
Network Transport Layer: Stop-and-Wait, Sliding Window

Qiao Xiang, Congming Gao, Qiang Su

<https://sngroup.org.cn/courses/cnns-xmuf25/index.shtml>

10/28/2025

Outline

- ❑ Admin and recap
- ❑ Overview of transport layer
- ❑ UDP
- ❑ Reliable data transfer, the stop-and-go protocols

Admin

- ❑ Midterm exam this afternoon
- ❑ Lab 3 due on Oct.29

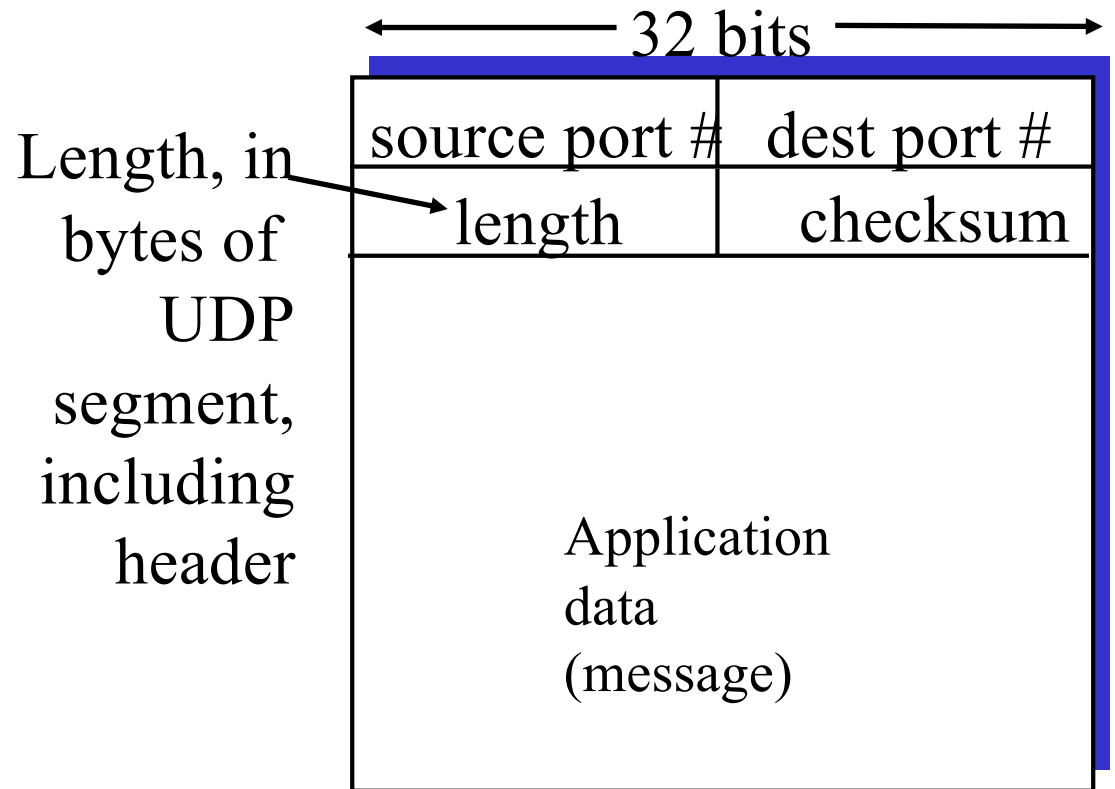
Recap: User Datagram Protocol [RFC 768]

- ❑ Often used for streaming multimedia apps

- loss tolerant
- rate sensitive

- ❑ Other UDP uses

- 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
- ❑ checksum: addition of segment contents to be zero
- ❑ sender puts checksum value into UDP checksum field

Receiver:

- ❑ compute sum of segment and checksum; check if sum zero
 - NO - error detected
 - YES - no error detected.
But maybe errors nonetheless?

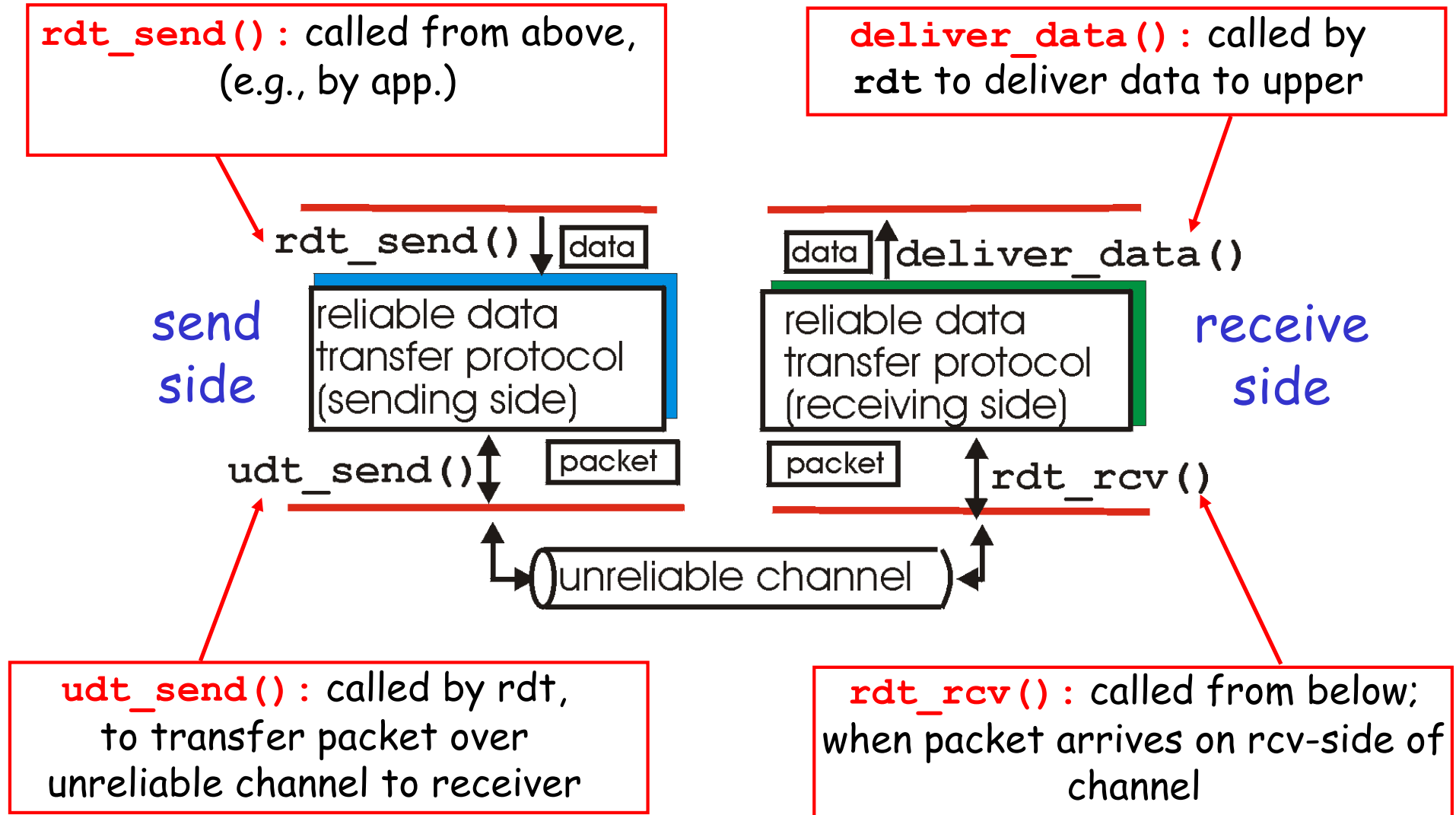
UDP Checksum: Algorithm

□ 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
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

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

Recap: Reliable Data Transfer Context

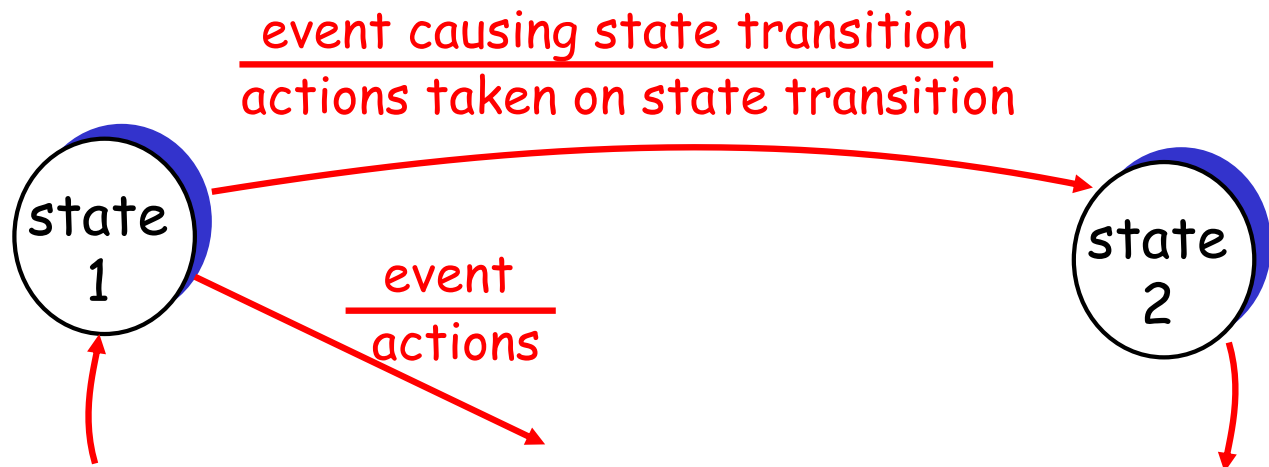


Reliable Data Transfer: Getting Started

We'll:

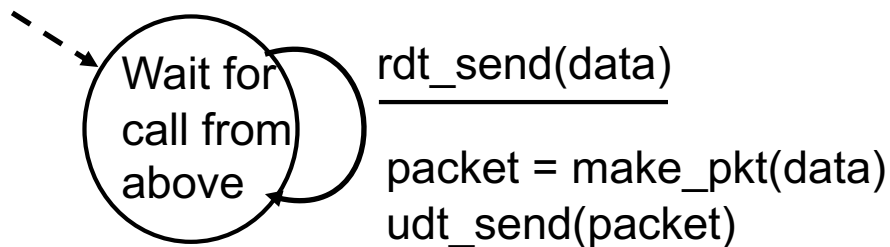
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
 - but control info will flow on both directions !
- use **finite state machines (FSM)** to specify sender, receiver

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

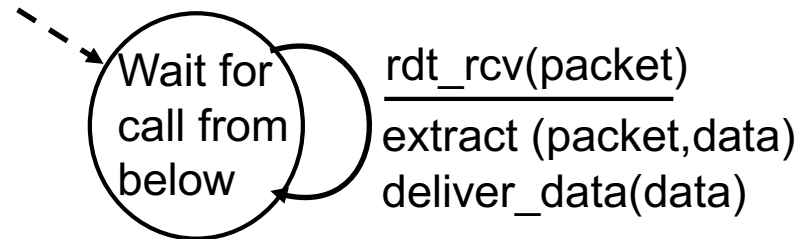


Rdt1.0: reliable transfer over a reliable channel

- separate FSMs for sender, receiver:
 - sender sends data into underlying channel
 - receiver reads data from underlying channel



sender



receiver

Exercise: Prove correctness of Rdt1.0.

