Chapter 3: The Data Link Layer

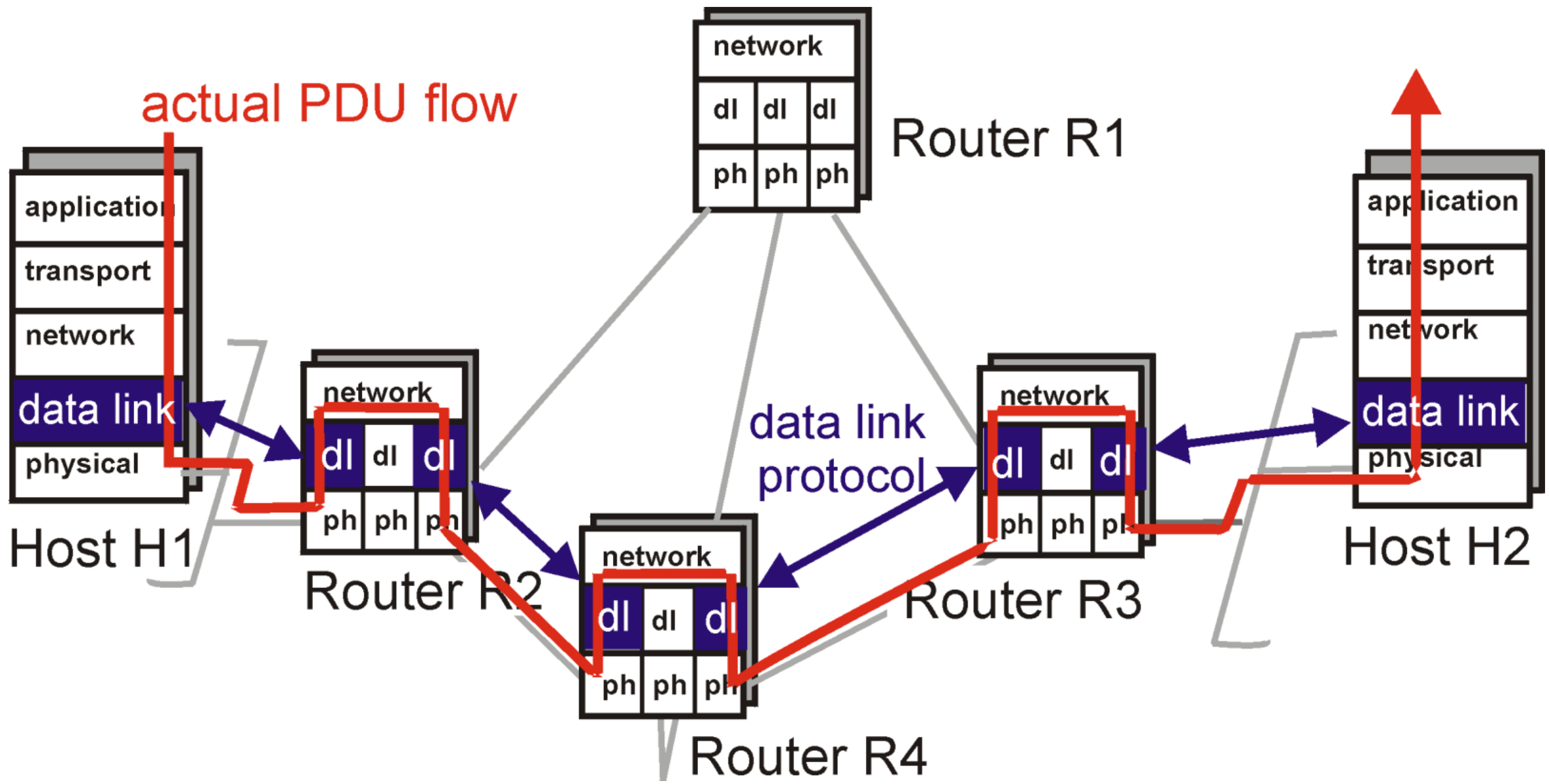
Our goals:

- understand principles behind data link layer services:
  - framing
  - error detection, correction
  - reliable data transfer
  - sharing a broadcast channel: multiple access

Overview:

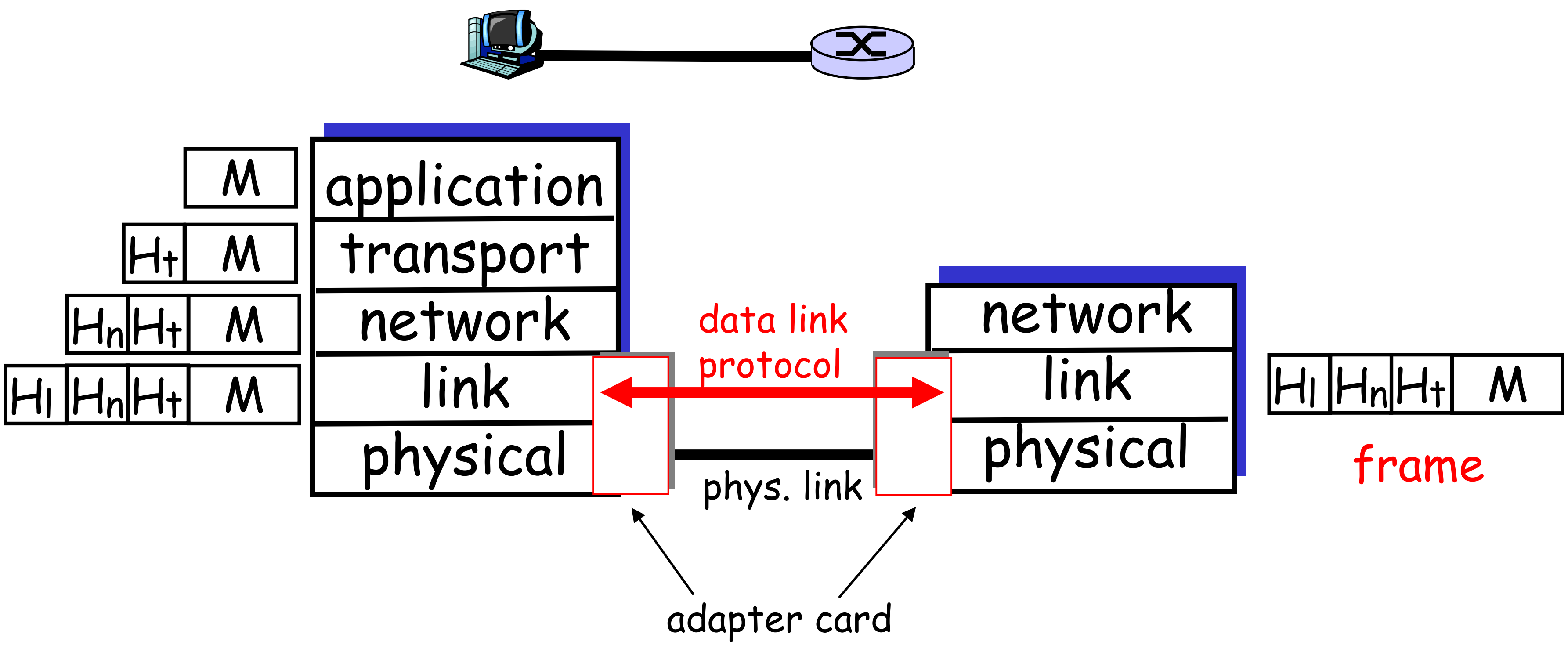
- link layer services
- framing
- error detection, correction
- reliable data transfer
- multiple access protocols

Link Layer: setting the context



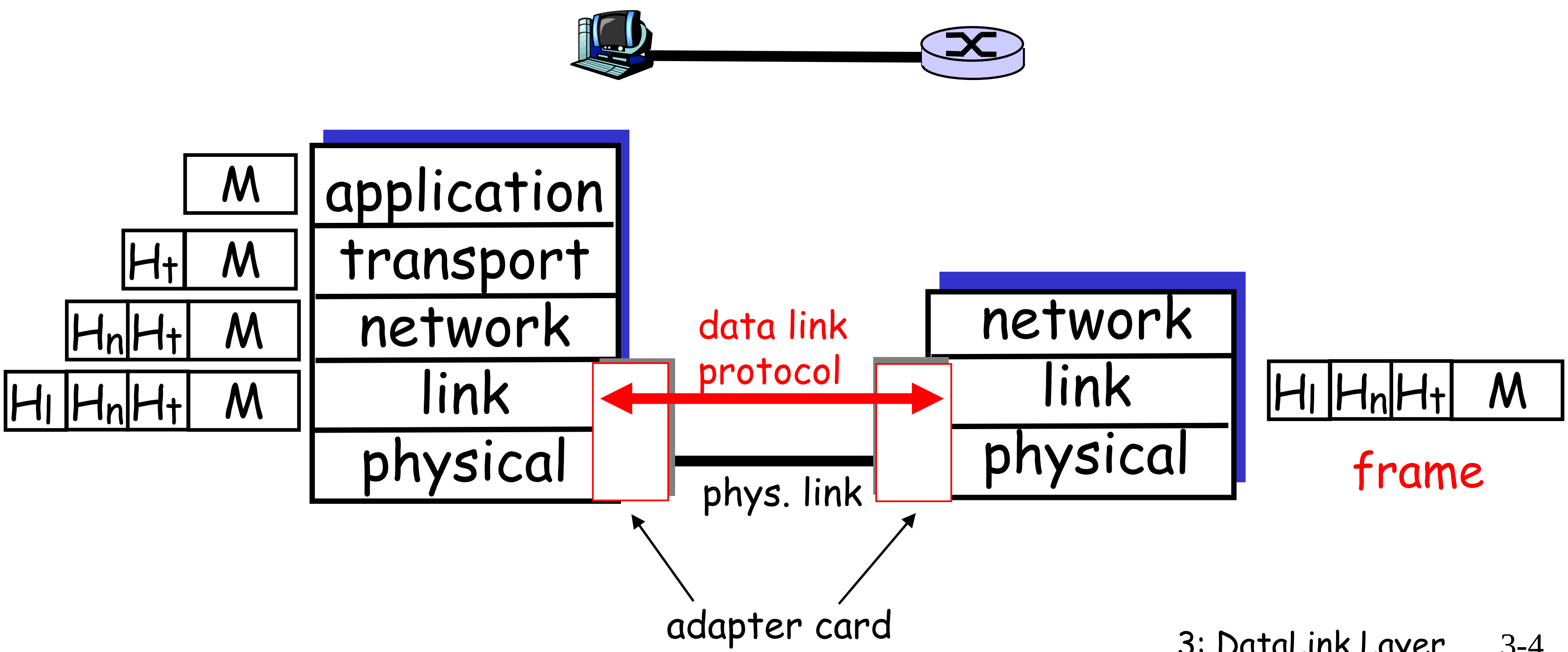
Link Layer: setting the context

- two *physically connected* devices:
  - host-router, router-router, host-host
- unit of data: *frame*



Link Layer: Implementation

- implemented in "adapter"
  - e.g., PCMCIA card, Ethernet card
  - typically includes: RAM, DSP chips, host bus interface, and link interface



Link Layer Services

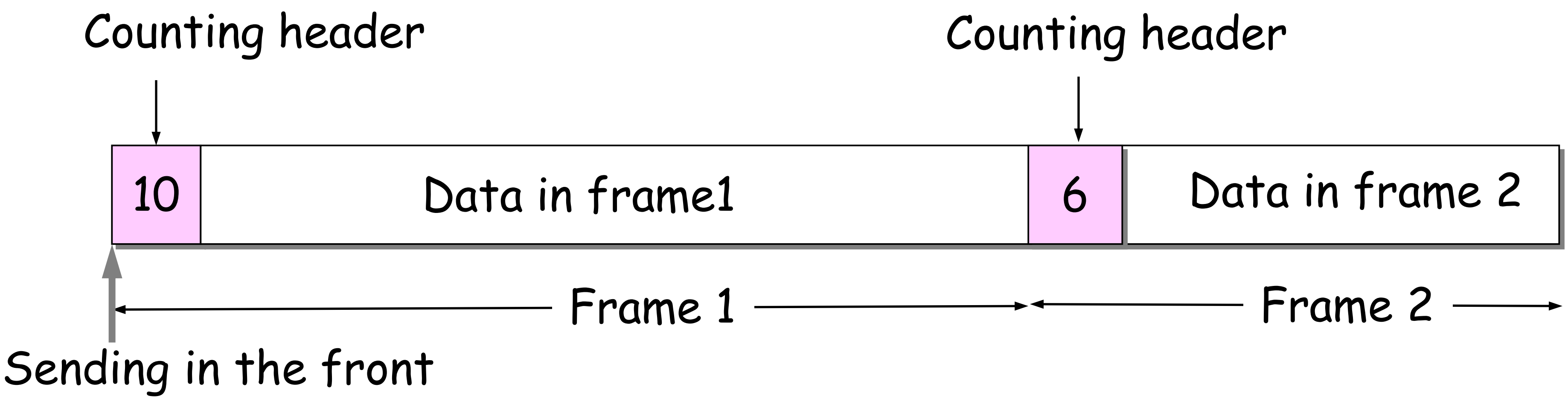
- Framing, link access:
  - encapsulate datagram into frame, adding header, trailer
  - implement channel access if shared medium,
  - 'physical addresses' used in frame headers to identify source, dest
- Reliable delivery between two physically connected devices:
  - Reliable data transfer protocol
  - seldom used on low bit error link (fiber, some twisted pair)
  - wireless links: high error rates

Link Layer Services (more)

- Flow Control:
  - pacing between sender and receivers
- *Error Detection*:
  - errors caused by signal attenuation, noise.
  - receiver detects presence of errors:
    - signals sender for retransmission or drops frame
- Error Correction:
  - receiver identifies *and corrects* bit error(s) without resorting to retransmission

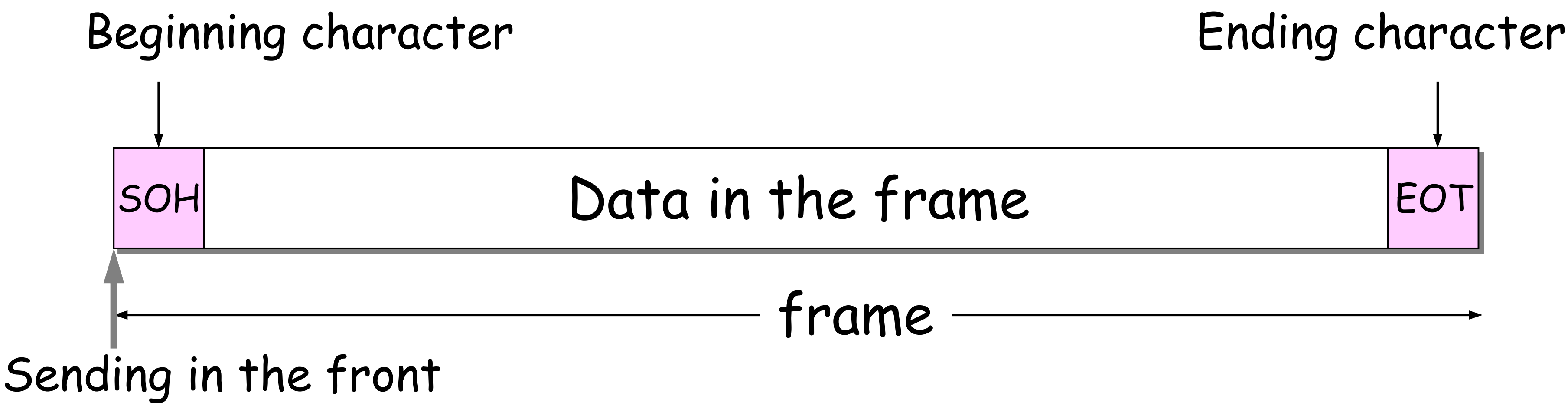
Framing

- Character count method :
  - encapsulate datagram into frame, adding header (character number)



