Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

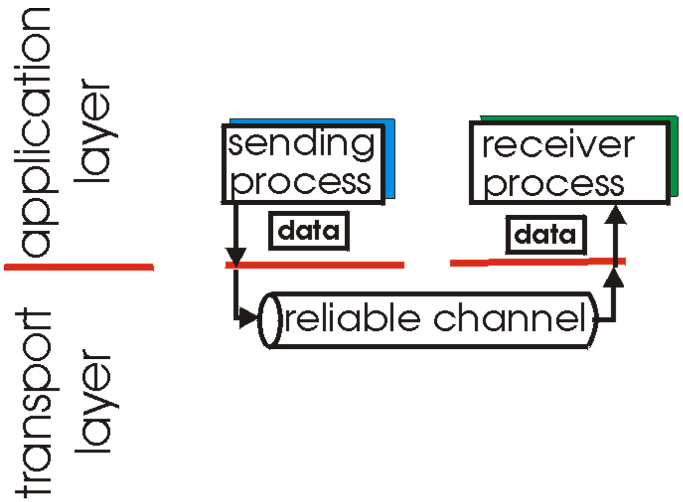
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

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

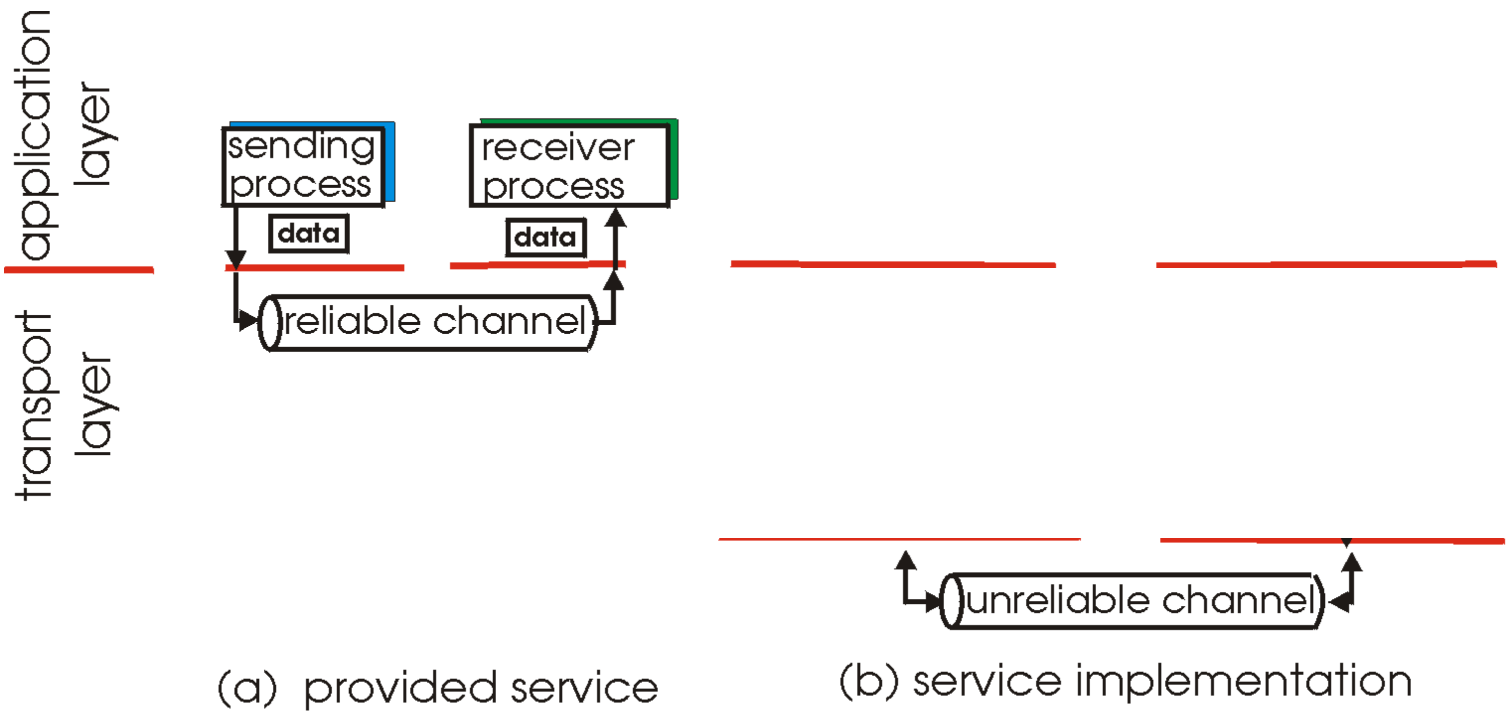


(a) provided service

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

Principles of reliable data transfer

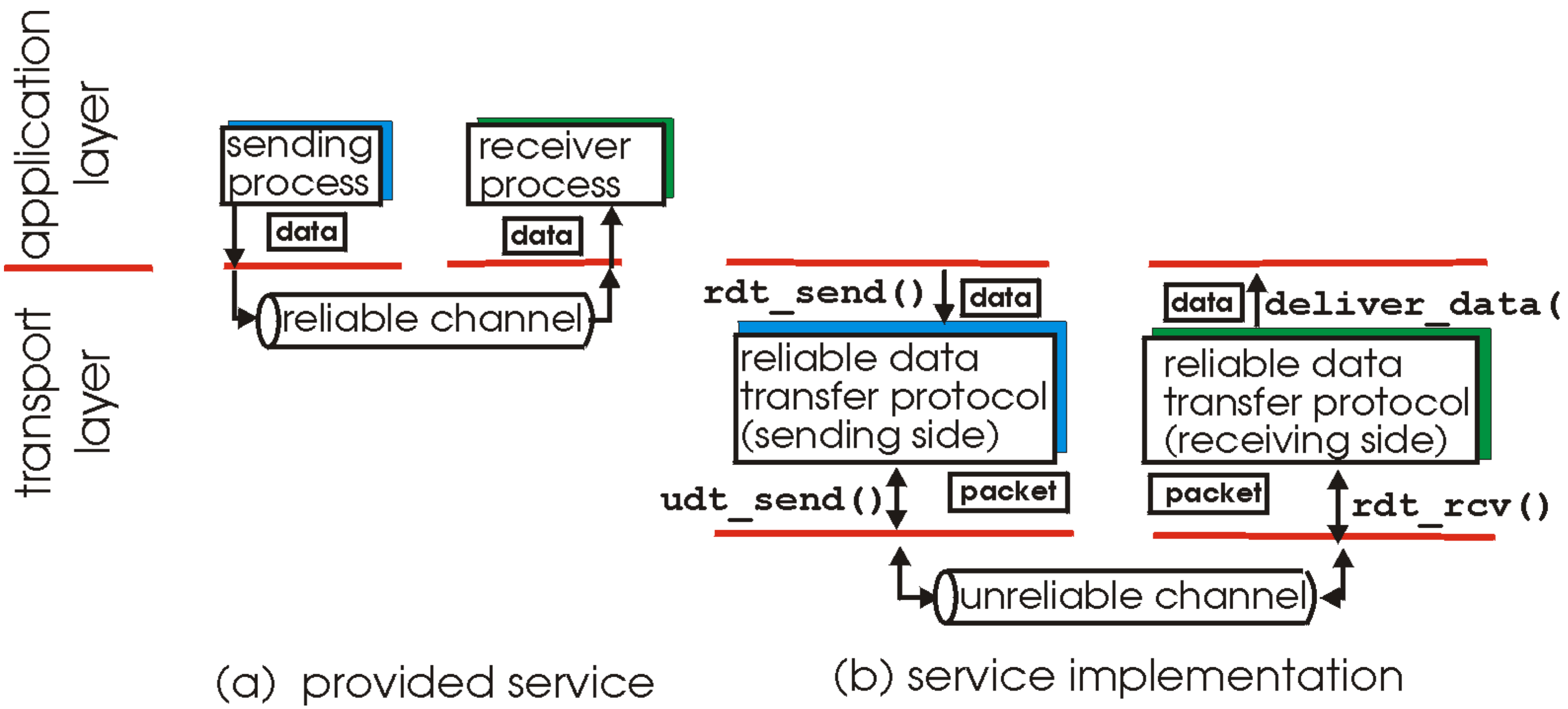
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

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

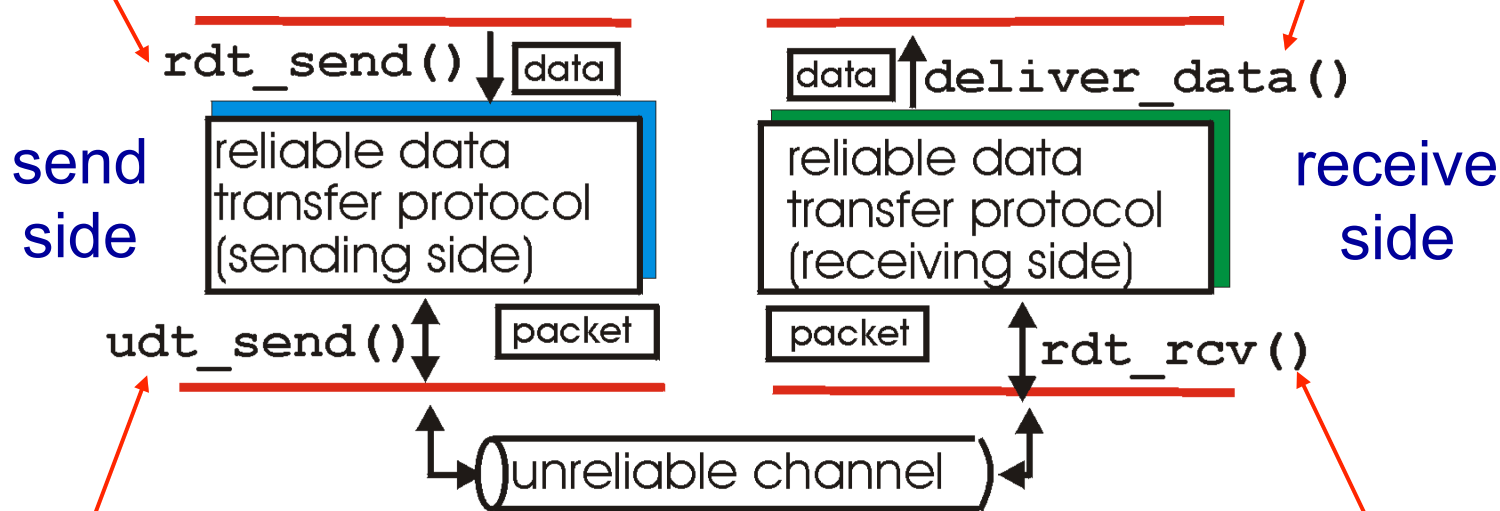


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: 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



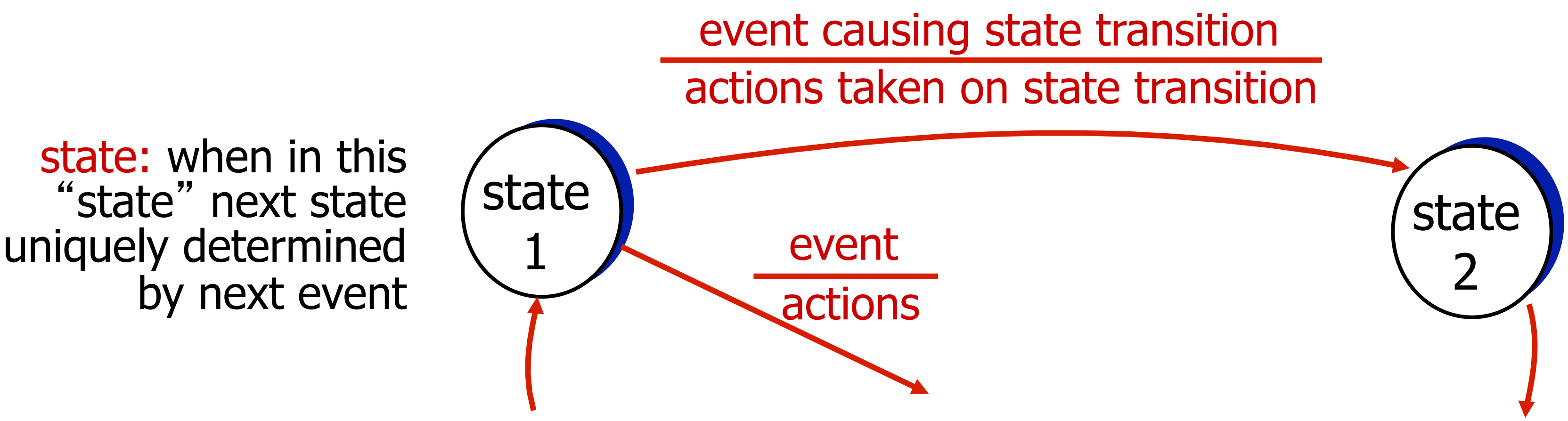
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