Correctness: for every single packet, one and only one copy is received by receiver correctly (no error) and in-order

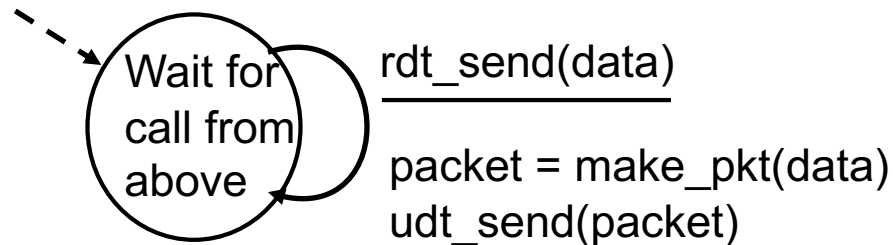
Potential Channel Errors

- ❑ bit errors
- ❑ loss (drop) of packets
- ❑ reordering or duplication

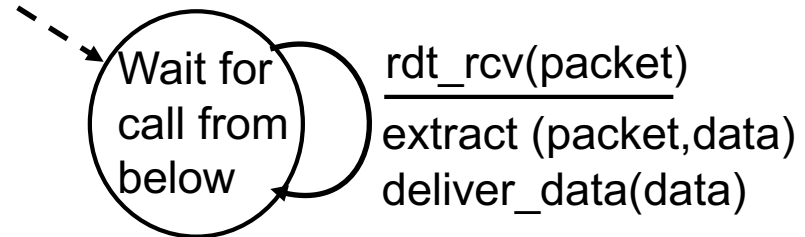
Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).

rdt2.0: Channel With Bit Errors

- Assume: Underlying channel **may only flip bits** in packet



sender



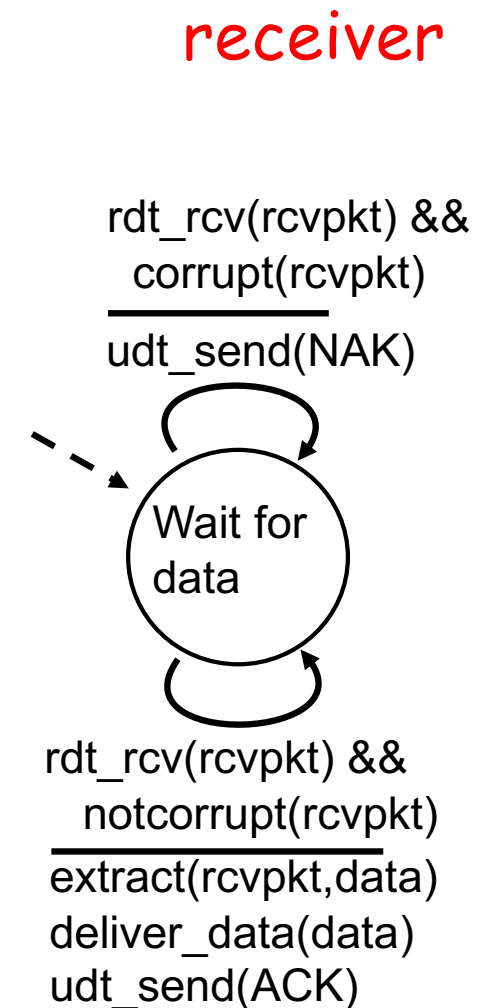
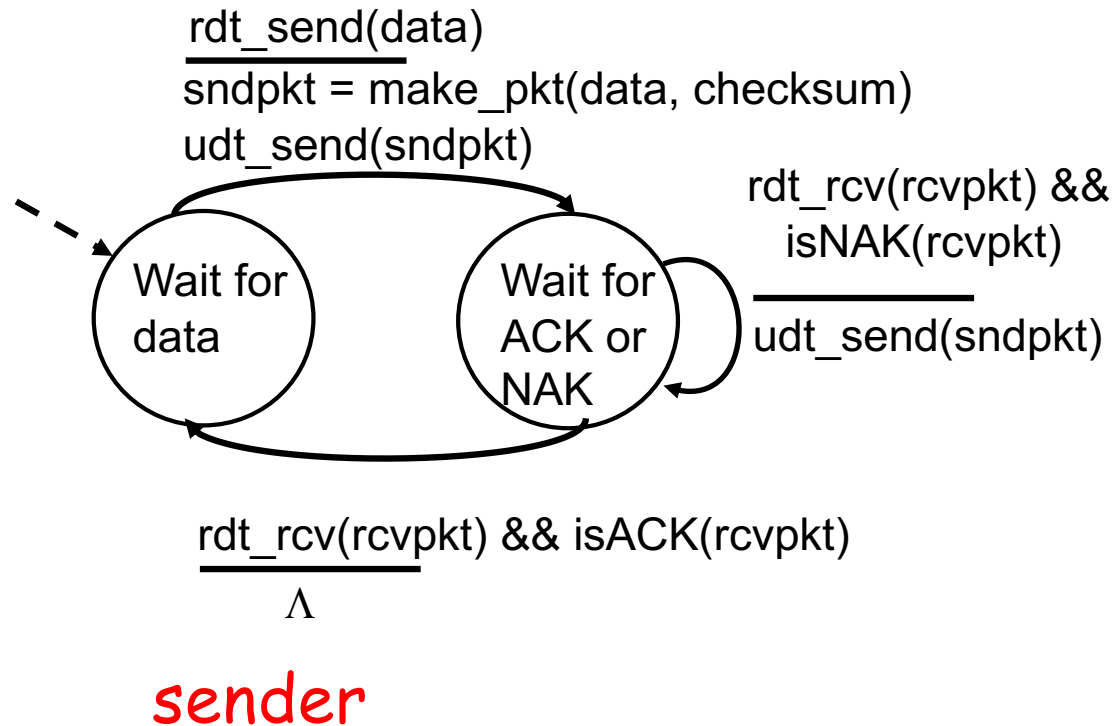
receiver

Exercise: What correctness requirement(s) rdt1.0 cannot provide?

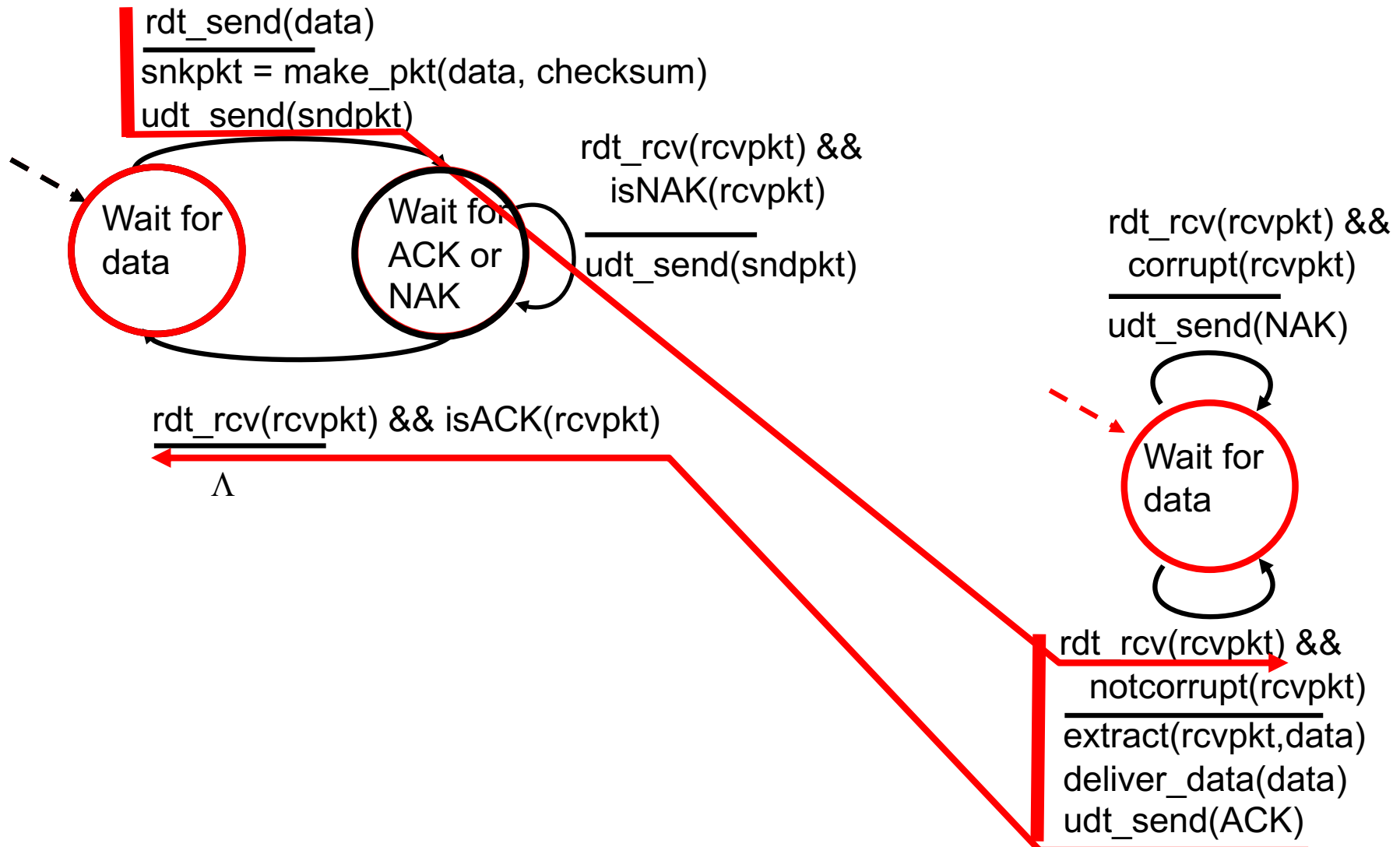
rdt2.0: Channel With Bit Errors

- ❑ New mechanisms in `rdt2.0` (beyond `rdt1.0`):
 - receiver error detection: recall: UDP checksum/Ethernet CRC detects bit errors
 - receiver feedback: control msgs (ACK,NAK) rcvr->sender
 - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
 - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
 - sender retransmission
 - sender retransmits pkt on receipt of NAK

