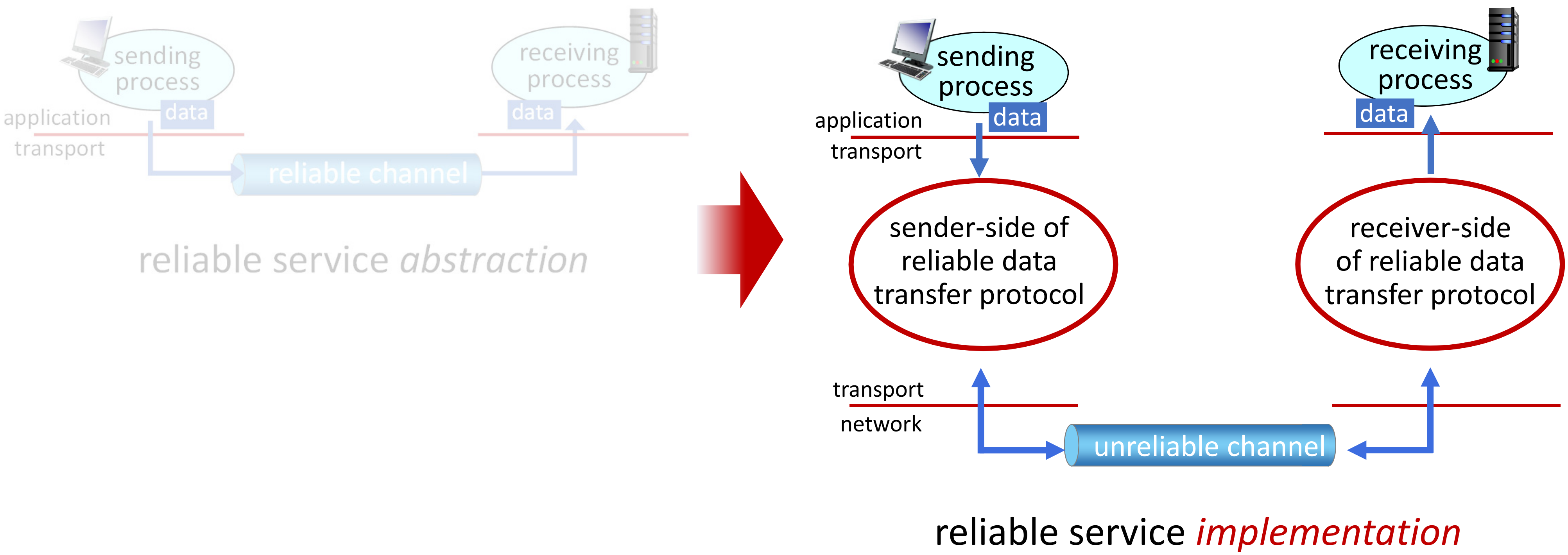
Chapter 3: roadmap

- 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

Principles of reliable data transfer

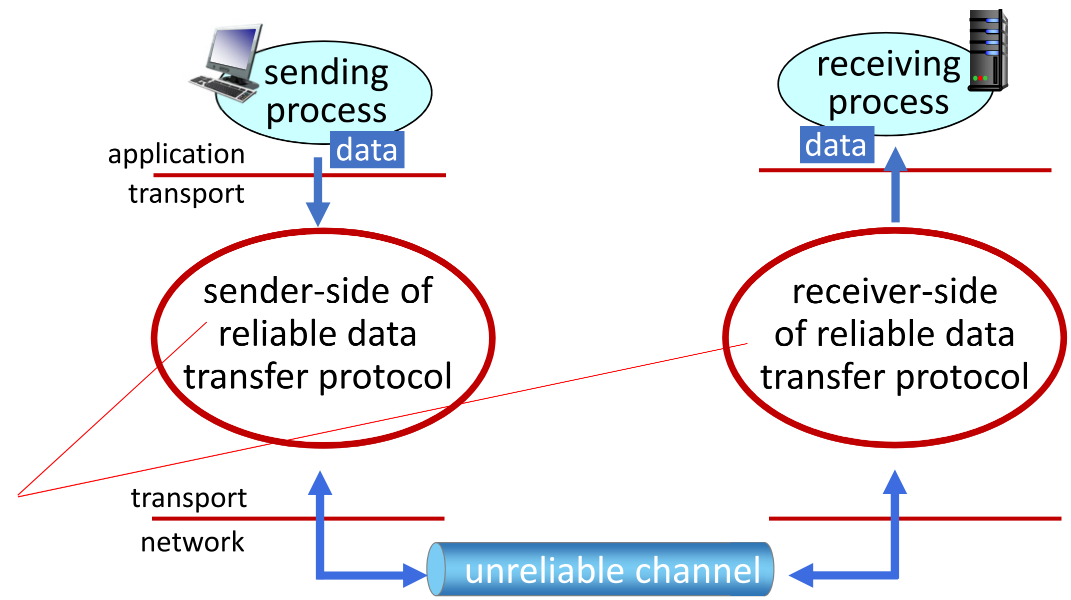
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