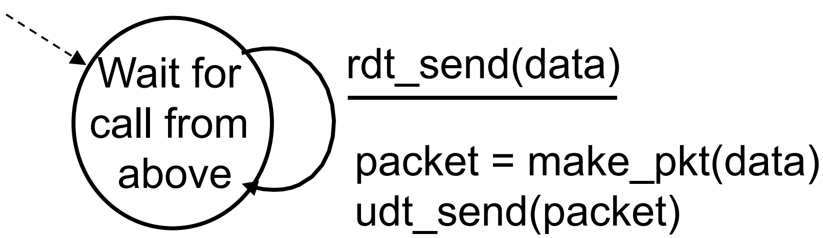
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

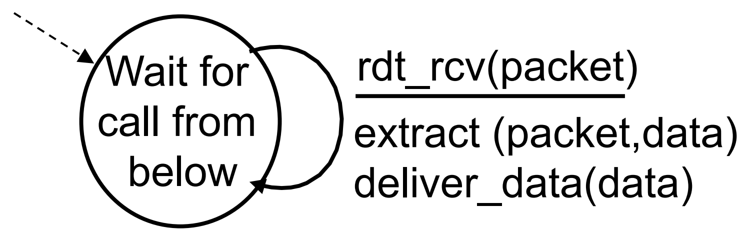


rdt1.0: reliable transfer over a reliable channel

- 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
- Clearly RDT is useless here ☺



sender



receiver

rdt2.0: channel with bit errors

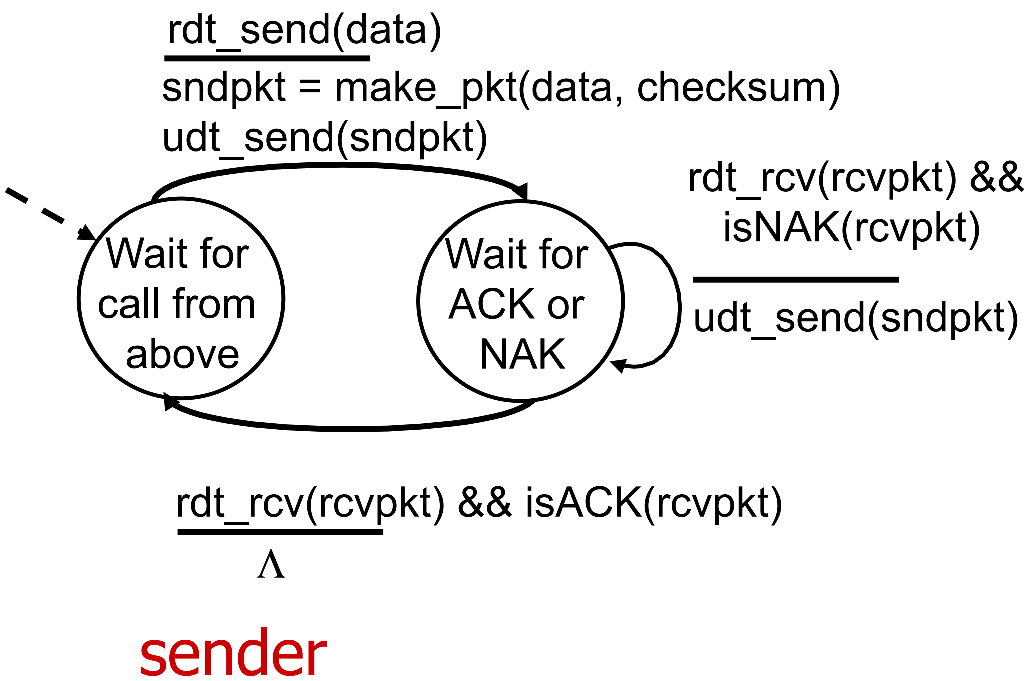
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

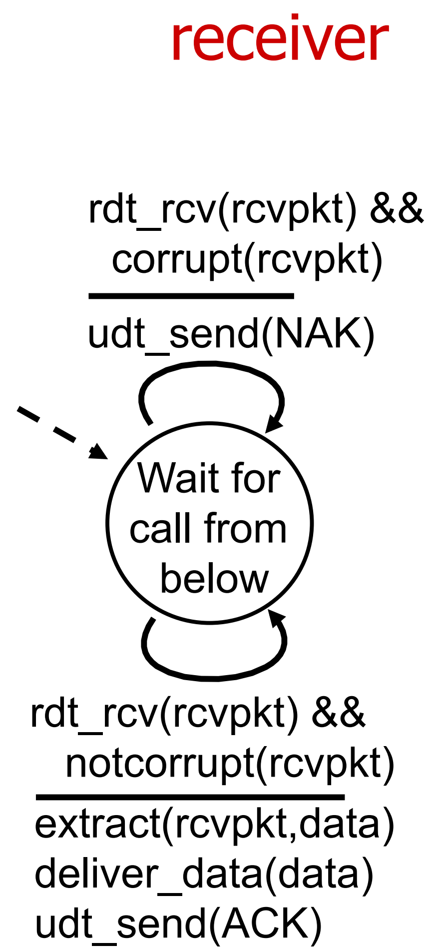
How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- Q: how to recover from errors?
- A: new mechanisms needed:
  - *feedback (control msgs from receiver to sender)*
    - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
    - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - *retransmission*
    - sender retransmits pkt on receipt of NAK

rdt2.0: FSM specification

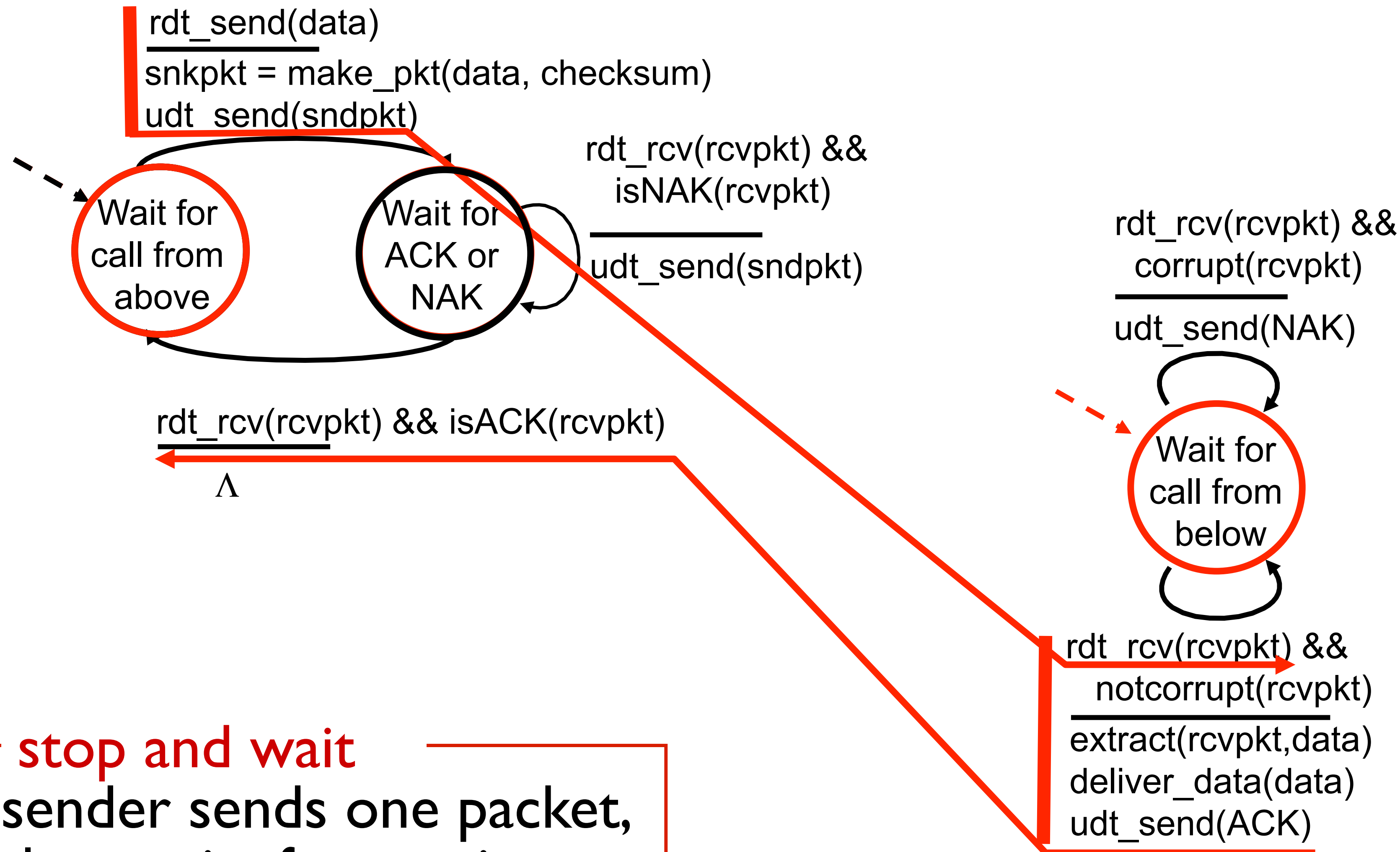




stop and wait

sender sends one packet, then waits for receiver response

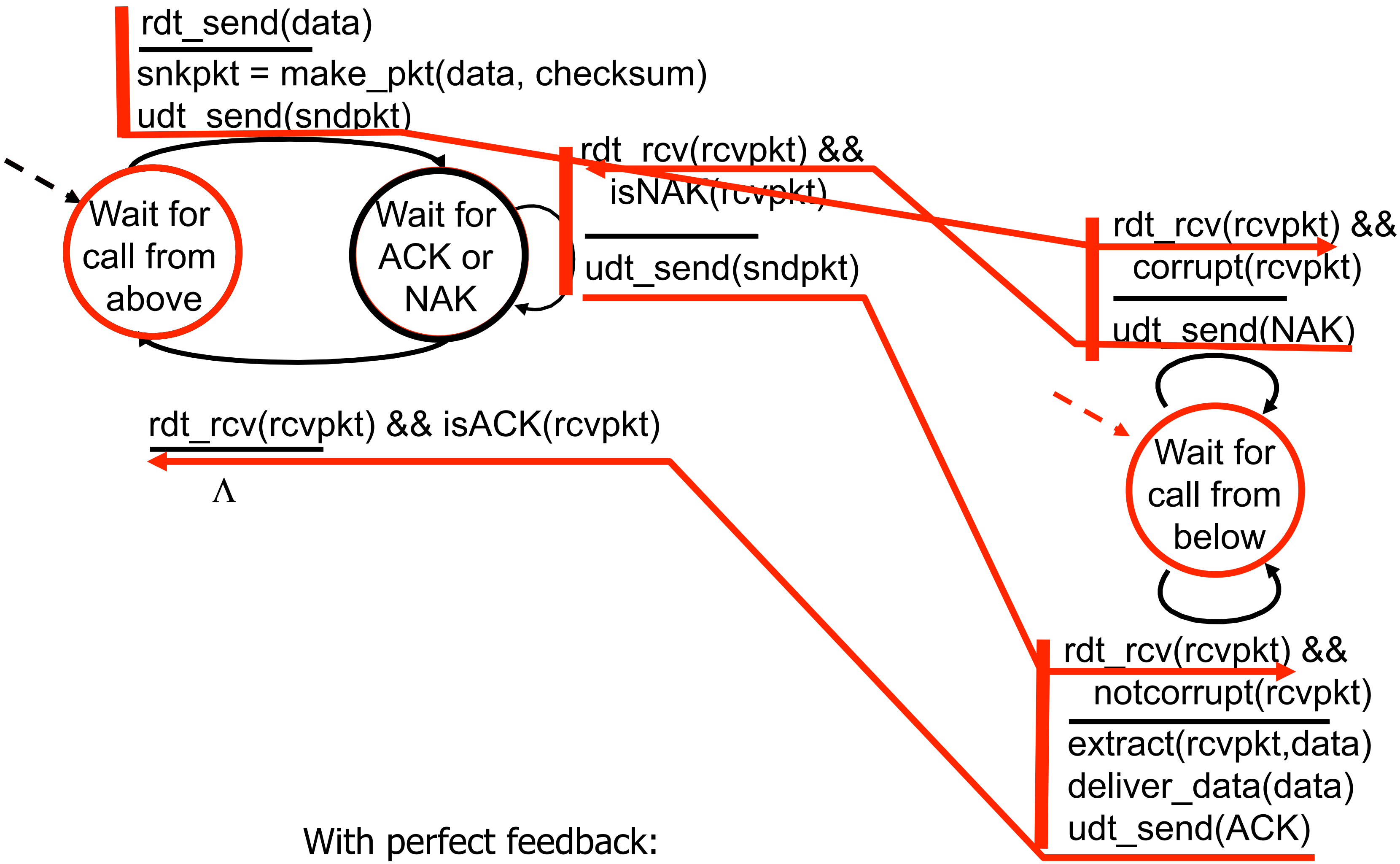
rdt2.0: operation with no errors



stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: error scenario



With perfect feedback:

sender's state and receiver's state are in perfect sync

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
  - Packet received or corrupted?
  - ACK corrupted not (exactly) the same as ACK being lost
- solutions
  - Feedback on feedback…
    - may be corrupted but we can detect that (e.g. through checksum).
  - Error detection+correction
    - Cannot handle lost acks
  - Retransmissions
    - can't just retransmit: possible duplicates