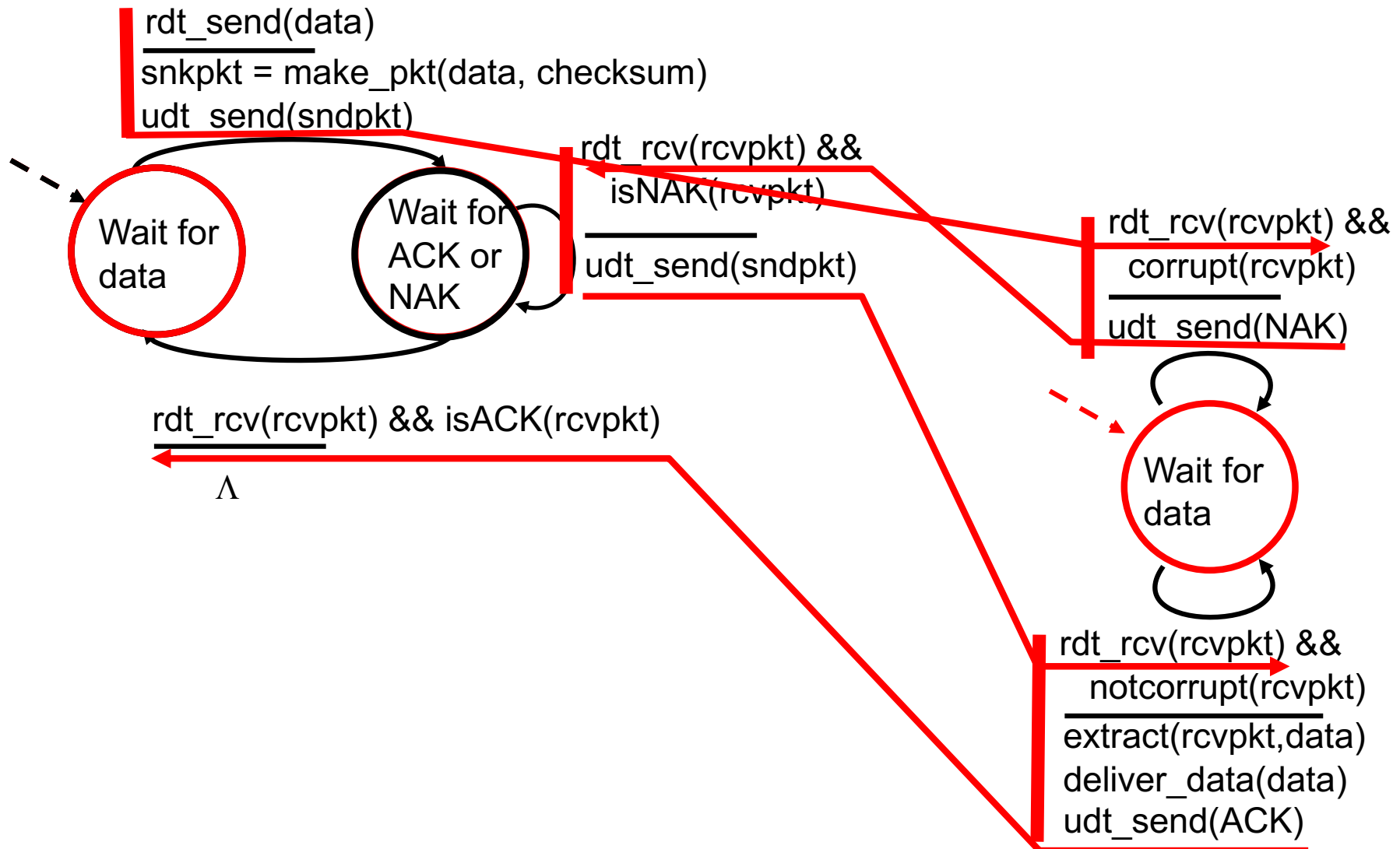
rdt2.0: FSM Specification



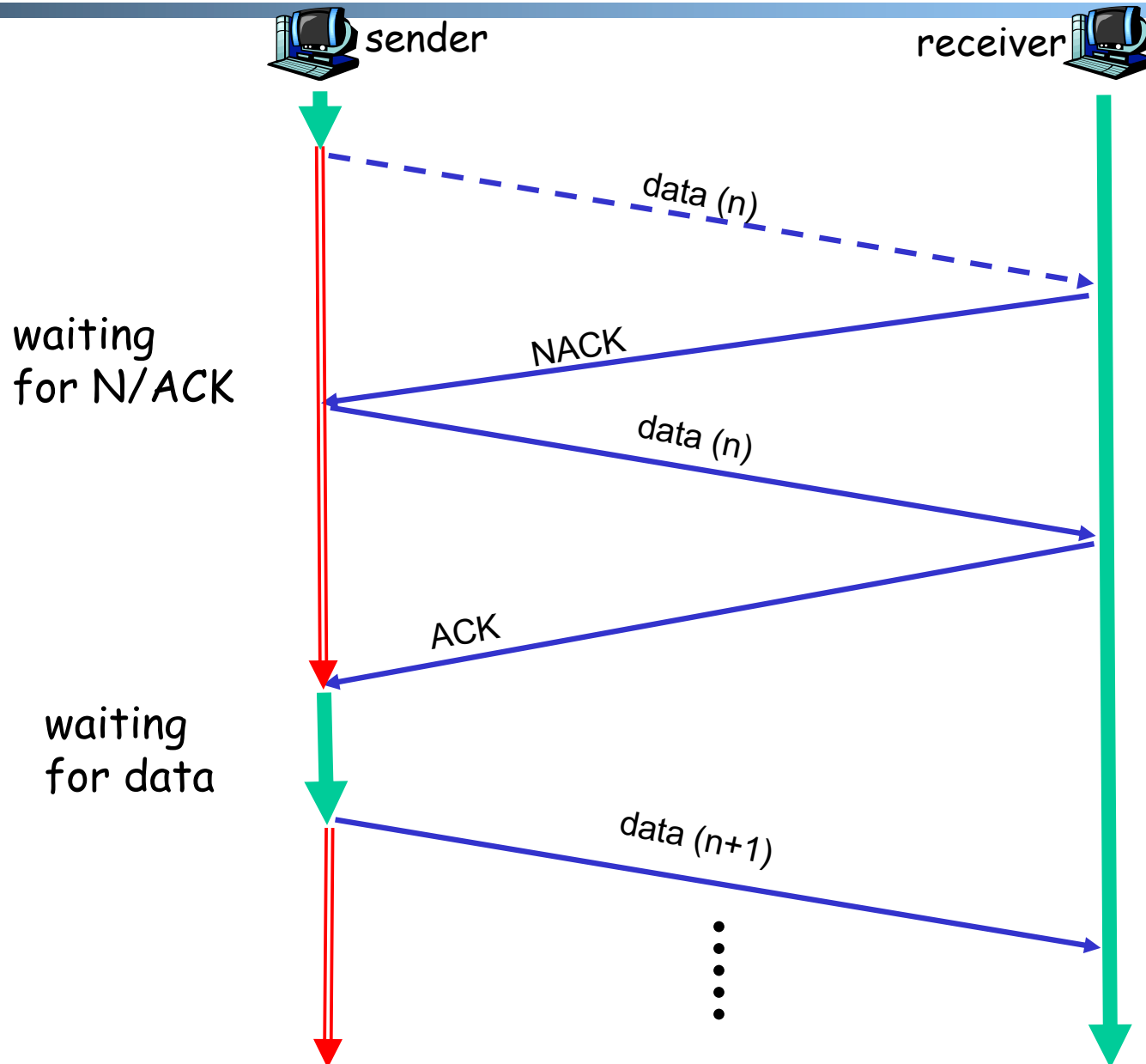
rdt2.0: Operation with No Errors



rdt2.0: Error Scenario



Rdt2.0 Analysis



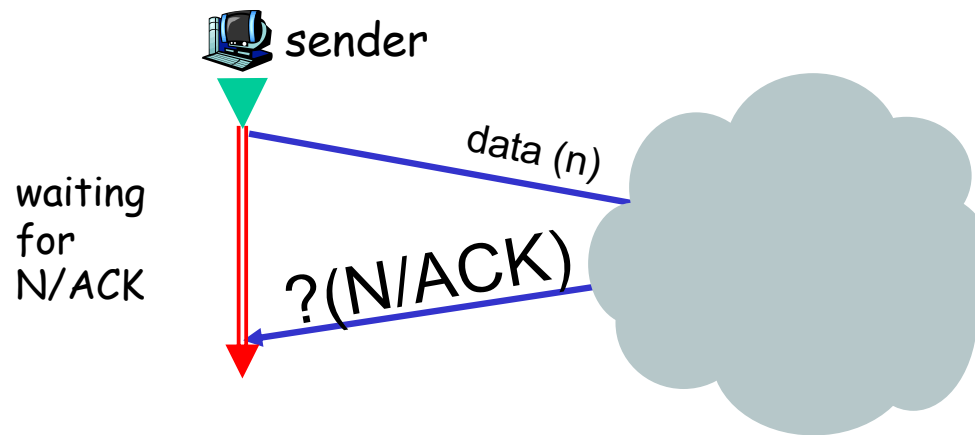
Execution traces
of rdt2.0:
 $\{\text{data}^* \text{ NACK}\}^*$
data deliver
ACK

Analyzing set of all
possible execution
traces is a common
technique to
understand and
analyze many types
of distributed
protocols.

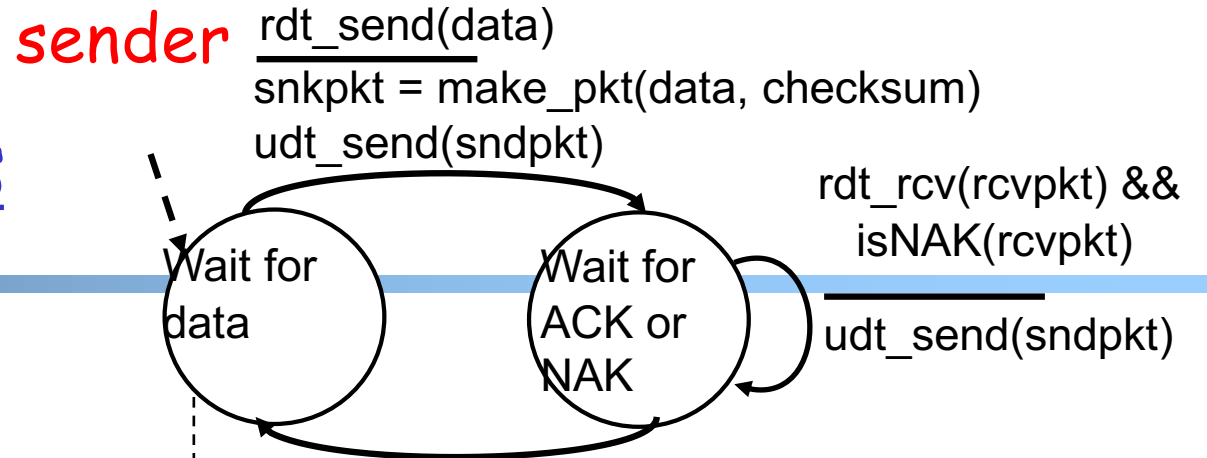
rdt2.0 is Incomplete!

What happens if ACK/NAK corrupted?

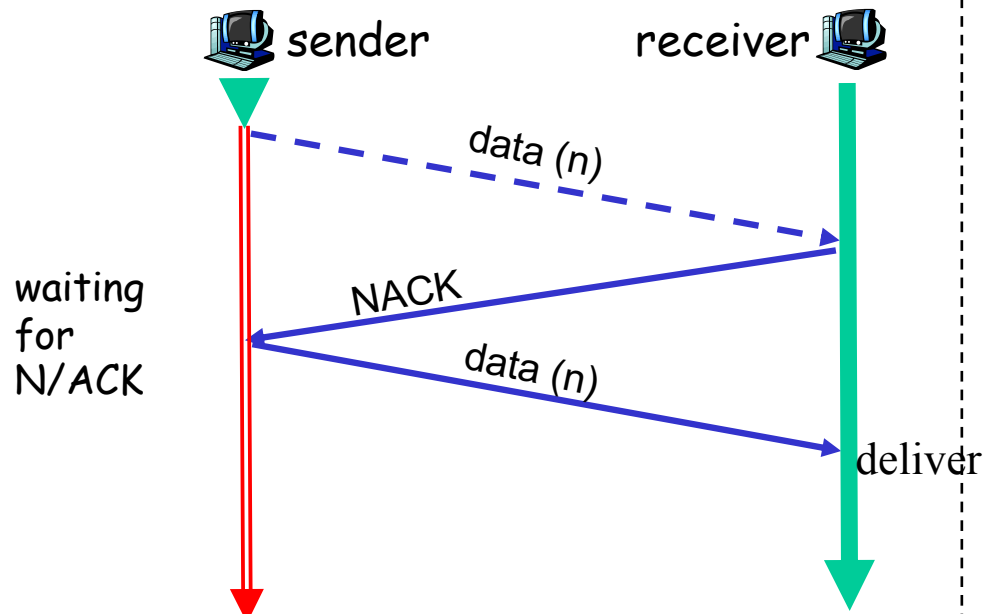
- Although sender receives feedback, but doesn't know what happened at receiver!



