

CSCE 463/612

Networks and Distributed Processing
Fall 2025

Transport Layer II

Dmitri Loguinov

Texas A&M University

October 7, 2025

Chapter 3: Roadmap

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

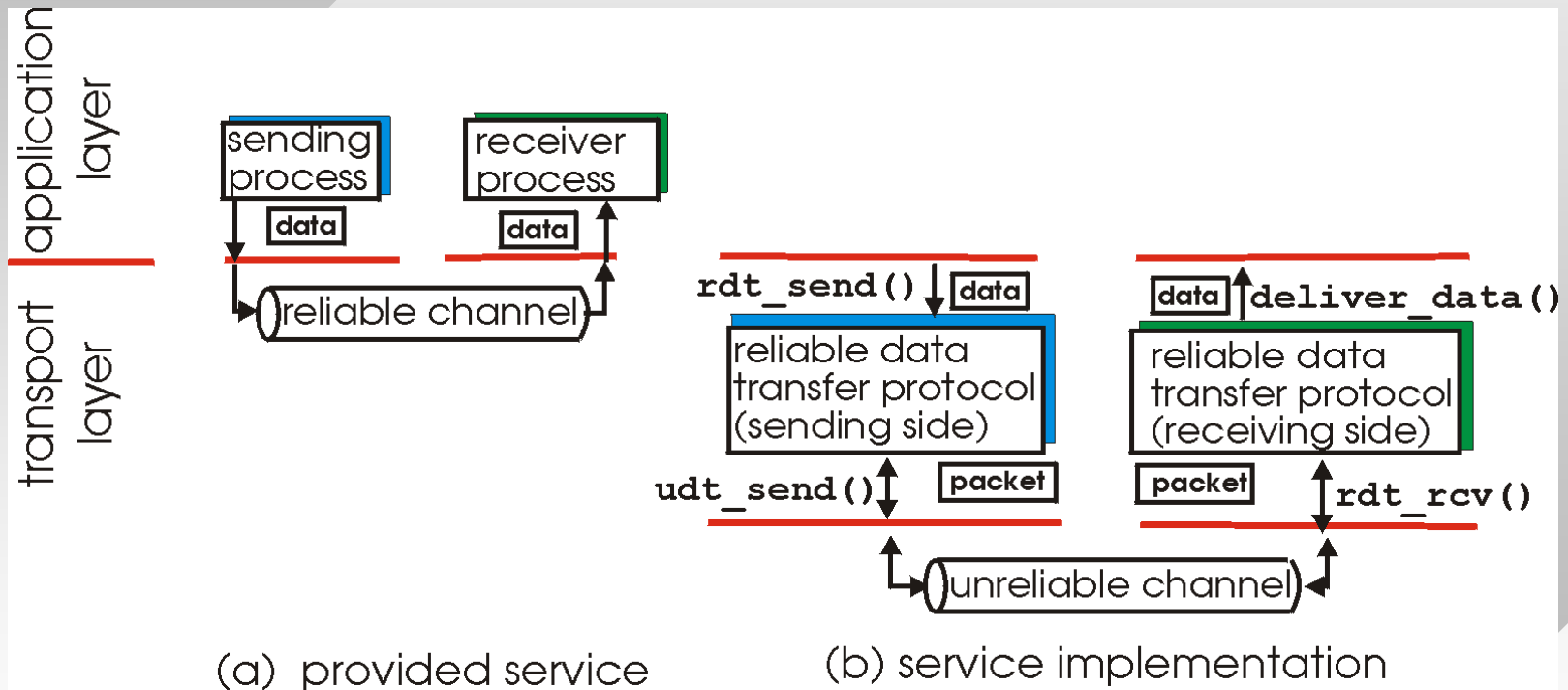
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

