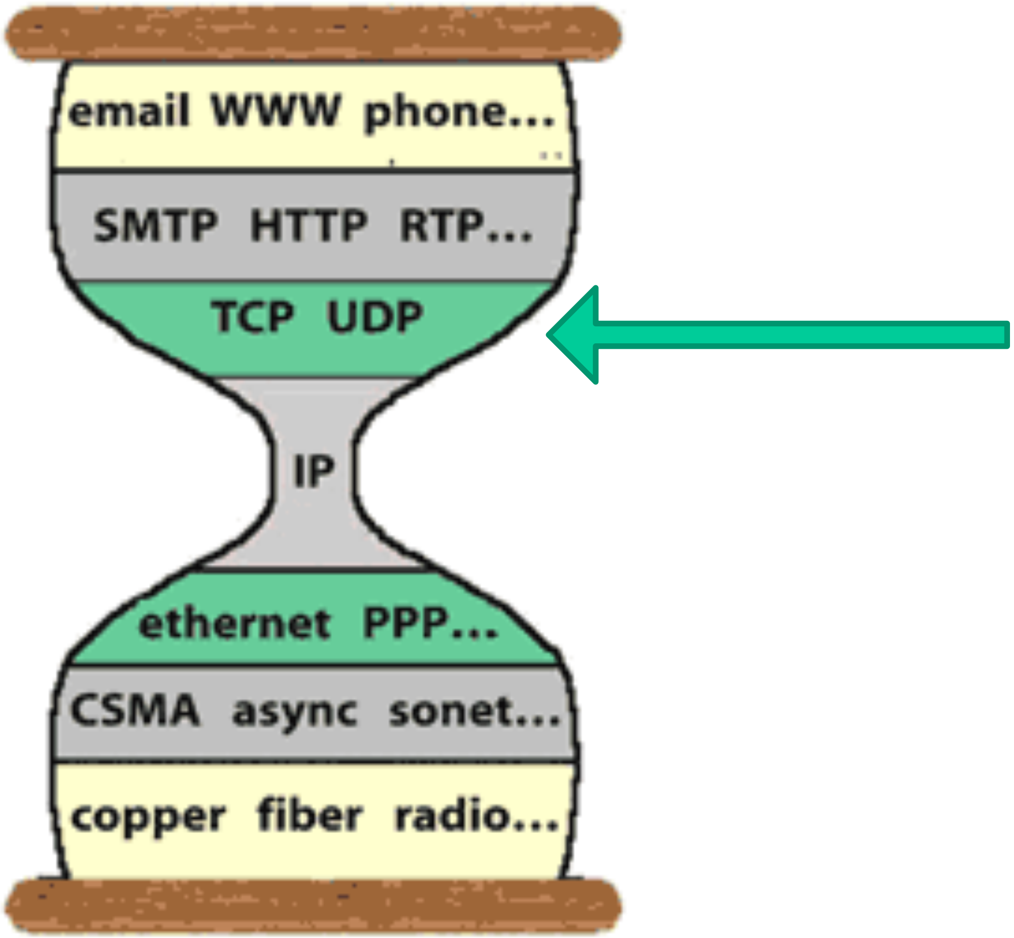
Lecture-5: Transport Layer



Chapter 3

3.1 Transport-layer services

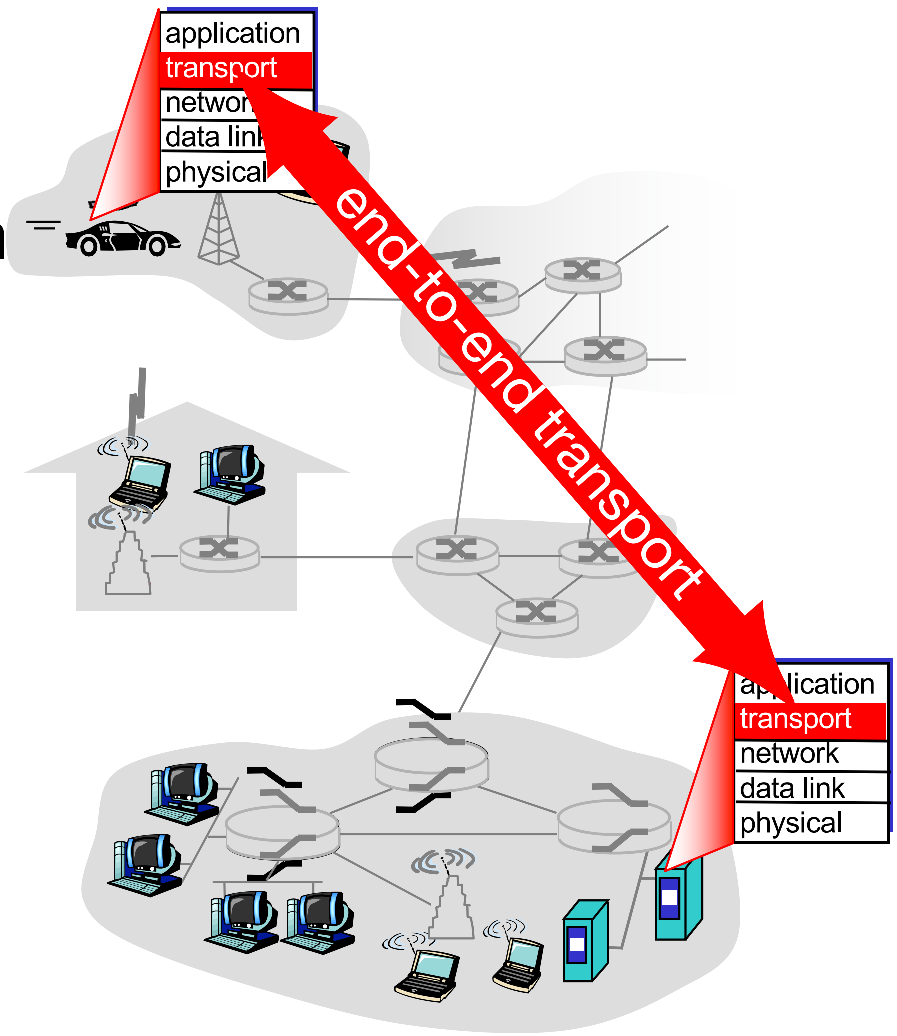
3.2 Multiplexing and de-multiplexing

3.3 Connectionless transport: UDP

3.4 Reliable data transfer

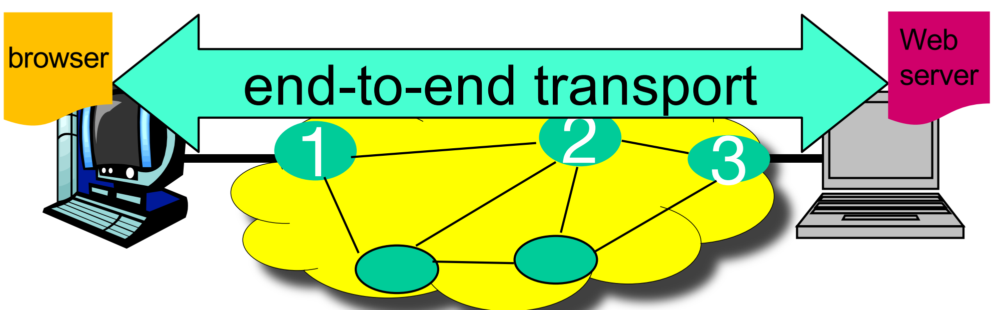
Transport Layer

- Transport protocols:
  - Run in end hosts
  - Offer a logical communication channel between 2 application processes
    - e.g. between a browser and a web server
- Multiple transport protocols exist, providing different transport services
  - **UDP, TCP**
  - **RTP**: realtime transport protocol
  - Latest development: **QUIC**



Transport vs. network layer

- *Transport layer:* logical pipe between *processes*
  - relies on network layer to deliver packets
- *Network layer:* delivering packets hop-by-hop, from a source host to a destination host



Household analogy: (from the textbook)

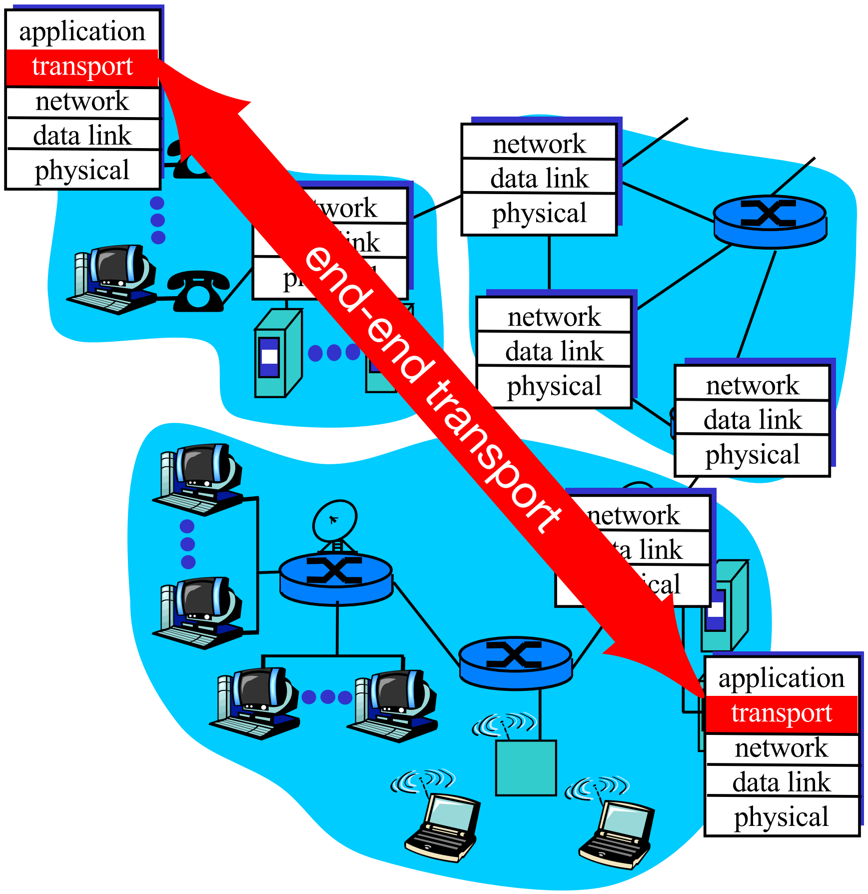
12 young kids sending letters to each other

- processes = kids
- hosts = houses
- application messages = letters in envelopes
- transport protocol = kids parents
- network-layer protocol = postal service

(not exactly right, unless we assume the kids can't read the envelope)

