Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Reliable data transfer

- Lead the list of the top 10 important problems in computer networking

- Not only in transport layer, but also in application layer and link layer

- We study this problem in the transport layer
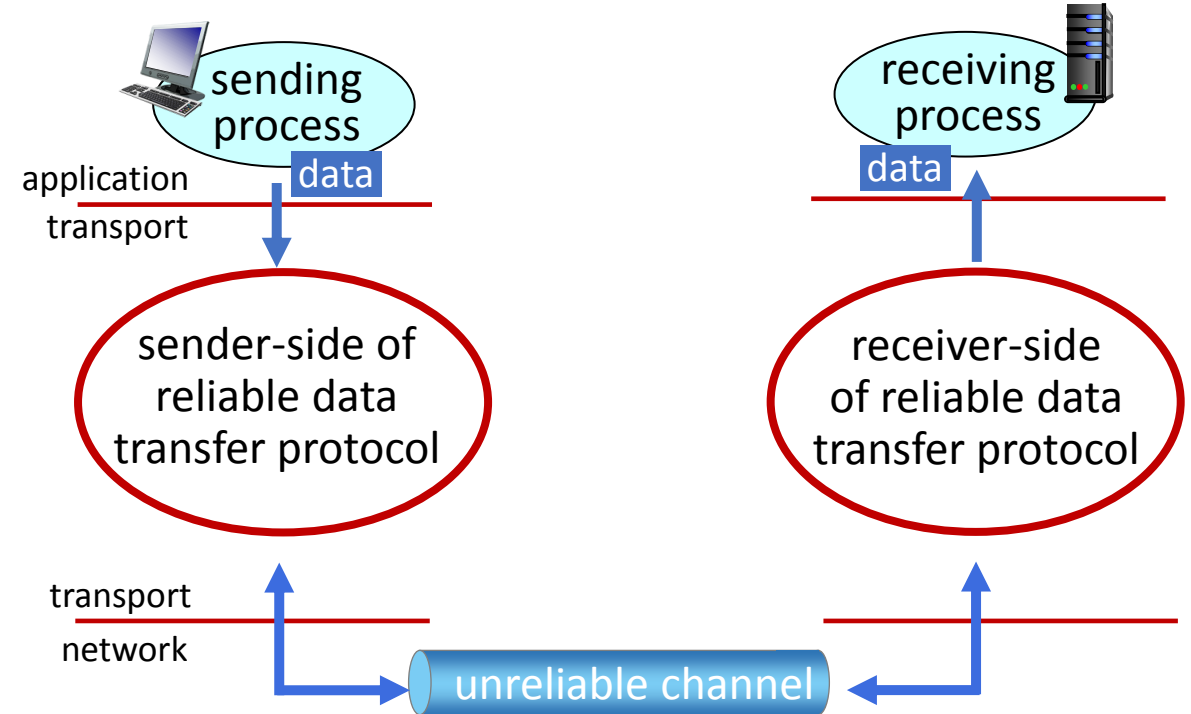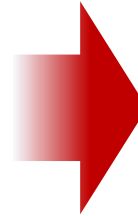
Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

reliable service *abstraction*

reliable service *implementation*

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

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*
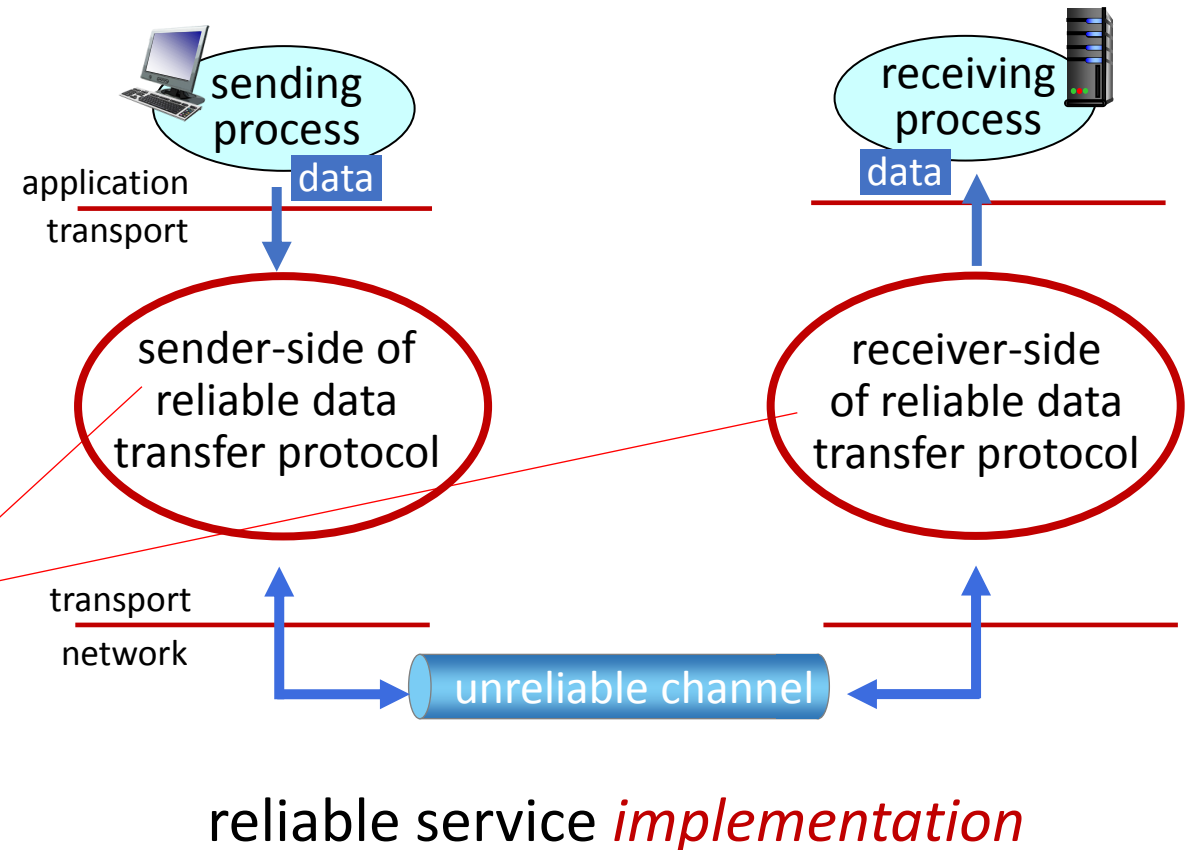
Principles of reliable data transfer