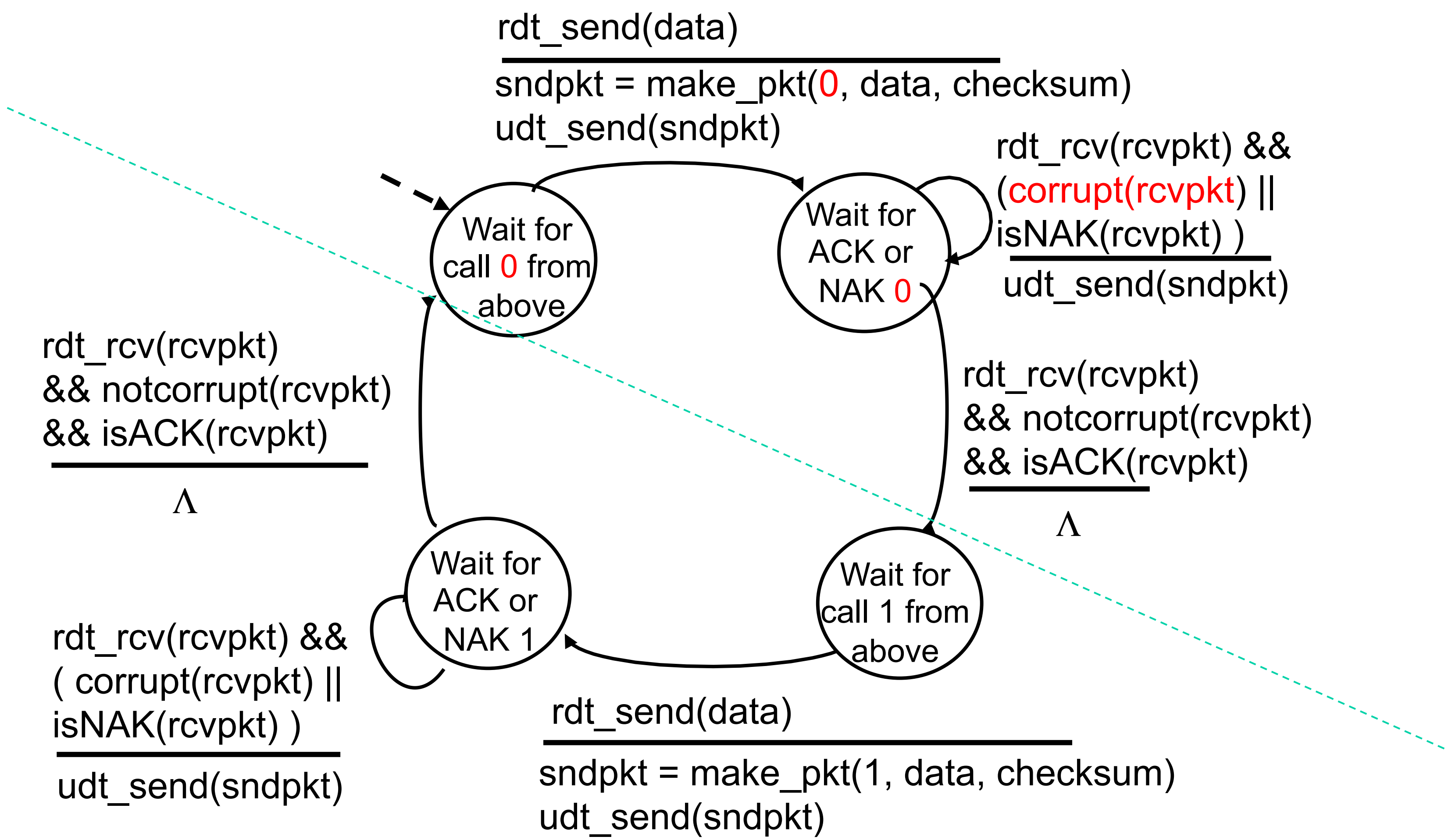
handling duplicates:

- sender conservatively retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt
- but receiver always ACKs/NAKs received packet (to help the server move on to next seqno)

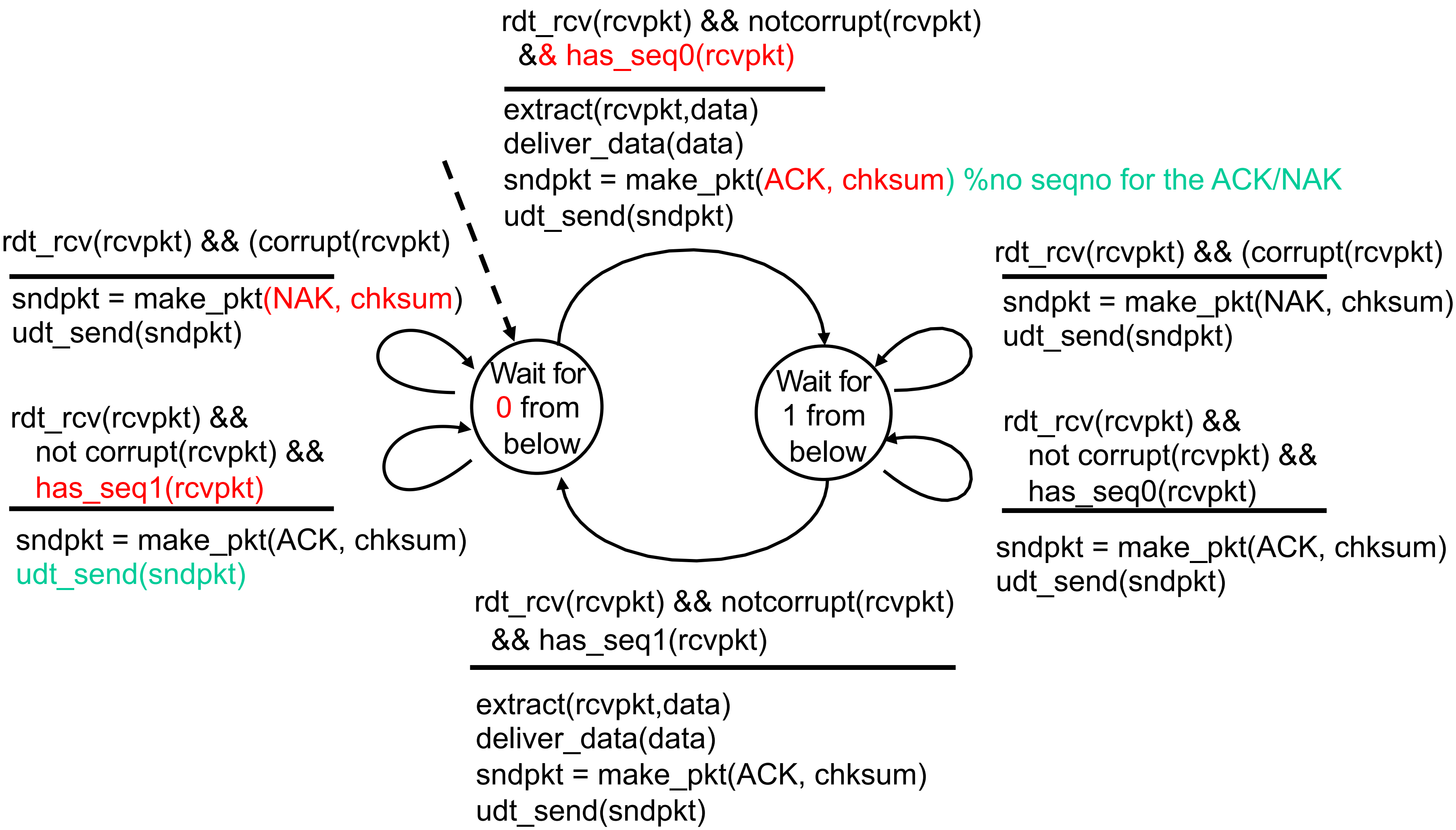
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

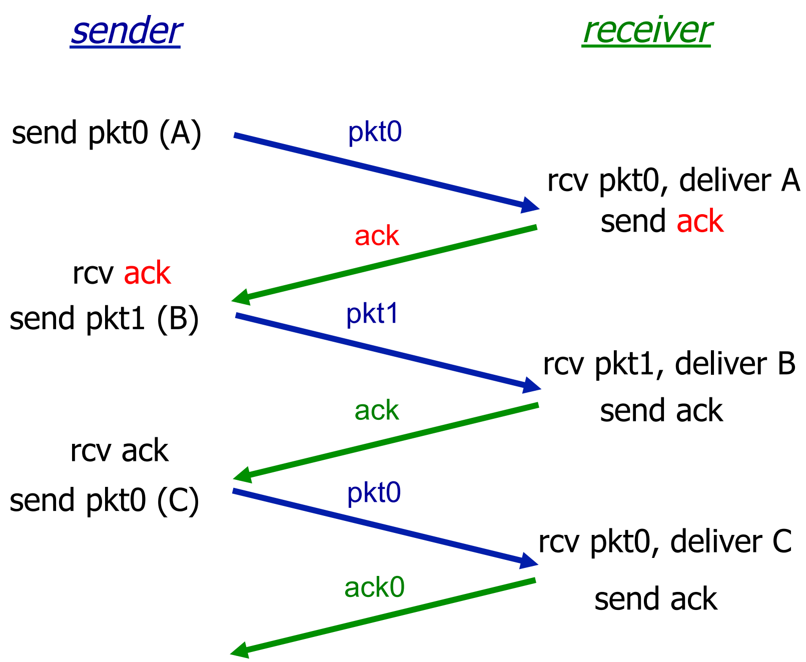
sender:

- must check if received ACK/NAK corrupted
- seq # added to pkt
  - two seq. #'s (0,1) will suffice for stop&wait
  - state indicates whether sender retransmits previous or transmits new pkt
  - twice as many states

receiver:

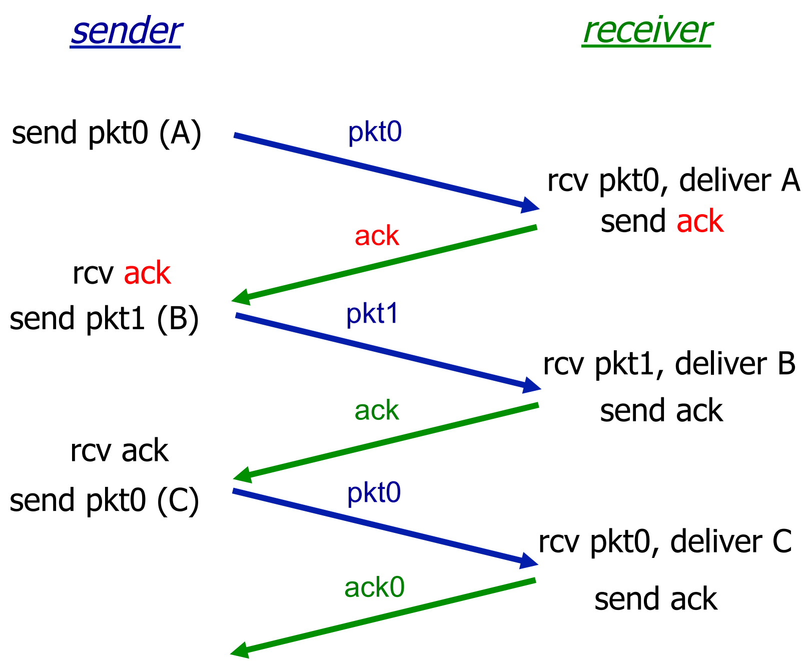
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
  - note: receiver can *not* know if its last ACK/NAK received OK at sender
- ACKs/NAKs don't need seq# for channel that don't drop packets
  - Always refer to most recently sent pkt
  - Seqnos necessary in ACK-only protocols and when channel drops packets

