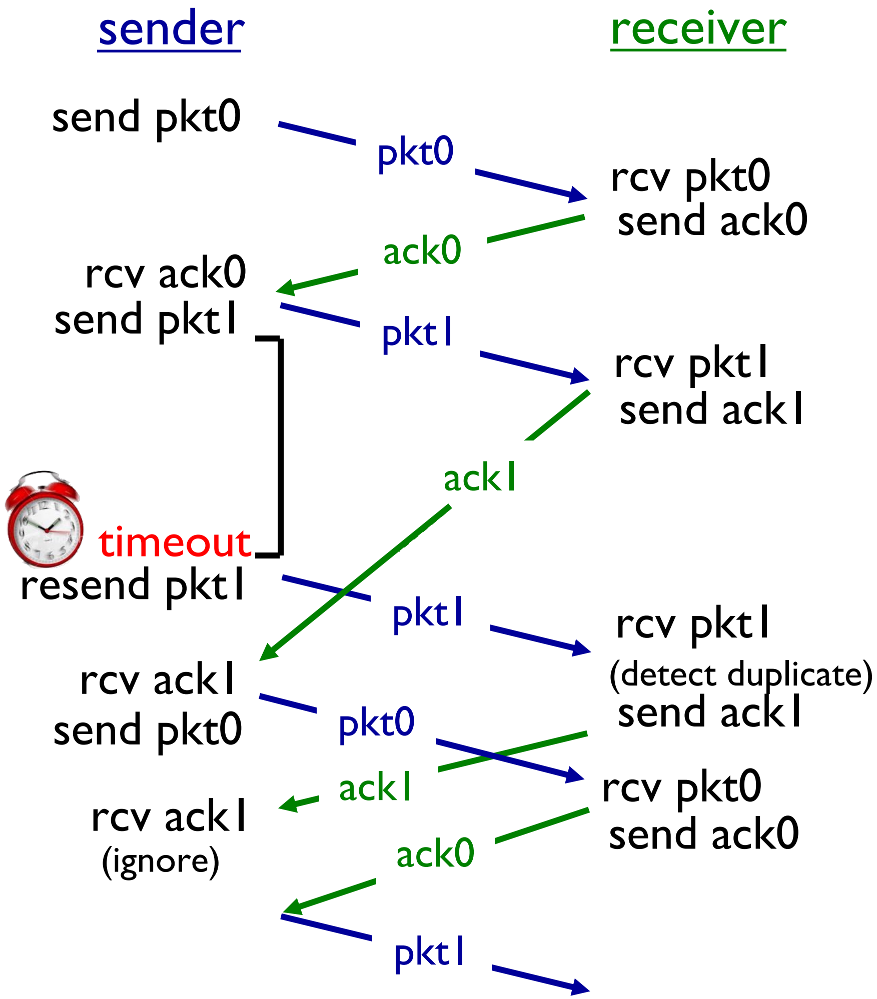
ack0

(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

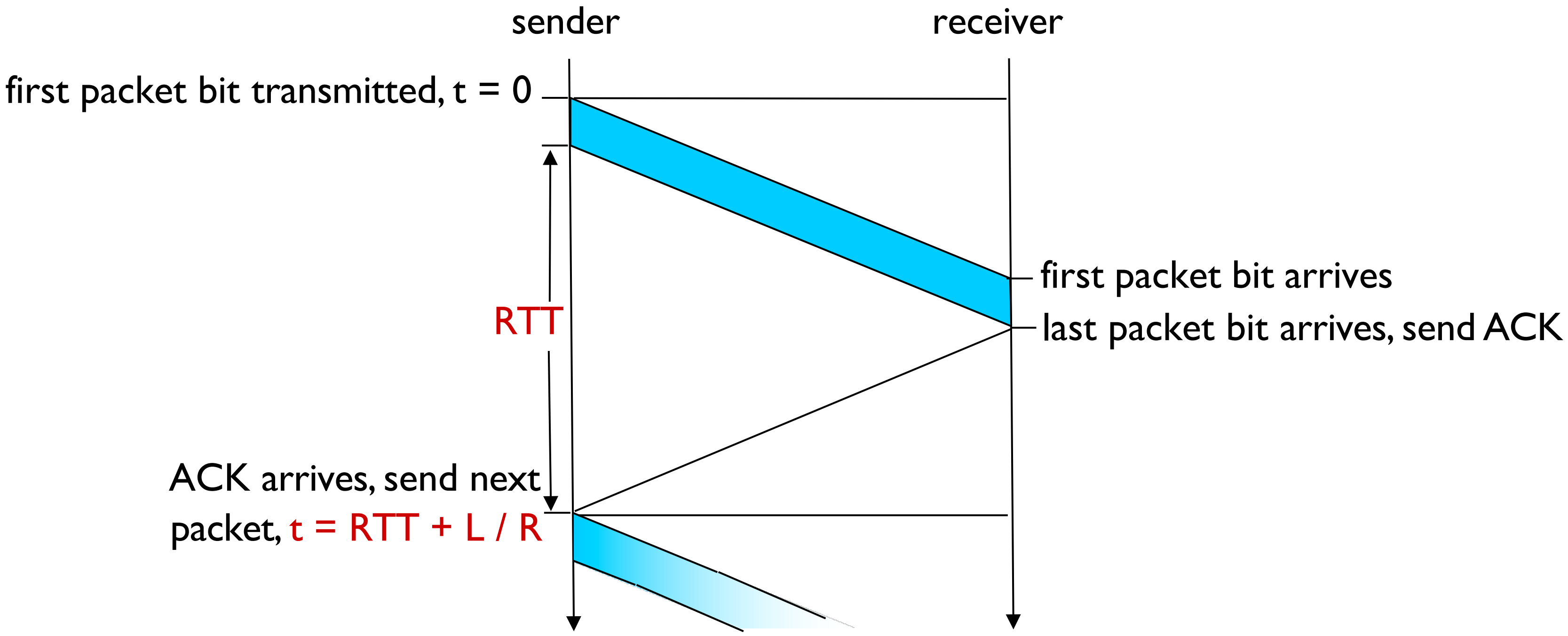
What should be the timeout value in this case?

