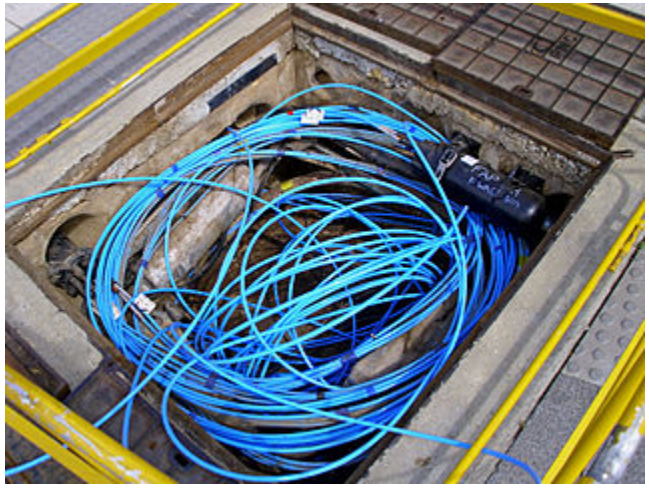
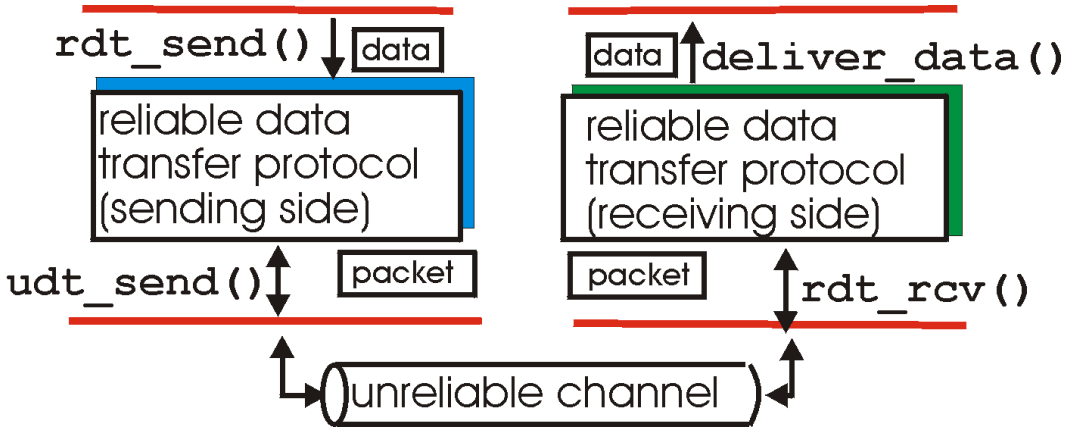
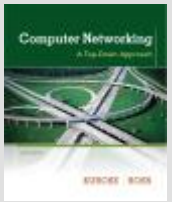
Principles of reliable data transfer





Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley



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

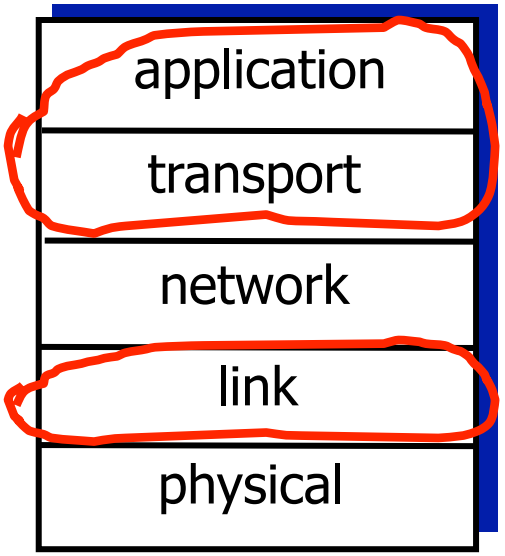
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Overview

- Reliable data transfer
  - Getting data there despite unreliable channel
  - Important for application, transport and link layers
  - Central problem in networking
- Our approach
  - Start simple, build increasingly sophisticated protocols to handle problems
  - Discuss in generality since problem occurs not just at transport layer

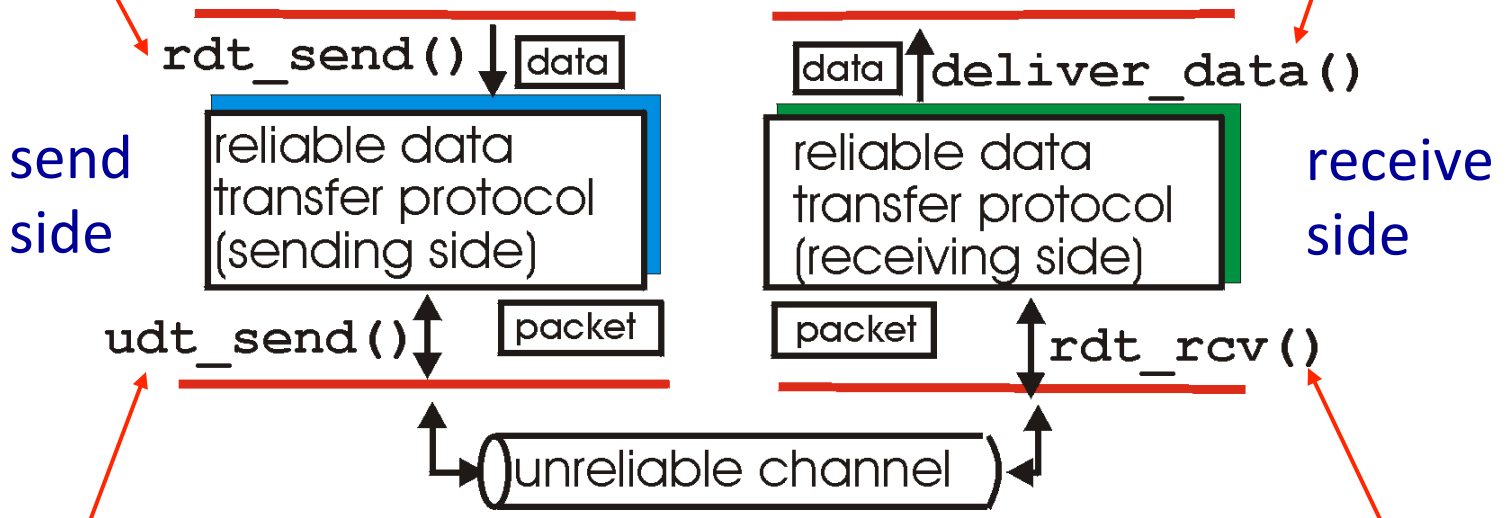


❖ Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above (e.g. by application). Passed data to deliver to receiver from upper layey.

