Network Transport Layer: Reliable Data Transfer

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf22/index.shtml

10/20/2022

This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Outline

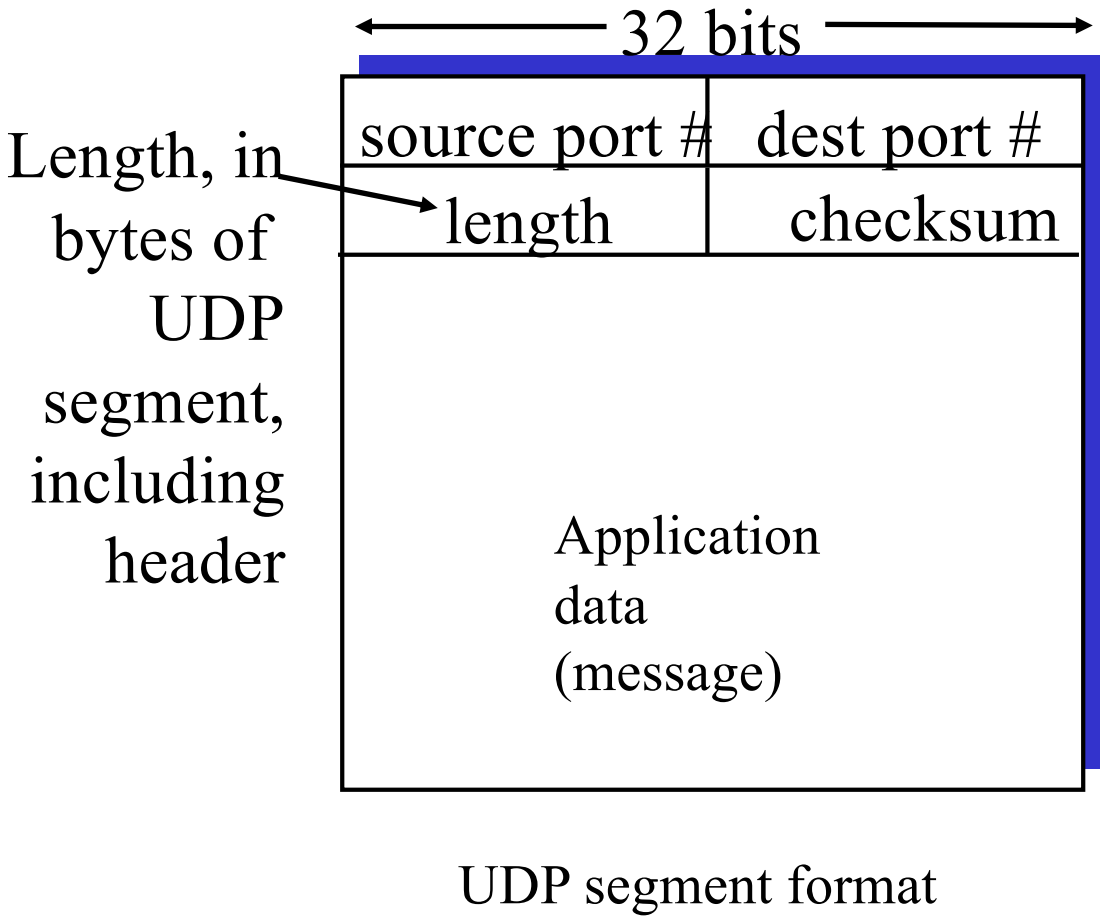
- Admin and recap
- Reliable data transfer

Recap

- Applications
  - Client-server applications
    - Single server
    - Multiple servers load balancing
  - Application overlays (distributed network applications) to
    - scale bandwidth/resource (BitTorrent)
    - distribute content lookup (Freenet, DHT, Chord) [optional]
    - distribute content verification (Block chain) [optional]
    - achieve anonymity (Tor) [optional]

UDP: 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:

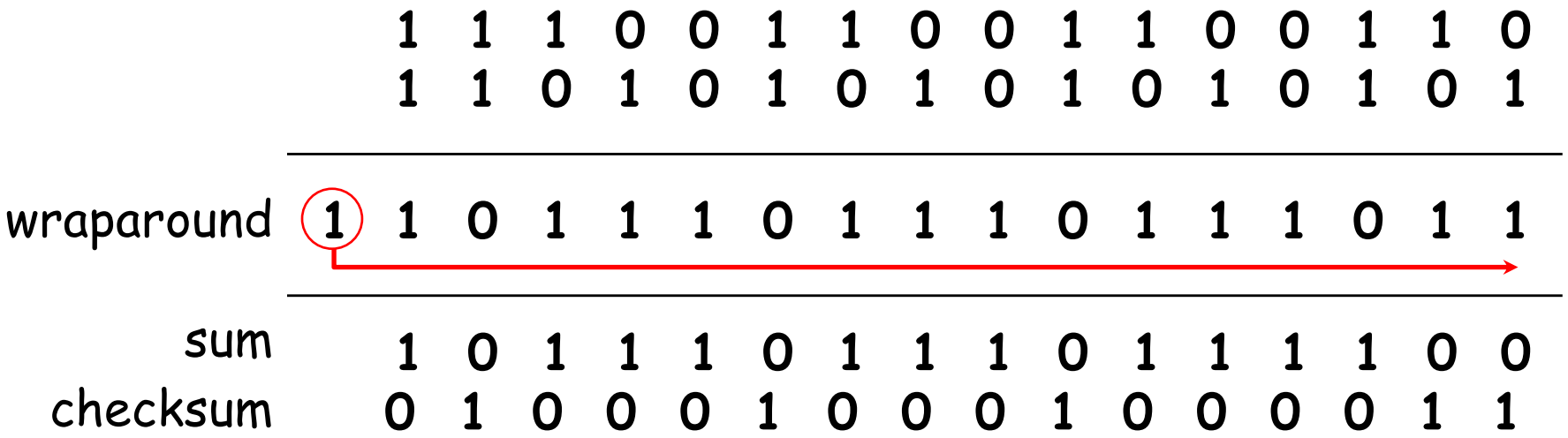
- treat segment contents as sequence of 16-bit integers
- checksum: addition of segment contents to be zero
- sender puts checksum value into UDP checksum field

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:



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

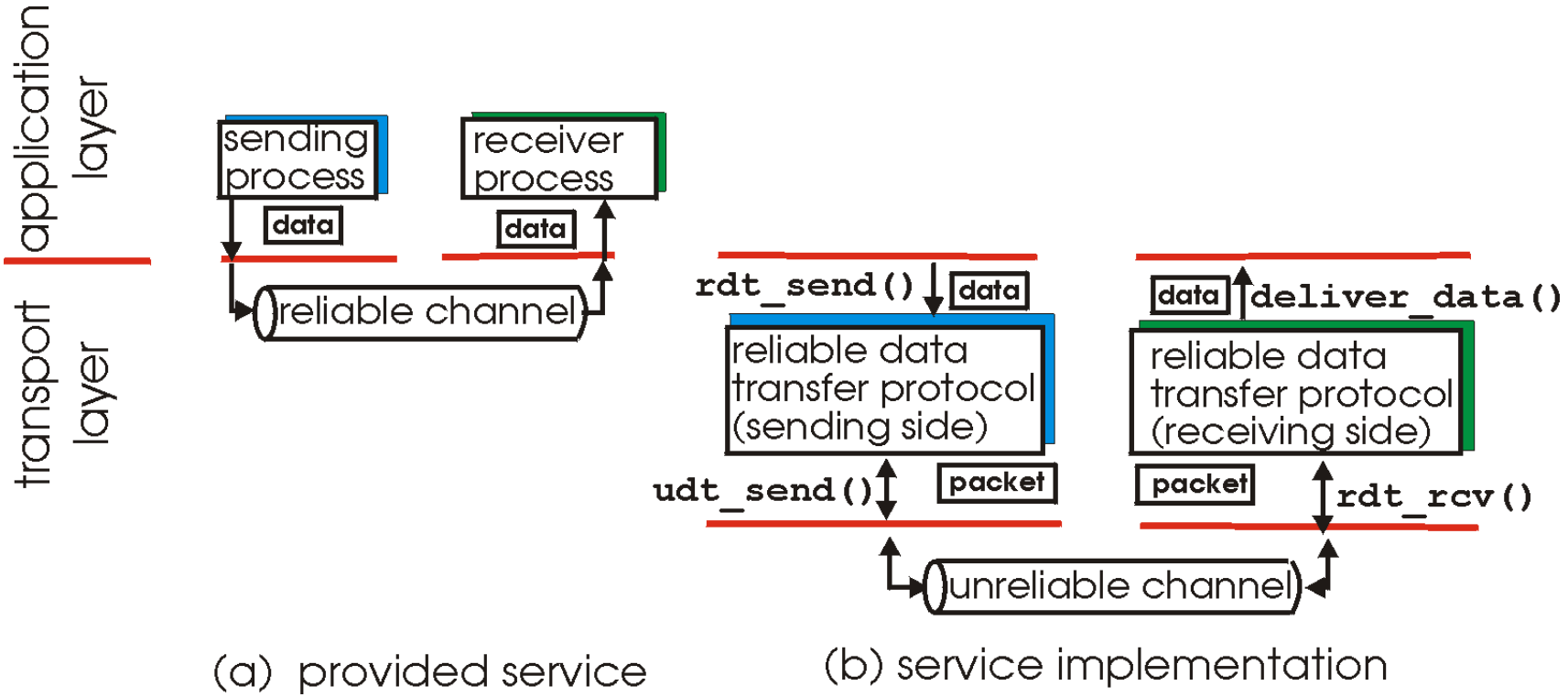
Outline

- Admin and recap
- *Reliable data transfer*

Principles of Reliable Data Transfer (RDT)

- Important in app., transport, link layers
- Foundation to other protocols
- We use the development of RDT to also better appreciate understanding distributed protocols