First two transport protocols

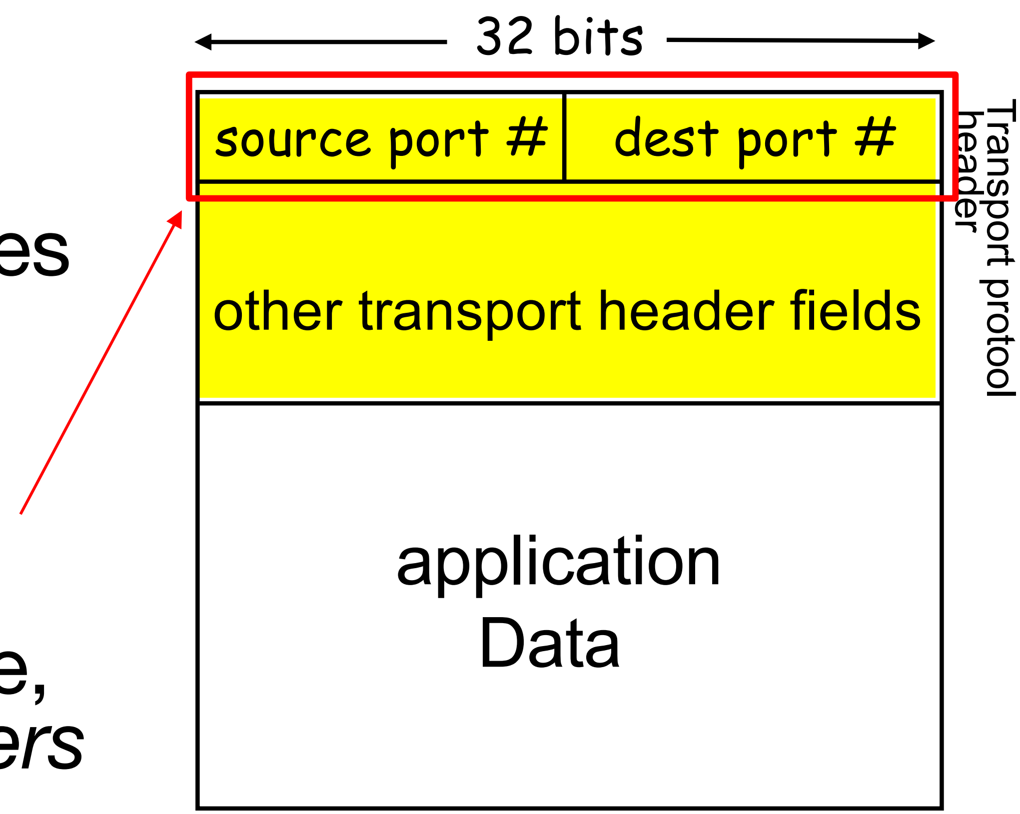
- **TCP**: Reliable, in-order *byte stream* delivery
  - connection setup & tear down
  - flow control
  - congestion control
- **UDP**: Unreliable *datagram* delivery



One common function among all transport protocols: multiplexing/ demultiplexing

How Demultiplexing Works

- A host receives an IP packet
  - It carries source and destination IP addresses
  - It carries a single transport-layer data *segment*
  - The segment transport header contains source, destination *port numbers*
- Host uses IP addresses & port numbers to direct each segment to the appropriate socket

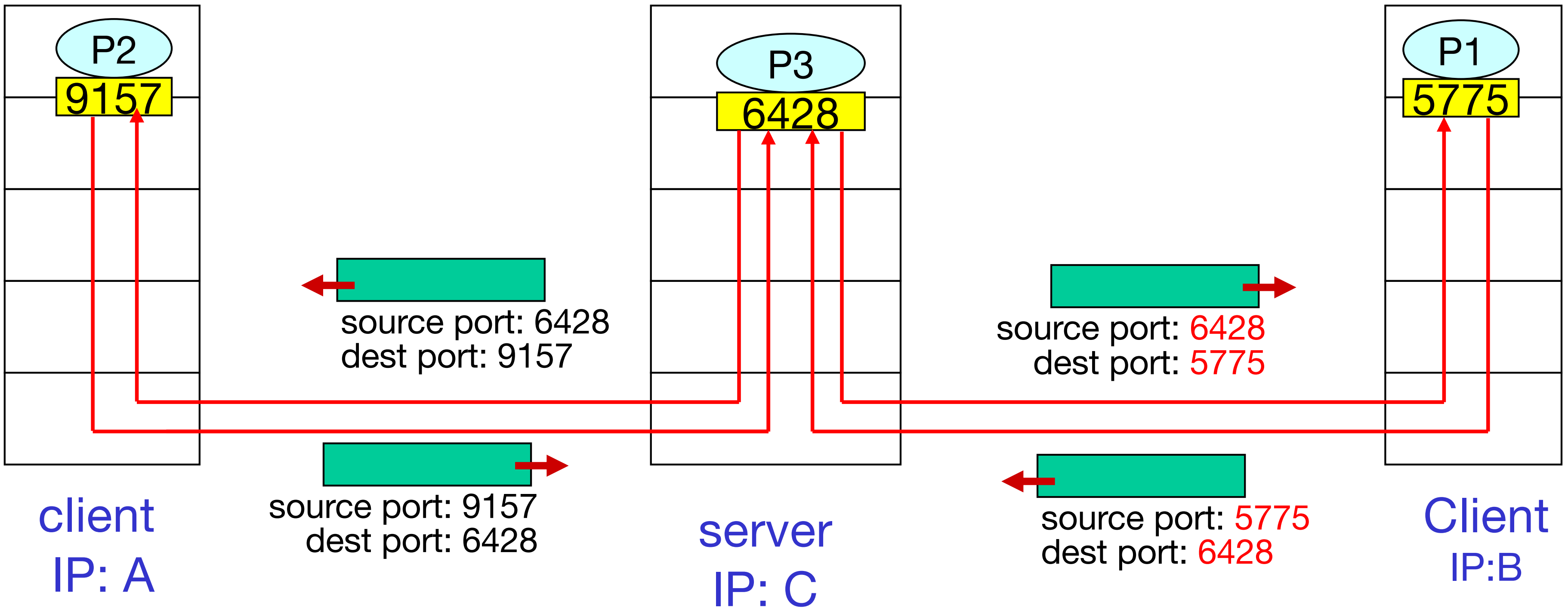


TCP/UDP segment format

Connectionless Demultiplexing

- When sending a packet to a UDP socket, one specifies
  - destination IP address
  - destination port #
- When destination host receives a UDP packet:
  - directs the packet to the socket listening to the destination port# carried in the packet
- Packets with *same destination address and port #* are directed to the same socket at the destination host
  - They may have different source IP addresses and/or source port#s

Connectionless transport: return a reply



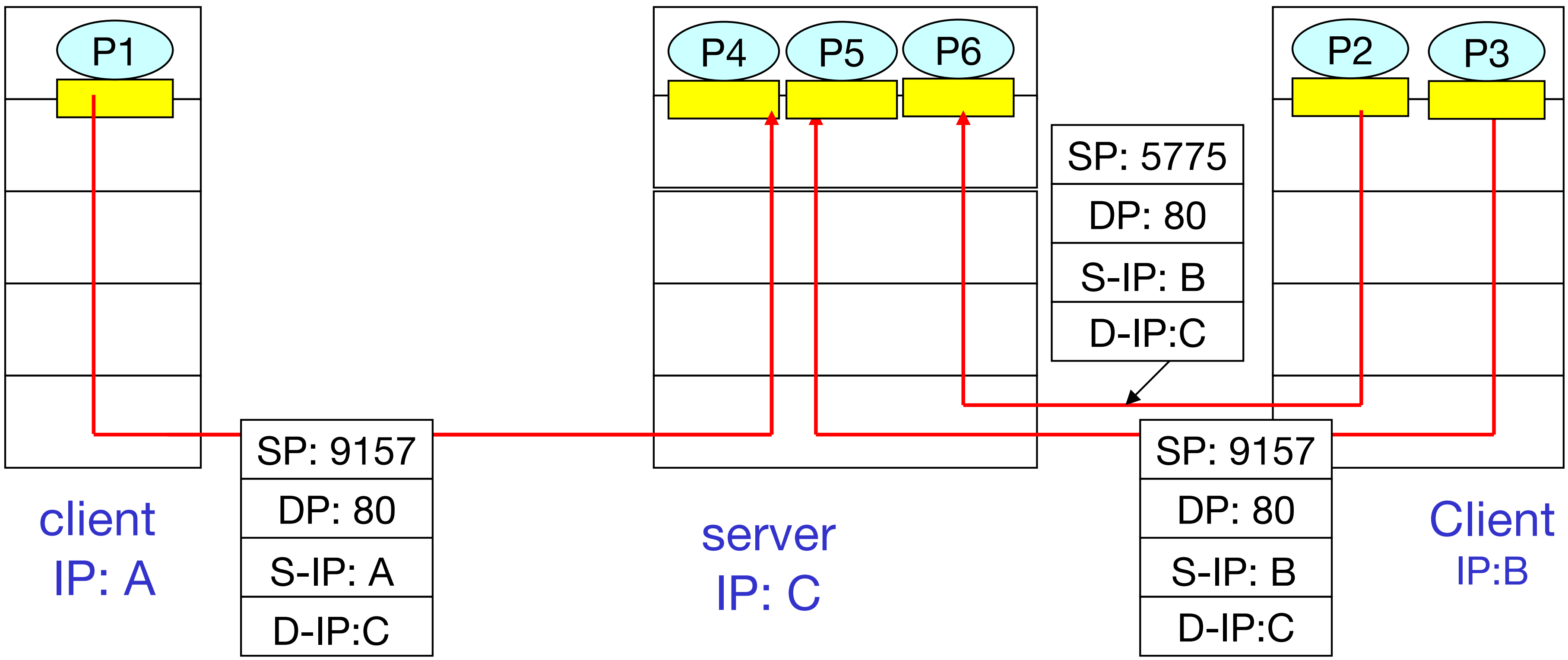
How a server figures out where to return a reply:

UDP specification (RFC768): "UDP module must be able to determine the source and destination internet addresses and the protocol field from the internet header. One possible UDP/IP interface would return the whole internet datagram including all of the internet header in response to a receive operation"

Connection-oriented Demultiplex

- A TCP socket is identified by 4-tuple:
  - source IP address
  - source port number
  - dest. IP address
  - dest. port number
- receiving host uses all the four values to direct a segment to appropriate socket
- A server host may support many simultaneous TCP sockets in parallel:
  - each socket identified by its own 4-tuple
- e.g. a web server creates separate sockets for each connected client

Connection-oriented demux (cont)



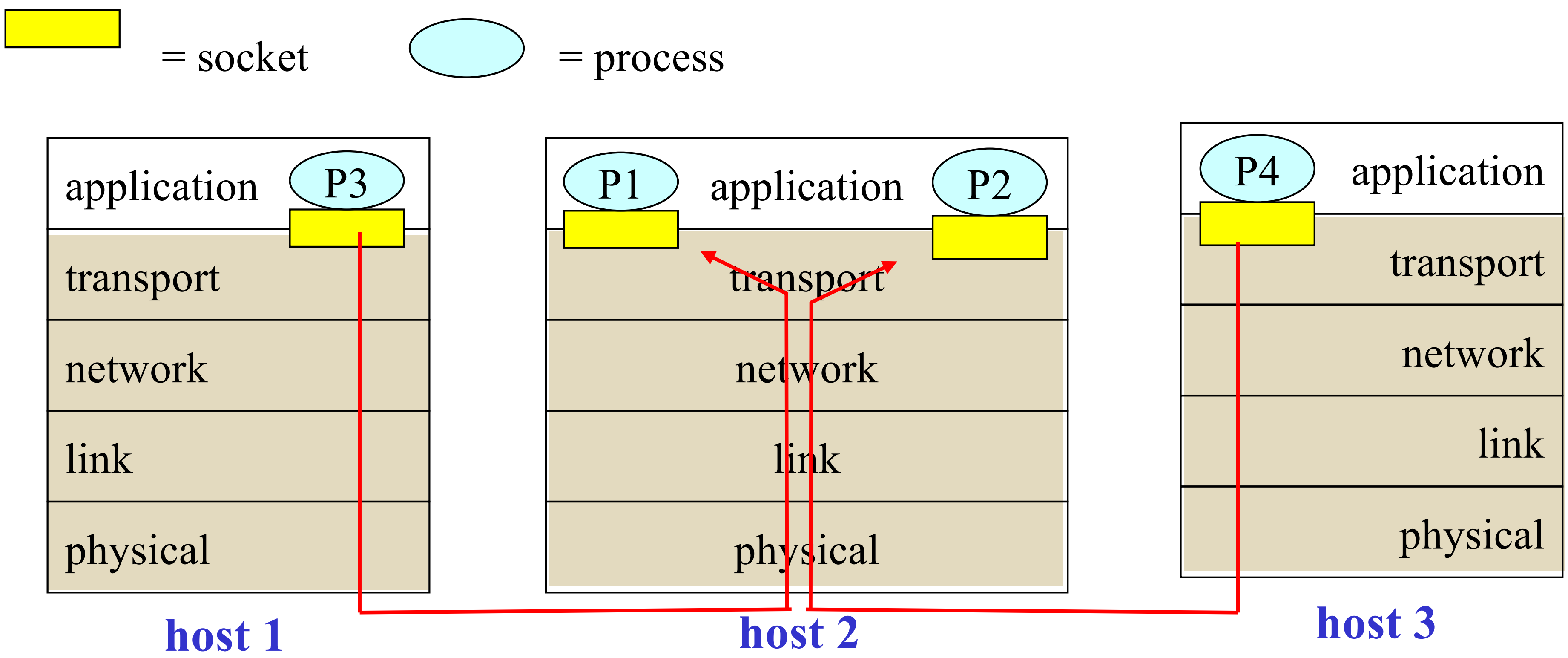
A server process can tell apart

- data from different hosts by IP addresses
- Data from the same host but different processes by source port numbers

