

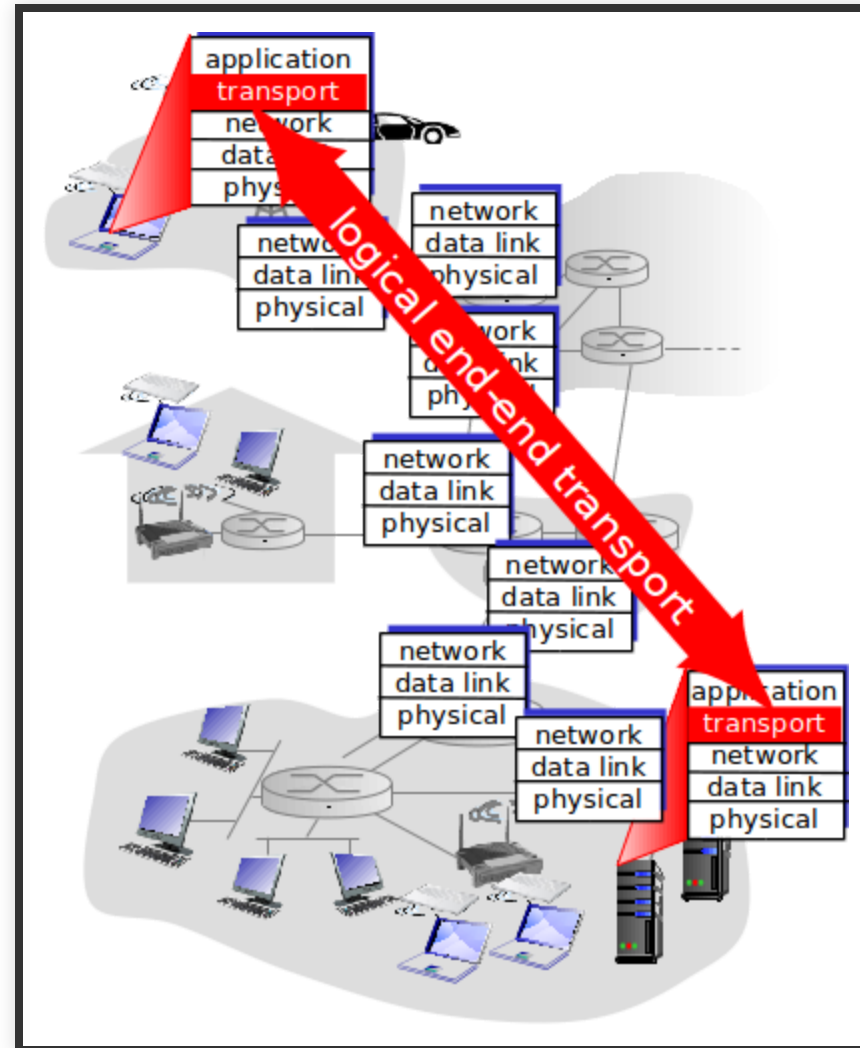


LECTURE 3 - TRANSPORT LAYER (1)

GOALS

- Understand principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
- Learn about Internet transport layer protocols:
 - UDP: connectionless transport
 - TCP: next week

TRANSPORT-LAYER SERVICES



TRANSPORT SERVICES AND PROTOCOLS

- Provide logical communication between app processes running on different hosts
- Transport protocols run in end systems
 - Sending side: breaks app messages into segments, passes to network layer
 - Receiving side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
 - Internet: TCP and UDP

TRANSPORT VS. NETWORK LAYER

- **Transport layer:** logical communication between processes
- **Network layer:** logical communication between hosts
 - Relies on, enhances, network layer services

HOUSEHOLD ANALOGY

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- Hosts = houses
- Processes = kids
- App messages = letters in envelopes
- Transport protocol = Ann and Bill who demux to in-house siblings
- Network-layer protocol = postal service

INTERNET TRANSPORT-LAYER PROTOCOLS

- Reliable, in-order delivery: TCP
 - Congestion control
 - Flow control
 - Connection setup
- Unreliable, unordered delivery: UDP
 - No-frills extension of “best-effort” IP
- Services not available:
 - Delay guarantees
 - Bandwidth guarantees

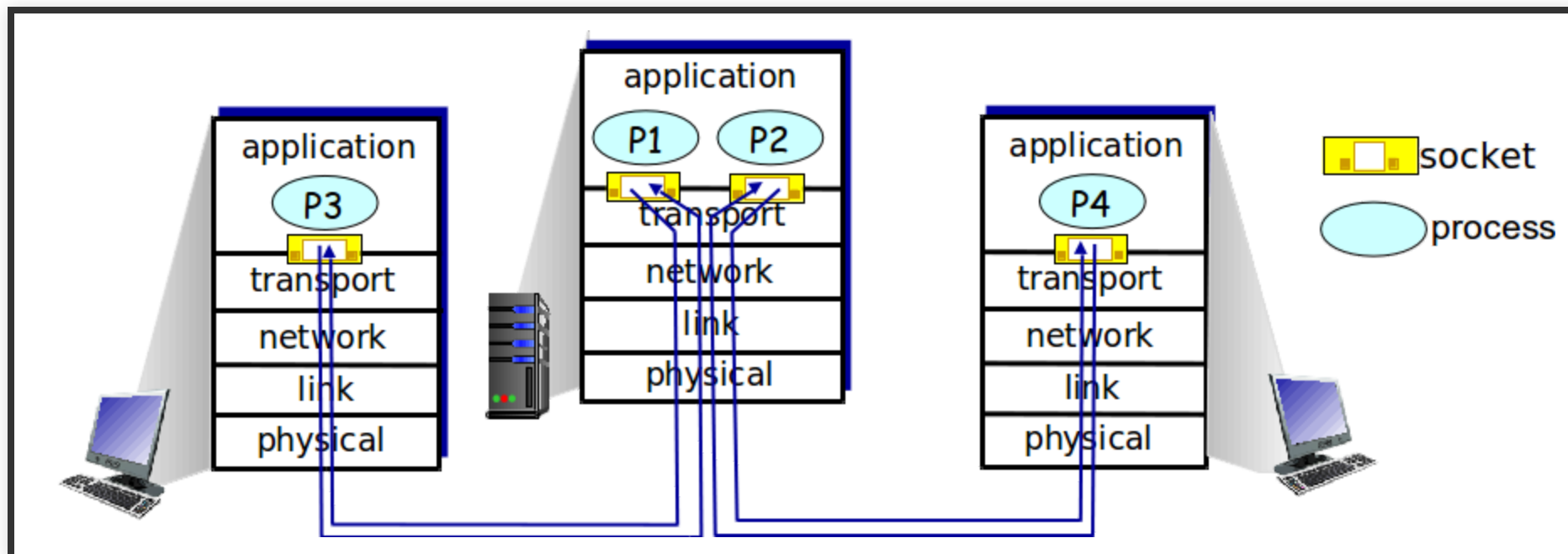
MULTIPLEXING/DEMULTIPLEXING

Multiplexing at sender:

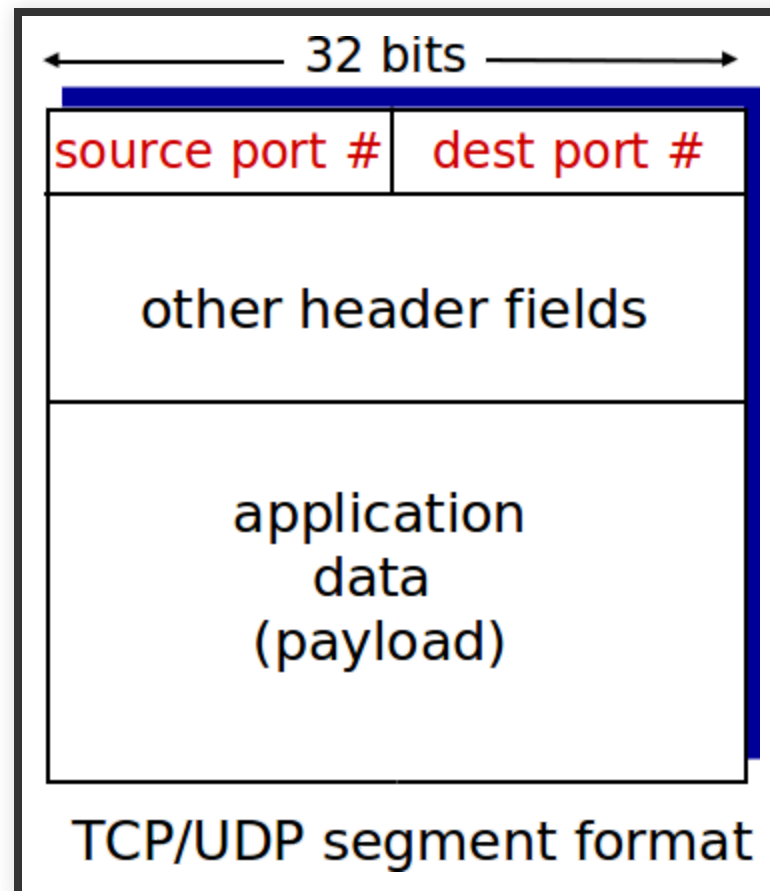
Handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing at receiver:

use header info to deliver received segments to correct socket



HOW DEMULTIPLEXING WORKS



HOW DEMULTIPLEXING WORKS

- Host receives IP datagrams
 - Each datagram has source IP address, destination IP address
 - Each datagram carries one transport-layer segment
 - Each segment has source, destination port number
- Host uses **IP addresses and port numbers** to direct segment to appropriate socket

CONNECTIONLESS DEMULTIPLEXING

Recall: created socket has host-local port #:

```
DatagramSocket mySocket = new DatagramSocket(12534);
```

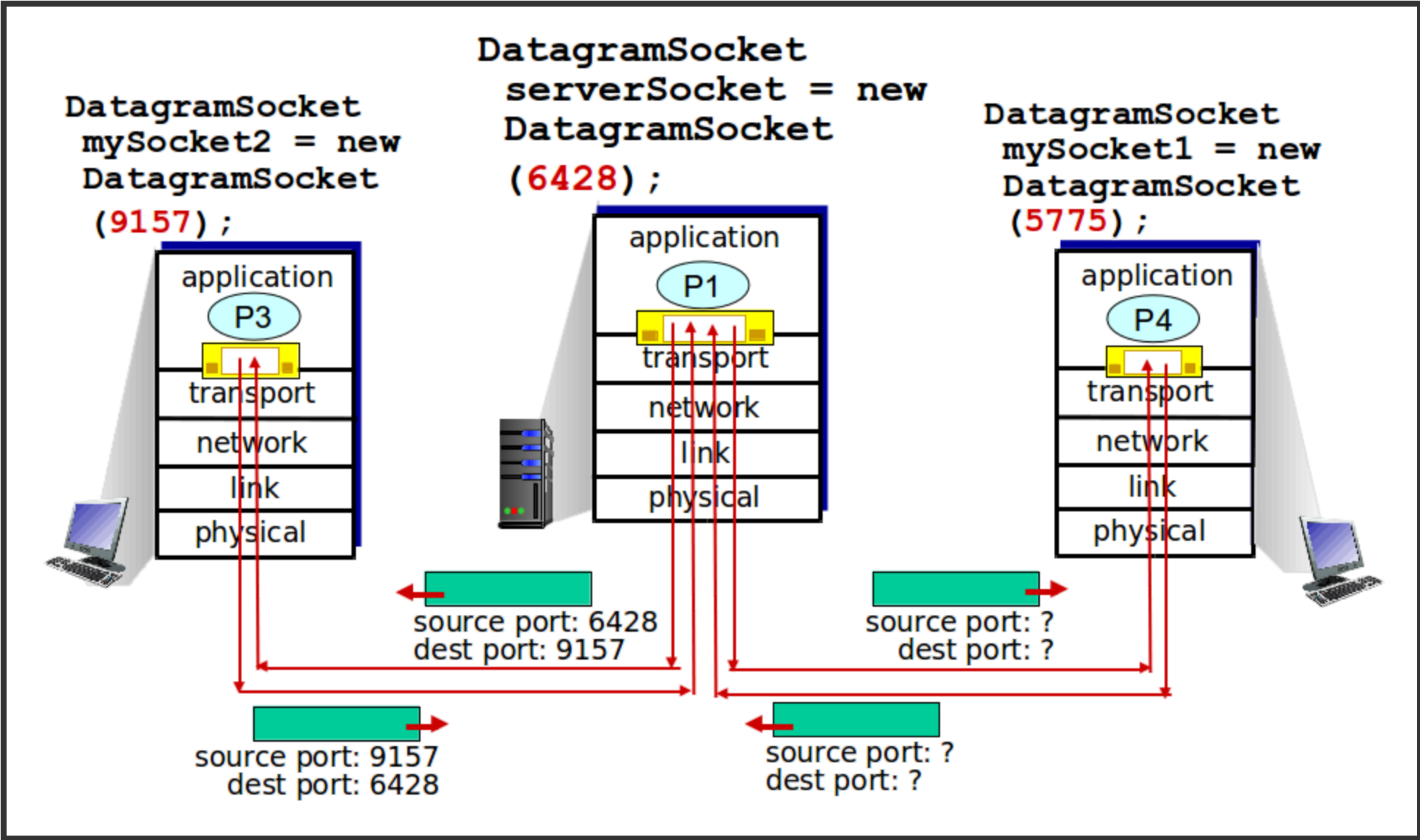
Recall: when creating datagram to send into UDP socket, must specify