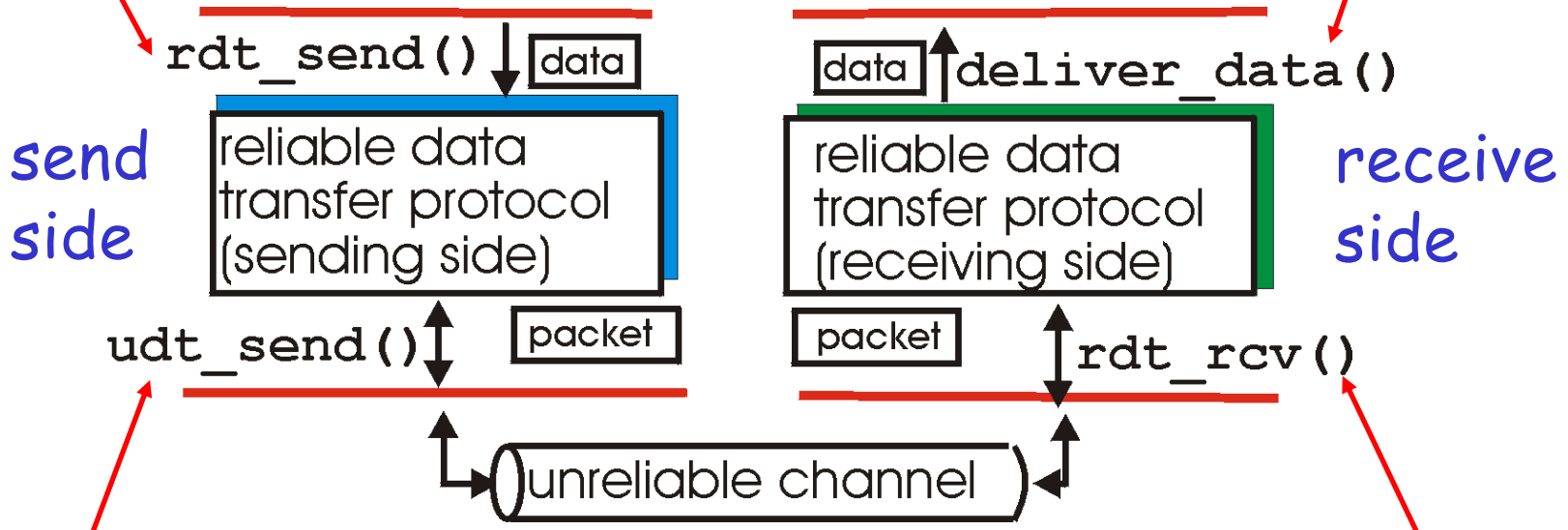
Reliable Data Transfer



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

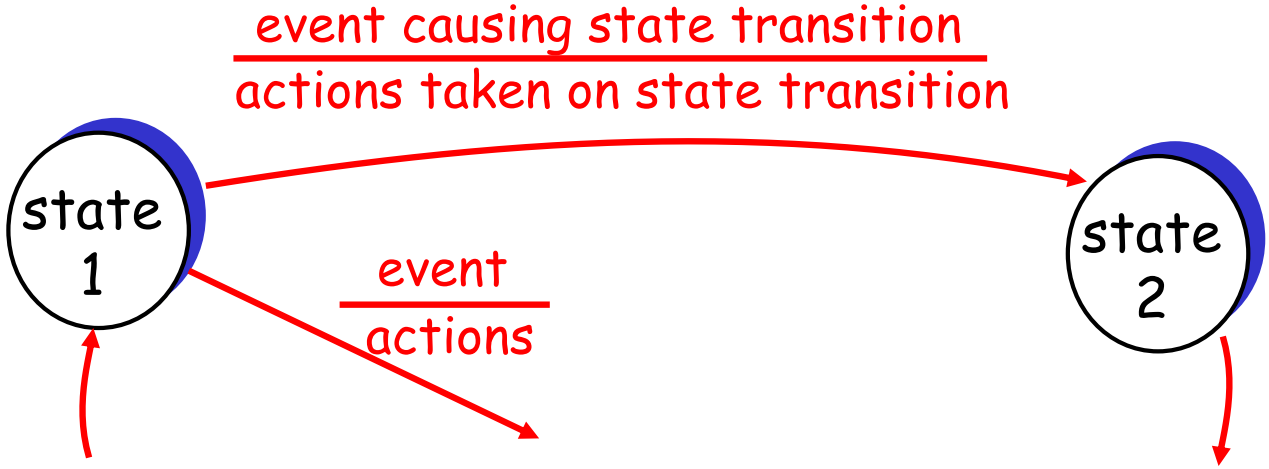
rdt_rcv(): called from below; when packet arrives on rcv-side of channel

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

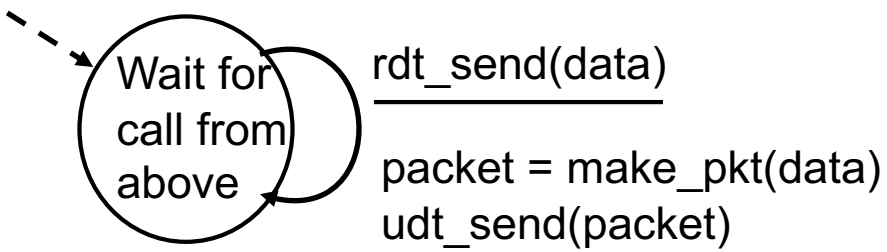


Outline

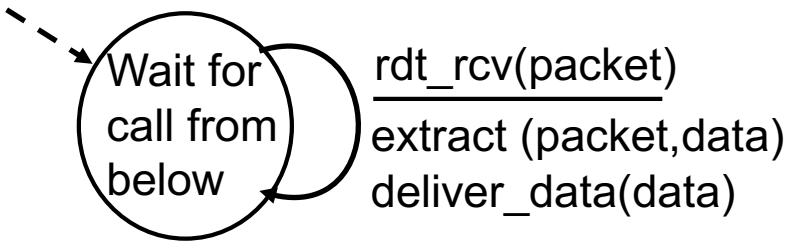
- Admin and review
- Overview of transport layer
- UDP and error checking
- Reliable data transfer
  - *perfect channel*

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

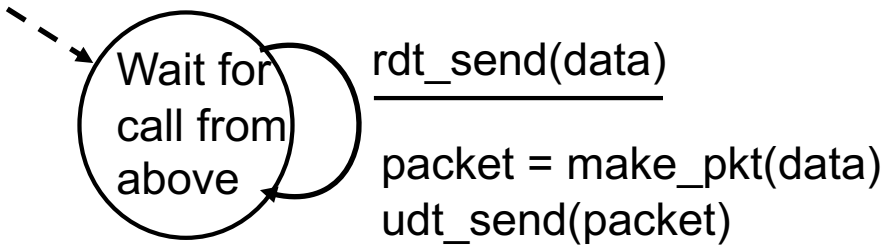
Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).

Outline

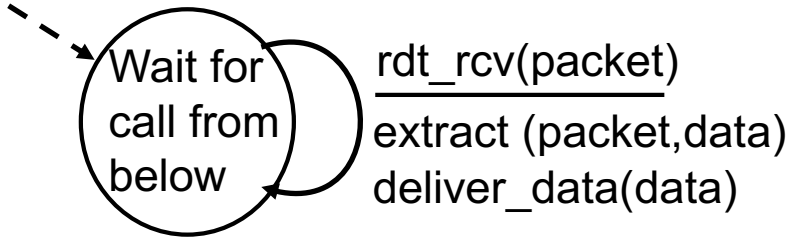
- Admin and recap
- Overview of transport layer
- UDP and error checking
- Reliable data transfer
  - perfect channel
  - *channel with bit errors*

rdt2.0: Channel With Bit Errors

- Assume: Underlying channel may only flip bits in packet



sender



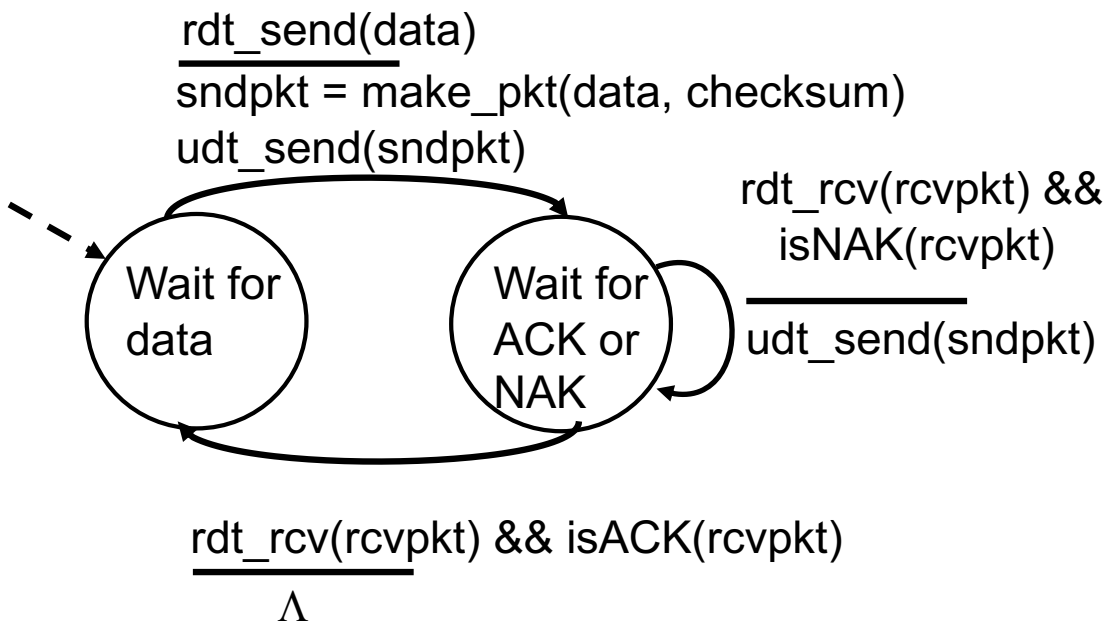
receiver

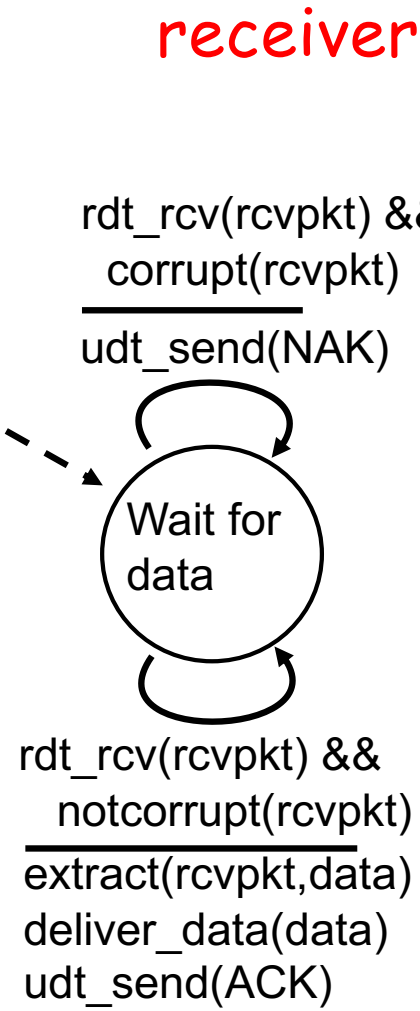
Exercise: What correctness requirement(s) rdt1.0 cannot provide?