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

sending process

data

application
transport

sender-side of reliable data transfer protocol

transport
network

reliable service *implementation*
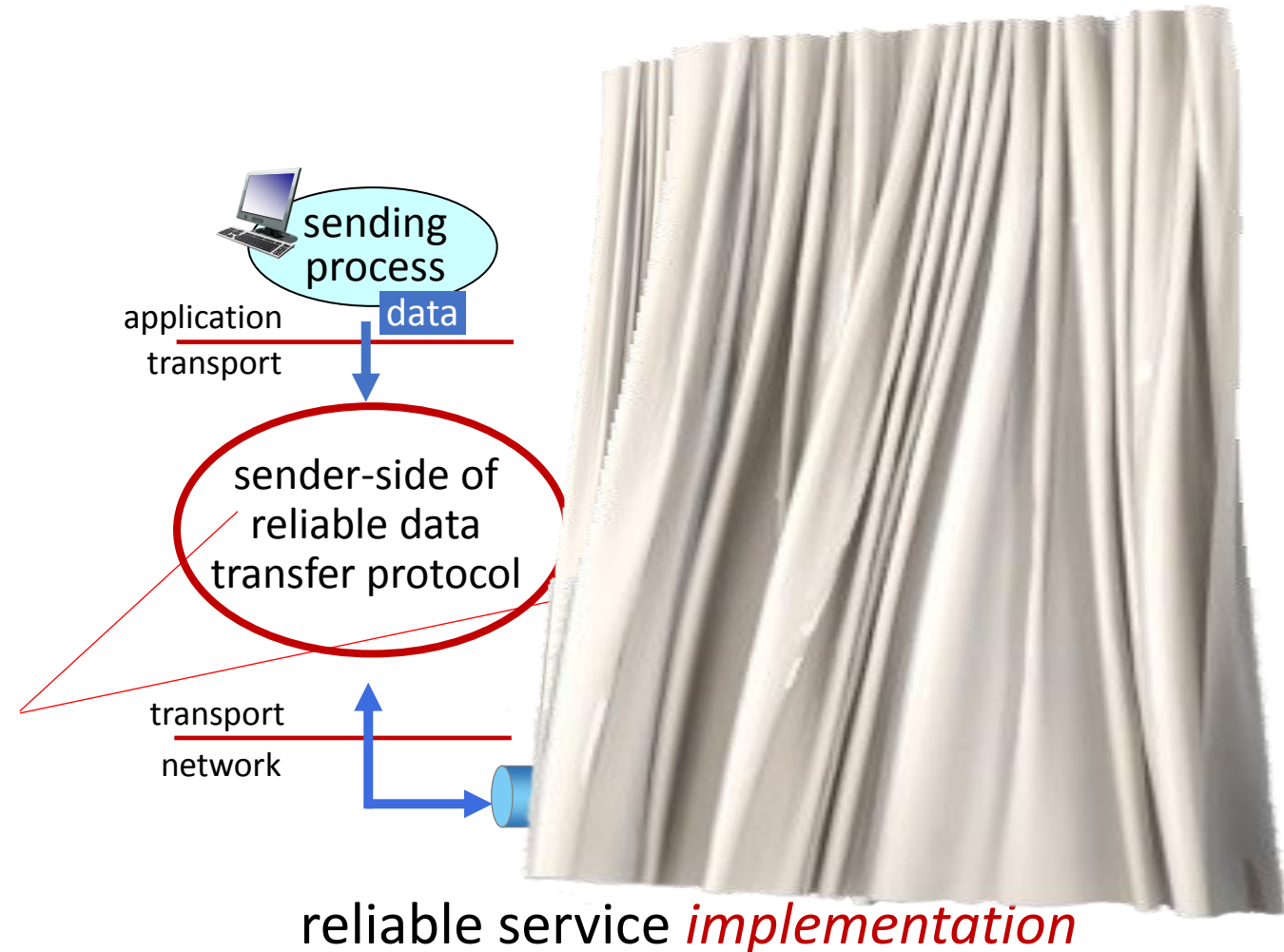
Reliable data transfer protocol (rdt): interfaces

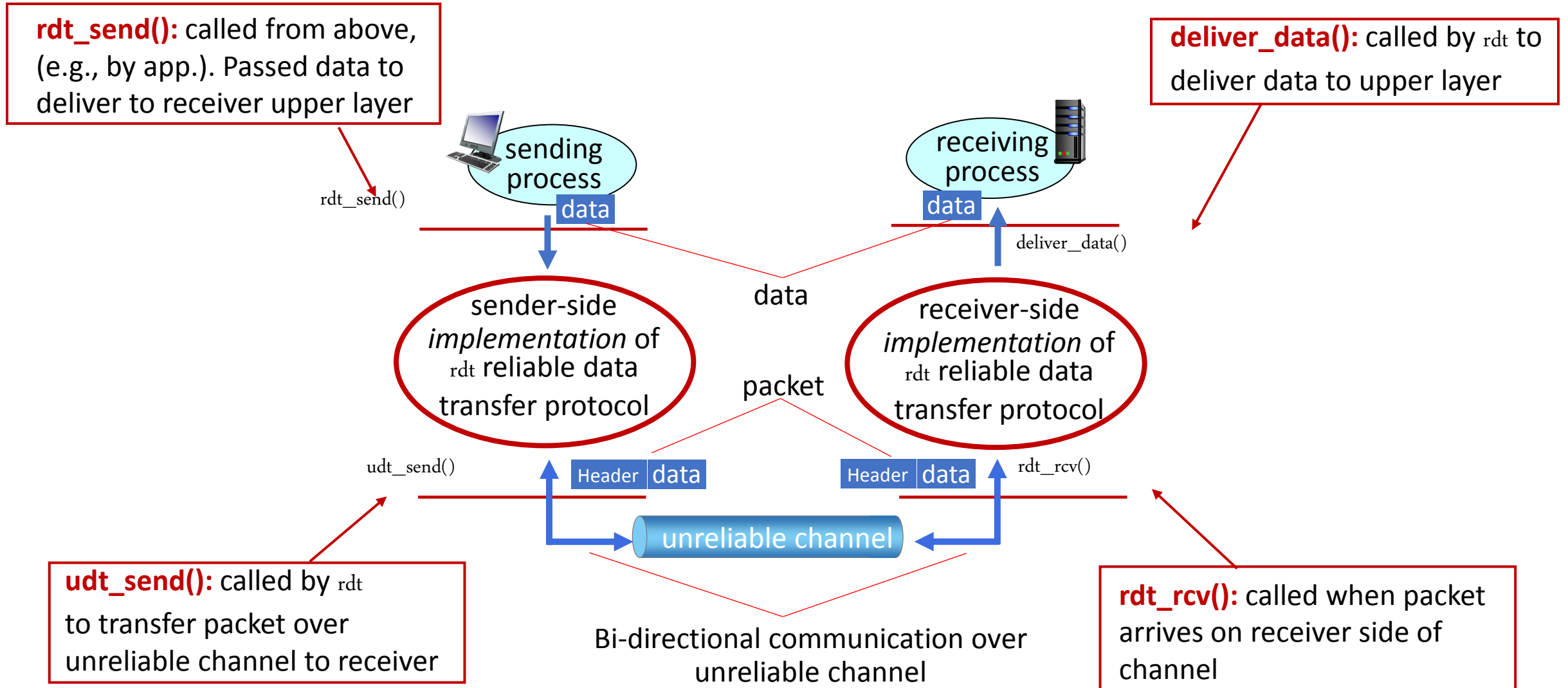**rdt_send():** called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

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

sending process

receiving process

rdt__send()

data

data

deliver__data()

sender-side *implementation* of rdt reliable data transfer protocol

data

receiver-side *implementation* of rdt reliable data transfer protocol

packet

udt_send()

Header | data

Header | data

rdt__rcv()

unreliable channel

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

Bi-directional communication over unreliable channel

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

Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)

- consider only unidirectional data transfer
  - but control info will flow in both directions!

- use finite state machines (FSM) to specify sender, receiver

$$\frac{\text{event causing state transition}}{\text{actions taken on state transition}}$$

state: when in this "state"
next state uniquely
determined by next
event

state 1

$$\frac{\text{event}}{\text{actions}}$$

state 2
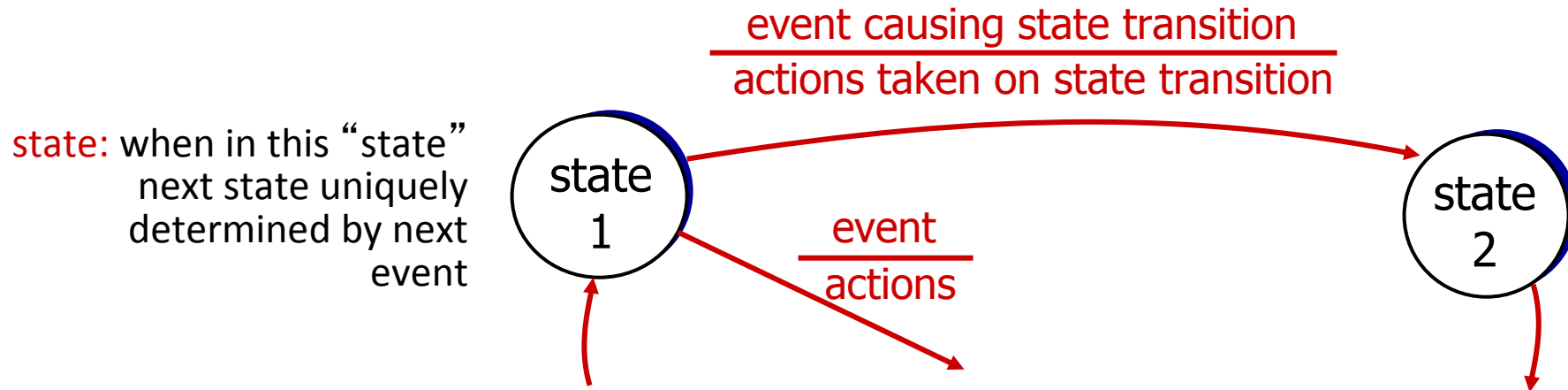
rdt1.0: reliable transfer over a reliable channel

