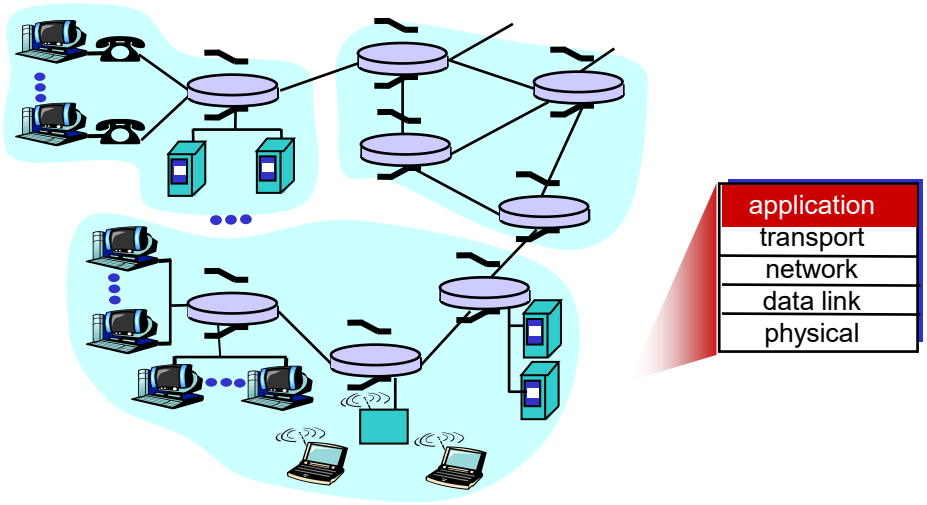
CS 4390
Computer Networks

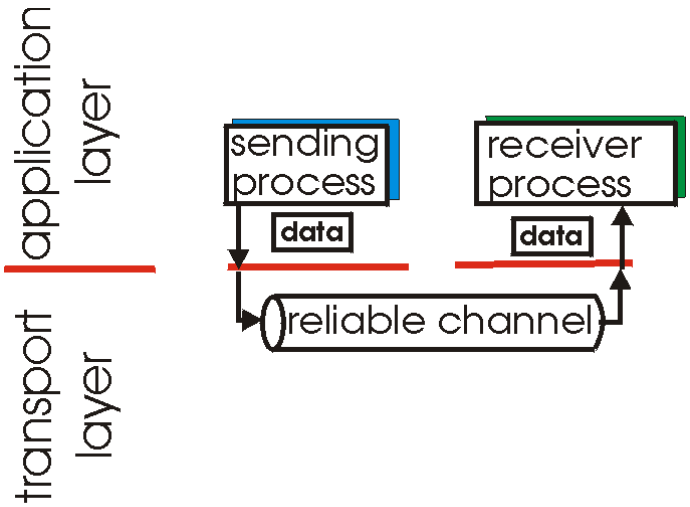


Transport Layer – Reliable Data Transfer

Adapted from "Computer Networking – a Top-Down Approach"

Principles of Reliable Data Transfer

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

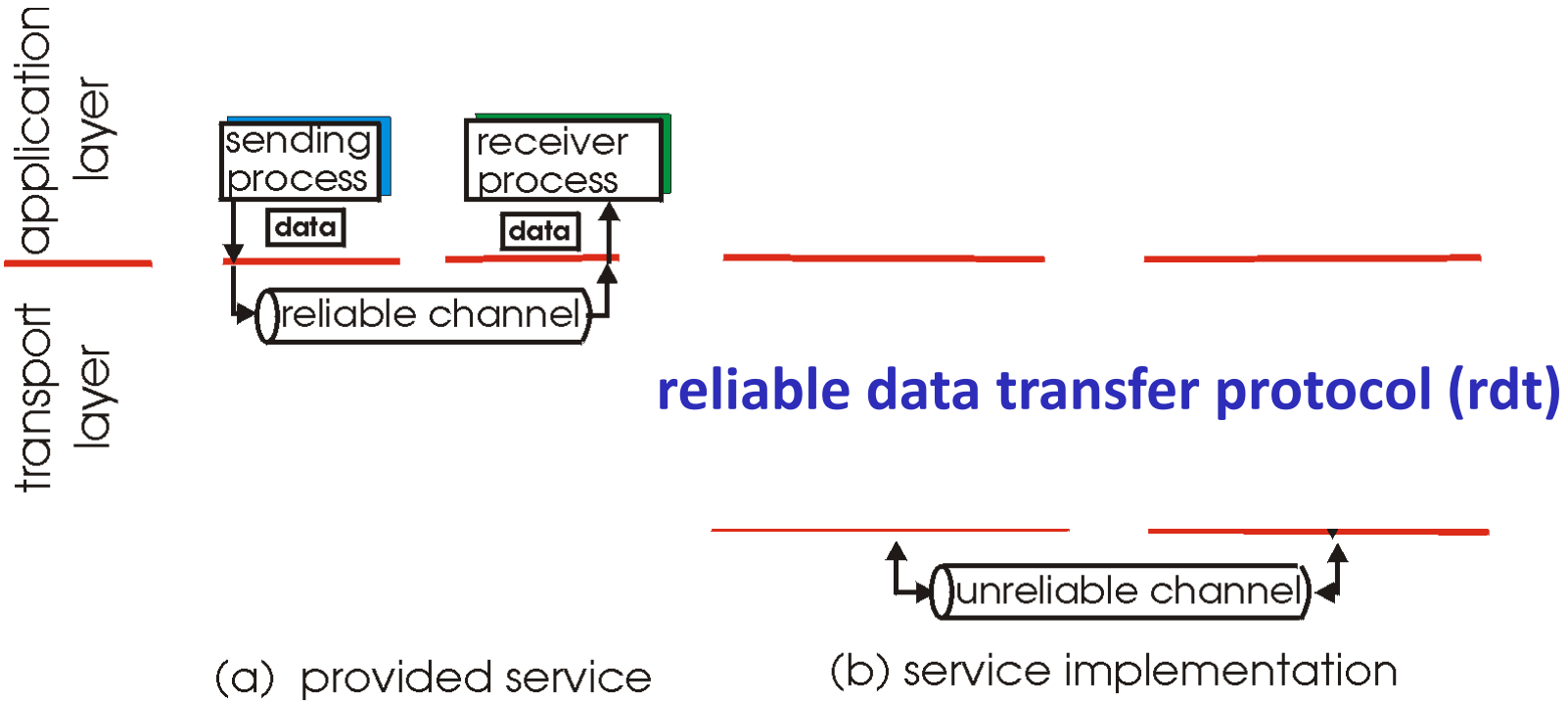


(a) provided service

How to develop algorithms and techniques for transport layer to provide reliable channel service to applications?