rdt2.0: Channel With Bit Errors

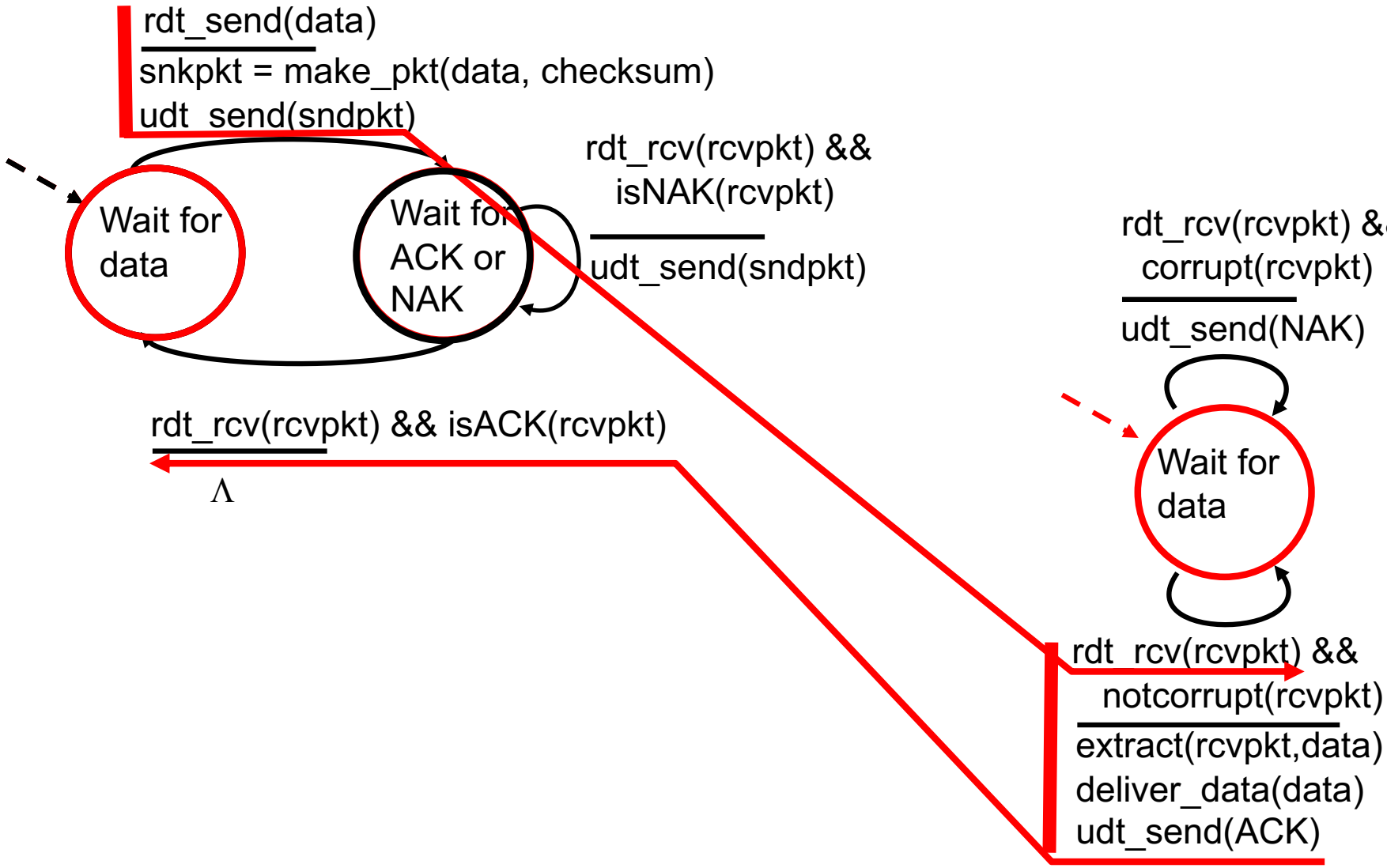
- New mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - receiver error detection: recall: UDP checksum/Ethernet CRC detects bit errors
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender
    - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
    - *negative acknowledgements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

rdt2.0: FSM Specification

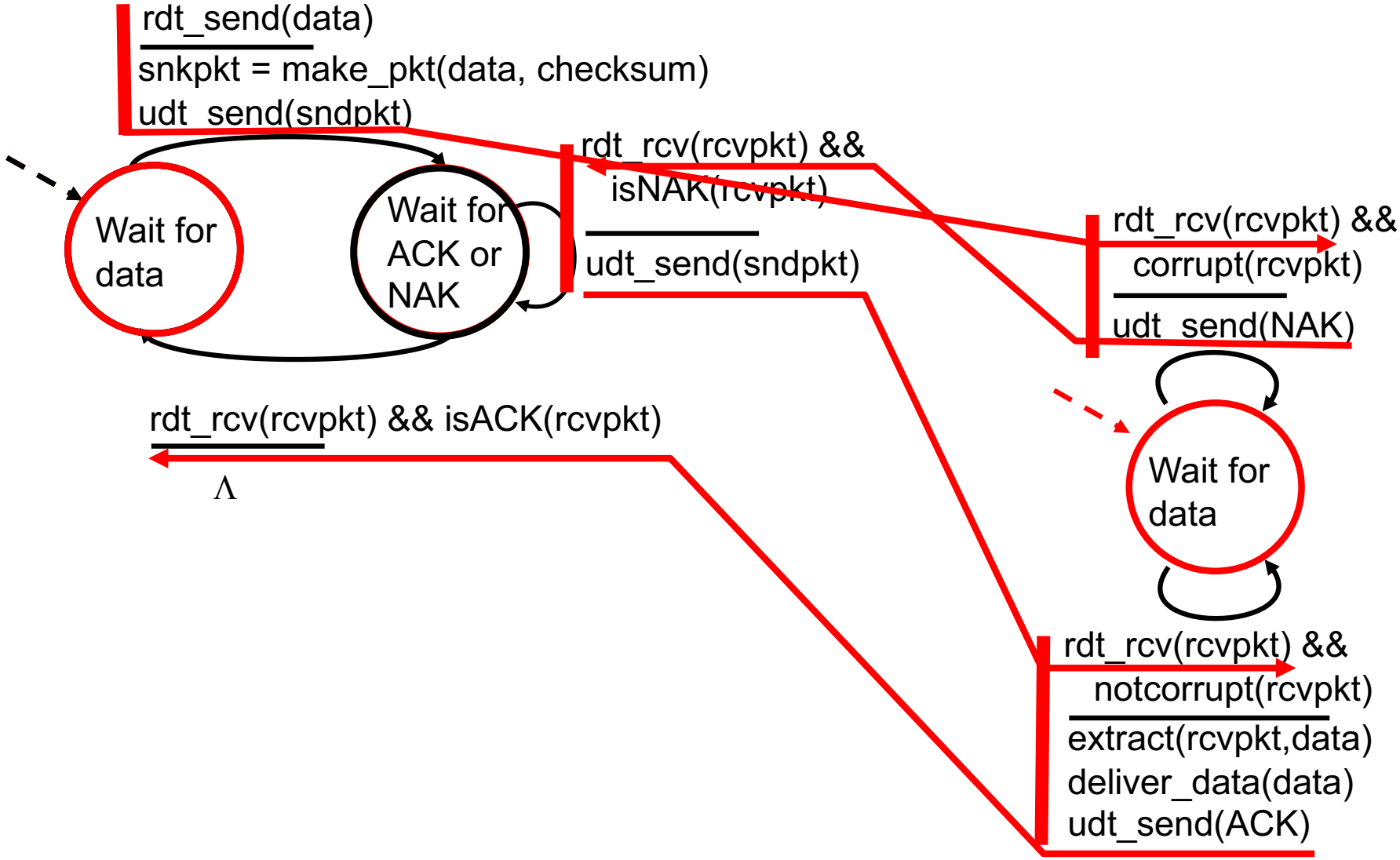




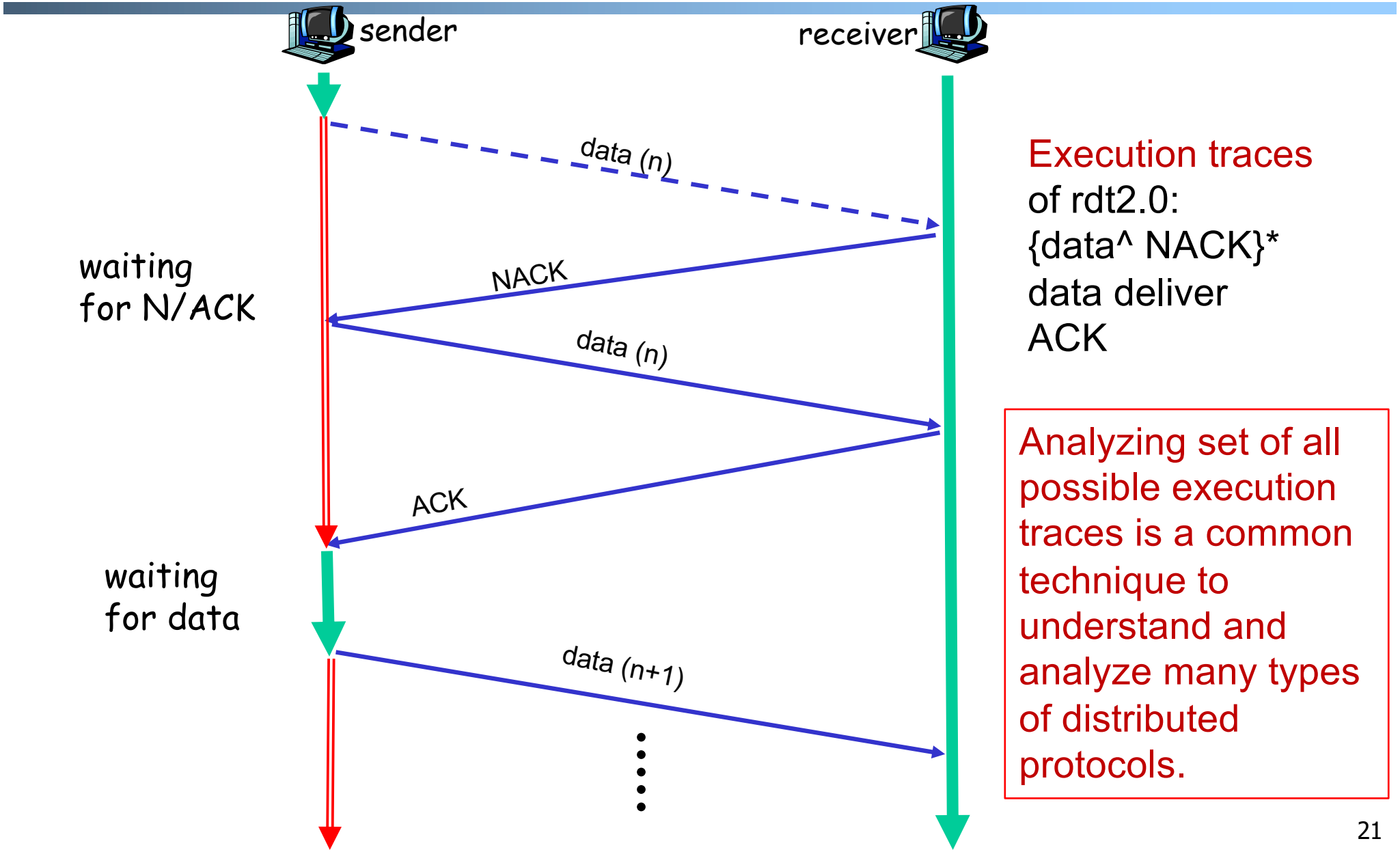
rdt2.0: Operation with No Errors



rdt2.0: Error Scenario



Rdt2.0 Analysis



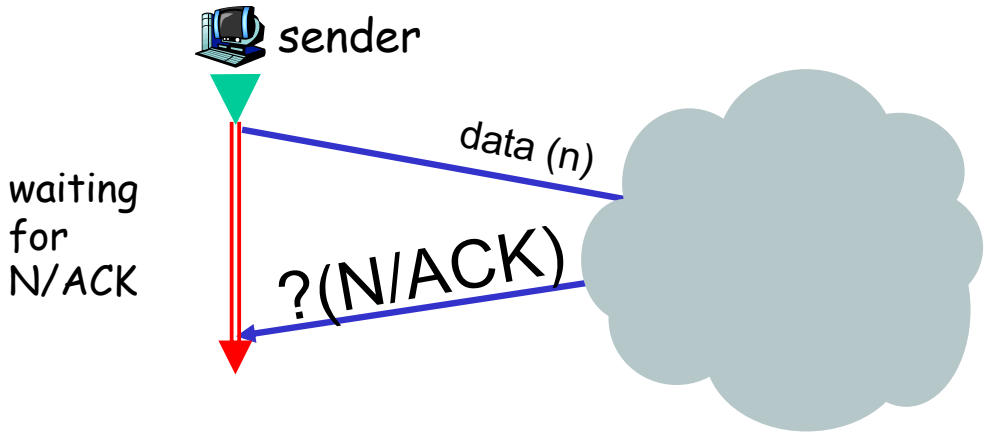
Execution traces of rdt2.0: {data^ NACK}* data deliver ACK

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

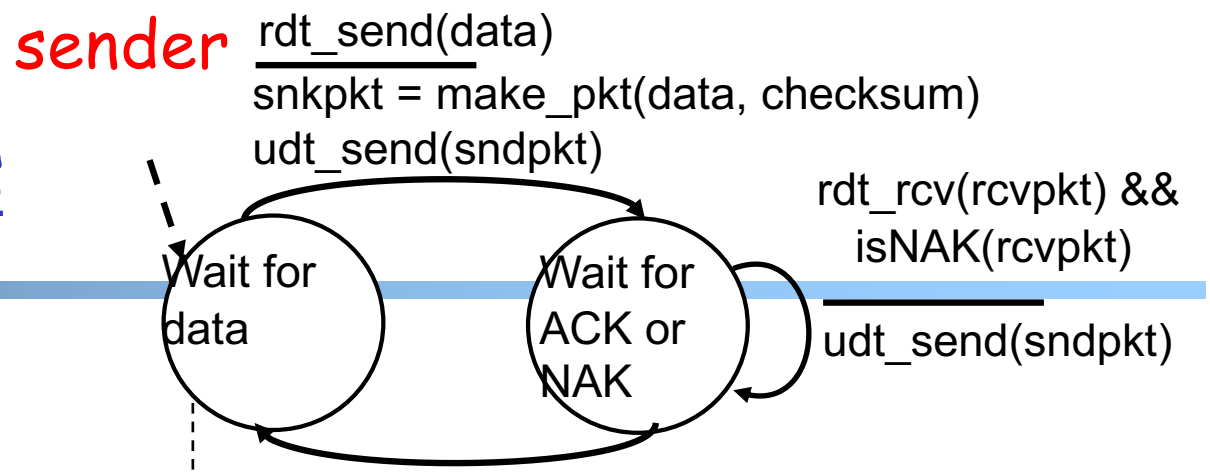
rdt2.0 is Incomplete!

What happens if ACK/NAK corrupted?

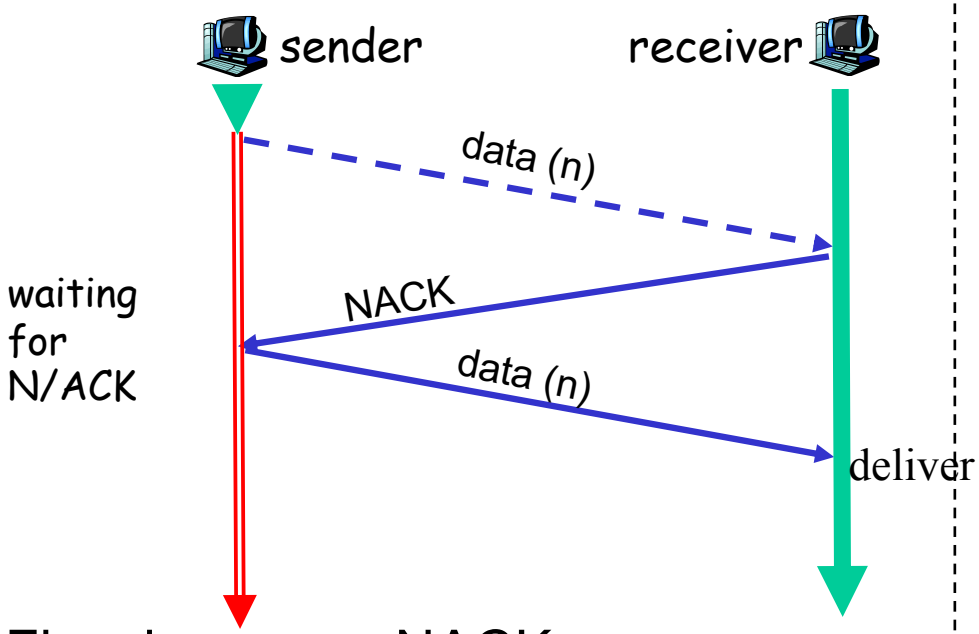
- Although sender receives feedback, but doesn't know what happened at receiver!



Two Possibilities

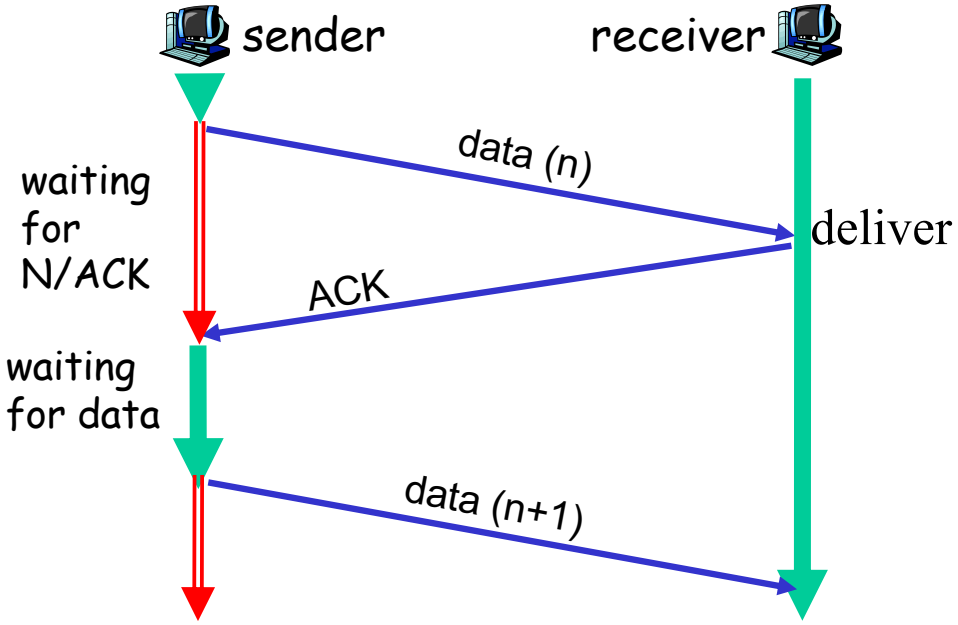


sender can't just guess NACK: if wrong, duplicate



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

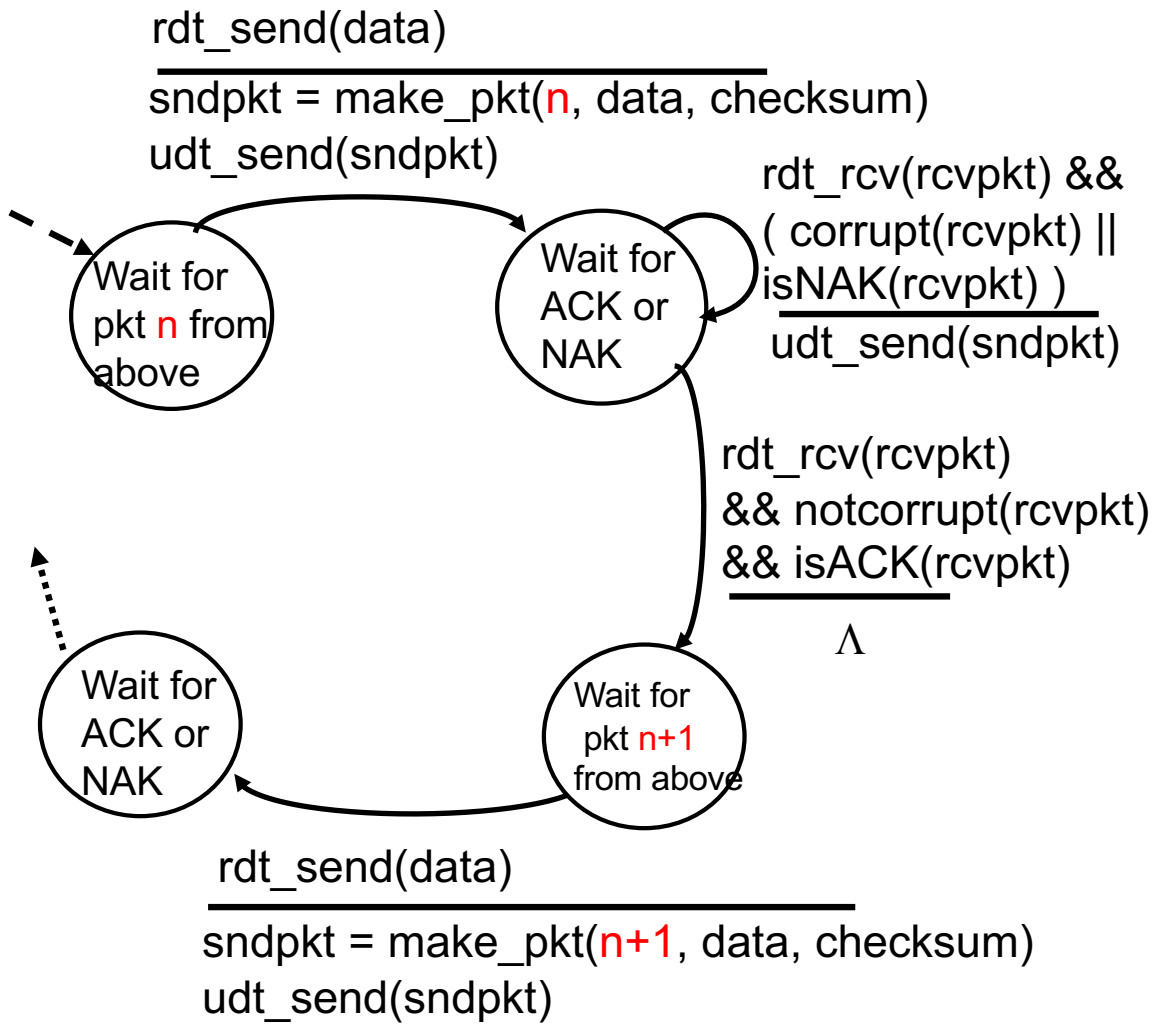
Handling ambiguity:

- 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

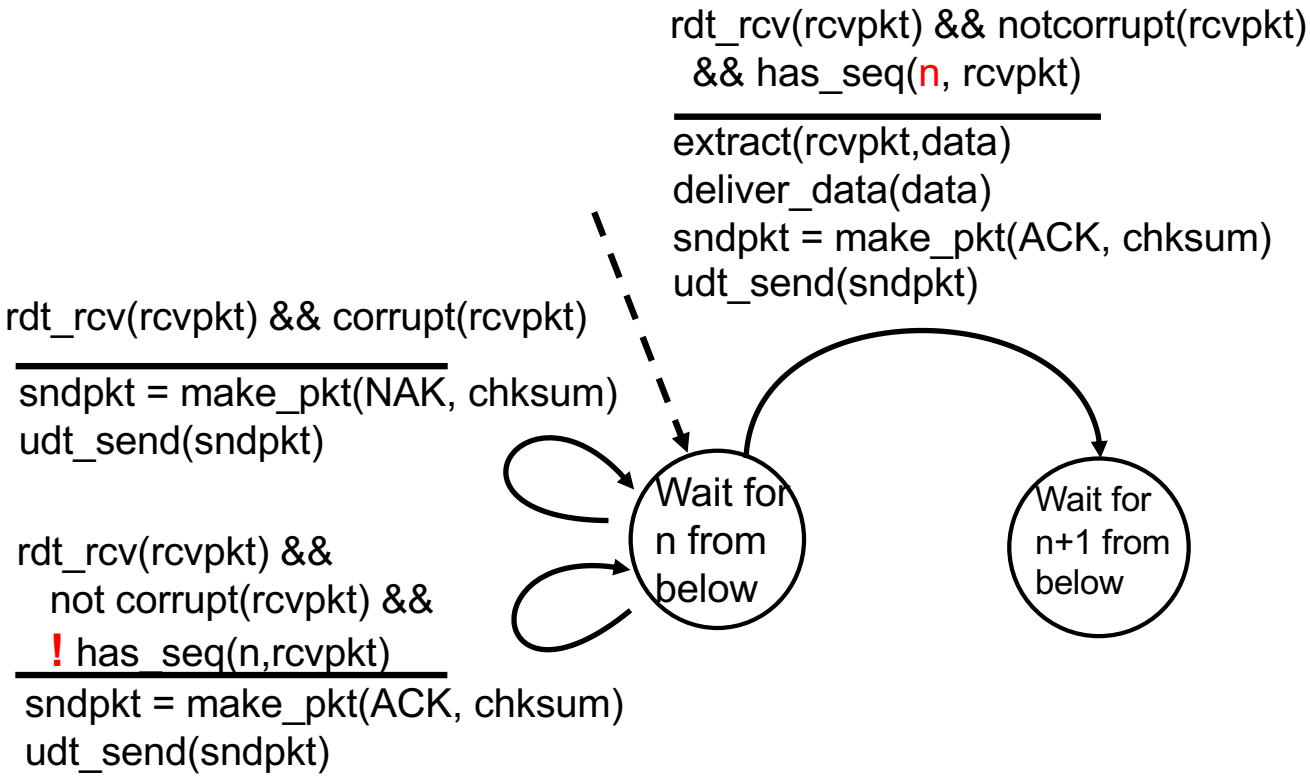
stop and wait
sender sends one packet, then waits for receiver response

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

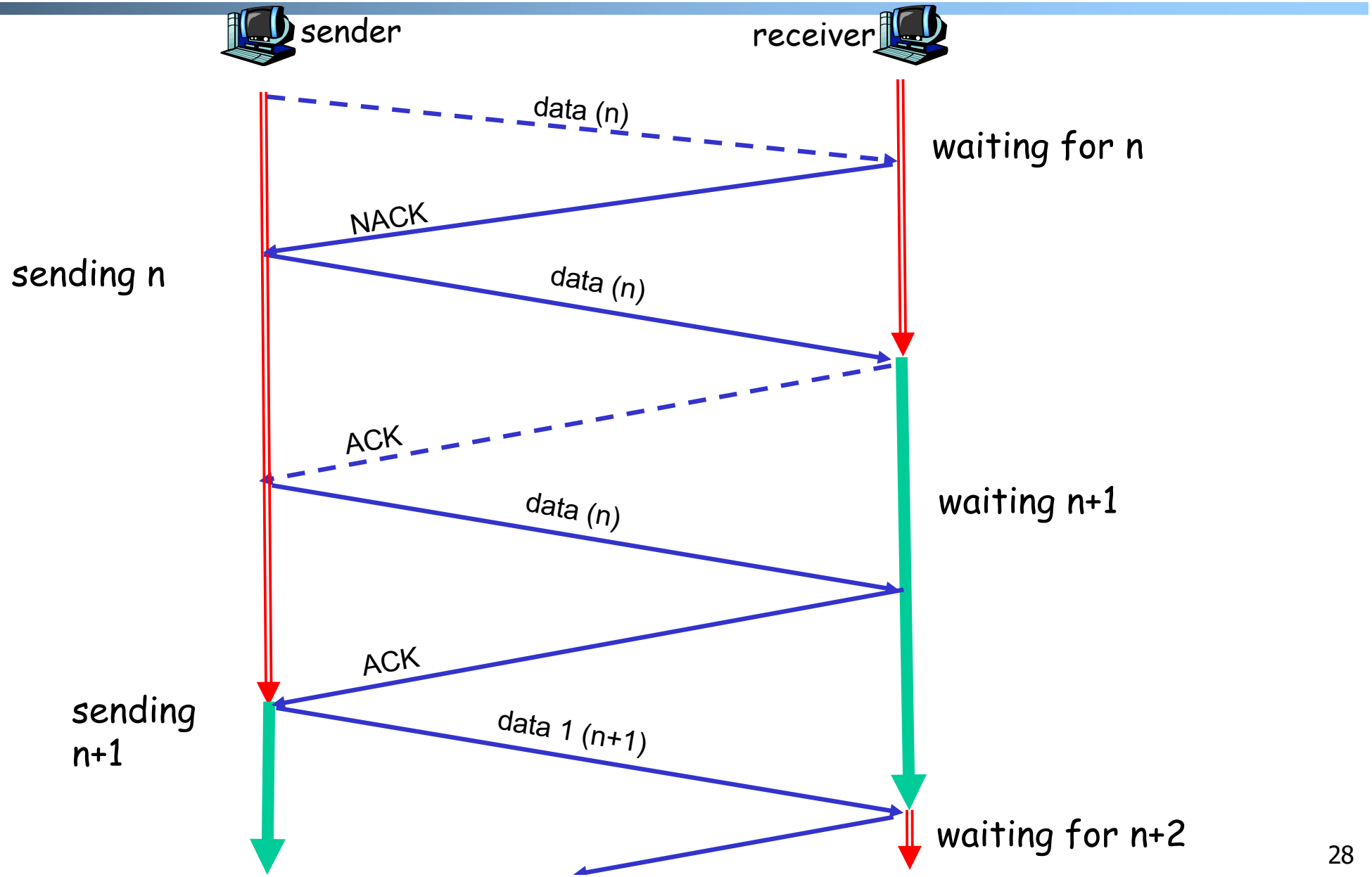
Sender:

- seq # added to pkt
- must check if received ACK/NAK corrupted

Receiver:

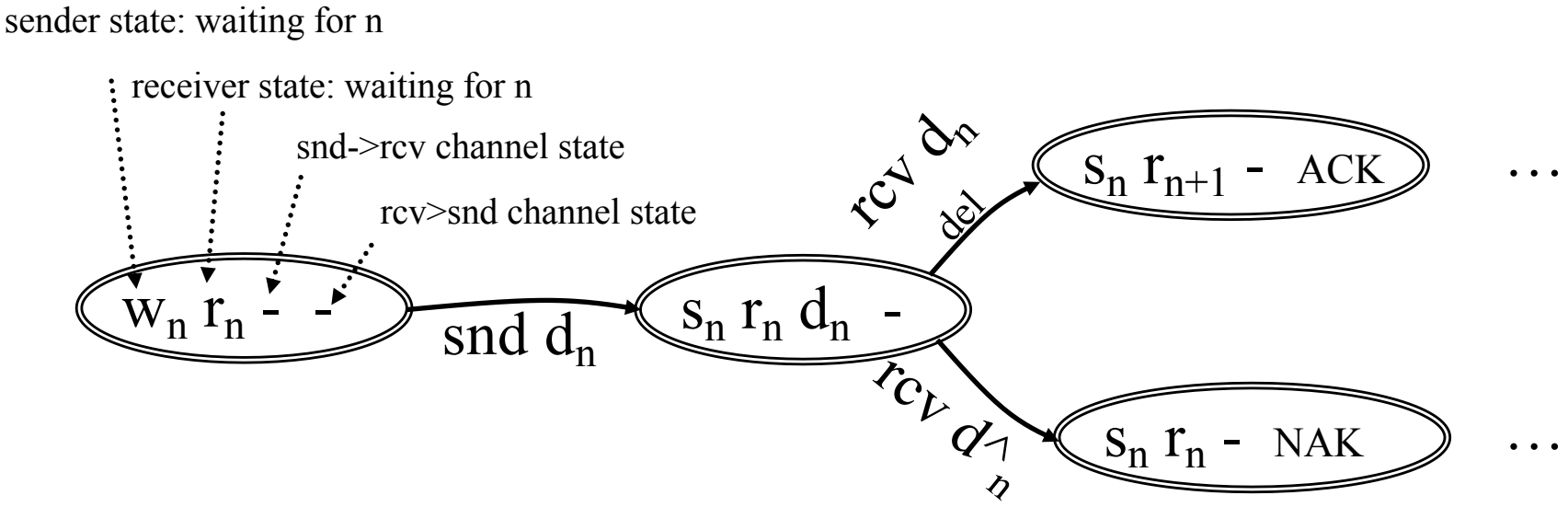
- must check if received packet is duplicate
  - by checking if the packet has the expected pkt seq #

rdt2.1b Analysis: Execution Traces?

