

# Computer Networks and Applications

COMP 3331/COMP 9331


Week 4

## Transport Layer Part 1

Reading Guide:  
Chapter 3, Sections 3.1 – 3.5

# Transport Layer

## our goals:

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

# Transport Layer 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

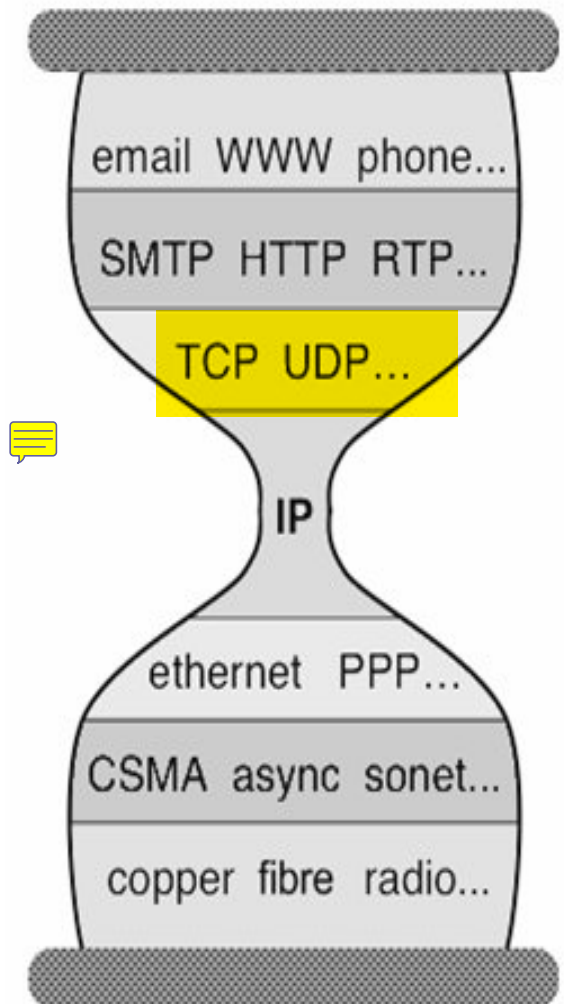
- segment structure
- reliable data transfer
- flow control
- connection management

## 3.6 principles of congestion control



## 3.7 TCP congestion control

# Transport layer

- ❖ Moving “down” a layer 
- ❖ Current perspective:
  - Application is the boss....
  - Usually executing within the OS Kernel 
  - The network layer is ours to command !!

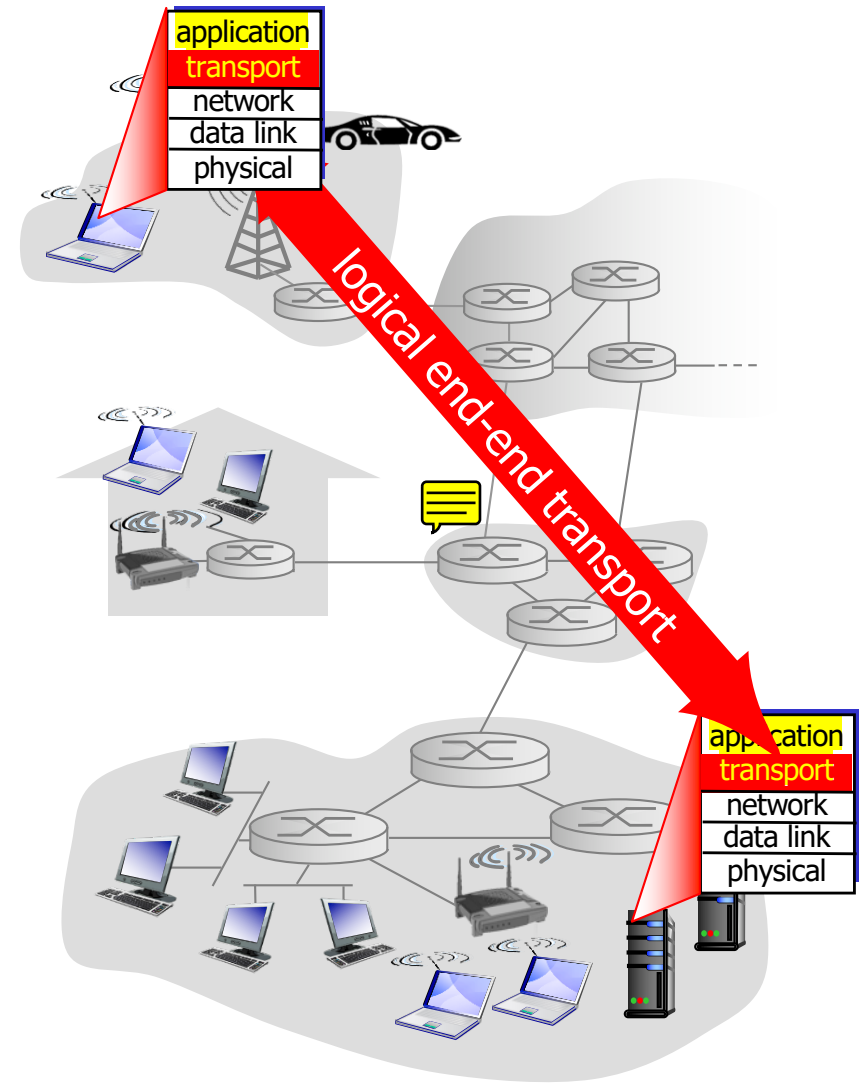


# Network layer (context)

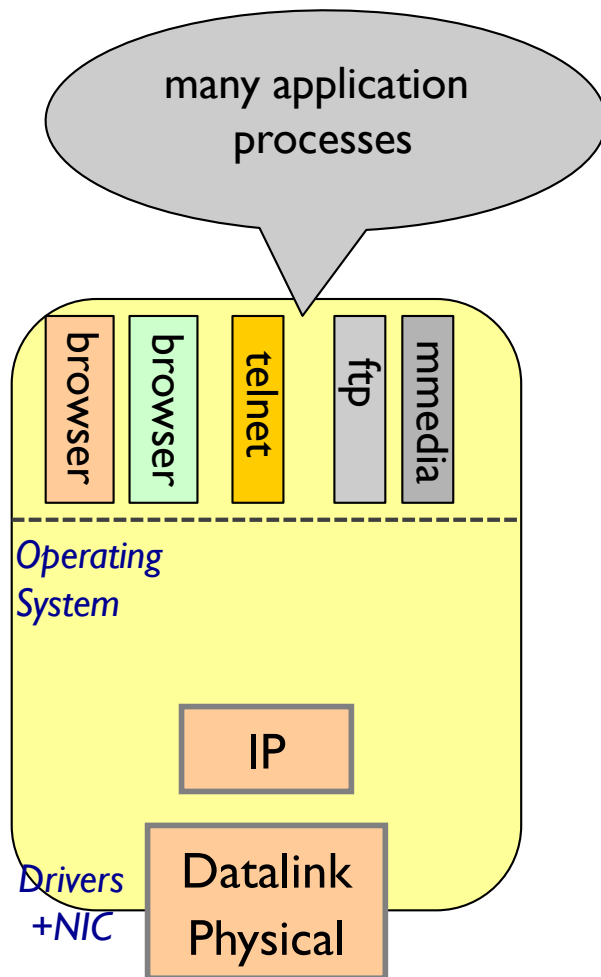
- ❖ What it does: finds paths through network
  - Routing from one end host to another
- ❖ What it doesn't:
  - Reliable transfer: “best effort delivery”
  - Guarantee paths 
  - Arbitrate transfer rates
- ❖ For now, think of the network layer as giving us an “API” with one function: *sendtohost(data, host)*
  - Promise: the data will go to that (usually!!) 

# Transport services and protocols

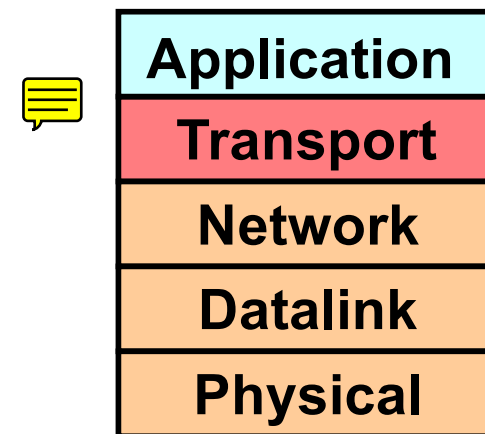
- ❖ provide *logical communication* between app processes running on different hosts
- ❖ transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
  - Exports services to application that network layer does not provide



# Why a transport layer?

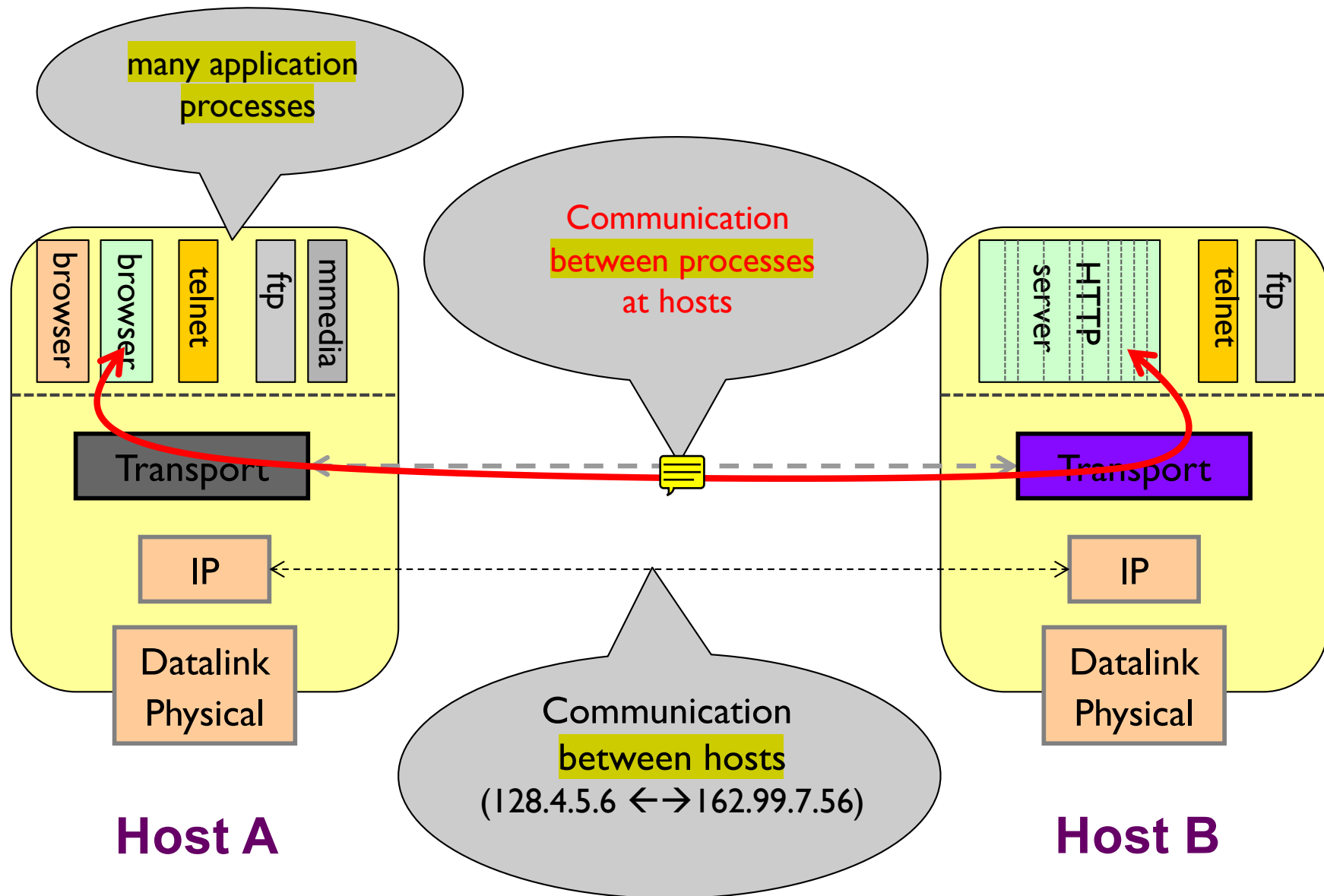


**Host A**



**Host B**

# Why a transport layer?





# Transport Layer 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

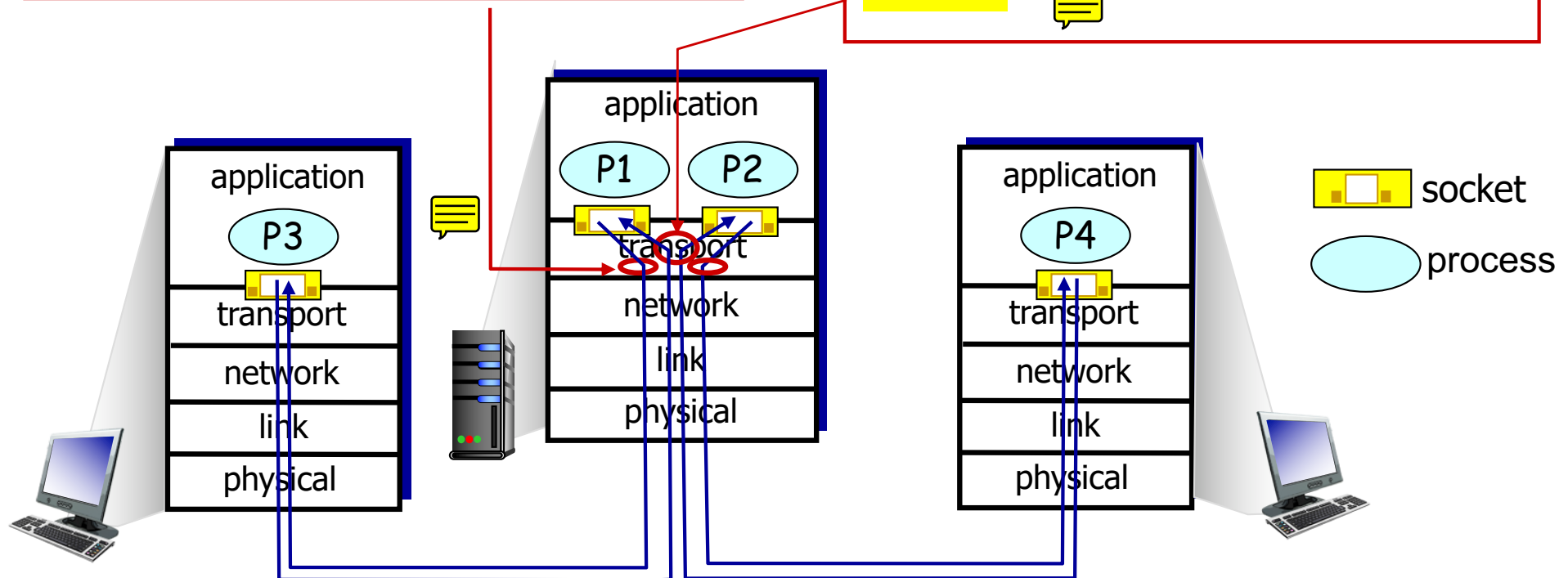
# Multiplexing/demultiplexing

## *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



**Note:** The network is a shared resource. It does not care about your applications, sockets, etc.

# 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 #

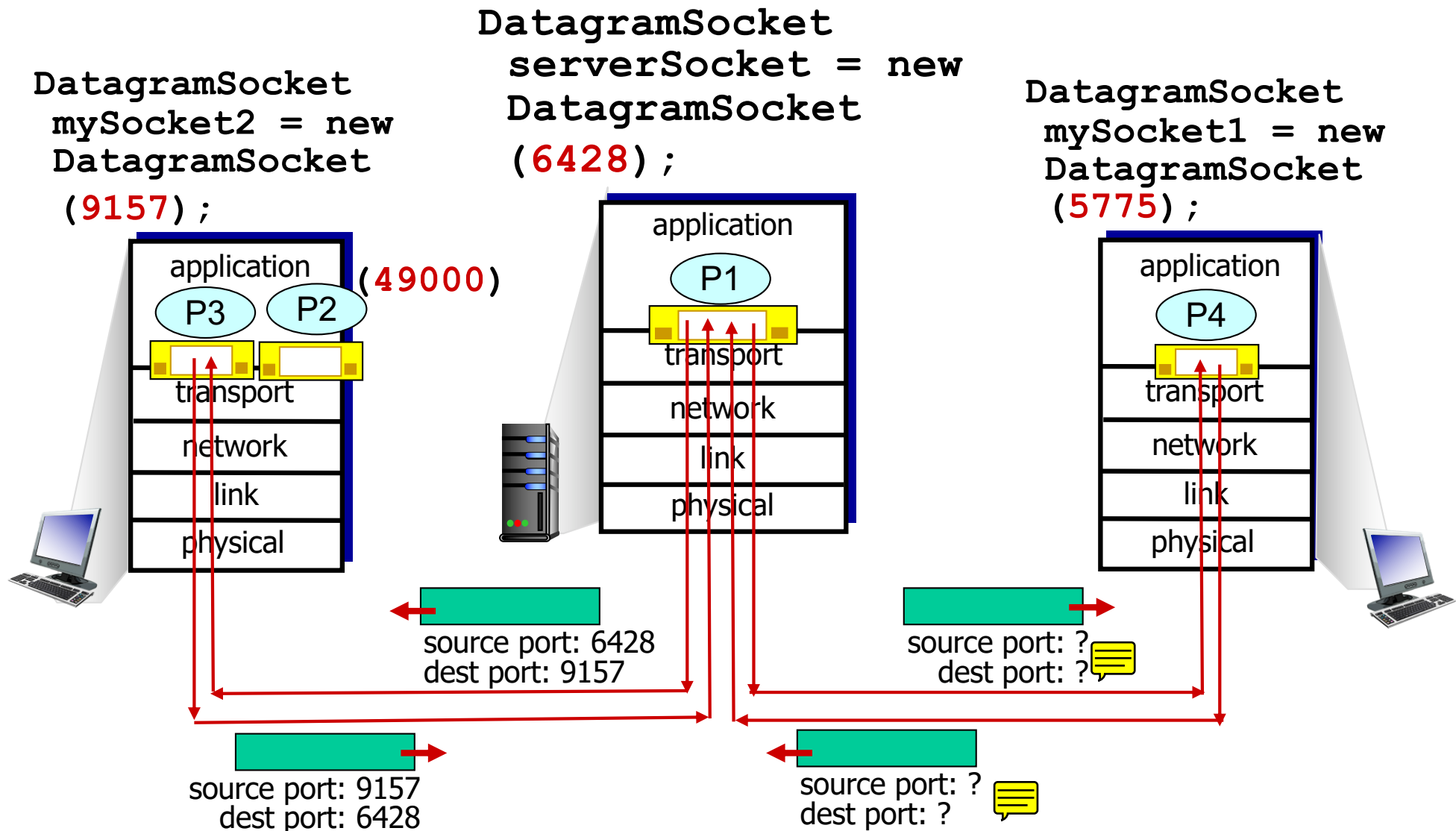
- ❖ when host receives UDP segment: 