- 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
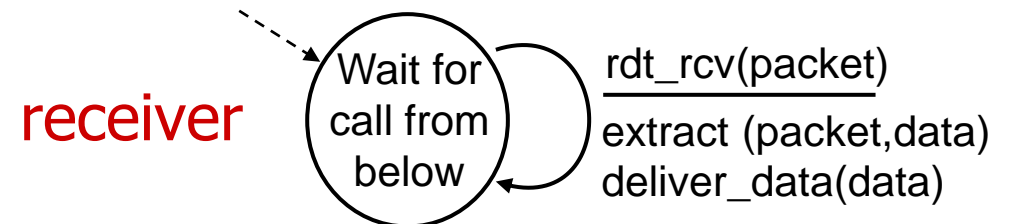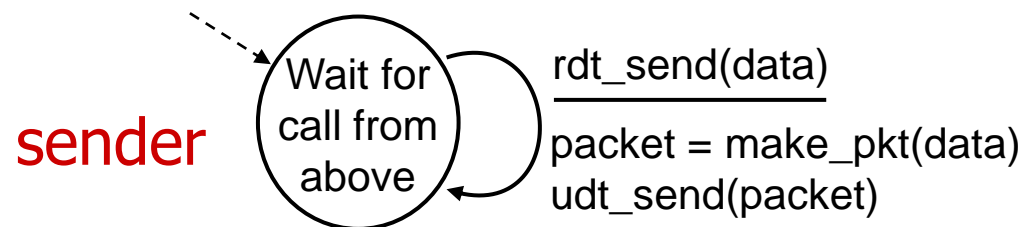
sender

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)

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
- *the* question: how to recover from errors?

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
- *the* question: how to recover from errors?
  - checksum (e.g., Internet checksum) to detect bit errors

  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK

  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors

  - sender *retransmits* pkt on receipt of NAK

ARQ: Automatic Repeat Request

stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications

sender

rdt_send(data)
‾‾‾‾‾‾‾‾‾‾‾‾
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾‾
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾‾
Λ

rdt2.0: FSM specifications

sender

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

(Wait for call from above)

(Wait for ACK or NAK)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

receiver

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

(Wait for call from below)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: FSM specification

rdt_send(data)
——————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
——————
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)
——————
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
——————
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
——————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender
  ▪ that's why we need a protocol!

rdt2.0: FSM specification

sender

rdt_send(data)
——————————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

$\Big($ Wait for call from above $\Big)$ → $\Big($ Wait for ACK or NAK $\Big)$

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
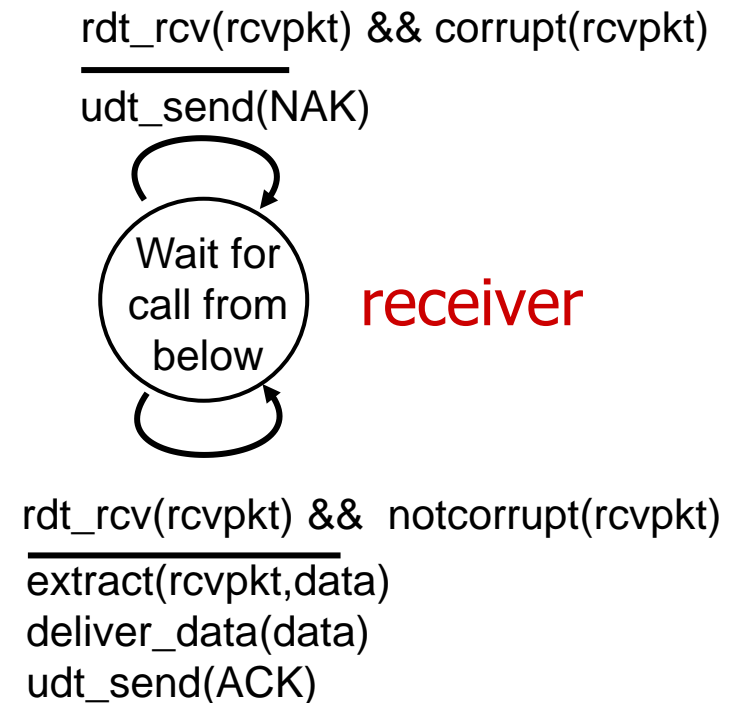——————————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
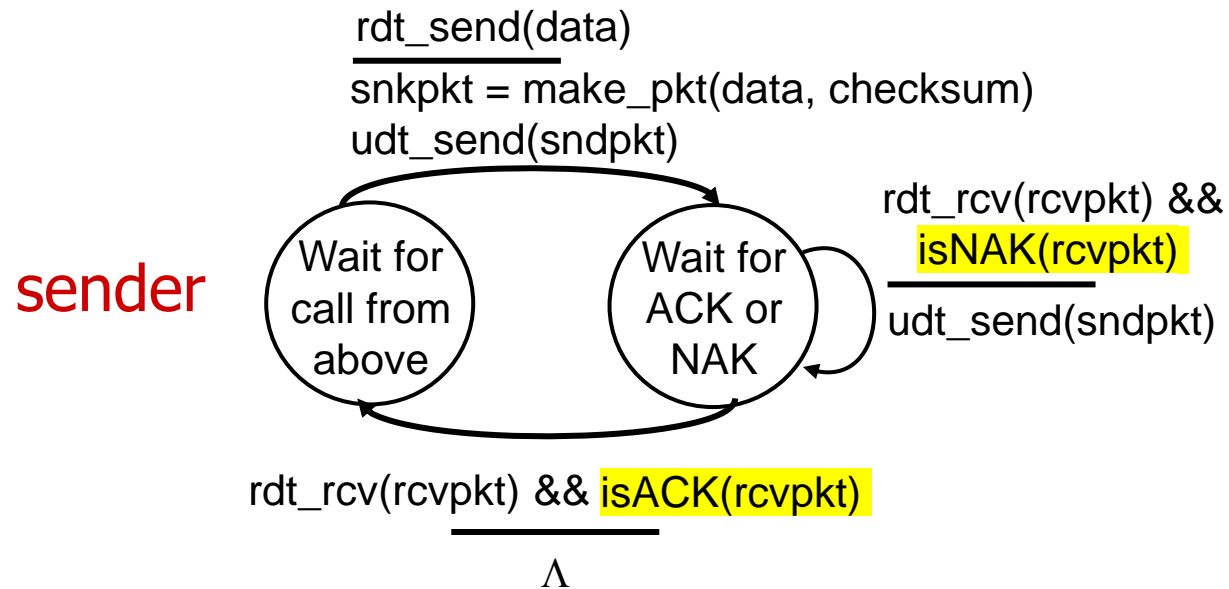——————————————
Λ

Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender
- that's why we need a protocol!

rdt2.0: operation with no errors

rdt2.0: operation with no errors

rdt_send(data)
---
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
---
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
---
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
---
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors

rdt_send(data)
<u> </u>
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
<u> </u>
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
<u> </u>
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
<u> </u>
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
<u> </u>
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

rdt2.0: corrupted packet scenario

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

rdt_send(data)
———————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
———————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
———————
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
———————
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
———————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

sender

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

receiver

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

rdt_send(data)
—————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
—————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
—————
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
—————
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
—————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

sender

rdt_send(data)
—————————————————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
—————————————————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
—————————————————
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
—————————————————
udt_send(NAK)

receiver

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
—————————————————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

rdt_send(data)
___
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
___
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
___
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
___
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
___
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

sender

rdt_send(data)
——————————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for
call from
above

Wait for
ACK or
NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
——————————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
——————————
Λ

receiver

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
——————————
udt_send(NAK)

Wait for
call from
below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
——————————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

sender

rdt_send(data)
‾‾‾‾‾‾‾‾‾‾‾
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
‾‾‾‾‾‾‾‾‾‾‾
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

rdt_send(data)
—————————
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
—————————
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
—————————
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
—————————
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
—————————
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs

rdt_send(data)
_____

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

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))
_____

udt_send(sndpkt)