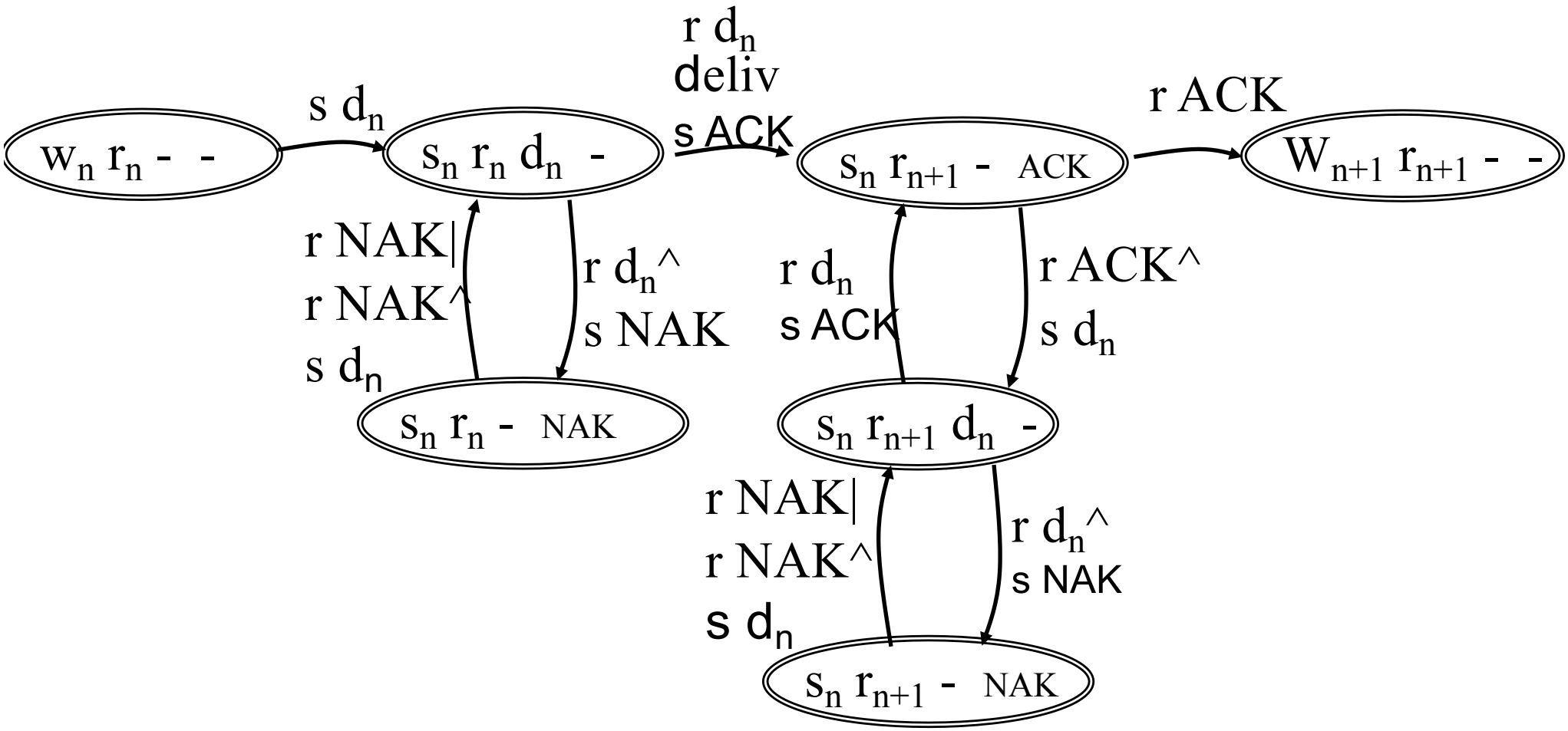


Protocol Analysis using (Generic) Execution Traces Technique

- 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

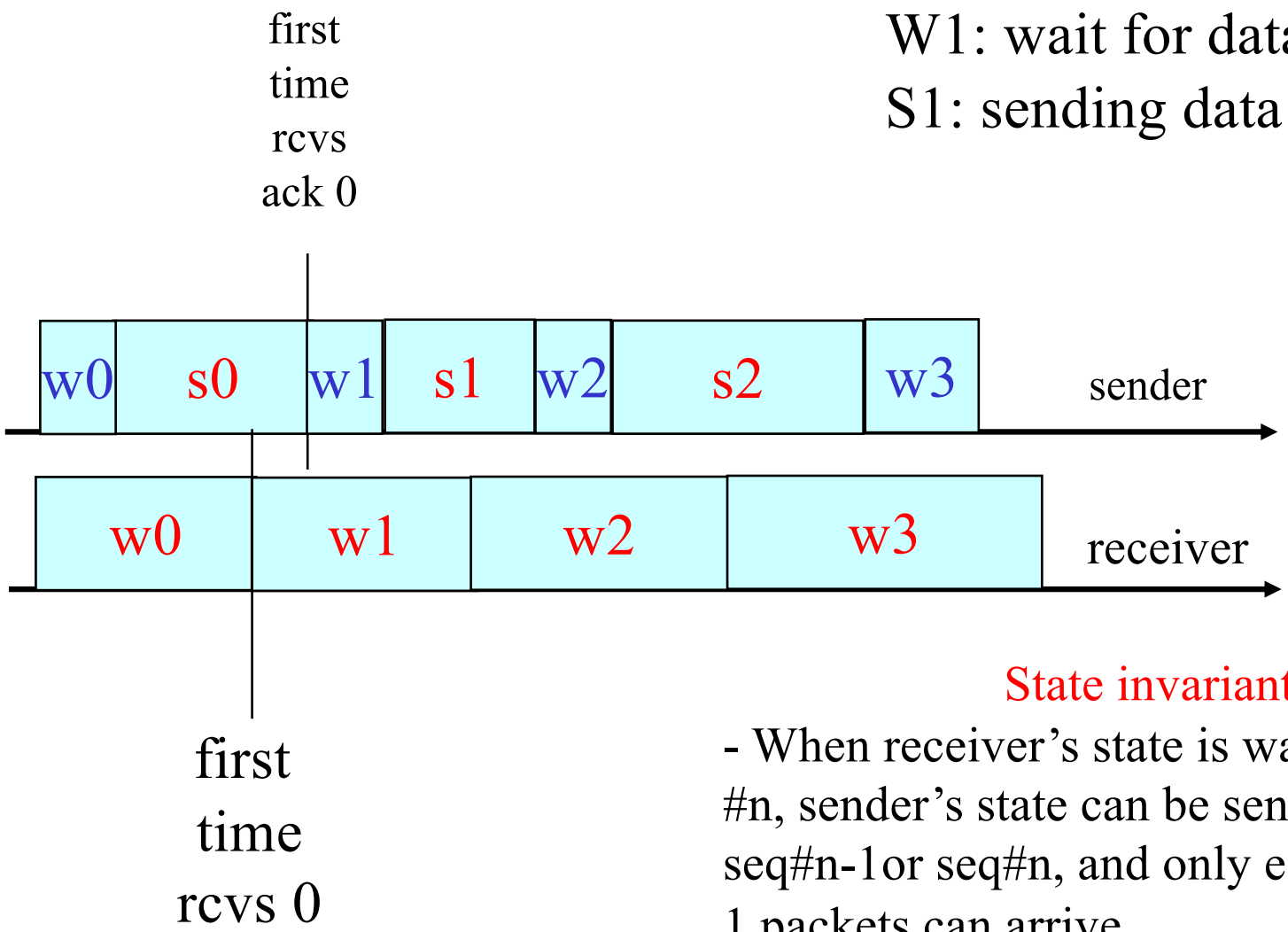


Recap: Protocol Analysis using (Generic) Execution Traces Technique



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

rdt2.1b Analysis: State Invariants



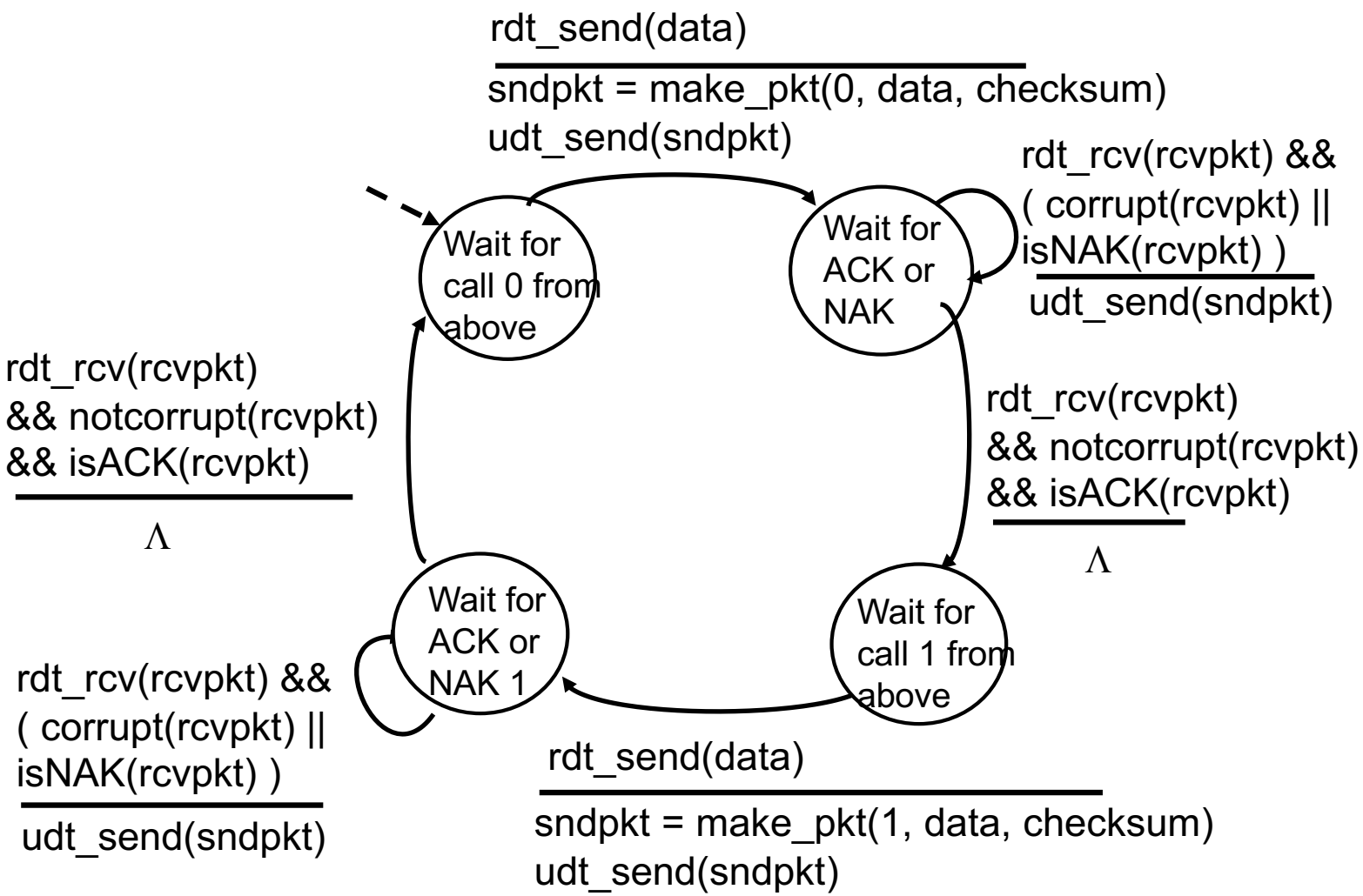
W1: wait for data with seq. 1