Rdt2.1 in action

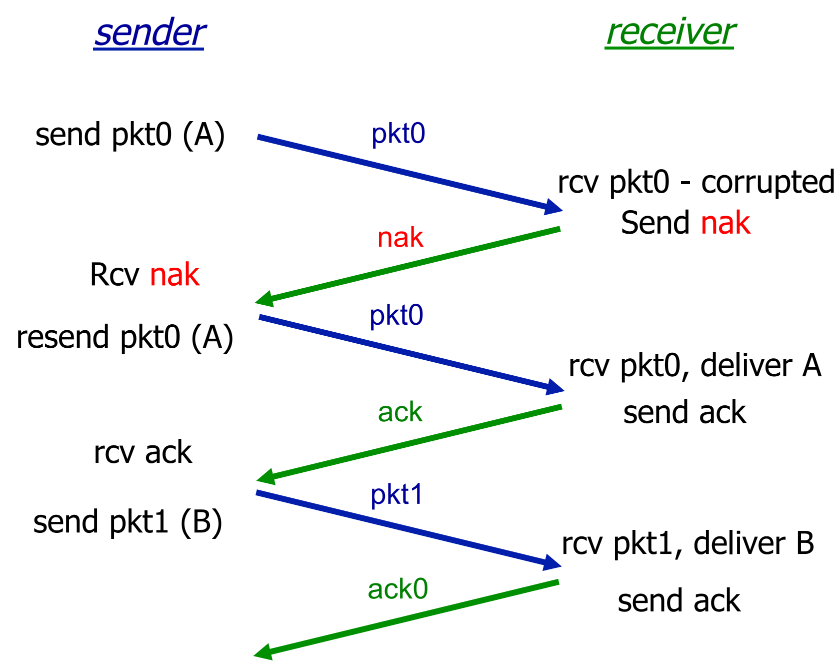


(a) No message or ACK corrupted

Rdt2.1 in action - continued

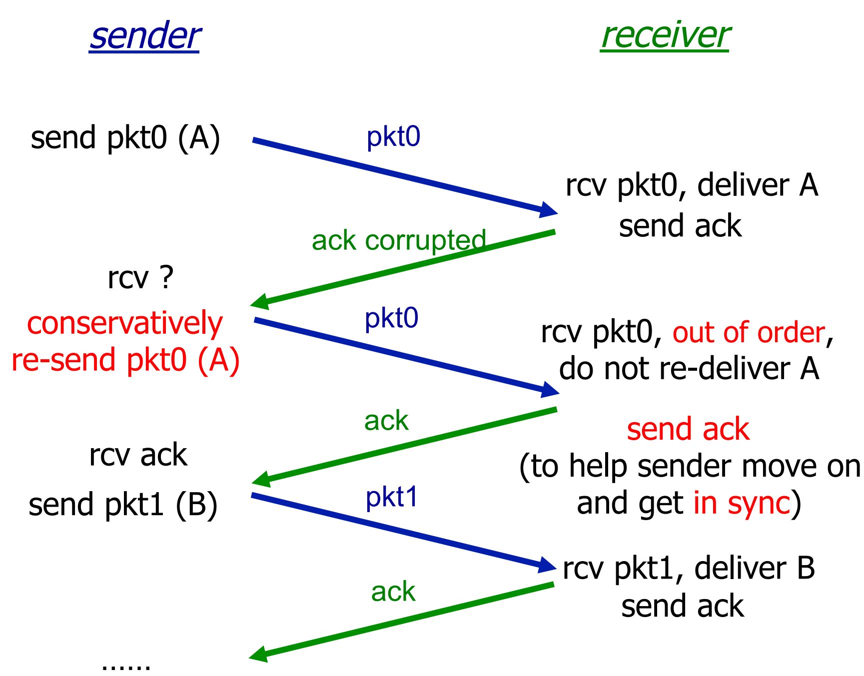


(a) No message or ACK corrupted

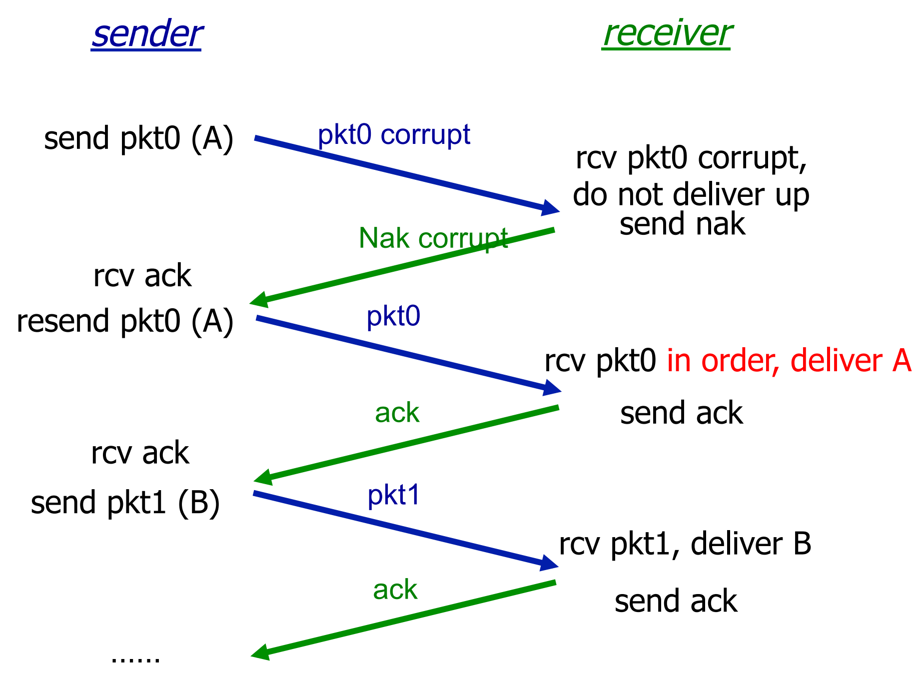


(b) Message corrupted, feedback ok

Rdt2.1 in action - continued



(c) Message ok, ACK corrupted

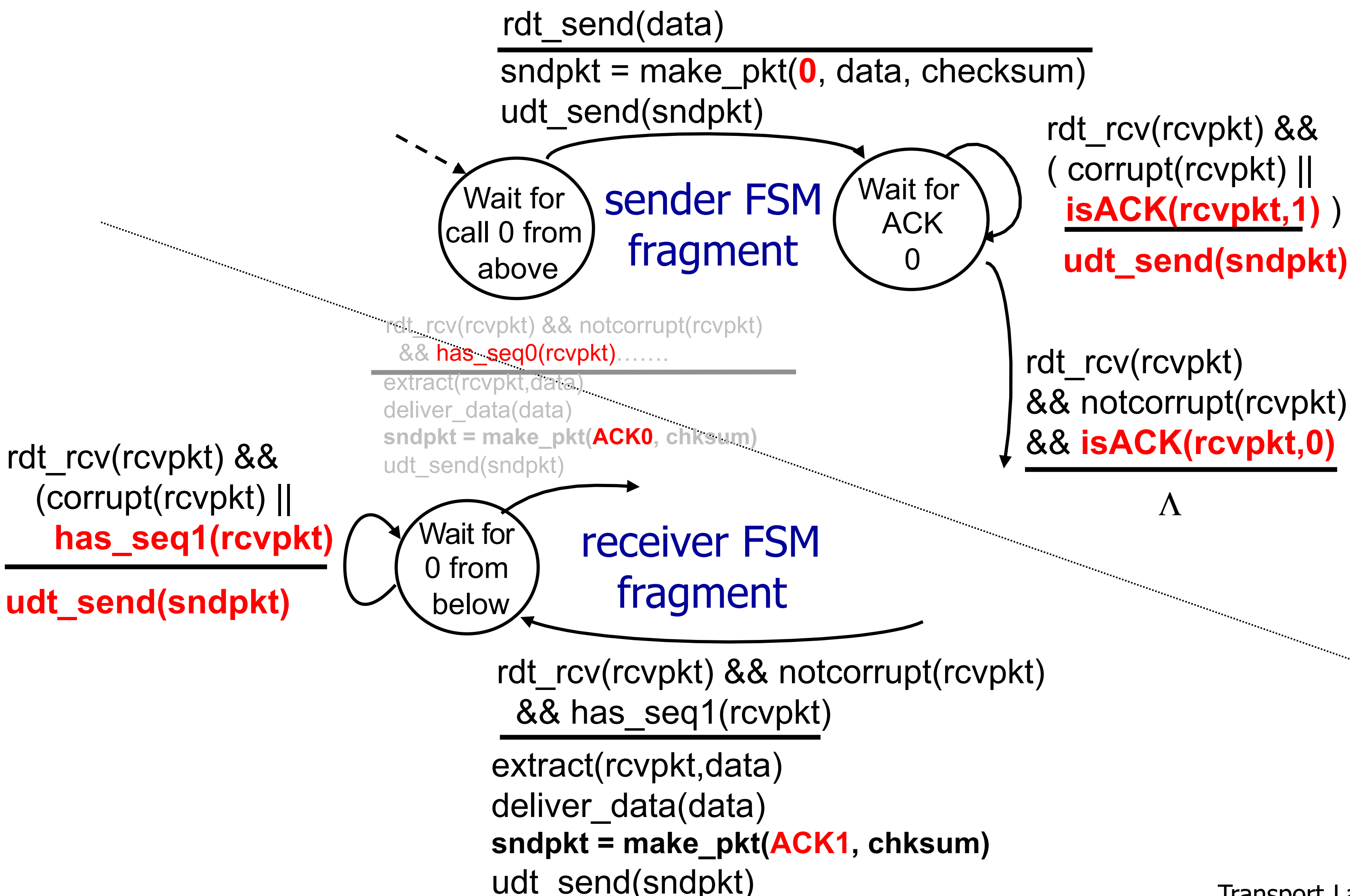


(d) Message corrupted, NAK corrupted

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 (=“duplicate ACK”)
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

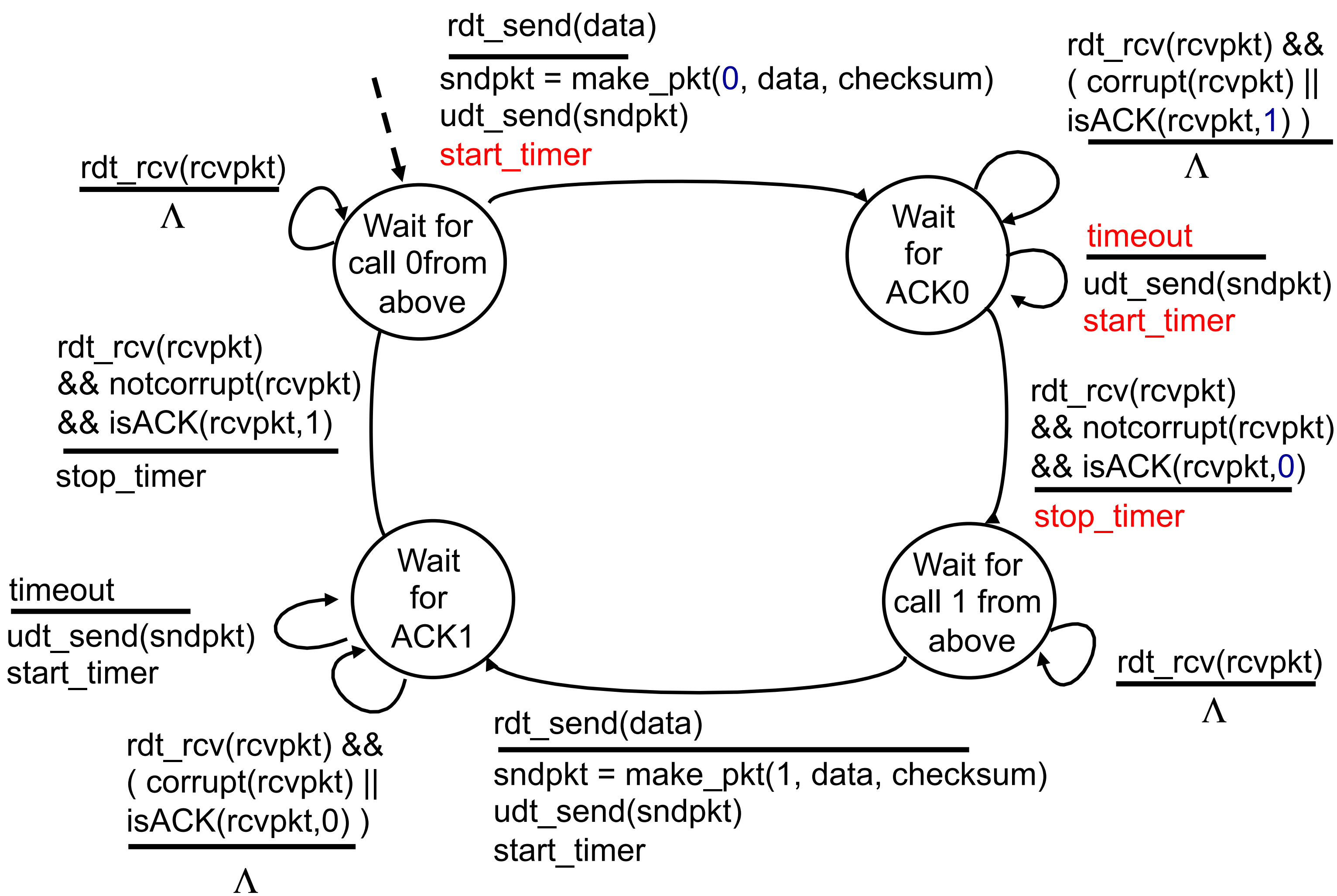
new assumption:
underlying channel can also lose packets (data, ACKs)

- How to detect loss?
- What to do in case of loss?
- checksum, seq. #, ACKs, retransmissions will be of help here, but not enough [why?]