Two Possibilities

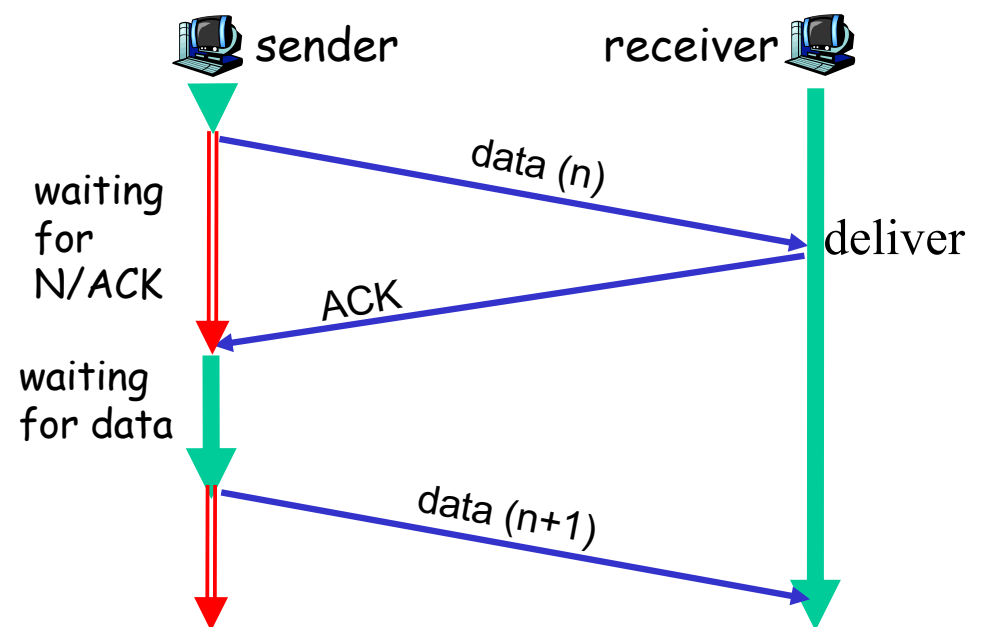


sender can't just guess NACK:
if wrong, duplicate



Fix miss guess NACK:
provide info for receiver to distinguish

sender can't just guess ACK:
if wrong, missing pkt



Home exercise: fix miss guess ACK

Handle Control Message Corruption

Handling ambiguity:

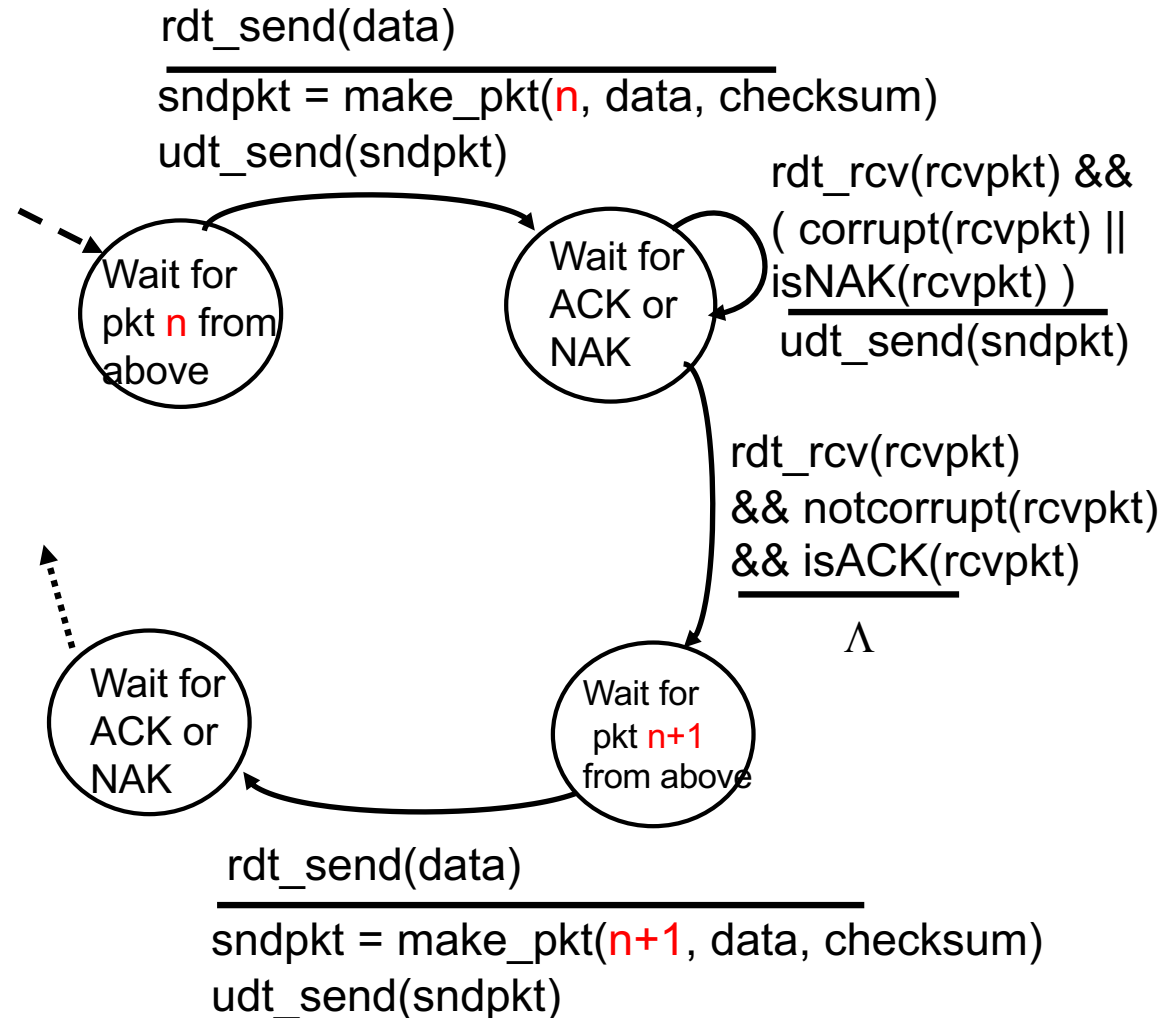
- ❑ sender adds *sequence number* to each pkt
- ❑ sender retransmits current pkt if ACK/NAK garbled
 - Guess NACK
- ❑ receiver discards (doesn't deliver up) duplicate pkt
 - fix effect of wrong guess

stop and wait

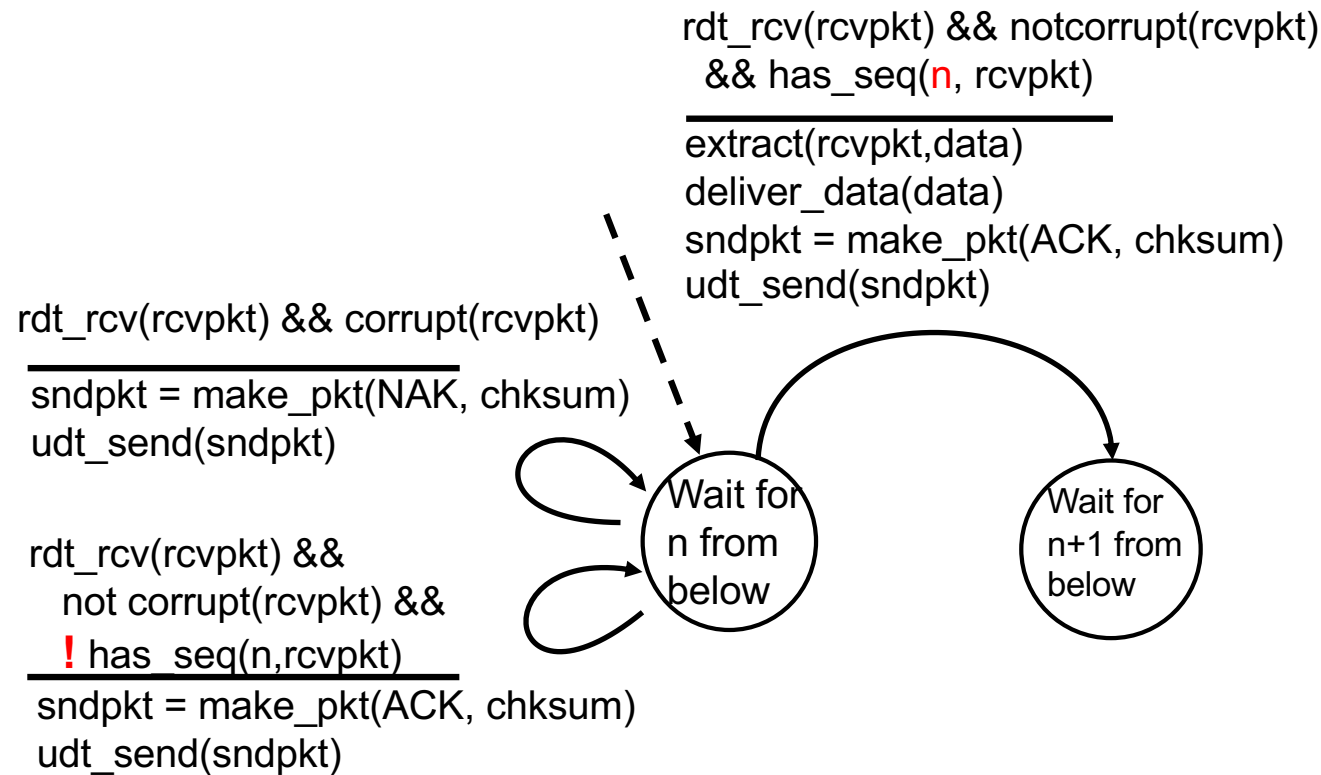
sender sends one packet,
then waits for receiver
response

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

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



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



rdt2.1b: Summary

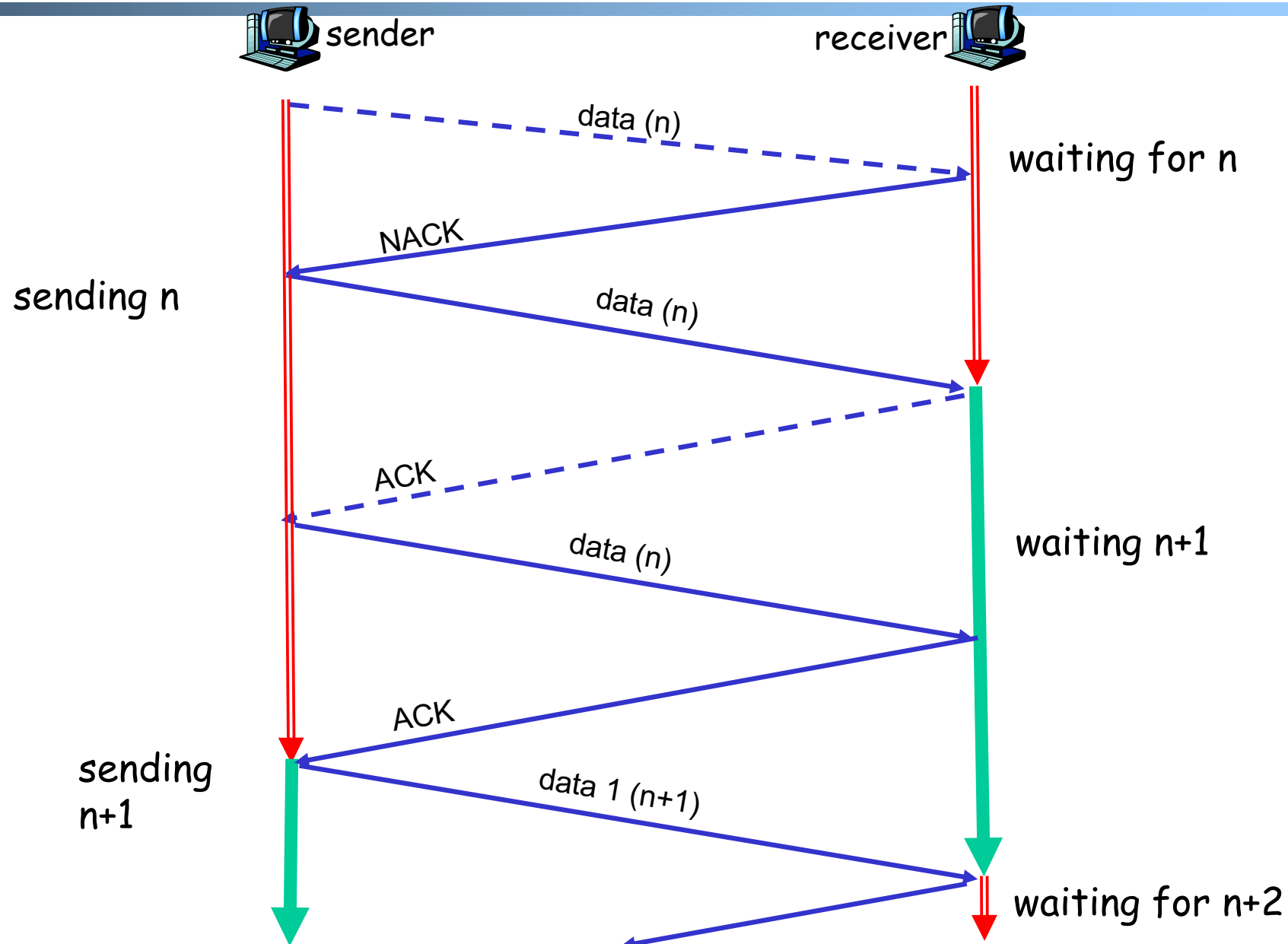
Sender:

- ❑ 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 #

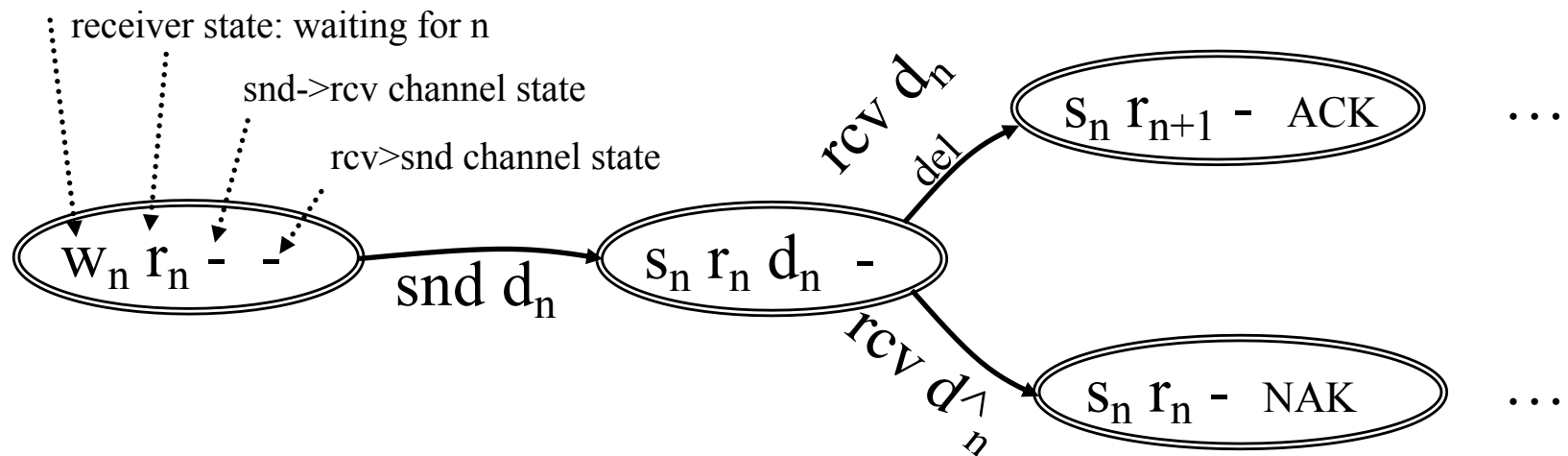
rdt2.1b Analysis: Execution Traces?



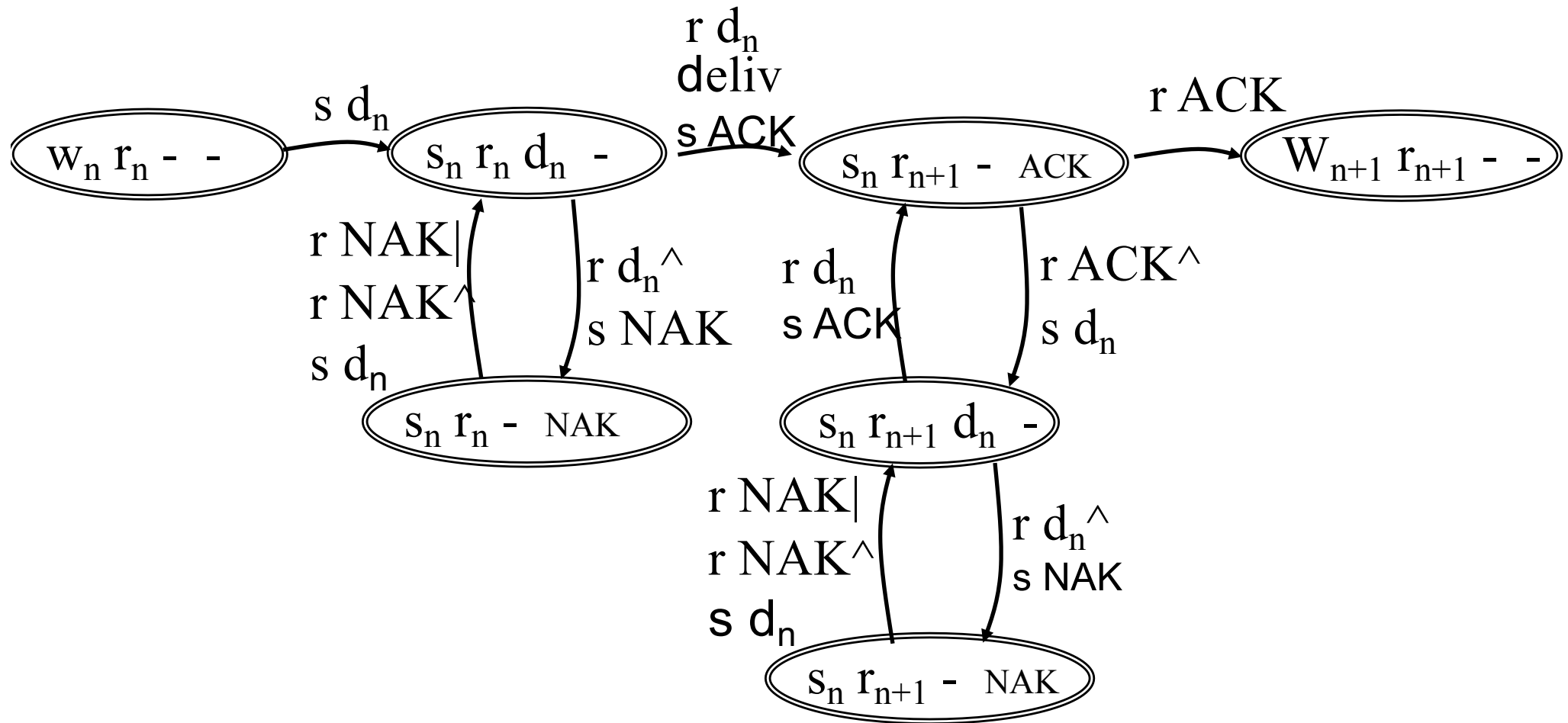
Protocol Analysis using (Generic) Execution Traces Technique

- Issue: how to systematically enumerate all potential execution traces to understand and verify correctness
- A systematic approach to enumerating exec. traces is to compute **joint sender/receiver/channels state machine**