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

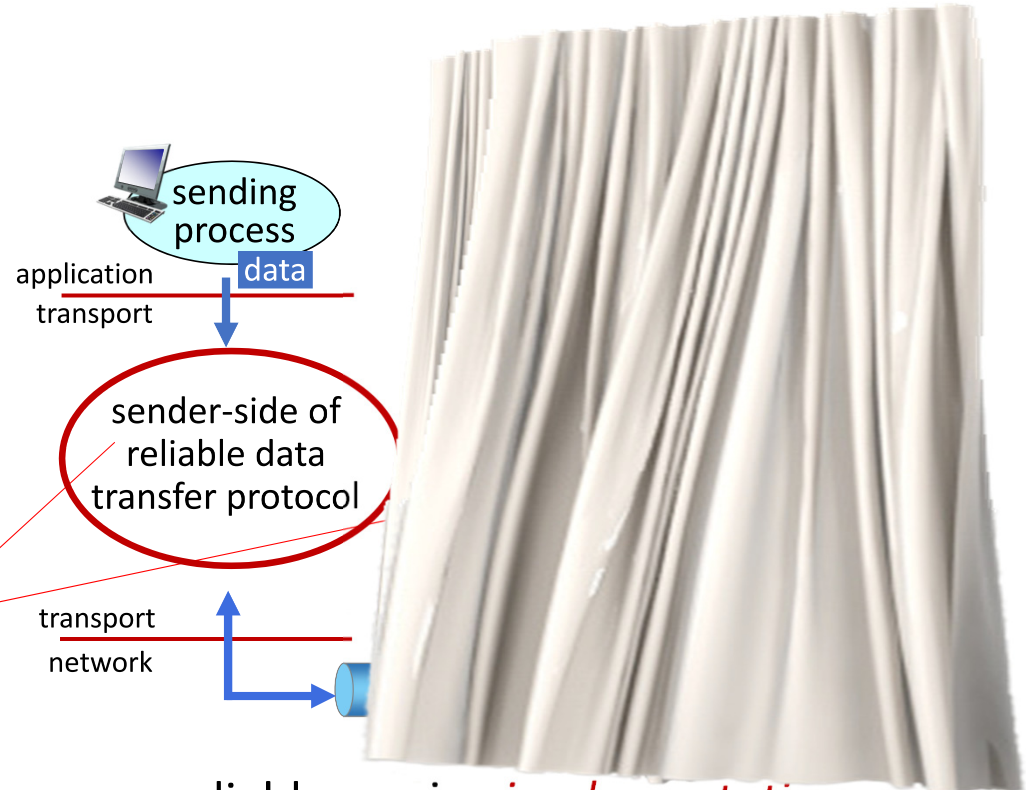


reliable service *implementation*

Principles of reliable data transfer

Sender and receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message

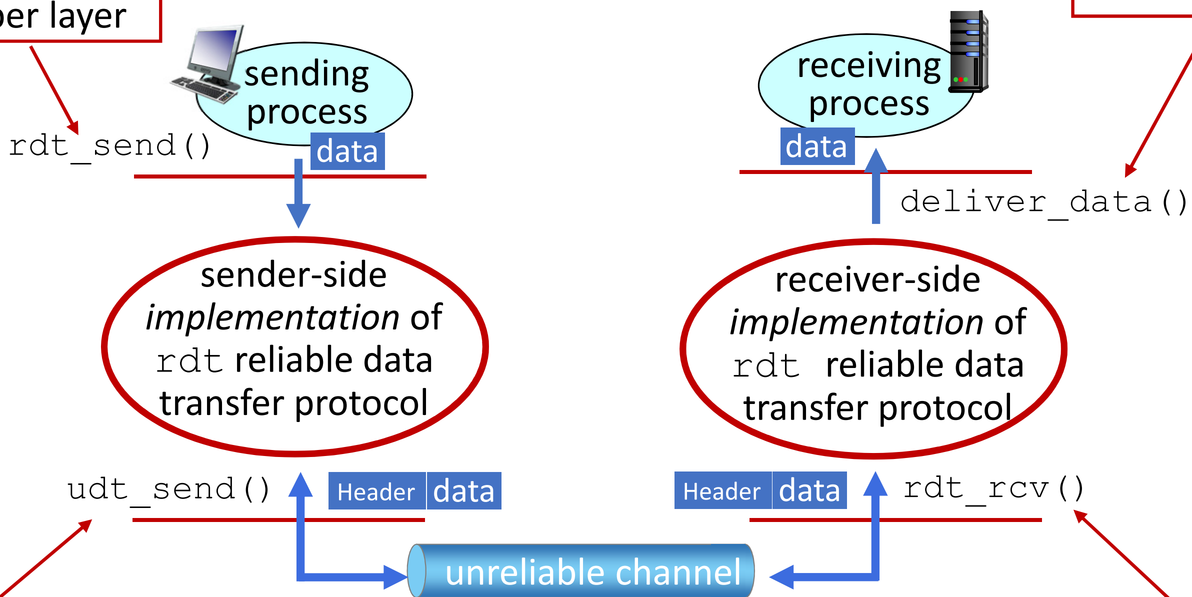


reliable service *implementation*

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

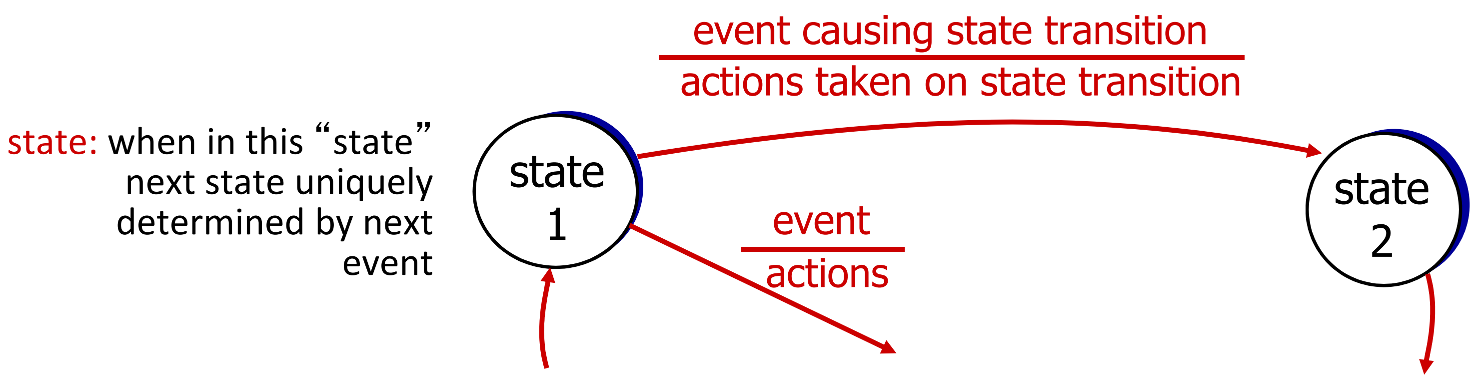


udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

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

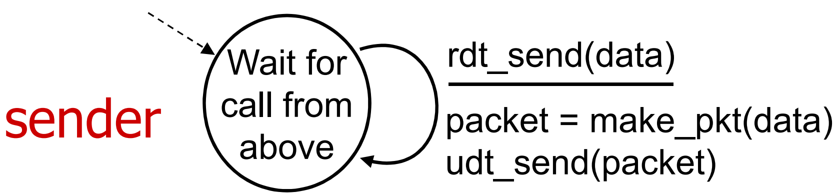
Reliable data transfer: getting started

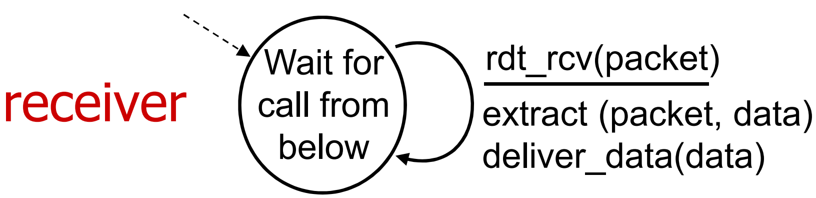
- We will incrementally develop sender and receiver sides of reliable data transfer protocol (rdt)
  - from simple to complex
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender and receiver



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





rdt2.0: channel with forward bit errors (but w/o loss)

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- the question: how to recover from errors?
  - retransmission if errors are detected
    - similar to the way how humans recover from "errors" during conversation