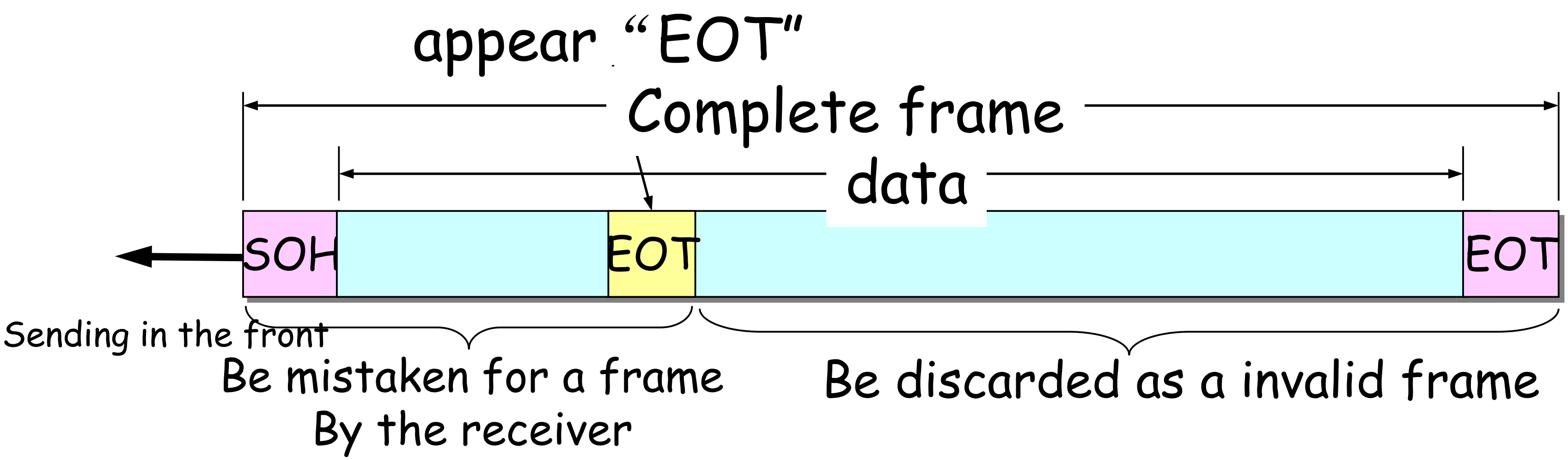
Framing

- First and tail bound method based on character:
  - encapsulate datagram into frame, adding header, trailer (character)



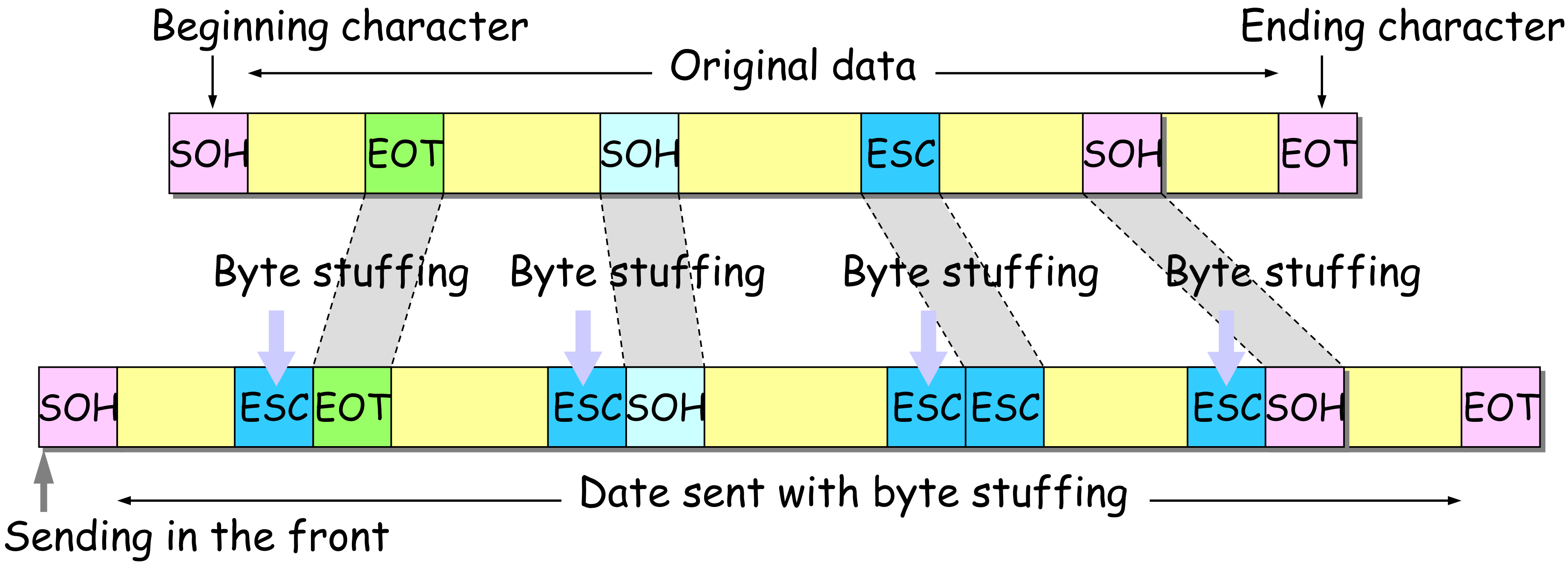
Framing

- First and tail bound method based on character:
  - encapsulate datagram into frame, adding header, trailer (character)



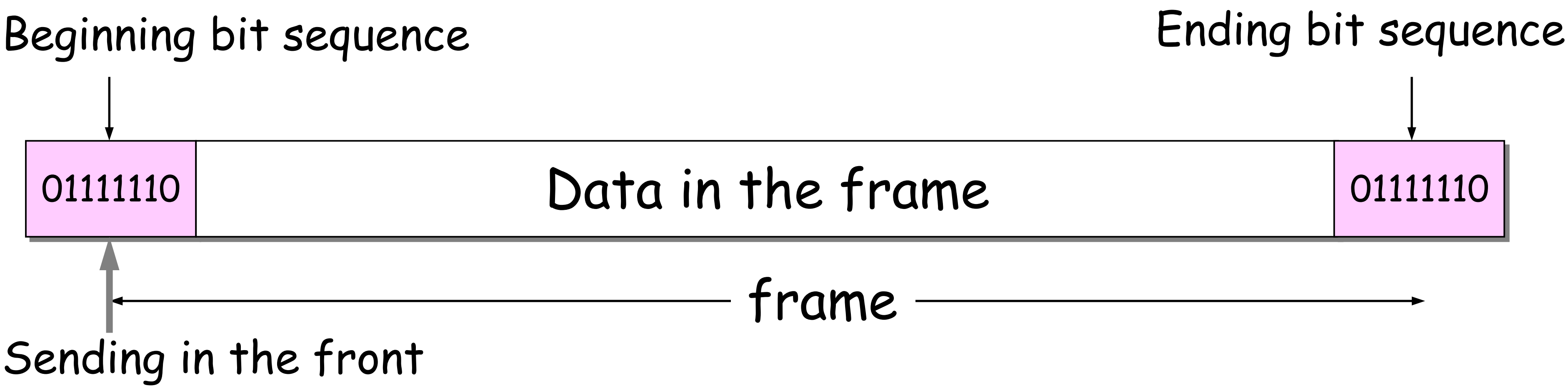
Framing

- First and tail bound method based on character:
  - Inserting/ stuffing a Escape character before the special character in the data part



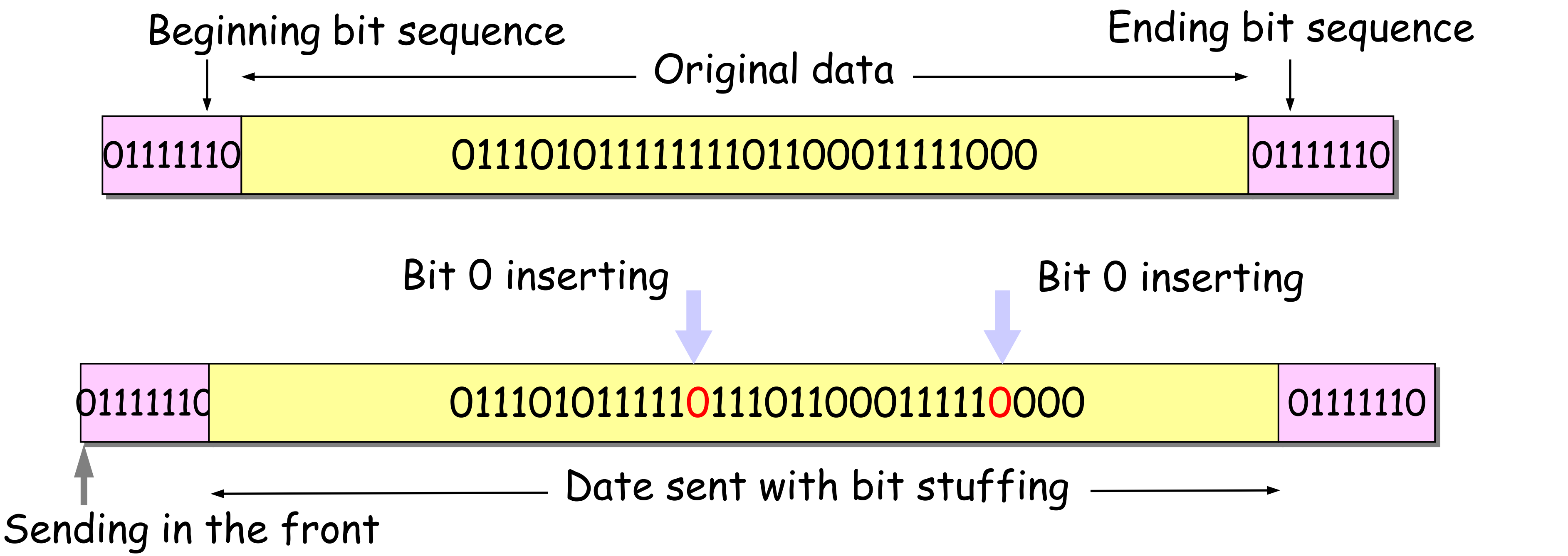
Framing

- First and tail bound method based on bit:
  - encapsulate datagram into frame, adding header, trailer (bit sequence)



Framing

- First and tail bound method based on bit:
  - Inserting a bit "0" after successive five bits "1" in the transmitter; vice versa



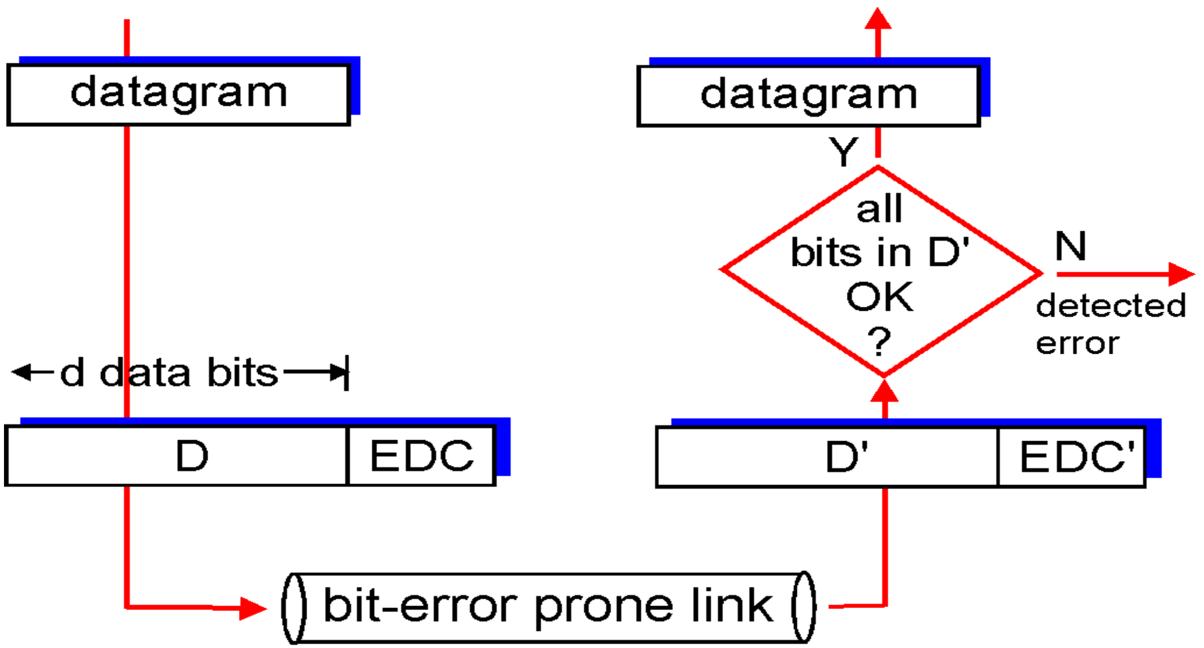
Framing

- Physical layer coding violation method:
  - encapsulate datagram into frame without stuffing
  - Only be used in the networks with redundancy coding technology in the physical layer
- For example (IEEE 802.11 with Manchester code):
  - Bit 1 with level jump mode from high to low
  - Bit 0 with level jump mode from low to high
  - Level jump modes from high to high or from low to low can be used for beginning and ending of a frame

Error Detection

EDC= Error Detection and Correction bits (redundancy)
D = Data protected by error checking, may include header fields

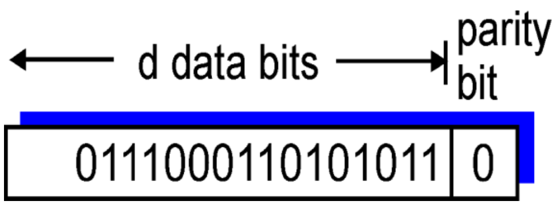
- Error detection not 100% reliable!
  - protocol may miss some errors, but rarely
  - larger EDC field yields better detection and correction