Kahoot ©

Check Canvas - will be optional extra-credit

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 s prop. delay, 8000 bit packet
  - time to transmit packet into channel:

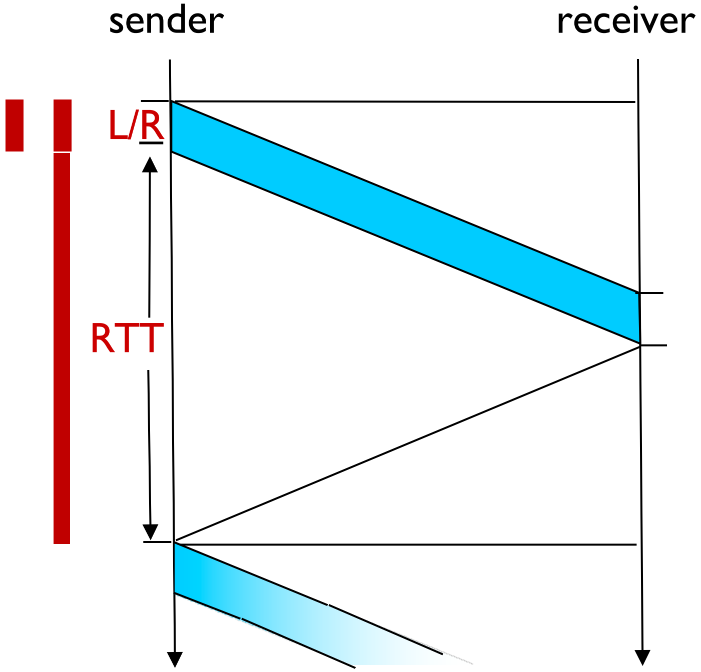
$$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

stop-and-wait suffers from very low link utilization

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

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

$$= 0.00027$$



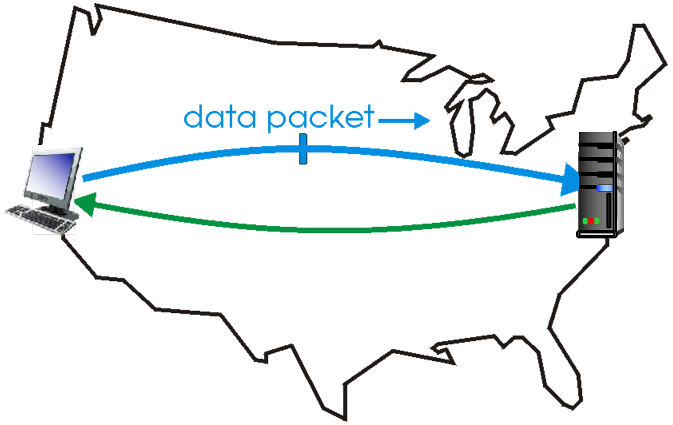
What is the root cause of this low utilization?

Protocol is limiting the performance of underline channel!

Pipelining allows to send multiple “in-flight” packets

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

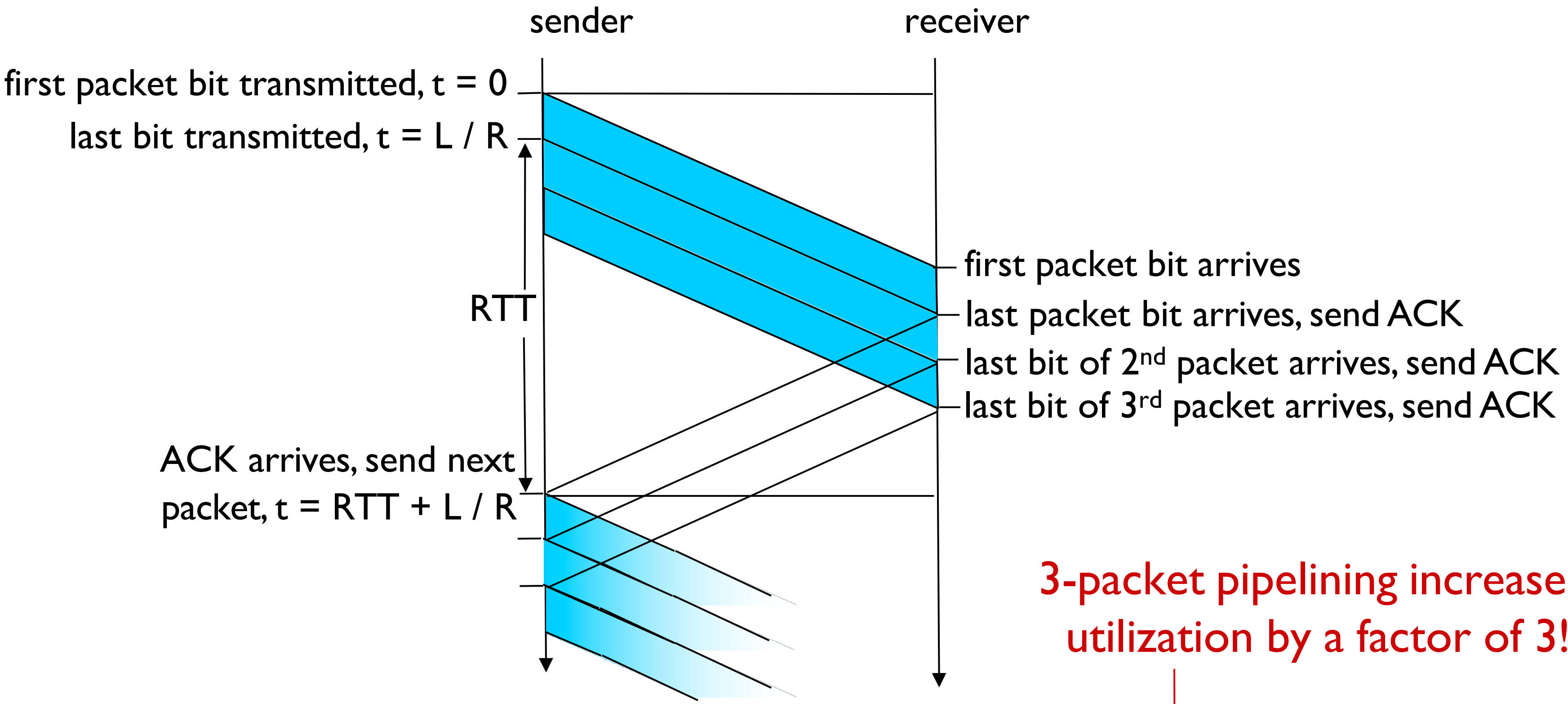
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

Say bye to stop-and-wait. Let’s adopt pipelining!

