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

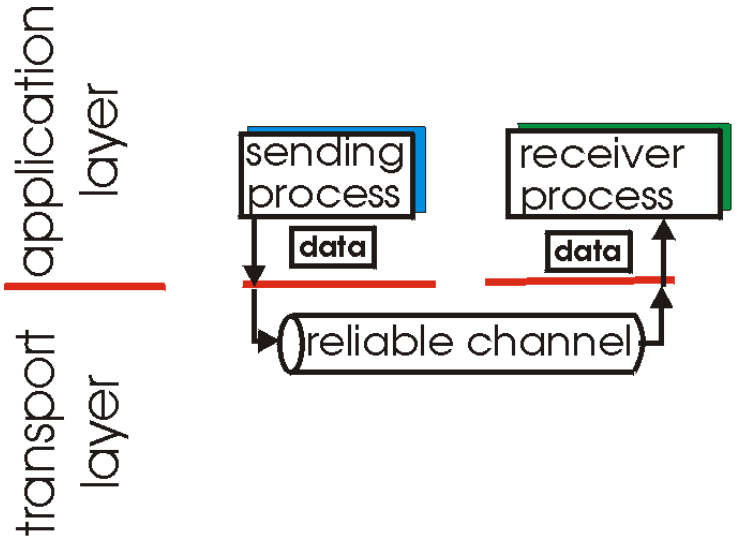
- segment structure
- reliable data transfer
- flow control
- connection management

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!

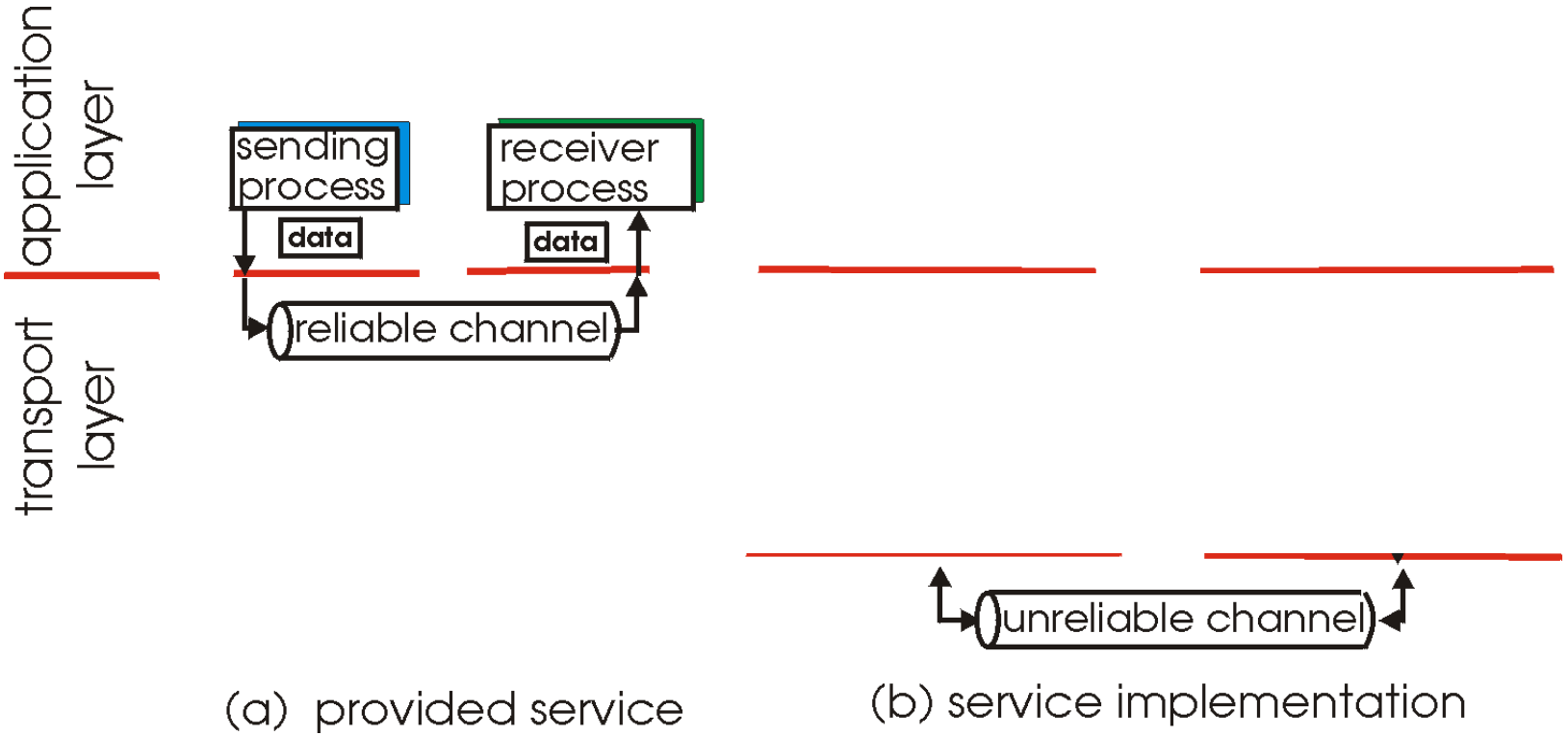


(a) provided service

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

Principles of reliable data transfer

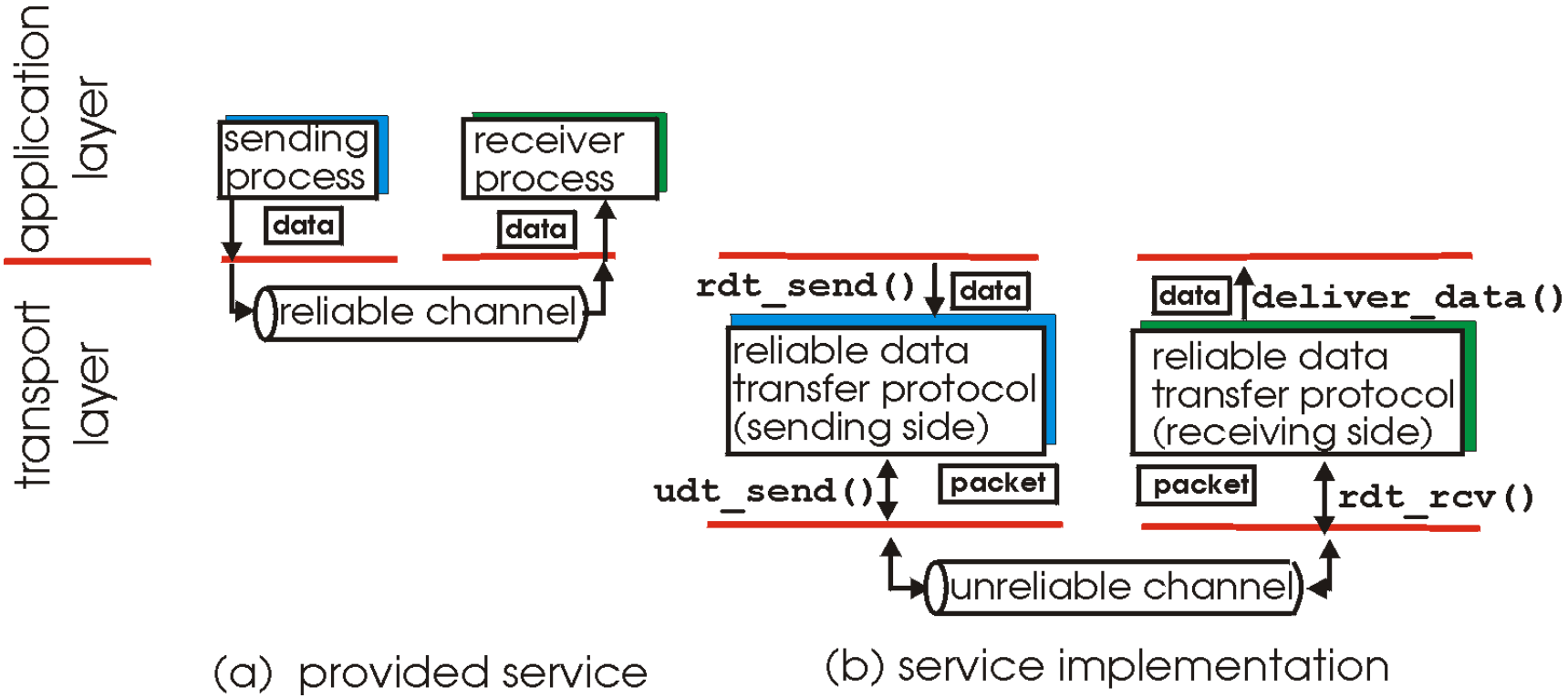
- important in application, transport, link layers
  - top-10 list of important networking topics!



- 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!