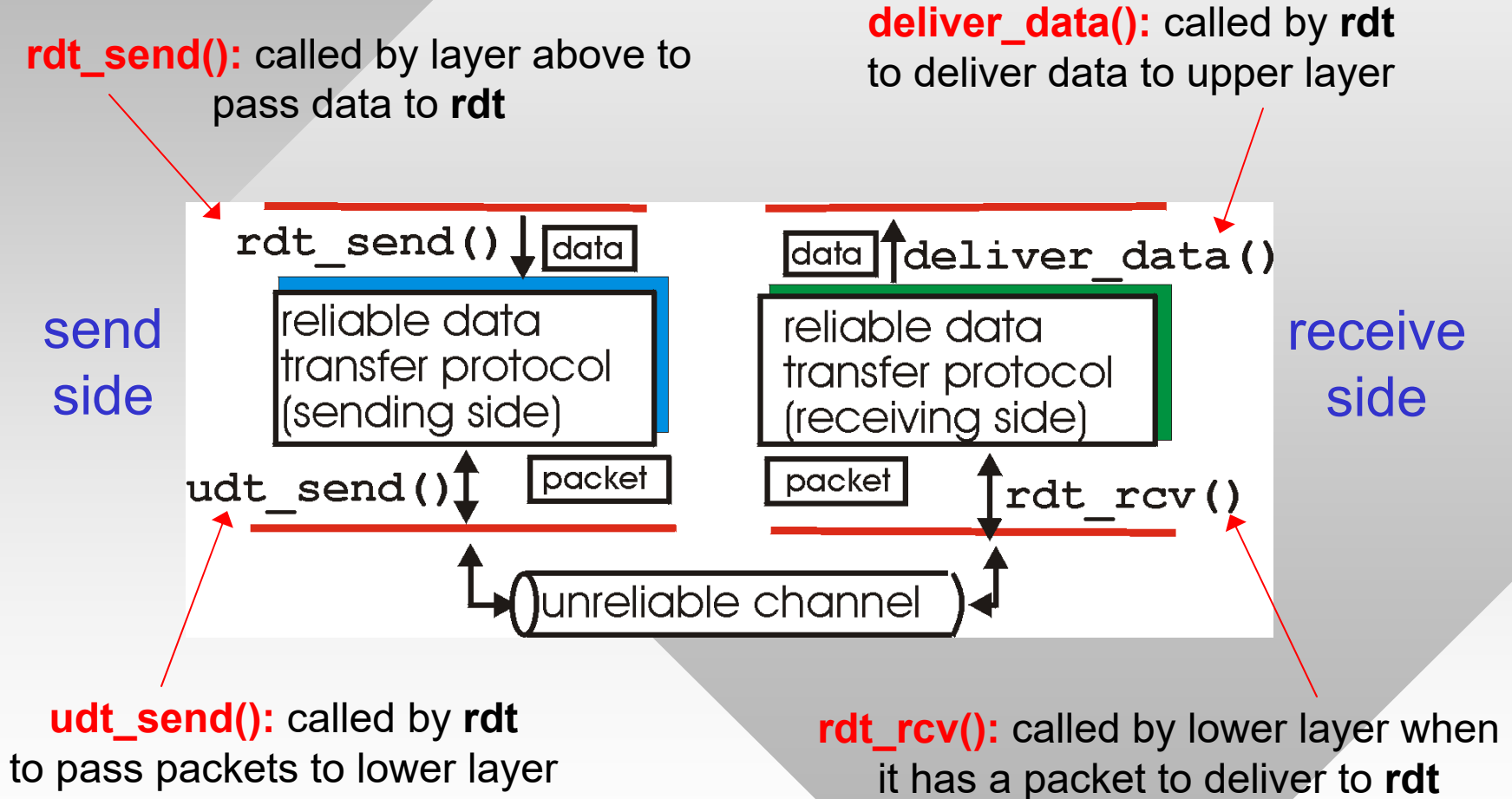
Principles of Reliable Data Transfer

- Important in application, transport, link layers



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

Reliable Data Transfer: Getting Started

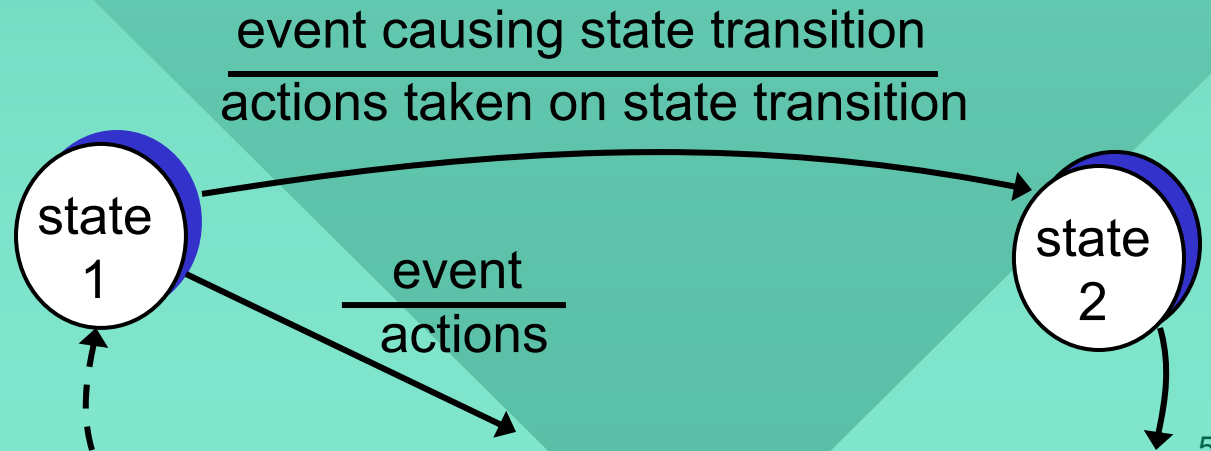


Reliable Data Transfer: Getting Started

We will:

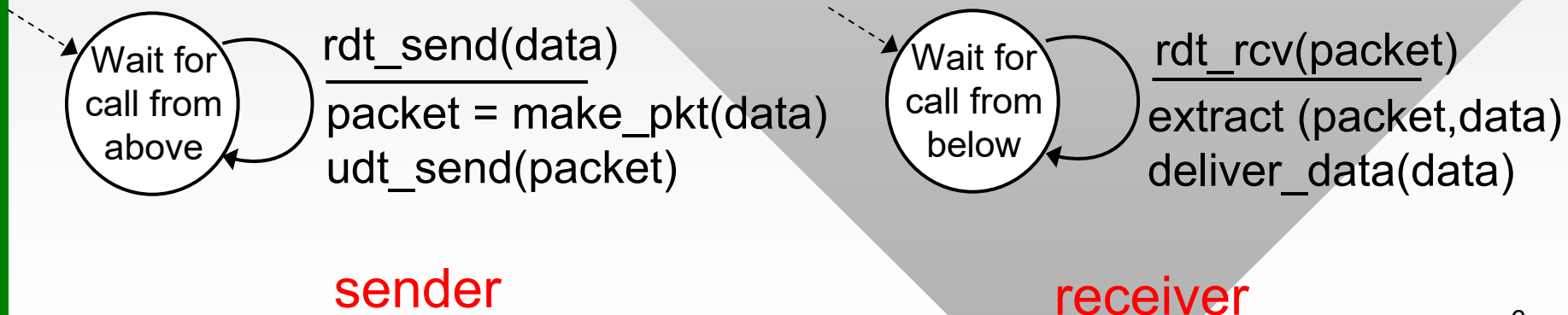
- Incrementally develop sender, receiver sides of *reliable data transfer* protocol (rdt)
- Consider only unidirectional data transfer
 - With receiver feedback, packets travel in both directions!
- Use **finite state machines** (FSM) to specify both sender and receiver

- From any state, the next state is uniquely determined by next event



Rdt1.0: Transfer Over a Reliable Channel

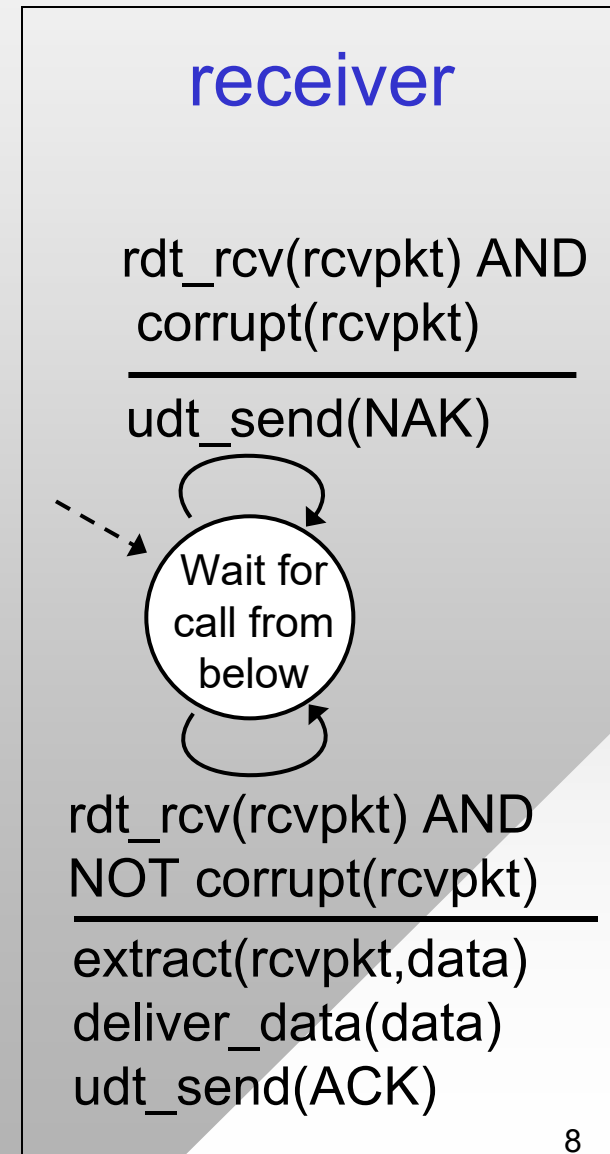
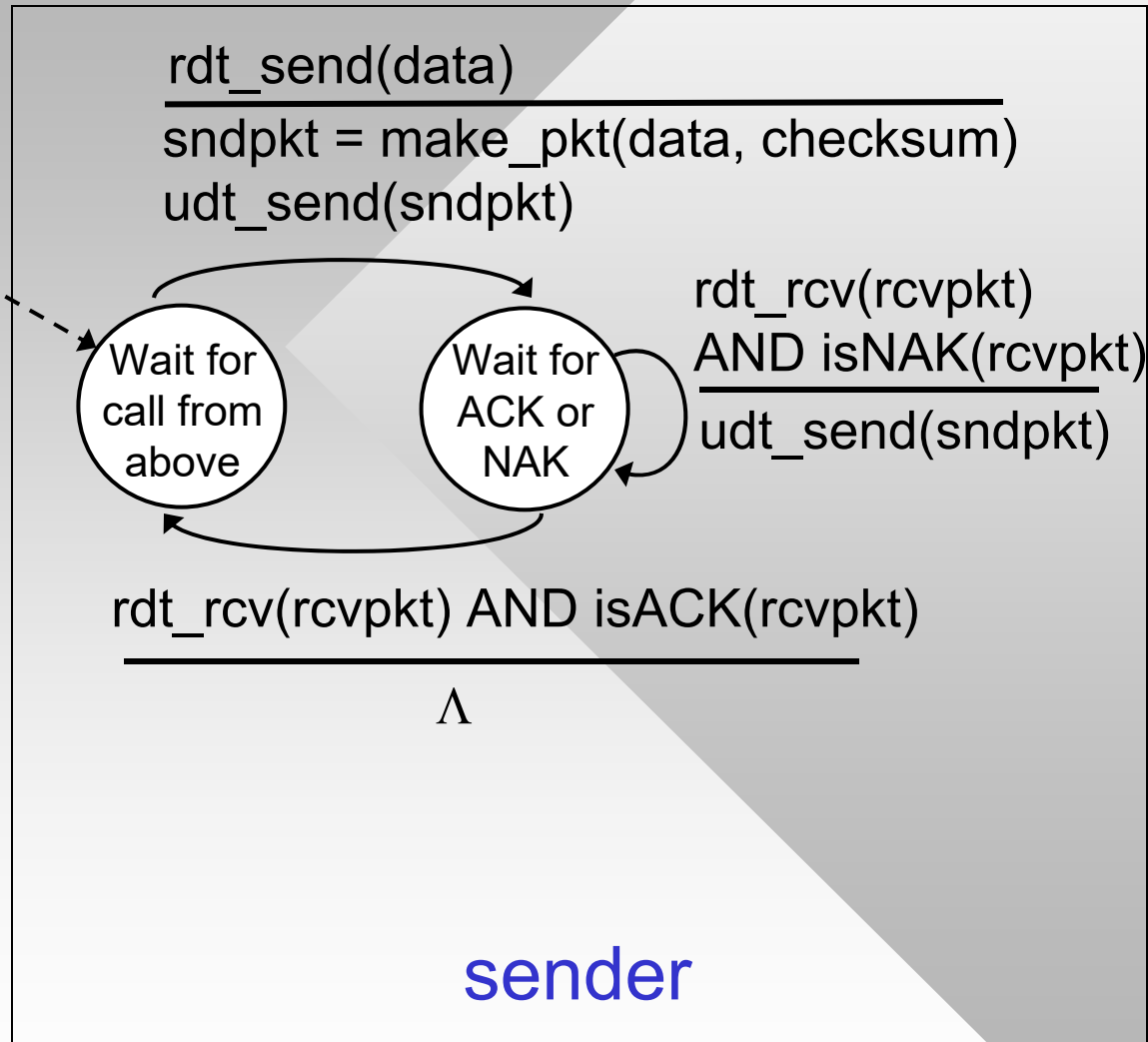
- Underlying channel perfectly reliable
 - No bit errors
 - No loss of packets
 - No reordering
- Separate FSMs for sender and receiver:
 - Sender transmits app data into underlying channel
 - Receiver passes data from underlying channel to app