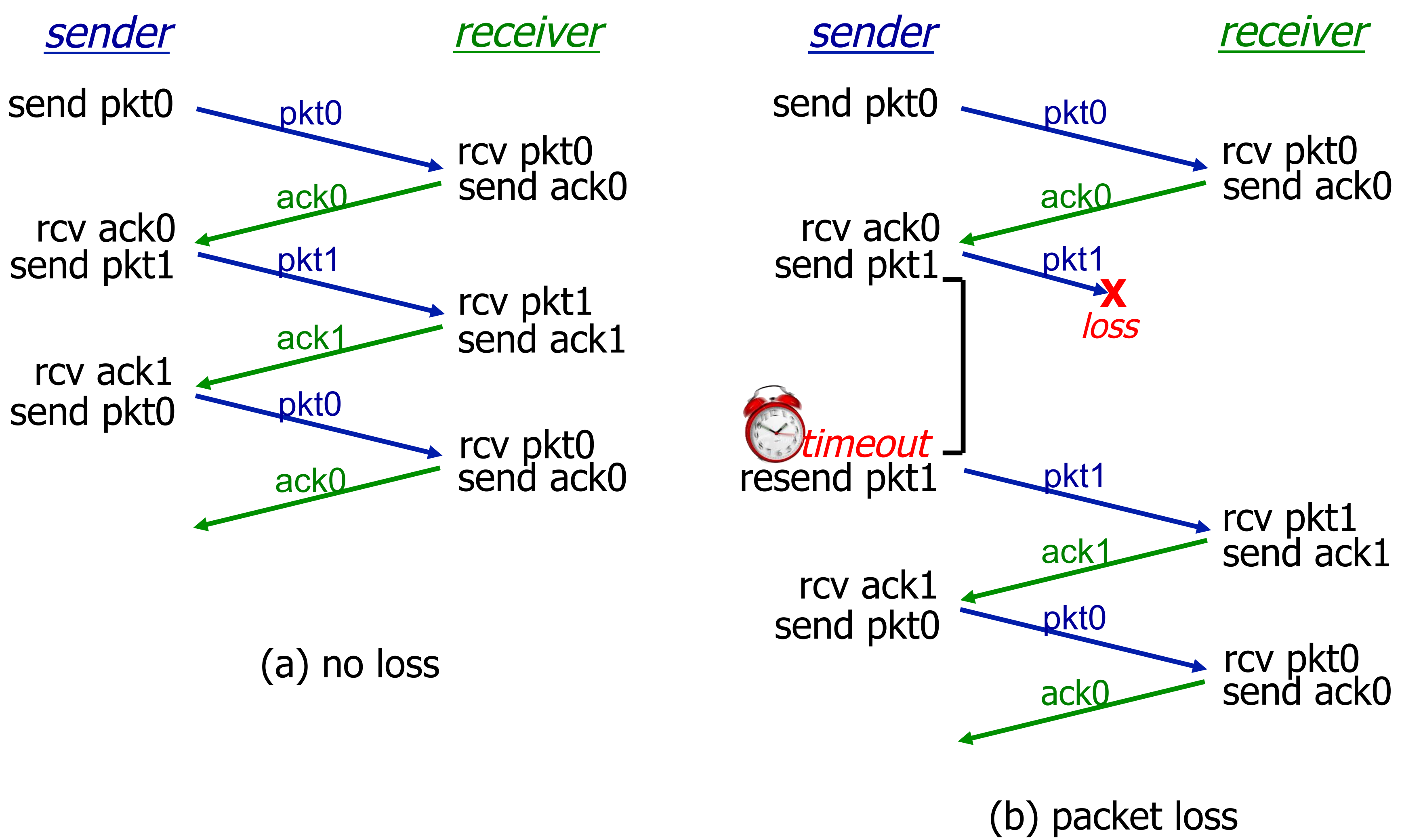
approach: detect loss at server with Timeout

- 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 pkt being ACKed
- requires countdown timer