deliver_data(): called by **rdt** to deliver data to upper

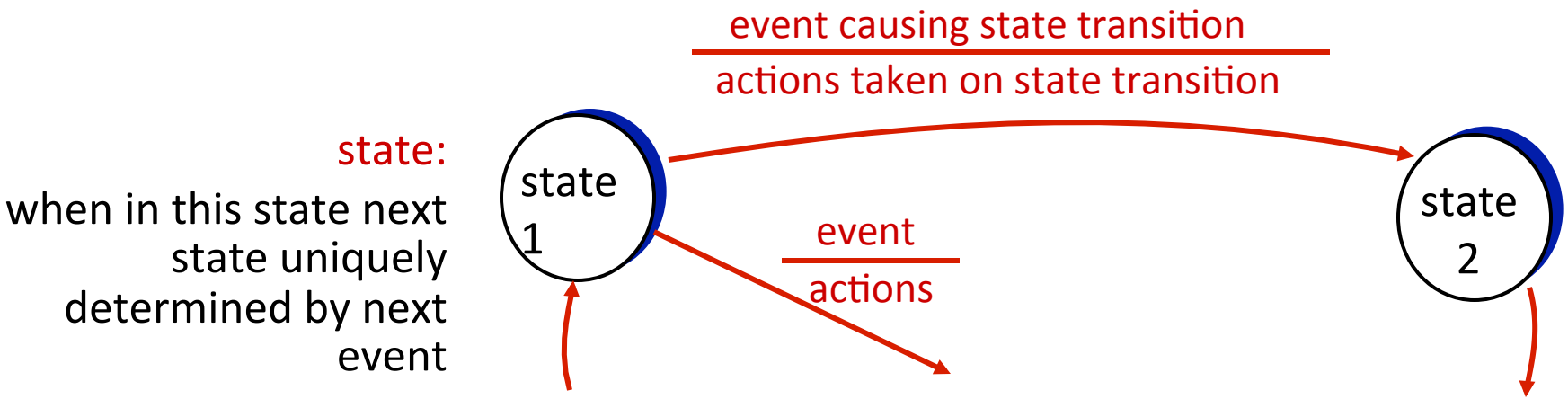


udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

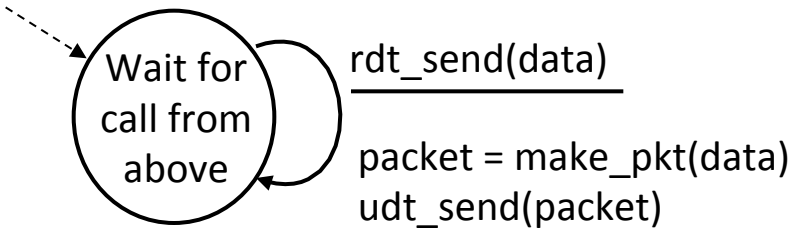
Reliable data transfer: getting started

- Incrementally develop sender and receiver sides
  - reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - But control info will flow on both directions!
- Specify using finite state machine (FSM)
  - One FSM for sender and one FSM for receiver

