Network Transport Layer: Transport Reliability: Sliding Windows; Connection Management; TCP

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf21/index.shtml

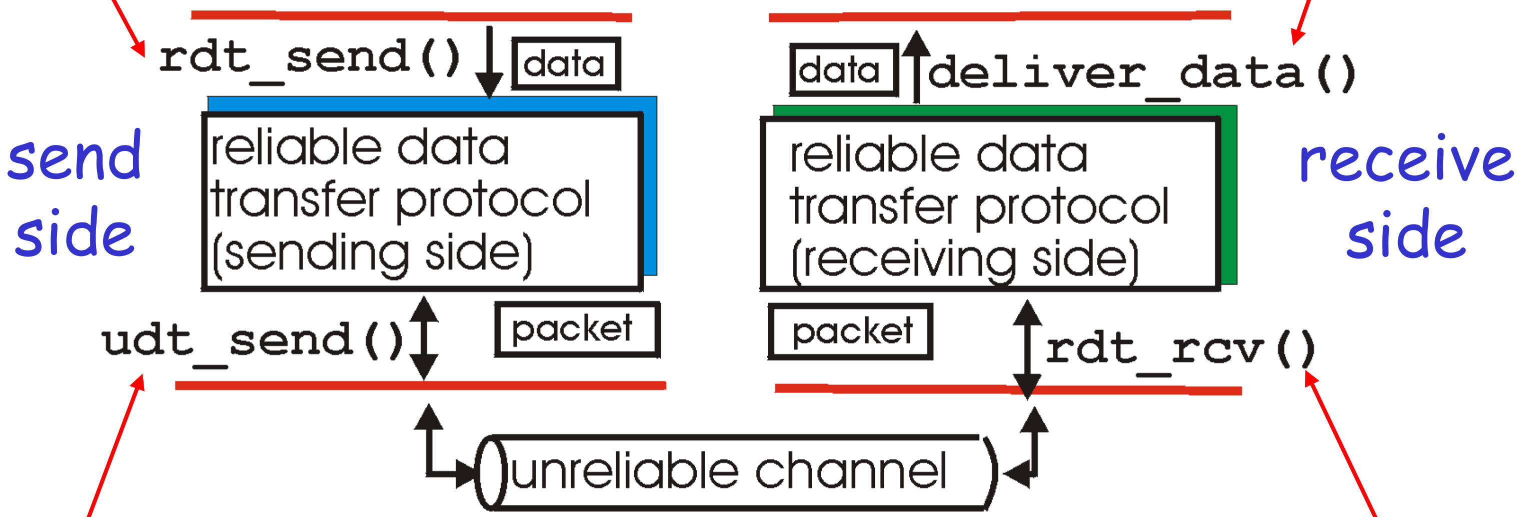
11/11/2021

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

Recap: Reliable Data Transfer Context

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

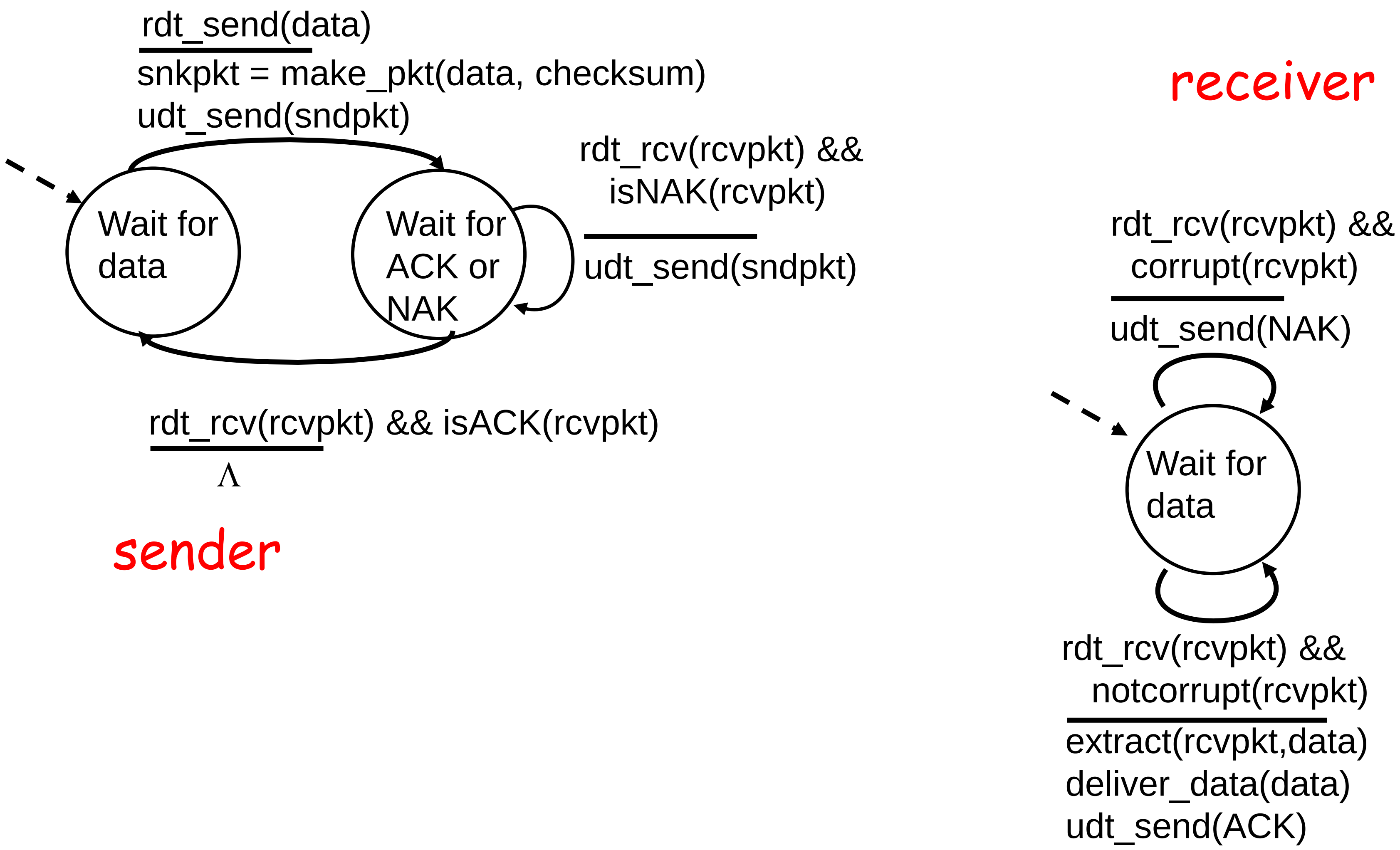
deliver_data(): called by **rdt** to deliver data to upper