- Destination IP address
- Destination port #
- when host receives UDP segment:
 - checks destination port # in segment
 - directs UDP segment to socket with that port #

CONNECTIONLESS DEMULTIPLEXING

- 💡 IP datagrams with **same dest. port #**, but different source IP addresses and/or source port numbers will be directed to **same socket** at dest

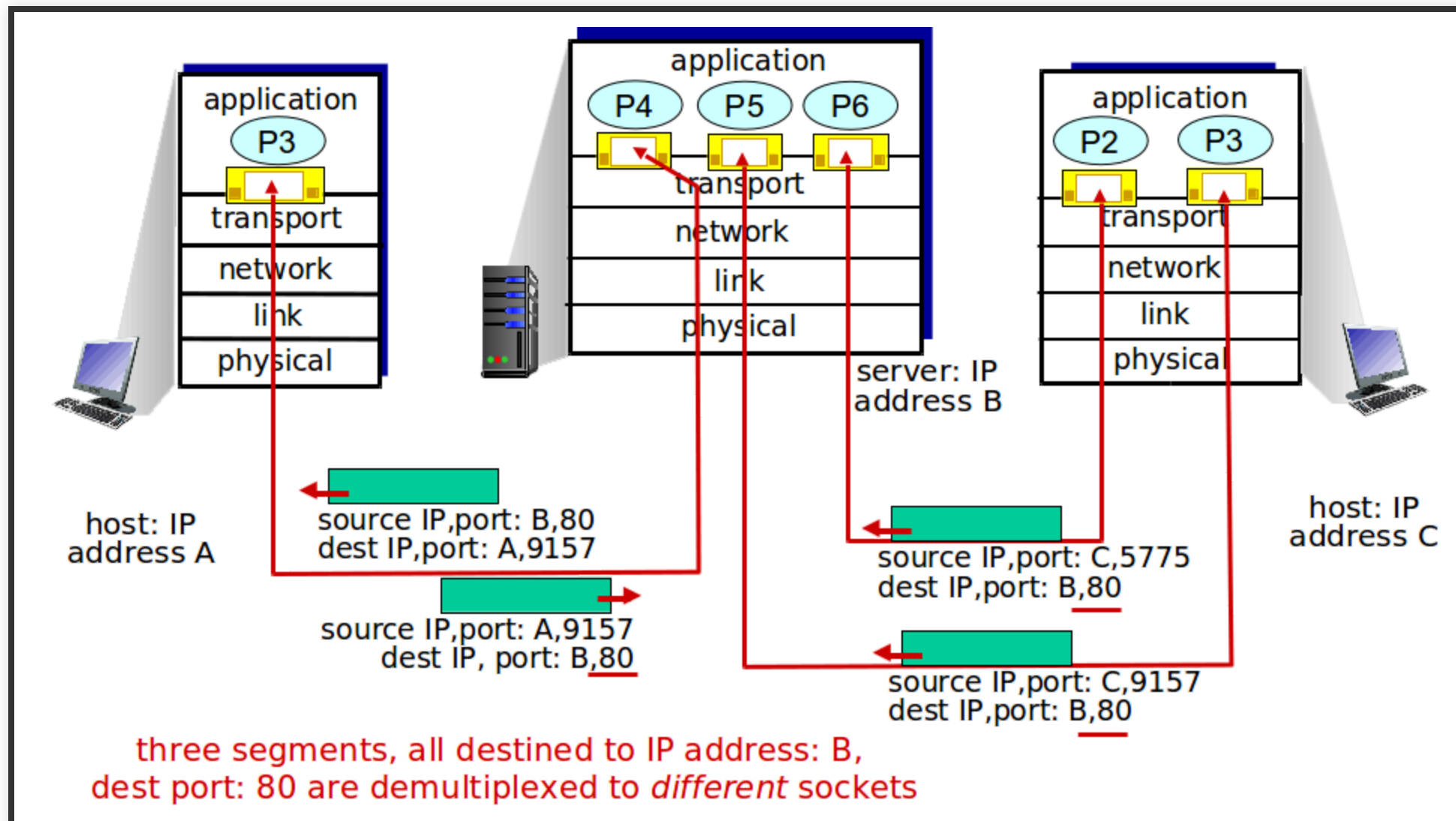
CONNECTIONLESS DEMUX: EXAMPLE



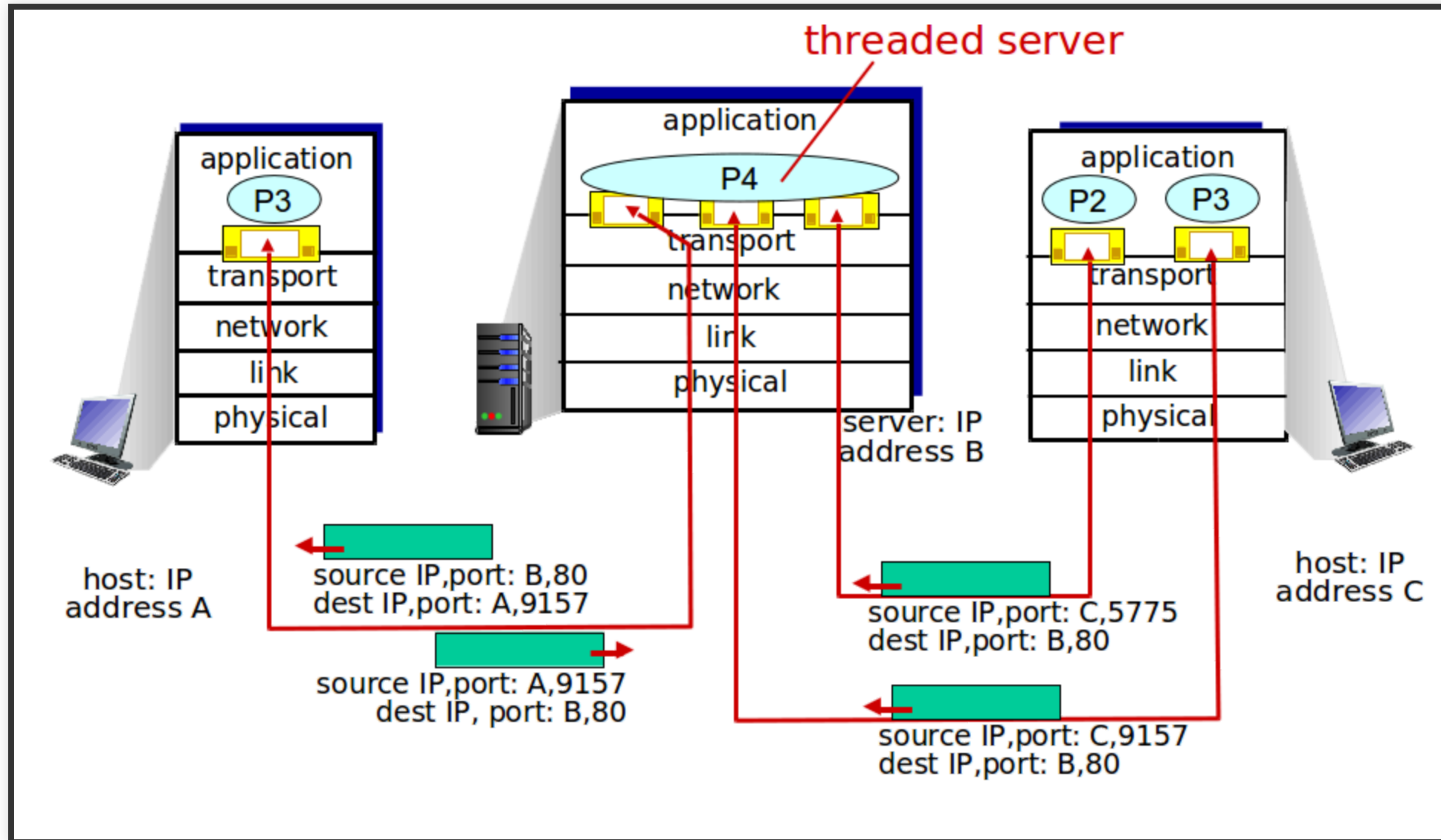
CONNECTION-ORIENTED DEMUX

- TCP socket identified by 4-tuple:
 - (Source IP address, Source port number, Dest IP address, Dest port number)
- Demux: receiver uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
 - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
 - Non-persistent HTTP will have different socket for each request

CONNECTION-ORIENTED DEMUX: EXAMPLE



CONNECTION-ORIENTED DEMUX: EXAMPLE



UDP: USER DATAGRAM PROTOCOL

- RFC 768
- "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

UDP: USER DATAGRAM PROTOCOL

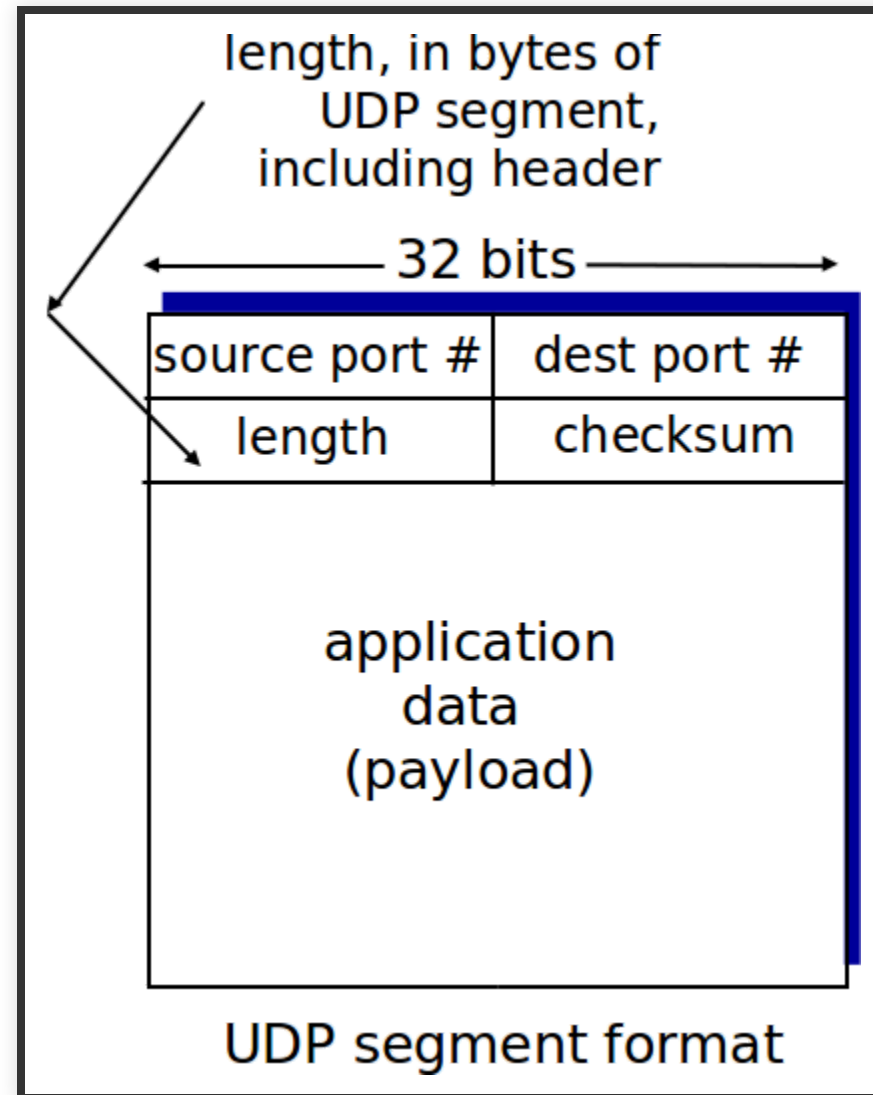
- UDP use:
 - Streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
- Reliable transfer over UDP:
 - Add reliability at application layer
 - Application-specific error recovery!

UDP: SEGMENT HEADER

❗ why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small header size
- No congestion control: UDP can blast away as fast as desired
- No breaks due to lost packets

UDP: SEGMENT HEADER



UDP CHECKSUM

- ❗ Goal: detect "errors" (e.g., flipped bits) in transmitted segment

UDP CHECKSUM

Sender:

- Treat segment contents, including header fields, as sequence of 16-bit integers
- Checksum: addition (one's complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

UDP CHECKSUM

Receiver:

- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
 - **[NO]** error detected
 - **[YES]** no error detected. But maybe errors nonetheless? More later...

INTERNET CHECKSUM: EXAMPLE

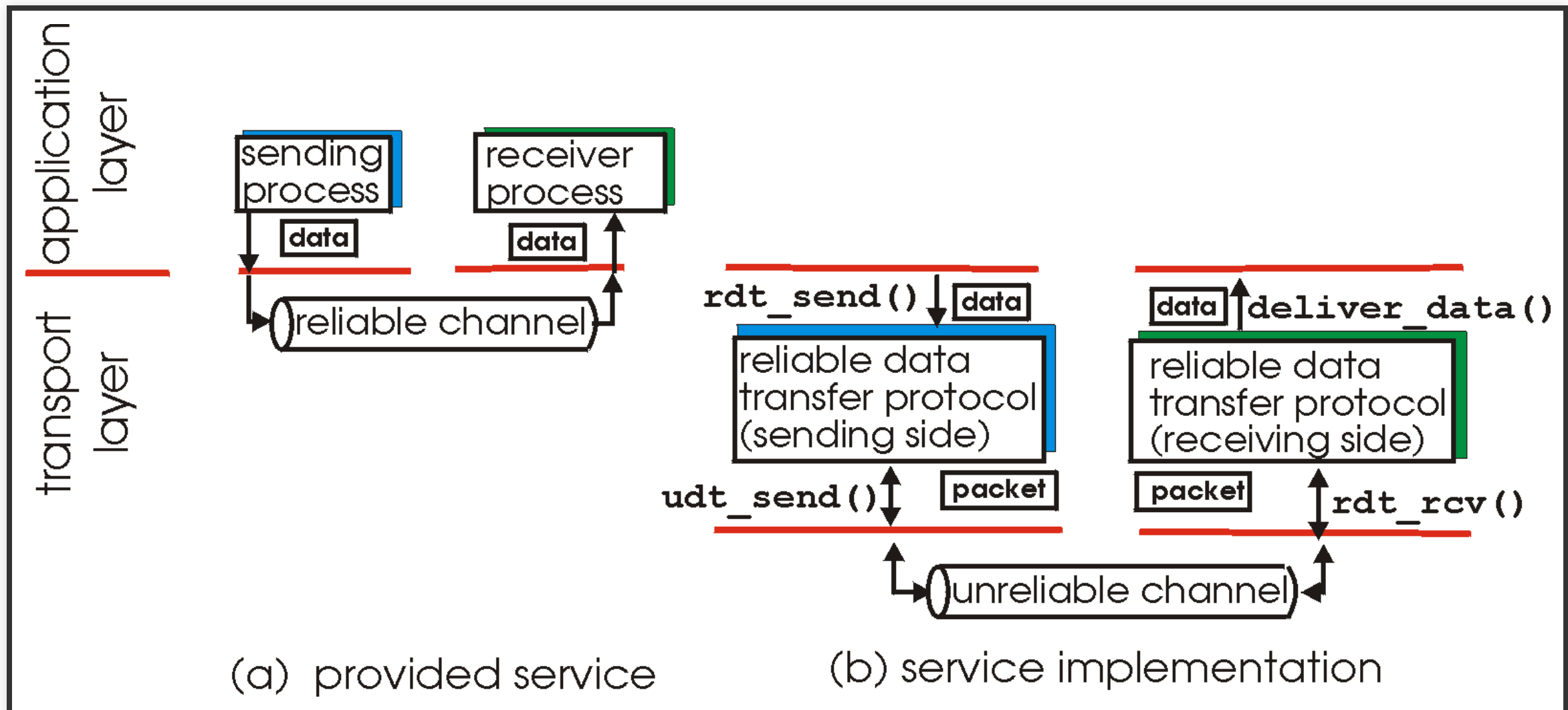
example: add two 16-bit integers

	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	
	<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
	<hr/>																
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	

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

PRINCIPLES OF RELIABLE DATA TRANSFER

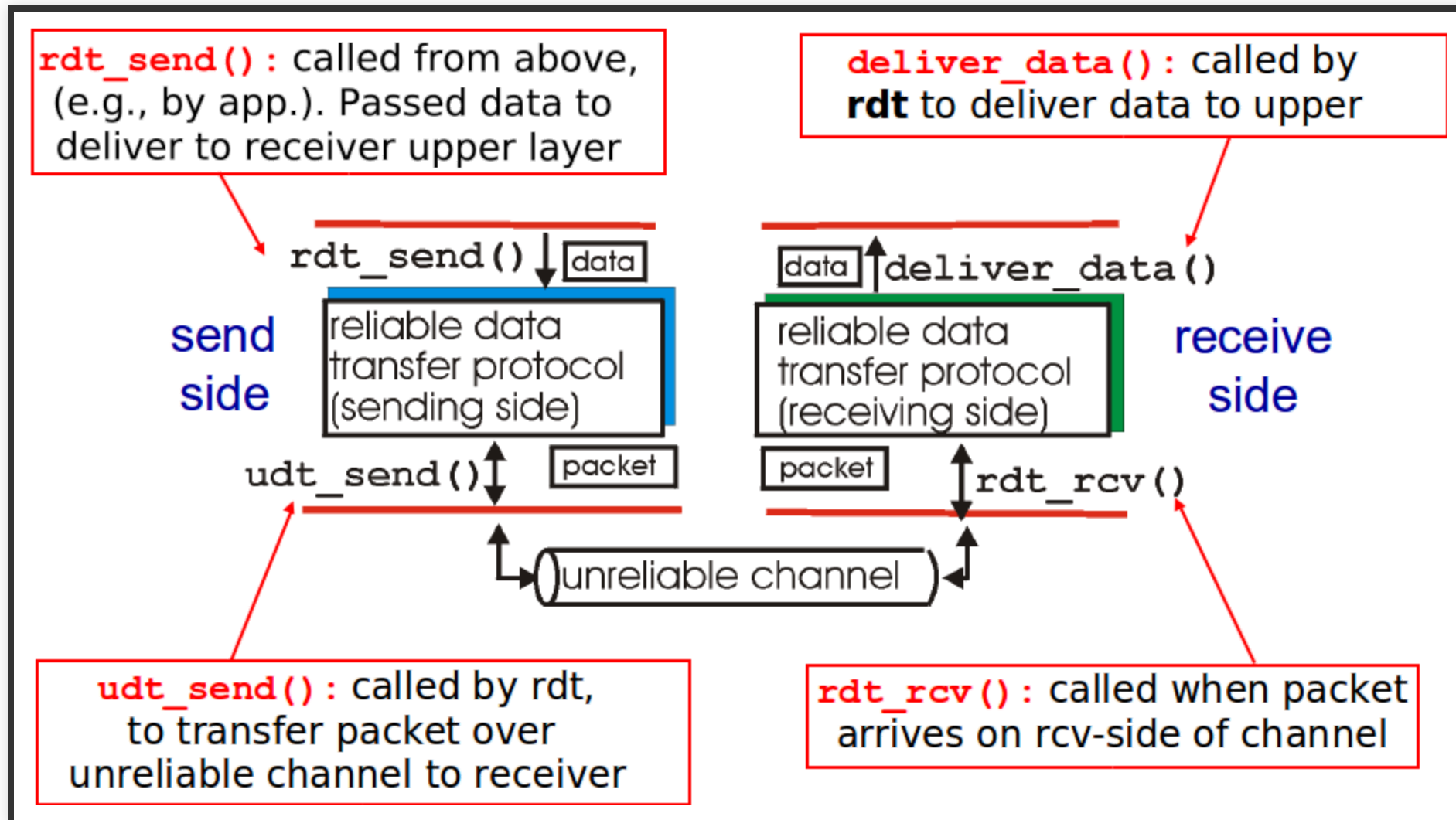
- Important in application, transport, link layers



PRINCIPLES OF RELIABLE DATA TRANSFER

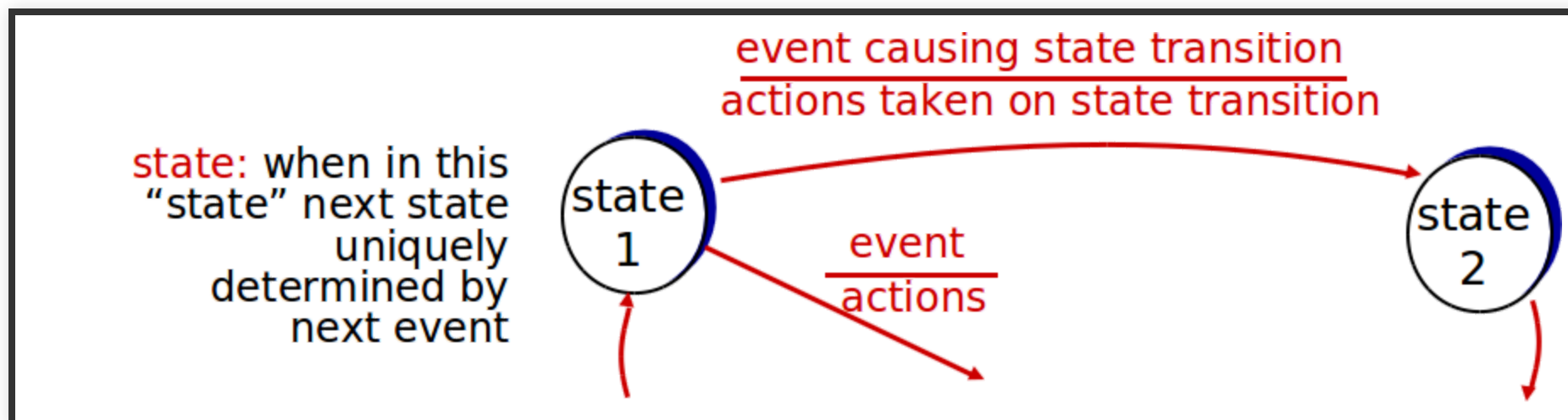
- 💡 characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

RELIABLE DATA TRANSFER: GETTING STARTED



RELIABLE DATA TRANSFER: GETTING STARTED

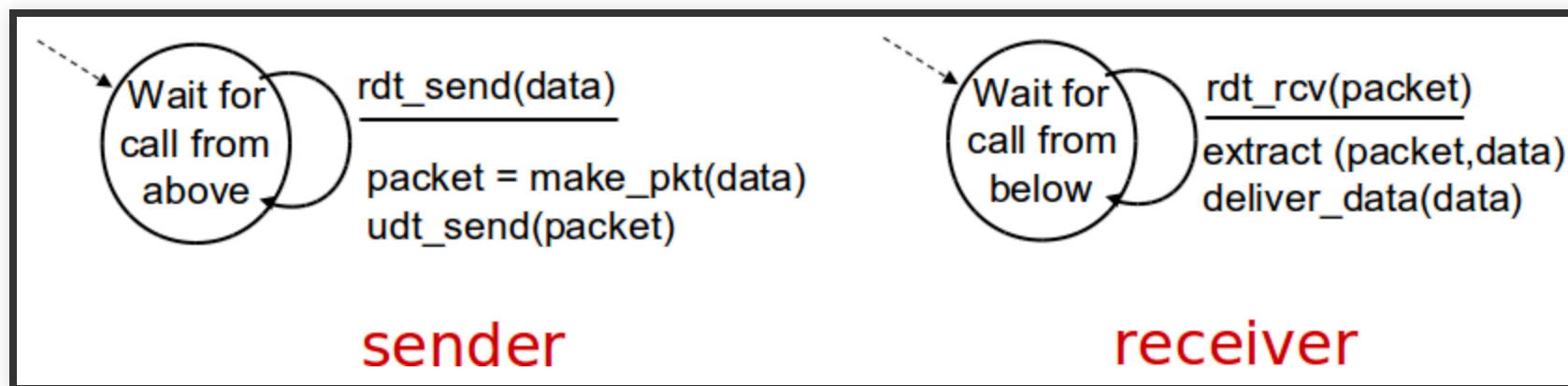
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
 - but control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver



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 reads data from underlying channel



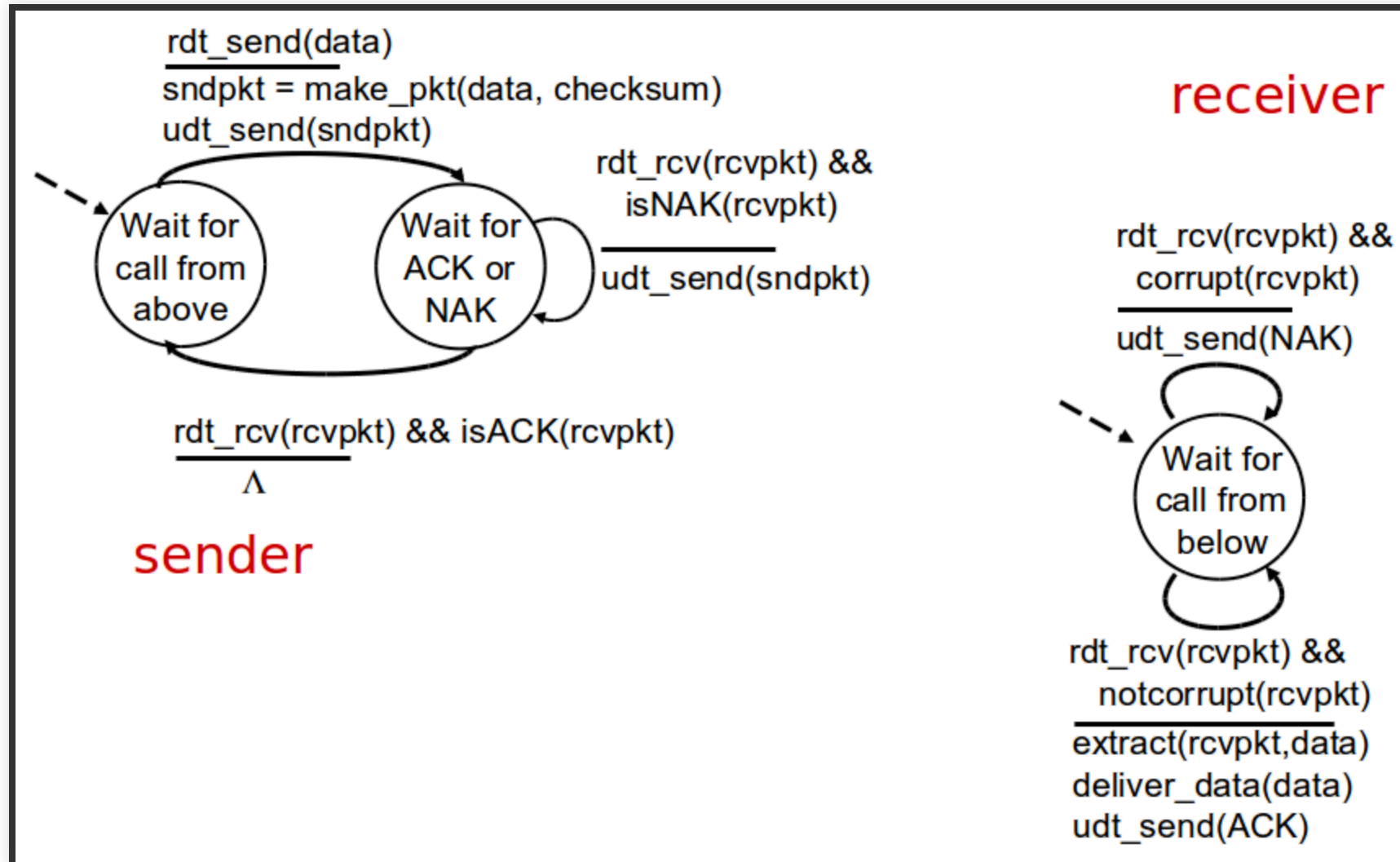
RDT2.0: CHANNEL WITH BIT ERRORS

- Underlying channel may flip bits in packet
 - Checksum to detect bit errors
- The question: how to recover from errors:
 - **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

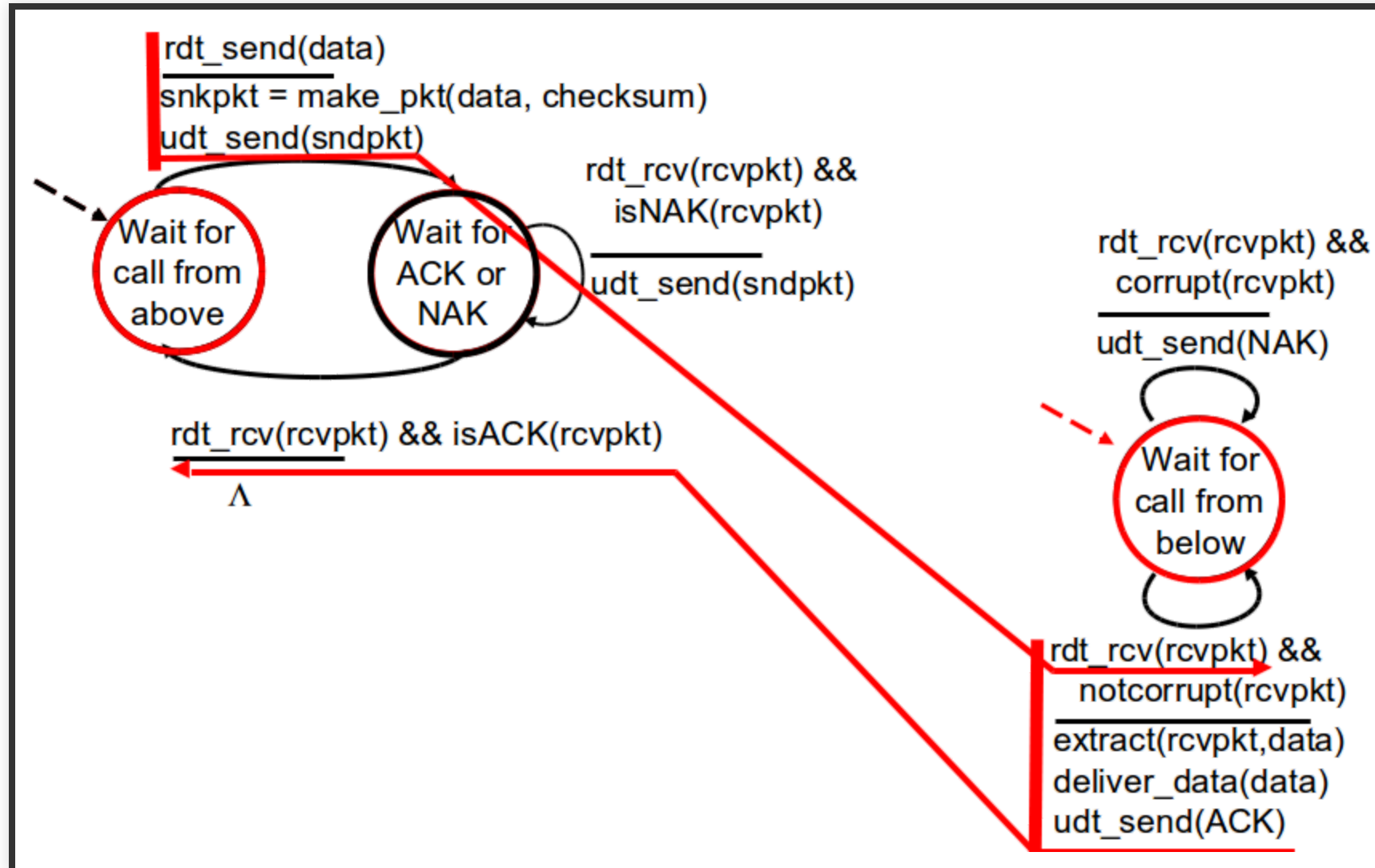
RDT2.0: CHANNEL WITH BIT ERRORS

- New mechanisms in Rdt2.0 (beyond Rdt1.0):
 - Error detection
 - Feedback: control messages (ACK,NAK) from receiver to sender

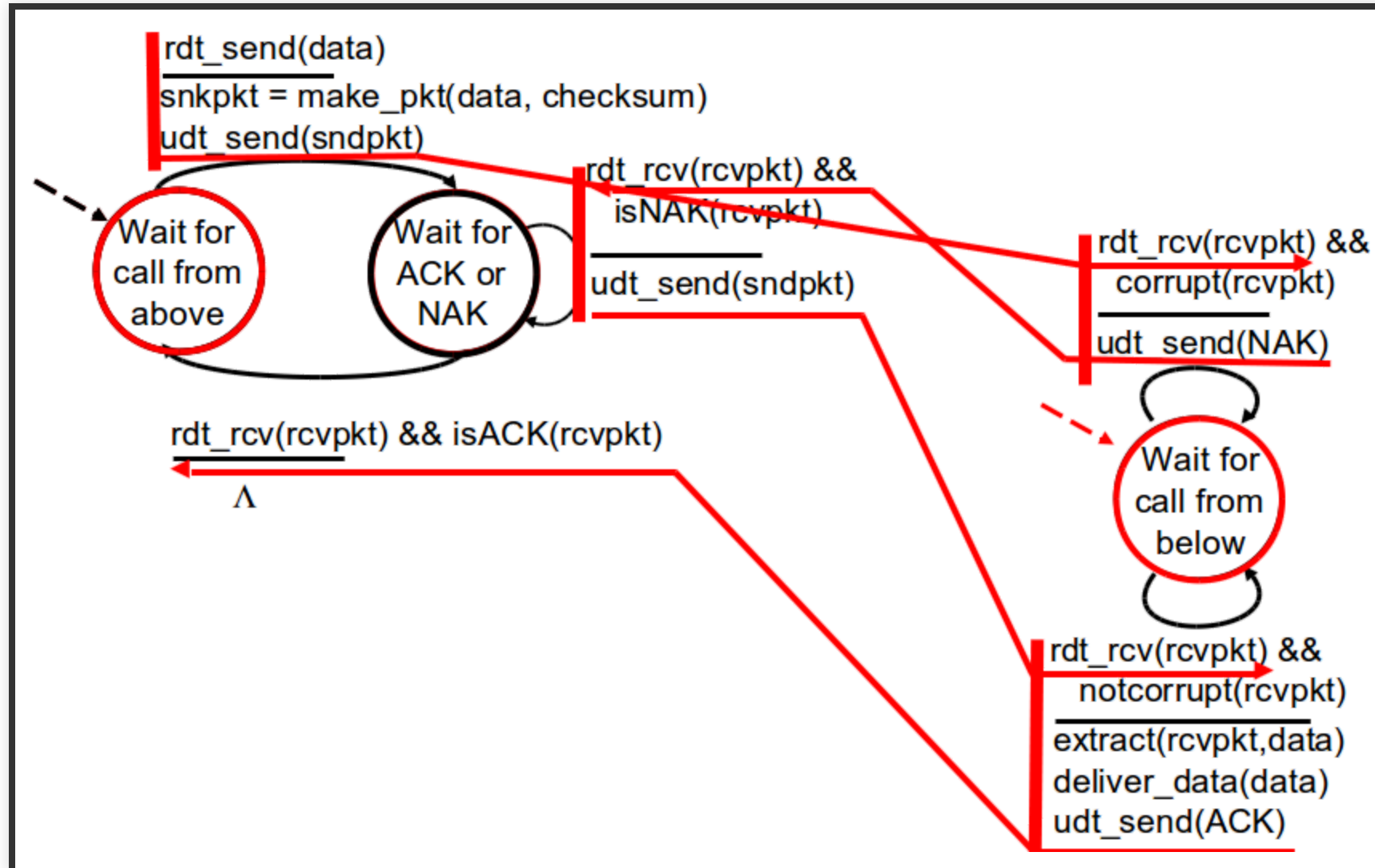
RDT2.0: FSM SPECIFICATION



RDT2.0: OPERATION WITH NO ERRORS



RDT2.0: ERROR SCENARIO



RDT2.0 HAS A FATAL FLAW!

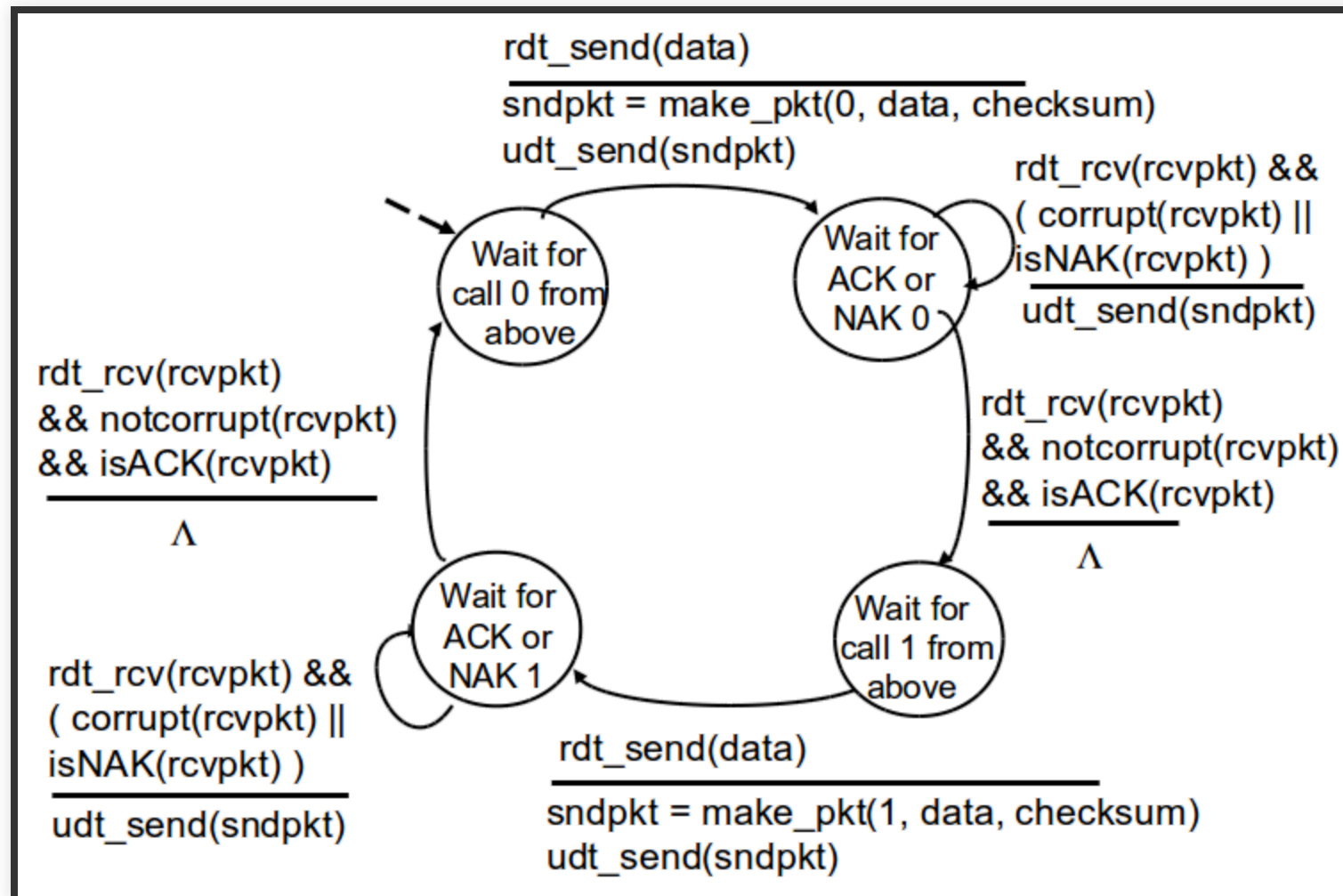
What happens if ACK/NAK corrupted?

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

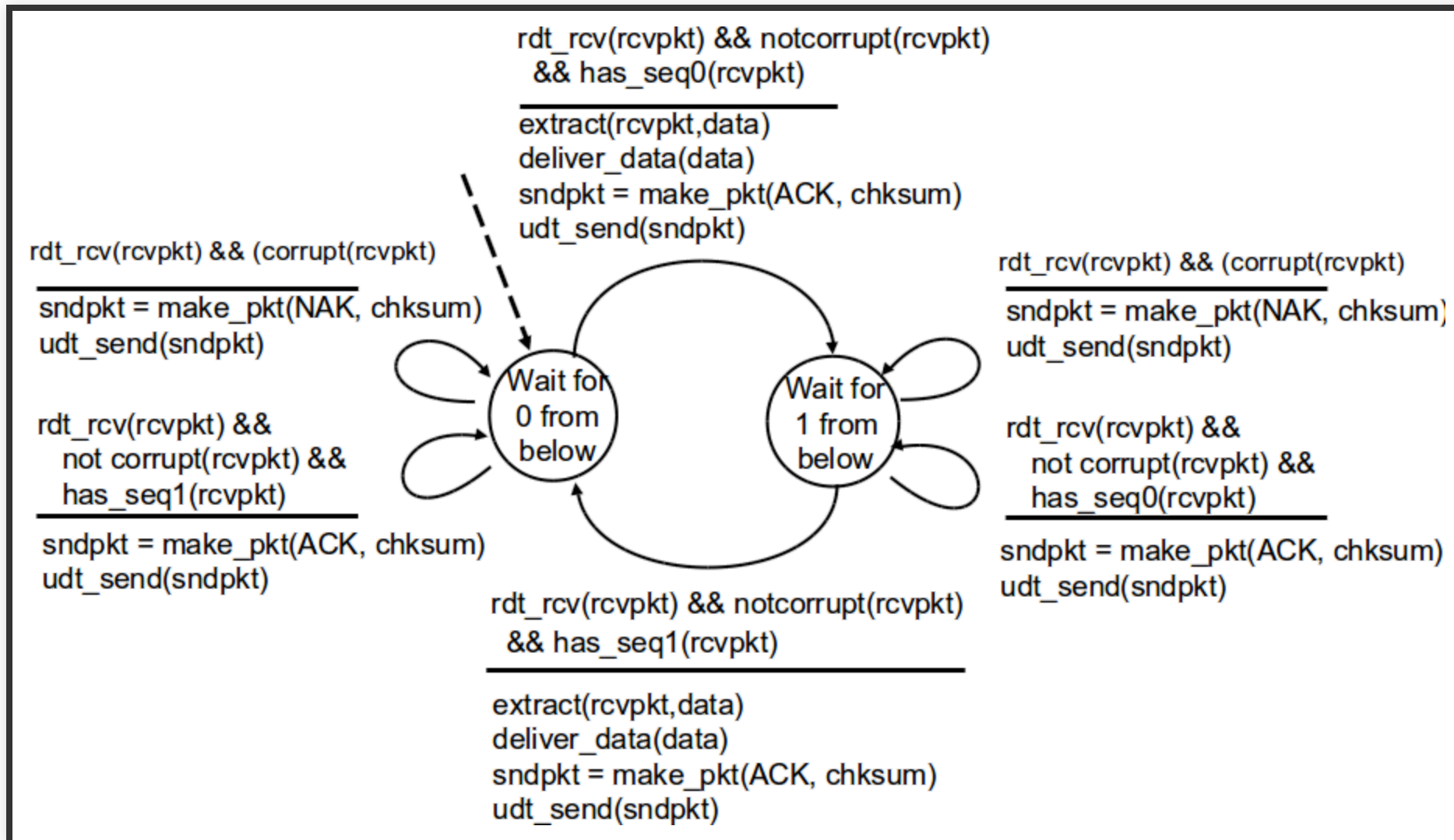
Handling duplicates:

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

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 "expected" pkt should have seq # of 0 or 1

RDT2.1: DISCUSSION

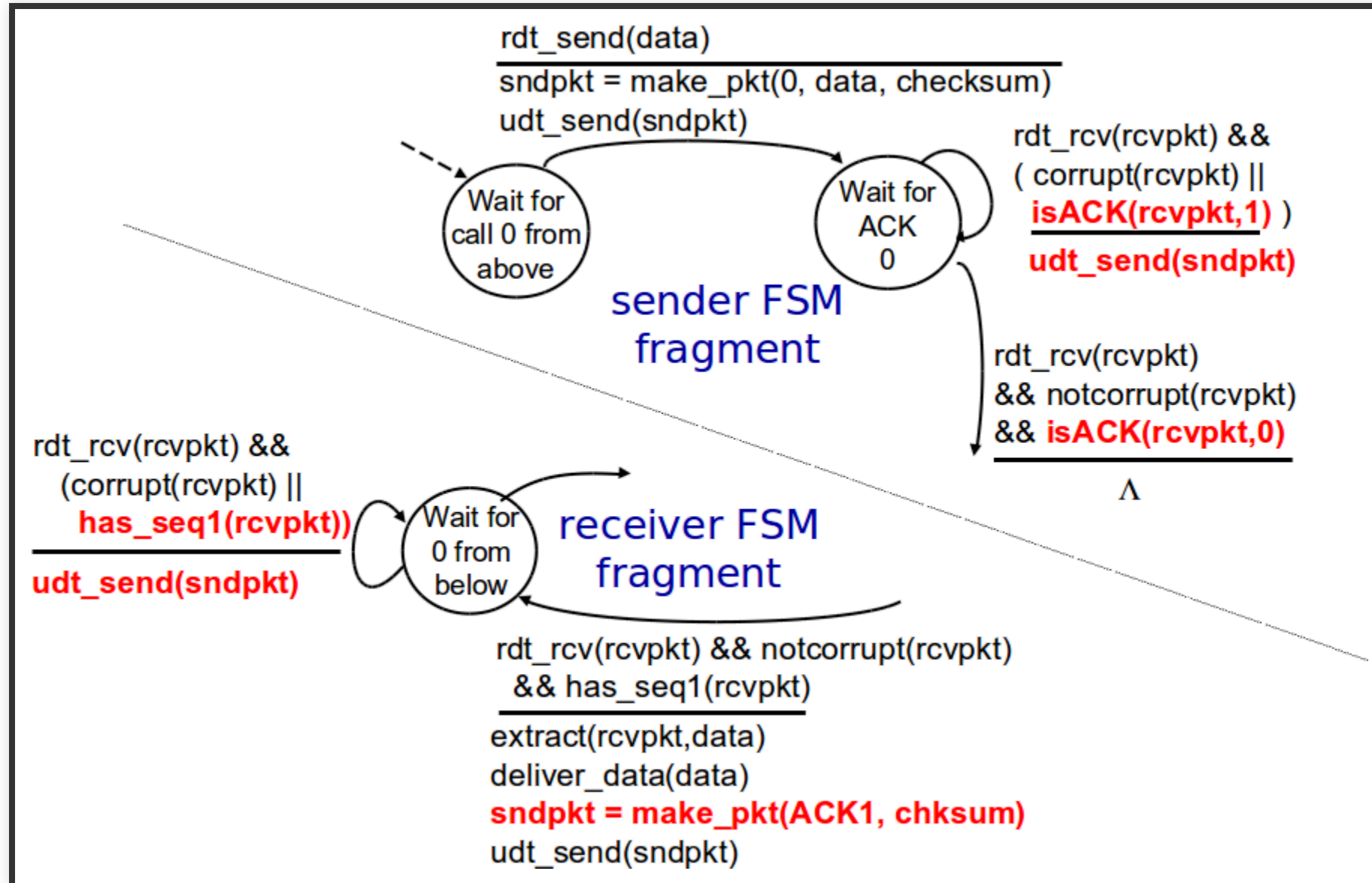
Receiver:

- 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, ACKs)

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

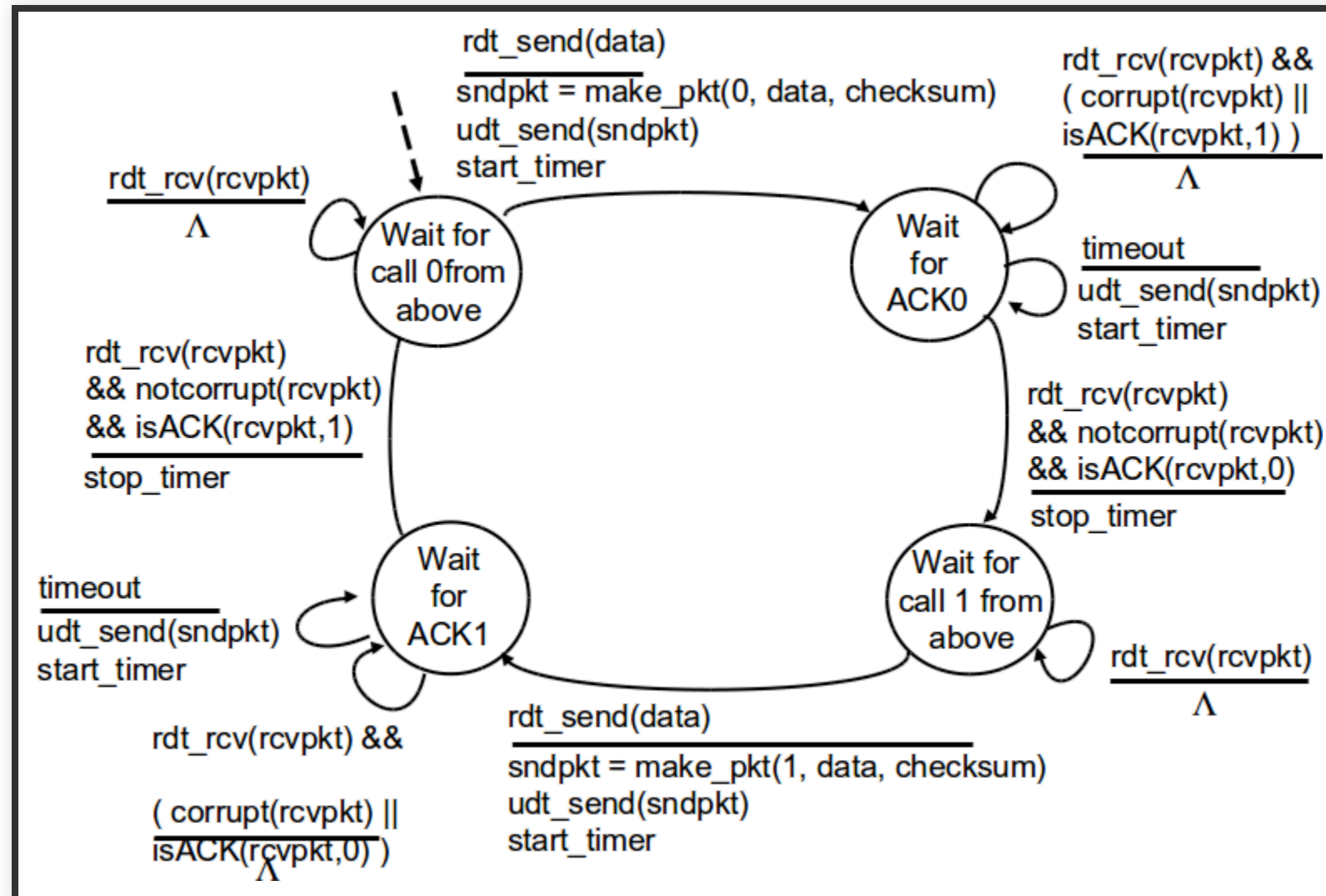
RDT3.0: CHANNELS WITH ERRORS AND LOSS

Approach:

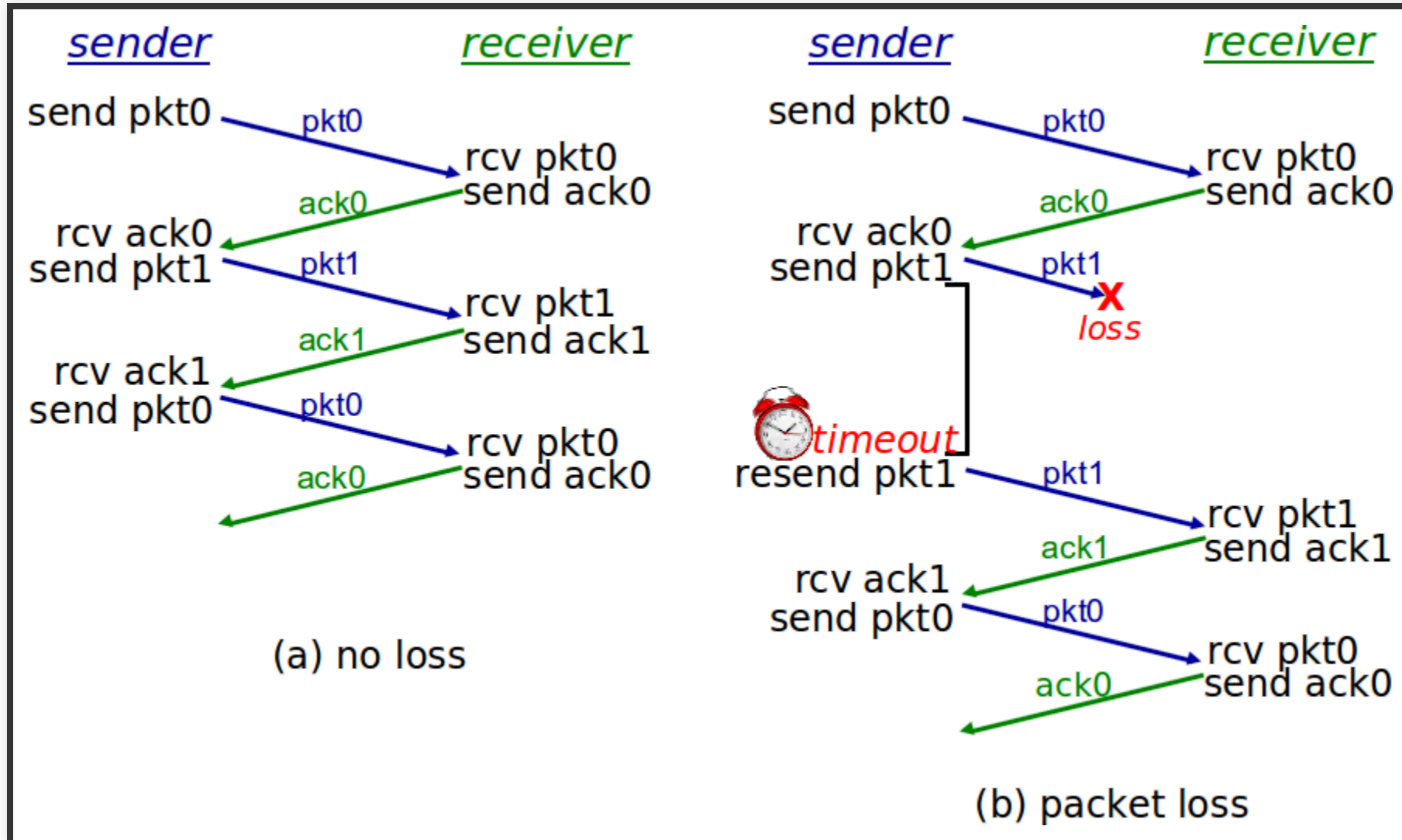
Sender waits “reasonable” amount of time for ACK

- 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 pkt being ACKed
- Requires countdown timer

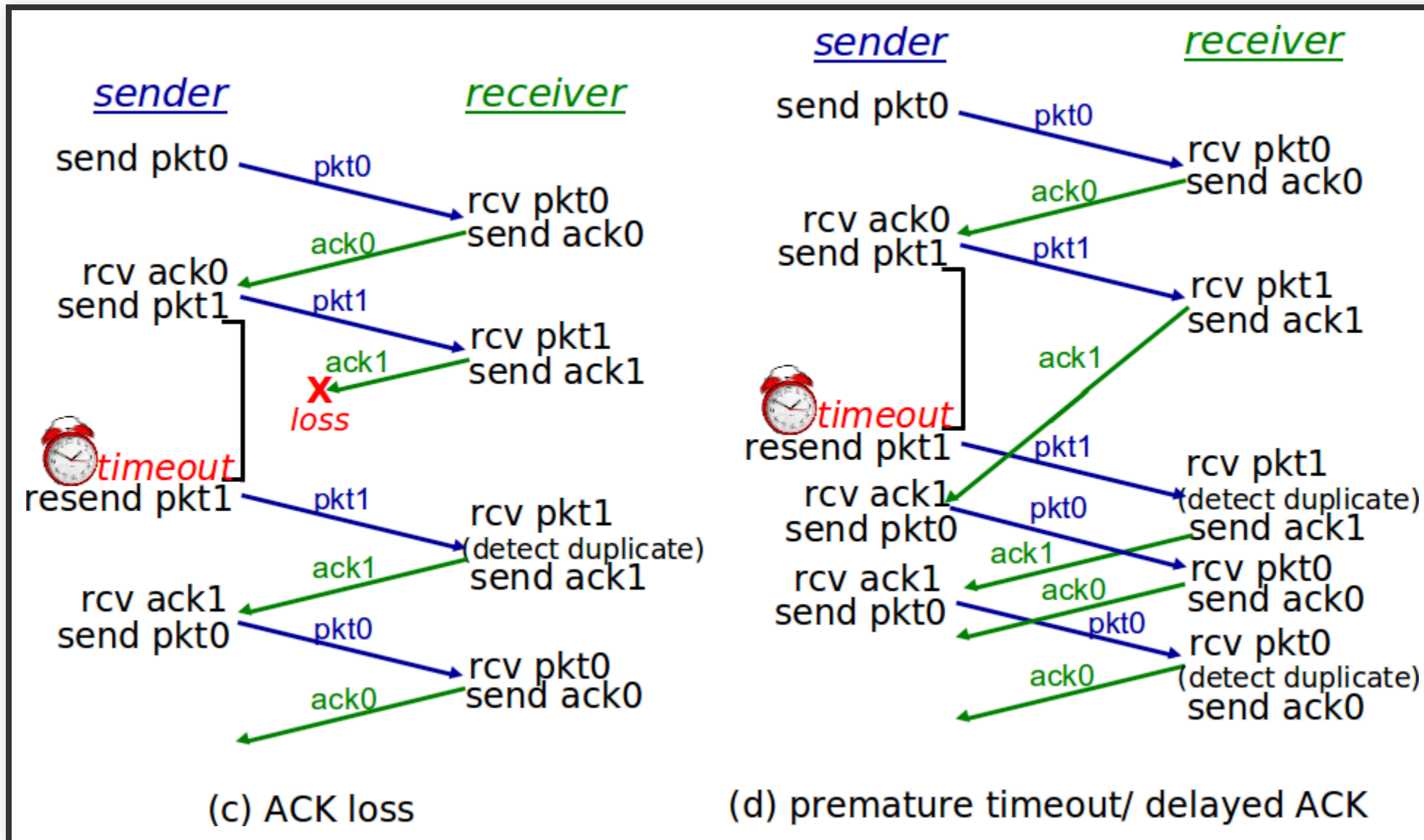
RDT3.0 SENDER



RDT3.0 IN ACTION



RDT3.0 IN ACTION



PERFORMANCE OF RDT3.0

Rdt3.0 is correct, but performance stinks

e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

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

$$U_{\text{sender}} = (L/R)/(RTT + L/R) = 0.008/30.008 = 0.00027$$

- if $RTT=30$ msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
 - **Network protocol limits use of physical resources!**

MECHANISMS

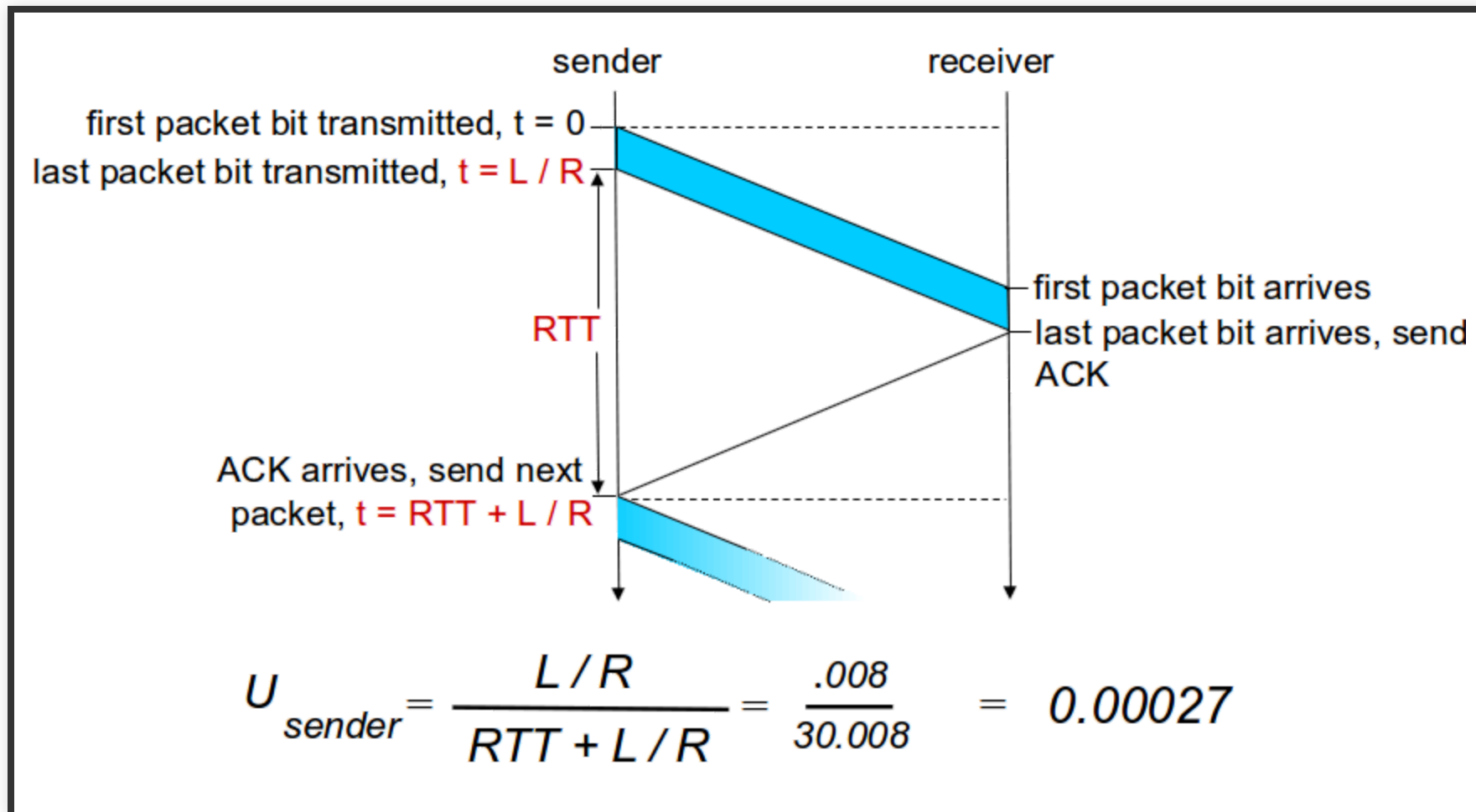
- Sequence Numbers
- Acknowledgements
- Timers
- Checksum

STOP AND WAIT

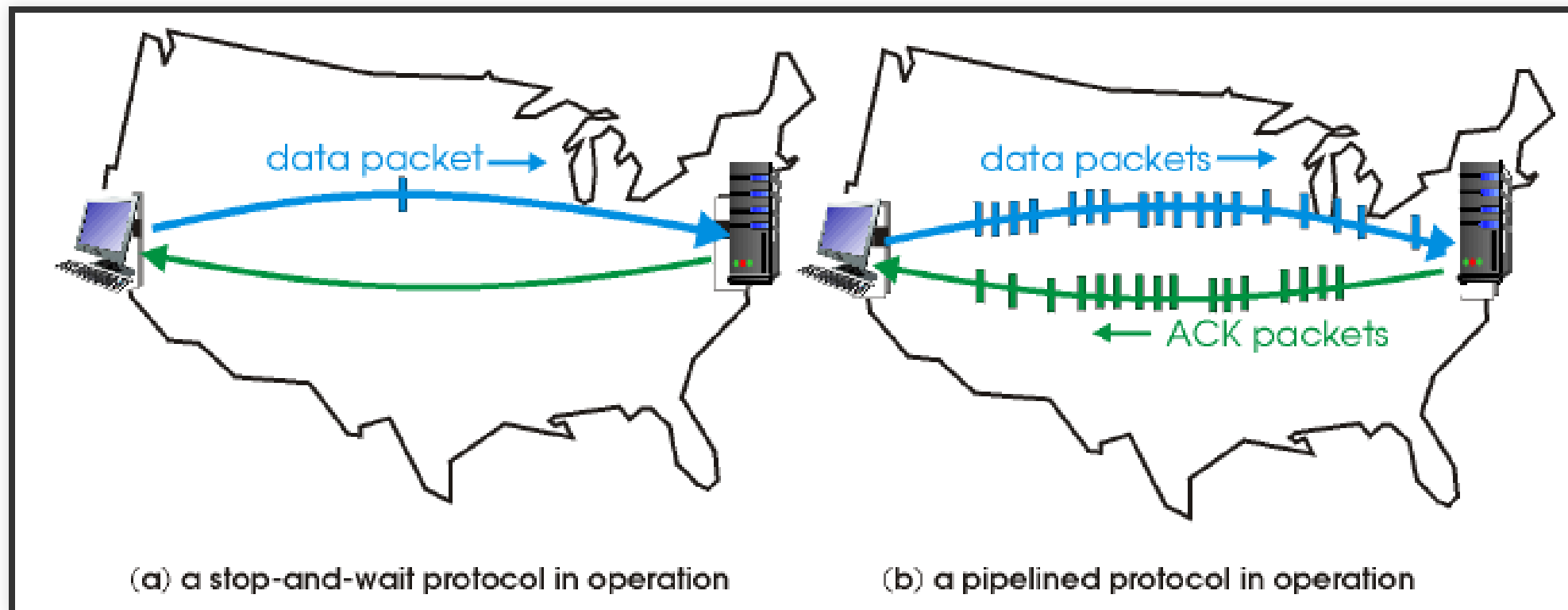
! Stop and wait

sender sends one packet, then waits for receiver response

RDT3.0: STOP-AND-WAIT OPERATION



PIPELINED PROTOCOLS



PIPELINED PROTOCOLS

! Pipelining:

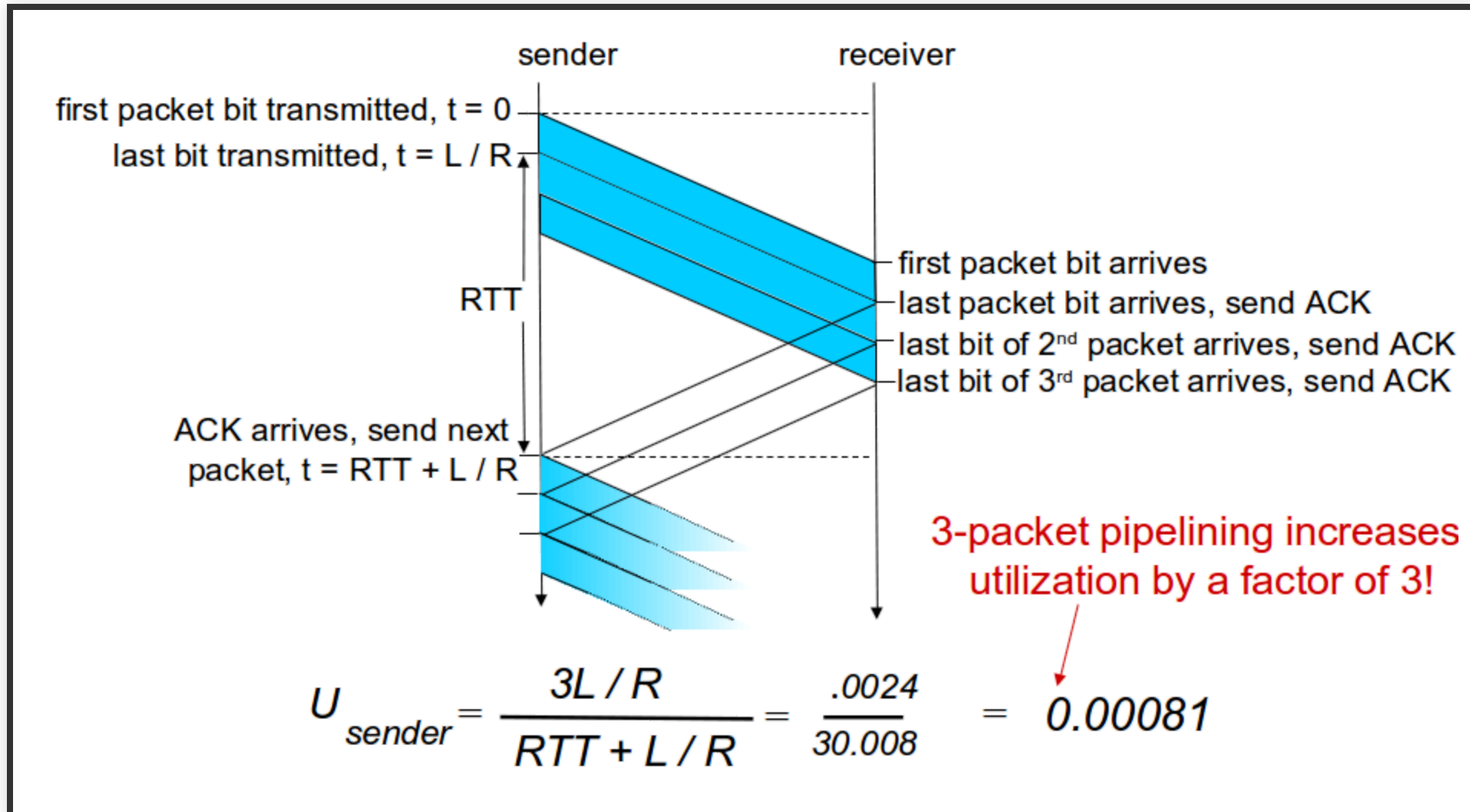
Sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

Two generic forms of pipelined protocols:

- go-Back-N
- selective repeat

PIPELINING: INCREASED UTILIZATION



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

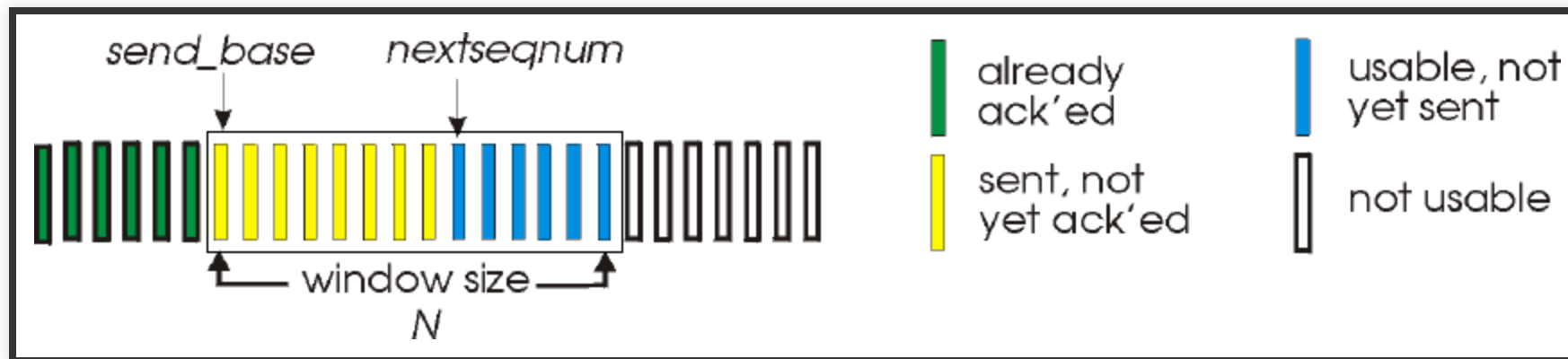
PIPELINED PROTOCOLS: OVERVIEW

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

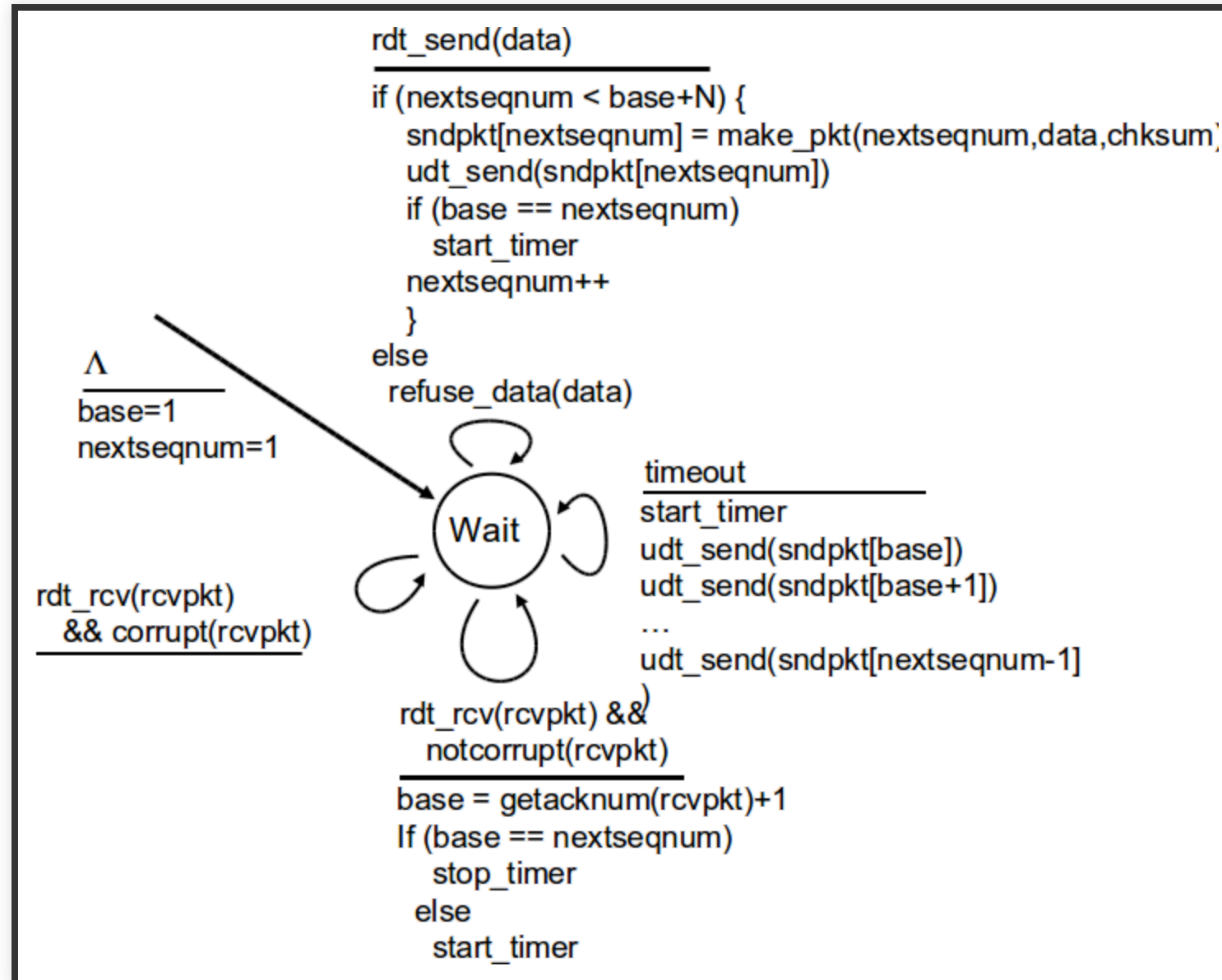
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



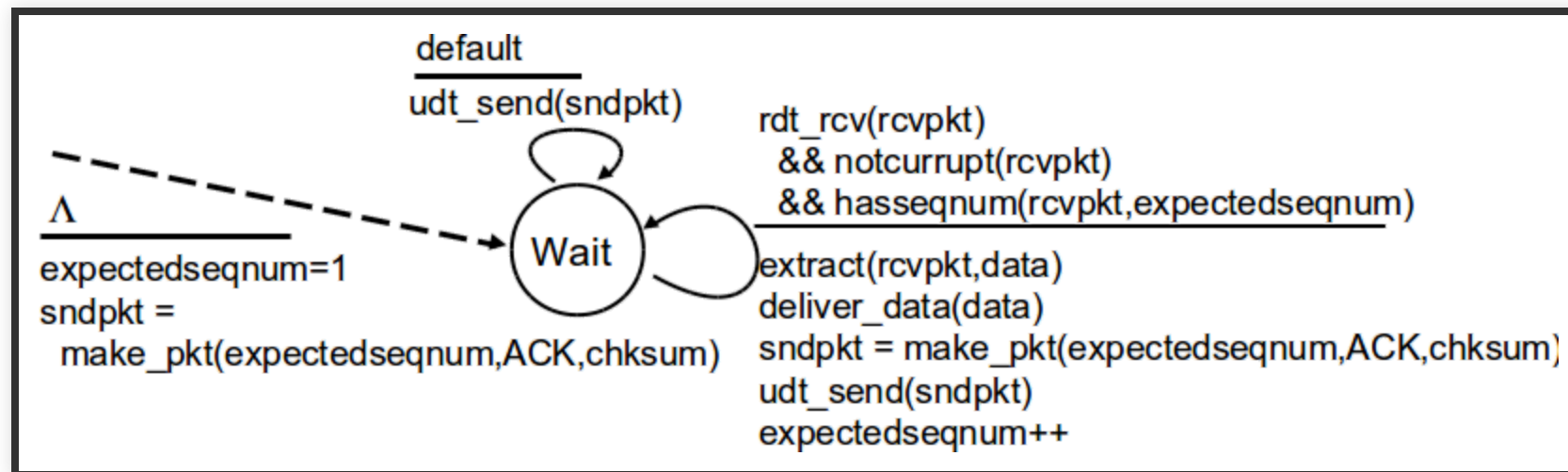
GO-BACK-N: SENDER

- 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 packet n and all higher seq # pkts in window

GBN: SENDER EXTENDED FSM



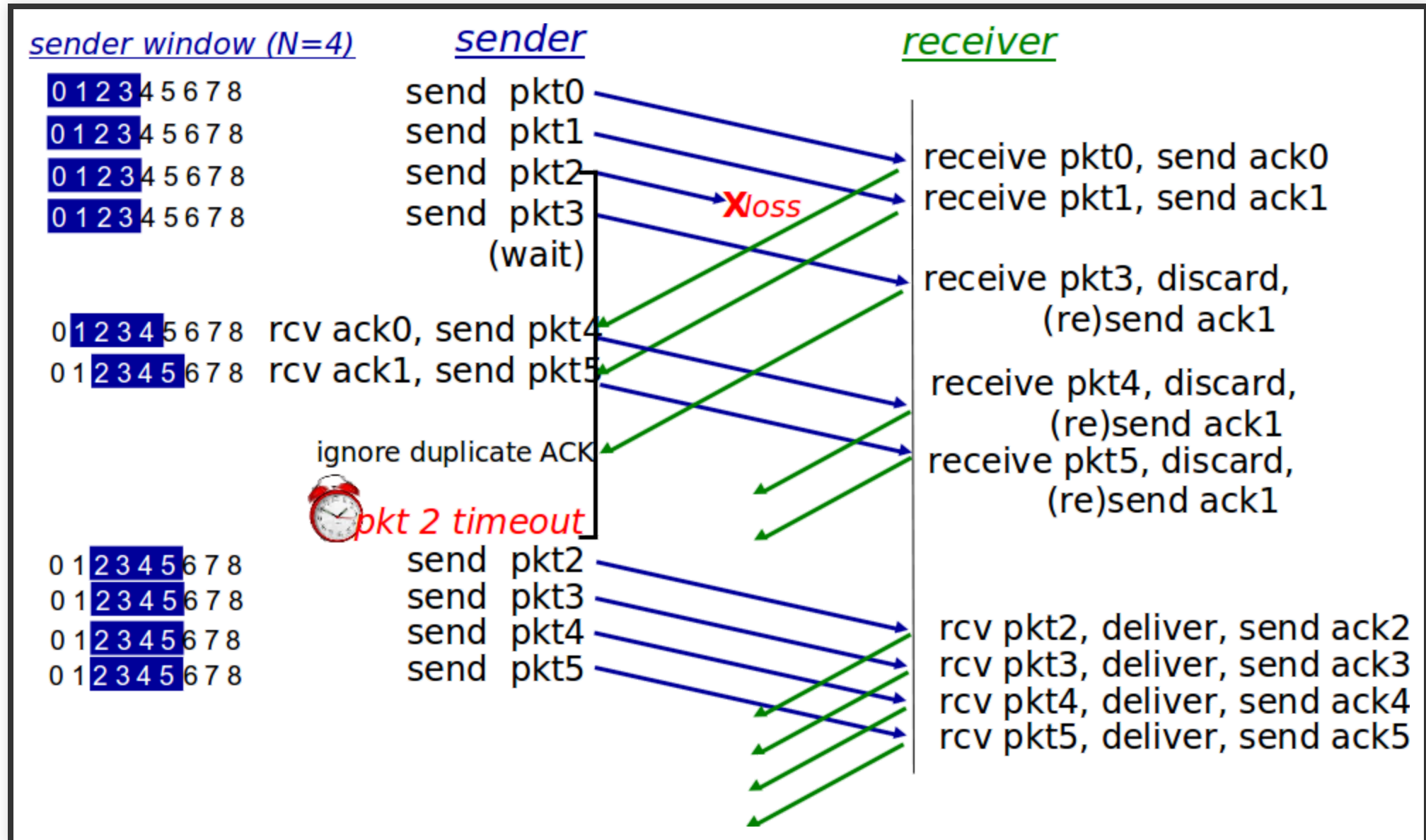
GBN: RECEIVER EXTENDED FSM



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 #

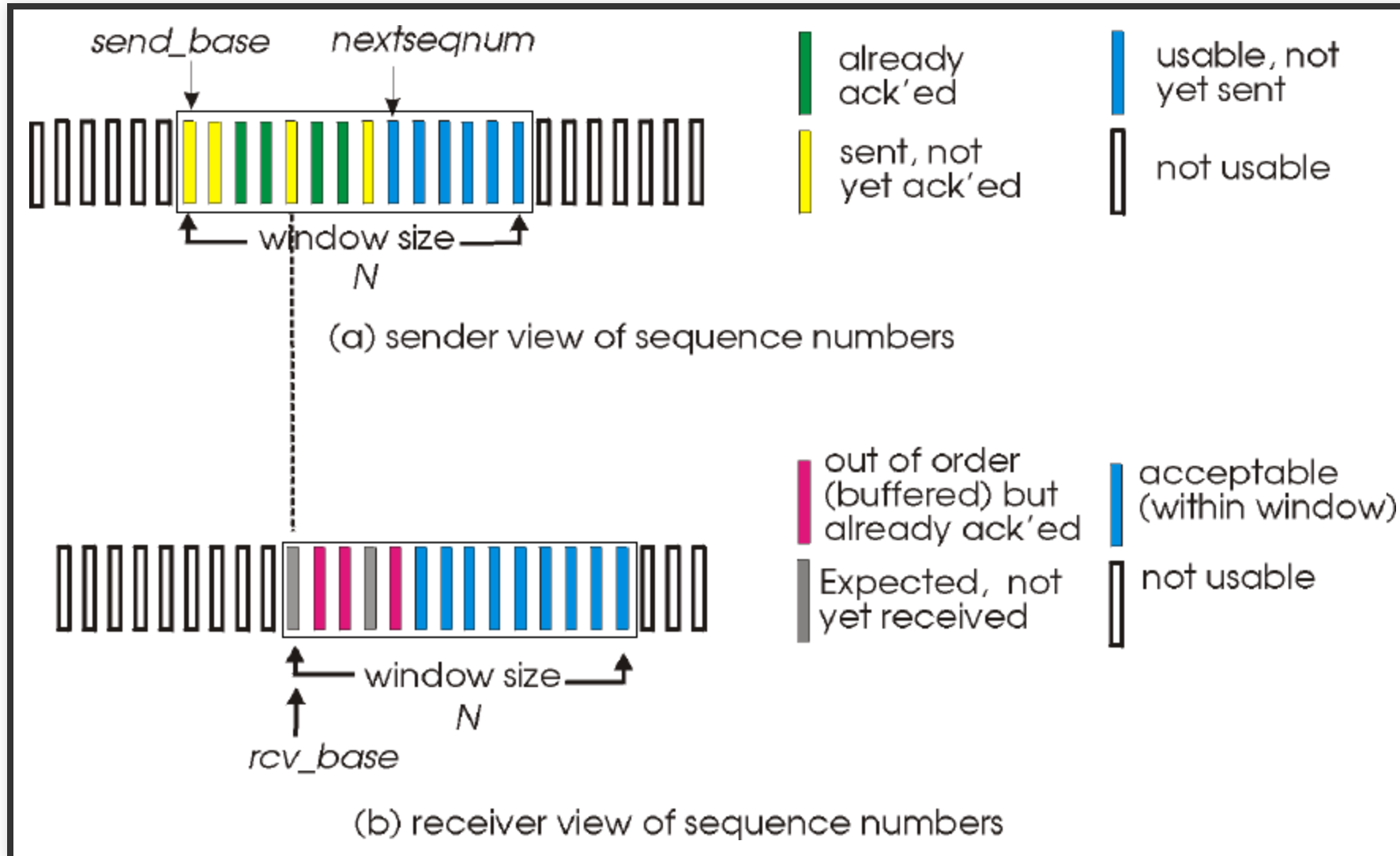
GBN IN ACTION



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
- sender window
 - N consecutive seq #'s
 - limits seq #'s of sent, unACKed pkts

SELECTIVE REPEAT: SENDER, RECEIVER WINDOWS



SELECTIVE REPEAT - SENDER

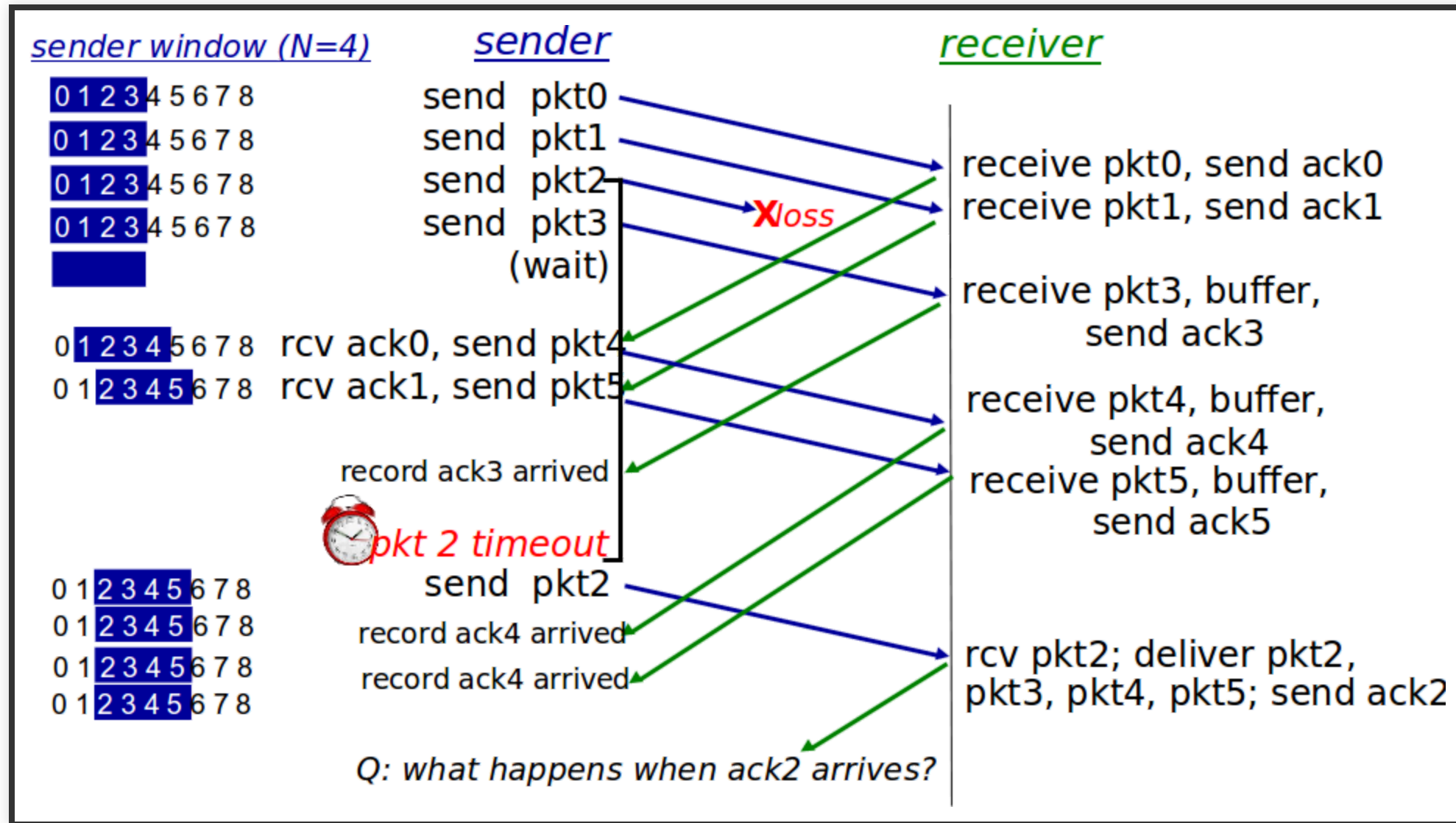
Data from above:

- 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 #

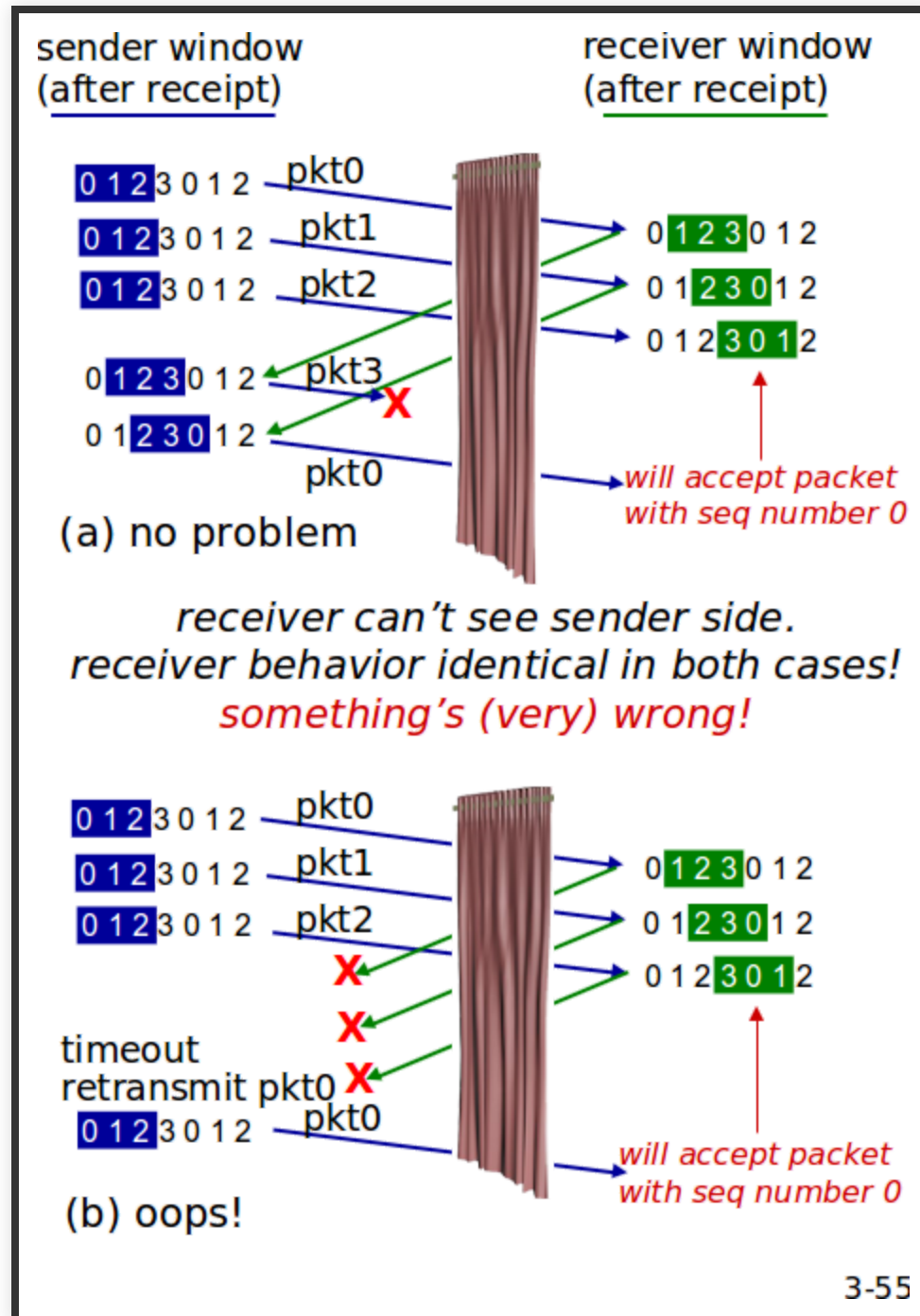
SELECTIVE REPEAT - RECEIVER

- 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




SELECTIVE REPEAT: DILEMMA



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)?

SUMMARY

- principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
- UDP