- 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 

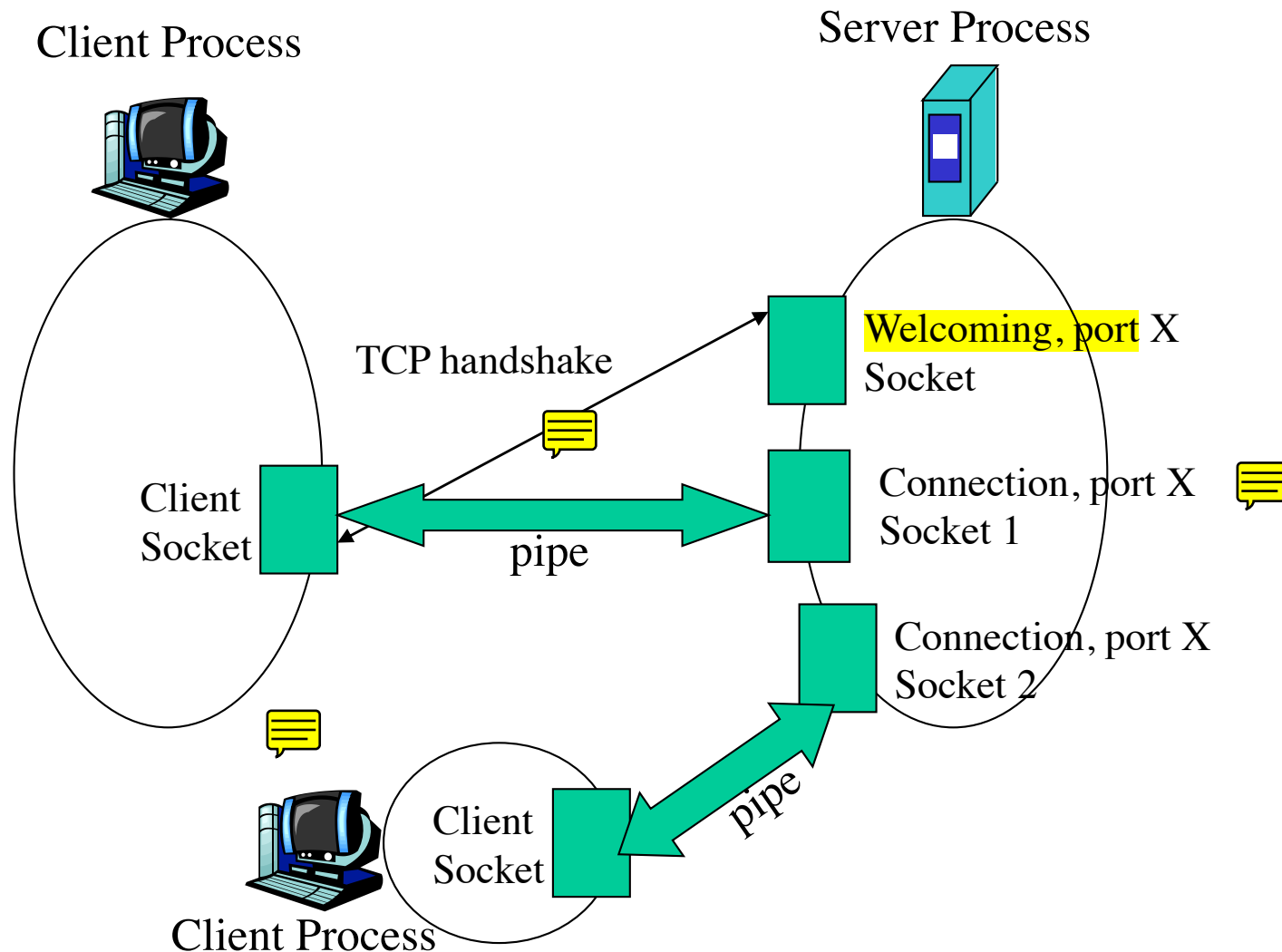
# Connectionless demux: example



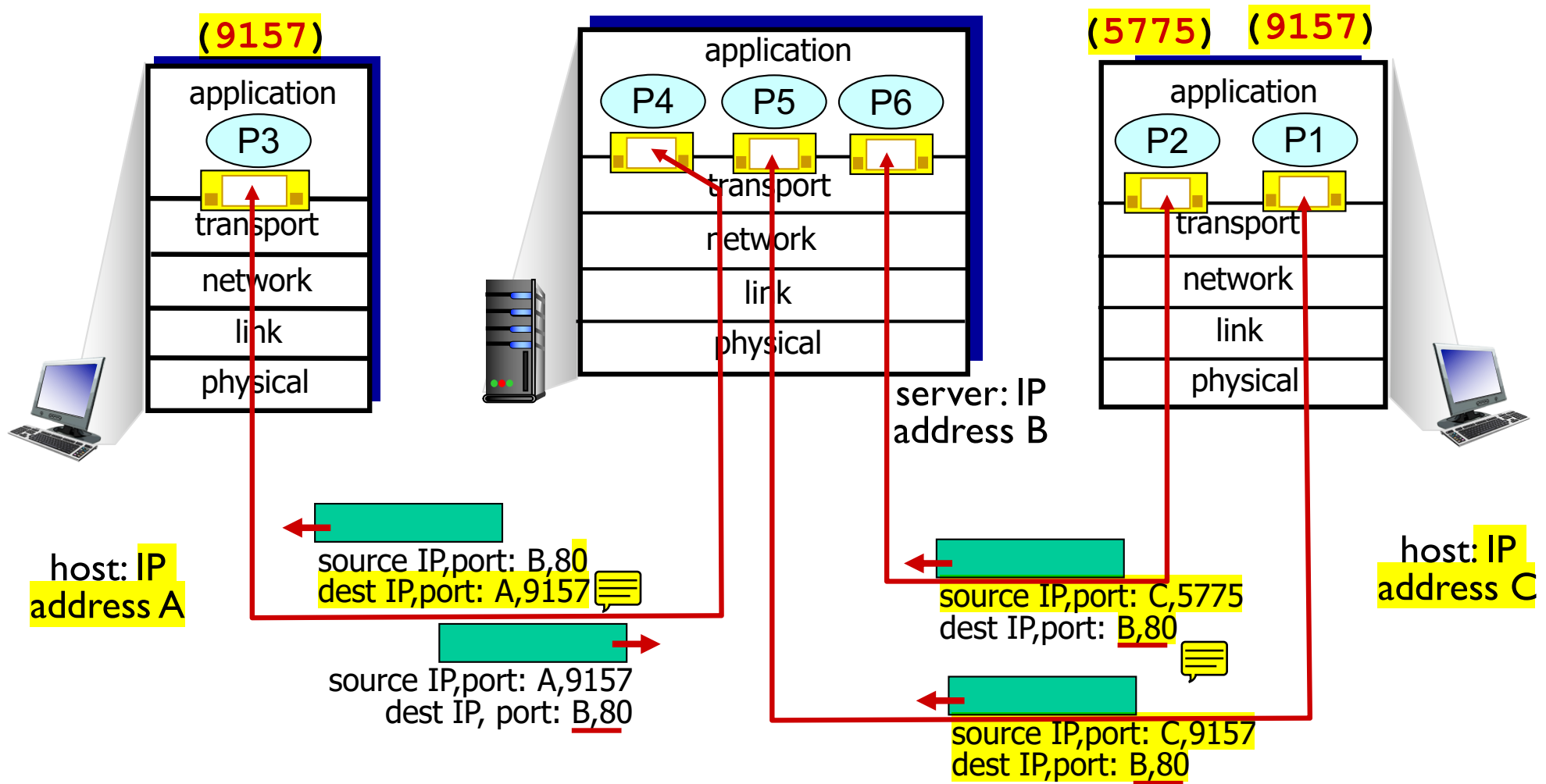
# Connection-oriented demux

- ❖ 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

# Revisiting TCP Sockets



# Connection-oriented demux: example



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

# May I scan your ports?

<http://netsecurity.about.com/cs/hackertools/a/aa121303.htm>

- ❖ Servers wait at open ports for client requests
- ❖ Hackers often perform *port scans* to determine open, closed and unreachable ports on candidate victims
- ❖ Several ports are well-known
  - <1024 are reserved for well-known apps
  - Other apps also use known ports
    - MS SQL server uses port 1434 (udp)
    - Sun Network File System (NFS) 2049 (tcp/udp)
- ❖ Hackers can exploit known flaws with these known apps
  - Example: Slammer worm exploited buffer overflow flaw in the SQL server
- ❖ How do you scan ports?
  - Nmap, Superscan, etc

<http://www.auditmypc.com/>

<https://www.grc.com/shieldsup>



## Quiz: UDP Sockets



- ❖ Suppose we use UDP instead of TCP for designing a web server where all requests and responses fit in a single UDP segment. Suppose 100 clients are simultaneously communicating with this web server. How many sockets are respectively at the server and each client?
  - a) 1, 1
  - b) 2, 1
  - c) 200, 2
  - d) 100, 1
  - e) 101, 1

## Quiz: TCP Sockets



- ❖ Suppose 100 clients are simultaneously communicating with a traditional HTTP/TCP web server. How many sockets are respectively at the server and each client?
  - a) 1, 1
  - b) 2, 1
  - c) 200, 2
  - d) 100, 1
  - e) 101, 1

## Quiz: TCP Sockets



- ❖ Suppose 100 clients are simultaneously communicating with a traditional HTTP/TCP web server. Do all of the sockets at the server have the same server-side port number?
  - a) Yes
  - b) No

# Transport Layer 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

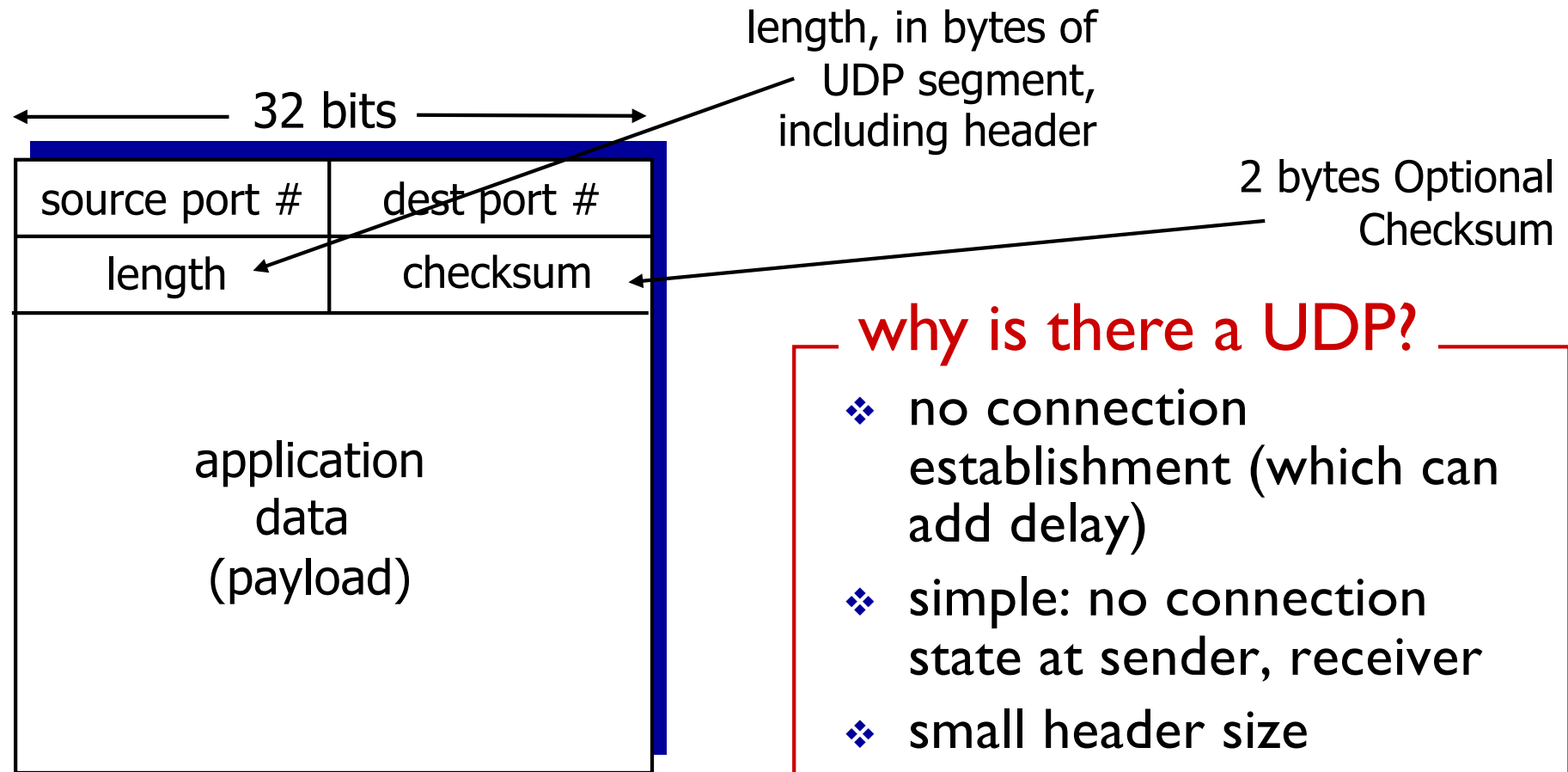
3.6 principles of congestion control

3.7 TCP congestion control

# 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: segment header



UDP segment format

## 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

# UDP checksum

- **Goal:** detect “errors” (e.g., flipped bits) in transmitted segment
  - Router memory errors
  - Driver bugs
  - Electromagnetic interference

## 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

## receiver:

- ❖ Add all the received together as 16-bit integers
- ❖ Add that to the checksum
- ❖ If the result is not 1111 1111 1111, there are errors !