Pipelining: increased utilization



sender

receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

ACK arrives, send next packet, t = RTT + L / R

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

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

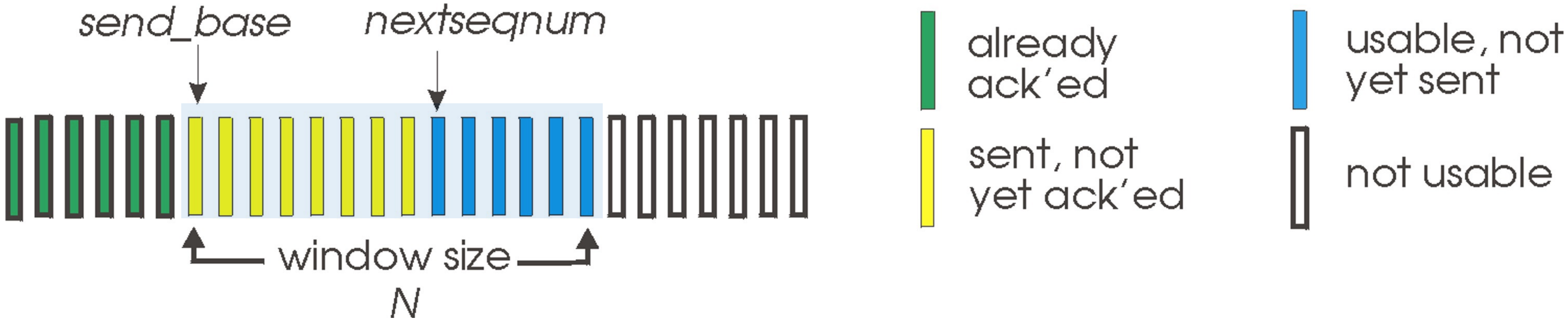
$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

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

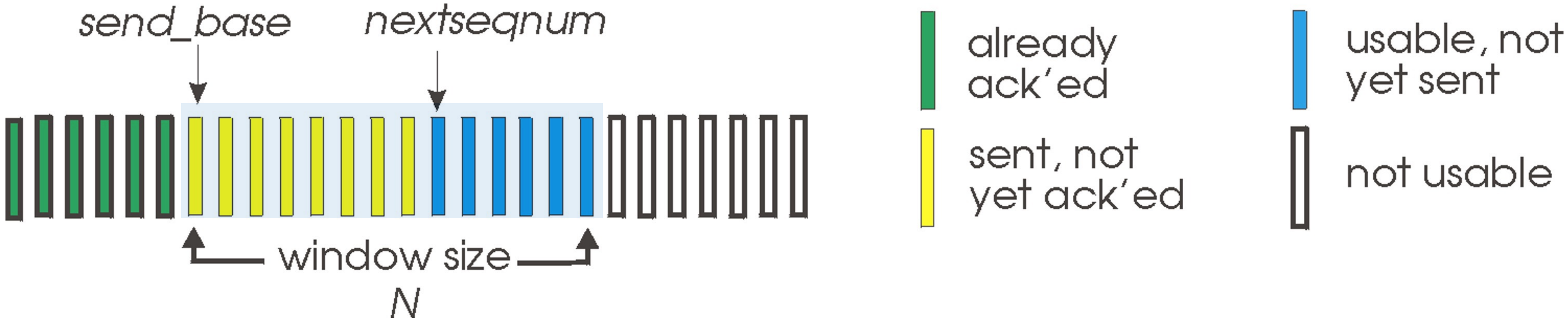


True or false?

- (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



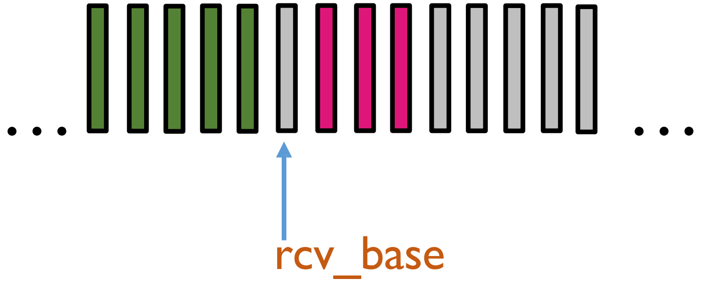
Answer key

- 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 need to buffer)
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