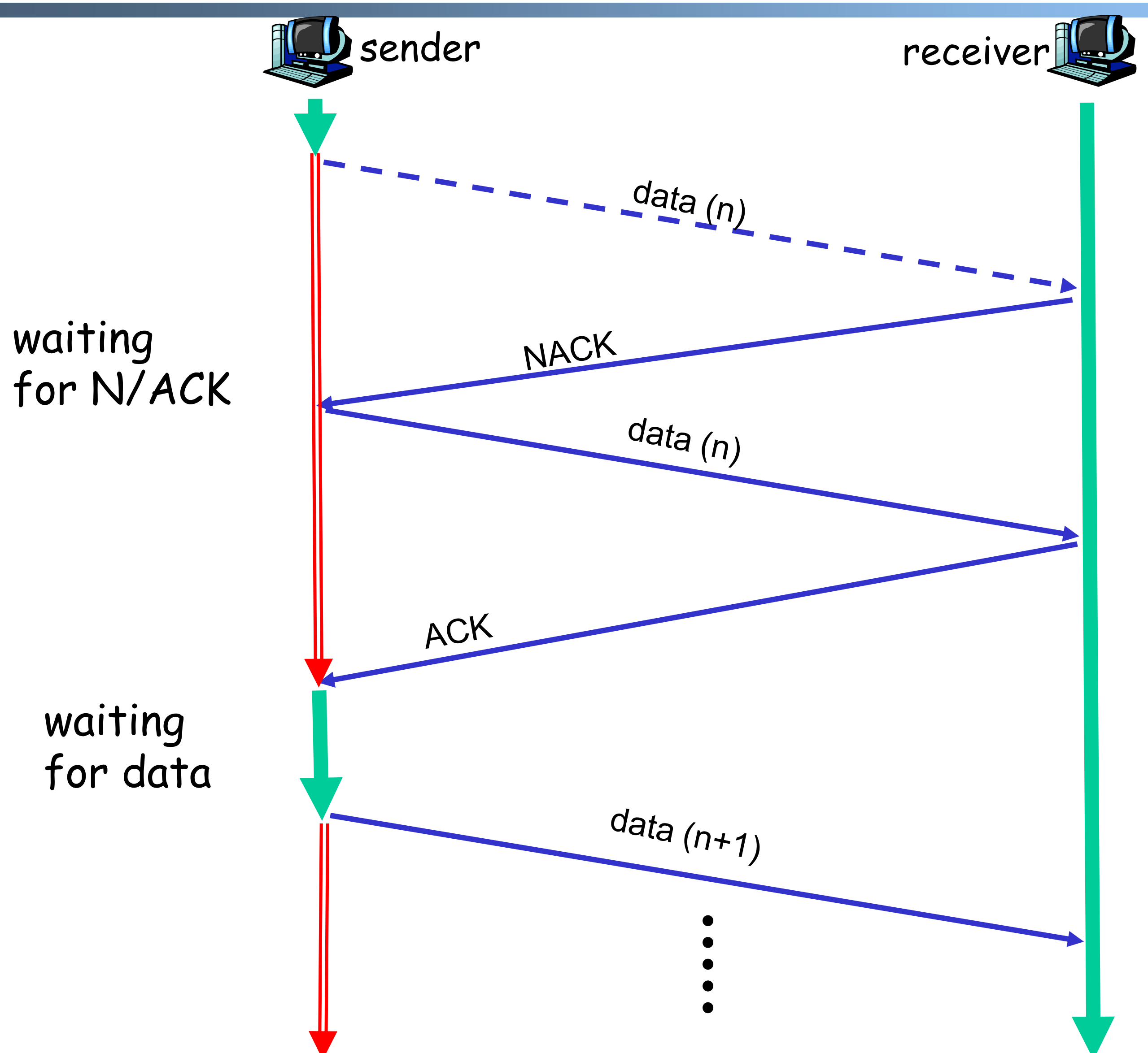
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Recap: rdt2.0: Reliability allowing only Data Msg Corruption

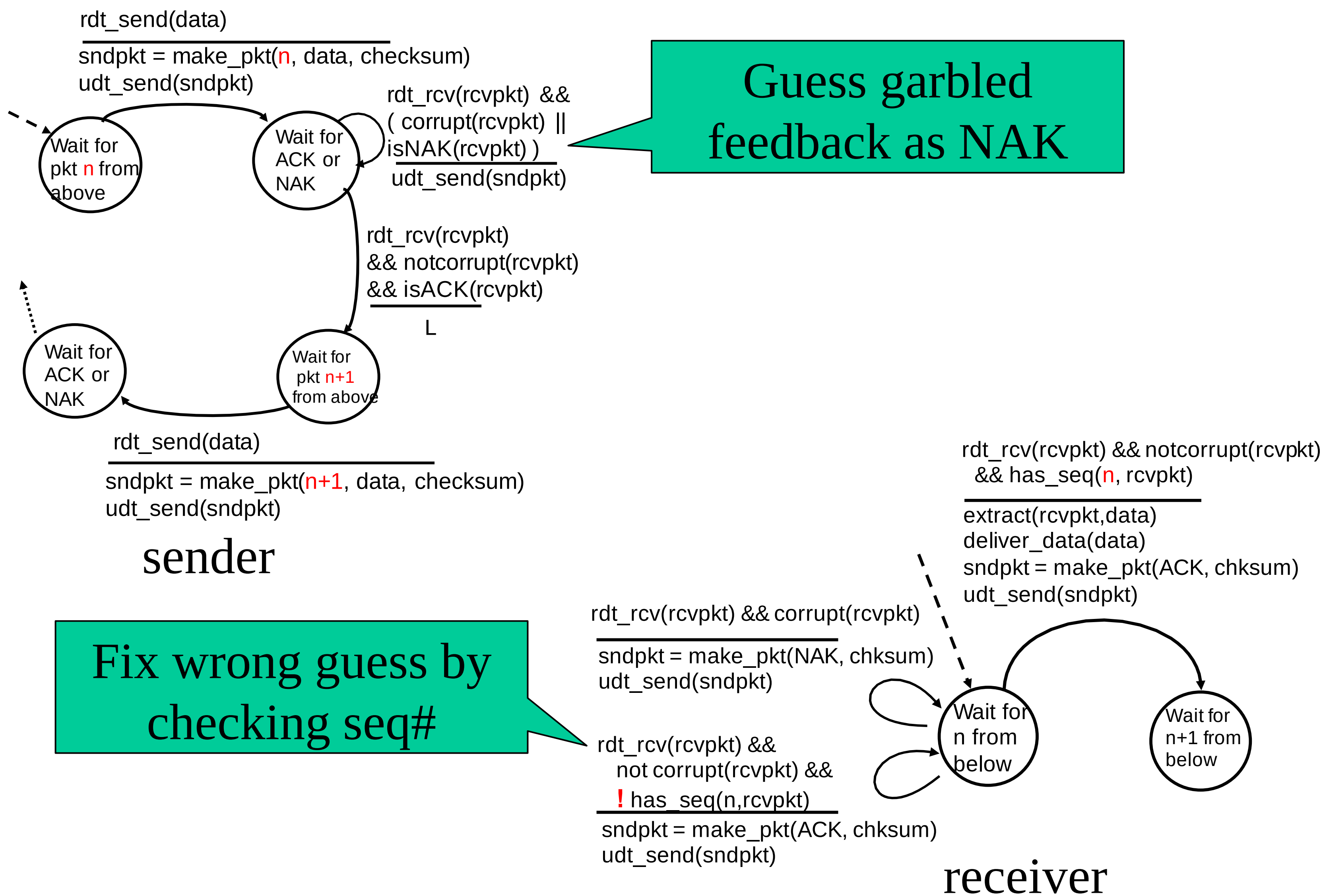


Recap: Rdt2.0 Analysis



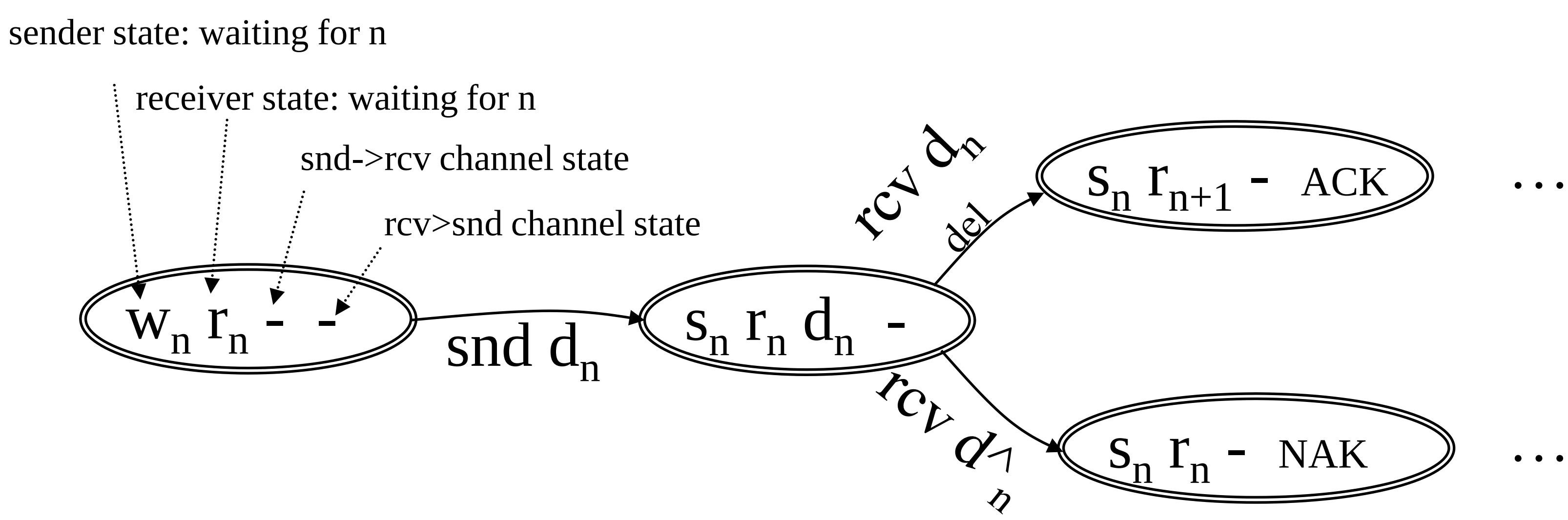
Execution traces of rdt2.0:
{data^ NACK}*
data deliver
ACK