Principles of Reliable Data Transfer

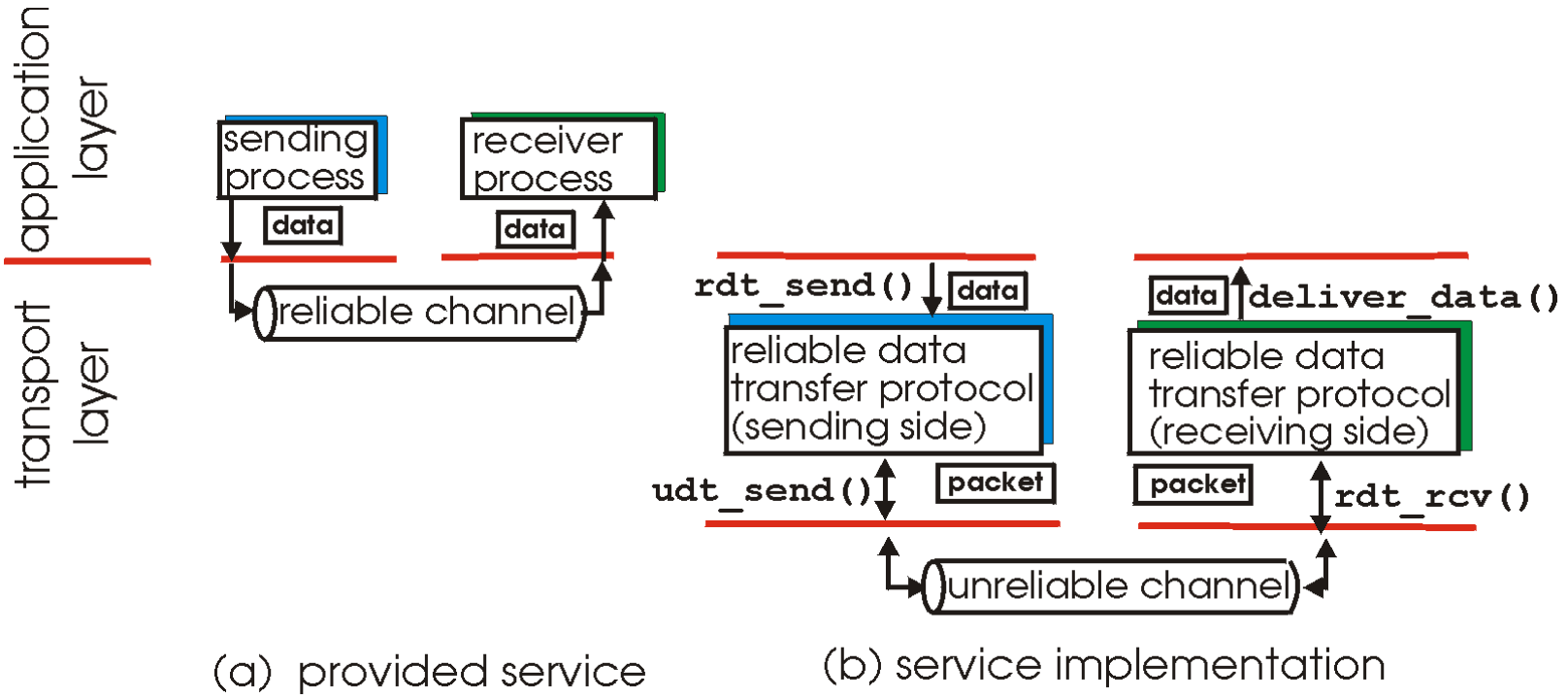
- important in *application, transport, link* layers
  - *top-10 list of important networking topics!*



- *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!

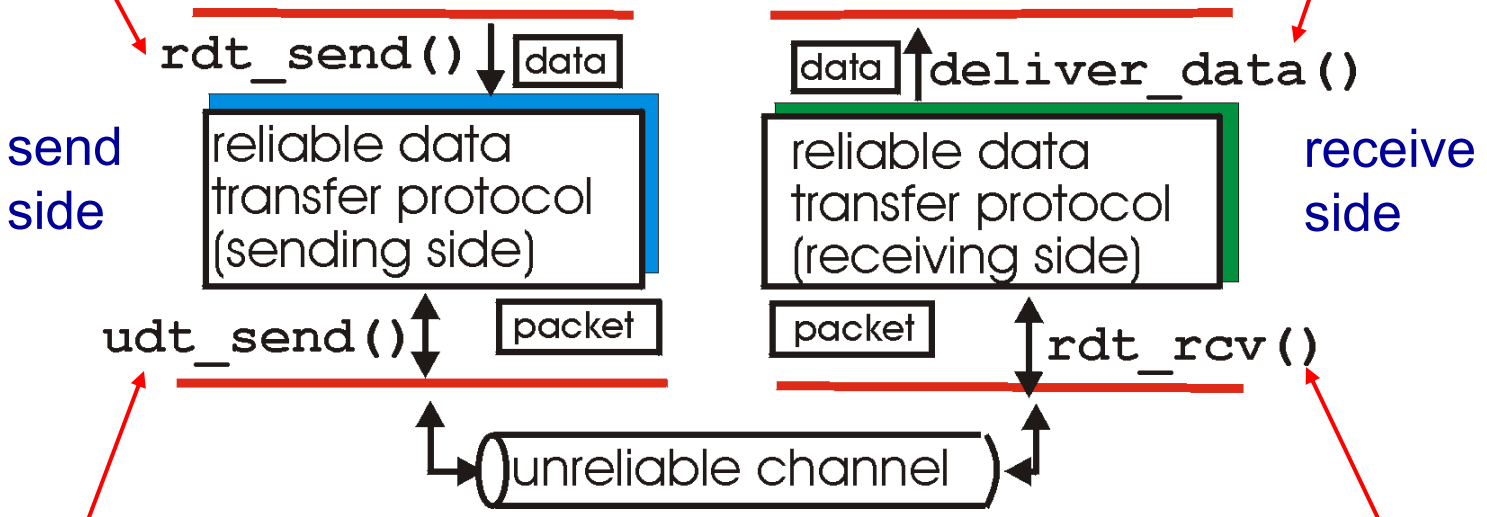


- *characteristics of unreliable channel* will determine ***complexity*** of reliable data transfer protocol (rdt)

Reliable Data Transfer: Getting Started

`rdt_send()`: called from above, (e.g., by app.). Passed data received from 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