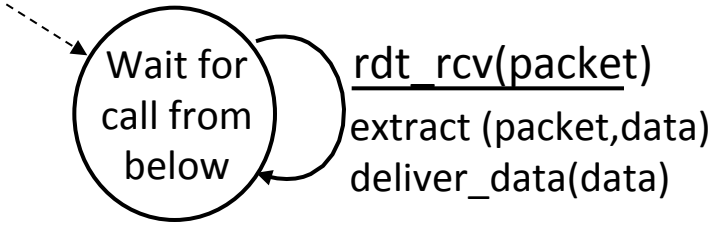


rdt1.0: transfer over reliable channel

- 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

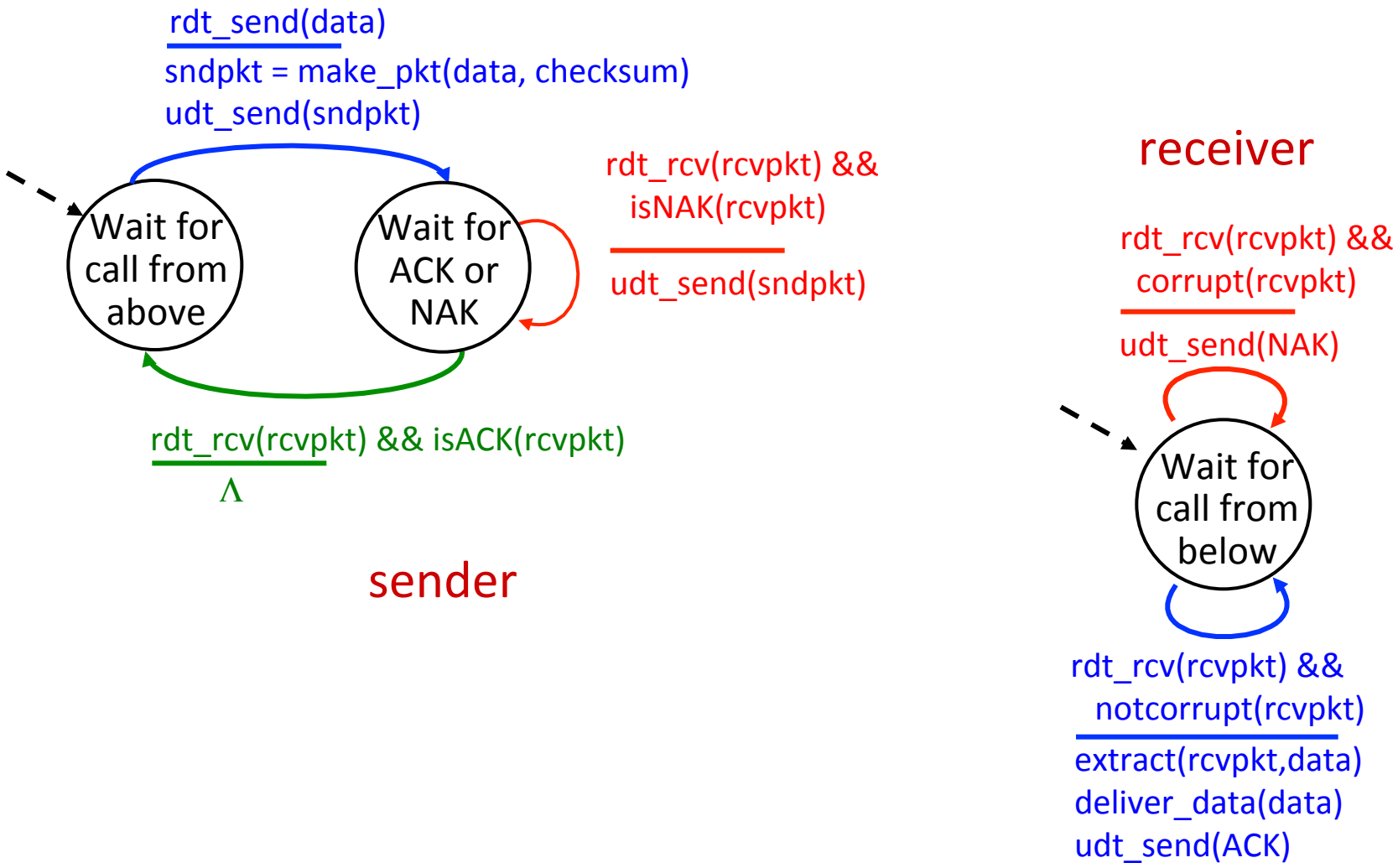


receiver

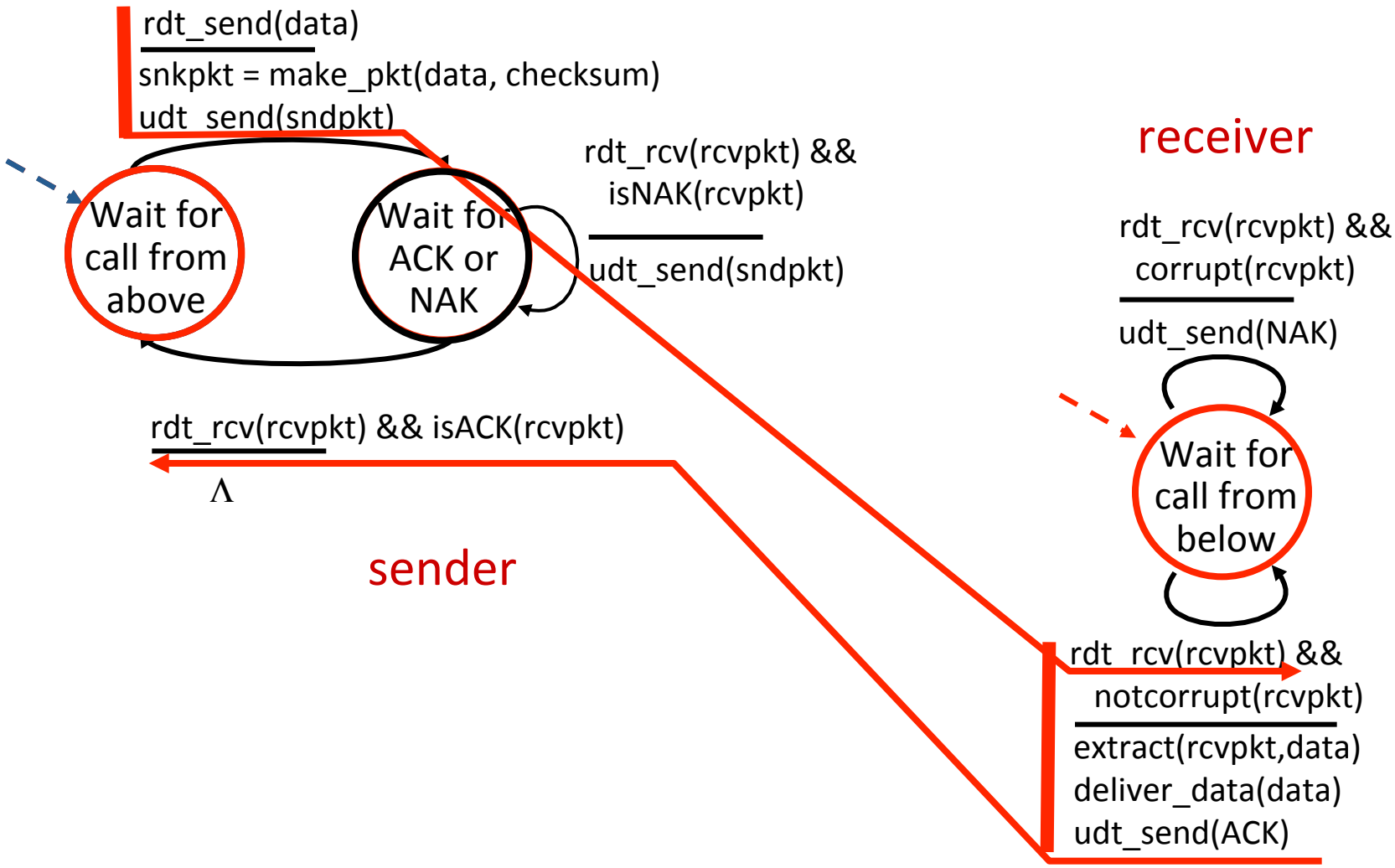
rdt2.0: channel with bit errors

- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- How to recover from errors?
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that packet received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - Sender retransmits packet on receipt of NAK
- New mechanisms in `rdt2.0`:
  - Error detection
  - Feedback: control msgs (ACK,NAK), receiver to sender
  - Known as an ARQ (Automatic Repeat reQuest) protocol

