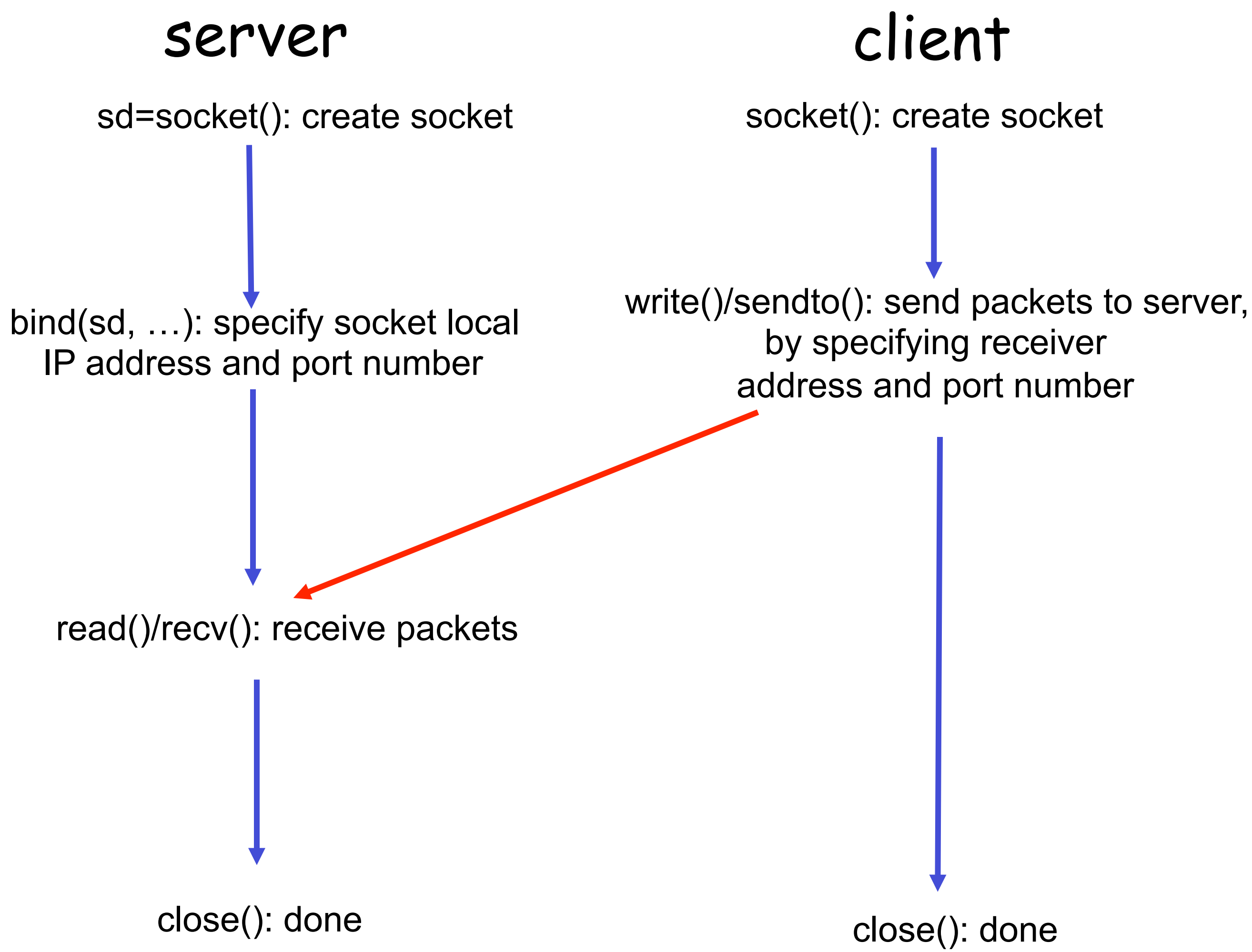
Transport Overview

KR 3.1-3.4

Outline

- Recap
- Overview of transport layer
- UDP
- Reliable data transfer, the stop-and-wait protocol

Connectionless UDP: Big Picture



Connection-oriented: Big Picture

server

sd=socket(): create socket

bind(sd, …): specify socket address

listen(sd, …): specify that socket sd is a listening socket

TCP connection setup

sd2=accept(sd, …):
get a connected connection
from the queue for socket sd;
create a new socket identified by sd2

read()/write(): do IO on socket sd2

close(sd2): done

client

socket(): create socket

bind(): specify socket address *optional*

connect(): initialize TCP handshake;
return until TCP handshake is done

read()/write(): do IO on the socket

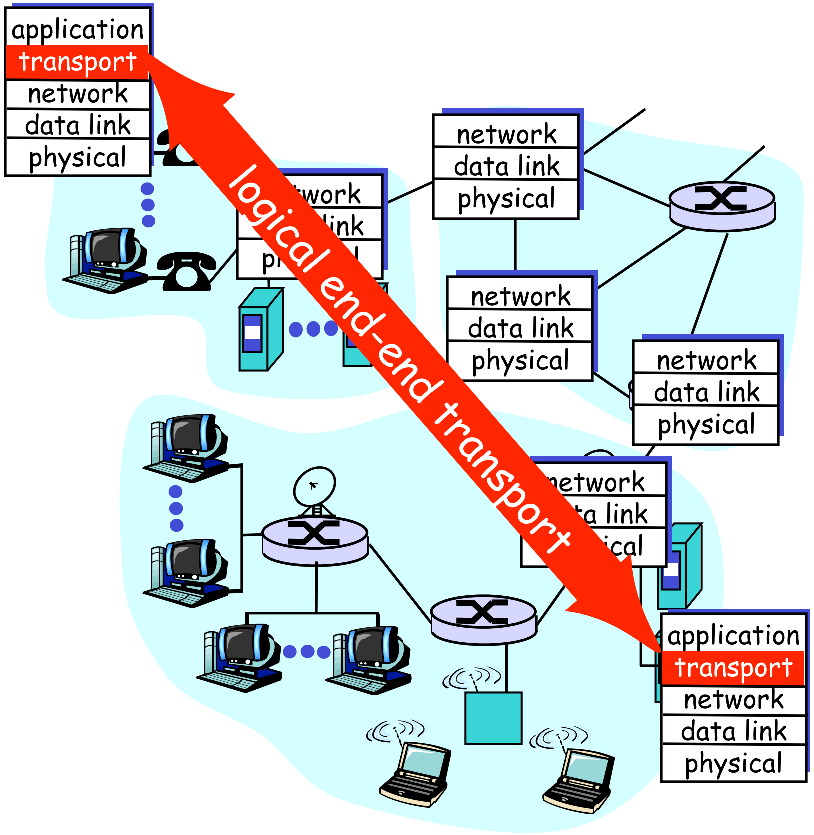
close(): done

Outline

- Recap
- Overview of transport layer
- UDP
- Reliable data transfer, the stop-and-go protocols

Transport Layer vs. Network Layer

- Provide *logical communication* between app' processes
- Transport protocols run in end systems
  - Sender: breaks app messages into segments, passes to network layer
  - Receiver: reassembles segments into messages, passes to app layer
- Transport vs. network layer services:
  - *Network layer:* data transfer between end systems
  - *Transport layer:* data transfer between processes
    - Relies on, enhances network layer services



Transport Layer Services and Protocols

- Reliable, in-order delivery (TCP)
  - Multiplexing
  - Reliability and connection setup
  - Congestion control
  - Flow control

- Unreliable, unordered delivery: UDP
  - Multiplexing

- Services not available:
  - Delay guarantees
  - Bandwidth guarantees

Transport Layer: Road Ahead

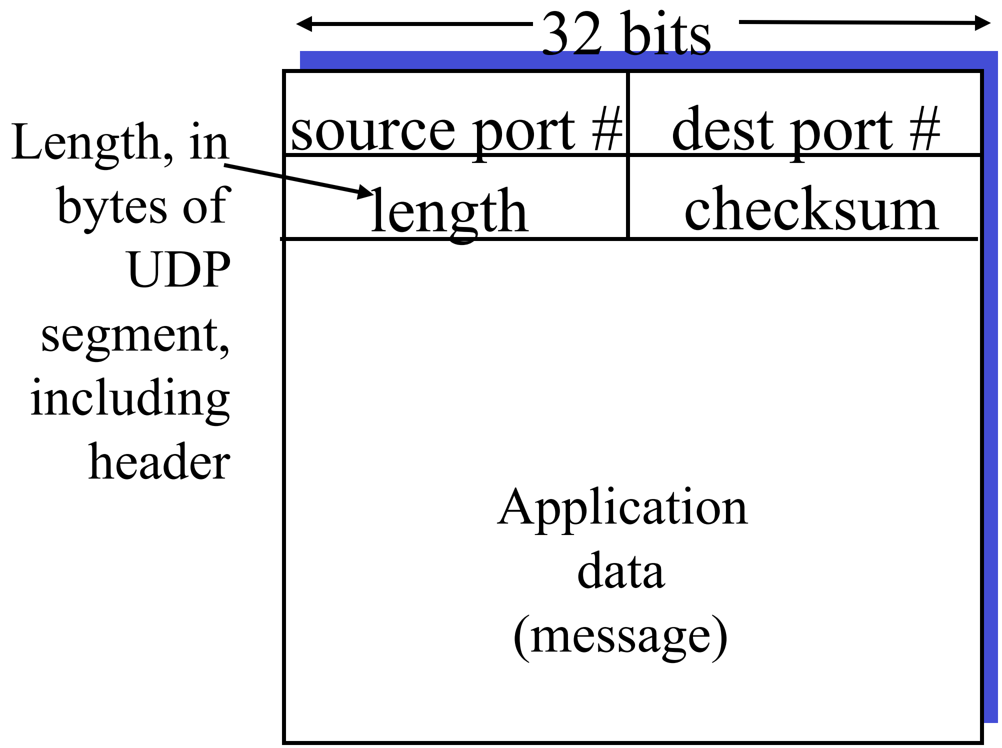
- Class 1:
  - Connectionless transport: UDP
  - Reliable data transfer using stop-and-wait
- Class 2:
  - Sliding window reliability
  - TCP reliability
    - Overview of TCP
    - TCP RTT measurement
    - TCP connection management
- Class 3:
  - Principles of congestion control
  - TCP congestion control; AIMD
- Class 4:
  - Performance modeling; TCP variants
  - The analysis and design framework for congestion control

Outline

- Recap
- Overview of transport layer
- UDP
- Reliable data transfer, the stop-and-go protocols

UDP: User Datagram Protocol [RFC 768]

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP usage
  - DNS
  - SNMP



UDP segment format

UDP Checksum

Goal: end-to-end detection of "errors" (e.g., flipped bits) in transmitted segment

Sender:

- Treat segment contents as sequence of 16-bit integers (represented in one's complement representation)
- Checksum: addition (1's complement sum) of segment contents to be 1111111111111111
- Sender puts checksum value into UDP checksum field

Receiver:

- Compute checksum of received segment
- Compute sum of segment and checksum; check if sum is 1111111111111111
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?*

One's Complement Arithmetic

- UDP checksum is based on one's complement arithmetic
  - One's complement was a common representation of signed numbers in early computers
- One's complement representation
  - Bit-wise NOT for negative numbers
  - Example: assume 8 bits
    - `00000000: 0`
    - `00000001: 1`
    - `01111111: 127`
    - `10000000: ?`
    - `11111111: ?`
  - Addition: conventional binary addition except adding any resulting carry back into the resulting sum
    - Example: -1 + 2

UDP Checksum: Algorithm

- Example checksum:

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

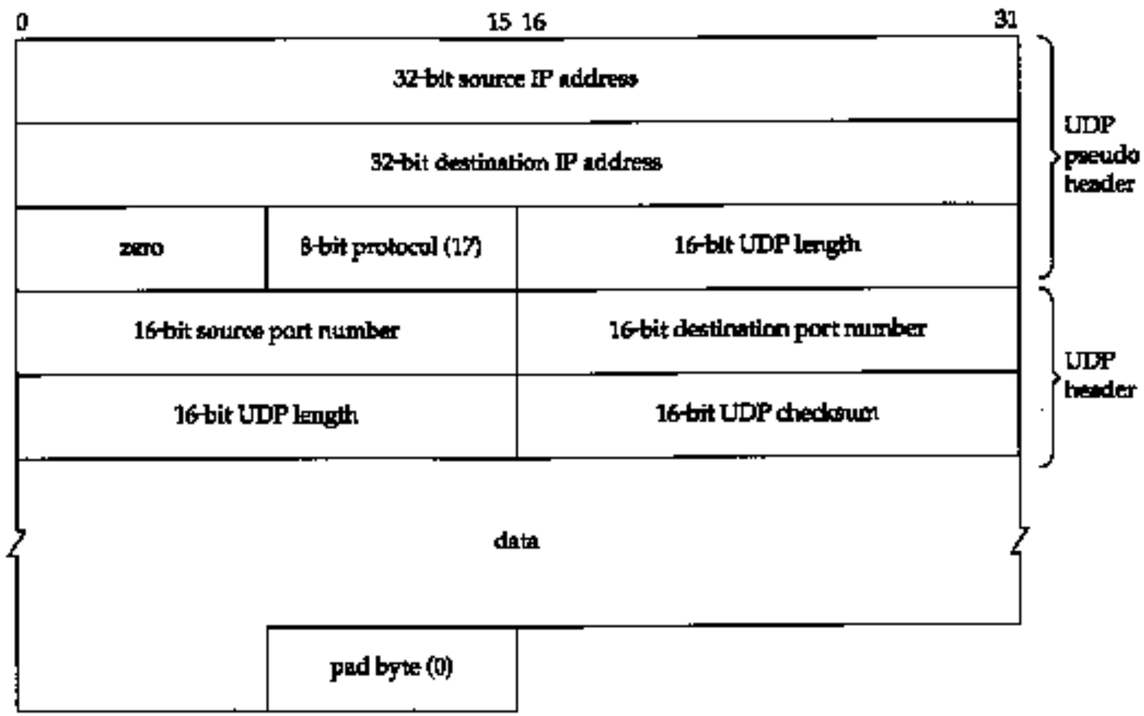
-What are some desirable properties of using one's complement representation?

- For fast implementation of computing UDP checksum, see http://www.faqs.org/rfcs/rfc1071.html

UDP Checksum: Coverage

Calculated over:

- A pseudo-header
  - IP Source Address (4 bytes)
  - IP Destination Address (4 bytes)
  - Protocol (2 bytes)
  - UDP Length (2 bytes)
- UDP header
- UDP data

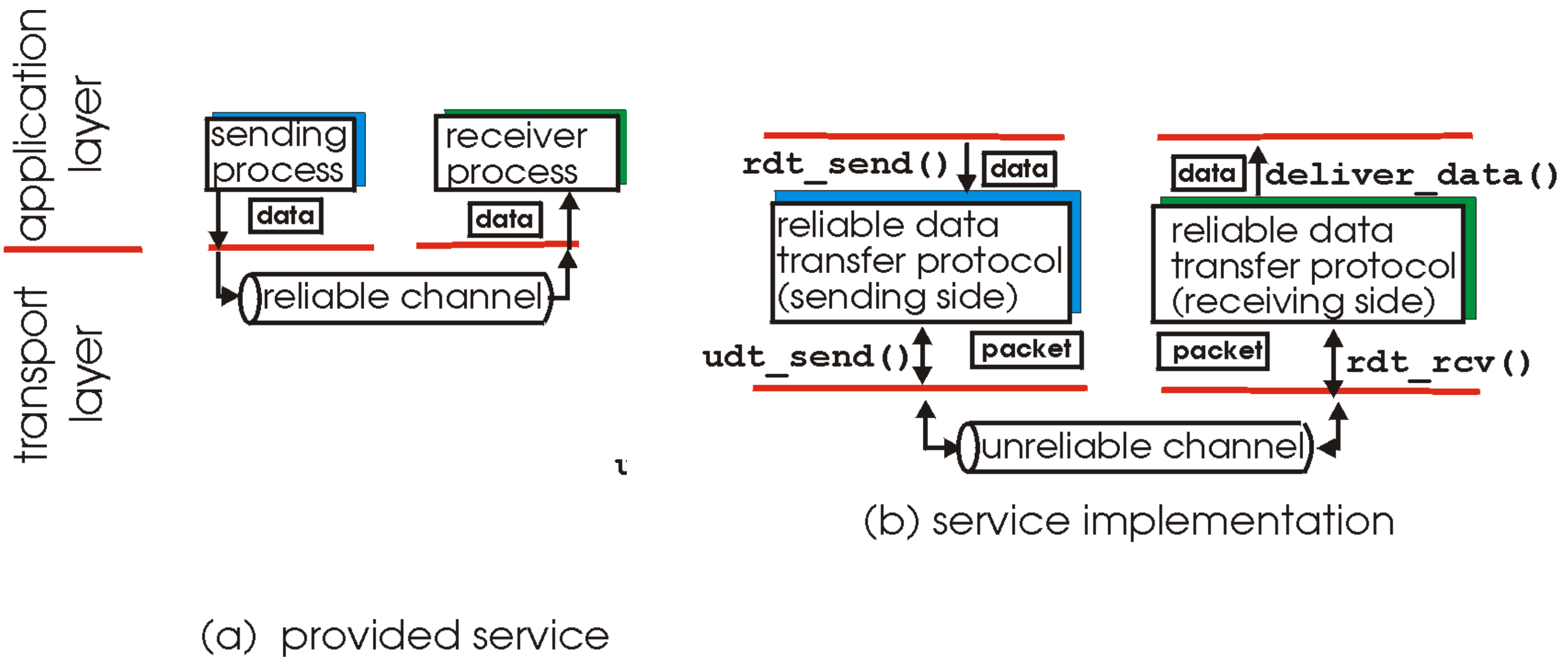


Outline

- Recap
- Overview of transport layer
- UDP
- Reliable data transfer

Principles of Reliable Data Transfer

- Important in app., transport, link layers
- Top-10 list of important networking topics!

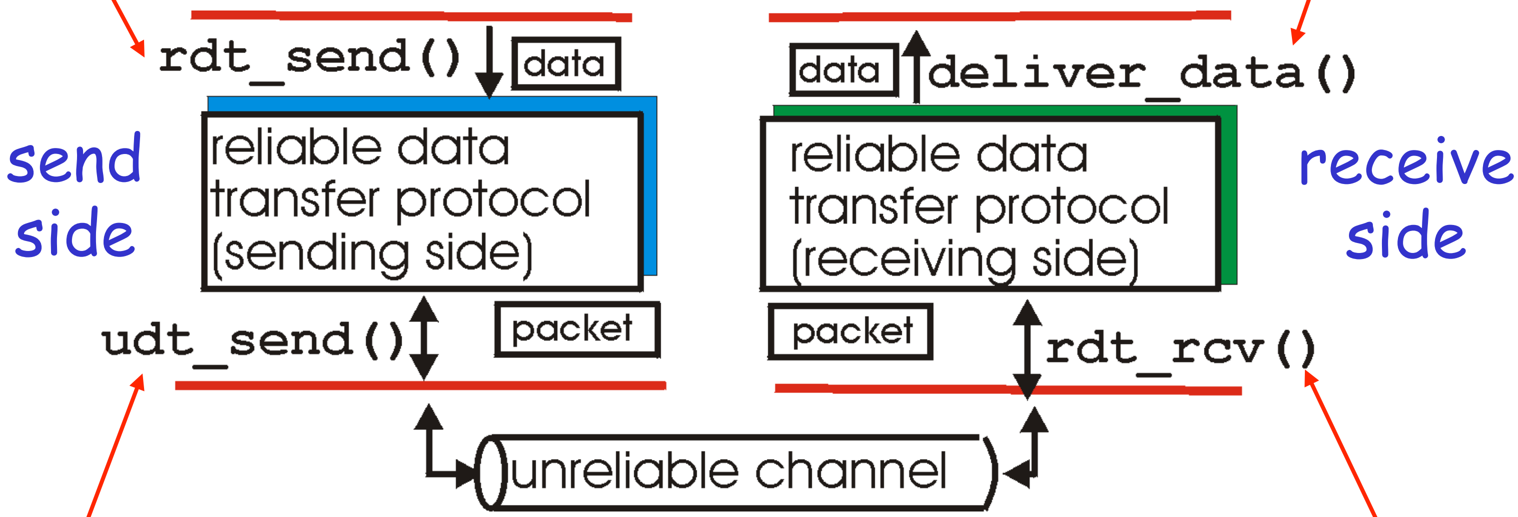


Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).

Reliable Data Transfer: Getting Started

rdt_send() : called from above, (e.g., by app.)

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



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

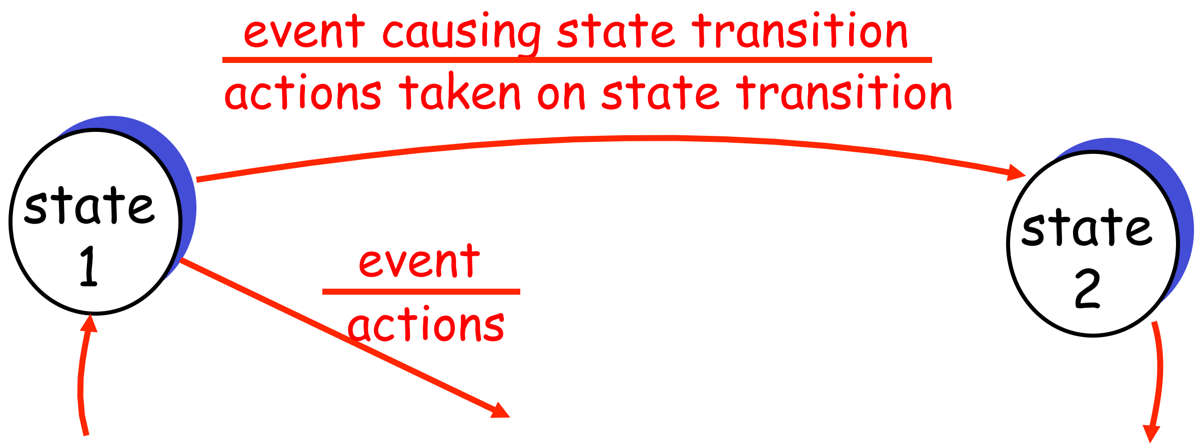
rdt_rcv() : called from below; when packet arrives on rcv-side of channel

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

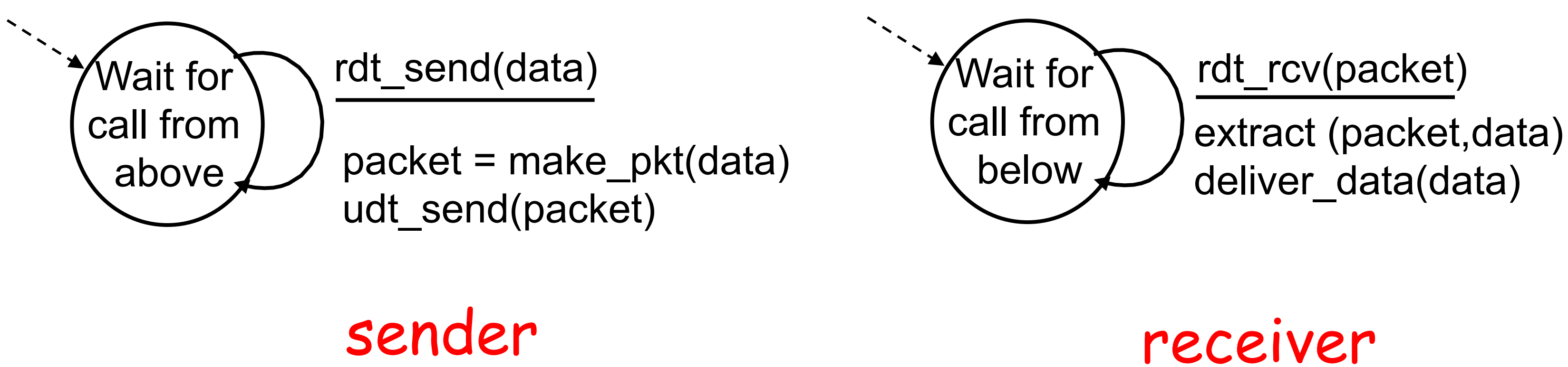


Outline

- Review
- Overview of transport layer
- UDP
- Reliable data transfer
  - Perfect channel