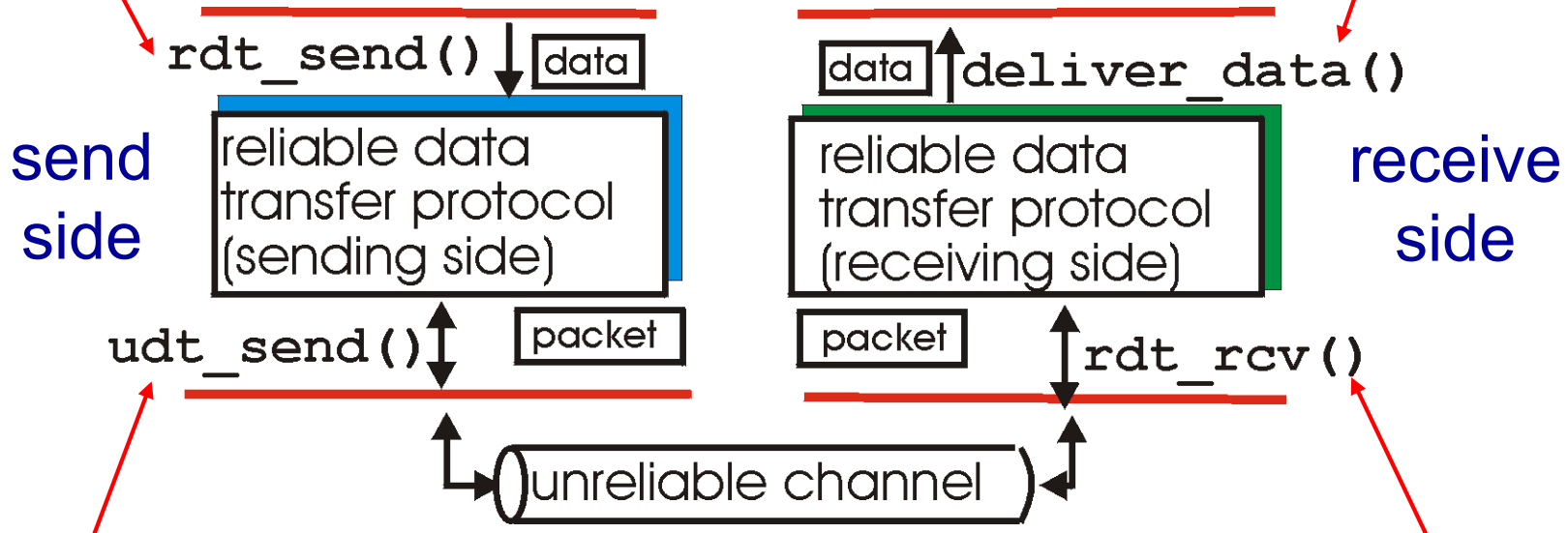


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



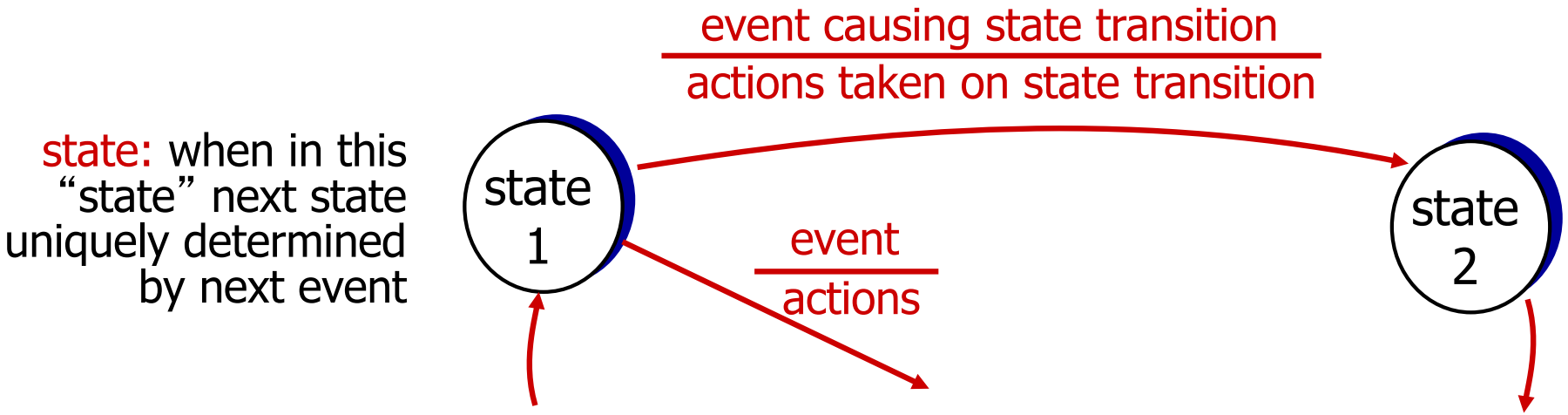
`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

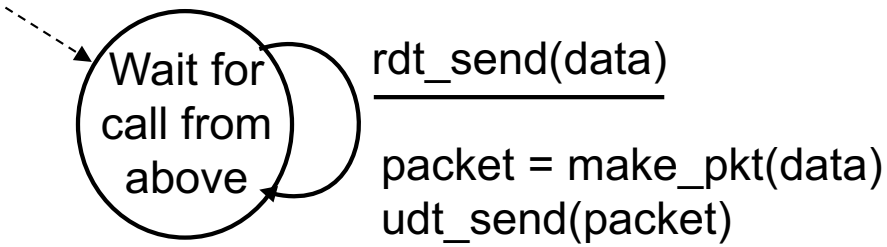
We will:

- 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

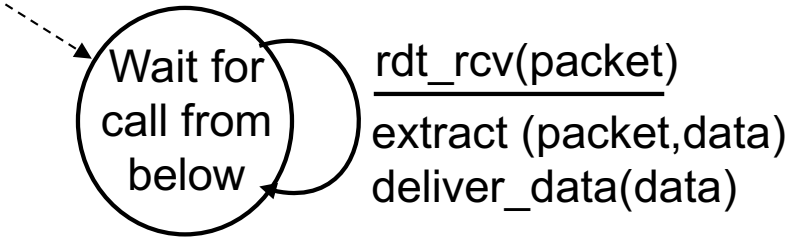


rdt1.0: reliable transfer over a reliable channel

- 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



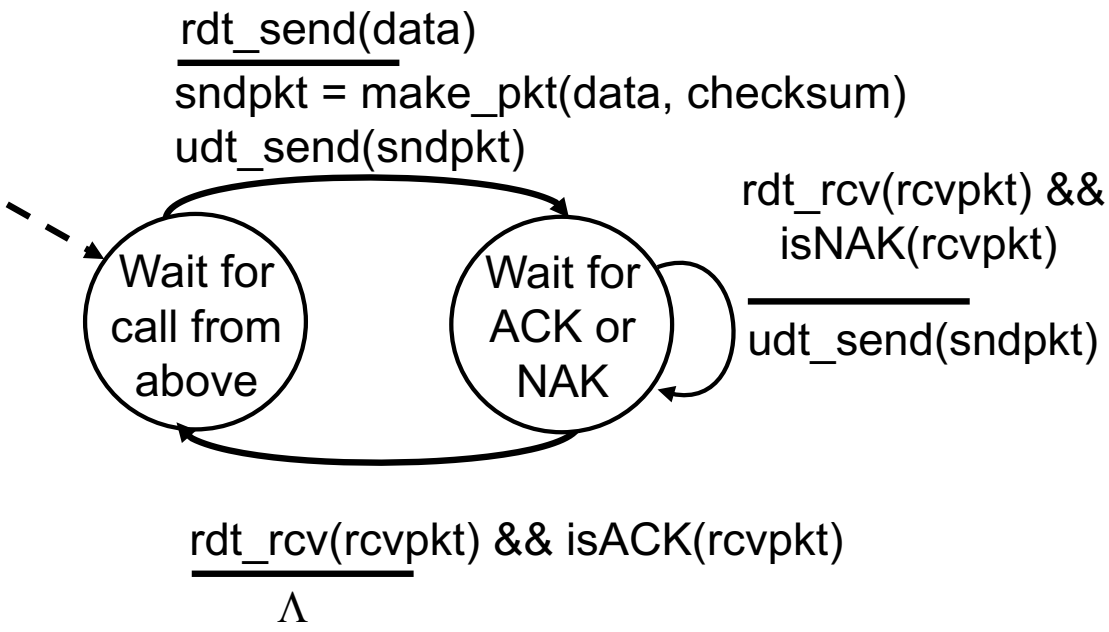
receiver

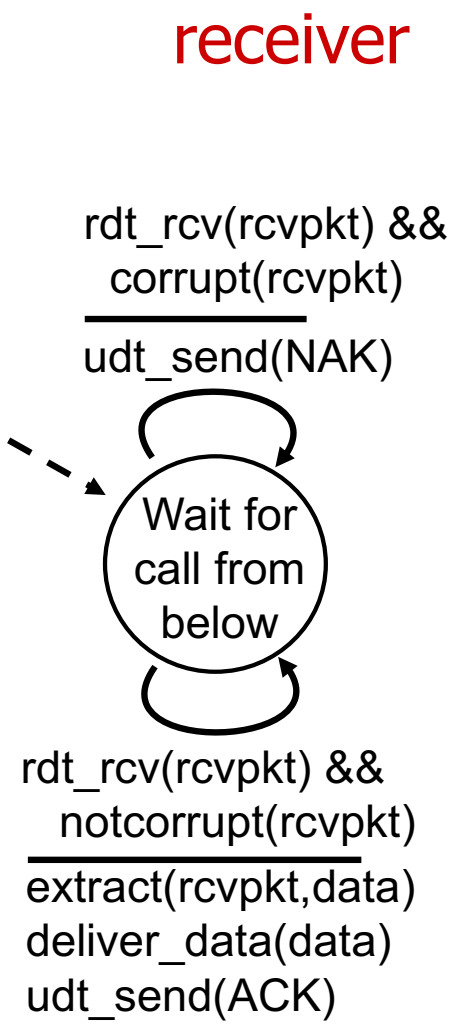
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - *error detection:* e.g., user checksum to detect bit errors

- Q: how to recover from errors?
- A: new mechanisms needed:
  - *feedback (control messages 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

