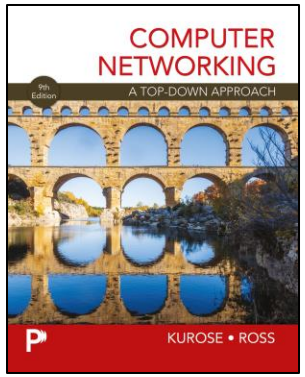


CSC 361
Computer Networks
Transport Layer
(Chapter 3 – UDP & RDT)

Wenjun Yang

Summer 2026

Chapter 3: Roadmap (2 of 7)

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

UDP: User Datagram Protocol (1 of 2)

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP: User Datagram Protocol (2 of 2)

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

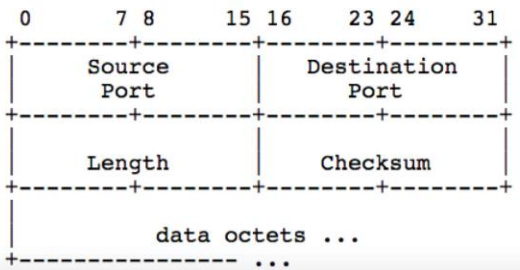
User Datagram Protocol

Introduction

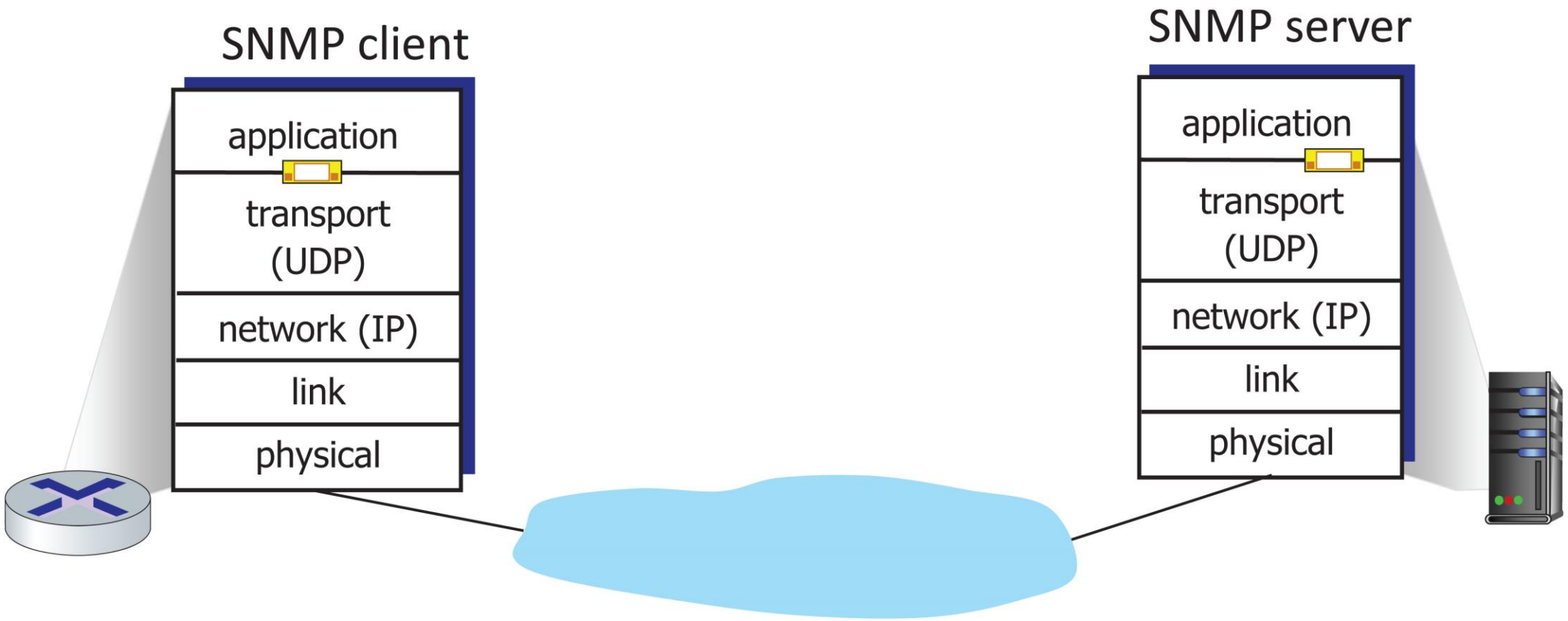
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format



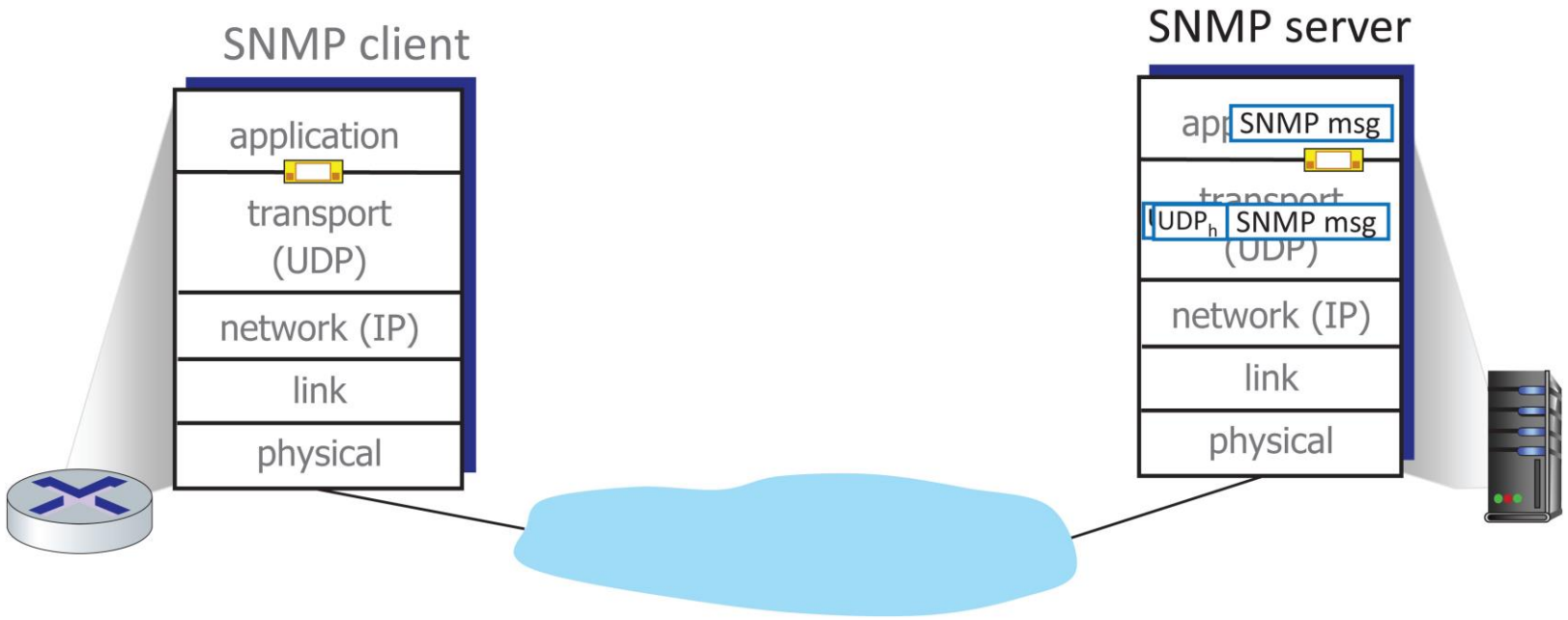
UDP: Transport Layer Actions (1 of 3)



UDP: Transport Layer Actions (2 of 3)

UDP sender actions:

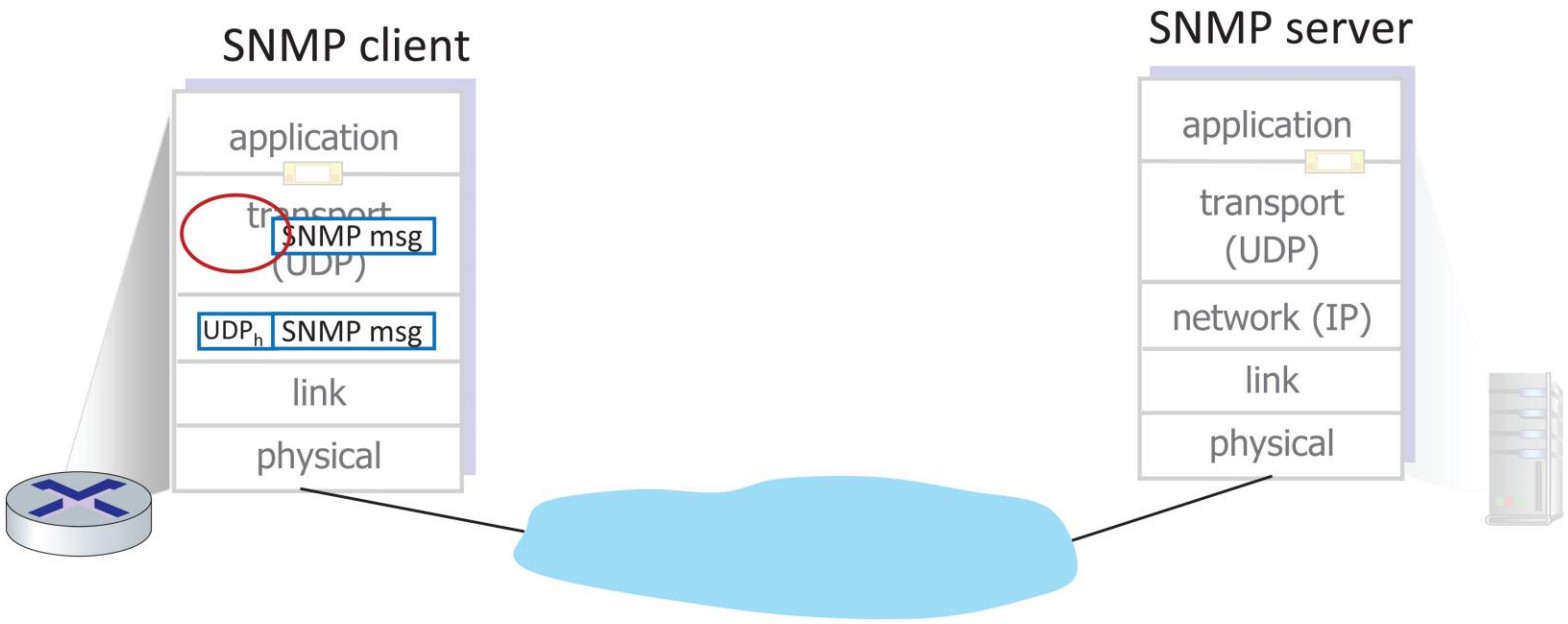
- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP



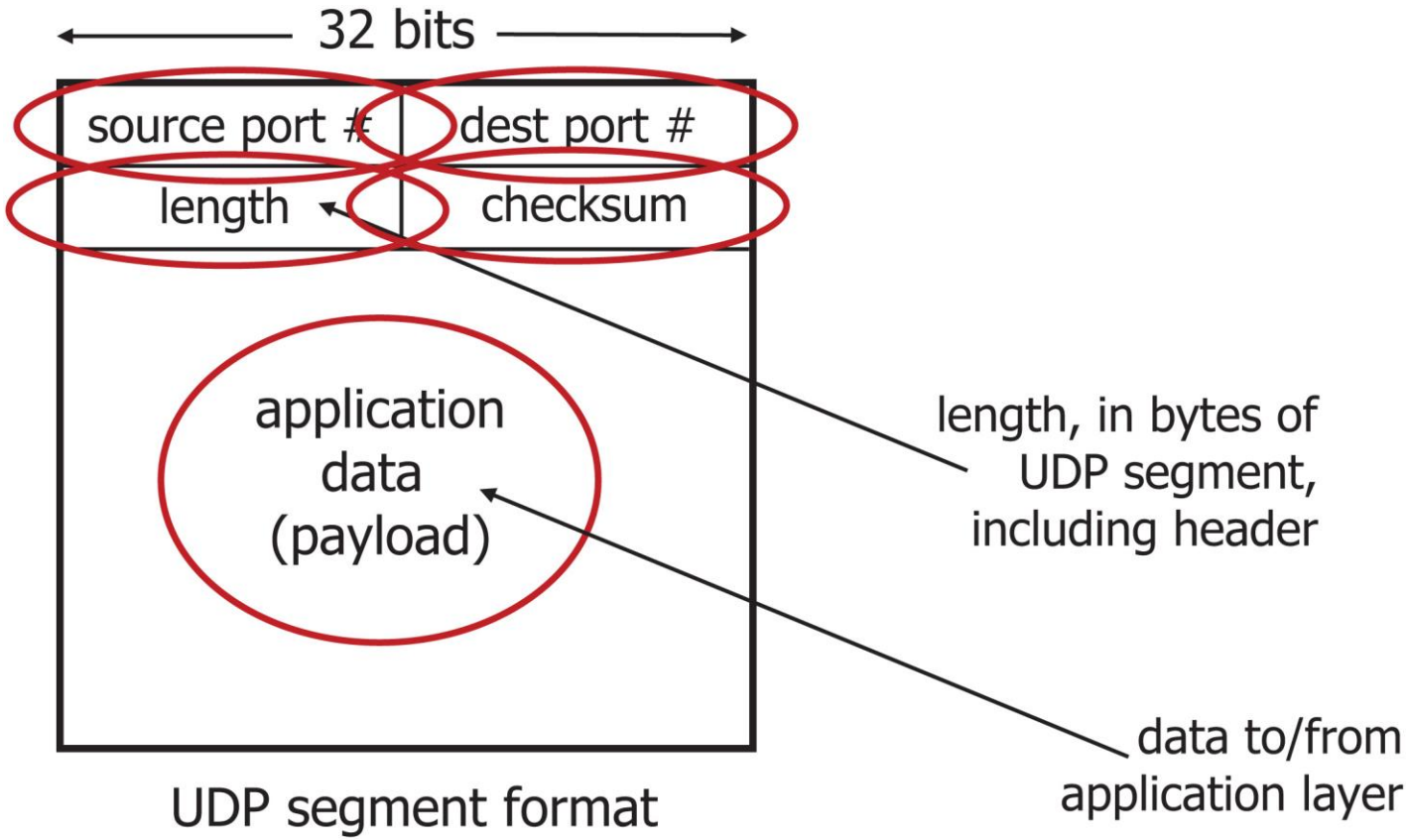
UDP: Transport Layer Actions (3 of 3)

UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket



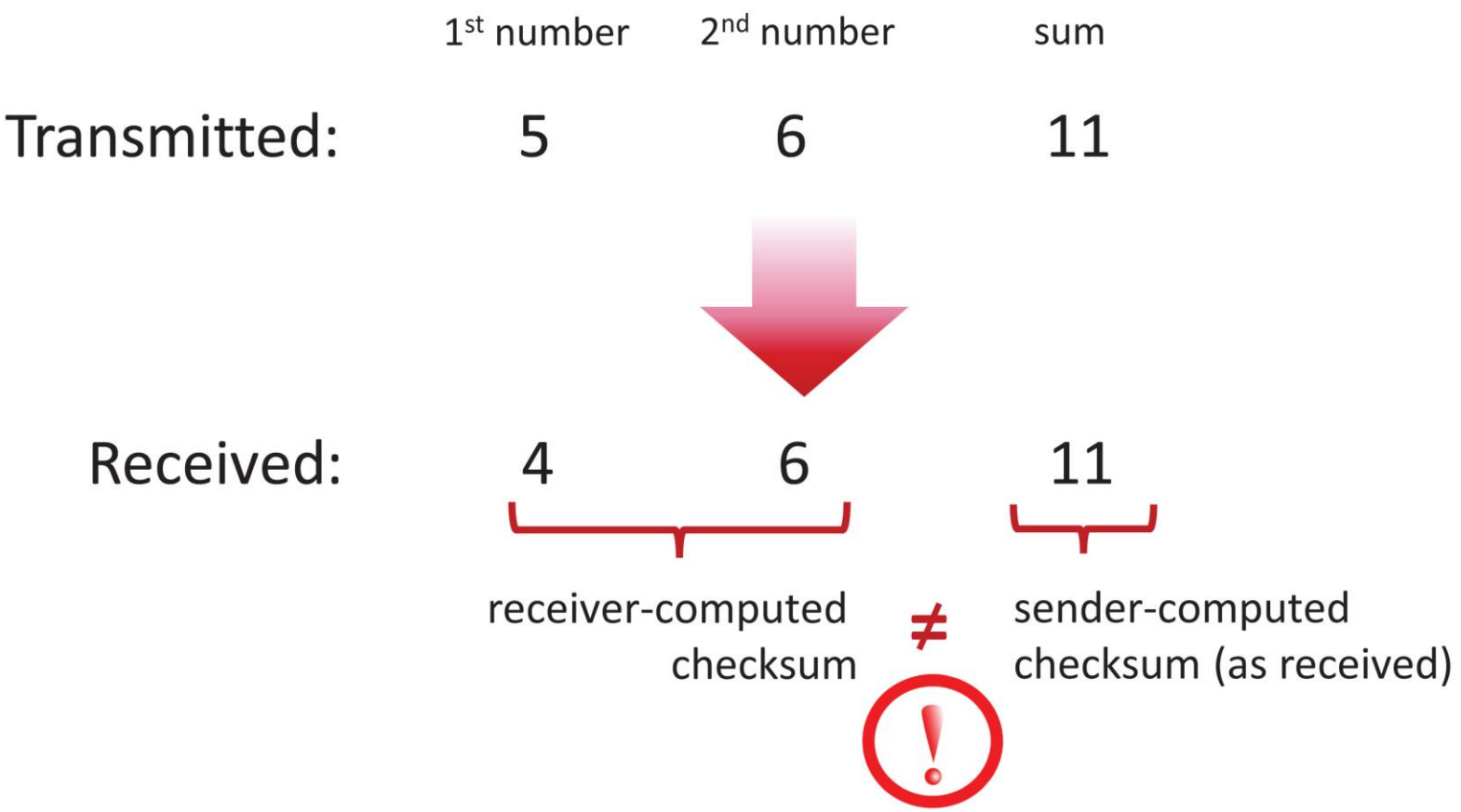
UDP Segment Header



UDP segment format

UDP Checksum

Goal: detect errors (**i.e.**, flipped bits) in transmitted segment



Internet Checksum

Goal: detect errors (**i.e.,** flipped bits) in transmitted segment

sender:

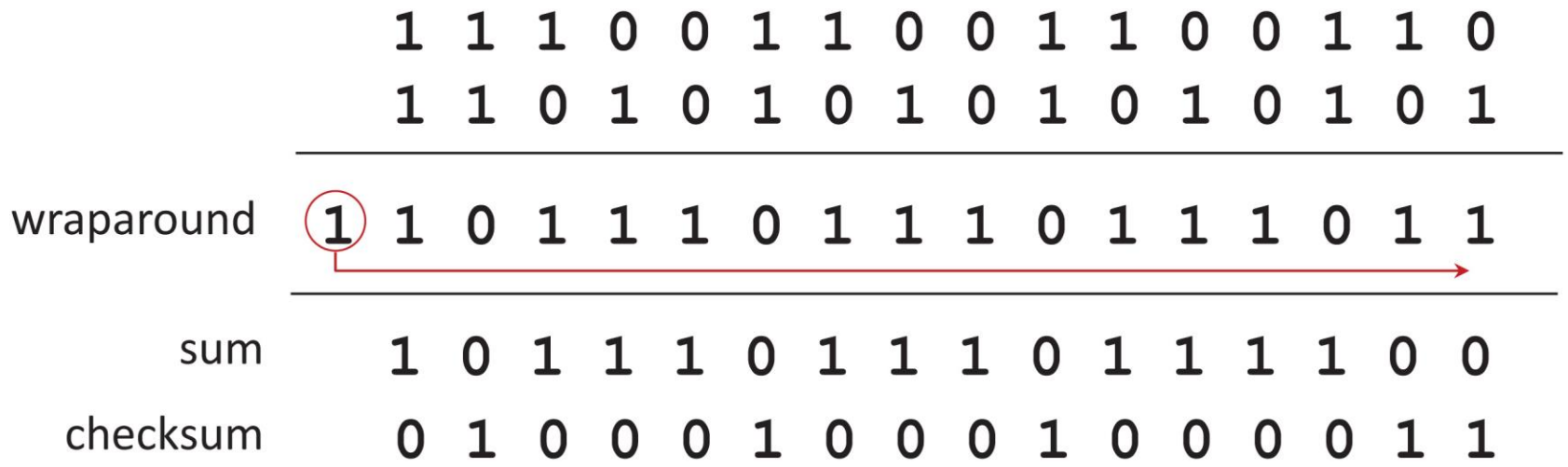
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. **But maybe errors nonetheless?** More later ….

Internet Checksum: an Example

example: add two 16-bit integers



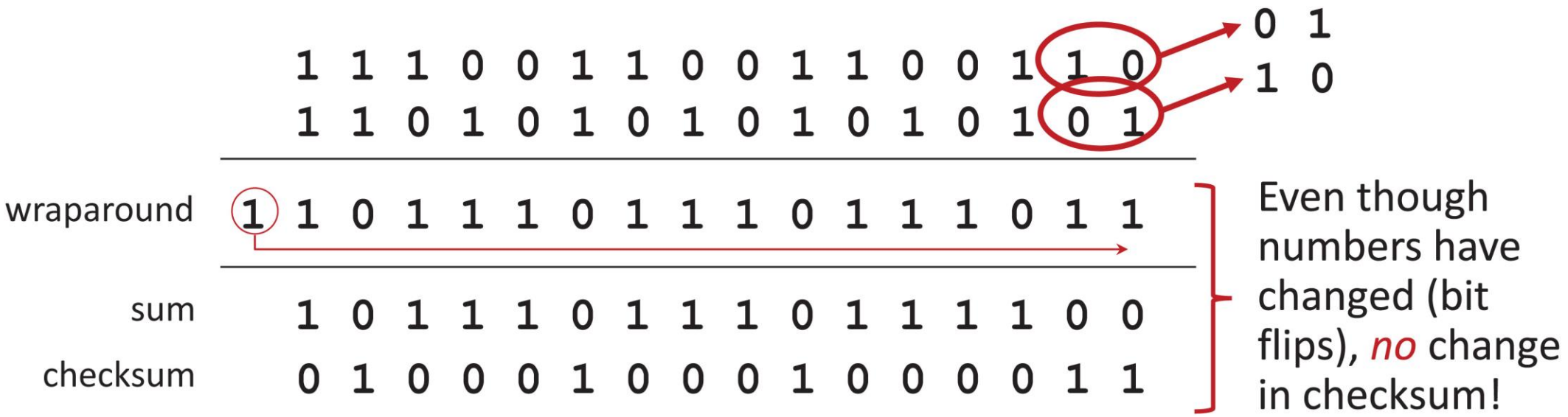
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

* Check out the online interactive exercises for more examples:

http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet Checksum: Weak Protection!

example: add two 16-bit integers

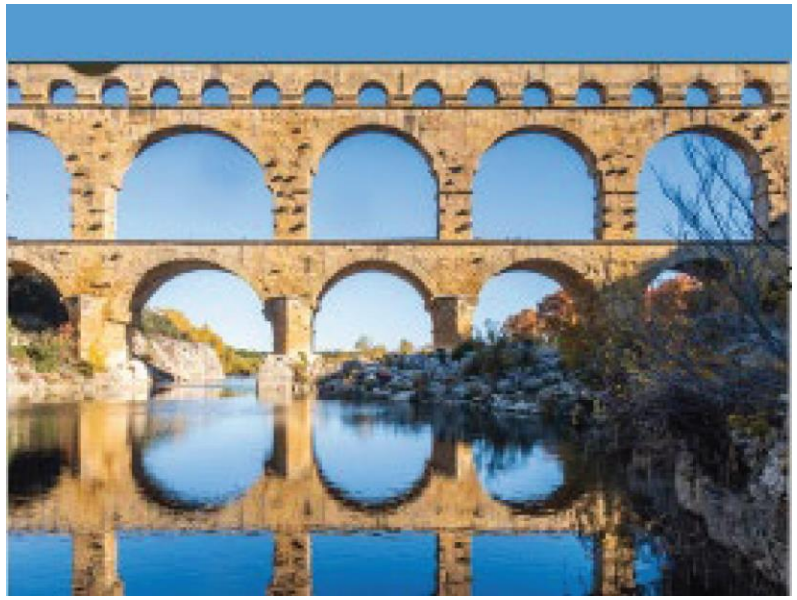


Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Chapter 3: Roadmap (3 of 7)