Rdt2.0: Channel With Bit Errors

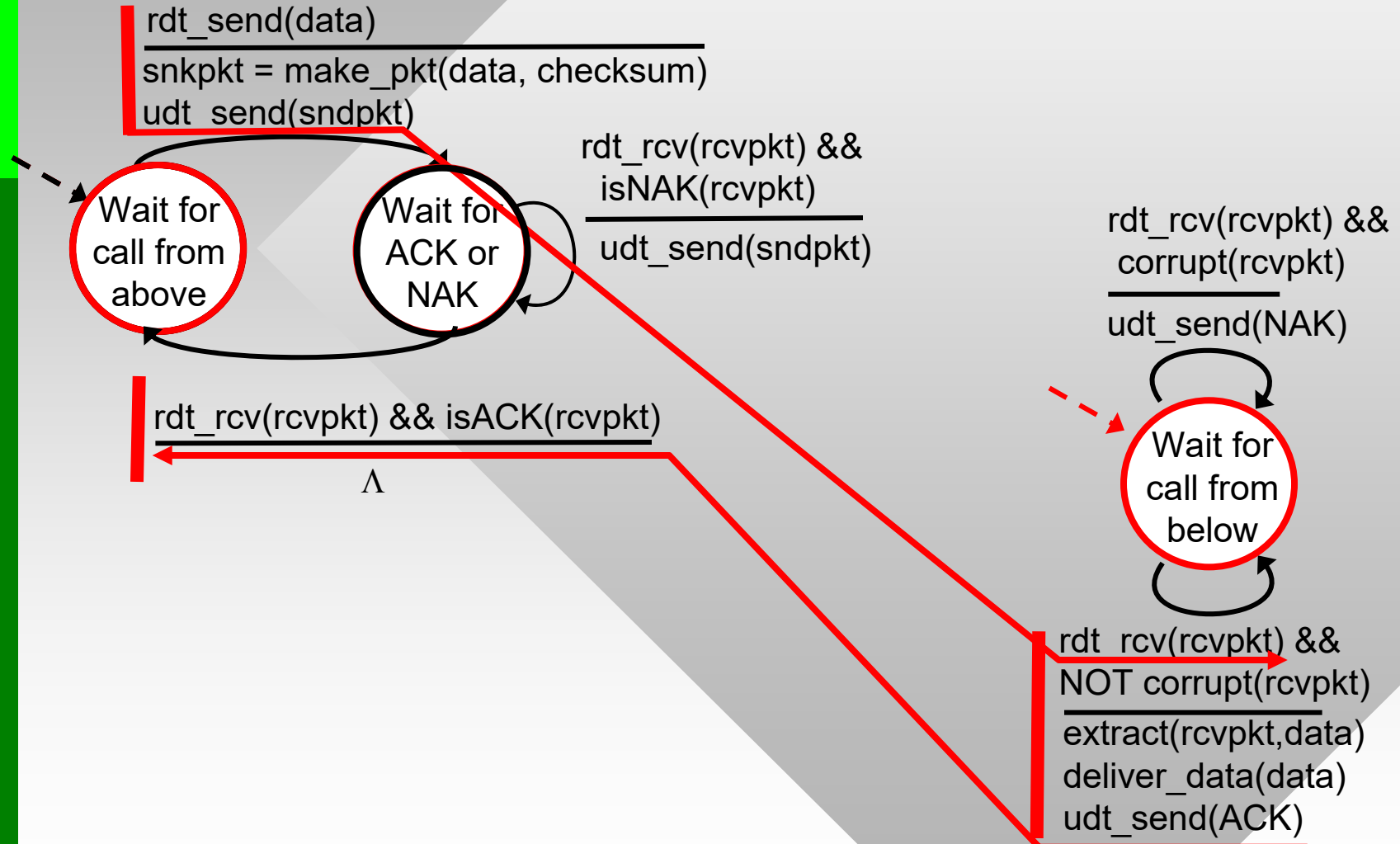
- Underlying channel may flip bits in packet (no loss)
 - Checksum to detect bit errors (assume perfect detection)
- Question: how to recover from errors?
- One possible approach is to use two feedback msgs:
 - *Positive acknowledgments (ACKs)*: receiver explicitly tells sender that packet was received OK
 - *Negative acknowledgments (NAKs)*: receiver explicitly tells sender that packet had errors
 - Sender retransmits packet on receipt of NAK
- New mechanisms in rdt 2.0 (beyond rdt 1.0):
 - Error detection
 - Receiver feedback (control msgs ACK/NAK)
 - Retransmission