Recap: rdt2.1b: Reliability allowing Data/Control Msg Corruption

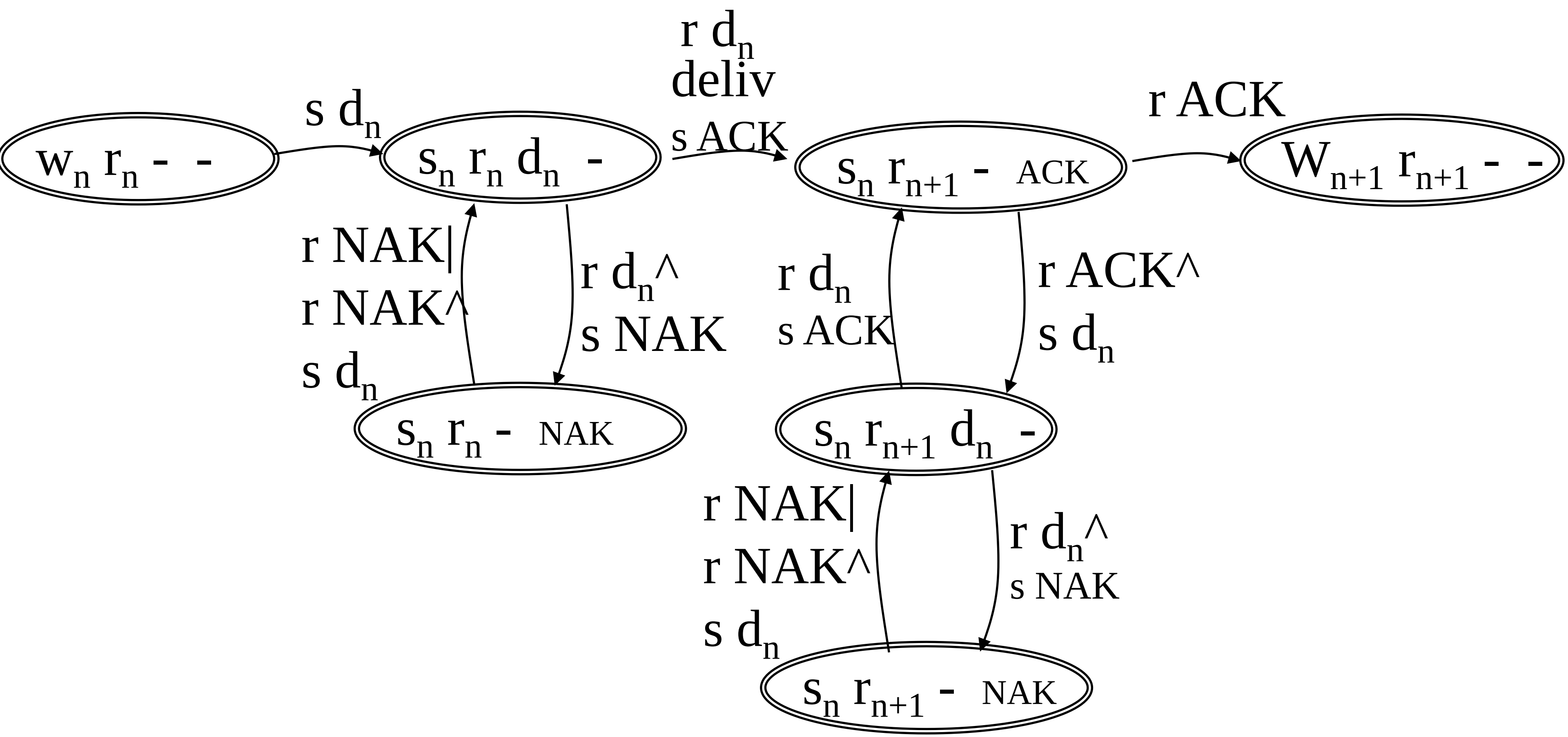


Protocol Analysis using (Generic) Execution Traces Technique

- 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



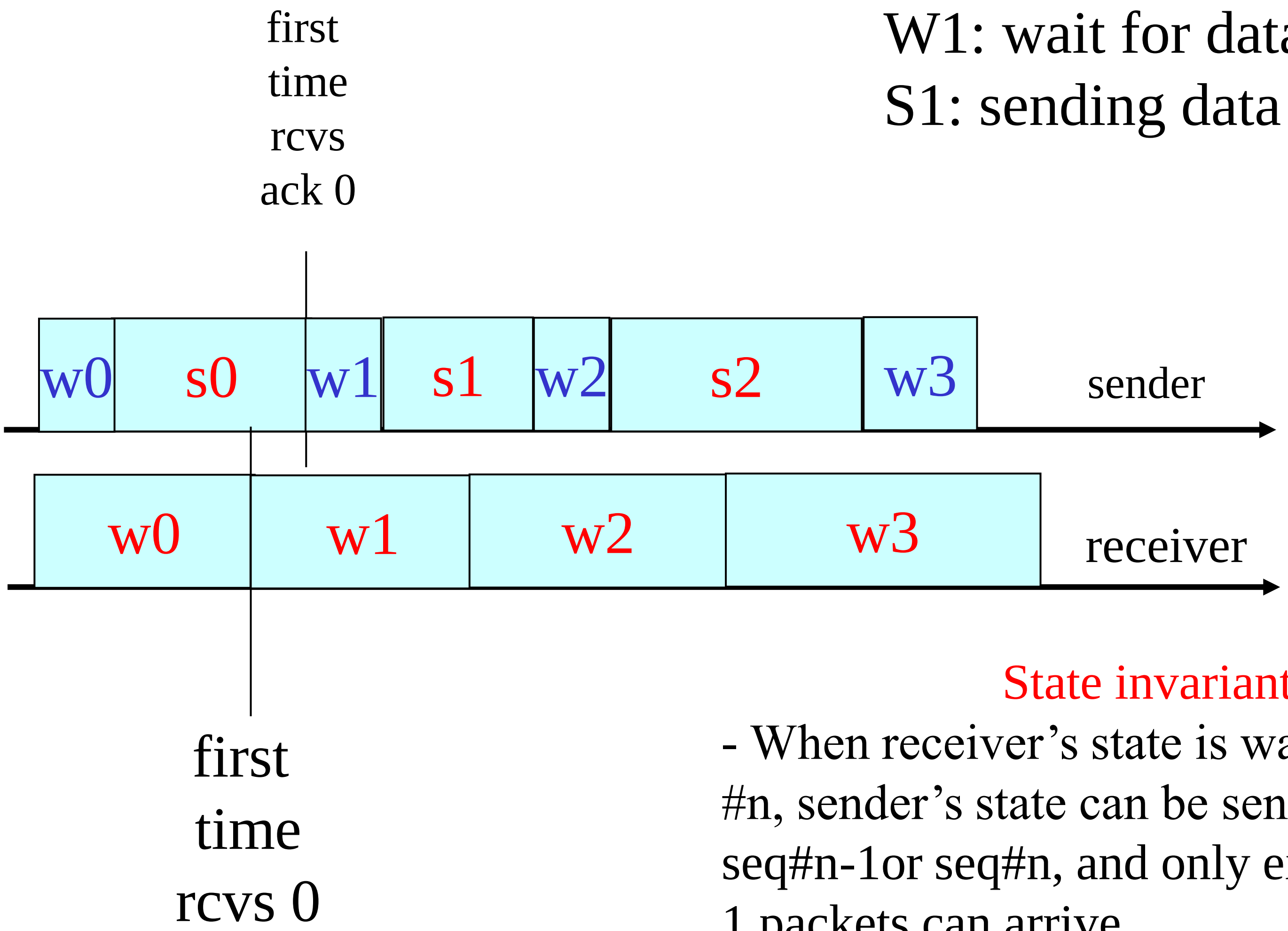
Recap: Protocol Analysis using (Generic) Execution Traces Technique