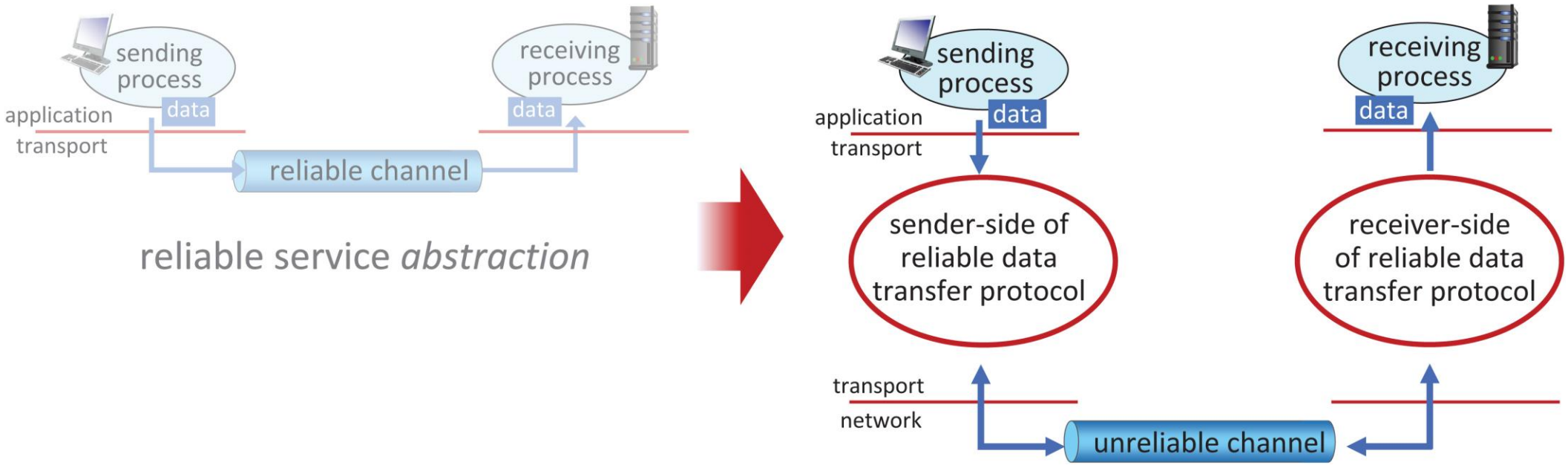
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



Principles of Reliable Data Transfer (1 of 4)

reliable service abstraction

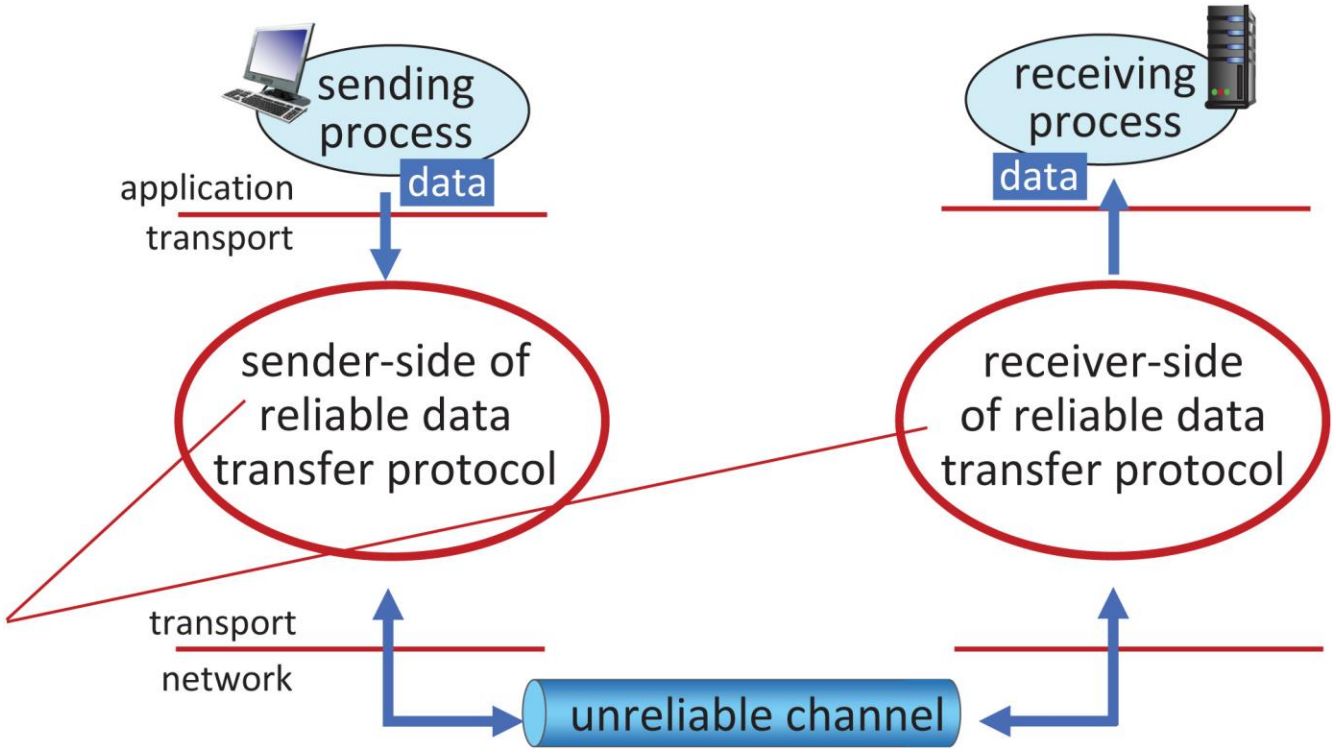
Principles of Reliable Data Transfer (2 of 4)



reliable service **implementation**

Principles of Reliable Data Transfer (3 of 4)

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

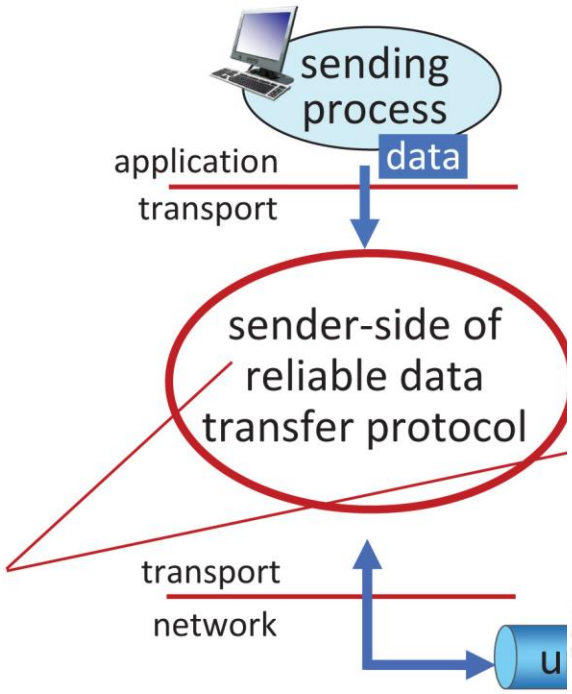


reliable service implementation

Principles of Reliable Data Transfer (4 of 4)

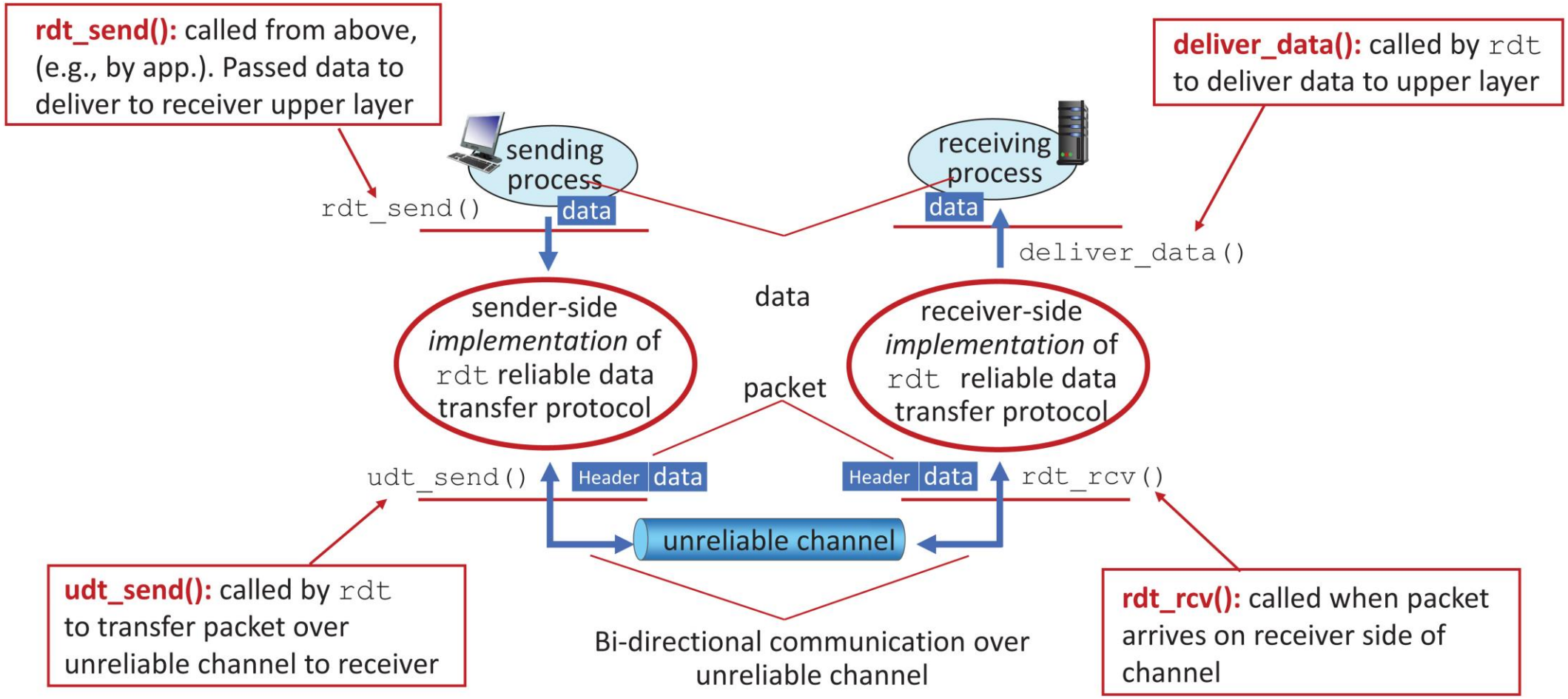
Sender, receiver do **not** know the "state" of each other, e.g., was a message received?

- unless communicated via a message



reliable service implementation

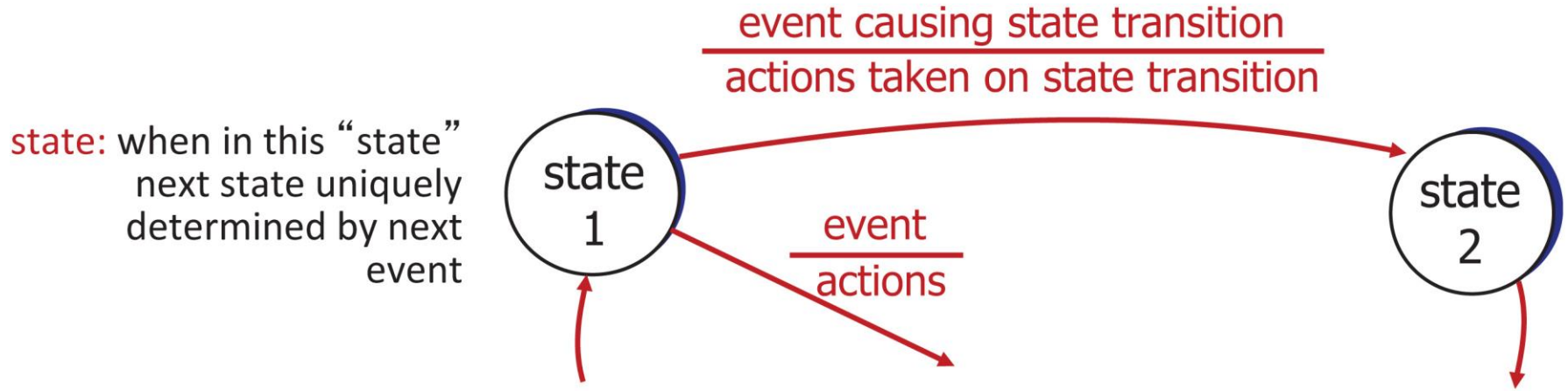
Reliable Data Transfer Protocol (rdt): Interfaces



Reliable Data Transfer: Getting Started

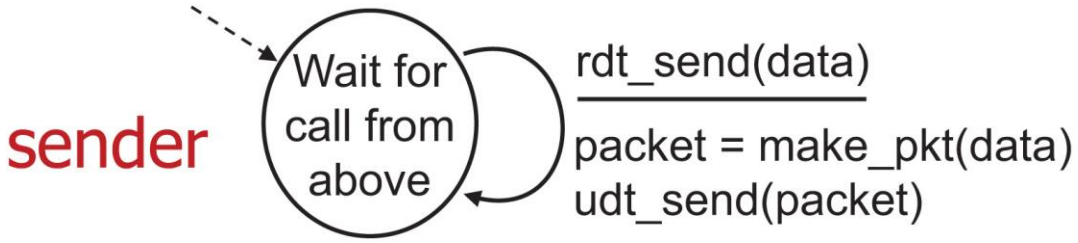
We will:

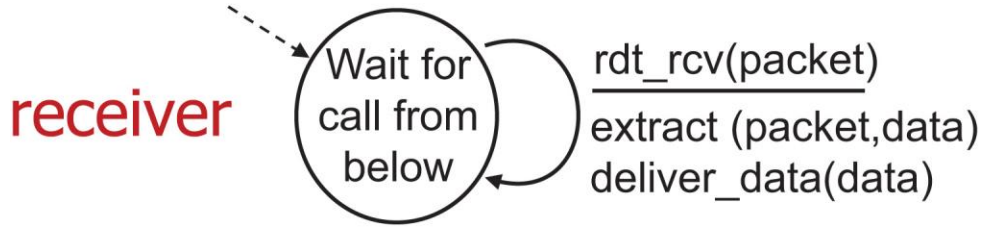
- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



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 reads data from underlying channel





rdt2.0: Channel With Bit Errors (1 of 2)

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- **the** question: how to recover from errors?

How do humans recover from "errors" during conversation?

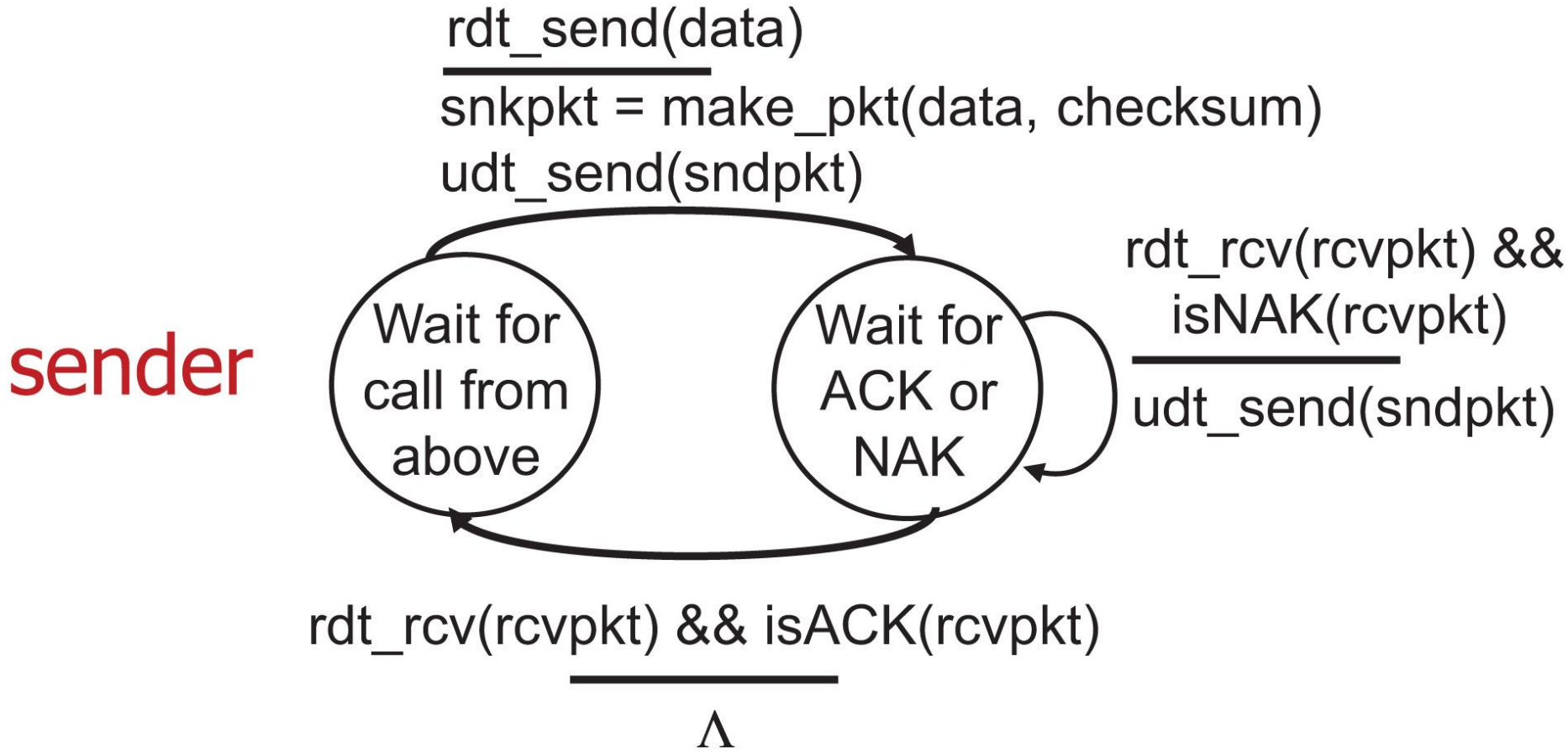
rdt2.0: Channel With Bit Errors (2 of 2)

- underlying channel may flip bits in packet
  - checksum to detect 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 ***r*etransmits** pkt on receipt of NAK

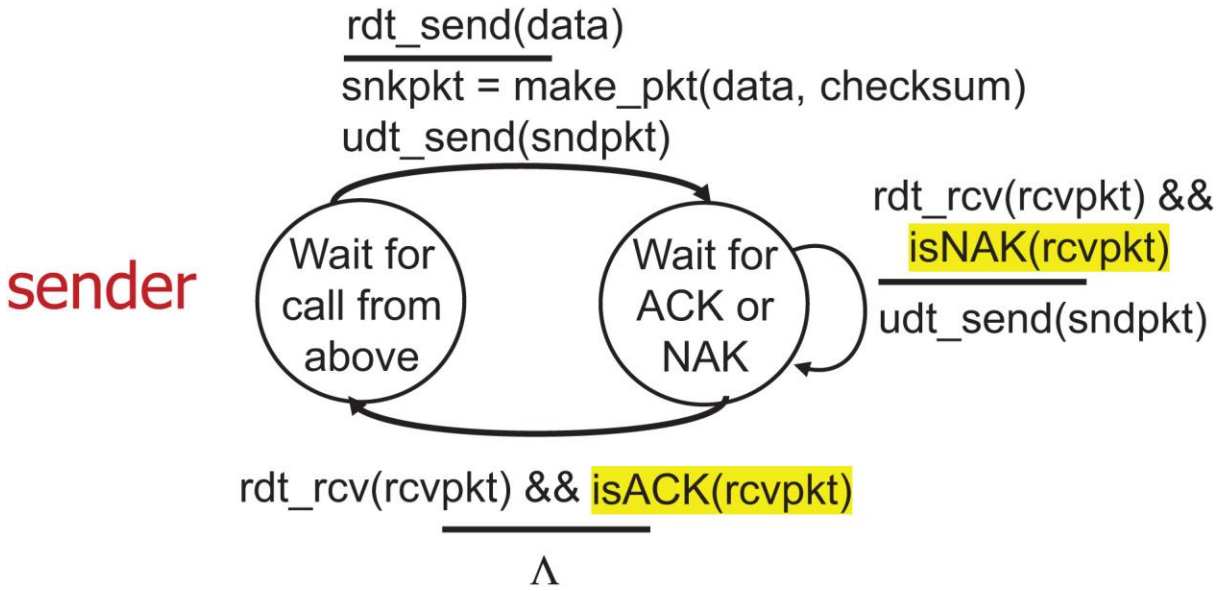
stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: FSM Specifications



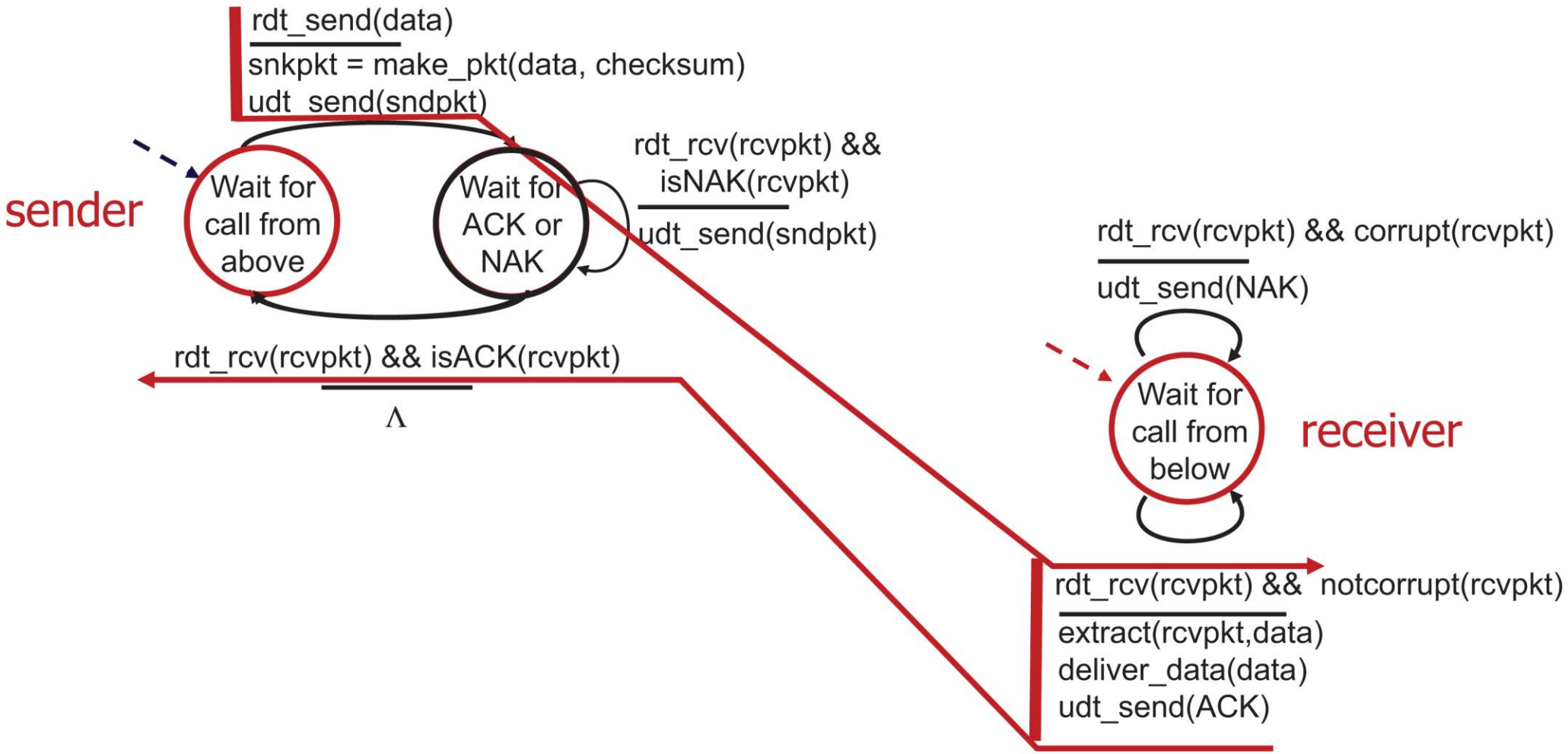
rdt2.0: FSM Specification



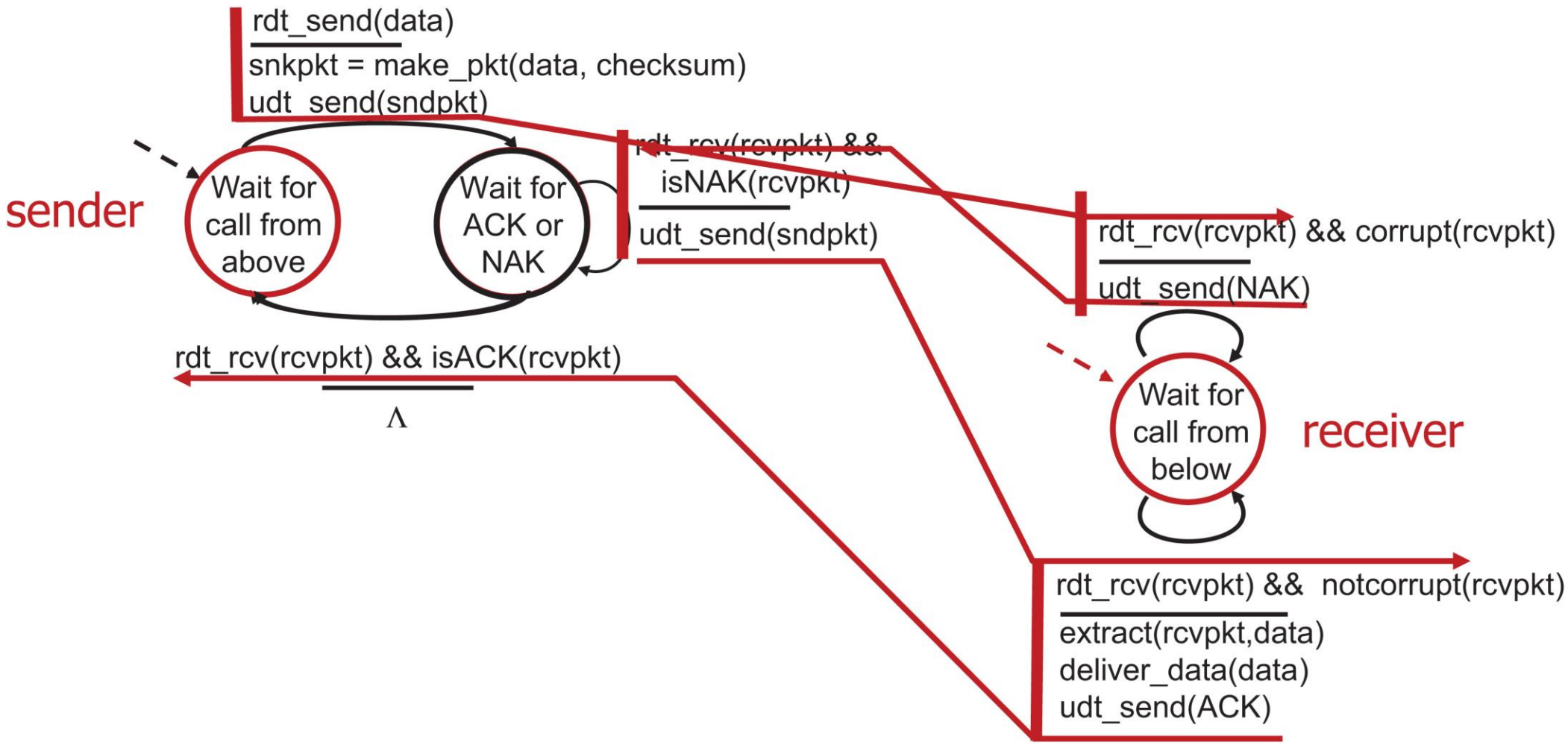
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

rdt2.0: Operation With No Errors



rdt2.0: Corrupted Packet 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 p kt if ACK/NAK corrupted
- sender adds **sequence number** to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

rdt3.0: Channels With Errors and Loss (1 of 2)

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

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

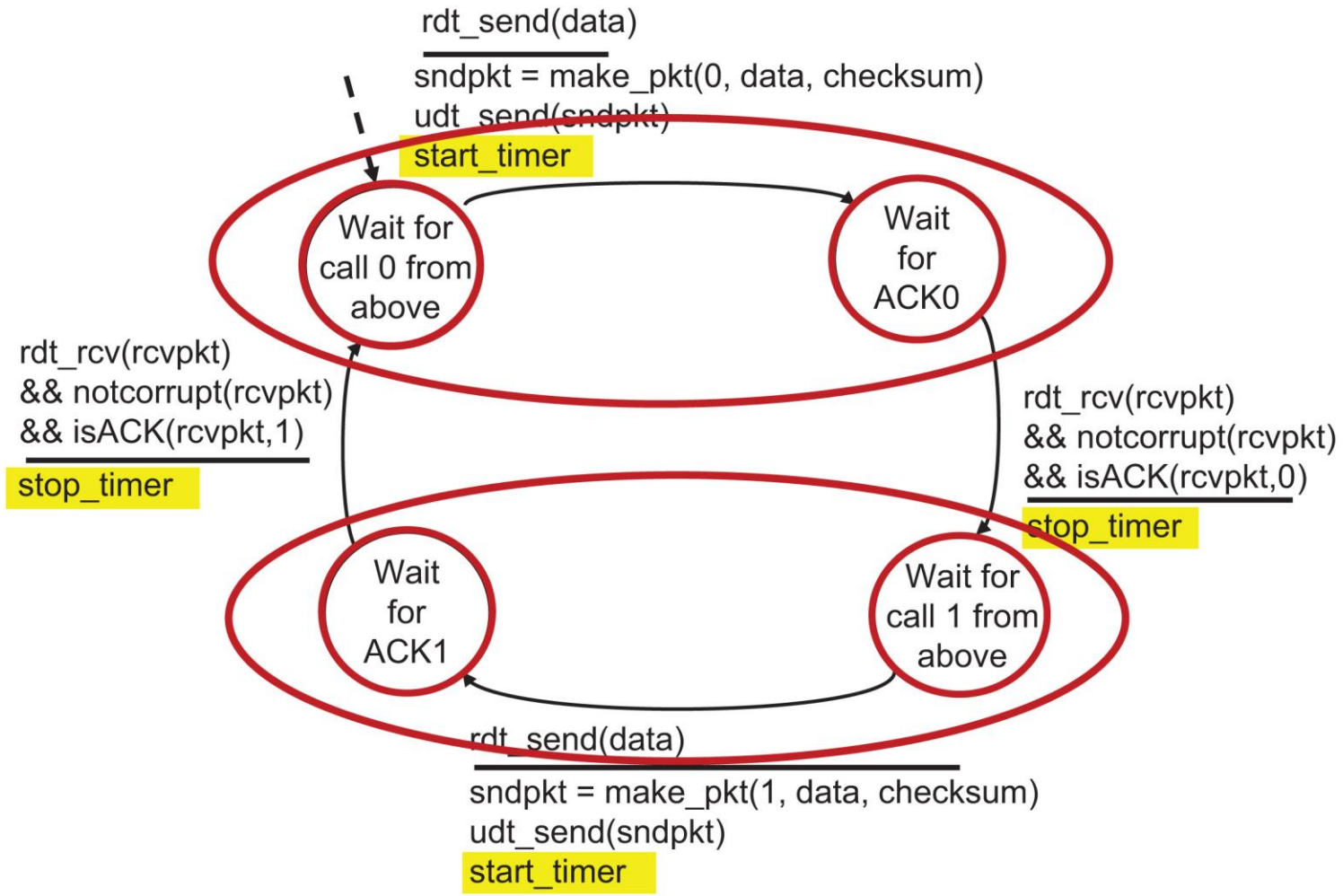
Q: How do **humans** handle lost sender-to-receiver words in conversation?

rdt3.0: Channels With Errors and Loss (2 of 2)

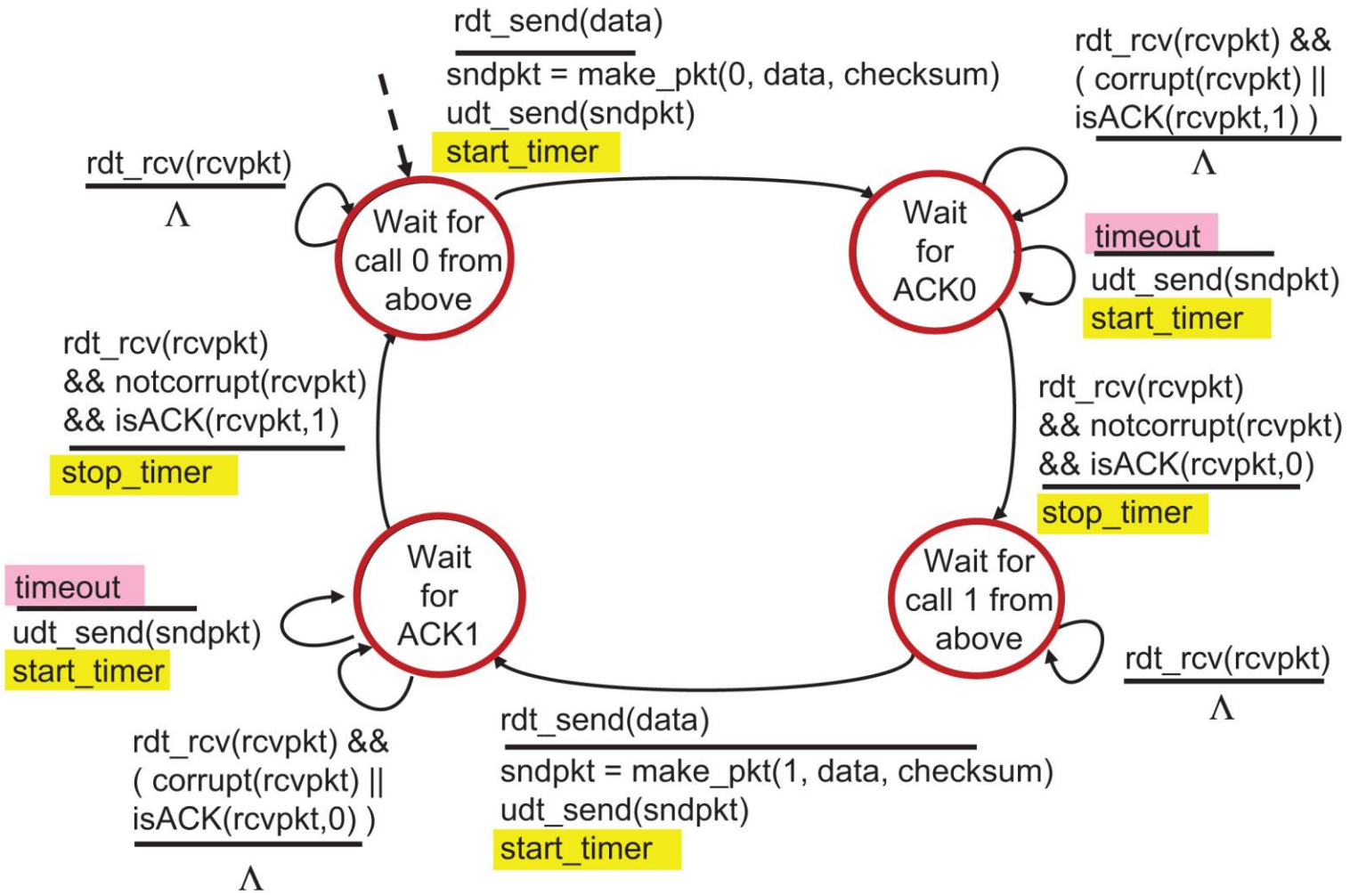
Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

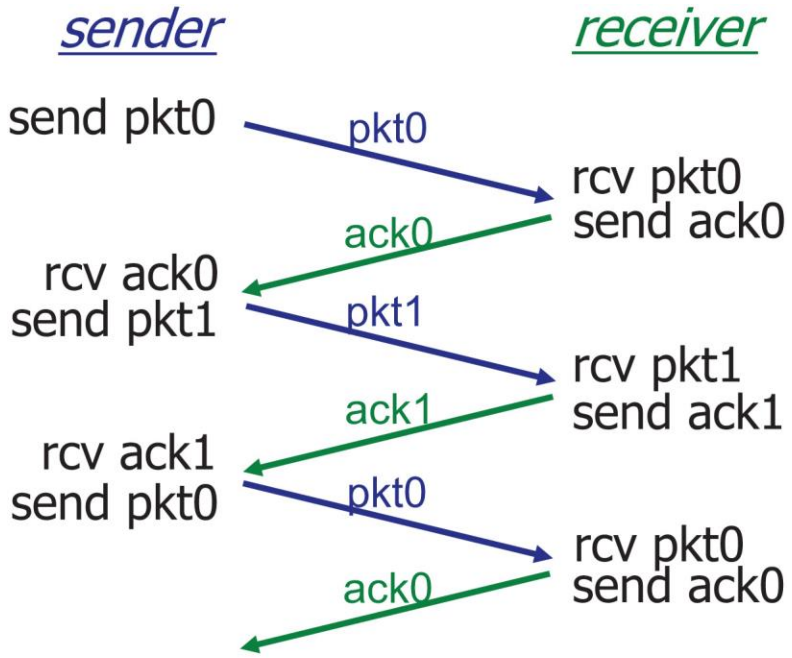
rdt3.0 Sender (1 of 2)



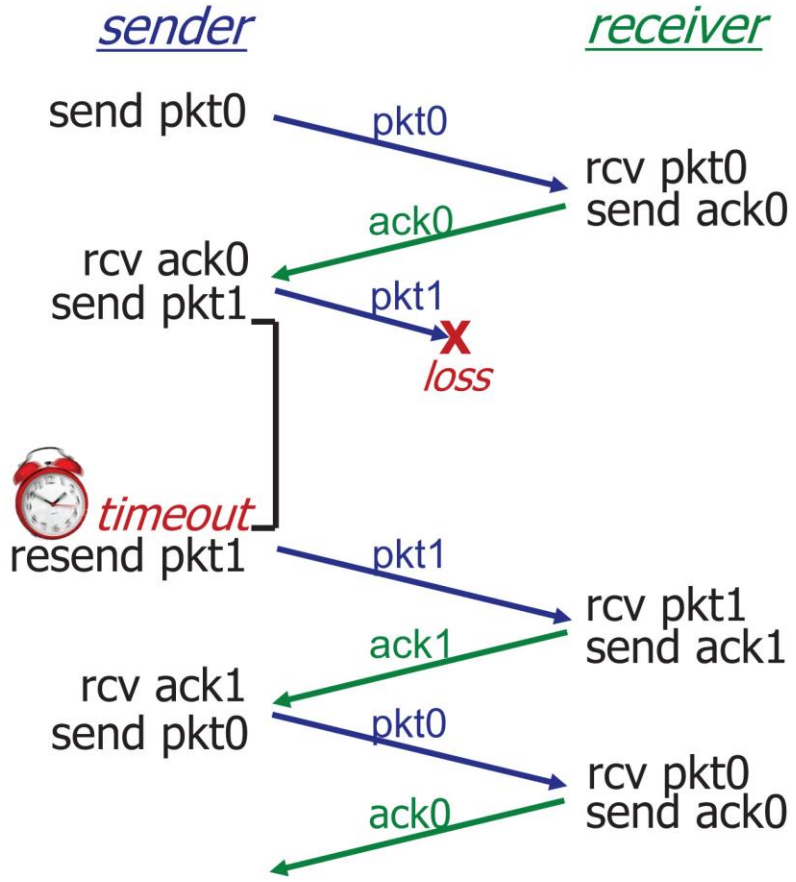
rdt3.0 Sender (2 of 2)



rdt3.0 in Action (1 of 2)

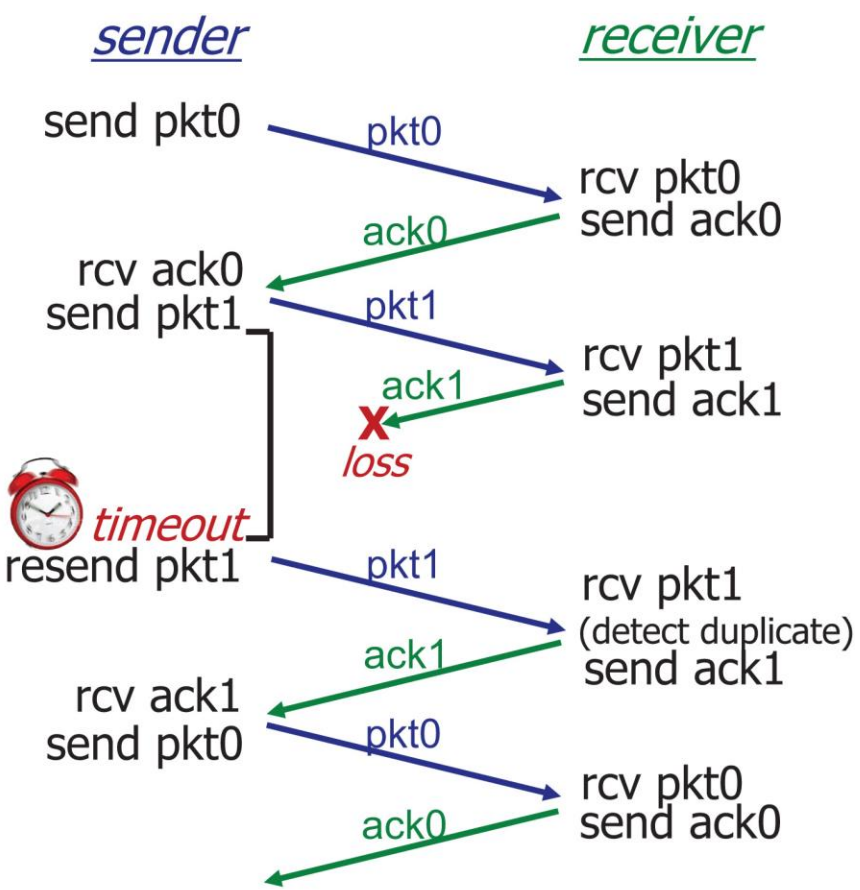


(a) no loss

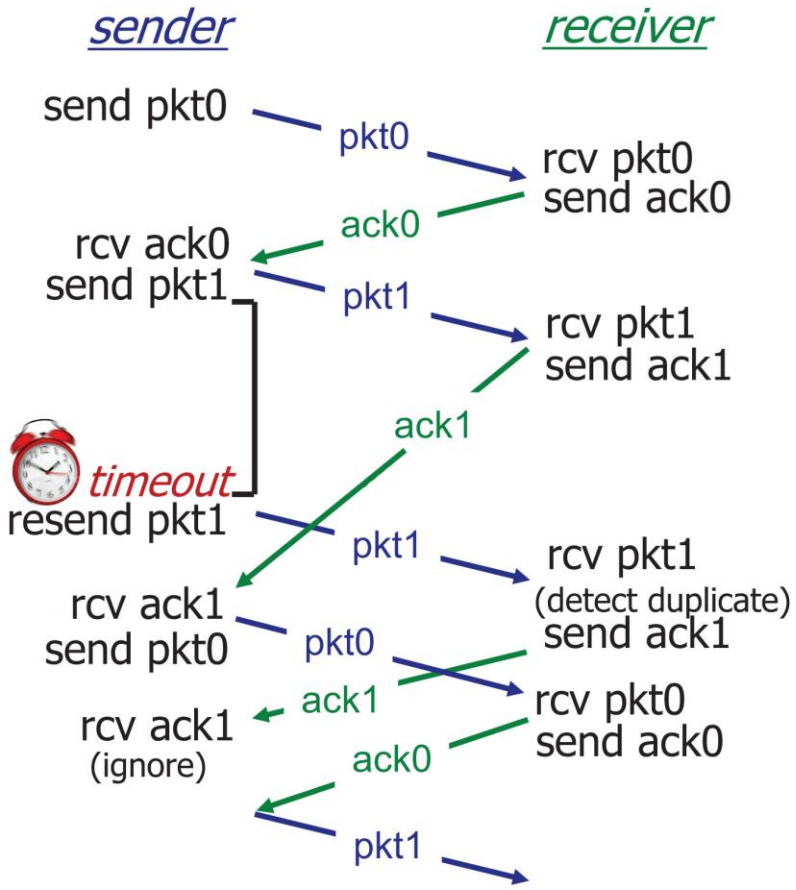


(b) packet loss

rdt3.0 in Action (2 of 2)



(c) ACK loss



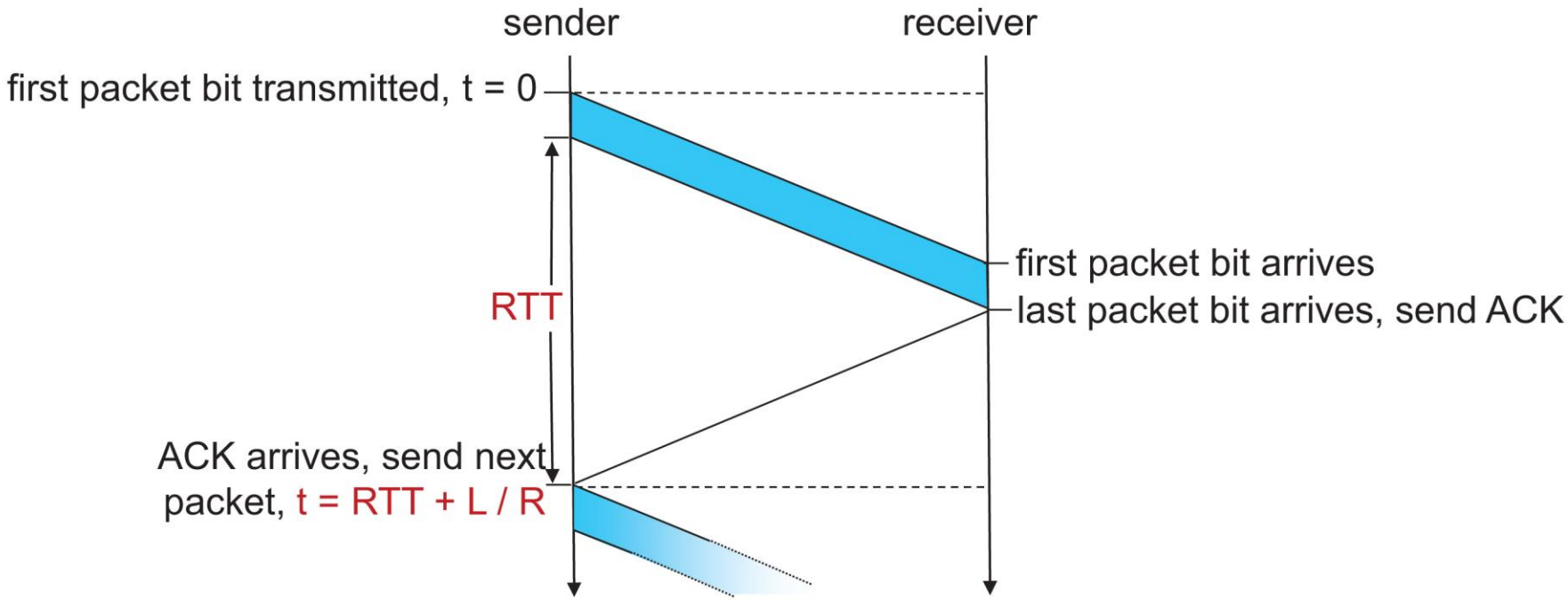
(d) premature timeout/ delayed ACK

Performance of rdt3.0 (stop-and-wait)