Rdt2.0: FSM Specification

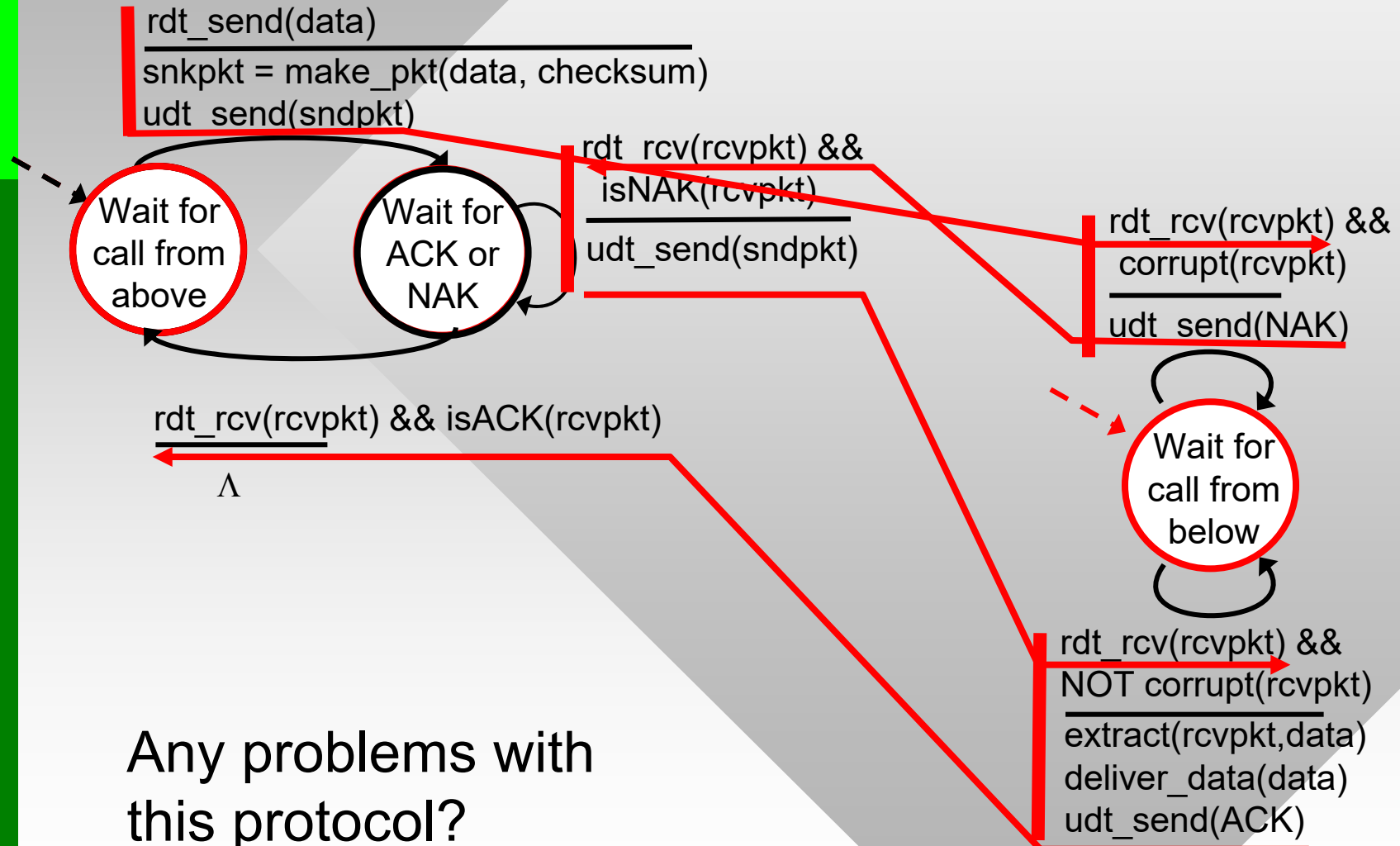


Λ = empty action, i.e., do nothing

Rdt2.0: Operation With No Errors

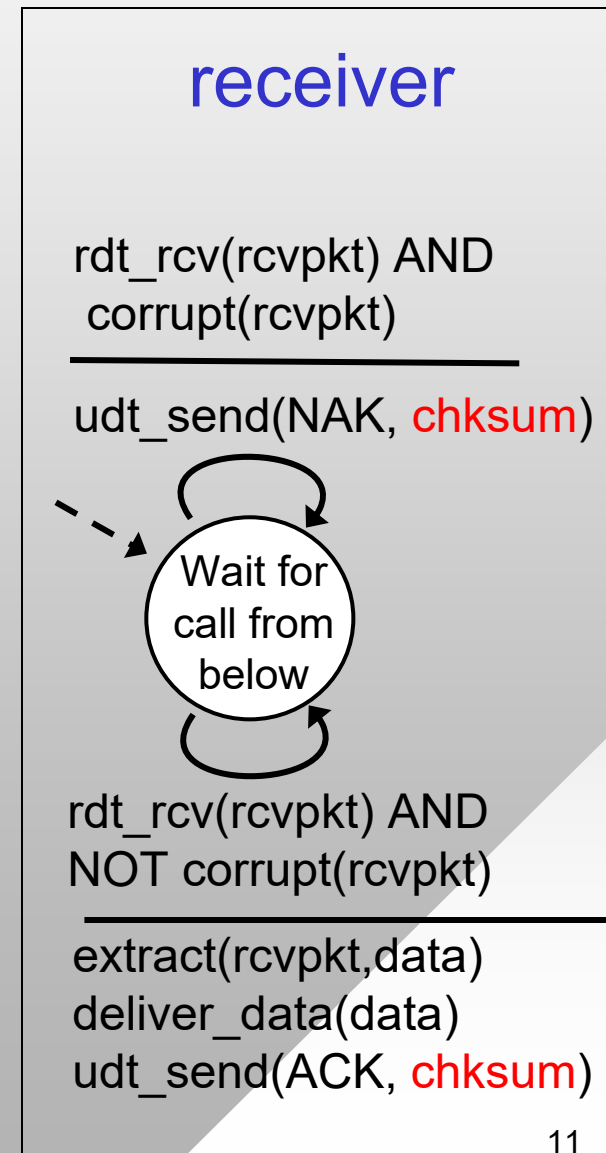
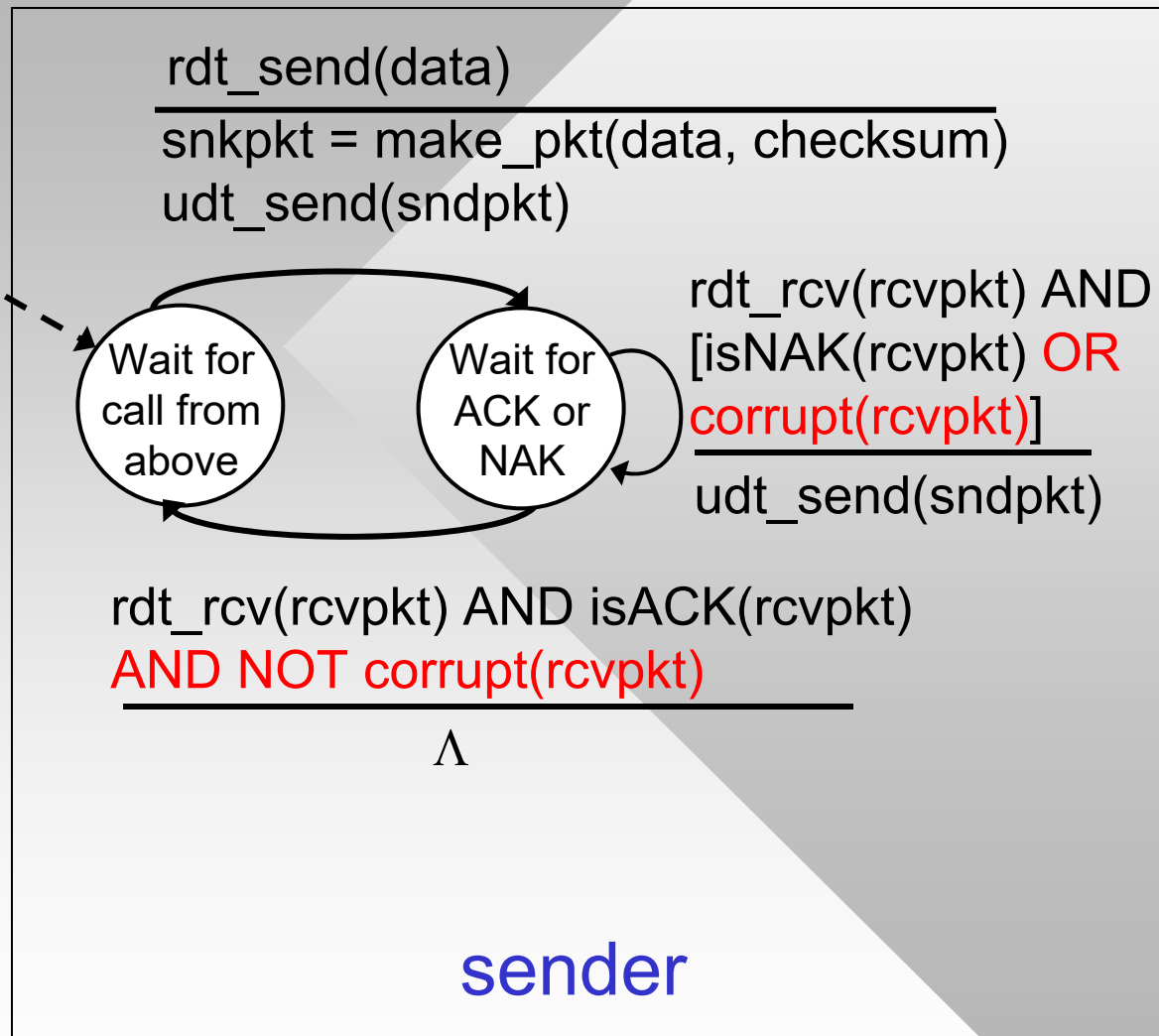