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

Recap: Protocol Analysis using State Invariants

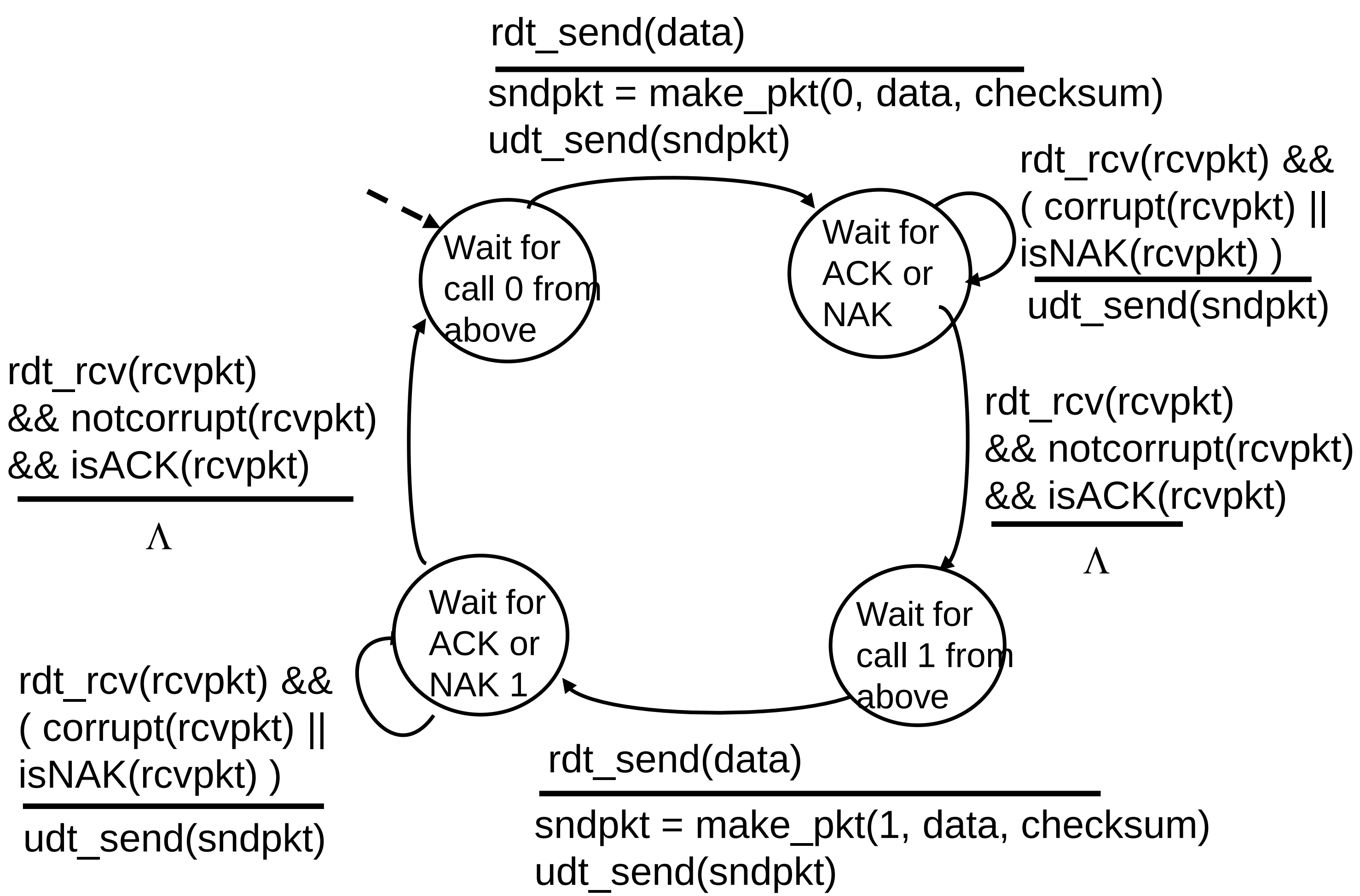
W1: wait for data with seq. 1
S1: sending data with seq. 1



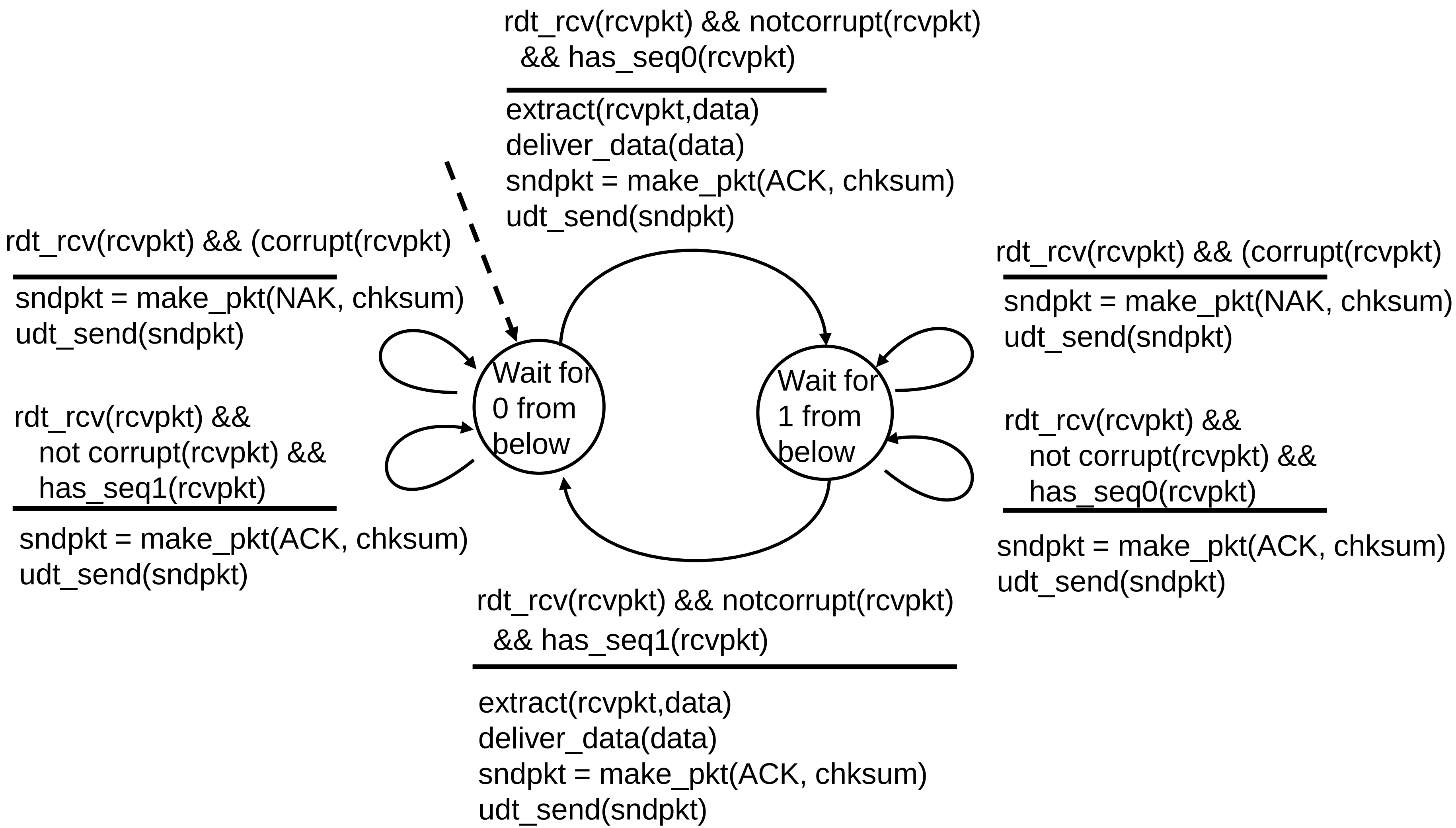
State invariant:

- 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. #

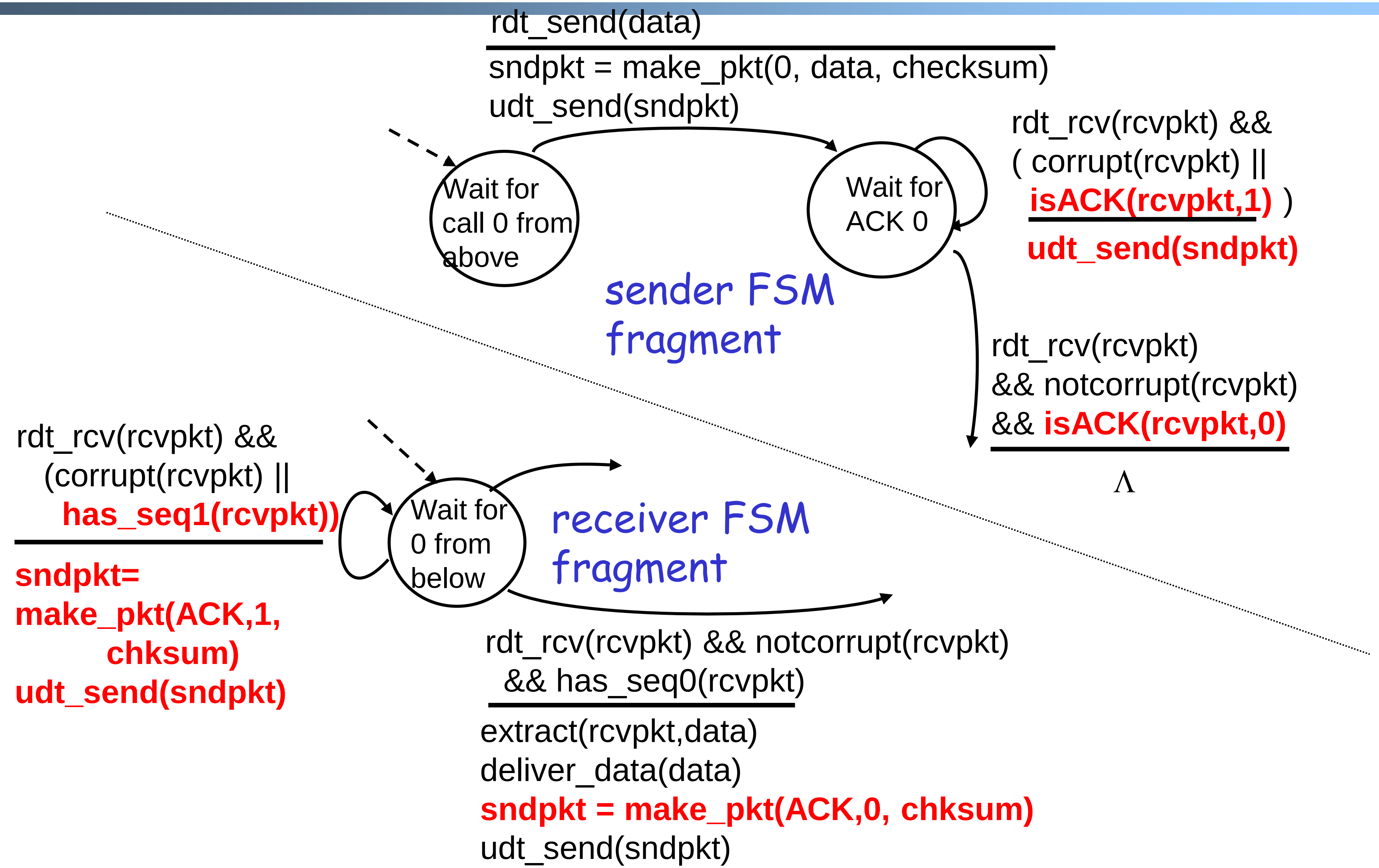
Receiver:

- 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)

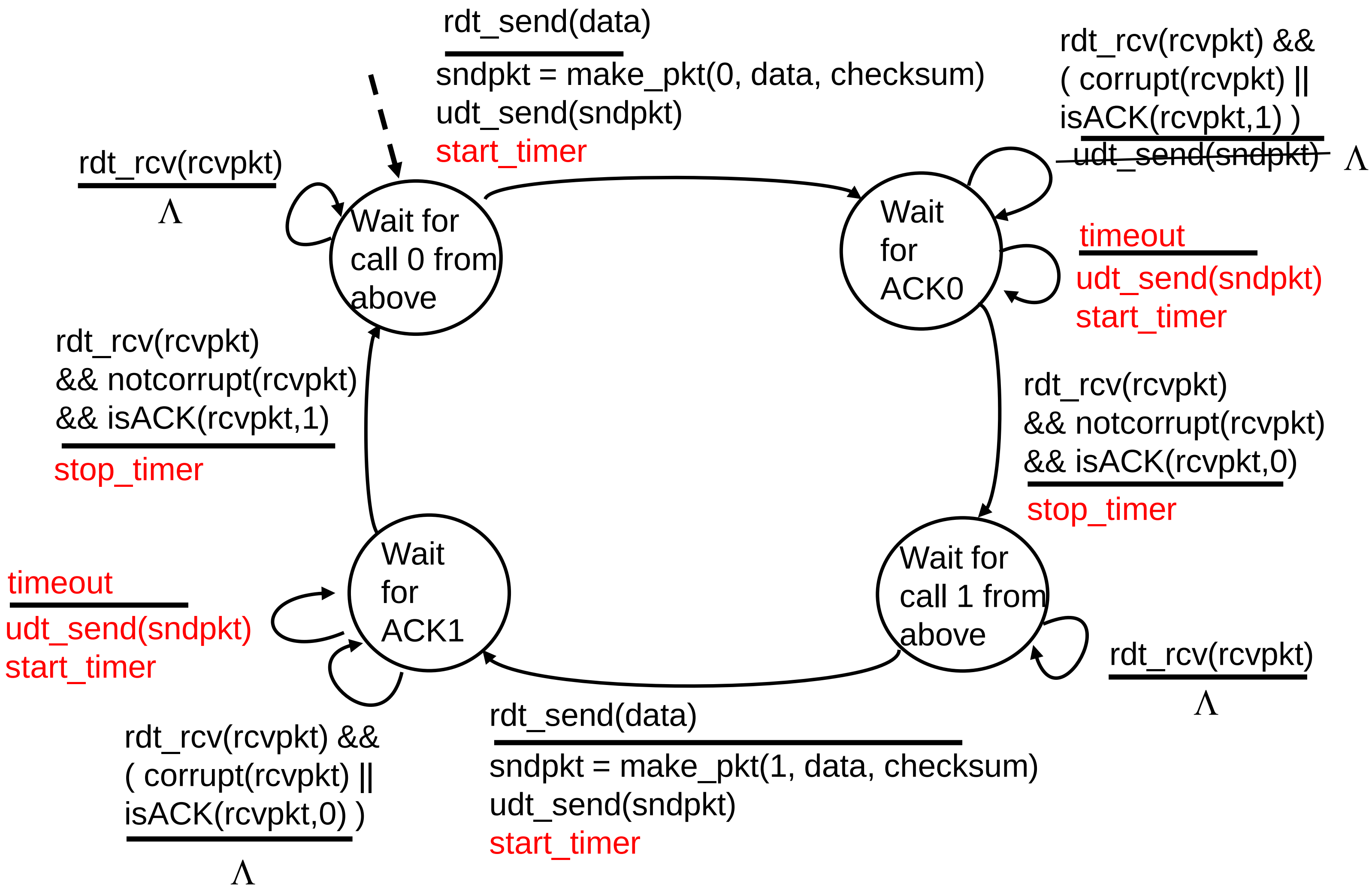
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: Does rdt2.2 work under losses?