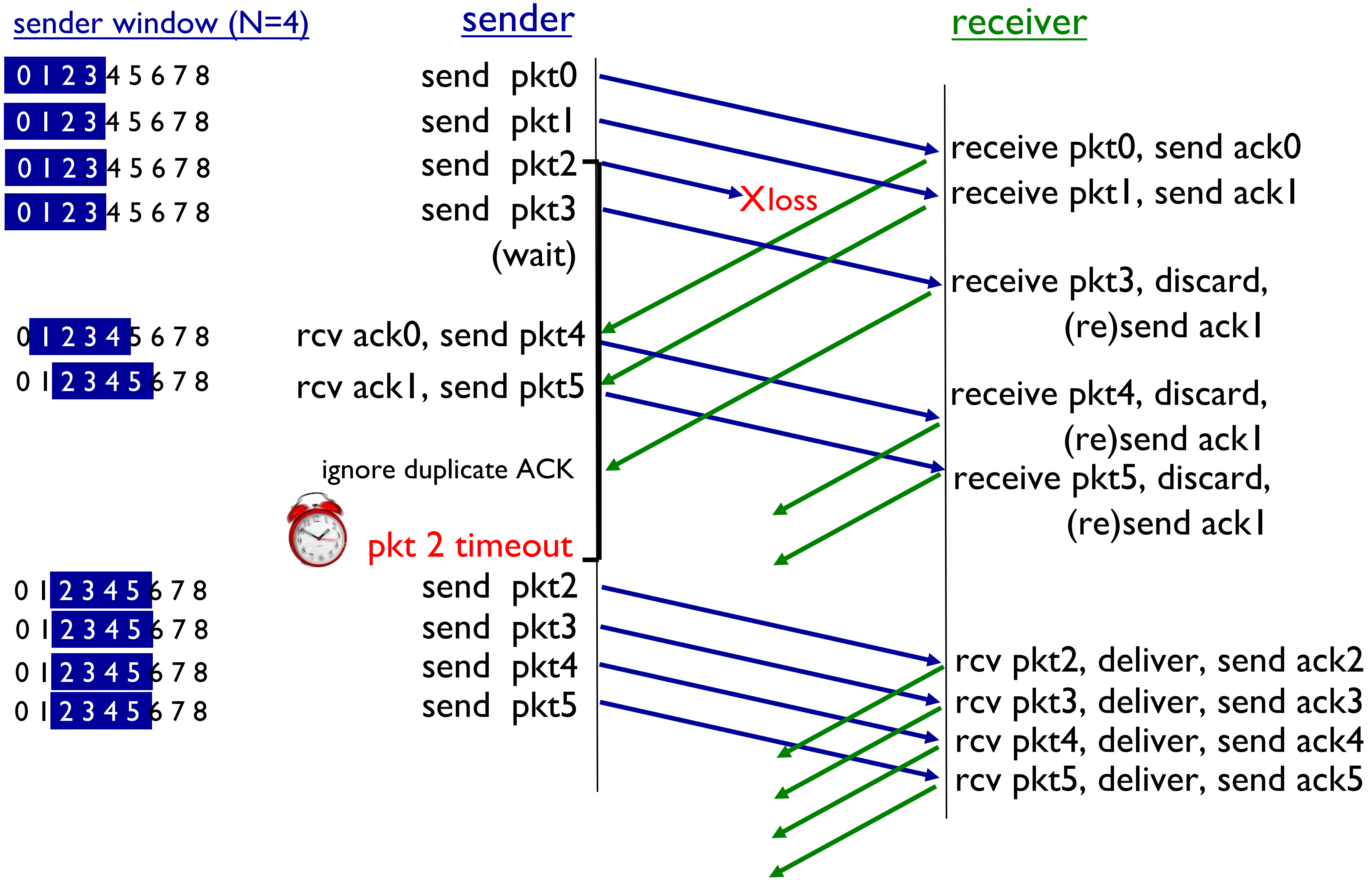


🟩 received and ACKed

🟥 Out-of-order: received but not ACKed

⬜ Not received

Go-Back-N 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

Xloss

receive pkt3, discard, (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

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

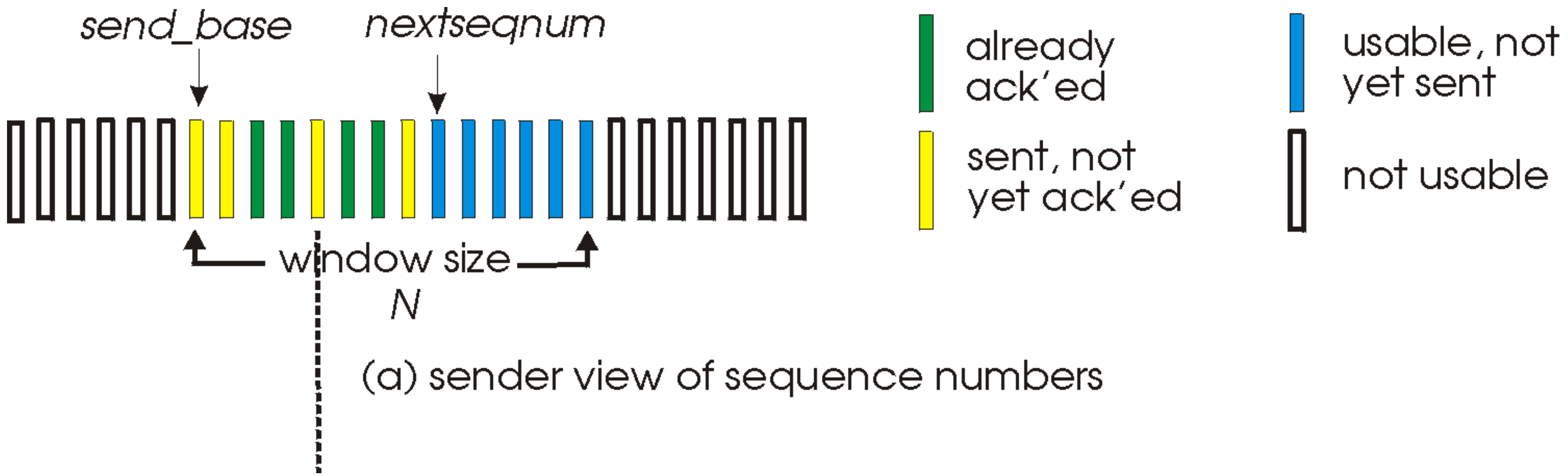
True or false?