Rdt2.0: Error Scenario



Any problems with this protocol?

Rdt2.0a: Handles Corrupted Feedback



Any problems?

Rdt2.0 and Rdt2.0a Have Fatal Flaws

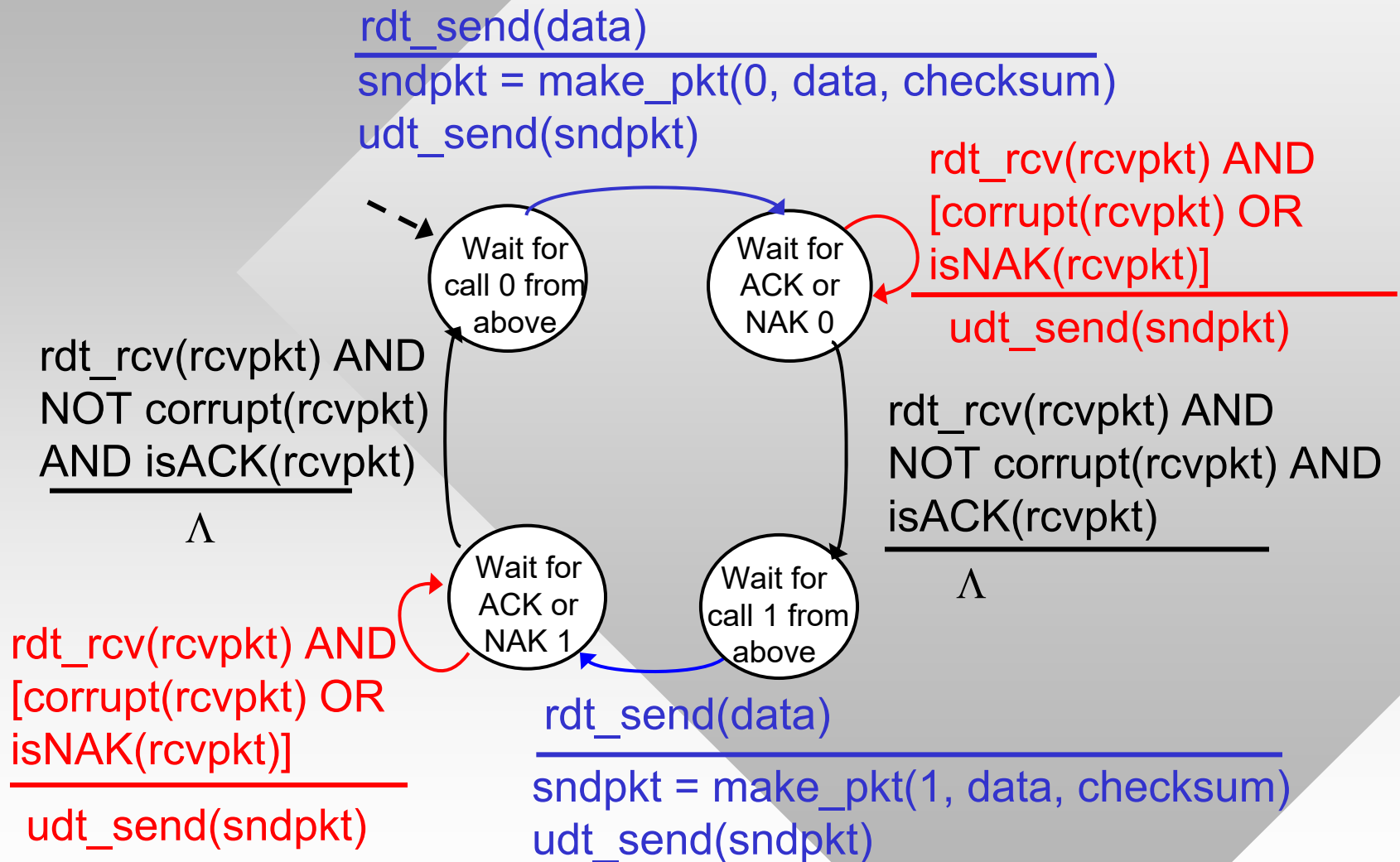
- Rdt 2.0 does not work when ACK/NAK is corrupted
 - Sender doesn't know what happened at receiver!
- Rdt 2.0a delivers duplicate packets to application

Proper algorithm:

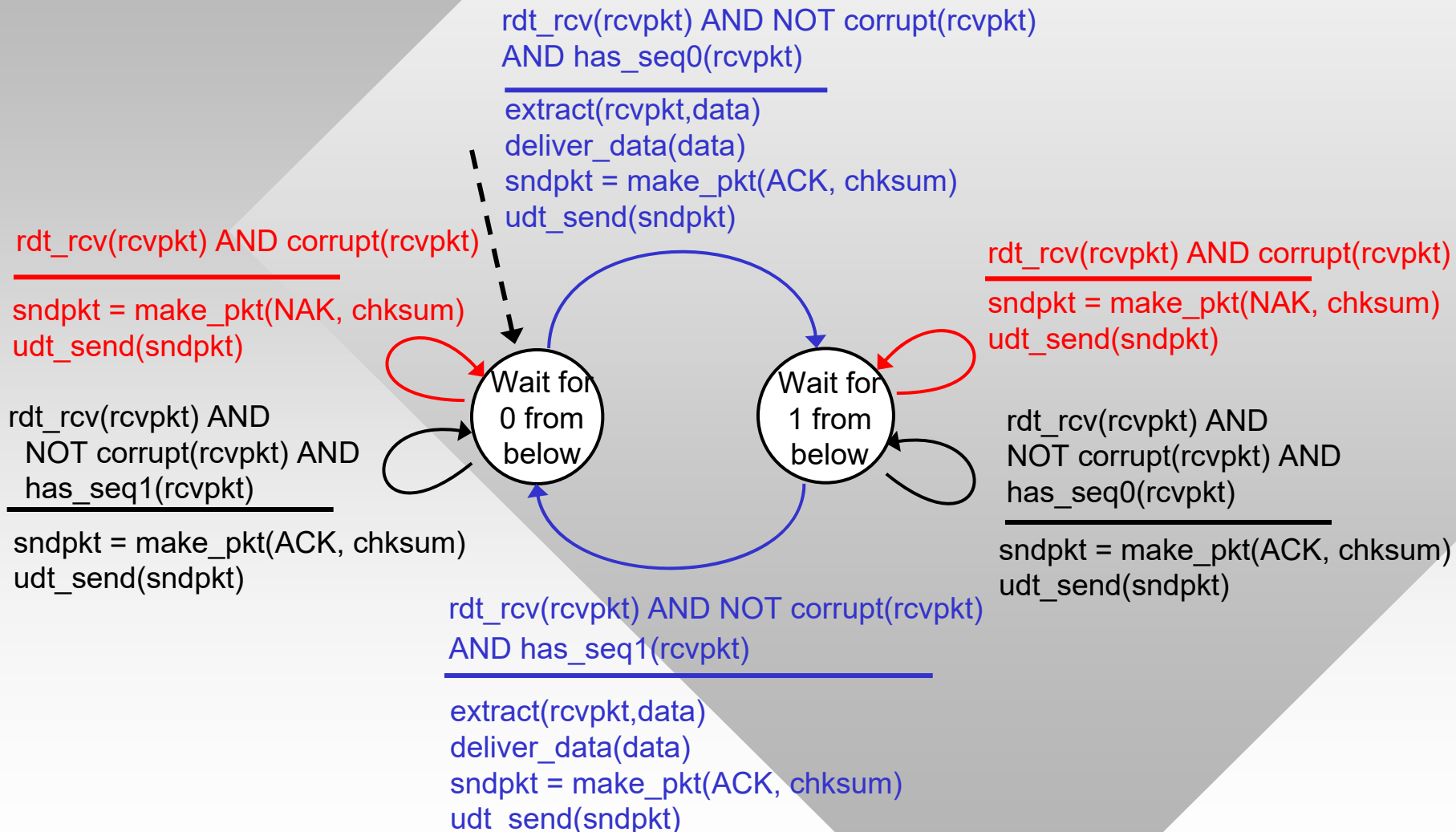
- Sender adds *sequence number* to each pkt
- Sender retransmits current pkt if ACK/NAK is garbled
- Receiver discards (doesn't deliver up) duplicate pkt

Stop-and-Wait protocol: sender sends one packet, then waits for receiver's response

Rdt2.1: Sender, Handles Garbled ACK/NAKs



Rdt2.1: Receiver, Handles Garbled ACK/NAKs



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
 - Protocol must remember whether current pkt has 0 or 1 sequence number

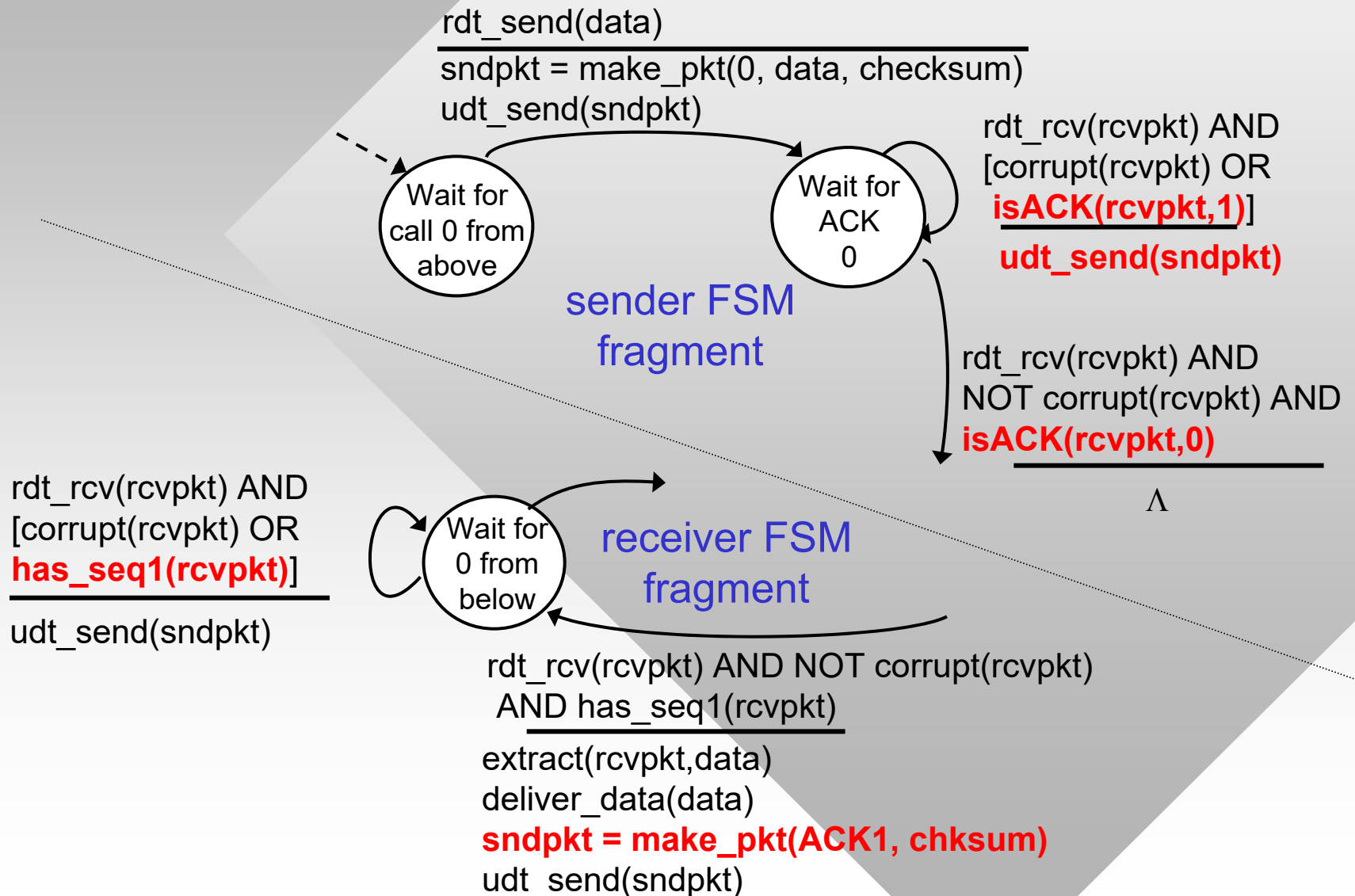
Receiver:

- Must check if received packet is duplicate
 - State indicates whether 0 or 1 is the expected packet seq #
- Note: receiver *cannot* know if its last ACK/NAK was received correctly at sender

Rdt2.2: NAK-Free Protocol

- Same functionality as rdt2.1, using ACKs only
 - Most protocols are easier to generalize without NAKs
- Instead of NAKs, receiver sends an ACK for **last packet received correctly**
 - Receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK received at sender results in same action as NAK: *retransmit current pkt*