Wait for
call 0 from
above

Wait for
ACK or
NAK 0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt) &&
isACK(rcvpkt)
_____

Λ

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
_____

Λ

Wait for
ACK or
NAK 1

Wait for
call 1 from
above

rdt_rcv(rcvpkt)
&& (corrupt(rcvpkt) ||
isNAK(rcvpkt))
_____

udt_send(sndpkt)

rdt_send(data)
_____

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt2.1: receiver, handling garbled ACK/NAKs

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq0(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
   not corrupt(rcvpkt) &&
   has_seq1(rcvpkt)
_____
 sndpkt = make_pkt(ACK, chksum)
 udt_send(sndpkt)

(**Wait for 0 from below**)   (**Wait for 1 from below**)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
   not corrupt(rcvpkt) &&
   has_seq0(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice. Why?
- 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 #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only

- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed

- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

rdt2.2: sender, receiver fragments

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

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
  isACK(rcvpkt,1))
_____
udt_send(sndpkt)

Wait for
call 0 from
above

Wait for
ACK
0

**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)

rdt3.0: channels with errors *and* loss

New channel assumption: underlying channel can also *lose* packets (data, ACKs)

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

Q: How do *humans* handle lost sender-to-receiver words in conversation?

rdt3.0: channels with errors *and* loss

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
- use countdown timer to interrupt after "reasonable" amount of time

timeout

rdt3.0 sender

rdt_send(data)
_____
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
_____
Λ

Wait for call 0 from above

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

Wait for ACK0

timeout
_____
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
_____
stop_timer

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

timeout
_____
udt_send(sndpkt)
start_timer

Wait for ACK1

Wait for call 1 from above

rdt_rcv(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
_____
Λ

rdt_send(data)
_____
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 in action

(a) no loss
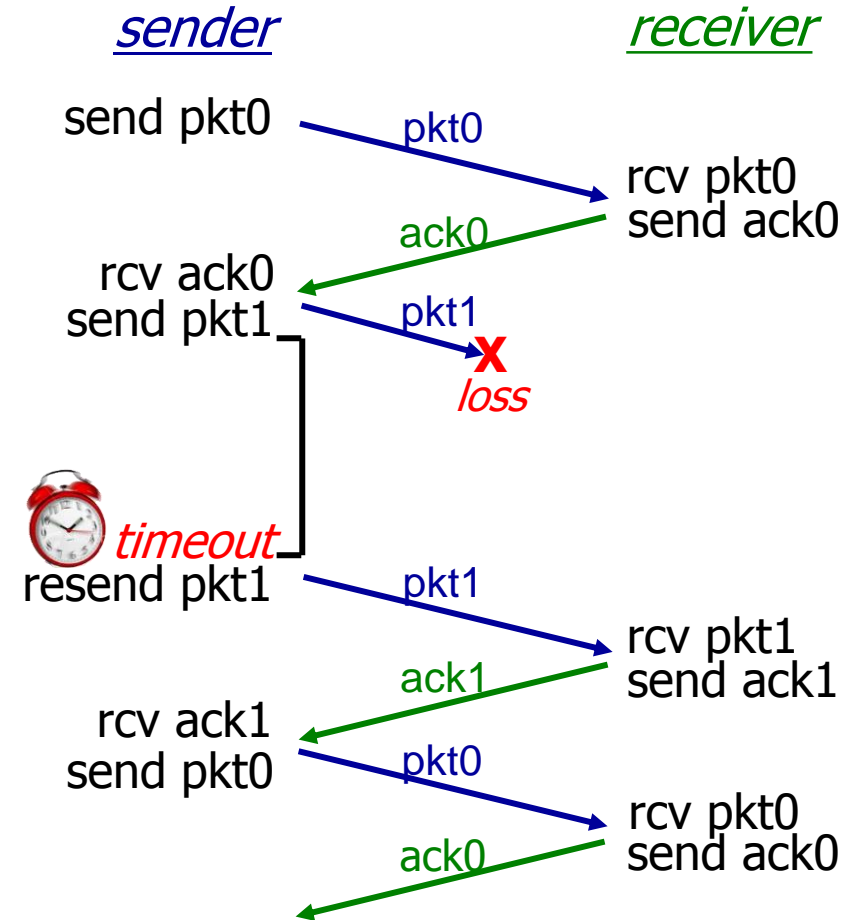
(b) packet loss

rdt3.0 in action
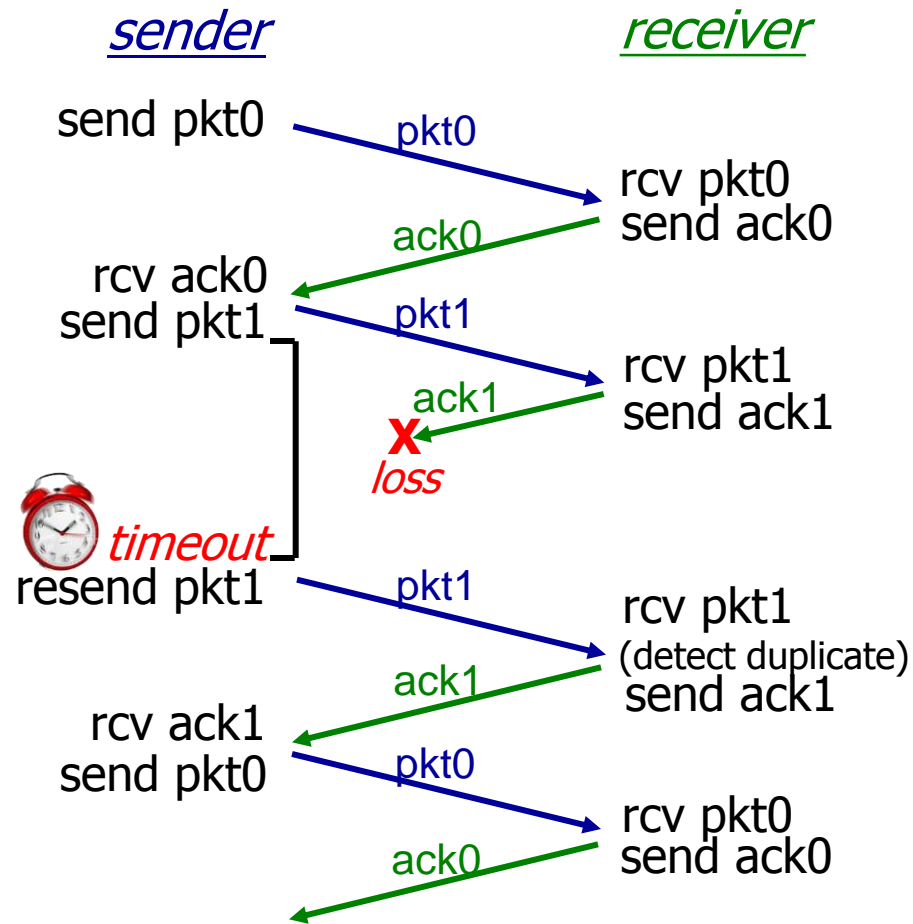
(c) ACK loss
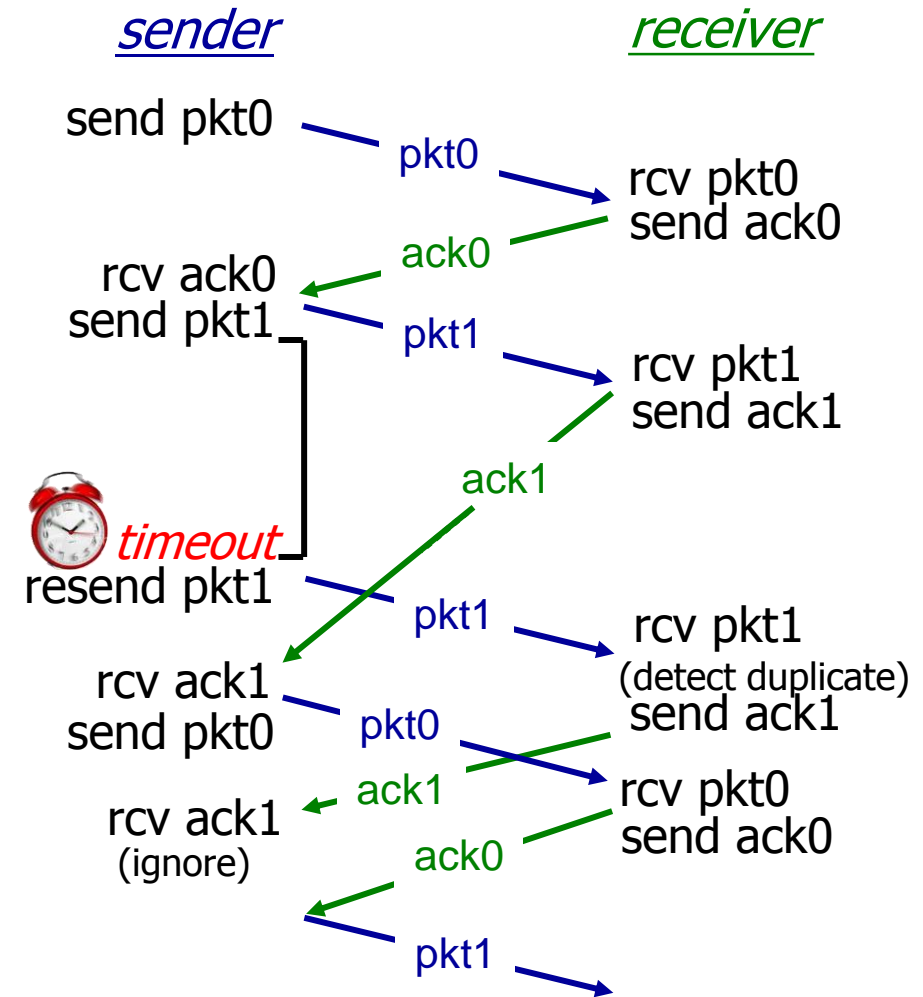
(d) premature timeout/ delayed ACK