# 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
<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



# UDP: Checksum

- Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.
- Checksum **header**, **data** and pre-pended **IP pseudo-header**
- But the header contains the checksum itself?
- What's IP pseudo-header?

bits	0 – 7	8 – 15	16 – 23	24 – 31
0	Source address			
32	Destination address			
64	Zeros	Protocol	UDP length	
96	Source Port		Destination Port	
128	Length		Checksum	
160+	Data			

UDP header {

{ IP pseduo-header

{ UDP payload

25

# Internet checksum: example

153.18.8.105			
171.2.14.10			
All 0s	17	15	
1087		13	
15		All 0s	
T	E	S	T
I	N	G	All 0s

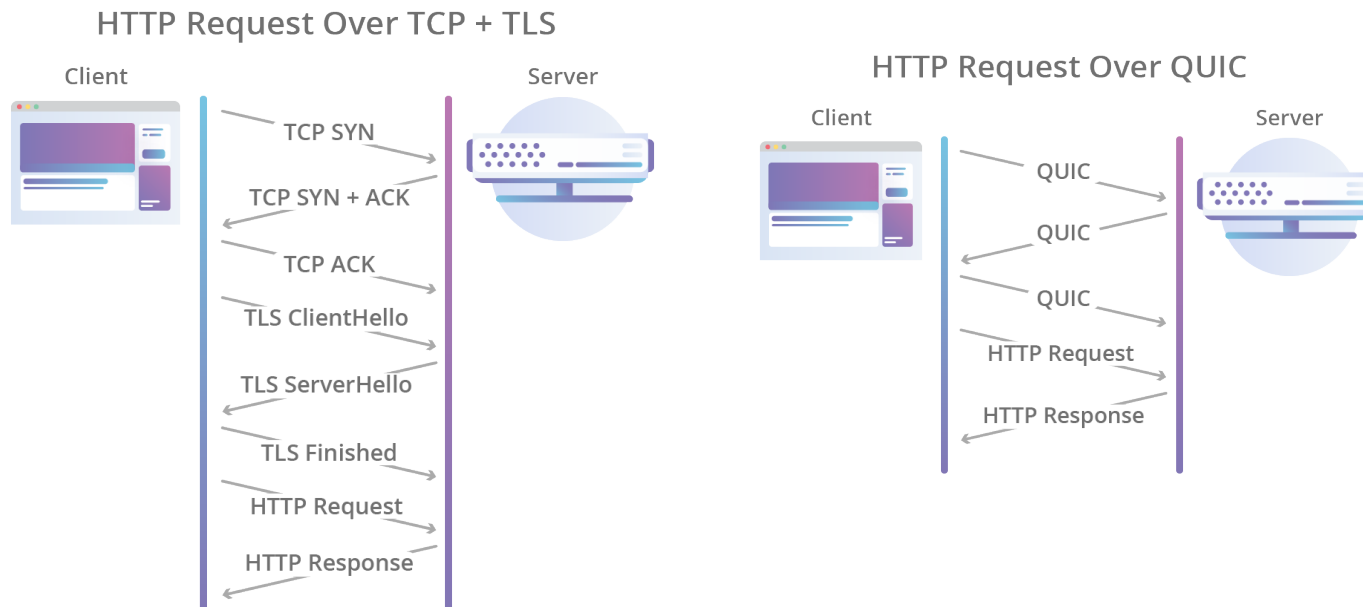
10011001	00010010	→	153.18
00001000	01101001	→	8.105
10101011	00000010	→	171.2
00001110	00001010	→	14.10
00000000	00010001	→	0 and 17
00000000	00001111	→	15
00000100	00111111	→	1087
00000000	00001101	→	13
00000000	00001111	→	15
00000000	00000000	→	0 (checksum)
01010100	01000101	→	T and E
01010011	01010100	→	S and T
01001001	01001110	→	I and N
01000111	00000000	→	G and 0 (padding)
<hr/>			
10010110	11101011	→	Sum
01101001	00010100	→	Checksum

# UDP Applications

- ❖ Latency sensitive/time critical
  - ❖ Quick request/response (DNS, DHCP)
  - ❖ Network management (SNMP)
  - ❖ Routing updates (RIP)
  - ❖ Voice/video chat
  - ❖ Gaming (especially FPS)
- ❖ Error correction unnecessary (periodic messages)

# QUIC: Quick UDP Internet Connections

- ❖ Core idea: HTTP/2 over UDP
  - Faster connection establishment
  - Overcomes HoL blocking due to lost packets
  - Improved congestion control
  - Forward error correction
  - Connection migration



# Transport Layer 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

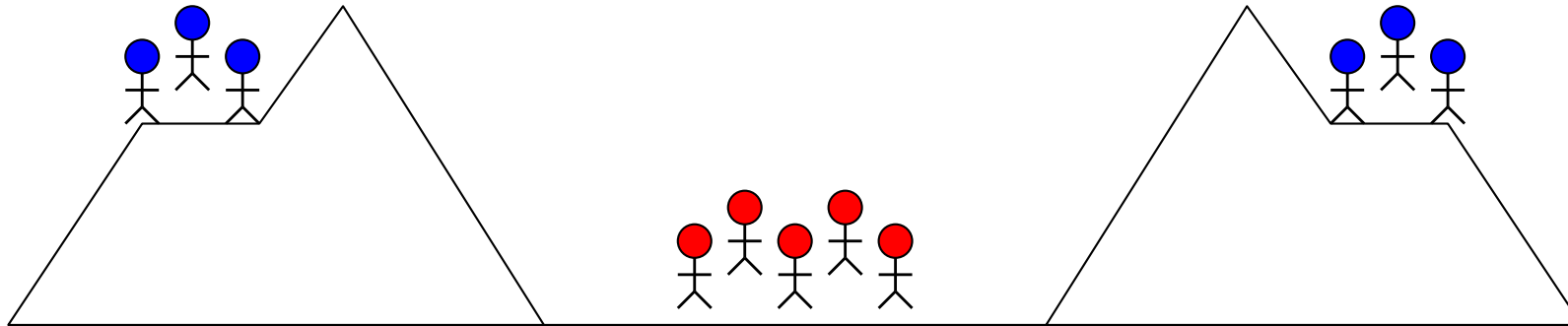
3.6 principles of congestion control

3.7 TCP congestion control

# Reliable Transport

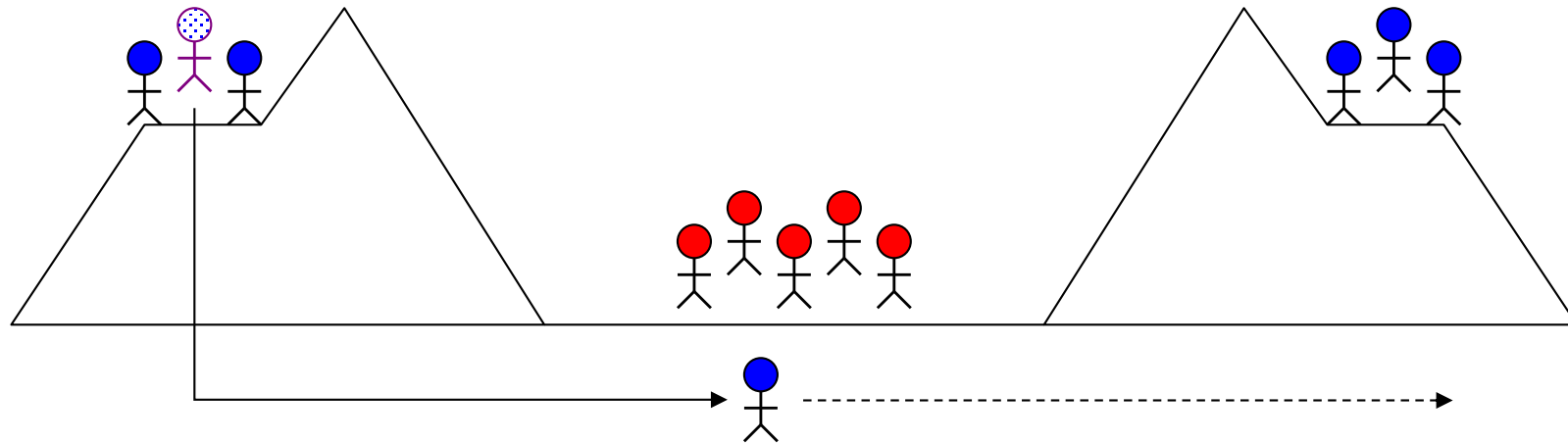
- In a perfect world, reliable transport is easy
- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost (*why?*)
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)

# The Two Generals Problem



- ❖ Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat
- ❖ Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)

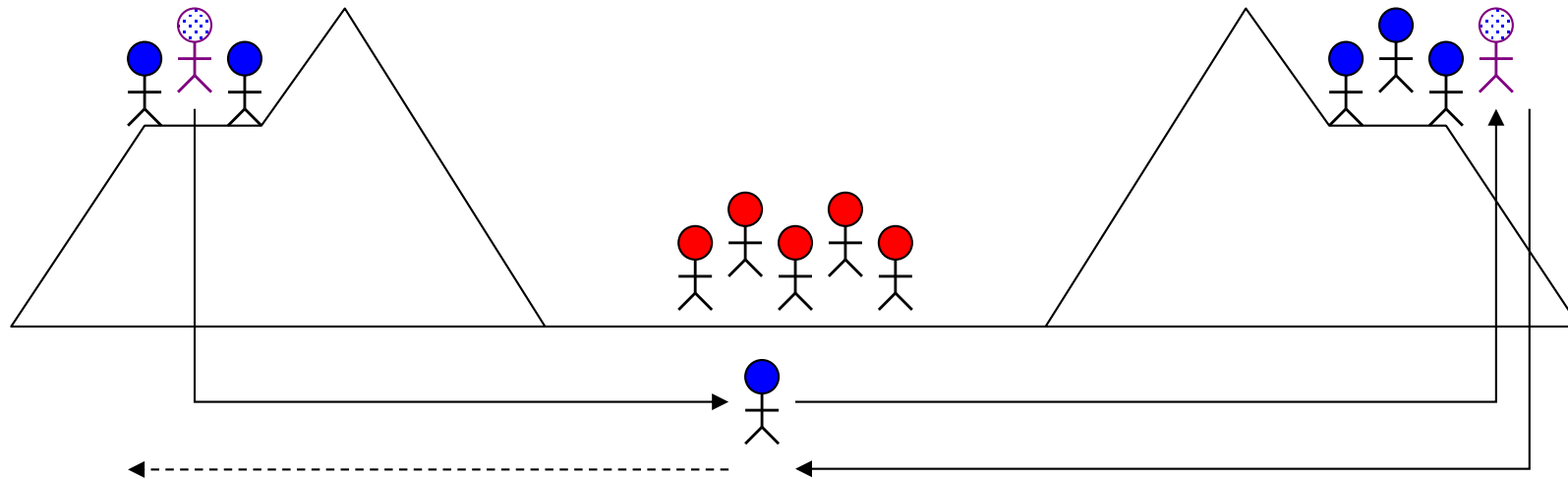
# The Two Generals Problem



- ❖ How to coordinate?
  - Send messenger: “Attack at dawn”
  - What if messenger doesn’t make it?



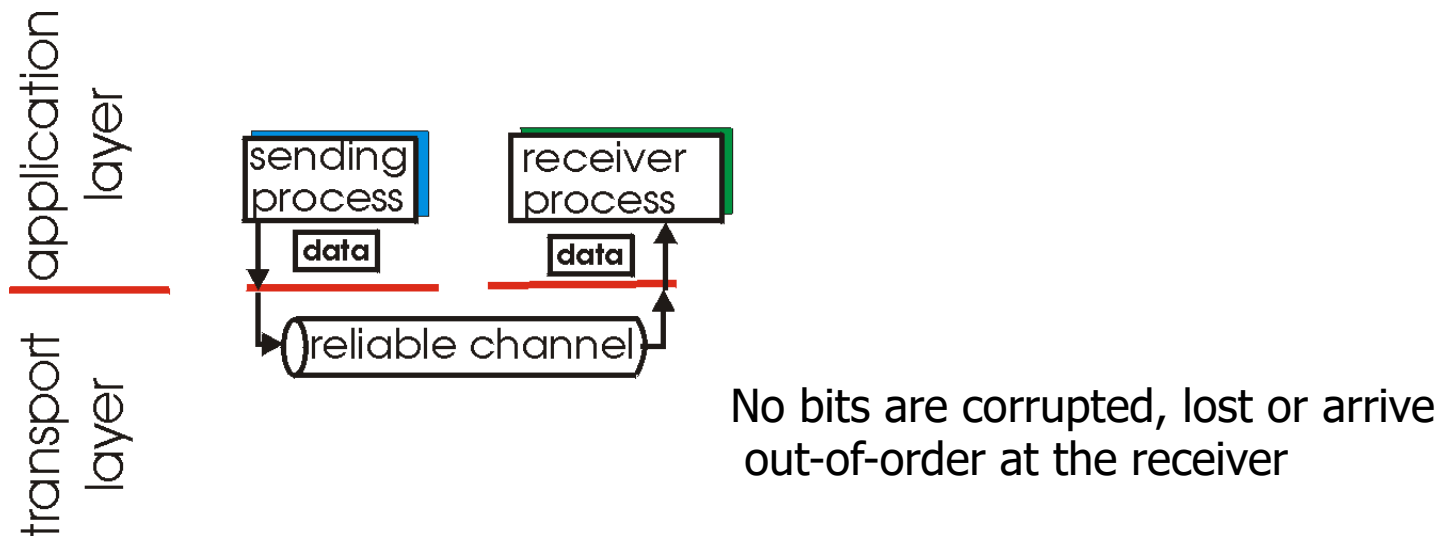
# The Two Generals Problem



- ❖ How to be sure messenger made it?
  - Send acknowledgement: “We received message”

# Principles of reliable data transfer

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

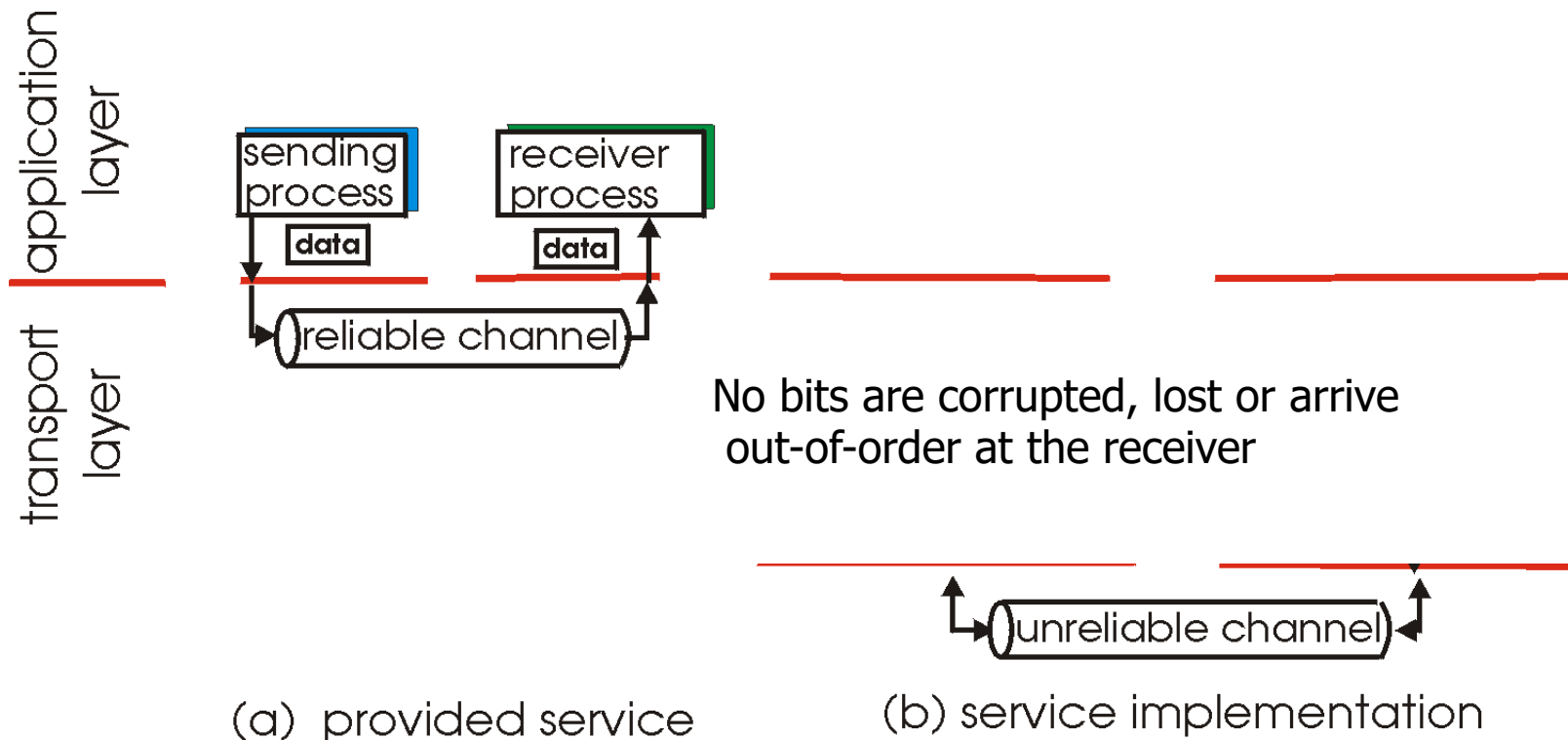


(a) provided service

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

# Principles of reliable data transfer

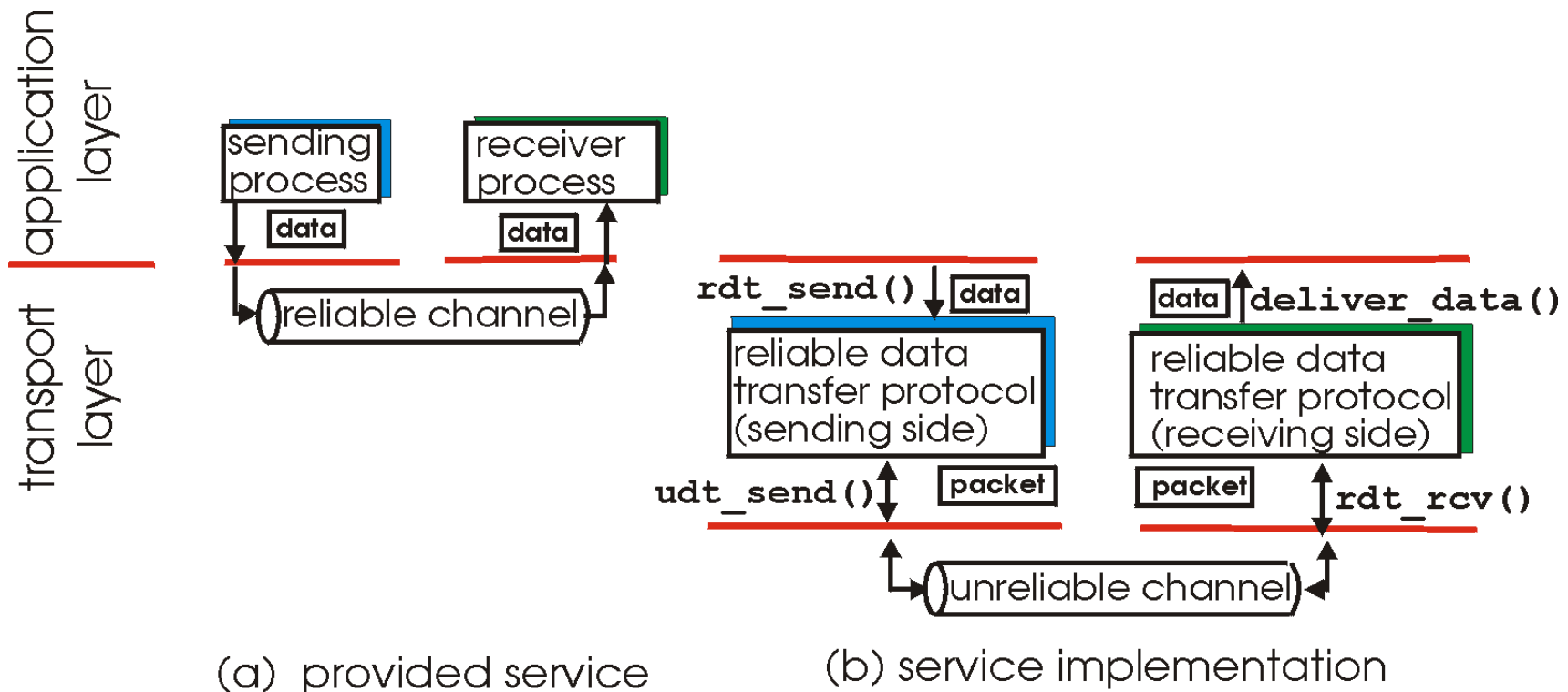
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



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

# Principles of reliable data transfer

- ❖ important in application, 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

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!
- Channel will not re-order packets

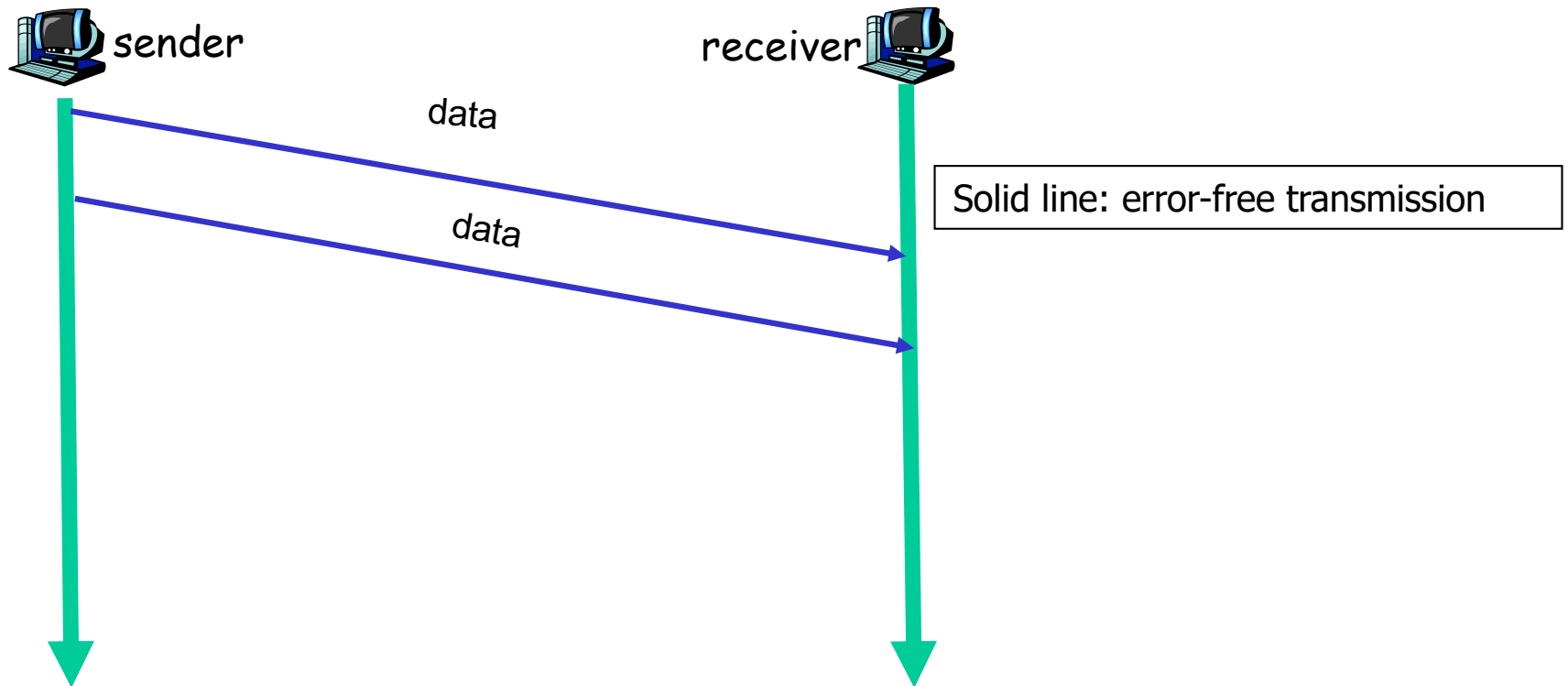
— stop and wait —

sender sends one packet,  
then waits for receiver  
response

# rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Transport layer does nothing !

# Global Picture of rdt1.0



# 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:*

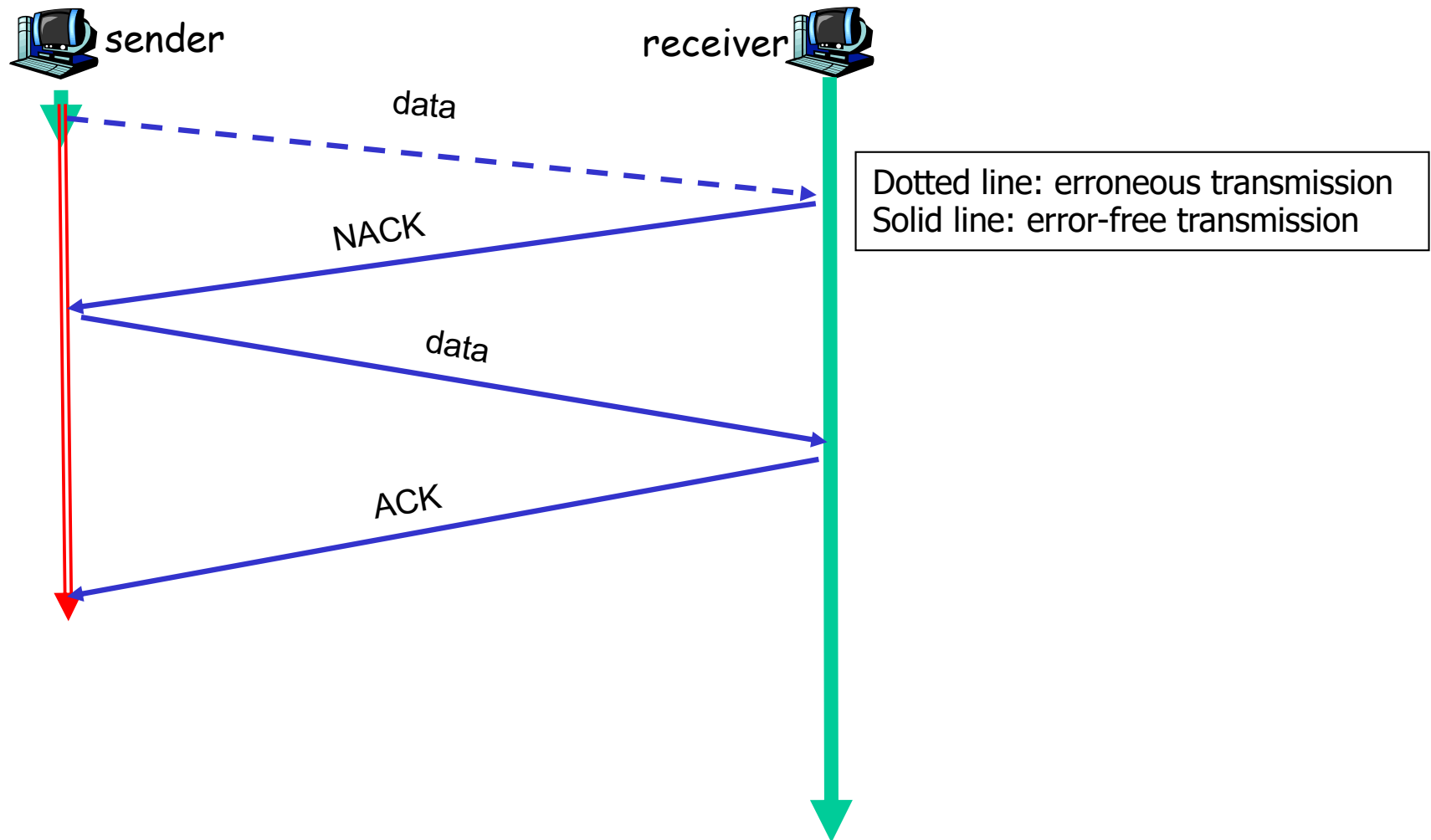
*How do humans recover from “errors”  
during conversation?*



# 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
- ❖ new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender
  - retransmission

# Global Picture of rdt2.0



# 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

## 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 “expected” pkt should have seq # of 0 or 1

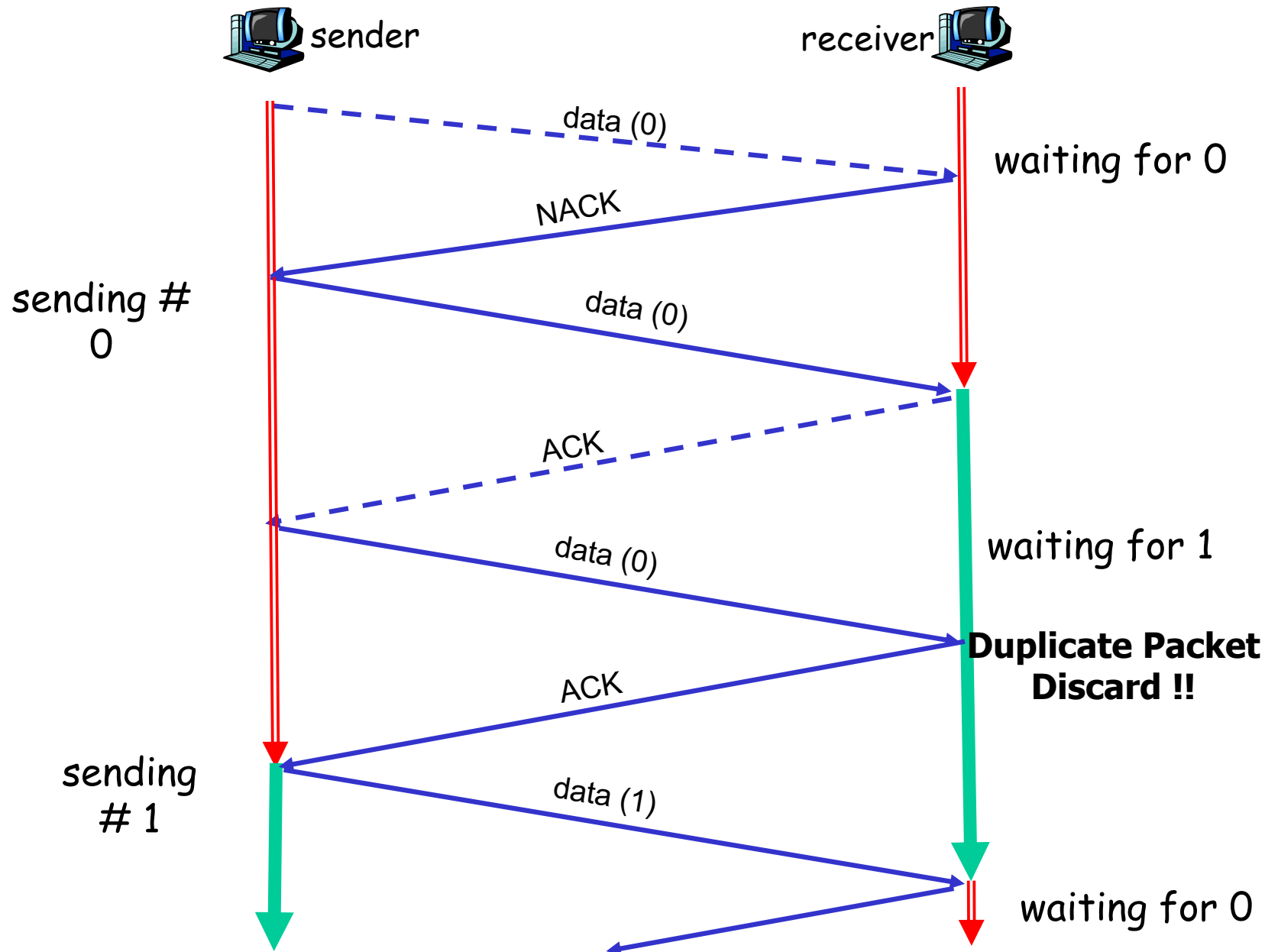
## receiver:

- ❖ 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

New Measures: Sequence Numbers, Checksum for ACK/NAK, Duplicate detection

# Another Look at rdt2.1

Dotted line: erroneous transmission  
Solid line: error-free transmission

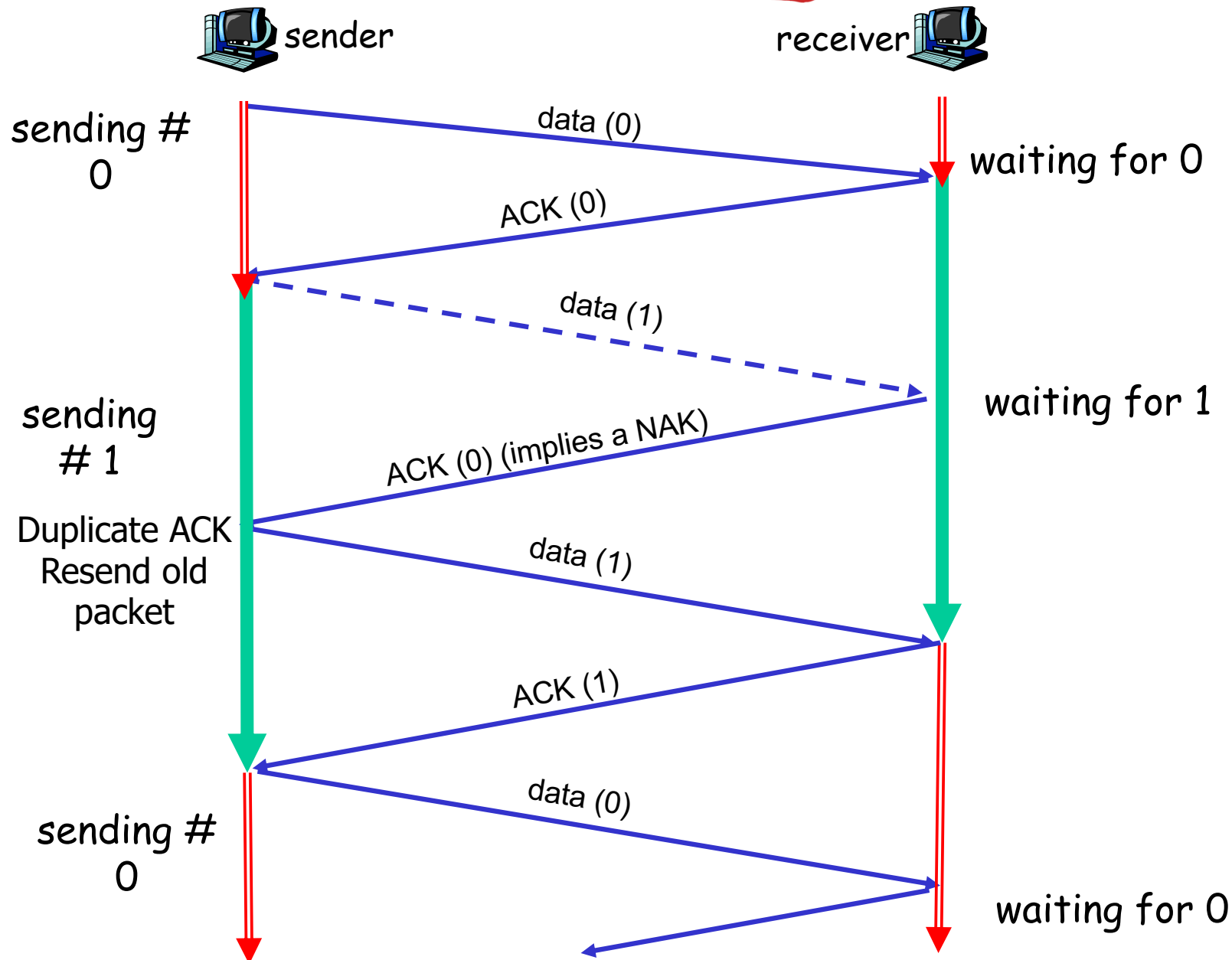


## rdt2.2: a NAK-free protocol

- ❖ 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: Example

Dotted line: erroneous transmission  
Solid line: error-free transmission



# rdt3.0: channels with errors *and* loss

## new assumption:

underlying channel can also lose packets (data, ACKs)

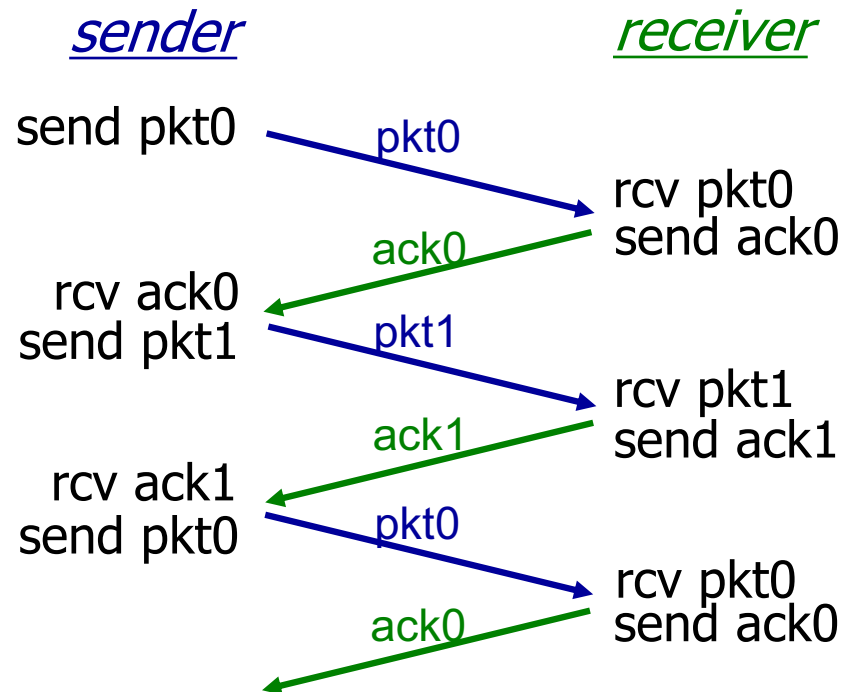
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

approach: sender waits “reasonable” amount of time for ACK

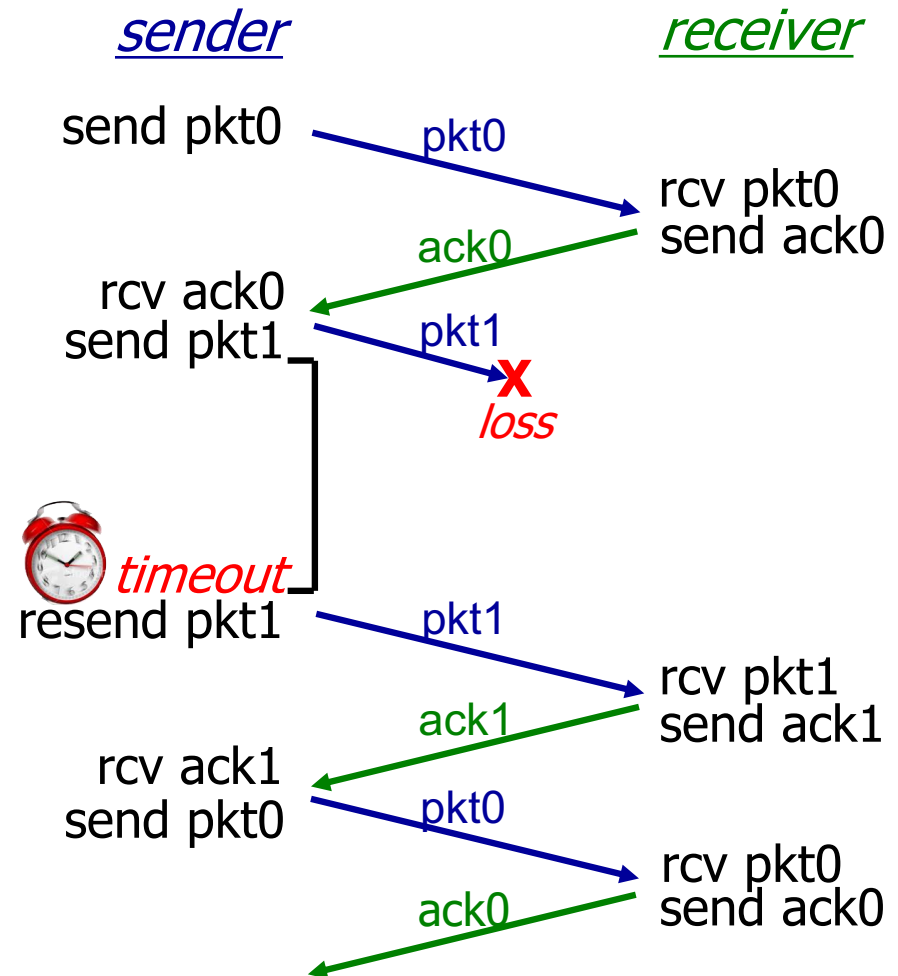
- ❖ retransmits if no ACK received in this time
- ❖ if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- ❖ requires countdown timer



# rdt3.0 in action

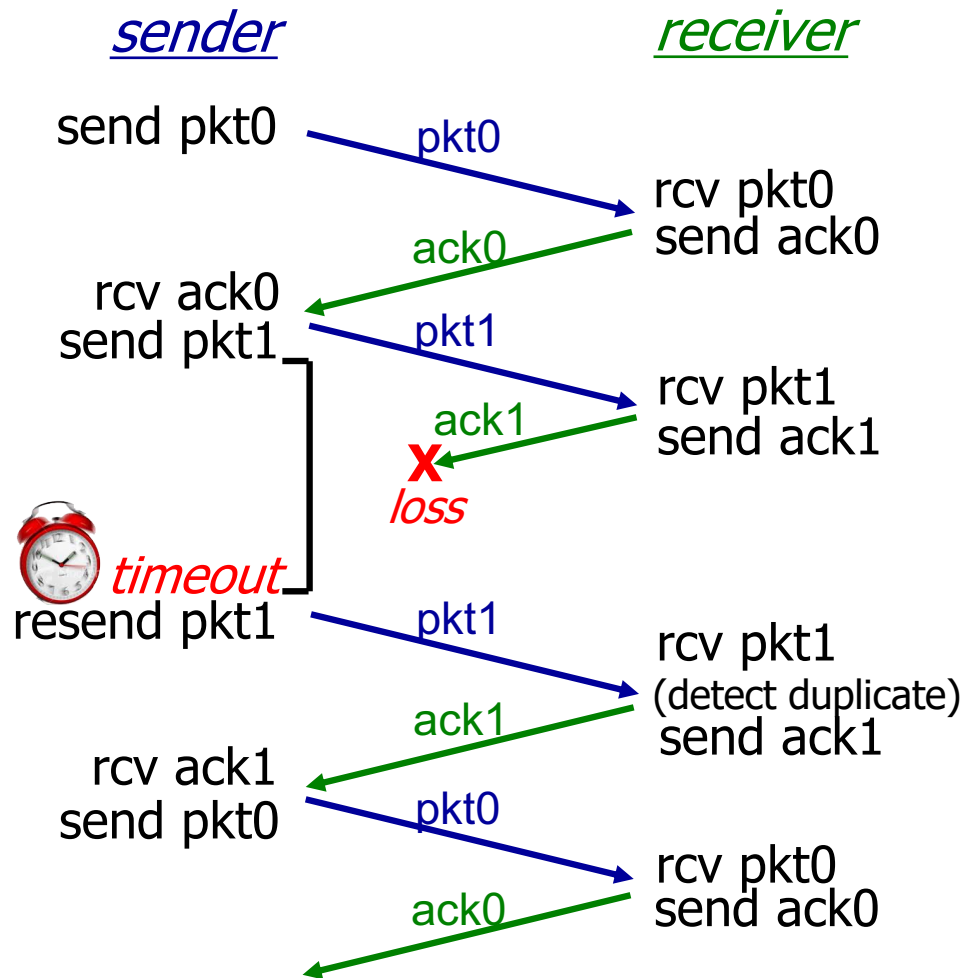


(a) no loss

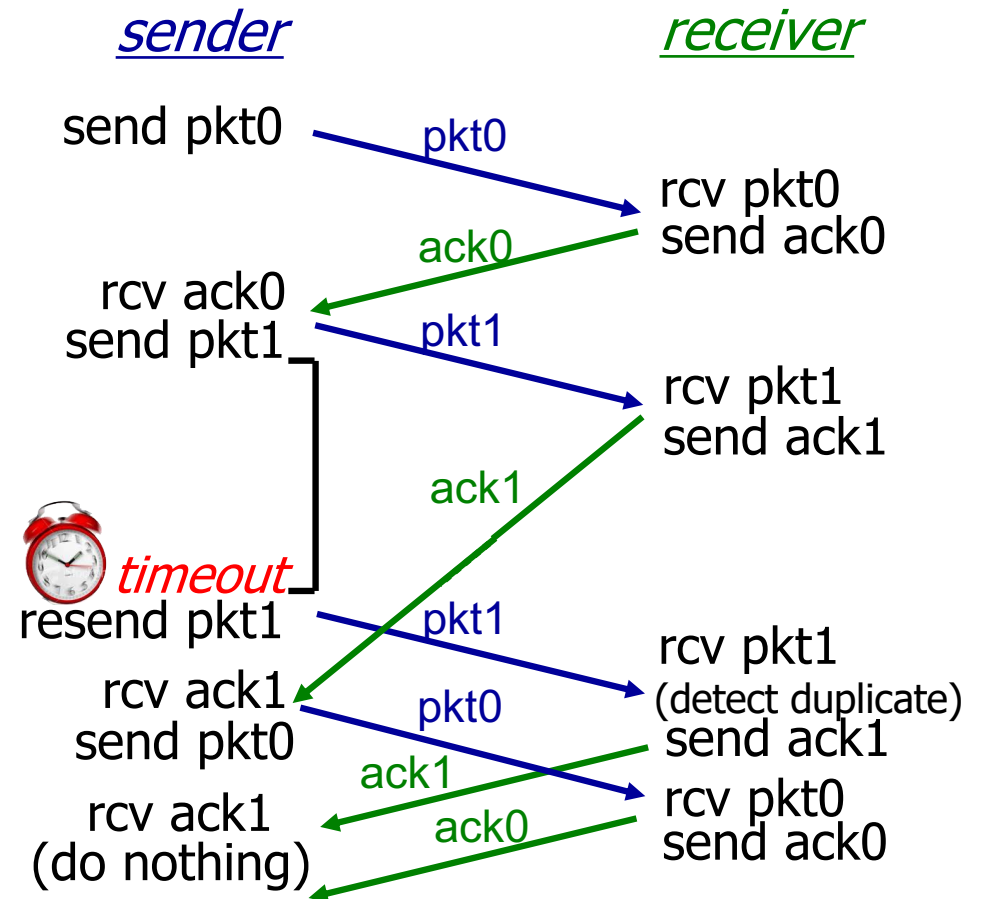


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Quiz: Reliable Data Transfer



- ❖ Which of the following are needed for reliable data transfer with only packet corruption (and no loss or reordering)? Use only as much as is strictly needed.
  - a) Checksums
  - b) Checksums, ACKs, NACKs
  - c) Checksums, ACKs
  - d) Checksums, ACKs, sequence numbers
  - e) Checksums, ACKs, NACKs, sequence numbers

# Quiz: Reliable Data Transfer



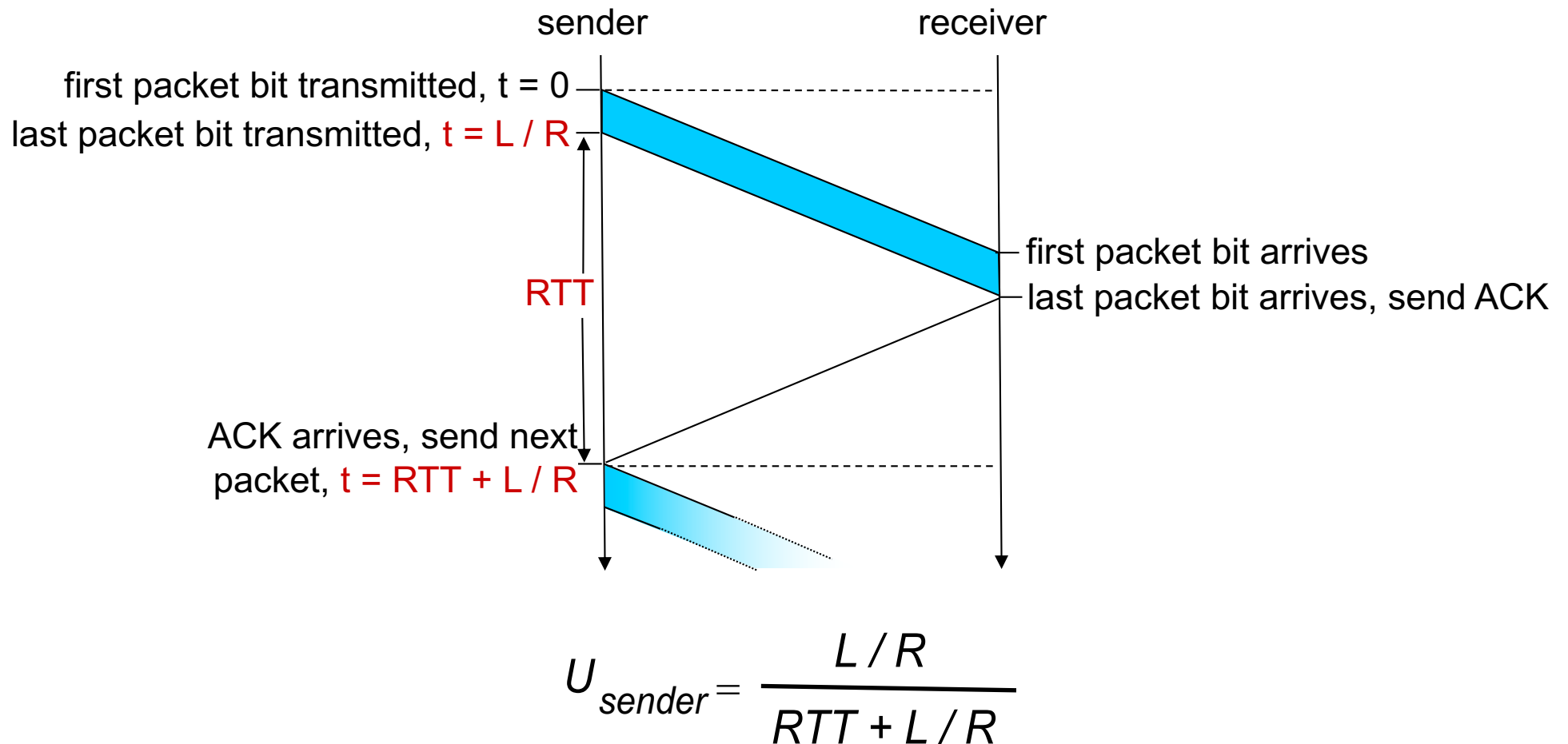
- ❖ If packets (and ACKs and NACKs) could be lost which of the following is true of RDT 2.1 (or 2.2)?
  - a) Reliable in-order delivery is still achieved
  - b) The protocol will get stuck
  - c) The protocol will continue making progress but may skip delivering some messages

# Quiz: Reliable Data Transfer



- ❖ Which of the following are needed for reliable data transfer to handle packet corruption and loss? Use only as much as is strictly needed.
  - a) Checksums, timeouts
  - b) Checksums, ACKs, sequence numbers
  - c) Checksums, ACKs, timeouts
  - d) Checksums, ACKs, timeouts, sequence numbers
  - e) Checksums, ACKs, NACKs, timeouts, sequence numbers

# rdt3.0: stop-and-wait operation



# Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 8000 bit packet and 30msec RTT:

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

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

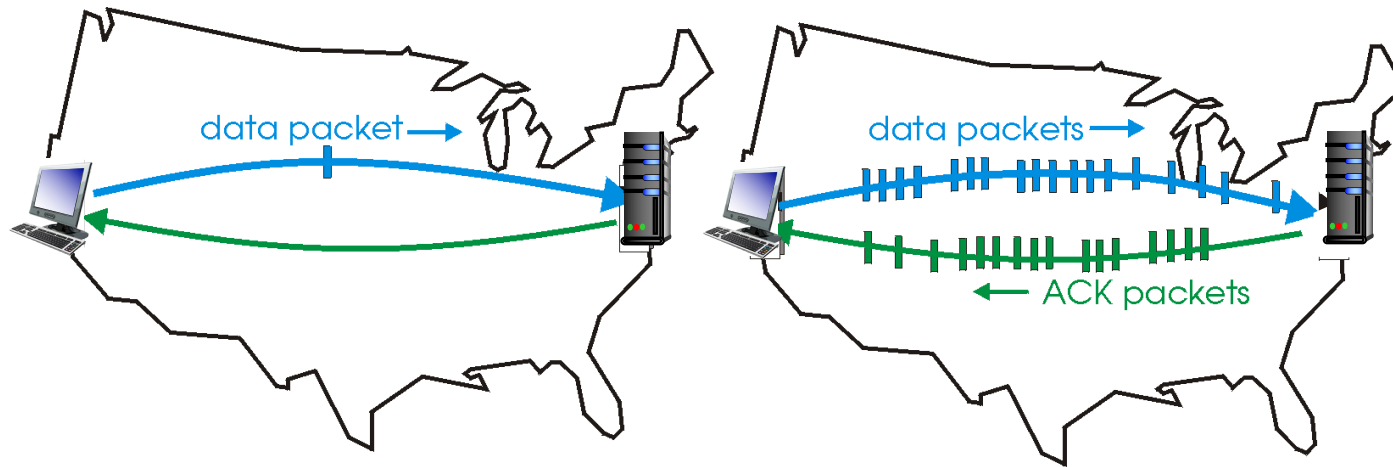
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- RTT=30 msec, 1KB pkt every 30.008 msec: 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