Parity Checking

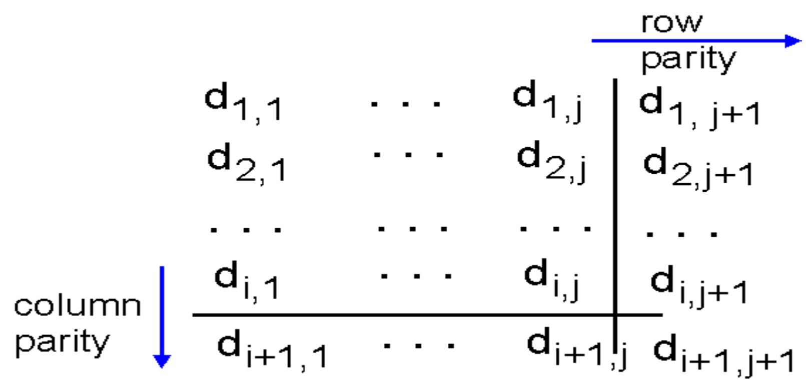
Single Bit Parity:

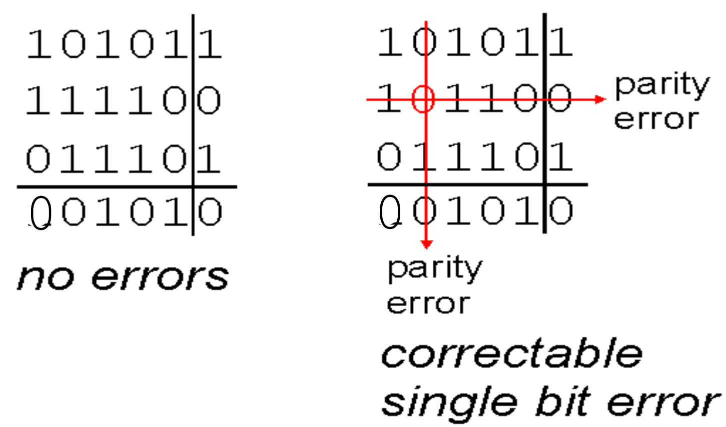
Detect single bit errors



Two Dimensional Bit Parity:

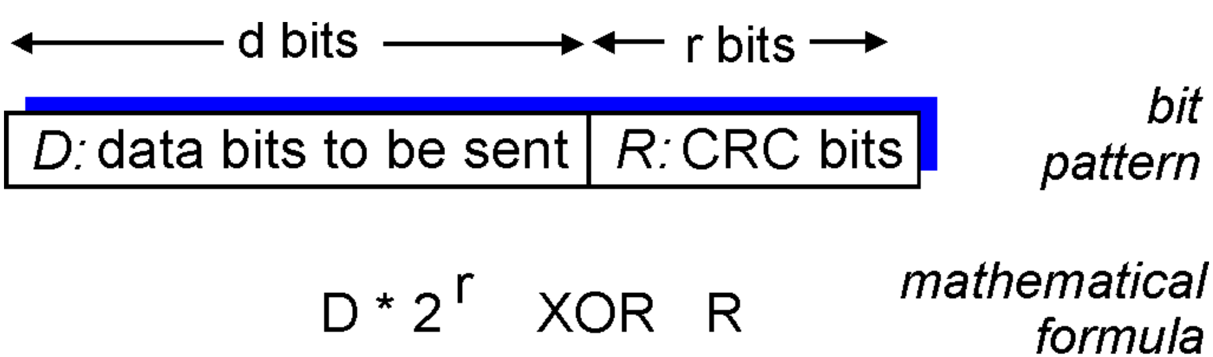
Detect *and correct* single bit errors





Checksumming: Cyclic Redundancy Check

- view data bits, D, as a binary number
- choose r+1 bit pattern (generator), G
- goal: choose r CRC bits, R, such that
  - <D,R> exactly divisible by G (modulo 2)
  - receiver knows G, divides <D,R> by G. If non-zero remainder: error detected!
  - can detect all burst errors less than r+1 bits
- widely used in practice (ATM, HDCL)



CRC Example

Want:

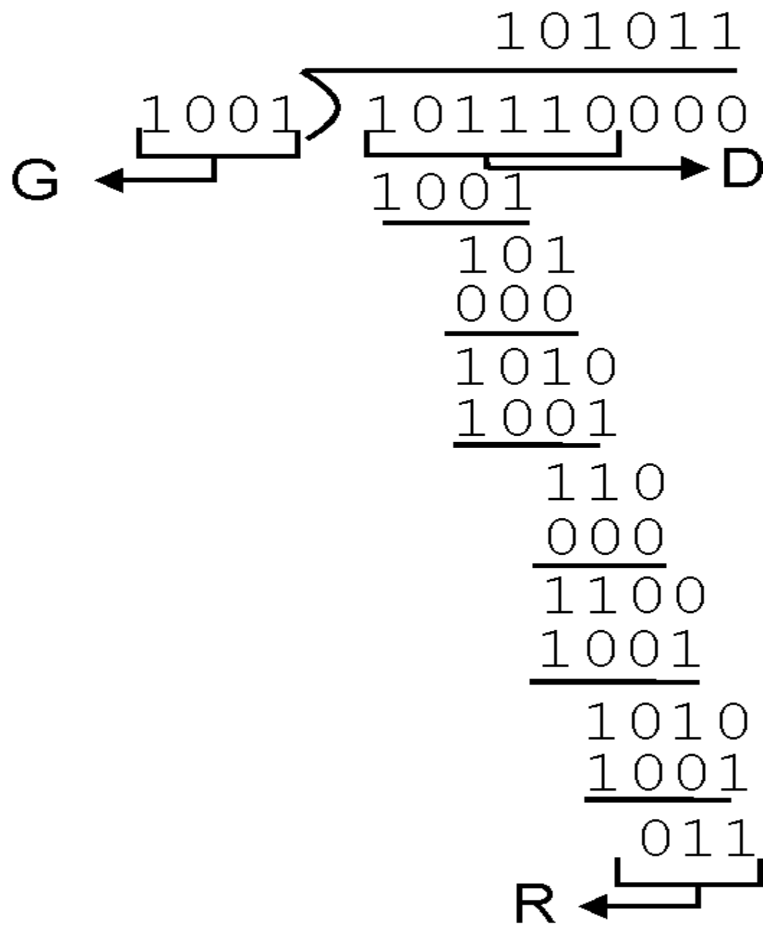
$D \cdot 2^r \text{ XOR } R = nG$

equivalently:

$D \cdot 2^r = nG \text{ XOR } R$

equivalently:

if we divide $D \cdot 2^r$ by G, want reminder R



$$R = \text{remainder}\left[\frac{D \cdot 2^r}{G}\right]$$

CRC well-known generator

- CRC-12: $x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$
- CRC-16 : $x^{16} + x^{15} + x^2 + 1$
- CRC-CCITT : $x^{16} + x^{12} + x^5 + 1$
- CRC-32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$

CCITT: Consultative Committee on International Telegraphy and Telephone

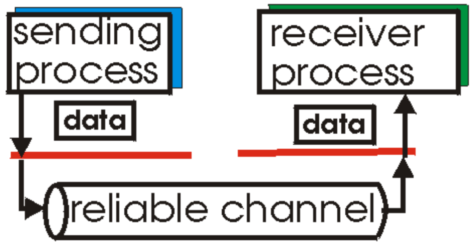
ITU-T: International Telecommunications Union – Telecommunications Standardization Sector

Exercise

- Consider the 5-bit generator, G=10011, and suppose that D has the value 101110.
- What is the value of R?

Principles of Reliable Data Transfer

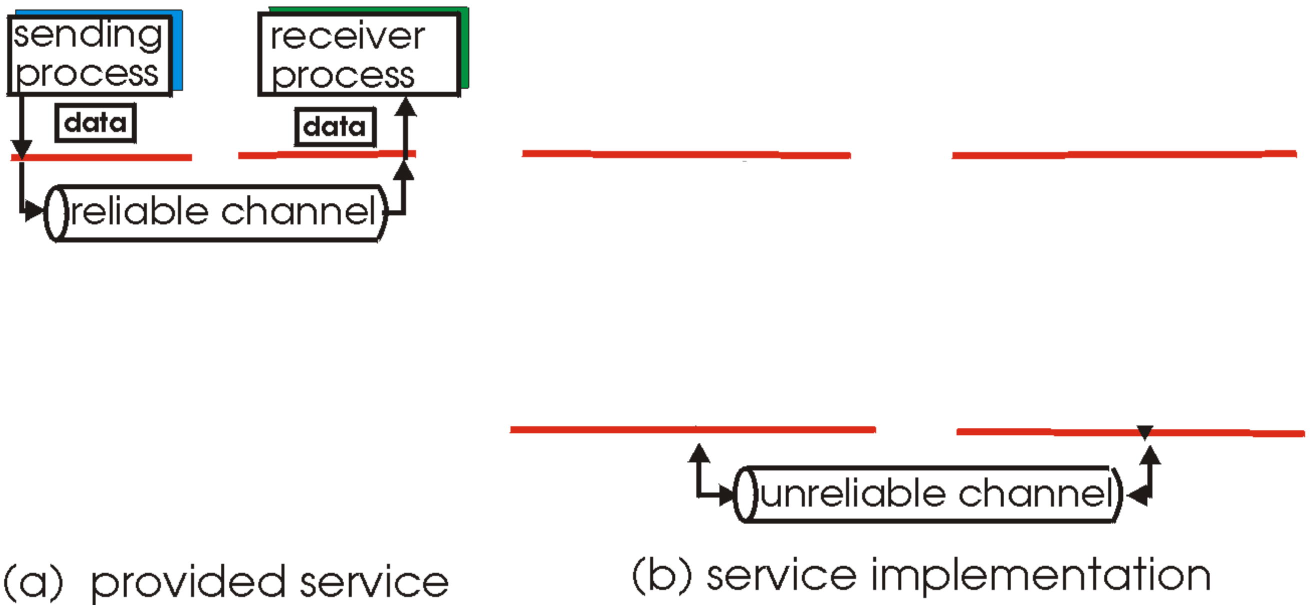
- Important in app., transport, link layers
- Top-10 list of important networking topics!



(a) provided service

Principles of Reliable Data Transfer

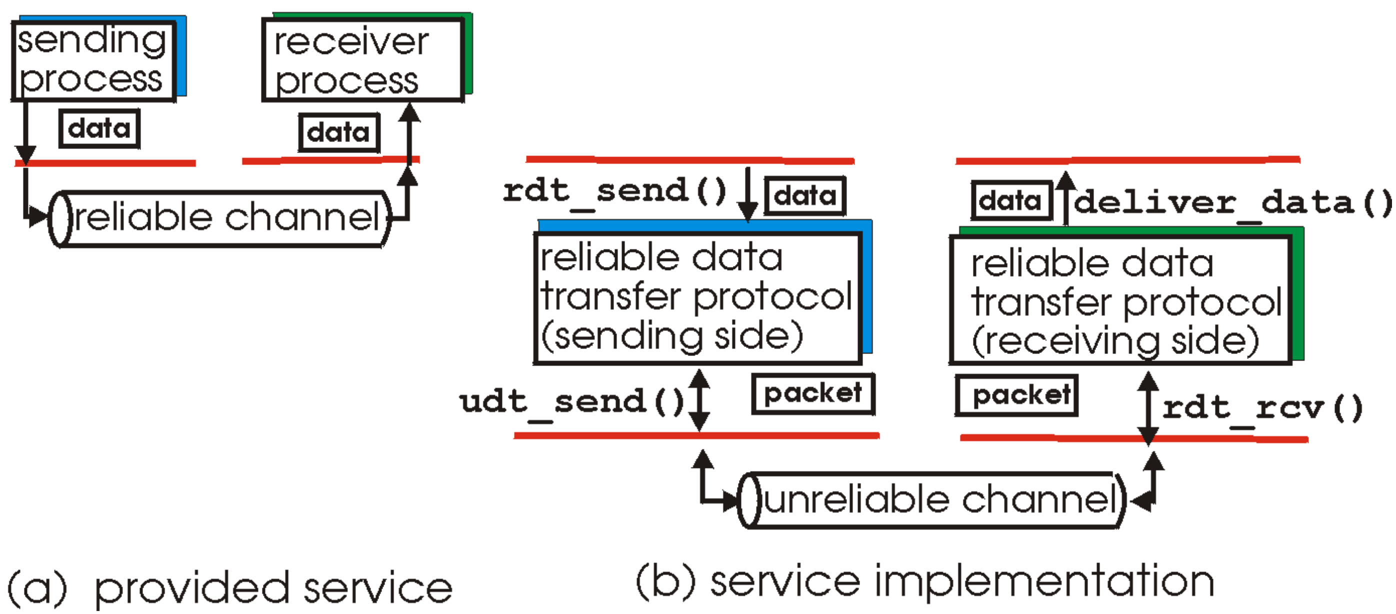
- Important in app., transport, link layers
- Top-10 list of important networking topics!