Performance of rdt3.0 (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:

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

rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0

RTT

first packet bit arrives

last packet bit arrives, send ACK

ACK arrives, send next
packet, $t = RTT + L / R$
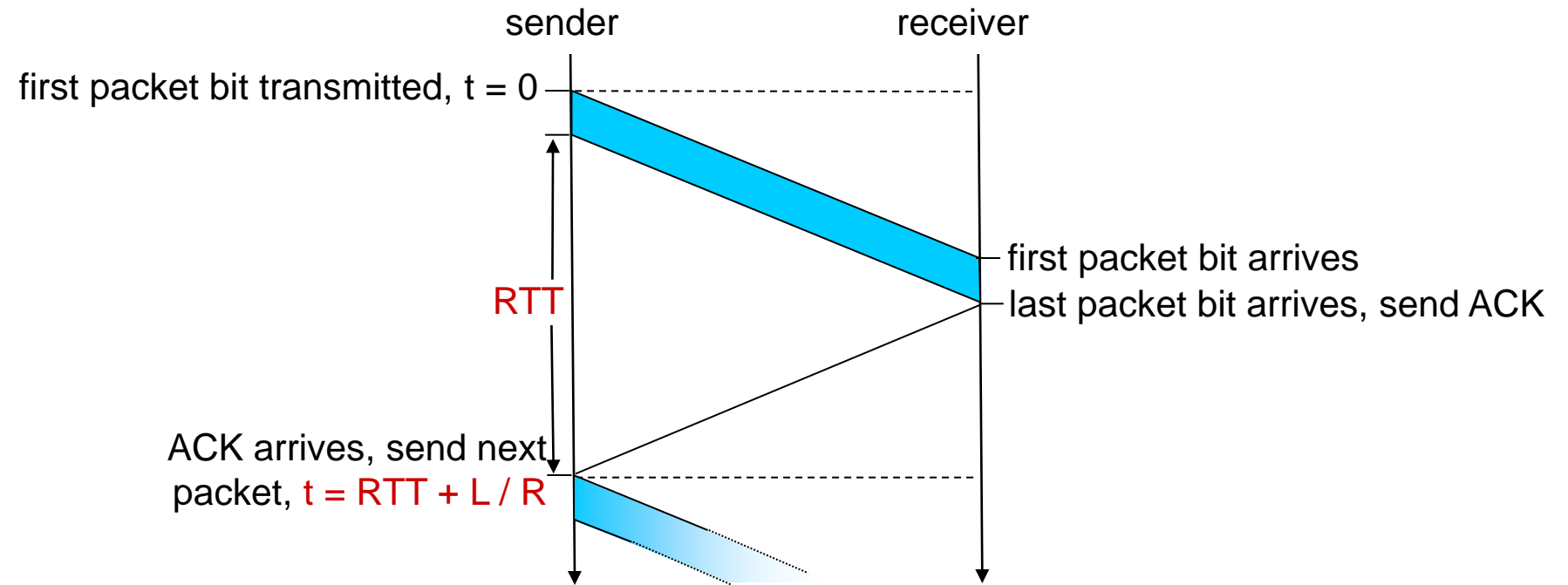
rdt3.0: stop-and-wait operation

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

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

$$= 0.00027$$
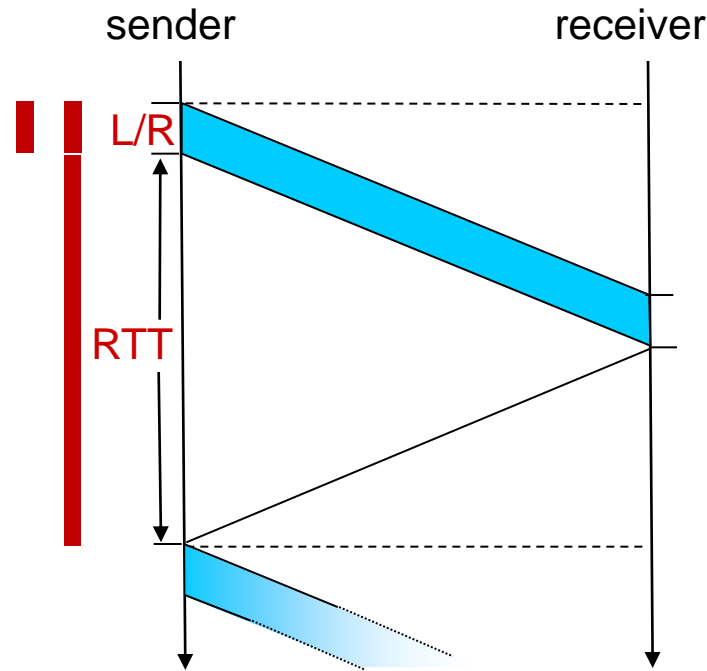
- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

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

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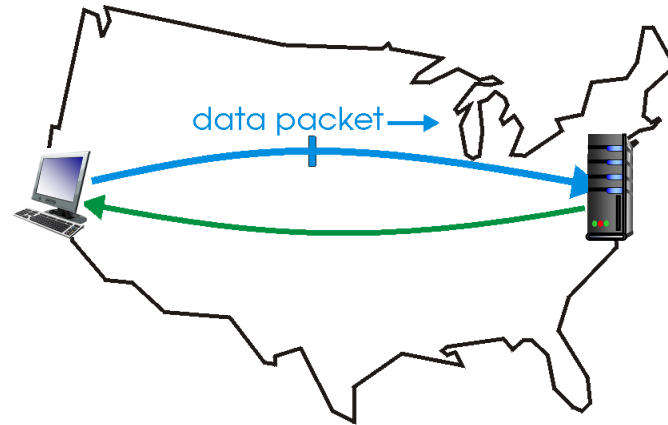
(a) a stop-and-wait protocol in operation