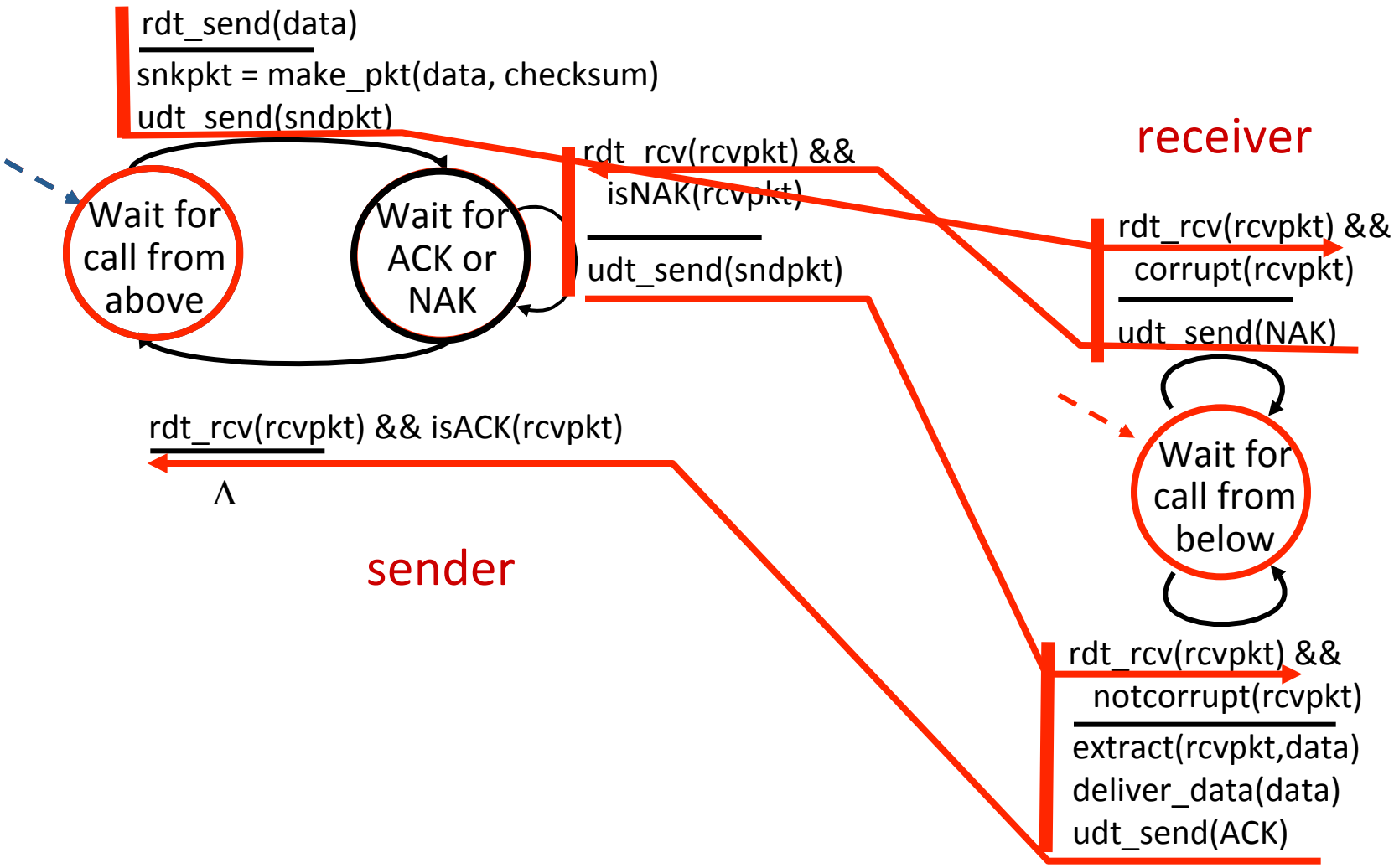
rdt2.0: FSM specification



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!
- Can't just retransmit: possible duplicate

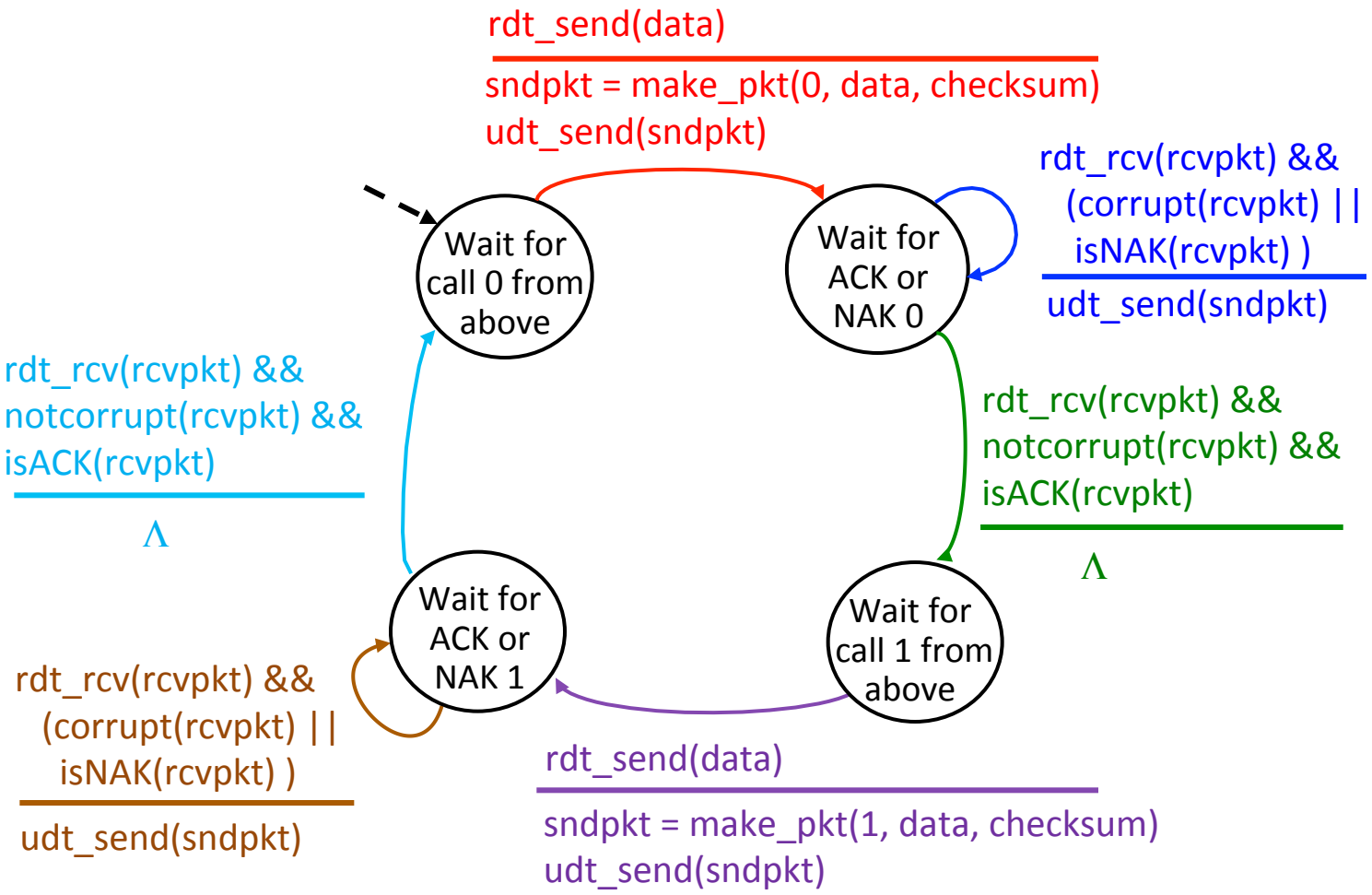
Handling duplicates:

- Sender retransmits current packet if ACK/NAK corrupted
- Sender adds *sequence number* to each packet
- Receiver discards (doesn't deliver up) duplicate packet

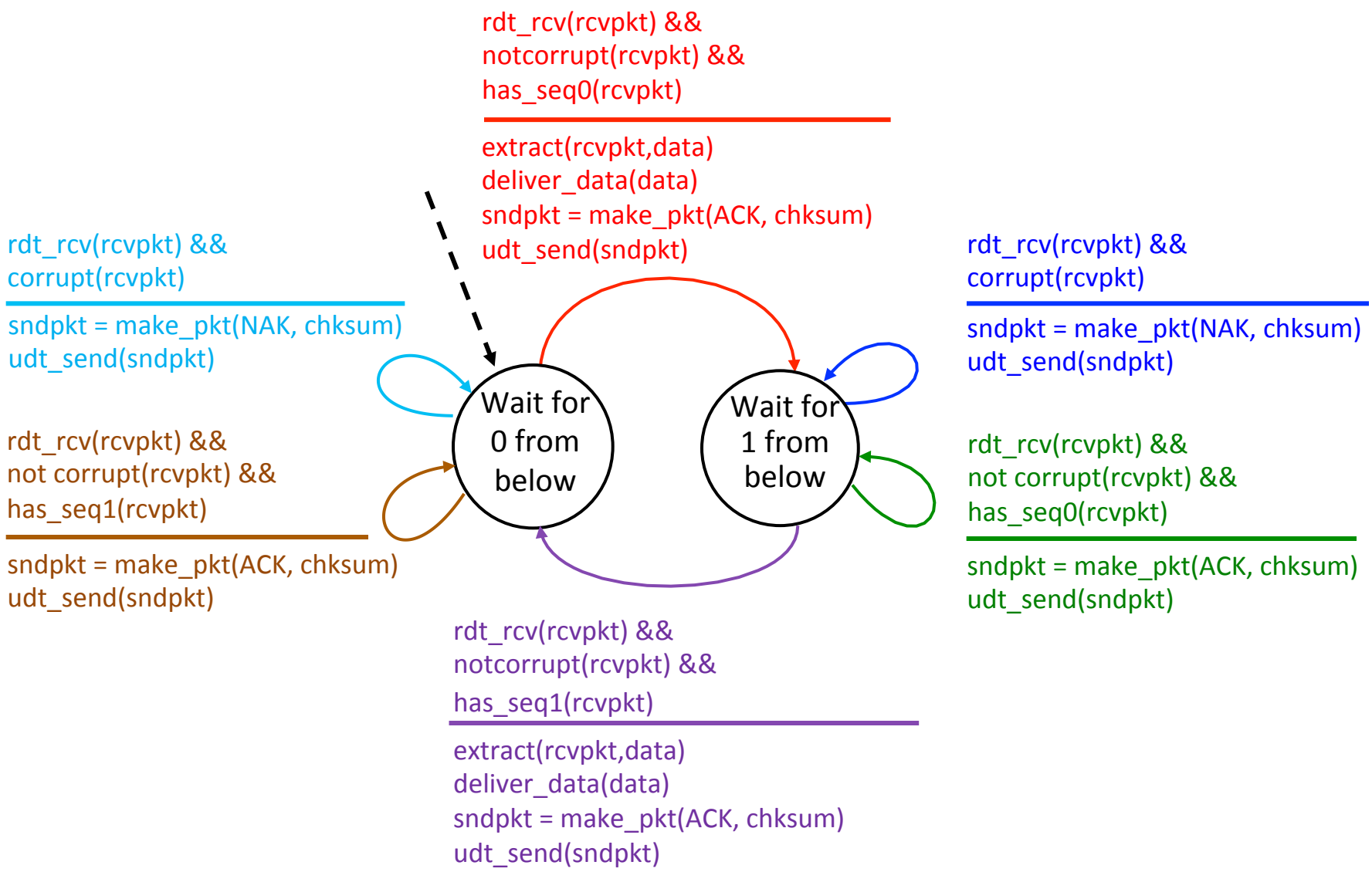
Stop and wait

Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAK



rdt2.1: receiver, handles garbled ACK/NAK



rdt2.1: discussion

Sender:

- Sequence # added to packet
- Two sequence #'s (0,1)
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must remember whether expected packet should have sequence # of 0 or 1

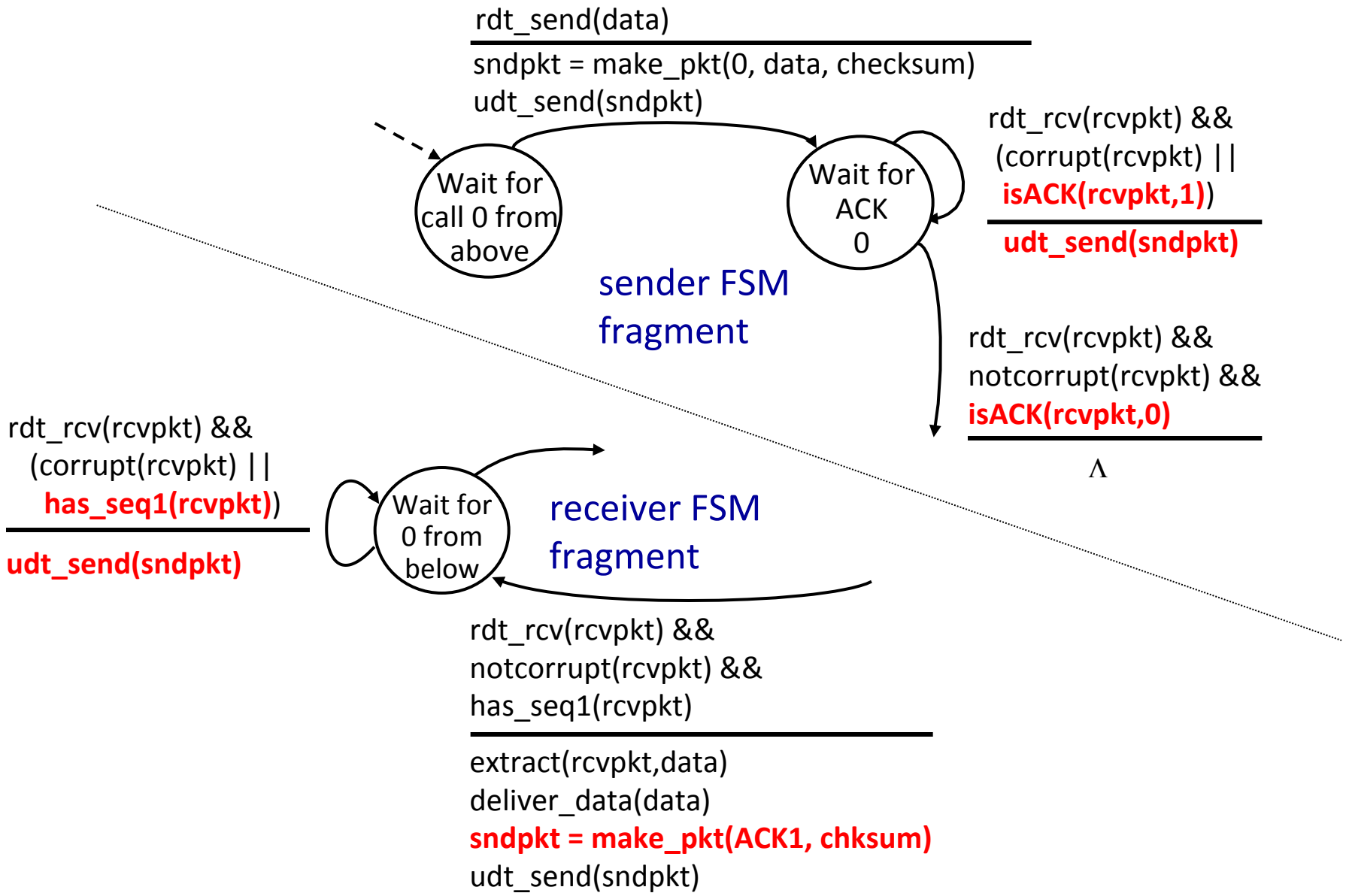
Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected packet sequence #
- 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 packet received OK
  - Receiver must *explicitly* include sequence # of packet being ACKed
  - An *alternating-bit protocol*
- Duplicate ACK at sender, same action as NAK:
  - Retransmit current packet

