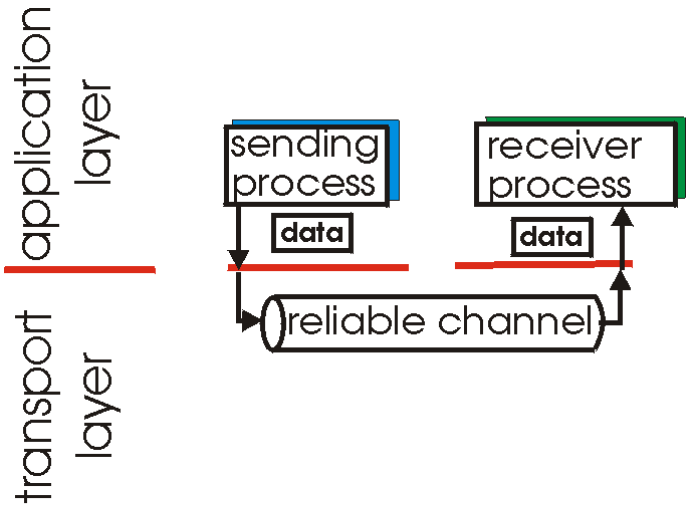
CS348: Computer Networks

Principles of Reliable Data Transfer

Dr. Manas Khatua
Asst. Professor
Dept. of CSE, IIT Guwahati
E-mail: manaskhatua@iitg.ac.in

Principles of Reliable Data Transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

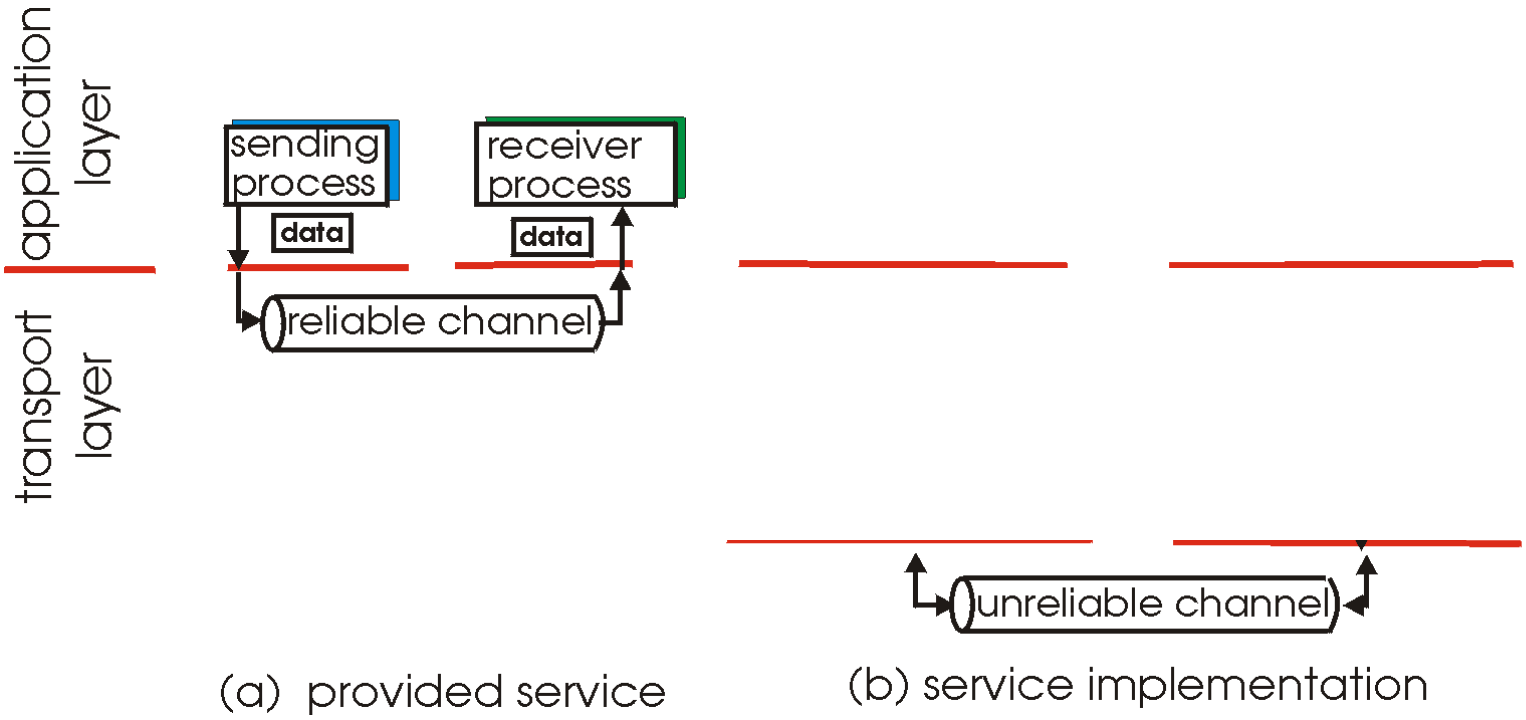


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Cont…

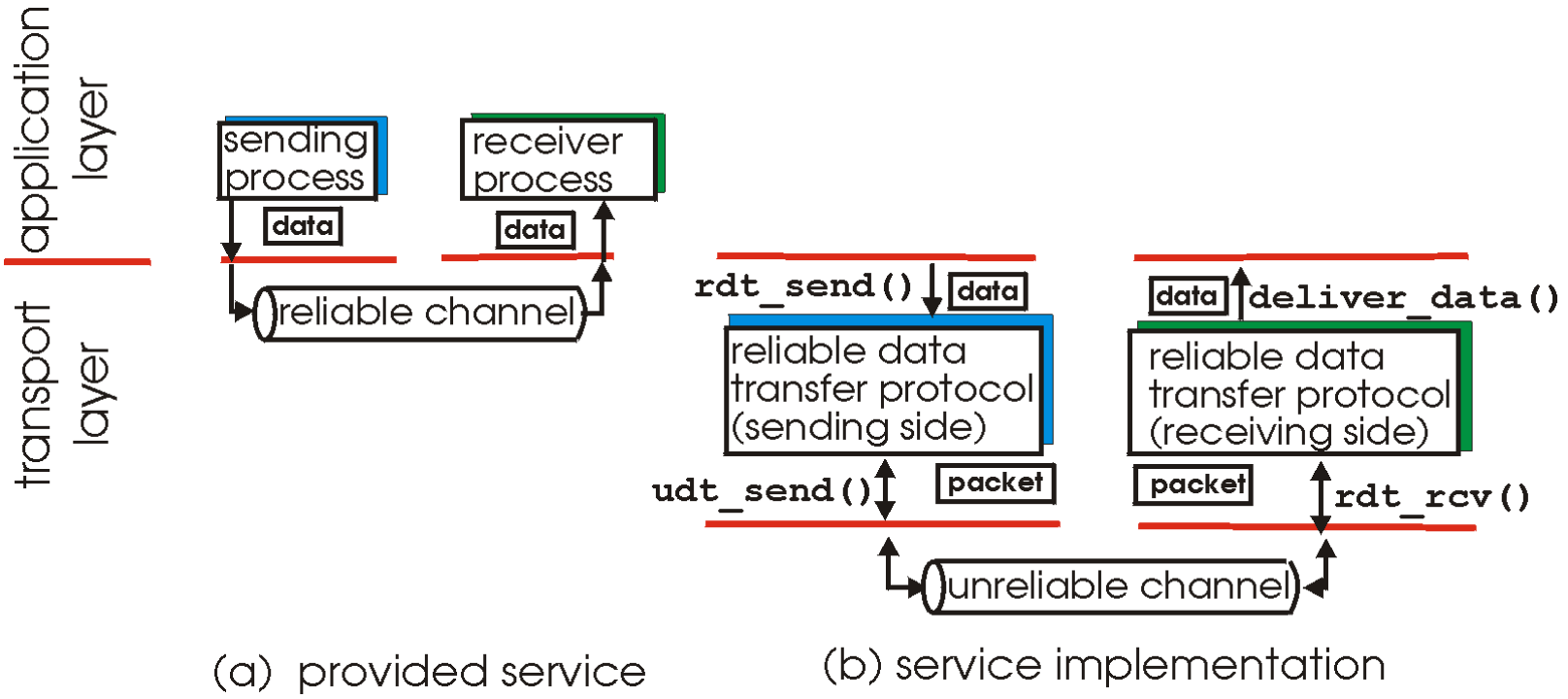
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Cont…

- important in application, transport, link layers
  - top-10 list of important networking topics!

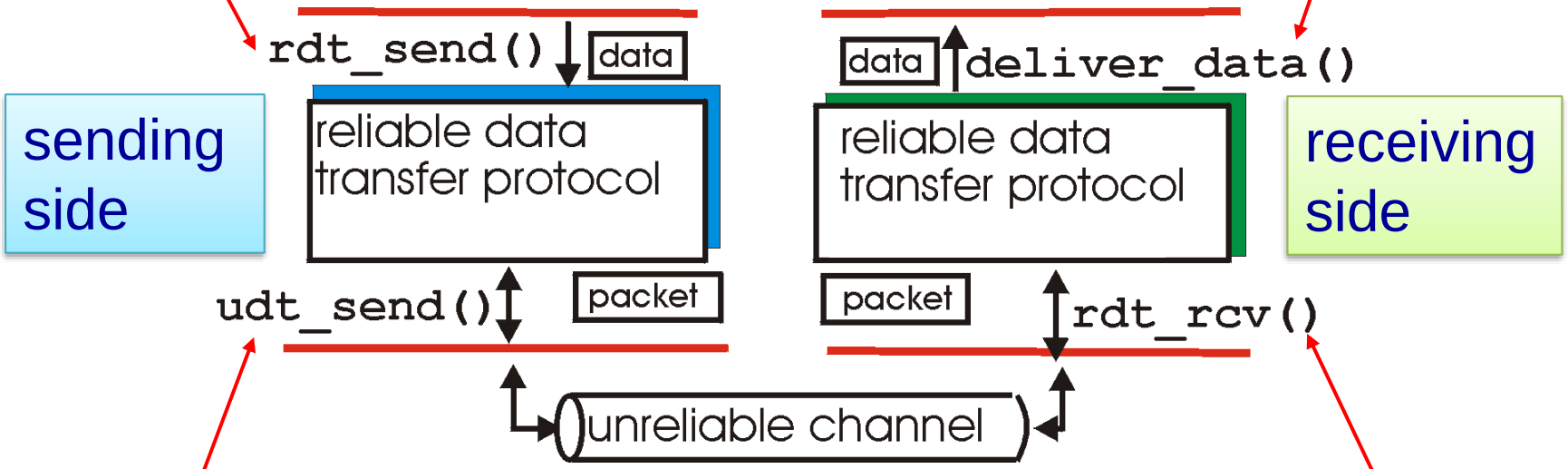


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

RDT: getting started

`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



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

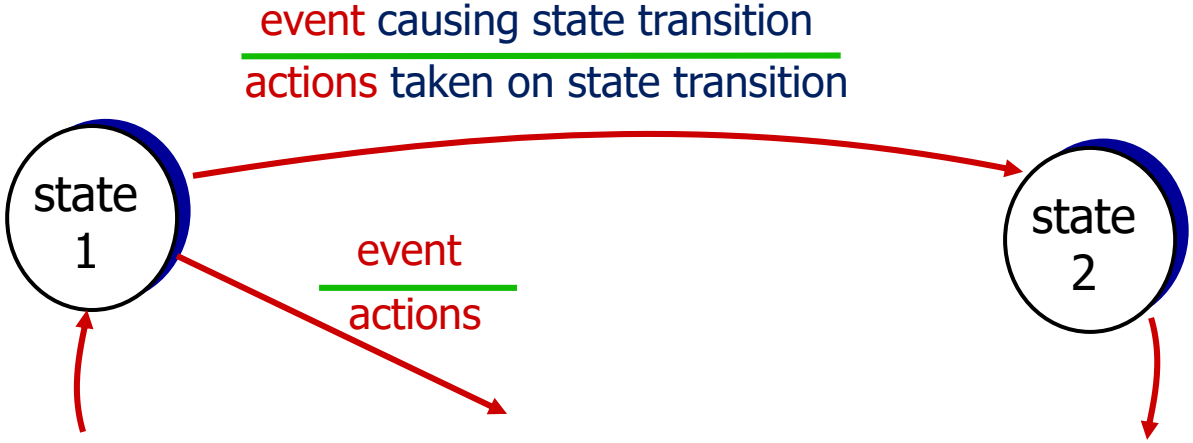
`rdt_rcv()`: called when packet arrives on rcv-side of channel

Finite state machine (FSM)

- use finite state machines (FSM) to specify sender, receiver
- consider only unidirectional data transfer
  - but control info will flow on both directions!