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

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

ACK arrives, send next
packet, t = RTT + L / R
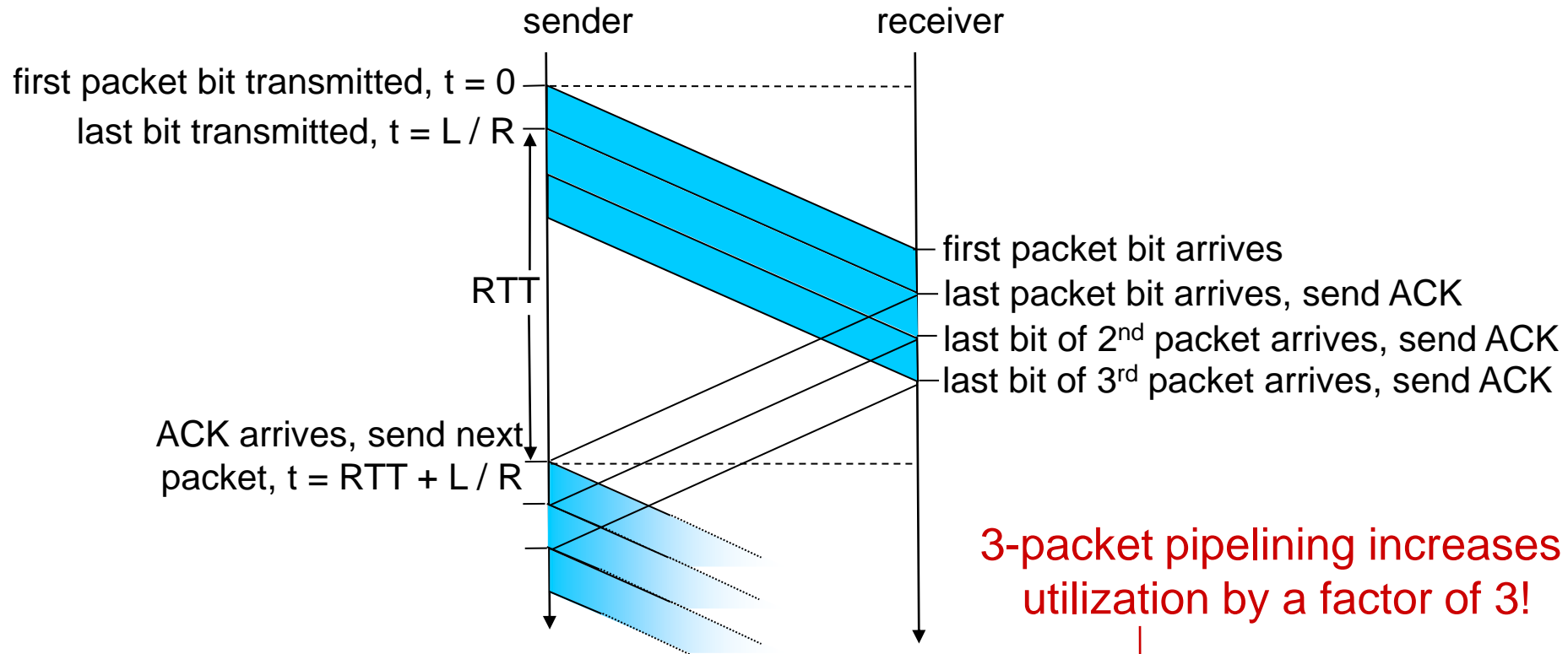
3-packet pipelining increases
utilization by a factor of 3!

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

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header
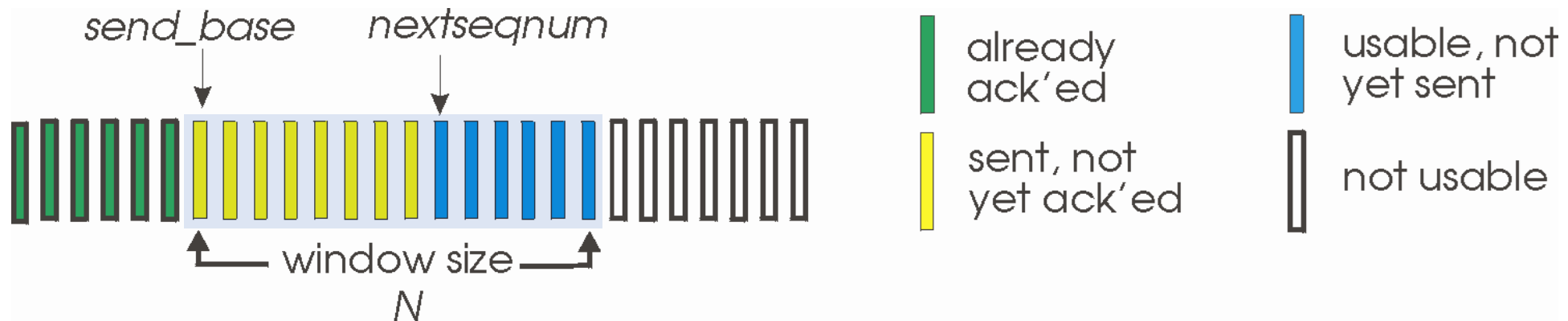
- *cumulative ACK:* ACK(n): ACKs all packets up to, including seq # n
  - on receiving ACK(n): move window forward to begin at $n+1$
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `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:

... rcv_base ...

received and ACKed

Out-of-order: received but not ACKed
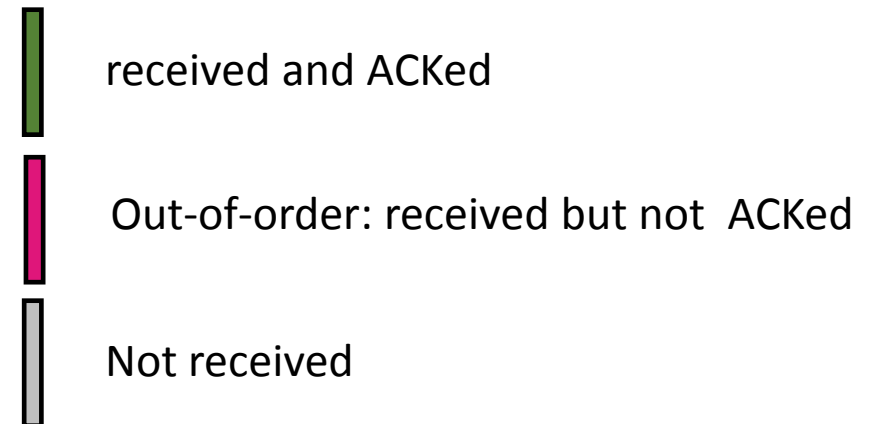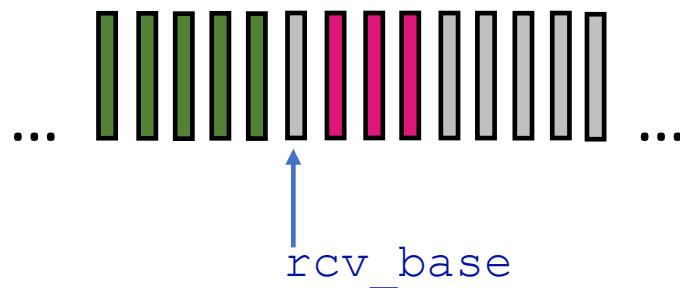
Not received