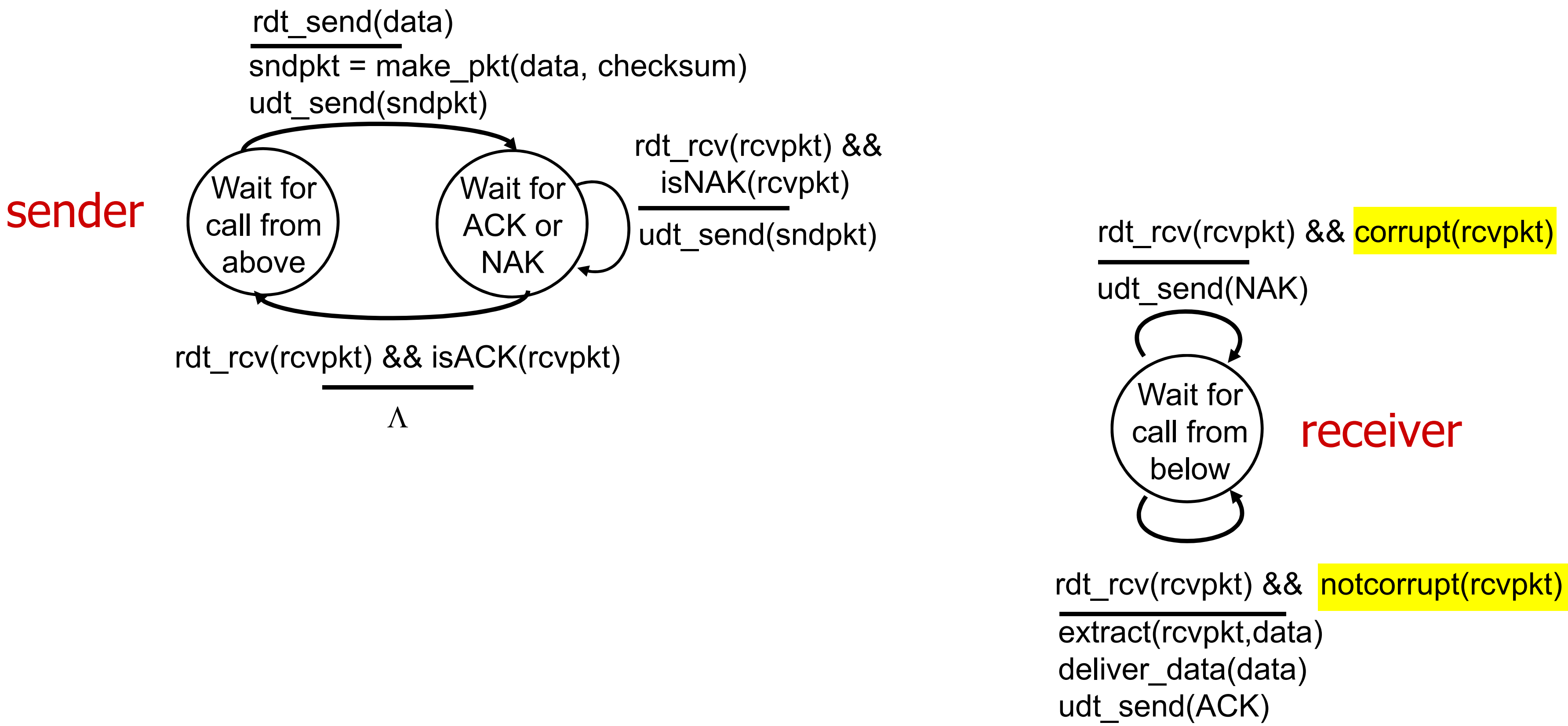
rdt2.0: channel with forward bit errors (but w/o loss)

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- how to know whether bit errors occur and then get recovered?
  - *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