- A FSM or **finite automaton** is a model of behavior composed of states, transitions and actions.
  - A **state** stores information about the past, i.e. it reflects the input changes from the system start to the present moment.
  - A **transition** indicates a **state change** and is described by a **condition/event** that would need to be fulfilled to enable the transition.
  - An **action** is a description of an **activity** that is to be performed at a given moment.

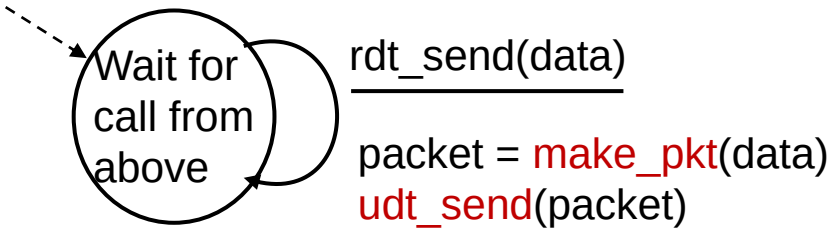
state: when in this "state", next state is uniquely determined by next event



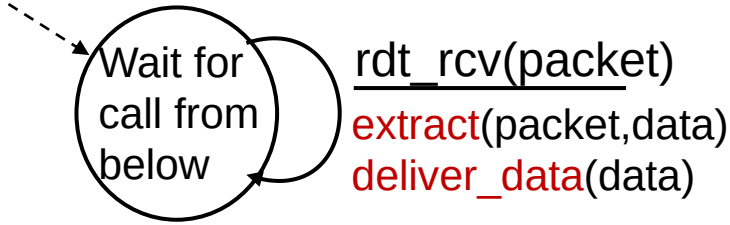
rdt1.0: transfer over a reliable channel

- 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



receiver

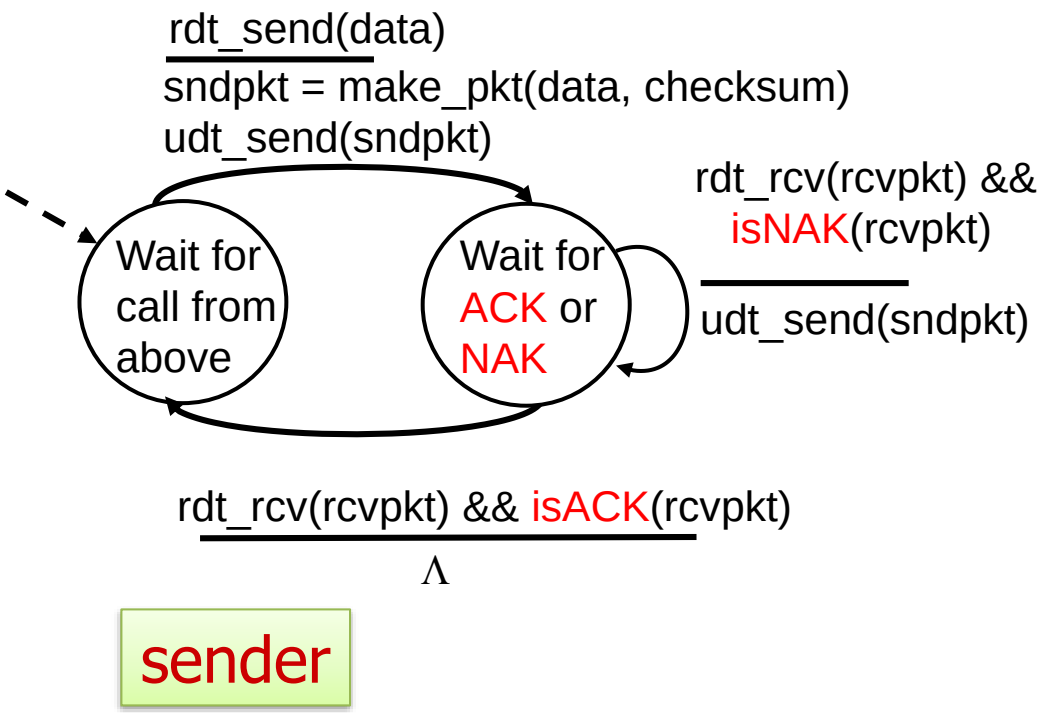
rdt2.0: channel with bit errors

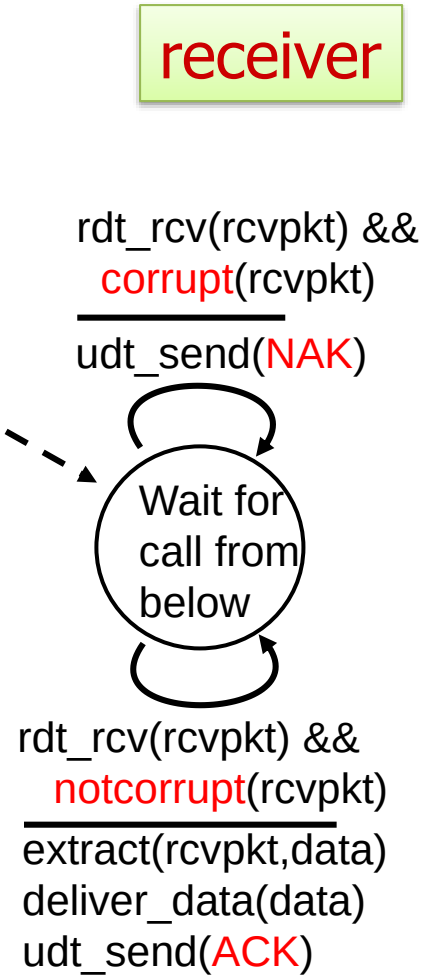
How do humans detect and recover from "errors" during conversation?

- underlying channel may flip bits in packet
  - checksum to detect bit errors at the receiver
  - But, how to recover from error?
    - *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
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - receiver feedback: control msgs (ACK,NAK)
  - retransmission

ARQ (**A**utomatic **R**epeat re**Q**uest protocols)