- U_{sender} : **utilization** – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

$$D_{trans} = \frac{L}{R} = \frac{8000 \ bits}{10^9 \ bits/sec} = 8 \ microsecs$$

rdt3.0: Stop-and-Wait Operation (1 of 2)

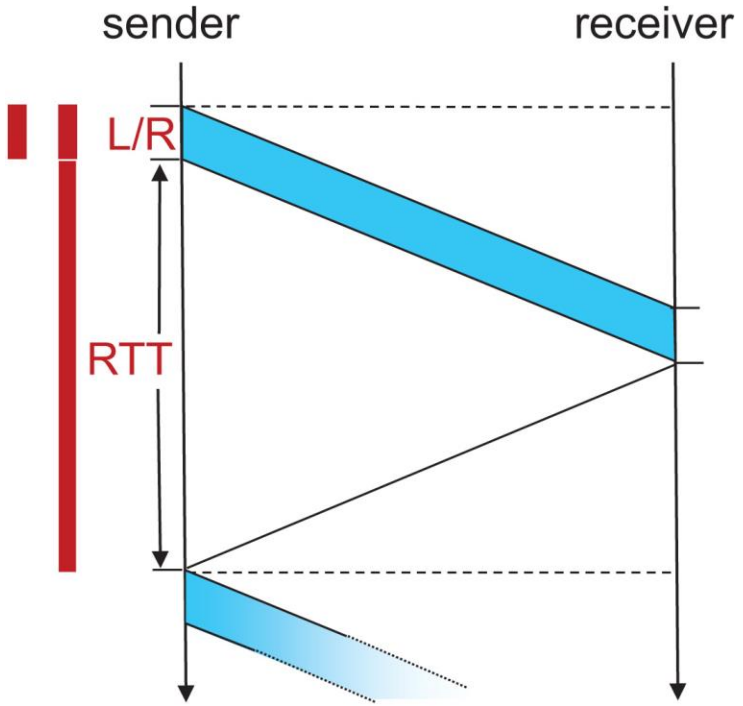


rdt3.0: Stop-and-Wait Operation (2 of 2)

$$U_{sender} = \frac{L/R}{RTT + L/R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

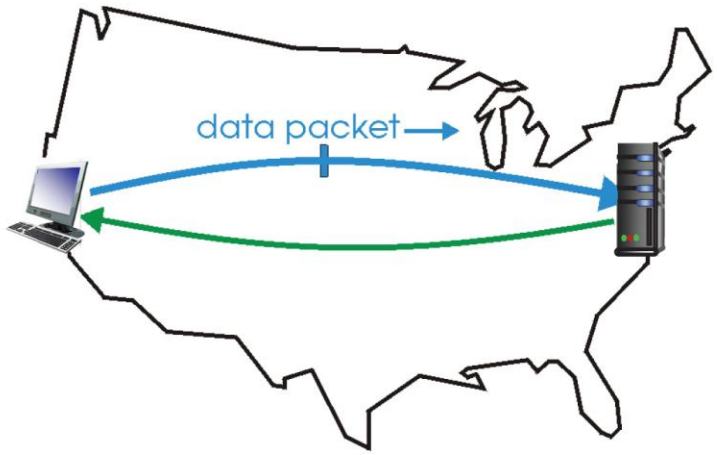


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

Rdt3.0: Pipelined Protocols Operation

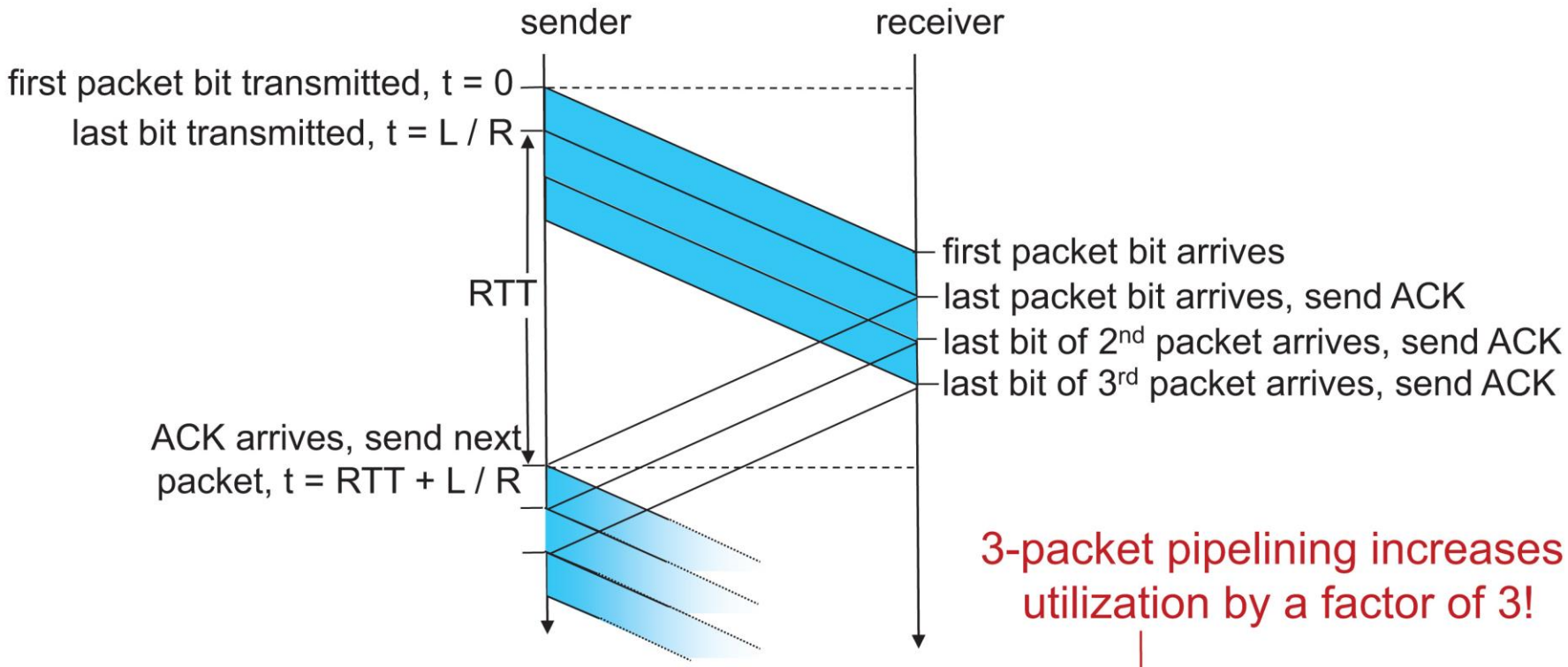
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

Pipelining: Increased Utilization

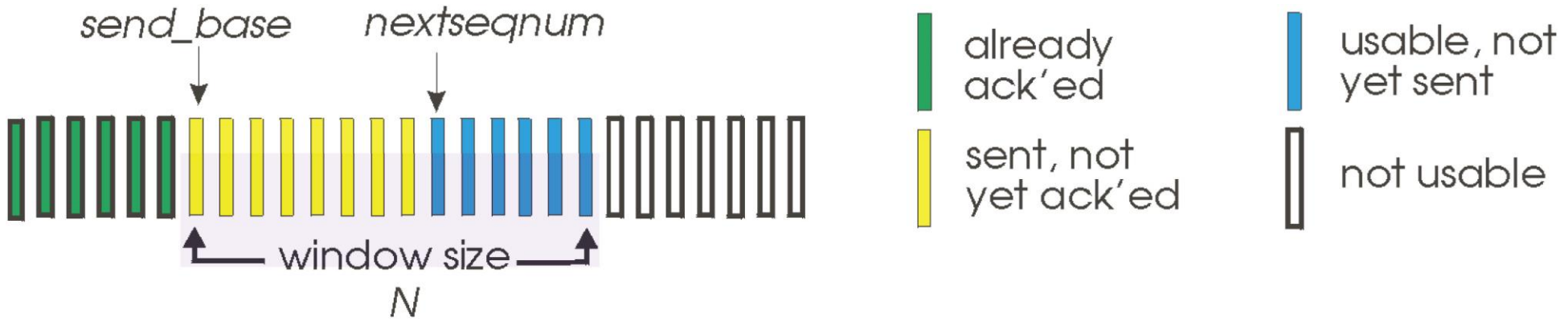


3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: Sender

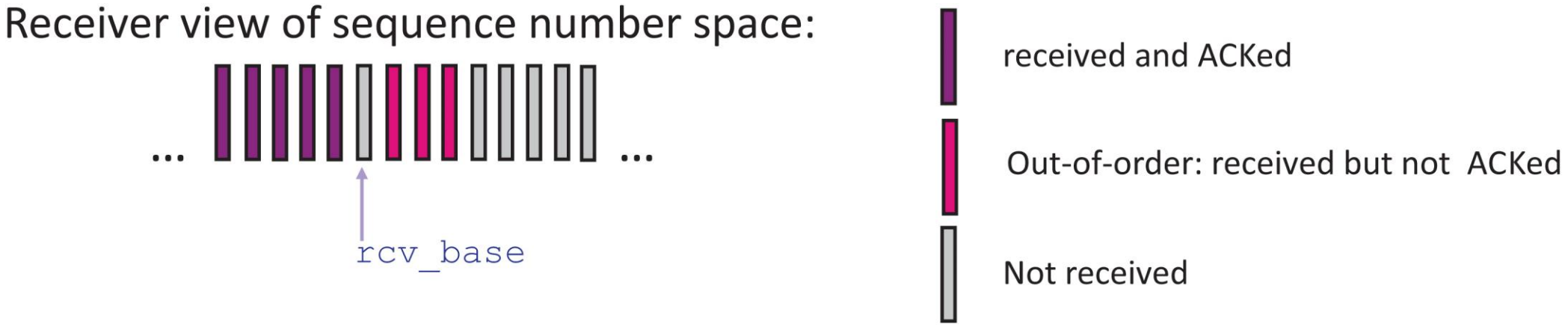
- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header



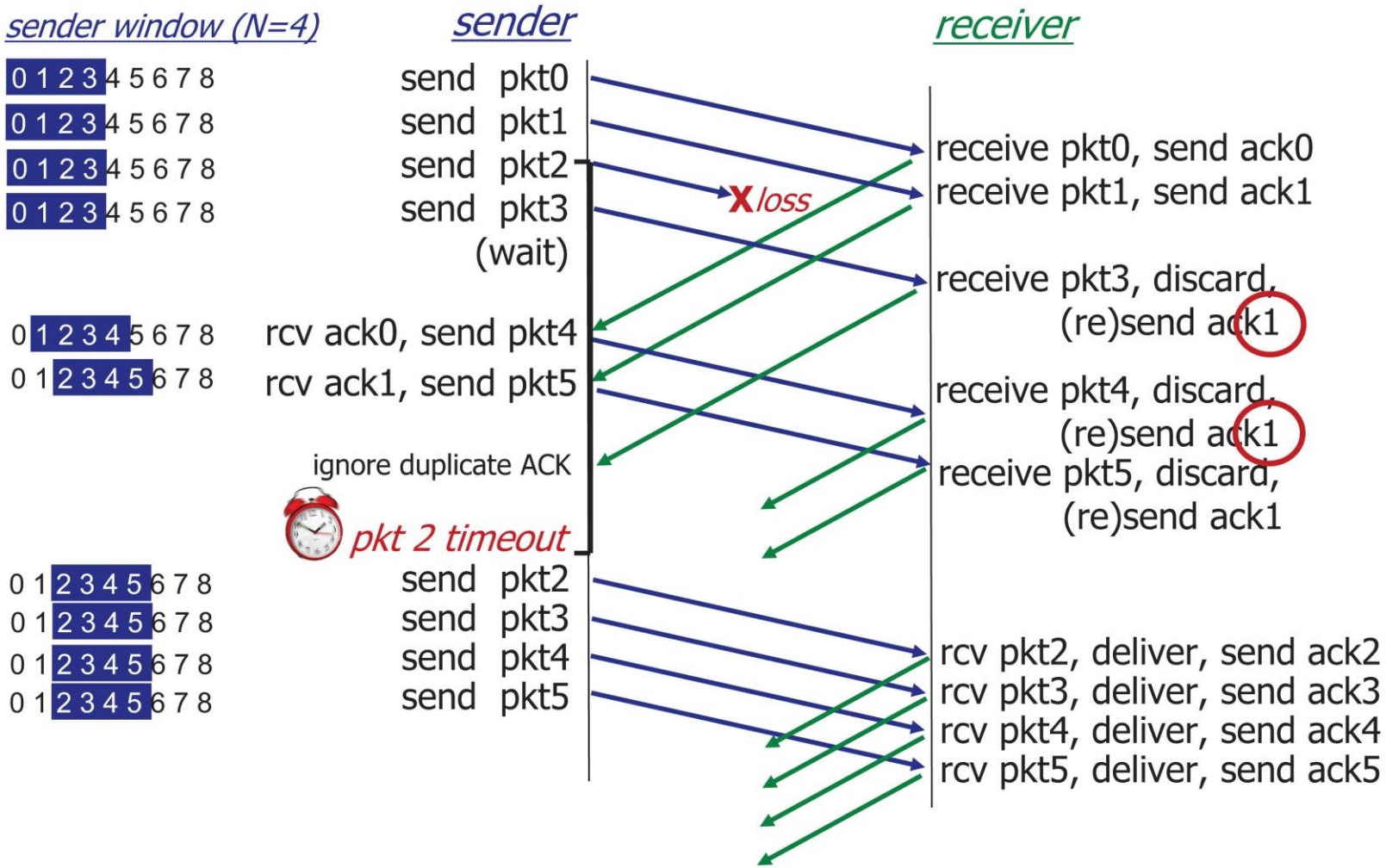
- cumulative ACK: ACK(**n**): ACK s all packets up to, including seq # **n**
  - on receiving ACK(**n**): move window forward to begin at **n+1**
- timer for oldest in-flight packet
- **timeout(n)**: retransmit packet n and all higher seq # packets in window