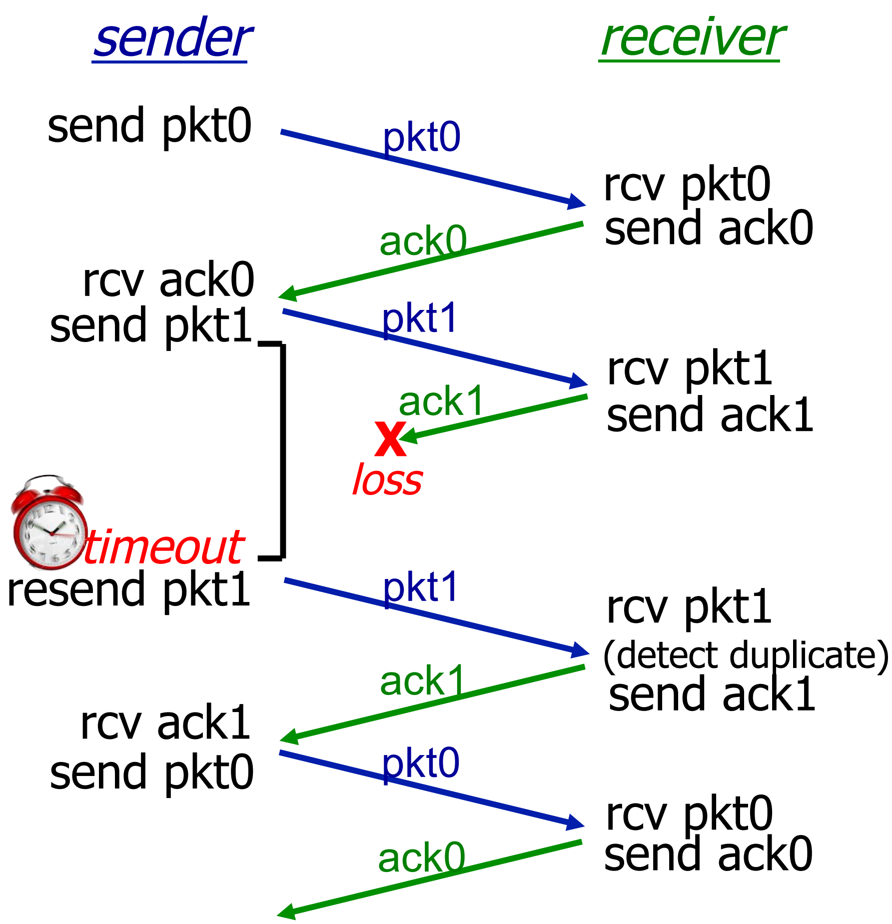
rdt3.0 sender



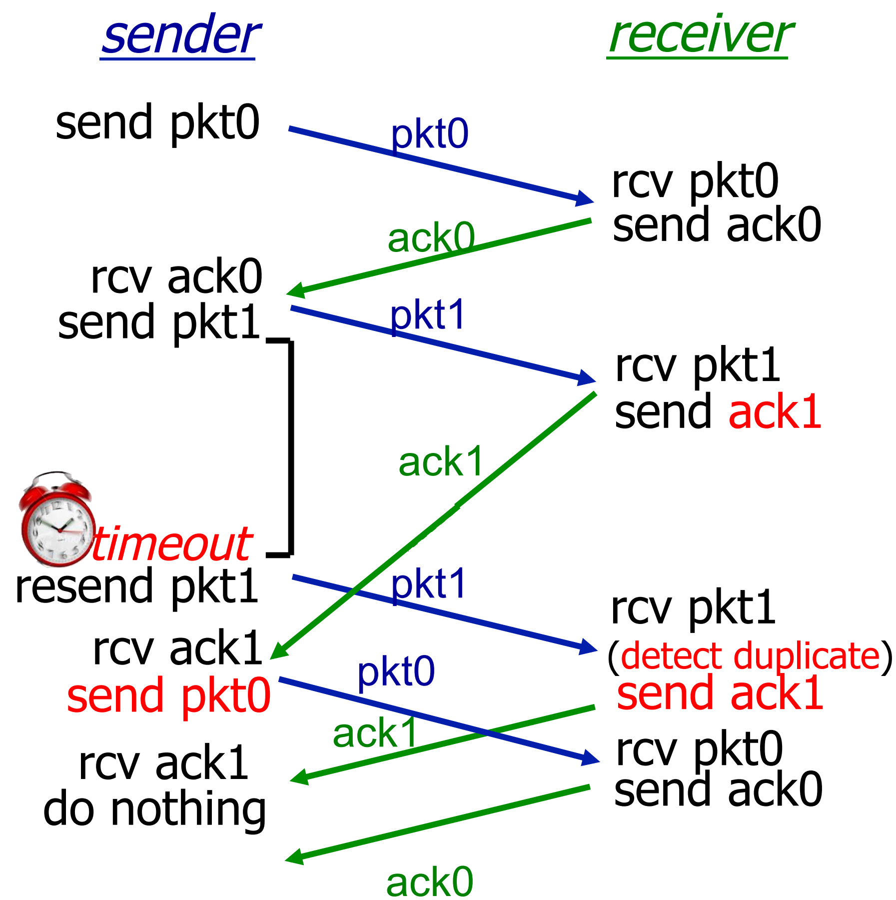
rdt3.0 in action



rdt3.0 in action



(c) ACK loss

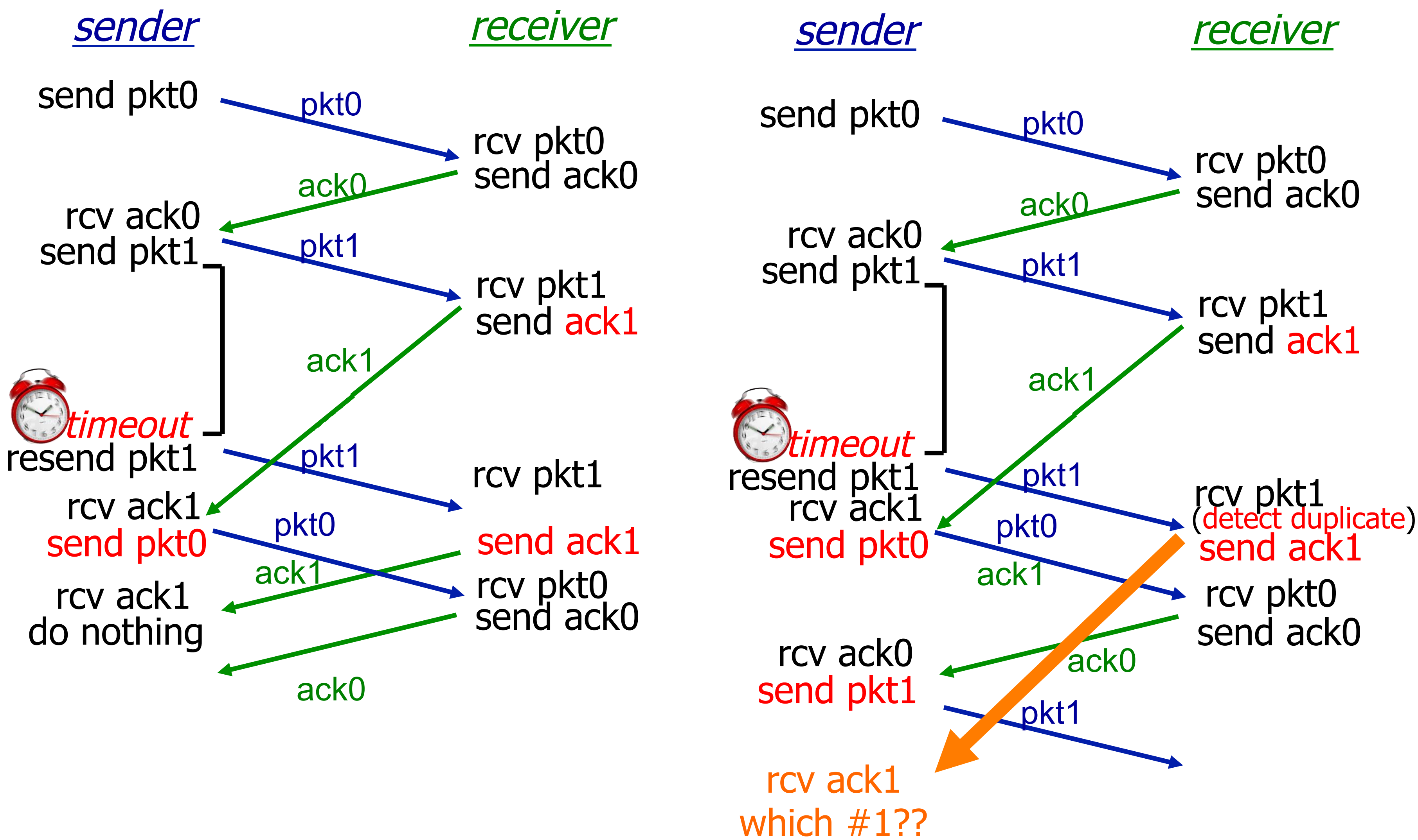


(d) premature timeout/ delayed ACK

Summary of RDT mechanisms

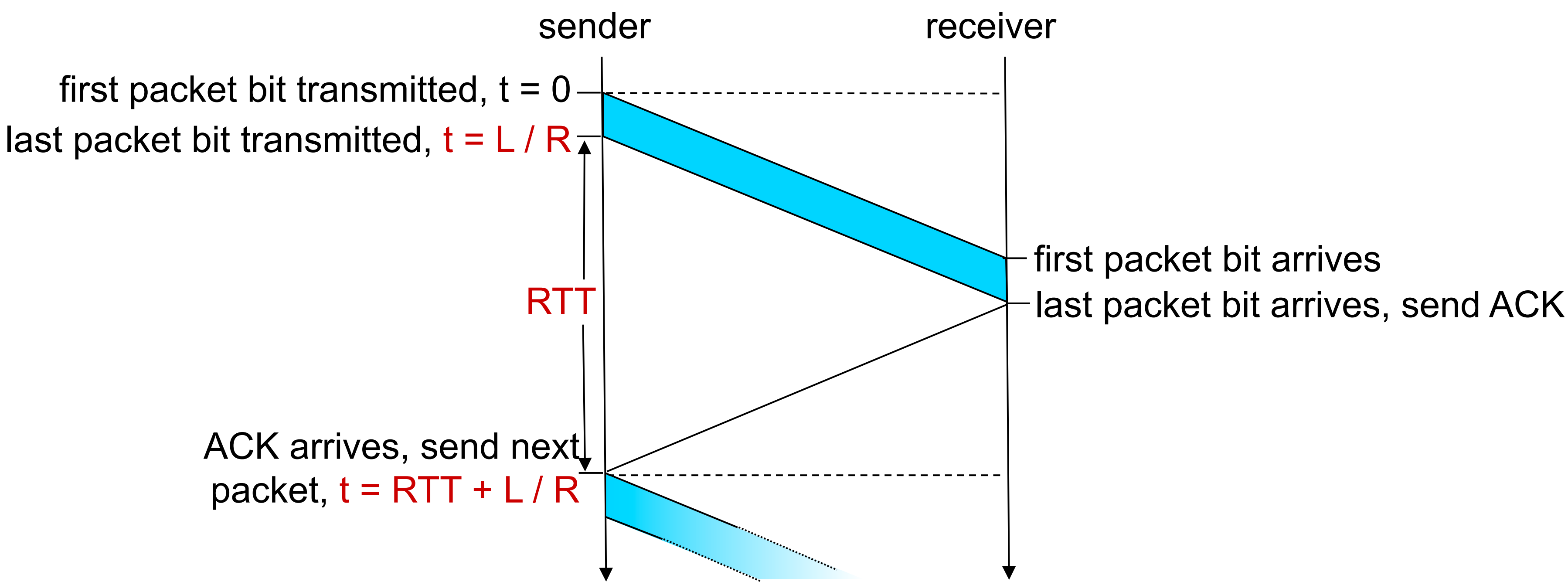
- Corrupted bits (RDT 2.*)
  - Detection via checksum
- Packet loss (RDT 3.*)
  - Detection mechanisms
    - Feedback: ACK, NAK (NAK-free possible using duplicate ACKs)
    - No feedback: Timeout
  - Recovery mechanisms
    - Retransmission
    - Sequence Numbers (to detect duplicates): choose seq.no
  - Retransmission protocols
    - Window=1: stop-and-wait
    - Window>1: GBN, SR
    - Sender and receiver state can be out of sync.
- Reordering in the network?
  - Sequence numbers
  - Reordering looks like loss: may trigger retransmissions in TCP

rdt3.0: delayed ack Q revisited



A: possible only if network reorders packets

rdt3.0: stop-and-wait operation



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

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

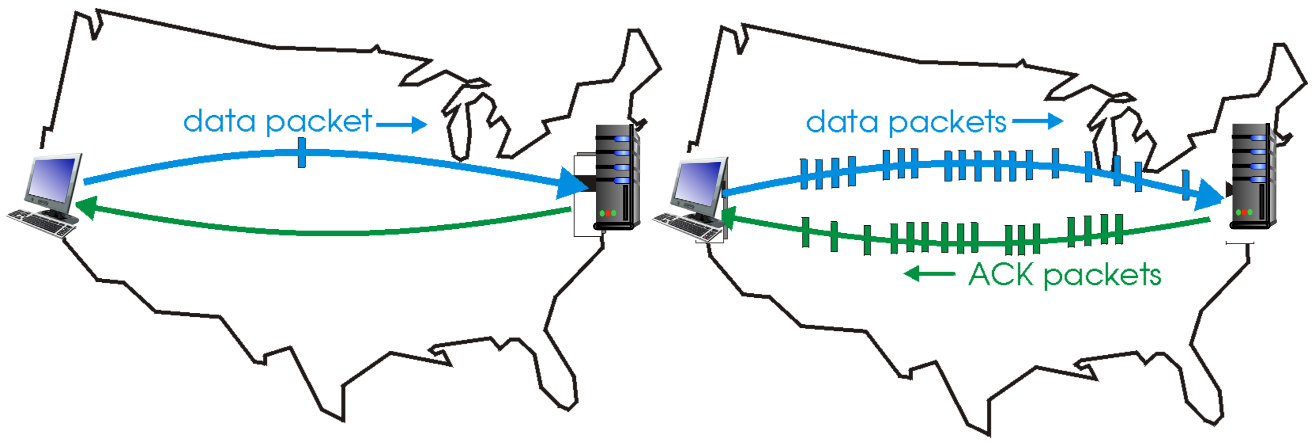
- U $_{sender}$ *utilization* (or "efficiency" in your HW): fraction of time sender busy sending

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

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
  - Example of a (bad) network protocol limits use of physical resources (fast link)!

Pipelined protocols

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

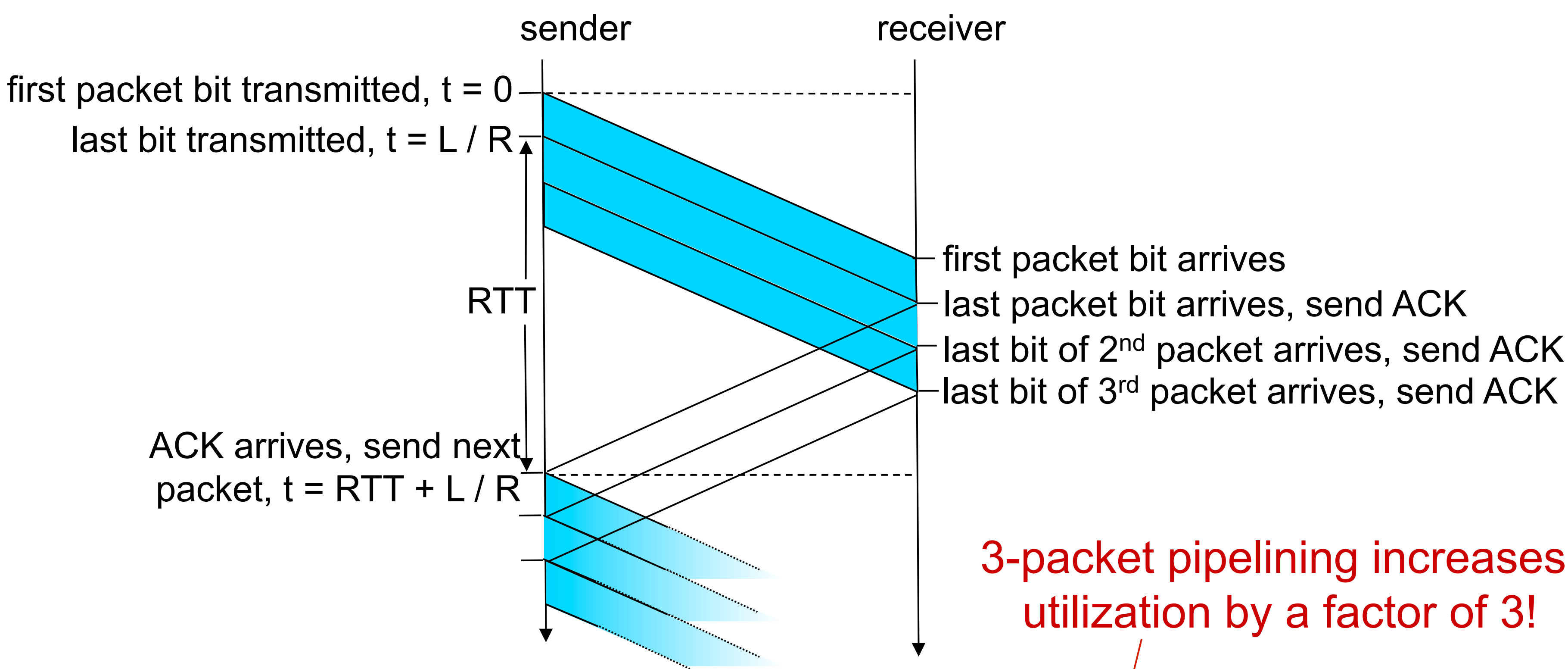


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization



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

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

Q: What is the best choice of the "window" of pipelined messages?

A: RTT/(L/R+1) keeps the channel always busy

Pipelined protocols: overview

Go-back-N:

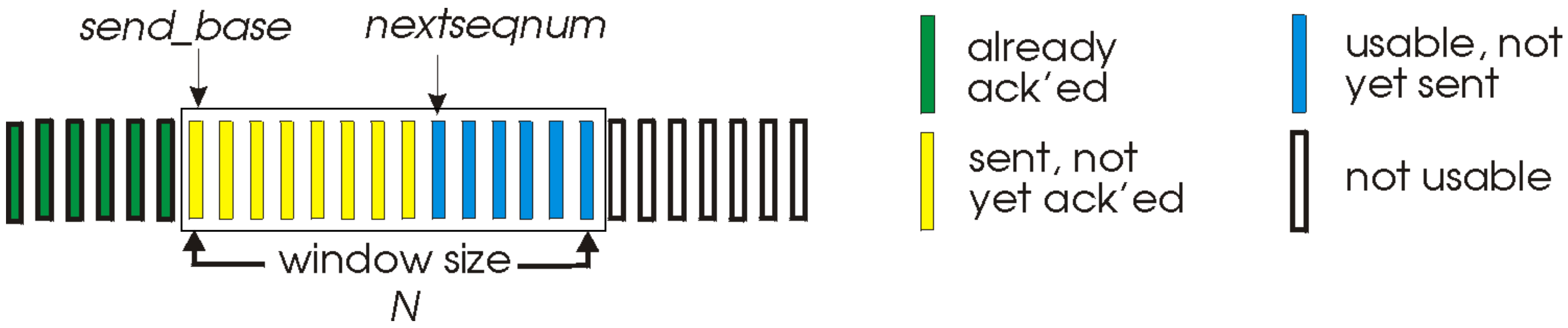
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there is a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

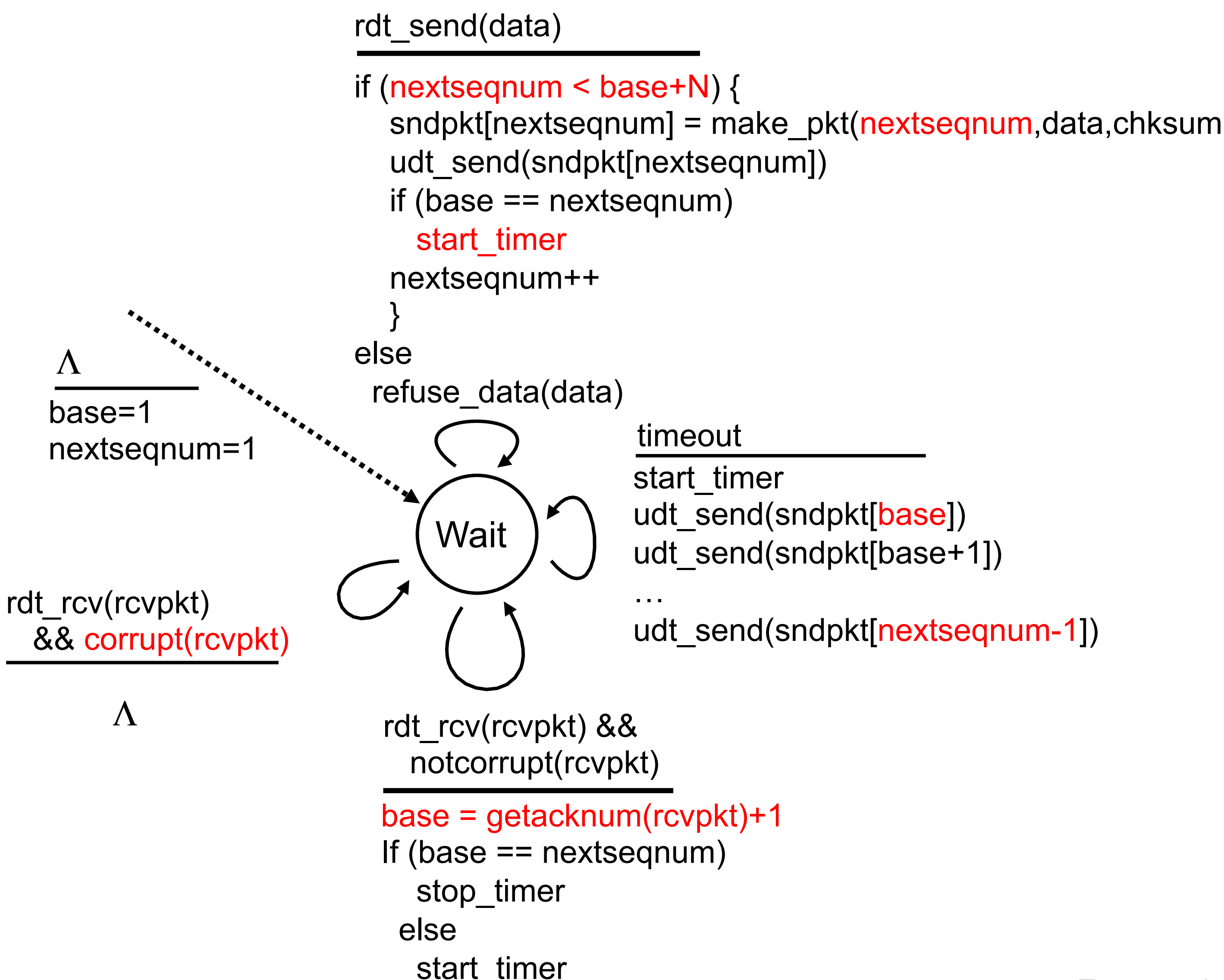
Go-Back-N: sender

- k-bit sequence # in pkt header
- "sliding window" of up to N, consecutive unack'ed pkts allowed