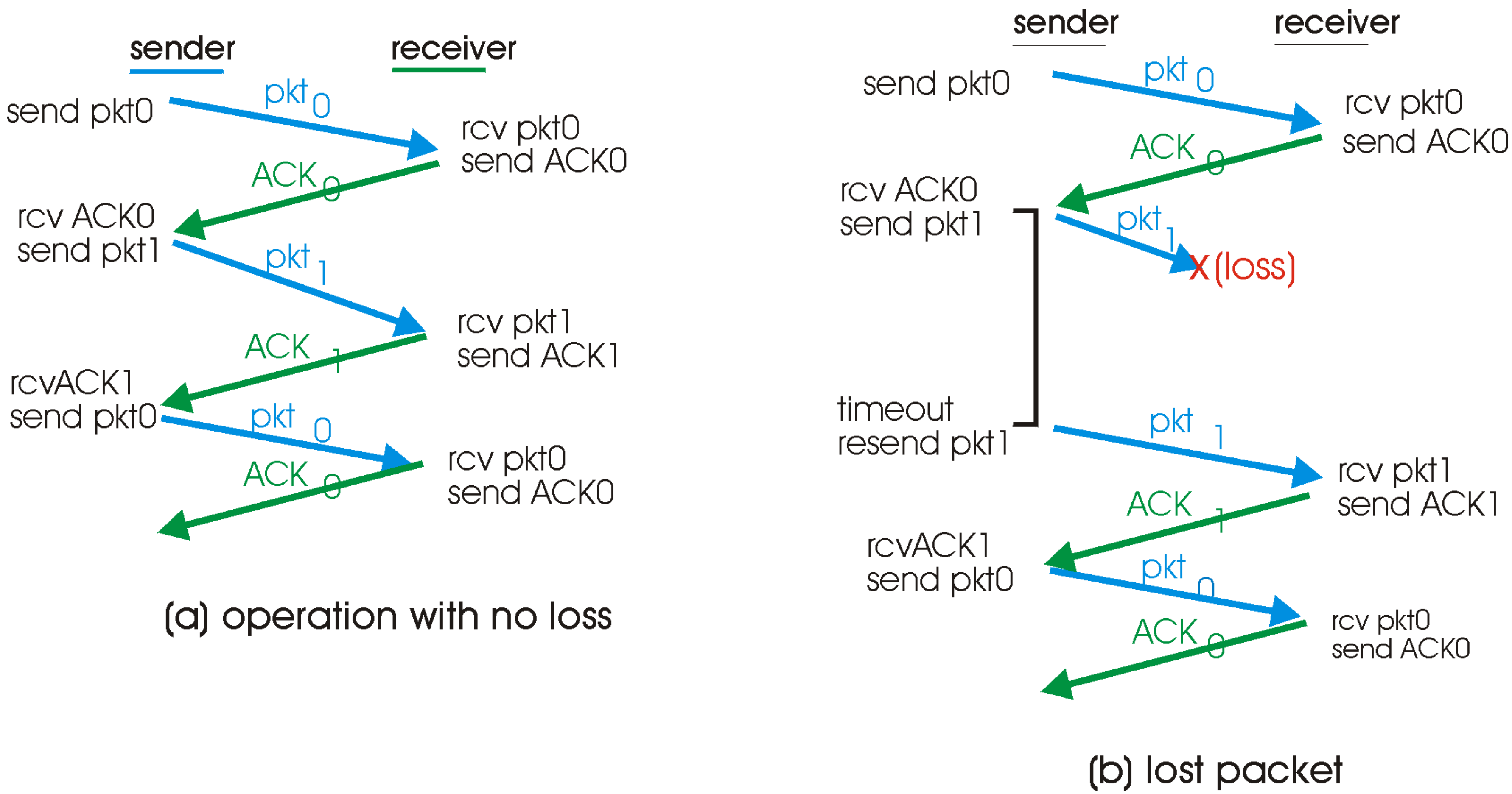
Approach: sender waits "reasonable" amount of time for ACK