Go-Back-N: Receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #



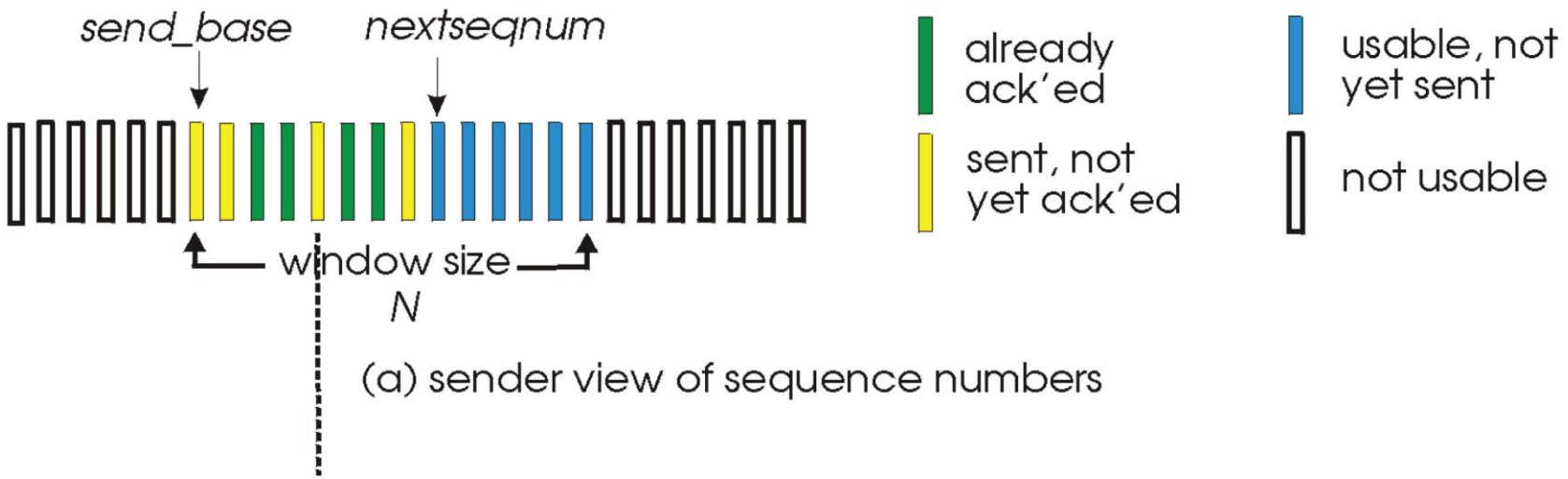
Go-Back-N in Action



Selective Repeat: The Approach

- pipelining: **multiple** packets in flight
- receiver individually ACKs all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) "window" over *N* consecutive seq #s
    - limits pipelined, "in flight" packets to be within this window

Selective Repeat: Sender, Receiver Windows



(a) sender view of sequence numbers

Selective Repeat: Sender and Receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase +N−1]:

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase +N−1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

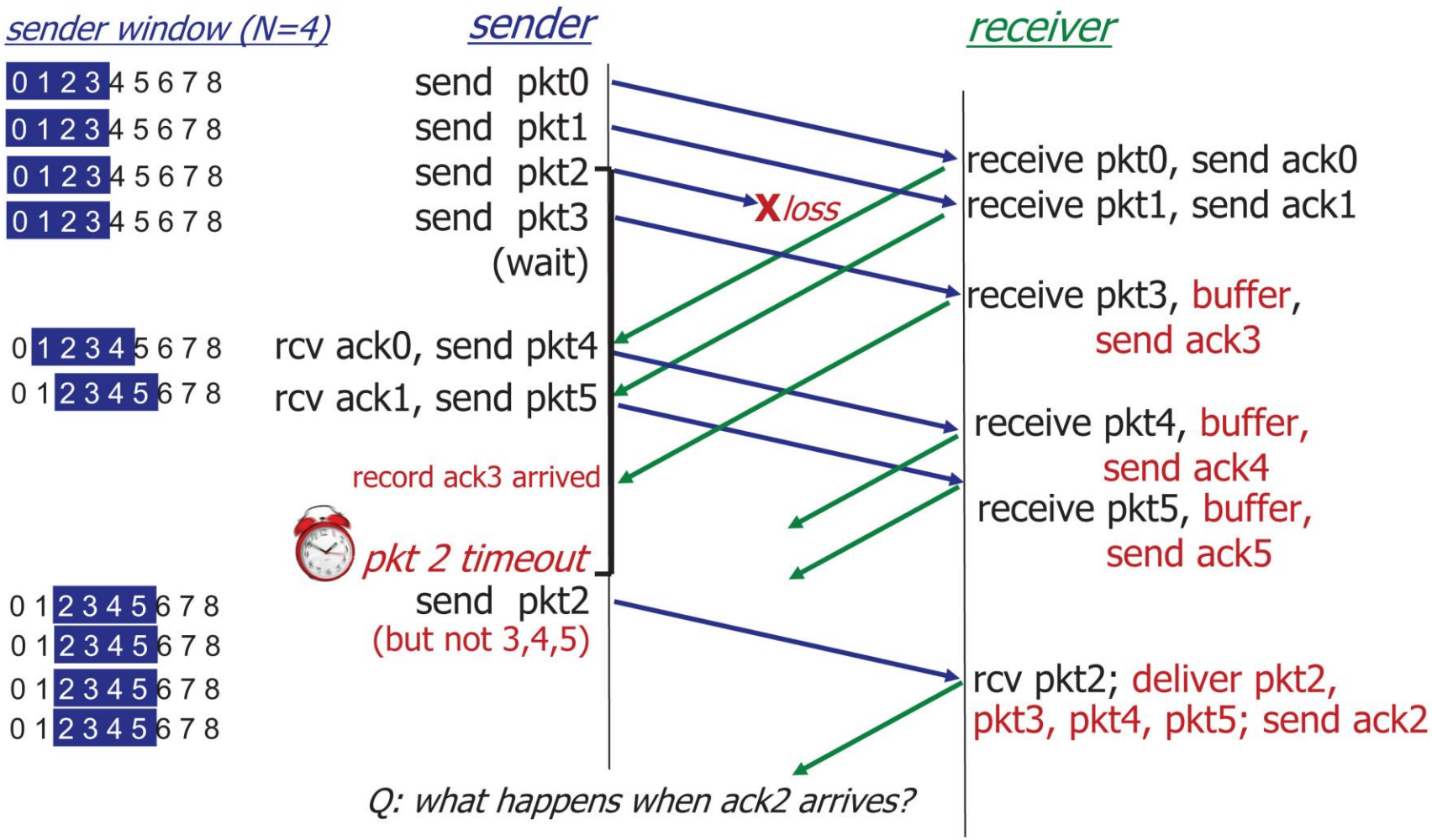
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

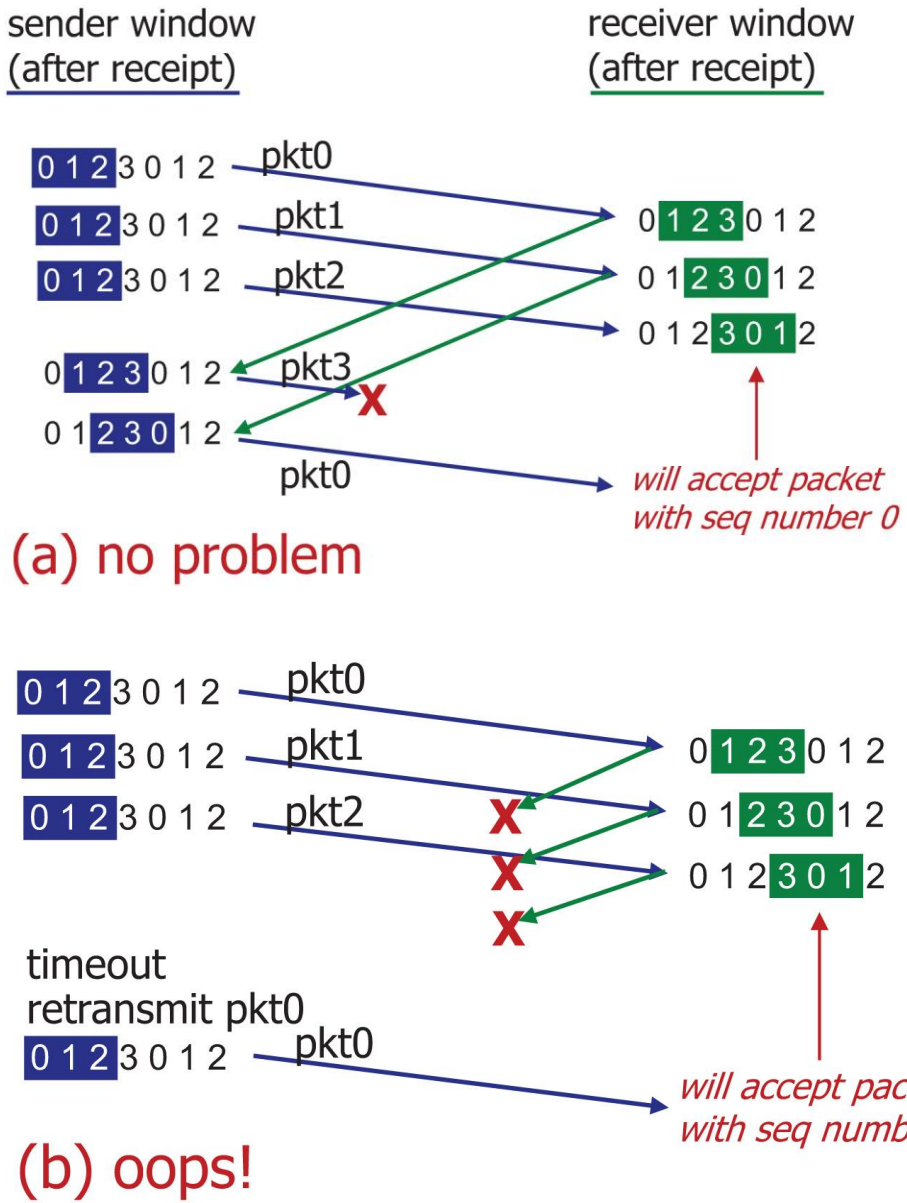
Selective Repeat in Action



Selective Repeat: a Dilemma! (1 of 2)

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



Selective Repeat: a Dilemma! (2 of 2)

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?