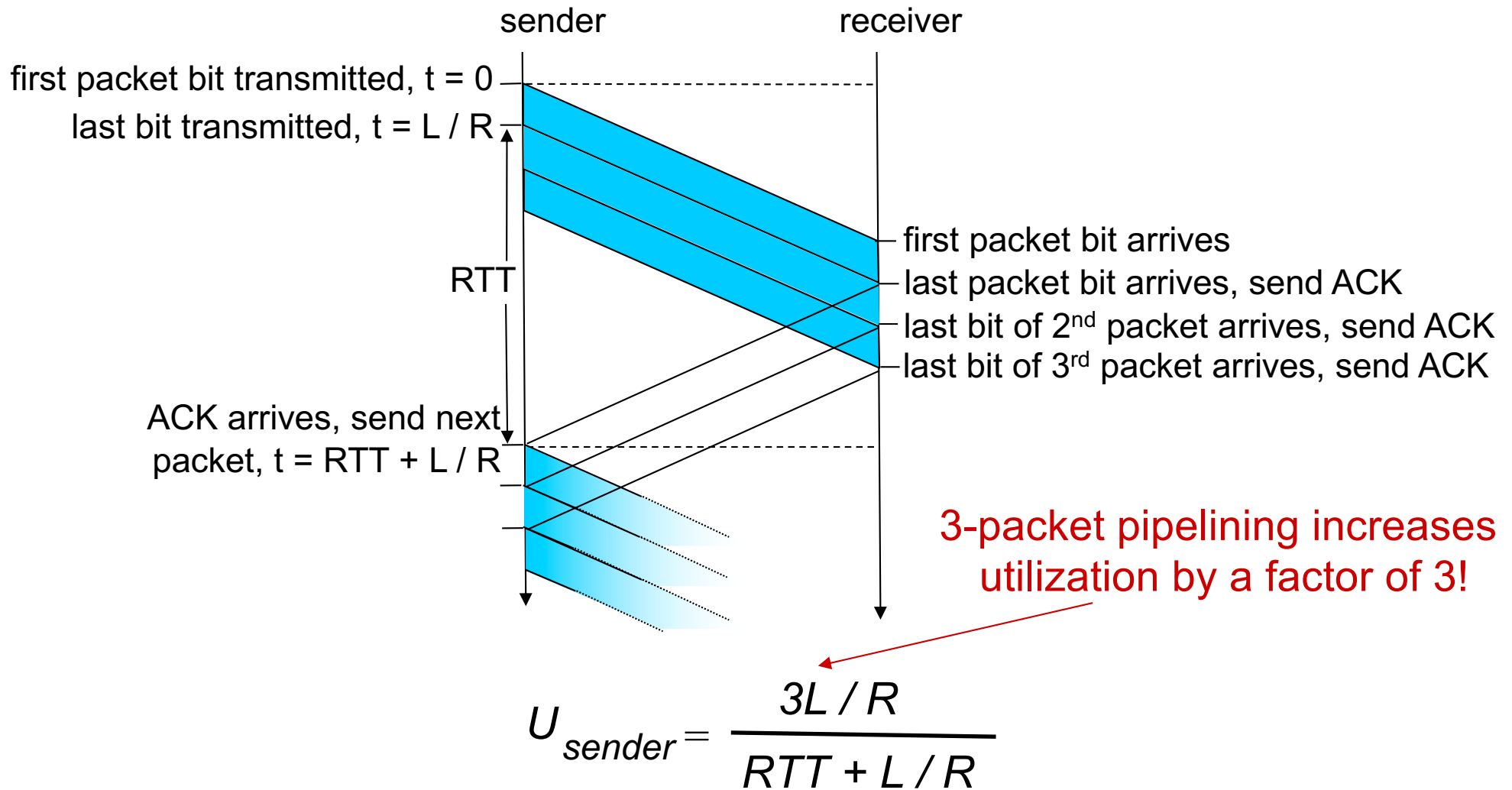
(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- ❖ two generic forms of pipelined (sliding window) protocols: *go-Back-N*, *selective repeat*



# Pipelining: increased utilization



# Pipelined protocols: overview

## Go-Back-N:

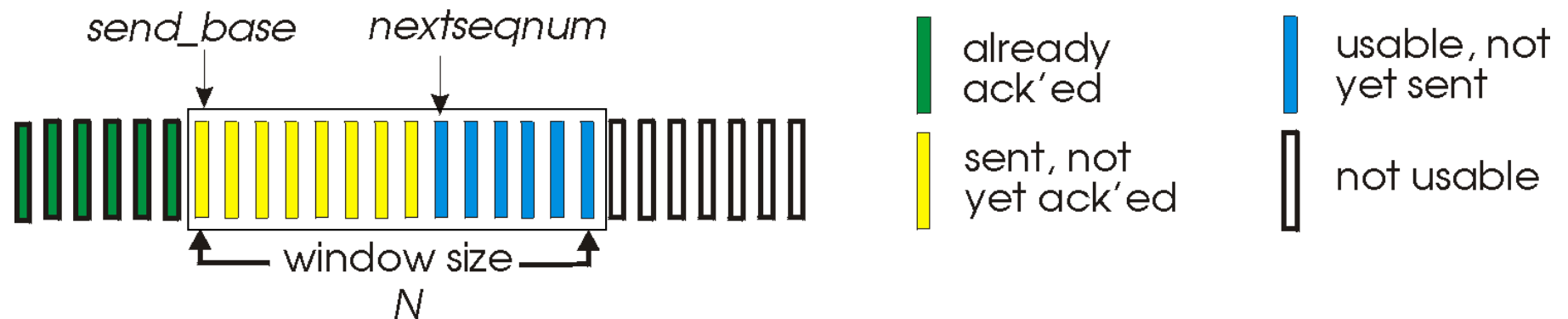
- Sender can have up to N unacked packets in pipeline
- Sender has **single timer** for oldest unacked packet, when timer expires, retransmit *all* unacked packets
- There is no buffer available at Receiver, out of order packets are discarded
- Receiver only sends **cumulative ack**, doesn't ack new packet if there's a gap

## Selective Repeat:

- Sender can have up to N unacked packets in pipeline
- Sender maintains timer for each unacked packet, when timer expires, retransmit only that unacked packet
- Receiver has buffer, can accept out of order packets
- Receiver sends **individual ack** for each packet

# Go-Back-N: sender

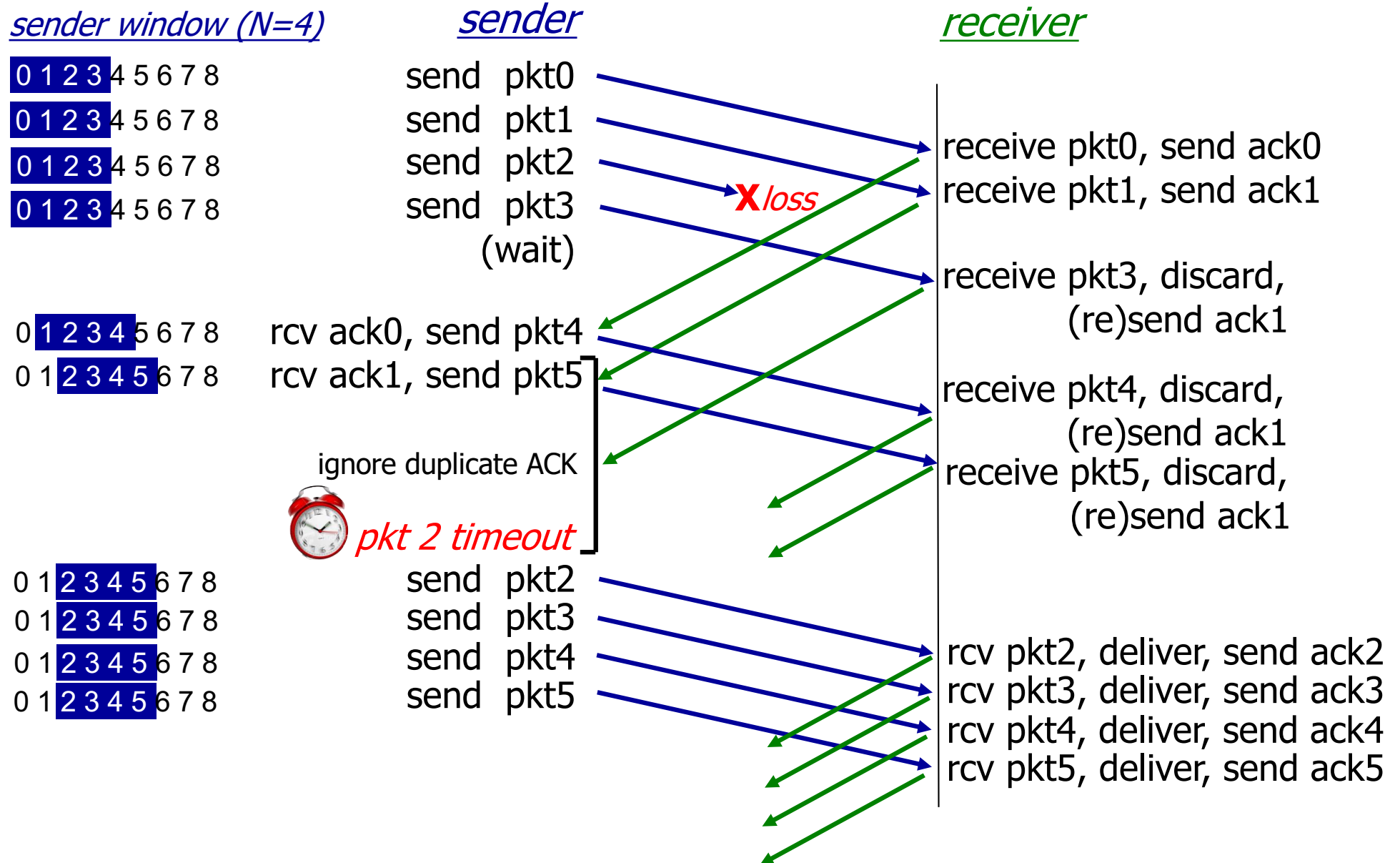
- ❖ k-bit seq # in pkt header
- ❖ “window” of up to N, consecutive unack’ed pkts allowed



- ❖ 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

Applets: [http://media.pearsoncmg.com/aw/aw\\_kurose\\_network\\_2/applets/go-back-n/go-back-n.html](http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html)  
[http://www.ccs-labs.org/teaching/rn/animations/gbn\\_sr/](http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/)

# GBN in action

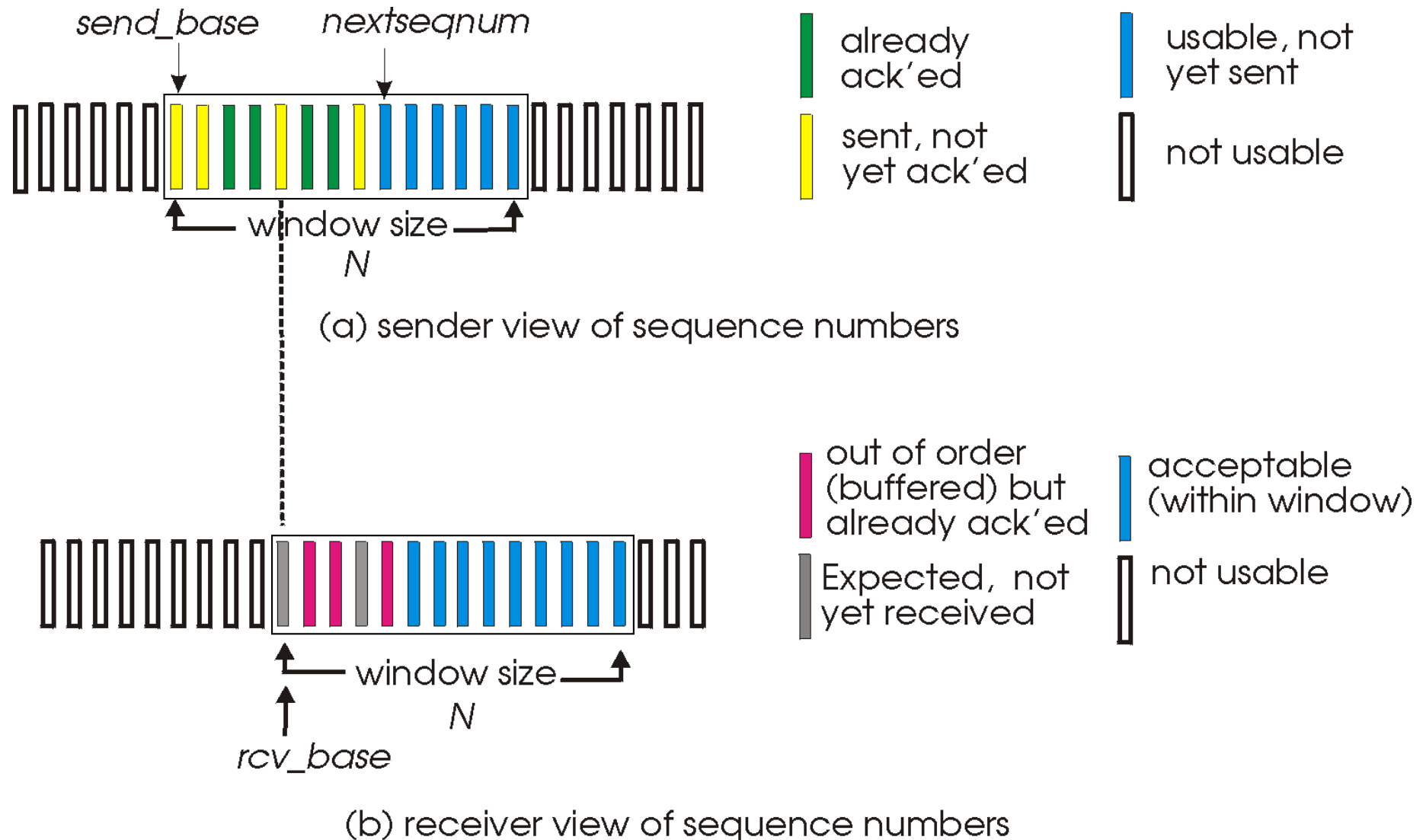


# Selective repeat

- ❖ 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

Applet: [http://media.pearsoncmg.com/aw/aw\\_kurose\\_network\\_3/applets/SelectRepeat/SR.html](http://media.pearsoncmg.com/aw/aw_kurose_network_3/applets/SelectRepeat/SR.html)

# 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 in action

sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived



*pkt 2 timeout*

send pkt2

record ack4 arrived

record ack5 arrived

receiver

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,  
send ack3

receive pkt4, buffer,  
send ack4

receive pkt5, buffer,  
send ack5

rcv pkt2; deliver pkt2,  
pkt3, pkt4, pkt5; send ack2

*Q: what happens when ack2 arrives?*



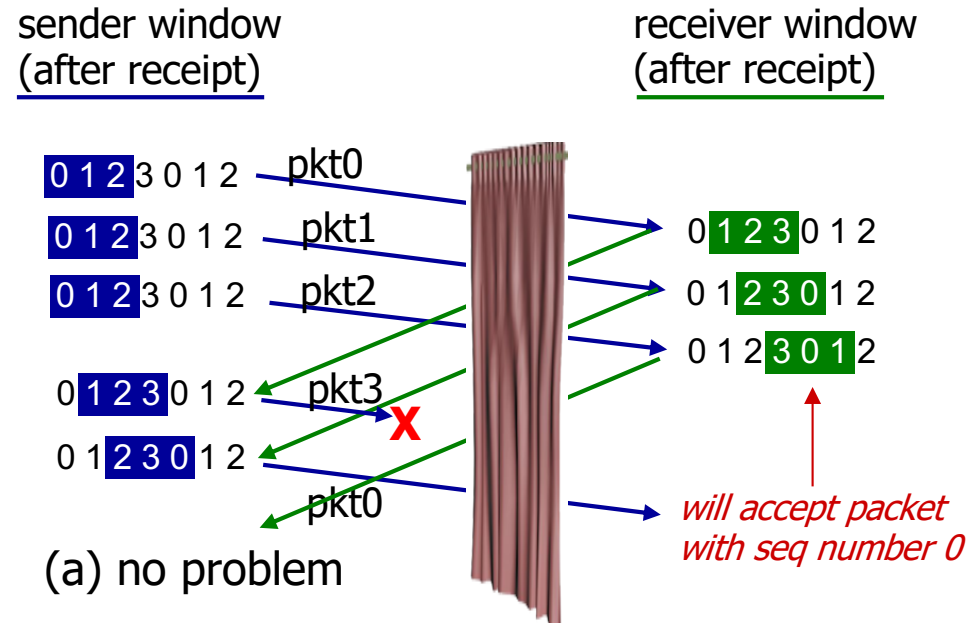
# Selective repeat: dilemma

example:

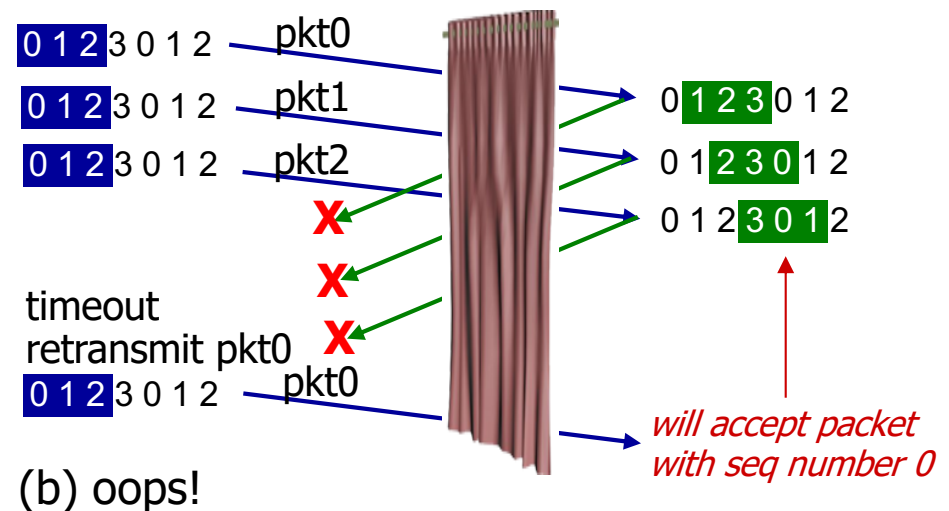
- ❖ seq #'s: 0, 1, 2, 3
- ❖ window size=3
- ❖ receiver sees no difference in two scenarios!
- ❖ duplicate data accepted as new in (b)

**Q:** what relationship between seq # size and window size to avoid problem in (b)?

**A:** Sender window size  $\leq \frac{1}{2}$  of Sequence number space



*receiver can't see sender side.  
receiver behavior identical in both cases!  
something's (very) wrong!*



# Recap: 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

## Quiz: GBN, SR



- ❖ Which of the following is not true?
  - a) GBN uses cumulative ACKs, SR uses individual ACKs
  - b) Both GBN and SR use timeouts to address packet loss
  - c) GBN maintains a separate timer for each outstanding packet
  - d) SR maintains a separate timer for each outstanding packet
  - e) Neither GBN nor SR use NACKs

## Quiz: GBN, SR



- ❖ Suppose a receiver that has received all packets up to and including sequence number 24 and next receives packet 27 and 28. In response, what are the sequence numbers in the ACK(s) sent out by the GBN and SR receiver, respectively?
  - a) [27, 28], [28, 28]
  - b) [24, 24], [27, 28]
  - c) [27, 28], [27, 28]
  - d) [25, 25], [25, 25]
  - e) [nothing], [27, 28]

# Transport Layer 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

# Practical Reliability Questions

- ❖ How do the sender and receiver keep track of outstanding pipelined segments?
- ❖ How many segments should be pipelined?
- ❖ How do we choose sequence numbers?
- ❖ What does connection establishment and teardown look like?
- ❖ How should we choose timeout values?

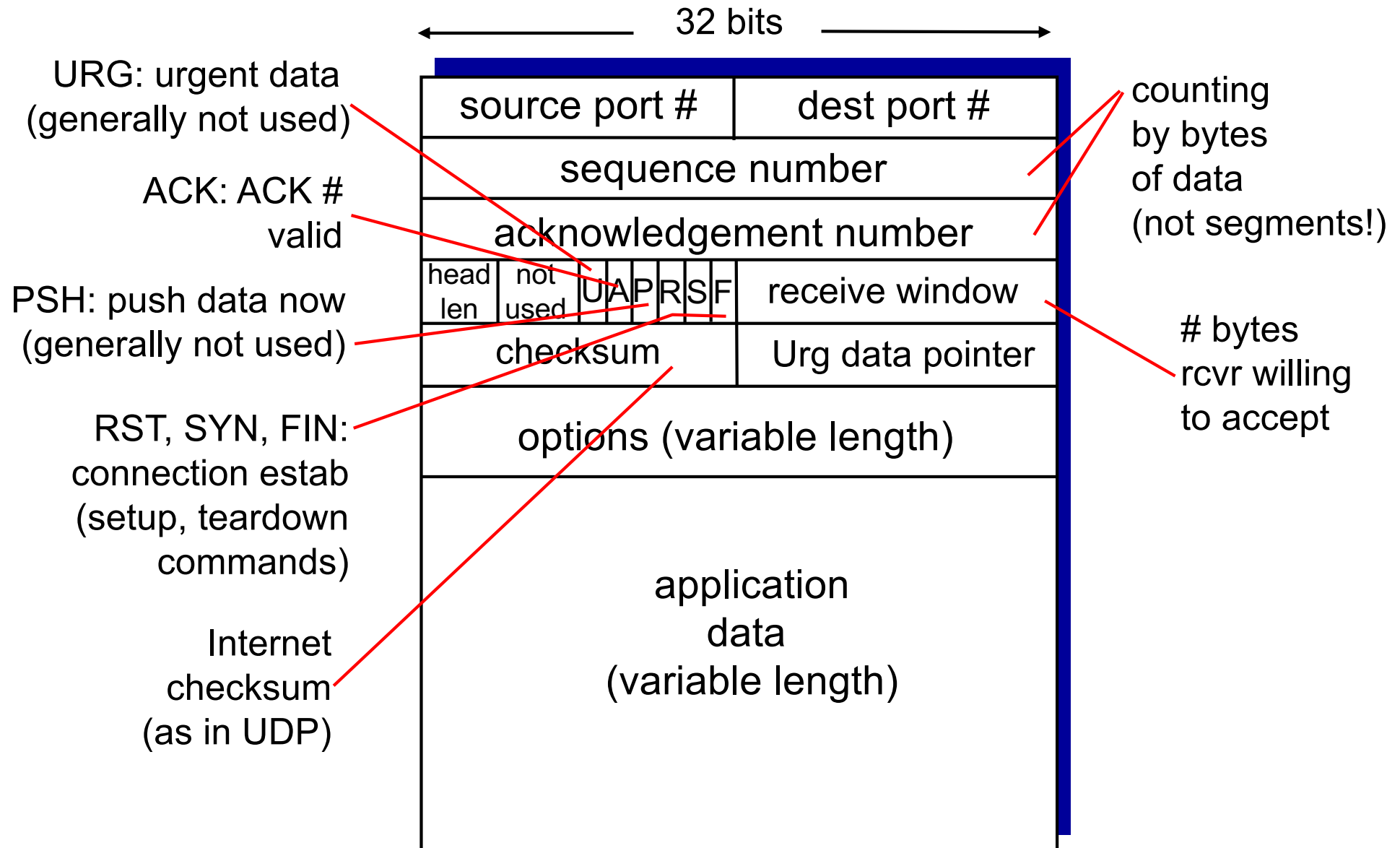
# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- ❖ **point-to-point:**
  - one sender, one receiver
- ❖ **reliable, in-order *byte stream*:**
  - no “message boundaries”
- ❖ **pipelined:**
  - TCP congestion and flow control set window size
- ❖ **send and receive buffers**
- ❖ **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- ❖ **connection-oriented:**
  - handshaking (exchange of control msgs) initializes sender, receiver state before data exchange
- ❖ **flow controlled:**
  - sender will not overwhelm receiver

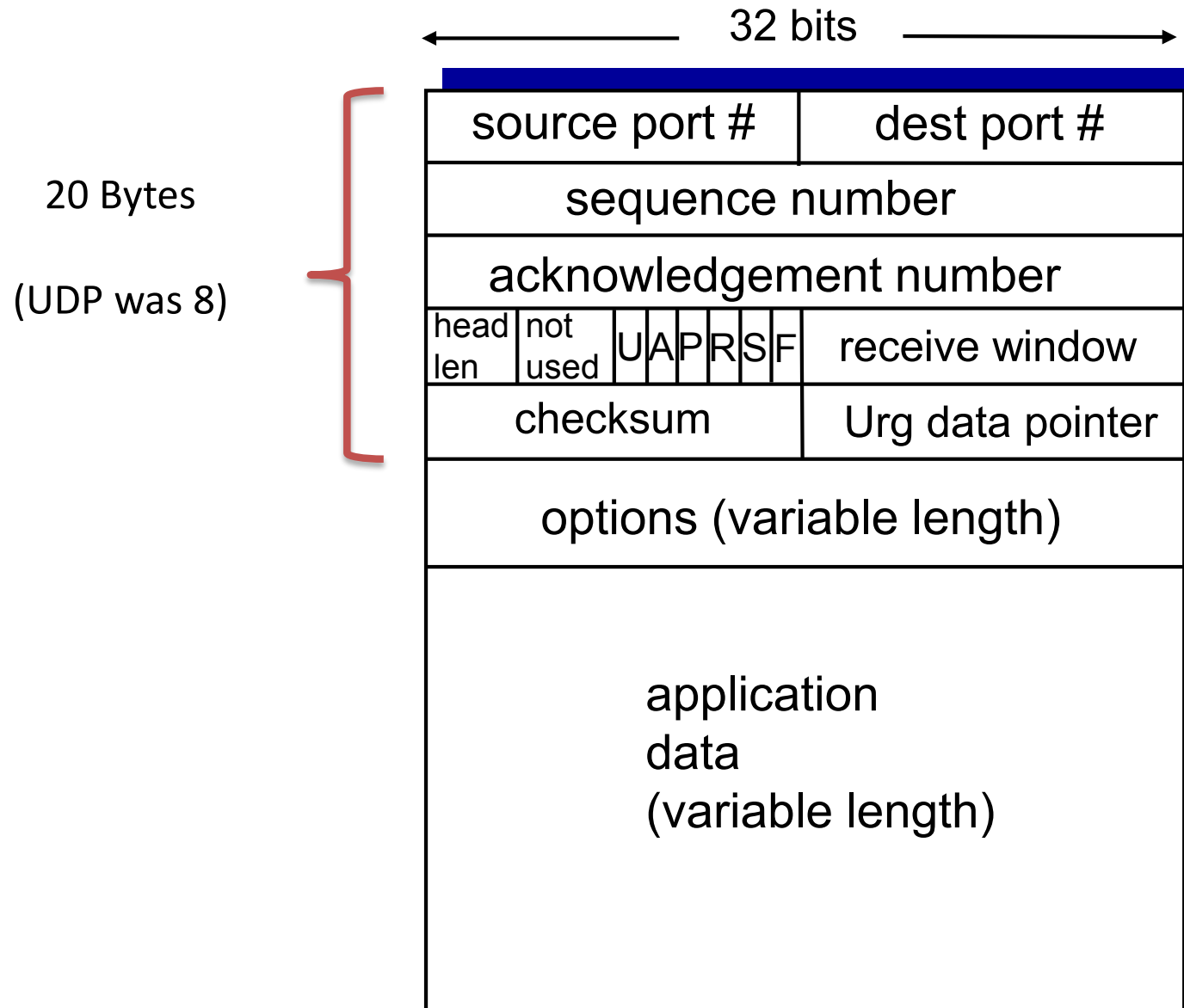


# TCP segment structure





# TCP segment structure



# Transport Layer 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

# Recall: Components of a solution for reliable transport

- ❖ Checksums (for error detection)
- ❖ Timers (for loss detection)
- ❖ Acknowledgments
  - cumulative
  - selective
- ❖ Sequence numbers (duplicates, windows)
- ❖ Sliding Windows (for efficiency)
  - Go-Back-N (GBN)
  - Selective Repeat (SR)