- 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 ptks

Selective repeat answer key

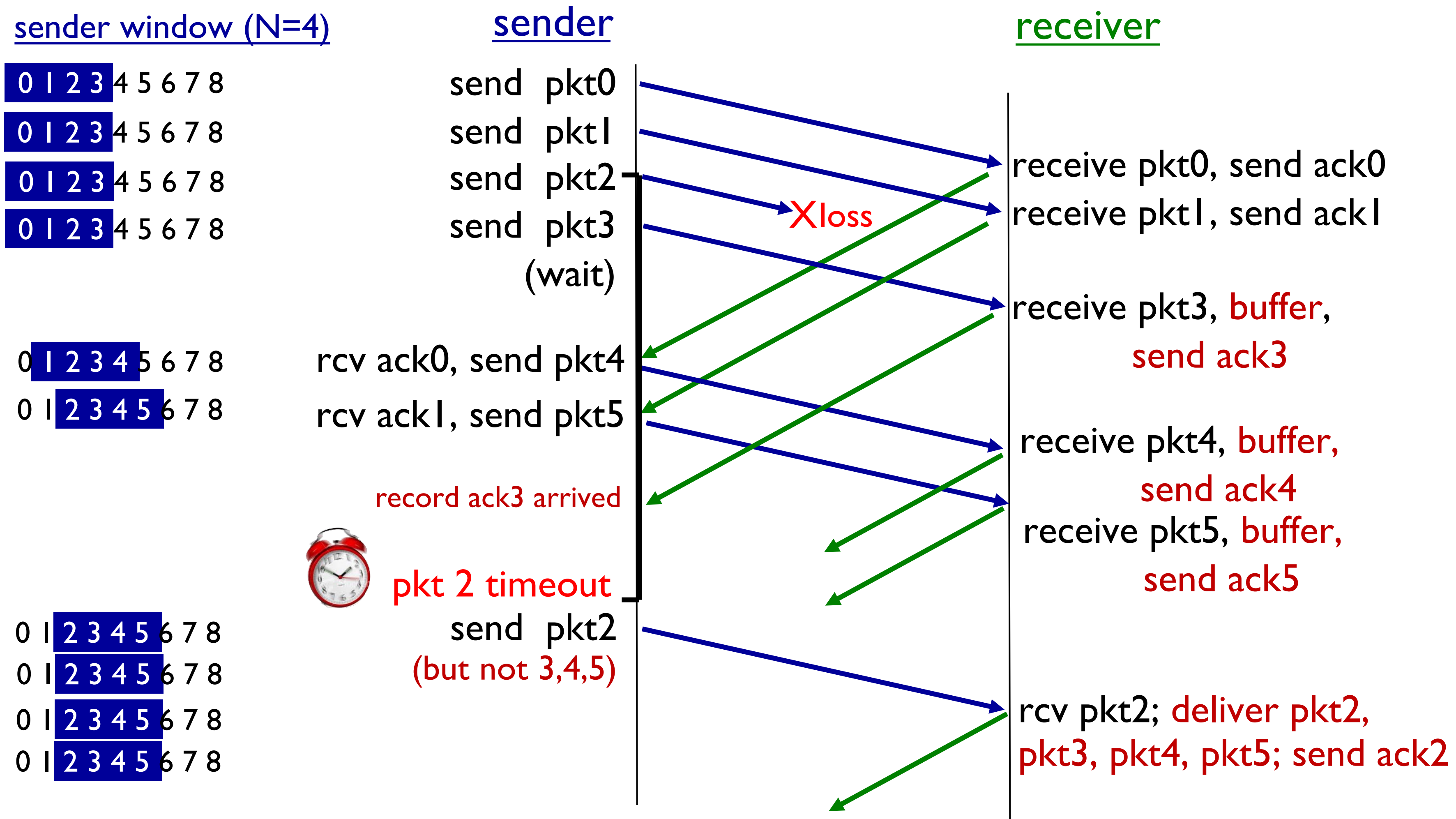
- Receiver should buffer packets for in-order delivery to app. layer
- Sender maintains timer for each in-flight pkt
    - Upon timeout sender retransmits that unACKed packet
- Sender window
    - N consecutive seq #s
    - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective Repeat 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

(but not 3,4,5)

X loss

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

Q: what happens when ack2 arrives?

Compare: GBN vs SR

- Which one uses more memory?
- Which one uses less processing overhead?
- Which one would help fight off very lossy network?

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?

When pipelining there is MORE to consider!

How many in-flight pkts are we allowing?

In other words, what should be the right window size?

Max window size is closely related with size of sequence number!

Consider 2-bit Sequence number

0, 1, 2, 3, 0, 1, 2, 3, …

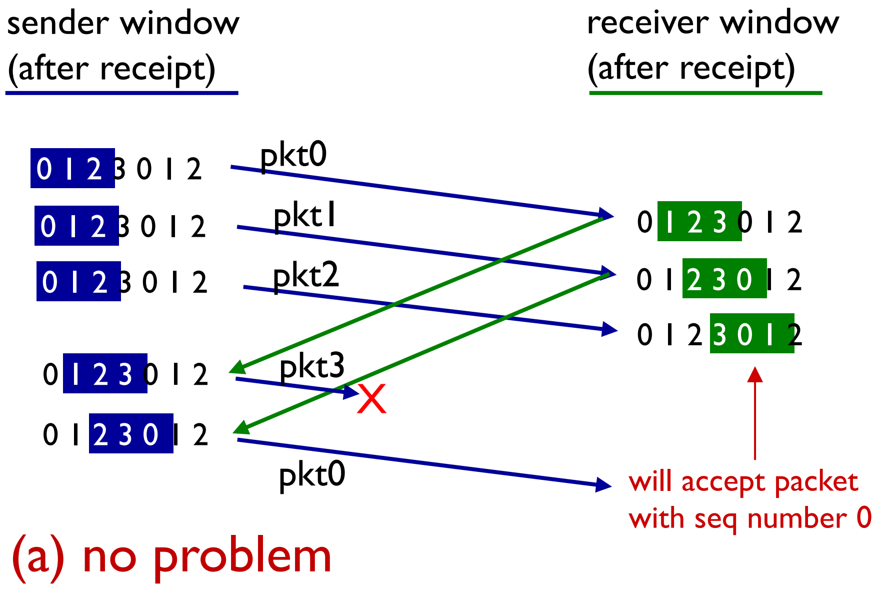
- Can we allow window size 5?
  - 0, 1, 2, 3, 0, 1, 2, 3, …
- How about window size 3?