rdt2.0: FSM specification

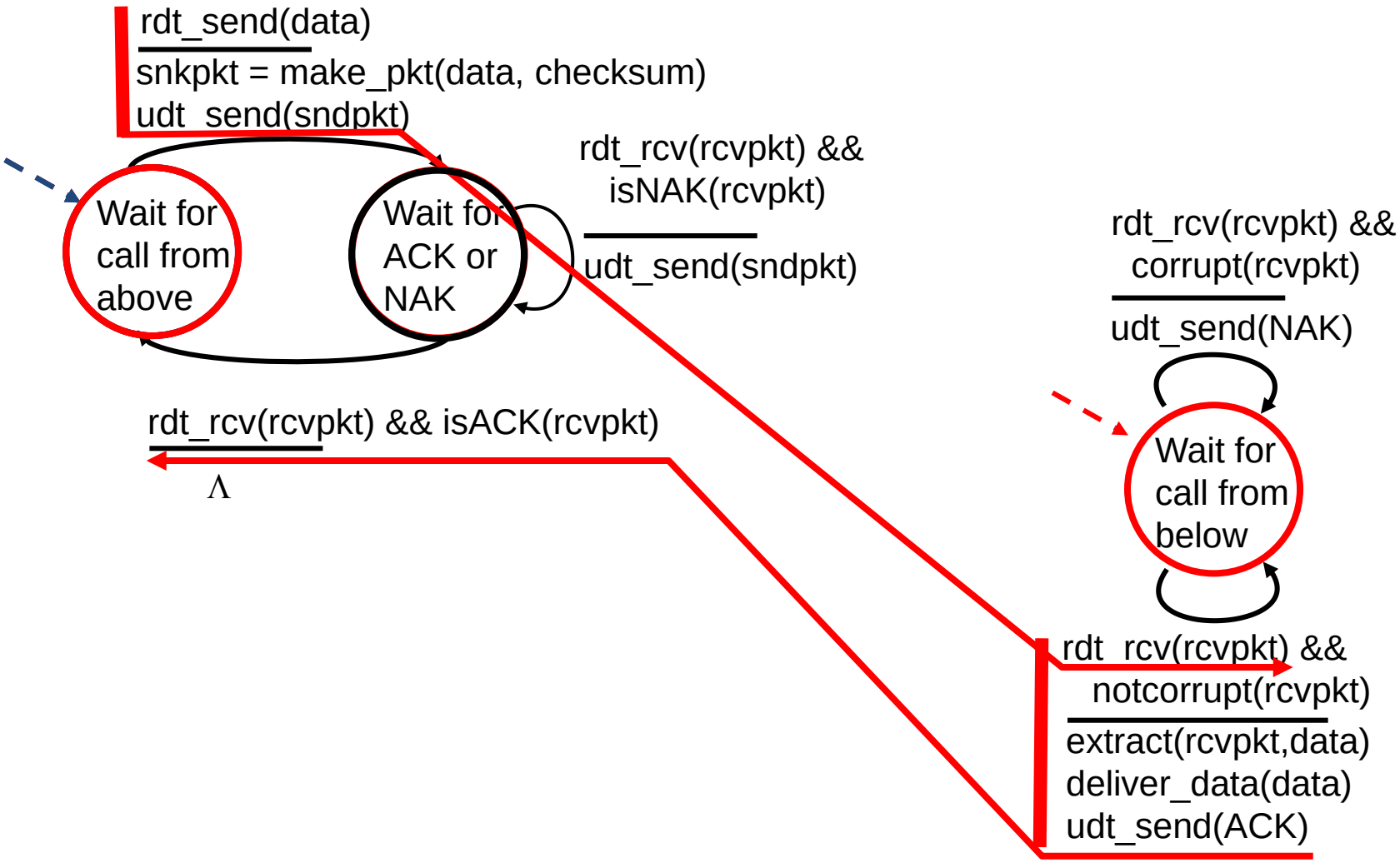




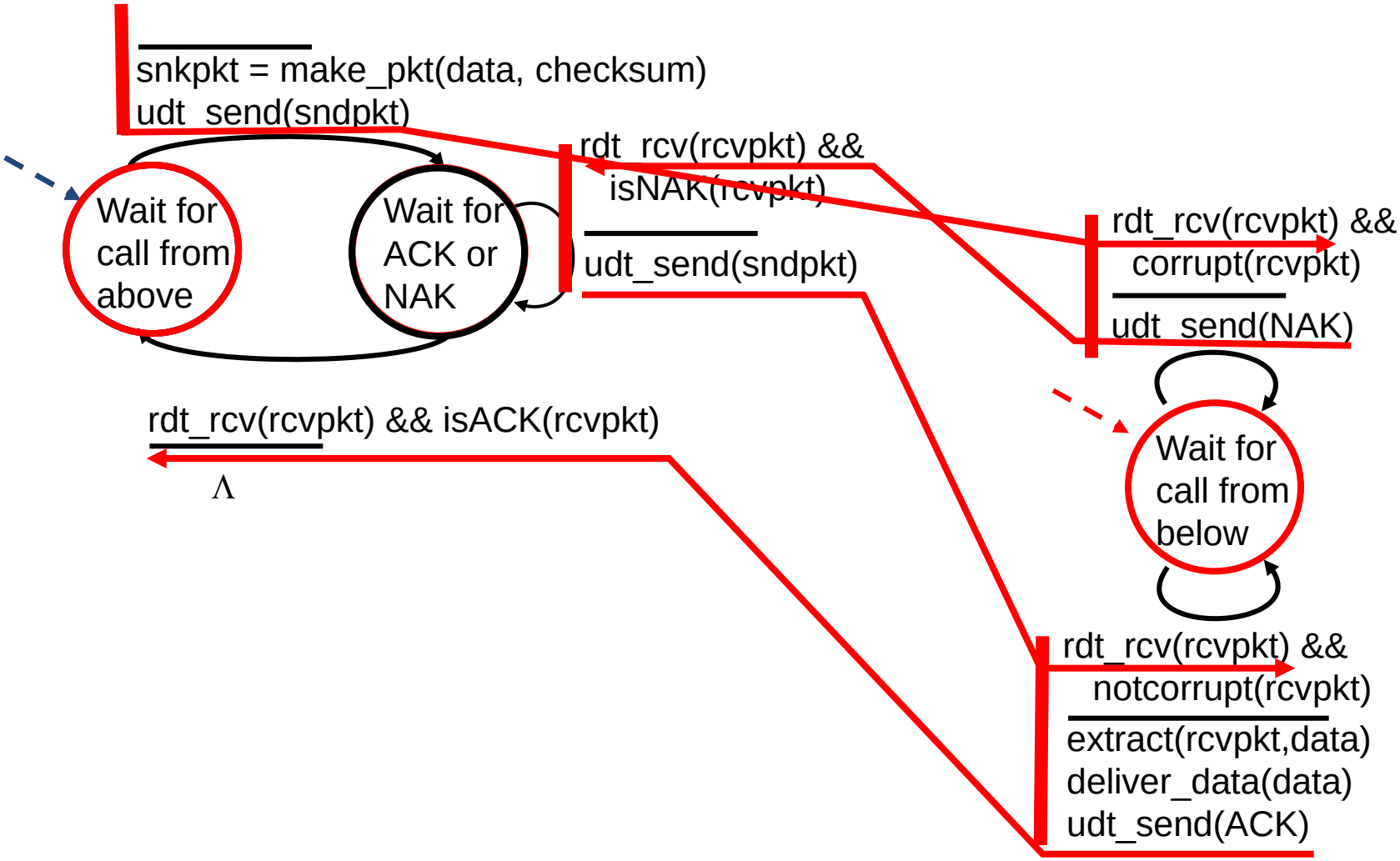
stop and wait

sender sends one packet,
then waits for receiver response

rdt2.0: no error scenario (with flow)



rdt2.0: error scenario (with flow)



rdt2.0: has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!

Possible solutions:

- Sender ask the receiver to retransmit the ACK/NAK pkt
- Use forward error correction (FEC)
- Sender retransmit the current data pkt
  - can't just retransmit: possible duplicate pkt !

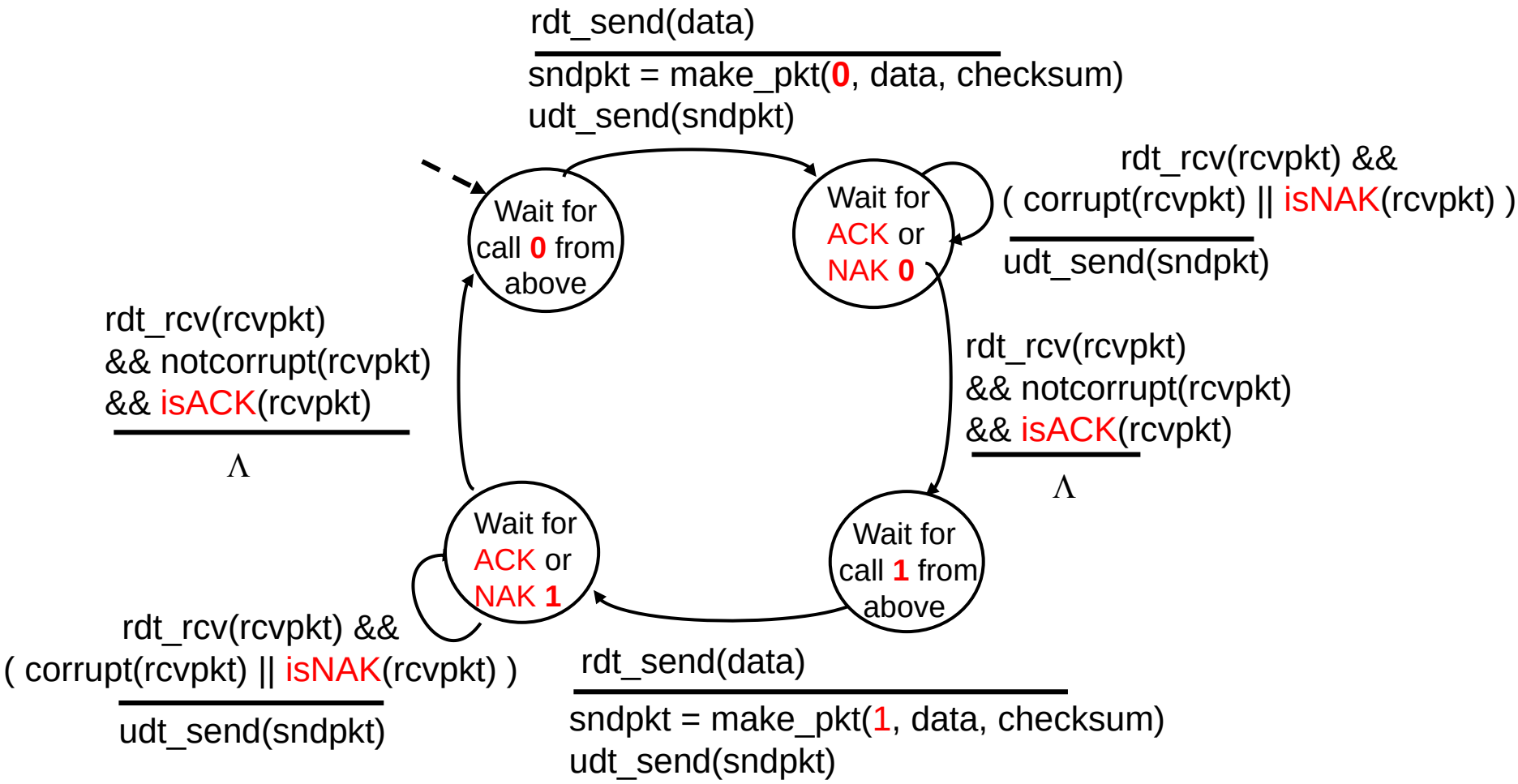
Handling duplicates:

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

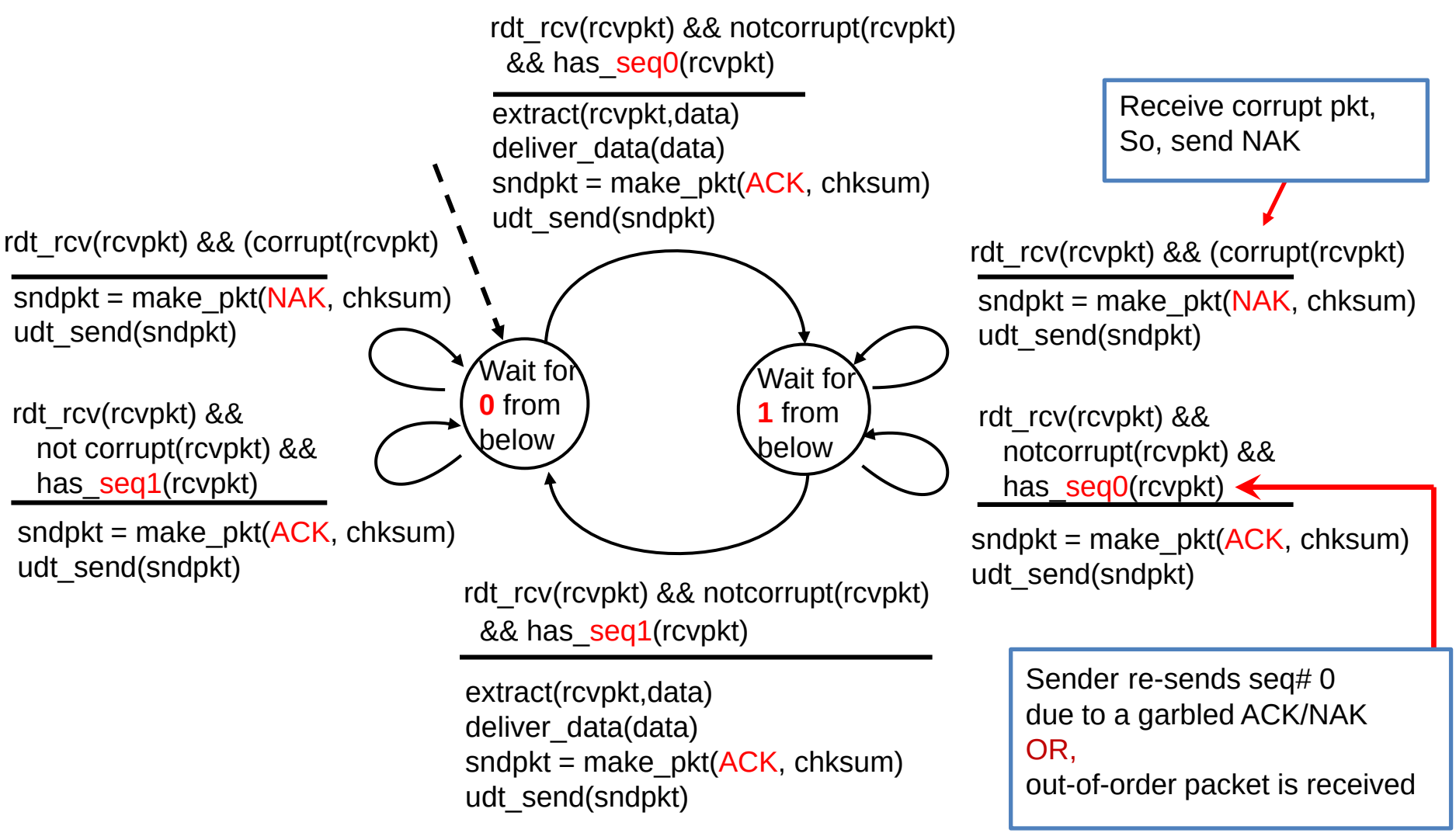
Sequence number:

- For stop-and-wait, one bit sequence number will be fine

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



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

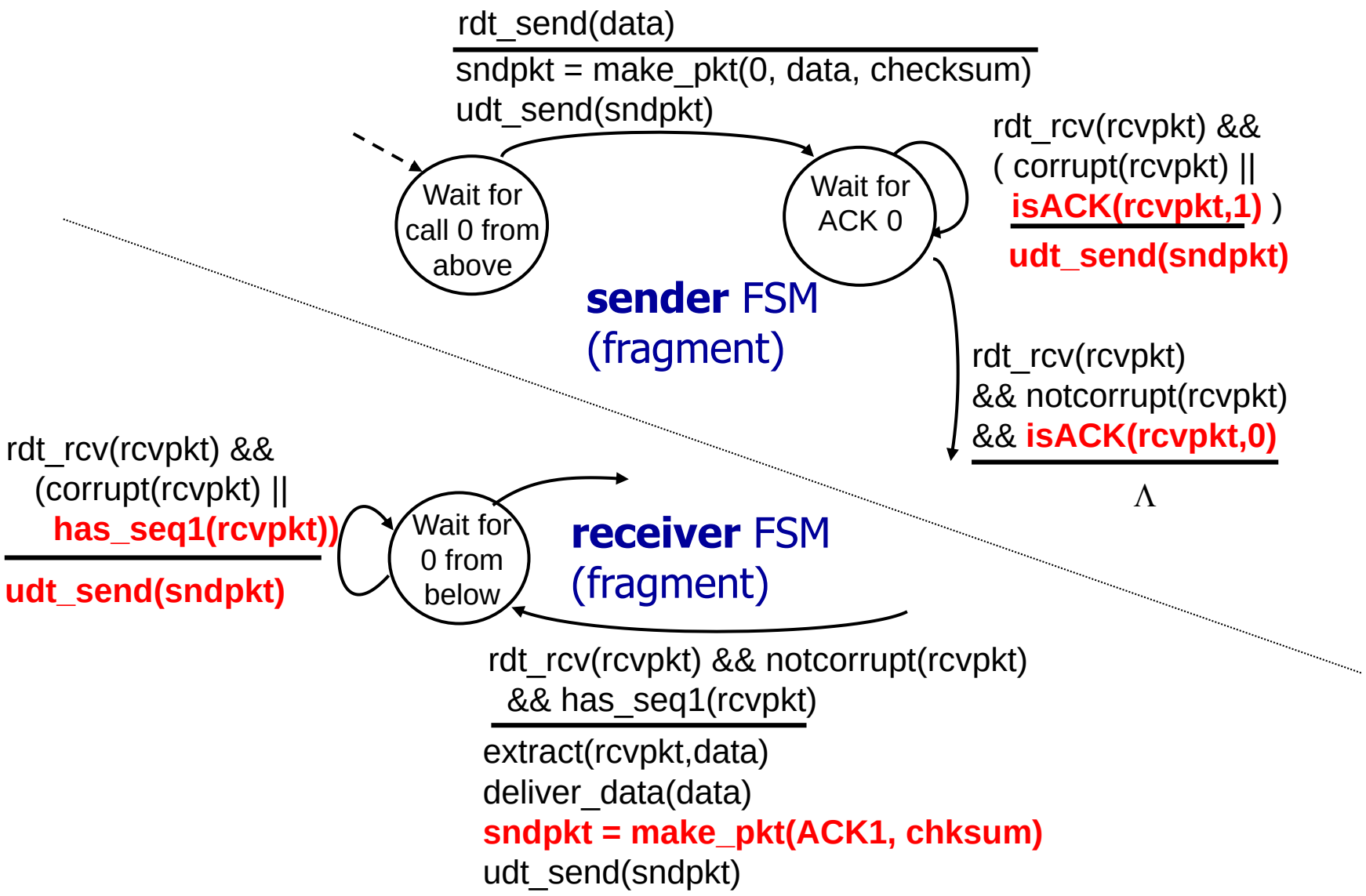
receiver:

- must check if received packet corrupted
- must check if received packet is duplicate **or** new
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver **can not know** if its last ACK/NAK received OK at sender

Question: Do we really need NAK?

Answer: **No!** Instead of NAK, receiver sends ACK for last pkt received OK. Receiver must explicitly include seq # of pkt being ACKed .

rdt2.2: sender, receiver fragments



rdt3.0: channels with error & loss

new scenario: underlying channel can also lose packets (data, ACKs)

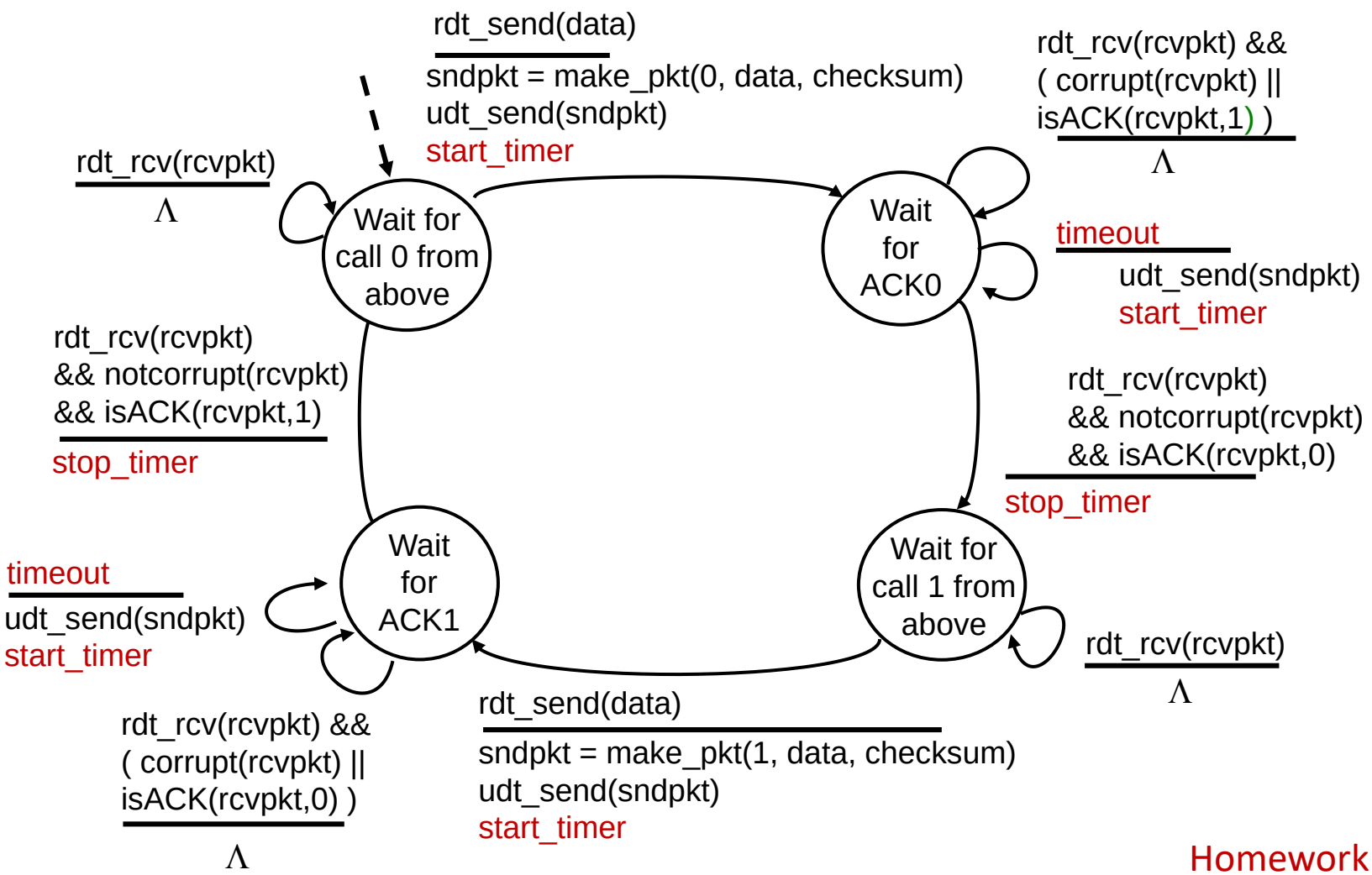
- checksum, seq. #, ACKs, retransmissions will be of help … but not enough !

approach:

sender waits "reasonable" amount of time for ACK

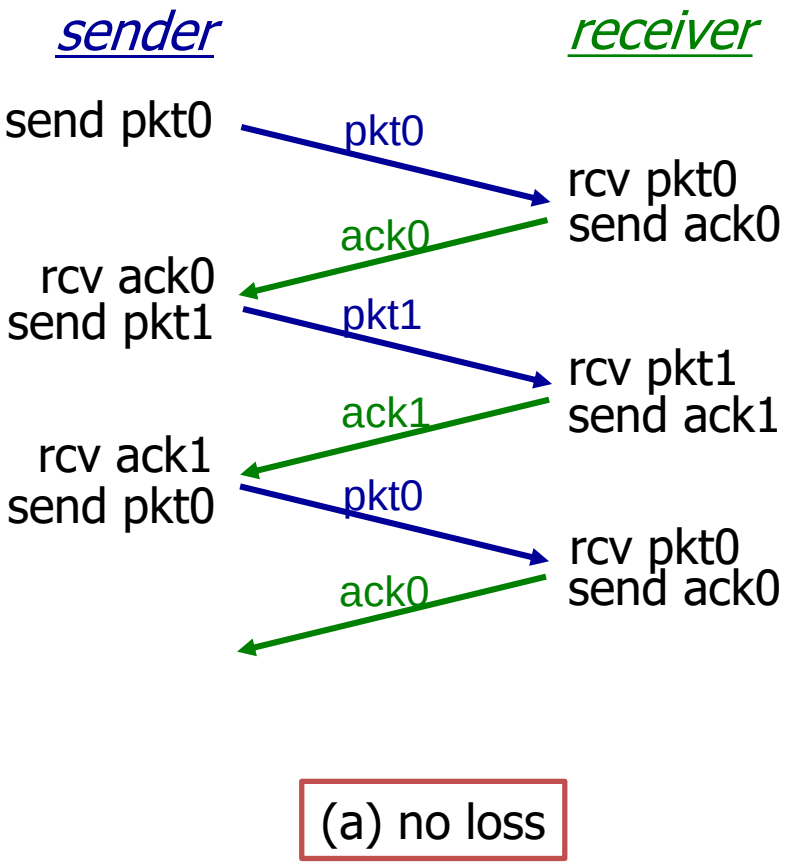
- retransmits if no ACK received in this time
- if pkt / ACK just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0: sender

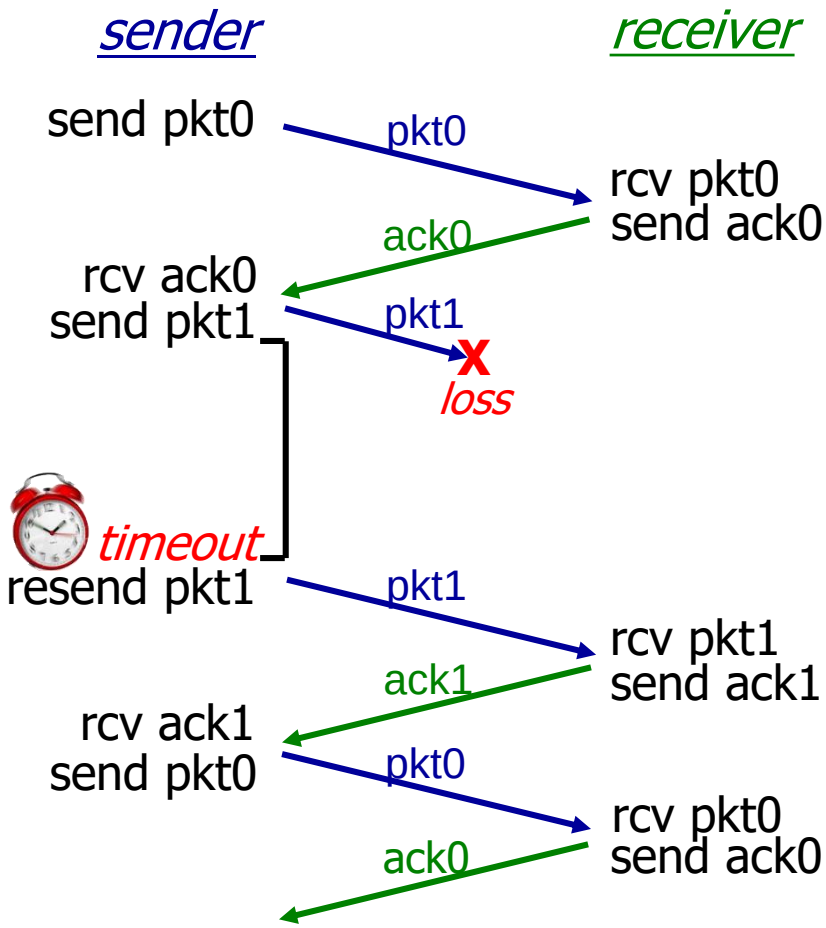


Homework:
FSM of the receiver

rdt3.0 in action (for pkt loss)