S1: sending data with seq. 1

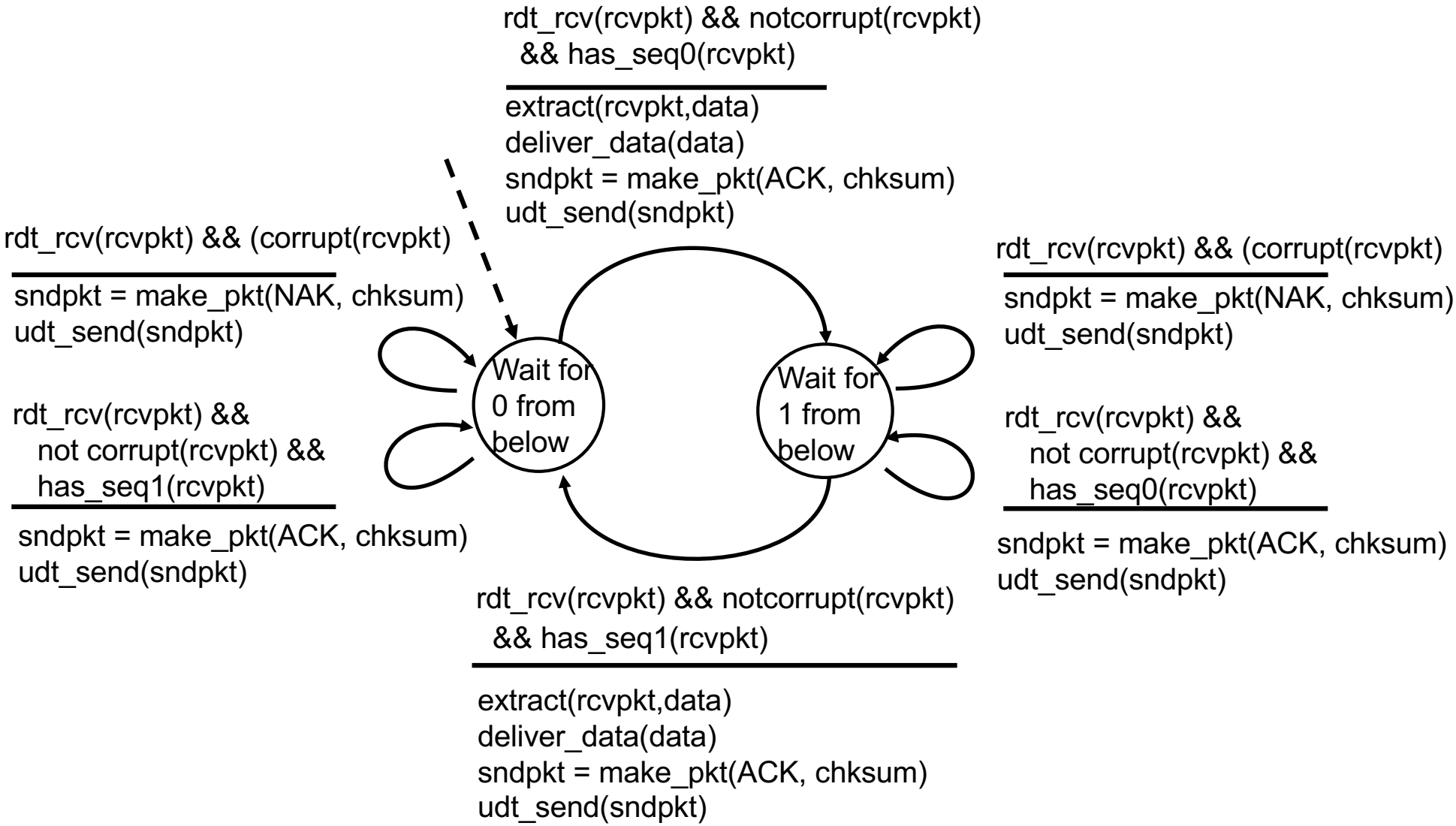
State invariant:

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

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. #

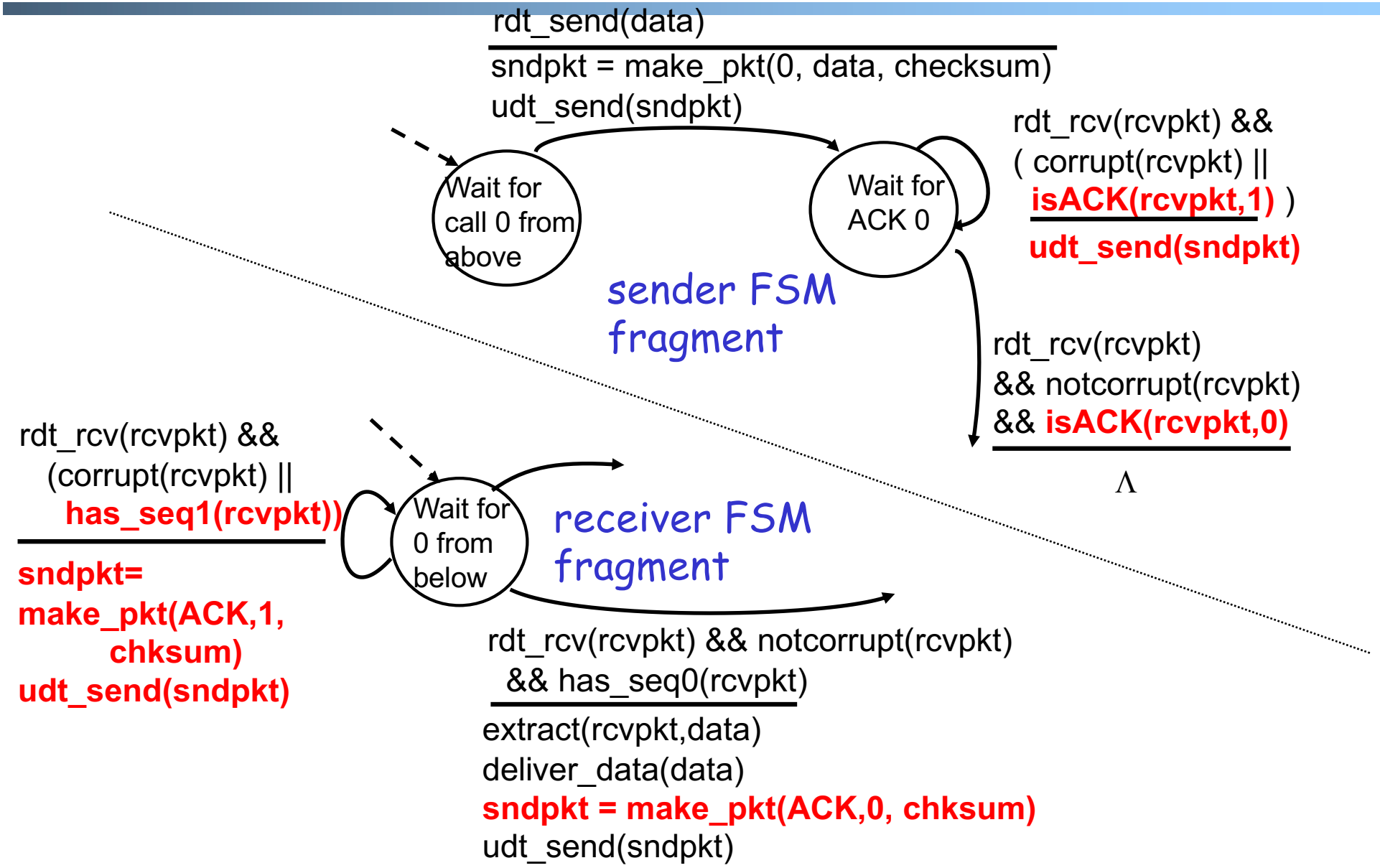
Receiver:

- 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)

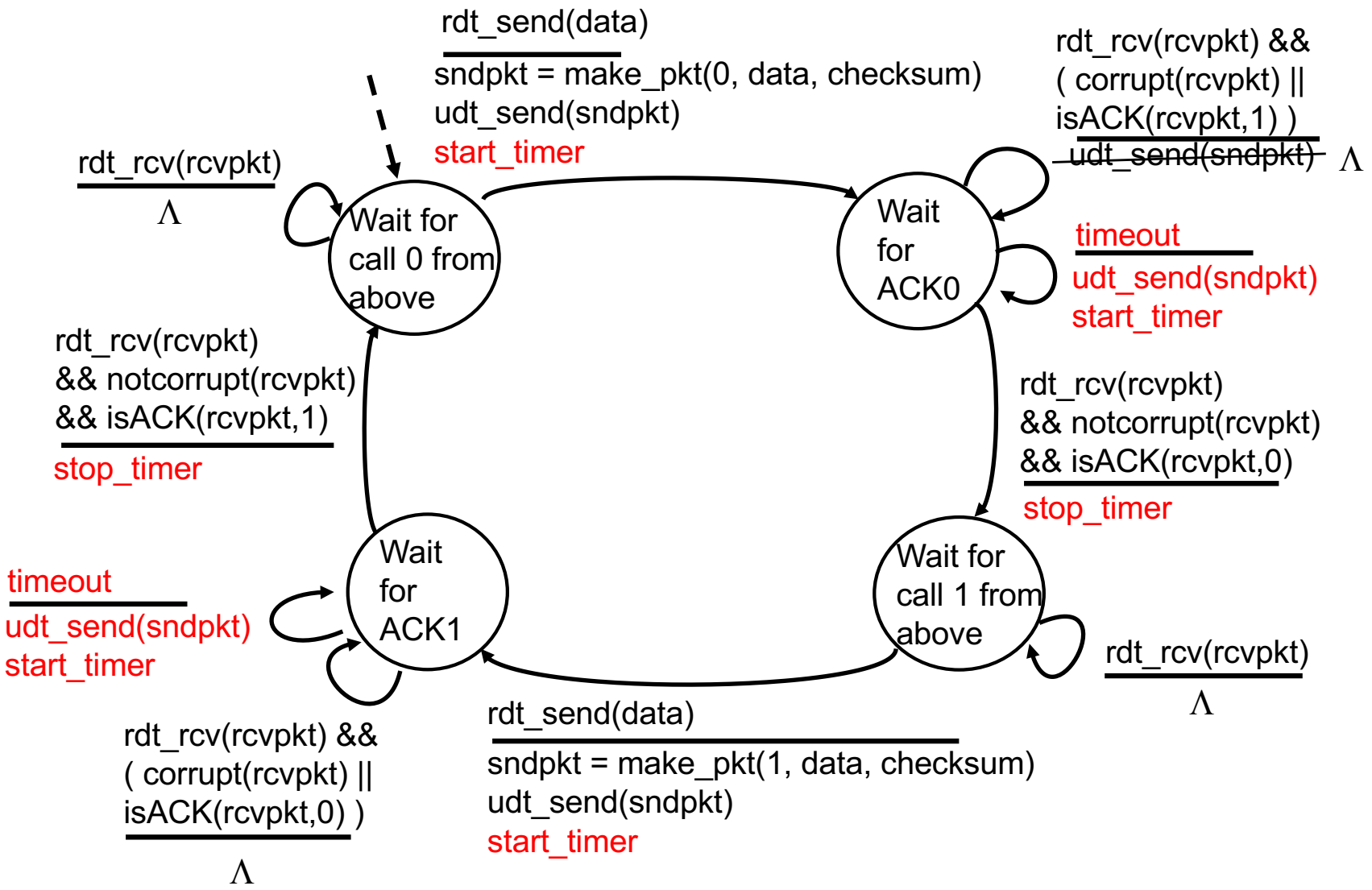
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: Does rdt2.2 work under losses?