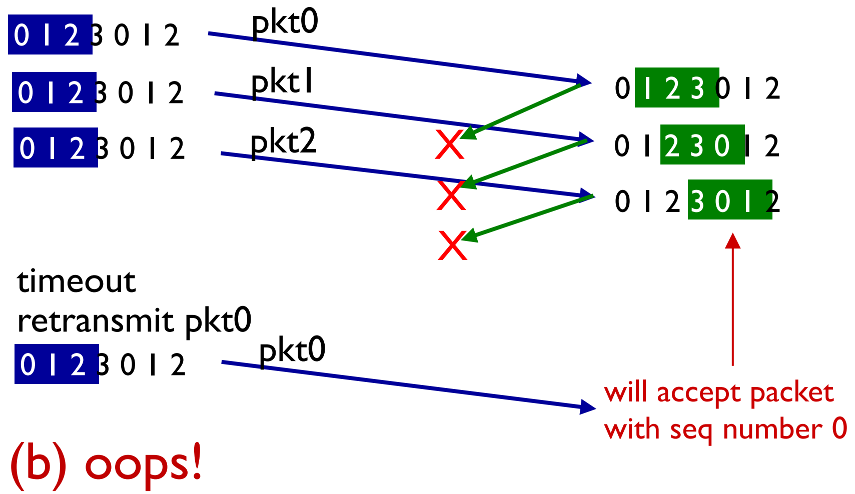
Remember: Receiver should be able to distinguish each packet within the same window

Seq no and window size

example:

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



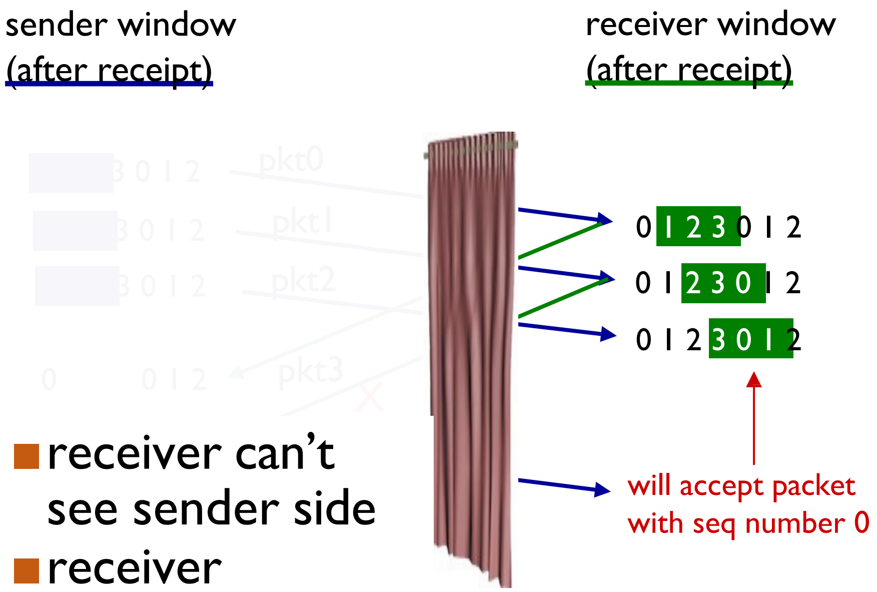


Why is this happening?

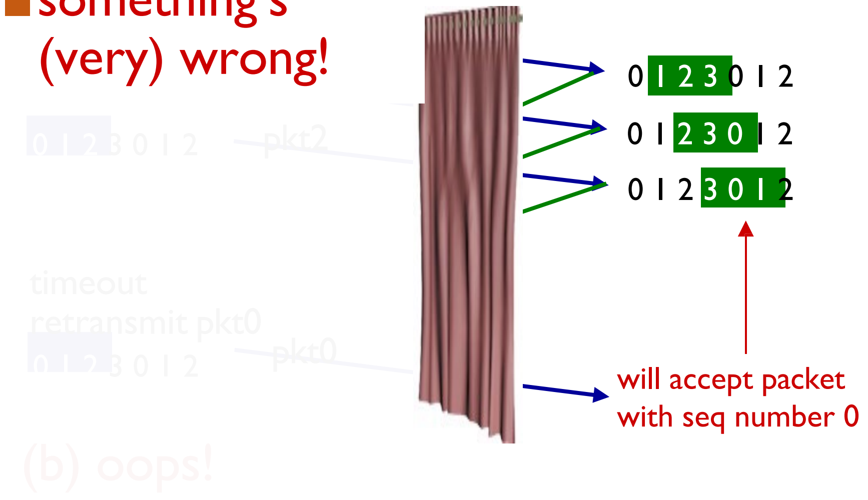
Seq no and window size

example:

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



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



BUT, WHY is this happening?

Sequence number with 2 bits

0, 1, 2, 3, 0, 1, 2, 3 …

- Sender's retransmission of 0 falls into receiver window
  - 0 is mistaken for new 0

- Same thing happens when sender retransmits 1
  - 1 is mistaken for new 1

0, 1, 2, 3, 0, 1, 2, 3 …

If we have infinite sequence number would this happen?

Need larger sequence number space!

0, 1, 2, 3, 4, 5, 0, 1 …

- In this example, seq number should span at least [0, 5]
- Or, window size should be limited