sender state: waiting for n

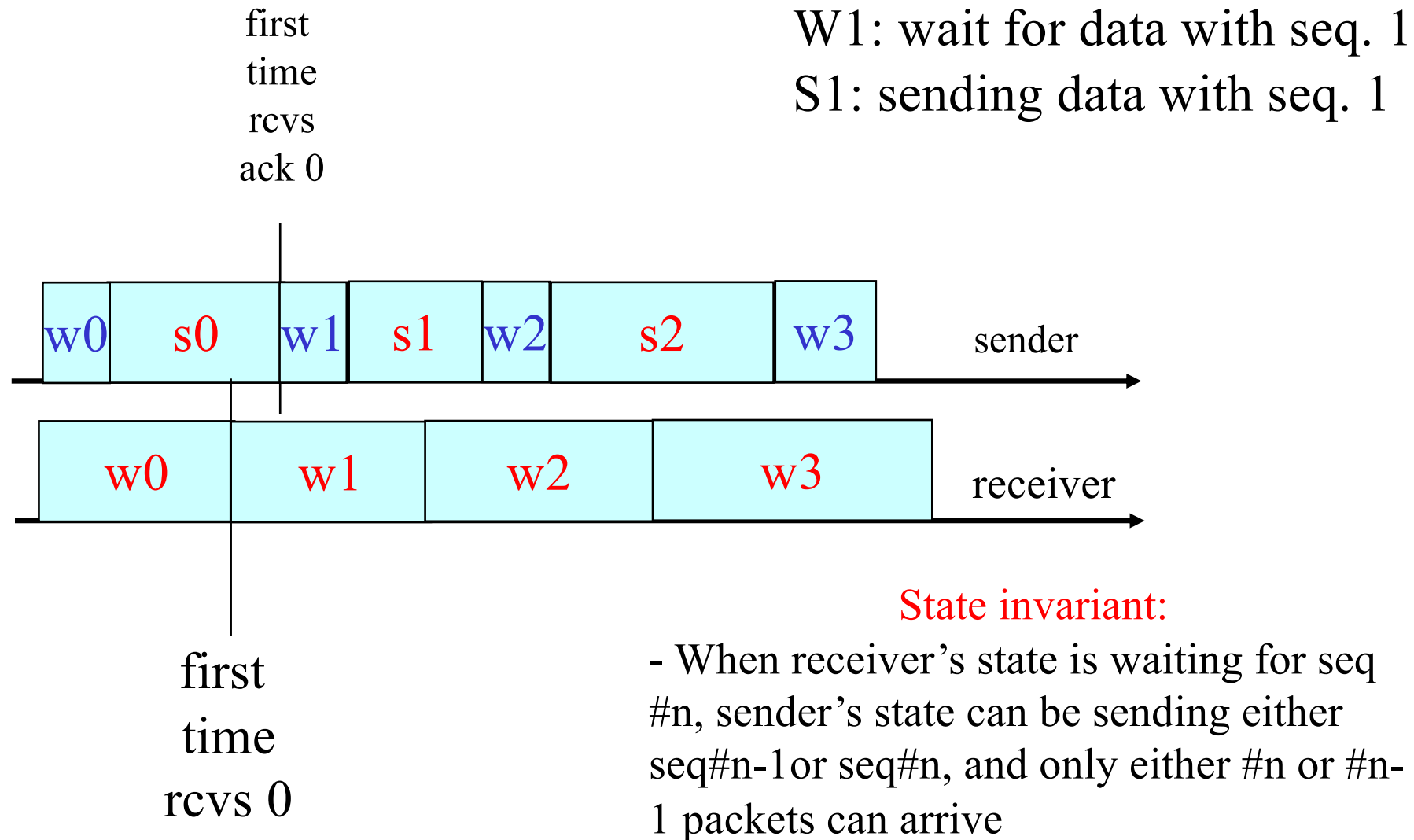


Recap: Protocol Analysis using (Generic) Execution Traces Technique

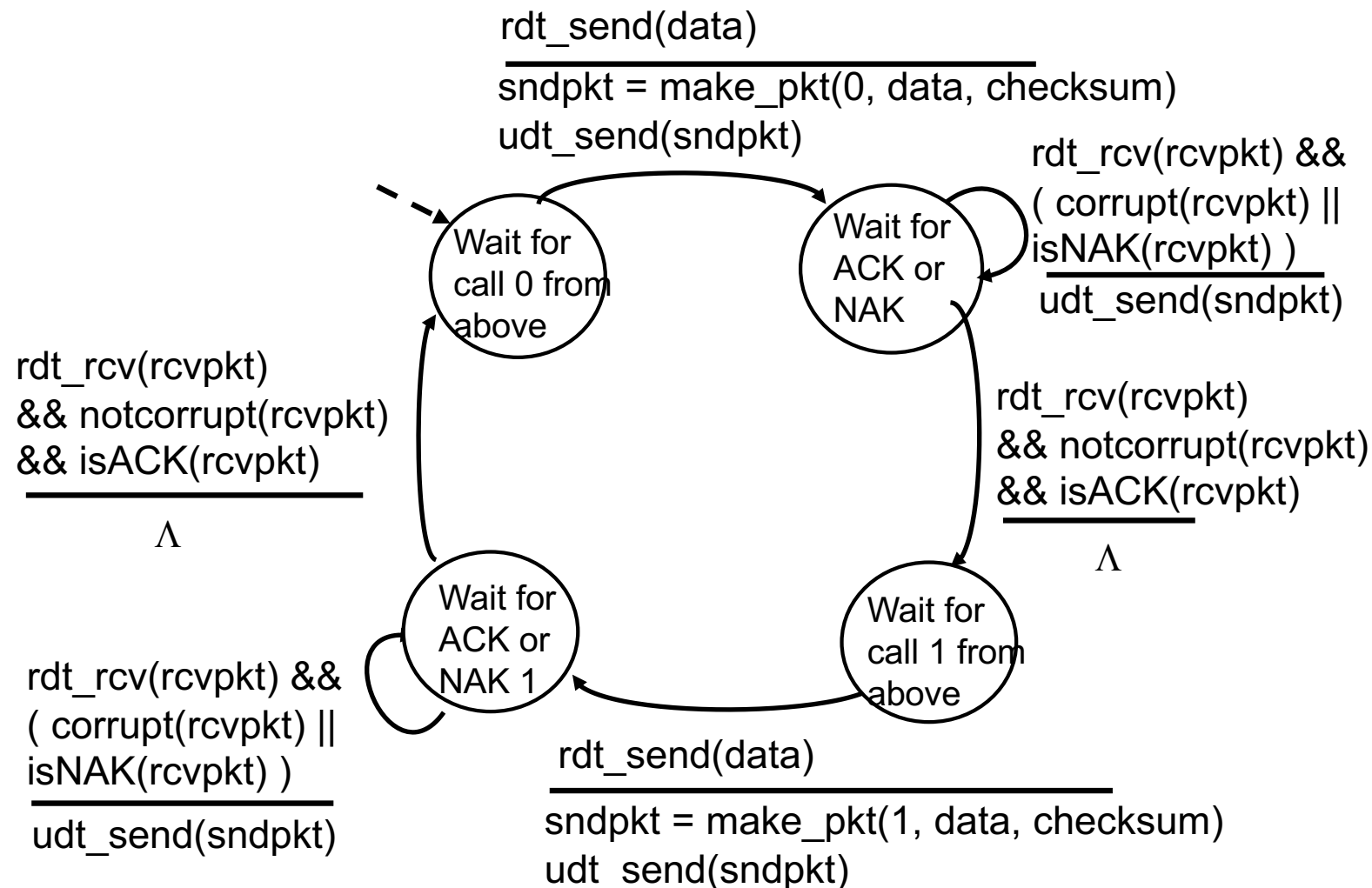


Execution traces of rdt2.1b are all that can be generated by the finite state machine above.

rdt2.1b Analysis: State Invariants

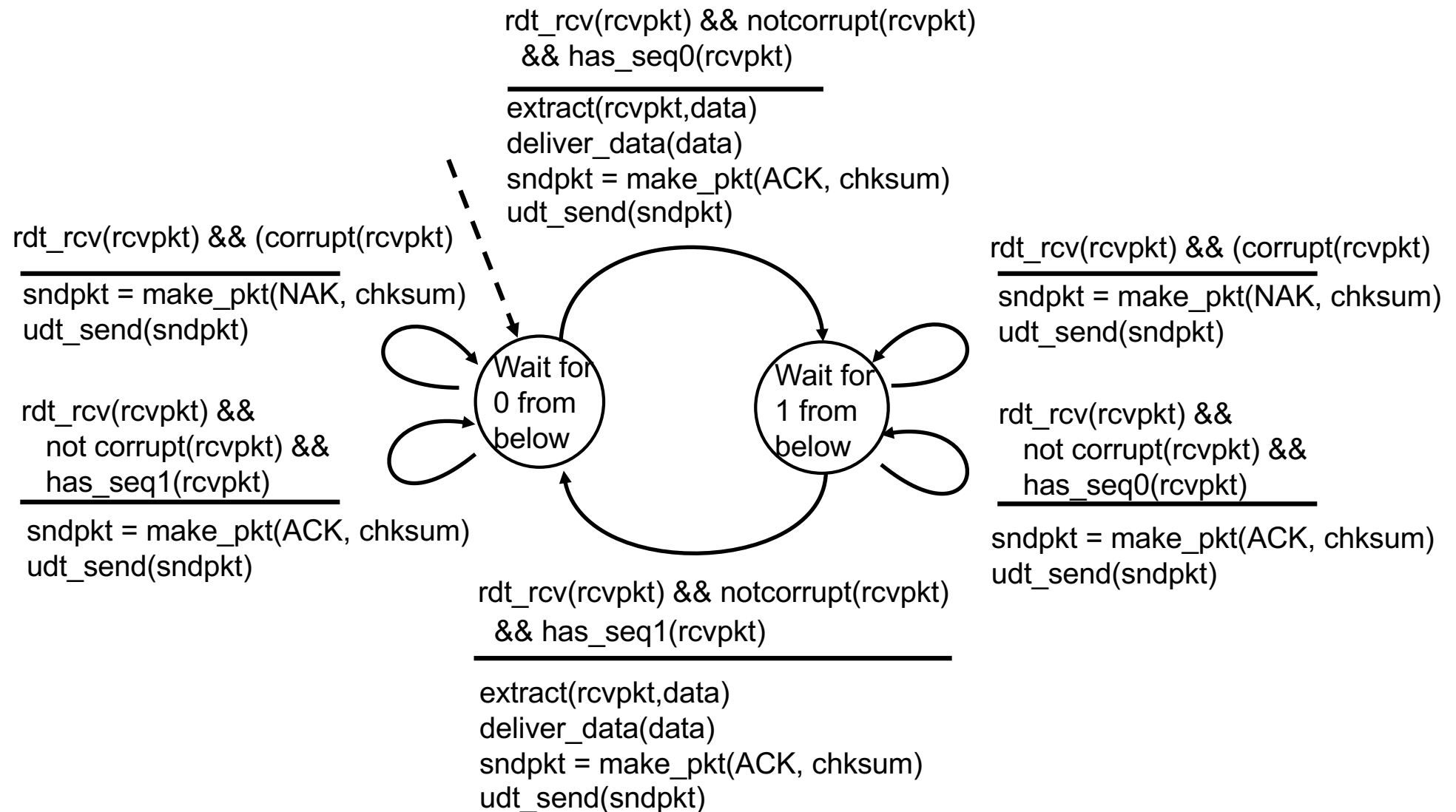


rdt2.1c: Sender, Handles Garbled ACK/NAKs: Using 1 bit (Alternating-Bit Protocol)



rdt2.1c: Receiver, Handles Garbled

ACK/NAKs: Using 1 bit



rdt2.1c: Summary

Sender:

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

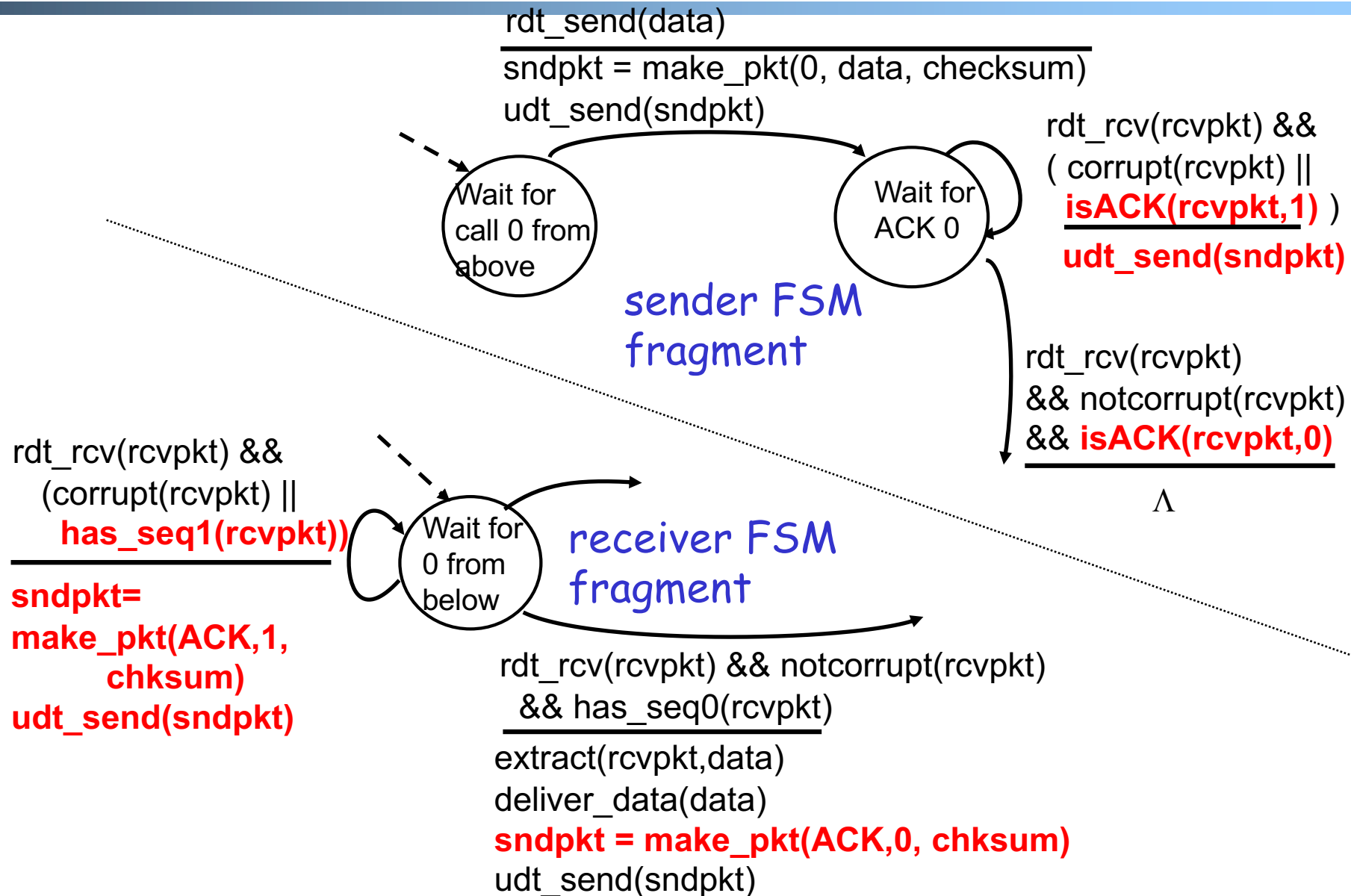
Receiver:

- ❑ must check if received packet is duplicate
 - state indicates whether 0 or 1 is expected pkt seq #

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

- ❑ Admin and review
- 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)