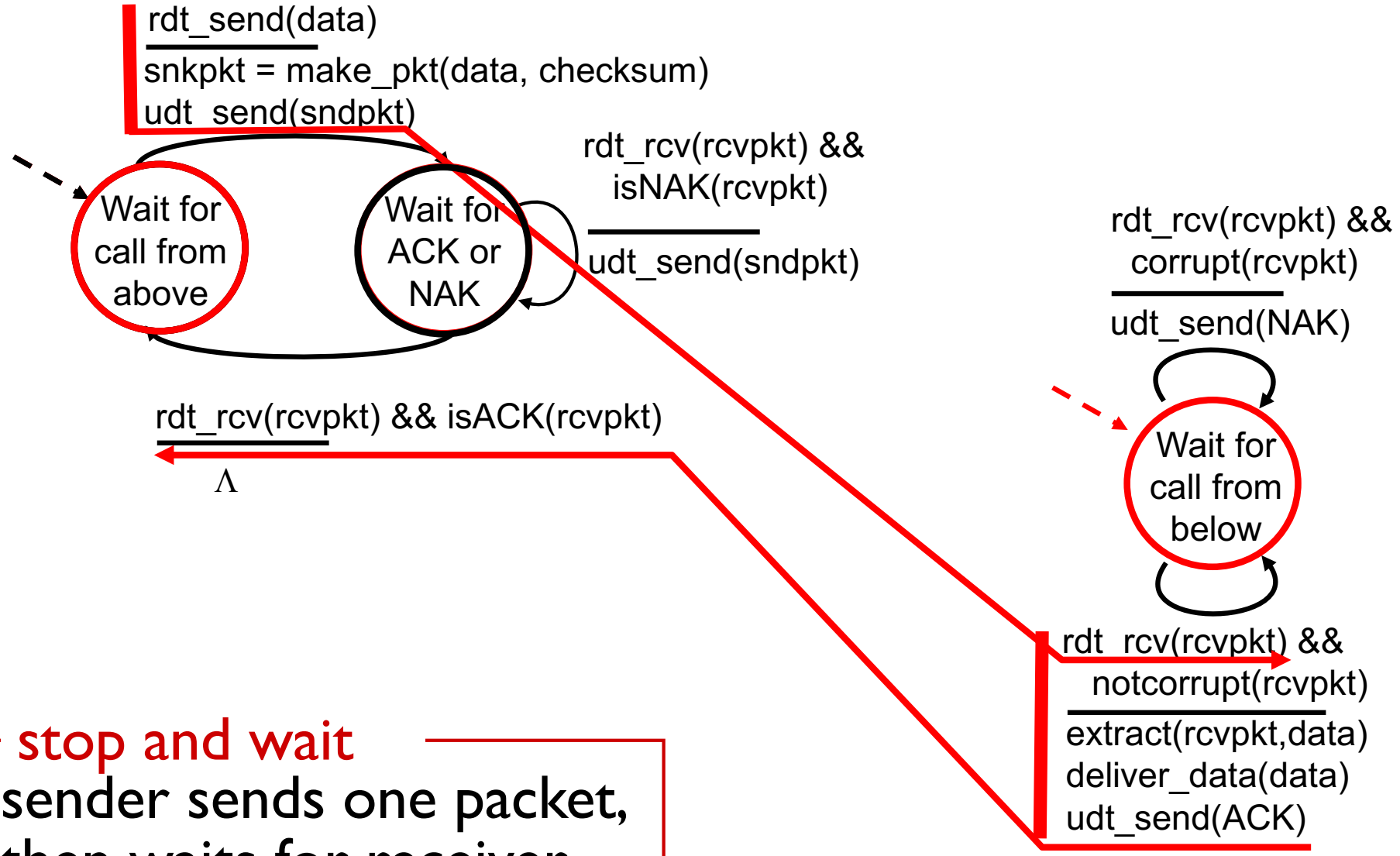




stop and wait

sender sends one packet, then waits for receiver response

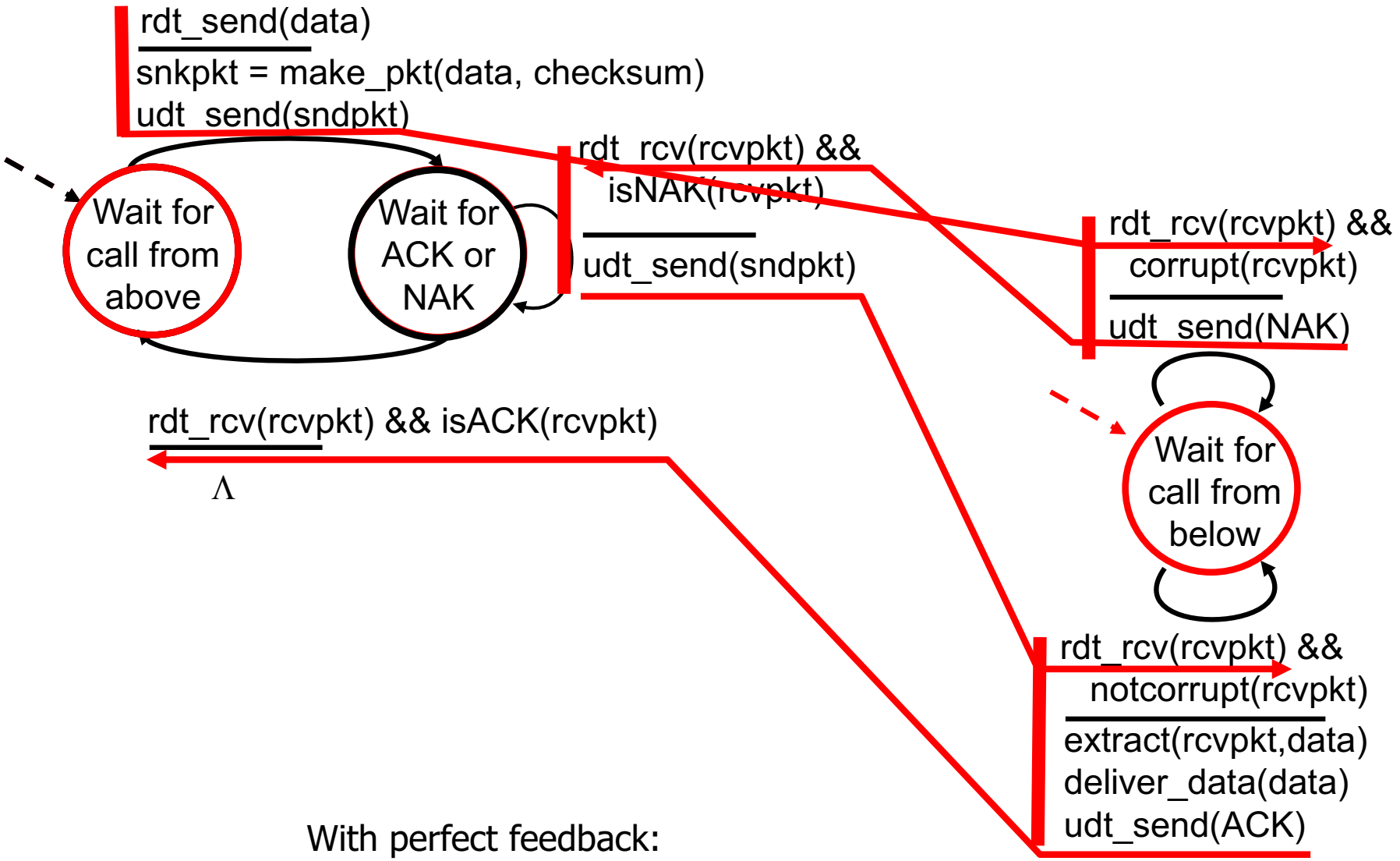
rdt2.0: operation with no errors



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 flaw

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
  - Is packet received or corrupted?
  - should we retransmit the same (may cause duplicates) or send the next (missing one)
- solutions
  - Feedback on feedback…
    - may be corrupted but we can detect that (e.g. through checksum).
  - Error detection+correction
    - Cannot handle lost acks
  - Retransmissions
    - cant just retransmit: possible duplicates

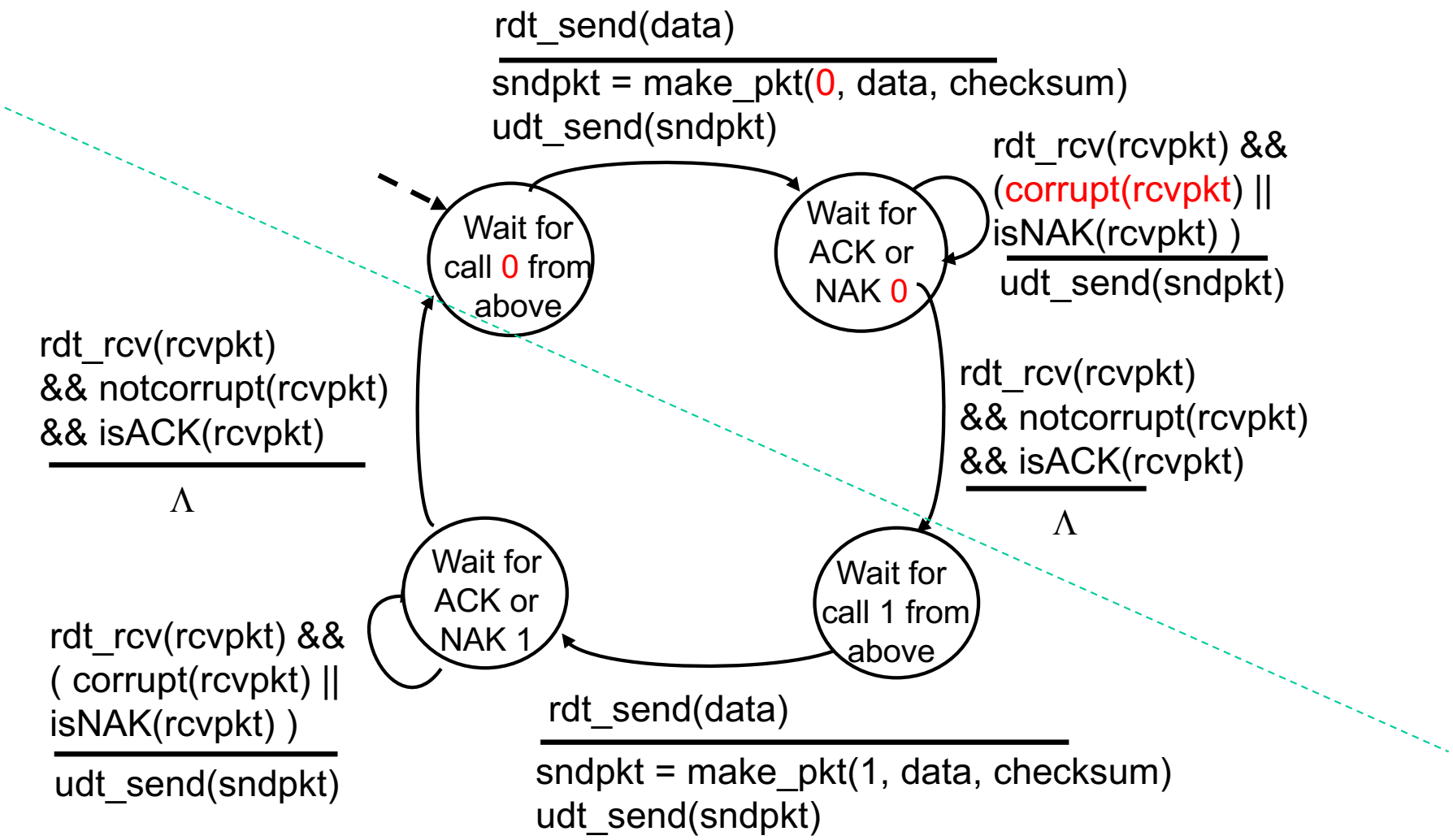
handling duplicates:

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

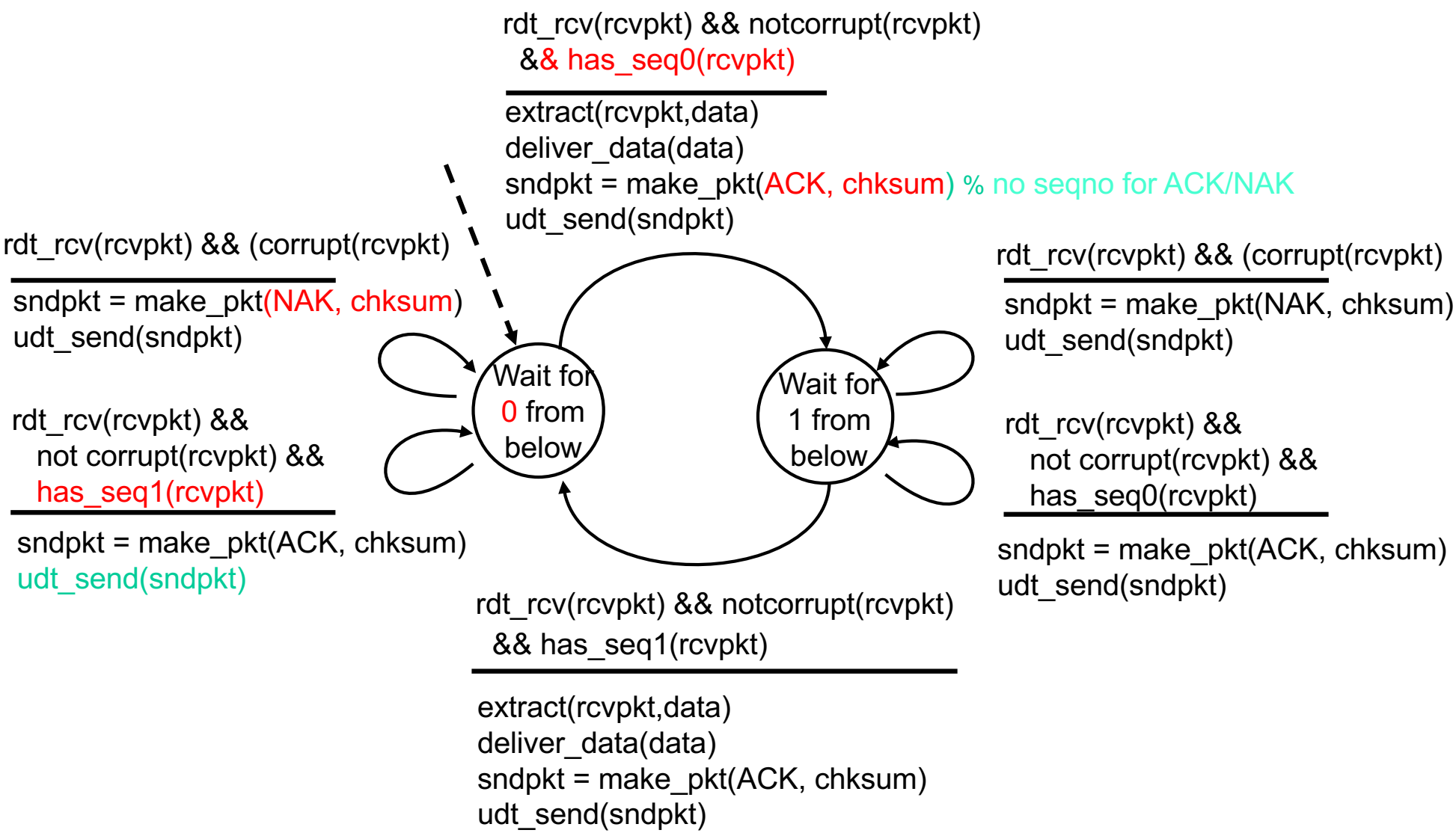
stop and wait

Sender still sends one packet, then waits for receiver response. But now both Sender and Receiver have double the number of states

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

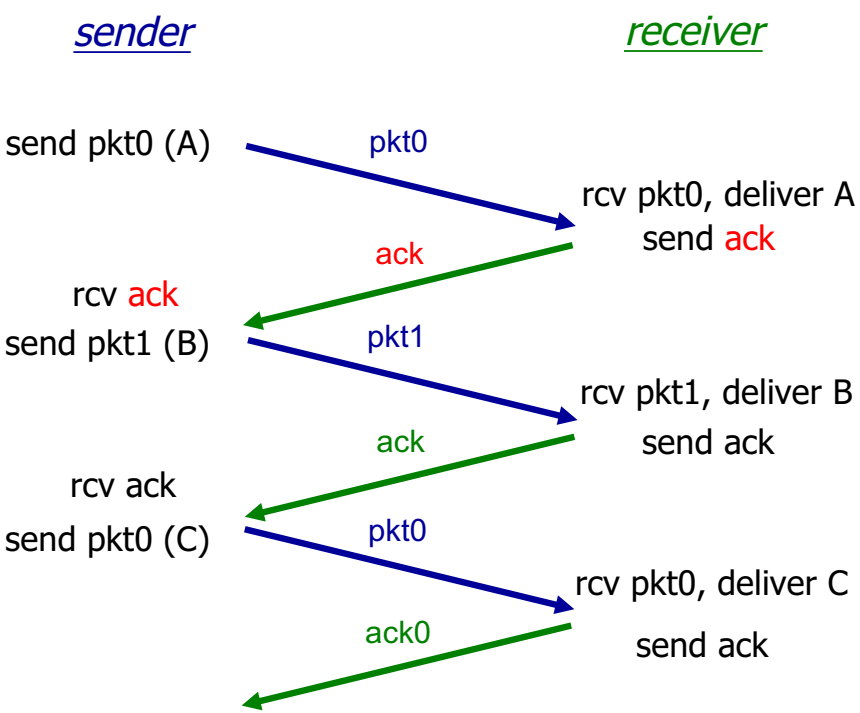
sender:

- 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:

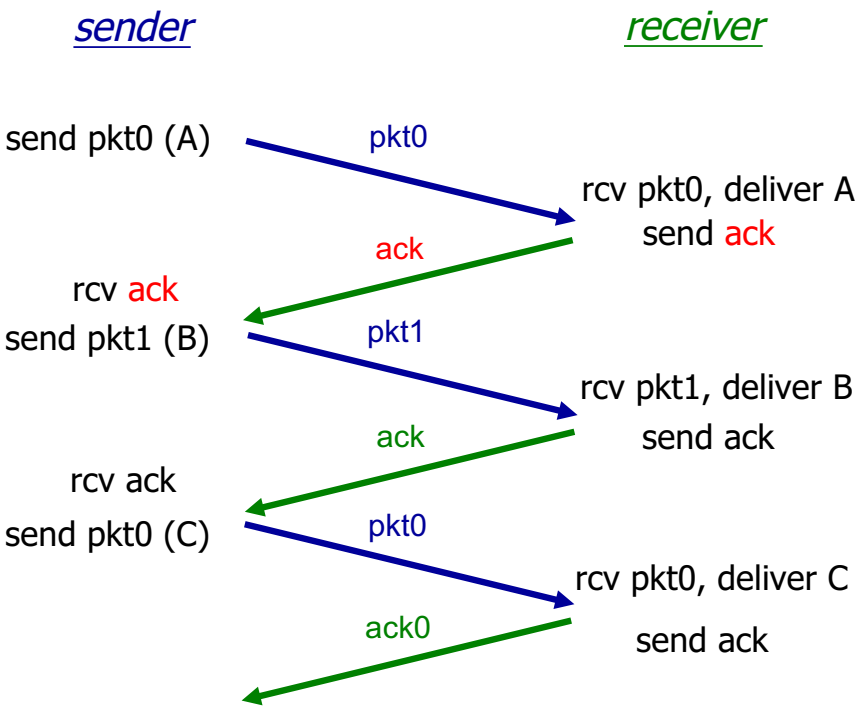
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
  - note: receiver can *not* know if its last ACK/NAK received OK at sender
  - If duplicate is received: do not pass up, but send ACK/NAK to help sender move on
- ACKs/NAKs don't need seq# for channel that doesn'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