- 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

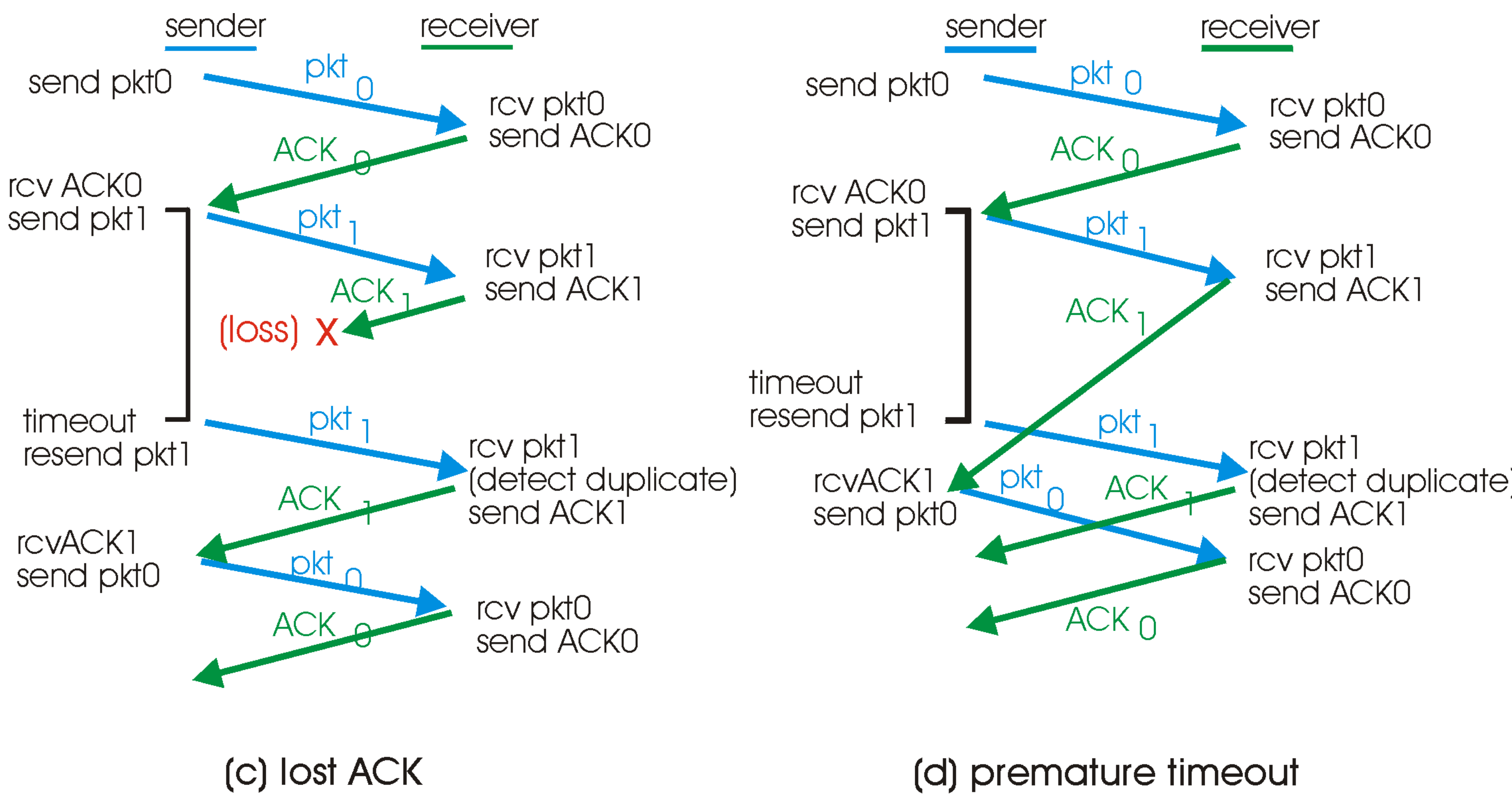
rdt3.0 Sender



rdt3.0 in Action



rdt3.0 in Action



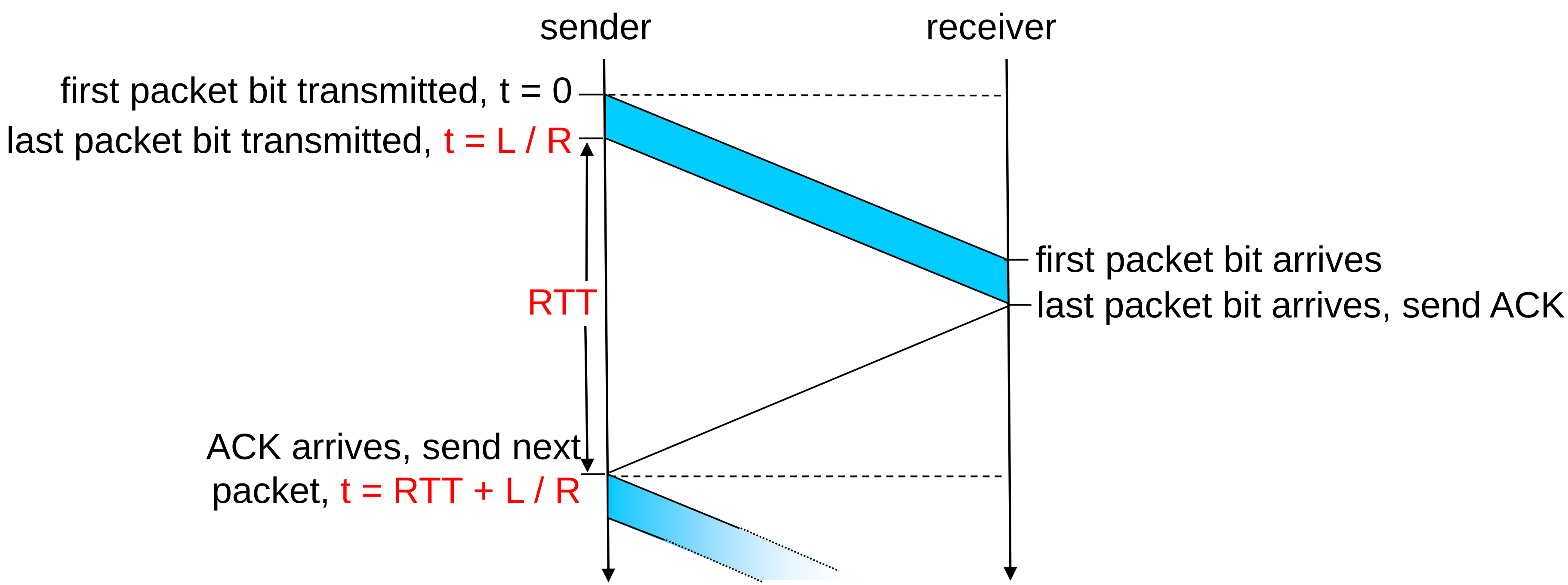
(c) lost ACK

(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:

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

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

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

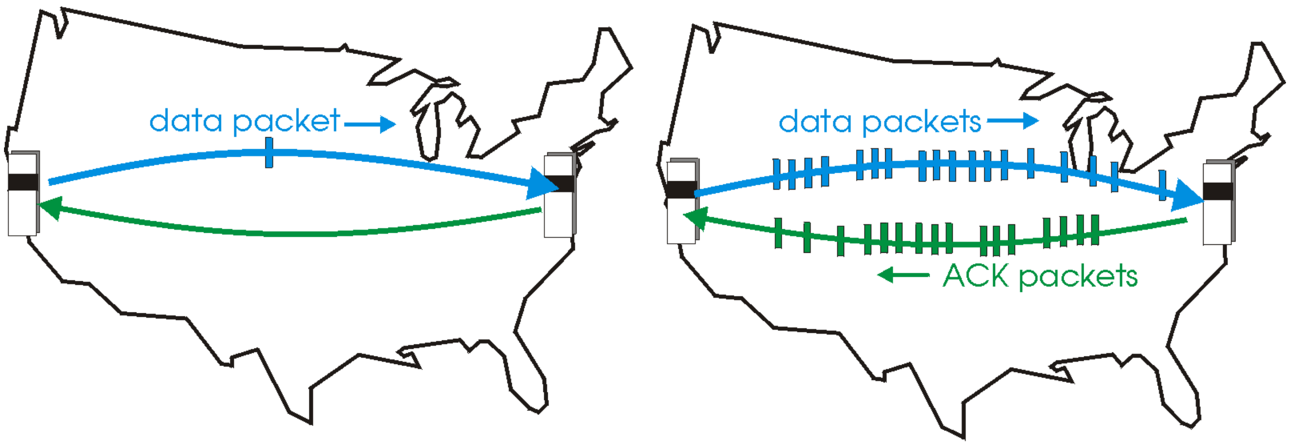
A Summary of Questions

- 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

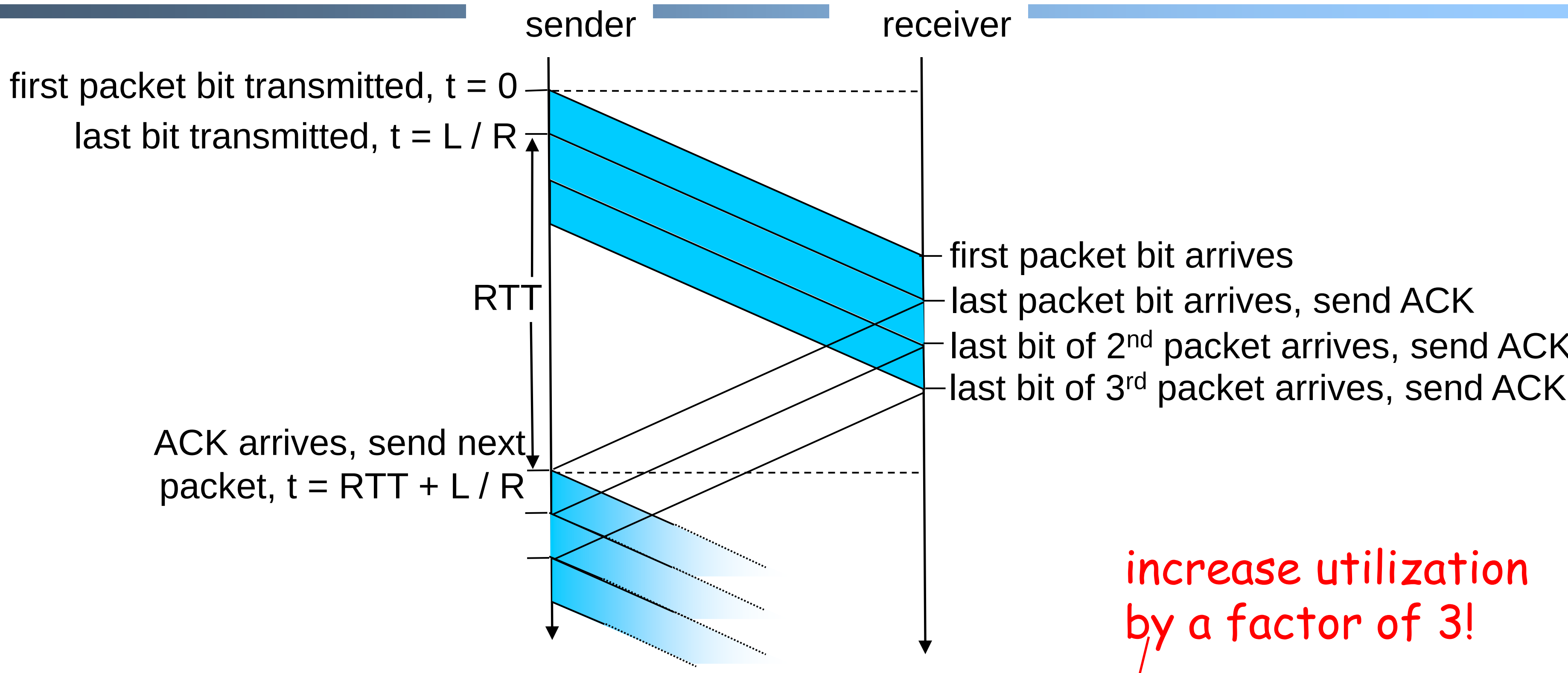
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: Increased Utilization



increase utilization by a factor of 3!

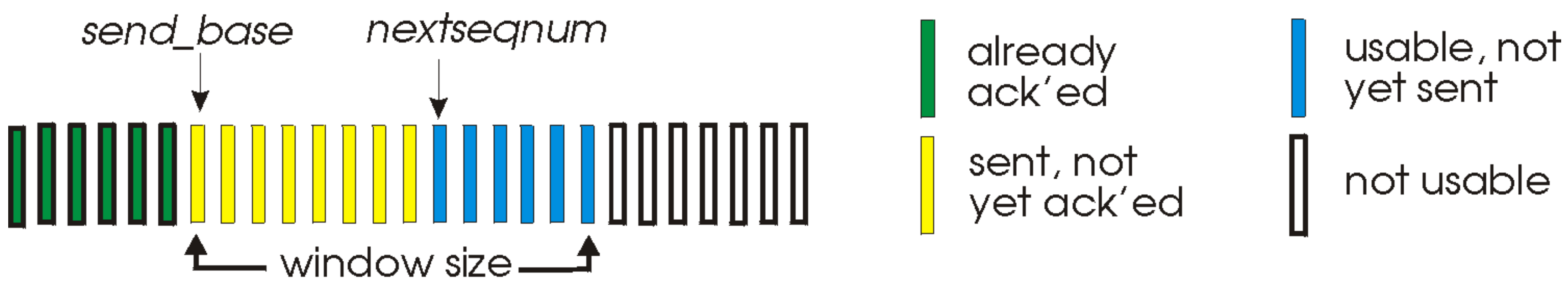
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