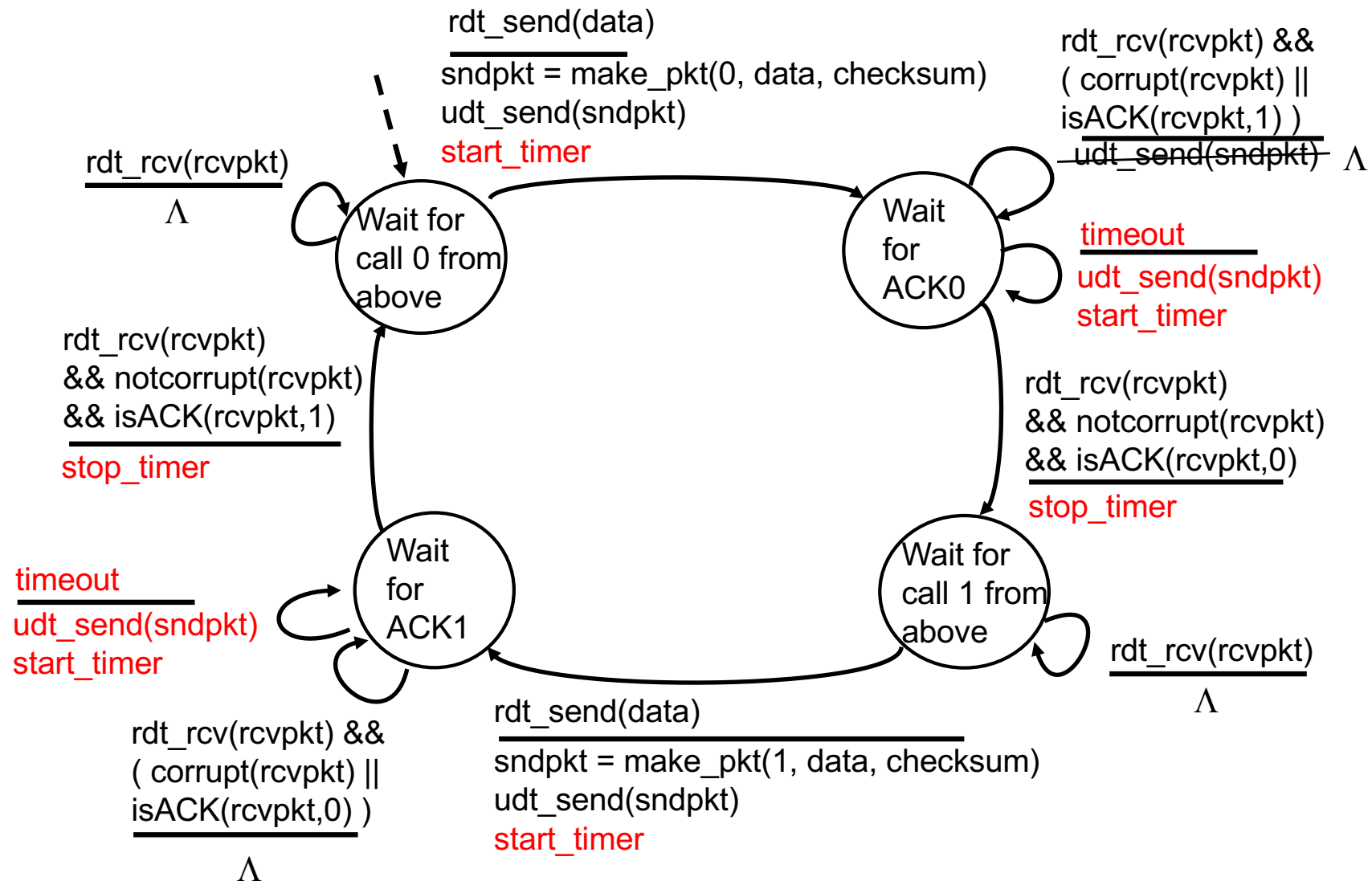
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: Does rdt2.2 work under losses?

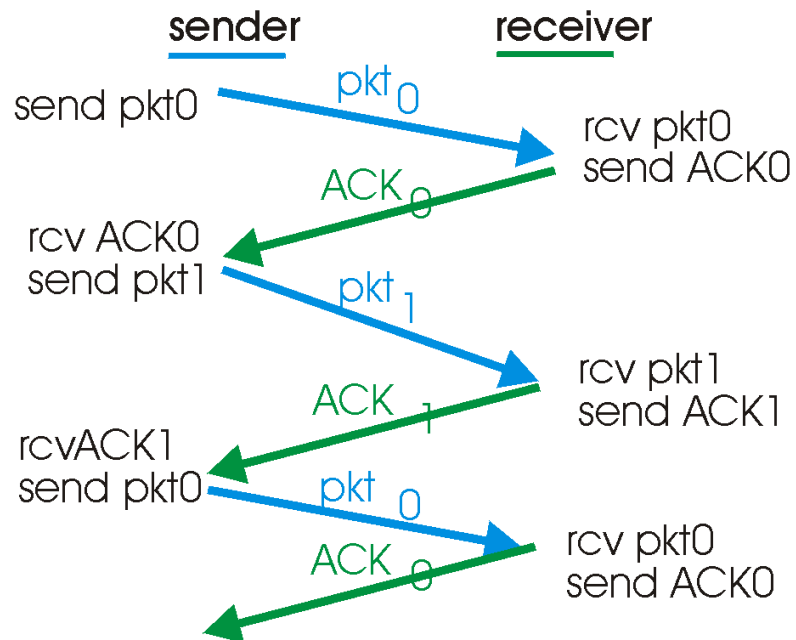
Approach: sender waits “reasonable” amount of time for ACK

