南京大學
NANJING UNIVERSITY

Computer Networks

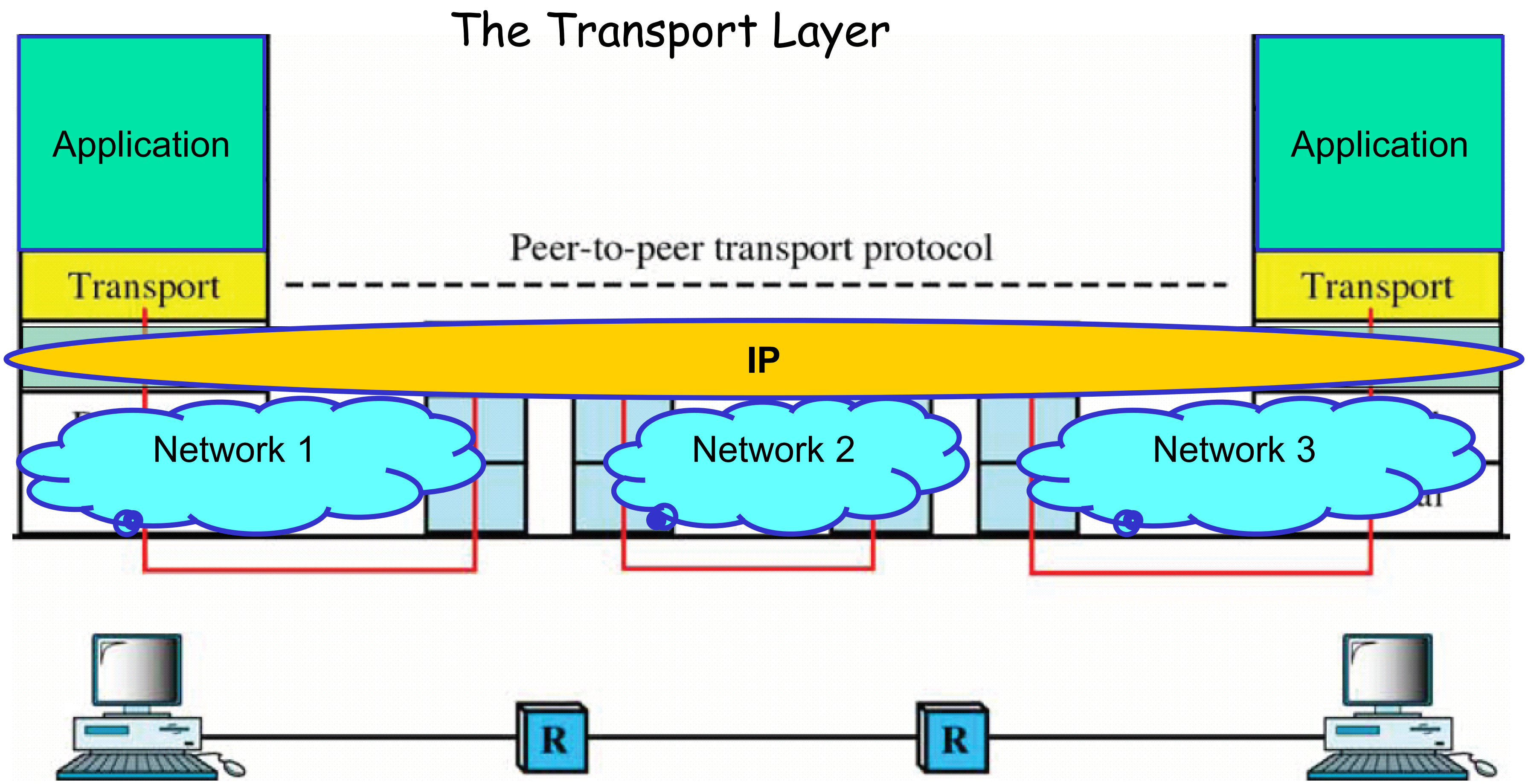
Wenzhong Li, Chen Tian

Nanjing University

Material with thanks to James F. Kurose, Mosharaf Chowdhury, and other colleagues.

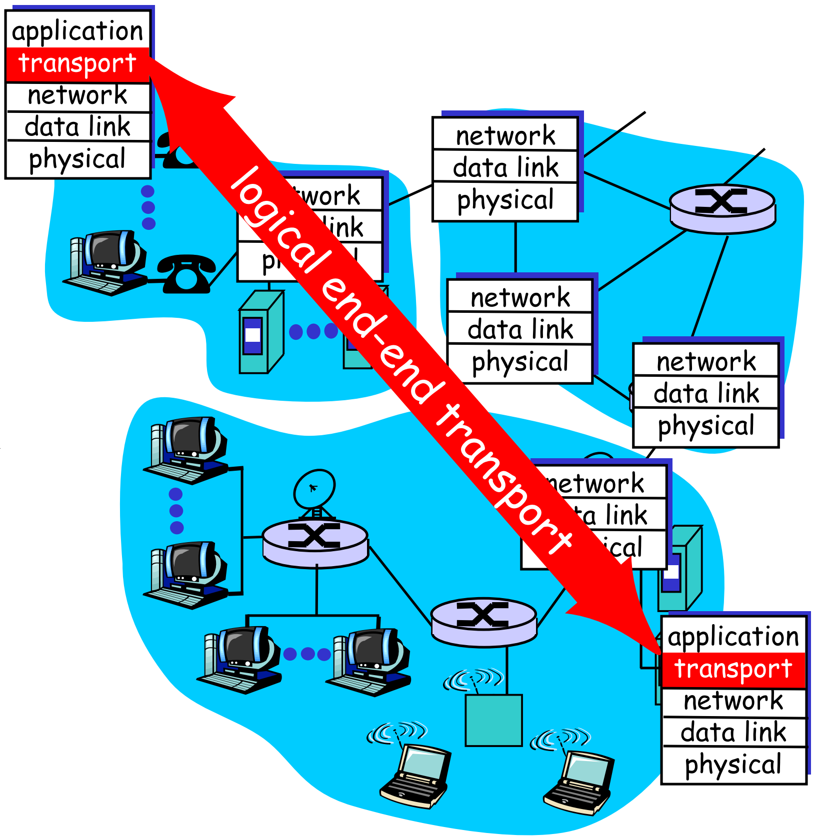
Transport layer basics

Transport Services and Mechanisms



Internet Transport Services

- 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
  - Receive side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP



Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (multiplexing/demultiplexing)
- IP provides a weak service model (best-effort)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
  - No guidance on how much traffic to send and when
  - Dealing with this is tedious for application developers

Multiplexing & demultiplexing

- Multiplexing (Mux)
  - Gather and combining data chunks at the source host from different applications and delivering to the network layer
- Demultiplexing (Demux)
  - Delivering correct data to corresponding sockets from multiplexed a stream

Role of the transport layer

- Communication between processes
  - Mux and demux from/to application processes
  - Implemented using *ports*

Role of the transport layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
  - Reliable, in-order data delivery
  - Well-paced data delivery
    - Too fast may overwhelm the network
    - Too slow is not efficient

Role of the transport layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
  - Also SCTP, MPTCP, SST, RDP, DCCP, …

Role of the transport layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist transport protocol
  - Only provides mux/demux capabilities

Role of the transport layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist transport protocol
- TCP offers a reliable, in-order, byte stream abstraction
  - With congestion control, but w/o performance guarantees (delay, b/w, etc.)

Applications and sockets

- Socket: software abstraction for an application process to exchange network messages with the (transport layer in the) operating system

- Transport layer addressing
  - <HostIP, Port>, called a socket

- Two important types of sockets
  - UDP socket: TYPE is SOCK_DGRAM
  - TCP socket: TYPE is SOCK_STREAM

Ports

- 16-bit numbers that help distinguishing apps
  - Packets carry src/dst port no in transport header
  - Well-known (0-1023) and ephemeral ports
- OS stores mapping between sockets and ports
  - Port in packets and sockets in OS
  - For UDP ports (SOCK_DGRAM)
    - OS stores (local port, local IP address) ←→ socket
  - For TCP ports (SOCK_STREAM)
    - OS stores (local port, local IP, remote port, remote IP) ←→ socket

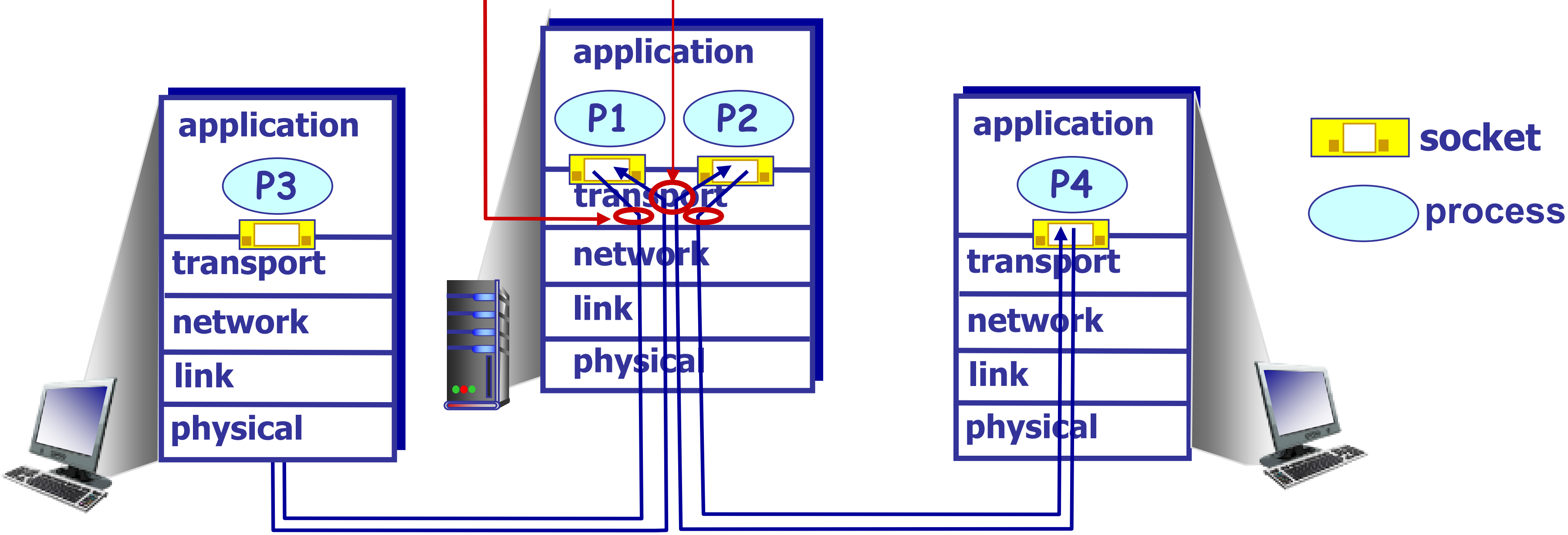
Multiplexing/demultiplexing

multiplexing at sender:

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

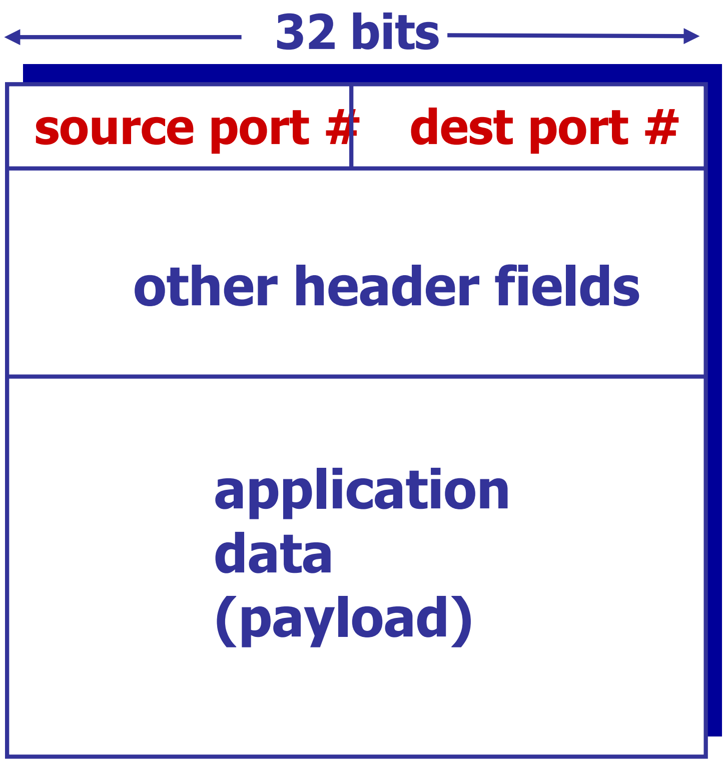
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- 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 & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

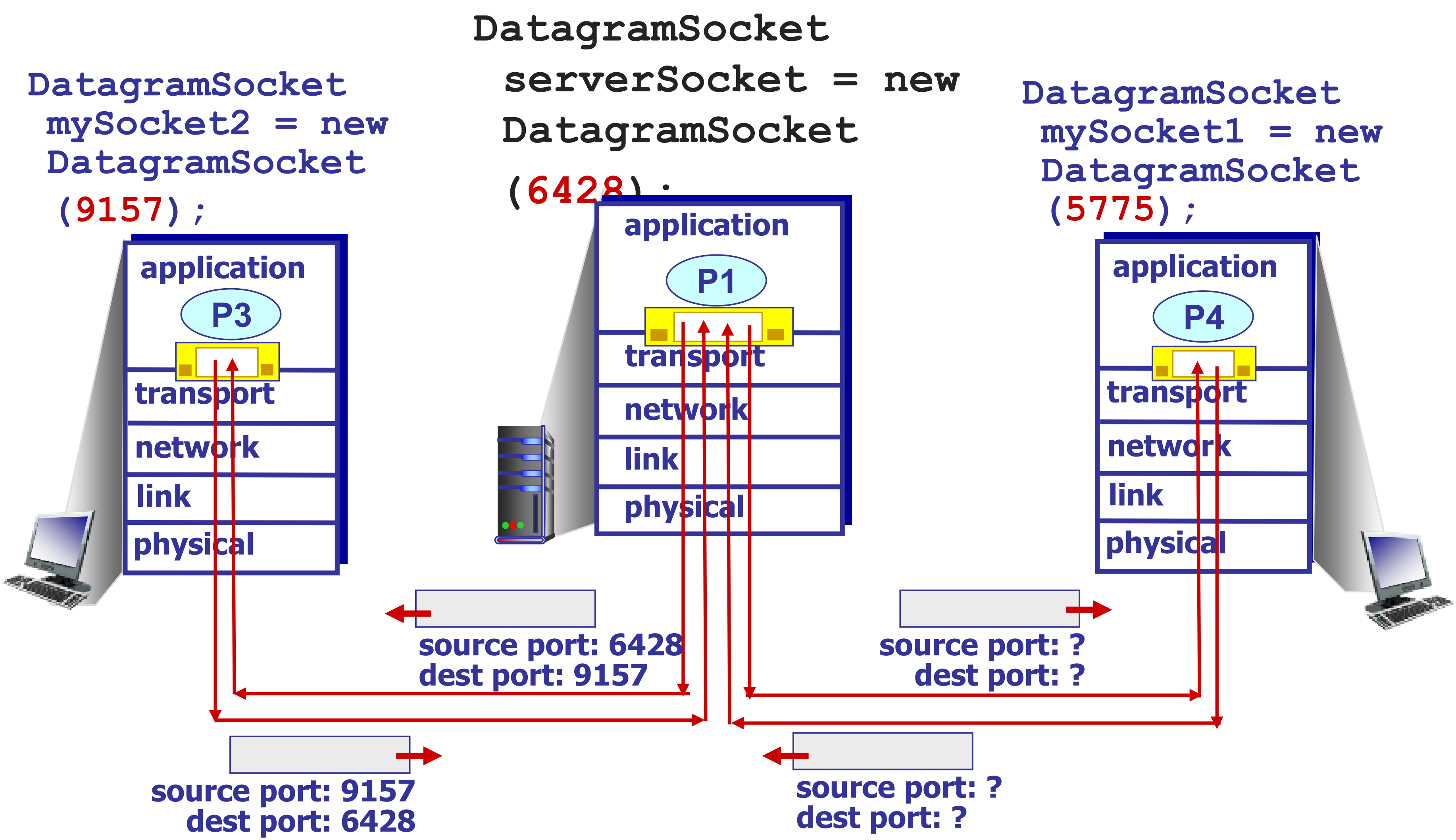
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- ***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 #

