Lecture 4

Transport Layer, Part I

Chapter 3: Transport Layer

Our goals:

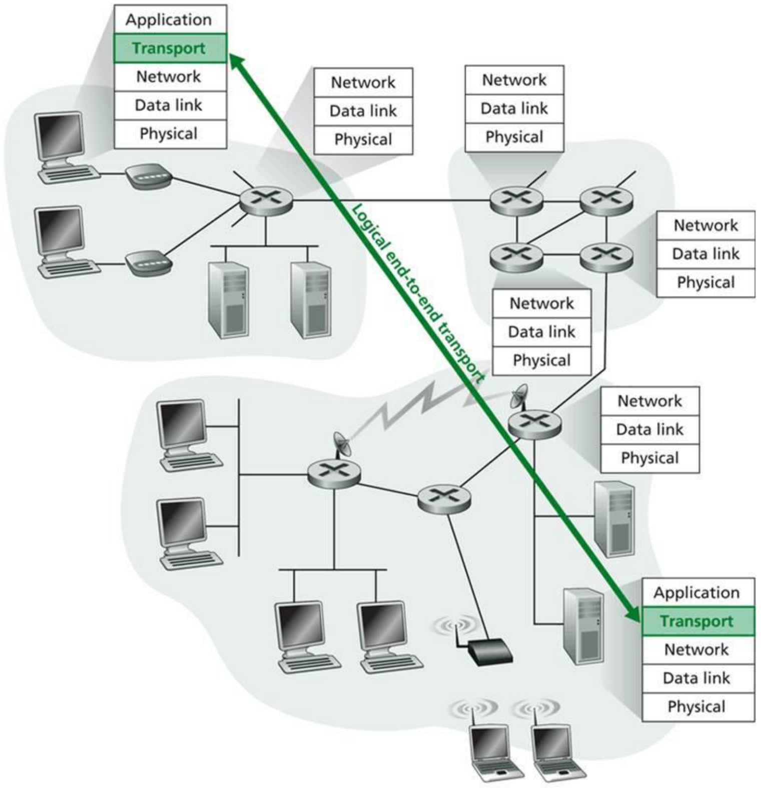
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion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

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 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

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

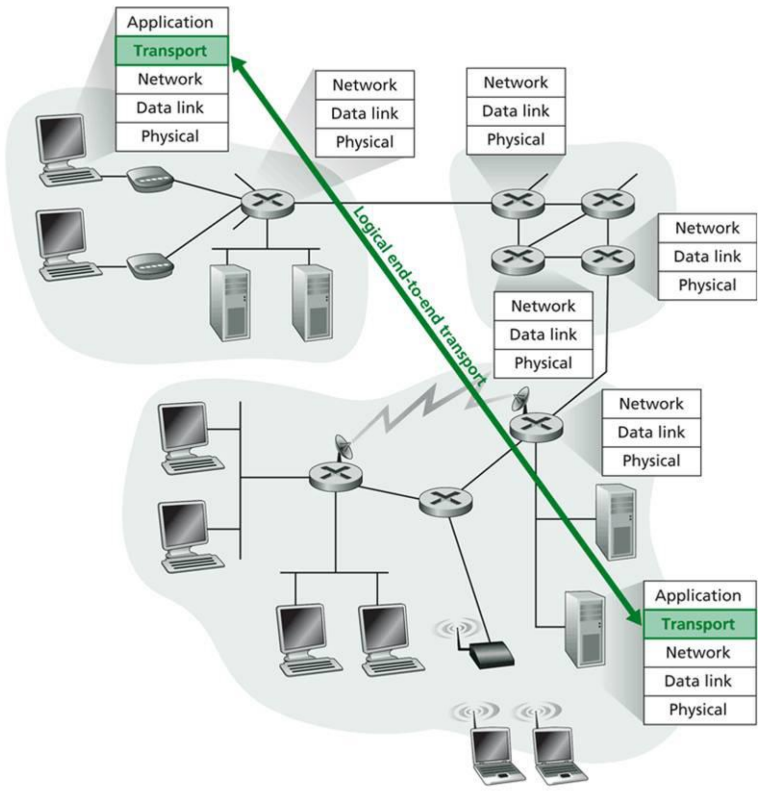
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

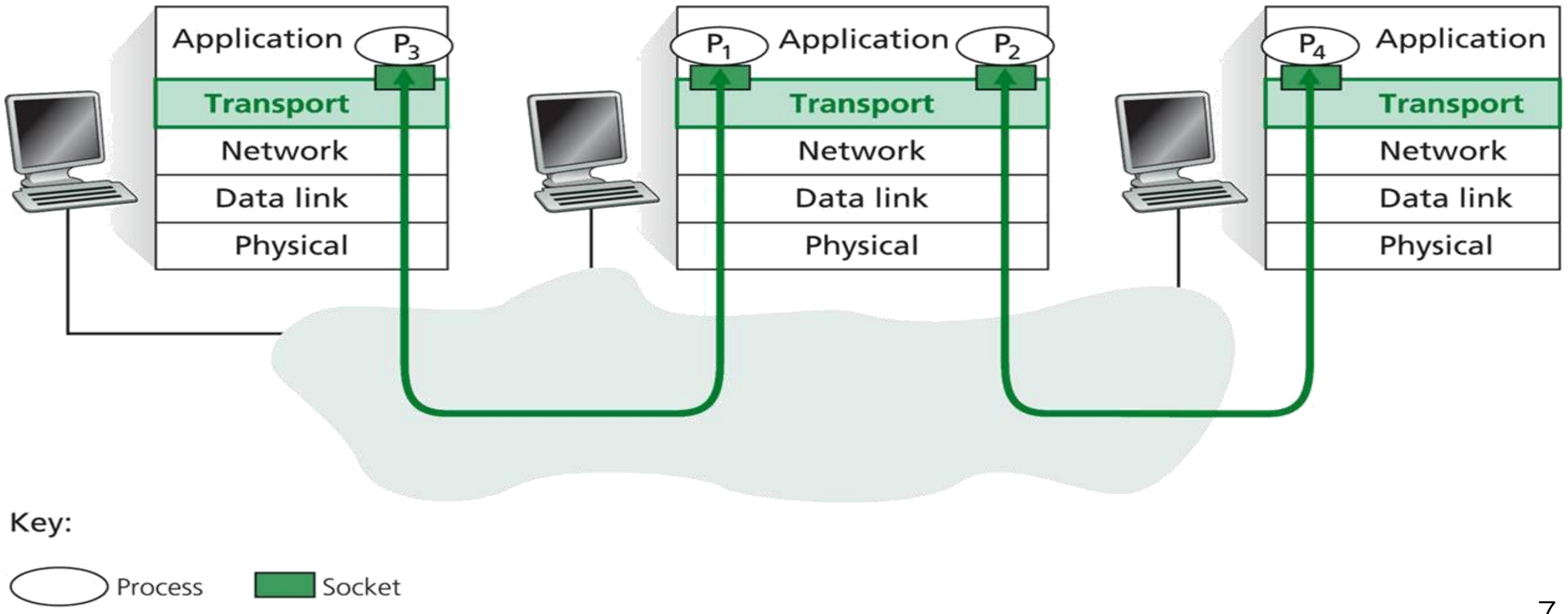
Internet transport-layer protocols

- 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



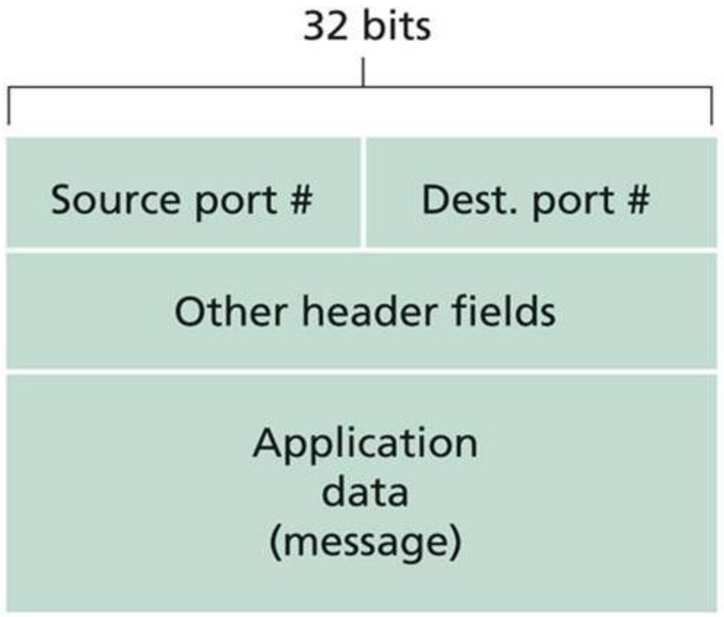
Multiplexing/demultiplexing

- Demultiplexing at rcv host
  - delivering received segments to correct socket
- Multiplexing at send host:
  - gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

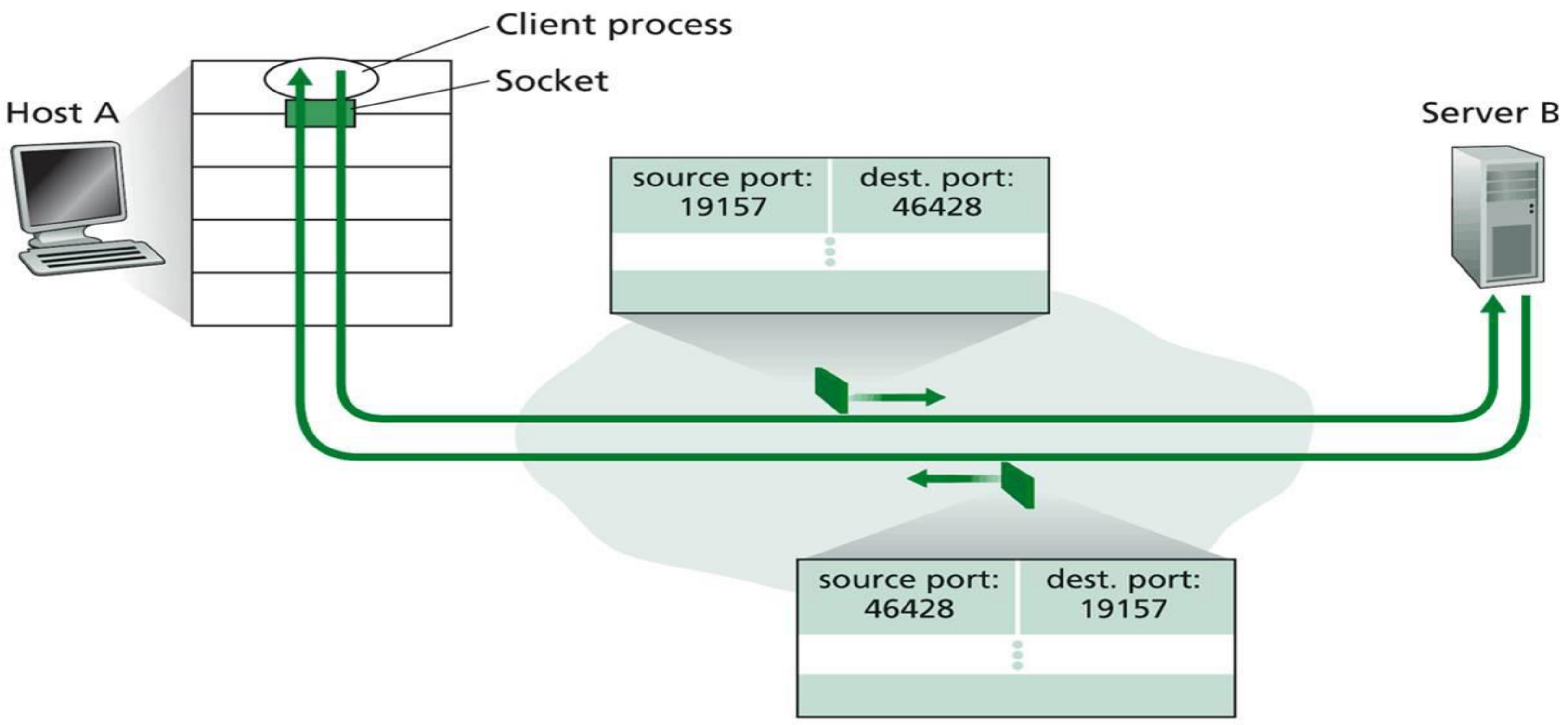
- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(19157);
DatagramSocket mySocket2 = new DatagramSocket(99222);
```

- 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

Connectionless demux (cont.)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

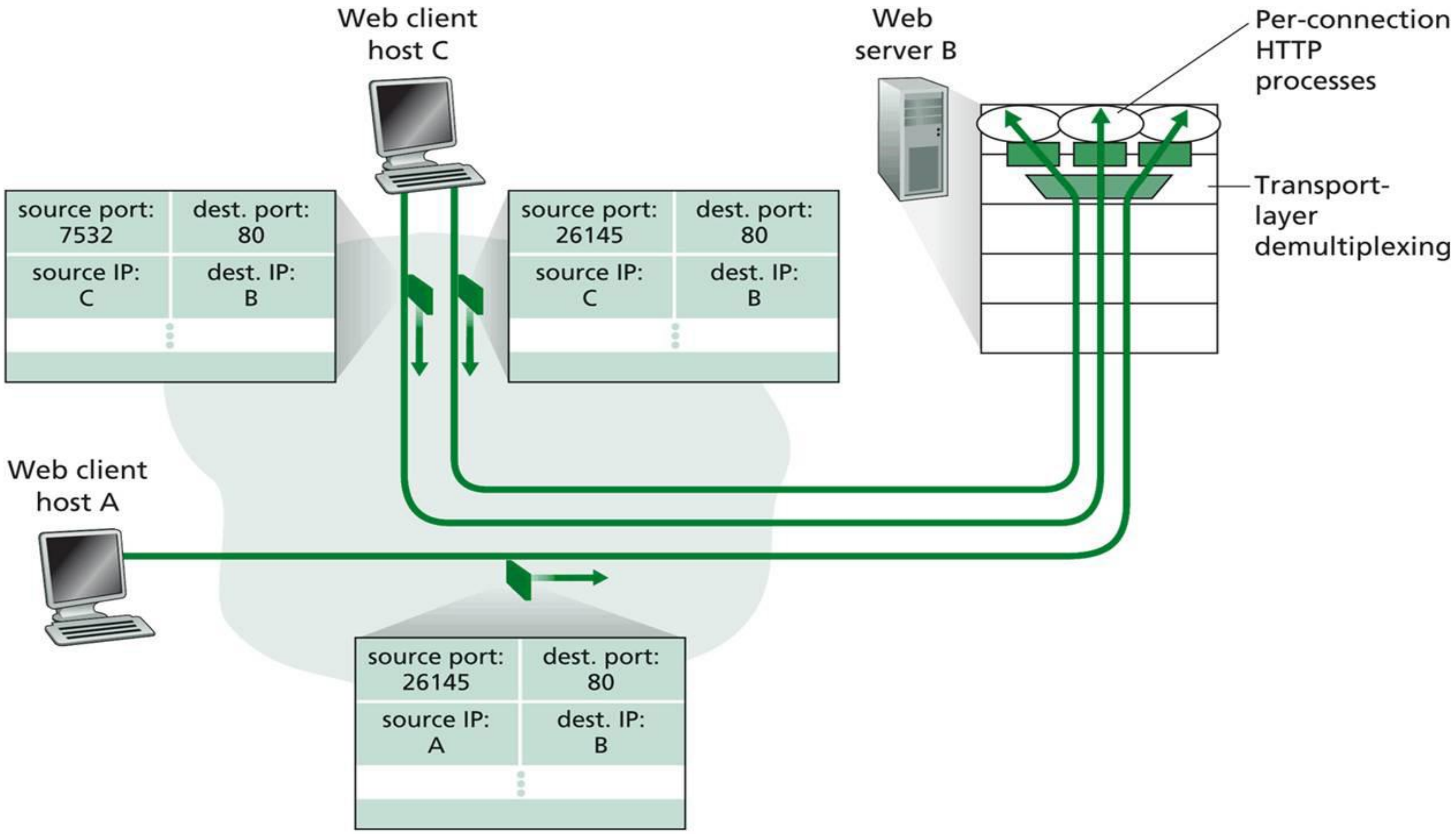


SP provides "return address"

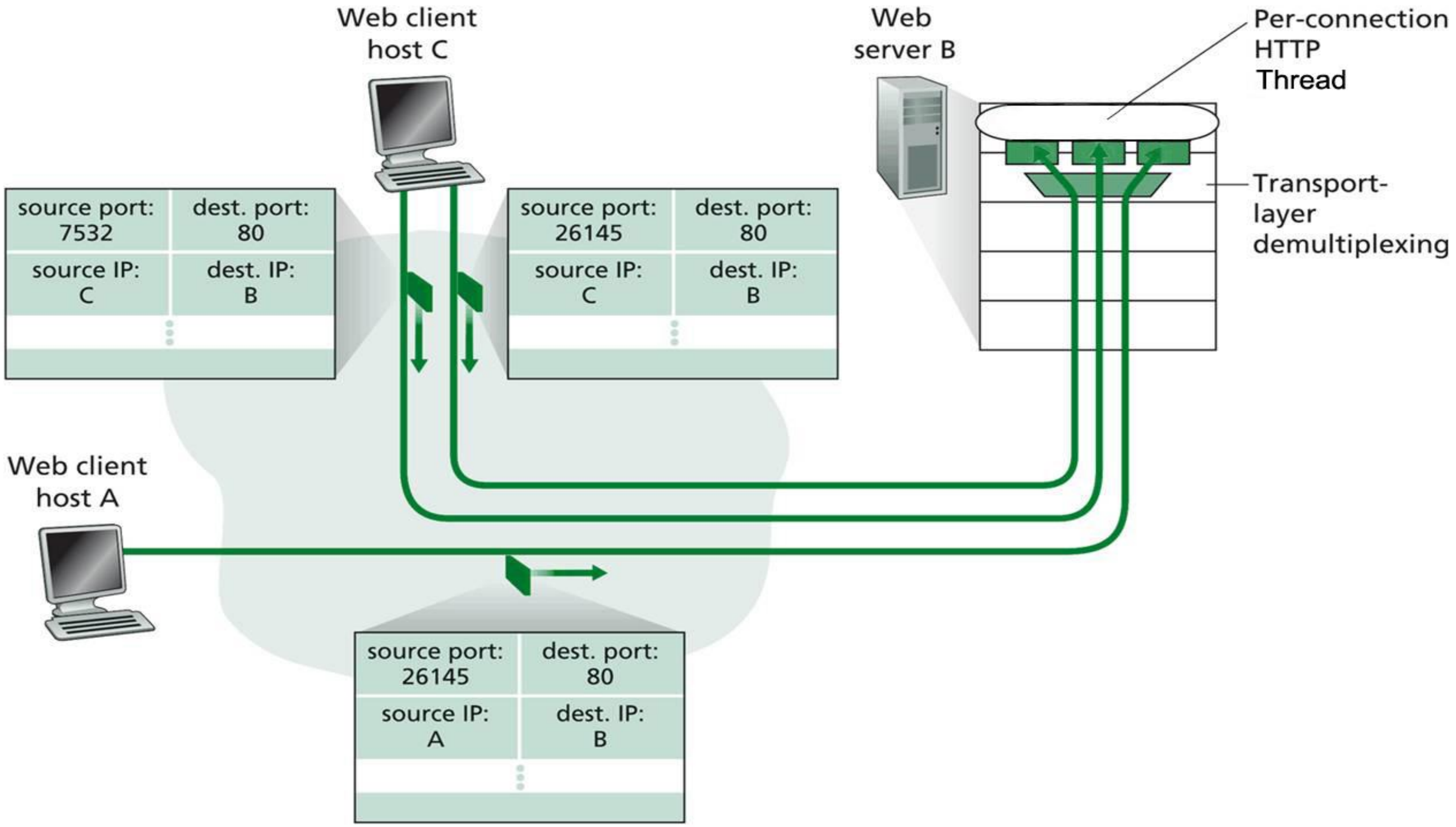
Connection-oriented demux

- 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

Connection-oriented demux (cont.)



Connection-oriented demux: Threaded Web Server



UDP: User Datagram Protocol [RFC 768]

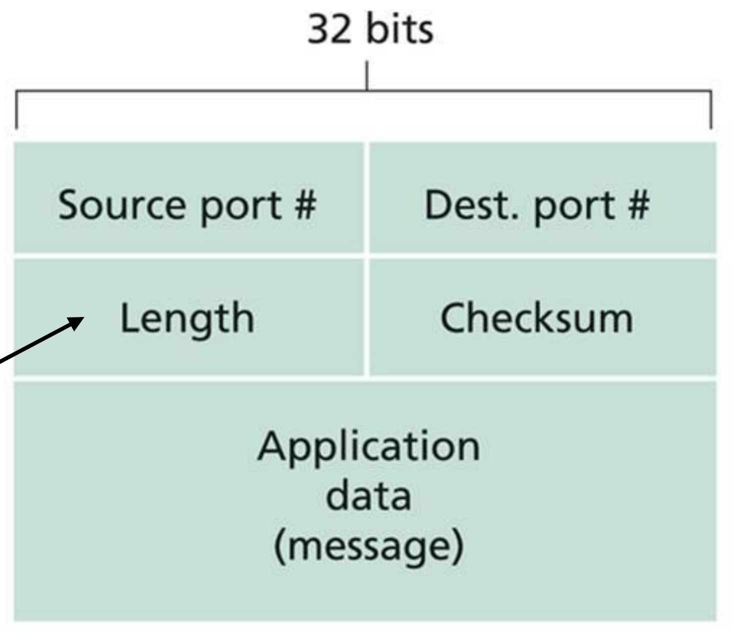
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability a
  - application-specific error recovery!

Length, in bytes of UDP segment, including header



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:
  - 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
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

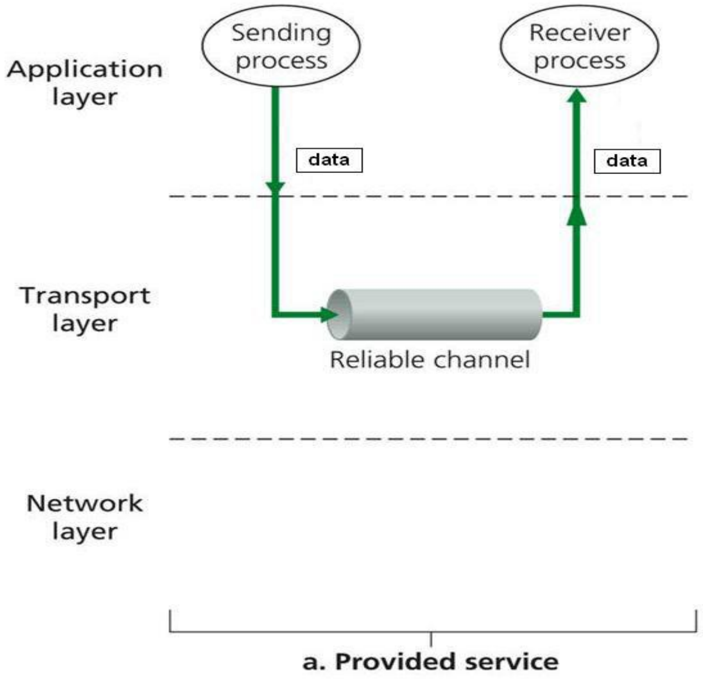
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Sender transmits both of 16-bit numbers and checksum!

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Principles of Reliable data transfer

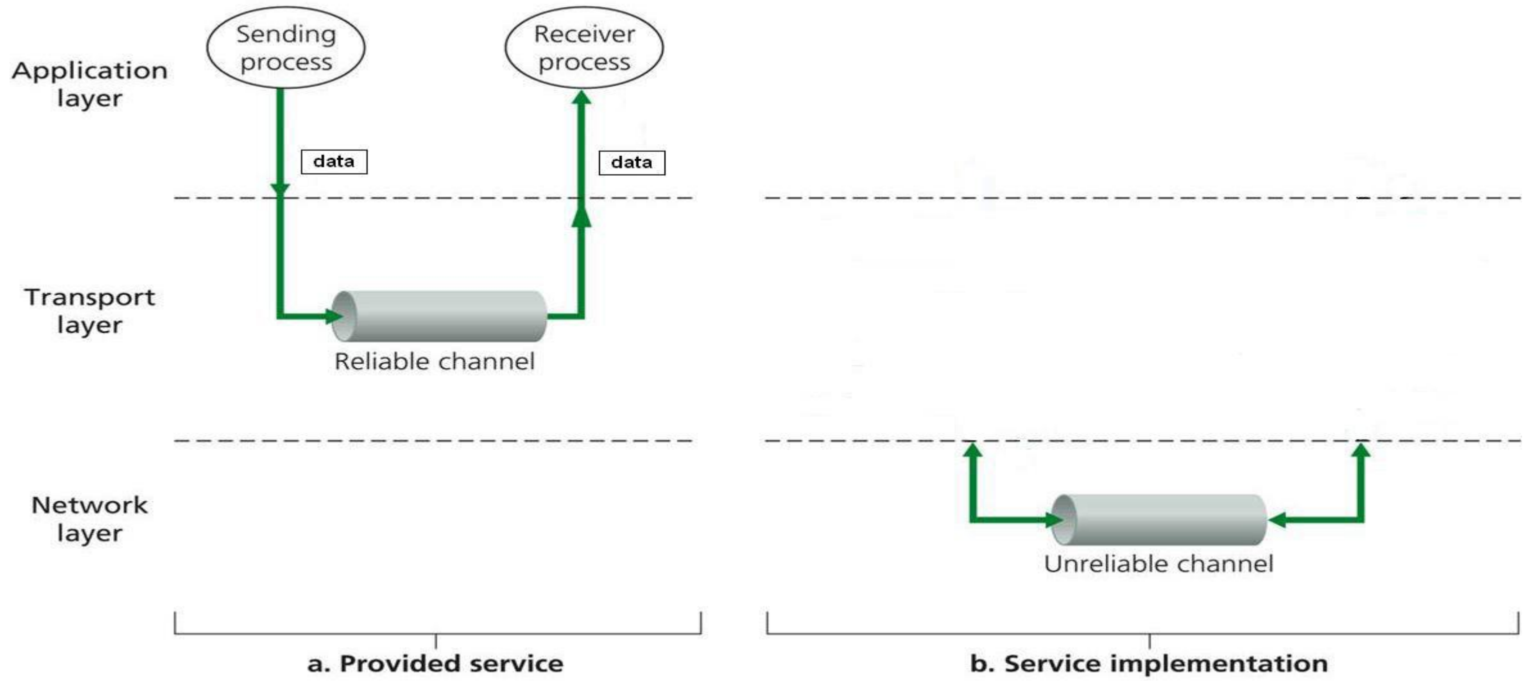
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

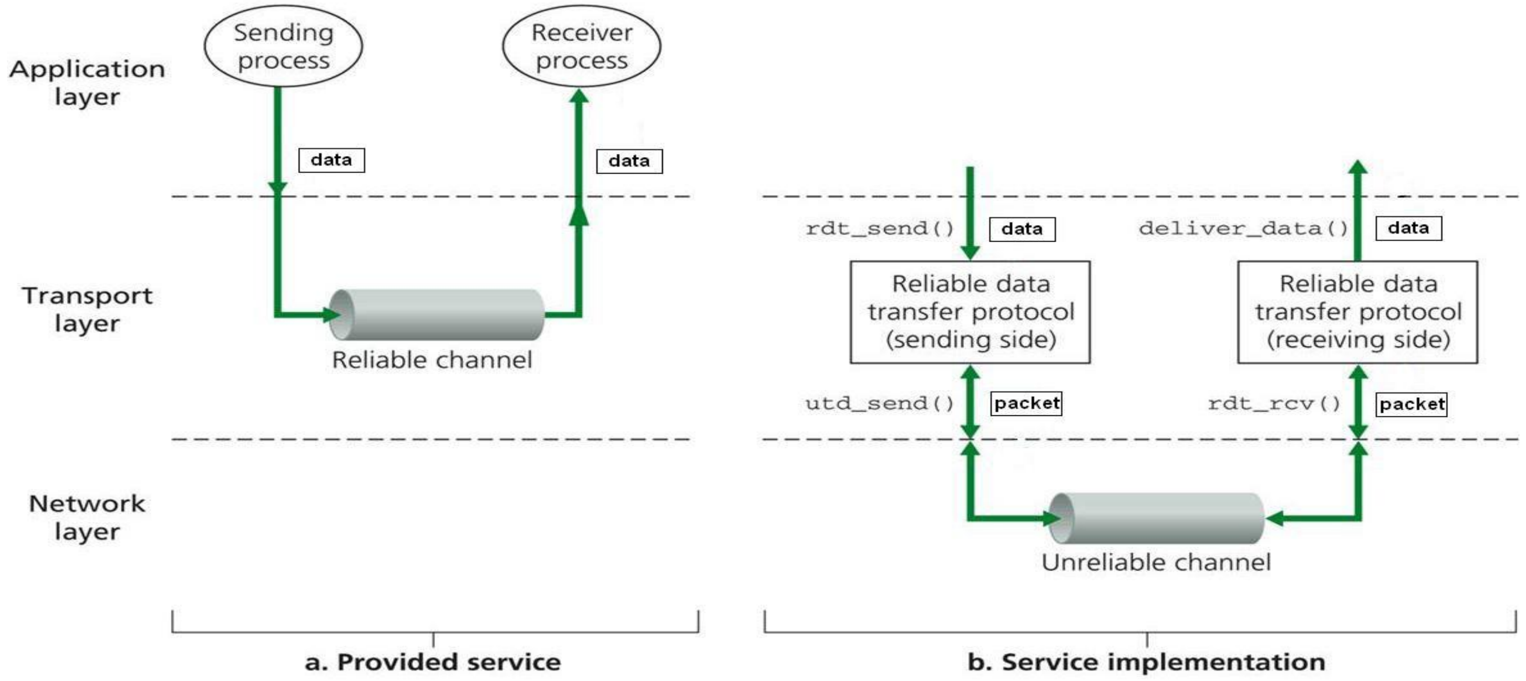
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

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



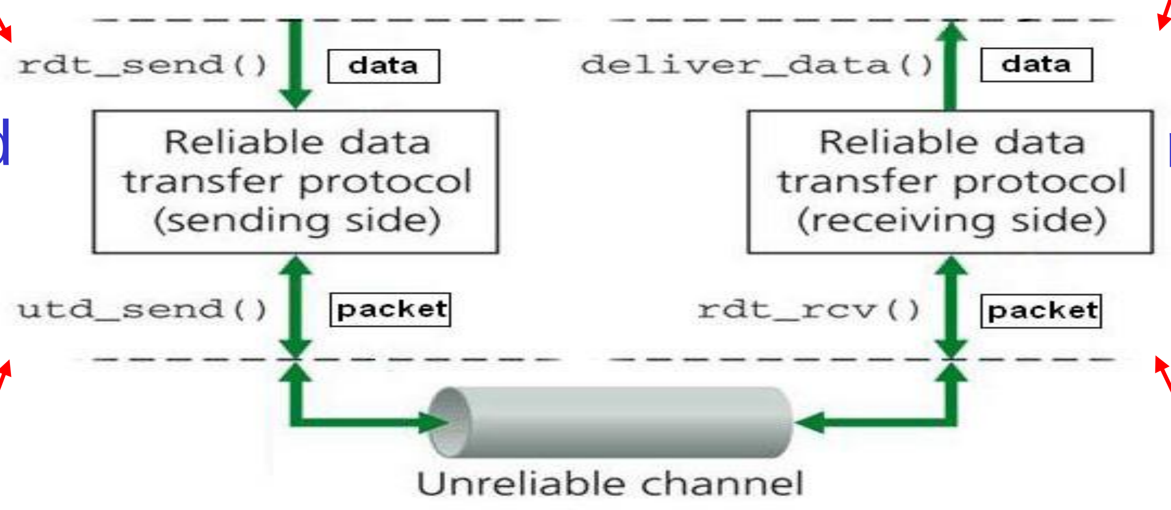
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

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

send side


rdt_send()
data
deliver_data()
data
Reliable data transfer protocol (sending side)
Reliable data transfer protocol (receiving side)
utd_send()
packet
rdt_rcv()
packet
Unreliable channel

receive side

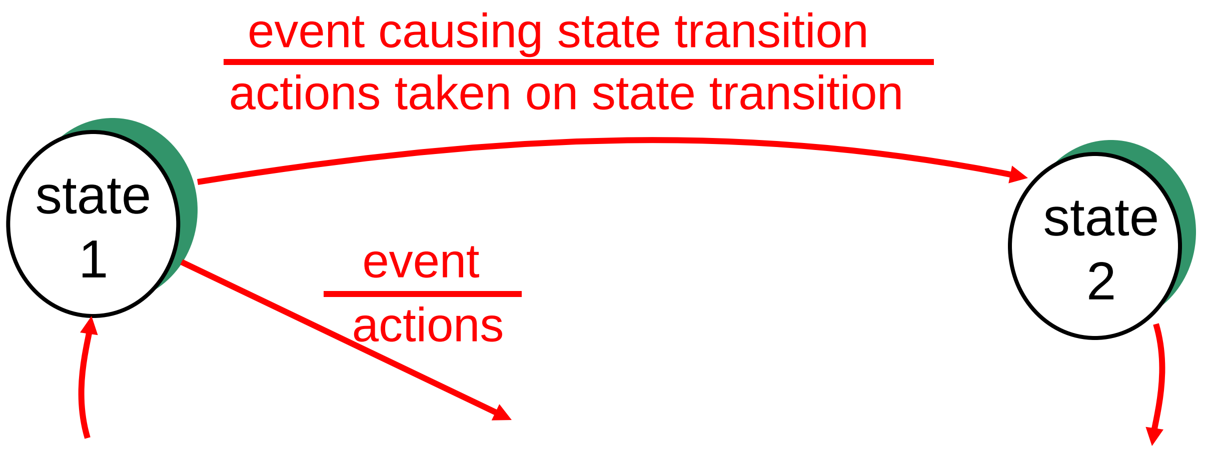
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

We'll:

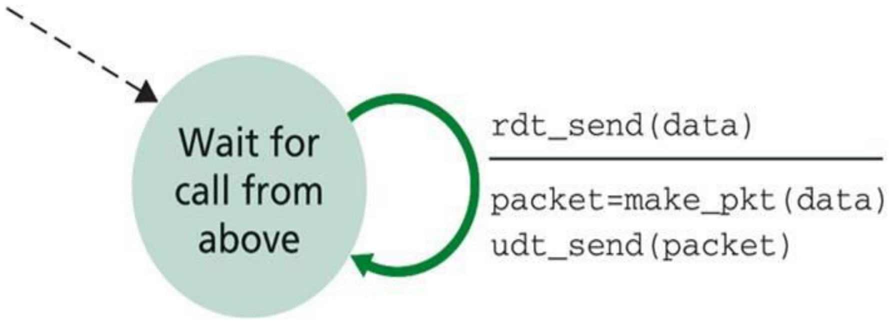
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver


event causing state transition
actions taken on state transition
state 1
state 2
event
actions

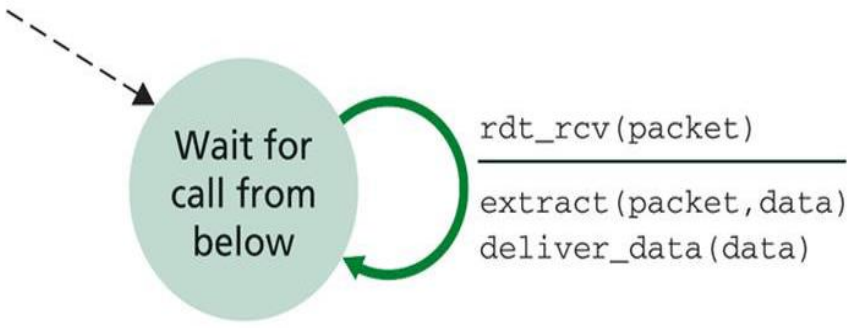
state: when in this "state" next state uniquely determined by next event

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



a. rdt1.0: sending side

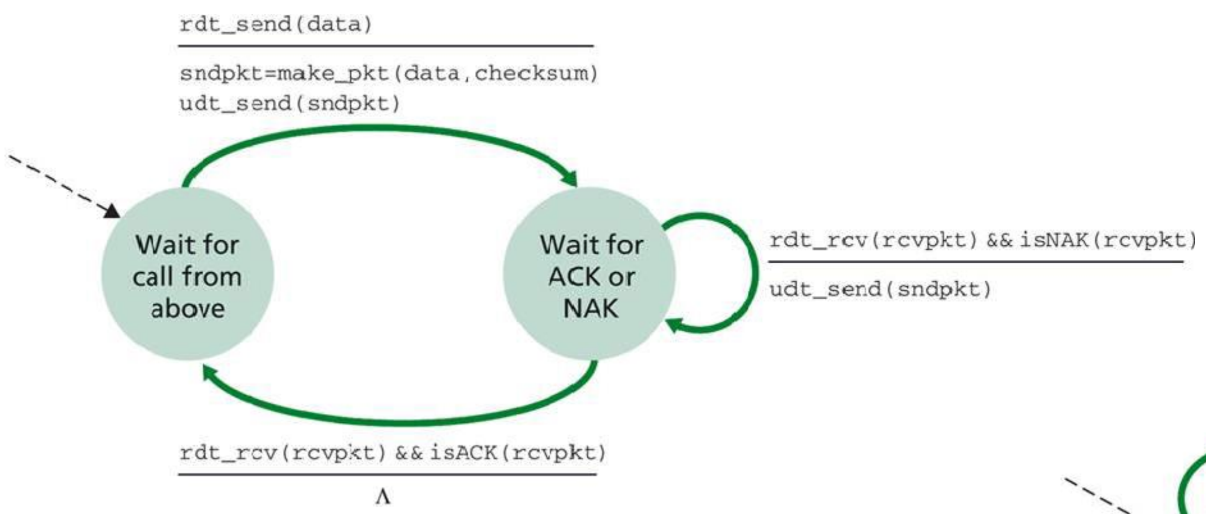


b. rdt1.0: receiving side

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  -
- *the* question: how to recover from errors:
  -
  -
  -
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

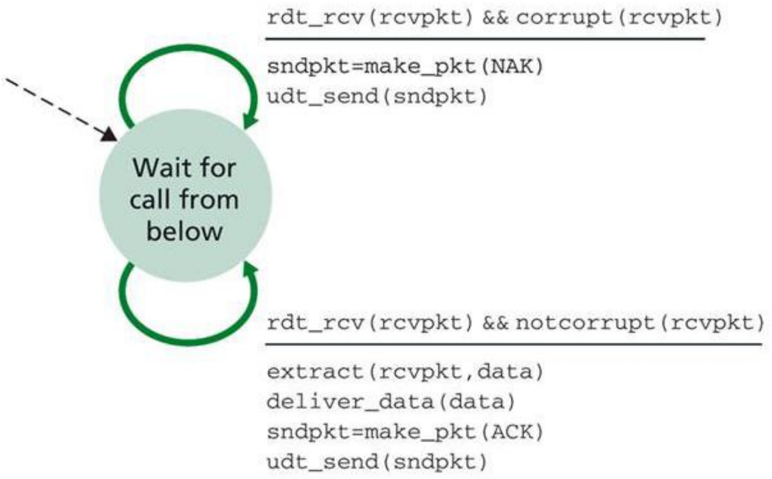
rdt2.0: FSM specification



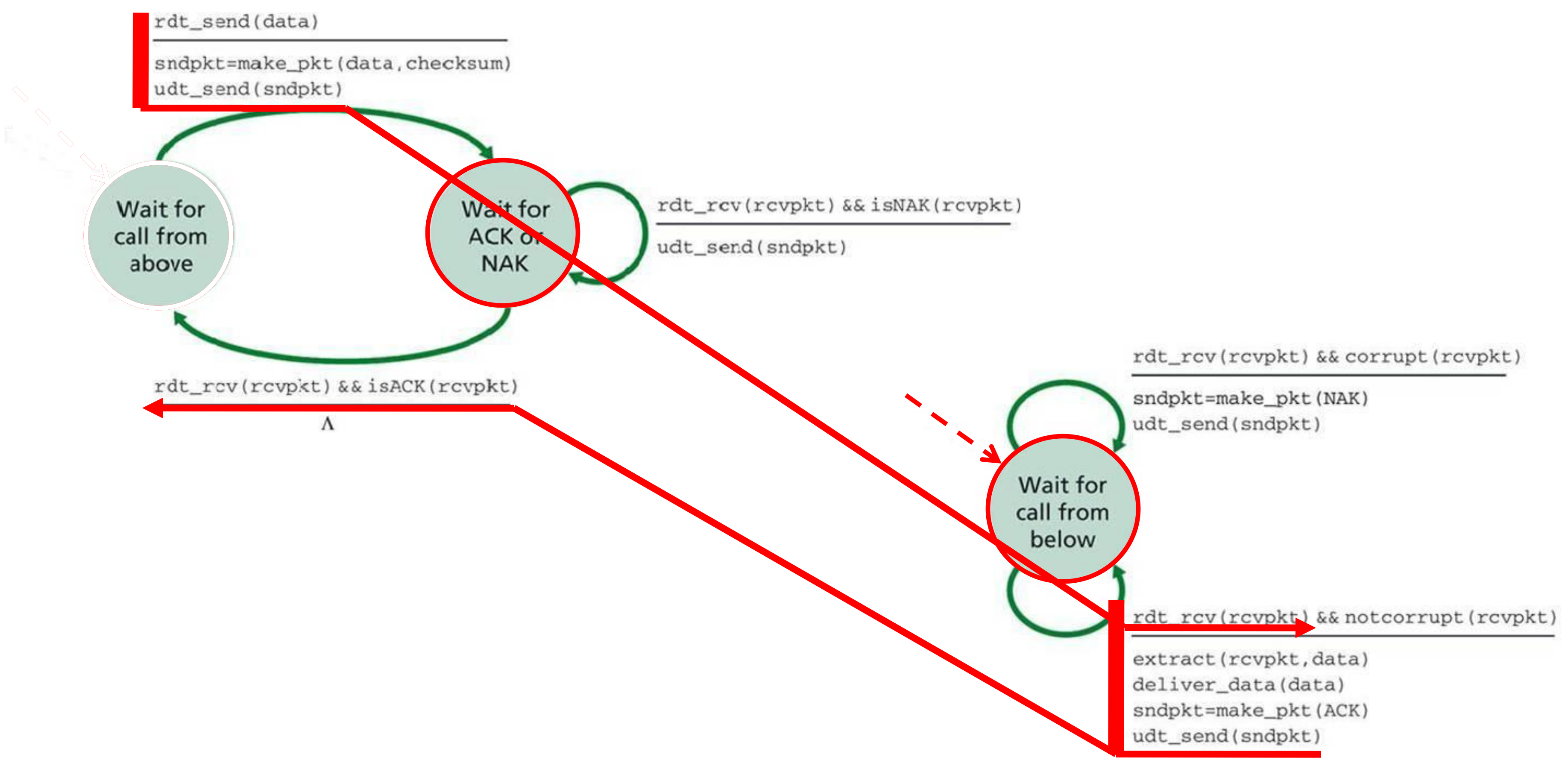
rdt_send(data)
sndpkt=make_pkt(data,checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

sender

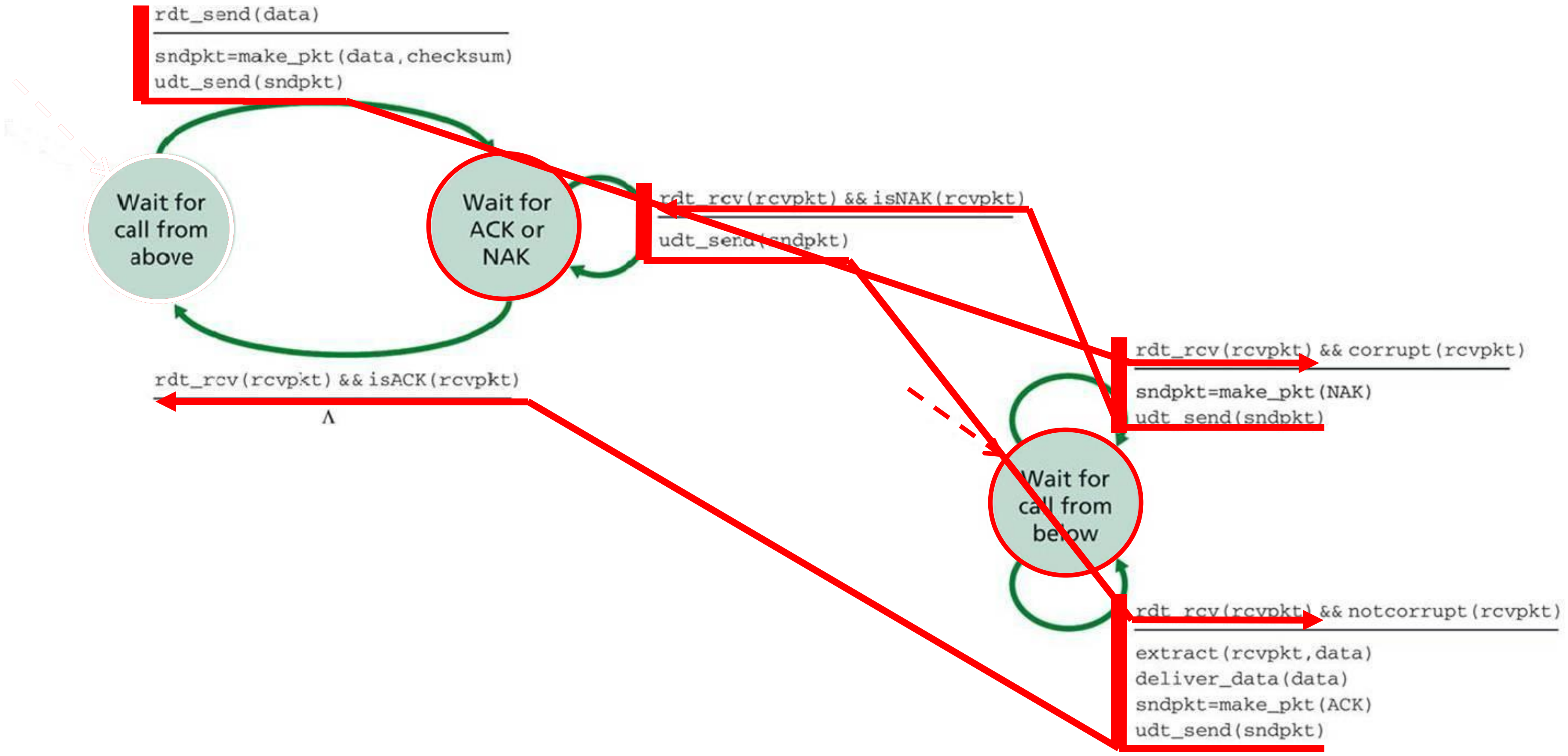
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- 리시버가 nak이라 해서 sender가 재전송으로 하려하는데, 사실 리시버가 실제로 ack를 보냈는데 nak으로 바뀐건지 아님 진짜 nak인지 구분이 안됨
- 그냥 재전송 할 수ㅡㄴ 없다. duplicate때문에

Handling duplicates:

- 일단은 재전송한다. ack/nak에 따라서
- 근데 이러면 receiver는 duplicate를 구분할 수 없음
- 그런고로 sender는 pkt에 sequence number를 추가한다.

Stop and wait

- Sender sends one packet, then waits for receiver response

rdt2.1: discussion

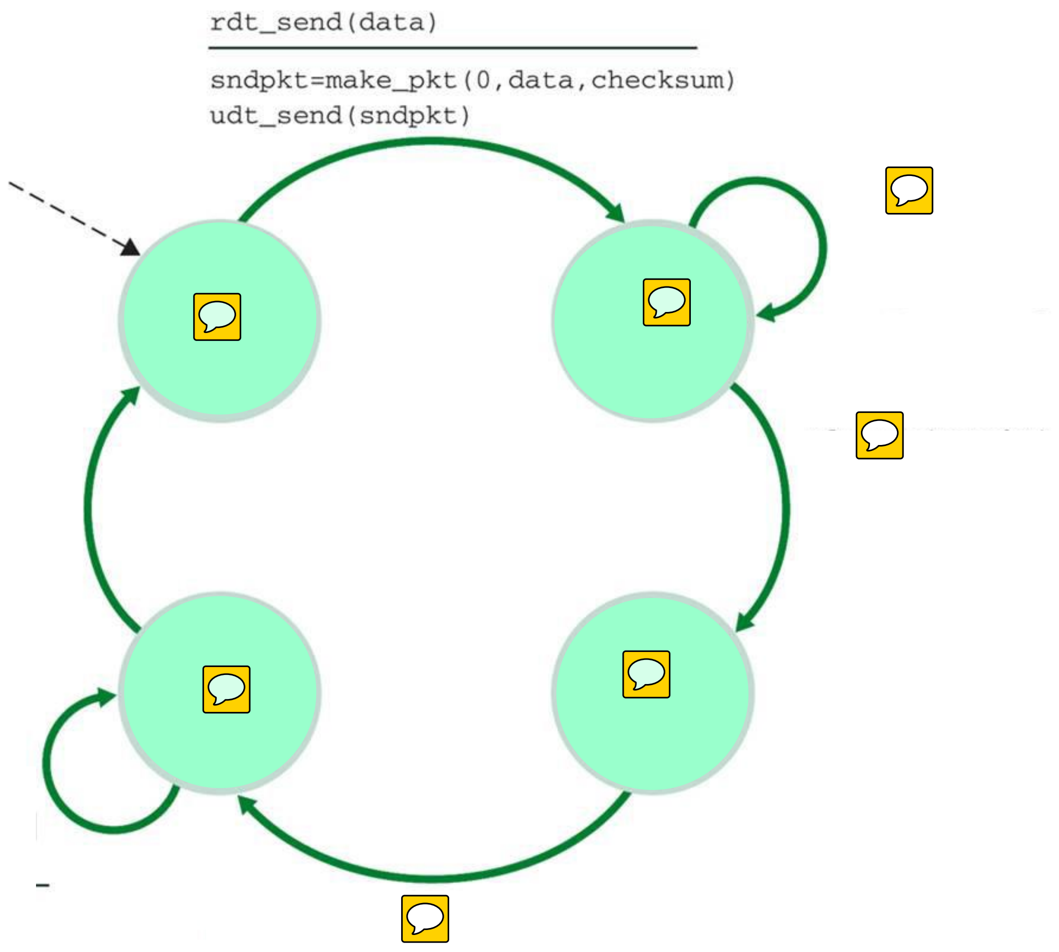
Sender:

- 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. #

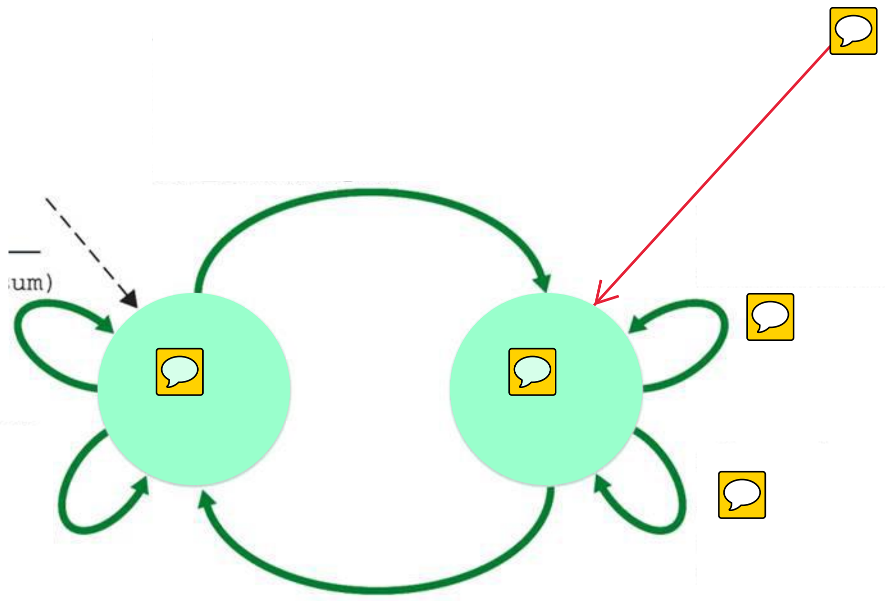
Receiver:

- 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

rdt2.1: sender, handles garbled ACK/NAKs



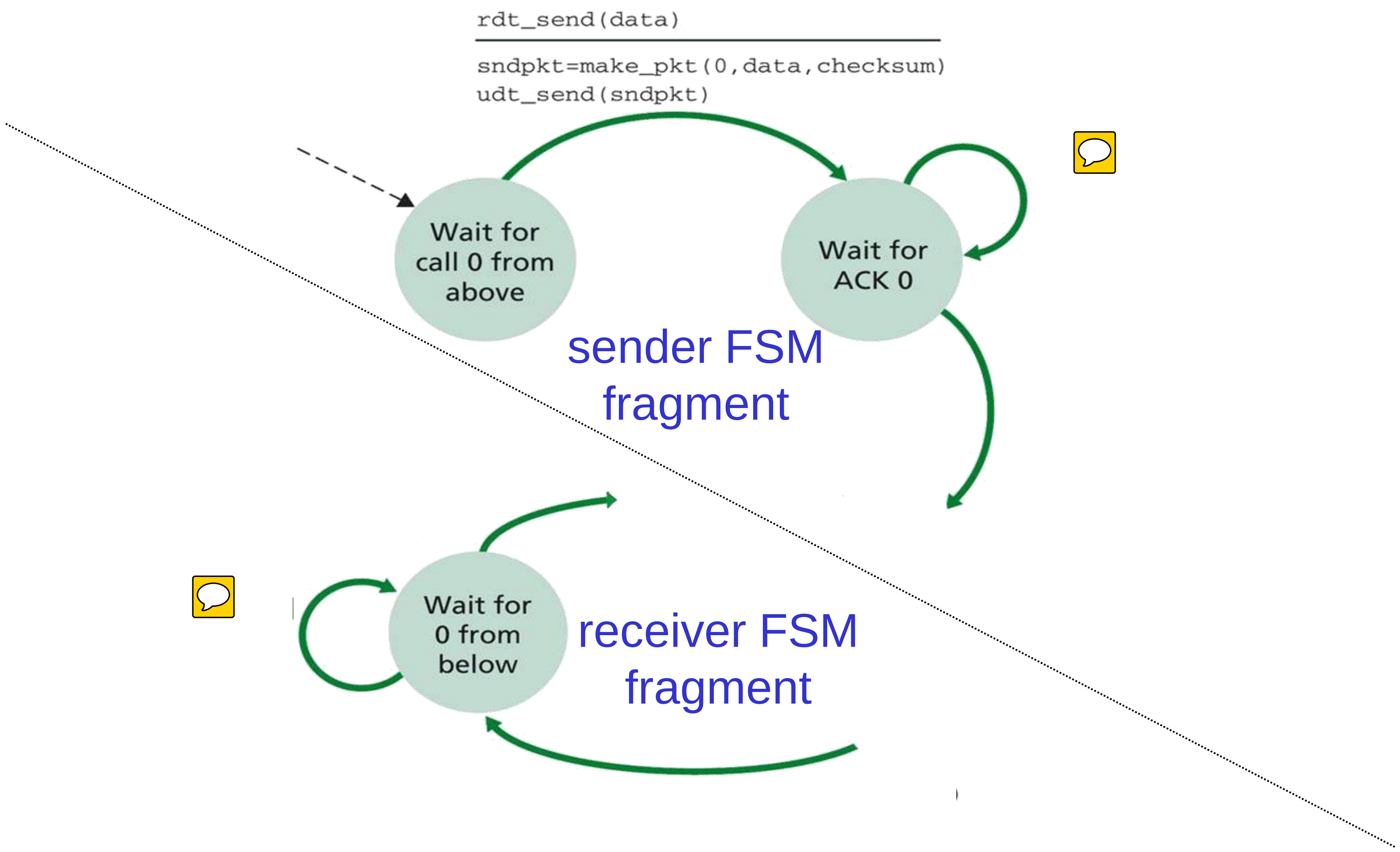
rdt2.1: receiver, handles garbled ACK/NAKs



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



sender FSM fragment

receiver FSM fragment

rdt3.0: channels with errors *and* loss

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

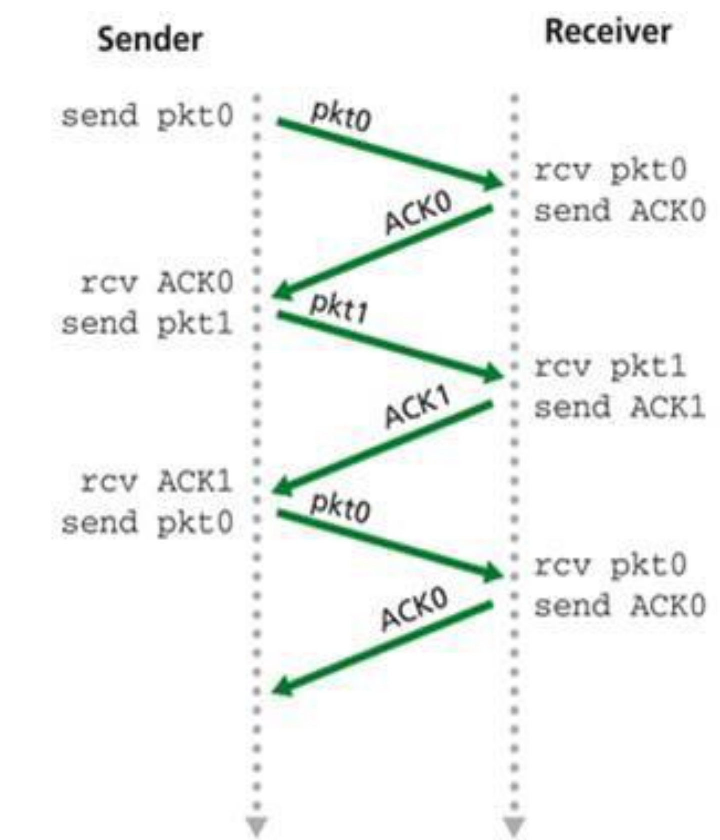
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach:

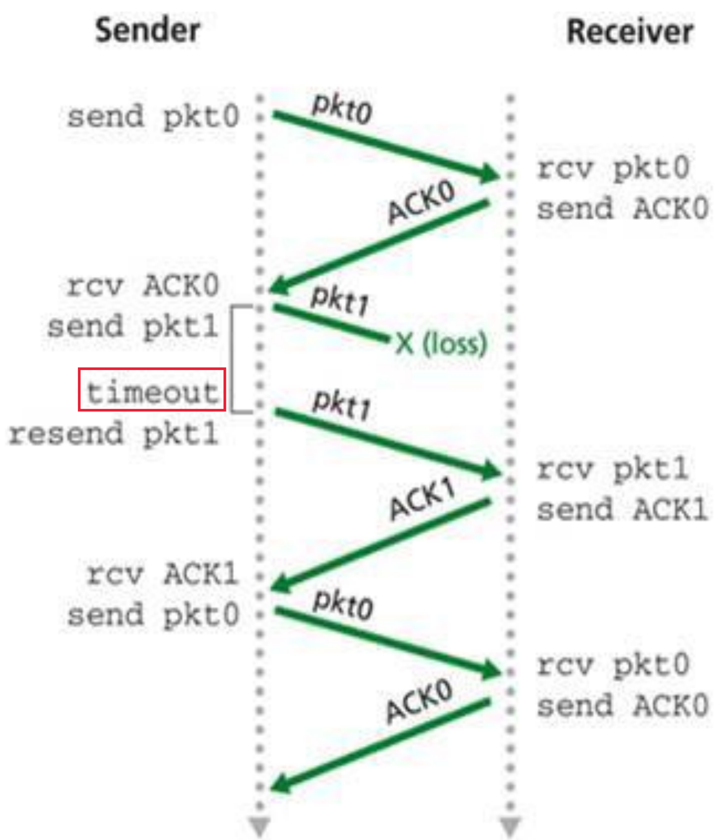
- if pkt (or ACK) just delayed (not lost):
  - retransmissino will be duplicate, but us of seq#, #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires

rdt3.0 sender

rdt3.0 in action

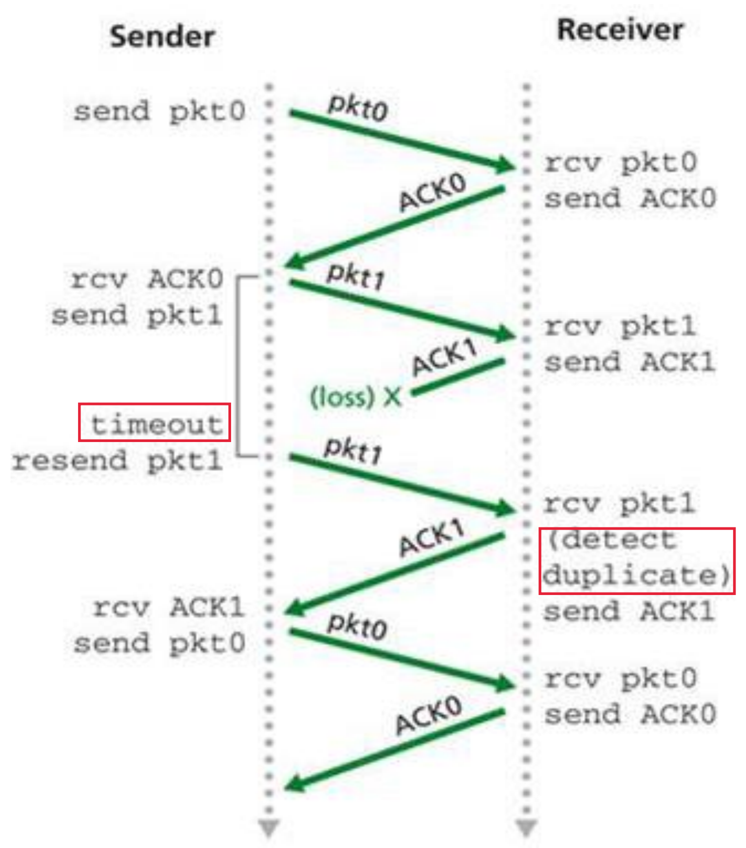


a. Operation with no loss

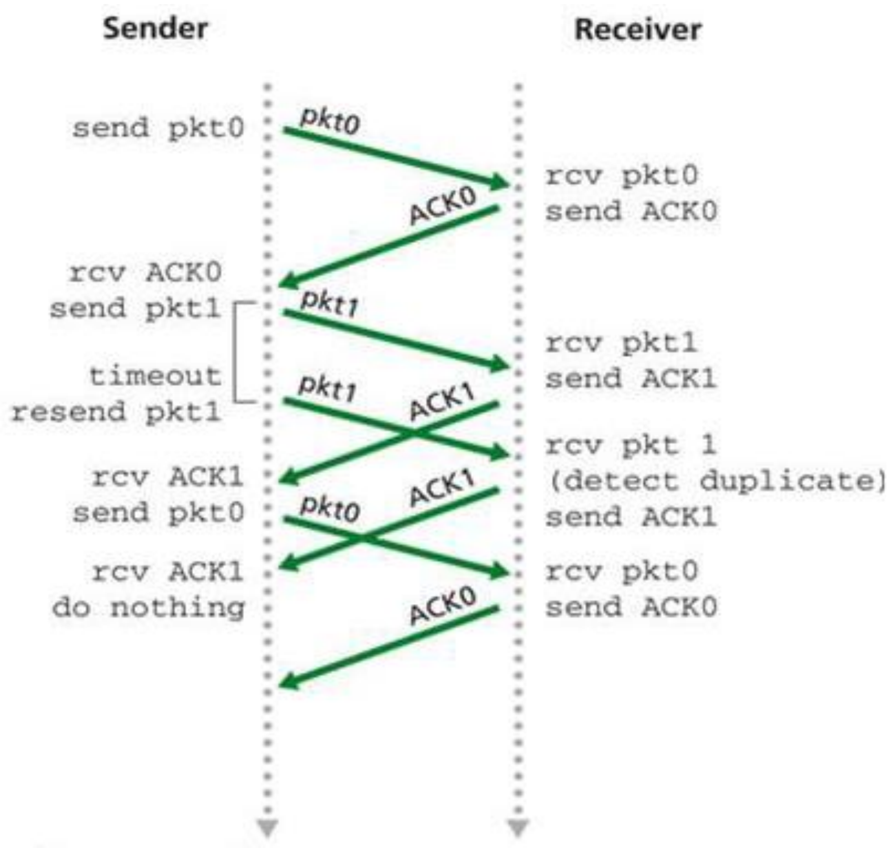


b. Lost packet

rdt3.0 in action



c. Lost ACK



d. Premature timeout

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

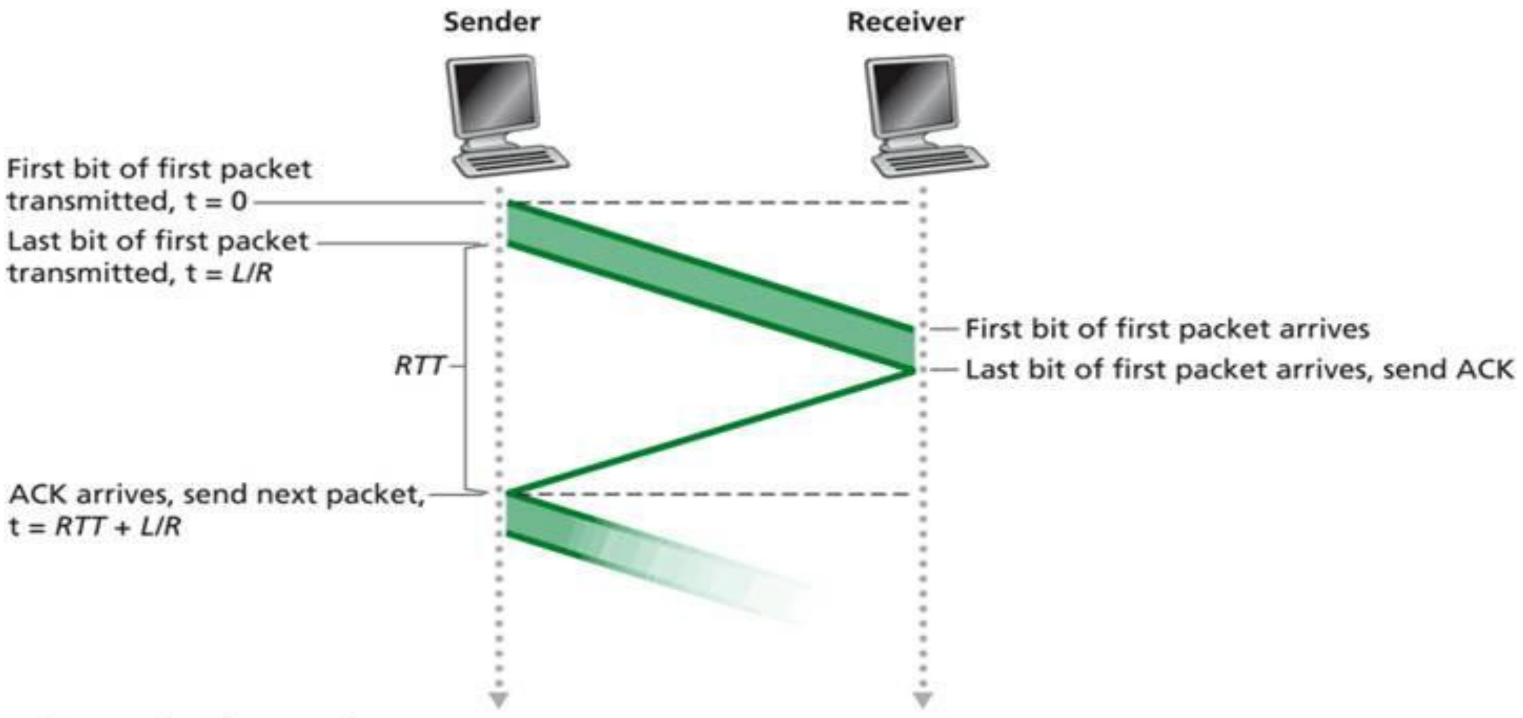
$$T_{\text{transmit}} = \frac{\text{L (packet length in bits)}}{\text{R (transmission rate, bps)}} = \frac{\text{8kb/pkt}}{\text{10**9 b/sec}} = \text{8 microsec}$$

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

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

– 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


Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
ACK arrives, send next packet, t = RTT + L/R
First bit of first packet arrives
Last bit of first packet arrives, send ACK

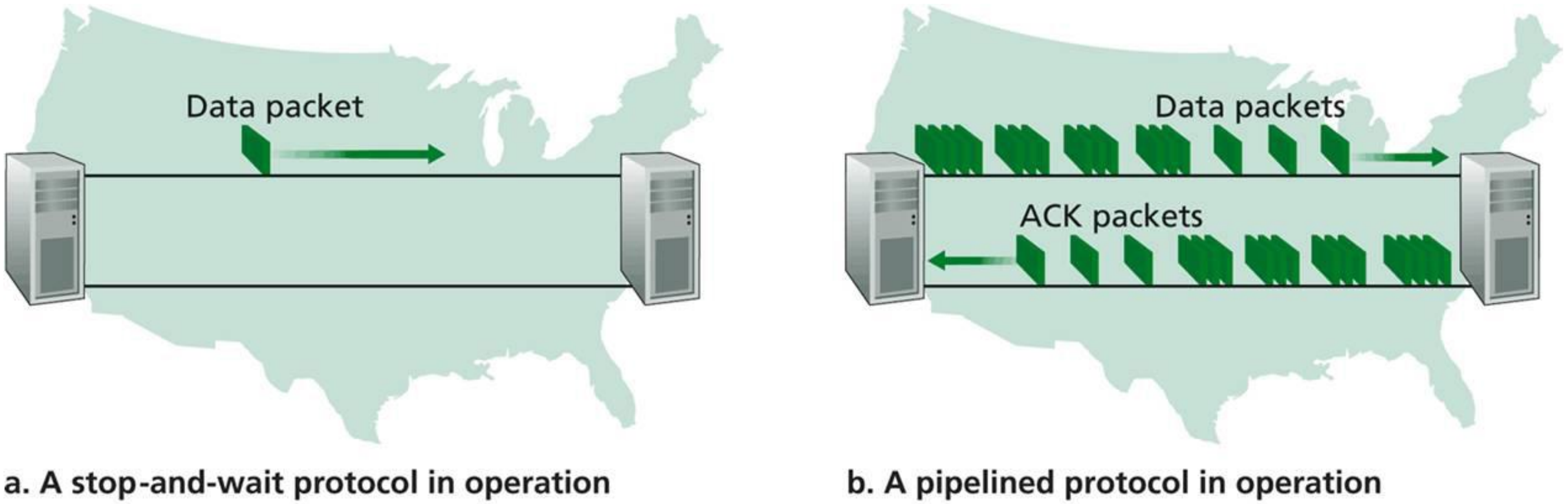
a. Stop-and-wait operation

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

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

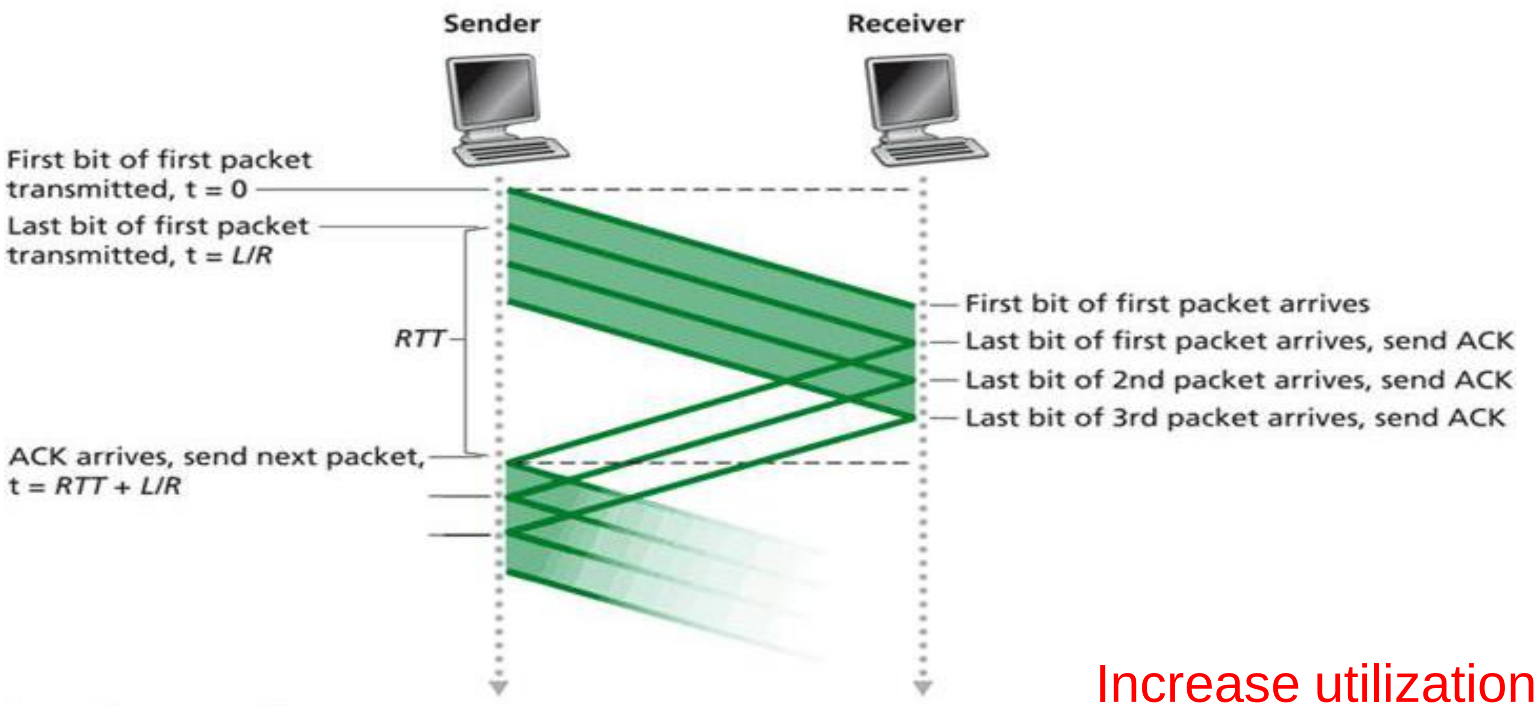


a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

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

Pipelining: increased utilization


Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
ACK arrives, send next packet, t = RTT + L/R
First bit of first packet arrives
Last bit of first packet arrives, send ACK
Last bit of 2nd packet arrives, send ACK
Last bit of 3rd packet arrives, send ACK

b. Pipelined operation

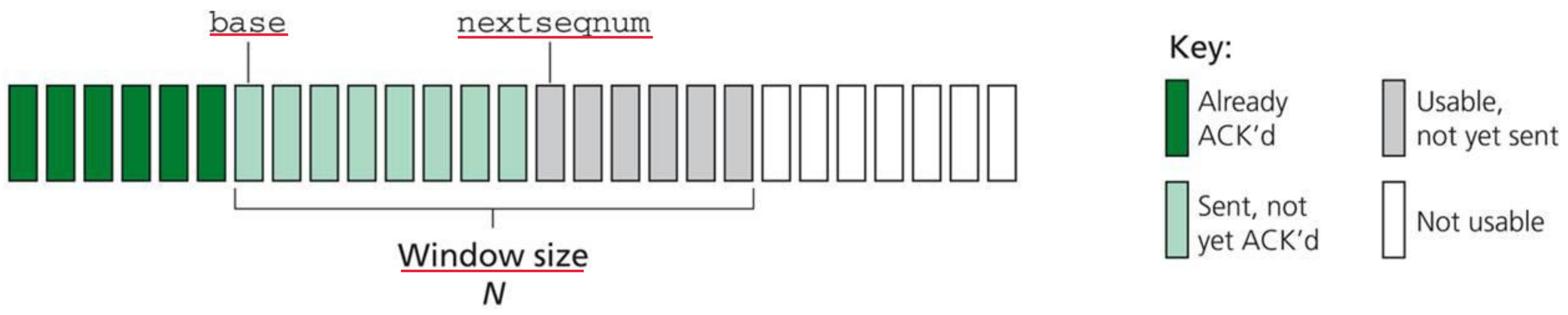
Increase utilization by a factor of 3!

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

Go-Back-N

Sender:

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



- ACK : ACK(n): ACKs all pkts up to, including seq # n - 'cumulative ACK'
  - may receive duplicate ACKs (see receiver)
- timer timer for all in - flight pkts
- *timeout* : timeout(n) : retransmit pkt n and all higher seq# pkts in window

GBN: sender extended FSM

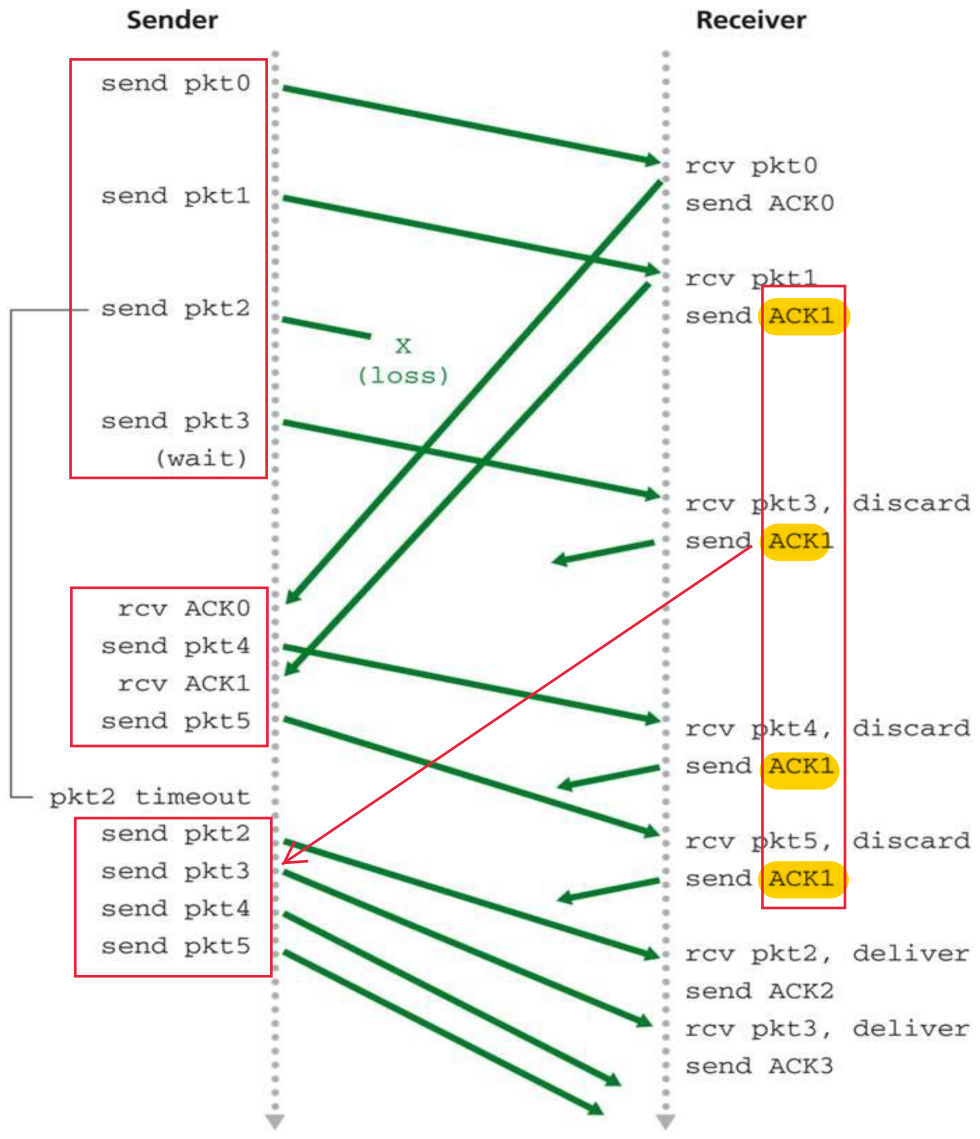
GBN: receiver extended FSM

ACK-only:

  - duplicate ACKs?
  -

- out-of-order pkt:
  -
  - Ack?

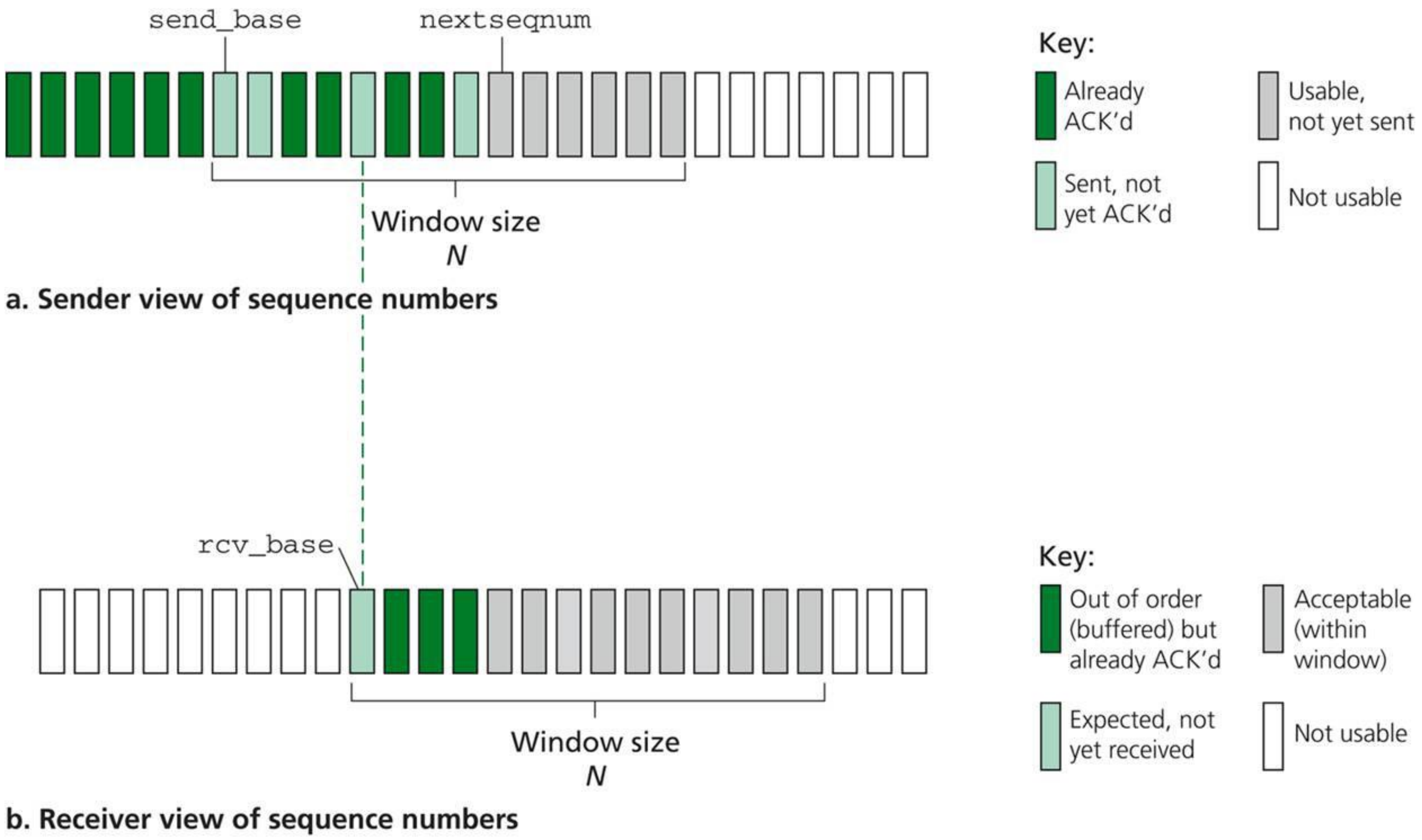
GBN in action



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



send_base
nextseqnum
Window size
N
a. Sender view of sequence numbers
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
rcv_base
Window size
N
b. Receiver view of sequence numbers
Key:
Out of order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received
Not usable

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

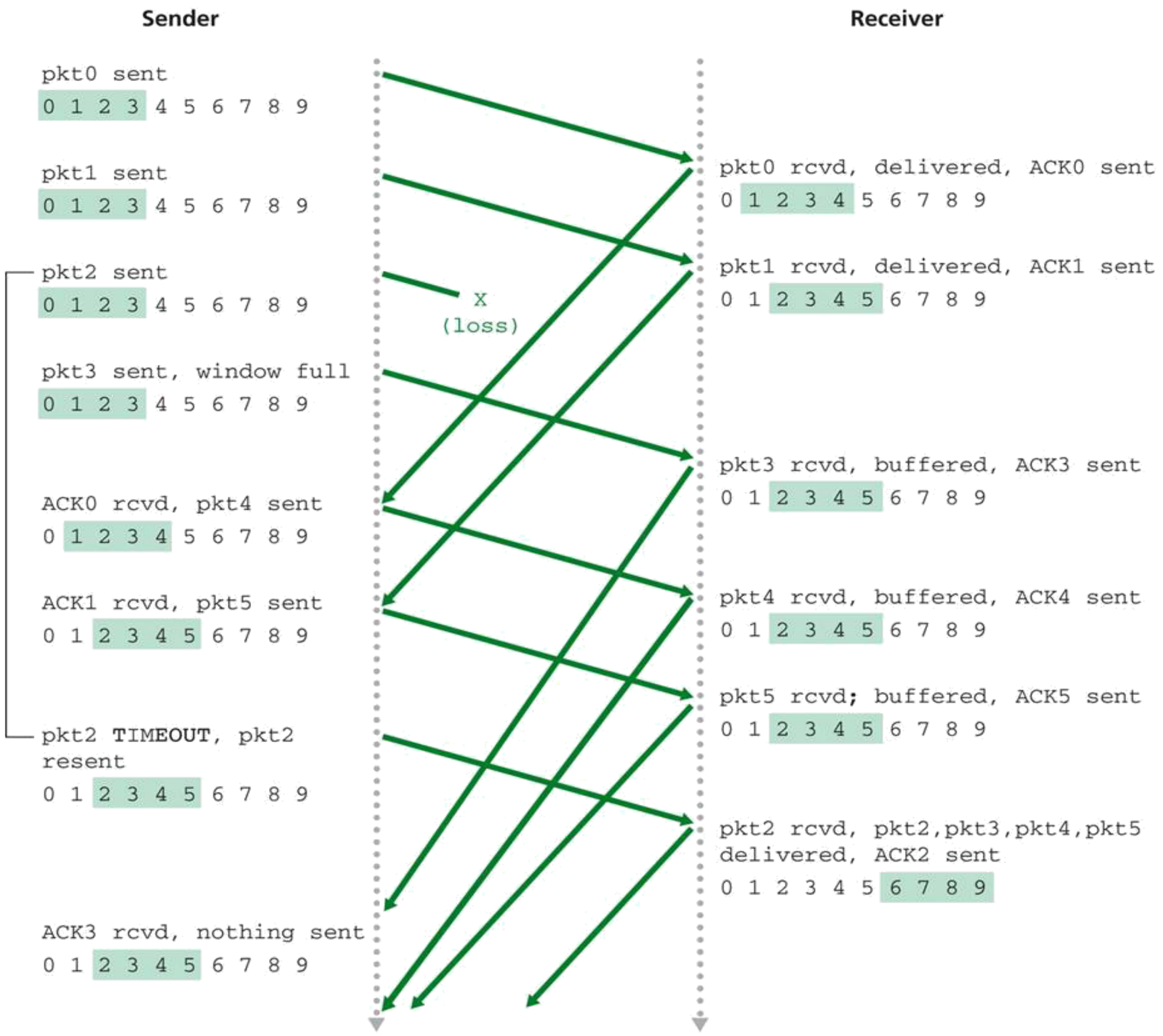
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



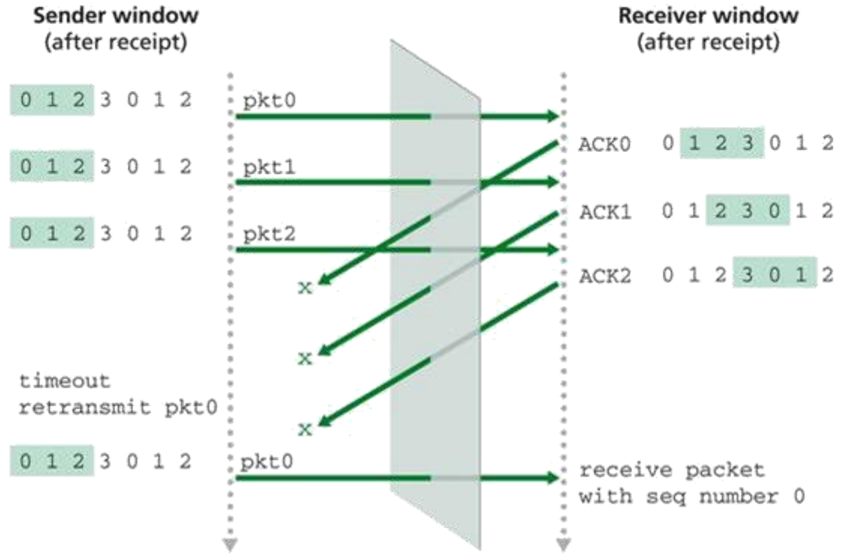
Selective repeat: dilemma

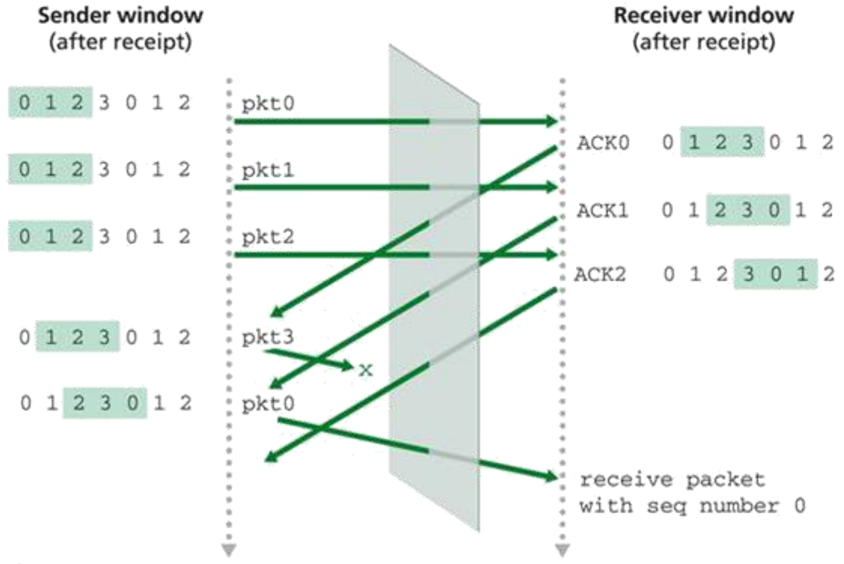
Example:

- 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?


Sender window
(after receipt)
Receiver window
(after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
x
timeout
retransmit pkt0
pkt0
receive packet
with seq number 0


Sender window
(after receipt)
Receiver window
(after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
pkt0
ACK0
ACK1
ACK2
x
receive packet
with seq number 0