Rdt1.0: Reliable Transfer over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - No reordering or duplication
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



Outline

- Recap
- Overview of transport layer
- UDP
- Reliable data transfer
  - Perfect channel
  - Channel with bit errors

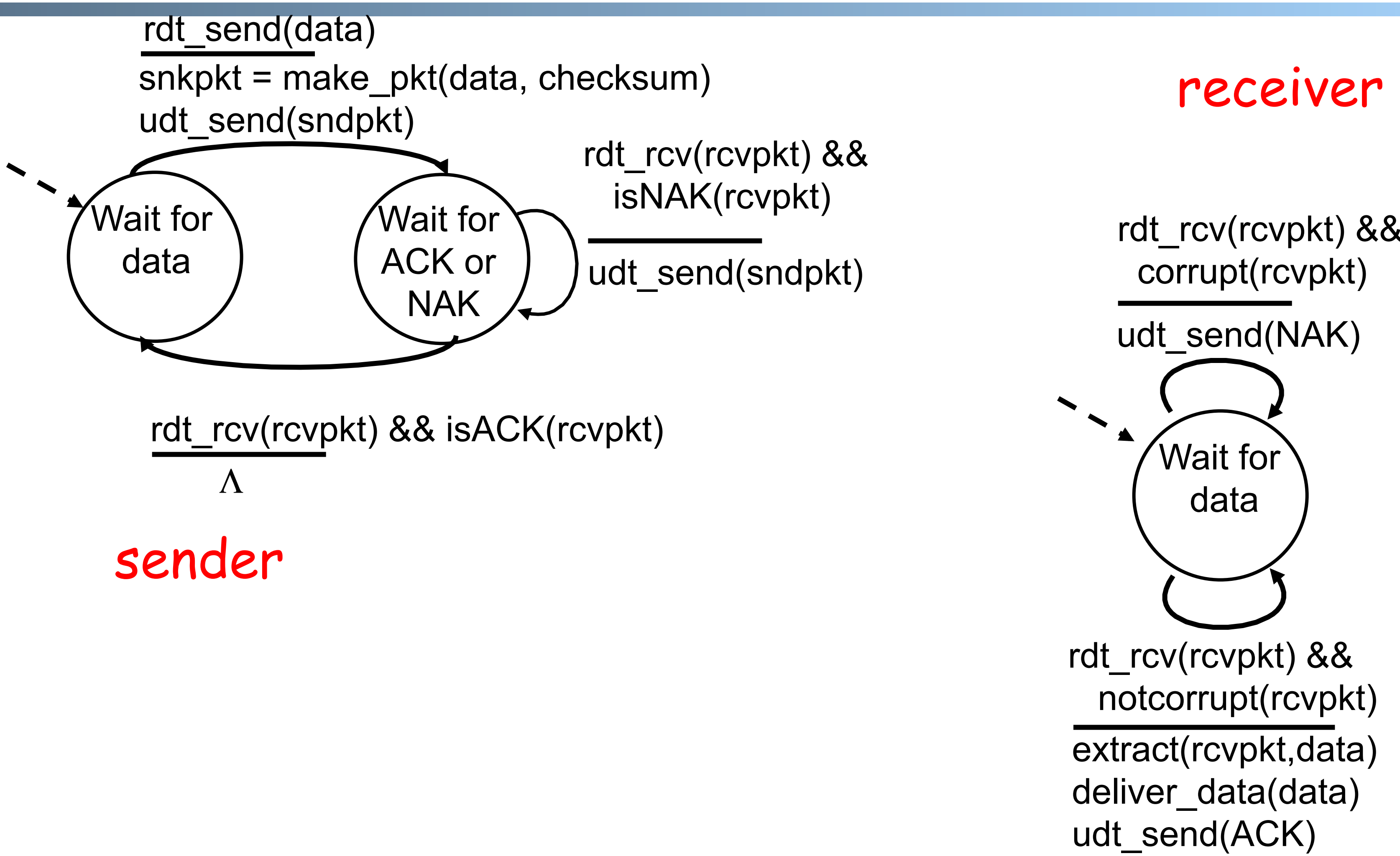
Rdt2.0: Channel With Bit Errors

- Underlying channel may flip bits in packet
  - No loss, duplication or reordering (reasonable?)

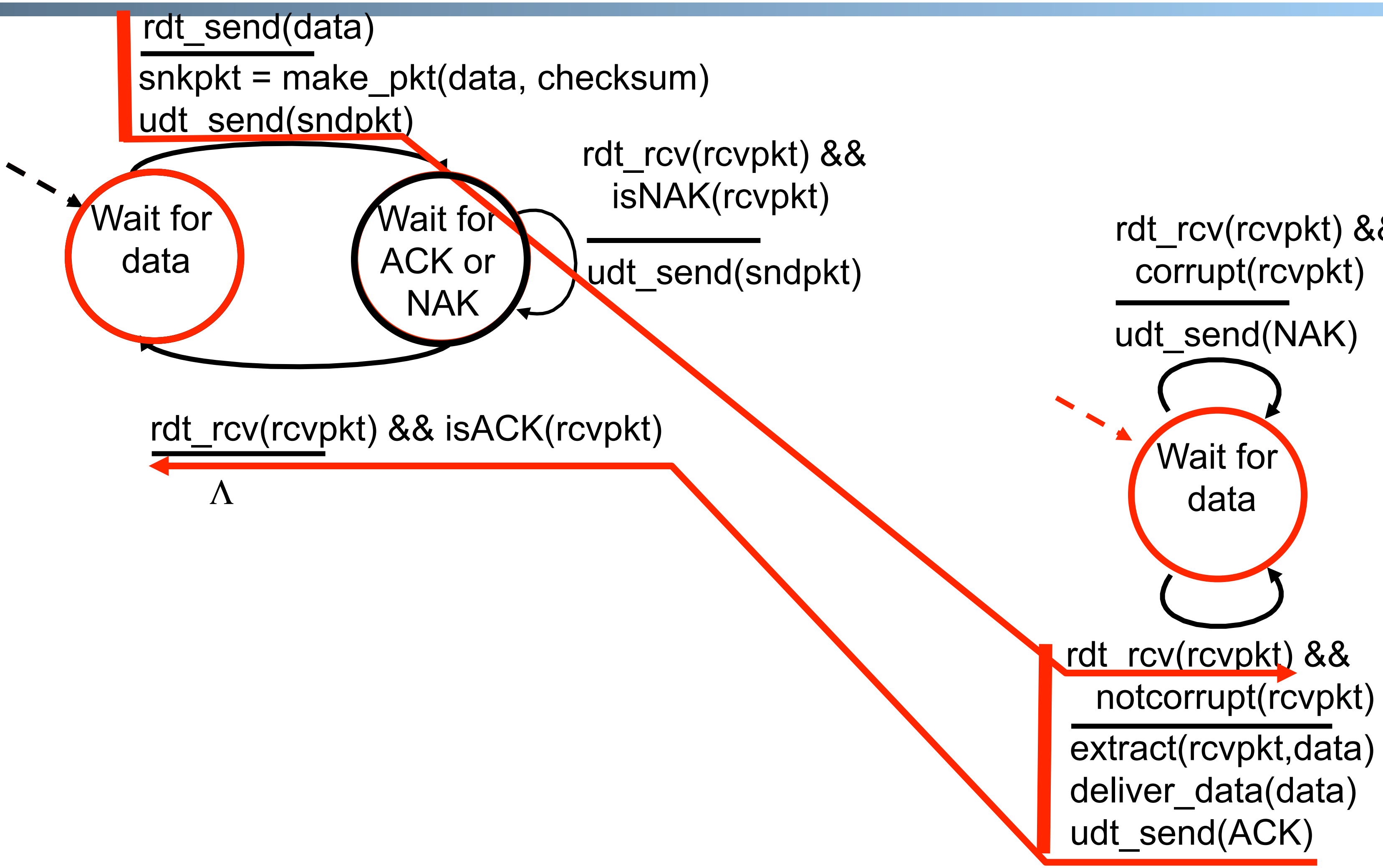
- *The* question: how to recover from data pkt errors:
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
    - Sender retransmits pkt on receipt of NAK

- New mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - Error detection: recall: UDP checksum to detect bit errors
  - Receiver feedback: control msgs (ACK,NAK) rcvr->sender

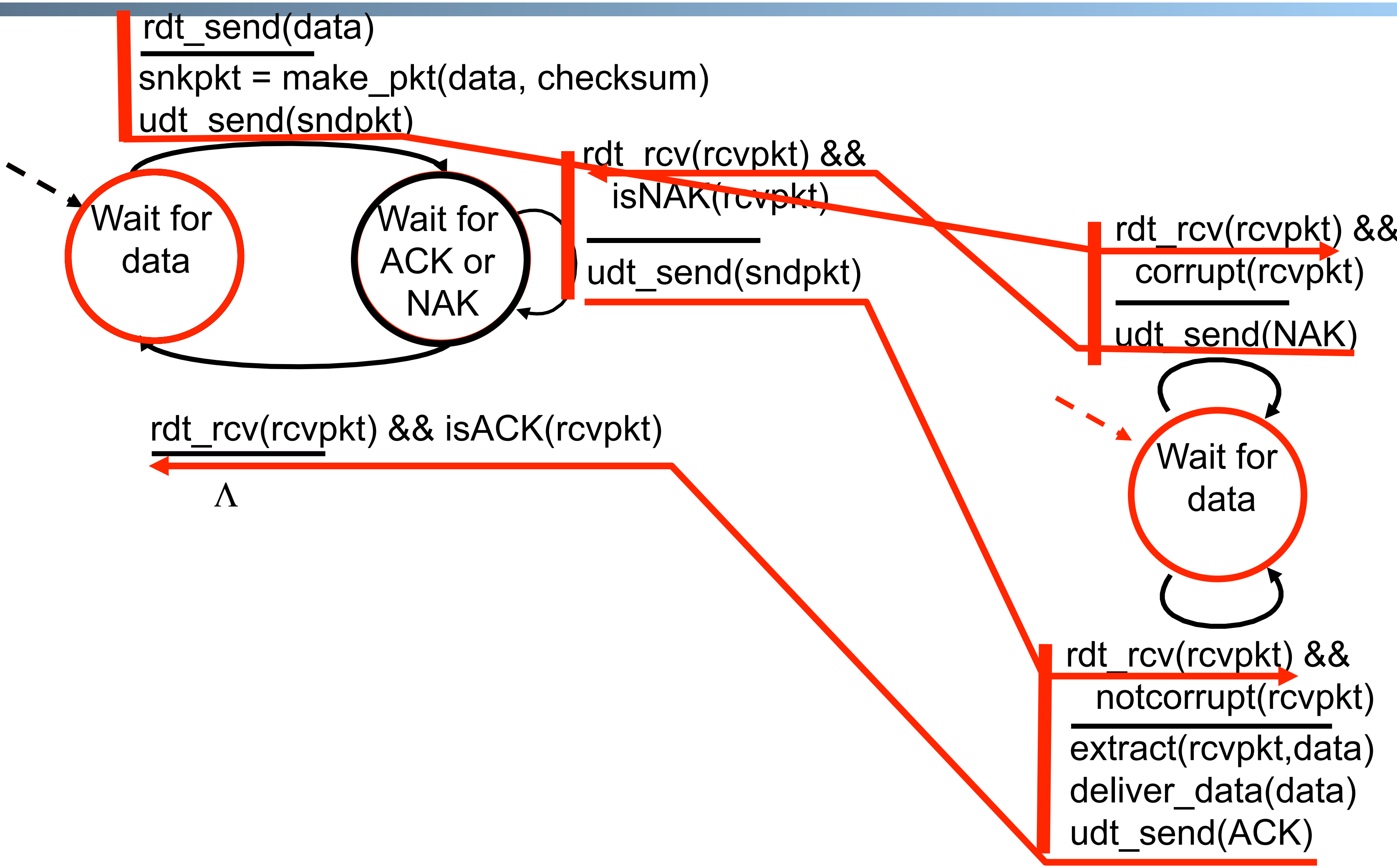
rdt2.0: FSM Specification



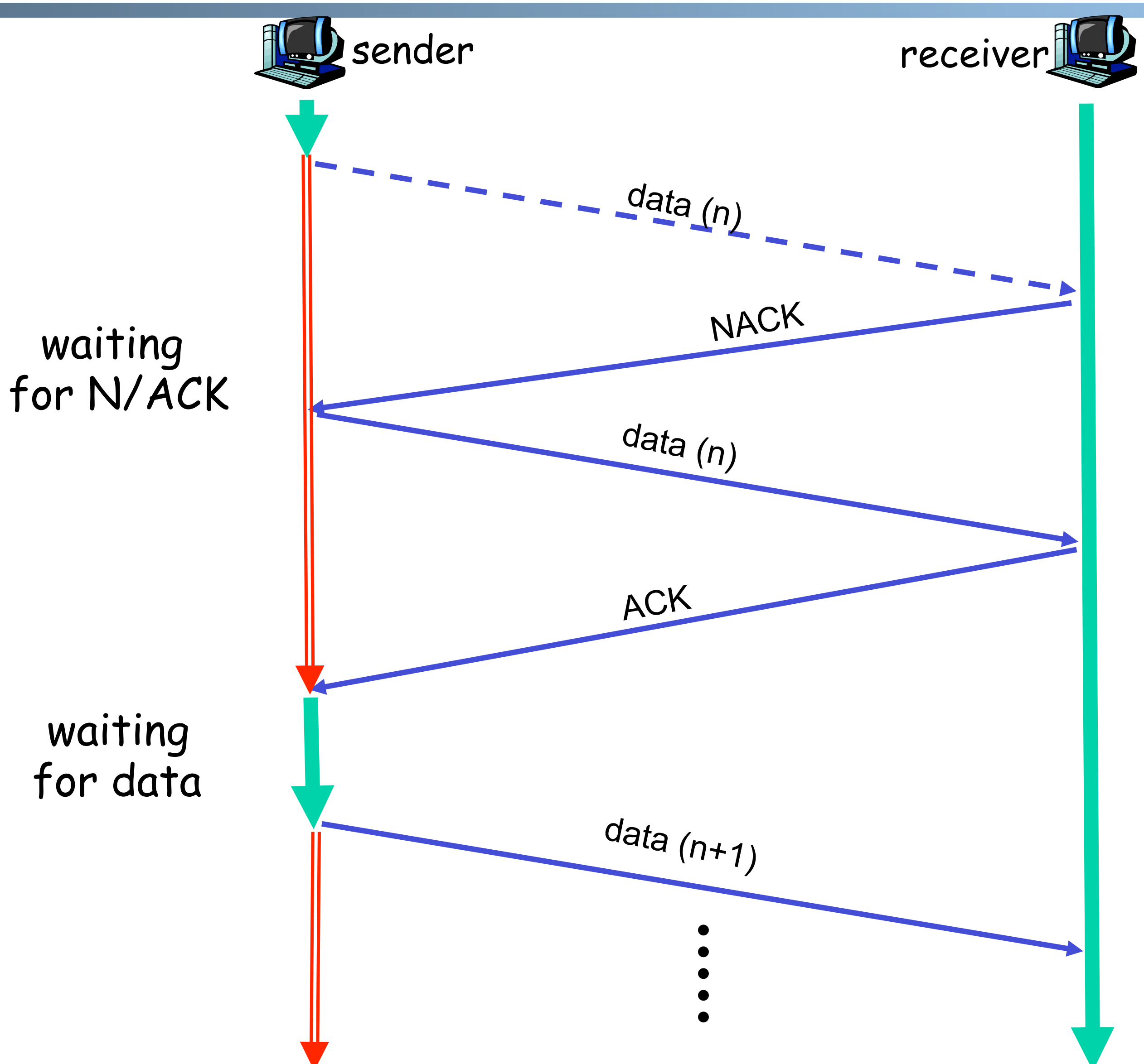
rdt2.0: Operation with No Errors



rdt2.0: Error Scenario



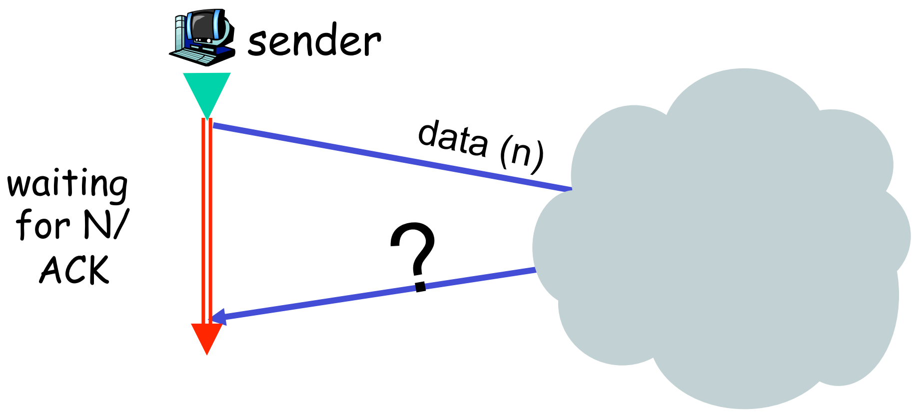
Big Picture of rdt2.0



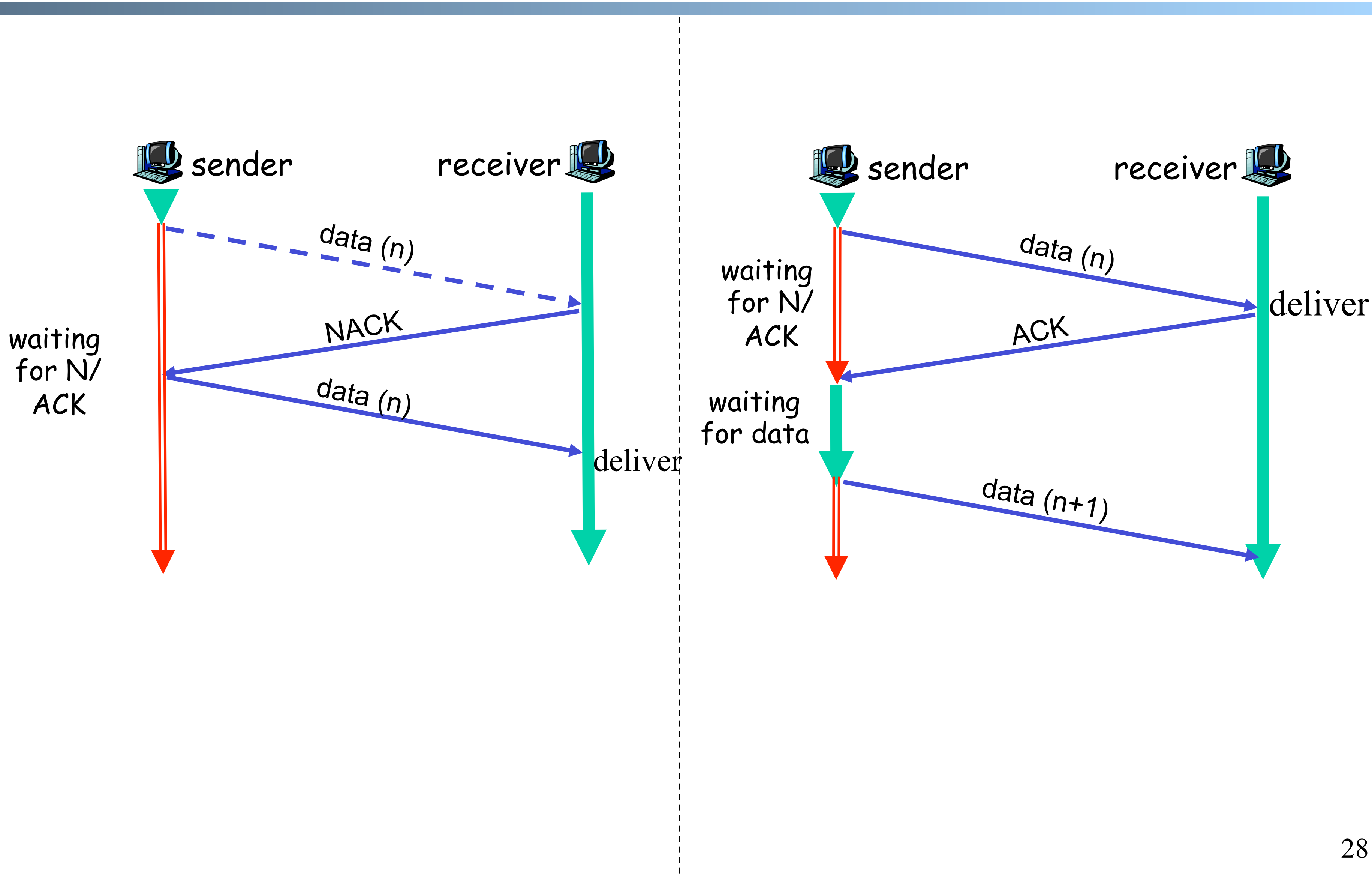
rdt2.0 is Incomplete!

What happens if ACK/NAK corrupted?

- Although sender receives feedback, but doesn't know what happened at receiver!



Two Possibilities



Handle Control Message Corruption

It is always harder to deal with control message errors than data message errors

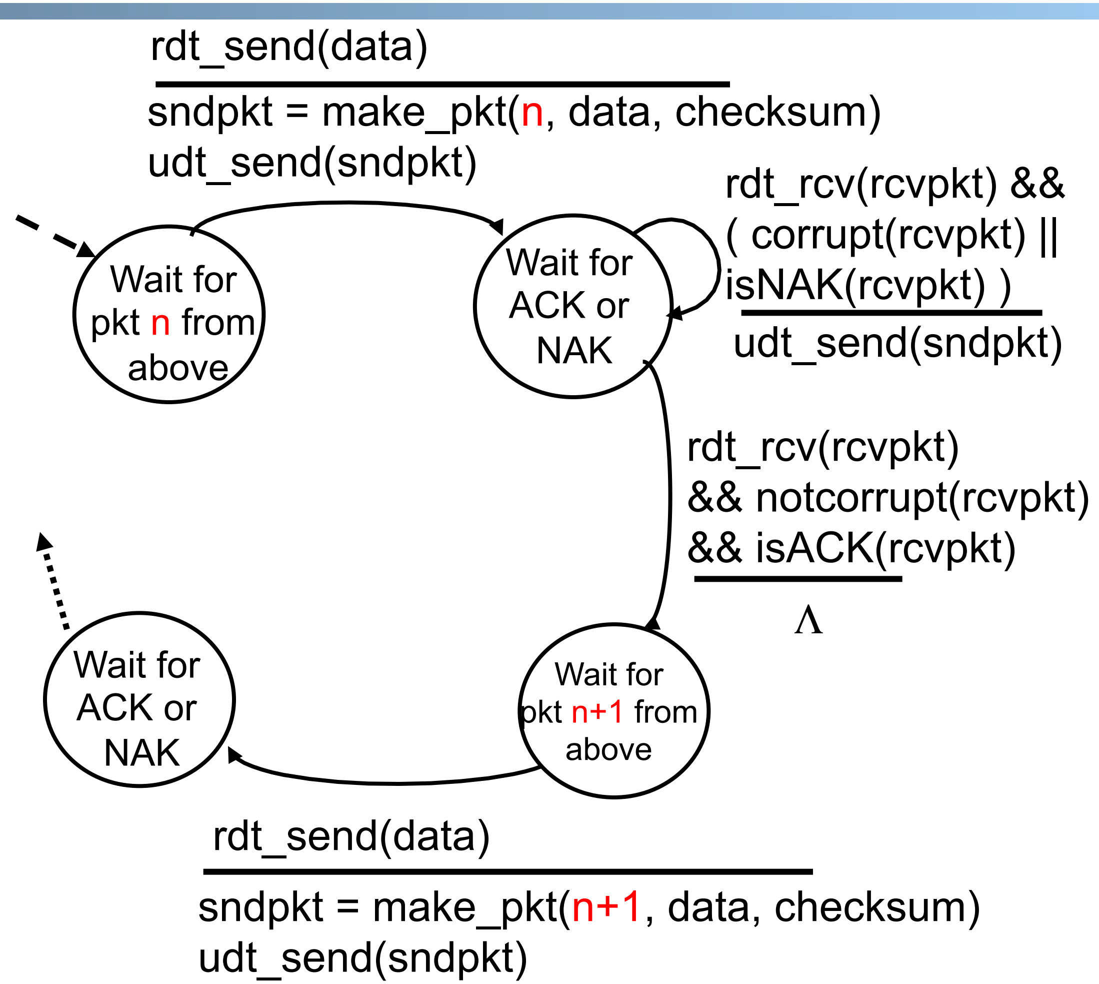
- Sender can't just retransmit: possible duplicate

Handling duplicates:

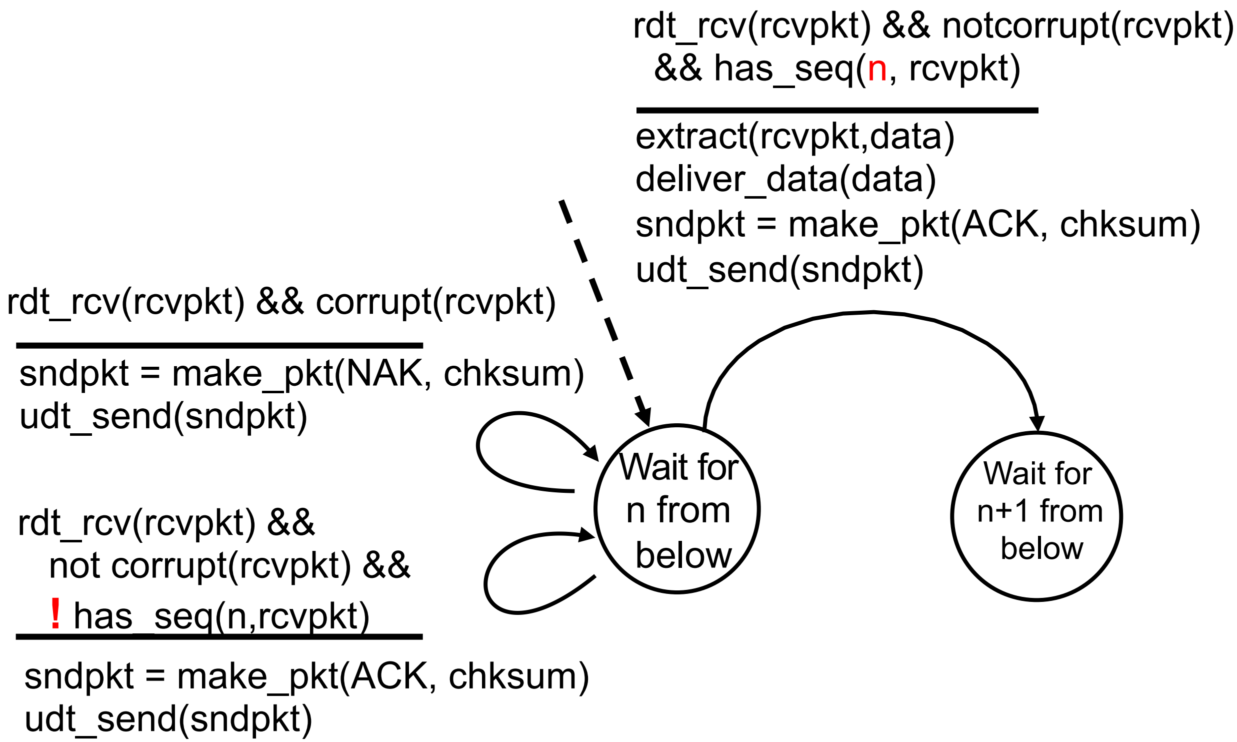
- Sender adds *sequence number* to each pkt
- Sender retransmits current pkt if ACK/NAK garbled
- Receiver discards (doesn't deliver up) duplicate pkt

stop and wait
sender sends one packet, then waits for receiver response

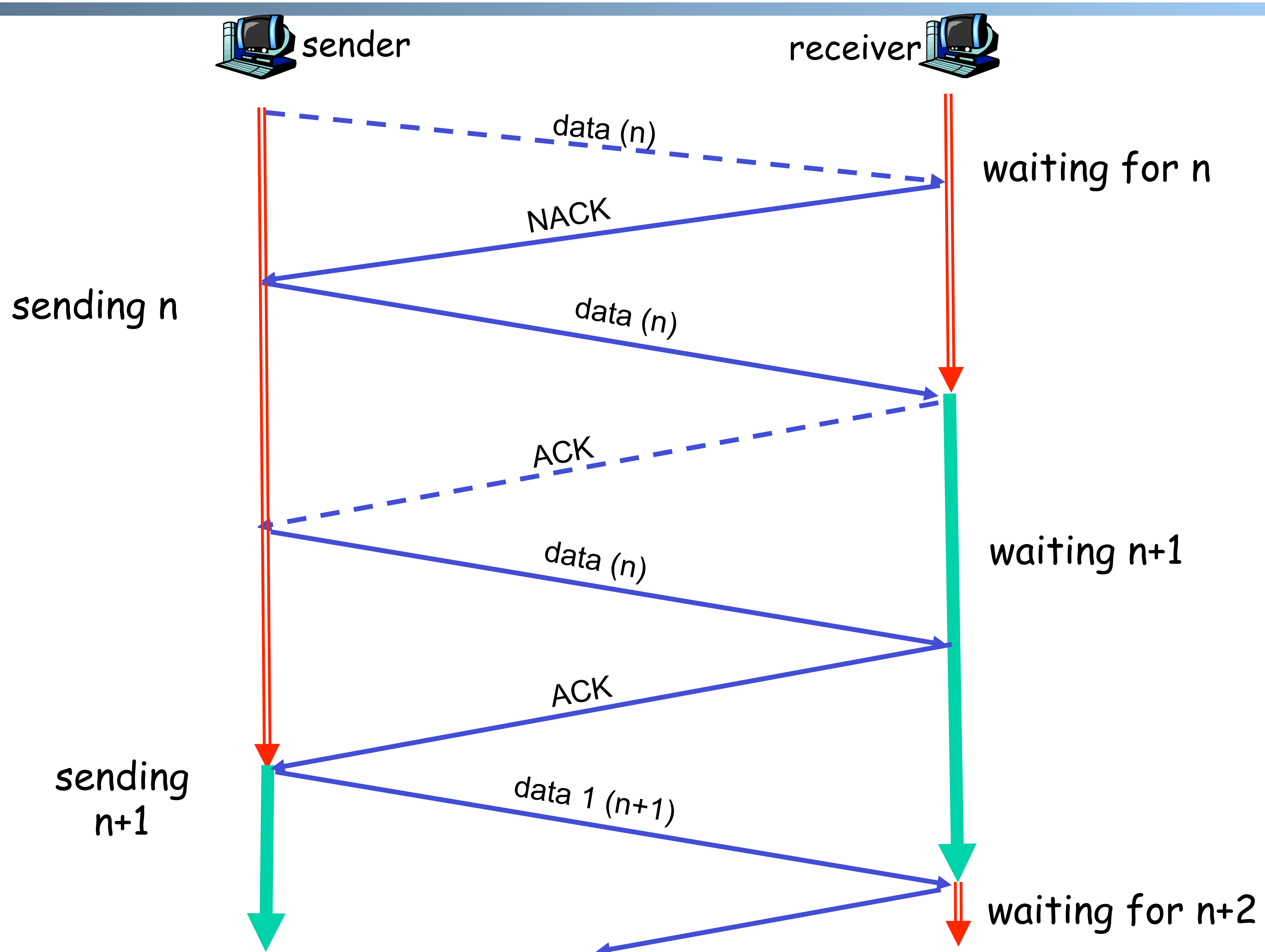
rdt2.1b: Sender, Handles Garbled ACK/ NAKs



rdt2.1b: Receiver, Handles Garbled ACK/NAKs

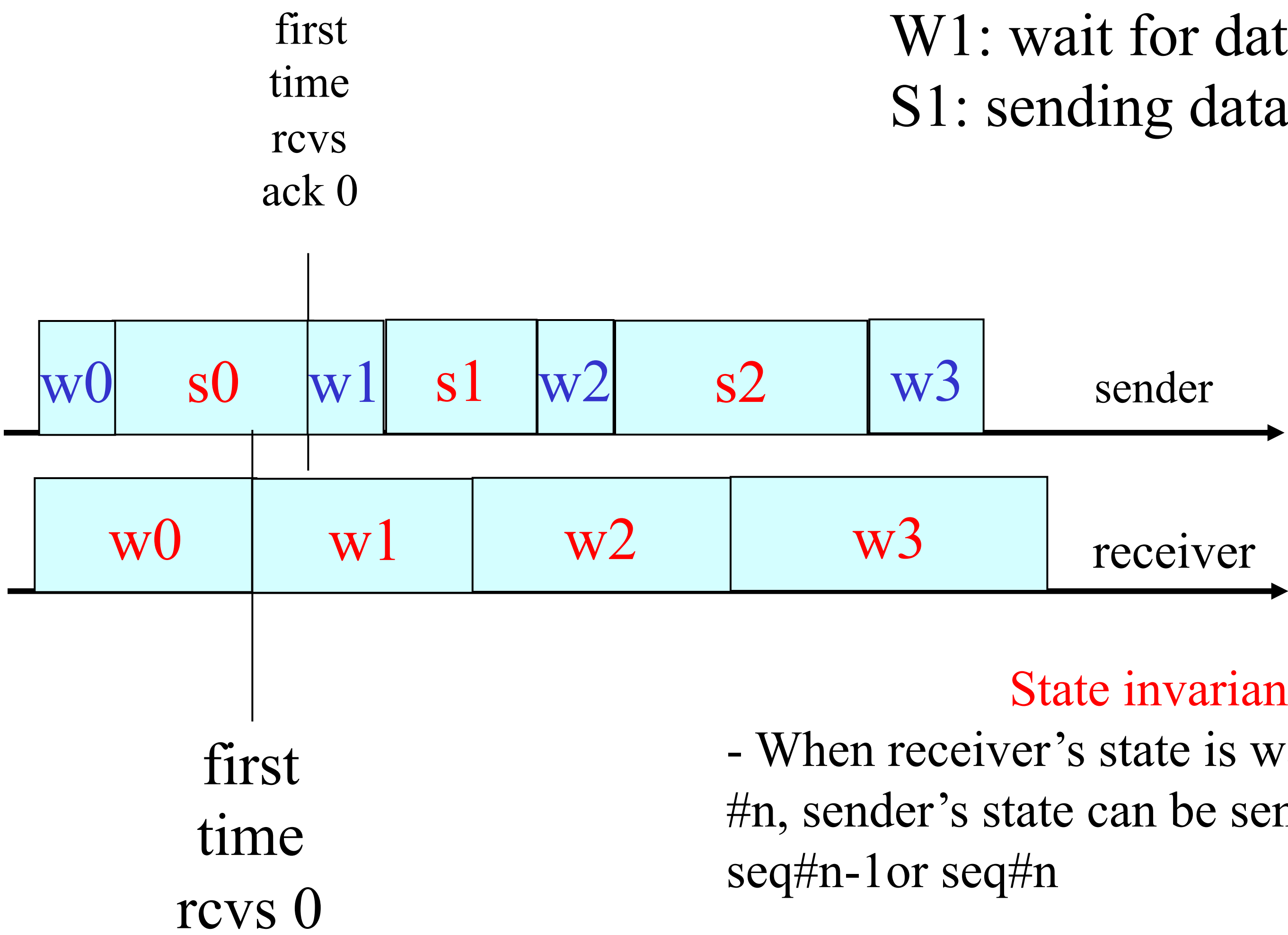


Another Look at rdt2.1b



State Relationship of rt2.1b

W1: wait for data with seq. 1
S1: sending data with seq. 1



State invariant:

- When receiver's state is waiting for seq #n, sender's state can be sending either seq#n-1or seq#n

rdt2.1b: Summary

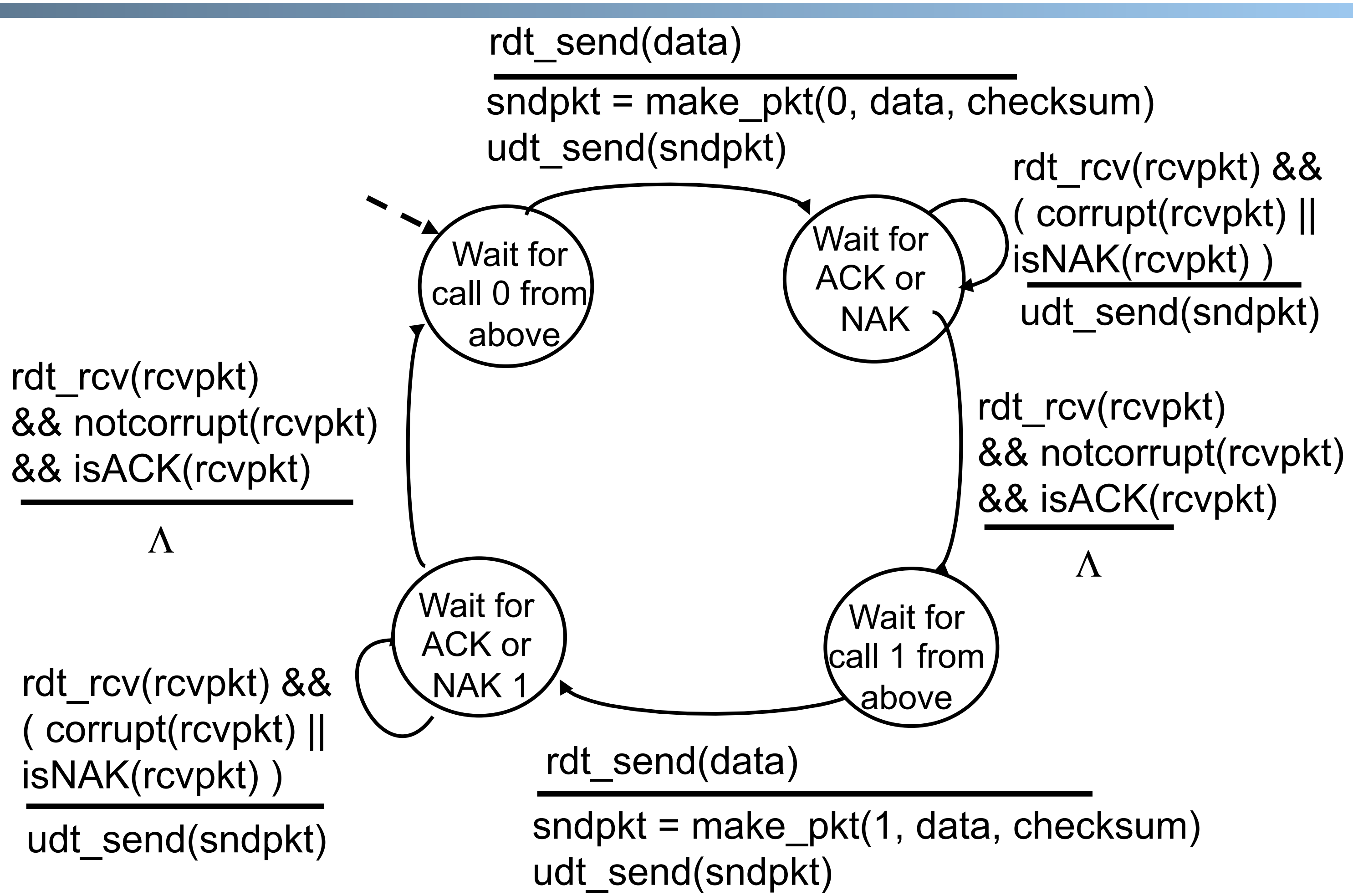
Sender:

- Seq # added to pkt
- Must check if received ACK/NAK corrupted

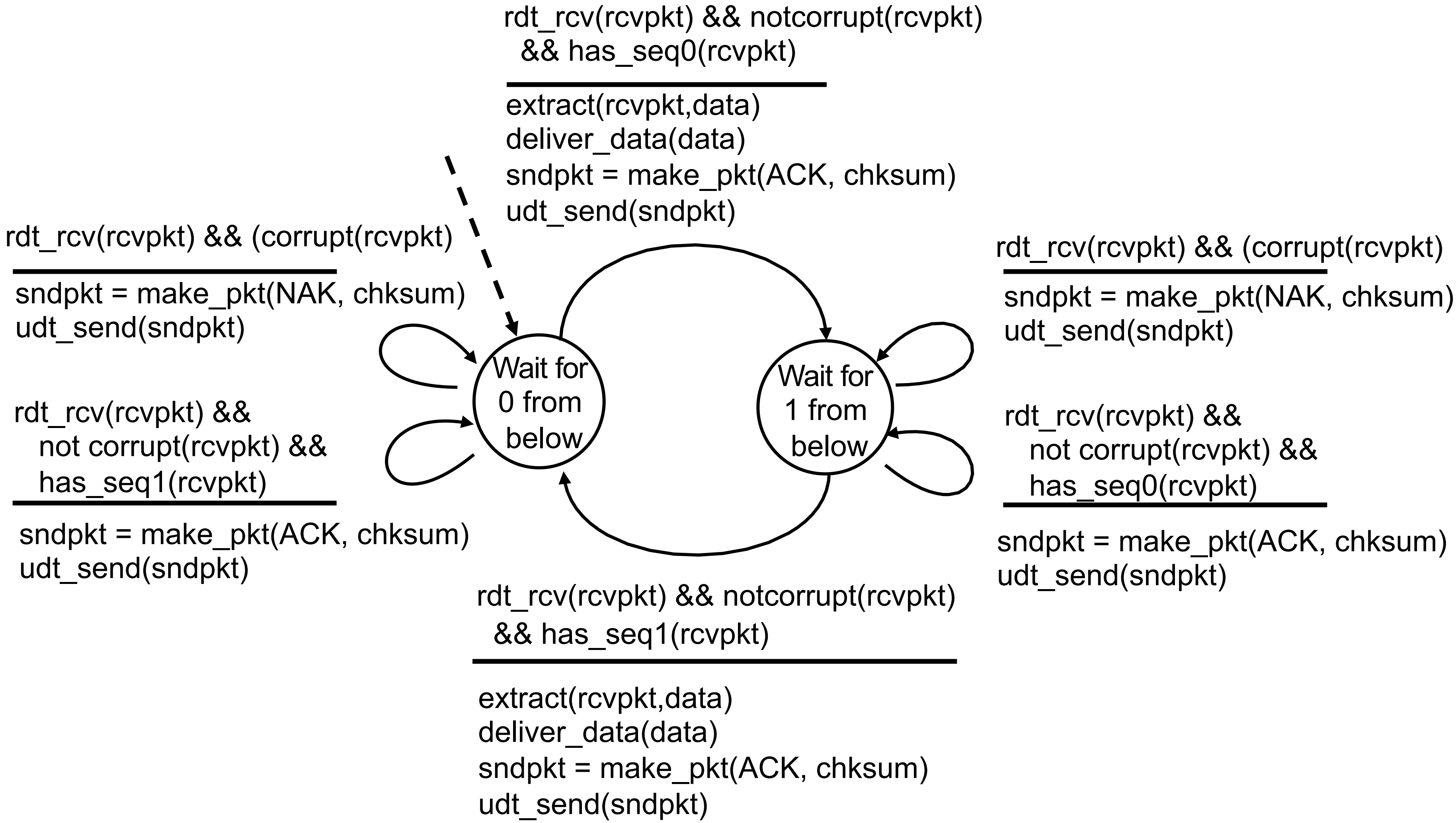
Receiver:

- Must check if received packet is duplicate
  - By checking if the packet has the expected pkt seq #

rdt2.1c: Sender, Handles Garbled ACK/ NAKs: Using 1 bit



rdt2.1c: Receiver, Handles Garbled ACK/NAKs: Using 1 bit



rdt2.1c: Discussion

Sender:

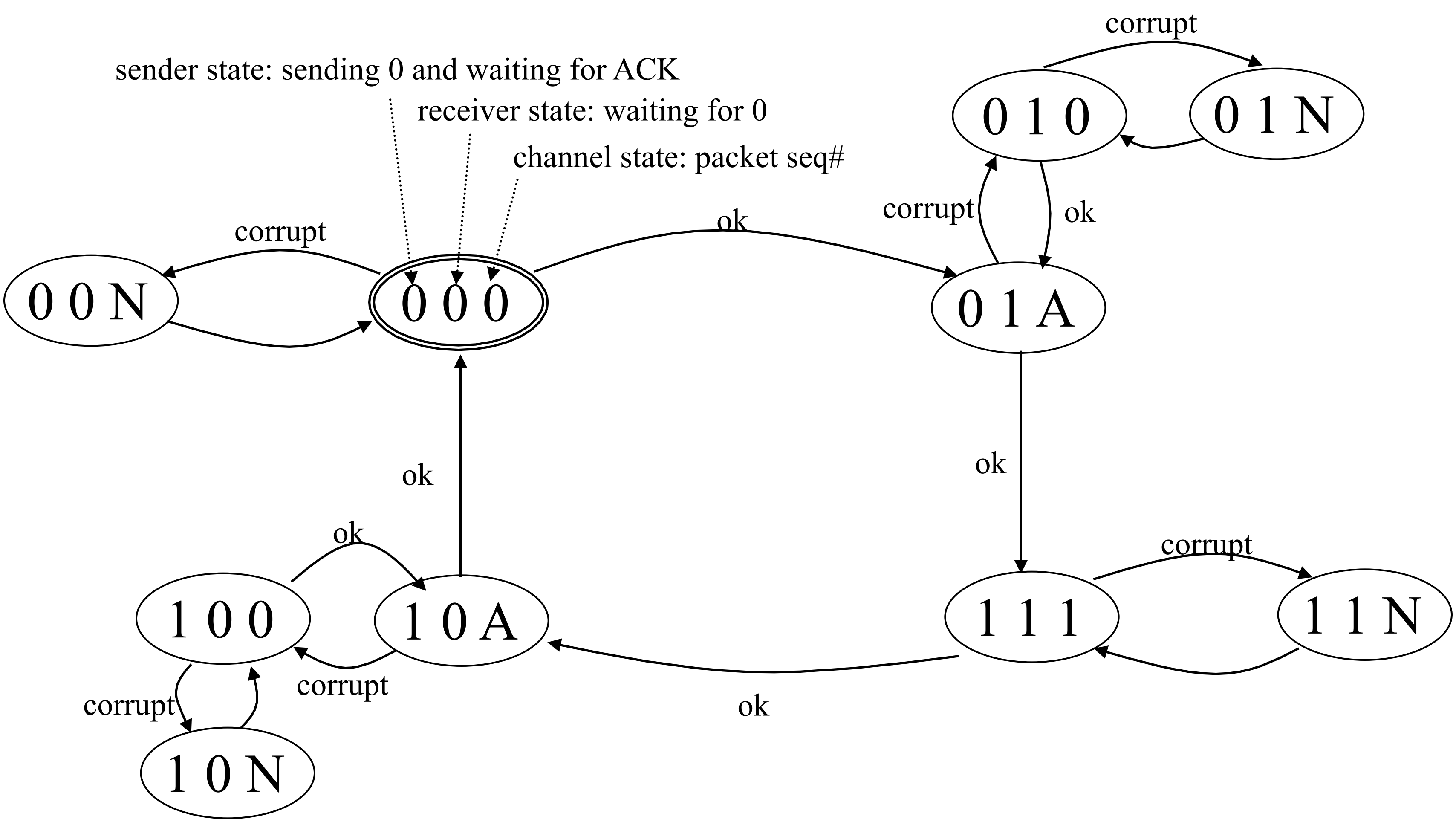
- State must “remember” whether “current” pkt has 0 or 1 seq. #

Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #

A Flavor of Protocol Correctness Analysis: rdt2.1c

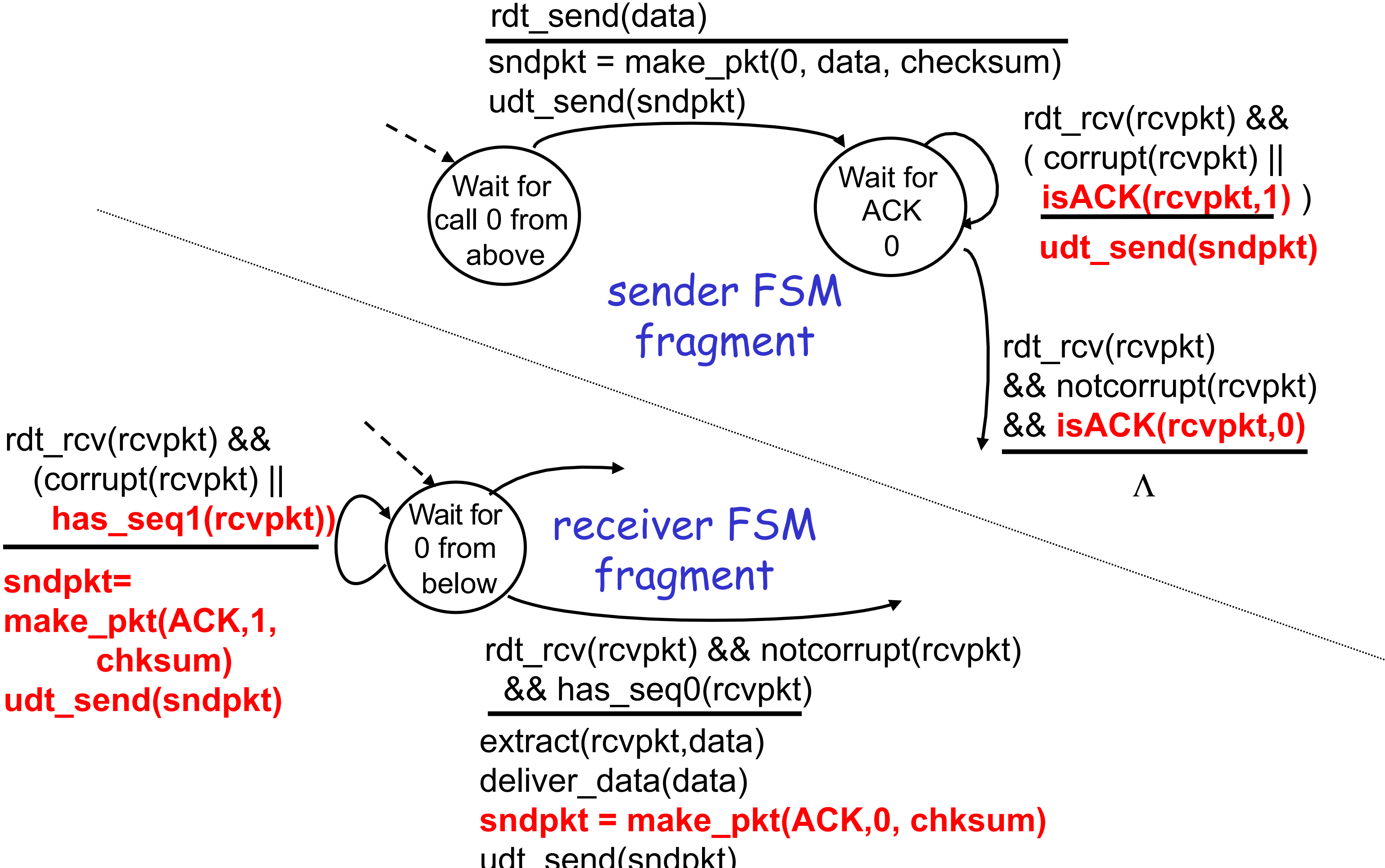
- Combined states of sender and channel
- Assume the sender always has data to send



rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1c, 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



Outline

- Review
- Overview of transport layer
- UDP
- Reliable data transfer
  - Perfect channel
  - Channel with bit errors
  - Channel with bit errors and losses

rdt3.0: Channels with Errors *and* Loss

New assumption: underlying channel can also lose packets (data or ACKs)

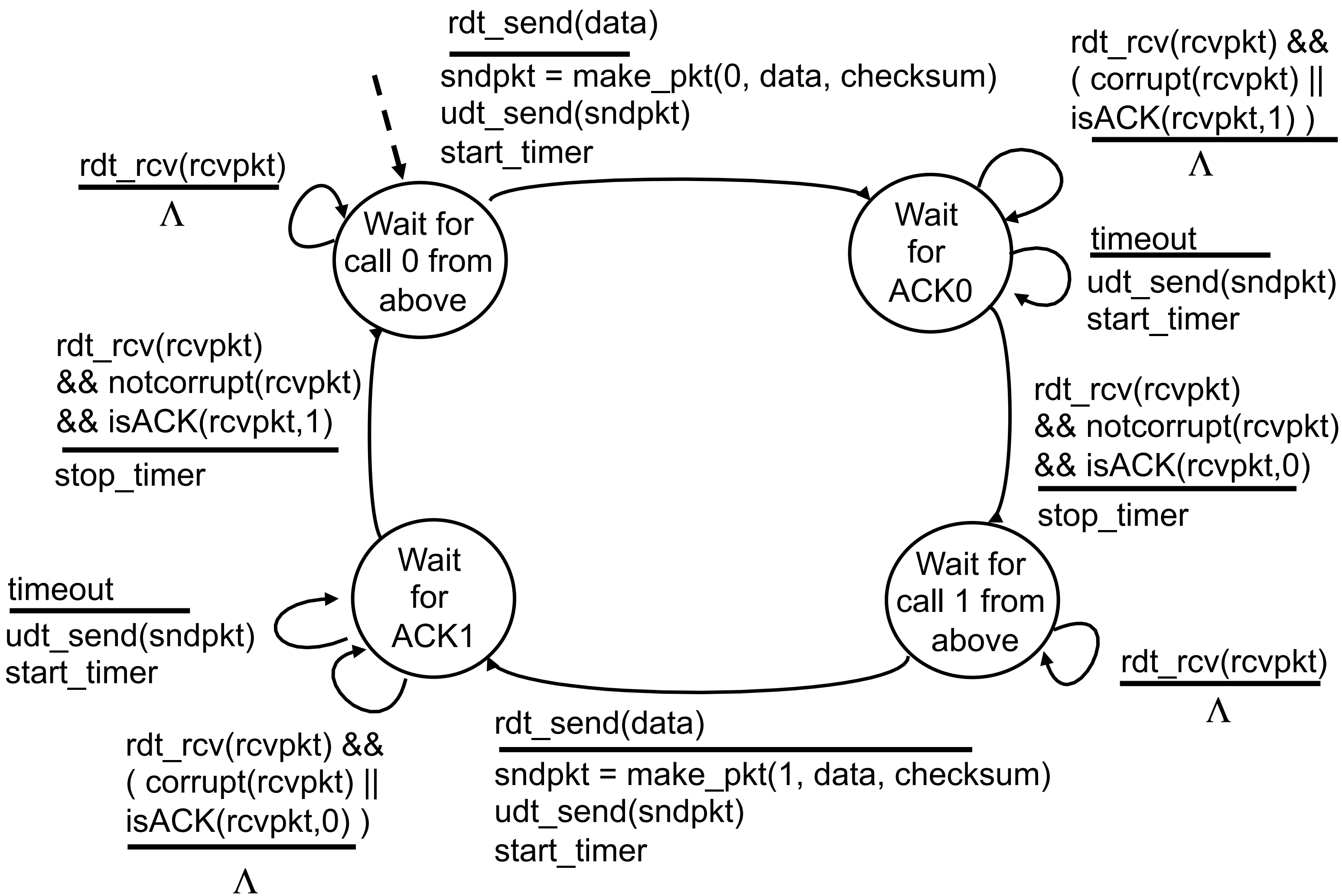
- Checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: how to deal with loss?

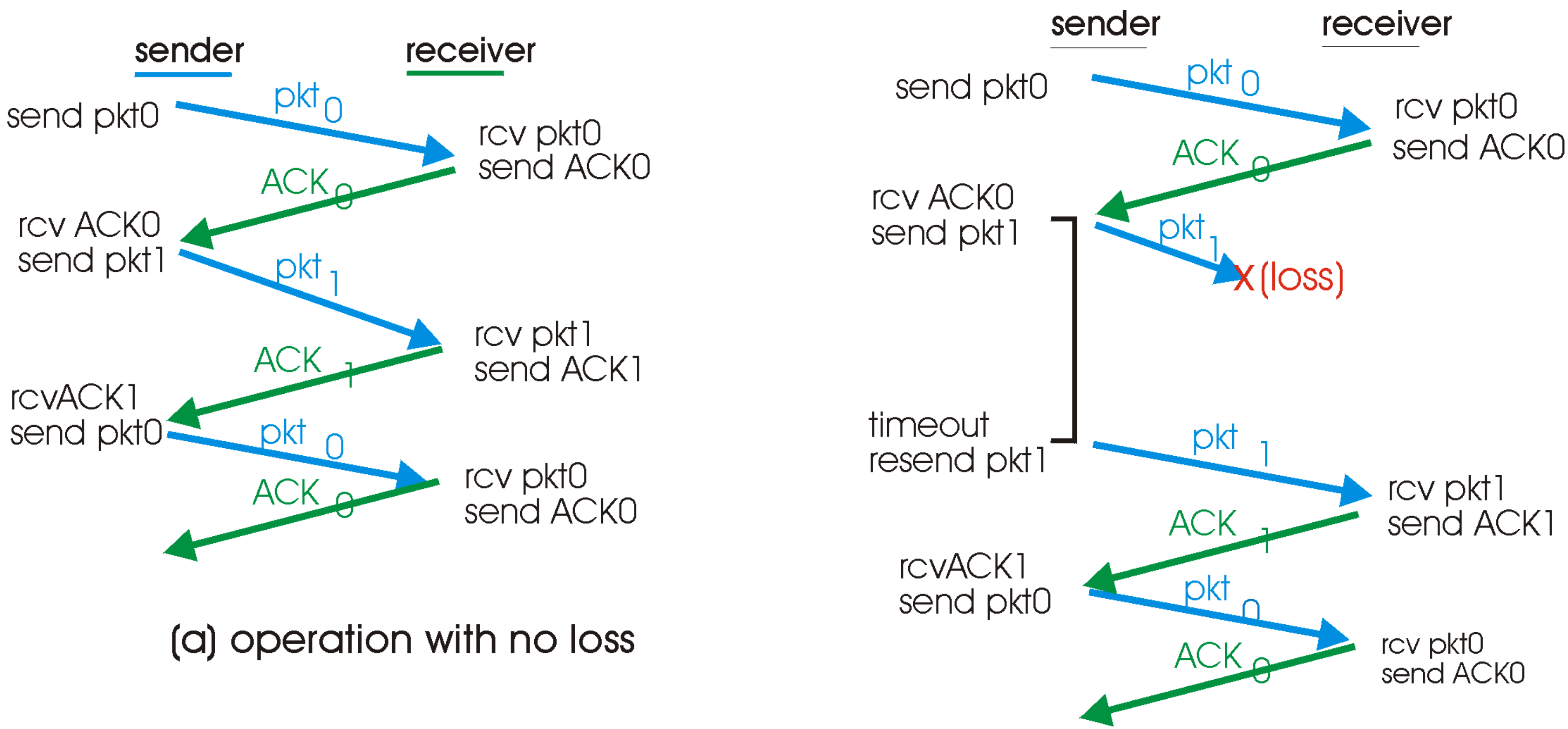
Approach: sender waits "reasonable" amount of time for ACK

- Requires countdown timer
- Retransmits if no ACK received in this time
- If pkt (or ACK) just delayed (not lost):
  - Retransmission will be duplicate, but use of seq. #'s already handles this
  - Receiver must specify seq # of pkt being ACKed

rdt3.0 Sender



rdt3.0 in Action

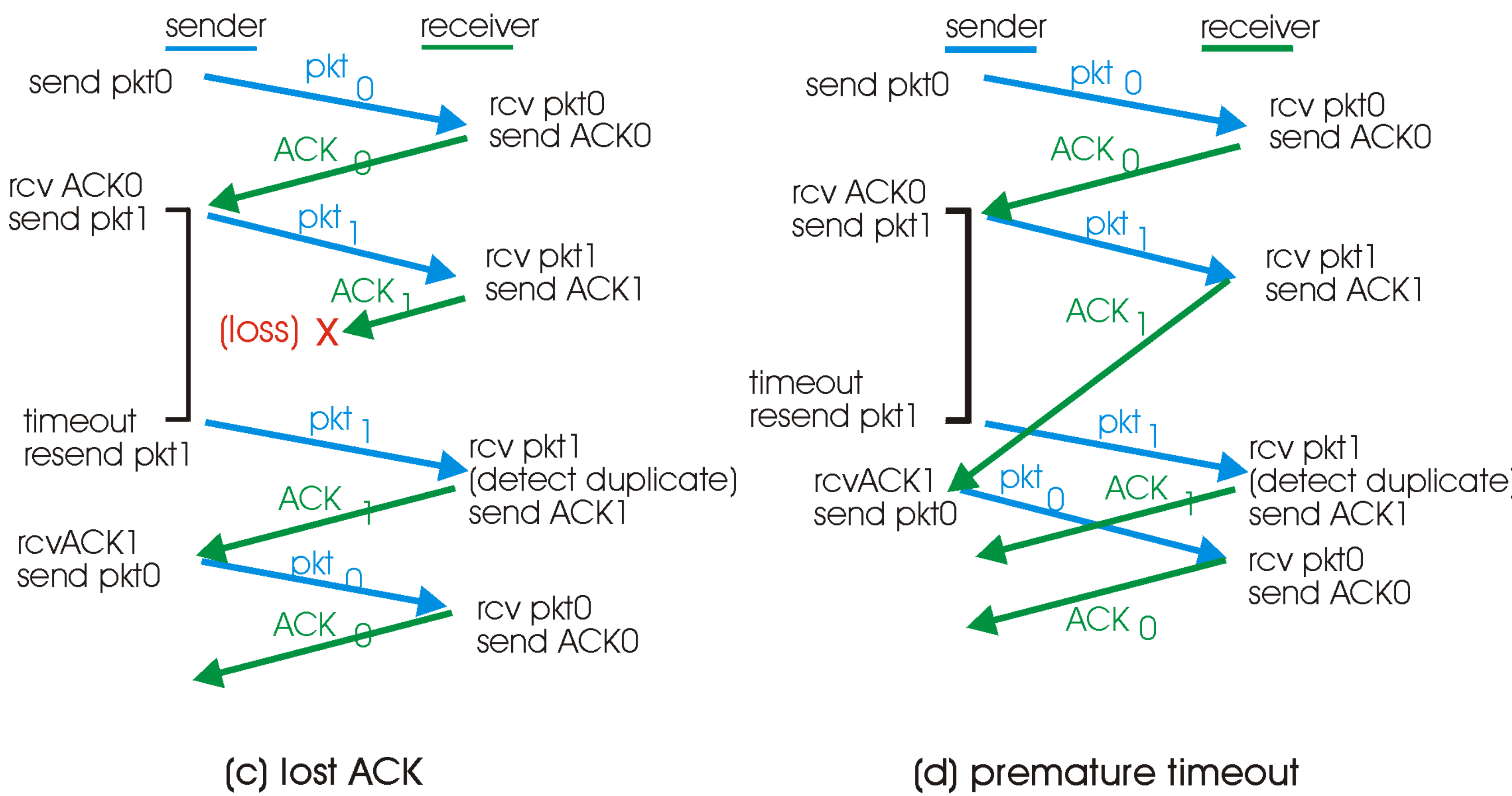


(a) operation with no loss

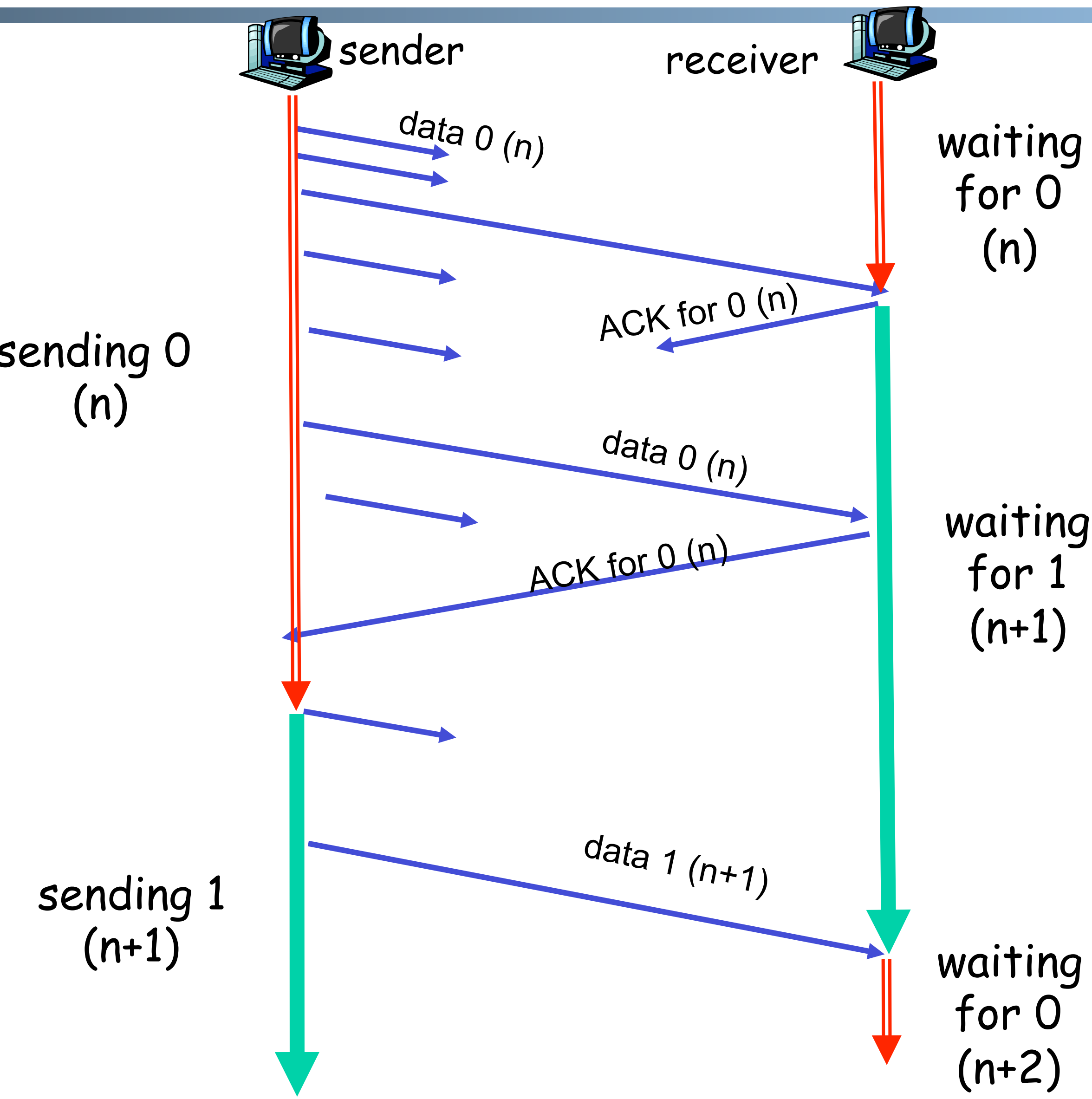
(b) lost packet

Question to think about: How to determine a good timeout value?

rdt3.0 in Action



rdt3.0

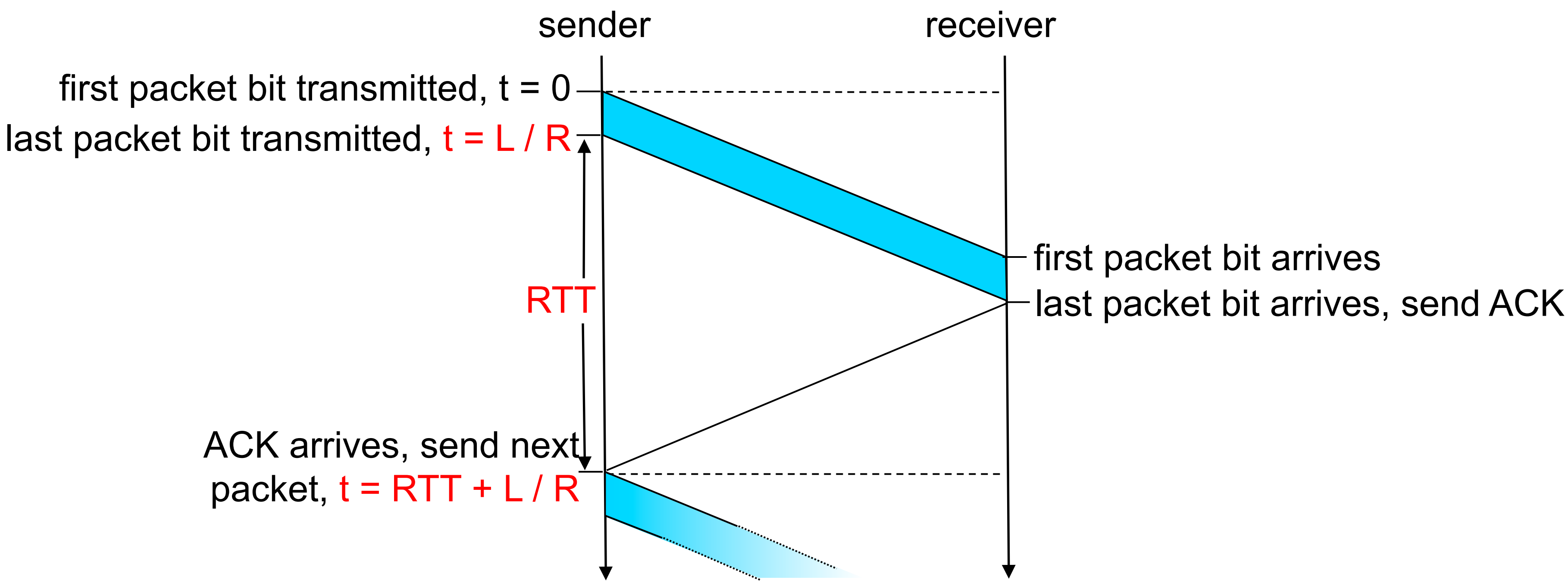


State consistency:

When receiver's state is waiting n+1, the state of the sender is either sending for n+1 or sending for n

When sender's state is sending for n, receiver's state is waiting for n or n + 1

rdt3.0: Stop-and-Wait Operation



What is U_{sender}: utilization – fraction of time sender busy sending?

Assume: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

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

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

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - Network protocol limits use of physical resources !

Questions

- How to improve the performance of rdt3.0?

- How to deal with loss: i.e., how to determine the "right" timeout value?

- What if there are reordering and duplication?