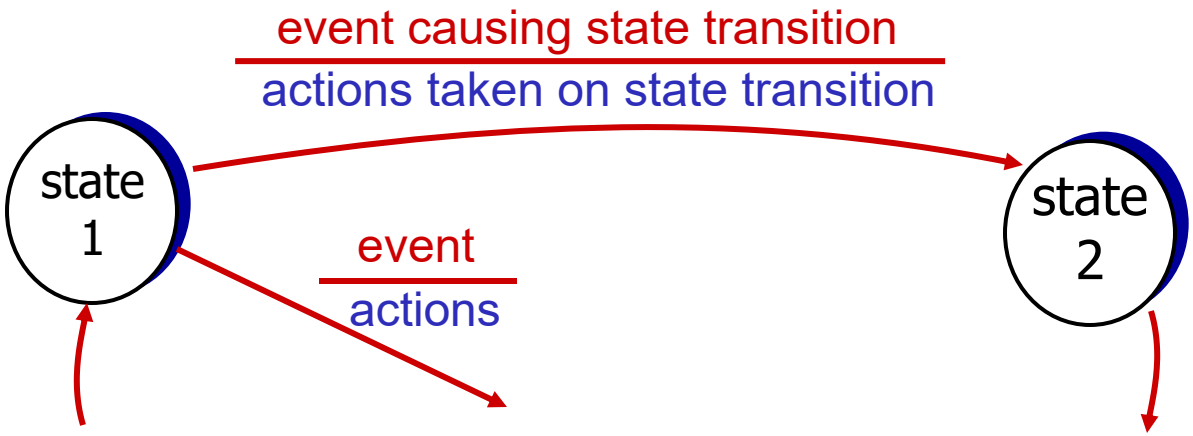
`rdt_rcv()`: called when packet arrives on rcv-side of channel

Reliable Data Transfer: Getting Started

we'll:

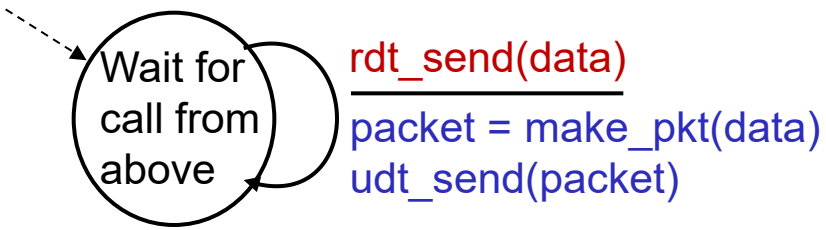
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only *unidirectional* data transfer
  - but control info will flow on both directions!
- use *finite state machines* (FSM) to specify sender, receiver

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

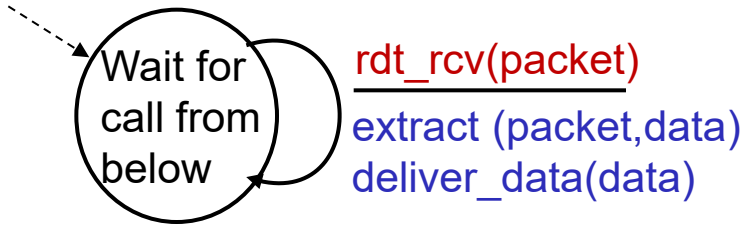


rdt1.0: Reliable Transfer over a Reliable Channel

- underlying channel *perfectly reliable* (not realistic!)
  - 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

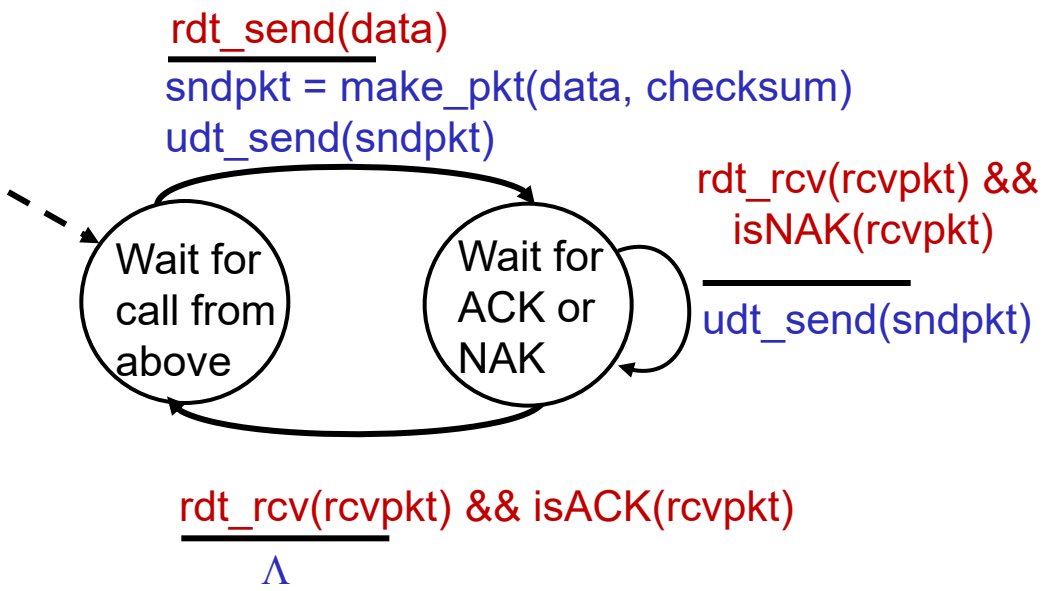
- underlying channel may flip bits in packet
  - use checksum to detect bit errors
- *the question*: how to recover from errors:

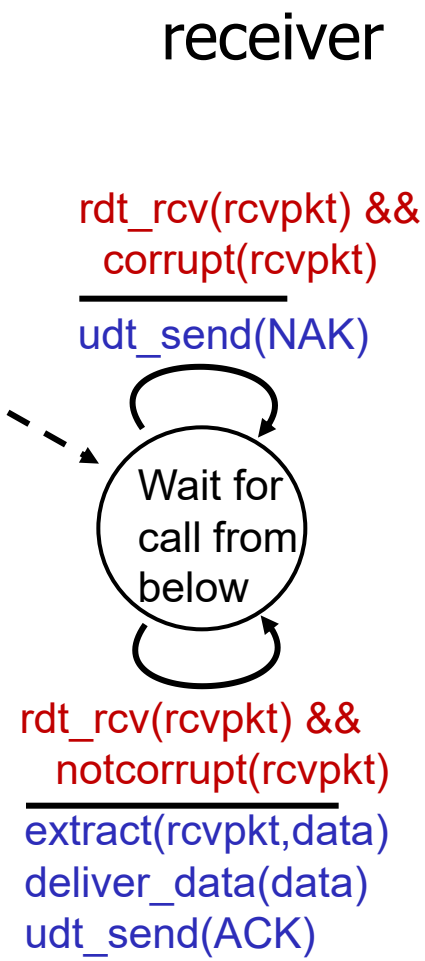
How do human recover from "errors" during conversation?

rdt2.0: Channel with Bit Errors

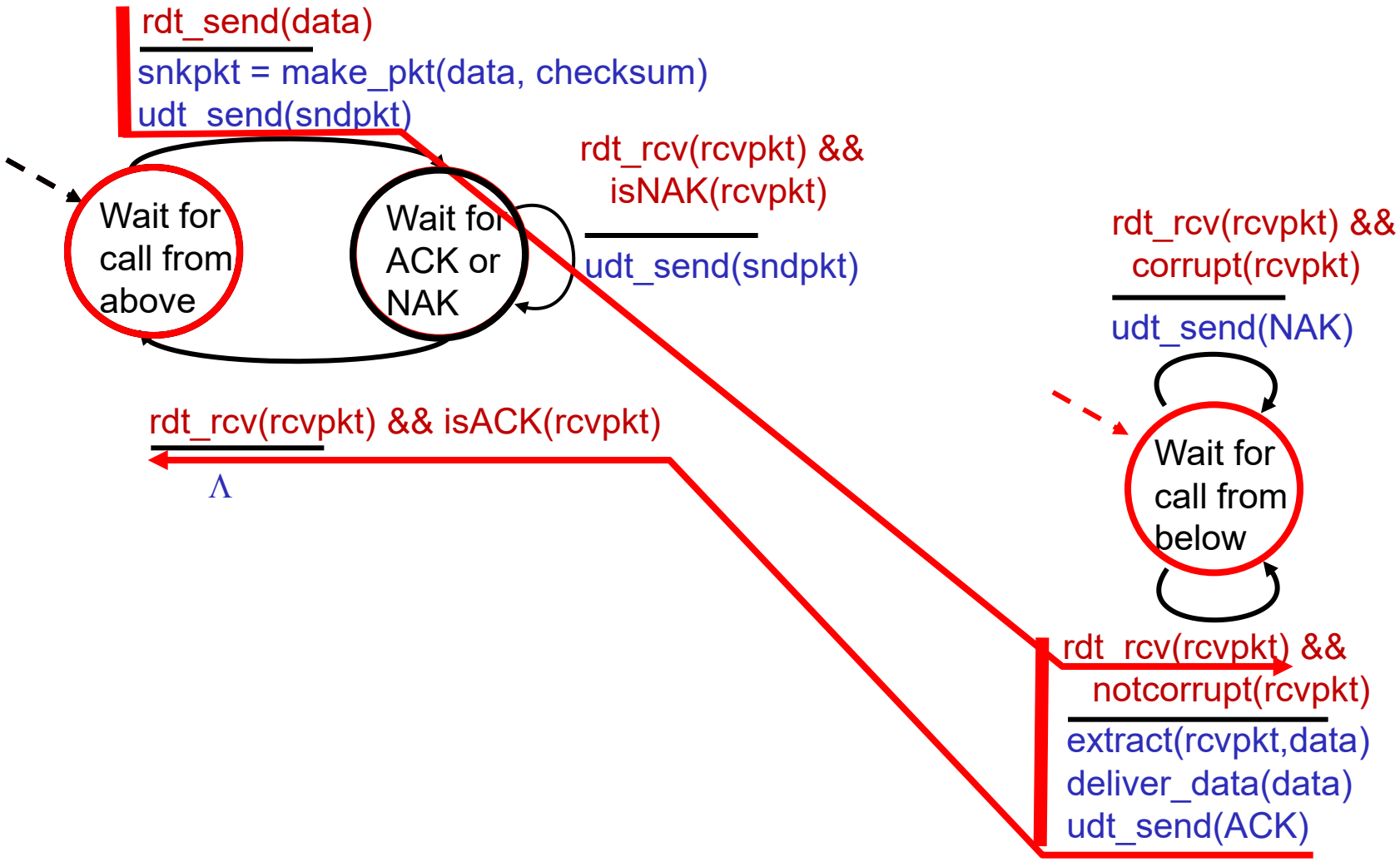
- underlying channel may flip bits in packet
  - how does the receiver know this – *error detection*
  - e.g. checksum to detect bit errors
- *the question*: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - *error detection*
  - *feedback*: control msgs (ACK, NAK) from receiver to sender

rdt2.0: FSM Specification

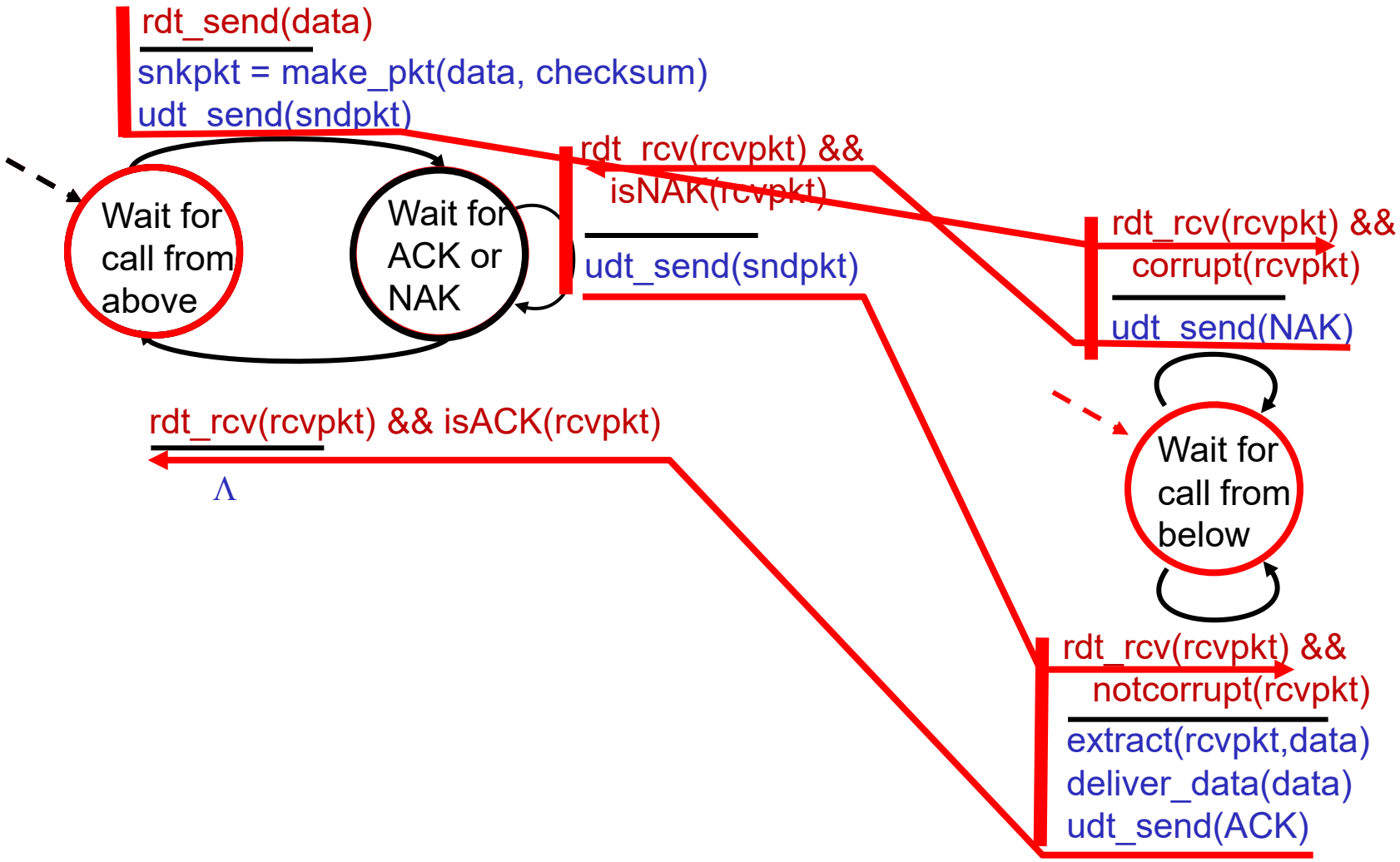




rdt2.0: Operation with no Errors



rdt2.0: Error Scenario



rdt2.0 has a Fatal Flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- sender retransmits current pkt if ACK/NAK corrupted
- may lead to *duplicate packets!*

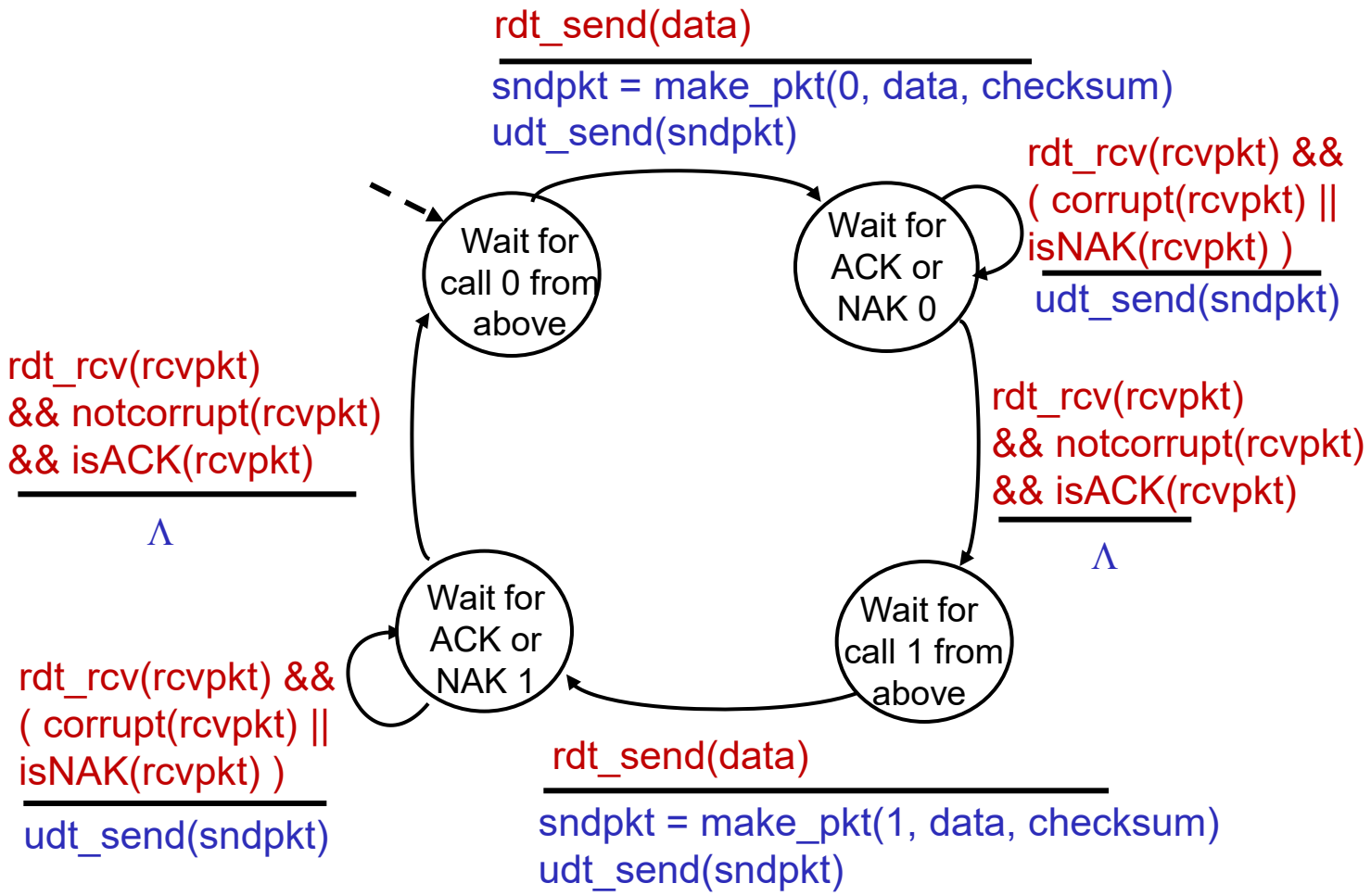
handling duplicates:

- How to detect packet duplication?
  - sender adds *sequence number* to each pkt
  - Receiver checks *sequence number*
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

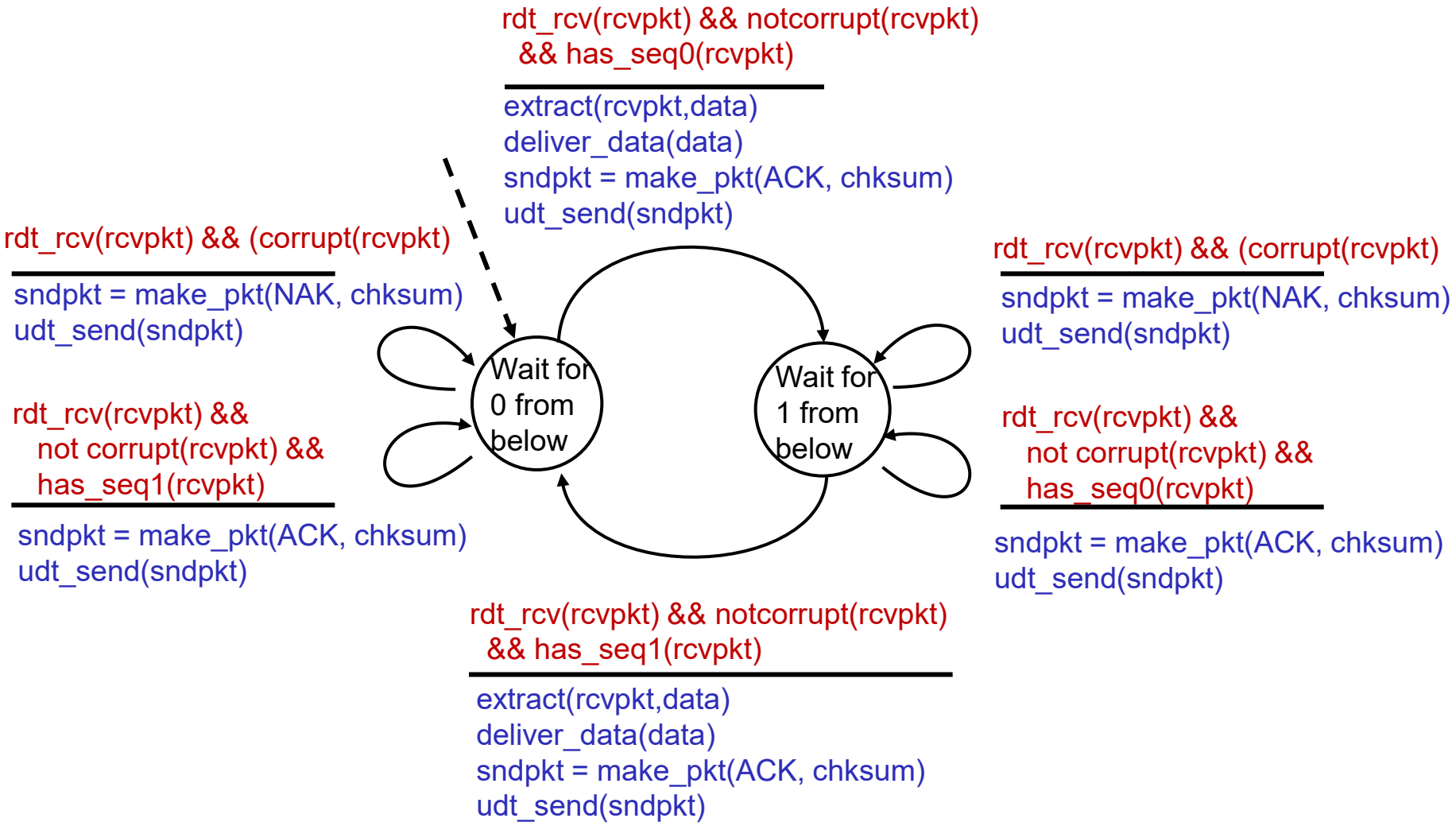
sender sends one packet, then waits for receiver response

rdt2.1: Handles Garbled ACK/NAKs – Sender



1-bit sequence numbers are used to detect duplicated packets, ACK 0 means ACK for packet with sequence number 0

rdt2.1: Handles Garbled ACK/NAKs – Recv.



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

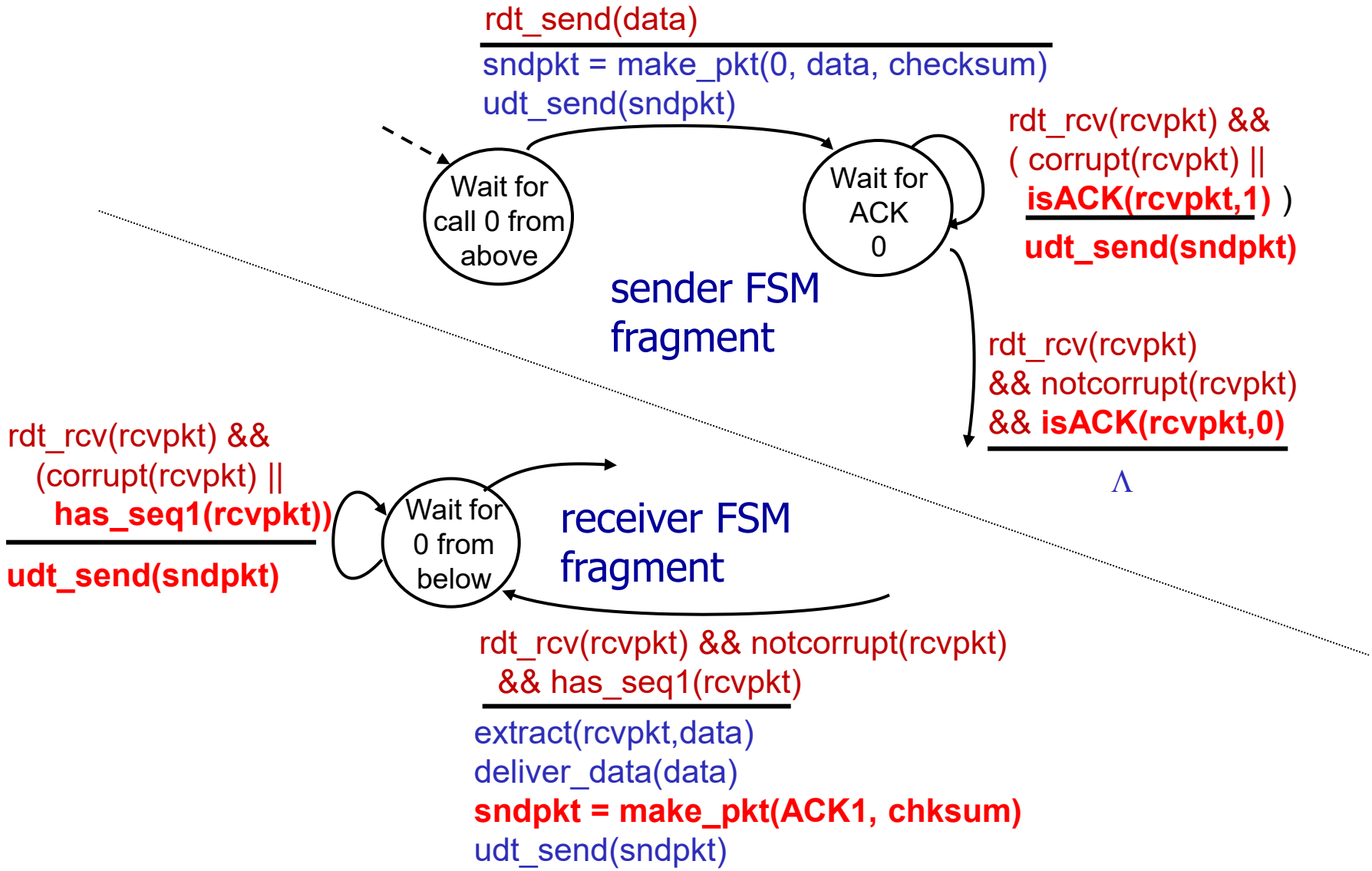
receiver:

- 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*

rdt2.2: Sender, Receiver Fragments



rdt3.0: Channels with Errors *and* Loss

new assumption:

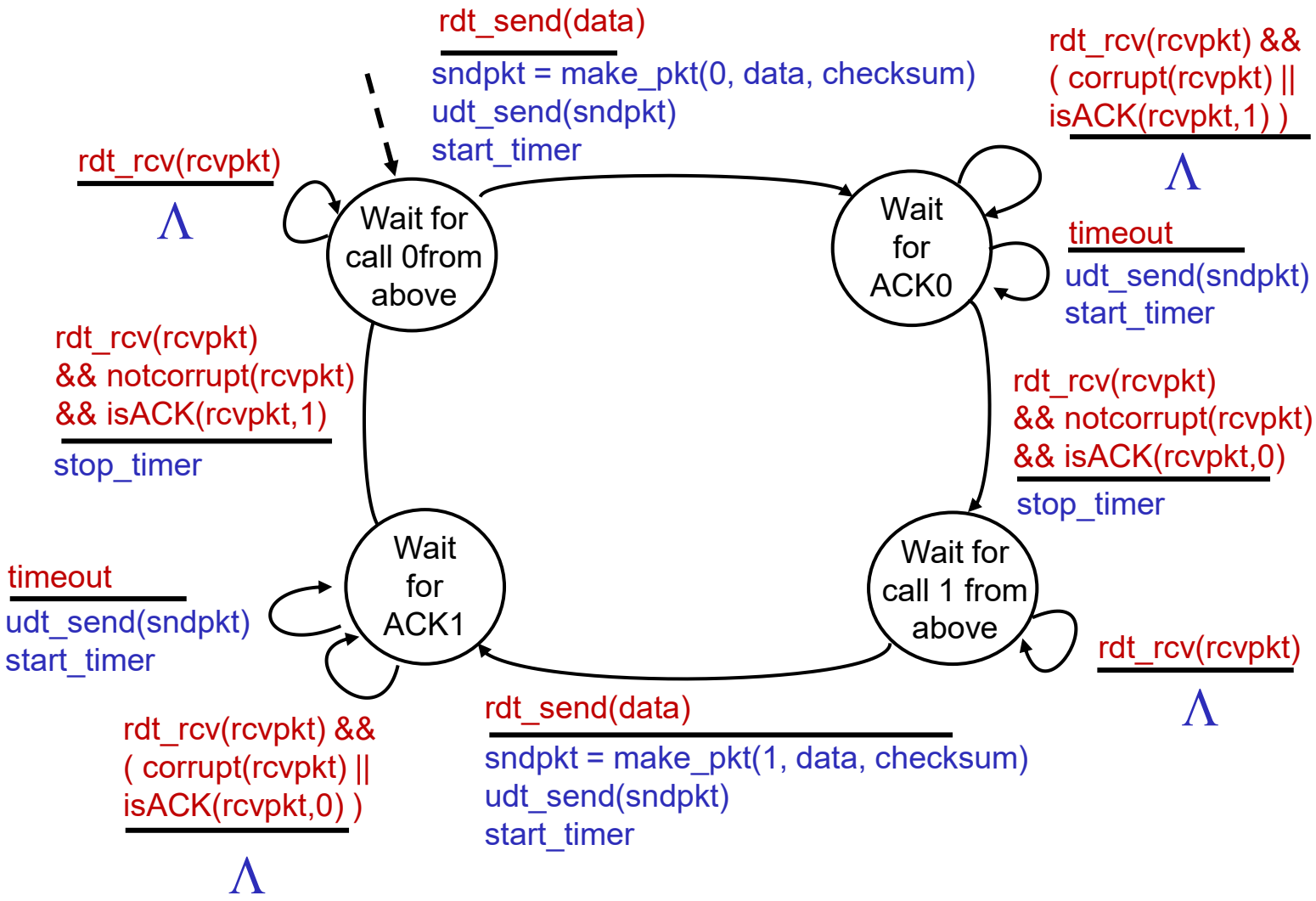
underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help … but *not enough!*
- *What if ACK packet were lost?*

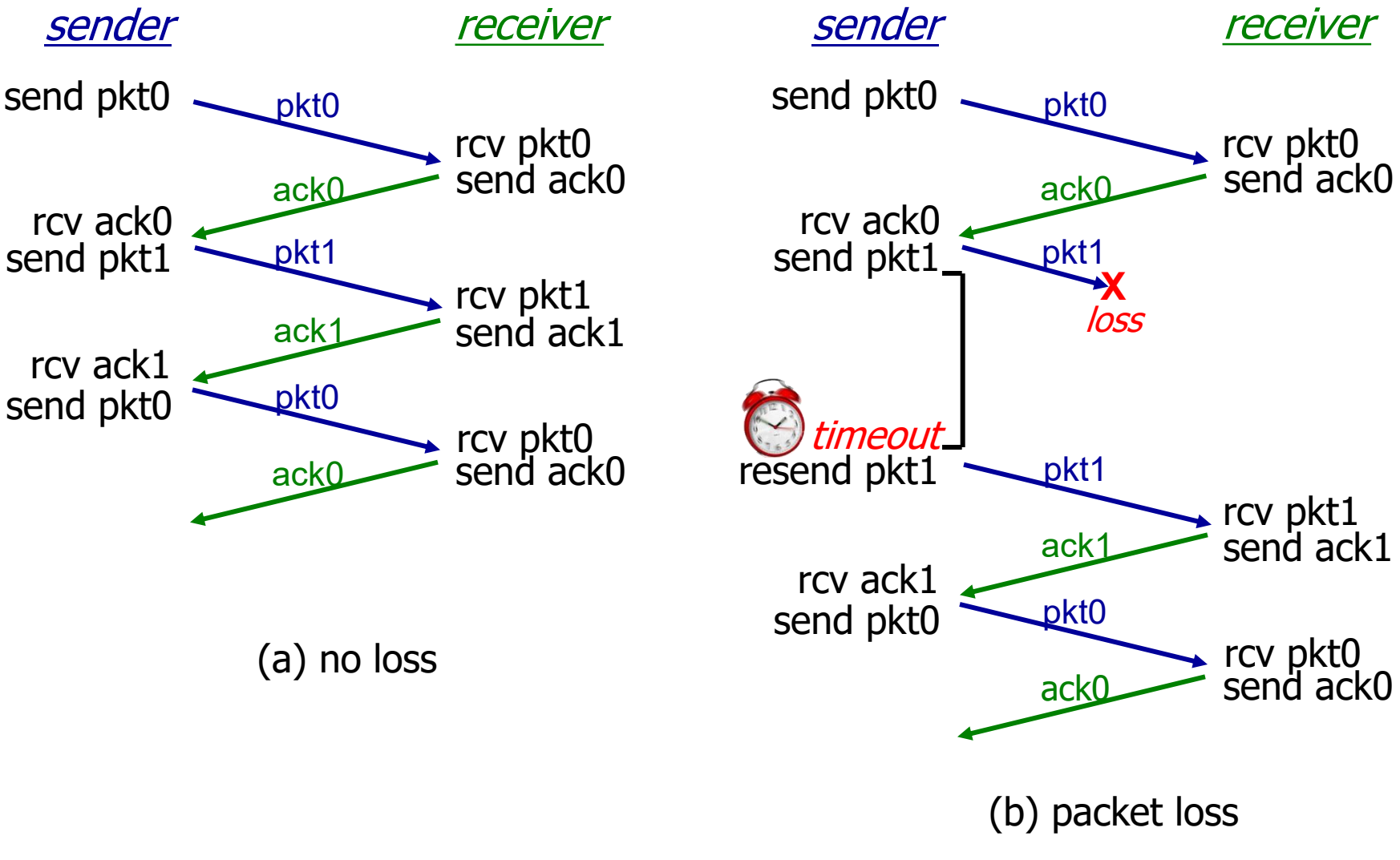
approach: sender waits "reasonable" amount of time for ACK