Multiplexing/demultiplexing

<u>Multiplexing at sender:</u>
gathering data from sockets, enveloping data with header (used for demultiplexing later)

<u>Demultiplexing at receiver:</u>
delivering received segments to correct socket



Each process is identified by IP address and port#

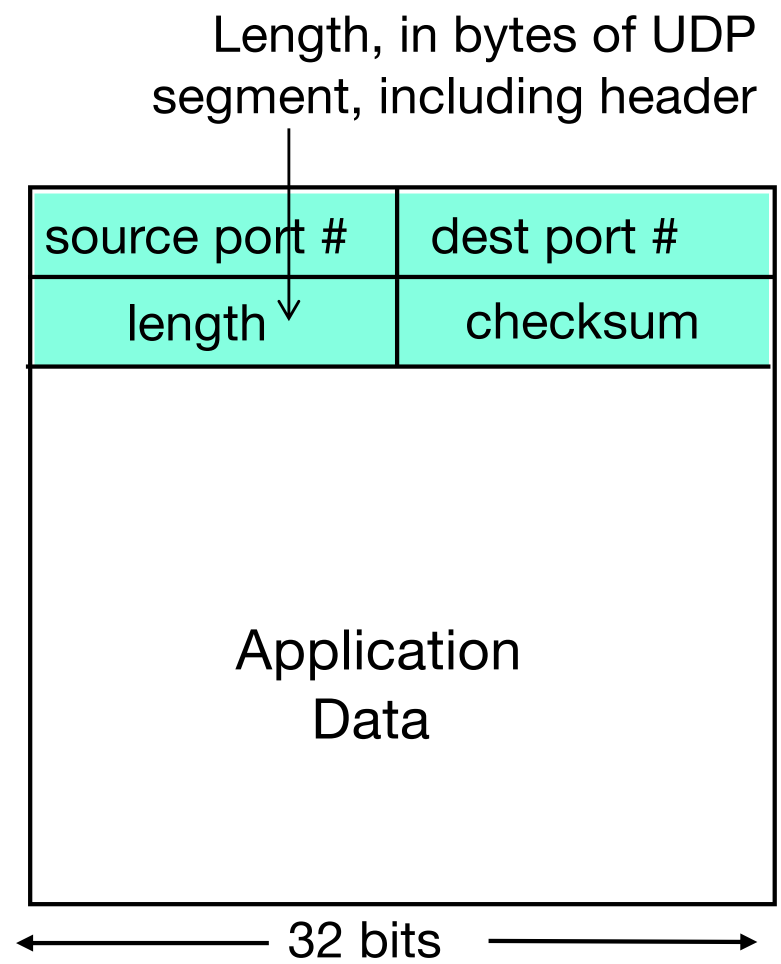
Now let's look at protocol specifics

UDP: User Datagram Protocol [RFC 768]

- A UDP segment may be lost, duplicated, or delivered out of order
- *connectionless*:
  - no prior handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- UDP usages:
  - DNS
  - streaming multimedia apps (loss tolerant, rate sensitive)
- If application requires reliable transfer: add reliability at application layer

UDP header format

- 👍 simple: performs demultiplexing only
  - no connection state at sender, receiver
- 👍 small header size
- 👎 no delivery reliability guarantee
- 👎 no congestion control: a UDP sender can blast away as fast as it wants



UDP segment format

UDP checksum

Goal: detect bit errors in the transmitted segment

Sender:

- treat segment content as a sequence of 16-bit integers
  - *the checksum field set to 0*
- checksum: adding up segment contents (1's complement sum)
- Put 1's complement of the resulting value into the checksum field
- UDP checksum is optional:
  - if don't need checksum, sender sets checksum field to 0

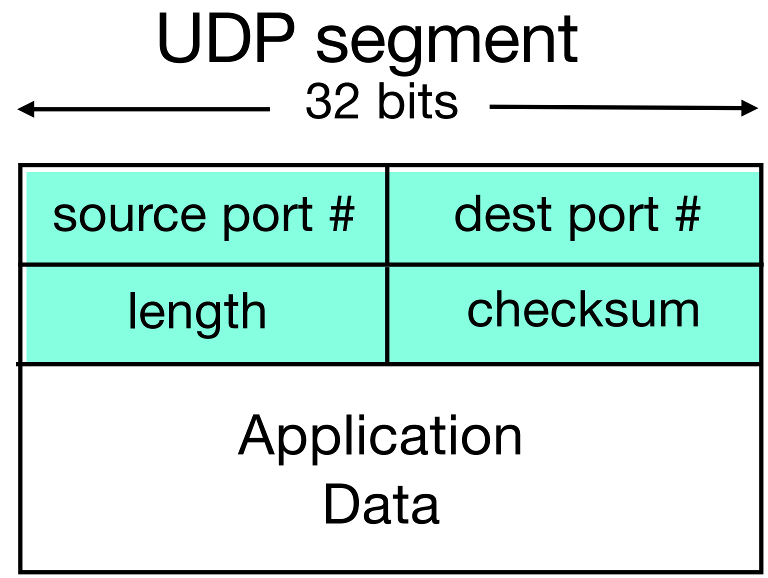
Receiver:

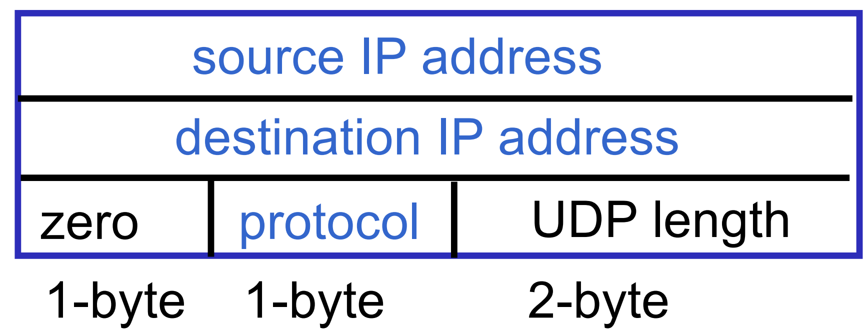
- Adds up the whole received UDP segment
  - Including the checksum field
- If the result is all 1's: no bit error

https://en.wikipedia.org/wiki/User_Datagram_Protocol#Checksum_computation

What included in UDP Checksum calculation

- checksum: computed over
  - the pseudo header, and
  - UDP header and data.
- pseudo header: protection against mis-delivered IP packets
  - pseudo header is not carried in UDP packet, nor counted in the length field





Reliable Data Transfer

The textbook dived into a detailed evolutionary explanation to show what factors are necessary for reliable data delivery

A simplified version of the Principles of Reliable Data Transfer

- 3 questions
  - How many different types of errors?
  - How to detect each type of errors?
  - How to recover from each type of errors?
- **3 types of errors**, and how to **detect** them
  - **Corrupted bits** in a packet: detected by checksum
  - **Packet loss**:
    - Receiver sends an Acknowledgment for received data
    - Sender sets alarm timer: if no ACK before timeout → data lost
  - Packets **arrived out of order**: detected by assigning each packet a sequence number
- **Recovery**
  - retransmitting the bit-error / lost packet
  - Pass to upper layer in-order

Three basic components in reliable data delivery by sender retransmission

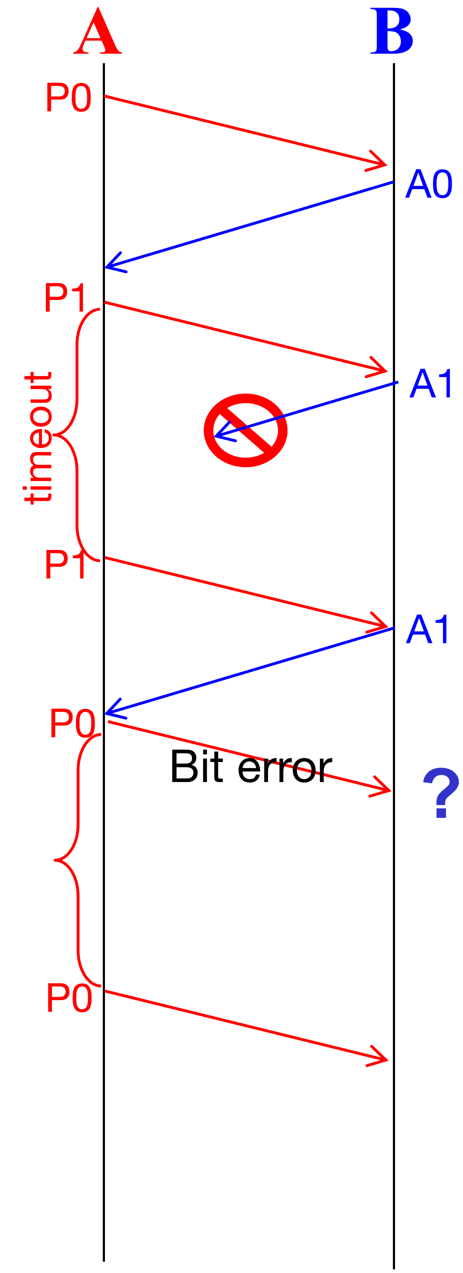
- Sender side:
  - Assign a sequence # to each piece of data: *uniquely* identifies individual packet
  - Set a retransmission timer after sending a packet
    - If ACK arrives before the timer expires: cancel the timer
    - When the timer expires: retransmit the packet
- Receiver side
  - After receiving expected data: send an Acknowledgment (ACK) to the data sender