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

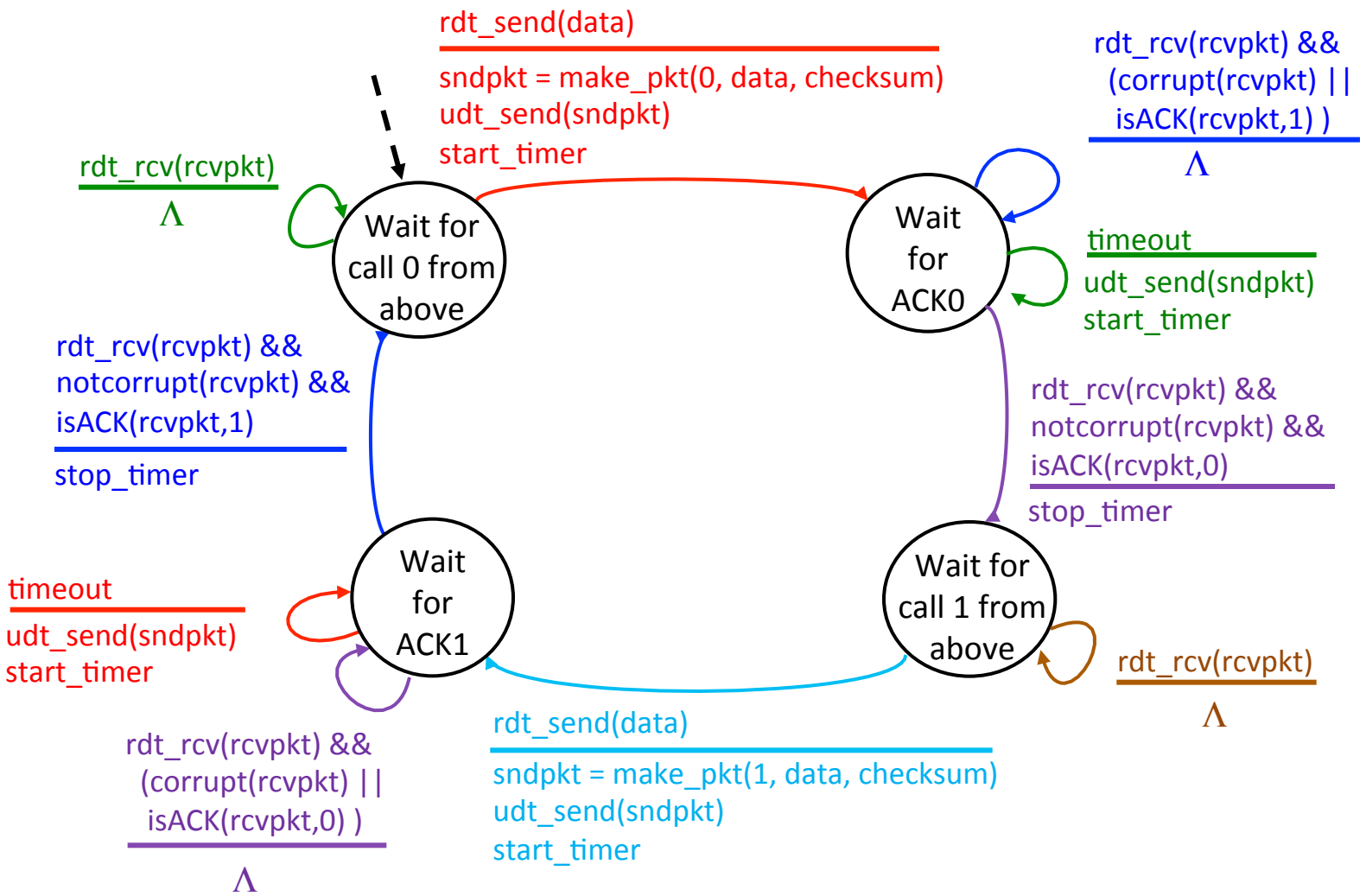
New assumption:

- Underlying channel can also lose packets
  - Lose data or ACKs
  - Checksum, seq. #, ACKs, retransmissions will be of help … but not enough