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



W

- 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

rdt_send(data)

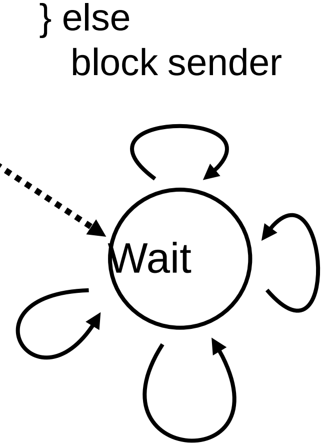
---

if (nextseqnum < base+W) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum) start_timer
    nextseqnum++
} else
    block sender

Λ

---

base=1
nextseqnum=1



timeout

---

start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

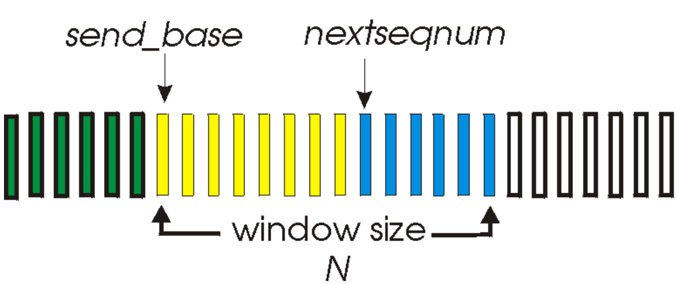
rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

---

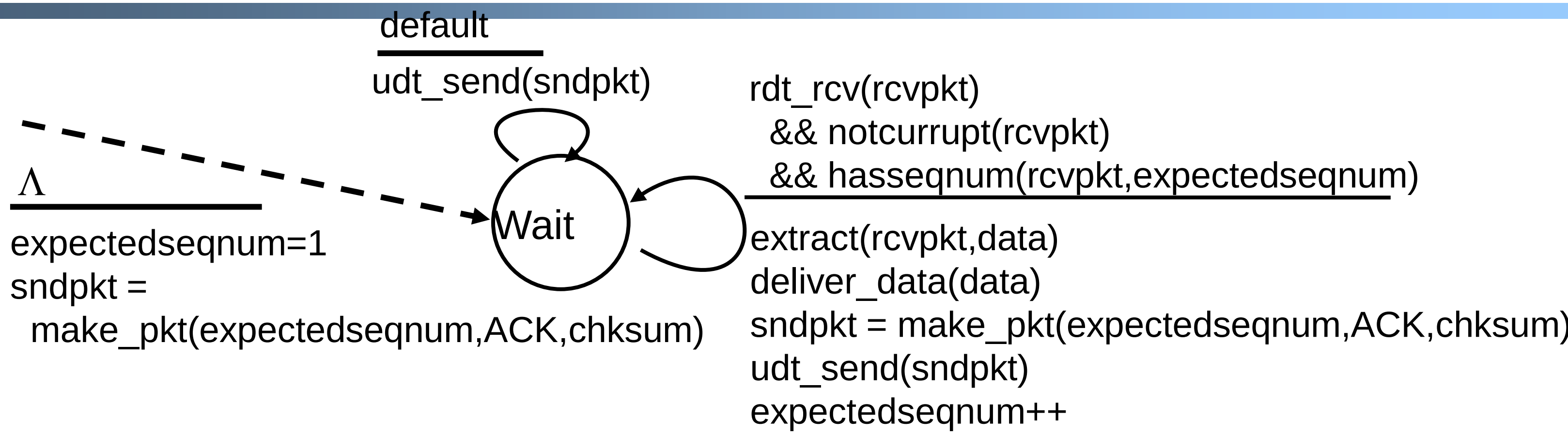
rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

---

if (new packets ACKed) {
  advance base;
  if (more packets waiting)
      send more packets
}
if (base == nextseqnum)
  stop_timer
else
  start_timer for the packet at new base



GBN: Receiver FSM

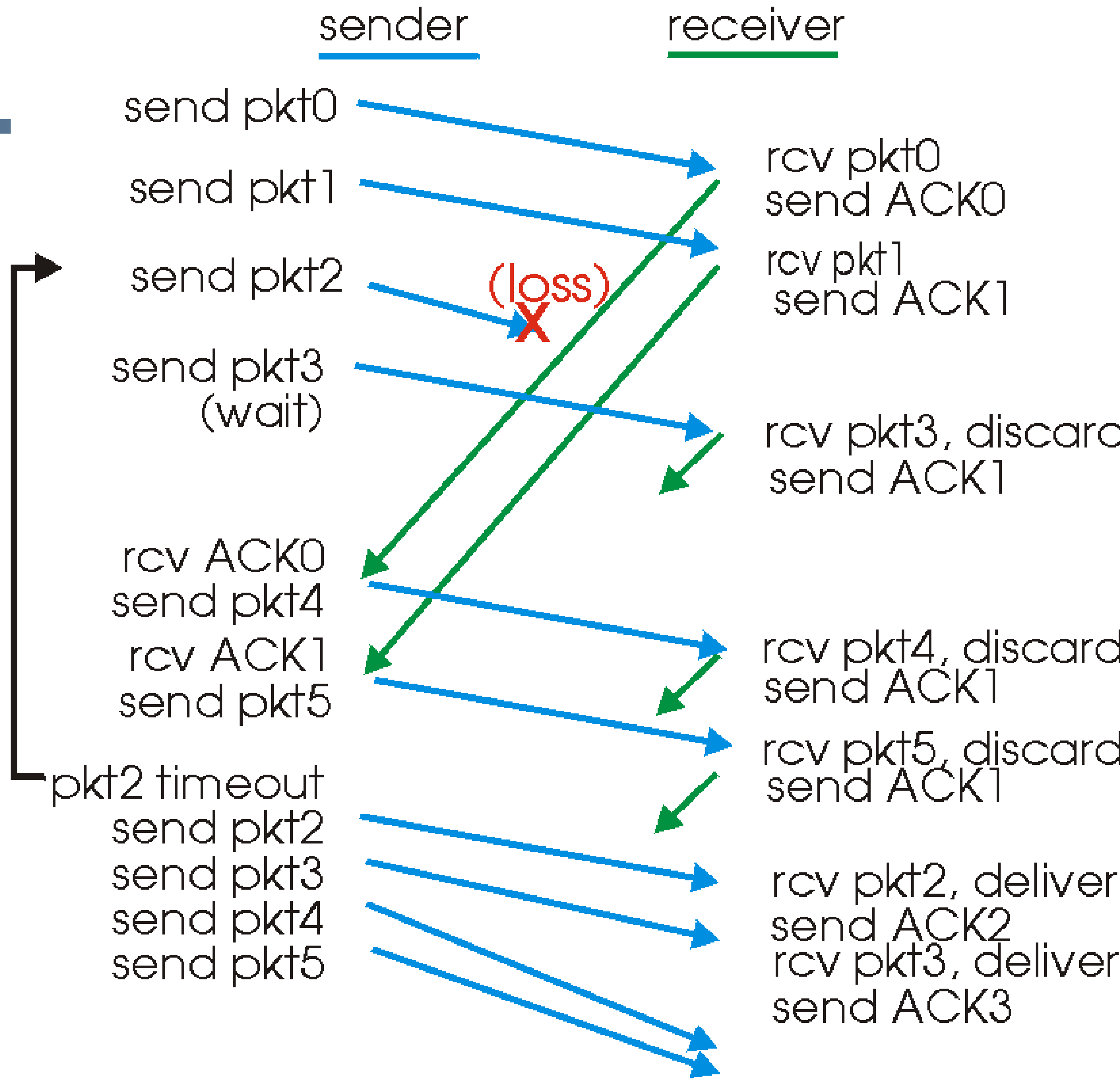


Only state: **`expectedseqnum`**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs

GBN in Action

window size = 4



Analysis: Efficiency of Go-Back-n

- Assume window size W
- Assume each packet is lost with probability p
- On average, how many packets do we send for each data packet received?

Selective Repeat

- Sender window
  - Window size W: W consecutive unACKed seq #'s
- Receiver *individually* acknowledges correctly received pkts
  - buffers out-of-order pkts, for eventual in-order delivery to upper layer
  - ACK(n) means received packet with seq# n only
  - buffer size at receiver: window size
- Sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt