

TDT506: Computer Networks

Instructor: Niklas Carlsson

Email: niklas.carlsson@liu.se

Notes derived from "*Computer Networking: A Top Down Approach*", by Jim Kurose and Keith Ross, Addison-Wesley.

The slides are adapted and modified based on slides from the book's companion Web site, as well as modified slides by Anirban Mahanti and Carey Williamson.

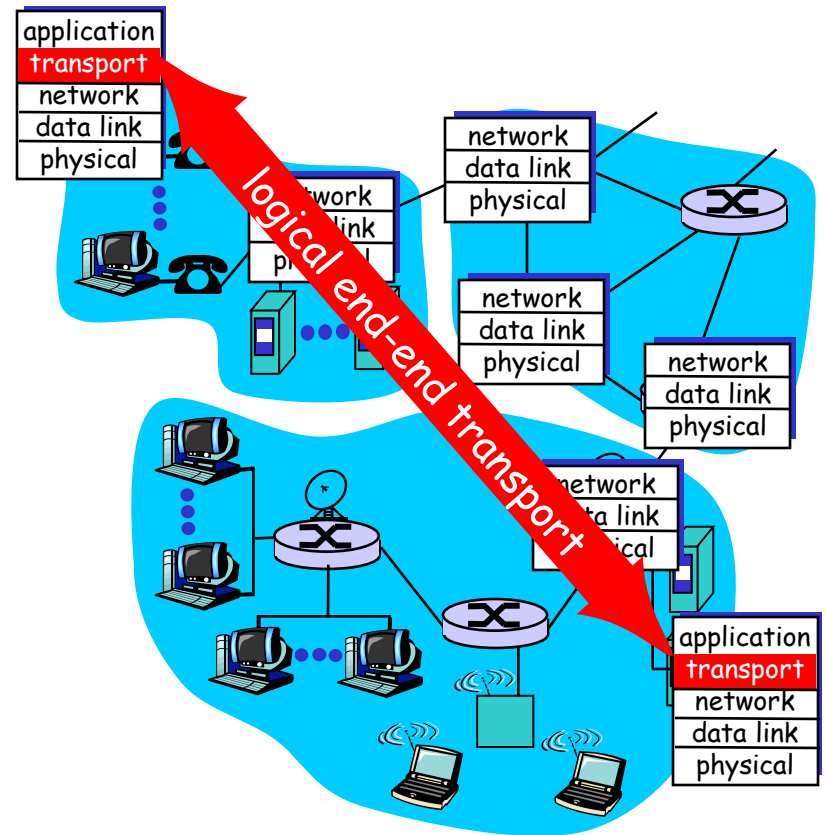
Chapter 3: Transport Layer

Our goals:

- ❑ understand principles behind transport layer services:
 - multiplexing and demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- ❑ learn about transport layer protocols in the Internet:
 - UDP: connectionless transport
 - TCP: connection-oriented transport
 - TCP congestion control

Transport services and protocols

- ❑ provide *logical communication* between app processes running on different hosts
- ❑ transport protocols run in end systems
 - send side: breaks app messages into *segments*, passes to network layer
 - rcv side: reassembles segments into messages, passes to app layer
- ❑ more than one transport protocol available to apps
 - Internet: TCP and UDP



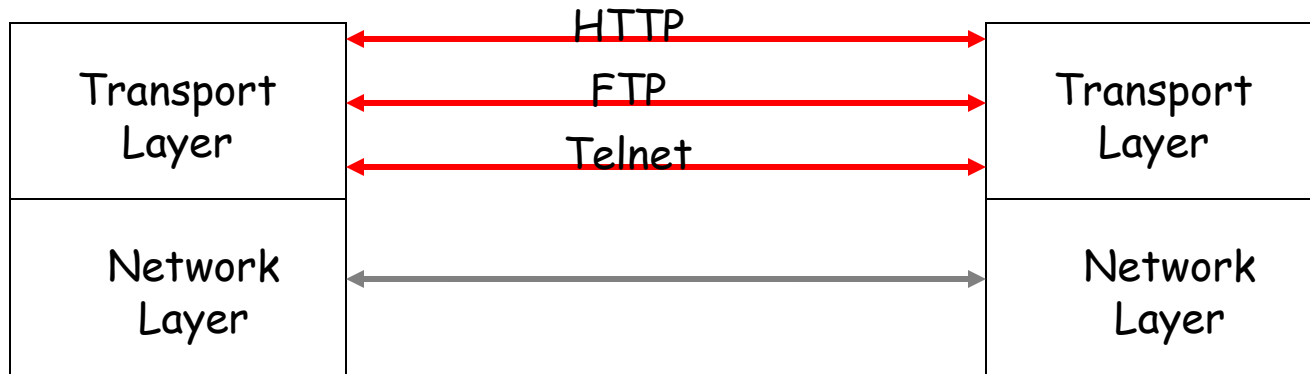
Transport vs. Network Layer

- ❑ *transport layer*: logical communication between processes
 - relies on, enhances, network layer services
 - PDU: Segment
 - extends "host-to-host" communication to "process-to-process" communication
- ❑ *network layer*: logical communication between hosts
 - PDU: Datagram
 - Datagrams may be lost, duplicated, reordered in the Internet - "best effort" service

TCP/IP Transport Layer Protocols

- ❑ reliable, in-order delivery (TCP)
 - connection setup
 - flow control
 - congestion control
- ❑ unreliable, unordered delivery: UDP
 - no-frills extension of “best-effort” IP
 - What does UDP provide in addition to IP?
- ❑ services not provided by IP (network layer):
 - delay guarantees
 - bandwidth guarantees

Multiplexing/Demultiplexing



- ❑ Use same communication channel between hosts for several logical communication processes
- ❑ How does Mux/DeMux work?
 - Sockets: doors between process & host
 - UDP socket: (dest. IP, dest. Port)
 - TCP socket: (src. IP, src. port, dest. IP, dest. Port)

Connectionless demux

- ❑ UDP socket identified by two-tuple:
 - (dest IP address, dest port number)
- ❑ When host receives UDP segment:
 - checks destination port number in segment
 - directs UDP segment to socket with that port number
- ❑ IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connection-oriented demux

- ❑ TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- ❑ recv host 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

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
- ❑ Why use UDP?
 - No connection establishment cost (critical for some applications, e.g., DNS)
 - No connection state
 - Small segment headers (only 8 bytes)
 - Finer application control over data transmission

UDP Segment Structure

- often used for voice and some real-time (streaming) apps

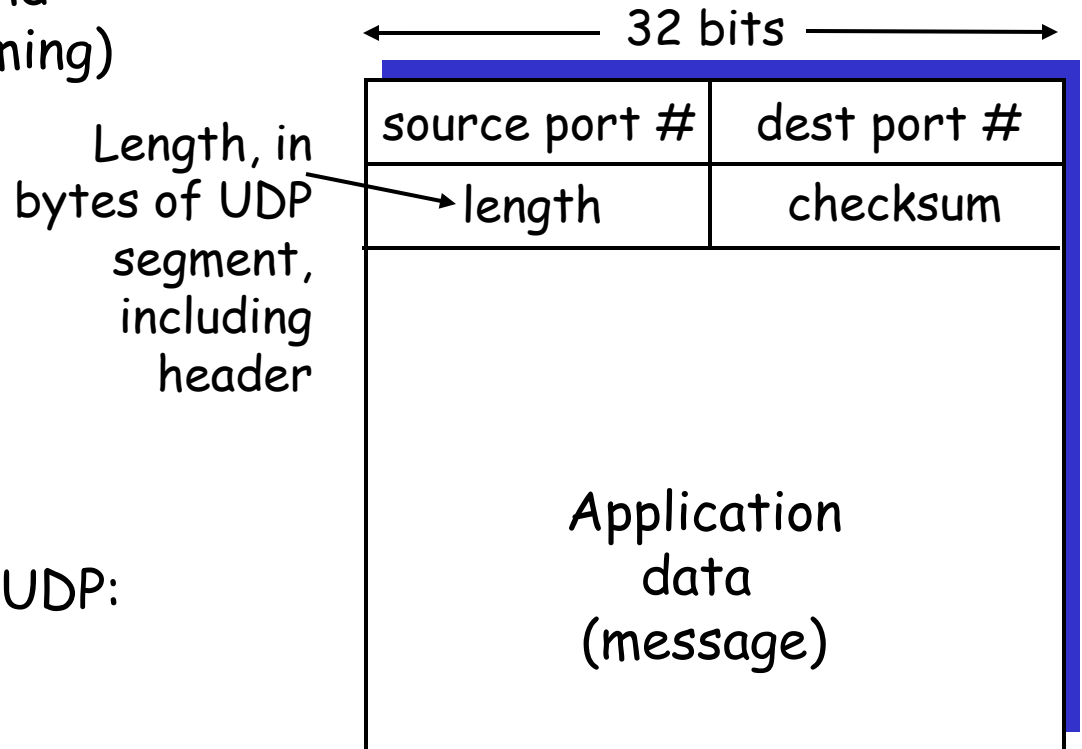
- loss tolerant
- rate sensitive

- other UDP uses

- DNS
- SNMP

- reliable transfer over UDP:
add reliability at application layer

- application-specific error recovery!



UDP segment format

UDP checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

Sender:

- ❑ treat segment contents as sequence of 16-bit integers
- ❑ checksum: addition (1's complement sum) of segment contents
- ❑ sender puts checksum value into UDP checksum field

Receiver:

- ❑ compute checksum of received segment
- ❑ check if computed checksum equals checksum field value:
 -

UDP checksum

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

Sender:

- ❑ treat segment contents as sequence of 16-bit integers
- ❑ checksum: addition (1's complement sum) of segment contents
- ❑ sender puts checksum value into UDP checksum field

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

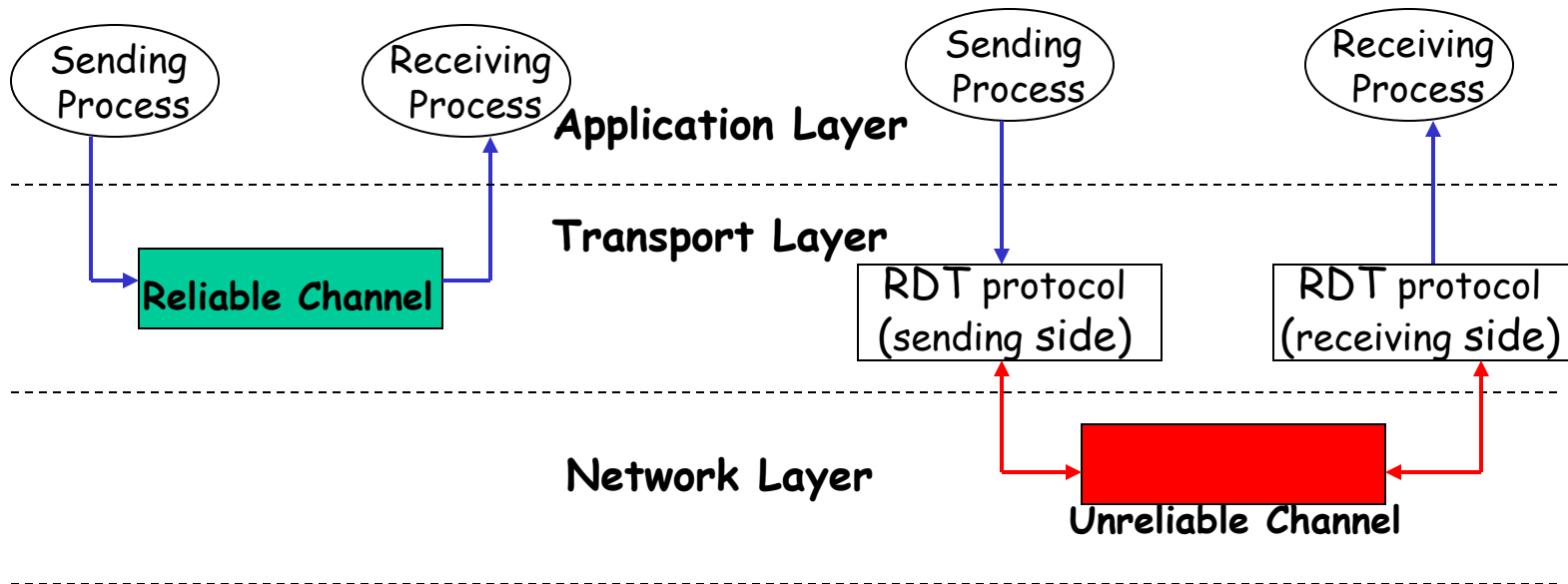
- ❑ **Note:** When adding numbers, a carryout from the most significant bit needs to be added to the result
- ❑ Example: add two 16-bit integers
- ❑ Weak error protection? Why is it useful?

		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
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
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

Reliable Data Transfer

Principles of Reliable Data Transfer

- ❑ important in application, transport, and link layers
- ❑ top-10 list of important networking topics!

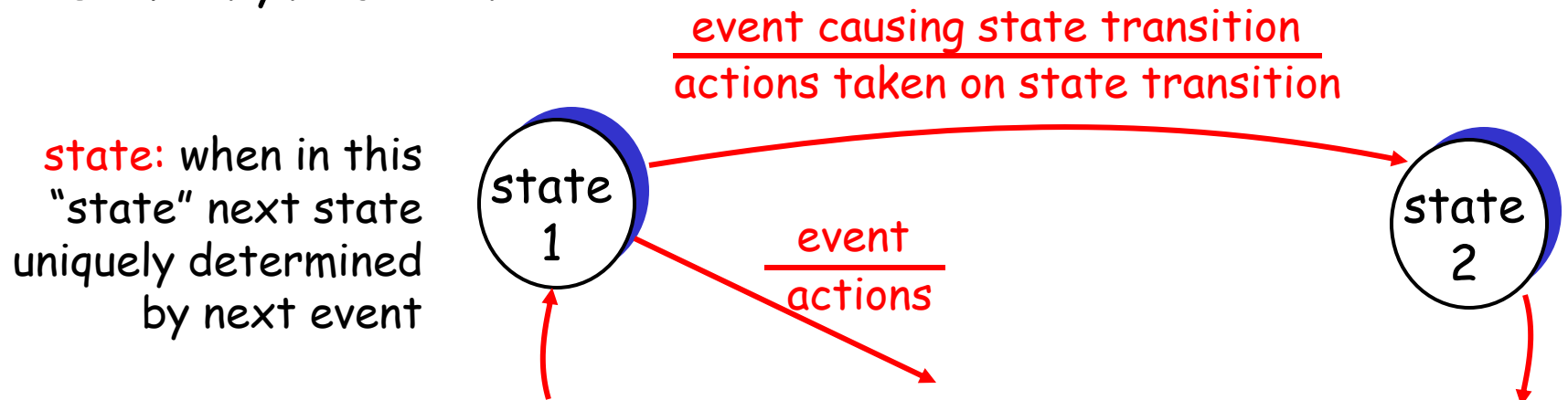


- ❑ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable Data Transfer: FSMs

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: Data Transfer over a Perfect Channel

- underlying channel perfectly reliable
 - no bit errors
 - no loss of packets

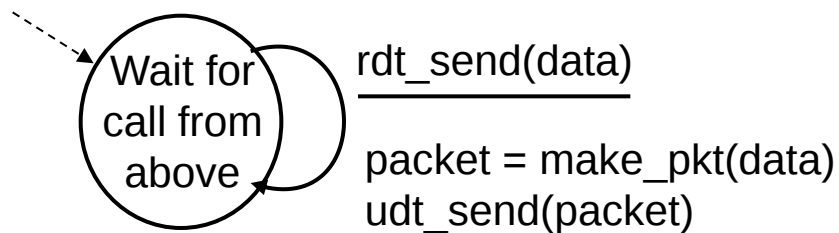
Rdt1.0: Data Transfer over a Perfect Channel

- underlying channel perfectly reliable

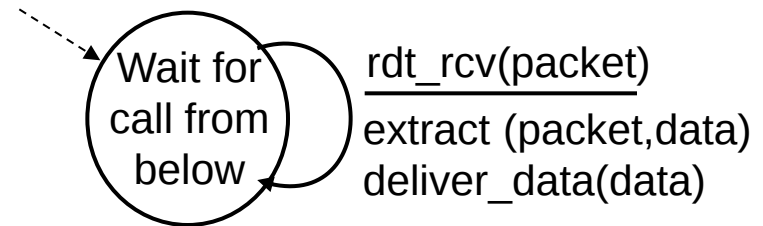
- no bit errors
- no loss of packets

- 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 [stop & wait protocol]

□ Assumptions

- All packets are received
- Packets may be corrupted (i.e., bits may be flipped)
- Checksum to detect bit errors

Rdt2.0: channel with bit errors [stop & wait protocol]

❑ Assumptions

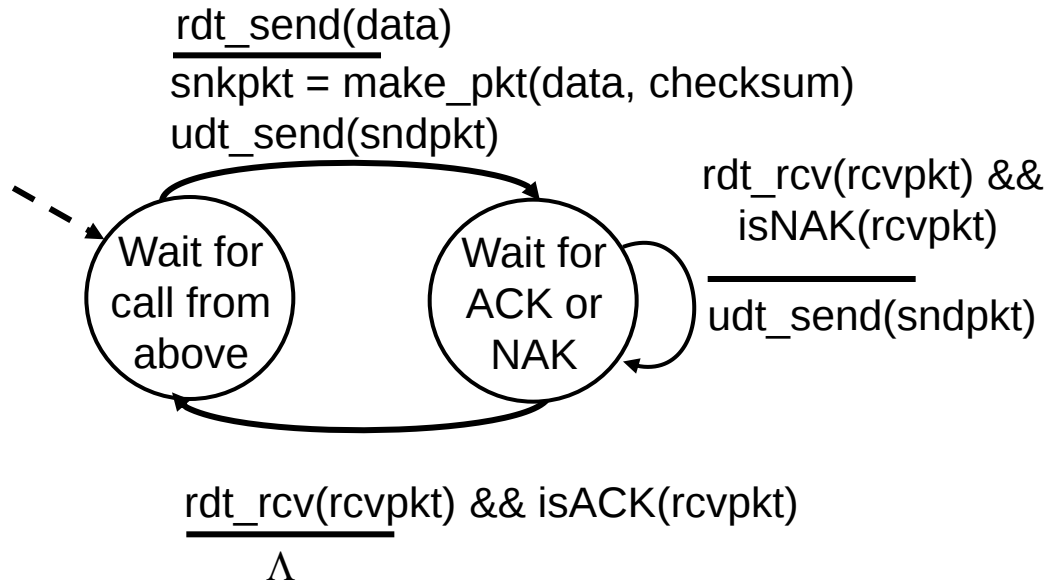
- All packets are received
- Packets may be corrupted (i.e., bits may be flipped)
- Checksum to detect bit errors

❑ How to recover from errors? Use ARQ mechanism

- *acknowledgements (ACKs)*: receiver explicitly tells sender that packet received correctly
- *negative acknowledgements (NAKs)*: receiver explicitly tells sender that packet had errors
- sender retransmits pkt on receipt of NAK

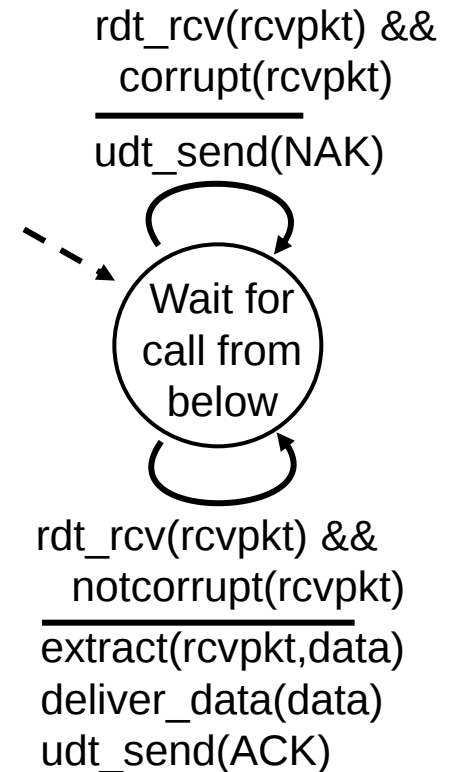
❑ What about error correcting codes?

rdt2.0: FSM specification

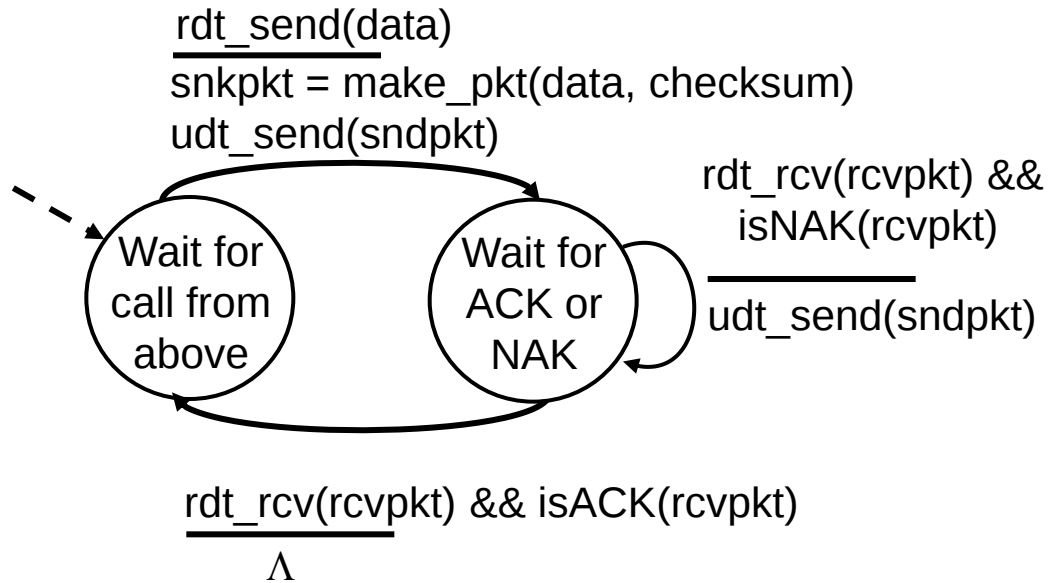


sender

receiver



rdt2.0: Observations

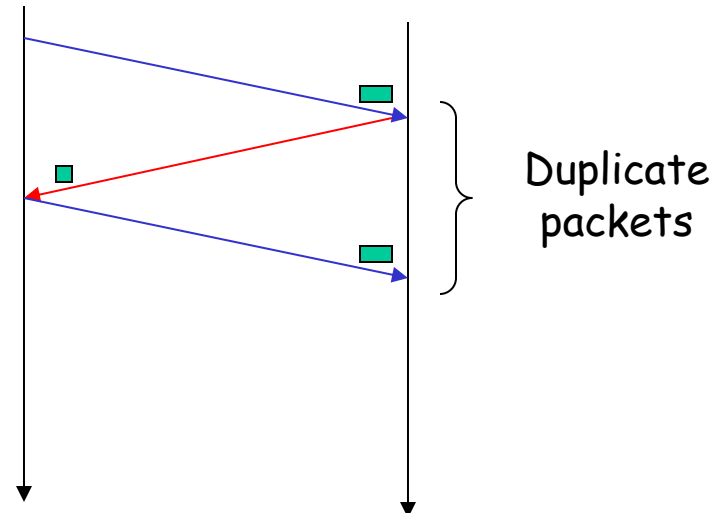


sender

1. A stop-and-Wait protocol

2. What happens when ACK or NAK has bit errors?

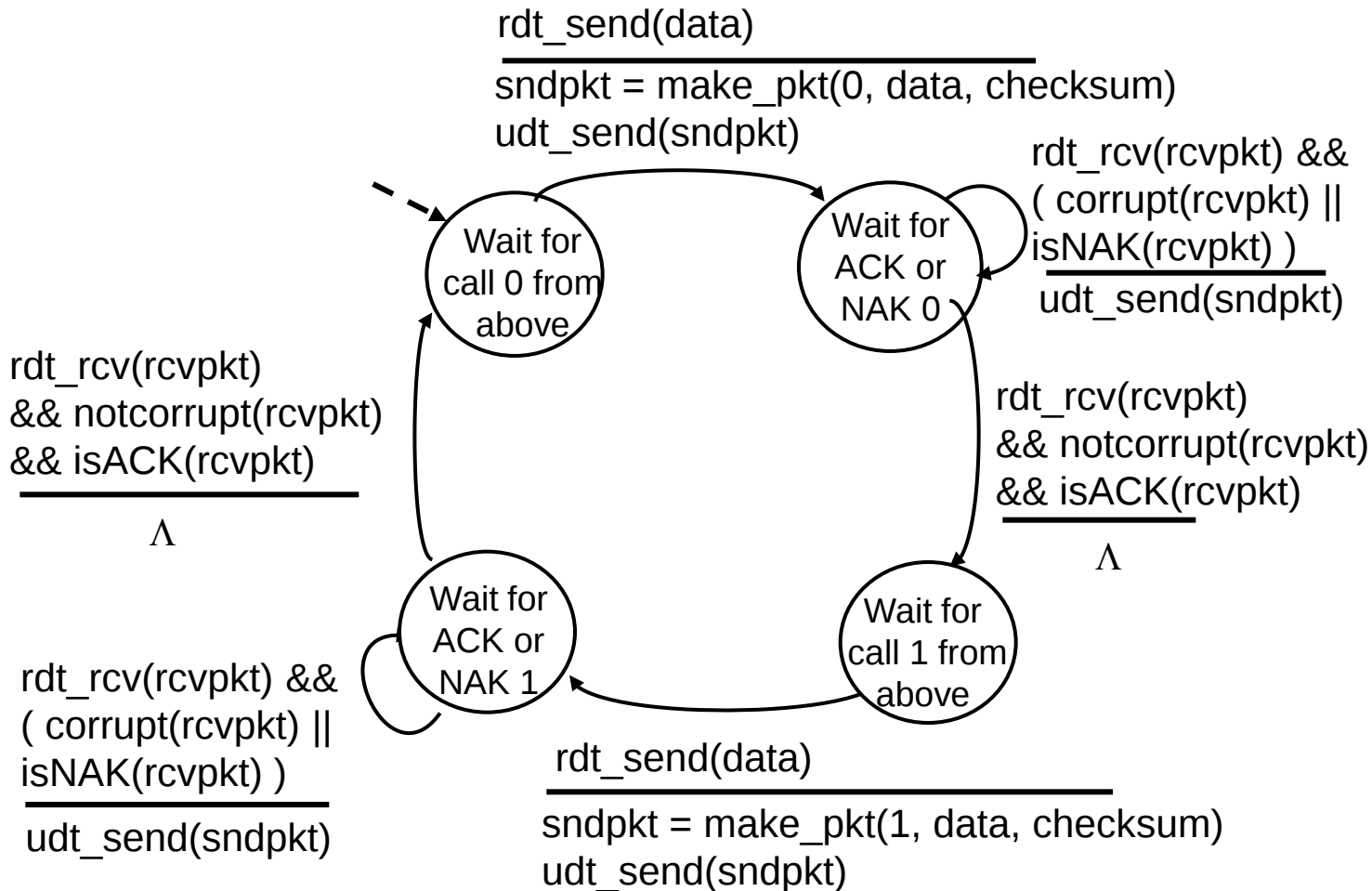
Approach 1: resend the current data packet?



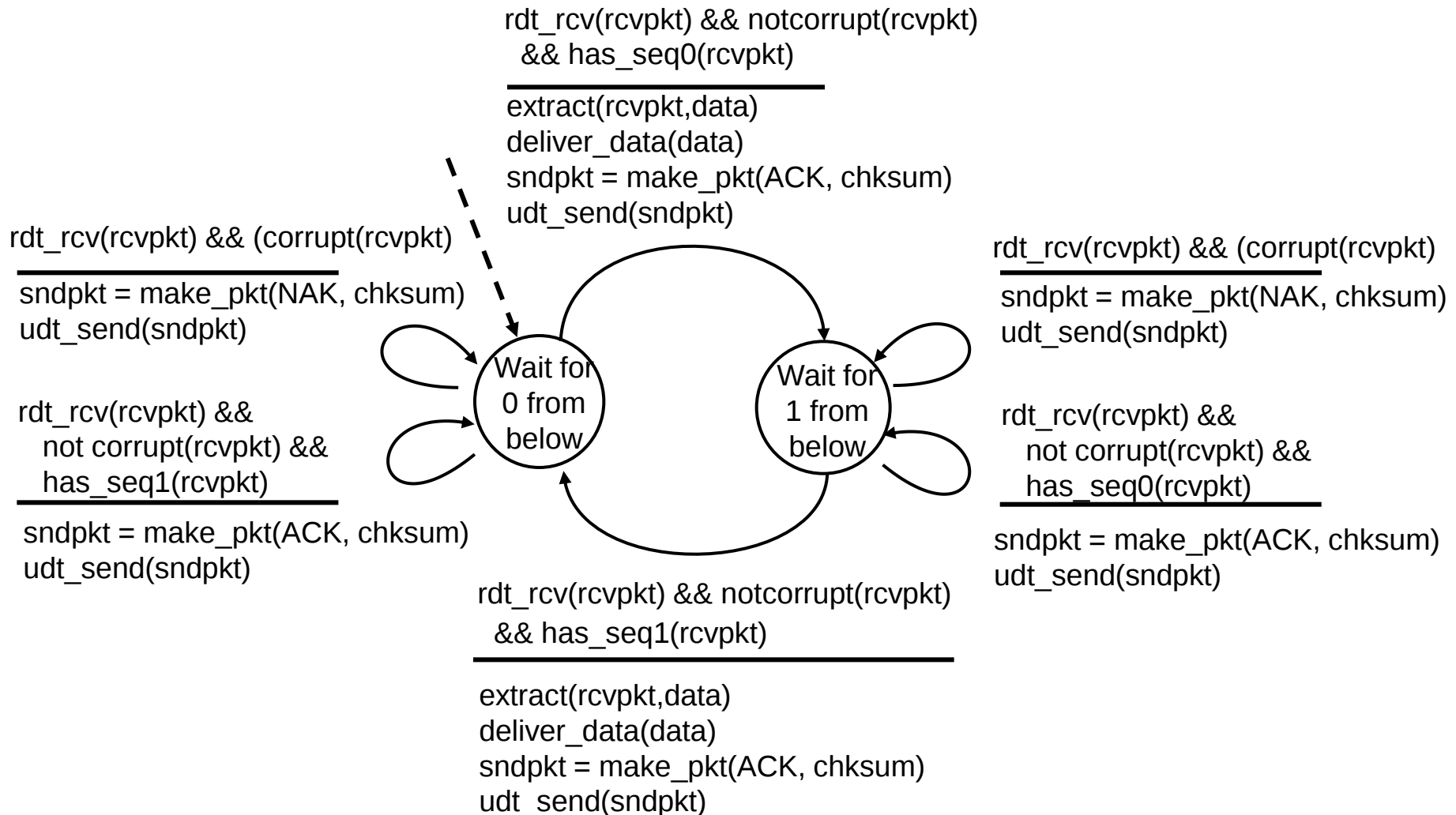
Handling Duplicate Packets

- ❑ sender adds *sequence number* to each packet
- ❑ sender retransmits current packet if ACK/NAK garbled
- ❑ receiver discards (doesn't deliver up) duplicate packet

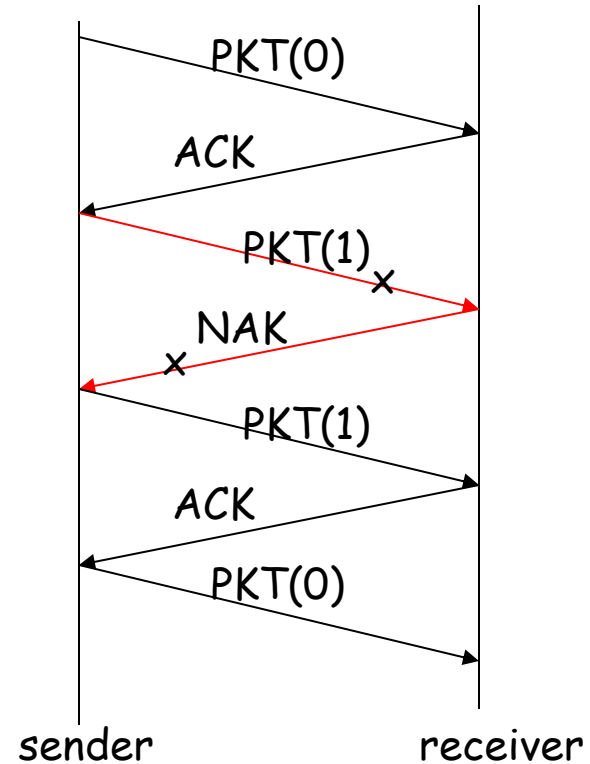
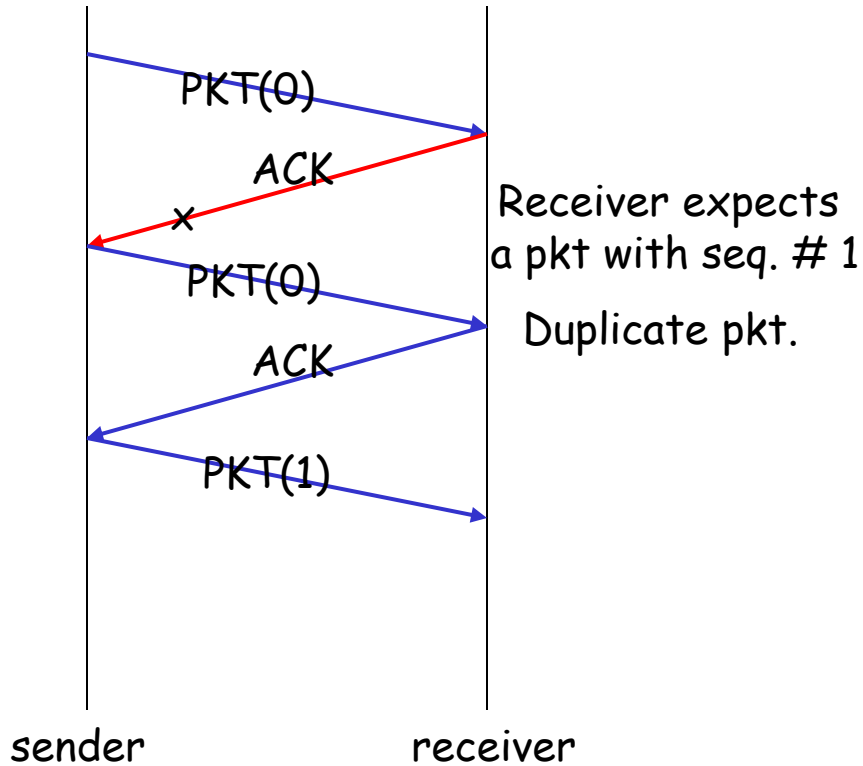
rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rtd2.1: examples



rdt2.1: summary

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. #

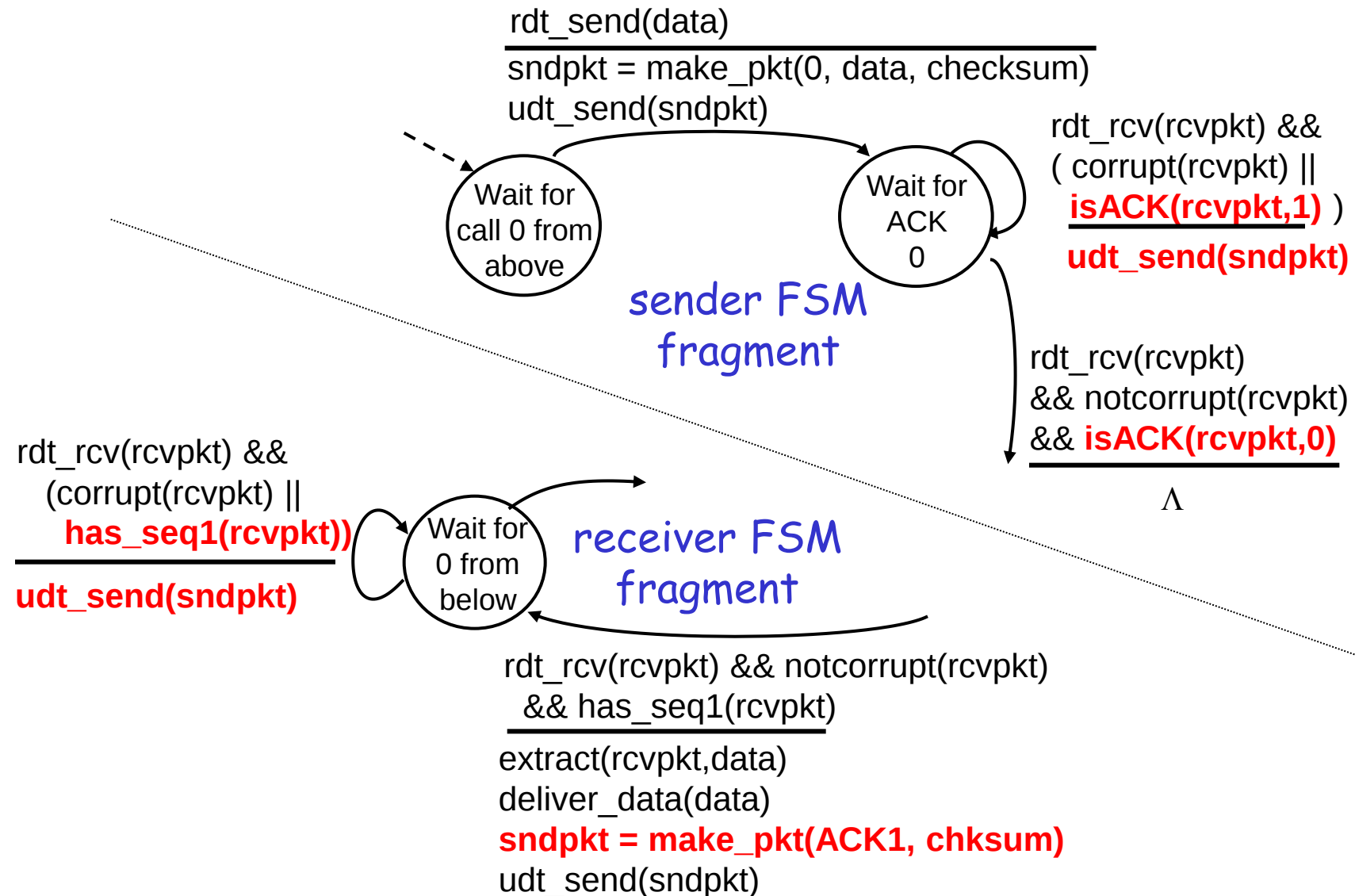
Receiver:

- ❑ 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



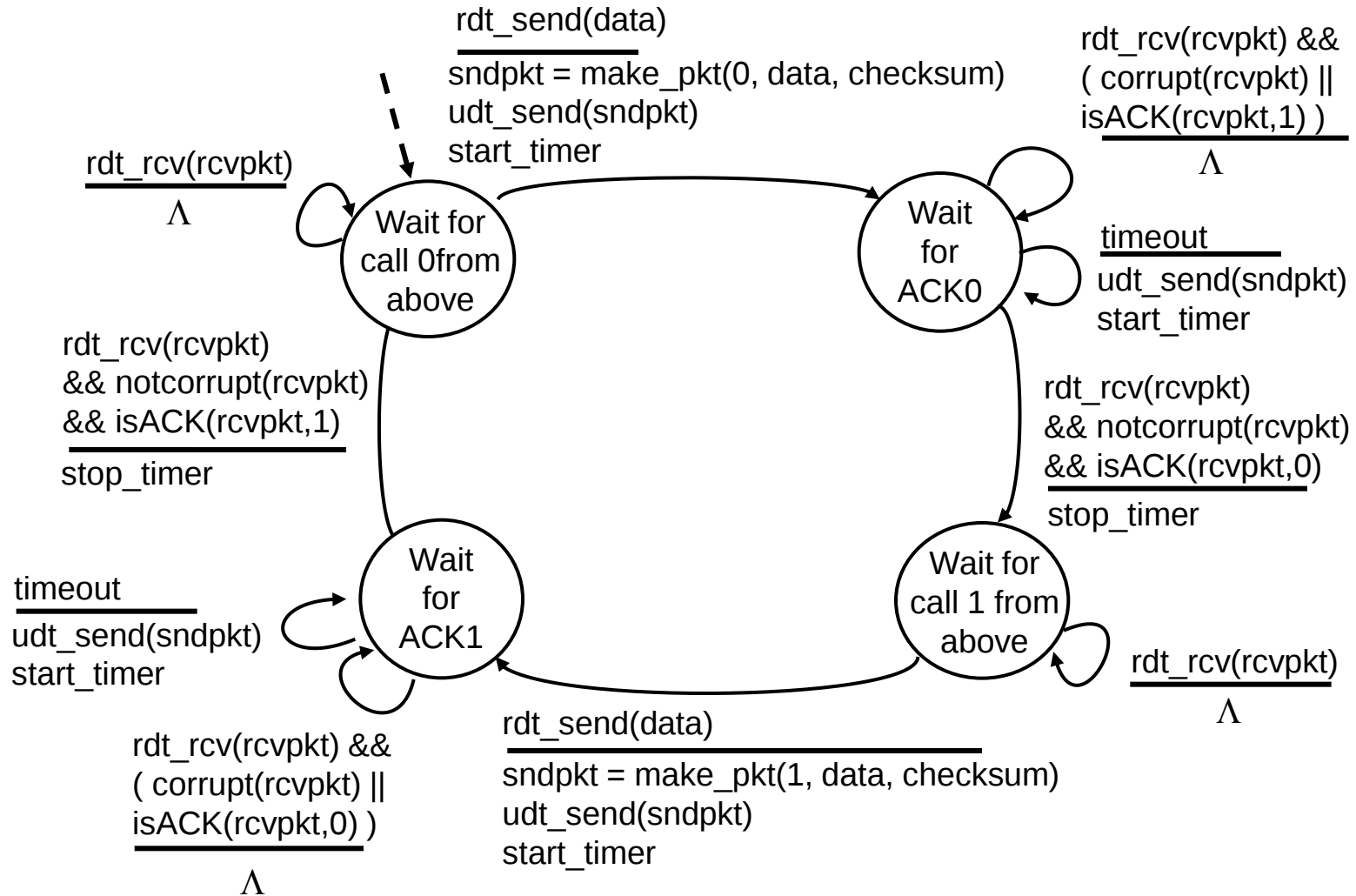
rdt3.0: The case of "Lossy" Channels

- Assumption: underlying channel can also lose packets (data or ACKs)

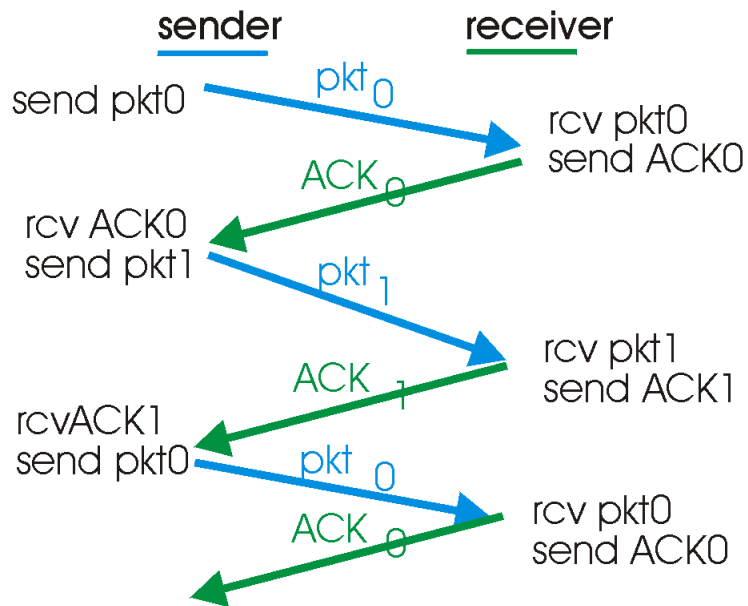
rdt3.0: The case of "Lossy" Channels

- ❑ Assumption: underlying channel can also lose packets (data or ACKs)
- ❑ Approach: sender waits "reasonable" amount of time for ACK (a Time-Out)
 - Time-out value?
 - Possibility of duplicate packets/ACKs?
- ❑ 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