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

Principles of Reliable Data Transfer

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

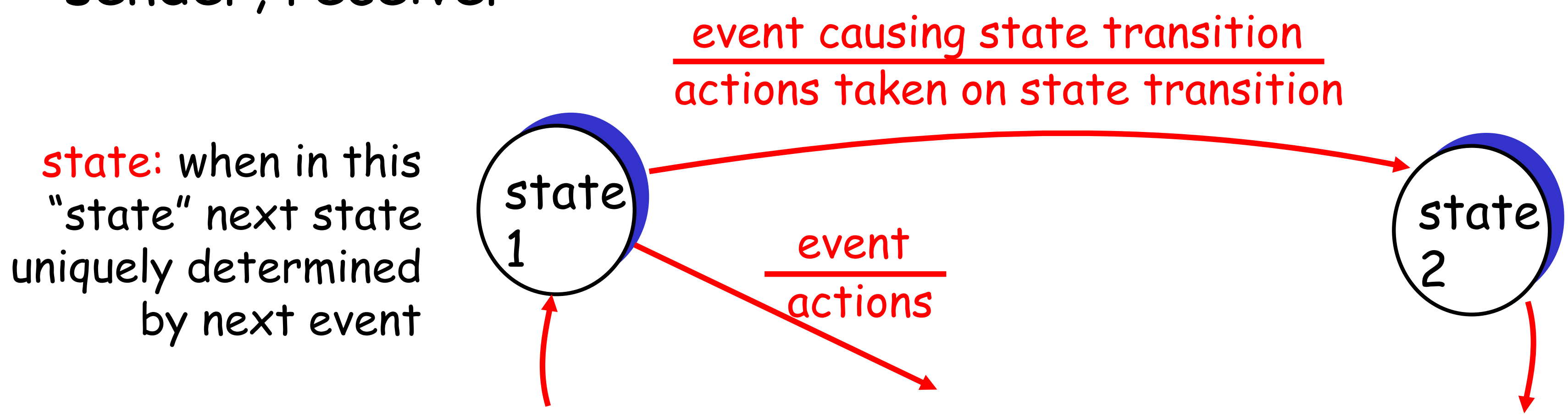


- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable Data Transfer: Getting Started

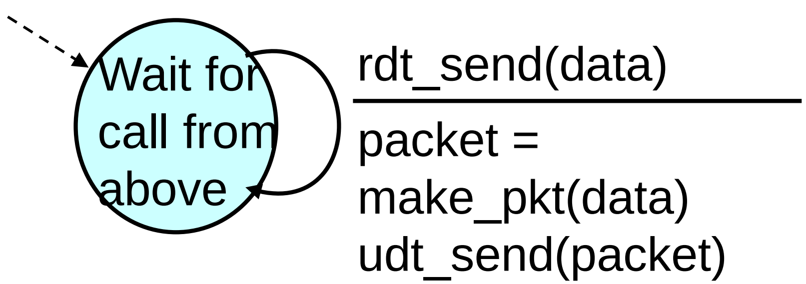
We'll:

- 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

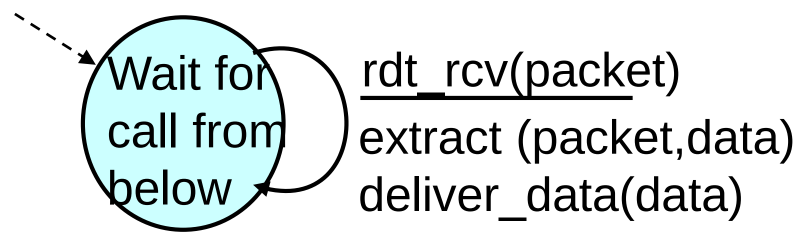


Rdt1.0: Reliable Transfer over a Reliable Channel

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



sender



receiver

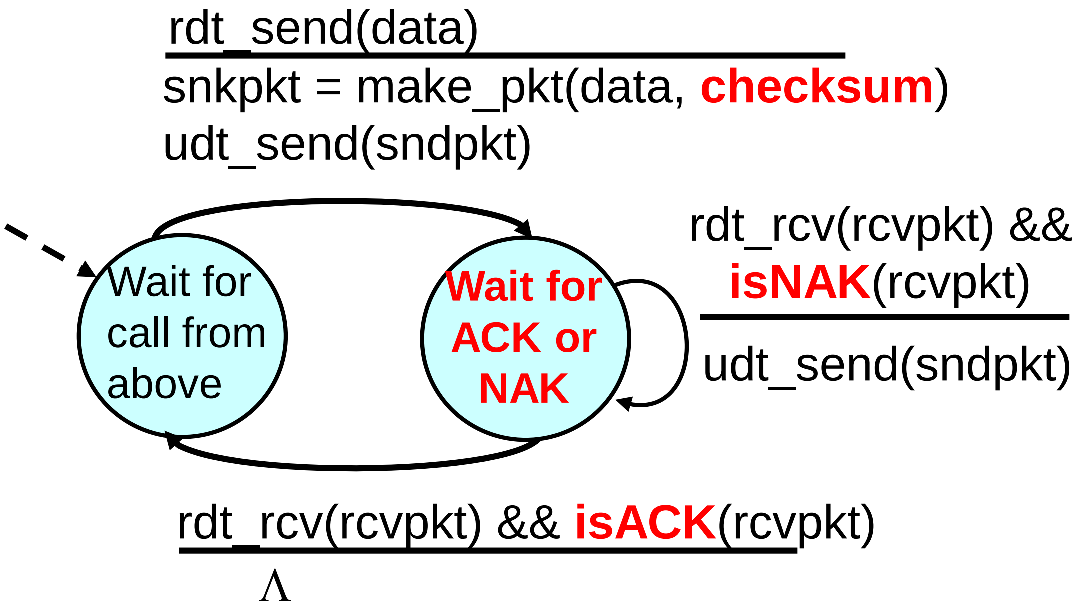
Rdt2.0: Channel with Bit Errors

- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- *The* question: how to recover from errors:
- How do humans recover from "errors" during conversation?

Rdt2.0: Channel with Bit Errors

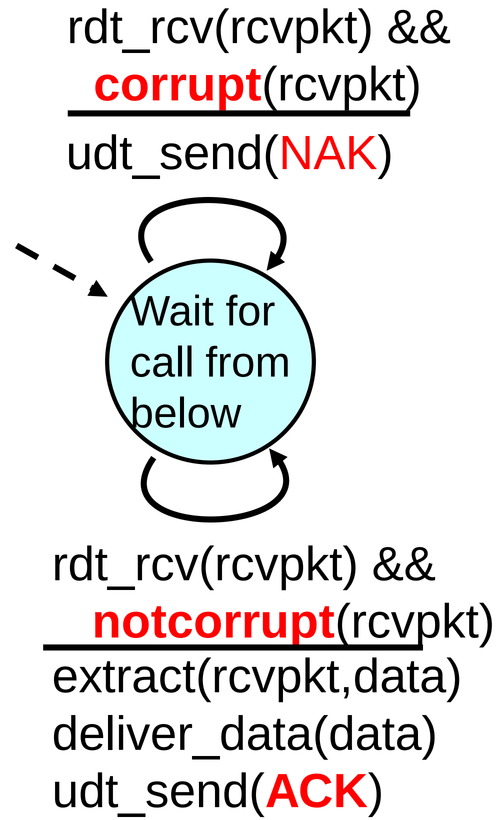
- *The question: how to recover from errors:*
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - Sender retransmits pkt on receipt of NAK
- New mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - Error detection
  - Receiver feedback: control msgs (ACK,NAK) rcvr->sender
  - Retransmission