Go-Back-N in action
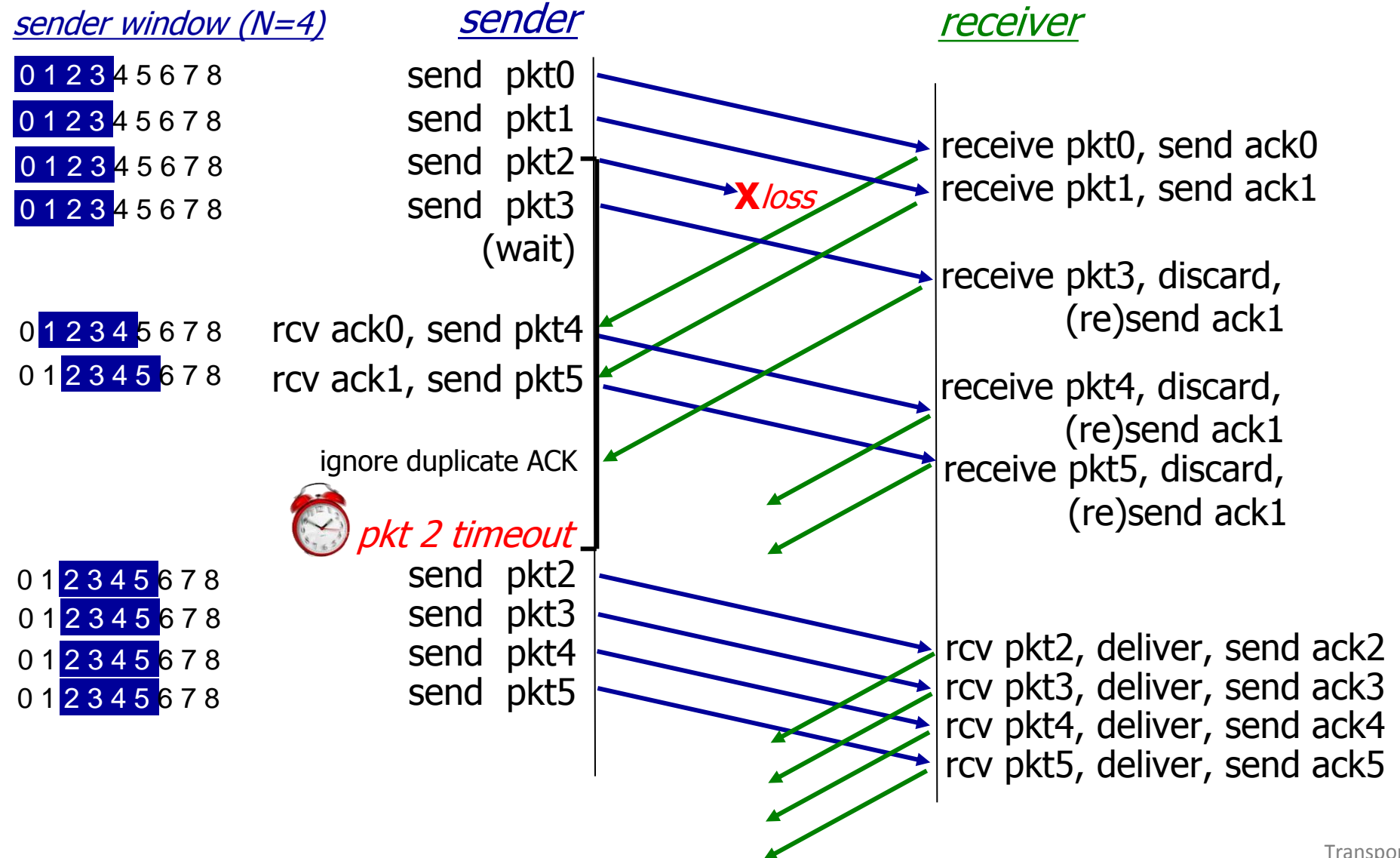
Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer

- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt

- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows

(a) sender view of sequence numbers
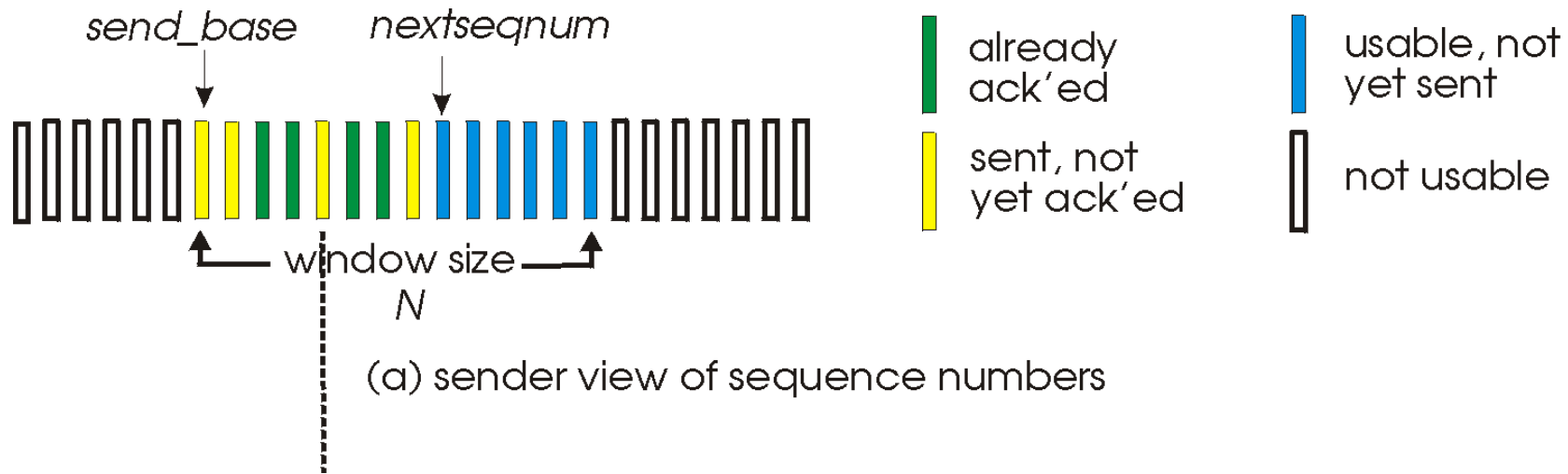
Selective repeat: sender and receiver