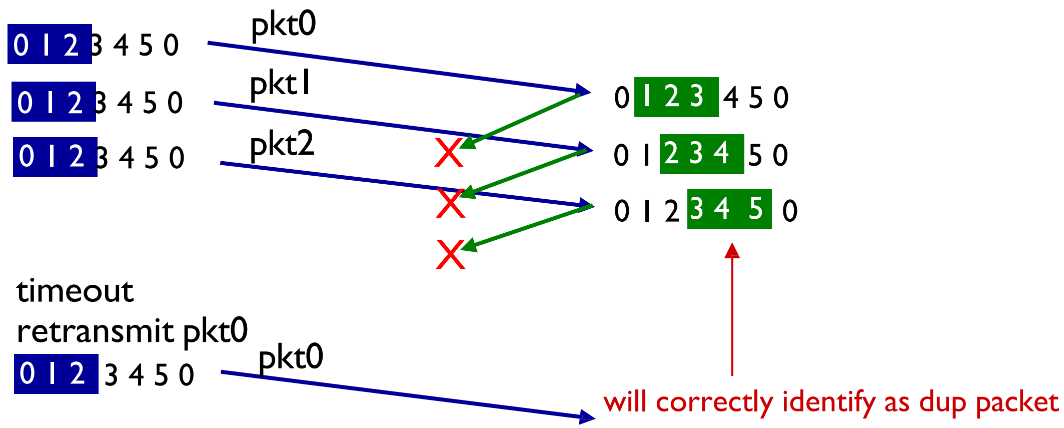
0, 1, 2, 3, 0, 1, 2, 3 …

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

Seq no ≥2 x 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

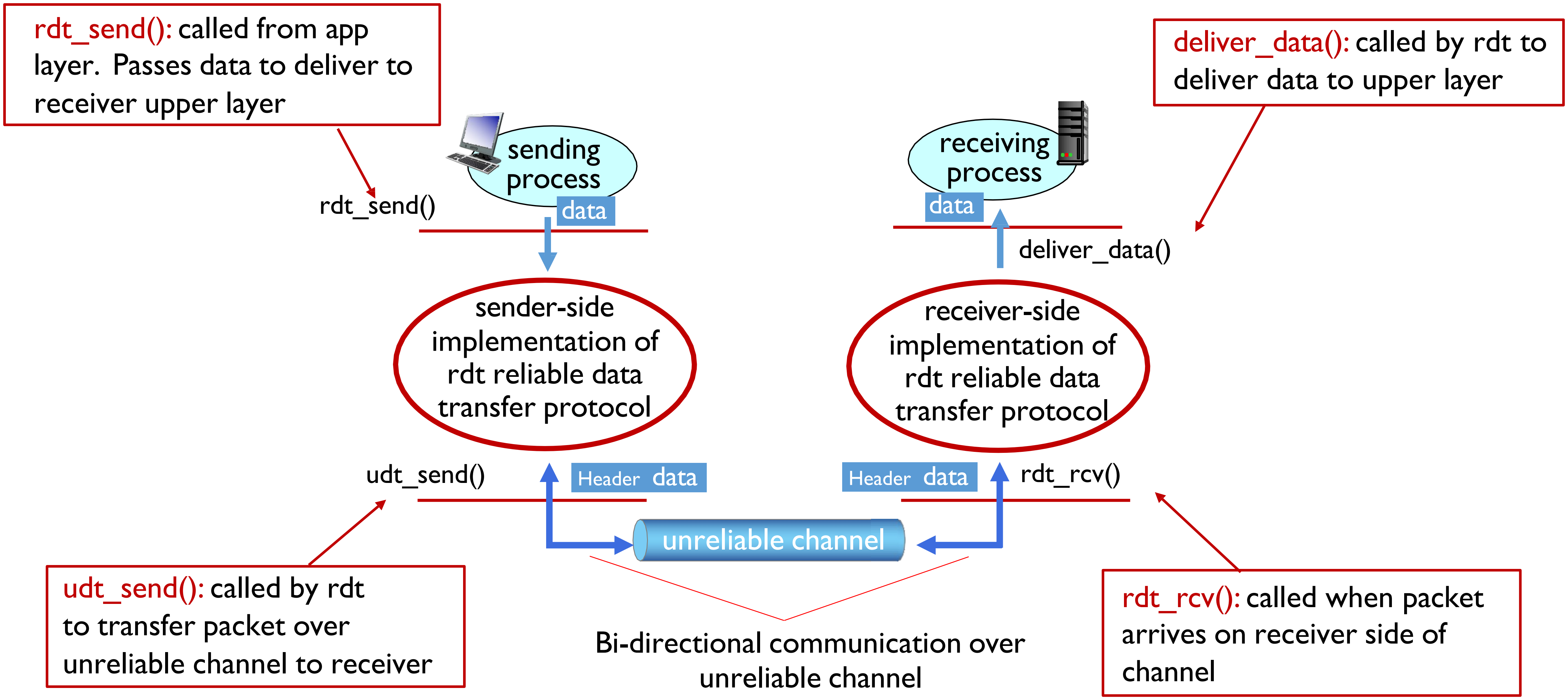
Backup Slides

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

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

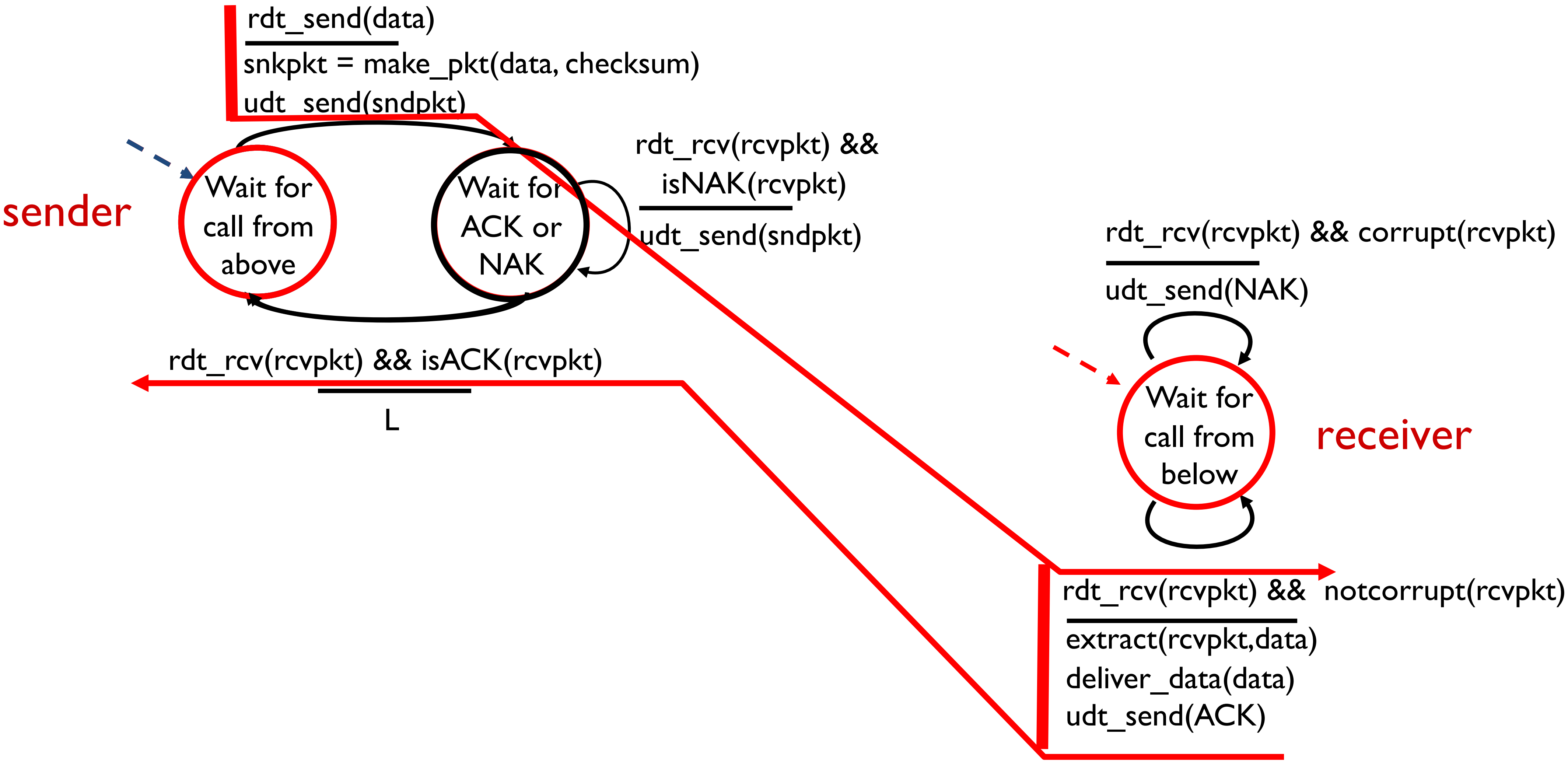
udt_send(): called by rdt to transfer packet over unreliable channel to receiver