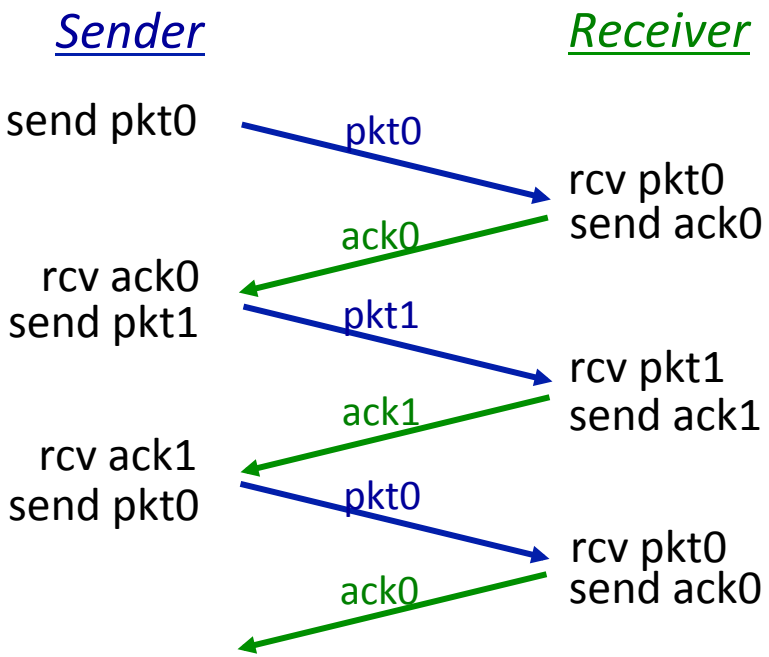
Approach:

- Sender waits "reasonable" amount of time for ACK
  - Retransmits if no ACK received in this time
  - If packet (or ACK) just delayed (not lost):
    - Retransmission will be duplicate, but sequence #'s already handles this
    - Receiver must specify sequence # of packet being ACKed
  - Requires countdown timer