The devils are always in the details

Design-1: Stop-and-Wait

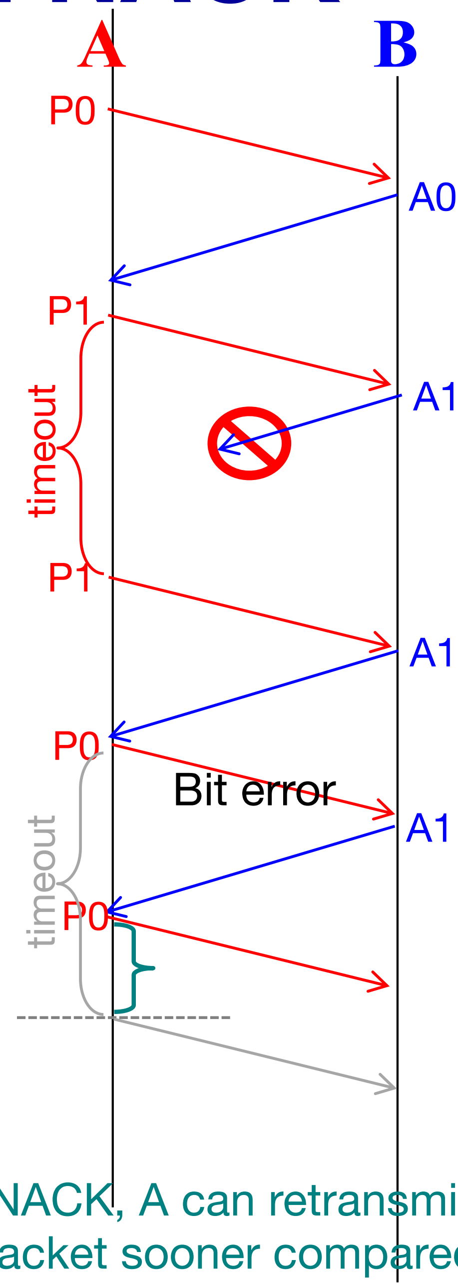
- Sender A sends one data packet, sets retransmission timer, then waits for ACK from receiver B
  - Each packet is assigned a seq#
  - we assume seq# has 1 bit
- When B received a packet with bit error:

  Option-1:
  - B does nothing
  - A times out and retransmits



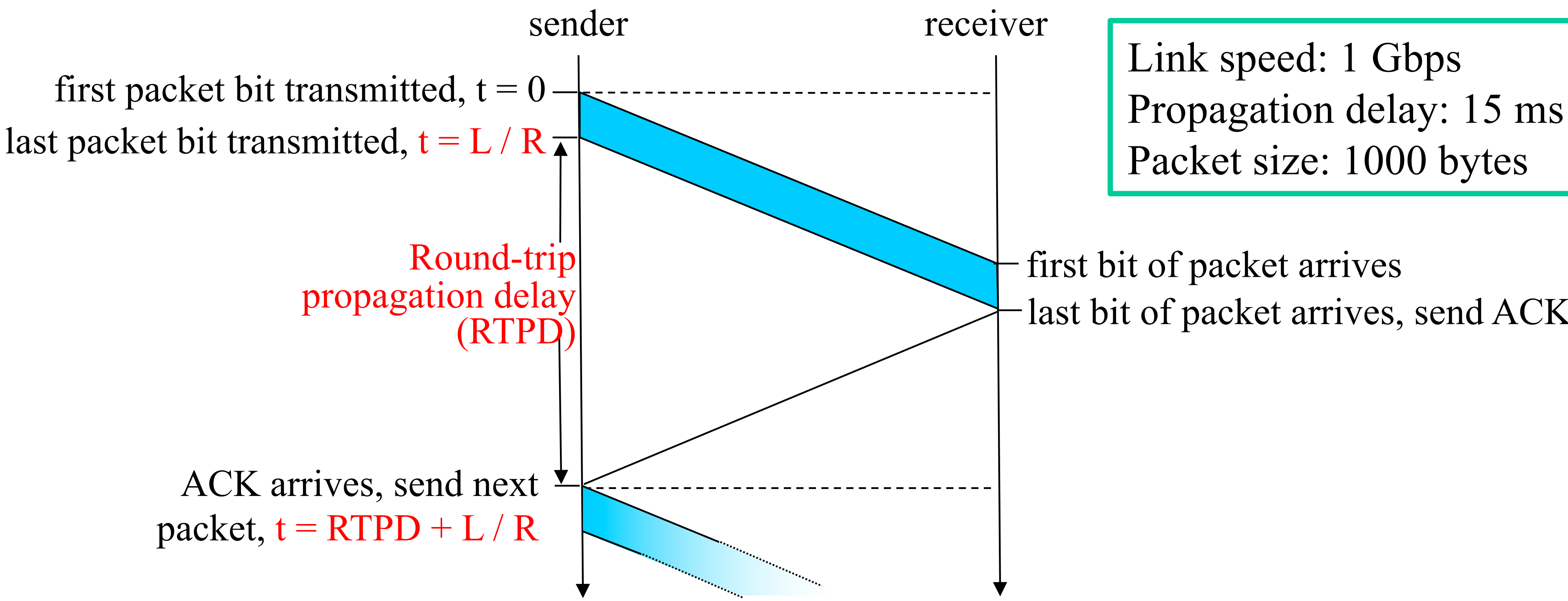
Design-2: Stop-and-Wait with NACK

- Sender A sends one data packet, sets timer, then waits for ACK from receiver B
  - Each packet is assigned a seq#
  - we assume seq# has 1 bit
- When B received a packet with bit error: Option-2:
  - B sends an ACK with the seq# of the *last* correctly received packet
  - A treats the *duplicate ACK* as negative-ACK (i.e. B did not get P_0): *retransmits* P_0



With NACK, A can retransmit lost packet sooner compared to wait-for-timeout

Stop-and-Wait in action



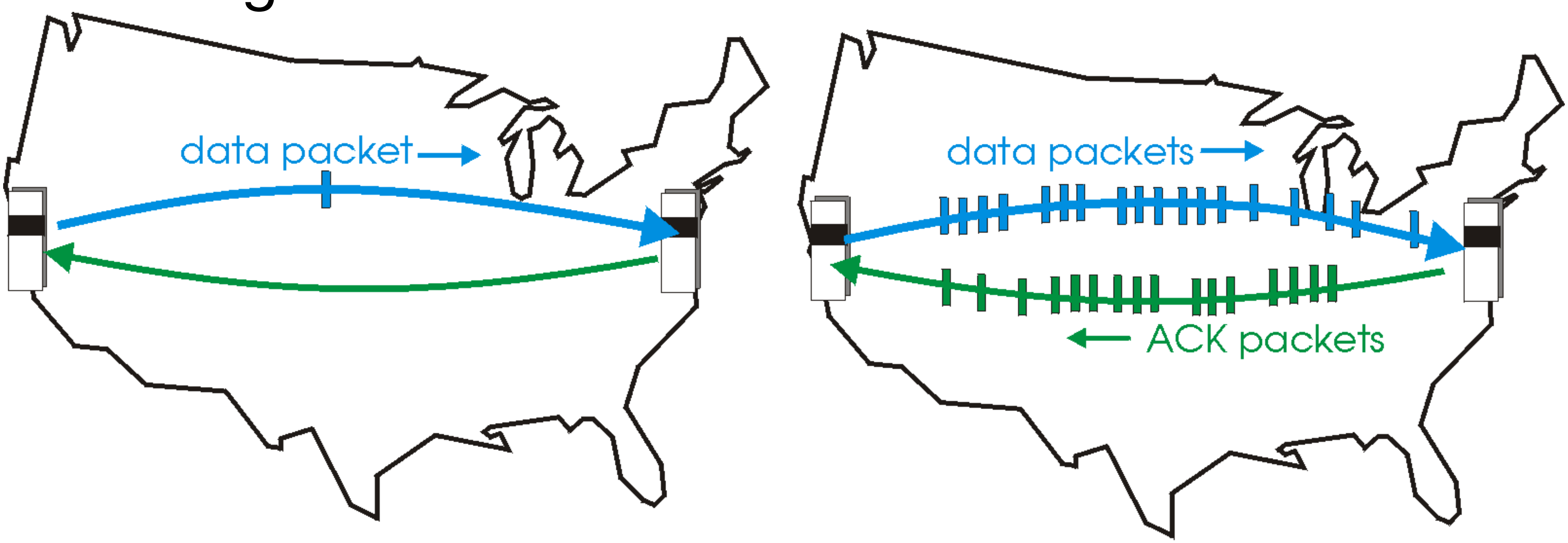
Link speed: 1 Gbps
Propagation delay: 15 ms
Packet size: 1000 bytes

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\mu s$$

$$U_{sender} = \frac{d_{trans}}{RTPD + d_{trans}} = \frac{0.008ms}{30.008ms} = 0.00027$$

Design-3: Pipelining packet transmission

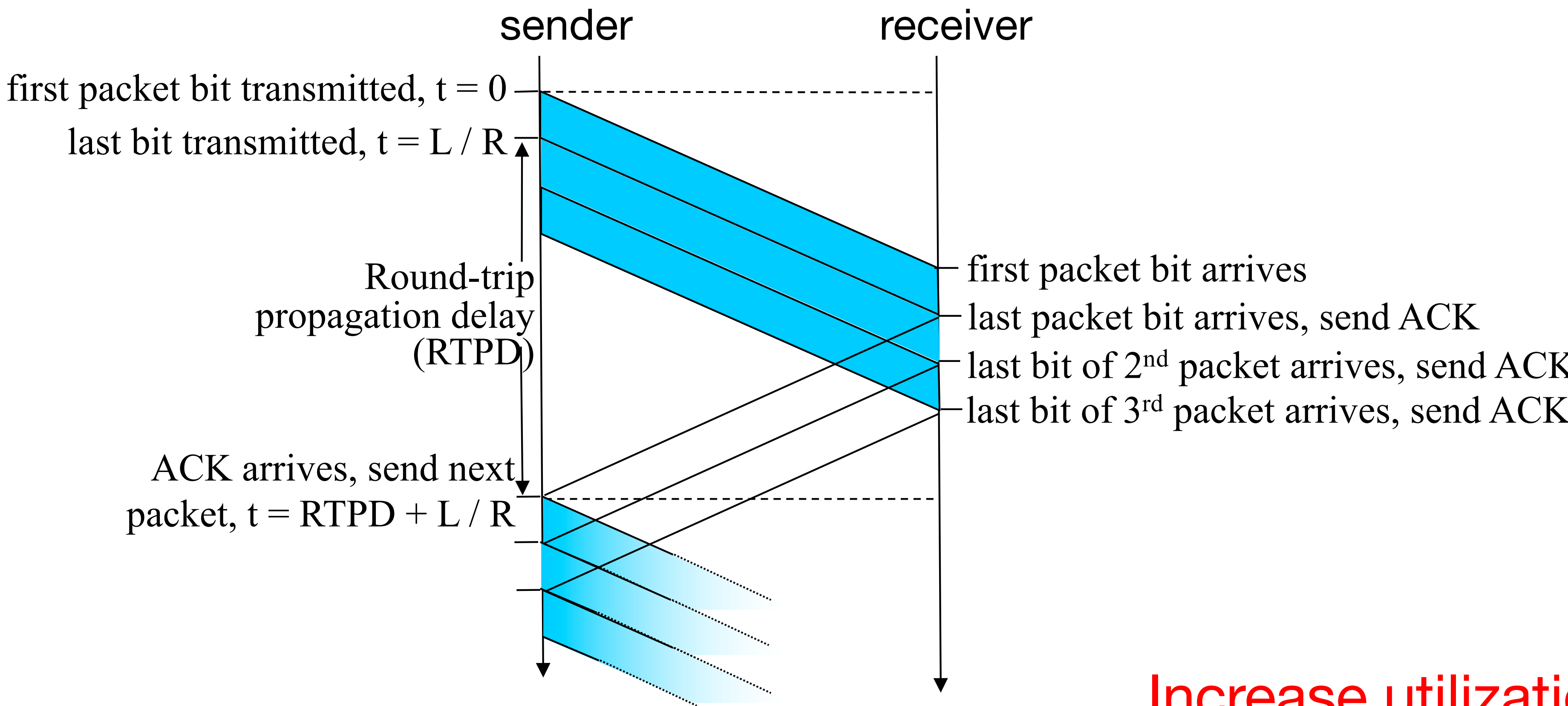
- Allowing multiple, yet-to-be-acknowledged, packets to be "in-flight"
  - Buffer in-flight packets at sender: if some packets get lost, need to retransmit
  - Buffer size determines how many packets can be in-flight: *flow control window*