Rdt2.0: FSM specification



sender

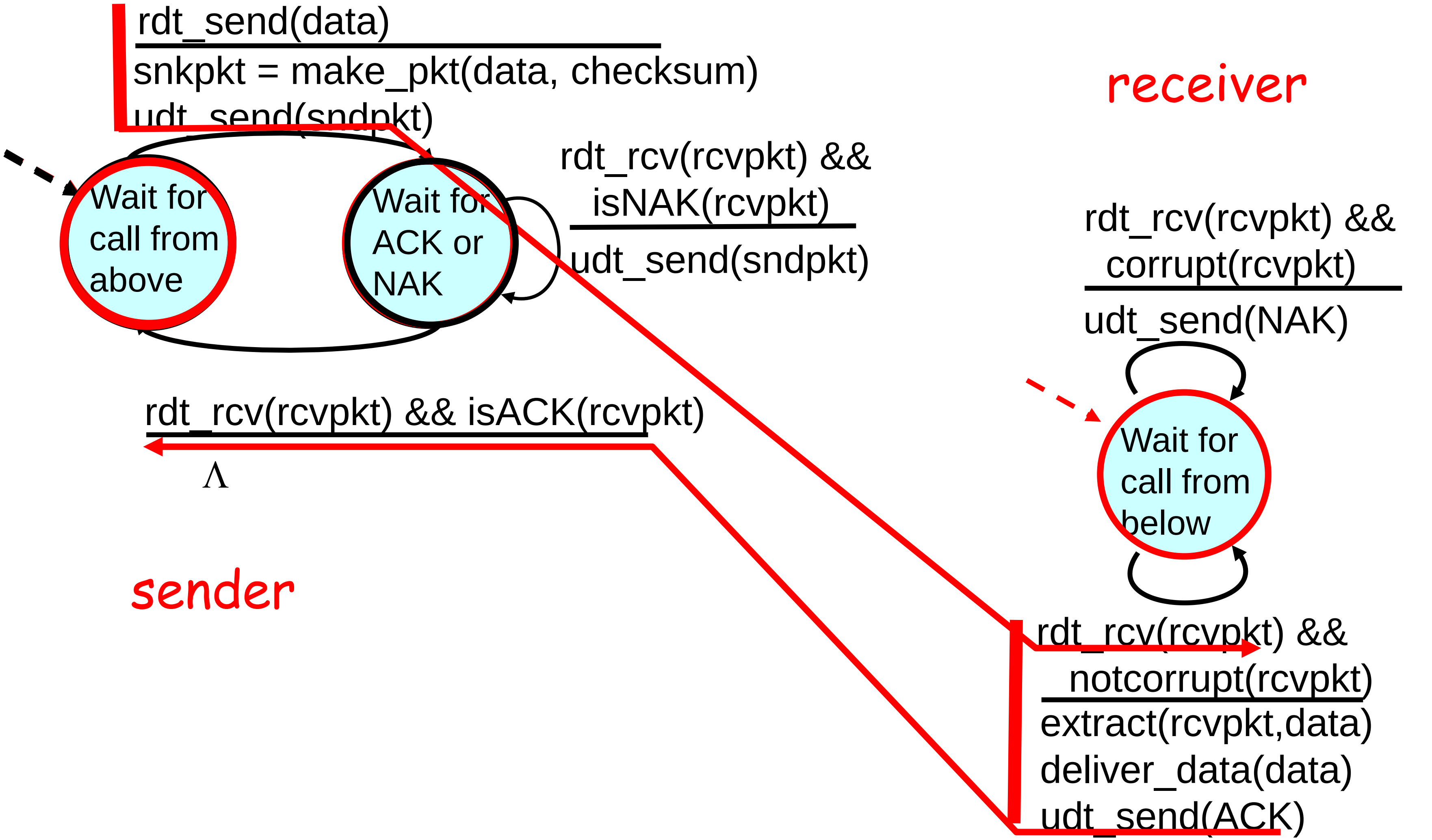
receiver



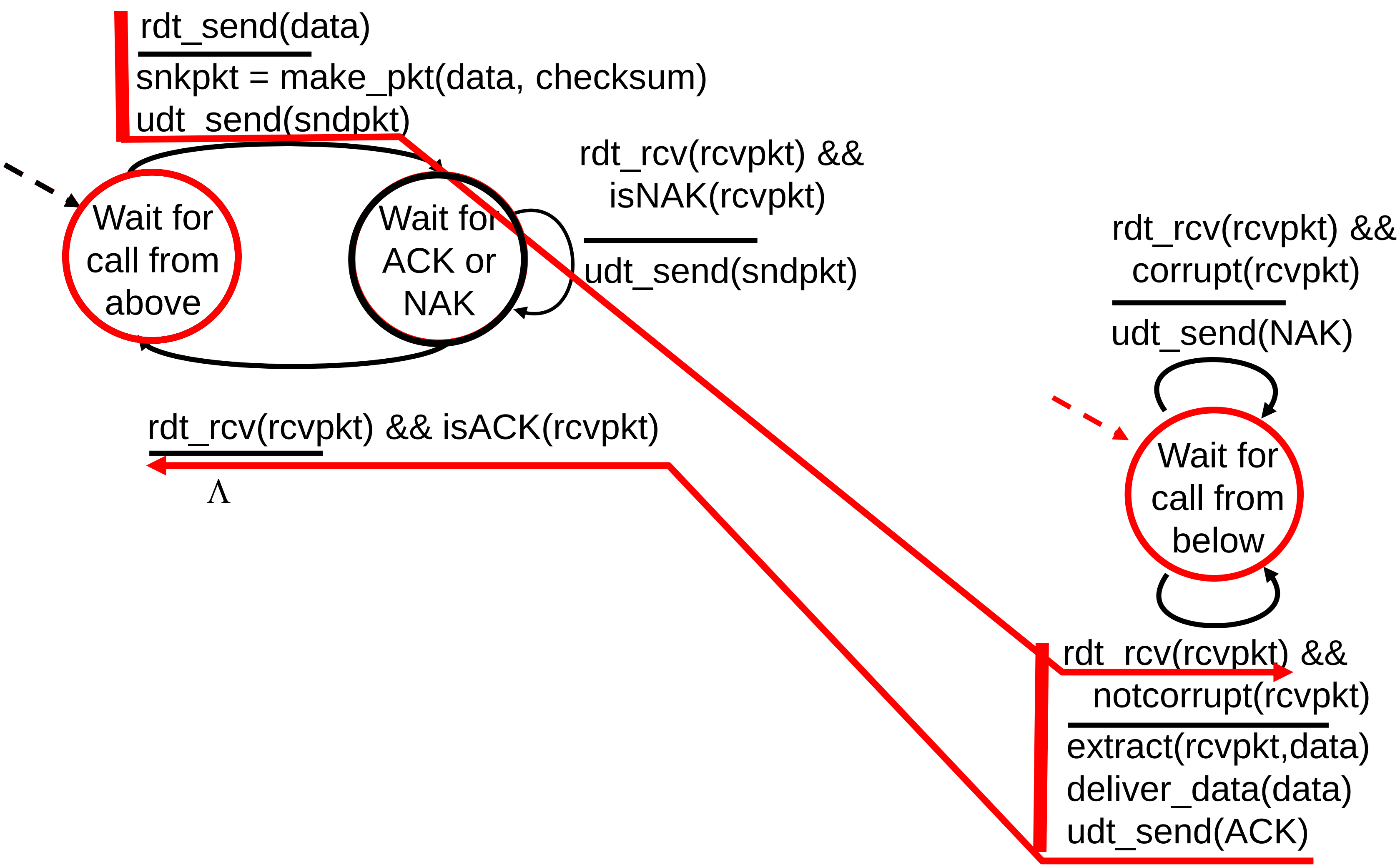
stop and wait

sender sends one packet, then waits for receiver response

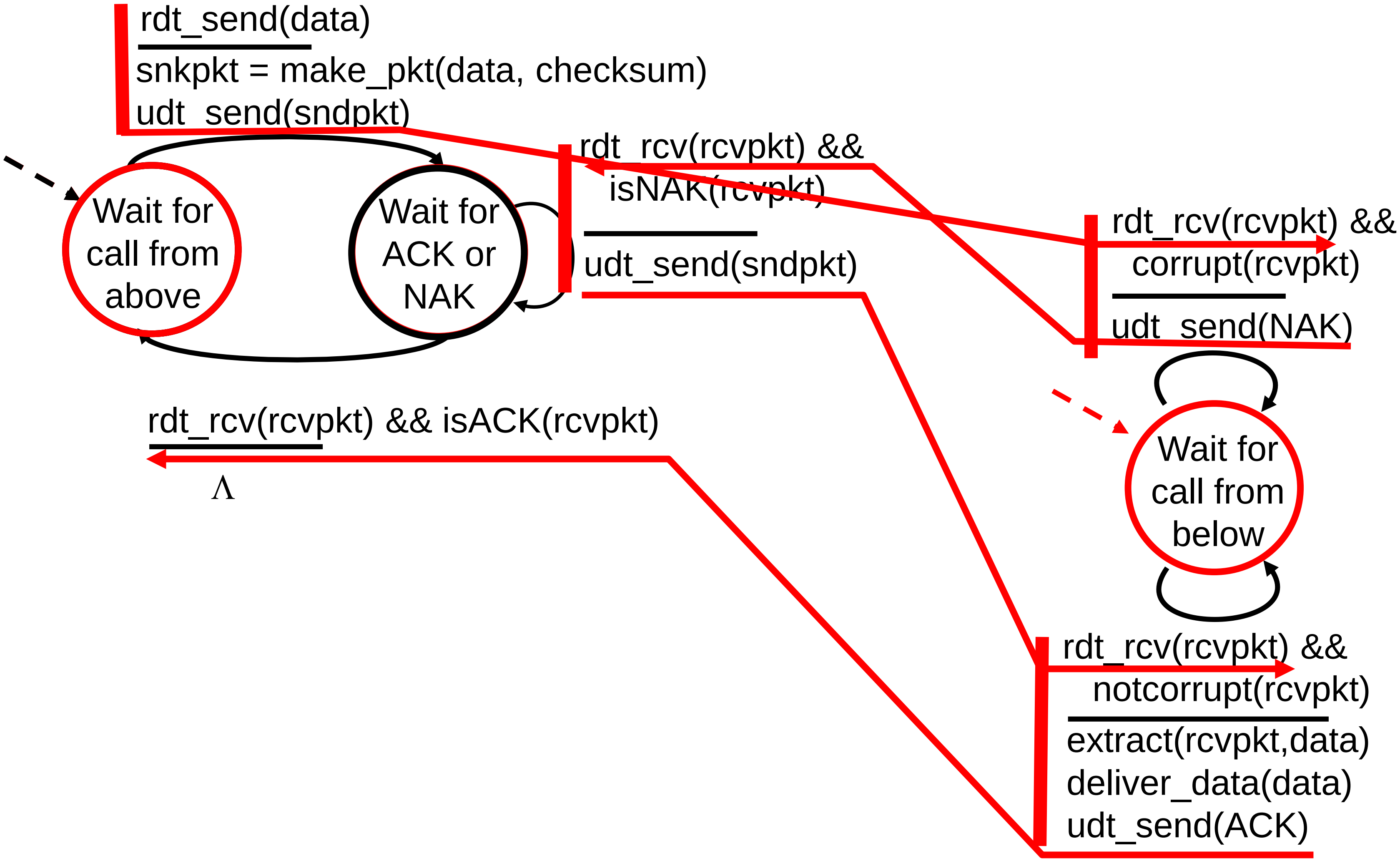
Rdt2.0: Operation with No Errors



Rdt2.0: operation with no errors



Rdt2.0: error scenario



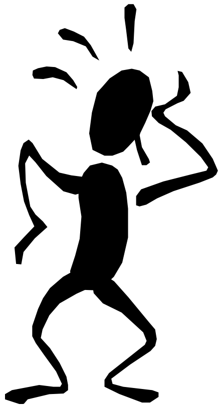
Rdt2.0 Has a Fatal Flaw!

What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- Can't just retransmit: possible duplicate

Handling duplicates:

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

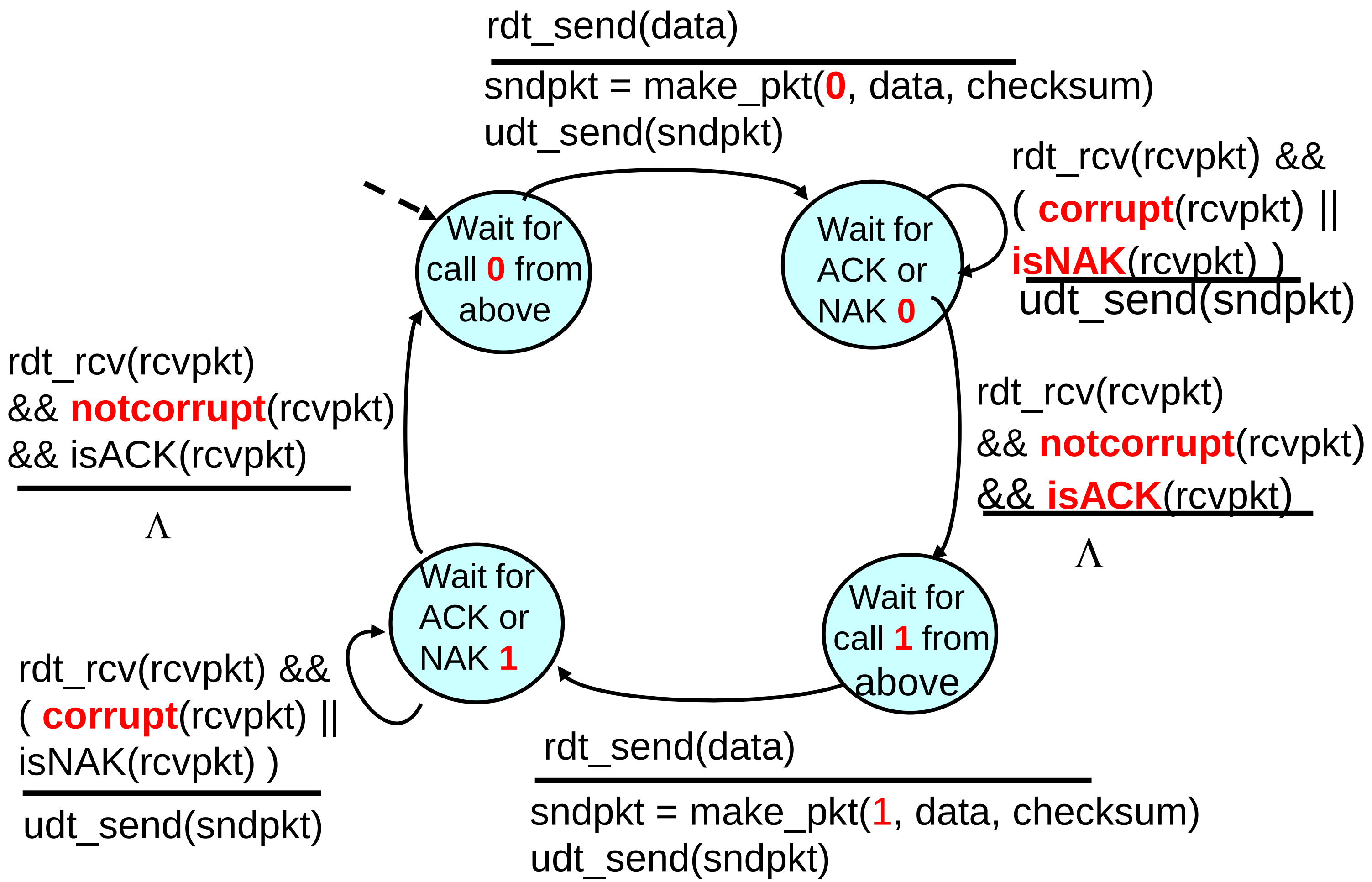


- Sender: whenever sender receives control message it sends a packet to receiver
  - A valid ACK: Sends next packet (if exists) with new sequence #
  - A NAK or corrupt response: resends old packet

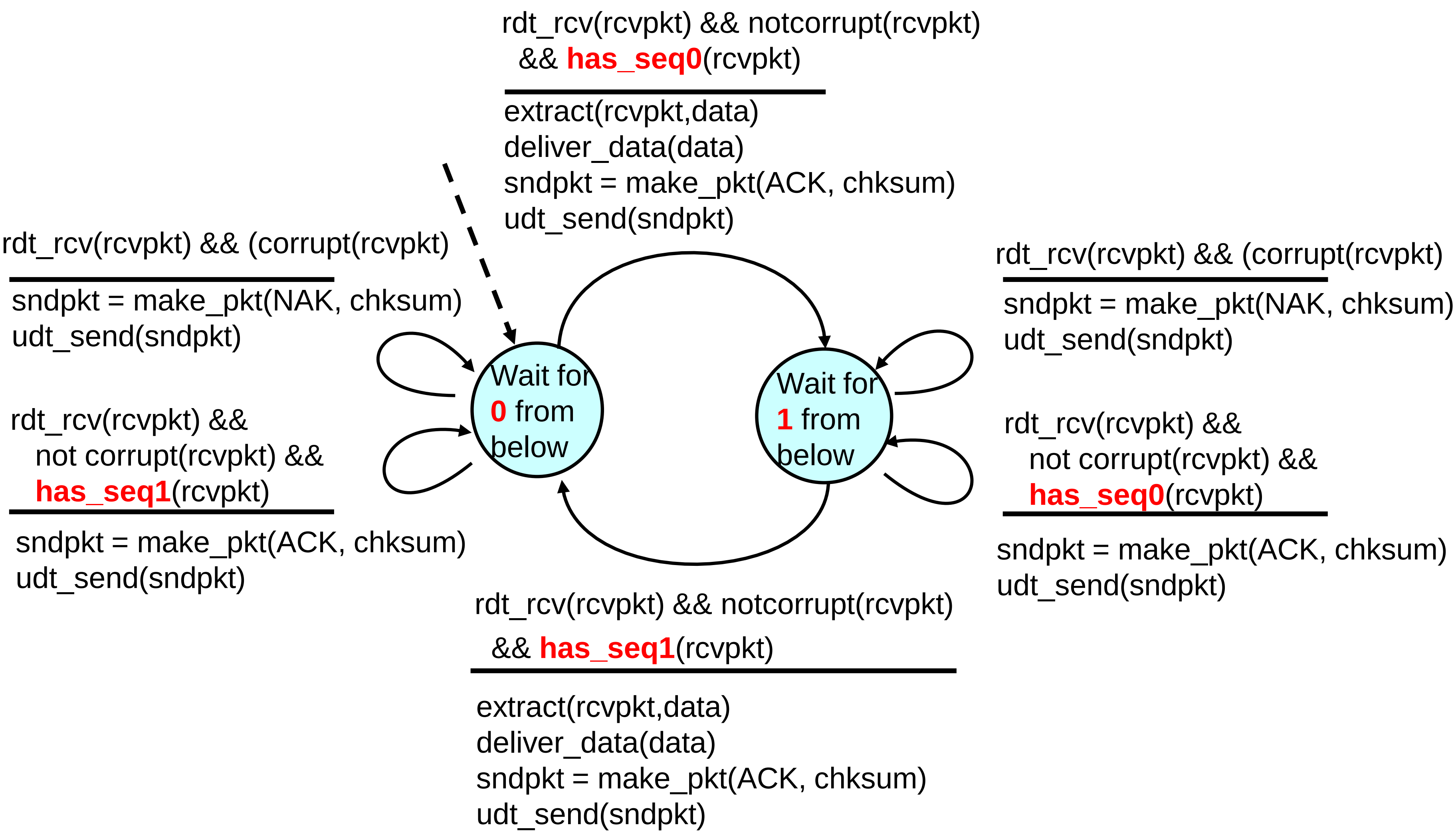
- Receiver: sends ACK/NAK to sender
  - If received packet is corrupt: send NAK
  - If received packet is valid and has different sequence # as prev packet: send ACK and deliver new data up
  - If received packet is valid and has same sequence # as prev packet, i.e., is a retransmission of duplicate: send ACK

- Note: ACK/NAK **do not** contain sequence #

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

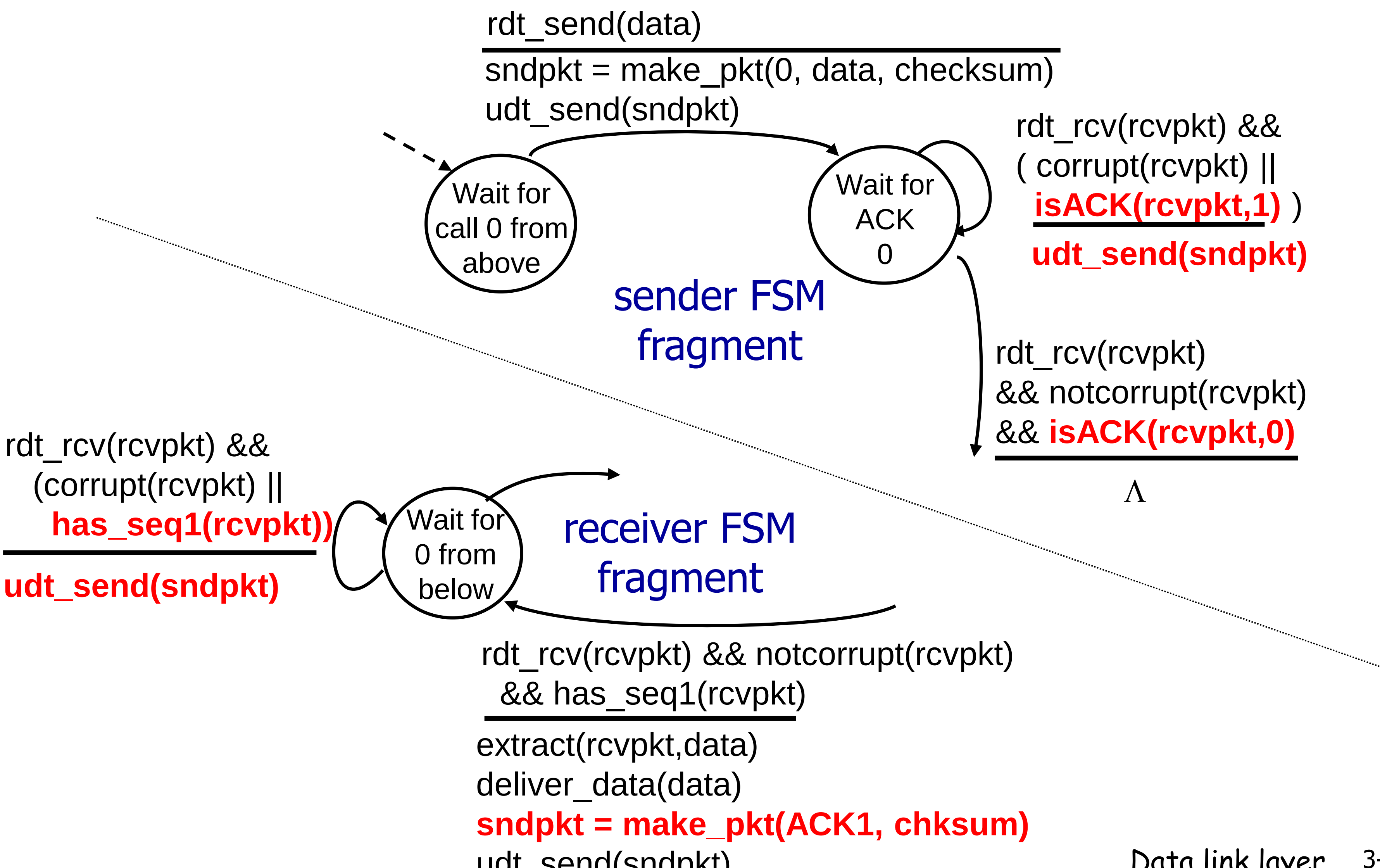
Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can *not* know if its last ACK/NAK received OK at sender