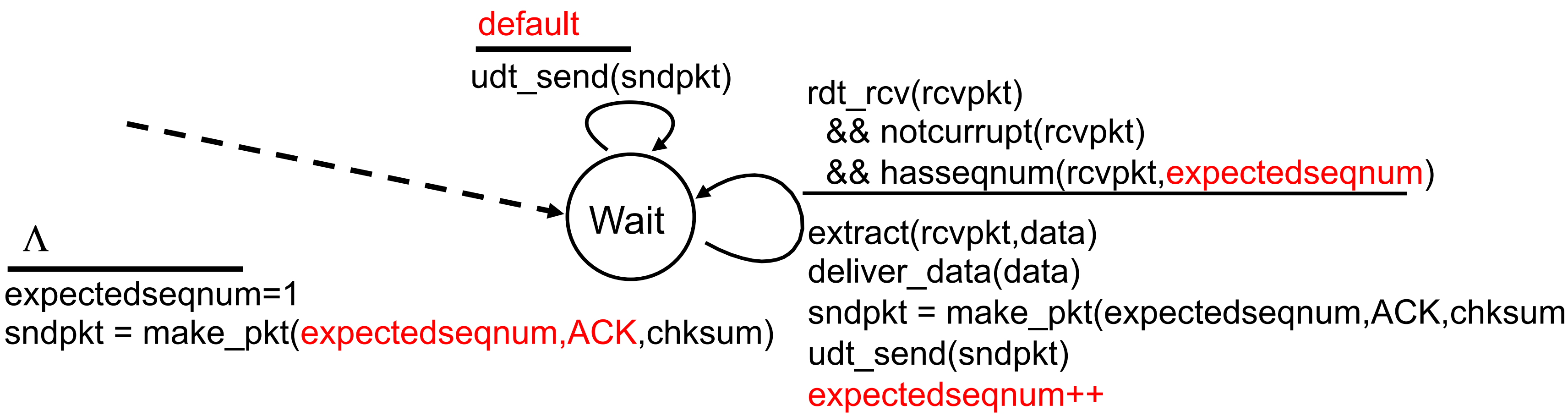


- Send_base: oldest in-flight packet (n): first yellow packet
- ACK(m): ACKs all pkts up to, including seq # m - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- *timeout(n)*: keep timer for n and retransmit packet n and all higher seq # pkts in window – i.e., all yellow packets

GBN: sender extended FSM

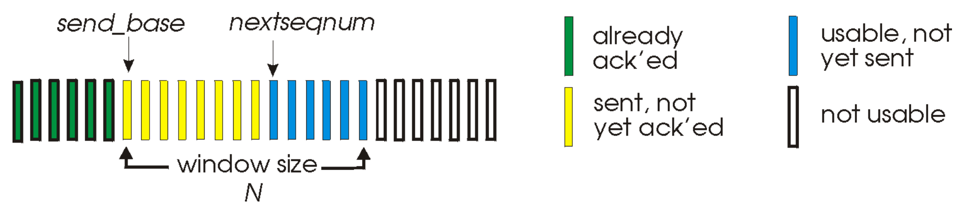


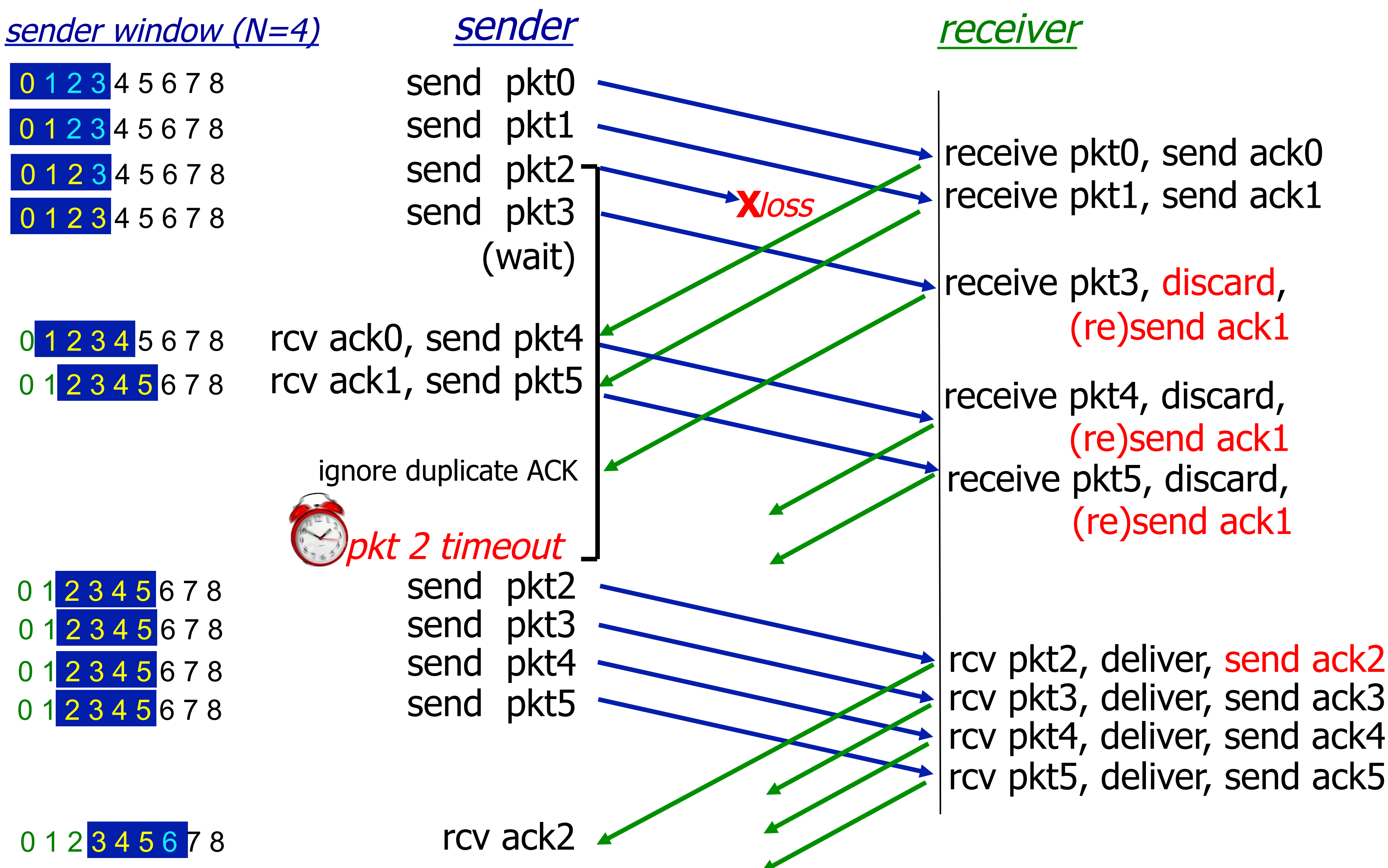
GBN: receiver extended FSM



- Keeps track of correctly-received pkt with highest *in-order* seq #:
  - if sender-receiver in sync, this should be the last green packet in Sender's window
- ACK-only: always send ACK for that particular packet
  - may generate duplicate ACKs
  - need only remember one number **`expectedseqnum`**
    - If sender-receiver in sync, this should be the 1st yellow packet in Sender's window
- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering! Why?*
  - re-ACK pkt with highest in-order seq #
- Really simple receiver!

GBN in action

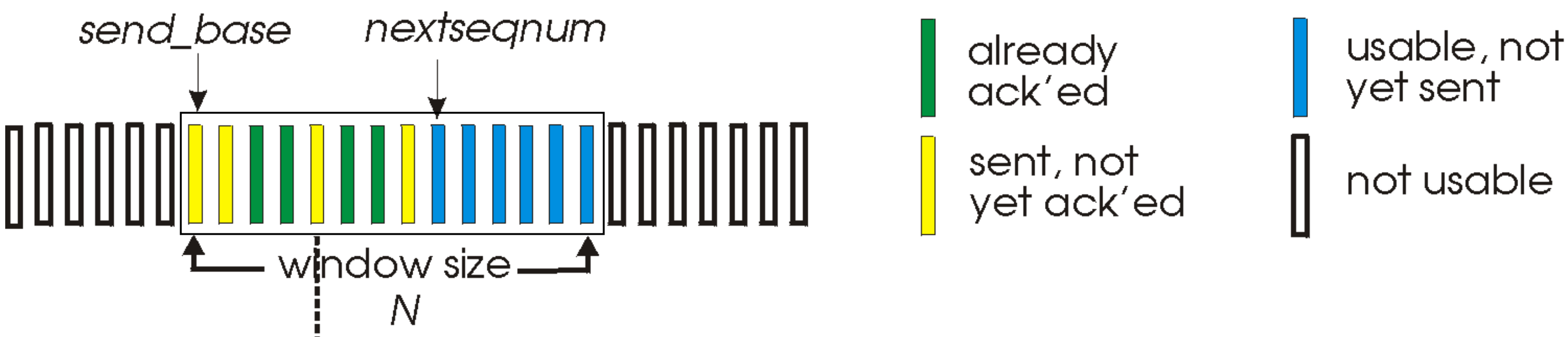




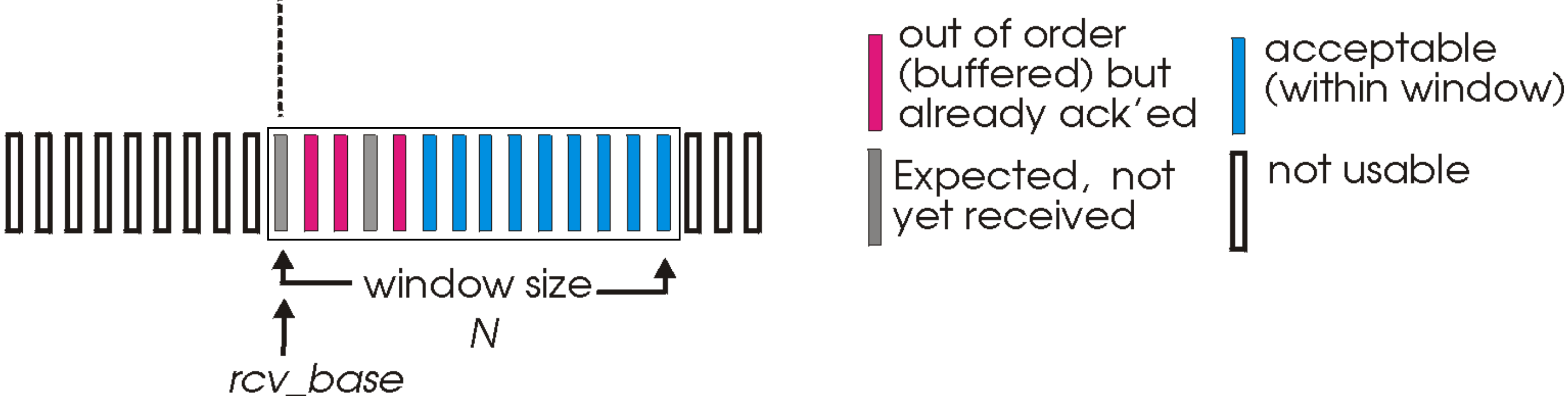
Selective Repeat (SR)

- Receiver *individually* acknowledges all correctly received pkts
  - whether they are received in-order or out-of-order
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only retransmits un-ACKed pkts
  - sender maintains timer for each unACKed pkt
- Both sender and receiver maintain windows
  - Sender window
    - *N* consecutive seq #' s
    - limits seq #s of sent, unACKed pkts
  - Receiver window
    - *N* consecutive seq #' s
    - Limits seq#s of buffered out-of-order packets

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective Repeat (FSM not shown)

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 n in [rcvbase, rcvbase+N-1]

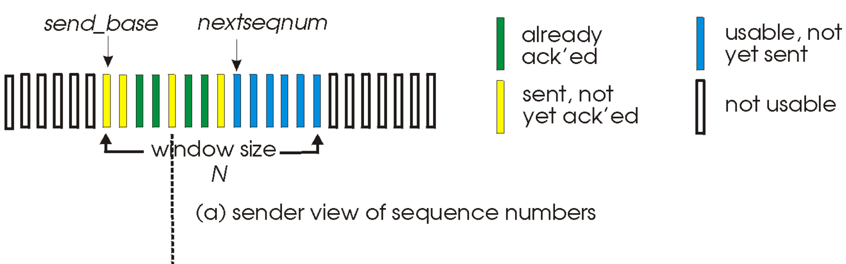
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver all buffered, in-order pkts), advance window to next not-yet-received pkt