- ❑ 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

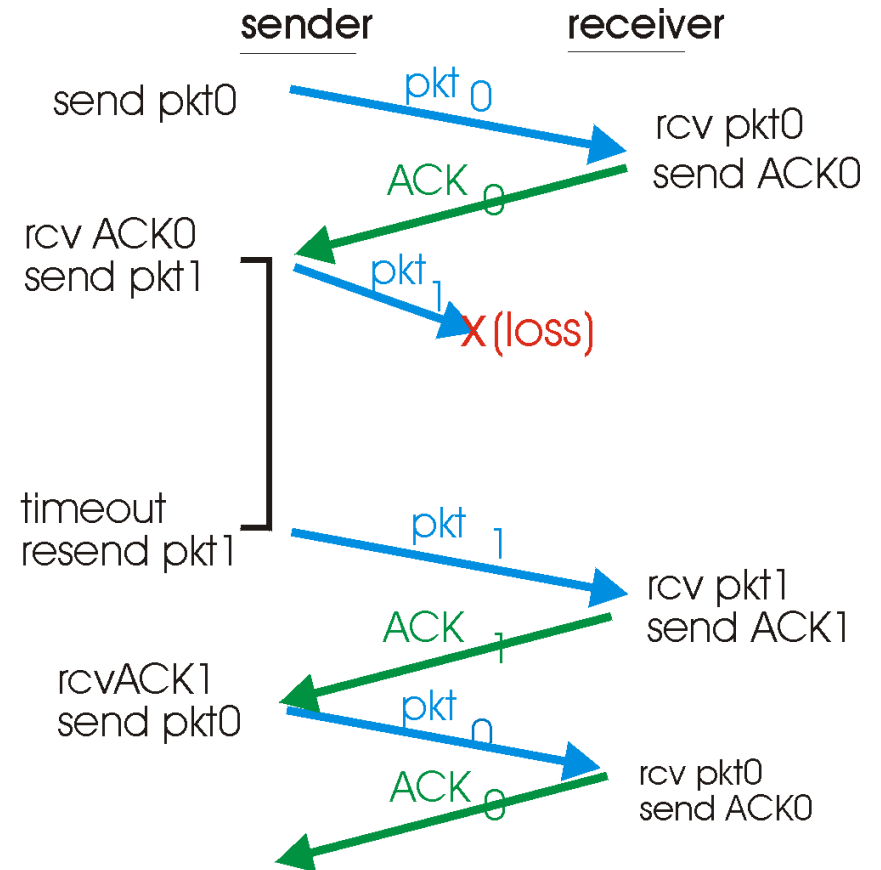
rdt3.0 Sender



rdt3.0 in Action

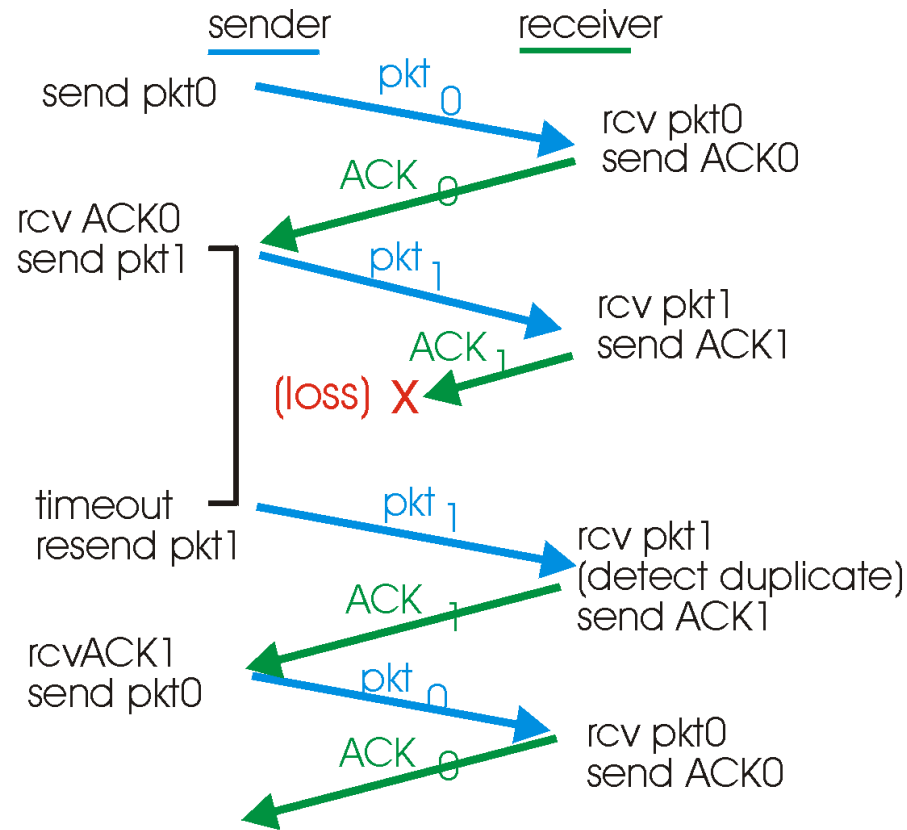


(a) operation with no loss

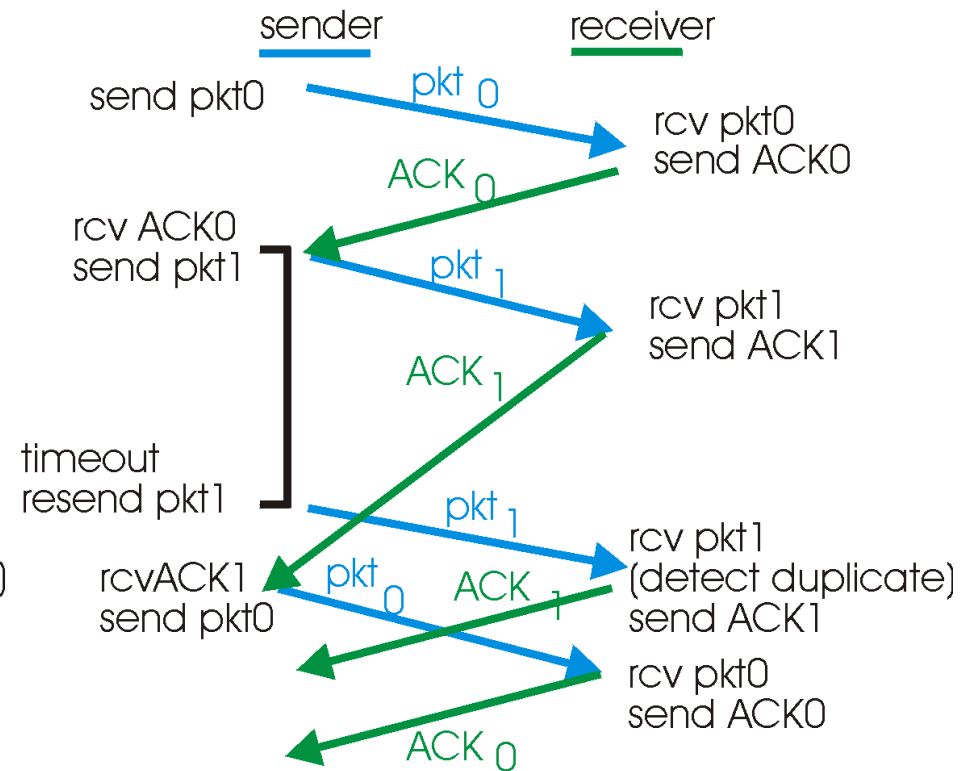


(b) lost packet

rdt3.0 in Action



(c) lost ACK

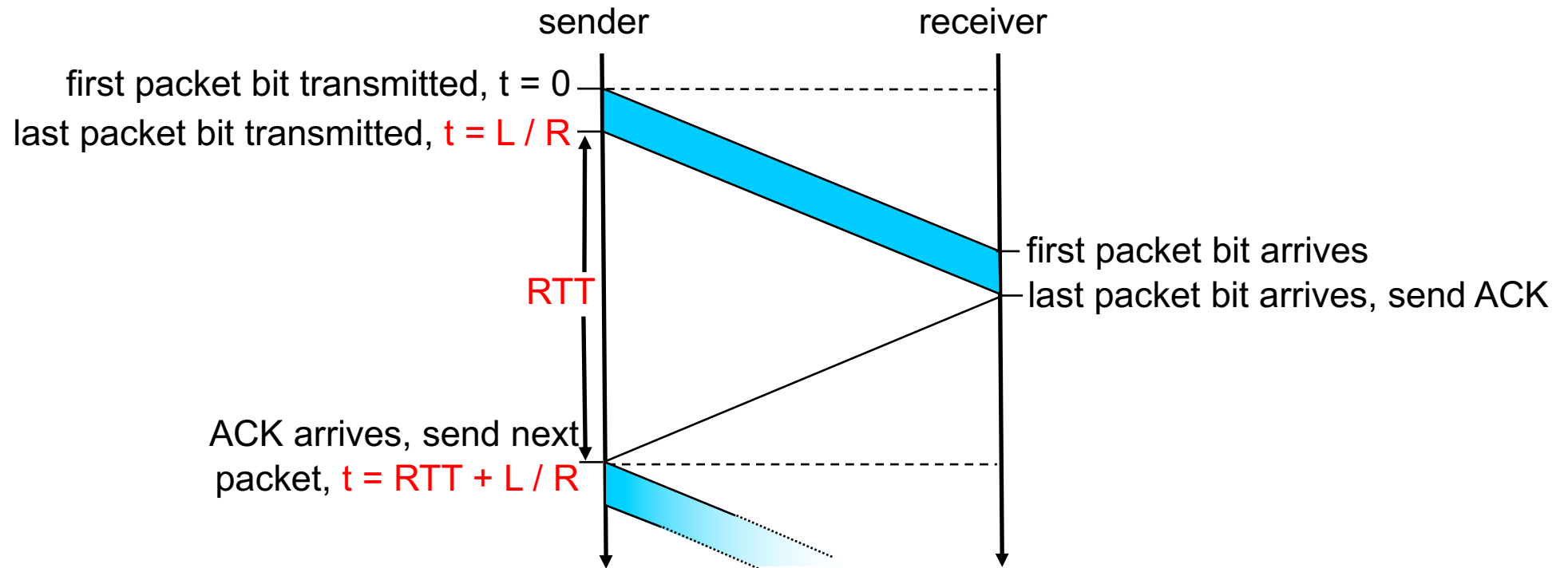


(d) premature timeout

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

Home exercise: What are execution traces of rdt3.0? What are some state invariants of rdt3.0?

rdt3.0: Stop-and-Wait Performance



What is U_{sender} : **utilization** – fraction of time link 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:

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

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

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

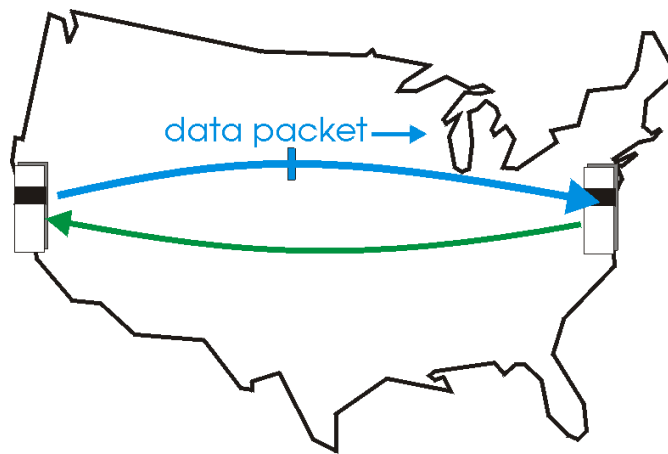
A Summary of Questions

- ❑ How to improve the performance of rdt3.0?
- ❑ What if there are reordering and duplication?
- ❑ How to determine the “right” timeout value?

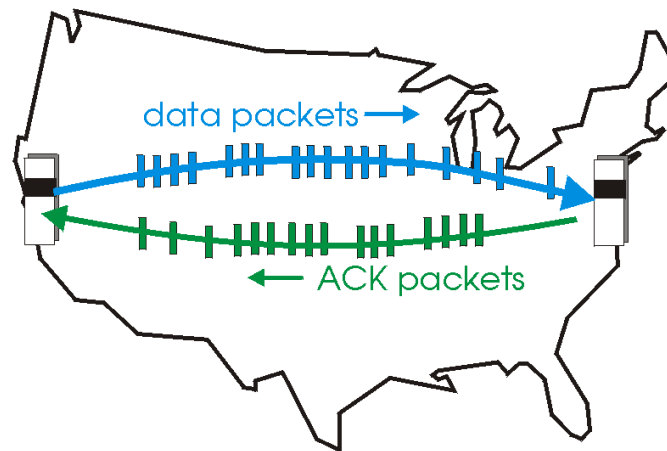
Sliding Window Protocols: Pipelining

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

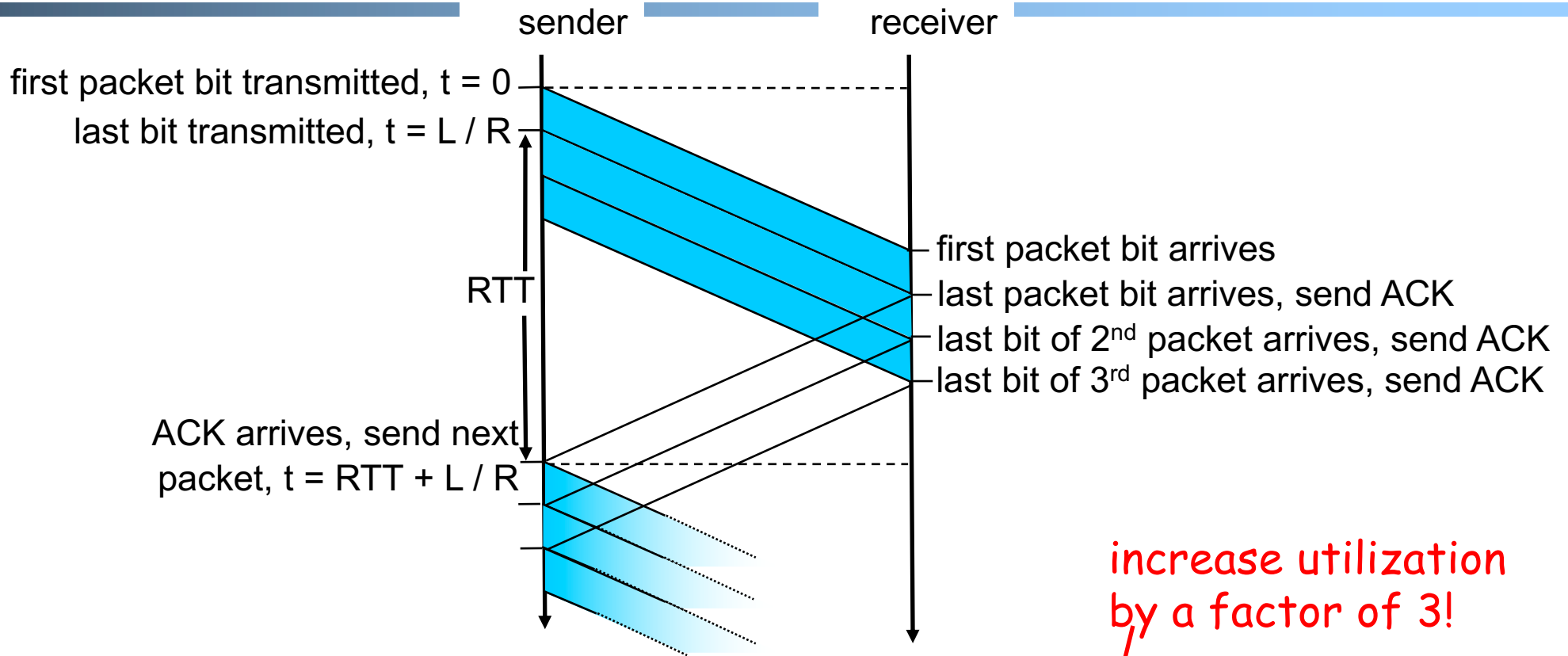


(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

Pipelining: Increased Utilization



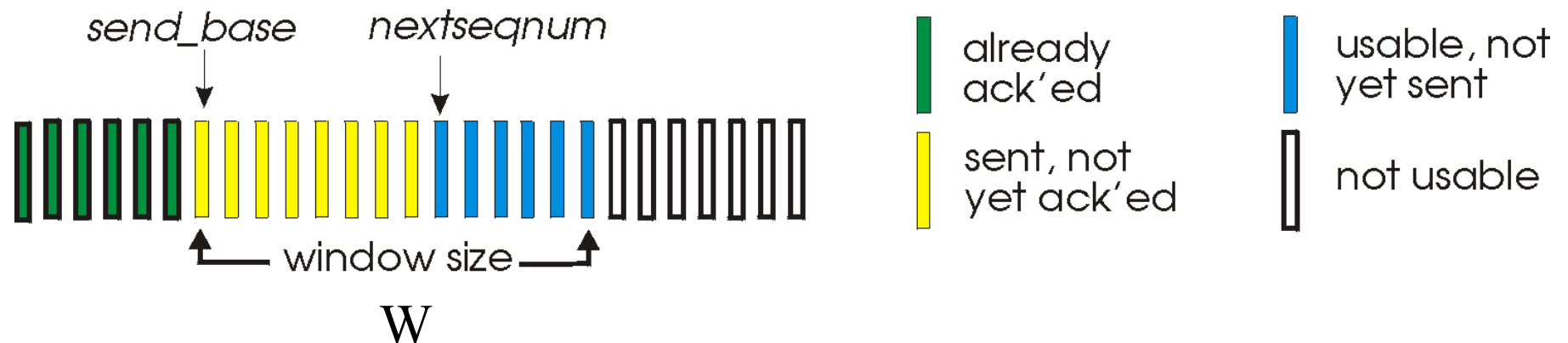
$$U_{\text{sender}} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Question: a rule-of-thumb window size?

Realizing Sliding Window: Go-Back-n

Sender:

- ❑ k-bit seq # in pkt header
- ❑ “window” of up to W , consecutive unack’ed pkts allowed



- ❑ **ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”**
 - note: ACK(n) could mean two things: I have received **upto and include** n, or I am waiting for n
- ❑ timer for the packet at base
- ❑ *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

GBN: Sender FSM