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

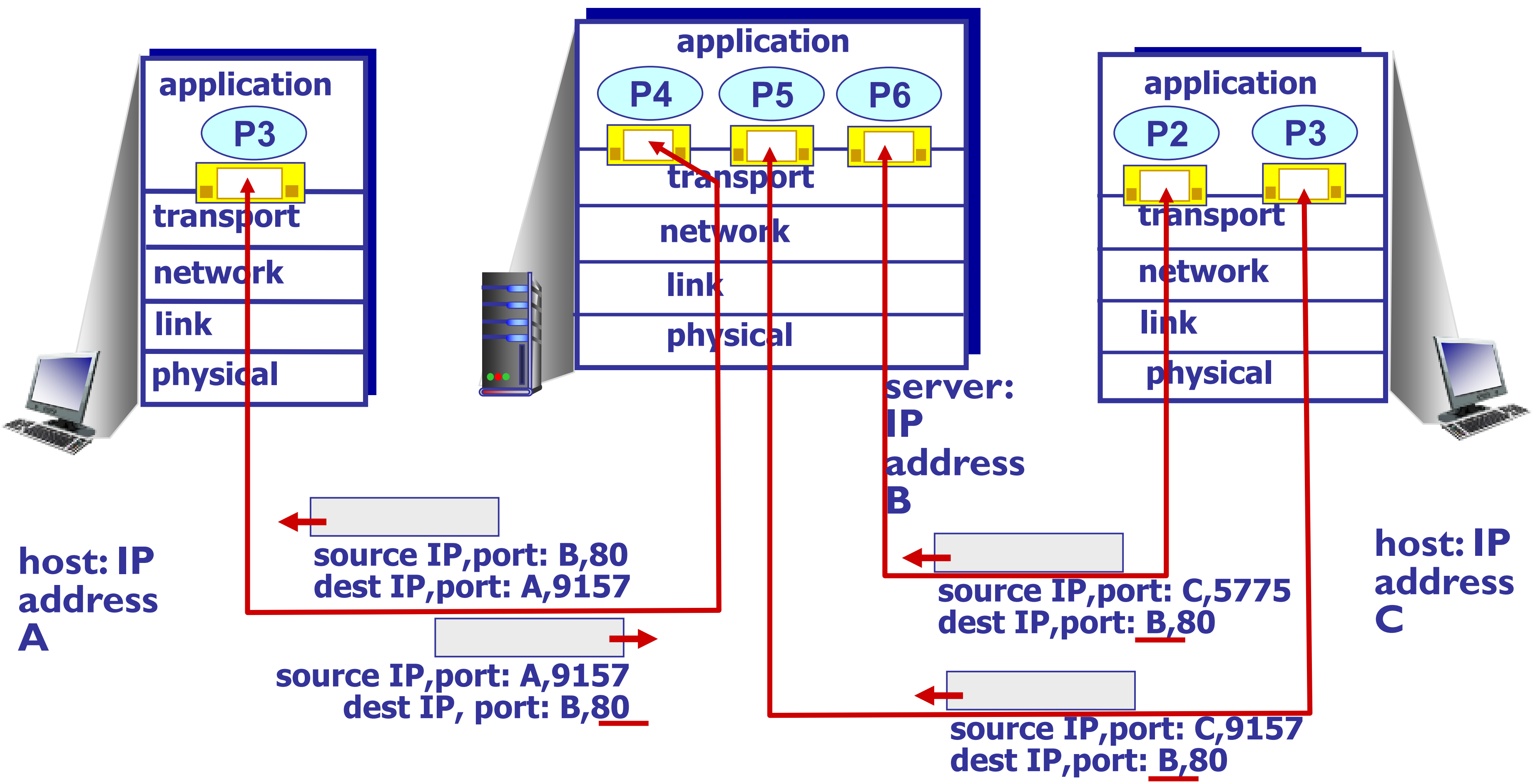


Connection-oriented demux

- 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

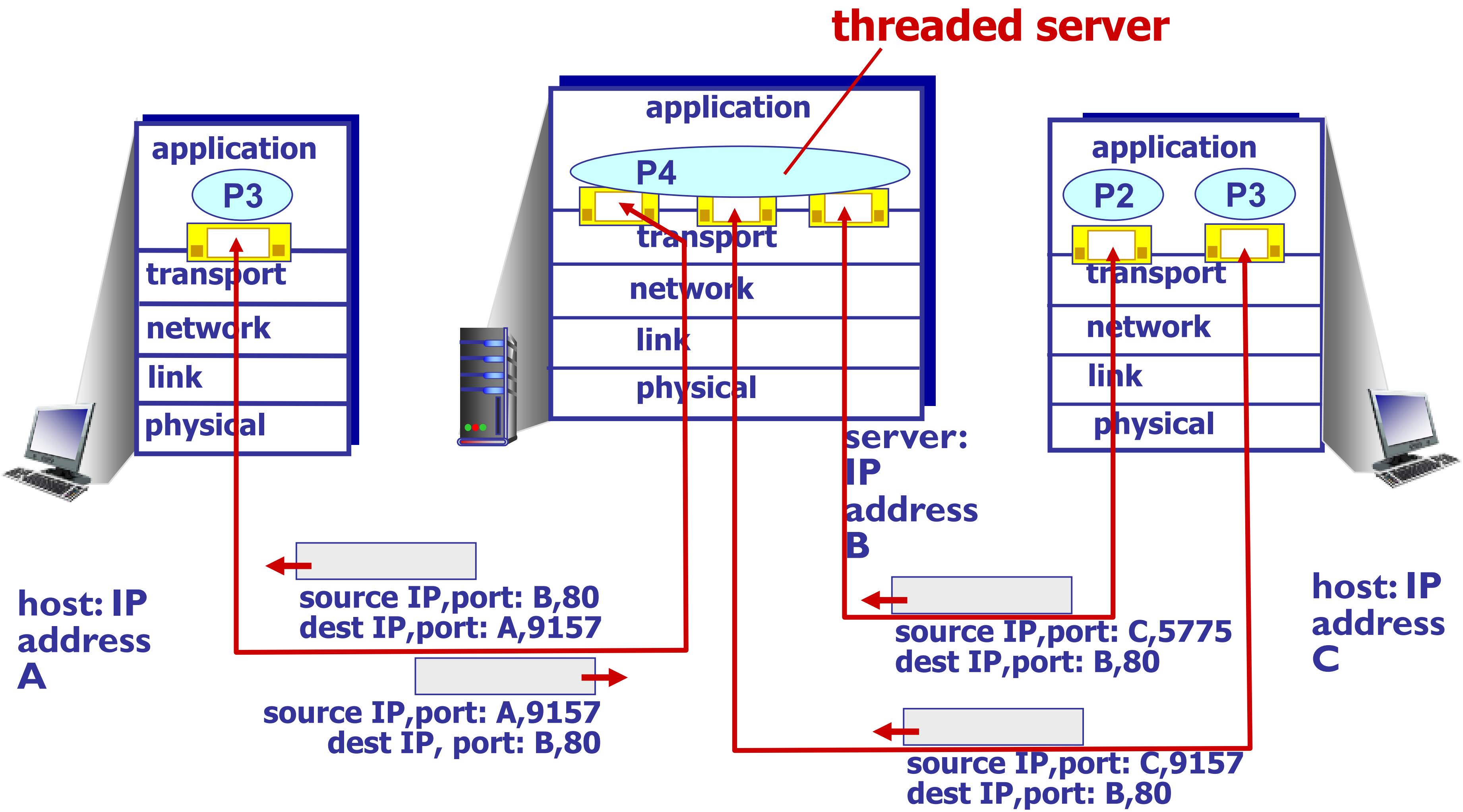
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



Design of reliable transport

Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (best-effort)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
  - No guidance on how much traffic to send and when
  - Dealing with this is tedious for application developers

Reliable transport

- In a perfect world, reliable transport is easy

@Sender

- Send packets

@Receiver

- Wait for packets

Reliable transport

- In a perfect world, reliable transport is easy
- All the bad things best-effort can do
  - A packet is corrupted (bit errors)
  - A packet is lost (*why?*)
  - A packet is delayed (*why?*)
  - Packets are reordered (*why?*)
  - A packet is duplicated (*why?*)

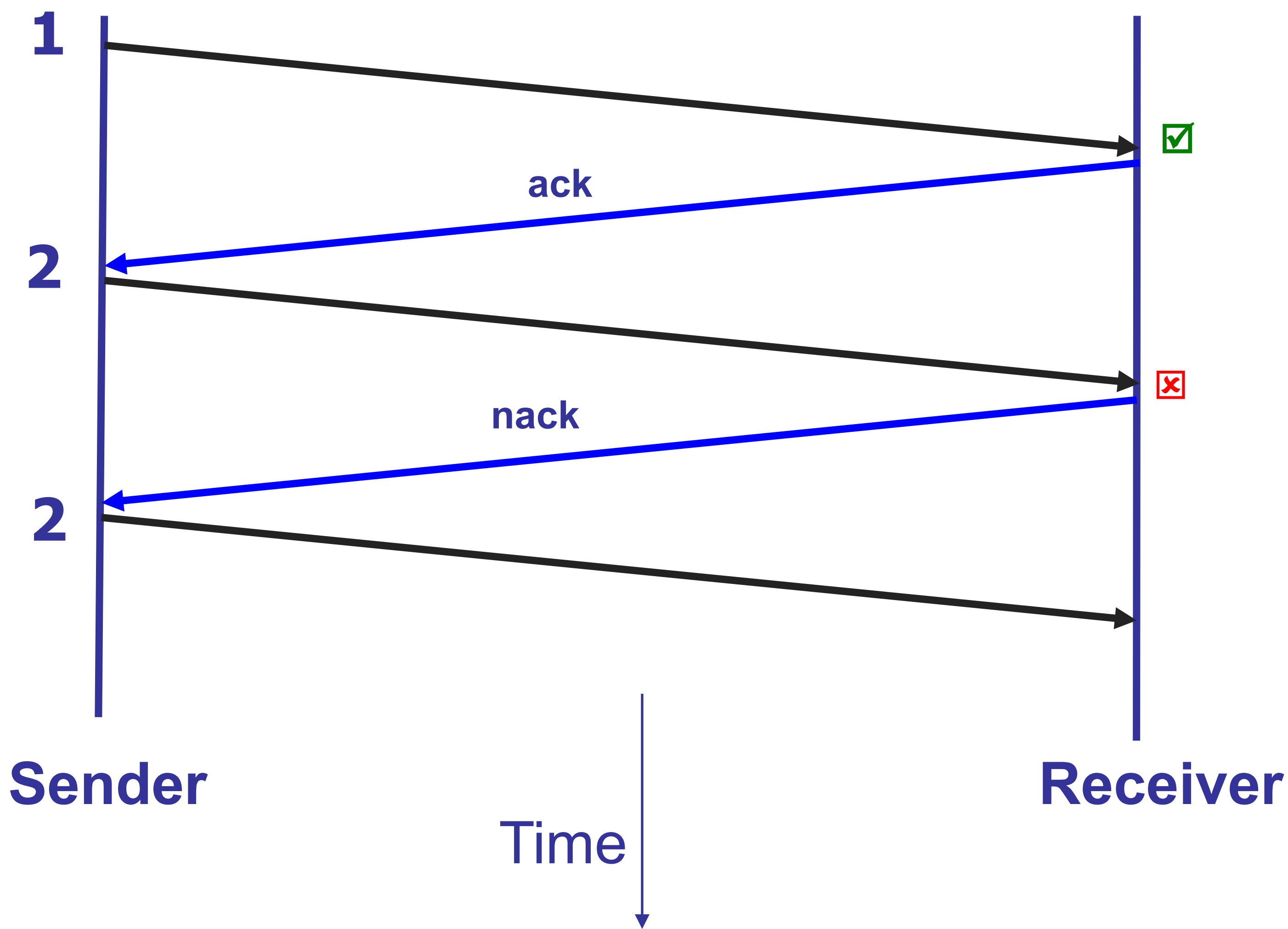
Reliable transport

- Mechanisms for coping with bad events
  - Checksums: to detect corruption
  - ACKs: receiver tells sender that it received packet
  - NACK: receiver tells sender it did not receive packet
  - Sequence numbers: a way to identify packets
  - Retransmissions: sender resends packets
  - Timeouts: a way of deciding when to resend packets
  - *Forward error correction: a way to mask errors without retransmission*
  - *Network encoding: an efficient way to repair errors*

Dealing with packet corruption

❖ *the* question: how to recover from errors:

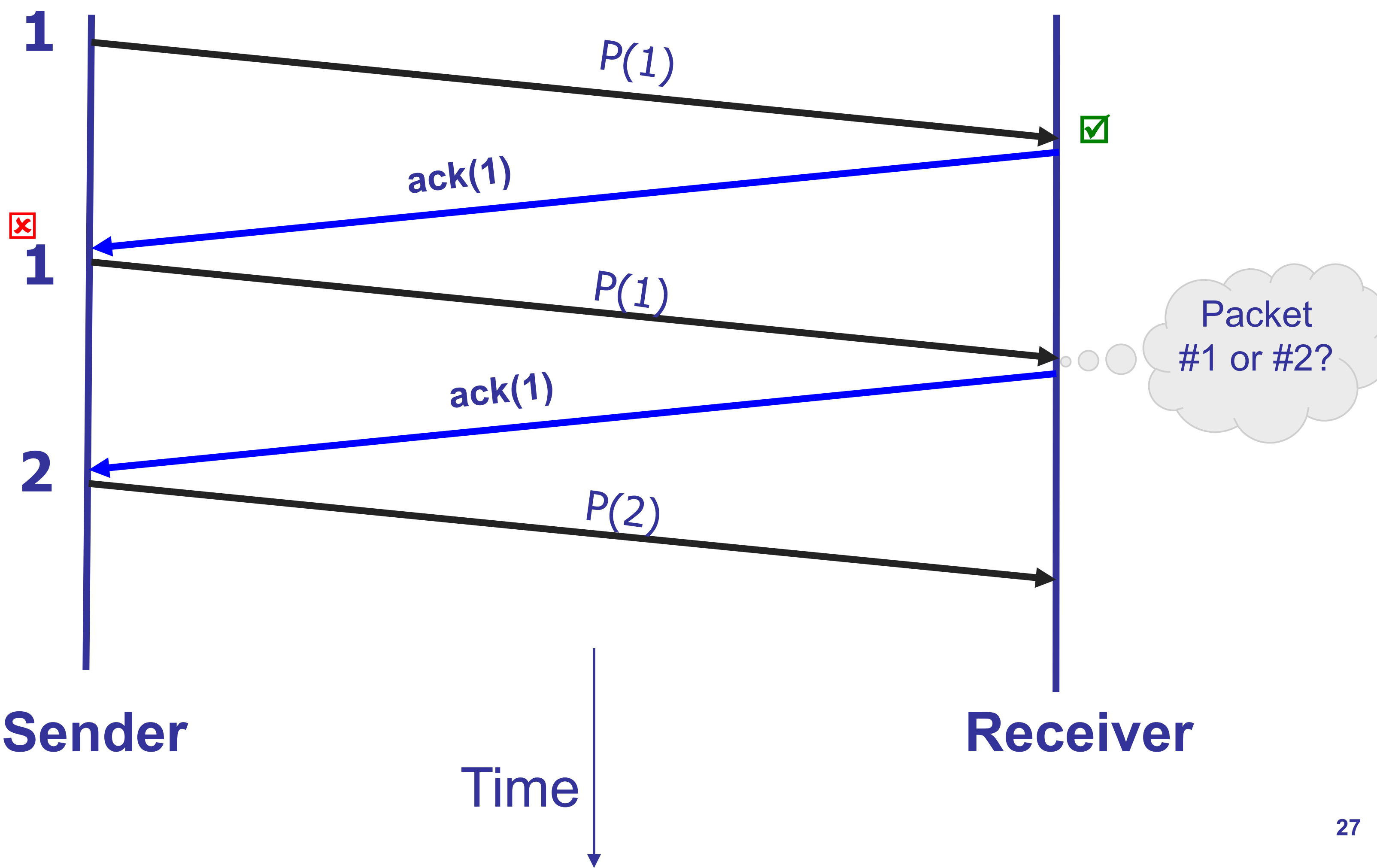
- *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
- *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
- sender retransmits pkt on receipt of NAK



Dealing with packet corruption

What if the ACK/NACK is corrupted?

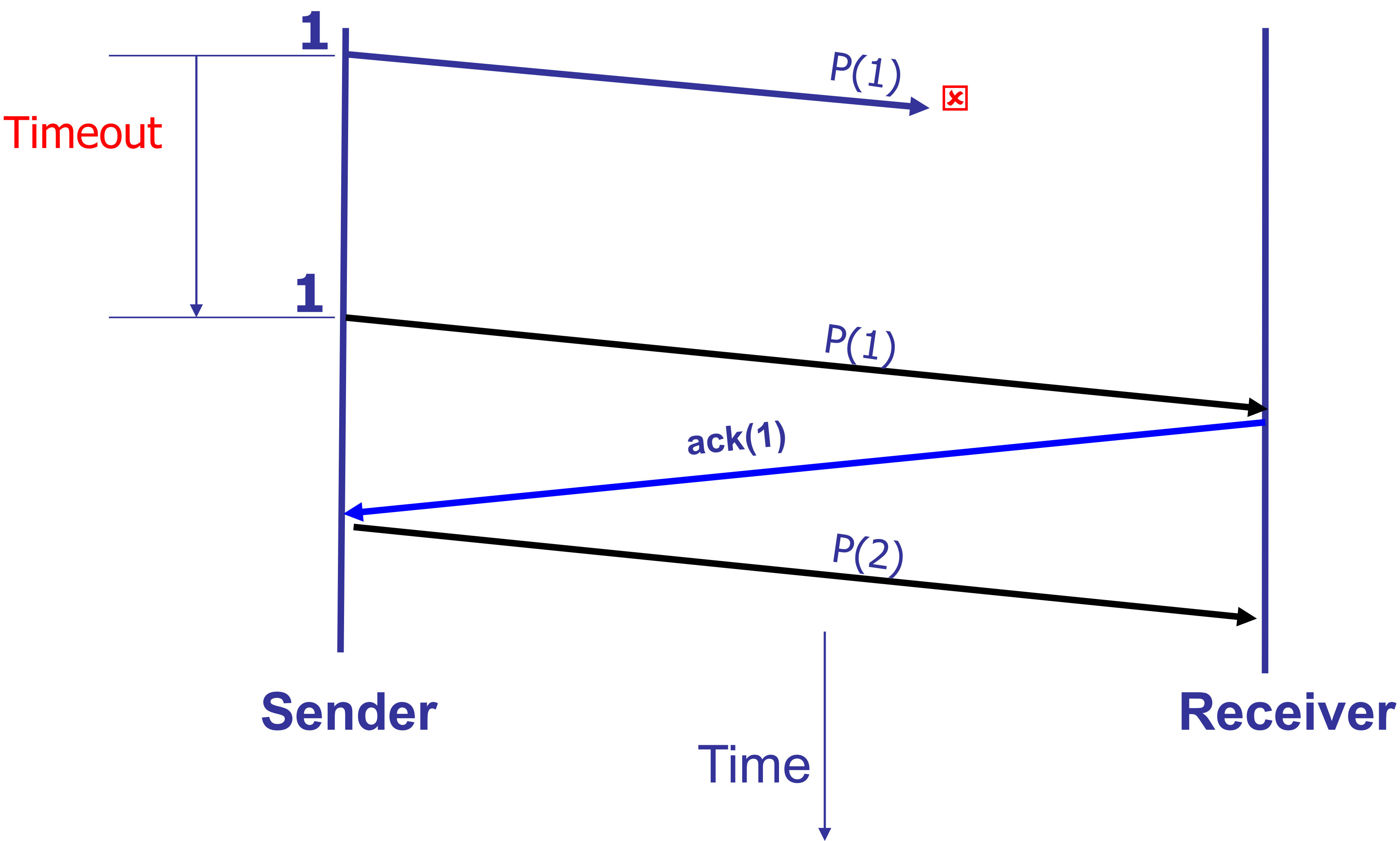
Data and ACK packets carry <u>sequence numbers</u>