# What does TCP do?

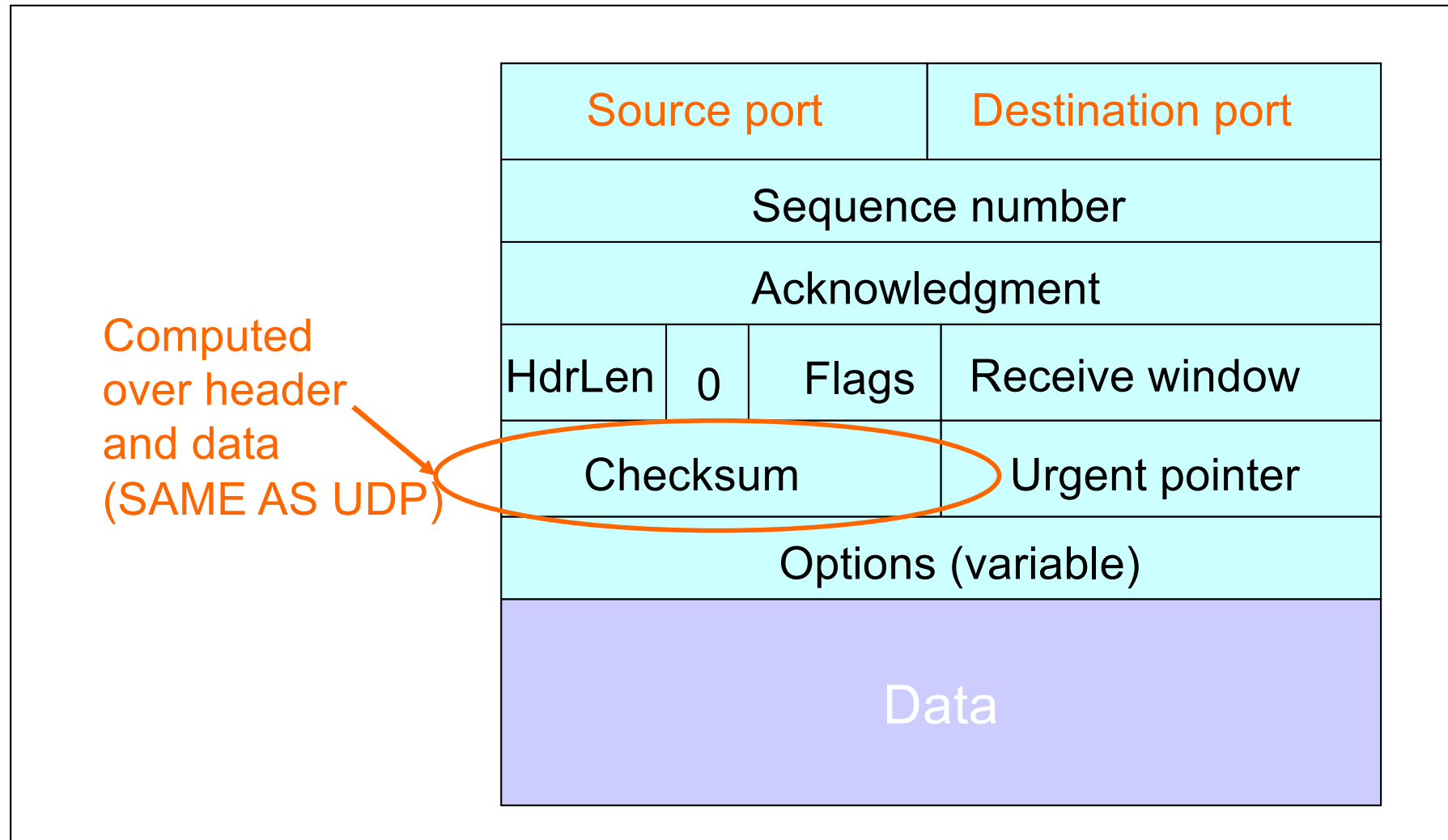
---

Many of our previous ideas, but some key differences

- ❖ Checksum

# TCP Header

---



# What does TCP do?

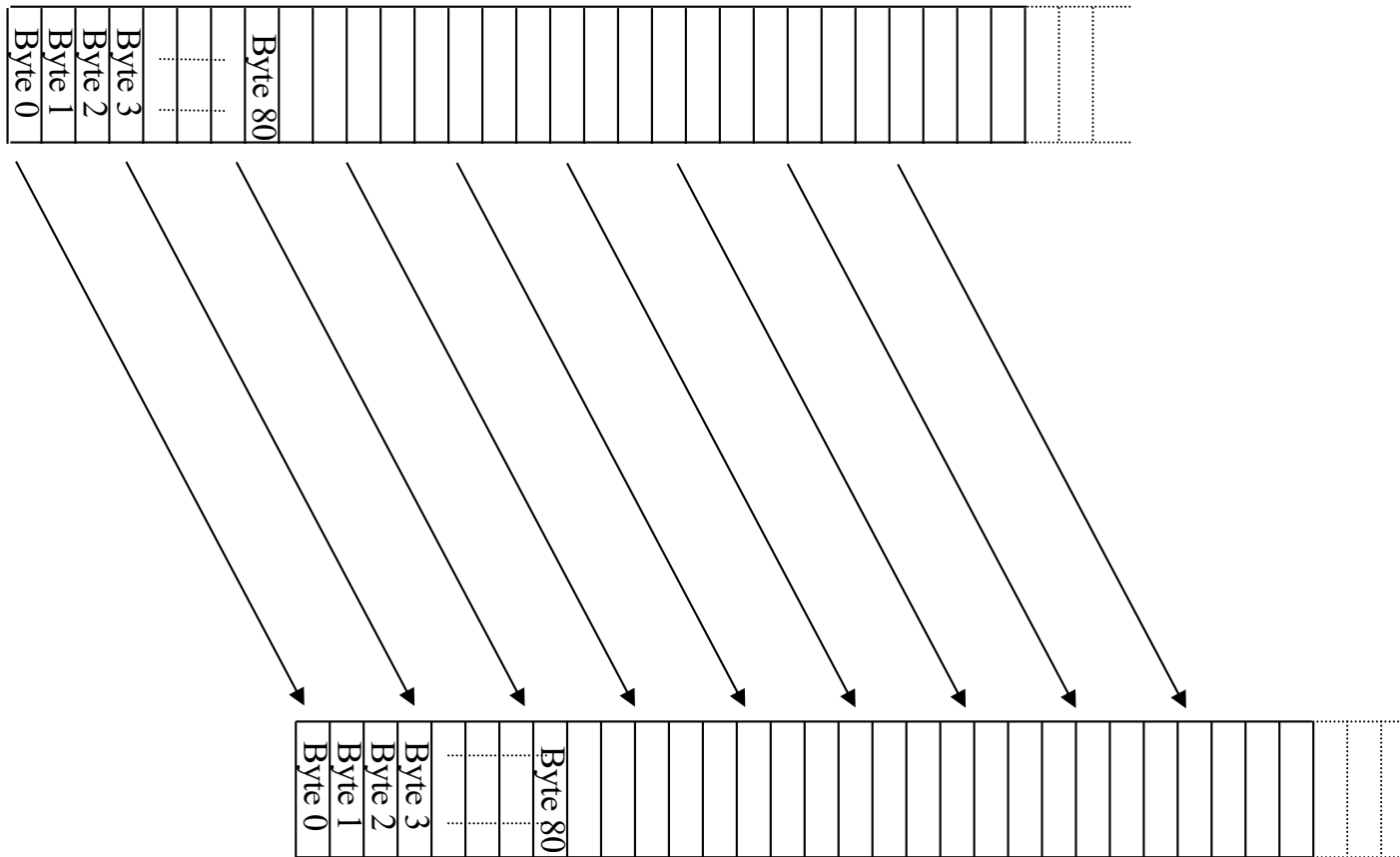
---

Many of our previous ideas, but some key differences

- ❖ Checksum
- ❖ **Sequence numbers are byte offsets**

# TCP “Stream of Bytes” Service ..

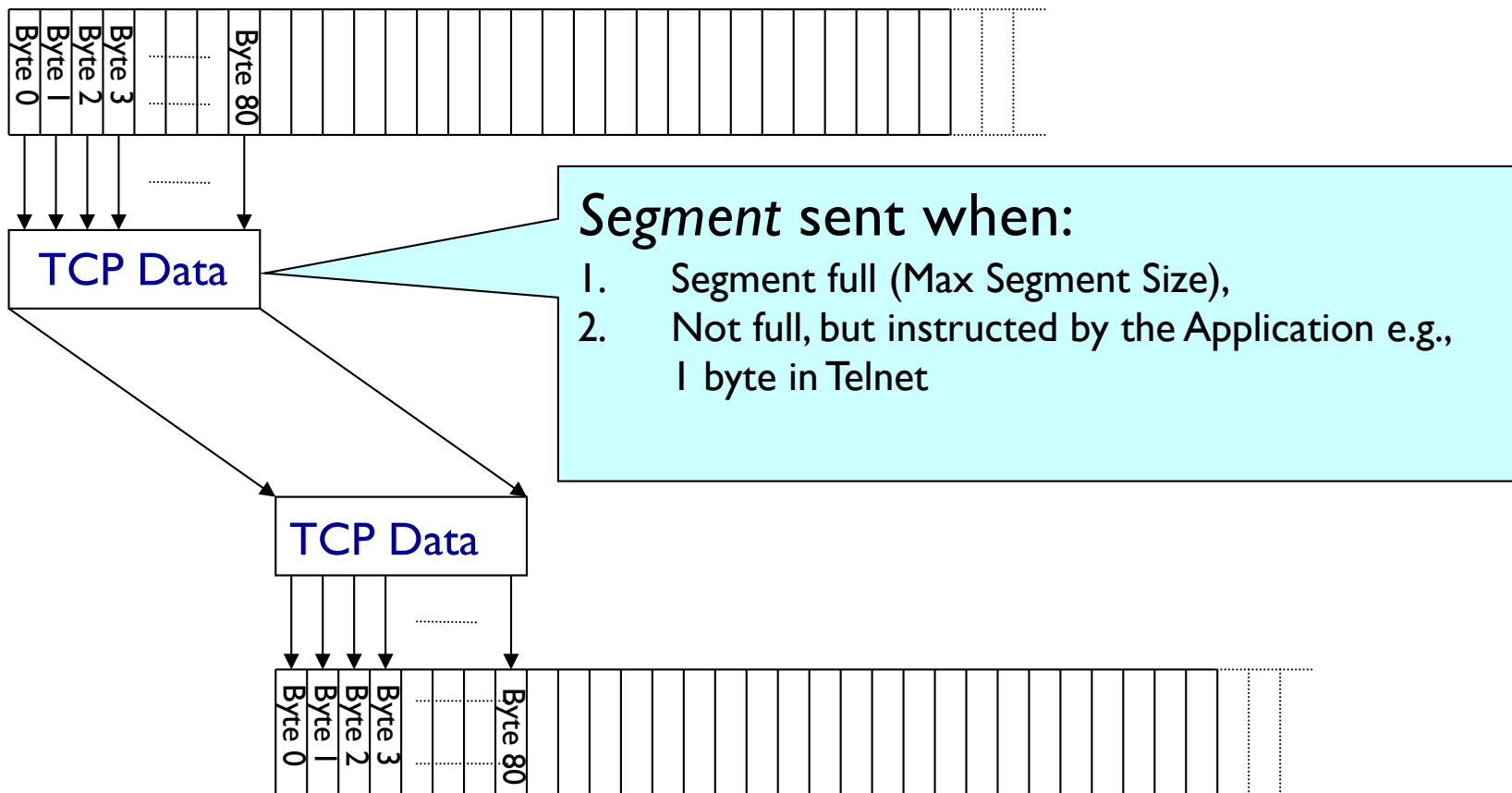
Application @ Host A



Application @ Host B

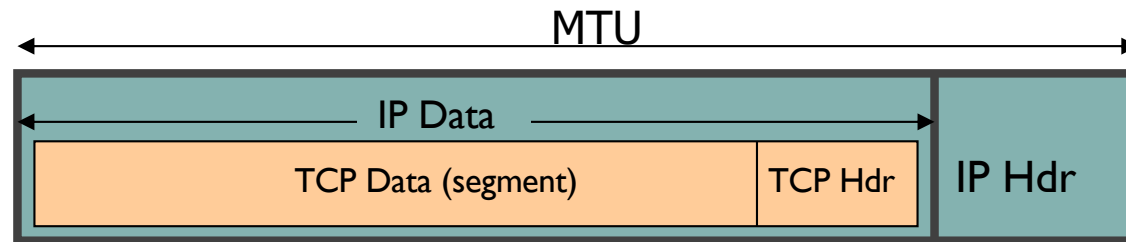
# .. Provided Using TCP “Segments”

Host A





# TCP Maximum Segment Size



## ❖ IP packet

- No bigger than Maximum Transmission Unit (MTU)
- E.g., up to 1500 bytes with Ethernet

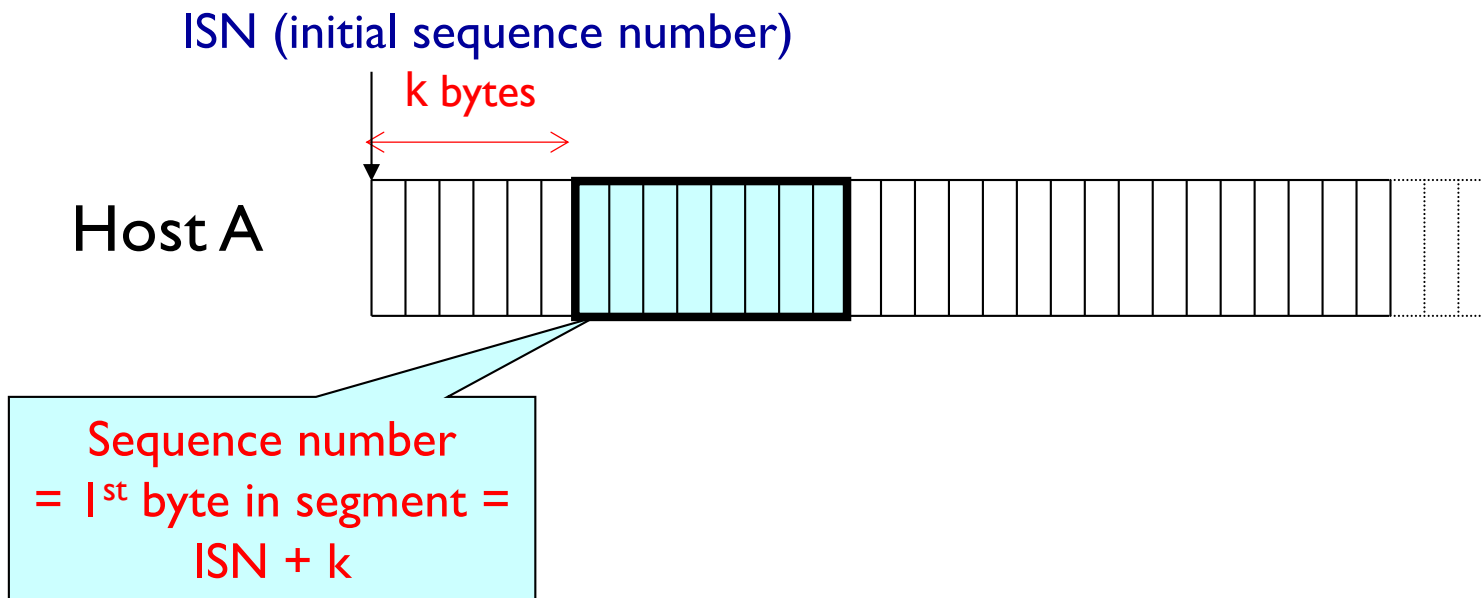
## ❖ TCP packet

- IP packet with a TCP header and data inside
- TCP header  $\geq 20$  bytes long

## ❖ TCP segment

- No more than Maximum Segment Size (MSS) bytes
- E.g., up to 1460 consecutive bytes from the stream
- $MSS = MTU - 20 \text{ (min IP header)} - 20 \text{ (min TCP header)}$

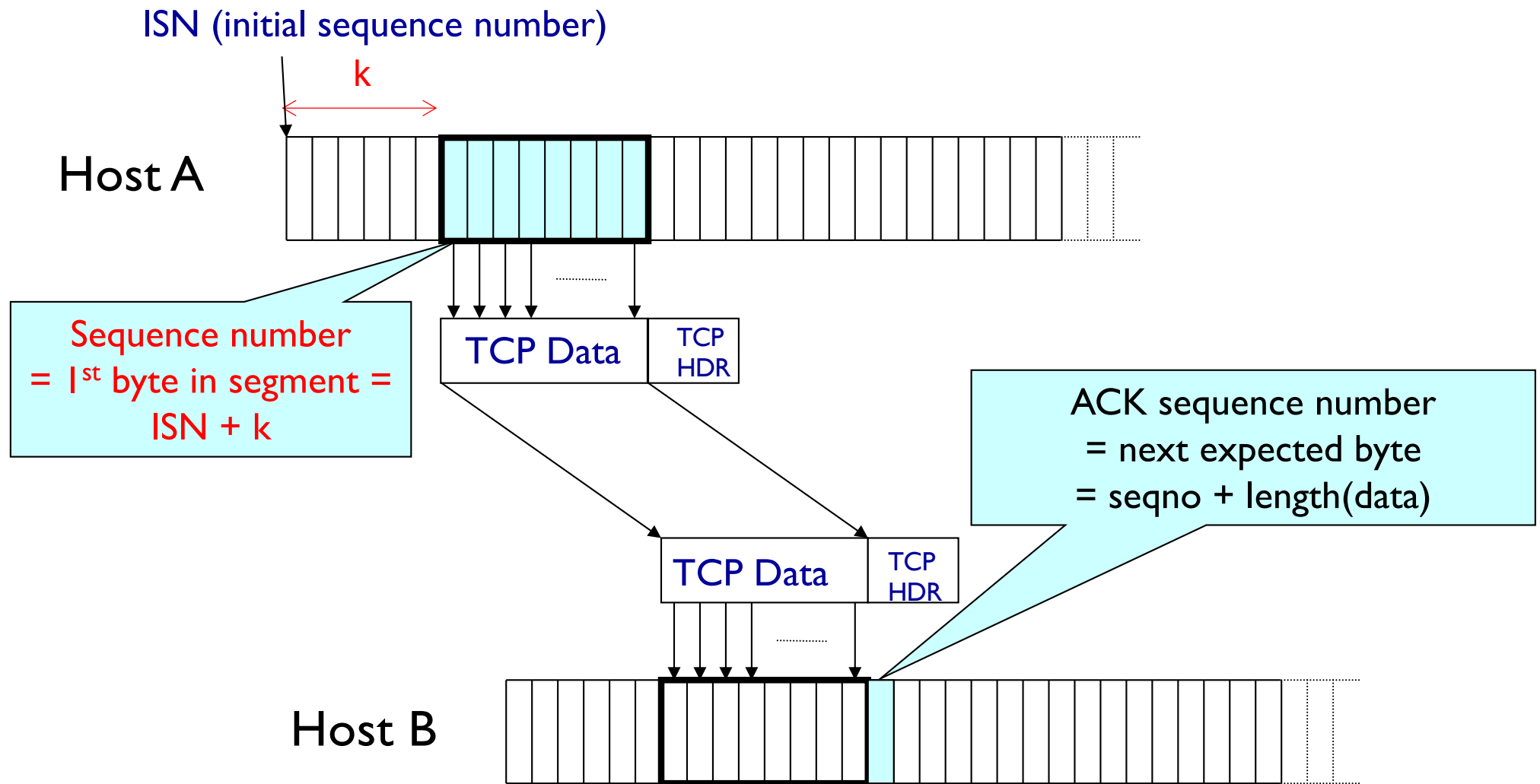
# Sequence Numbers



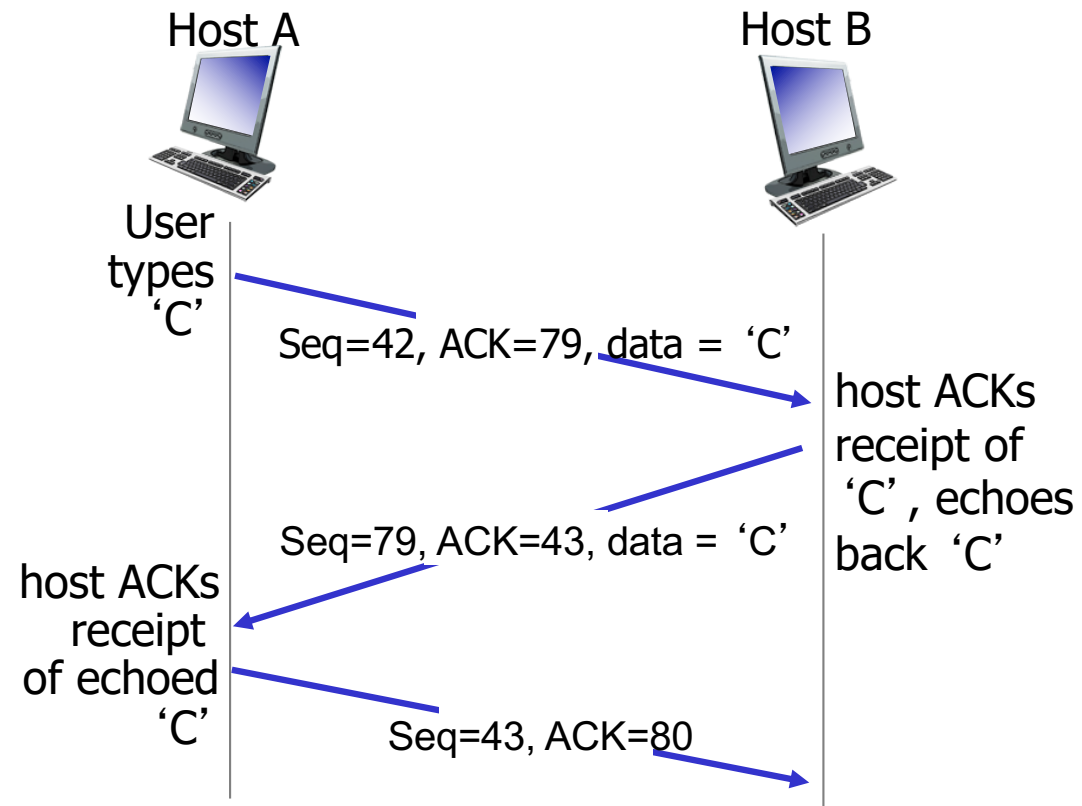
## Sequence numbers:

- byte stream “number” of first byte in segment's data

# Sequence & Ack Numbers



# TCP seq. numbers, ACKs



simple telnet scenario

# Transport Part I: Summary

- ❖ principles behind transport layer services:
  - multiplexing, demultiplexing
  - UDP
  - reliable data transfer
  - Pipelined Protocols for reliable data transfer

## ❖ **Next Week:**

- TCP
  - TCP Flow Control
  - TCP Connection Management