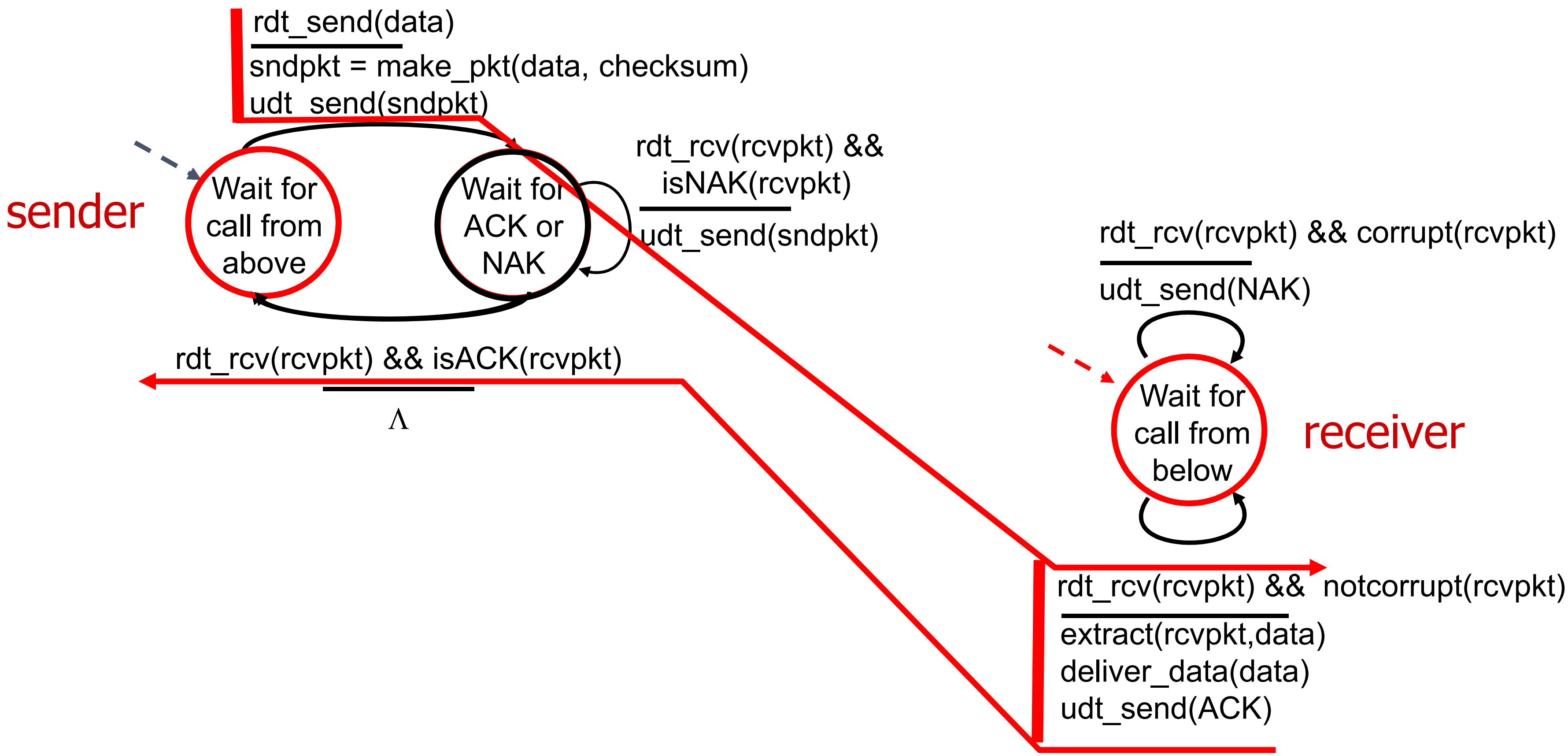
stop and wait

sender sends one packet, then waits for receiver response

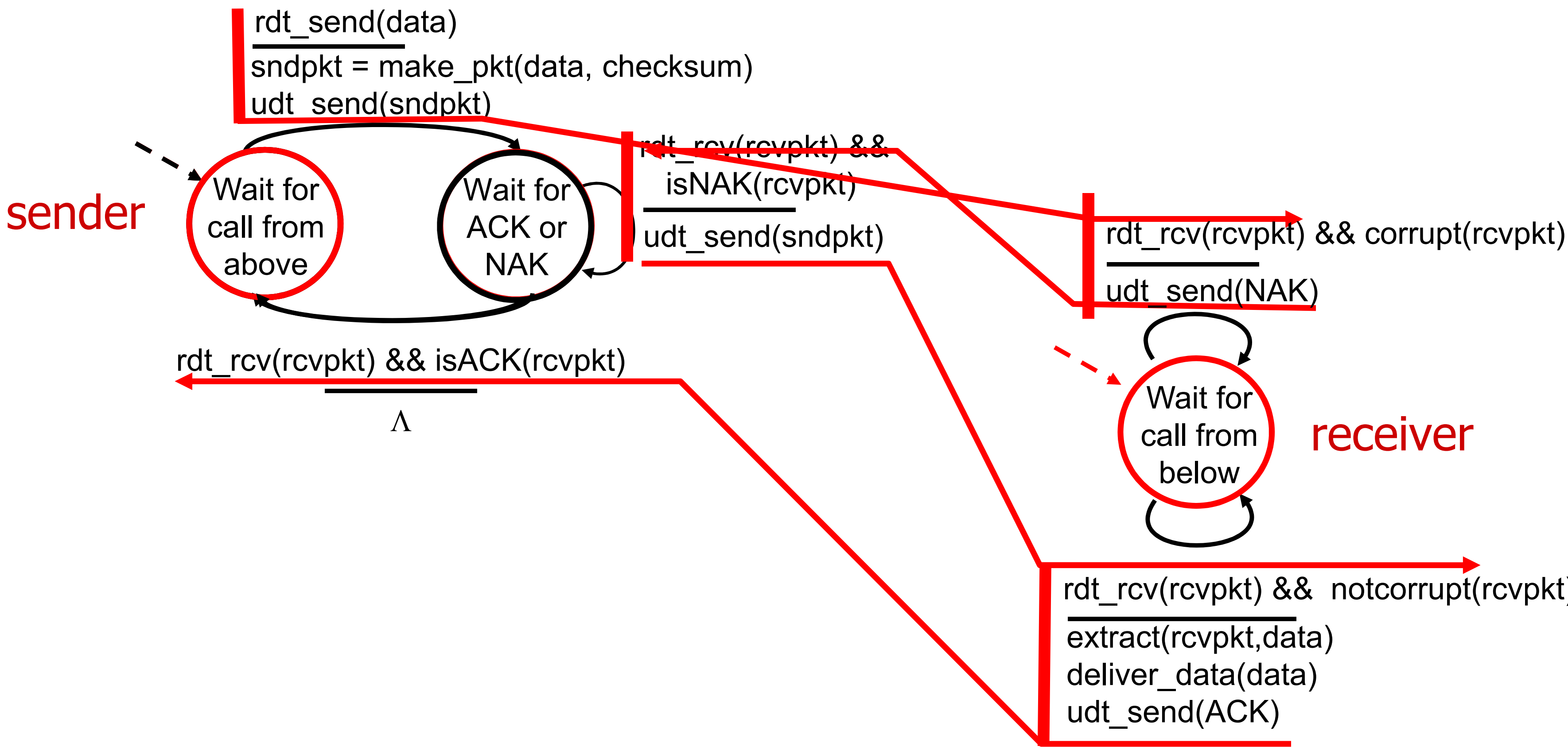
rdt2.0: FSM specifications



rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw when ACK/NAK may corrupt

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
  - rdt2.1 can use {0,1} for seq #
- receiver discards (doesn't deliver up) duplicate pkt

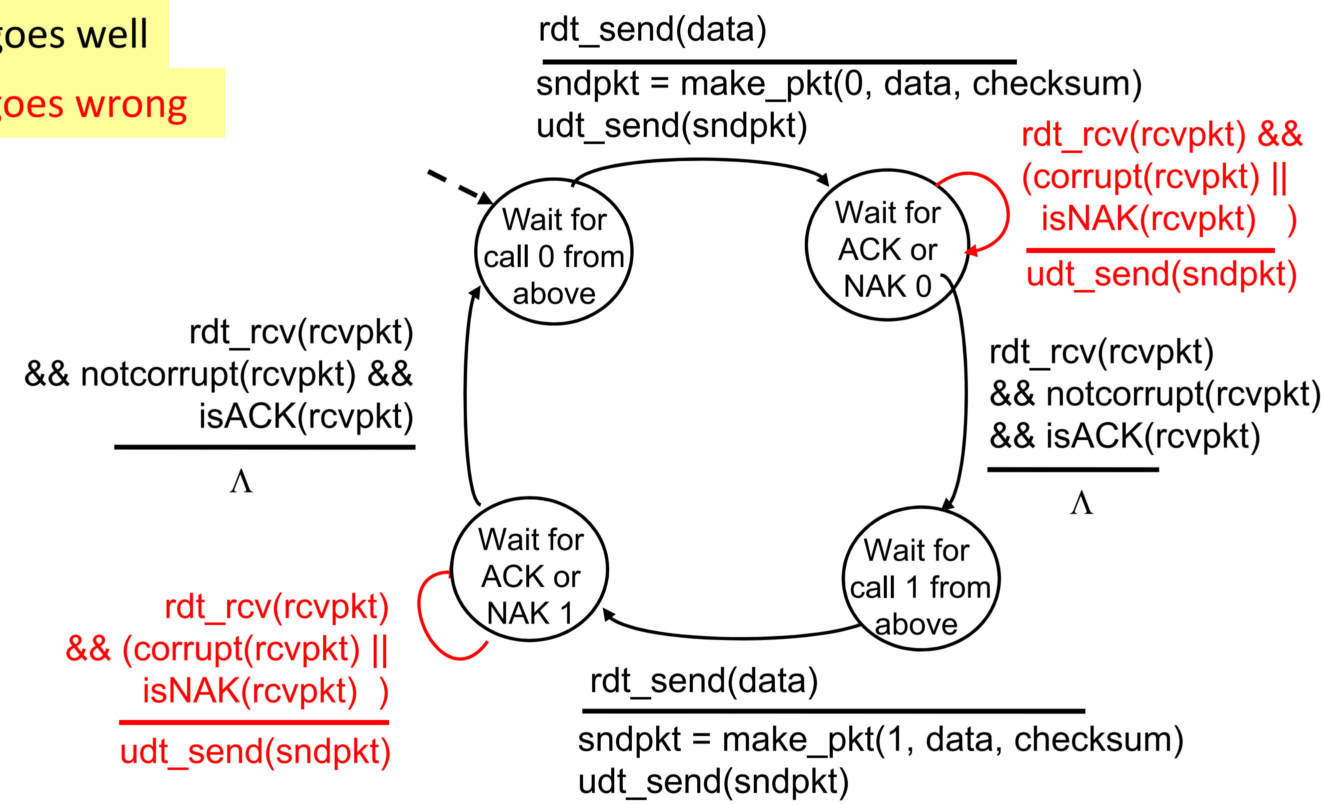
stop and wait

sender sends one packet,
then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs

If everything goes well

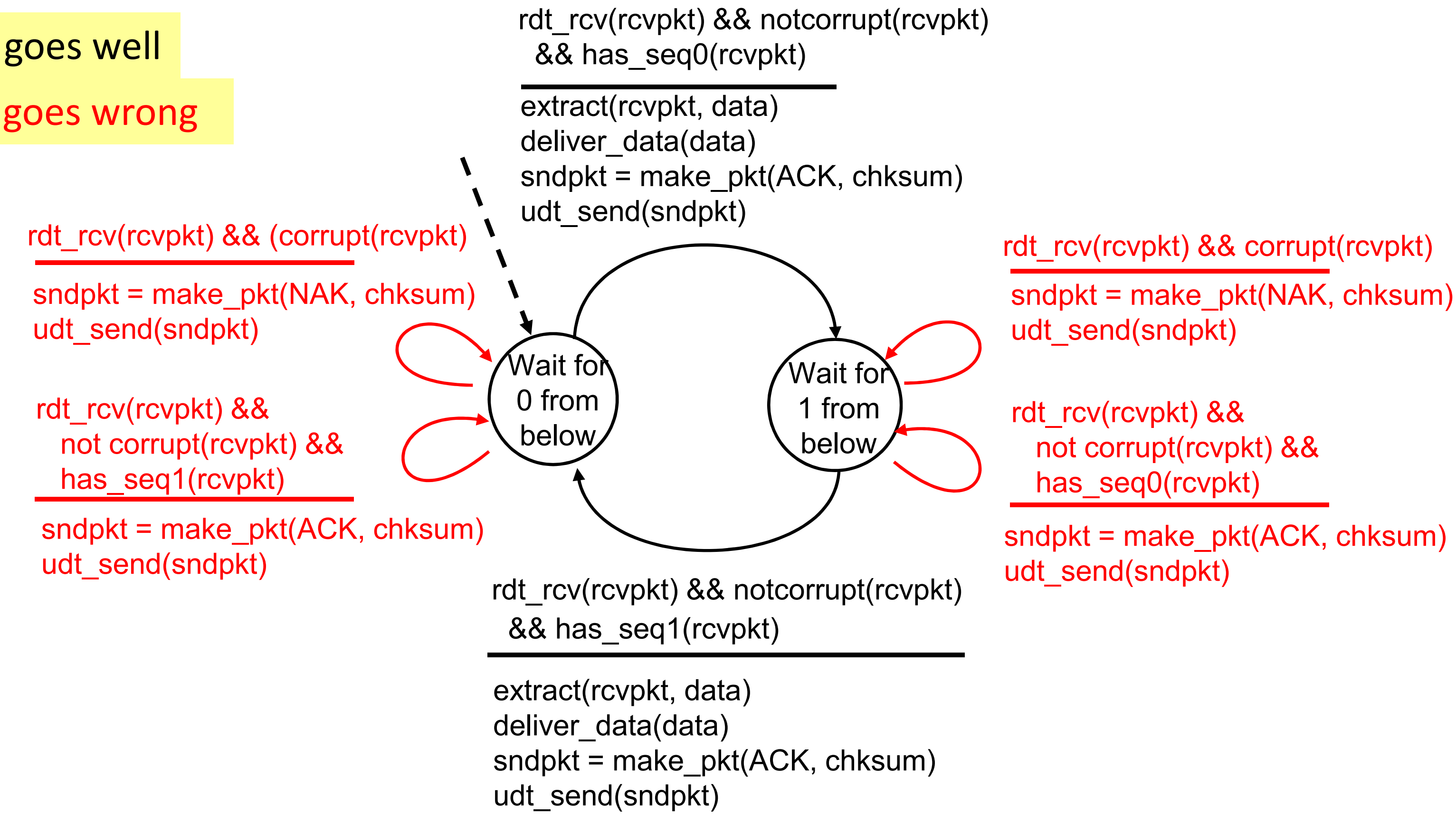
If something goes wrong



rdt2.1: receiver, handling garbled ACK/NAKs

If everything goes well

If something goes wrong



rdt2.1: discussion

sender:

- seq # is added to pkt
- two sequence numbers {0,1} will suffice. Why?
  - for receiver to distinguish *duplicate* or *new* packet
    - in case ACK/NAK is corrupted
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

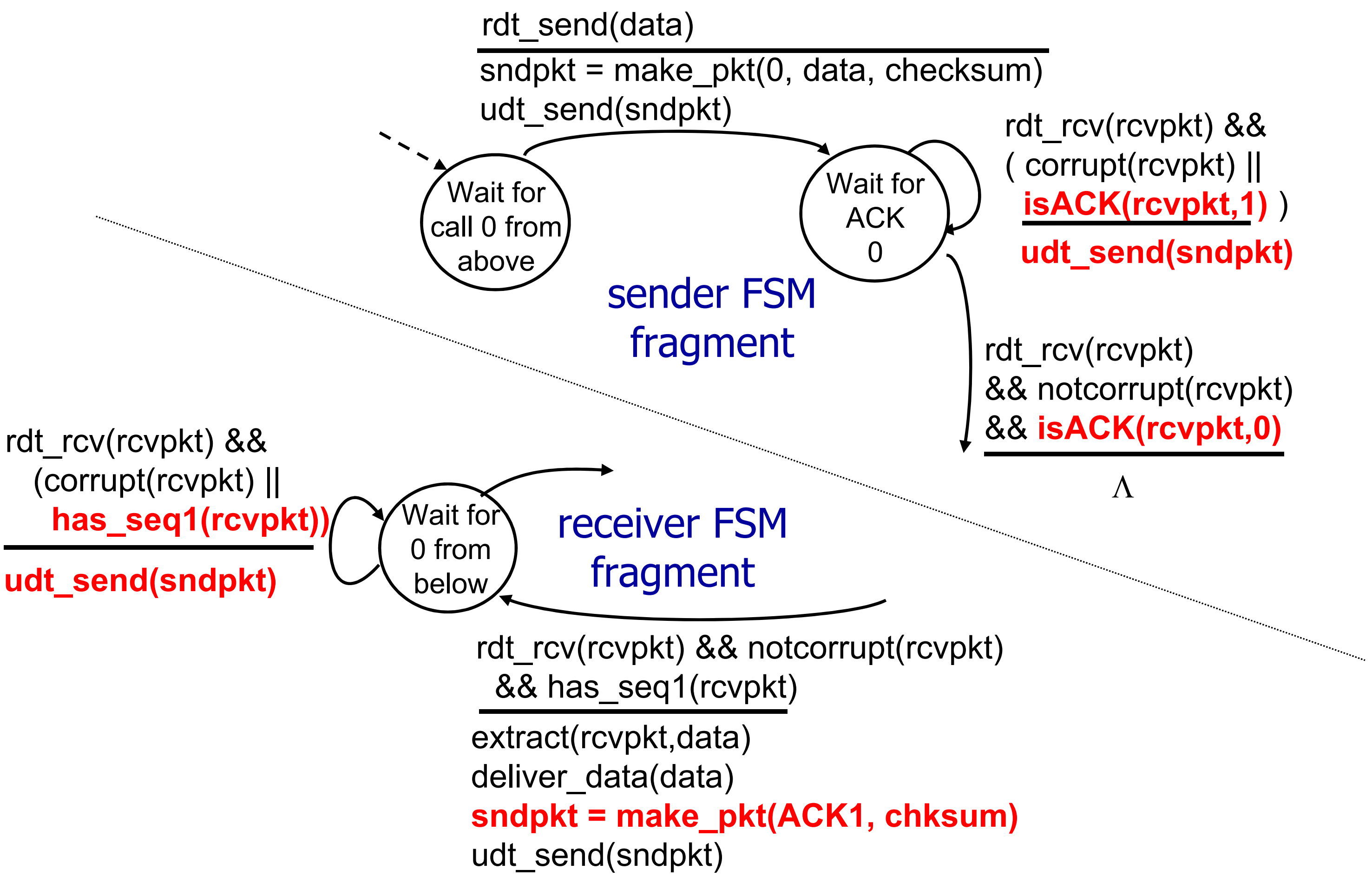
receiver:

- *cannot* know if its last ACK/NAK received OK at sender
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK and seq # (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, (rdt 3.0 and the following rdt including) TCP uses this NAK-free approach

rdt2.2: sender, receiver fragments



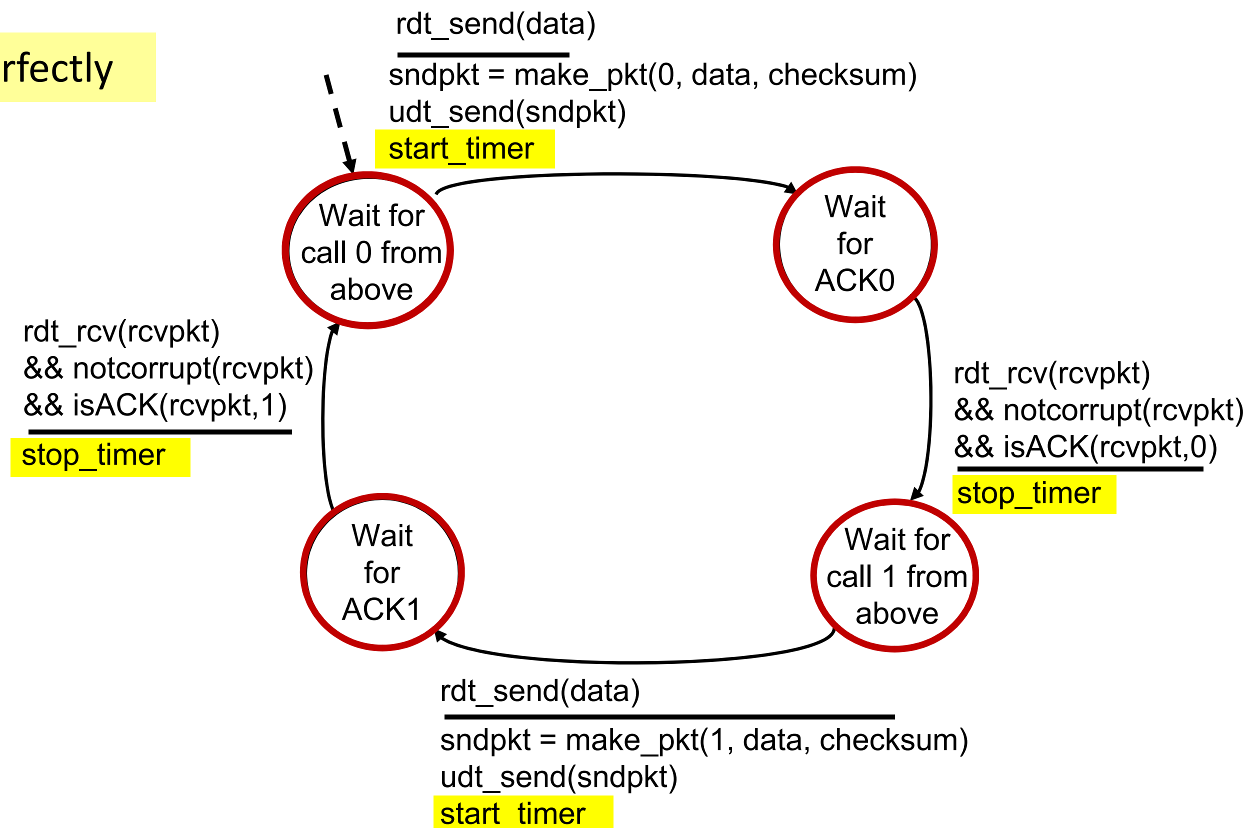
rdt3.0: channels with errors *and loss*

- new channel assumption: underlying channel can also
  - *drop/lose* packets (data, ACKs)
  - delay packets (but still in-order)
- checksum, sequence #, ACKs, retransmissions are not enough to handle it
- new approach: sender waits for "reasonable" amount of time for ACK
  - uses countdown timer and retransmits data packet once (and only if) timeout
    - timeout means that no ACK is received in this interval and the timer expires
    - rdt3.0 does nothing when receiving an ACK with wrong seq #
  - if a packet or ACK is just over-delayed (instead of lost):
    - after timeout, sender will retransmit the data packet
    - receiver detects a duplicate transmission, because seq # already handles this!