Rdt2.2: a NAK-free Protocol

- Same functionality as rdt2.1, 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



Rdt3.0: Channels with Errors and Loss

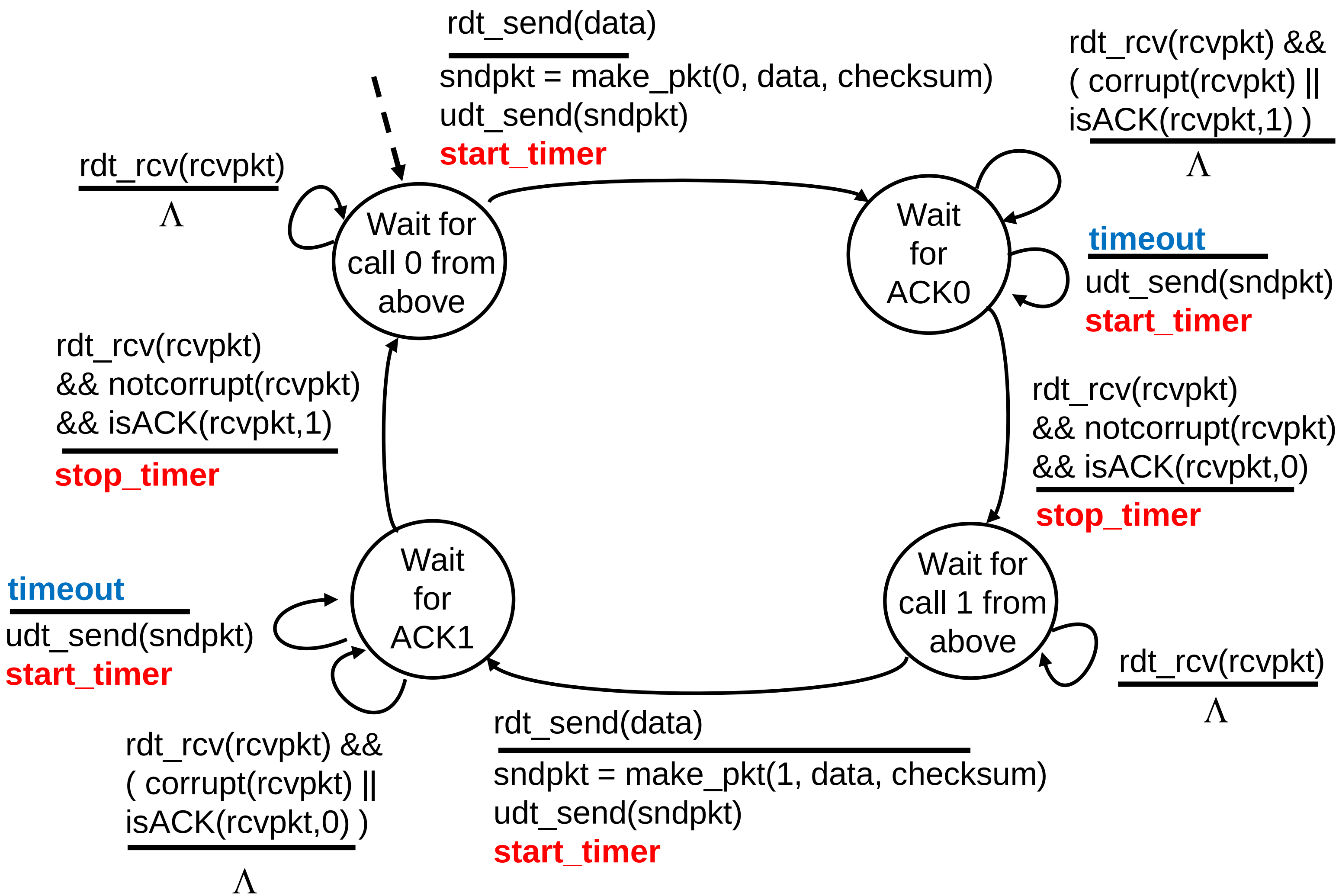
New assumption: Underlying channel can also lose packets (data or ACKs)

- Checksum, seq. #, ACKs, retransmissions will be of help, but not enough
- Data lost?
- ACK lost?

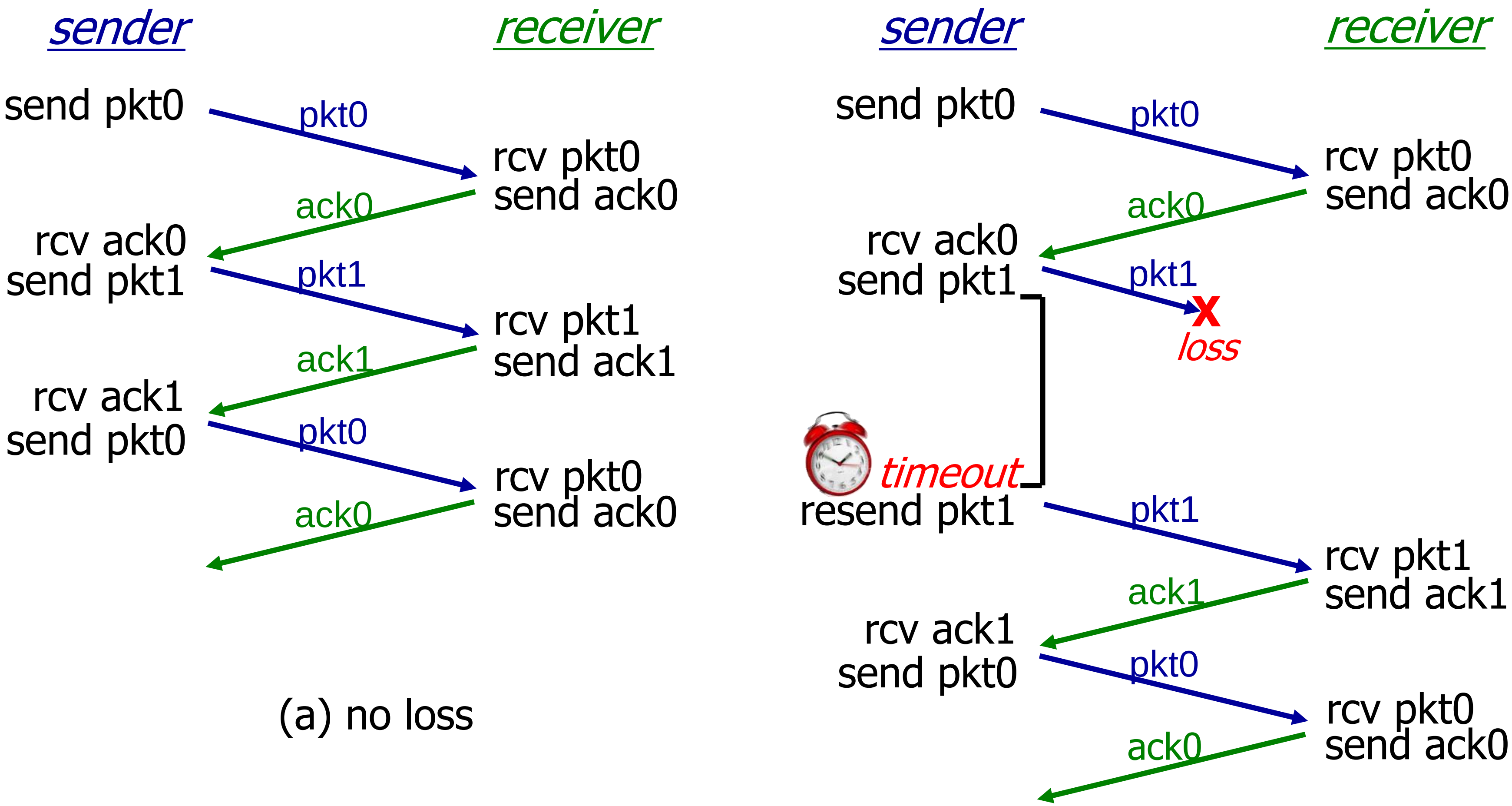
Approach: sender waits "reasonable" amount of time for ACK

- 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
- Requires countdown timer

Rdt3.0 sender



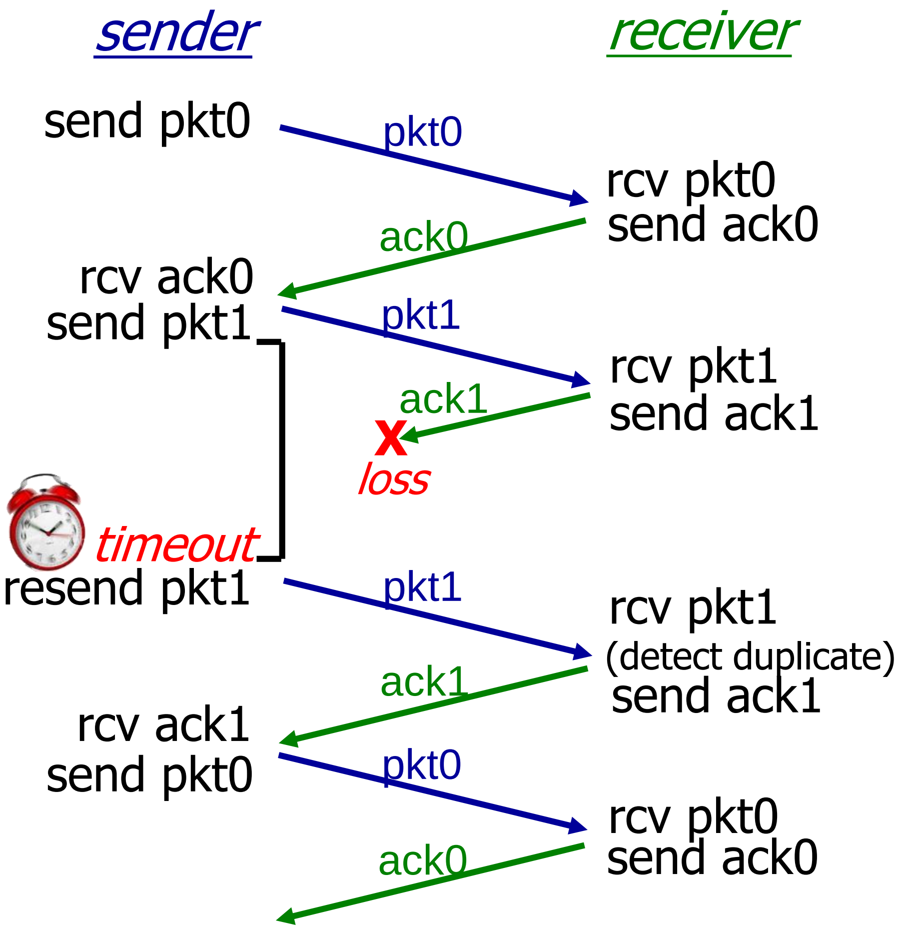
rdt3.0 in action



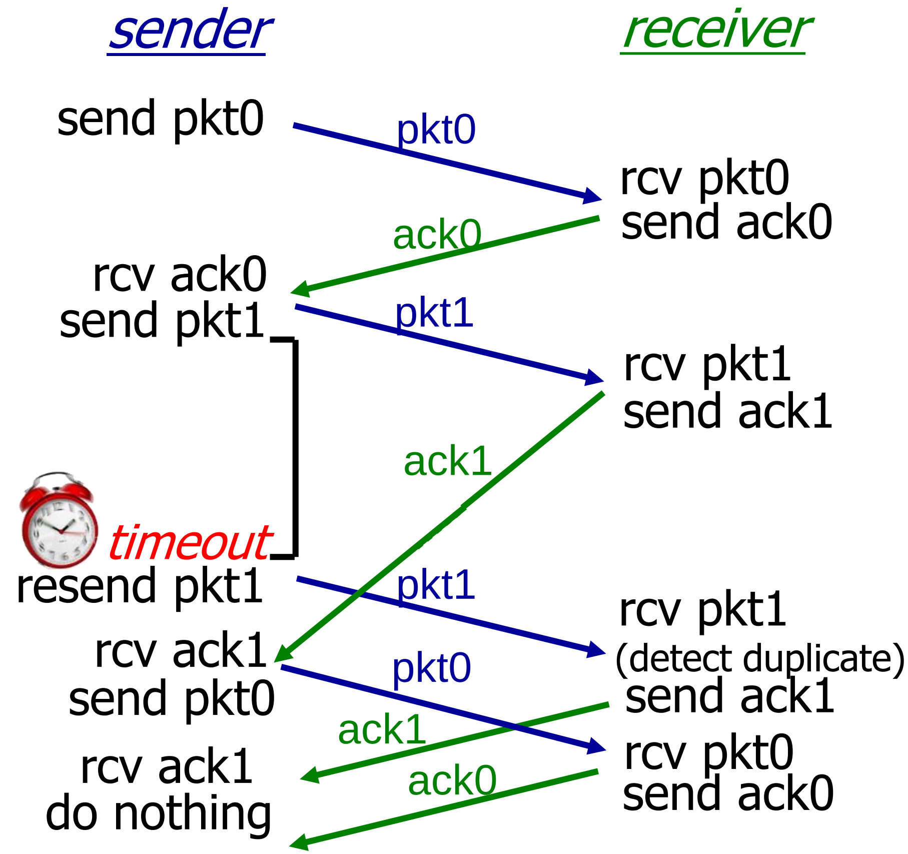
(a) no loss

(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Summary of the Protocols

- Rdt1.0: all packets arrive correctly
- Rdt2.0: all packets arrive, with possible errors only in data packets, and introducing ACK and NAK (no error)
- Rdt2.1: corrupted ACKs/NAKs
- Rdt2.2: similar to rdt2.1 remove NAKs
- Rdt3.0: Allows packets to be lost and errors