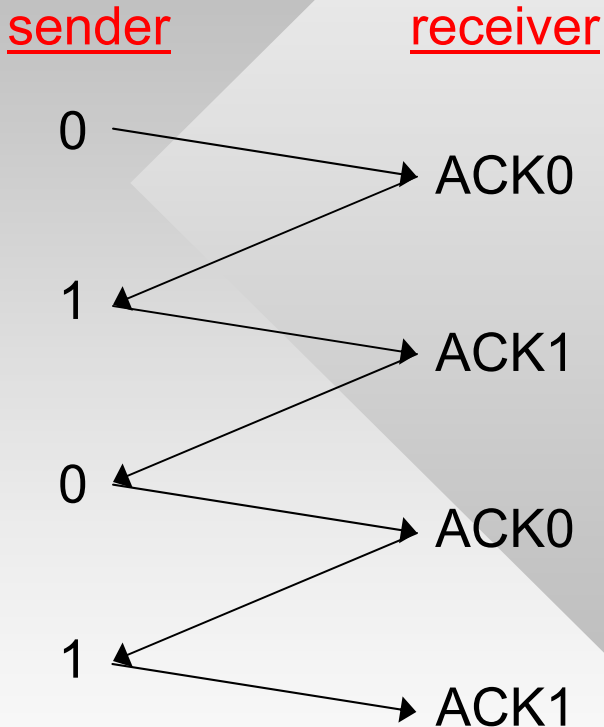
Rdt2.2: Sender, Receiver Fragments



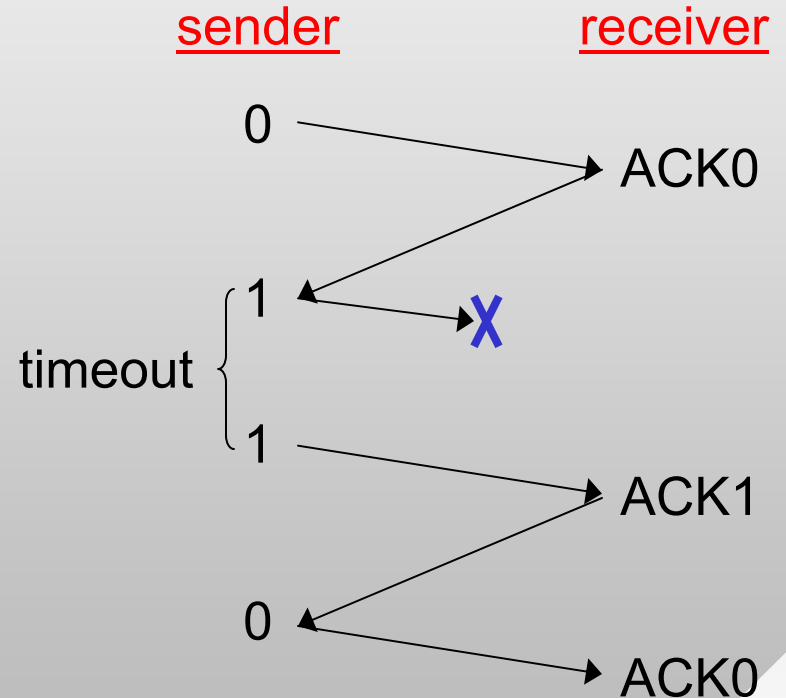
Rdt3.0: Channels With Errors *and* Loss

- New assumption: underlying channel can also lose data packets or ACKs
 - Still no reordering
- Checksum, sequence numbers, ACKs, retransmissions will be of help, but **not enough**
 - Why not?
- Approach: sender waits a “reasonable” amount of time for ACK
 - Retransmits if no ACK received in this time
 - Sender requires a timer
- Redundant retransmission is now possible
 - ACK is lost or arrives after the timer expires
 - The use of seq. #'s in data pkts and ACKs already handles this

Rdt3.0 in Action (No Corruption)

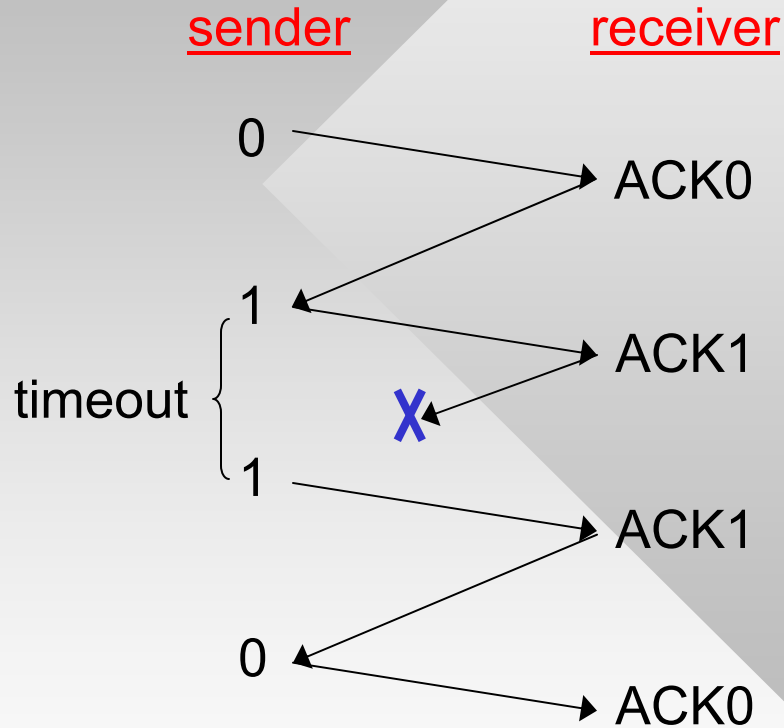


no loss

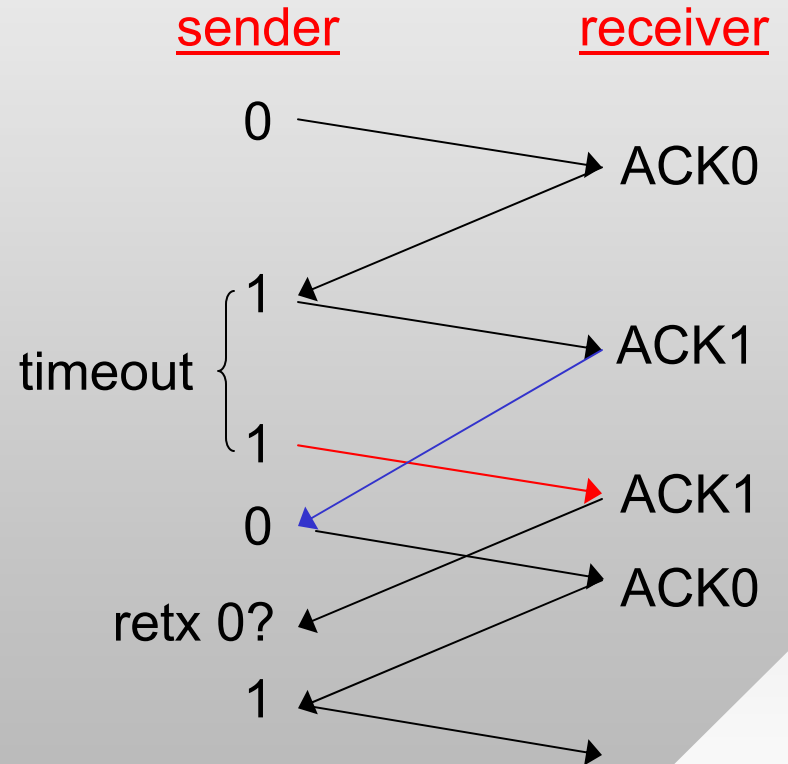


forward loss

Rdt3.0 in Action (No Corruption)