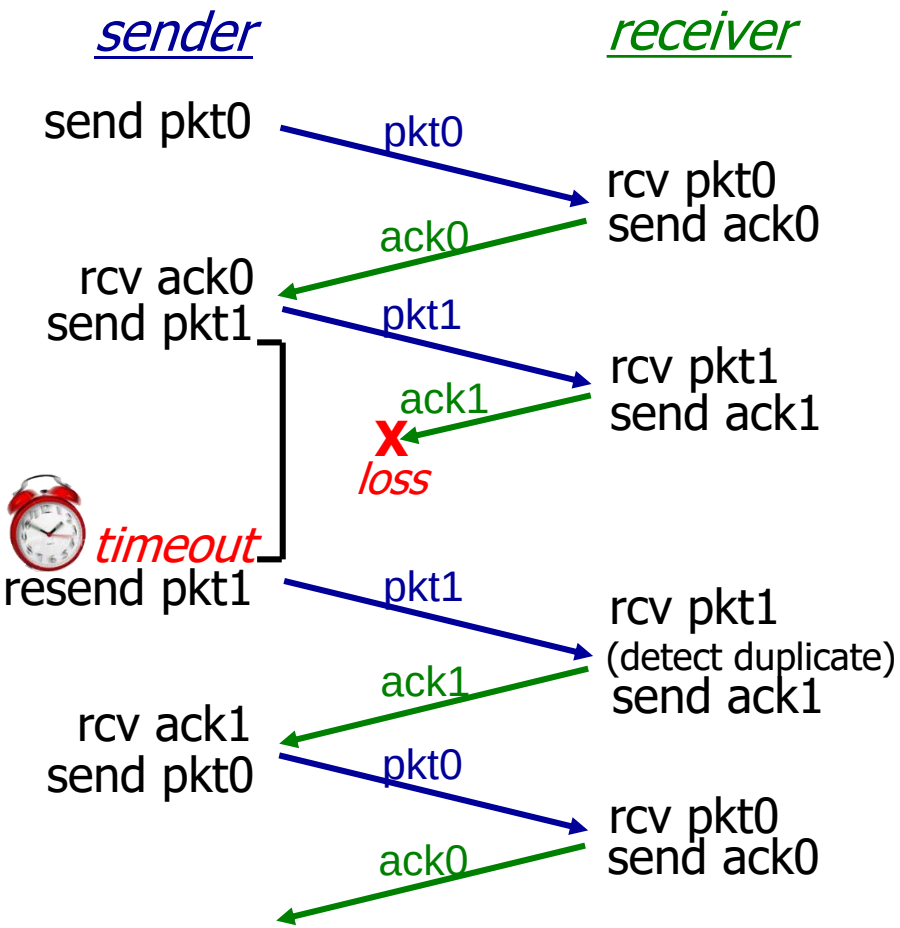


(a) no loss

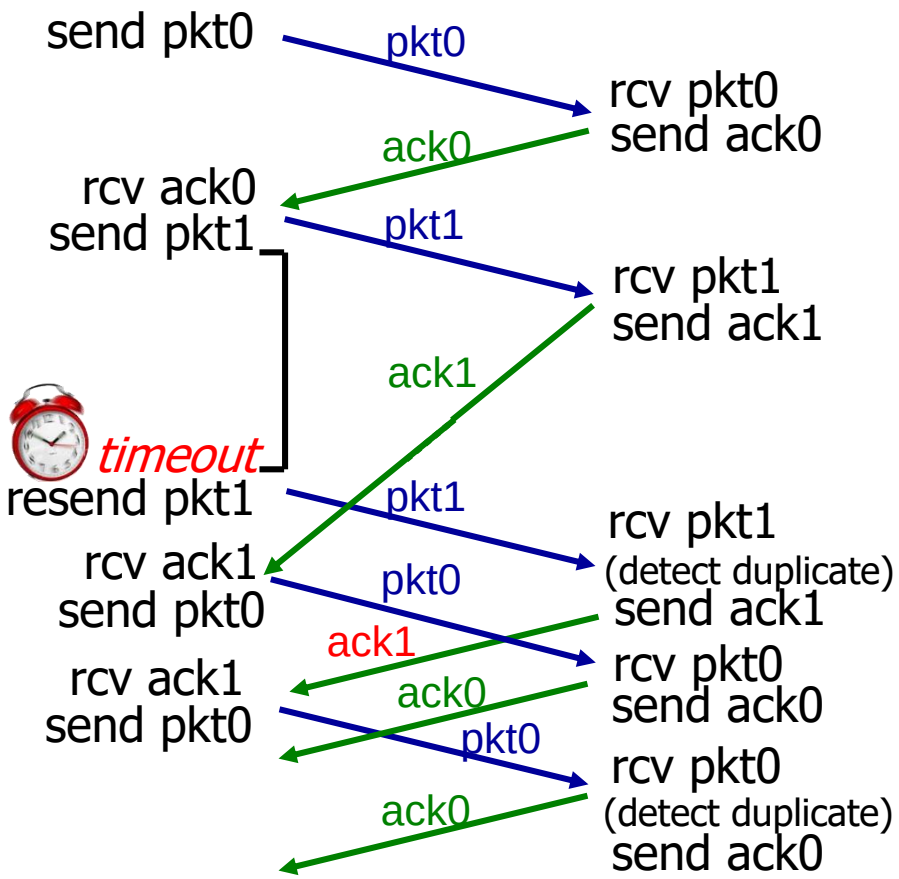


(b) packet loss

Cont…



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 1000 bytes packet:

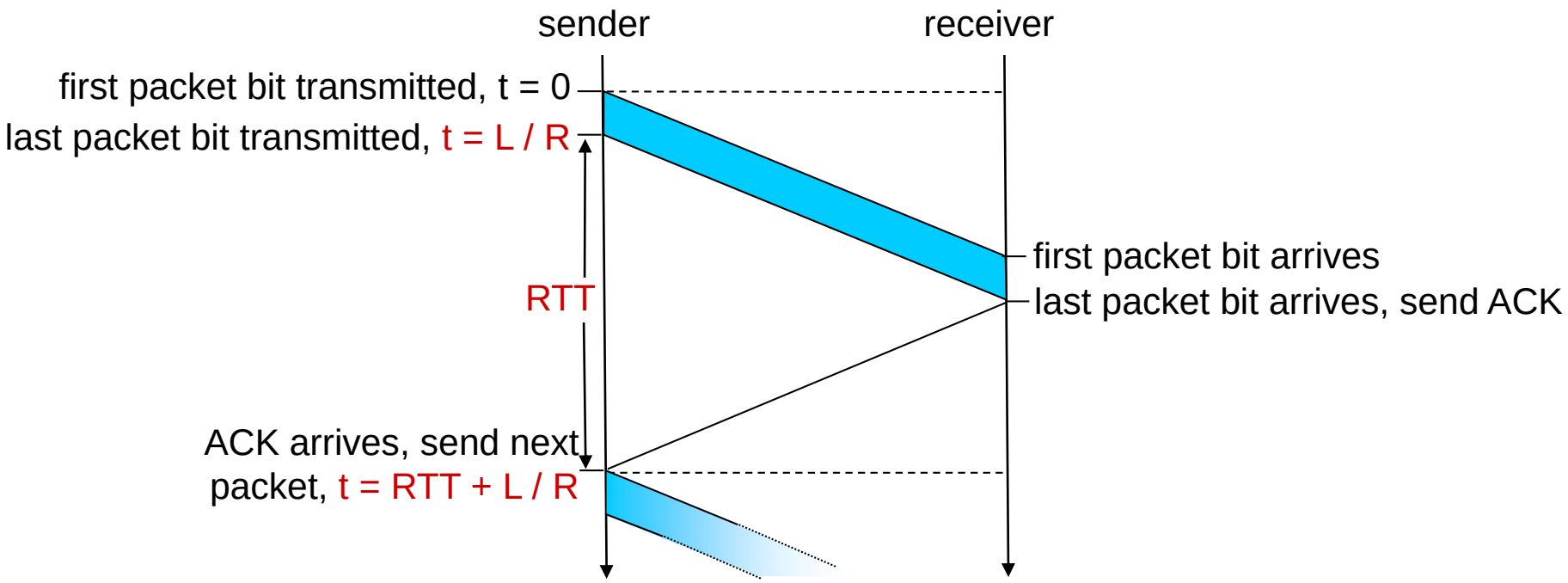
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

  - U_{sender}: *utilization* – fraction of time sender busy sending

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

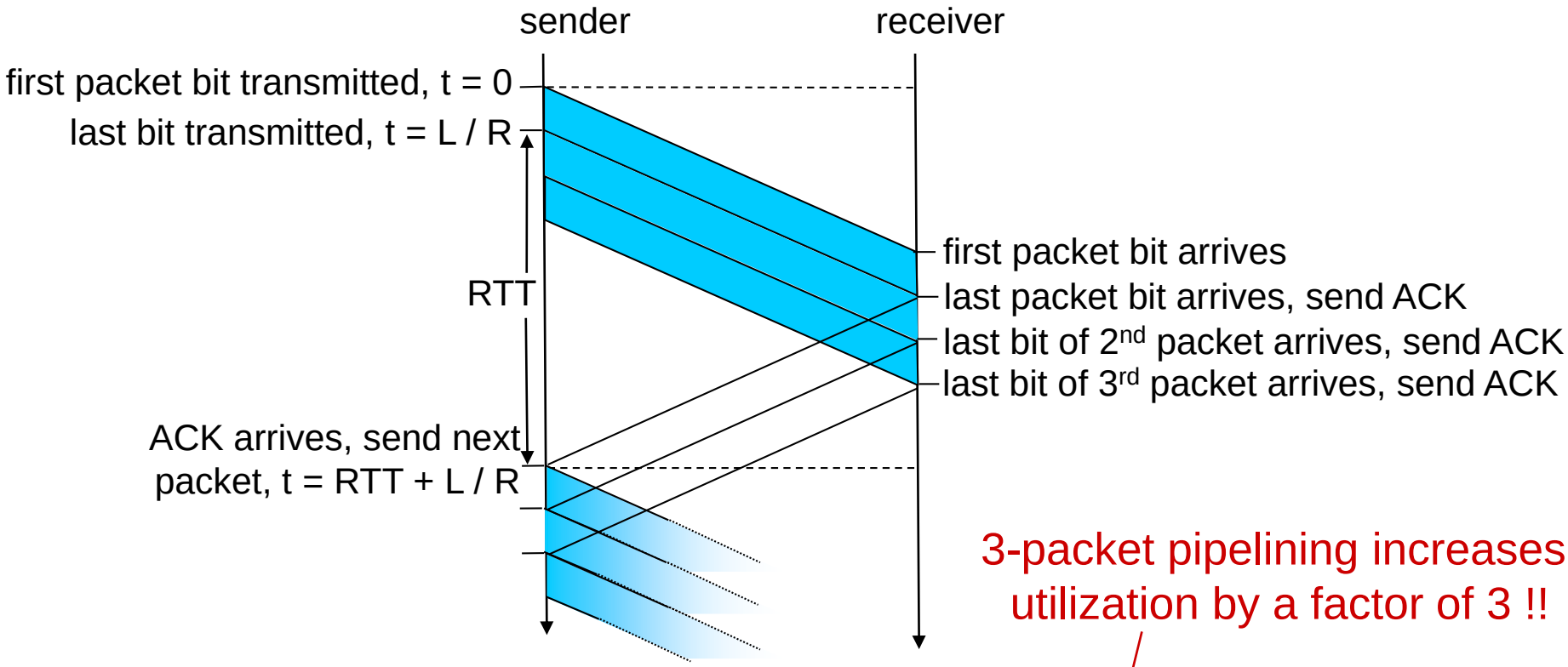
  - if RTT=30 msec, 1KB pkt every 30 msec: 33 kB/sec thrughput over 1 Gbps link !
- network protocol limits the use of physical resources!