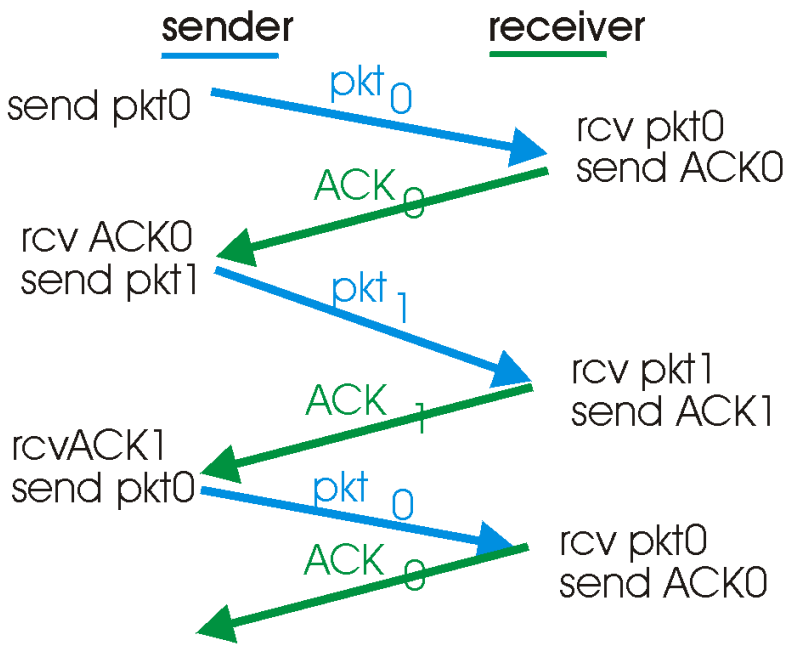
Approach: sender waits "reasonable" amount of time for ACK

- 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

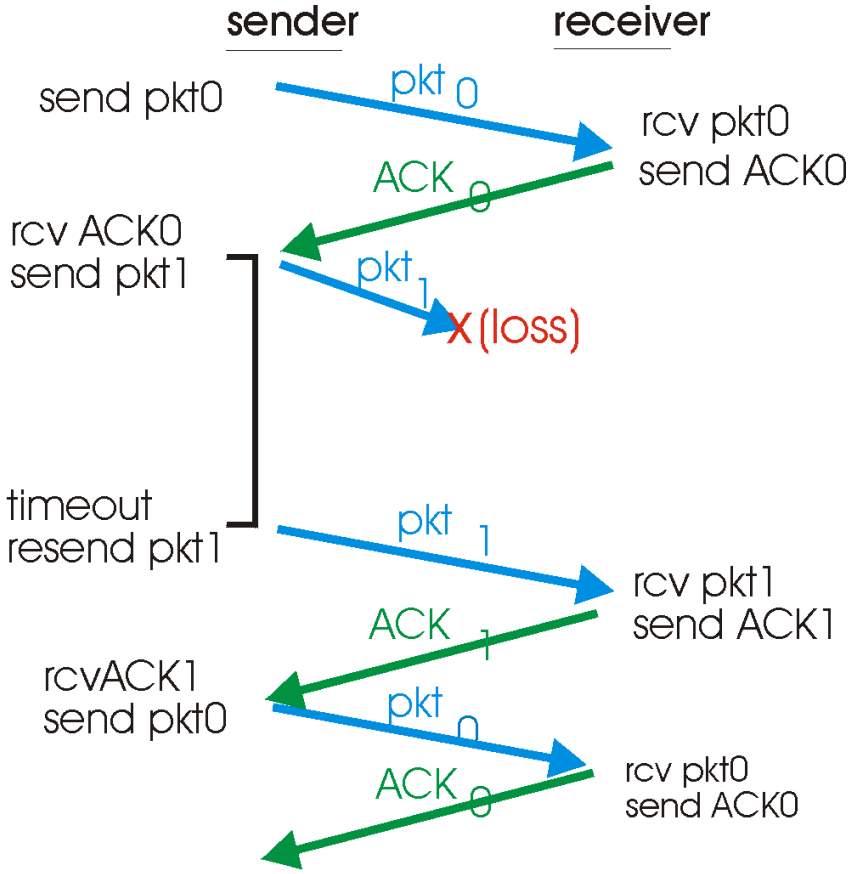
rdt3.0 Sender



rdt3.0 in Action

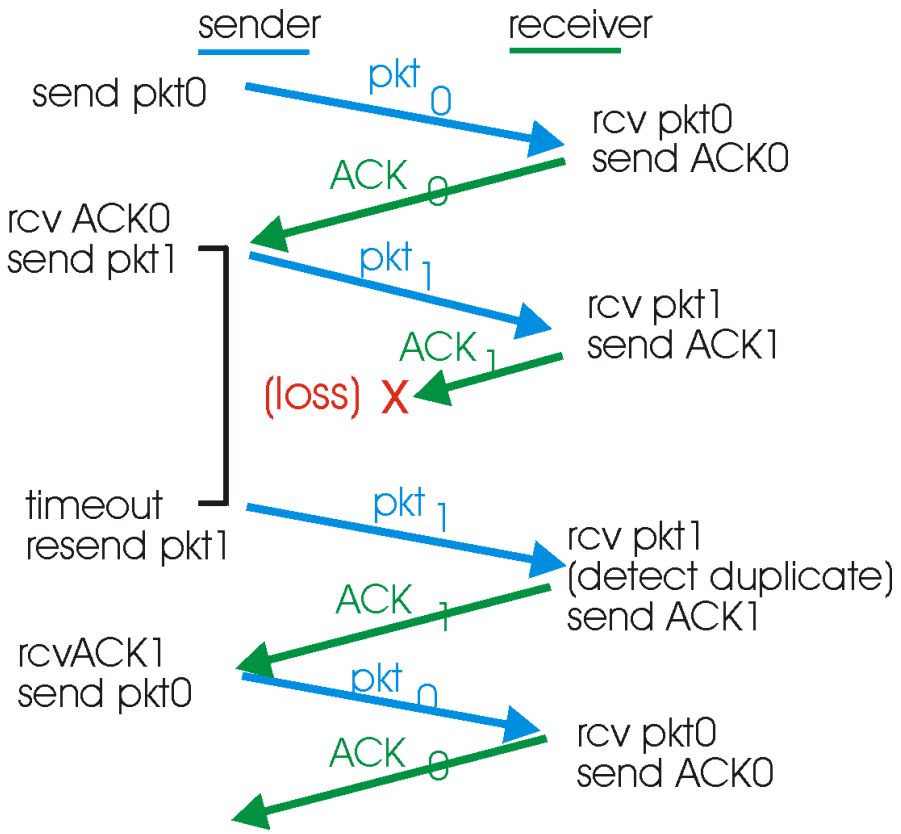


(a) operation with no loss

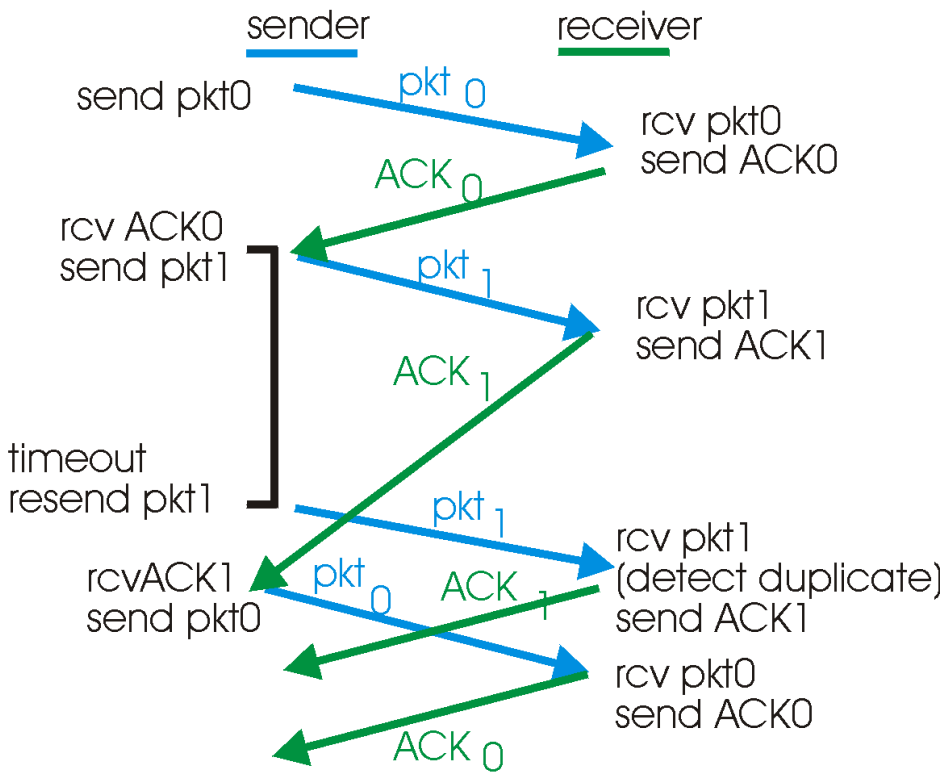


(b) lost packet

rdt3.0 in Action



(c) lost ACK

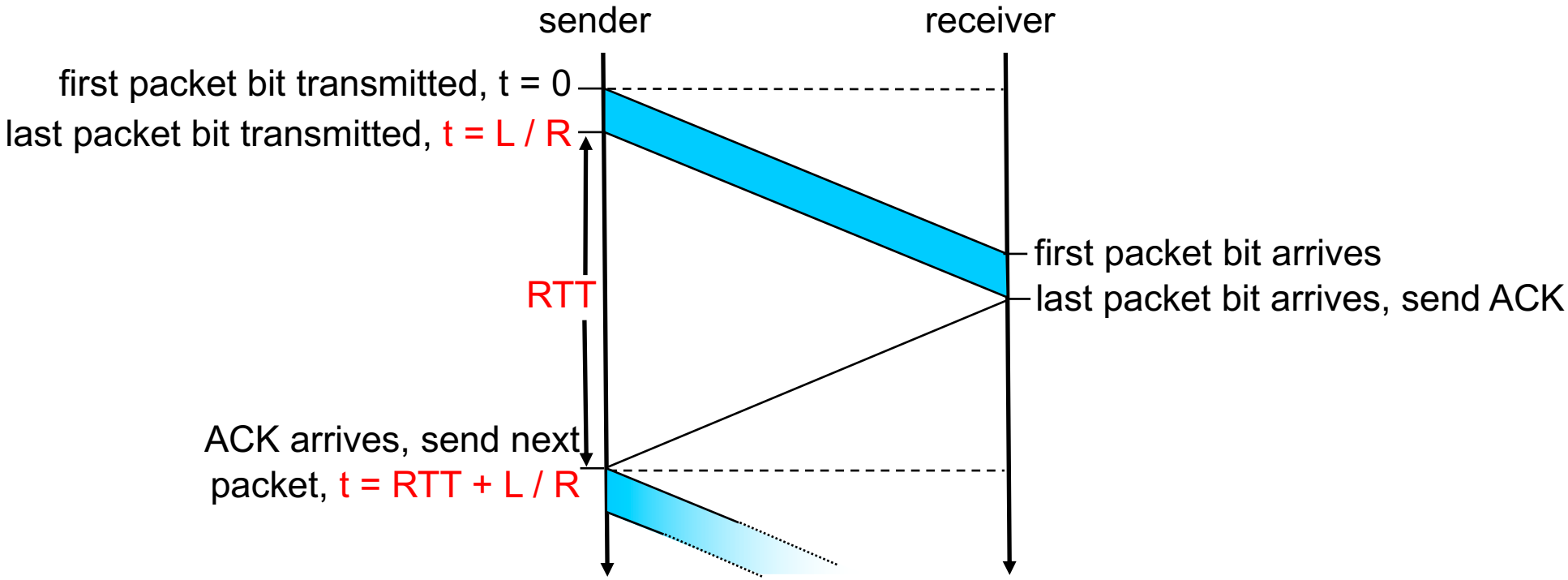


(d) premature timeout

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