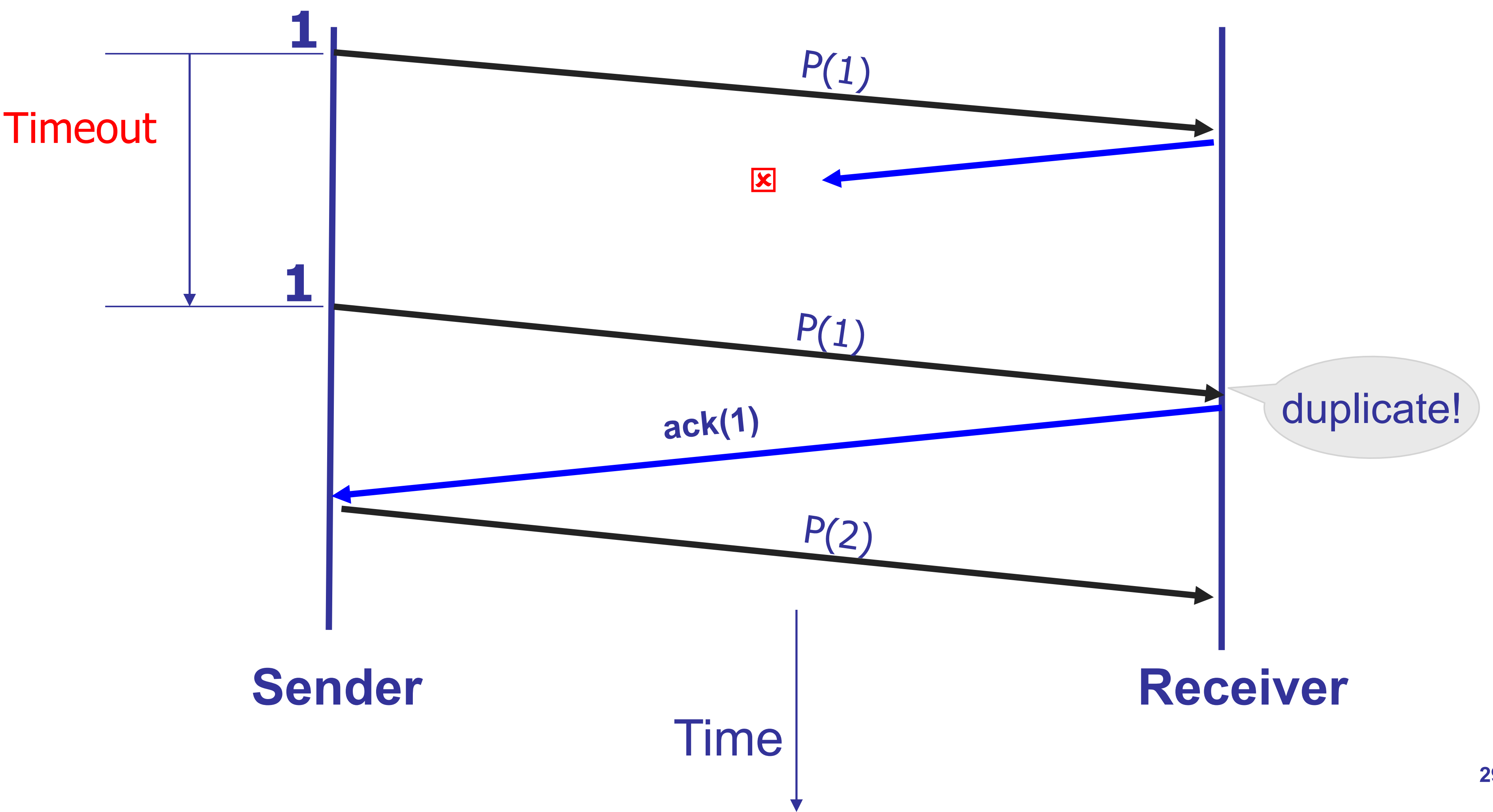
Dealing with packet loss

Timer-driven loss detection

Set timer when packet is sent; retransmit on timeout

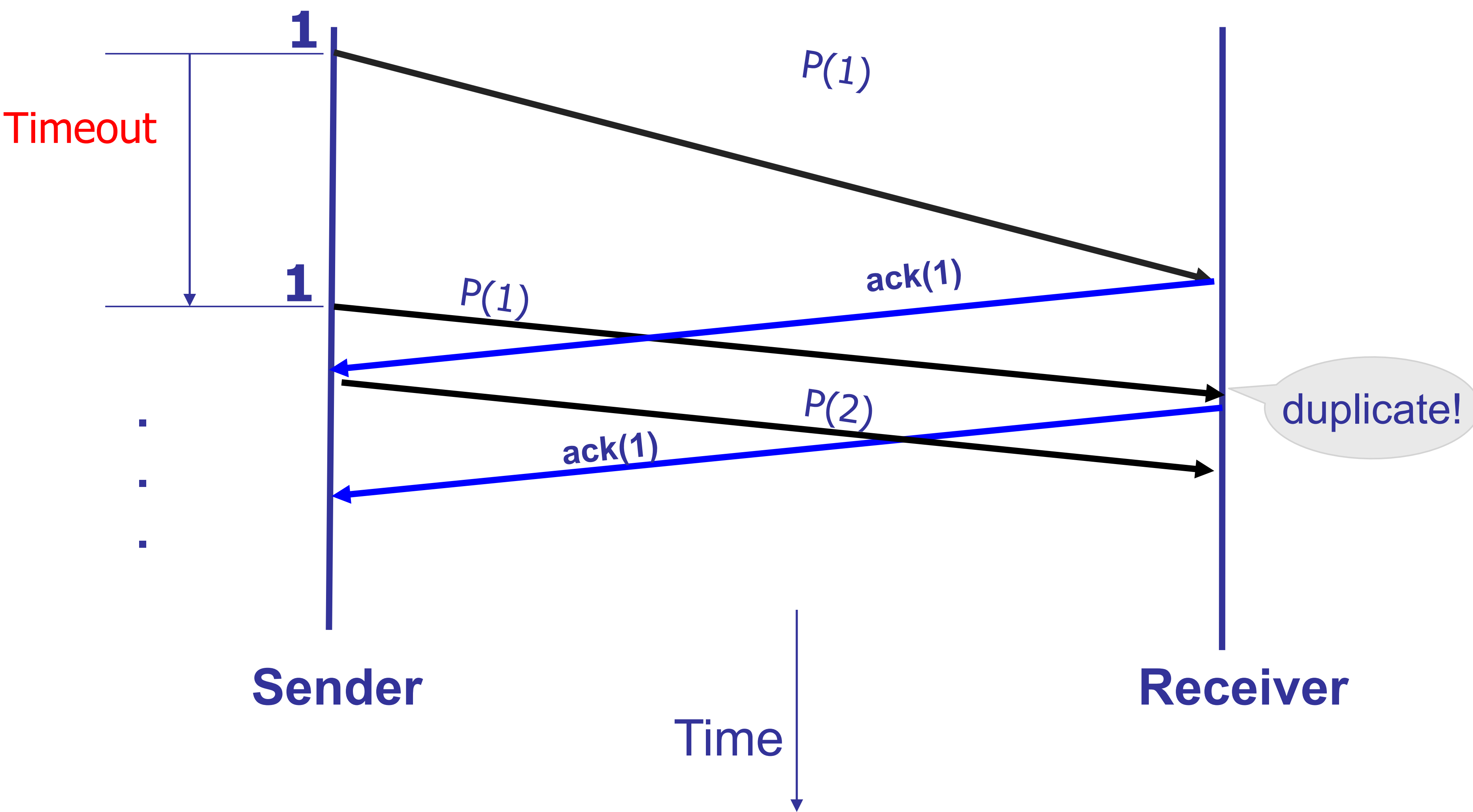


Dealing with packet loss (of ack)



Dealing with packet loss

Timer-driven retransmission can lead to <u>duplicates</u>



Components of a solution

- Checksums (to detect bit errors)
- Timers (to detect loss)
- Acknowledgements (positive or negative)
- Sequence numbers (to deal with duplicates)

Designing a reliable transport protocol？

A Solution: “Stop and Wait”

@Sender

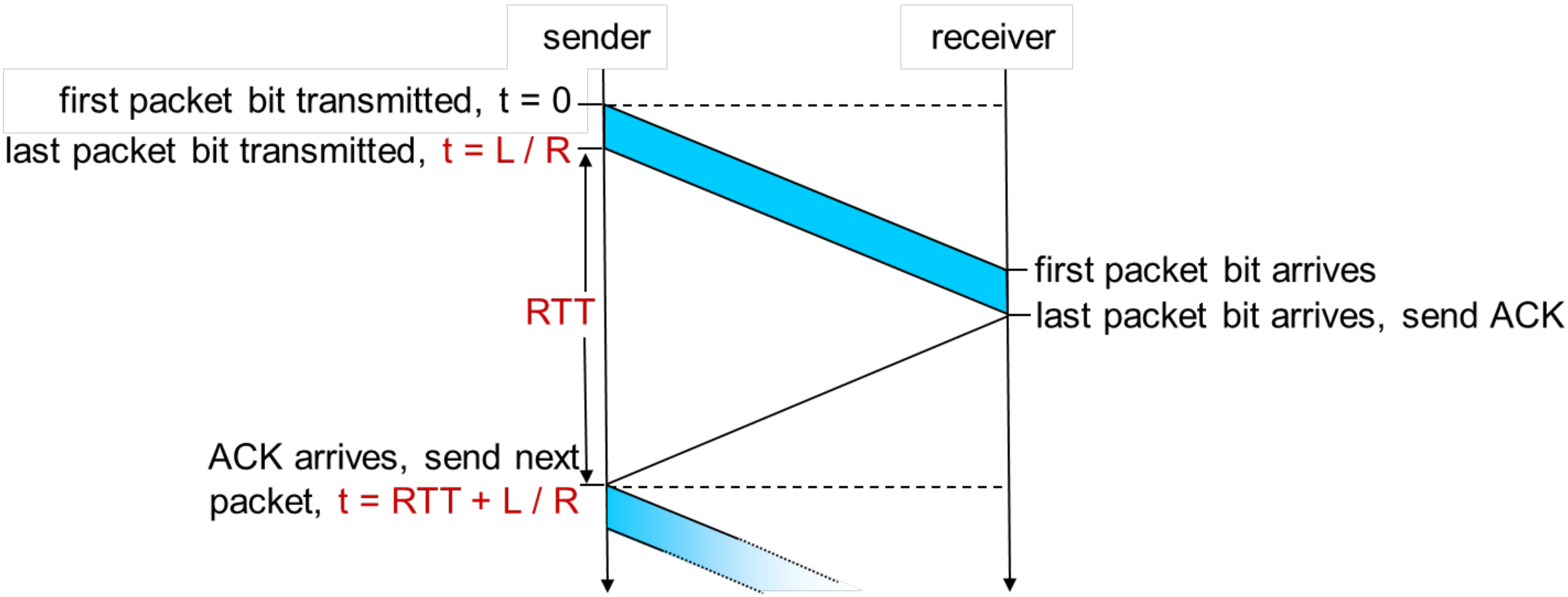
- Send packet(I); (re)set timer; wait for ack
- If (ACK)
  - I++; repeat
- If (NACK or TIMEOUT)
  - repeat