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
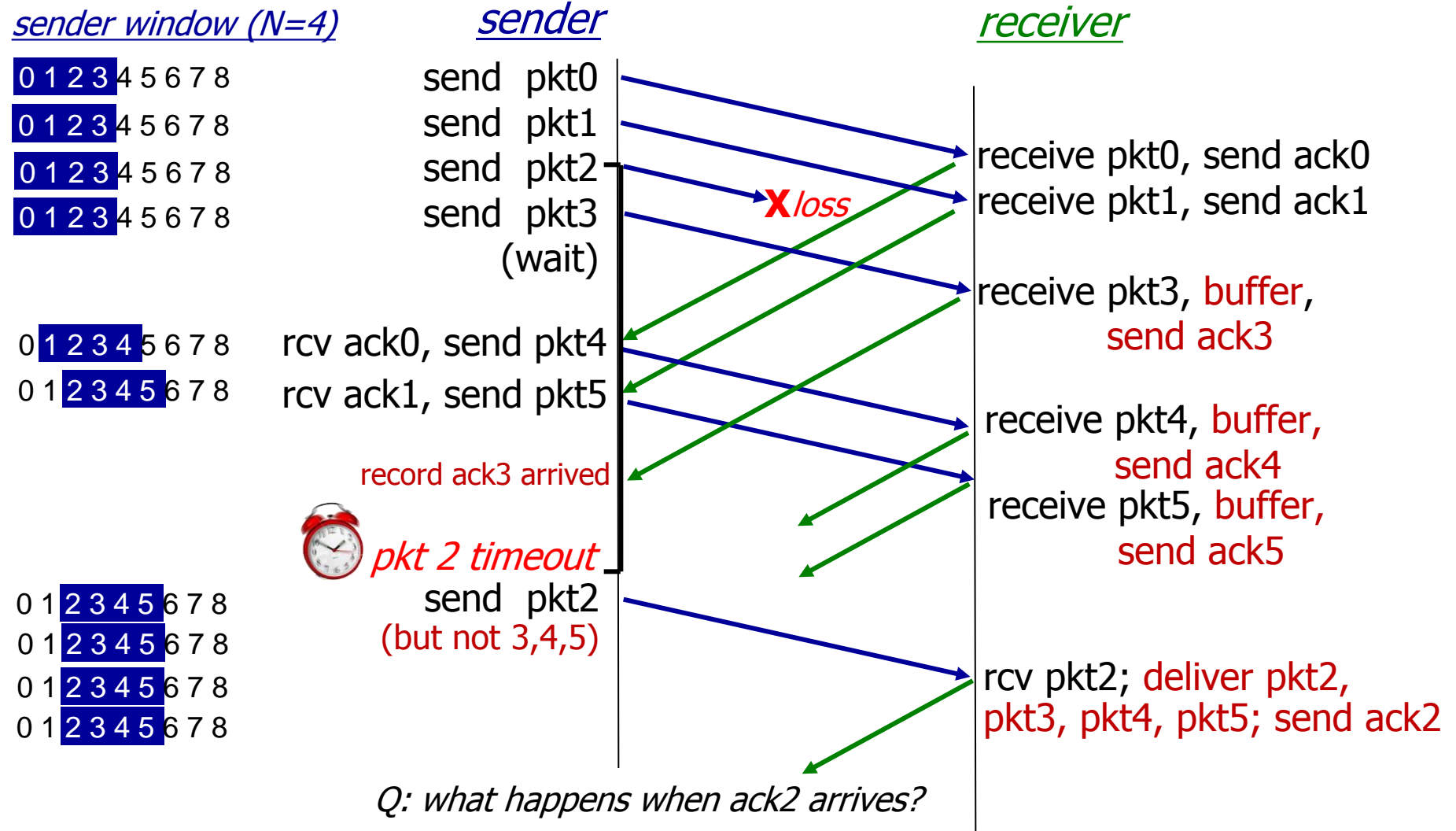
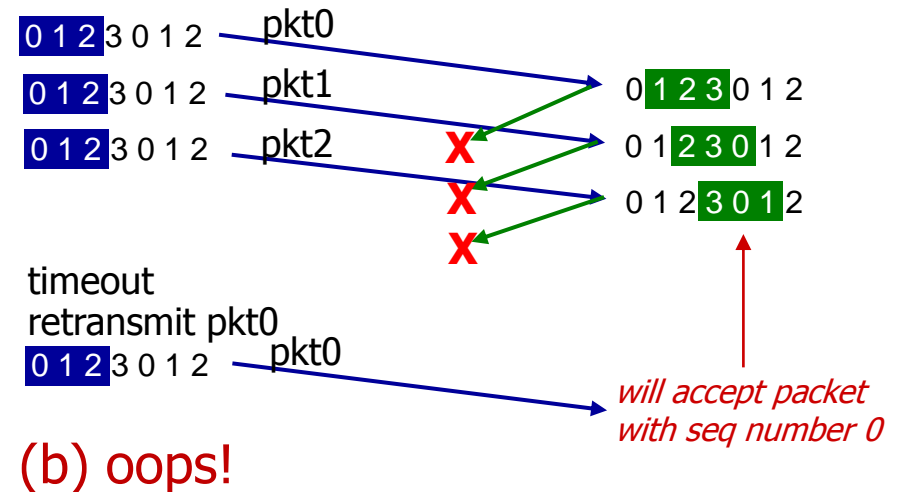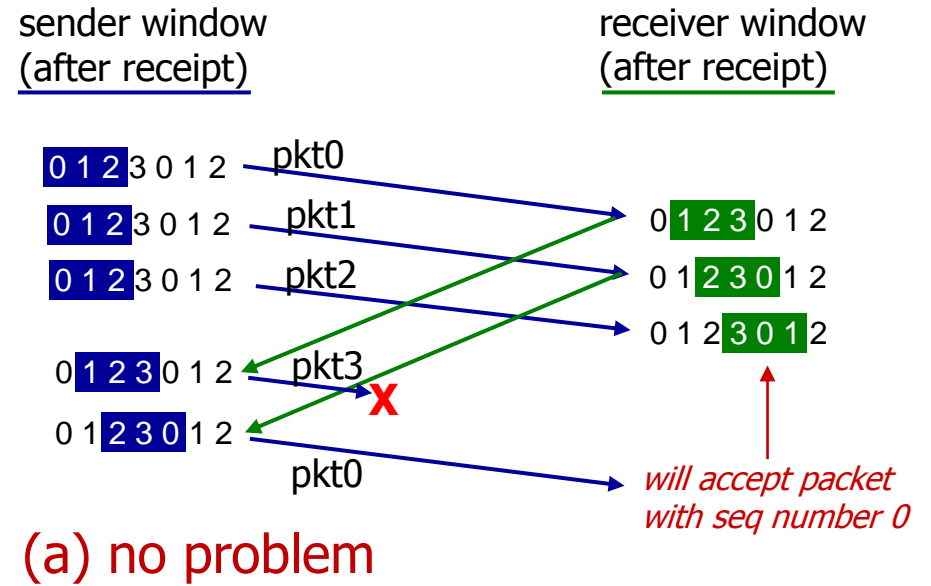### packet n in [rcvbase-N,rcvbase-1]
- ACK(n)

otherwise:
- ignore

Selective Repeat in action

sender window (N=4)

sender

receiver

0 1 2 3 4 5 6 7 8   send pkt0

0 1 2 3 4 5 6 7 8   send pkt1

0 1 2 3 4 5 6 7 8   send pkt2

0 1 2 3 4 5 6 7 8   send pkt3

(wait)

X *loss*

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, buffer,
         send ack3

0 1 2 3 4 5 6 7 8   rcv ack0, send pkt4

0 1 2 3 4 5 6 7 8   rcv ack1, send pkt5

record ack3 arrived

receive pkt4, buffer,
         send ack4
receive pkt5, buffer,
         send ack5

pkt 2 timeout

0 1 2 3 4 5 6 7 8   send pkt2

0 1 2 3 4 5 6 7 8   (but not 3,4,5)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?

Selective repeat: a dilemma!

example:

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

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
X
0 1 2 3 0 1 2
pkt0

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*

(a) no problem

0 1 2 3 0 1 2 pkt0
0 1 2 3 0 1 2 pkt1
0 1 2 3 0 1 2 pkt2   X
X
X

timeout
retransmit pkt0
0 1 2 3 0 1 2 pkt0

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*

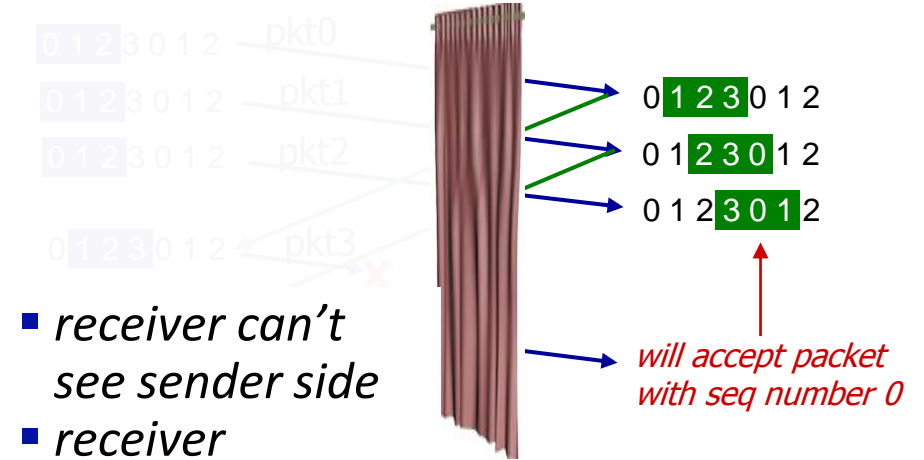## (b) oops!

Selective repeat: a dilemma!

example:

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

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?
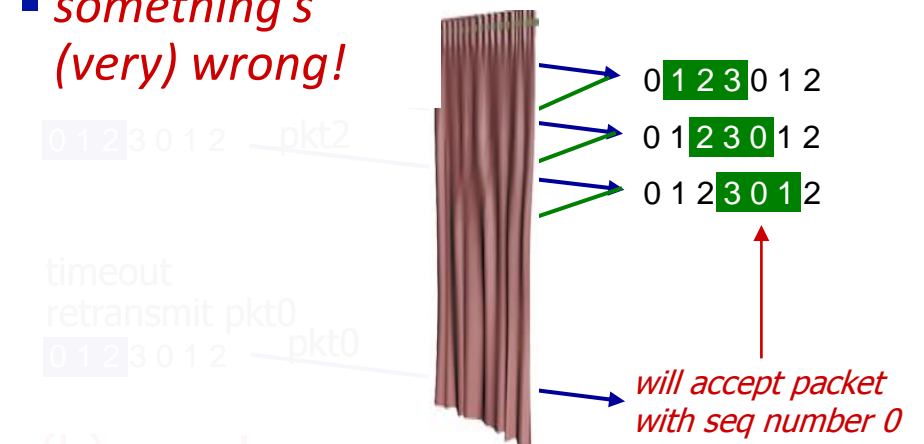
sender window
(after receipt)

receiver 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

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

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

(b) oops!