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

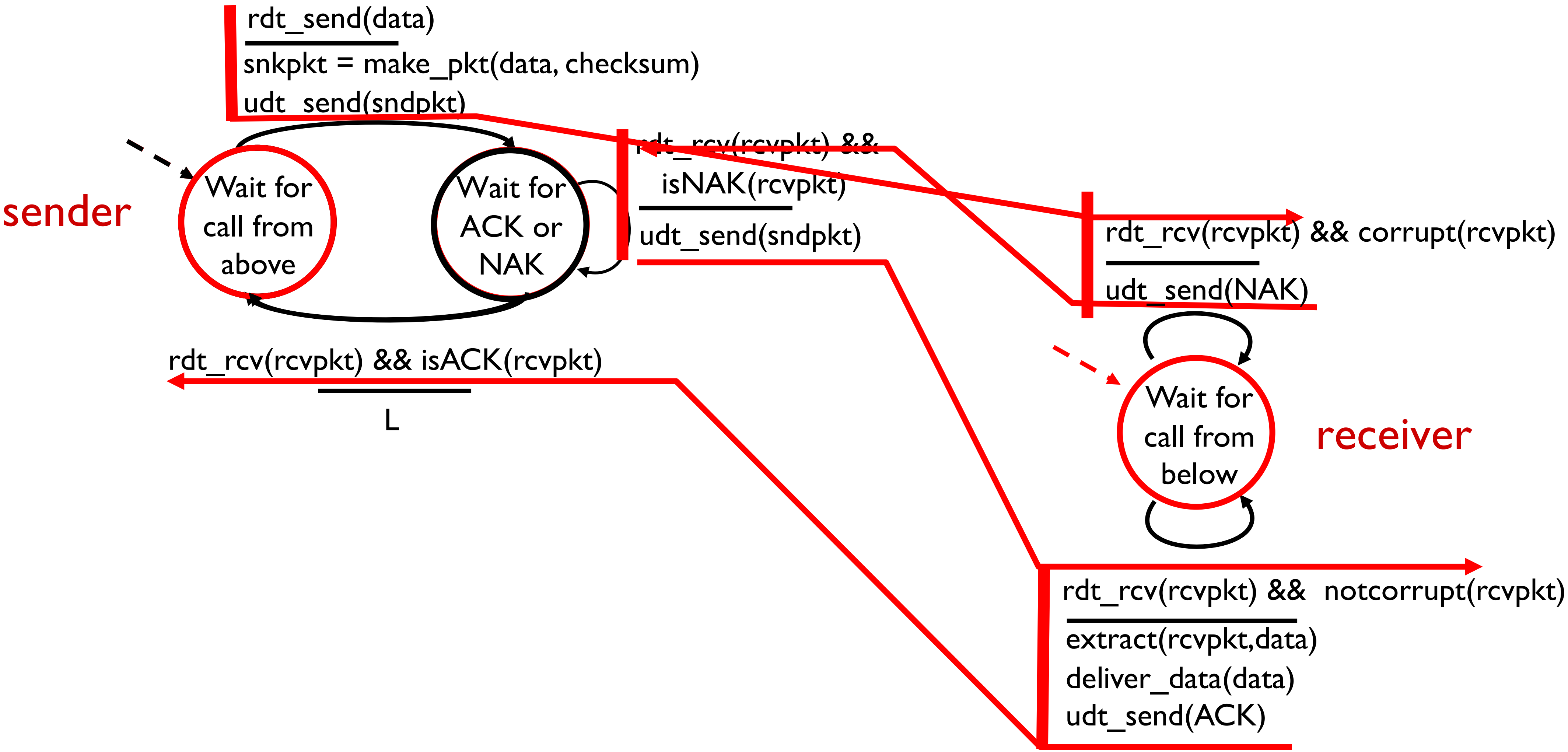
Bi-directional communication over unreliable channel

sending process
data
rdt_send()
sender-side implementation of rdt reliable data transfer protocol
udt_send()
Header data
unreliable channel
receiving process
data
deliver_data()
receiver-side implementation of rdt reliable data transfer protocol
Header data
rdt_rcv()

rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



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

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?

Acknowledgements

Slides are adopted from Kurose’ Computer Networking Slides