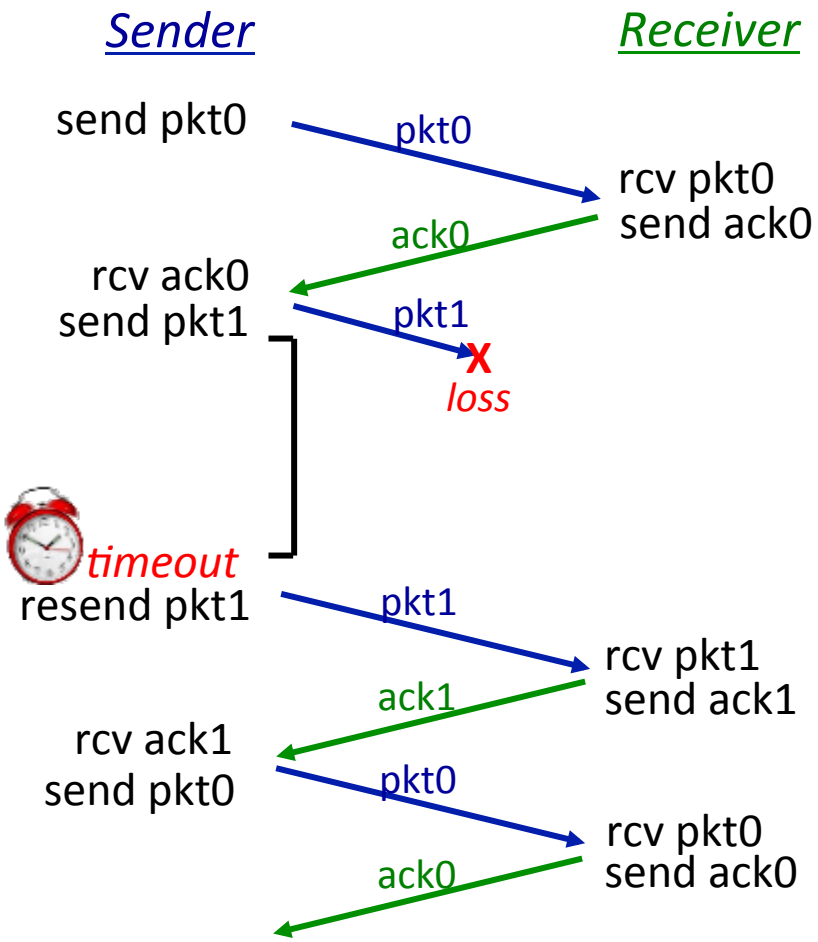
rdt3.0 sender



rdt3.0 in action

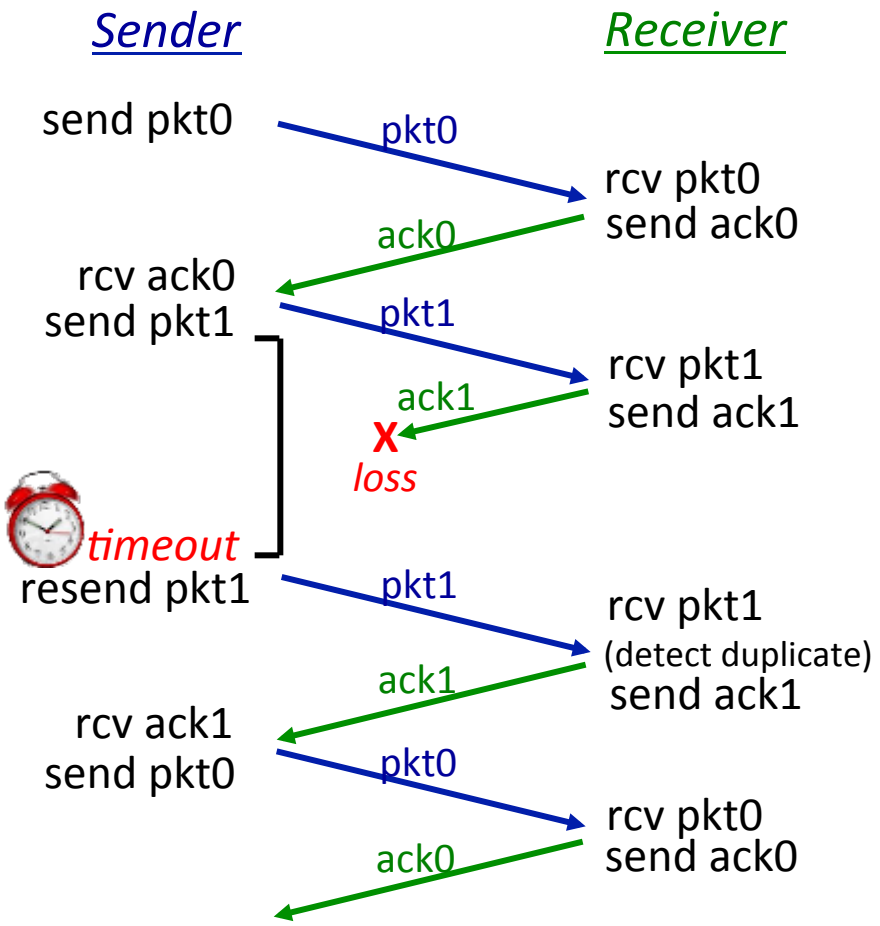


(a) no loss

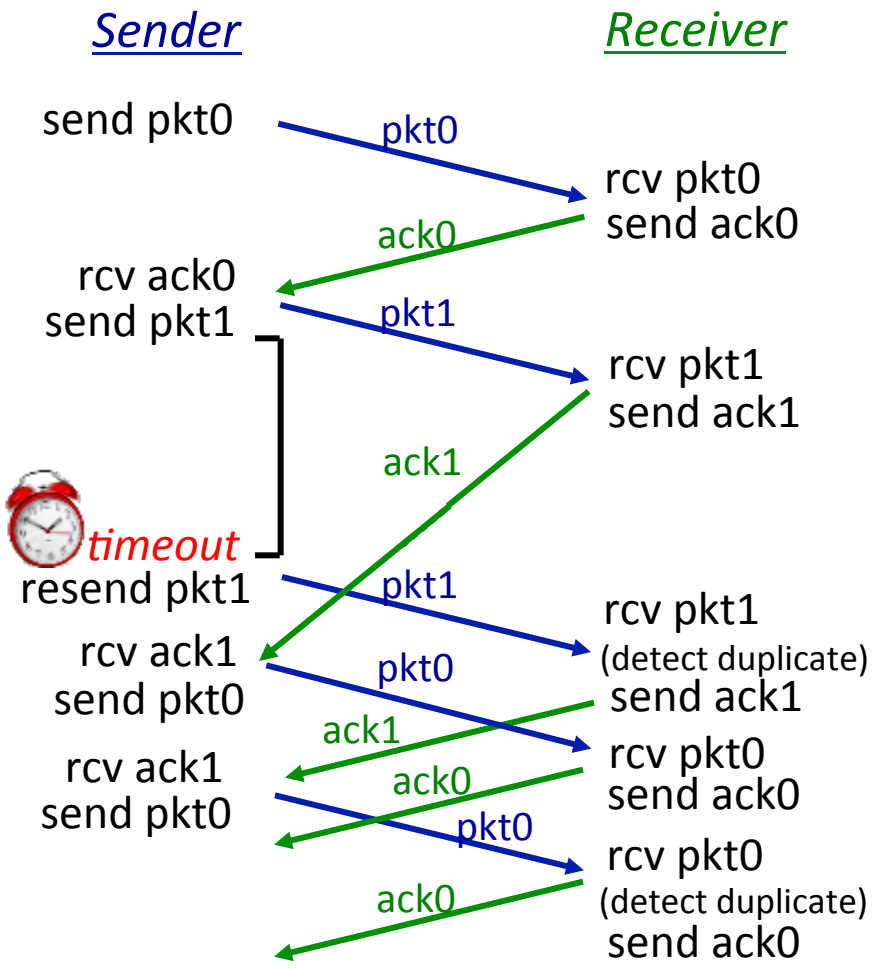


(b) packet loss

rdt3.0 in action



(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, 8000 bit packet:

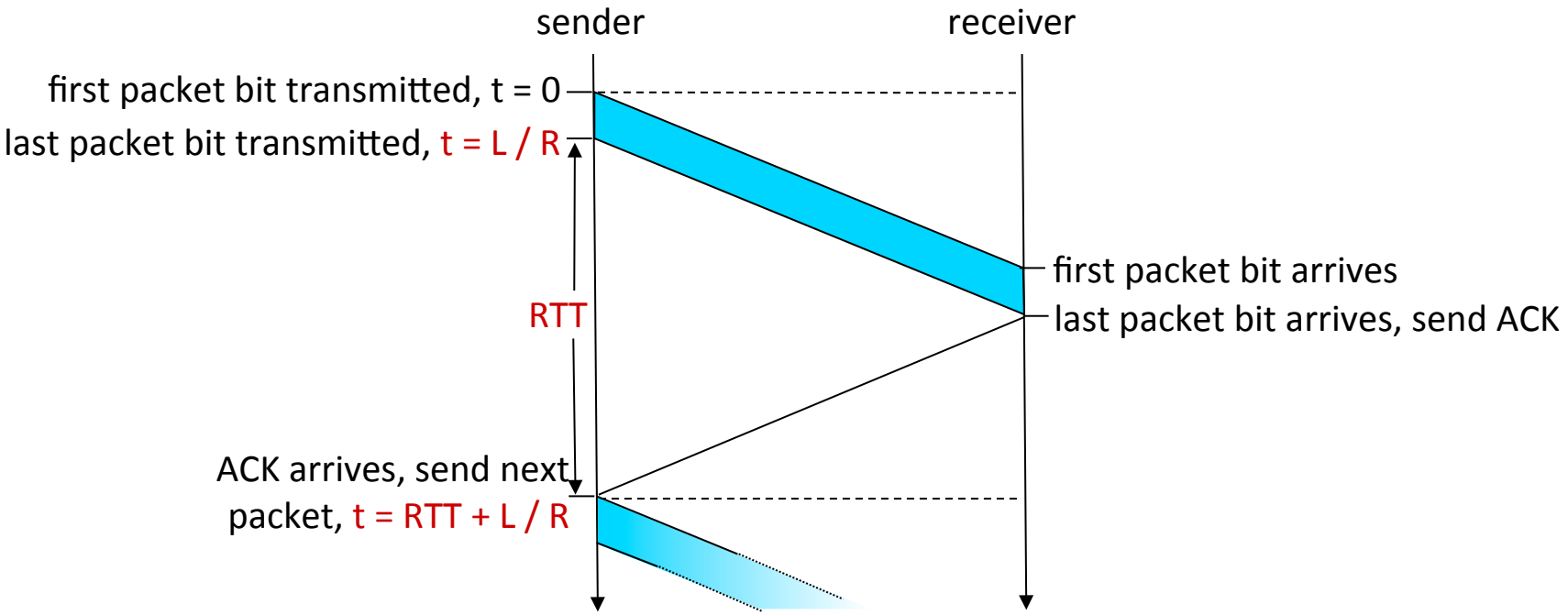
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

  - 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 packet every 30 msec: 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



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

Summary

- Reliable data transport
  - Tricky if packets corrupted, lost, or delayed
  - Uses acknowledge messages, ACKs
  - Uses sequence numbers
  - Uses timers
- rdt3.0
  - Stop-and-wait protocol
  - Alternating bit protocol
  - ARQ (Automatic Repeat reQuest) protocol
  - But terrible network utilization
    - We'll fix next time