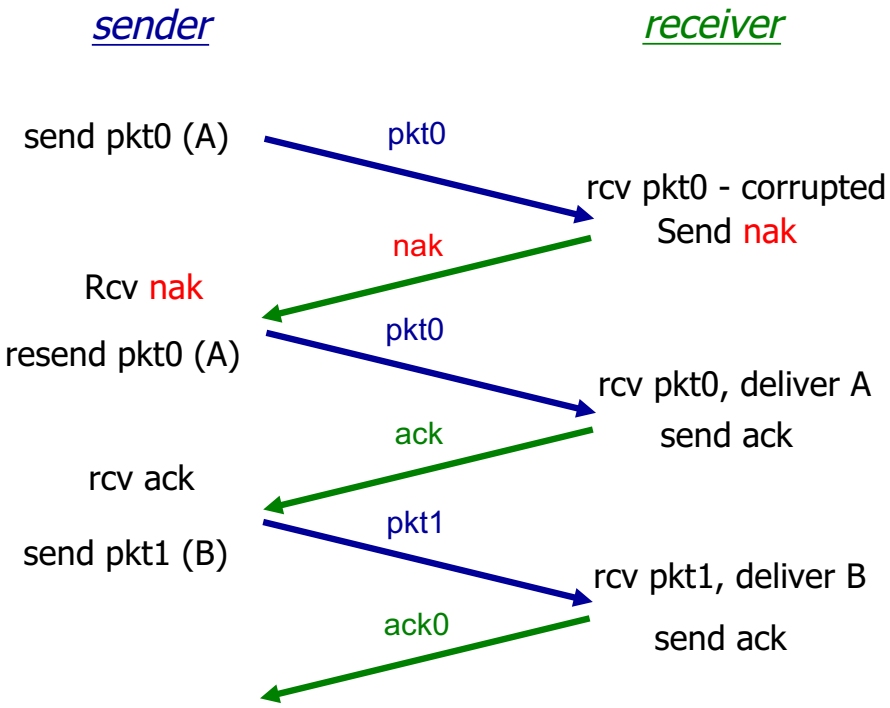


(a) No message or ACK corrupted

Rdt2.1 in action - continued

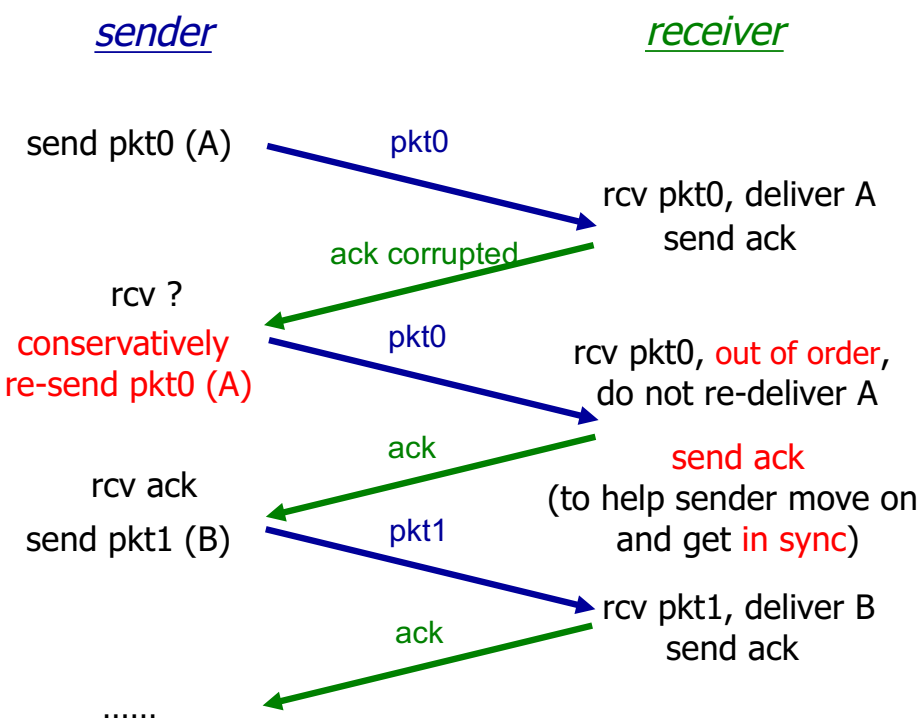


(a) No message or ACK corrupted

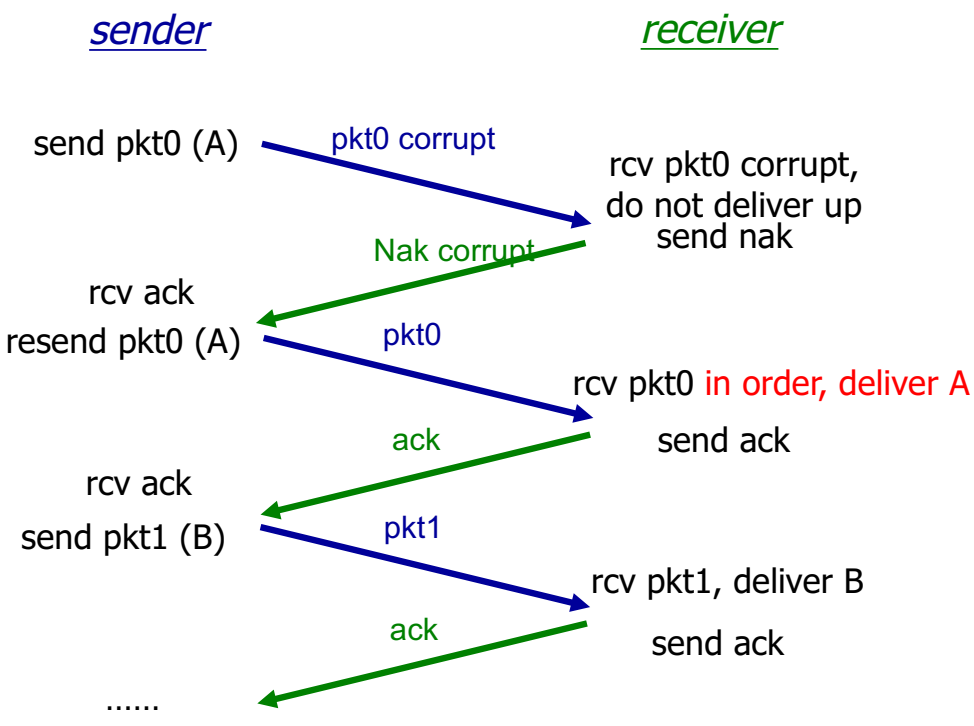


(b) Message corrupted, feedback ok

Rdt2.1 in action - continued



(c) Message ok, ACK corrupted

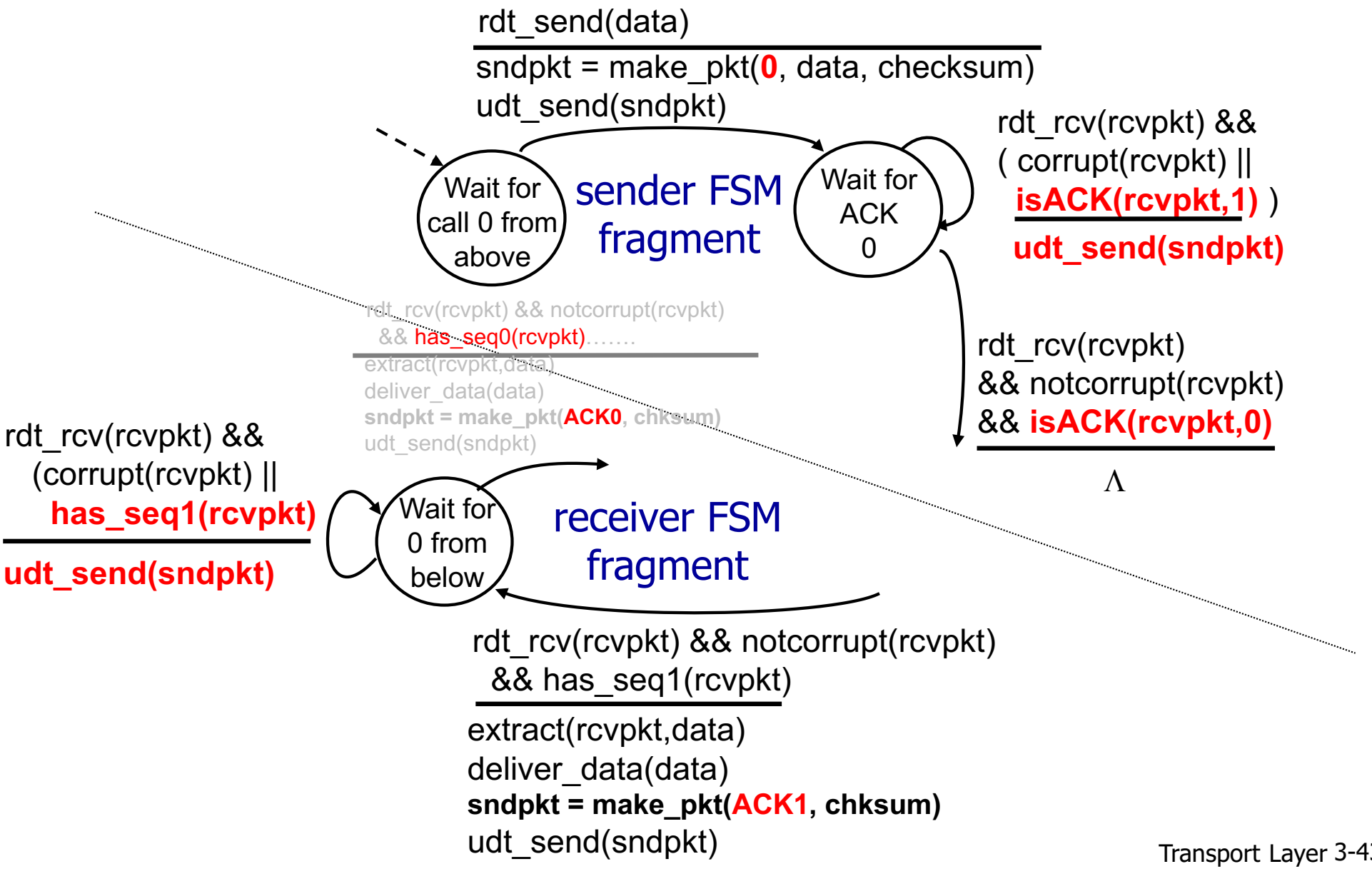


(d) Message corrupted, NAK corrupted

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



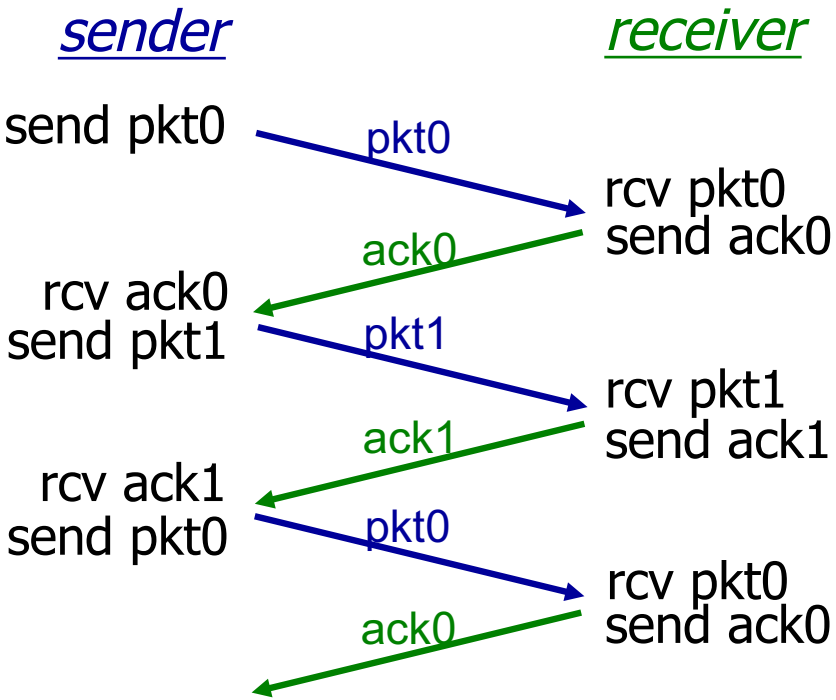
rdt3.0: channels with errors *and* loss