rdt3.0: stop-and-wait protocol



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

Pipelining: increased utilization

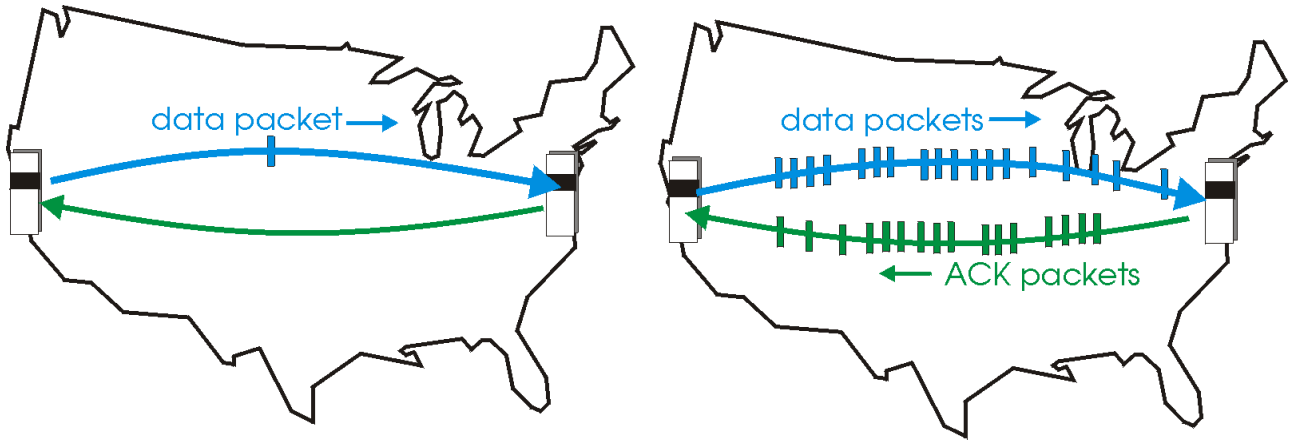


3-packet pipelining increases utilization by a factor of 3 !!

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

Cont…

- pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

- two generic forms of pipelined protocols:
  - *go-Back-N*
  - *selective repeat*

Pipelined protocols: overview

Go-back-N:

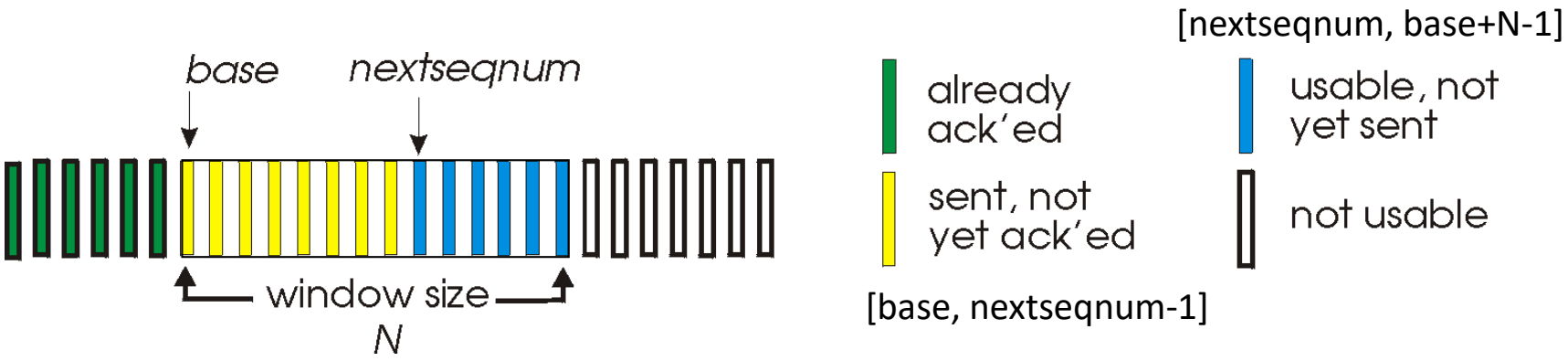
- sender can have up to *N* unacked packets in pipeline
- receiver only sends *cumulative ACK*
  - Doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to *N* unacked packets in pipeline
- receiver sends *individual ACK* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

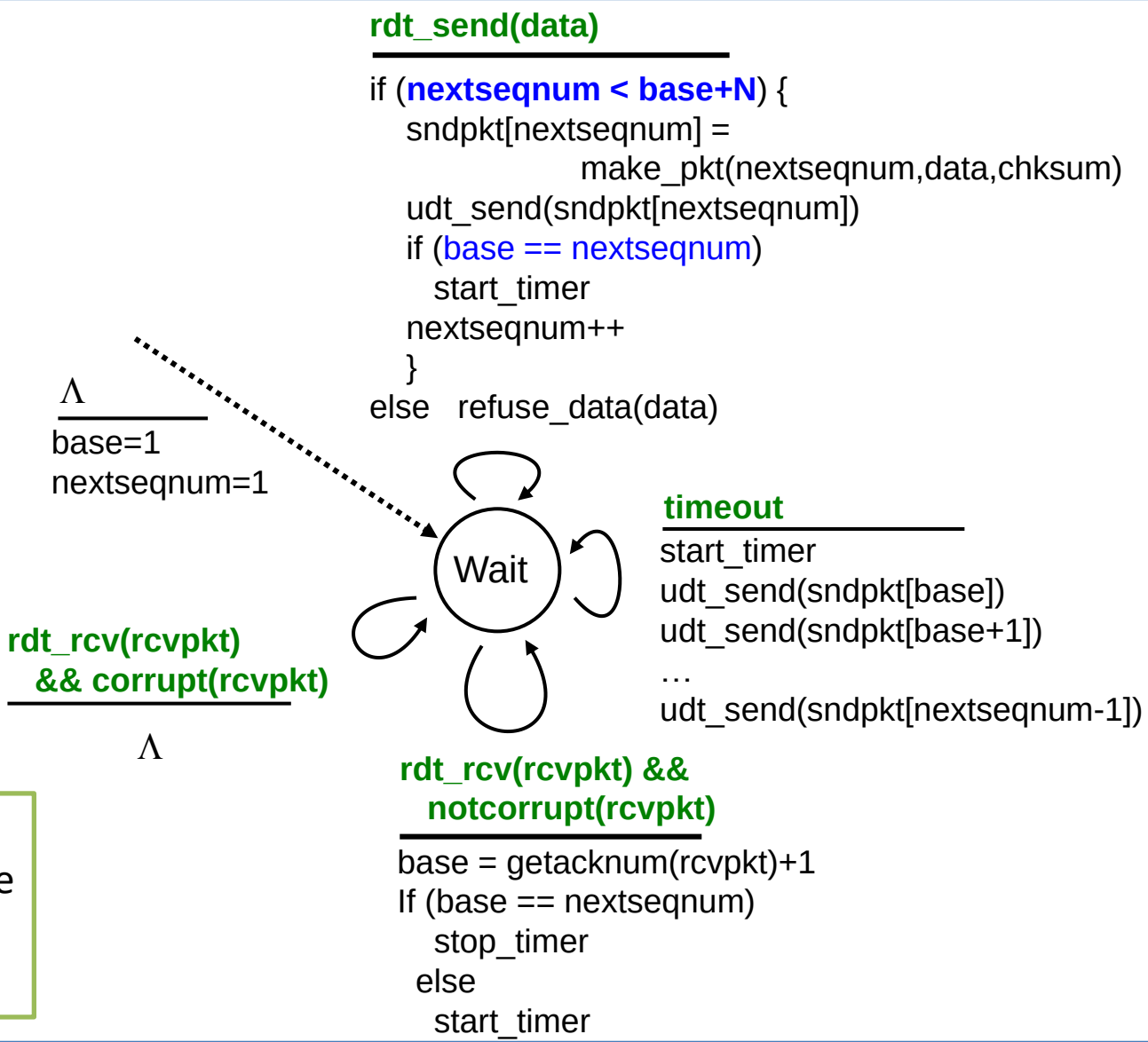
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed
- GBN is refer as *sliding window* protocol



- *base*: seq# of the oldest unacked pkt
- *nextseqnum*: seq# of the next pkt to be sent
- *ACK(n)*: ACKs all pkts up to, including seq # *n* - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- *timer* for oldest in-flight pkt
- *timeout(n):* retransmit packet *n* and all higher seq # pkts in window

GBN: sender extended FSM

ACK-based
NAK-free
Extended FSM



GBN sender respond to:

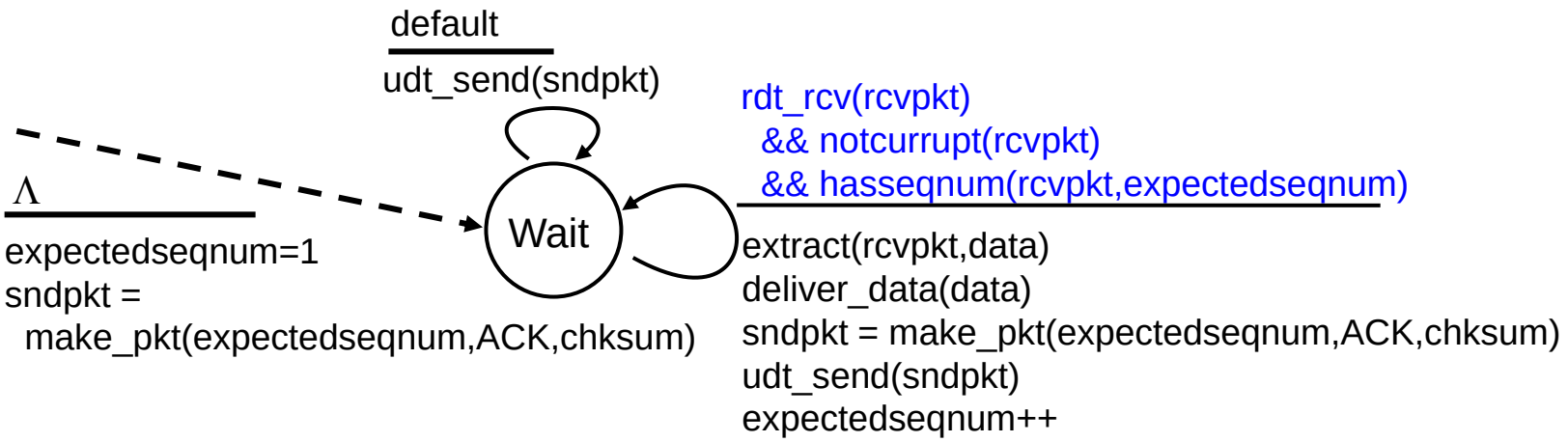
1) Invocation from above
2) Receipt of an ACK
3) Timeout