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

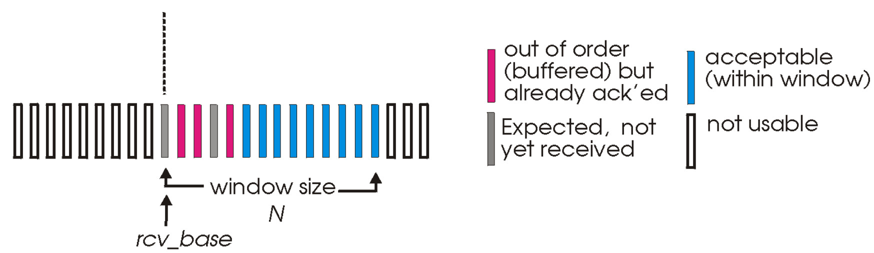
- ACK(n) [Why not ignore?]

otherwise:

- ignore

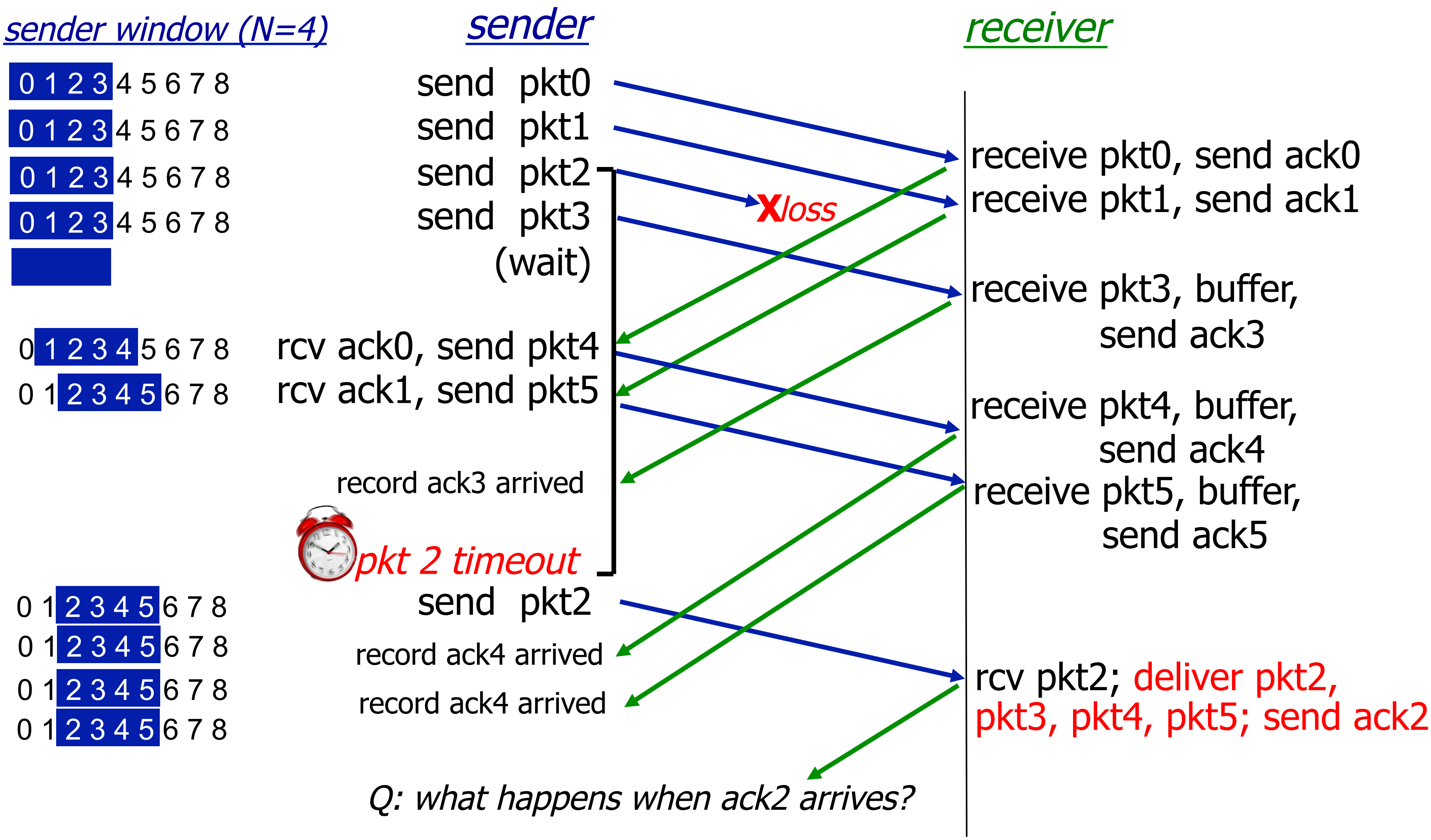


(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective repeat in action



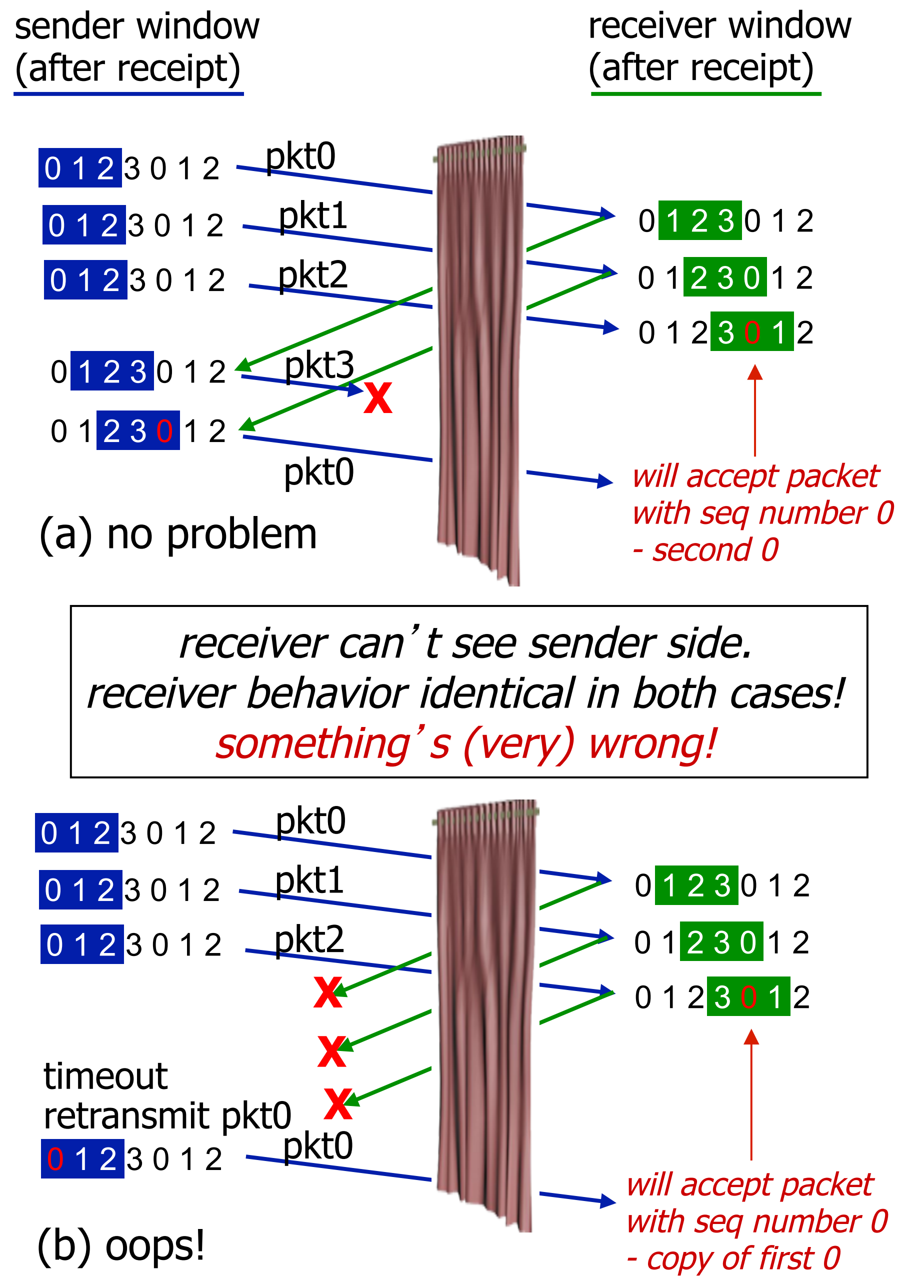
Lack of sync + finite seq#: ambiguity

Example:

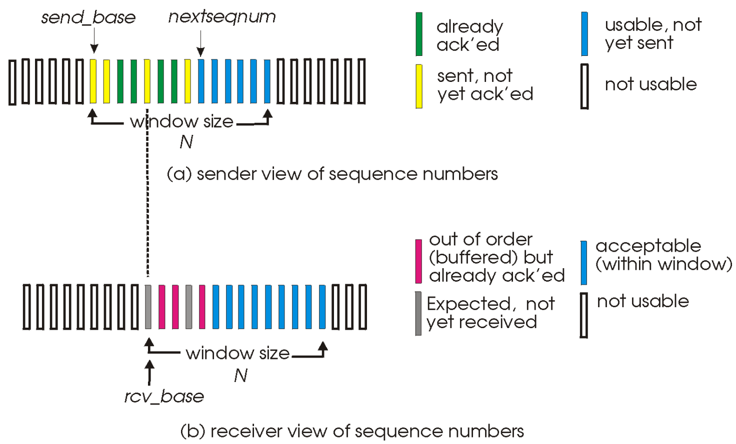
- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

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

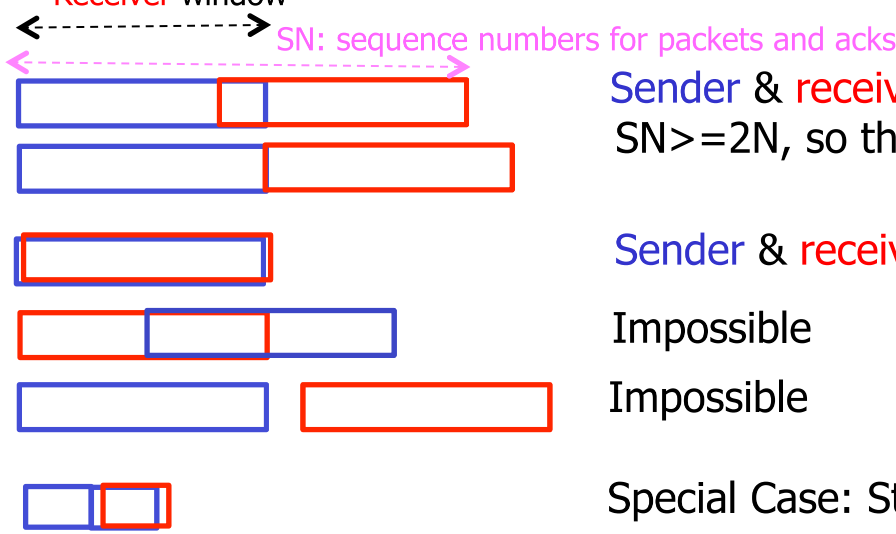
A: SN>=2N because sender and receiver window must have overlap of at least 1



SR: sender, receiver windows & Seq No



N= Sender window
= Receiver window



Sender & receiver windows overlap. At most consecutive.
SN>=2N, so that seqnums for packets and acks are unique

Sender & receiver windows in perfect sync

Impossible

Impossible

Special Case: Stop and Wait: Sender W=2, Receiver W=1

Go-back-N vs SR: Mechanisms

Go-back-N:

- Sender can have up to N unack'ed packets in pipeline
  - sliding window
- Rcvr sends *cumulative* ack for last in-order packet
  - maintains expectedseqnum
  - doesn't accept or ack out-of-order packet
- Sender maintains timer for oldest unacked packet
  - if timer expires, retransmit all unack'ed packets

Selective Repeat:

- Sender can have up to N unack'ed packets in pipeline
  - sliding window
- Rcvr sends *individual ack* for each packet
  - maintains Rcvr window
  - buffers and acks all packets within Rcvr window
- Sender maintains timer for each unacked packet
  - when timer expires, retransmit only that one unack'ed packet

GBN vs SR: Performance

- Compared to Stop-and Wait
  - They both fill the pipeline
- Loss rate
  - Light loss:
    - SR: selectively retransmits what is needed
    - GBN: a single packet lost causes unnecessary retransmission of all packets in the window
  - Heavy loss
    - GBN: ok
- Complexity:
  - GBN is simpler – less state

Practice GBN vs SR

- Sample Midterm
  - Problem 4
    - https://eee.uci.edu/16s/18105/hws/eecs148-midterm-s15.pdf
- Companion Website
  - http://wps.pearsoned.com/ecs_kurose_compnetw_6/
  - Applets for Ch.3
    - http://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198700.cw/index.html
  - interactive exercises for Ch.3 #2
    - http://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198700.cw/index.html