timeout

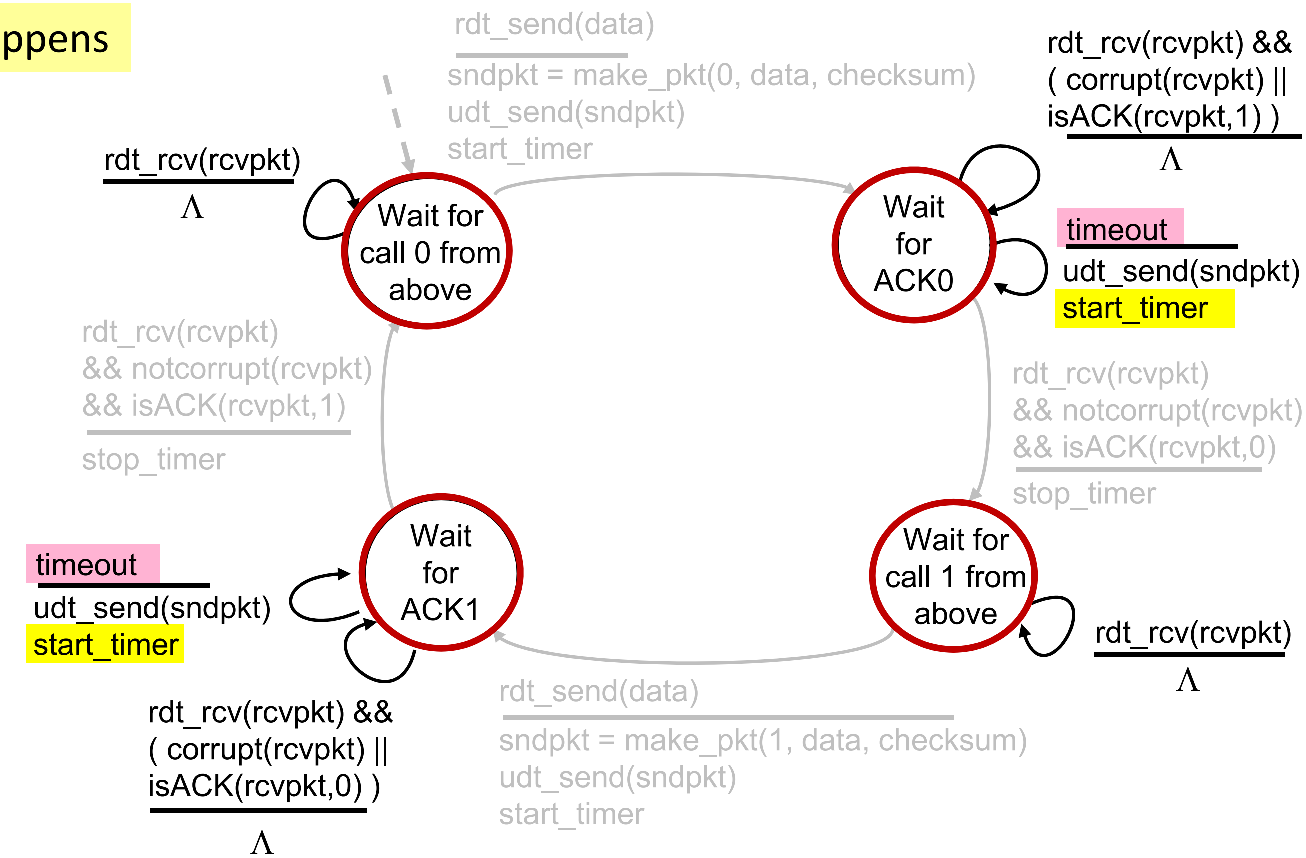
rdt3.0 sender

If everything goes perfectly

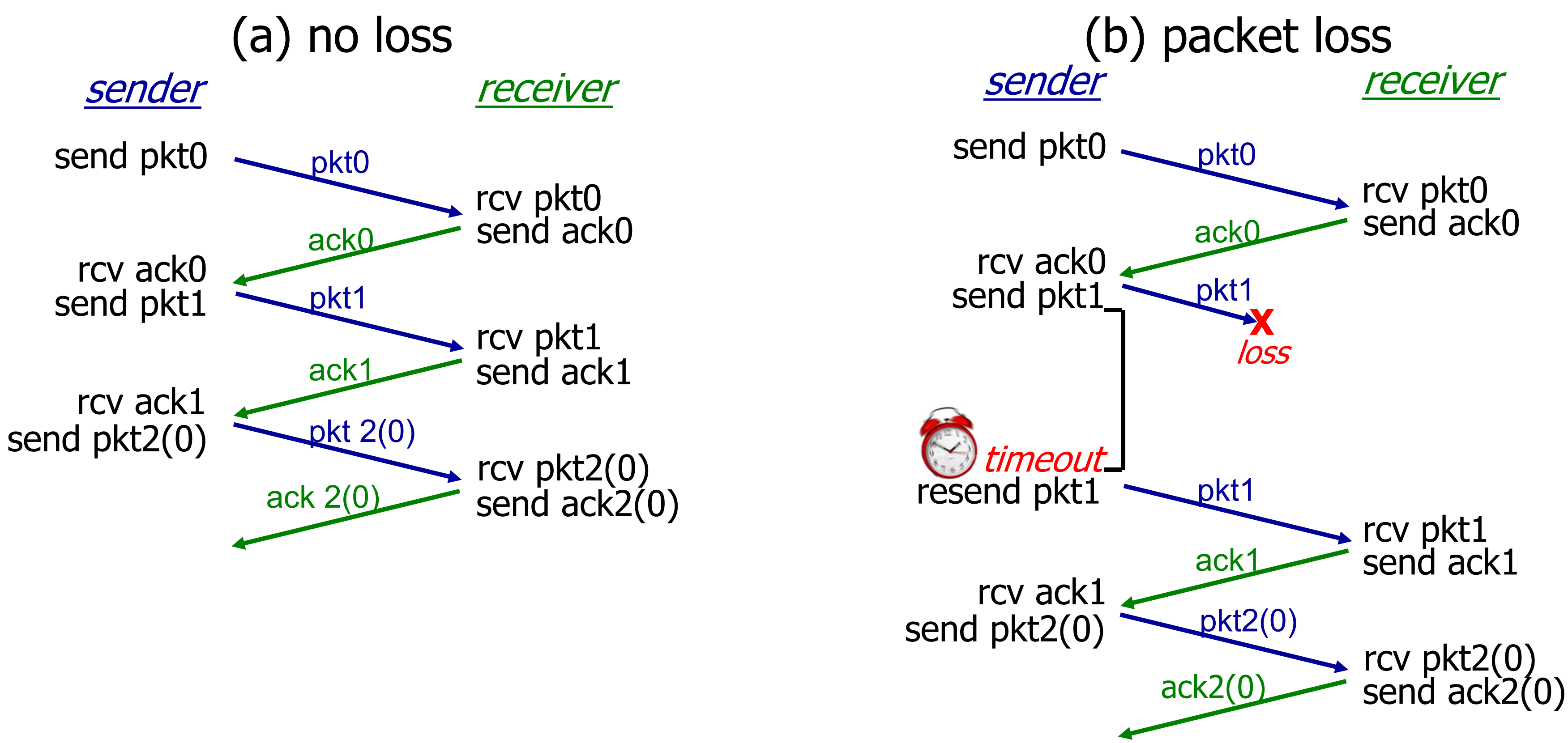


rdt3.0 sender

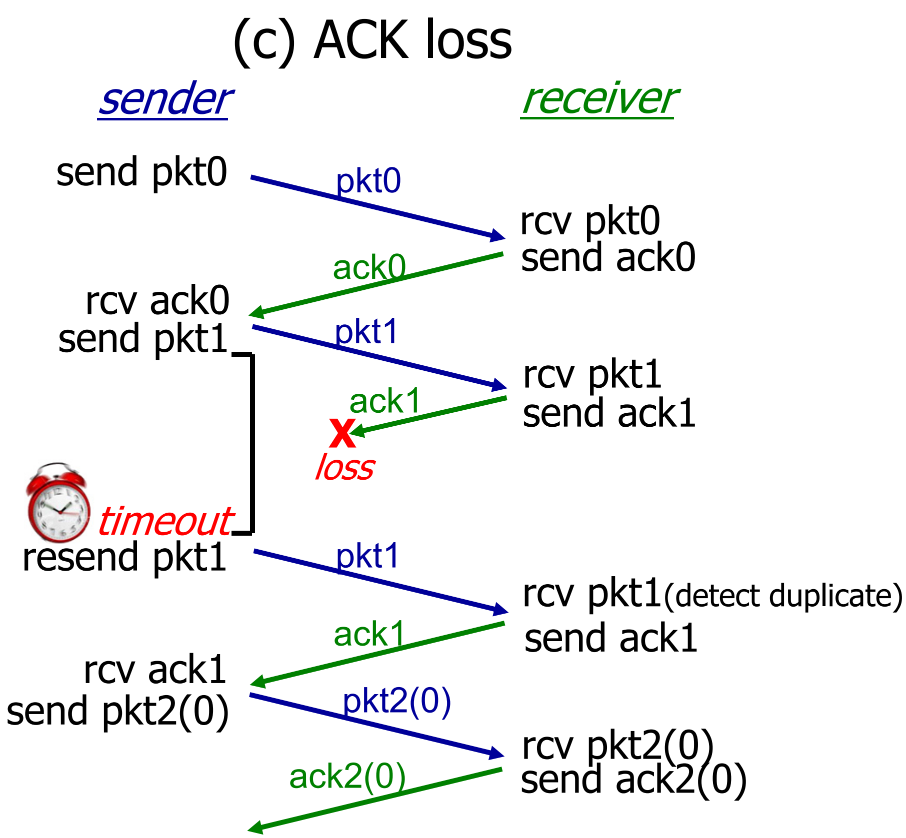
If something bad happens

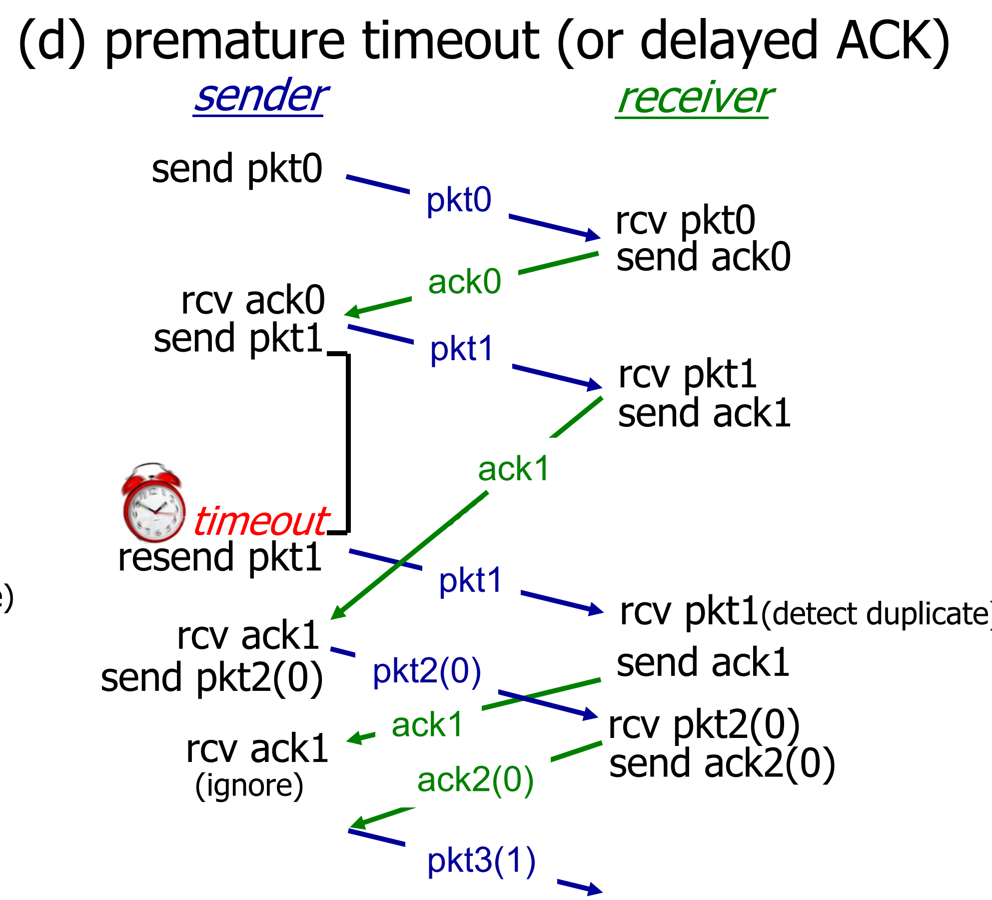


rdt3.0 in action



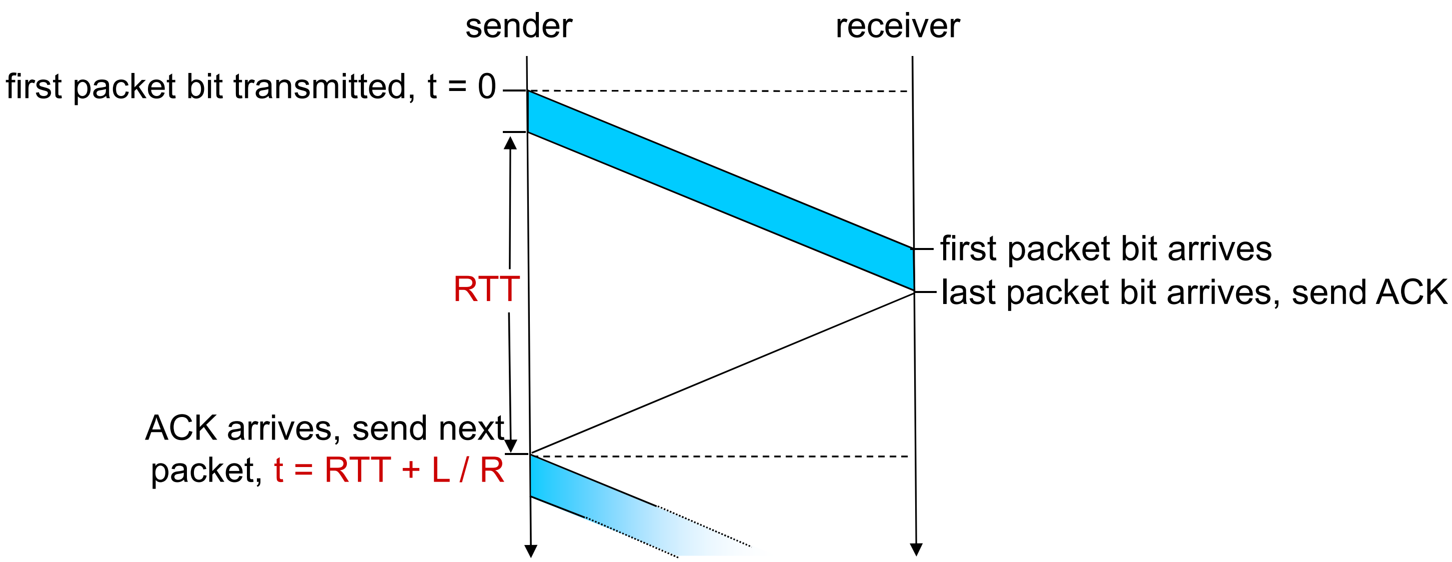
rdt3.0 in action





Performance of rdt3.0 (which is stop-and-wait)

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



rdt3.0: stop-and-wait operation

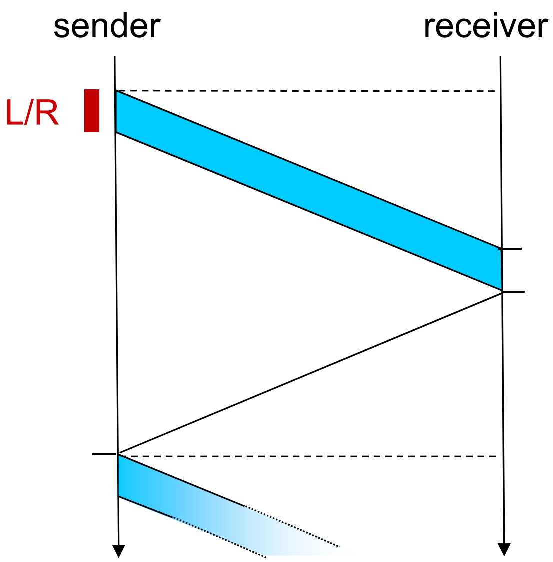
example: 1Gbps link, 15ms propagation delay, 8000-bit packet