@Receiver

- Wait for packet
- If packet is OK, send ACK
- Else, send NACK
- Repeat

- A correct reliable transport protocol, but an extremely inefficient one

Stop & Wait is inefficient



L: packet size
R: bandwidth of the link
RTT = 2*PropDelay: roundtrip time

If (L/R<< RTT) then
Throughput ~ DATA/RTT

Orders of magnitude

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

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

  - **if RTT=30 msec,**

- **U $_{sender}$: *utilization* – fraction of time sender busy sending**

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

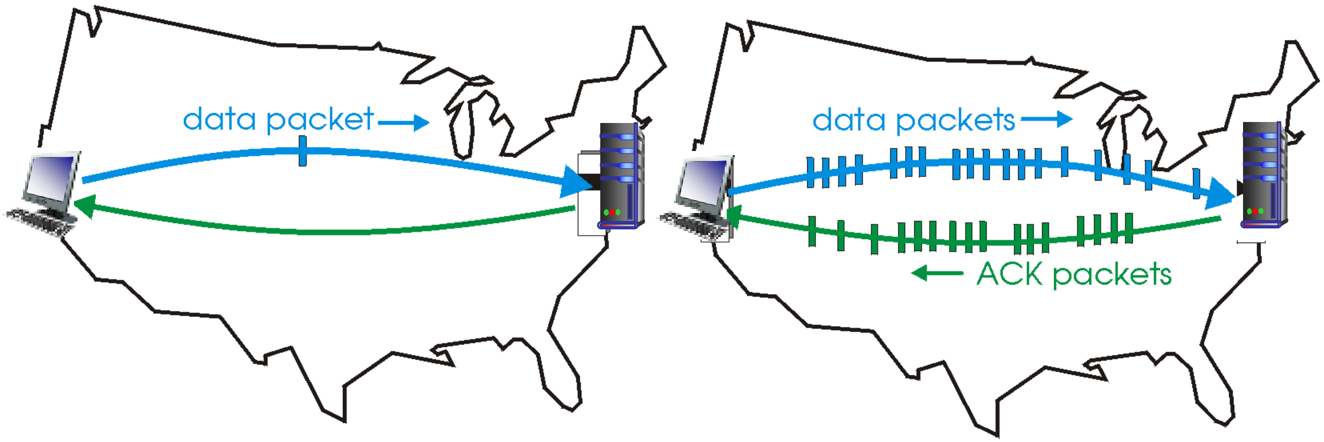
  - **33kB/sec thruput over 1 Gbps link!**

- **network protocol limits use of physical resources!**

Pipelined protocols

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

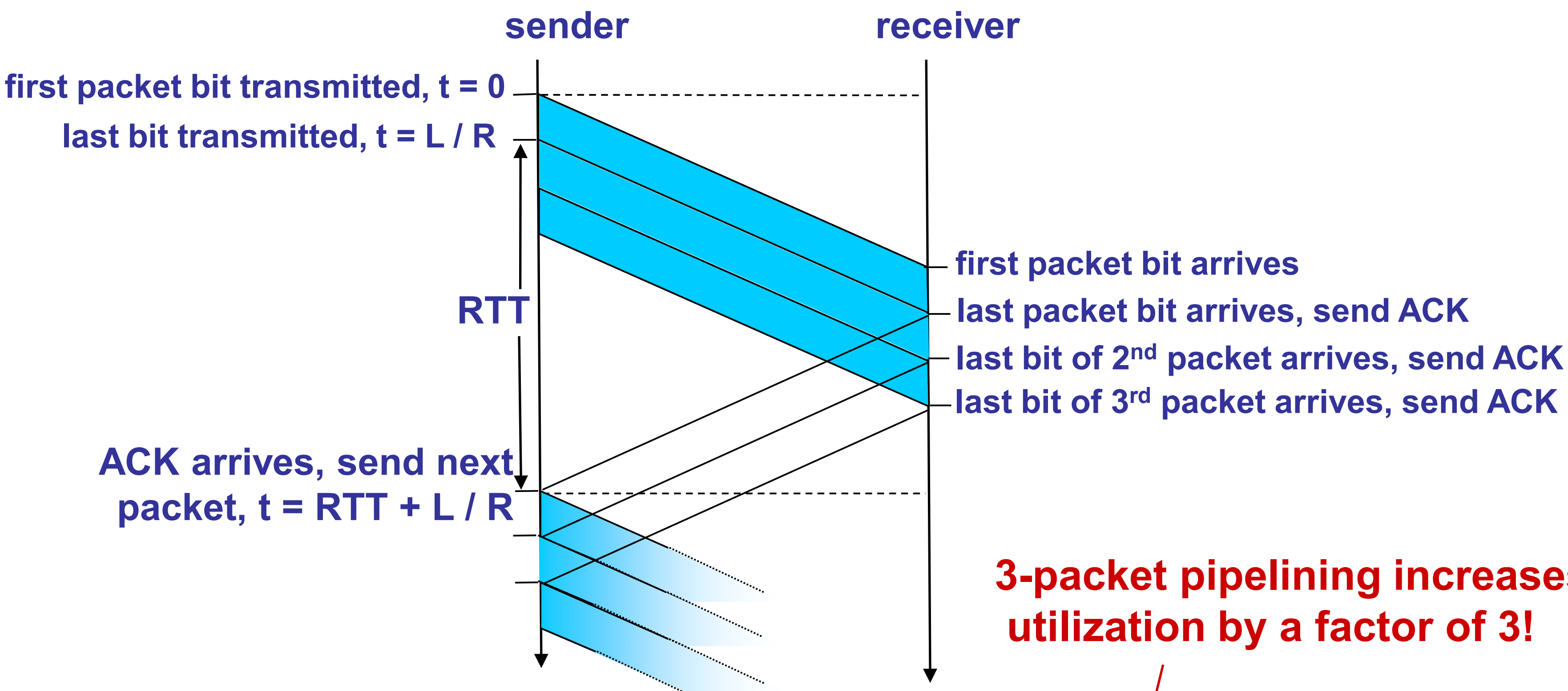
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: increased utilization



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

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

Three design decisions

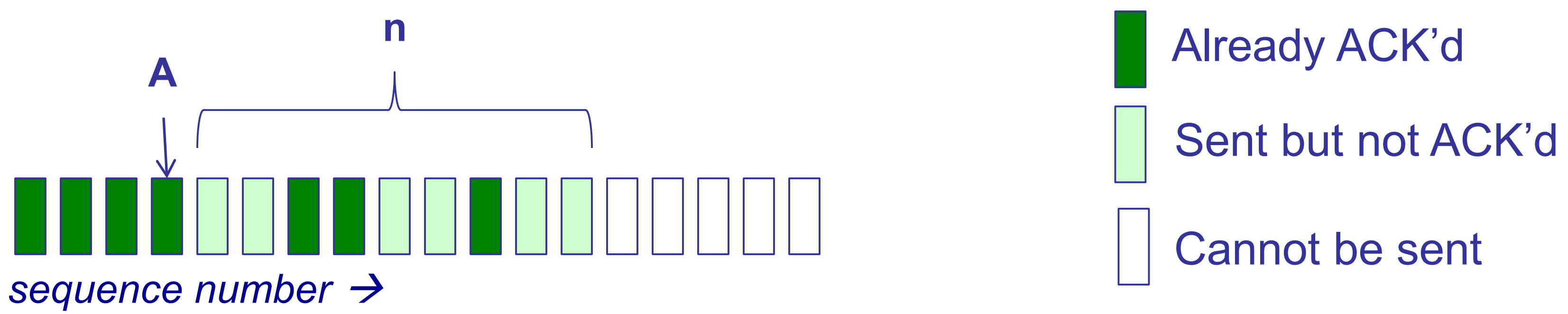
- Which packets can sender send?
  - Sliding window
- How does receiver ack packets?
  - Cumulative
  - Selective
- Which packets does sender resend?
  - Go-Back N (GBN)
  - Selective Repeat (SR)

Sliding window

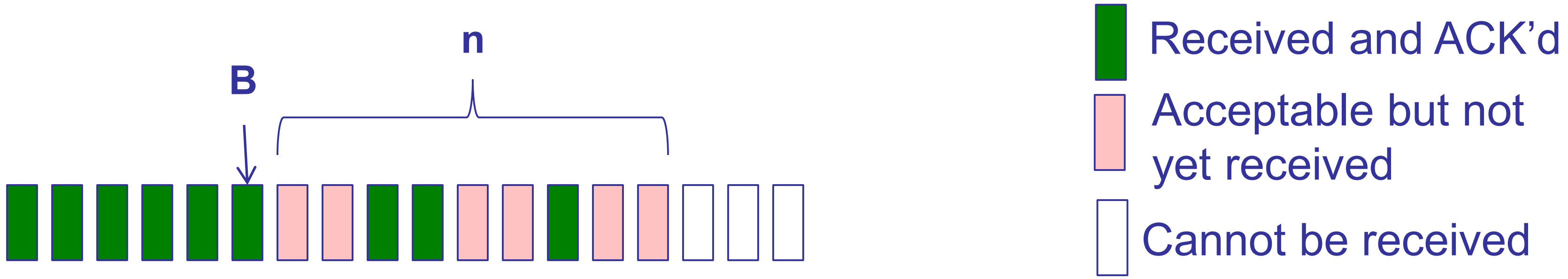
- Window = set of adjacent sequence numbers
  - The size of the set is the window size; assume window size is n
- General idea: send up to n packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets "slides" on successful reception/acknowledgement
  - Window contains all packets that might still be in transit
- Sliding window often called "packets in flight"

Sliding window

- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}



- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}



Throughput of sliding window

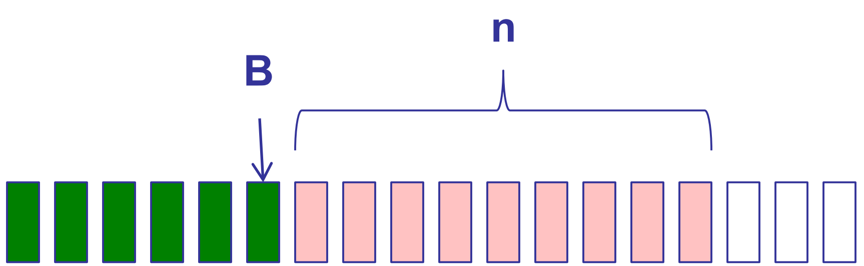
- If window size is n, then throughput is roughly
  - MIN(n*DATA/RTT, Link Bandwidth)
- Compare to Stop and Wait: Data/RTT
- What happens when n gets too large?

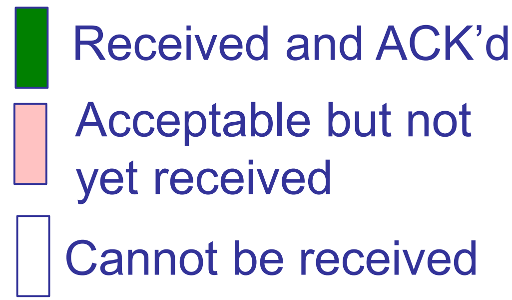
Acknowledgements w/ sliding window

- Two common options
  - Cumulative ACKs: ACK carries next in-order sequence number that the receiver expects

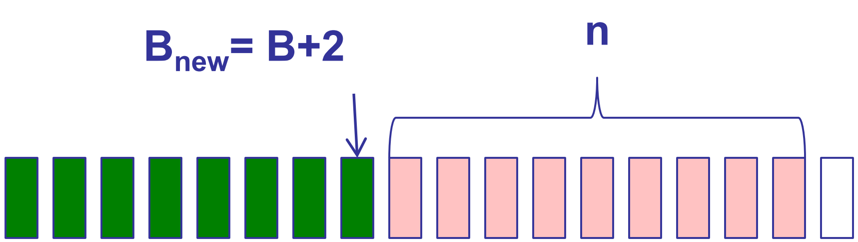
Cumulative acknowledgements

- At receiver





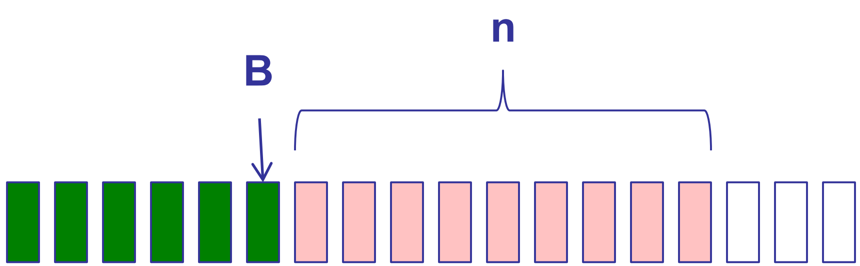
- After receiving B+1, B+2

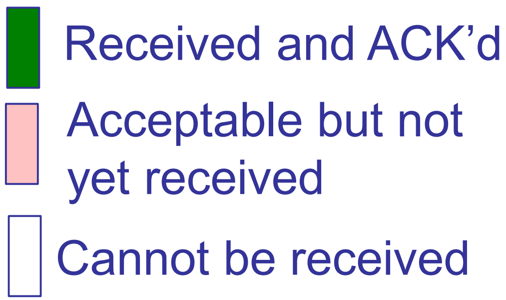


- Receiver sends ACK(B+3) = ACK(B_{new}+1)

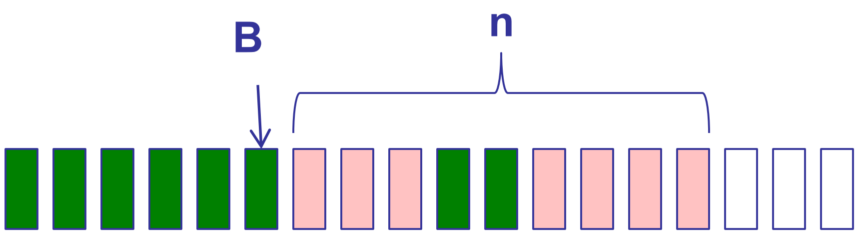
Cumulative acknowledgements (cont'd)

- At receiver





- After receiving B+4, B+5



- Receiver sends ACK(B+1)

Acknowledgements w/ sliding window

- Two common options
  - Cumulative ACKs: ACK carries next in-order sequence number the receiver expects
  - Selective ACKs: ACK individually acknowledges correctly received packets

- Selective ACKs offer more precise information but require more complicated book-keeping

Sliding window protocols

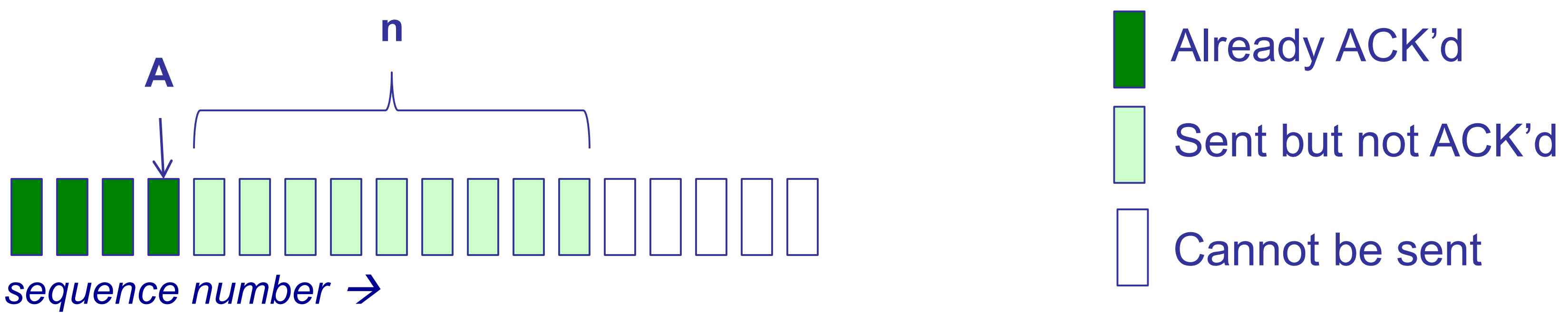
- Resending packets: two canonical approaches
  - Go-Back-N
  - Selective Repeat
- Many variants that differ in implementation details

Go-Back-N (GBN)

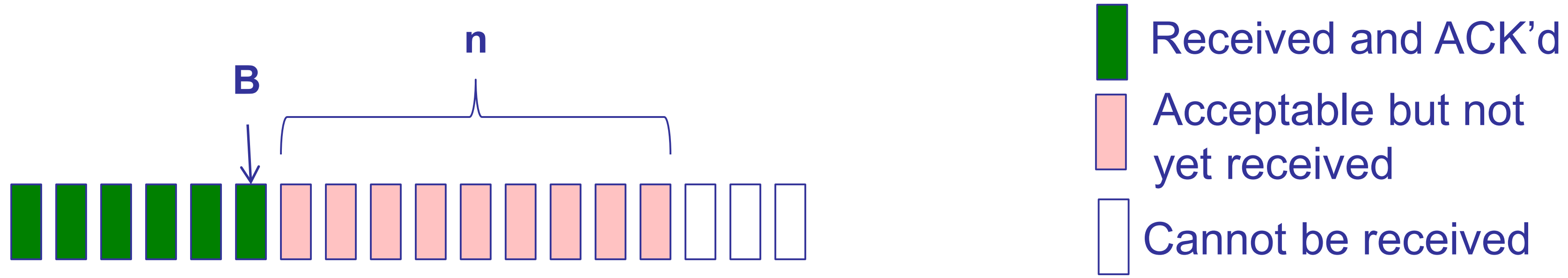
- Sender transmits up to n unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses cumulative acknowledgements
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding ack (A+1)
- If timeout, retransmit A+1, … , A+n

Sliding window with GBN

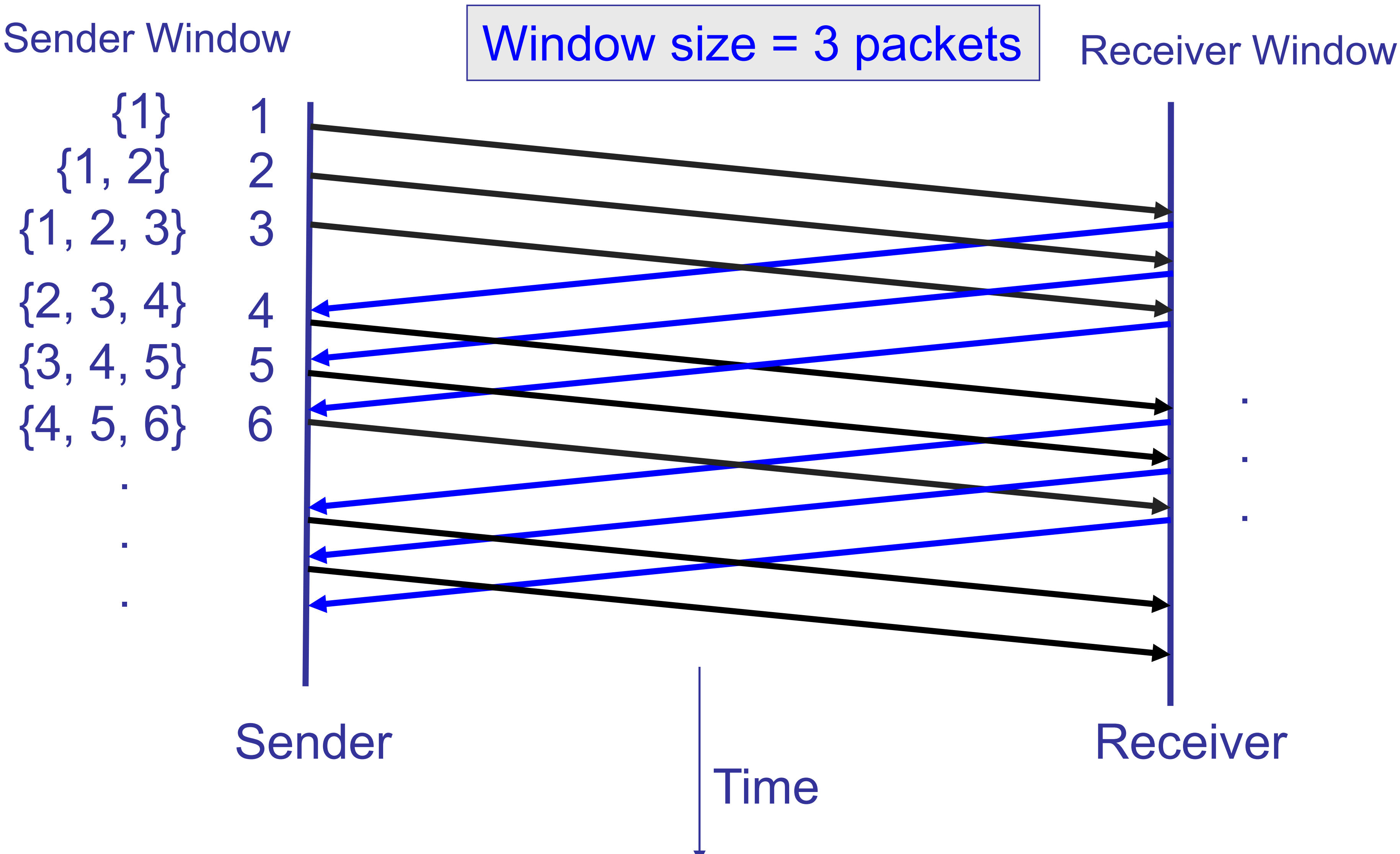
- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}