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

(b) a pipelined protocol in operation

Pipelining increases network utilization



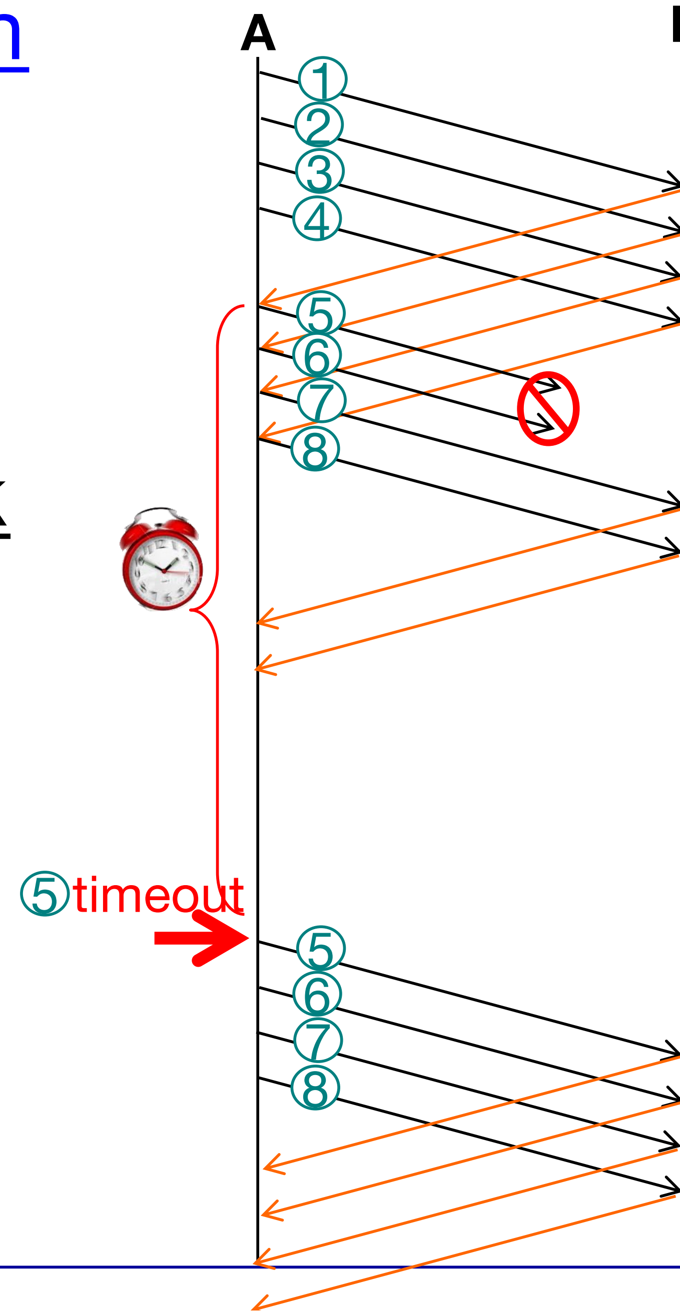
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 \times \frac{L}{R}}{RTPD + \frac{L}{R}} = \frac{0.024}{30.008} = 0.0008$$

What if some packet(s) get lost?

Go-Back-N (GBN) retransmission

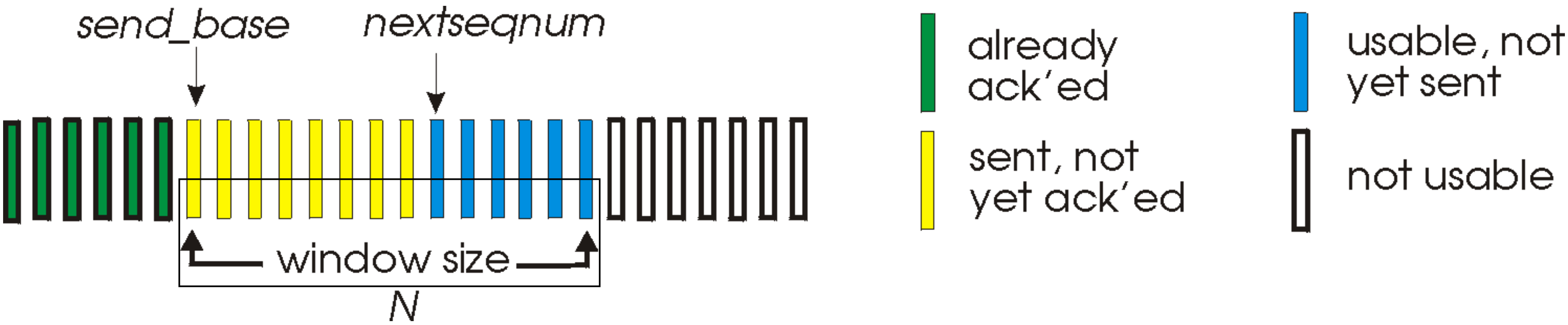
- Sender can send up to N unacknowledged packets
  - N = Flow control window size
- Receiver sends <u>cumulative ack</u>
  - acknowledge the last in-order arrived packet
- Sender sets timer for oldest unack'ed packet
  - when the timer expires, retransmit ***all*** the unack'ed packets within the window



Go-Back-N in detail

Sender:

- "window" of up to N *consecutive* unack'ed packets allowed

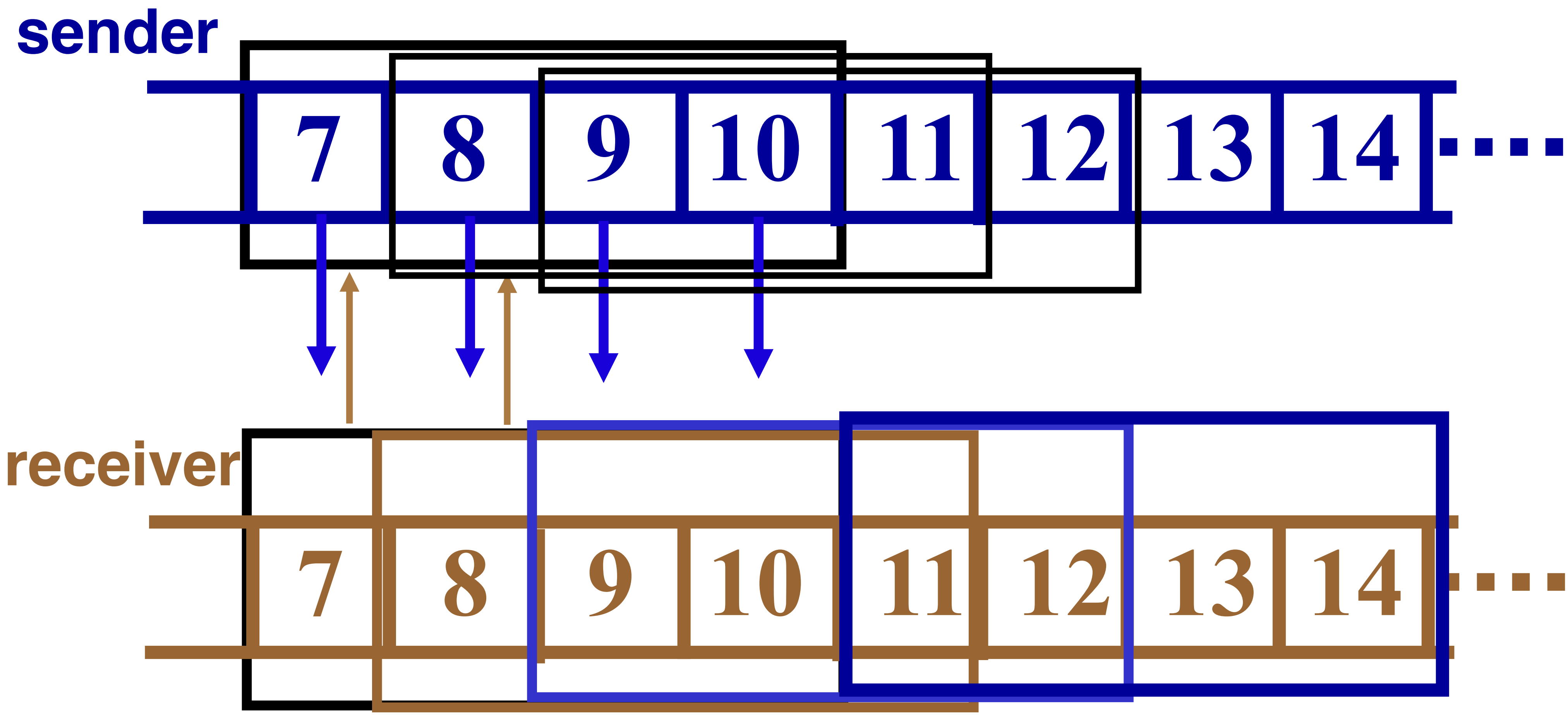


Receiver

- When receive a packet with seq= **`expectedseqnum`**, send an ACK
- When receive an out-of-order packet: discard
  - No need to buffer, since the sender will send all packets starting from the 1st missed packet
- Receiver only needs to keep track a single control variable: **`expectedseqnum`**