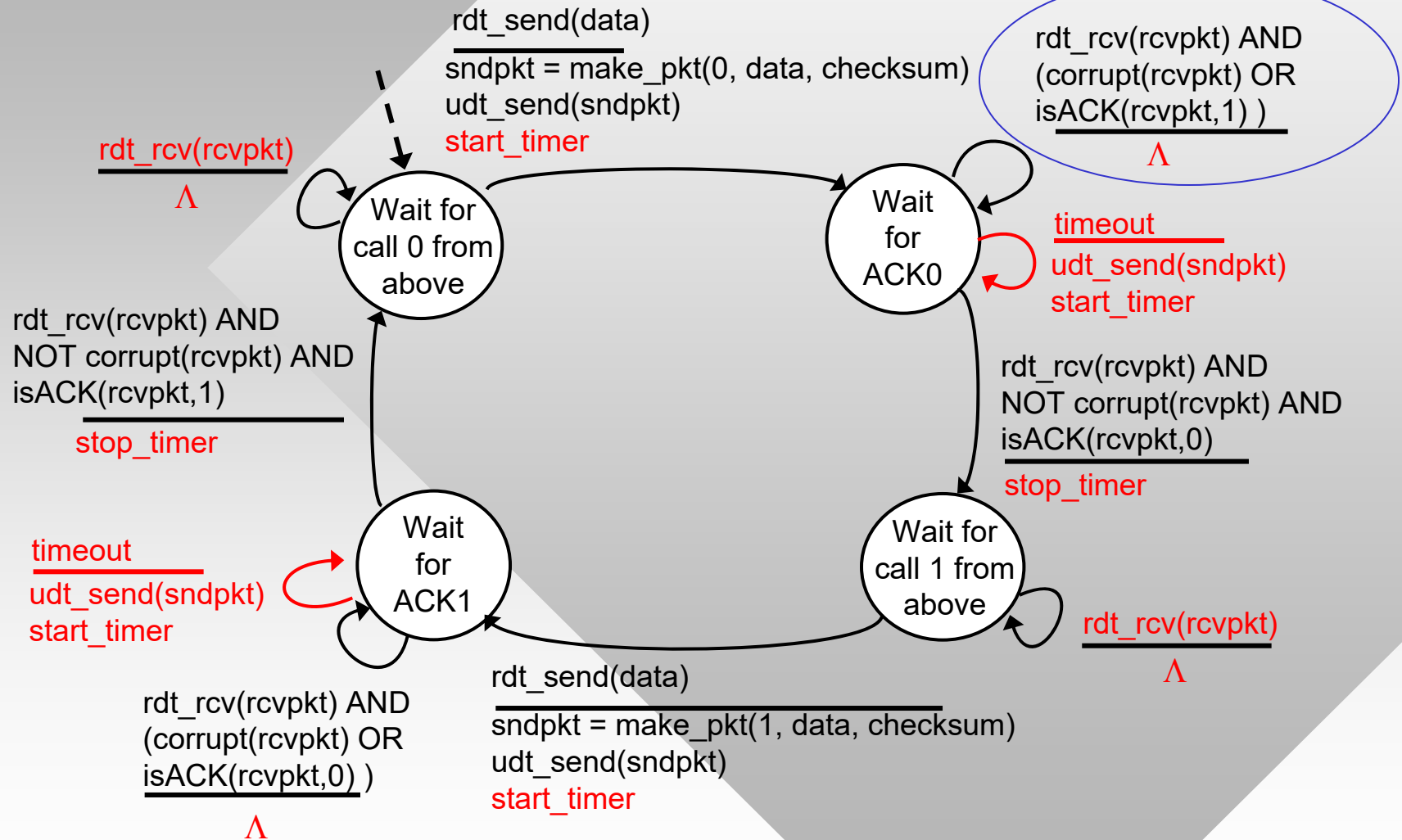
reverse loss



premature timeout

Rdt3.0 Sender

Must not retx: ACK1 may be from a premature timeout on pkt1



Performance of Rdt3.0

Notation: KB = Kilobyte;
Kbps = Kilobits/sec
Gbps = Gigabits/sec

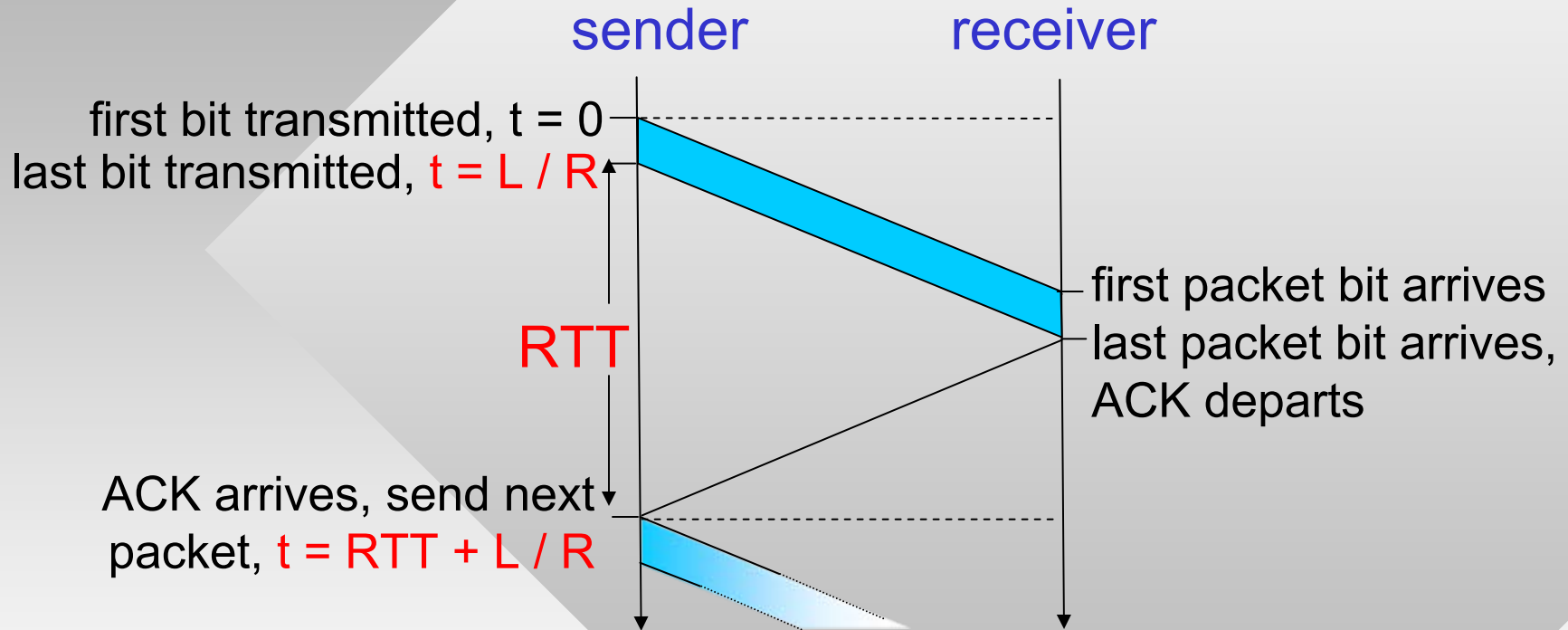
- Rdt 3.0 works, but performance is low
- Example: 1 Gbps link, 15 ms end-to-end propagation delay, 1 KB packets, no loss or corruption:

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8 \text{ Kbits/pkt}}{10^9 \text{ bits/sec}} = 8 \text{ microsec}$$

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

- Server spends 0.008 ms being busy and 30 ms being idle, thus its link utilization is **only 0.027%**
- 1-KB pkt every 30 ms → 264 Kbps throughput
- *Network protocol limits use of physical resources!*

Rdt3.0: Stop-and-Wait Operation



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

Performance of Rdt3.0

- Next assume that 10% of data packets are corrupted/lost (no loss in retransmissions or ACKs) and the timeout is 1 second
 - 90% of packets take $(RTT + L/R) \approx 30$ ms to complete, while 10% require $[\text{timeout} + RTT + 2L/R] \approx 1.03$ sec
 - Average per-packet delay $0.9 \cdot 0.03 + 0.1 \cdot 1.03$ sec = 130 ms
 - Average rate 7.7 pkts/s or 61.5 Kbps
- Rdt3.0 similar to HTTP 1.0 or non-pipelined HTTP 1.1
- Next time we'll improve this using **pipelining**, which allows multiple unack'ed packets at any time
- Quiz #2: chapter 2 problems and **system notes**
 - P1, P3-P11, P13-P14, P20-P21