Performance of rdt3.0

- Rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

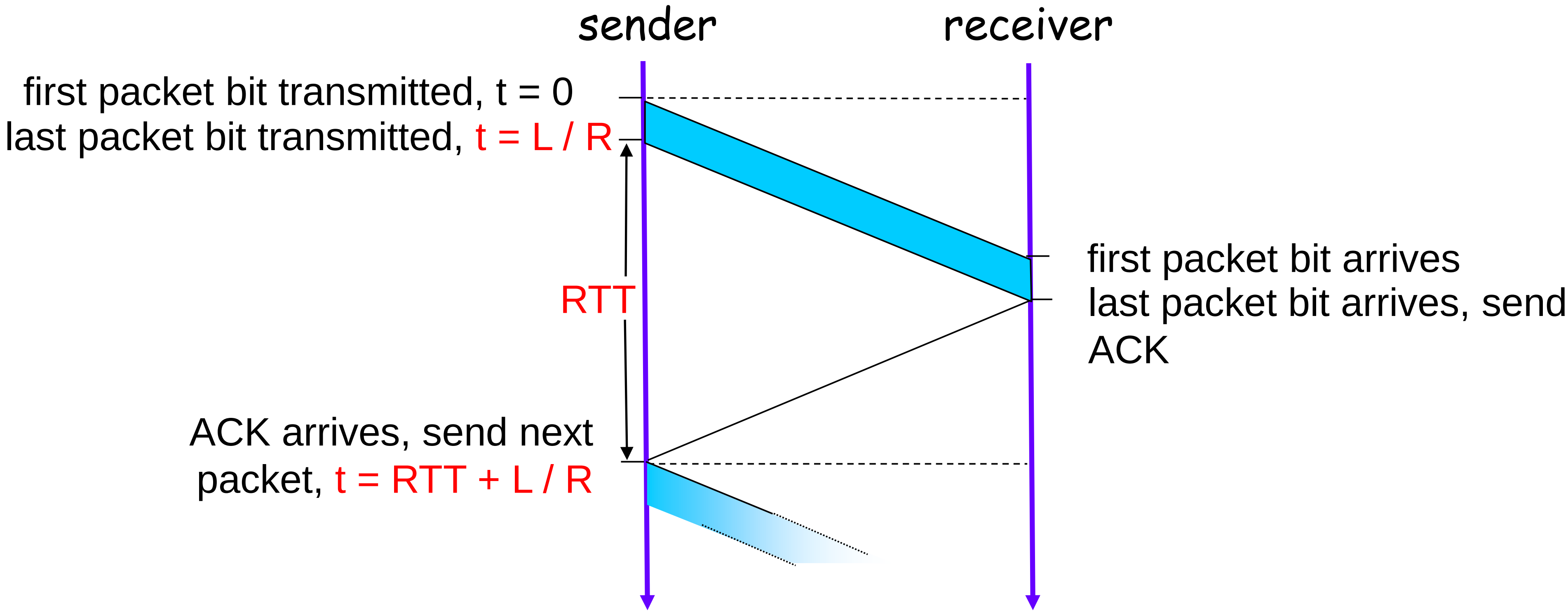
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

  - U_{sender}: utilization – fraction of time sender busy sending

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

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

Rdt3.0: Stop-and-Wait Operation

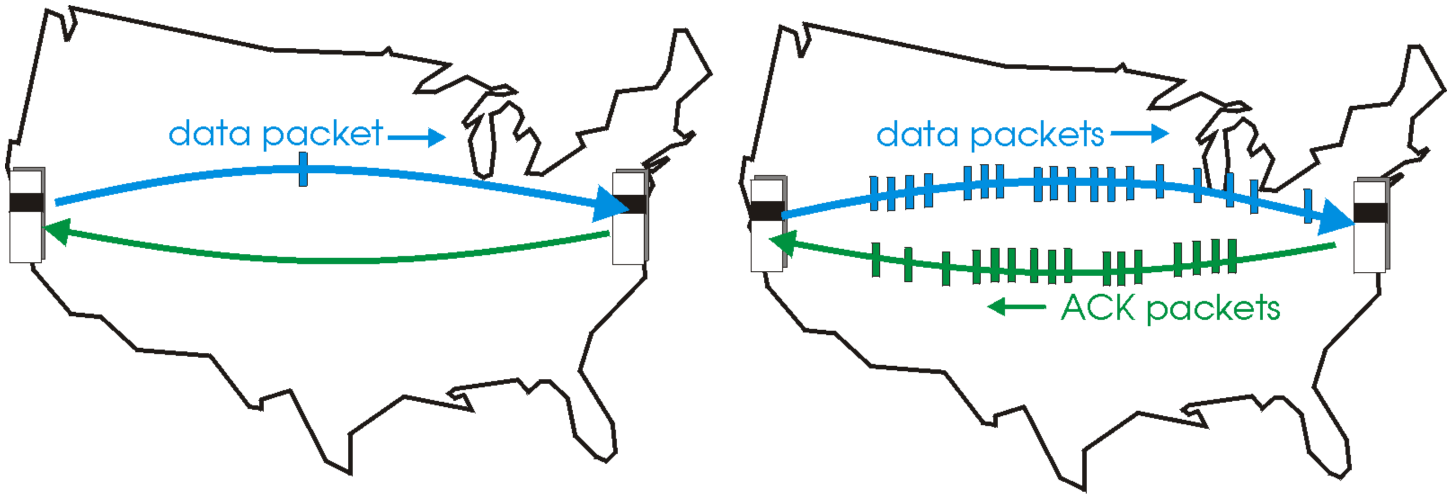


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined Protocols

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

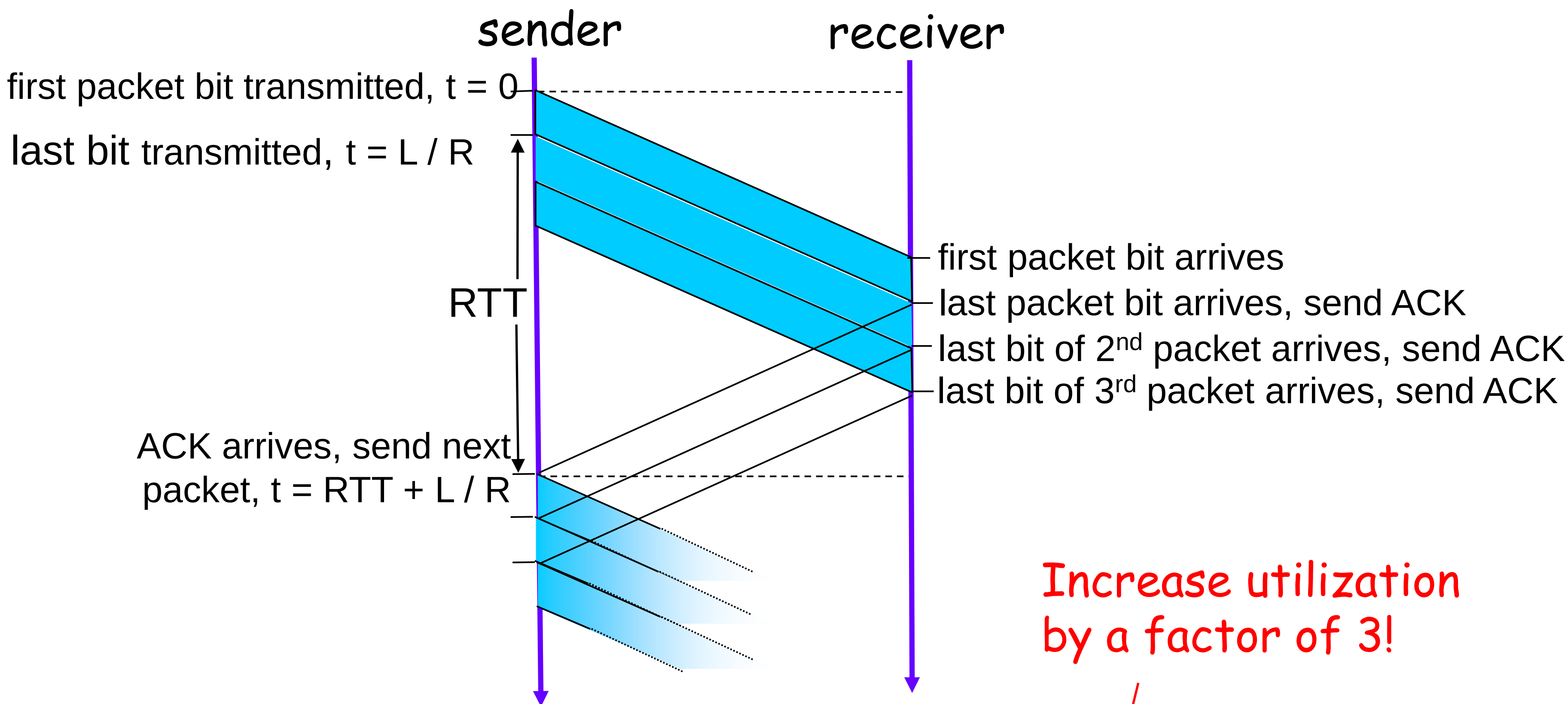
(b) a pipelined protocol in operation

Pipelined Protocols

- Advantage: much better bandwidth utilization than stop-and-wait
- Disadvantage: More complicated to deal with reliability issues, e.g., corrupted, lost, out of order data.
  - Two generic approaches to solving this
    - *Go-Back-N protocols*
    - *Selective repeat protocols*

- Note: *TCP is not exactly either*

Pipelining: Increased Utilization



Increase utilization by a factor of 3!

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

Pipelined protocols: overview

Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

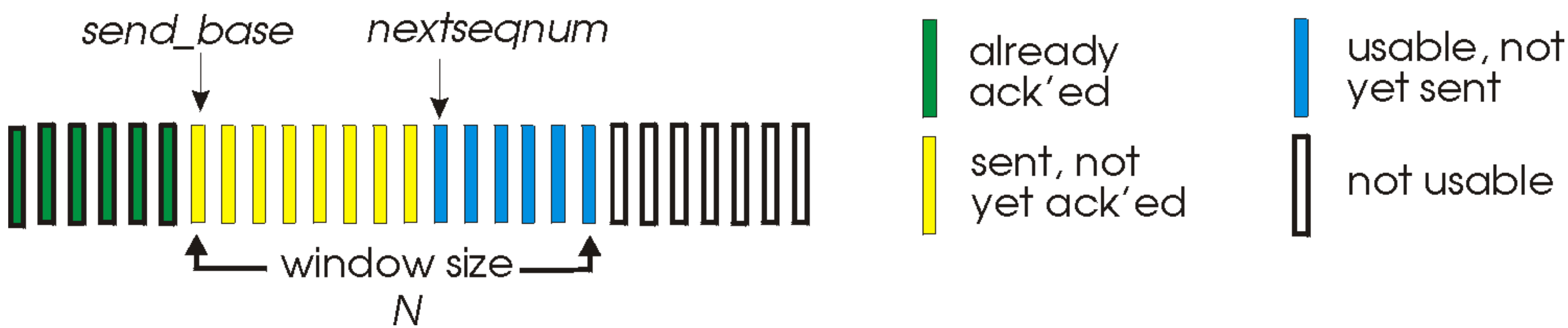
Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N

Sender:

- k-bit seq # in pkt header
- "Window" of up to N, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - May receive duplicate ACKs (see receiver)
- Timer for oldest in-flight pkt
- *Timeout(n)*: retransmit pkt n and all higher seq # pkts in window
- Called a sliding-window protocol

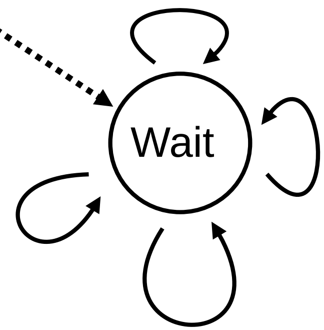
GBN: sender extended FSM

rdt_send(data)

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

Λ

base=0
nextseqnum=0



timeout

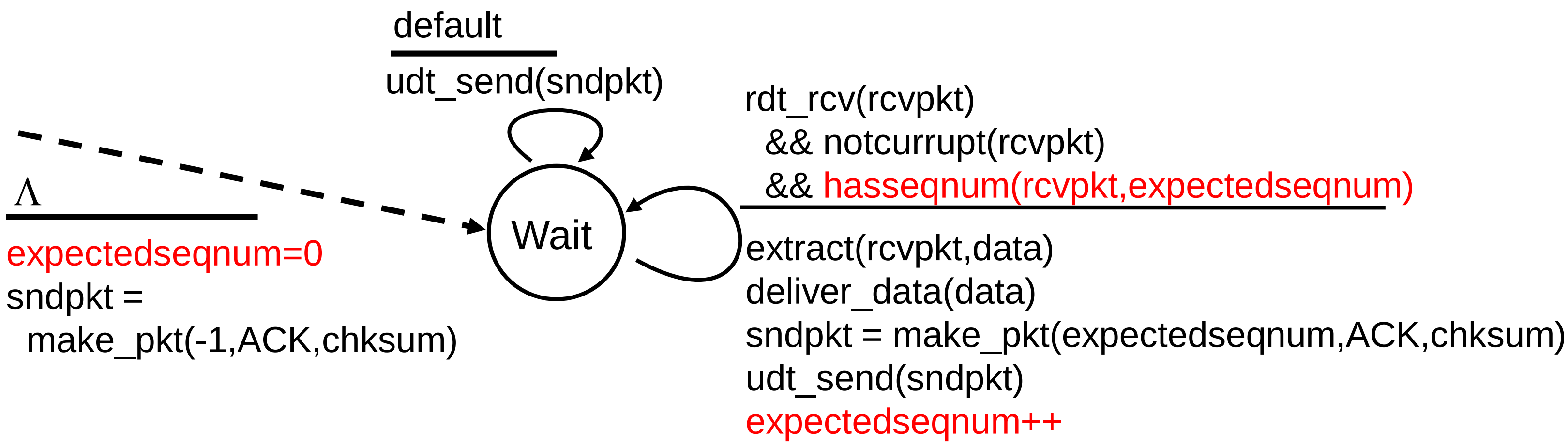
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