GBN: receiver extended FSM

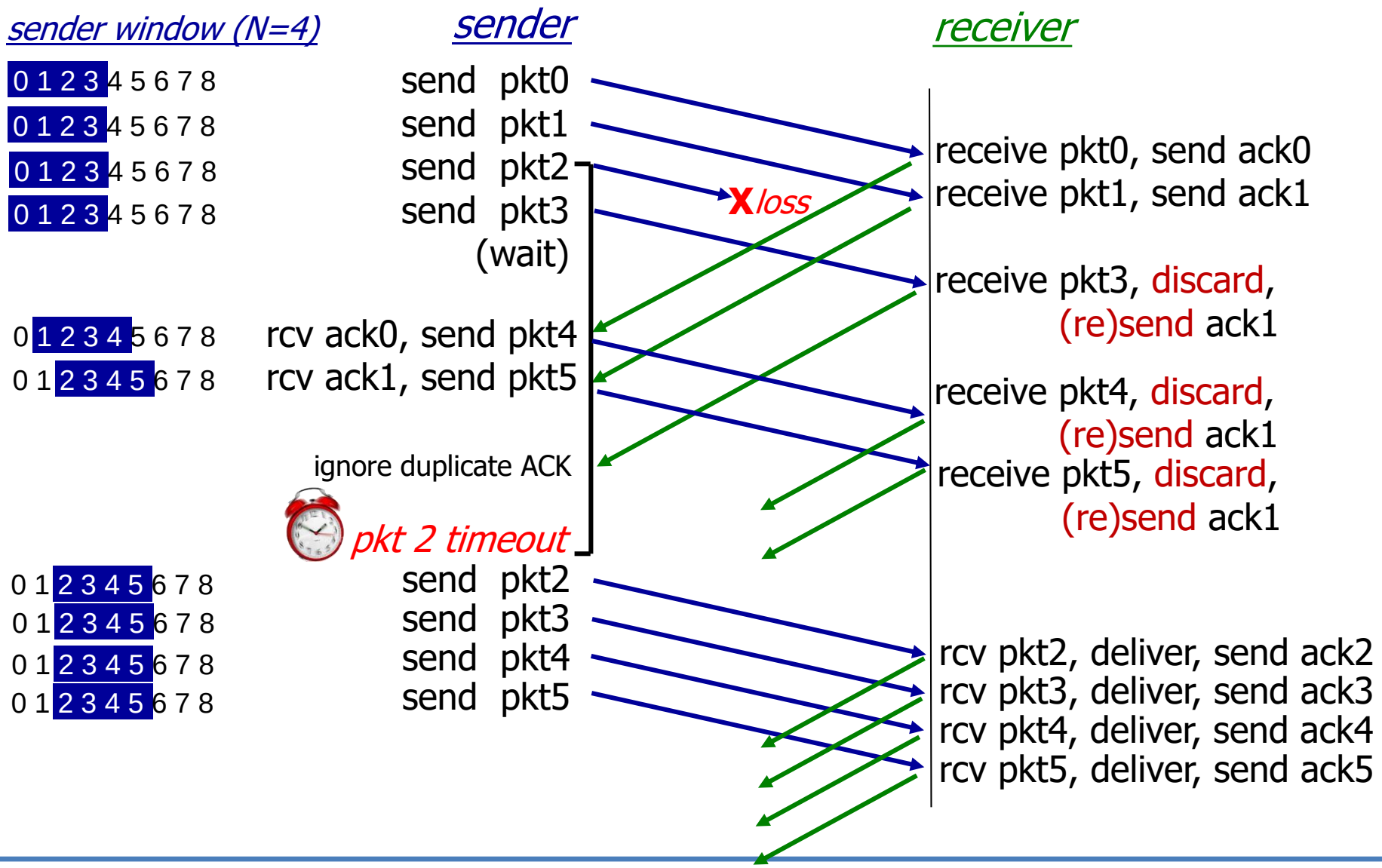
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only to remember **`expectedseqnum`**

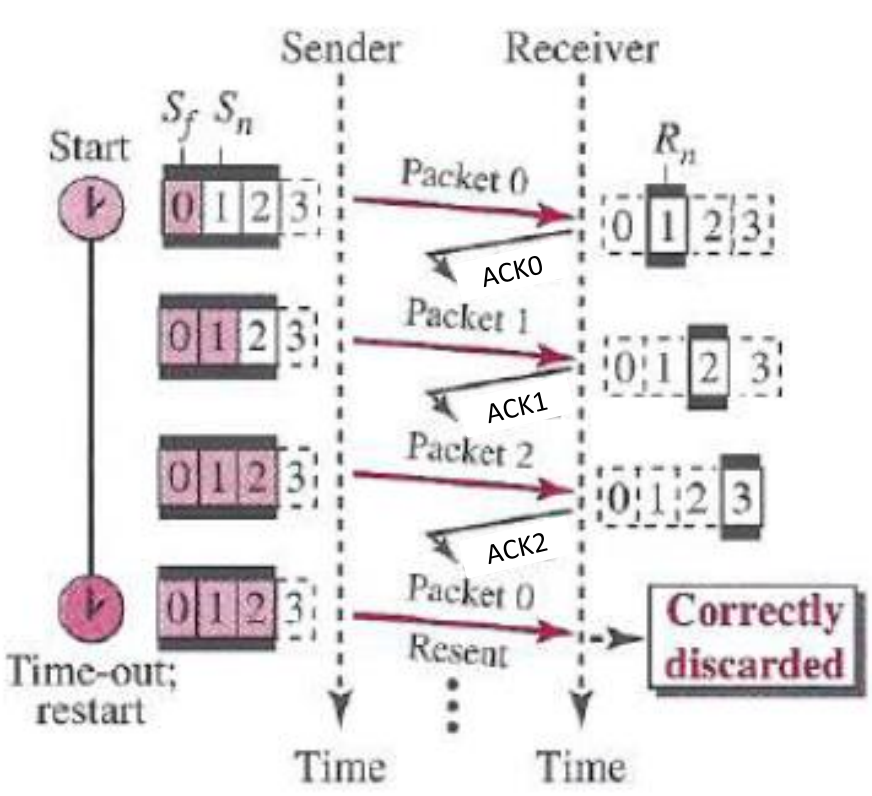
- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with **highest in-order seq #**



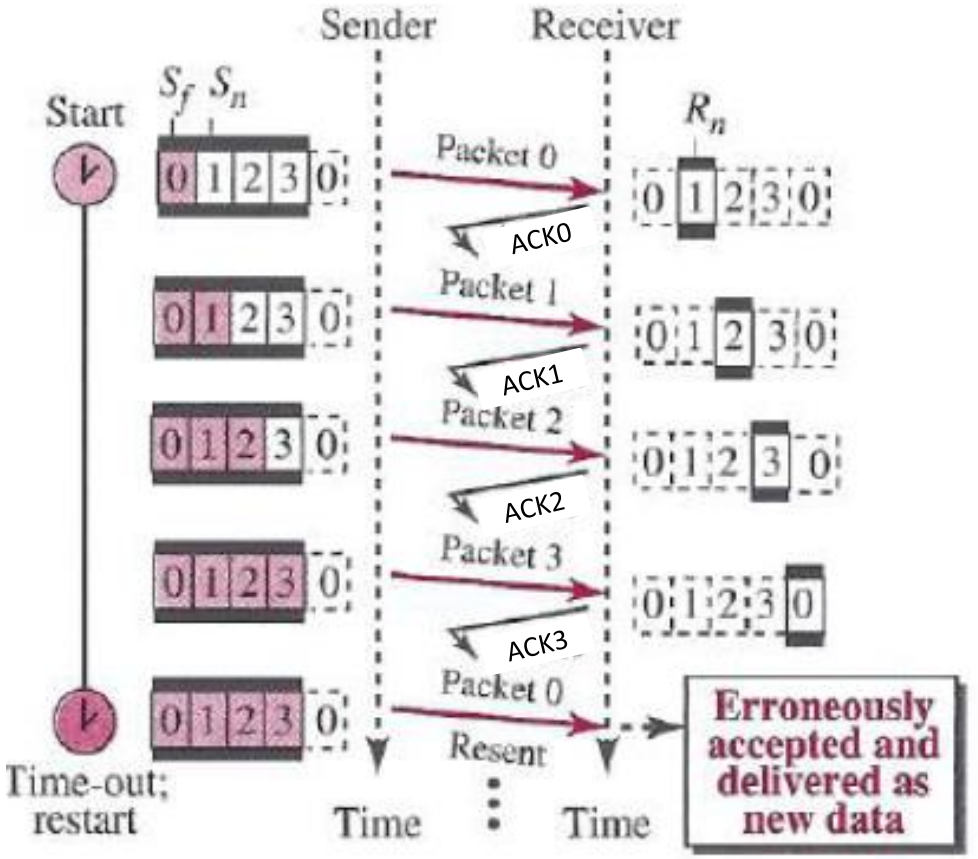
GBN in action



Send Window Size in GBN



a. Send window of size $< 2^m$



b. Send window of size $= 2^m$

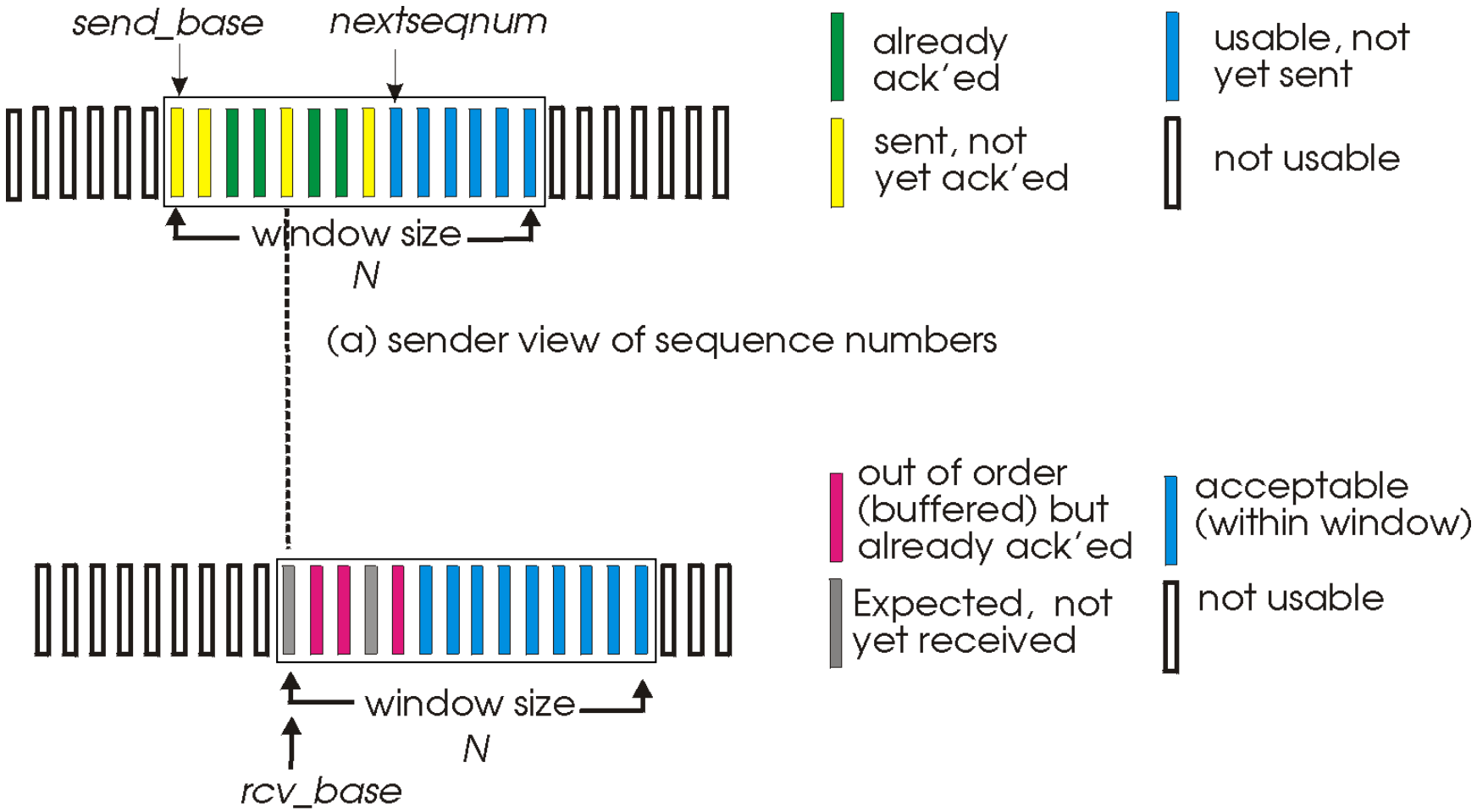
Selective repeat (SR)

- In GBN,
  - receiver discards out-of-ordered pkts even if they are received correctly !
  - Can we do buffering to avoid unnecessary retransmission?

In SR,

- receiver *individually* acknowledges 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
- receiver window
  - *N* consecutive seq #'s
  - limits seq #s of acceptable pkts

SR: sender, receiver windows



Sender, Receiver – Events & Actions

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

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

receiver

pkt with seq# in [rcvbase, rcvbase+N-1]

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

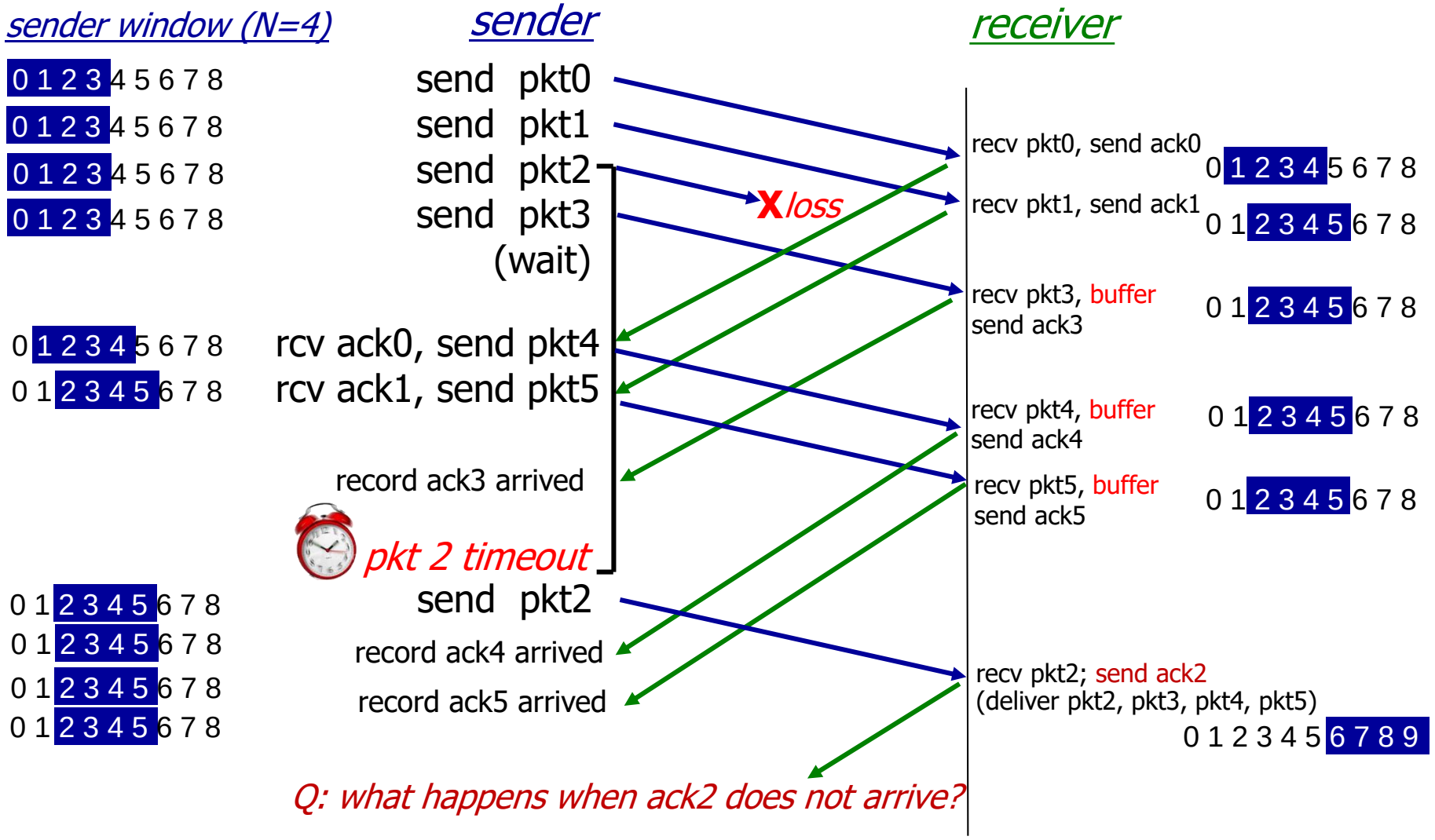
pkt with seq# in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

SR in action


sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
recv pkt0, send ack0
recv pkt1, send ack1
recv pkt3, buffer
send ack3
rcv ack0, send pkt4
rcv ack1, send pkt5
recv pkt4, buffer
send ack4
recv pkt5, buffer
send ack5
record ack3 arrived
pkt 2 timeout
send pkt2
record ack4 arrived
record ack5 arrived
recv pkt2; send ack2
(deliver pkt2, pkt3, pkt4, pkt5)
Q: what happens when ack2 does not arrive?

SR: dilemma !

sender window (after receipt)

receiver window (after receipt)

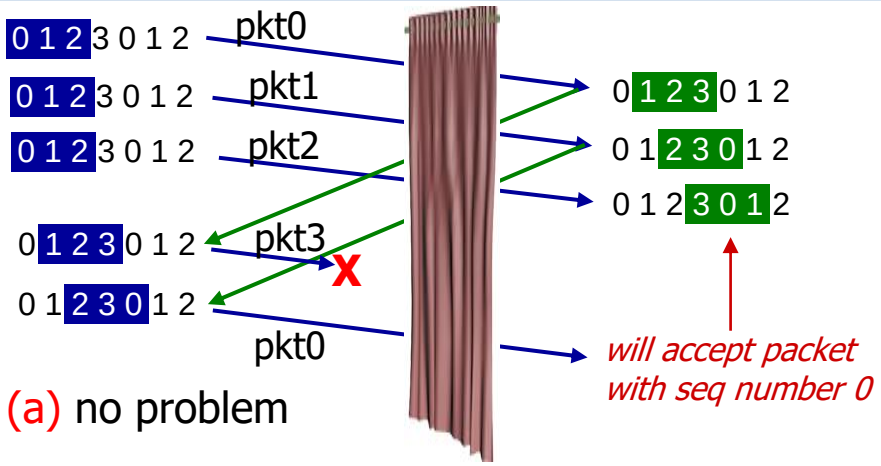
example:

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

❖ receiver sees no difference in two scenarios!

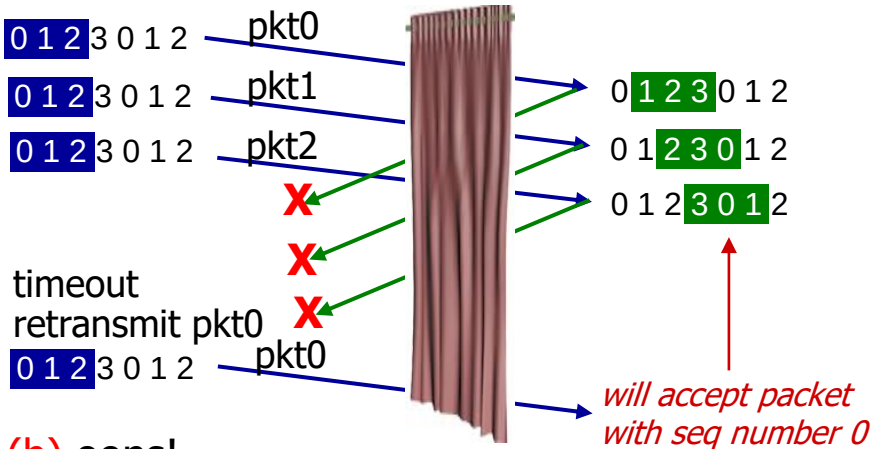
❖ duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



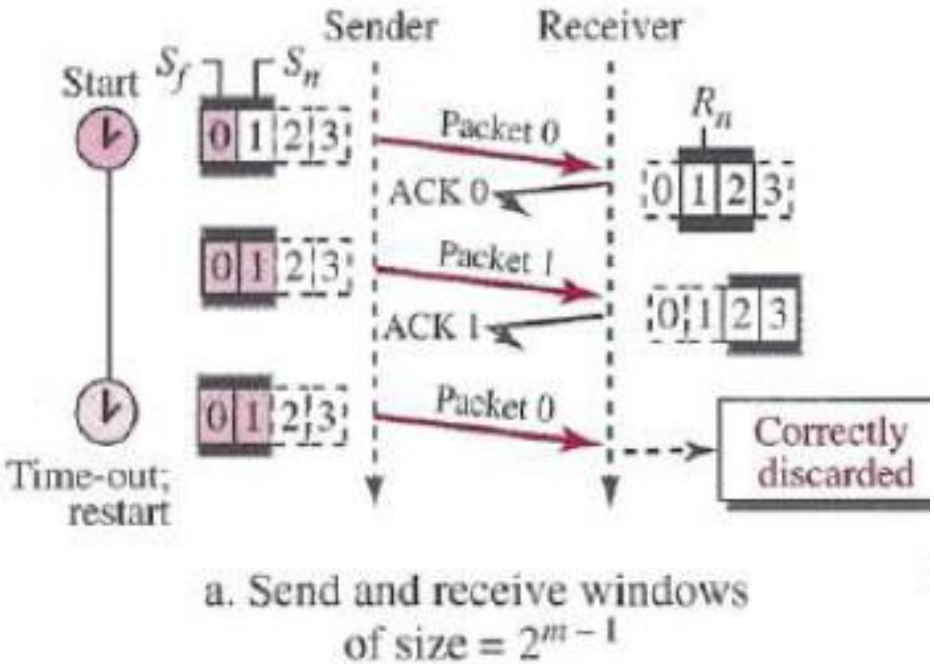
(a) no problem

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



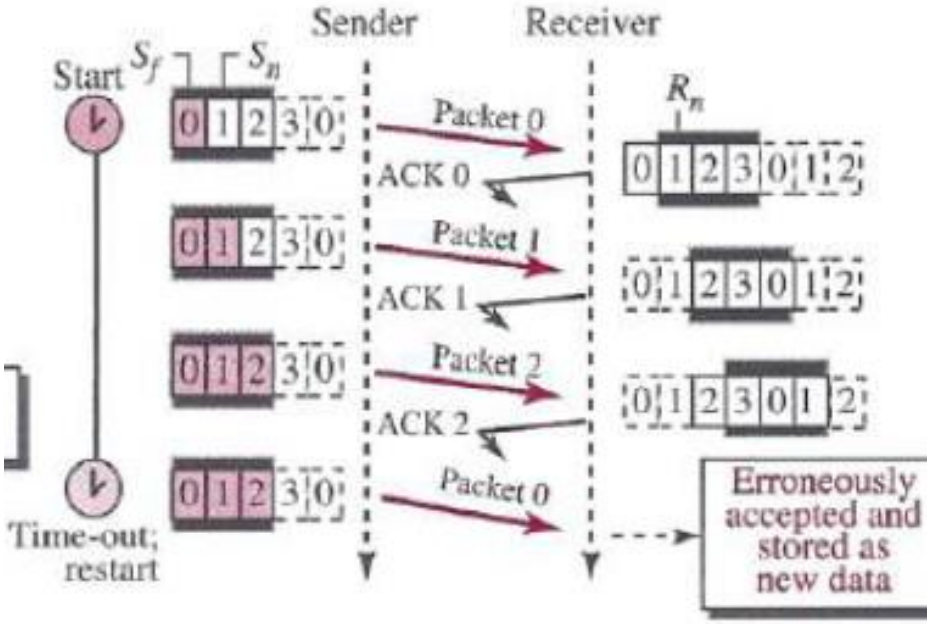
(b) oops!

Window Size in SR



a. Send and receive windows of size = 2^{m-1}

A: window size <= ½(seq# size)



b. Send and receive windows of size > 2^{m-1}

Thanks!

Content of this PPT are taken from:

1) **Computer Networks: A Top Down Approach**, by J.F. Kuros and K.W. Ross, 6th Eds, 2013, Pearson Education.

2) **Data Communications and Networking**, by B. A. Forouzan , 5th Eds, 2012, McGraw-Hill.

3) **Chapter 3 : Transport Layer,** PowerPoint slides of "Computer Networking: A Top Down Approach", 6th Eds, J.F. Kurose, K.W. Ross