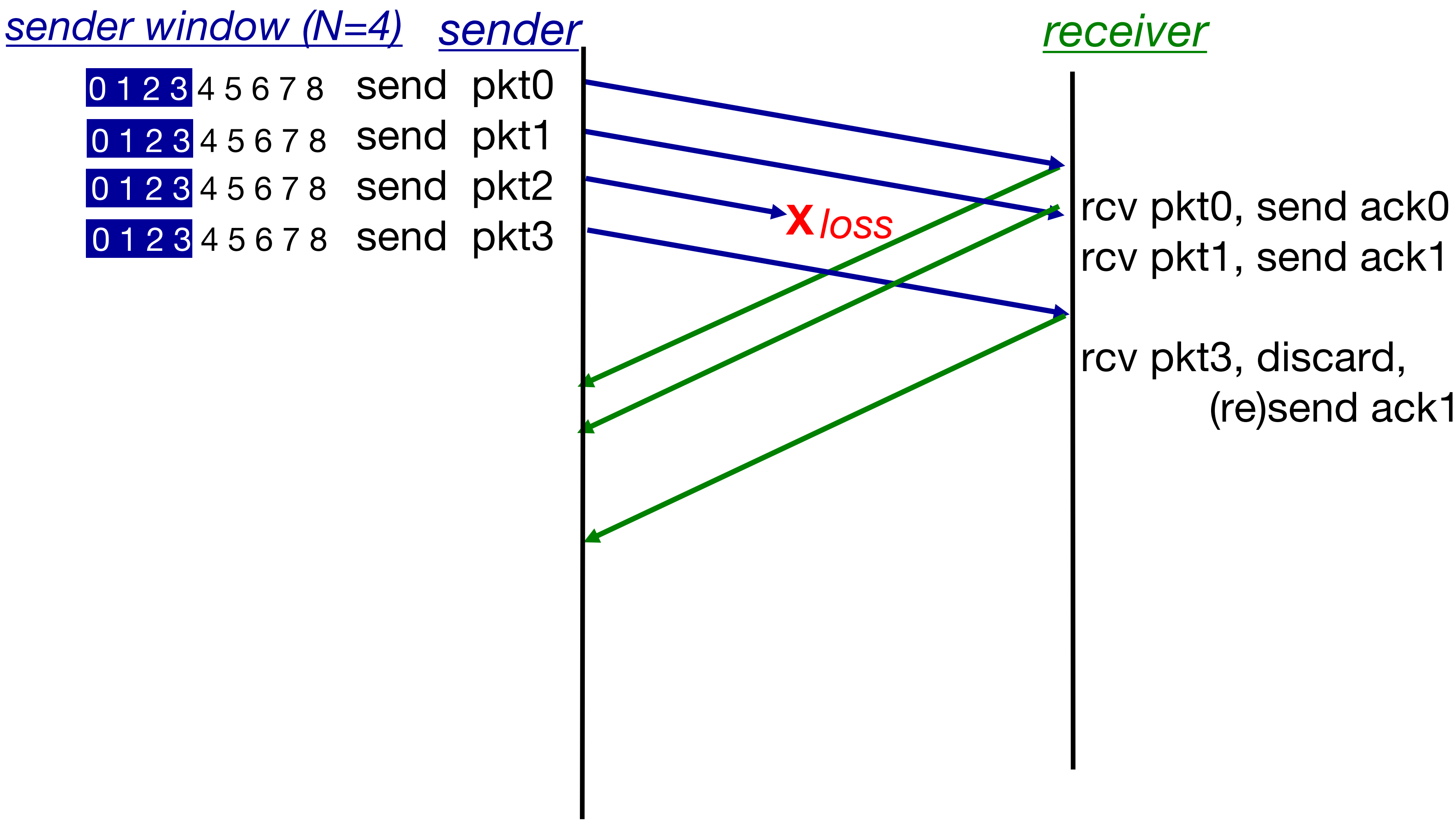
Flow control window at both ends for reliable data delivery

sender's window moves forward upon arrival of the ACK for the first un-ACKed packet

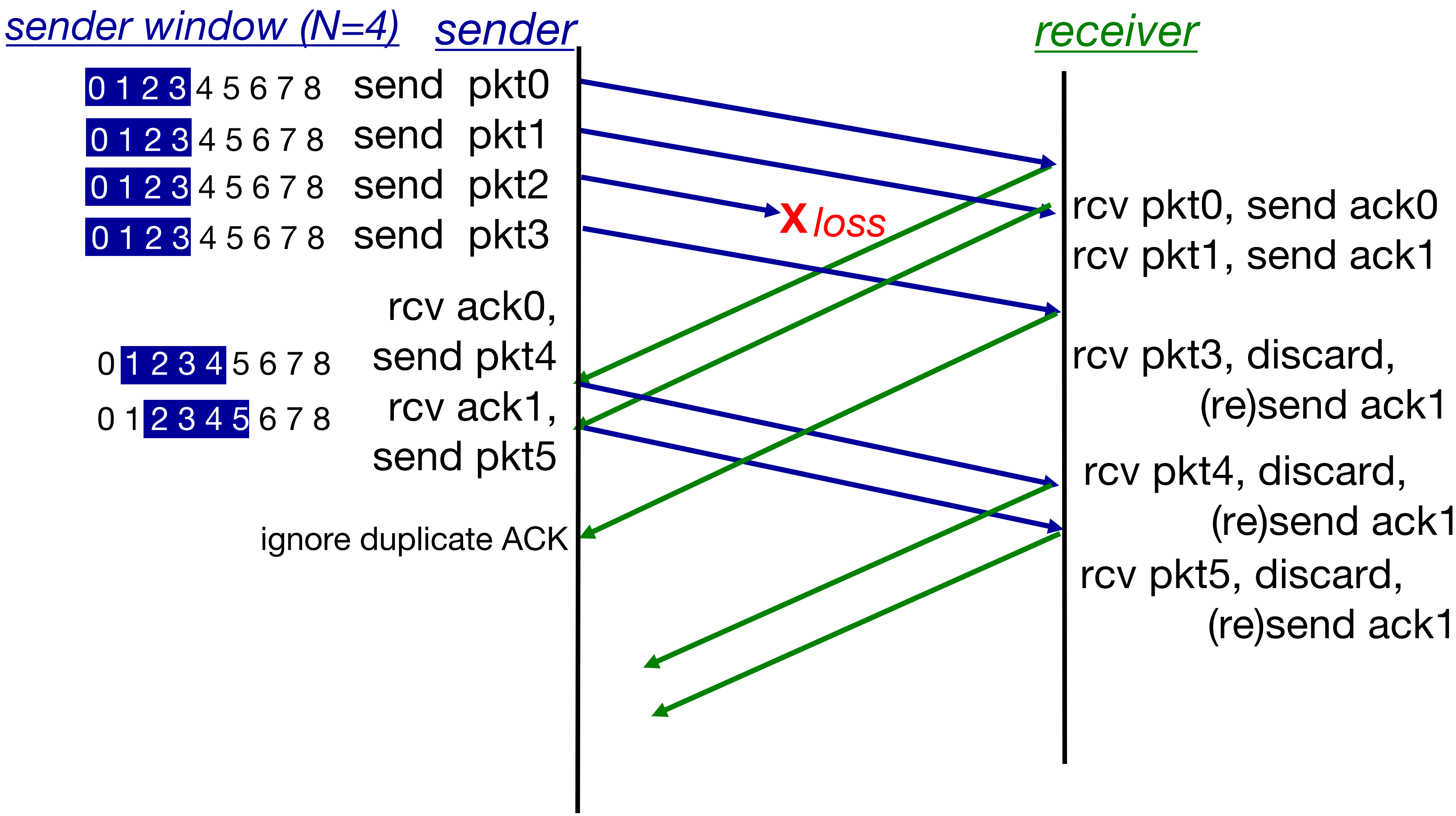


Receiver's window moves forward upon in-order arrival of each (error-free) data packet

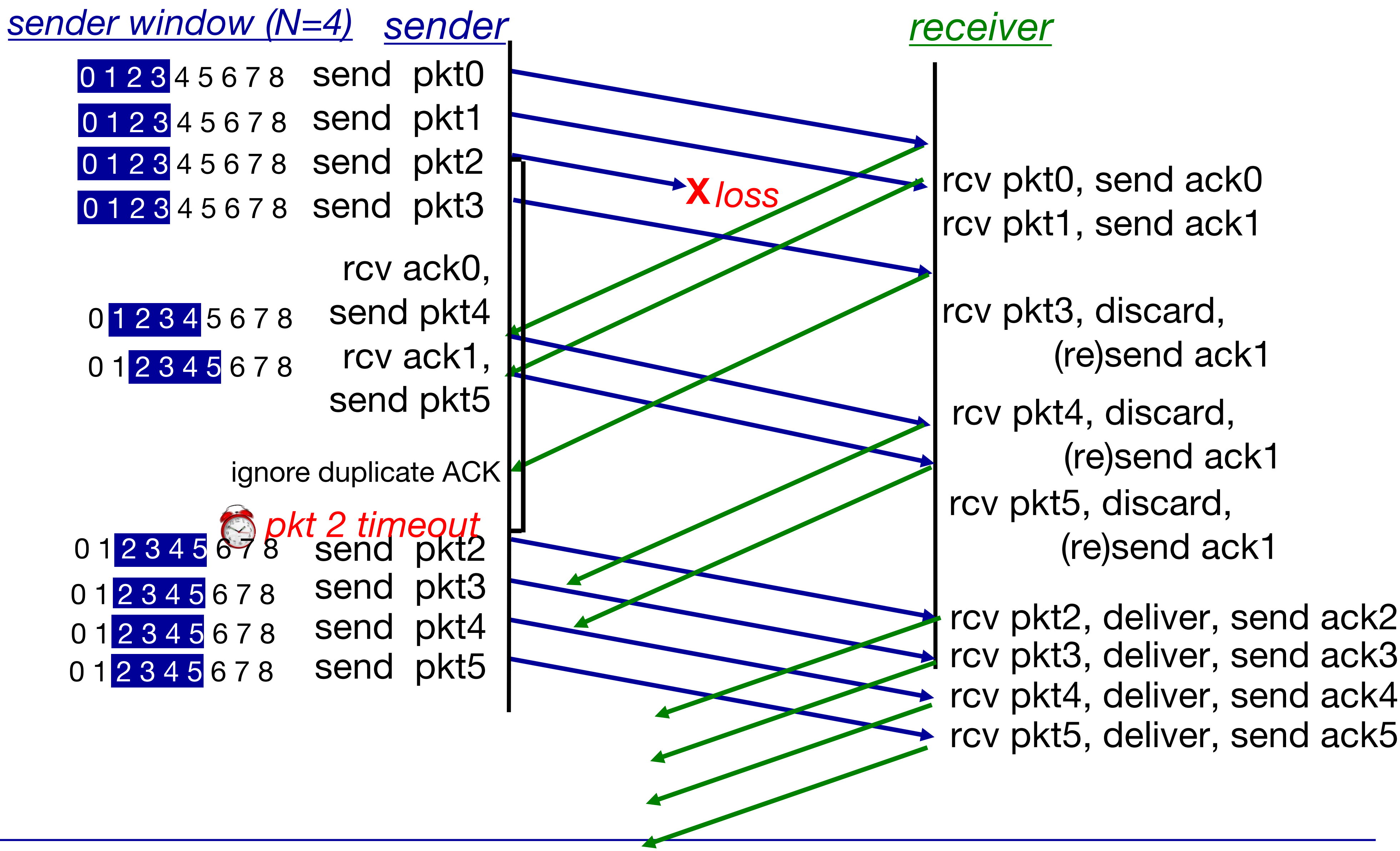
#Go-Back-N in action



#Go-Back-N in action



#Go-Back-N in action

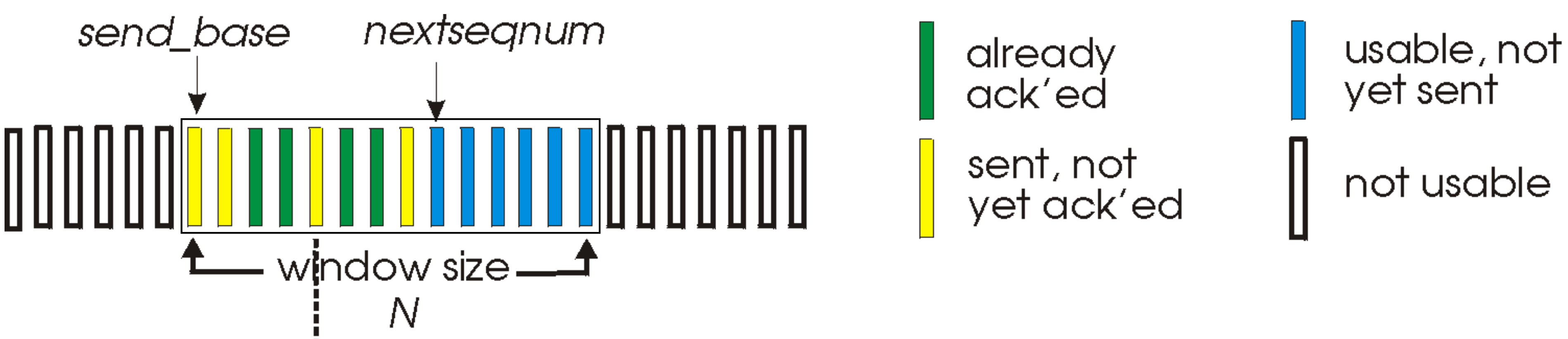


Can we do better than Go-back-N

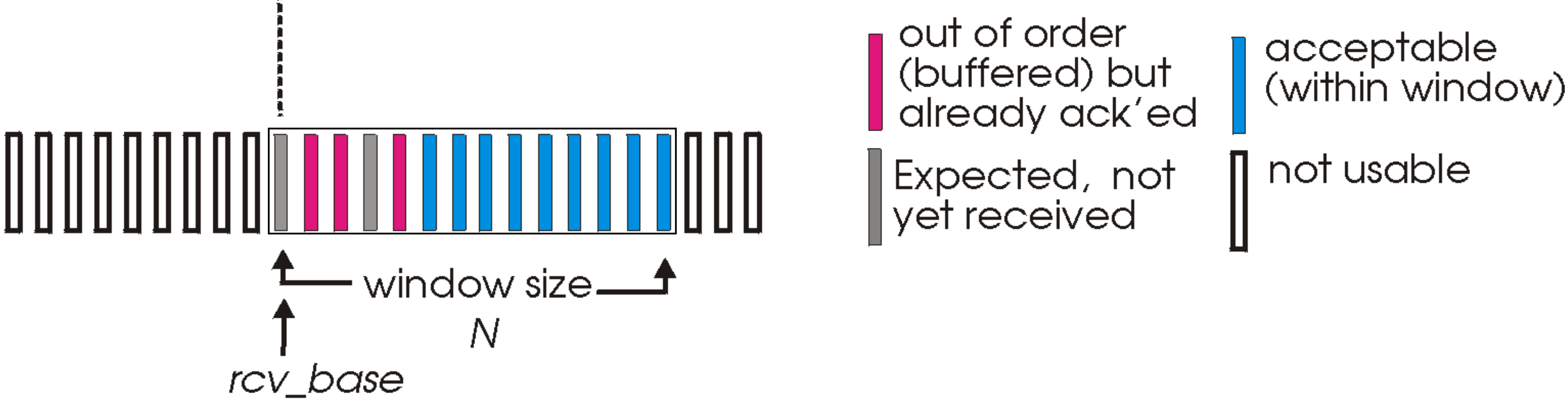
Selective repeat

- Sender can send up to N unacked packets
- Receiver acknowledges each correctly received packet
  - Receiver buffers packets that arrived out-of-order
  - When the missing packets received: Receiver can deliver data to upper layer
- Sender maintains **a** timer for the first unack'ed packet
  - When a timer expires, retransmit only that unack'ed packet
- Flow control window: works as before
  - Sender can send N consecutive packets
  - N controls the *number* of packets between
    [the first unACKed packet, the last one that can be sent]

Selective Repeat: sender, receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective Repeat

Sender

data from upper layer:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in

[sendbase, sendbase+N]:

- 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*

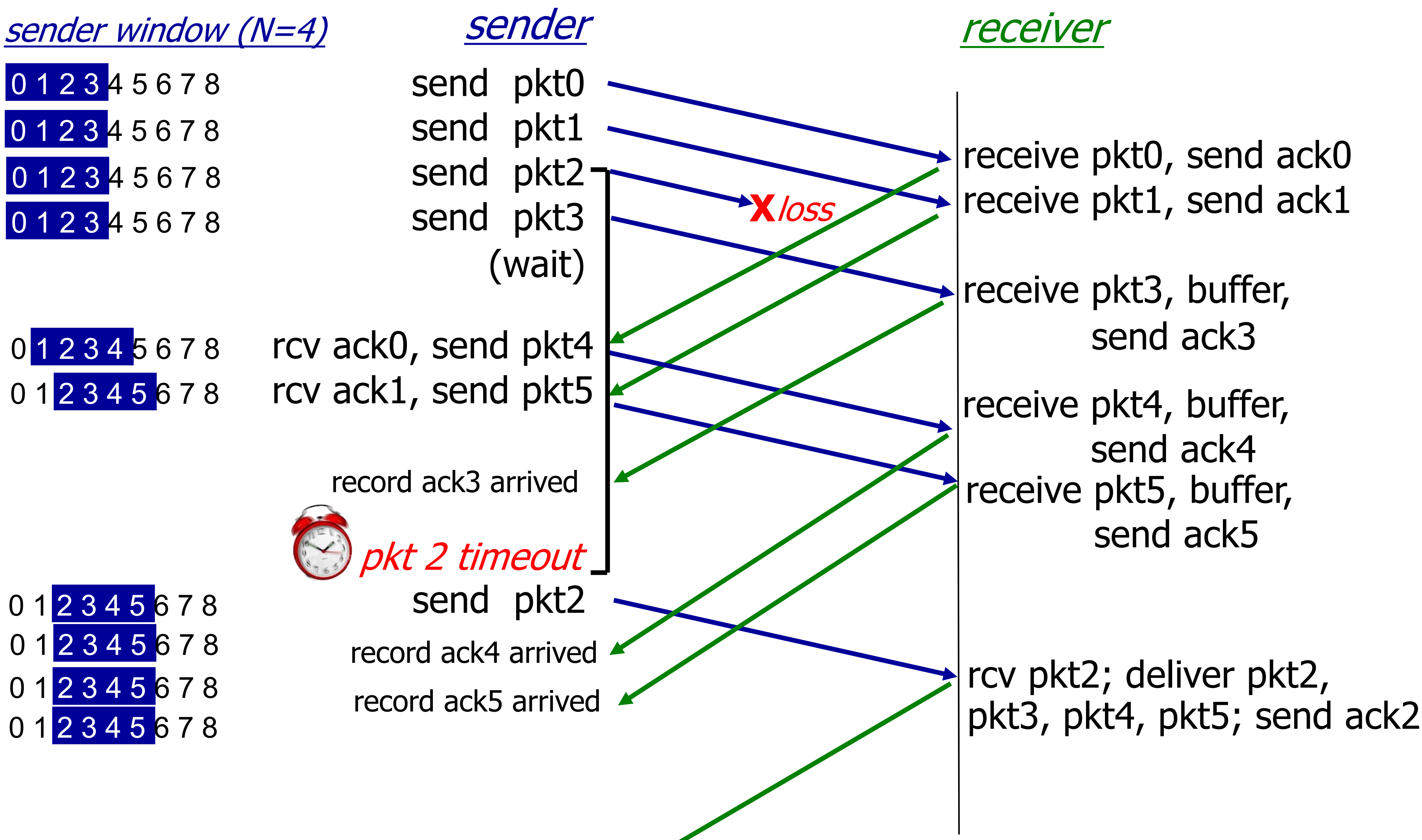
[rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

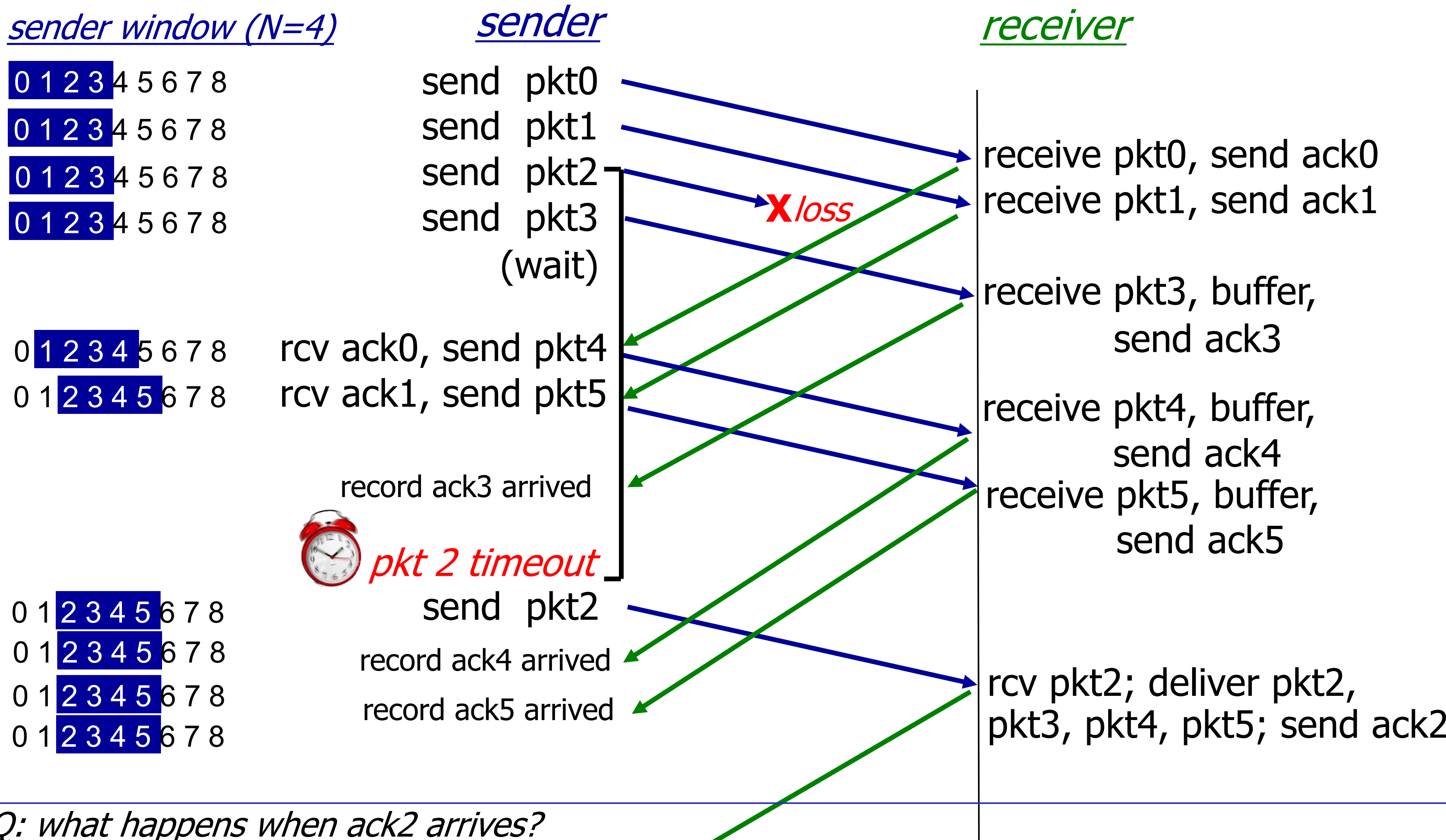
- ignore

Selective repeat in action



Q1: what happens when ack2 arrives?

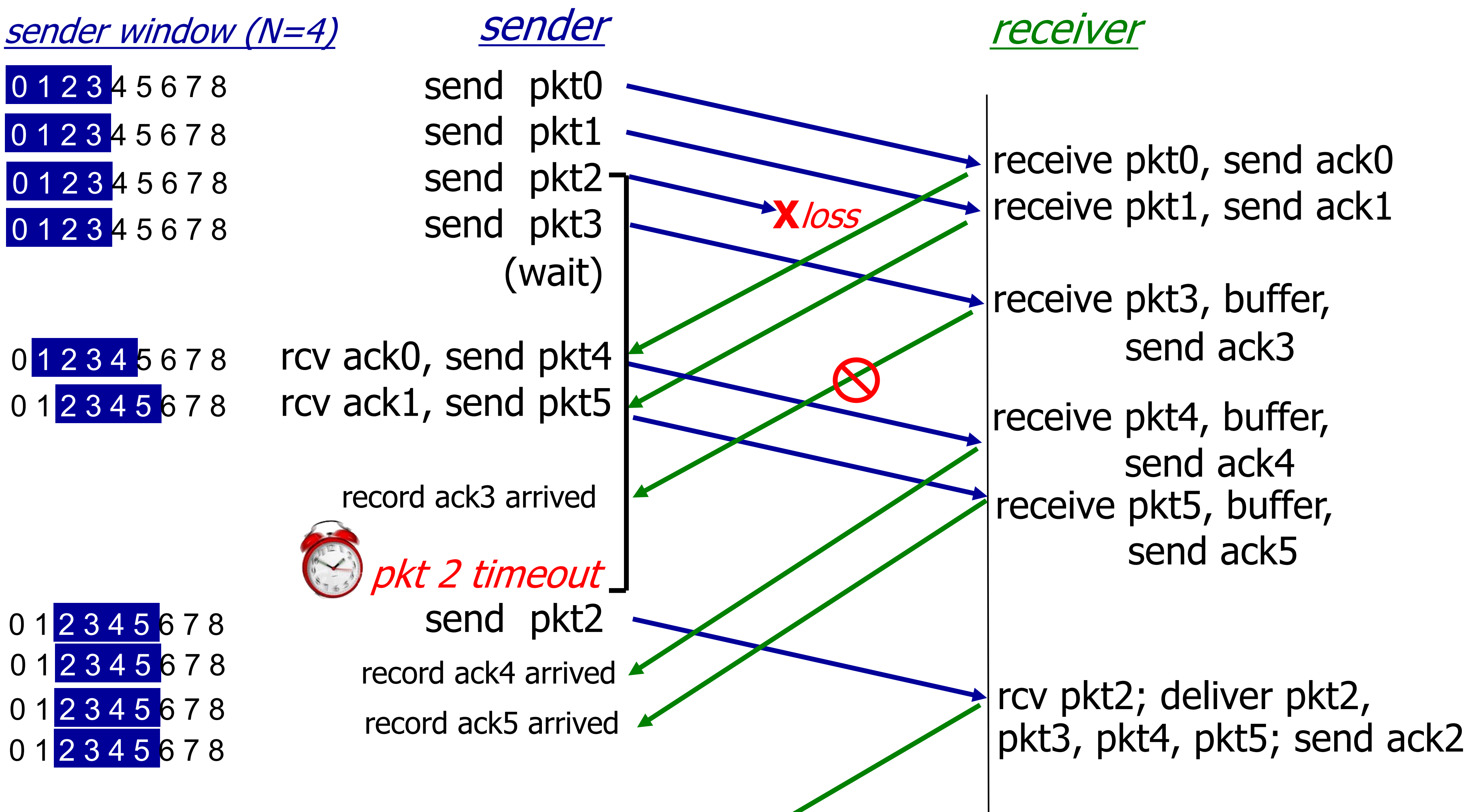
Selective repeat in action



Q: what happens when ack2 arrives?

- If ACK3 arrived: move flow control window to the right by 4 positions

Selective repeat in action



Q: what happens when ack2 arrives?

- If ACK3 arrived: move flow control window to the right by 4 positions
- If ACK3 lost: move flow control window to the right by 1

Q2: what if some ACK is lost?

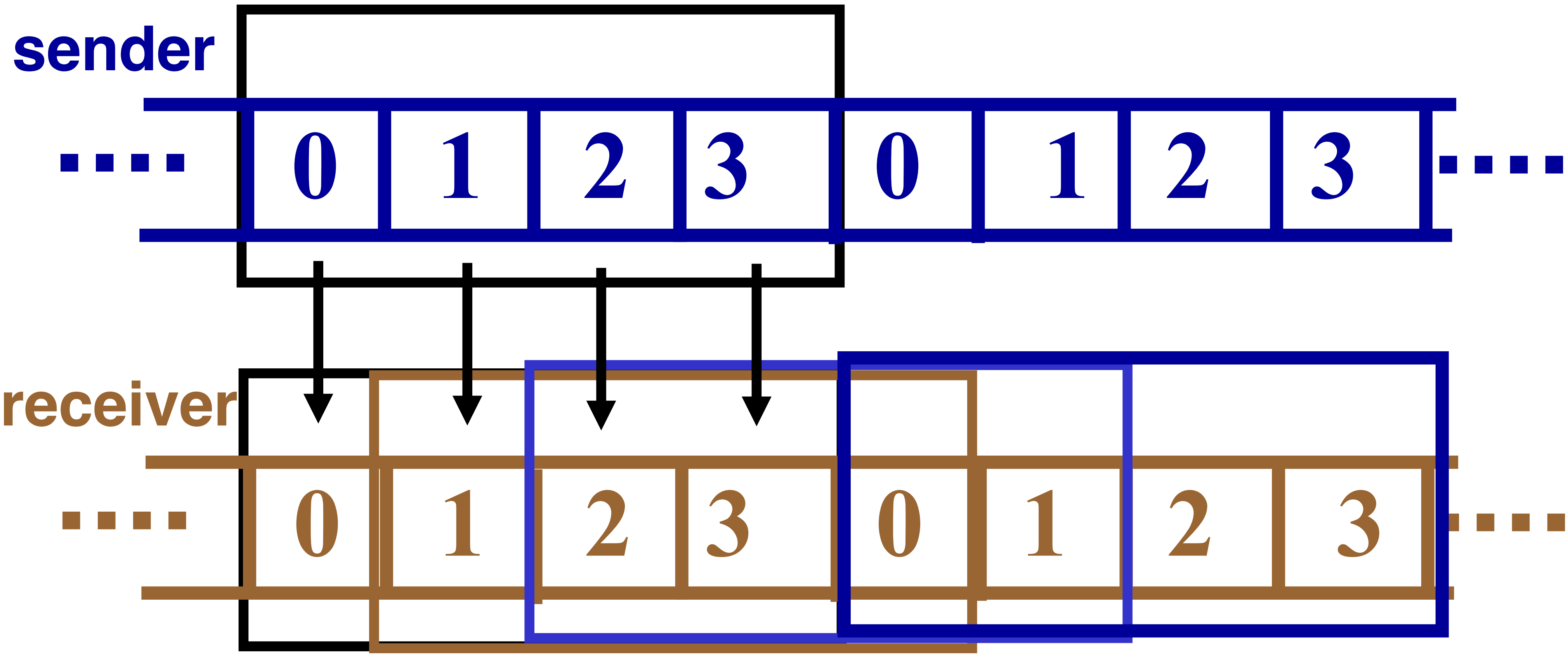
Summary of Selective Repeat

- Receiver informs the sender of received packets that arrive out of order
  - This avoids unnecessary retransmission of already received packets, when they arrived out of order
- Receiver also ACKs the previous window
  - This avoids the sender window freezes
- Improvement: receiver sends cumulative ACK <u>in addition</u> to Selective Repeat
  - No need to wait for timeout, move the sender window forward ASAP

The relationship between window size & sequence number, window size & throughput

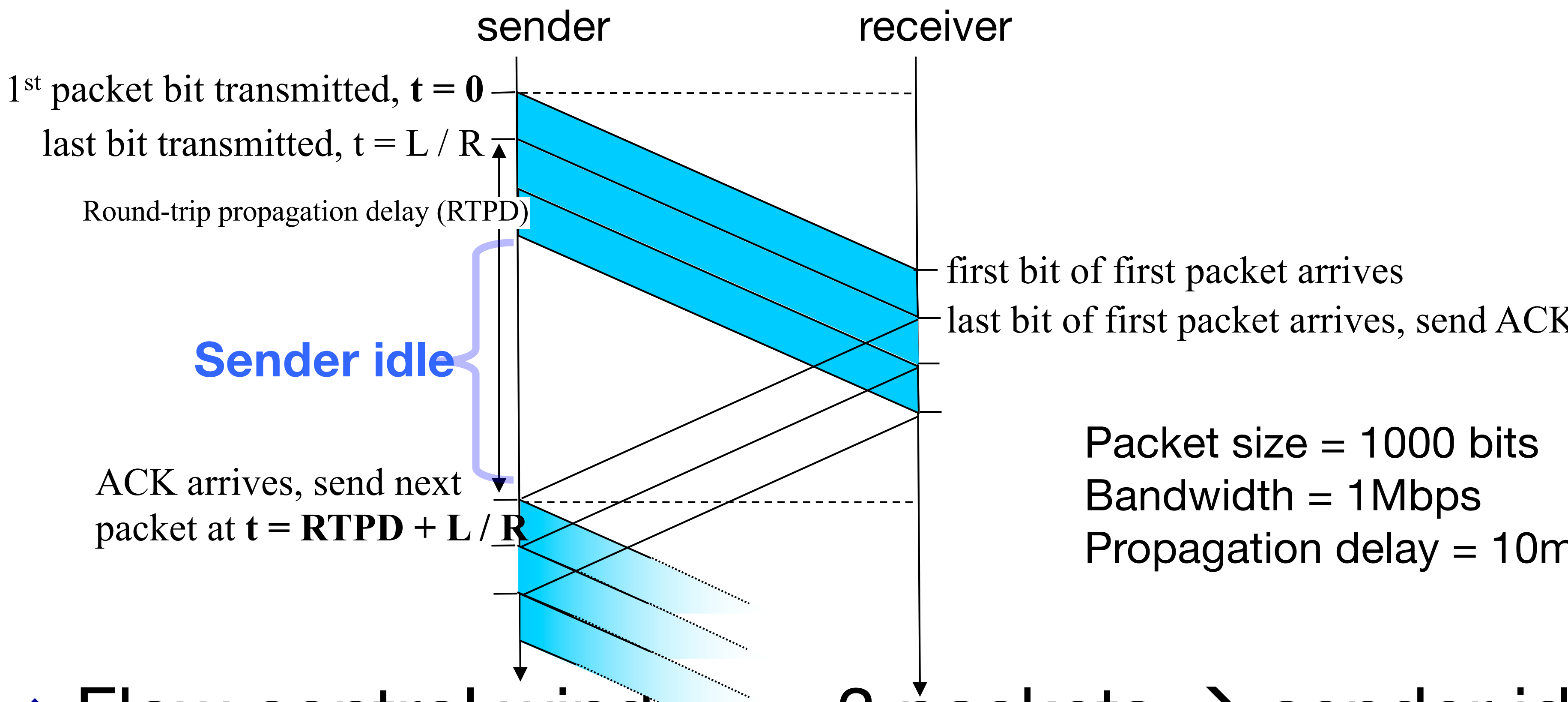
Relationship between flow control window size & seq# range

Example: **Window size = 4, is 2-bit seq# enough?**



window-size ≤ (Max. seq# + 1) / 2

Relationship between flow control window size and throughput: an example



Packet size = 1000 bits
Bandwidth = 1Mbps
Propagation delay = 10msec

- Flow control window = 3 packets → sender idle from time to time
- What is the effective throughput (without packet loss)?
- To keep sender busy all the time: window size = ?

- Window = 3, Round trip propagation delay = 20msec,

  effective throughput = 1Mbps x 3/21

- To achieve full utilization (=sender busy transmtting all the time): Window = 21 packets
  - How many bits would be needed for the seq# field?

- Generally speaking: *in the absence* of packet losses,
  - When Window / RTT < bandwidth,

    Throughput = window size / RTT
  - When Window / RTT ≥ bandwidth,

    Throughput = bandwidth