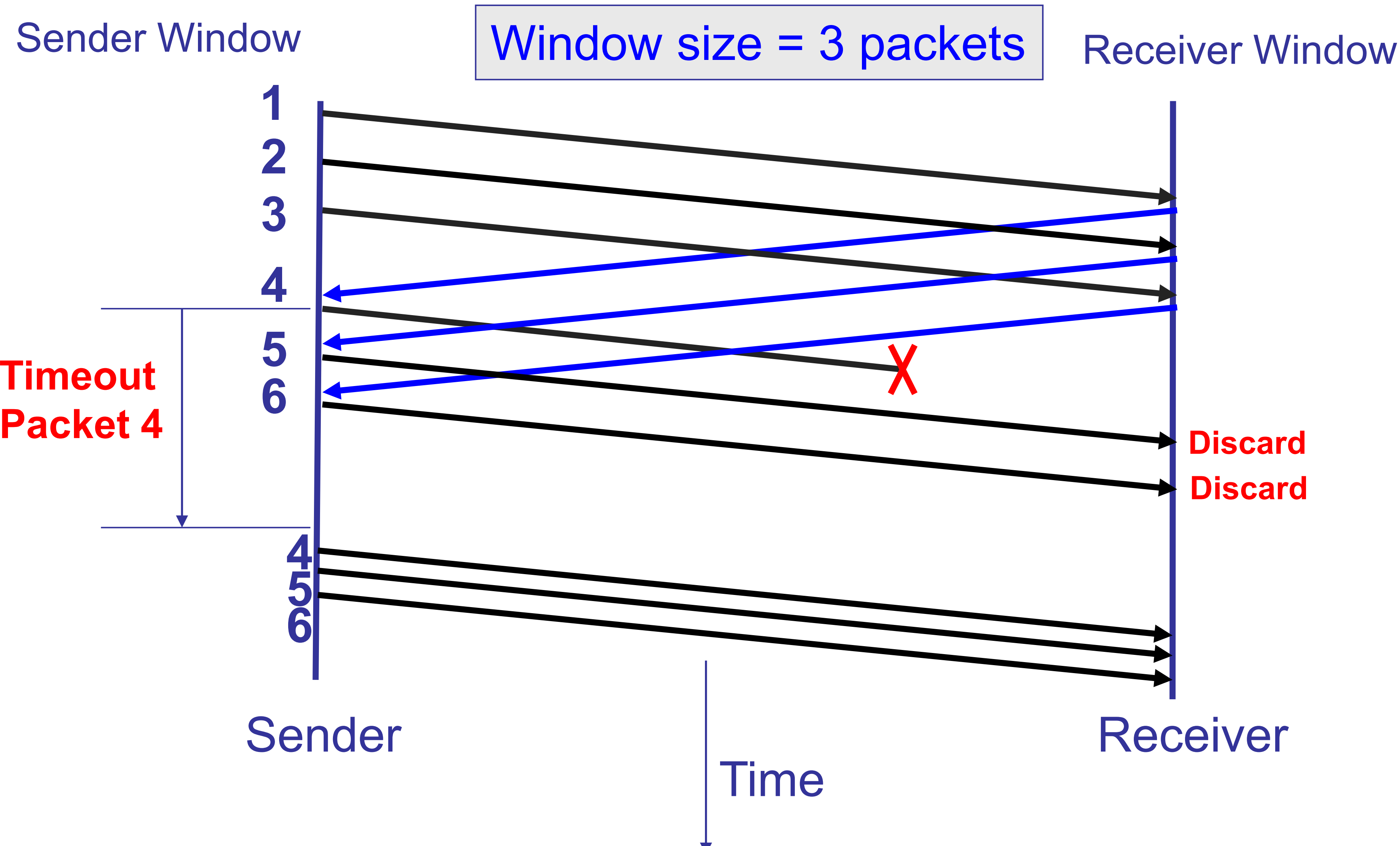
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}



GBN example w/o errors



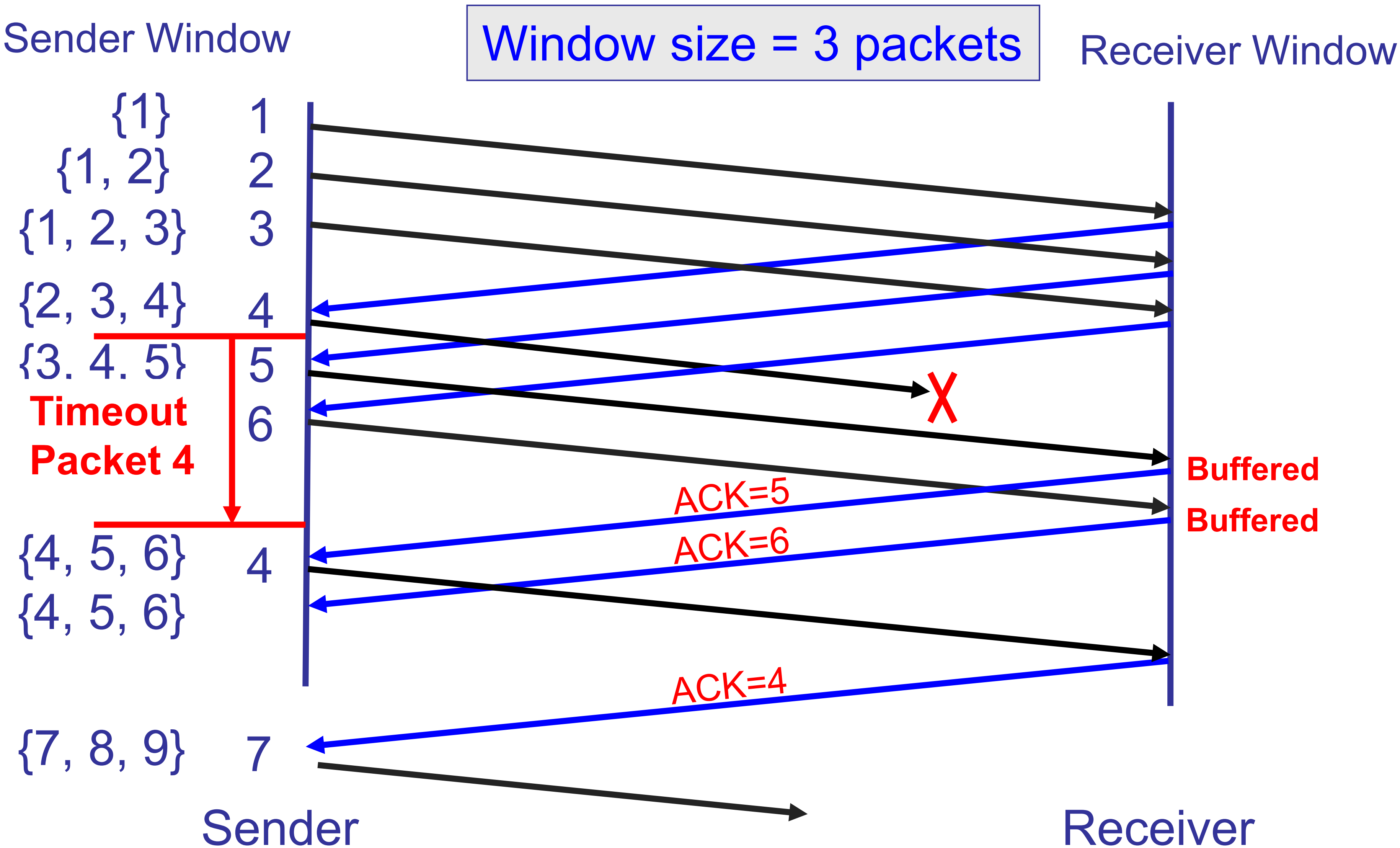
GBN example with errors



Selective Repeat (SR)

- Sender: transmit up to n unacknowledged packets
- Assume packet k is lost, k+1 is not
  - Receiver: indicates packet k+1 correctly received
  - Sender: retransmit only packet k on timeout
- Efficient in retransmissions but complex book-keeping
  - Need a timer per packet

SR example with errors



GBN vs. Selective Repeat

- When would GBN be better?
  - When error rate is low; wastes bandwidth otherwise
- When would SR be better?
  - When error rate is high; otherwise, too complex

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Implementation complexity depends on protocol details (GBN vs. SR)

Components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - Cumulative
  - Selective
- Sequence numbers (duplicates, windows)
- Sliding windows (for efficiency)
- Reliability protocols use the above to decide when and what to retransmit or acknowledge

Summary

- Transport Layer Service
  - Addressing
  - Multiplexing
- Design of reliable transport
  - Dealing with packet corruption
  - Flow control
    - Stop-and-wait
    - Sliding Window

Homework

- 第3章:R5, R8, P1, P19