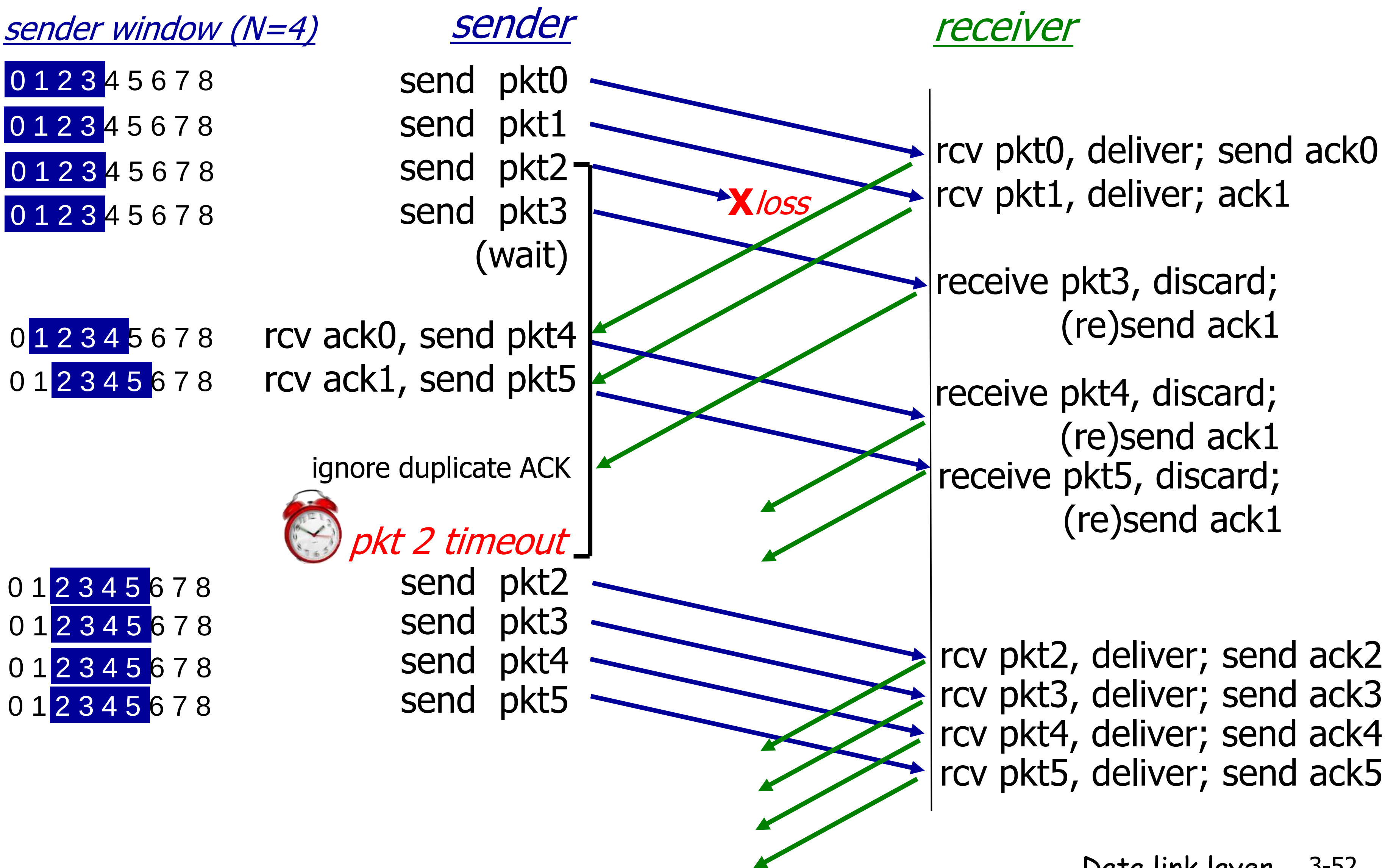
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
else
    start_timer

GBN: receiver extended FSM



- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember **expectedseqnum**
- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

GBN in action



Exercise

- Via the GBN protocol, a sender sent packets with seq # from 0 to 5. If the sender has only received ACK0 and ACK2 when a timeout happens, which packets should the sender retransmit?

- GBN is easy to code but might have performance problems

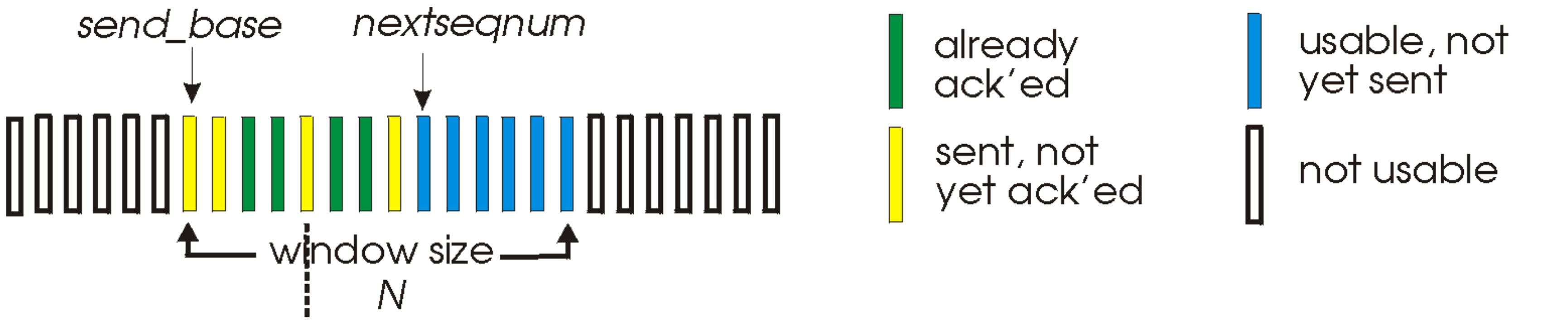
- In particular, if many packets are in pipeline at one time (bandwidth-delay product large) then one error can force retransmission of huge amounts of data!

- Selective Repeat protocol allows receiver to buffer data and only forces retransmission of required packets

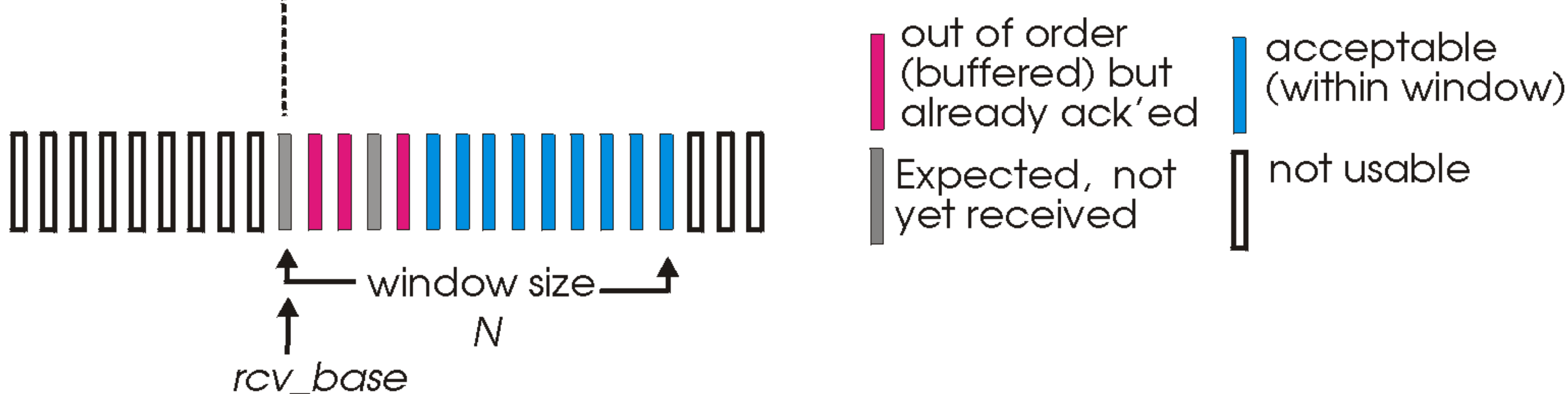
Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
  - Sender timer for each unACKed pkt
  - Compare to GBN which only had timer for base packet
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed pkts

Selective Repeat: Sender, Receiver Windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective Repeat

sender

Data from above :

- If next available seq # in window, send pkt

Timeout(n):

- Resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

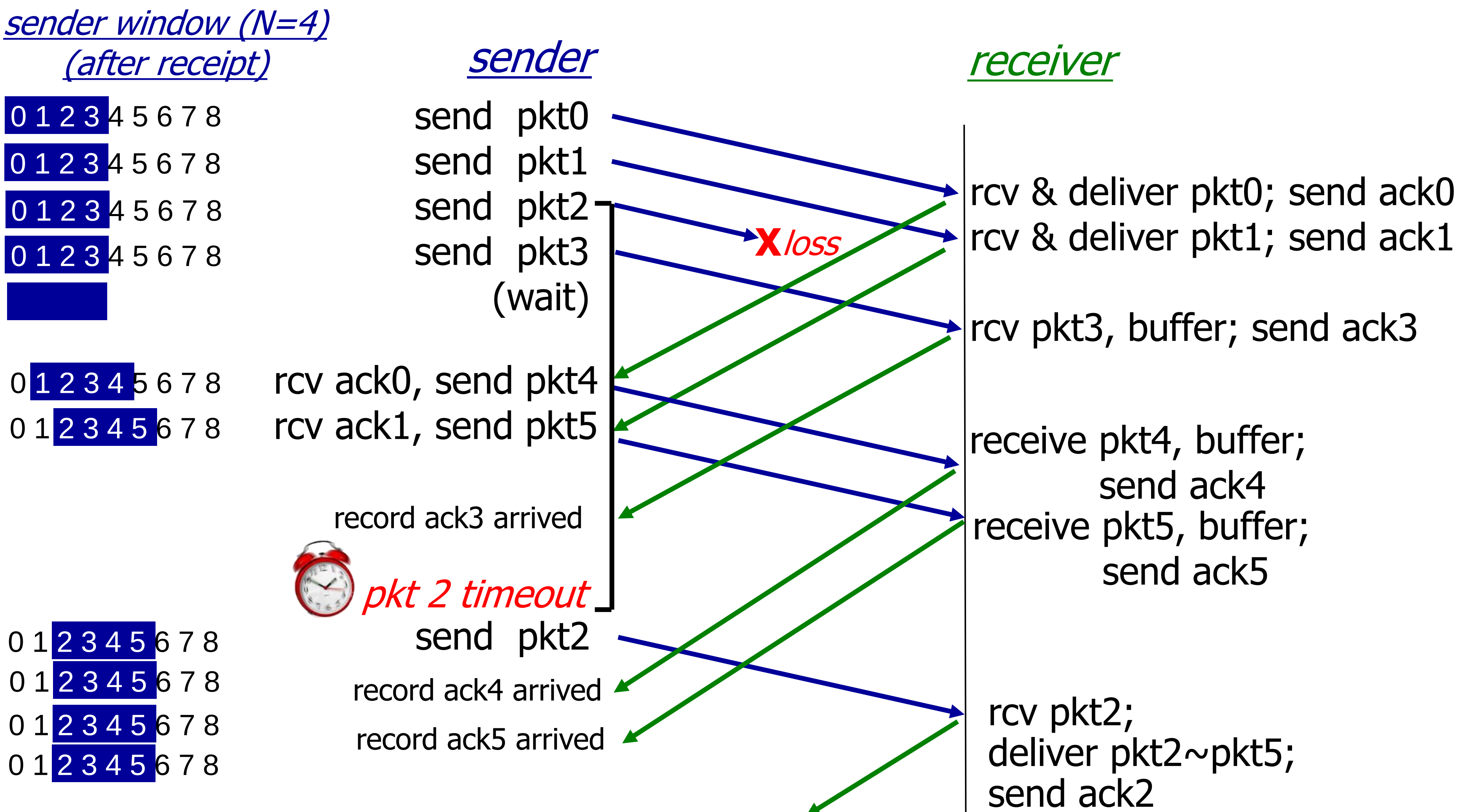
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

Otherwise:

- Ignore

Selective repeat in action



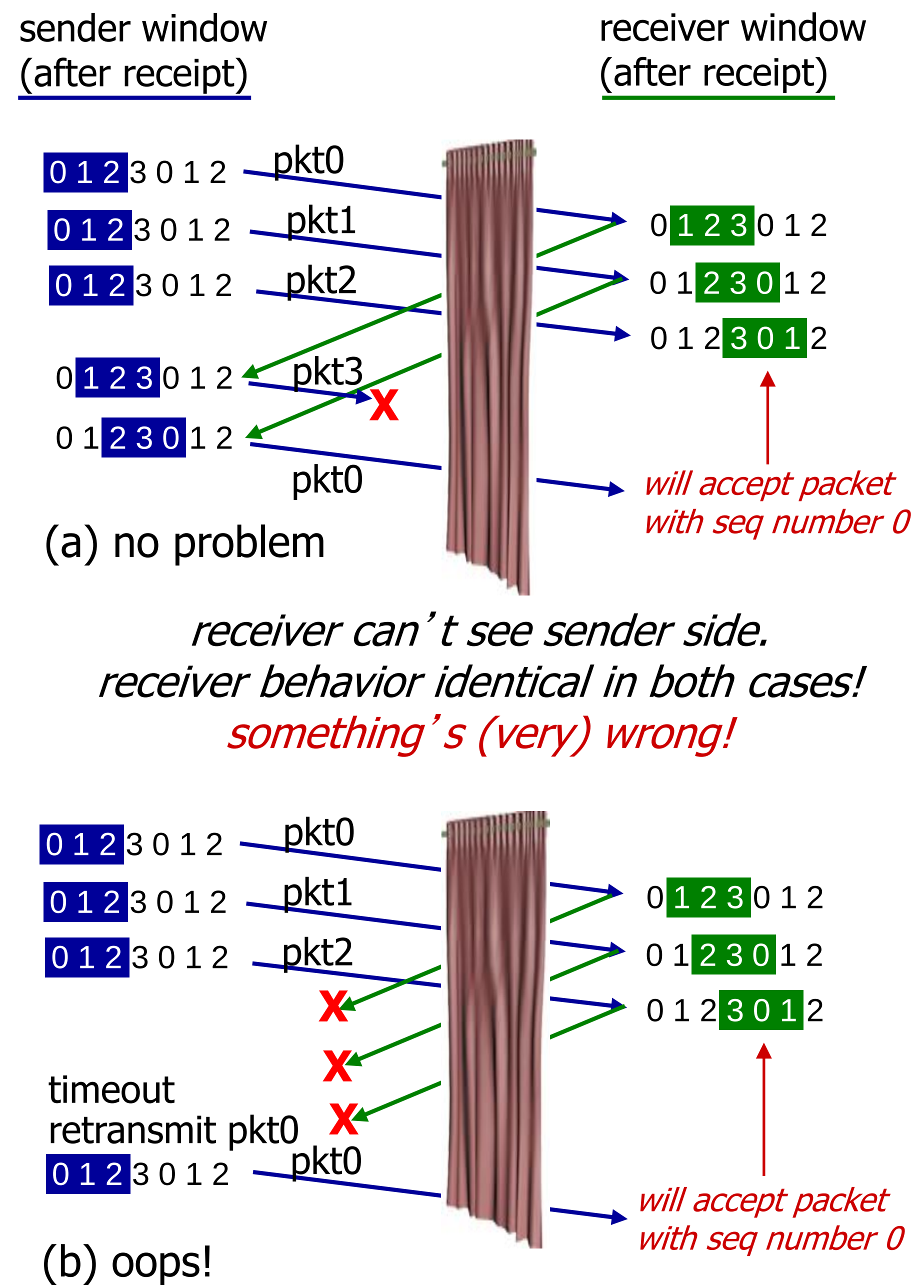
Q: what happens when ack2 arrives?

Selective repeat: dilemma

example:

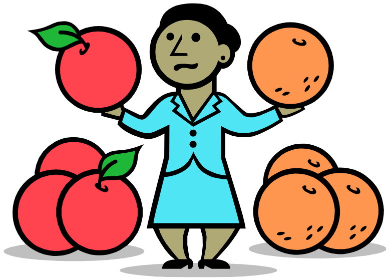
- ❒ seq #'s: 0, 1, 2, 3
- ❒ window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



GBN vs. Selective Repeat

- Selective repeat is more complicated as it needs buffering at the receiver, but only retransmit packets required for retransmission



- GBN is simpler, but can lead to large number of unnecessary retransmission