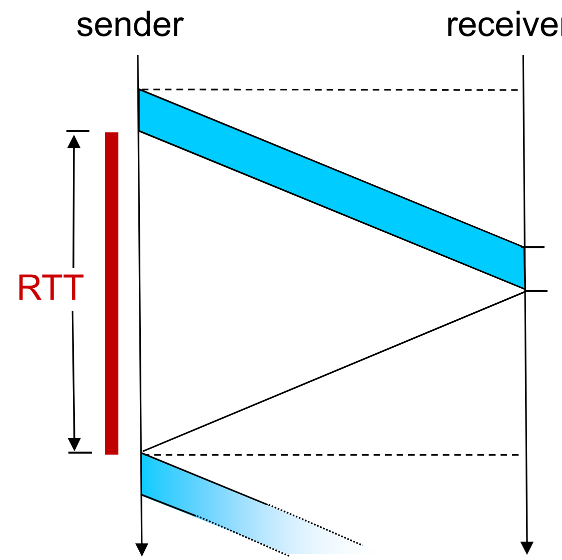
- transmission time:

$$\frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/s}} = 8\,\mu s = 0.008\,ms$$

- RTT (round-trip propagation time):

$$RTT = 2 \cdot 15\,ms = 30\,ms$$





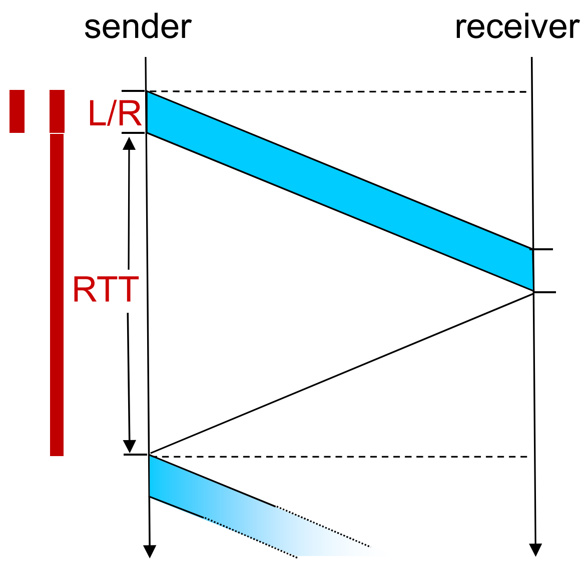
rdt3.0: stop-and-wait operation

example: 1Gbps link, 15ms propagation delay, 8000-bit packet

- Utilization:

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

$$= \frac{0.008}{0.008 + 30} = 0.00027$$



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