new assumption:

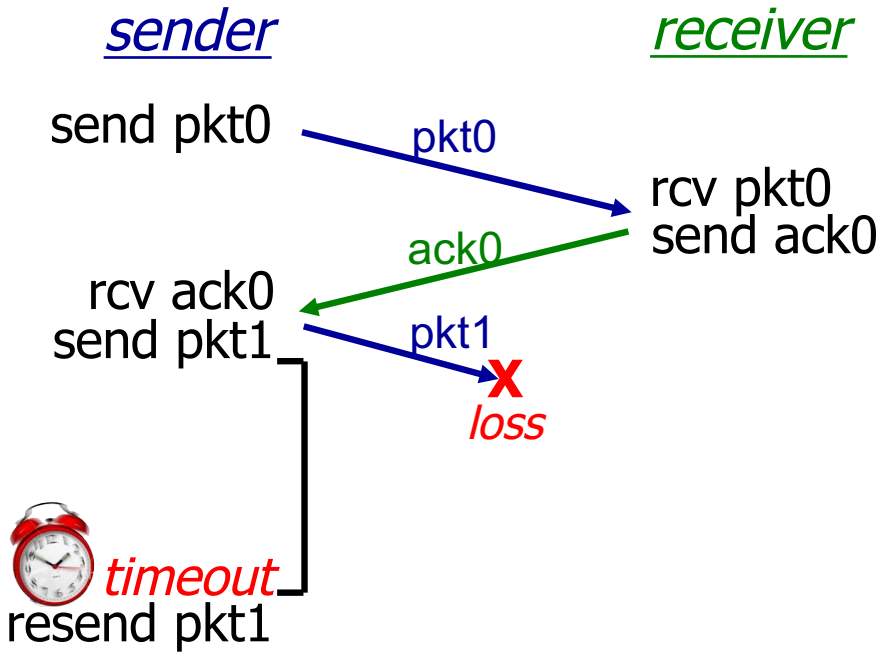
underlying channel can also lose packets (data, ACKs)

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

rdt3.0 in action



(a) no loss



(b) packet loss

rdt3.0: channels with errors *and* loss

new assumption:

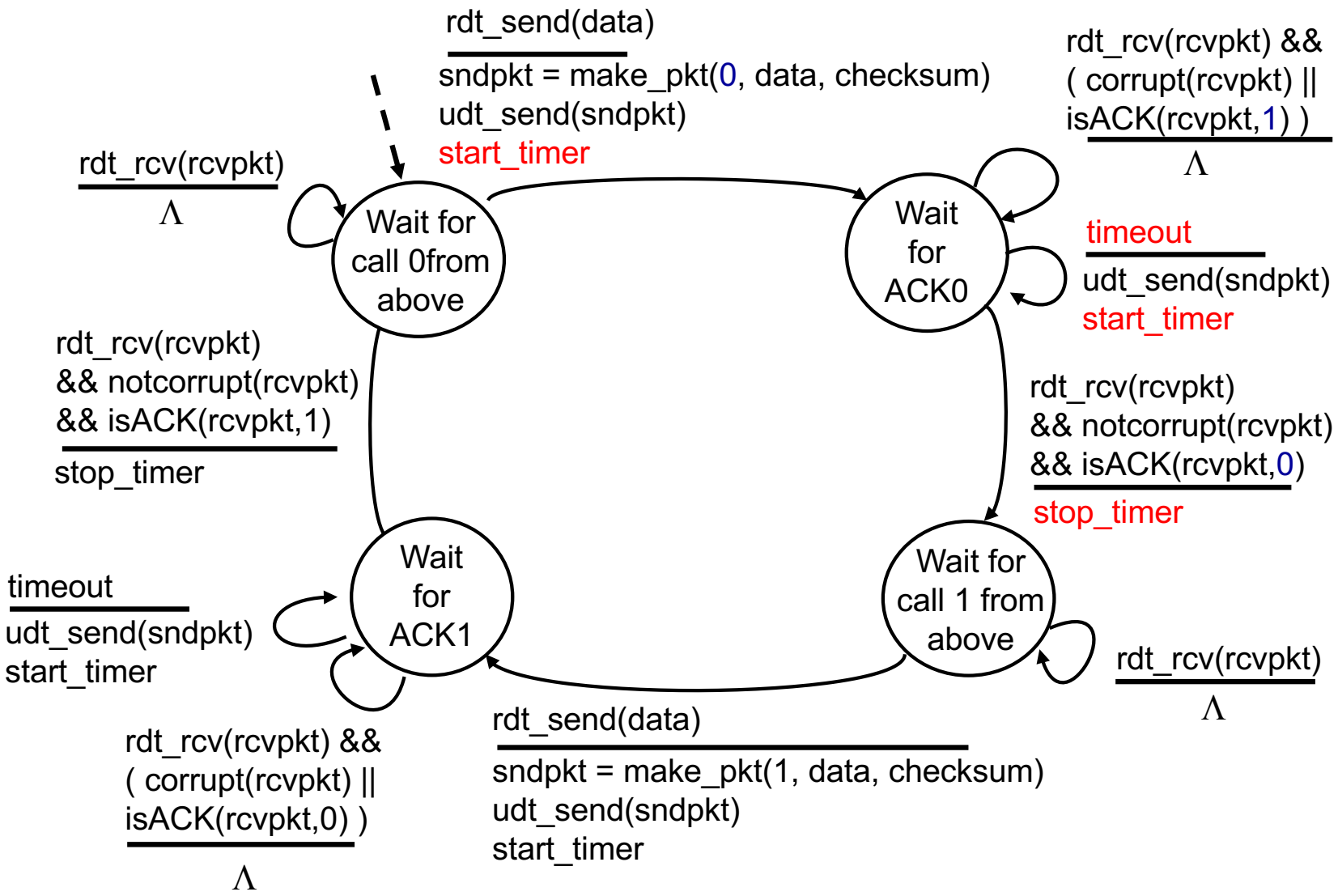
underlying channel can also lose packets (data, ACKs)

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

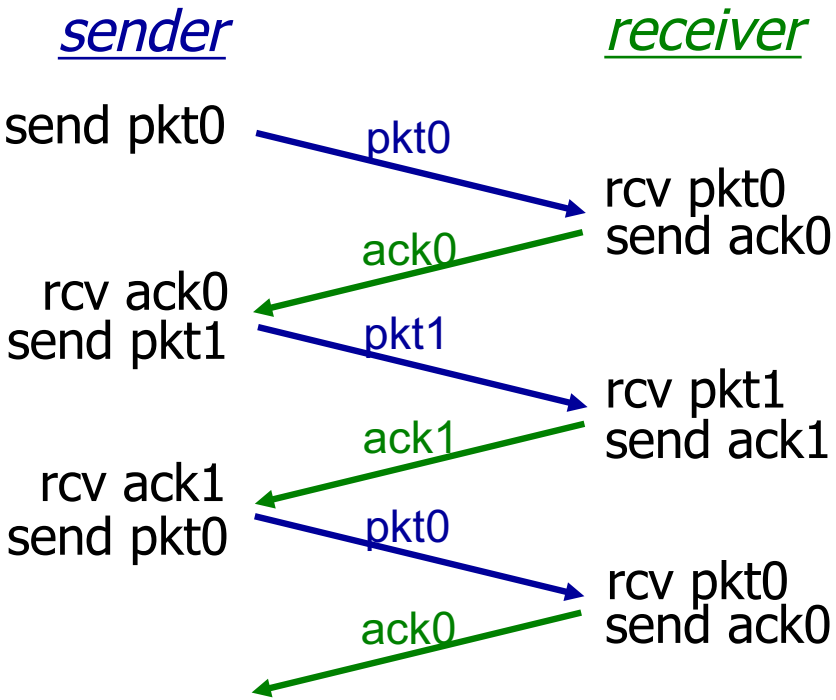
approach: detect loss at server with Timeout