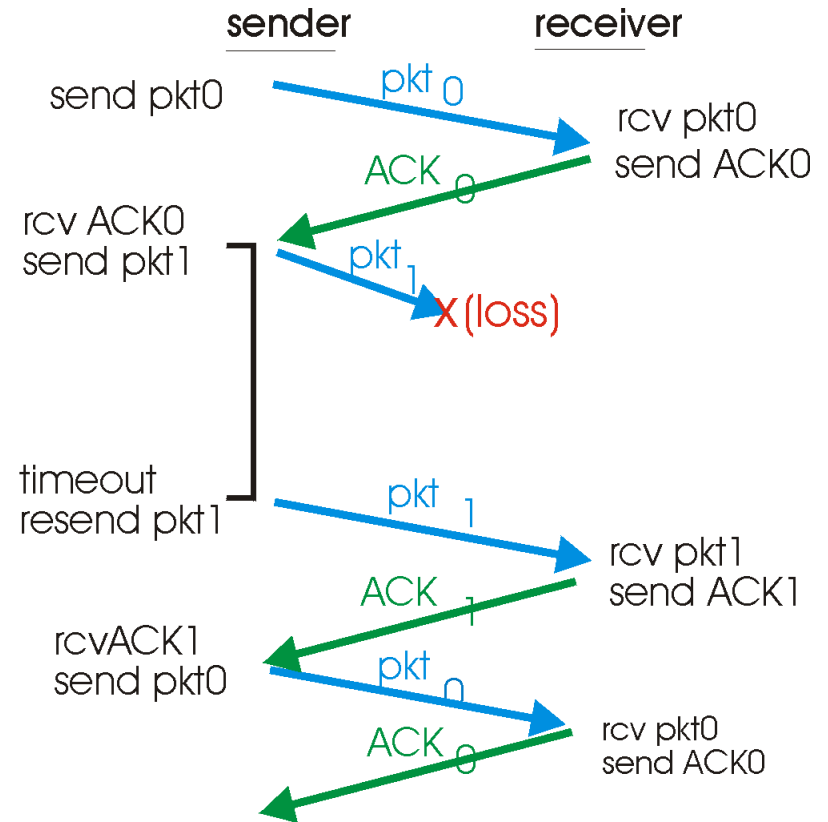
rdt3.0 sender



rdt3.0 in action

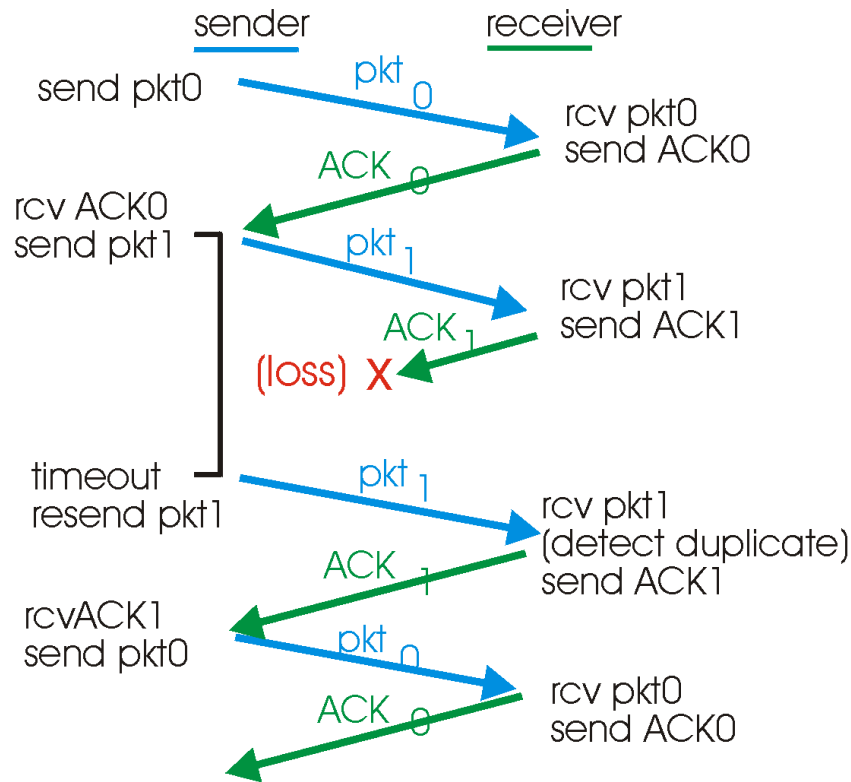


(a) operation with no loss

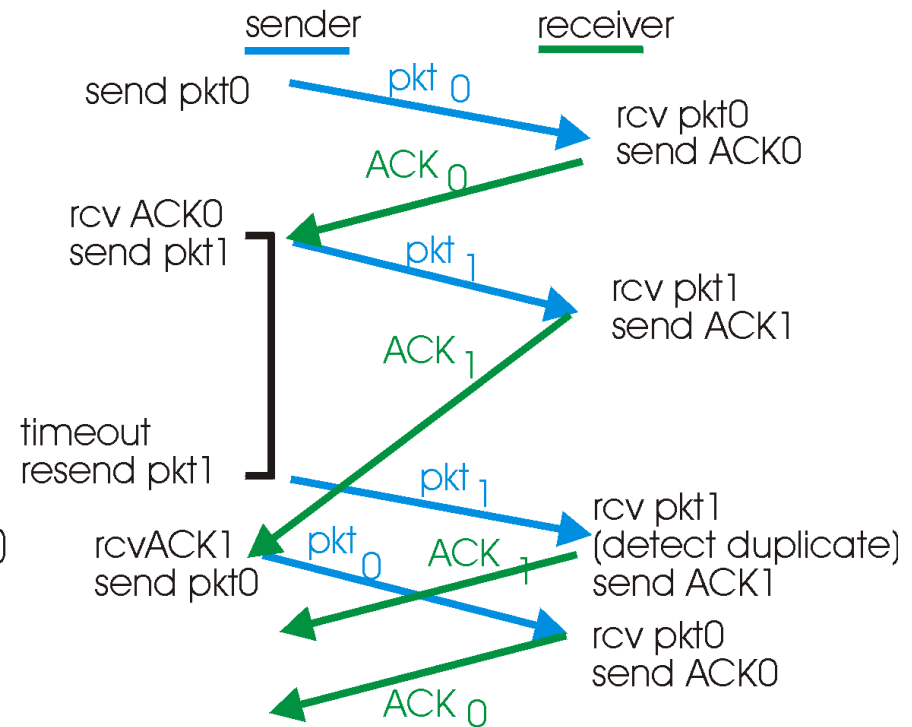


(b) lost packet

rdt3.0 in action

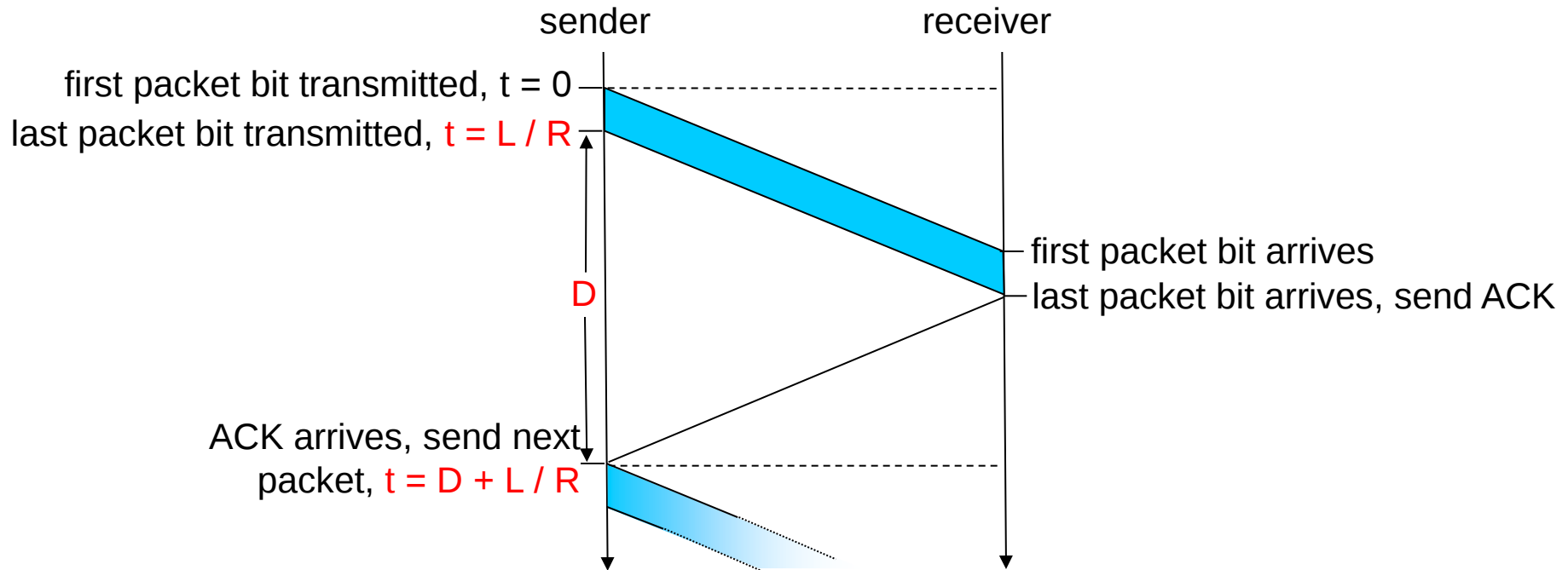


(c) lost ACK



(d) premature timeout

stop-and-wait operation



Pipelining: Motivation

- Stop-and-wait allows the sender to only have a single unACKed packet at any time
- example: 1 Mbps link (R), end-2-end round trip propagation delay (D) of 92ms, 1KB packet (L):

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb}}{10^3 \text{ kb/sec}} = 8 \text{ ms}$$

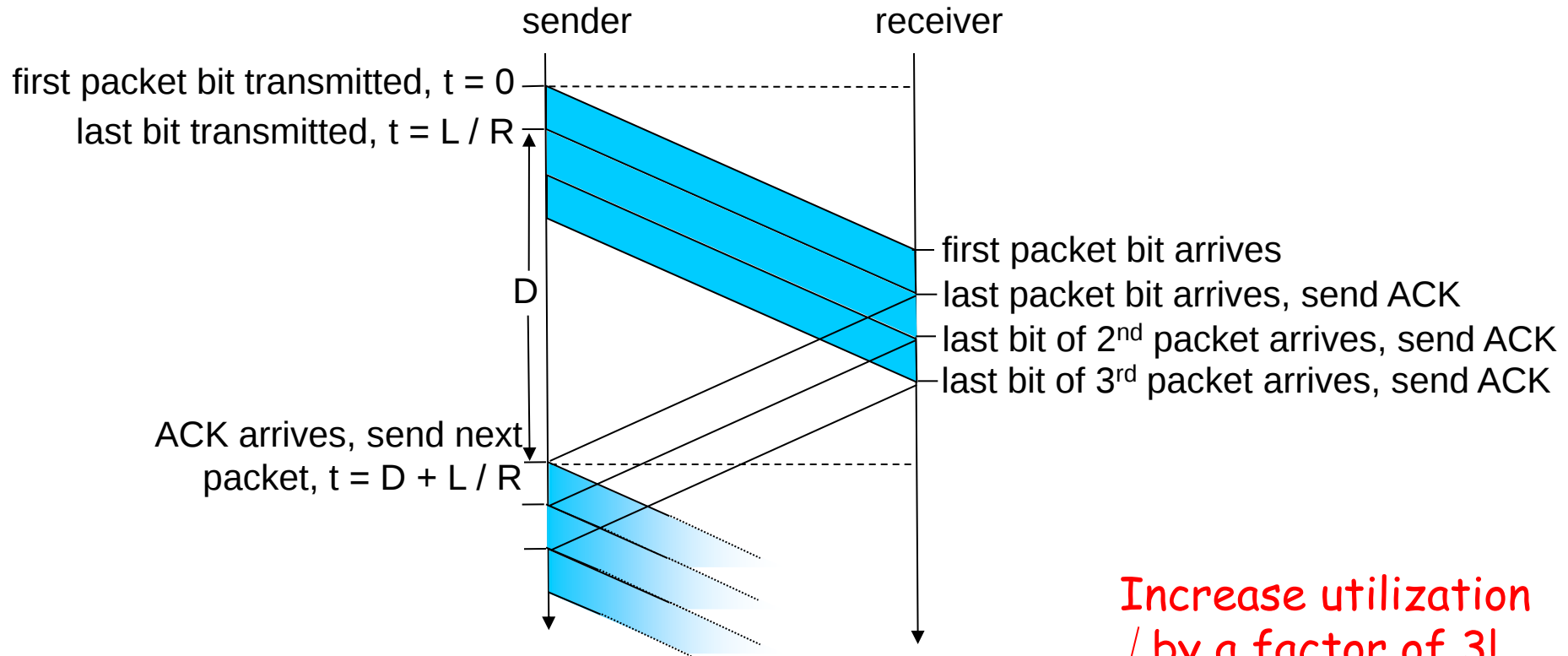
$$U_{\text{sender}} = \frac{L / R}{D + L / R} = \frac{8\text{ms}}{100\text{ms}} = 0.08$$

- 1KB pkt every 100 ms -> 80Kbps throughput on a 1 Mbps link
- What does bandwidth x delay product tell us?

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

Pipelining: increased utilization



$$U_{\text{sender}} = \frac{3 * L / R}{D + L / R} = \frac{24\text{ms}}{100\text{ms}} = 0.24$$

Increase utilization
by a factor of 3!

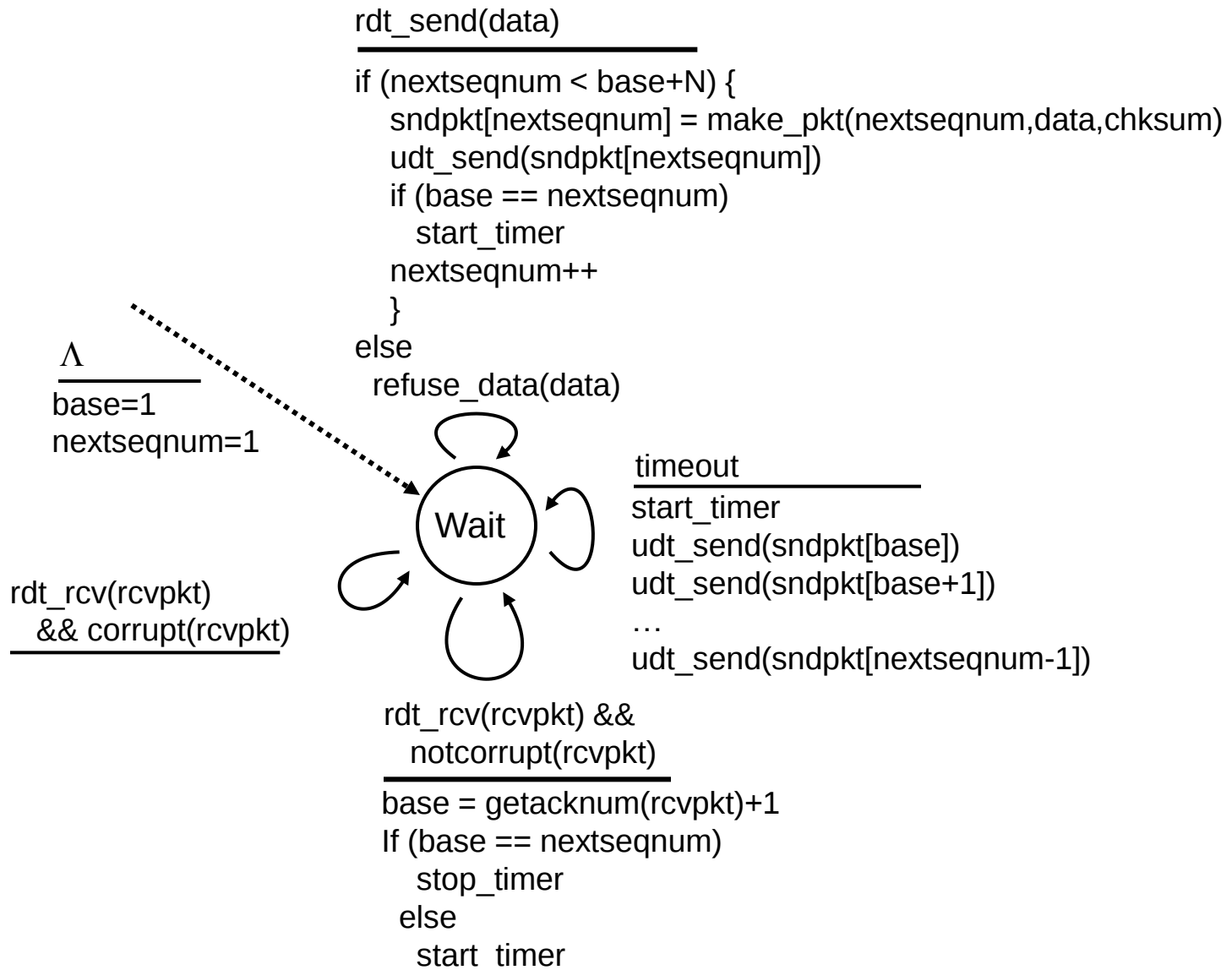
Pipelined protocols

- ❑ Two generic forms of pipelined protocols
 - *go-Back-N*
 - *selective repeat*

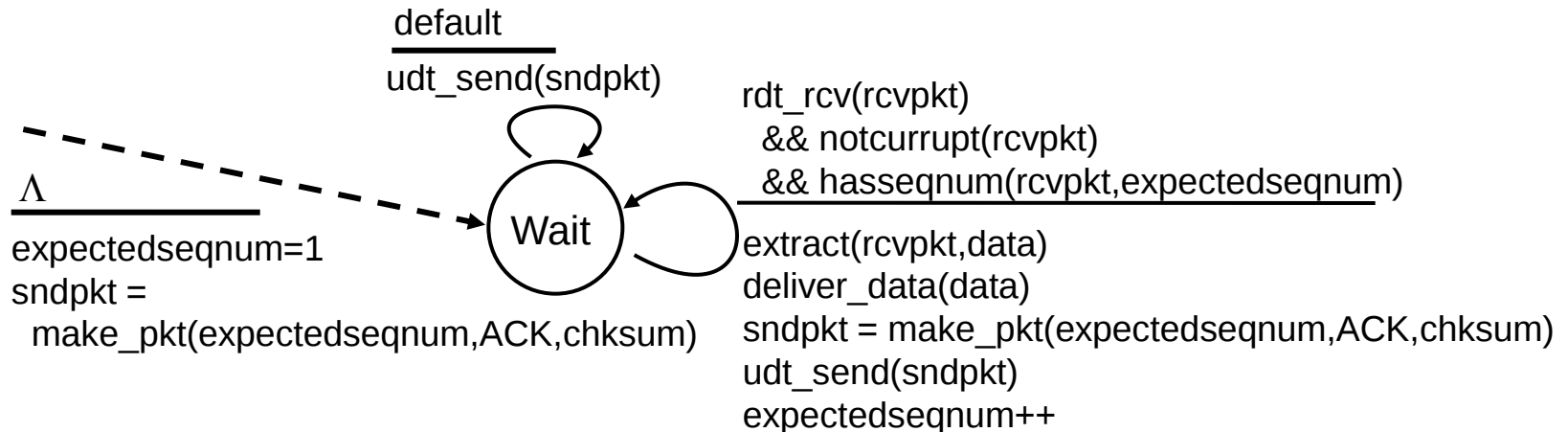
Go-Back-N

- ❑ Allow up to N unACKed pkts in the network
 - N is the Window size
- ❑ Sender Operation:
 - If window not full, transmit
 - ACKs are cumulative
 - On timeout, send all packets previously sent but not yet ACKed.
 - Uses a single timer - represents the oldest transmitted, but not yet ACKed pkt

GBN: sender 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: ...

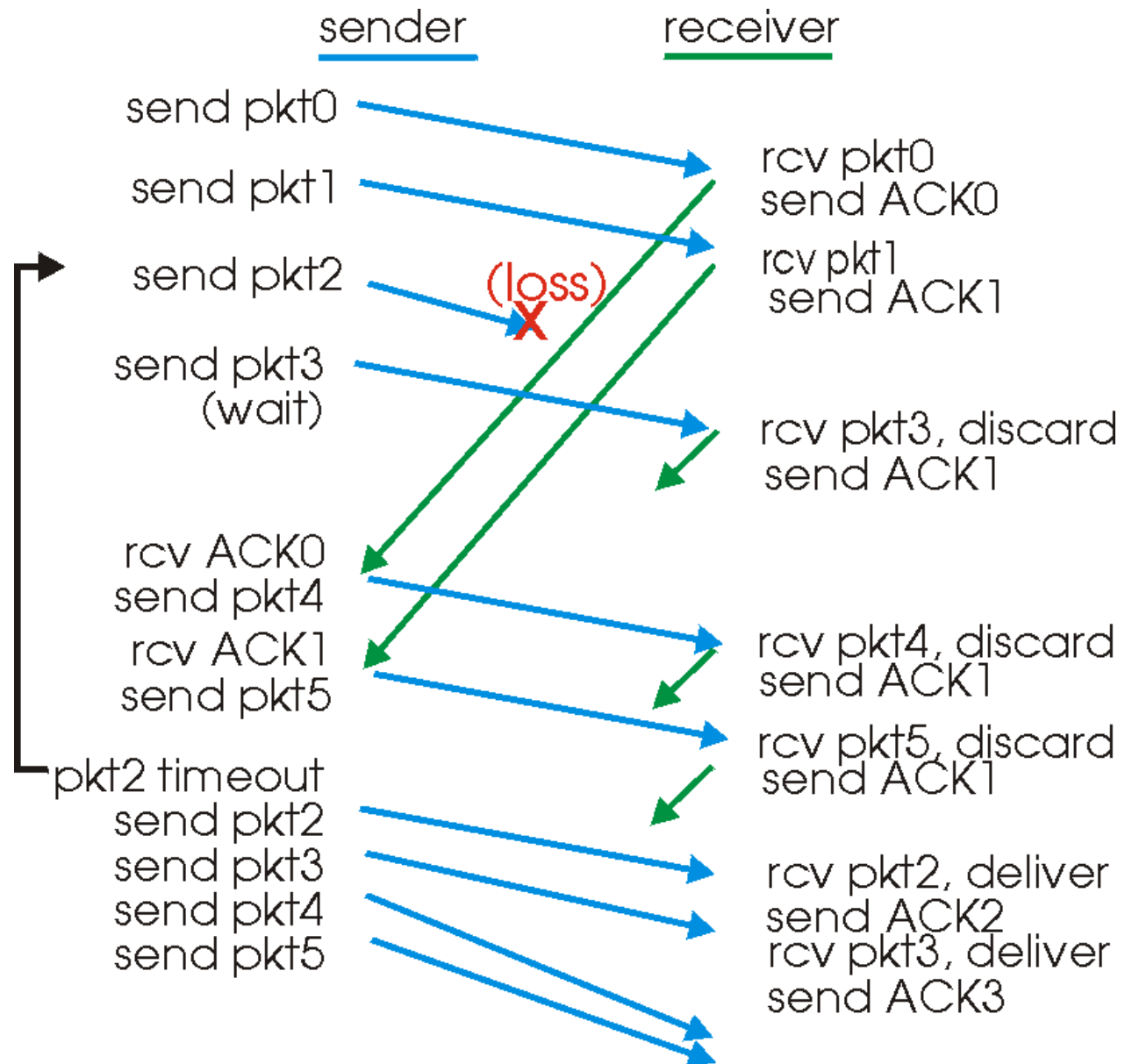
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember `expectedseqnum`

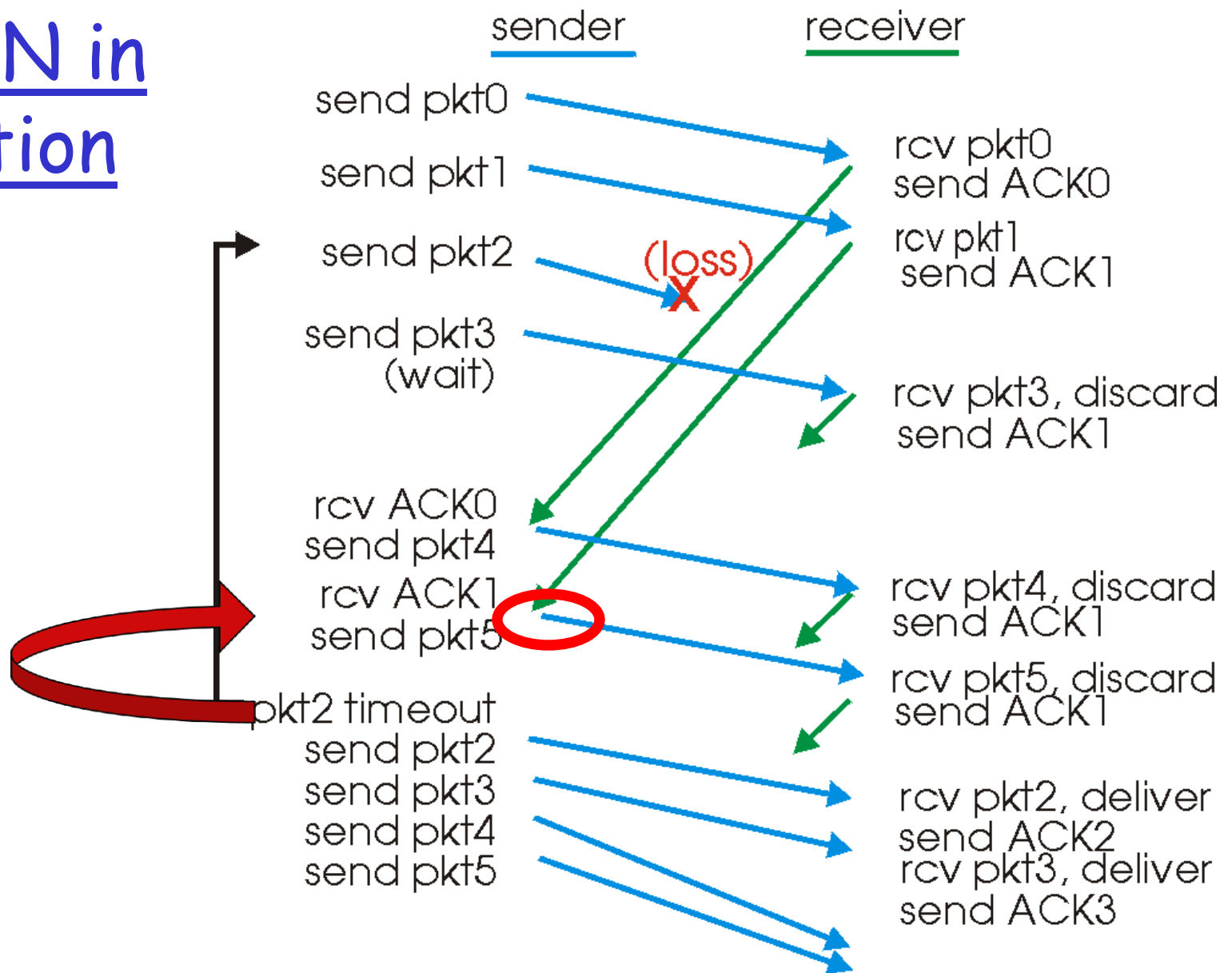
□ out-of-order pkt:

- discard (don't buffer) -> **no receiver buffering!**
- Re-ACK pkt with highest in-order seq #

GBN in action



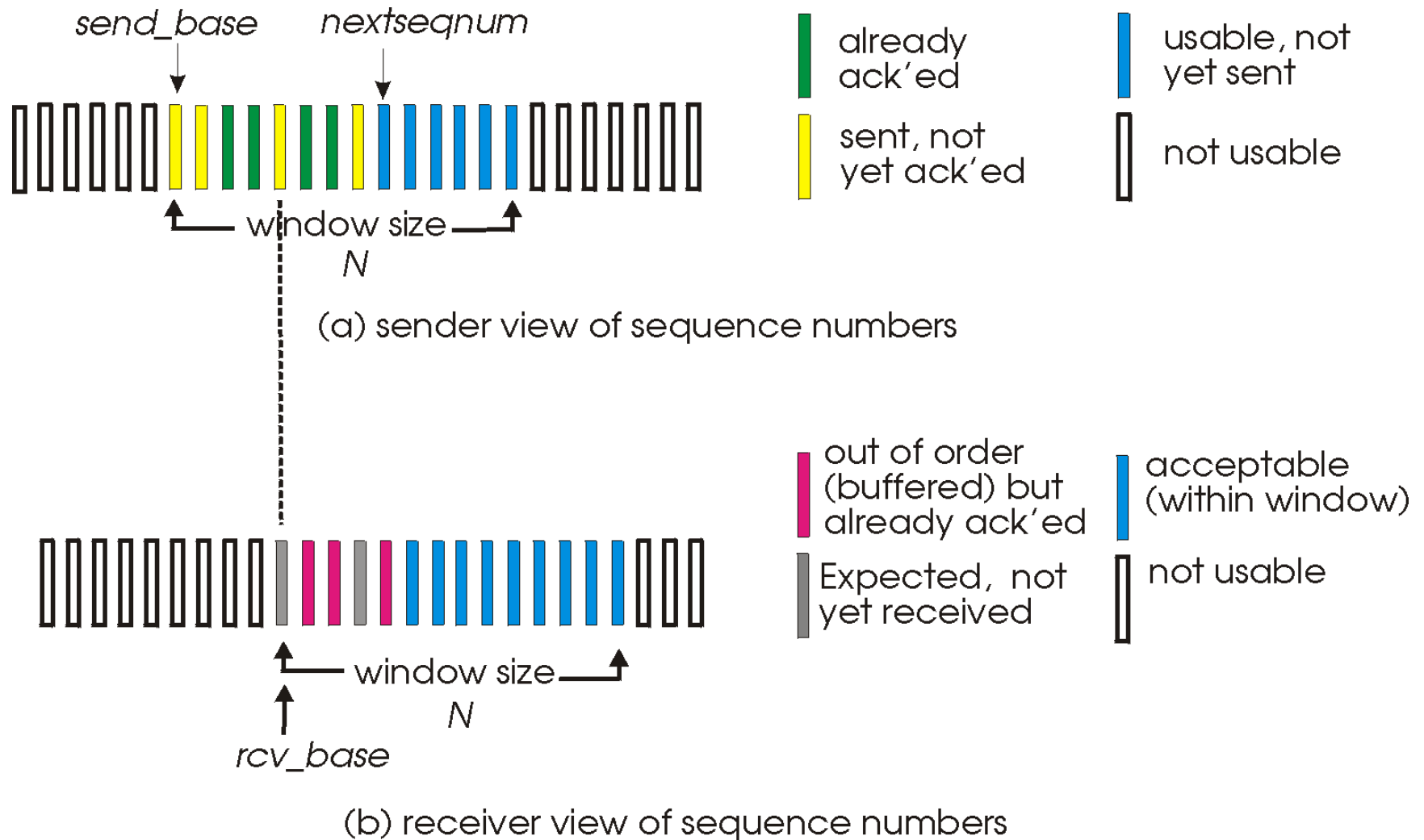
GBN in action



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
 - again 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 #

—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

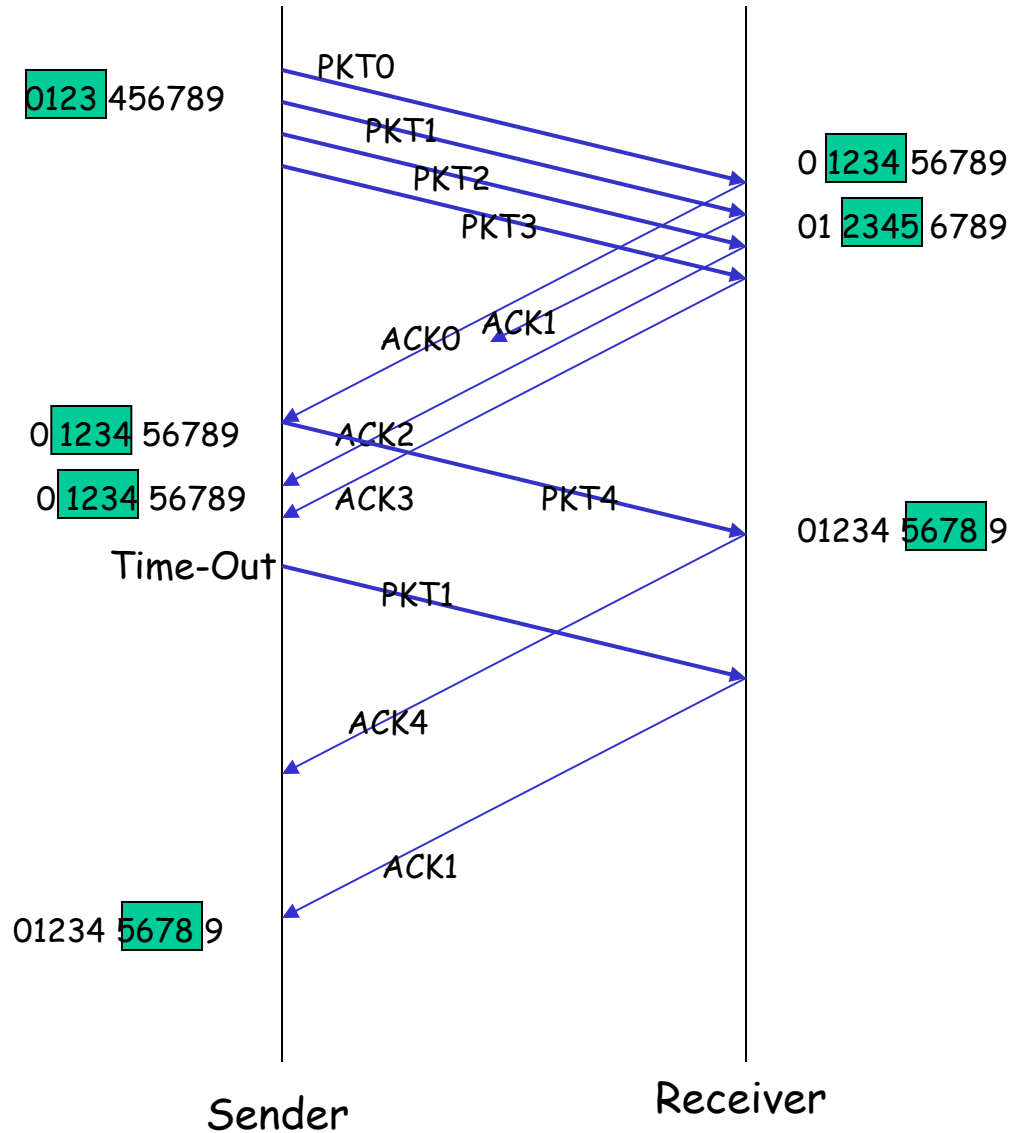
pkt n in [rcvbase-N, rcvbase-1]

- ❑ ACK(n)

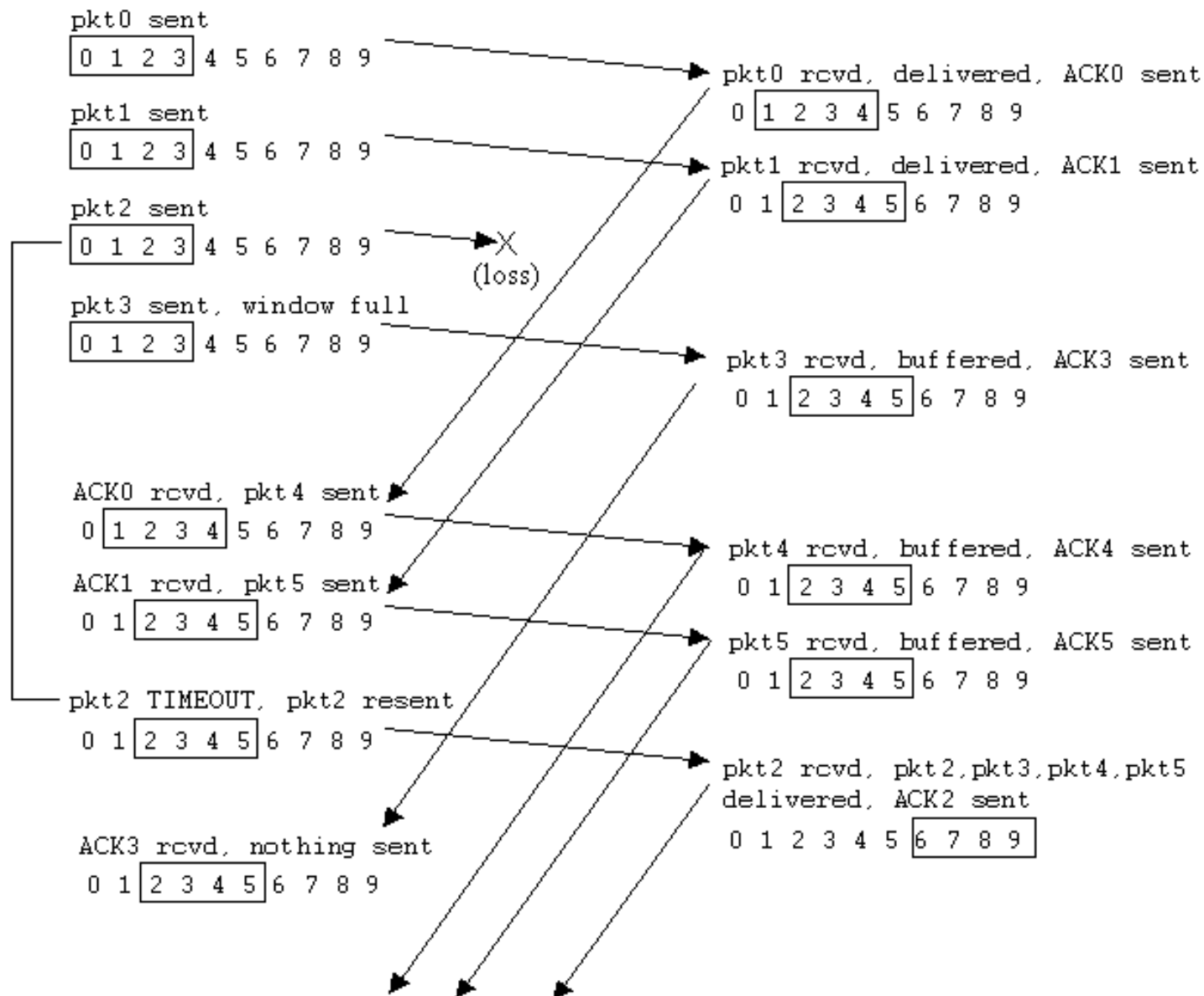
otherwise:

- ❑ ignore

Selective Repeat Example



Another Example



Selective repeat: dilemma

Example:

- ❑ seq #'s: 0, 1, 2, 3
- ❑ window size=3

- ❑ receiver sees no difference in two scenarios!
- ❑ incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