- 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 pkt being ACKed
- requires *countdown timer*

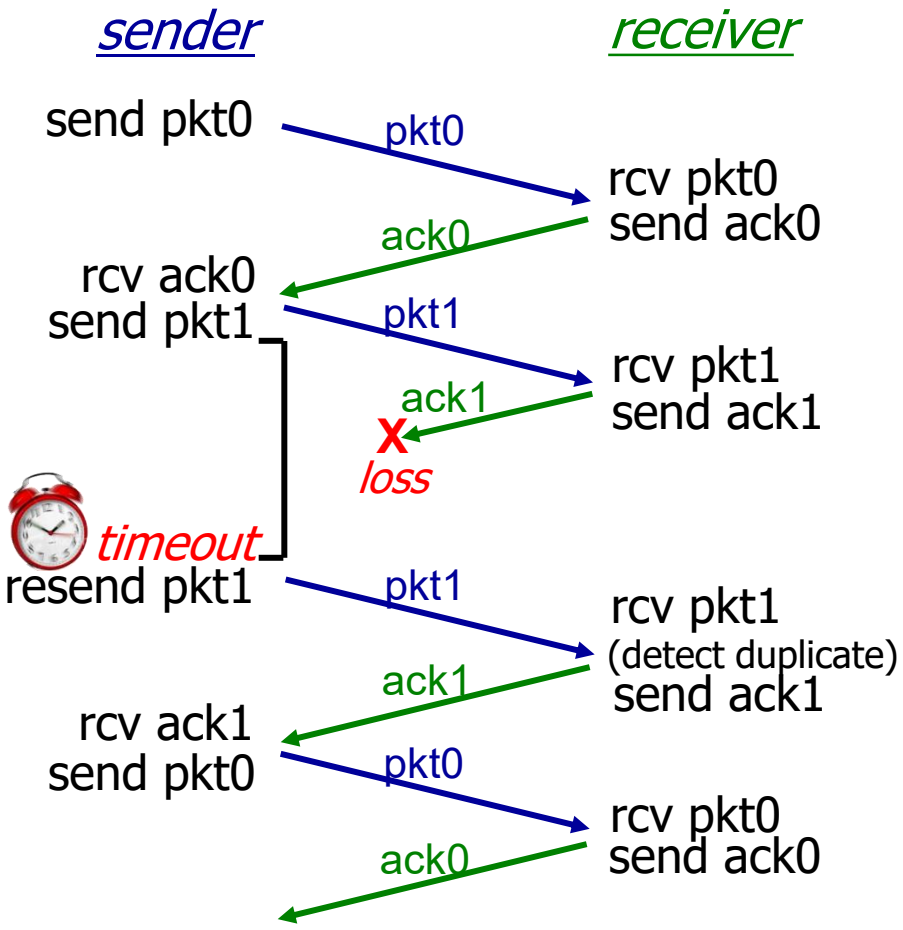
rdt3.0 Sender



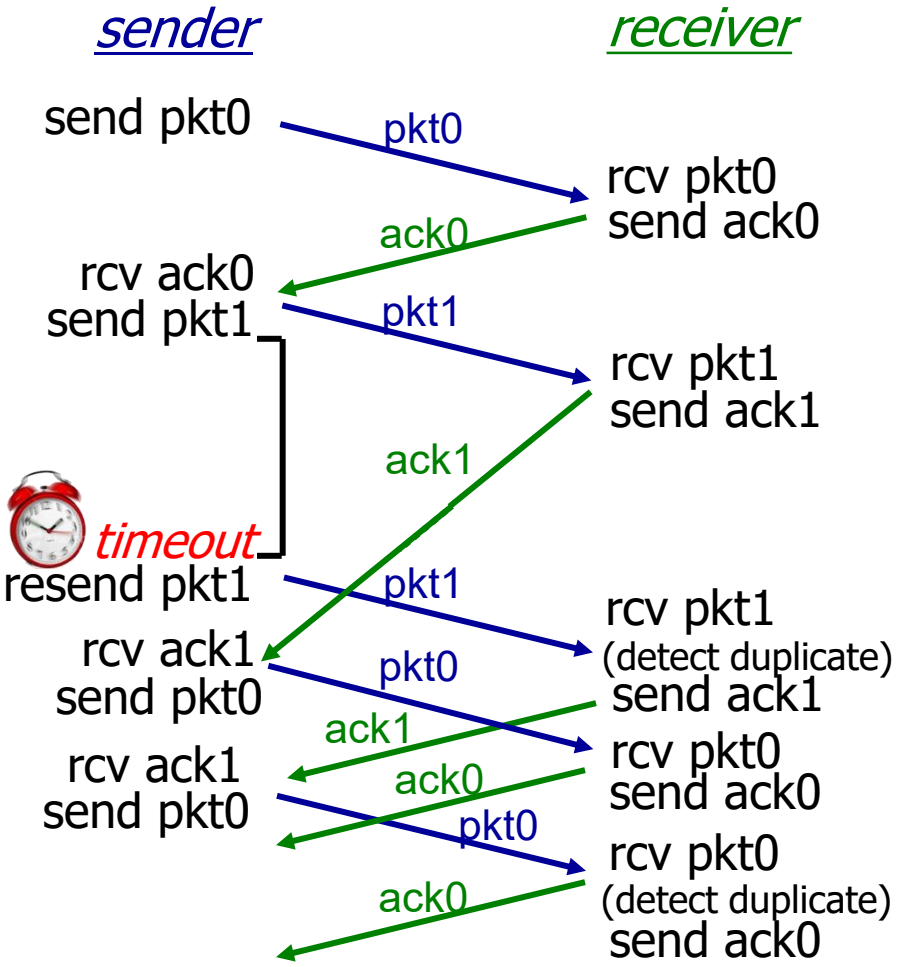
rdt3.0 in Action



rdt3.0 in Action



(c) ACK loss



(d) premature timeout/ delayed ACK