- 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

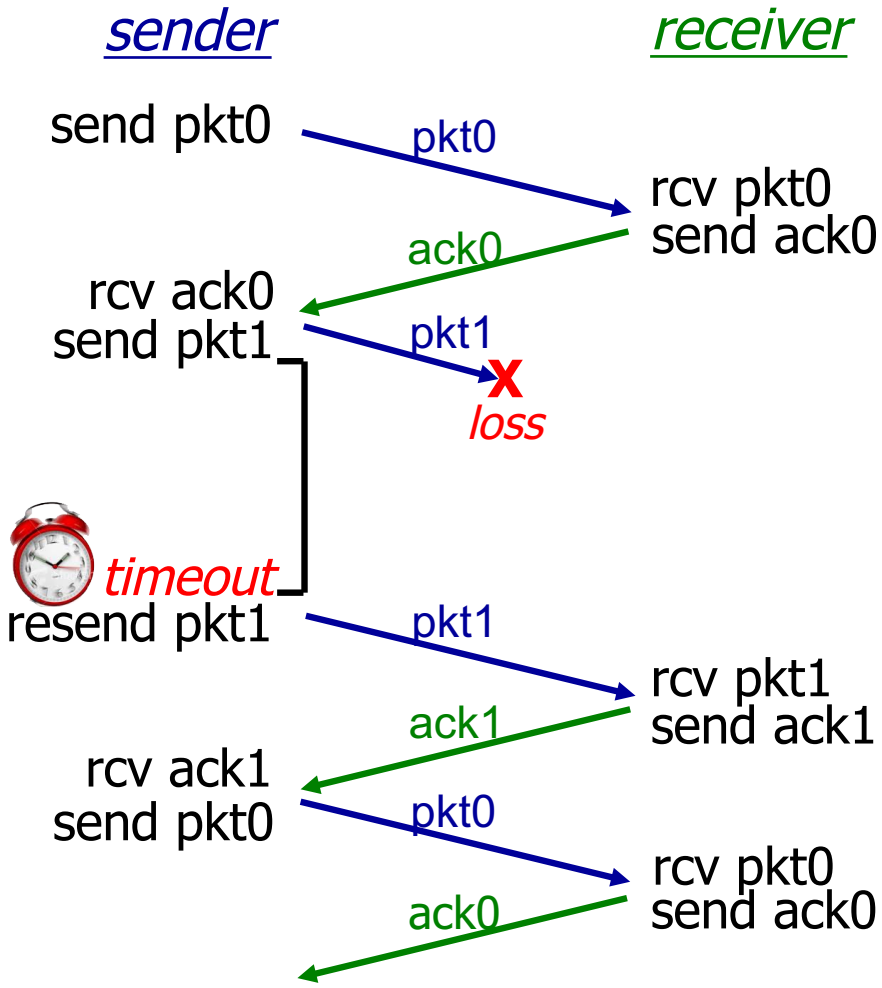
rdt3.0 sender ("alternating bit protocol")



rdt3.0 in action

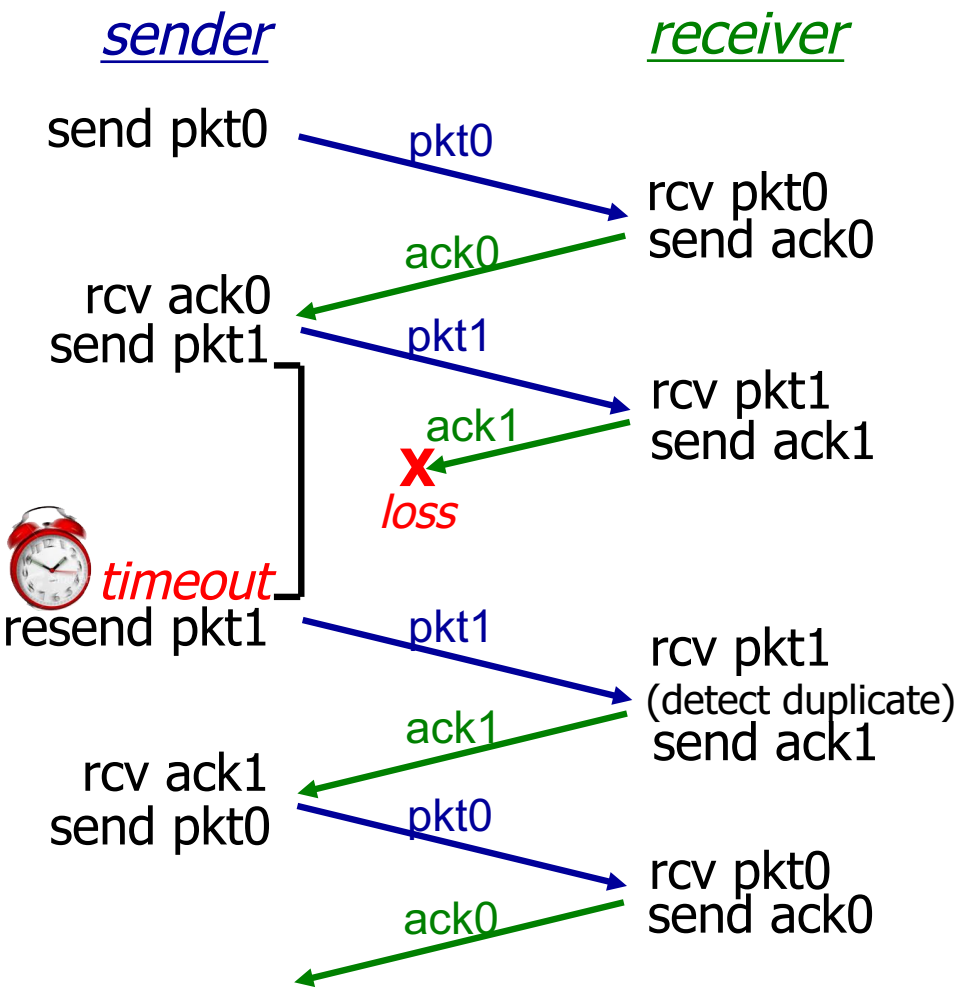


(a) no loss

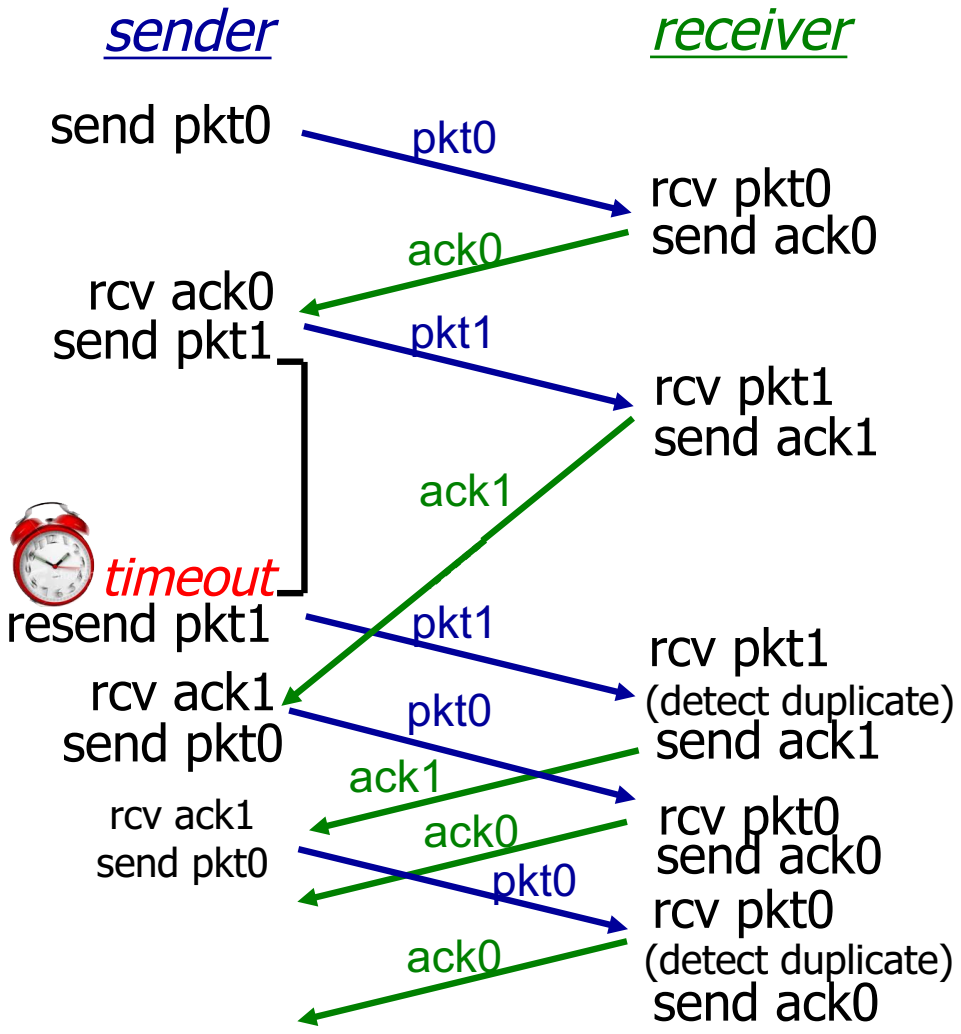


(b) packet loss

rdt3.0 in action

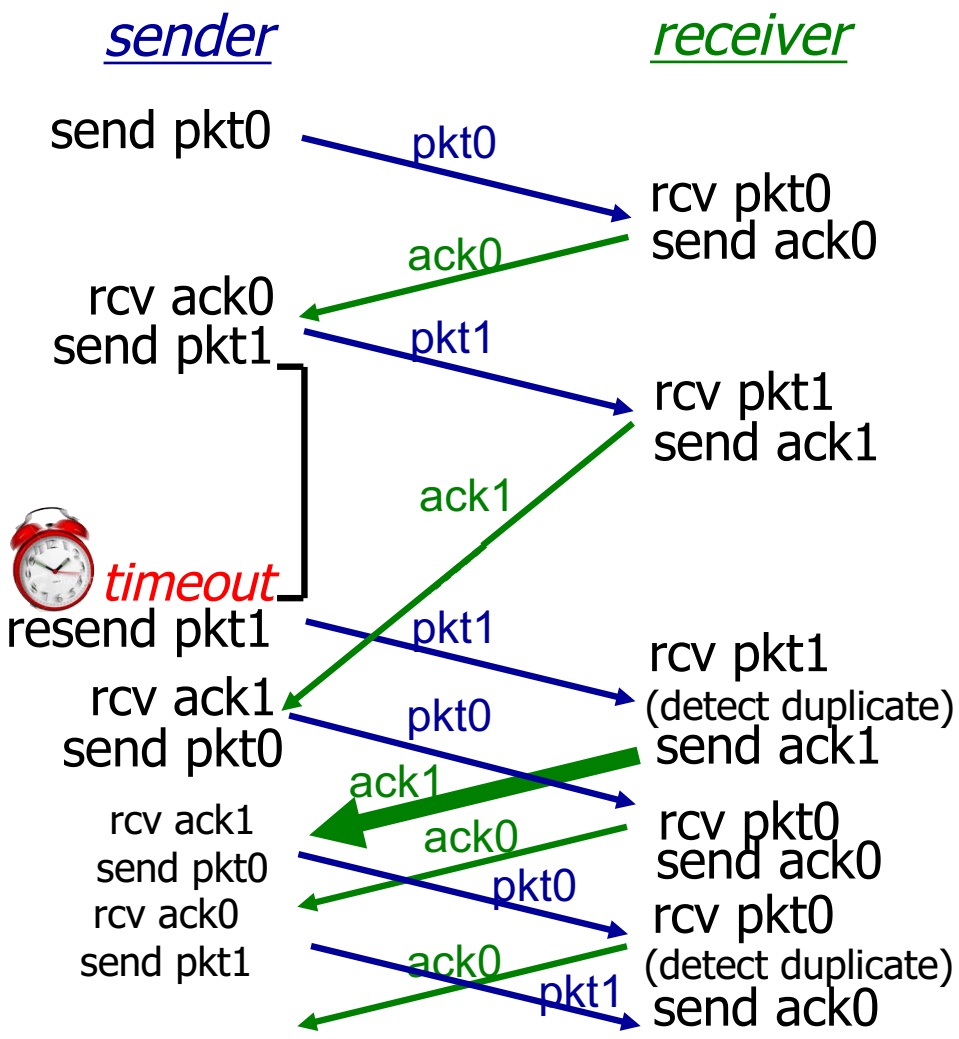


(c) ACK loss



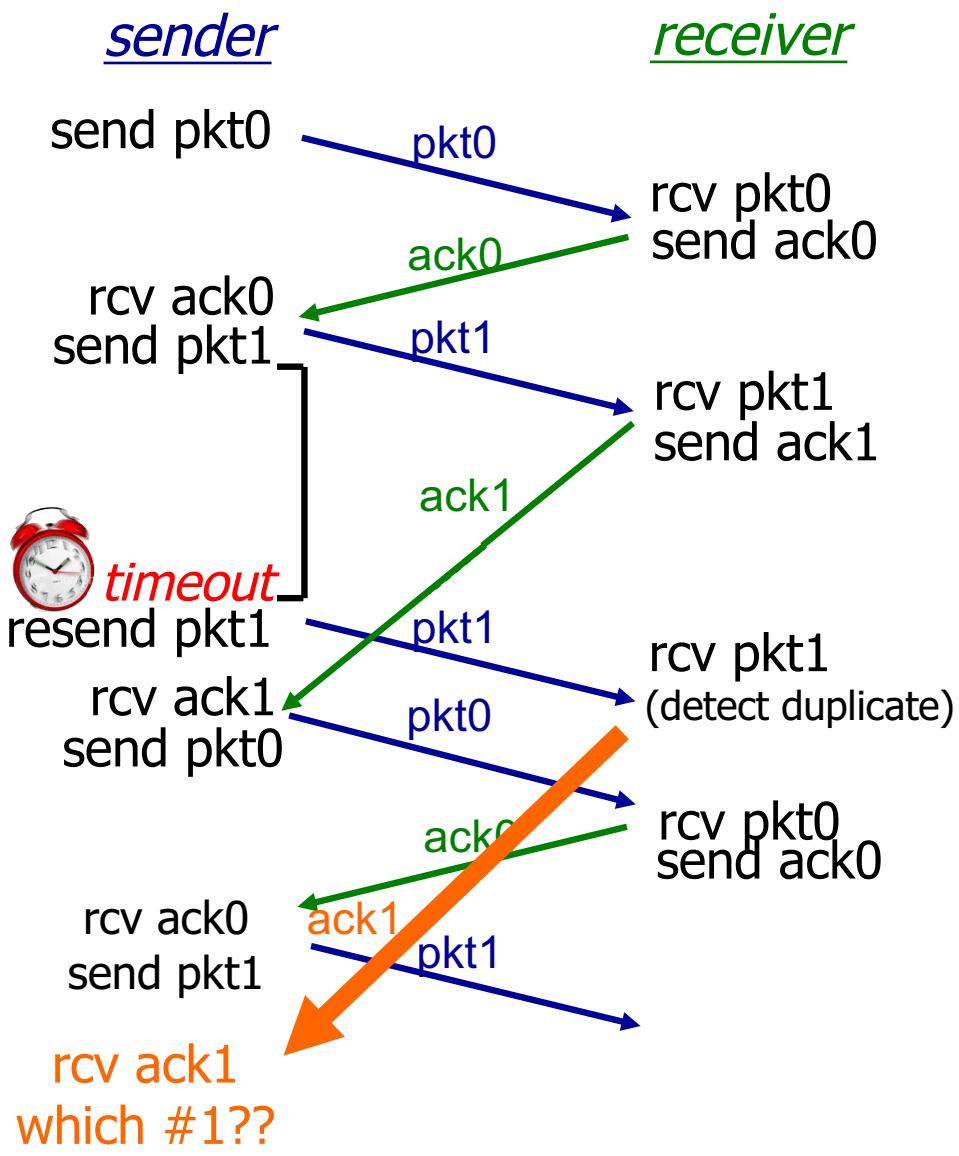
(d) premature timeout/ delayed ACK

rdt3.0 in action



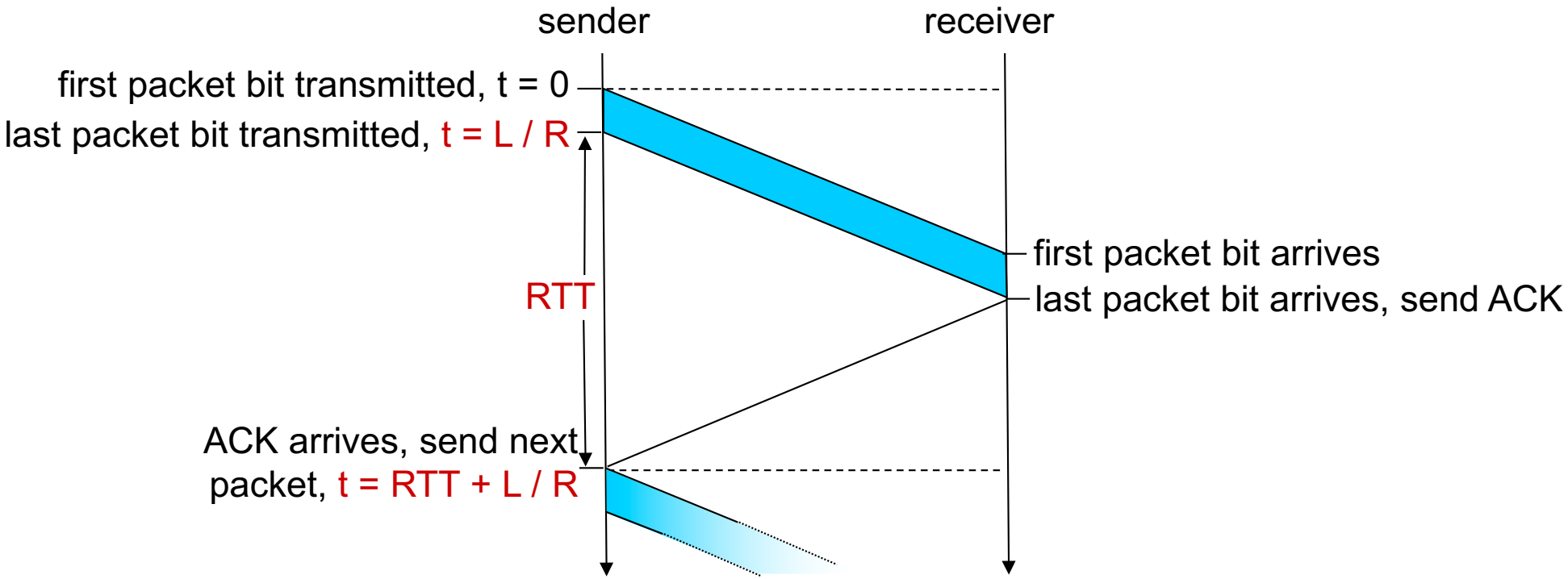
(d) premature timeout/ delayed ACK

rdt3.0: delayed ack- what if?



A: possible only if network reorders packets

rdt3.0: stop-and-wait operation



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