

# Computer Networks and Applications

COMP 3331/COMP 9331

Week 4

Major Concepts:

1. Multiplexing-demultiplexing
2. Checksum
3. Reliable data transfer

## Transport Layer Part 1

Reading Guide:  
Chapter 3, Sections 3.1 – 3.5.2

## Transport layer: overview

### *Our goal:*

- 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

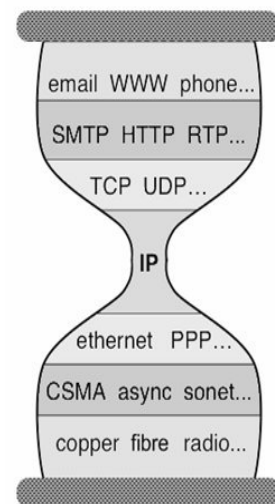
# Transport layer: roadmap

- ❖ **Transport-layer services**
- ❖ Multiplexing and demultiplexing
- ❖ Connectionless transport: UDP
- ❖ Principles of reliable data transfer
- ❖ Connection-oriented transport: TCP
- ❖ Principles of congestion control
- ❖ TCP congestion control
- ❖ Evolution of transport-layer functionality

3

## Transport layer

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



4

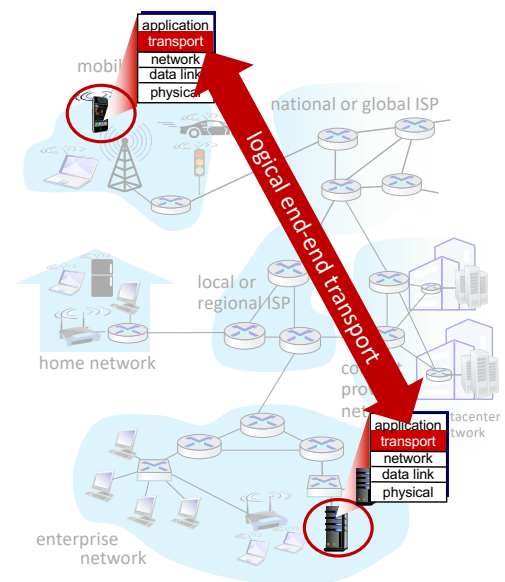
# Network layer (some context)

- ❖ 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 transmission 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!!)

5

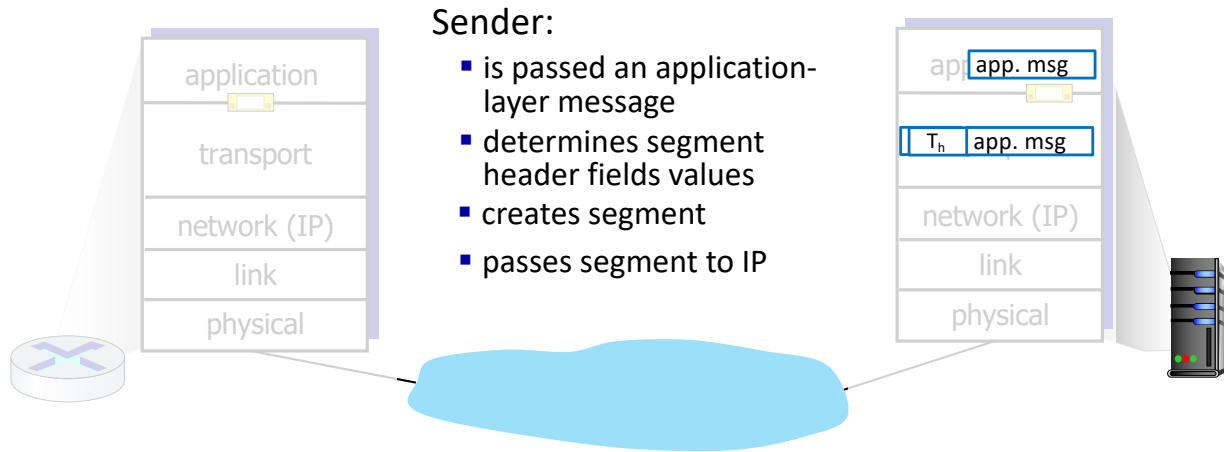
## Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



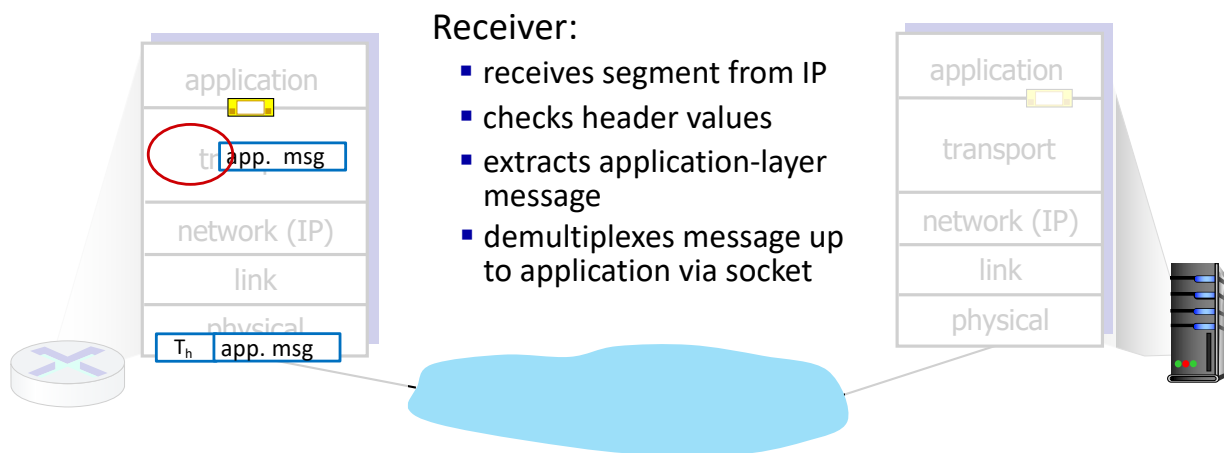
6

# Transport Layer Actions



7

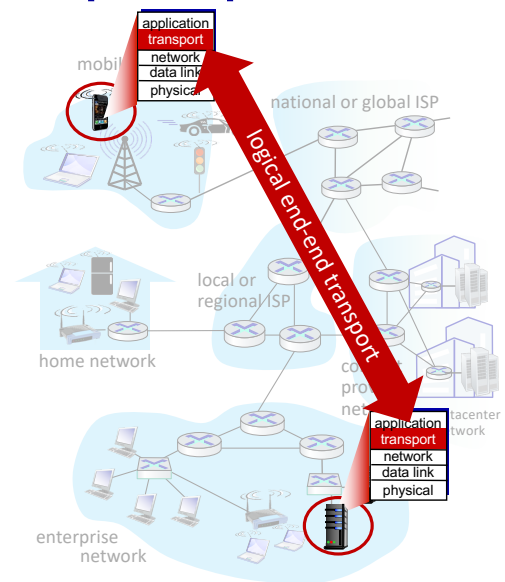
# Transport Layer Actions



8

# Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

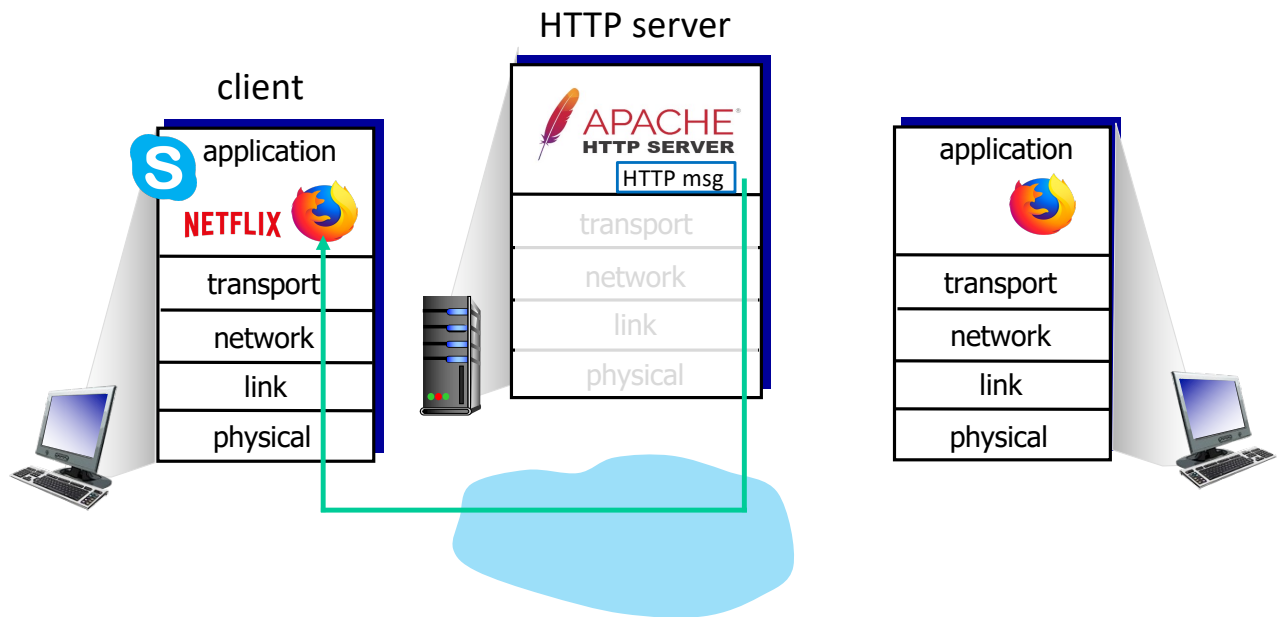


9

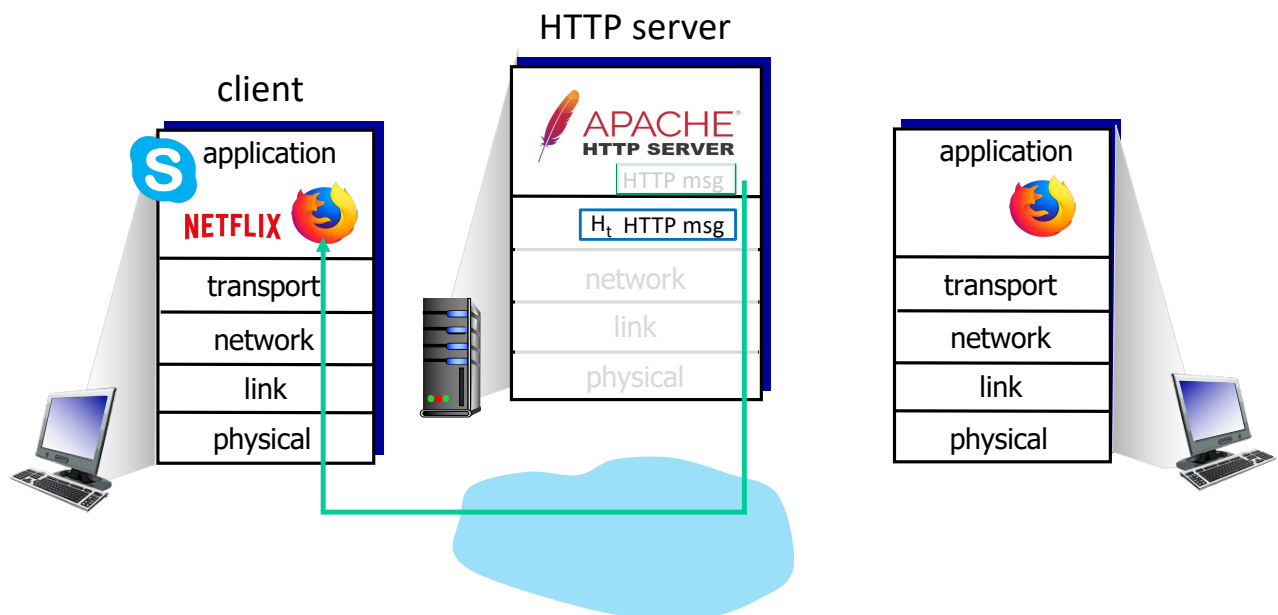
## Transport layer: roadmap

- ❖ Transport-layer services
- ❖ **Multiplexing and demultiplexing**
- ❖ Connectionless transport: UDP
- ❖ Principles of reliable data transfer
- ❖ Connection-oriented transport: TCP
- ❖ Principles of congestion control
- ❖ TCP congestion control
- ❖ Evolution of transport-layer functionality

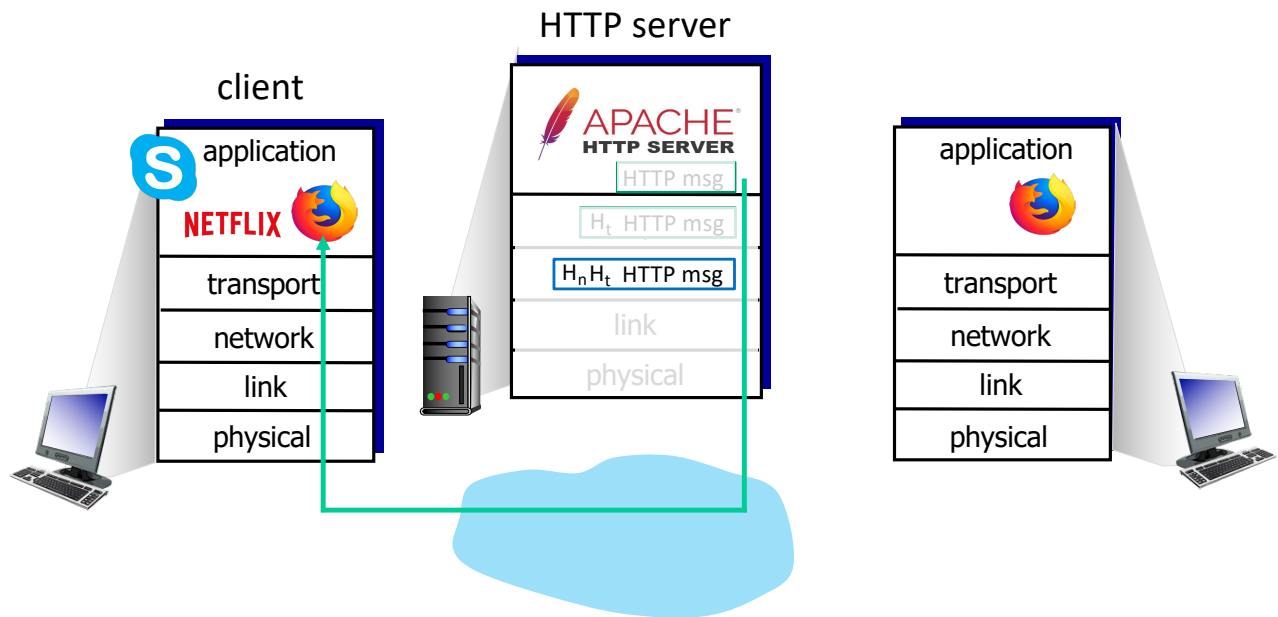
10



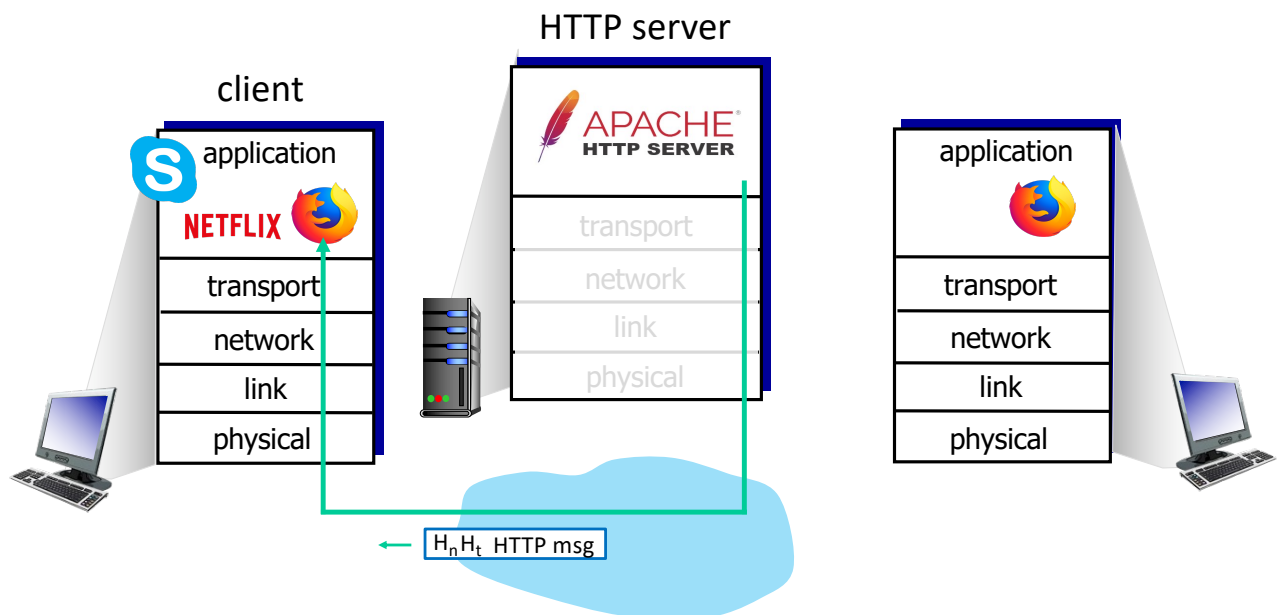
11



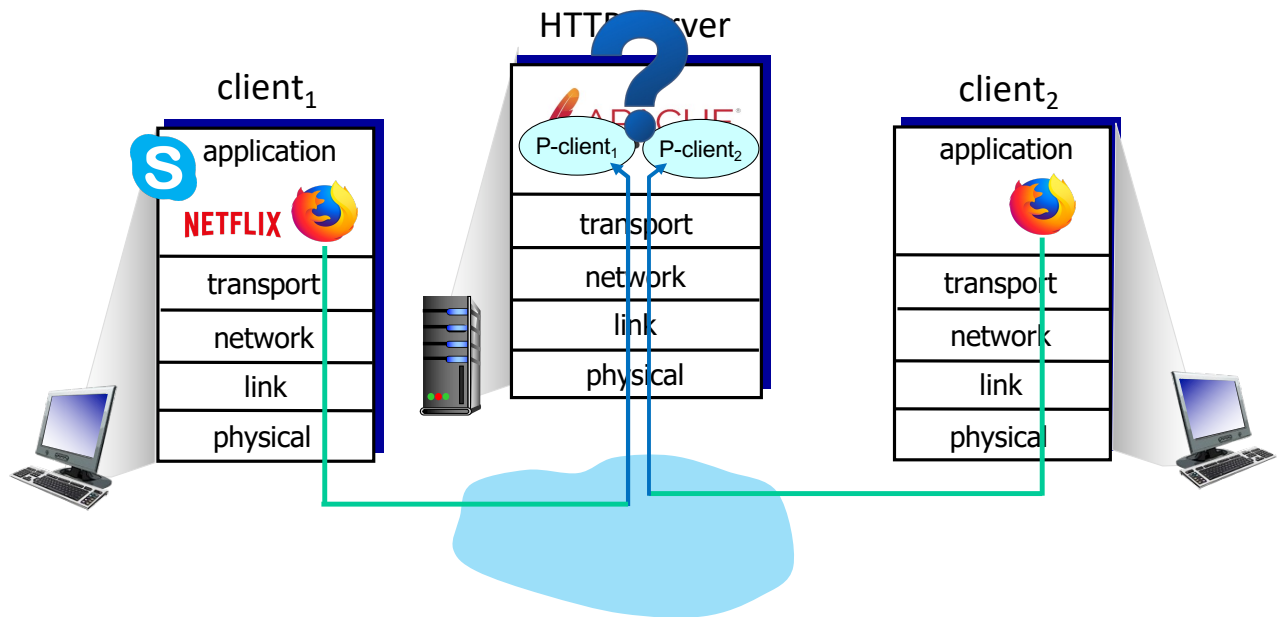
12



13



14



15

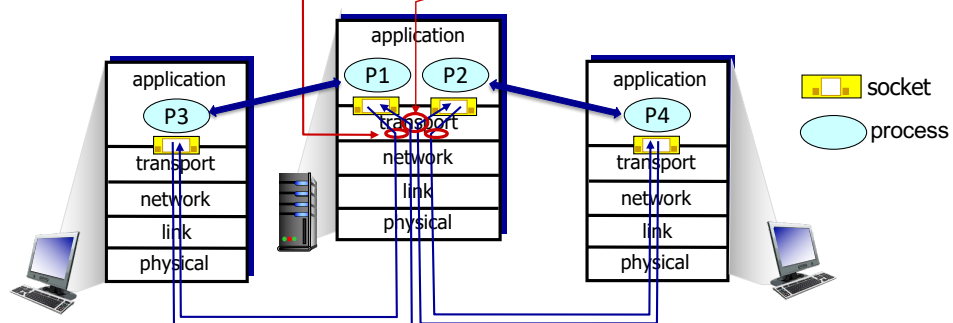
## Multiplexing/demultiplexing

### *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



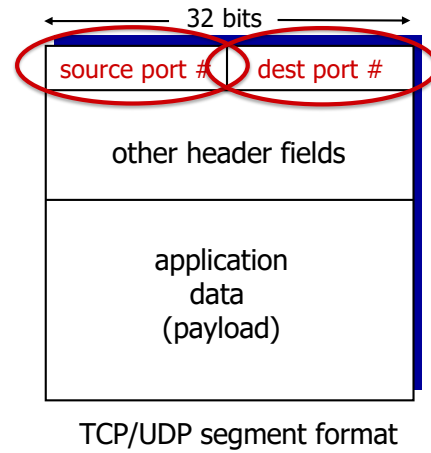
**Note:** The network is a shared resource. It does not care about your applications, sockets, etc.

16



# How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address (not shown in the pic)
  - 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



17

## Connectionless demultiplexing (UDP)

*Recall:*

- when creating socket, either specify *host-local* port # (or let OS pick random available port):

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

- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

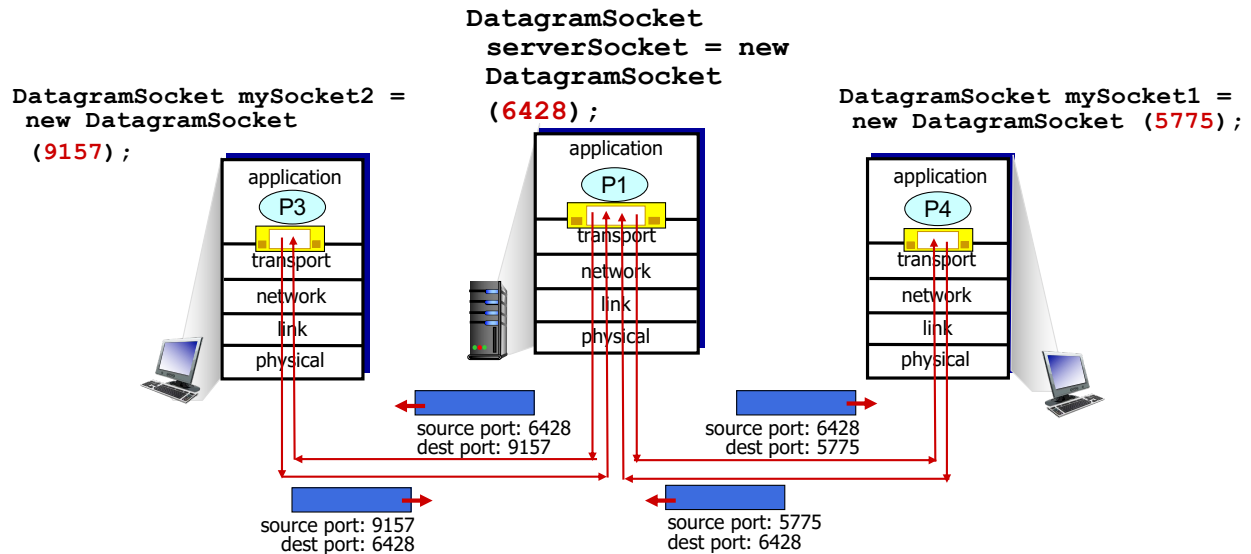
- checks destination port # in segment
- directs UDP segment to socket with that port #



IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

18

## (UDP) Connectionless demultiplexing: an example



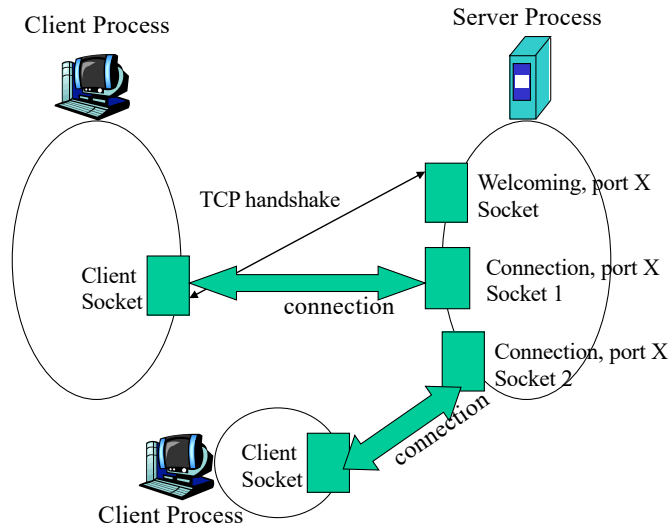
19

## Connection-oriented demultiplexing (TCP)

- TCP socket identified by **4-tuple**:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket
- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

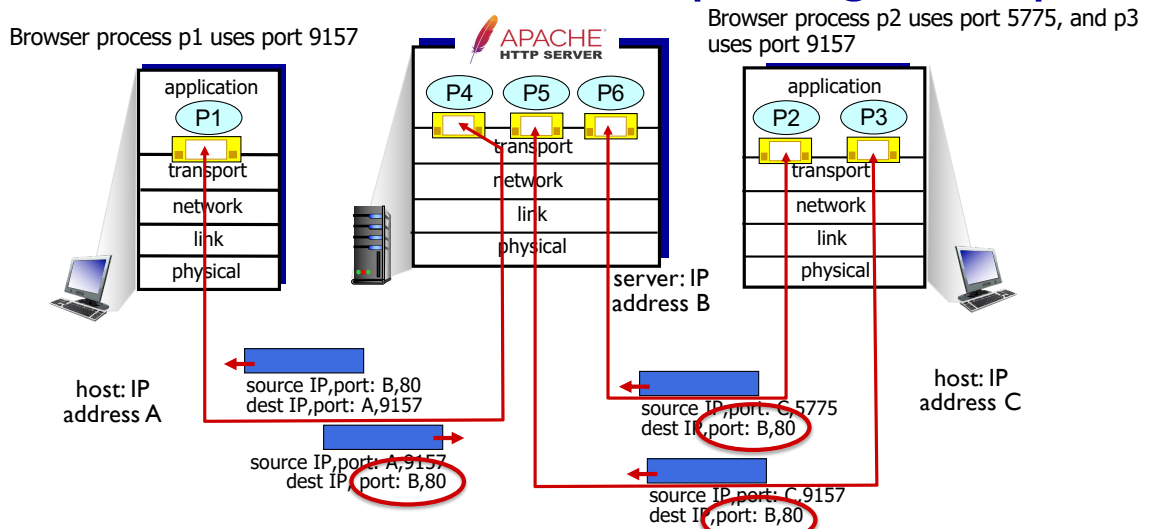
20

# Revisiting TCP Sockets



21

## Connection-oriented demultiplexing: example



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

22

# Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination IP and port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers (more later in the course)

23

## 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>

24

## Quiz: UDP Sockets



Suppose we use UDP instead of TCP for communicating with 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 active at the server and each client?

- a) 1, 1
- b) 2, 1
- c) 200, 2
- d) 100, 1
- e) 101, 1

25

## Quiz: TCP Sockets



Suppose 100 clients are simultaneously communicating with a traditional HTTP/TCP web server. How many sockets are active respectively at the server and each client?

- a) 1, 1
- b) 2, 1
- c) 200, 2
- d) 100, 1
- e) 101, 1

26

## Quiz: TCP Sockets



Suppose 100 clients are simultaneously communicating with a traditional HTTP/TCP web server. Do all the TCP sockets at the server have the same server-side port number?

- a) Yes
- b) No

27

## Transport layer: roadmap

- ❖ Transport-layer services
- ❖ Multiplexing and demultiplexing
- ❖ **Connectionless transport: UDP**
- ❖ Principles of reliable data transfer
- ❖ Connection-oriented transport: TCP
- ❖ Principles of congestion control
- ❖ TCP congestion control
- ❖ Evolution of transport-layer functionality

28

# UDP: User Datagram Protocol

- “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 RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

29

# UDP: User Datagram Protocol

- Applications that use UDP:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

30

# UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD  
RFC 768 J. Postel  
ISI  
28 August 1980

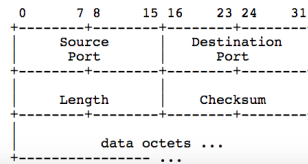
## User Datagram Protocol

### Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

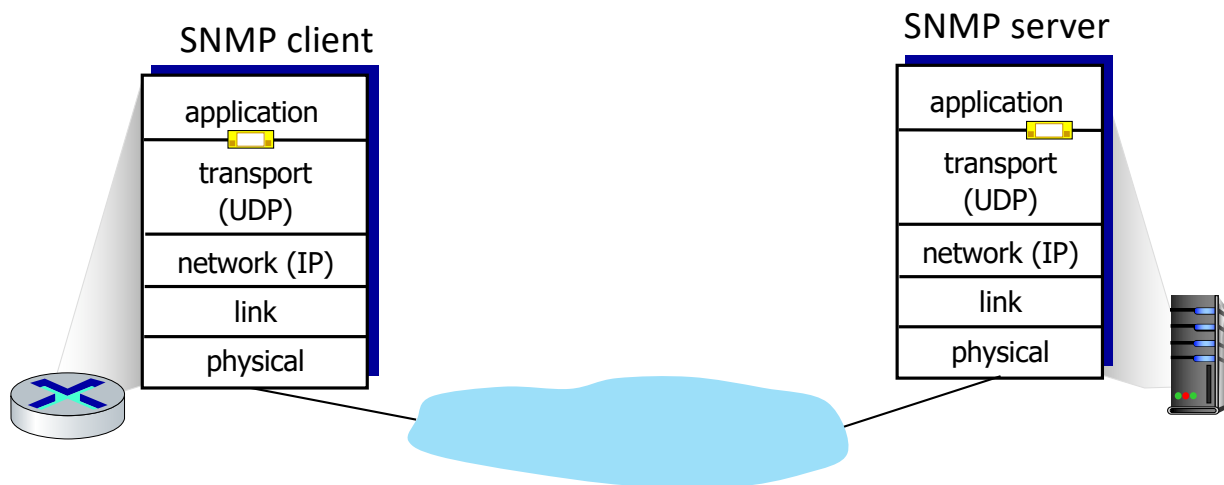
This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

### Format



31

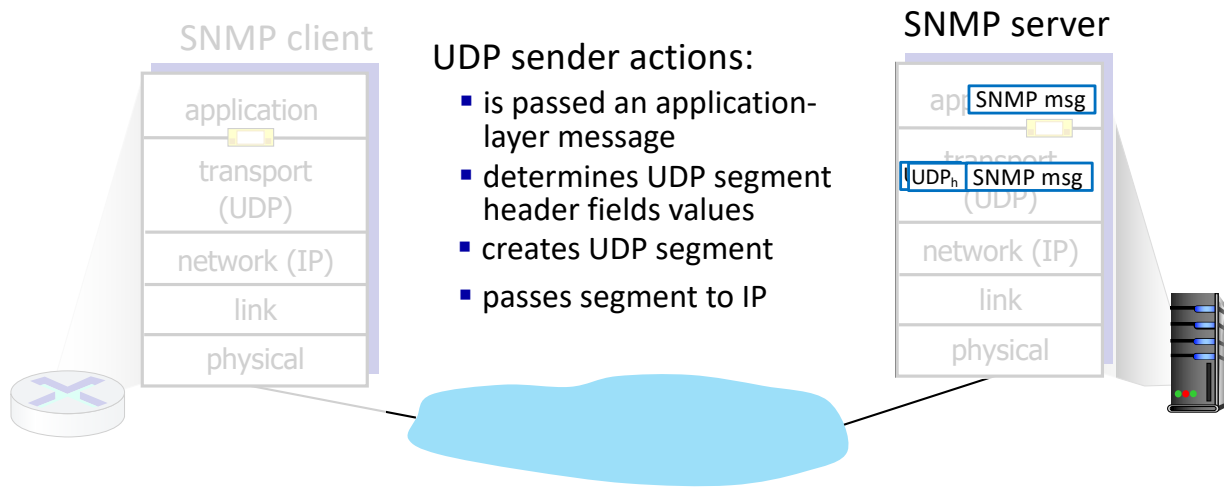
# UDP: Transport Layer Actions



32

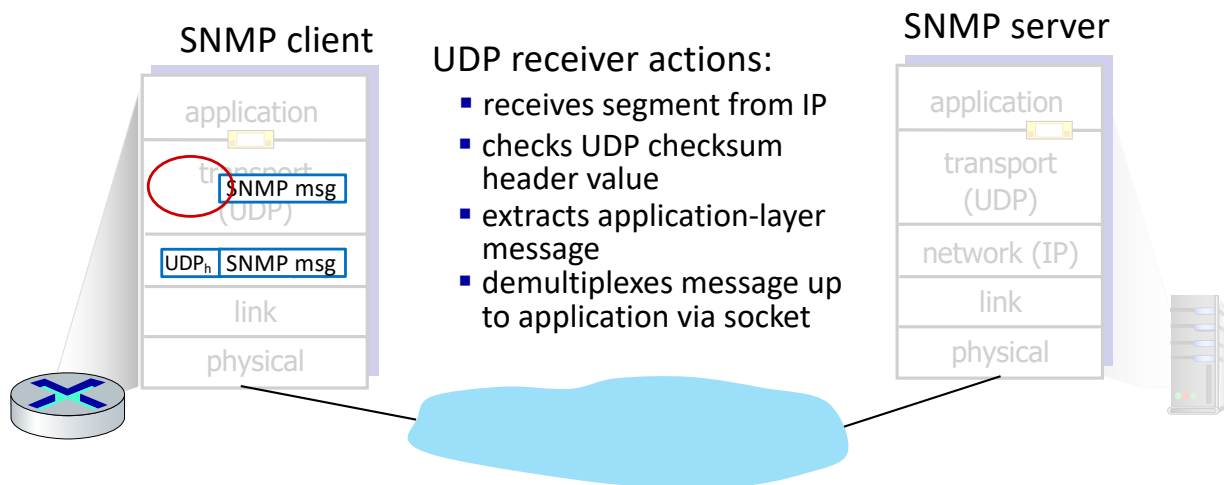


# UDP: Transport Layer Actions



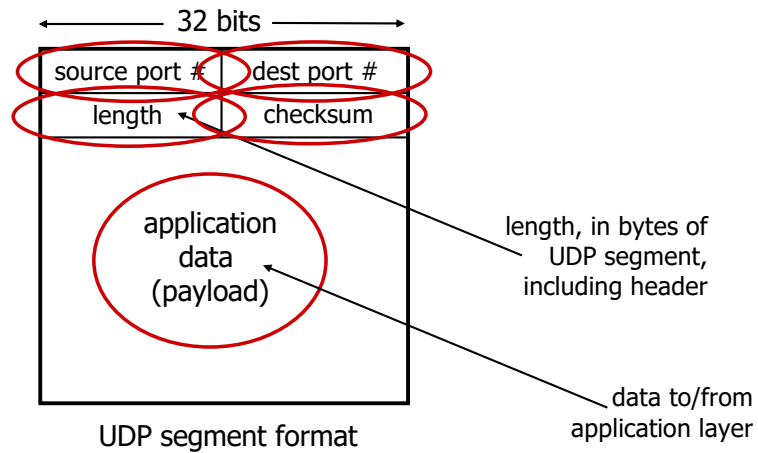
33

# UDP: Transport Layer Actions



34

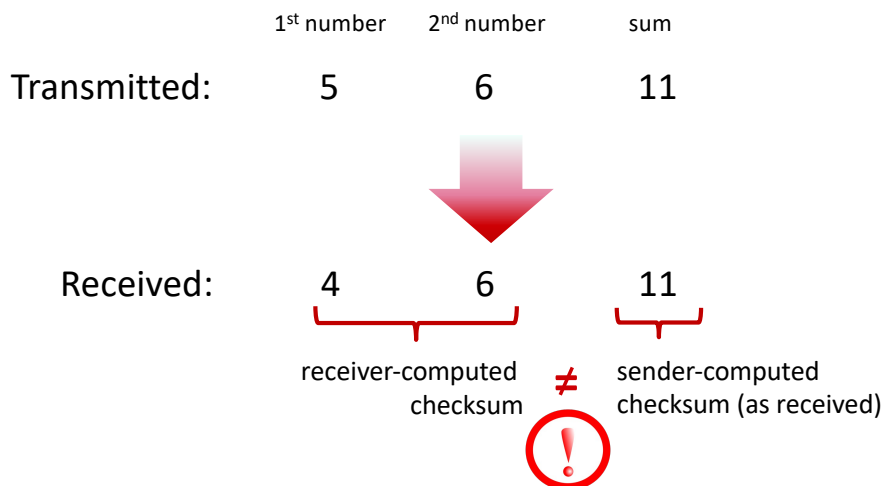
# UDP segment header



35

## UDP checksum

**Goal:** detect errors (*i.e.*, flipped bits) in transmitted segment



36

# Internet checksum

**Goal:** detect errors (*i.e.*, flipped bits) in transmitted segment

## sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- **checksum:** addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

## receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. *But maybe errors nonetheless? More later ....*

37

# Internet checksum: an 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

38

# Internet checksum: weak protection!

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
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

Even though numbers have changed (bit flips), *no* change in checksum!

39

## UDP Checksum in Practice

- 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** (some fields from the IP header)
- But the header contains the checksum itself?

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

40

## 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

**Note: TCP Checksum computation is exactly similar**

41

## 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 managed by periodic messages

42

# Summary: UDP

- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

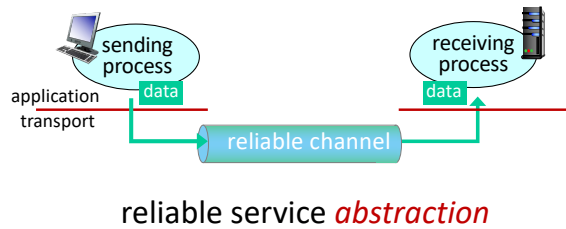
43

## Transport layer: roadmap

- ❖ Transport-layer services
- ❖ Multiplexing and demultiplexing
- ❖ Connectionless transport: UDP
- ❖ Principles of reliable data transfer
- ❖ Connection-oriented transport: TCP
- ❖ Principles of congestion control
- ❖ TCP congestion control
- ❖ Evolution of transport-layer functionality

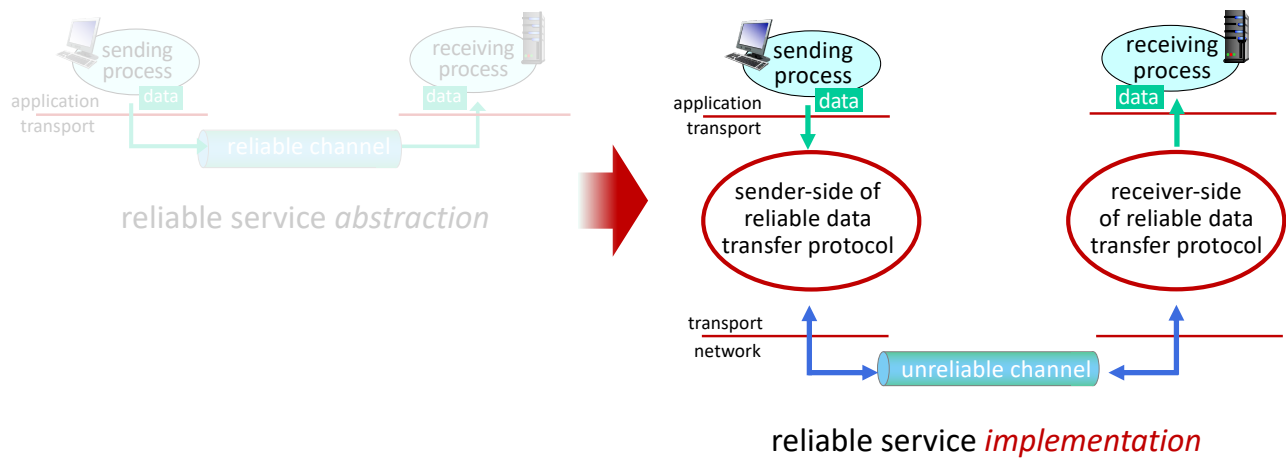
44

# Principles of reliable data transfer



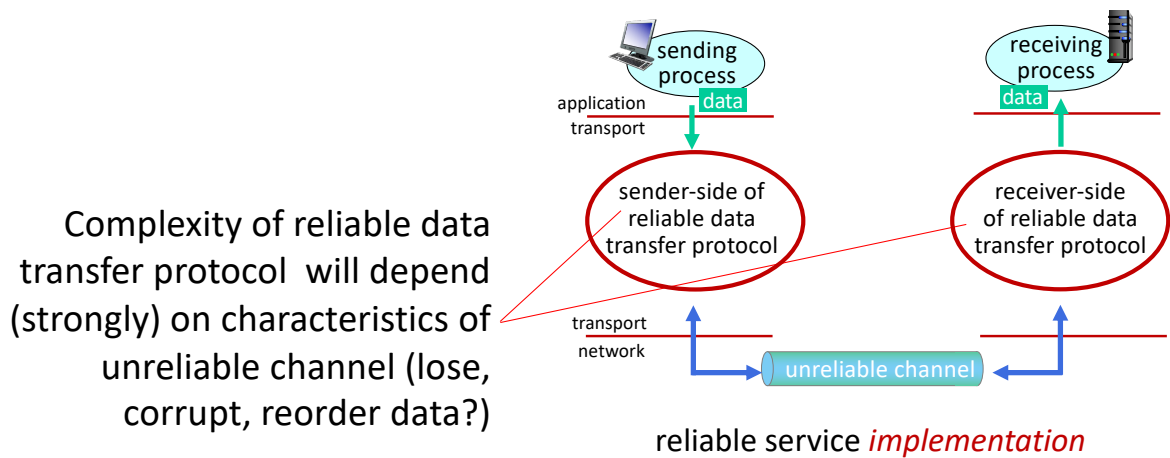
45

# Principles of reliable data transfer



46

# Principles of reliable data transfer

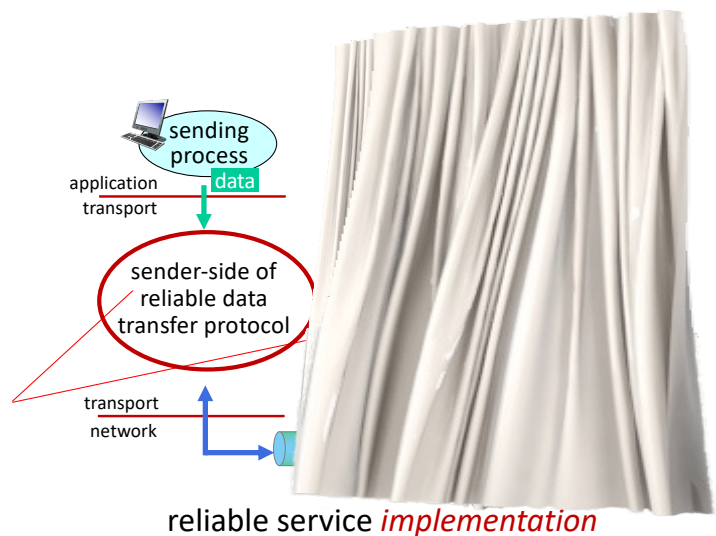


47

# Principles of reliable data transfer

Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

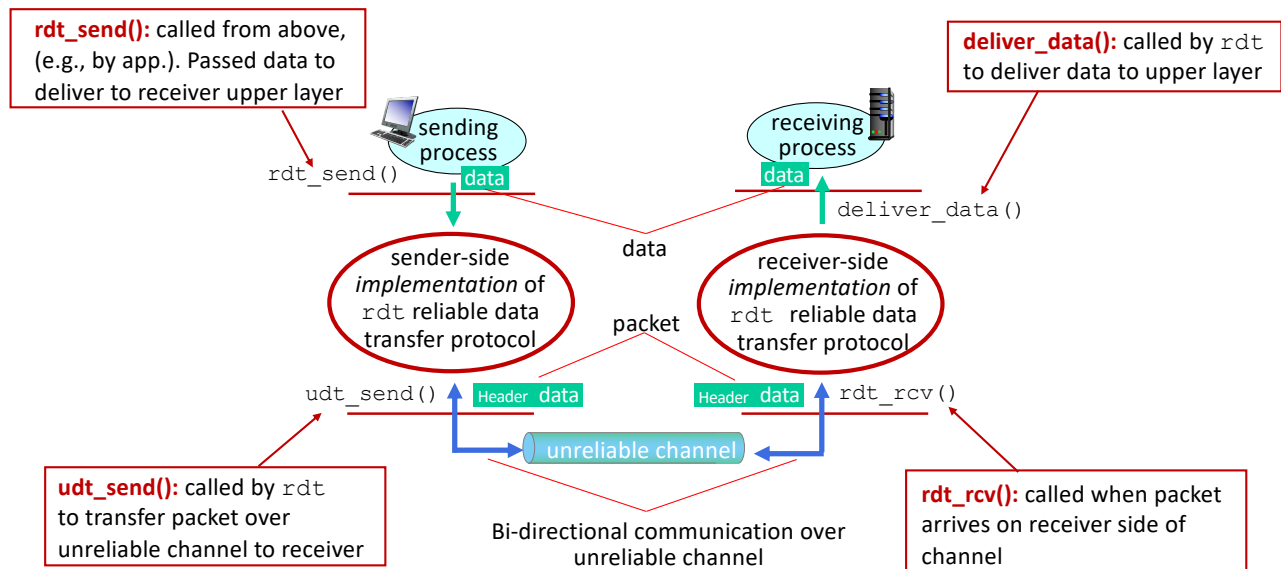
- unless communicated via a message



48



# Reliable data transfer protocol (rdt): interfaces

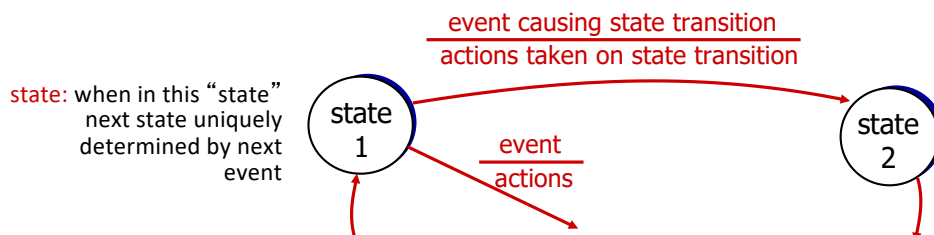


49

## Reliable data transfer: getting started

**We will:**

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- Book uses finite state machines (FSM) to specify sender, receiver
  - We won't use them in the lecture, and you won't be asked exam questions on them



50

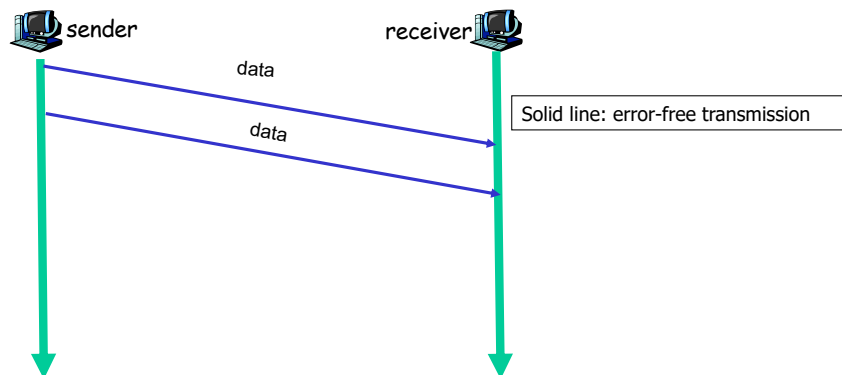
# rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Nothing to do



51

## Global Picture of rdt1.0



52

## rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the question*: how to recover from errors?

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

53

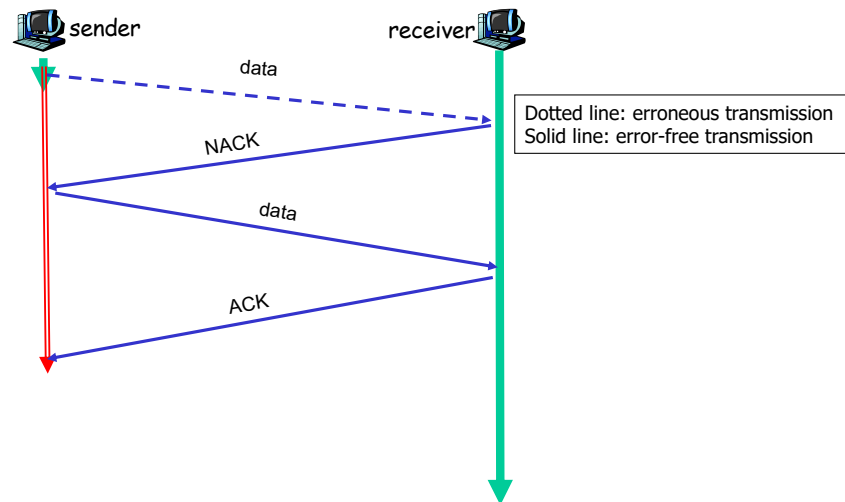
## rdt2.0: channel with bit errors

- 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

**stop and wait**  
sender sends one packet, then waits for receiver response

54

## Global Picture of rdt2.0



55

## 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

56

## rdt2.l: 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

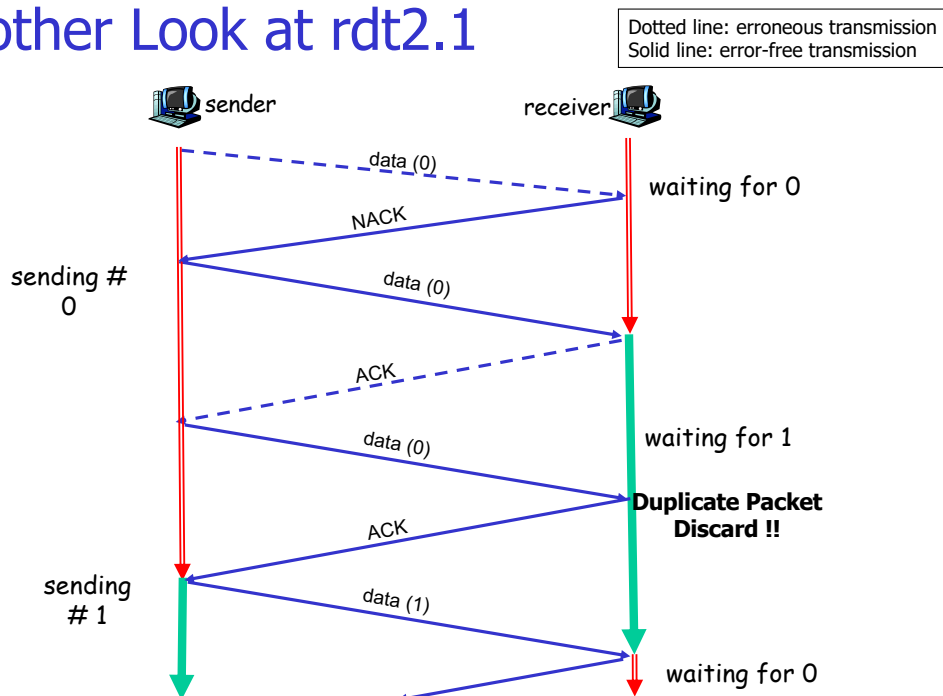
receiver:

- 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/NACK, Duplicate detection

57

## Another Look at rdt2.1



58

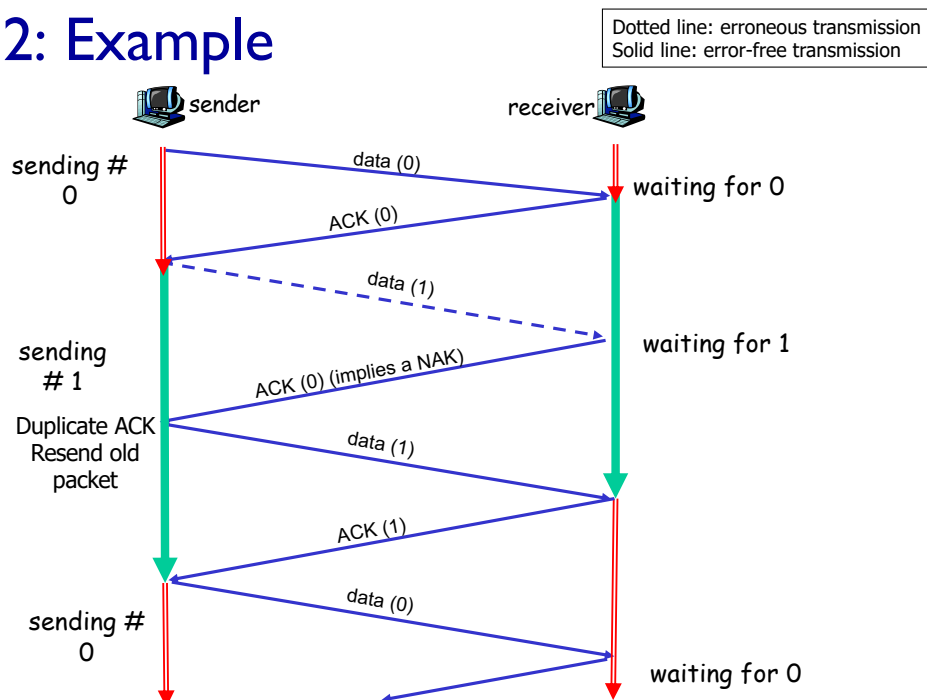
## rdt2.2: a NAK-free protocol

- 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*

As we will see, TCP uses this approach to be NAK-free

59

### rdt2.2: Example



60

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

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

- checksum, sequence #s, ACKs, retransmissions will be of help ... but not quite enough

*Q:* How do *humans* handle lost sender-to-receiver words in conversation?

61

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

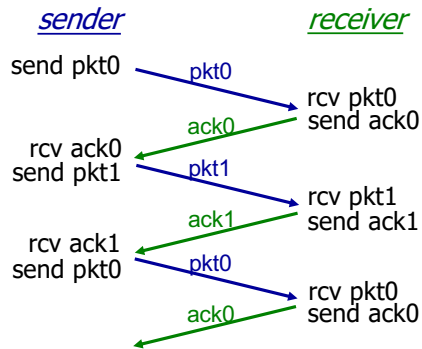
*Approach:* sender waits “reasonable” amount of time for ACK

- 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 packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time
- No retransmission on duplicate ACKs

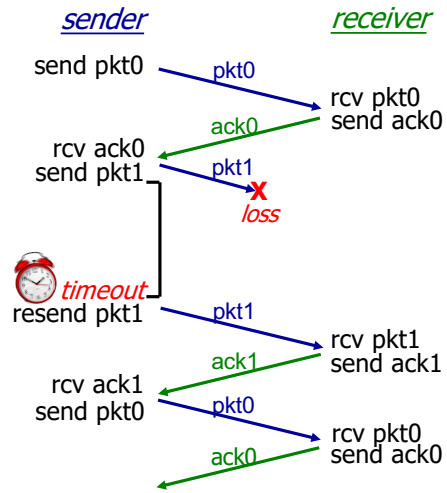


62

## rdt3.0 in action



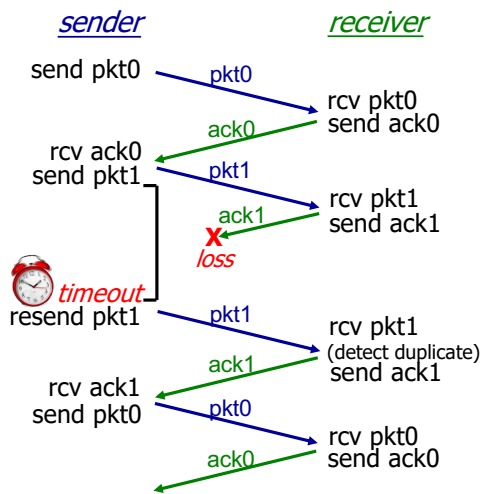
(a) no loss



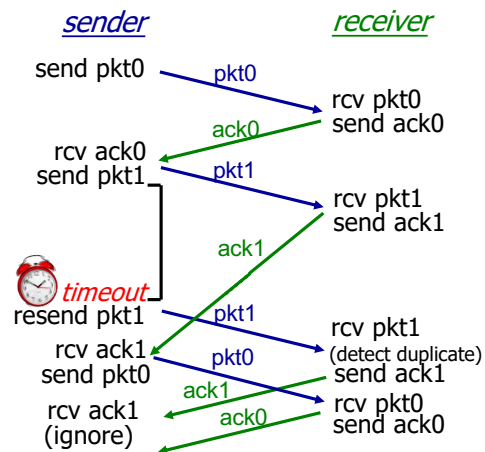
(b) packet loss

63

## rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

64



## 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

65

## 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

66

## 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

67

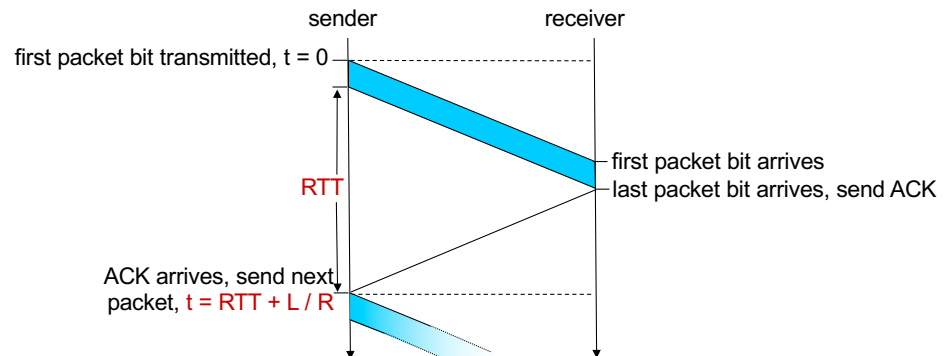
## Performance of rdt3.0 (stop-and-wait)

- $U_{\text{sender}}$ : **utilization** – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

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

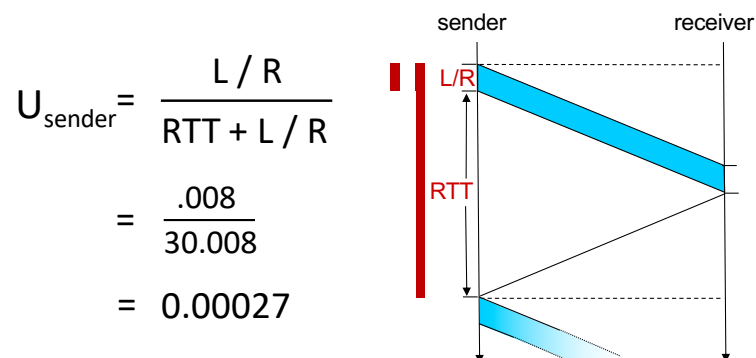
68

## rdt3.0: stop-and-wait operation



69

## rdt3.0: stop-and-wait operation



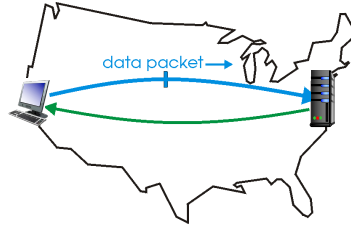
- rdt 3.0 protocol performance is very poor!
- Protocol limits performance of underlying infrastructure (channel)

70

# rdt3.0: pipelined protocols operation

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

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

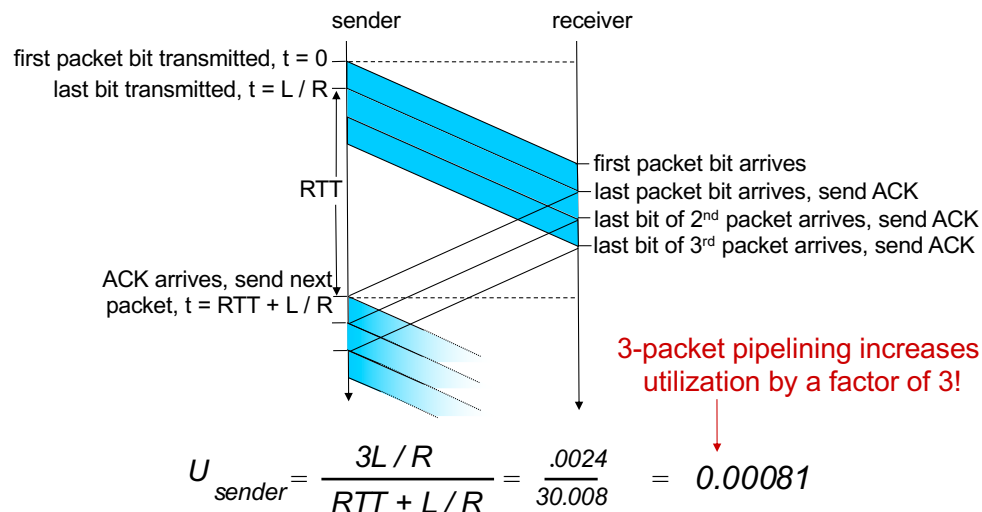


(a) a stop-and-wait protocol in operation

- Go Back N, Selective Repeat

71

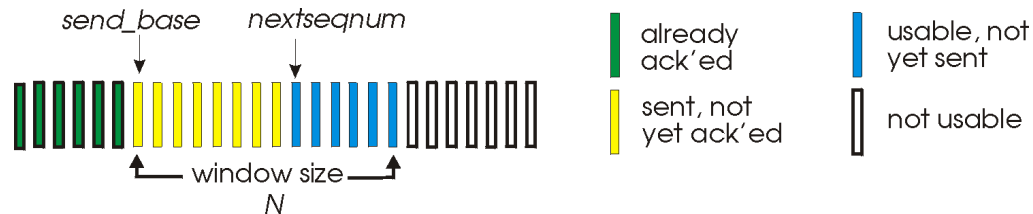
## Pipelining: increased utilization



72

## Go-Back-N: sender

- sender: “window” of up to  $N$ , consecutive transmitted but unACKed pkts
  - $k$ -bit seq # in pkt header



- **cumulative ACK:**  $ACK(n)$ : ACKs all packets up to, including seq #  $n$ 
  - on receiving  $ACK(n)$ : move window forward to begin at  $n+1$
- timer for oldest in-flight packet
- $timeout(n)$ : retransmit packet  $n$  and all higher seq # packets 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/)

73

## Go-Back-N: receiver

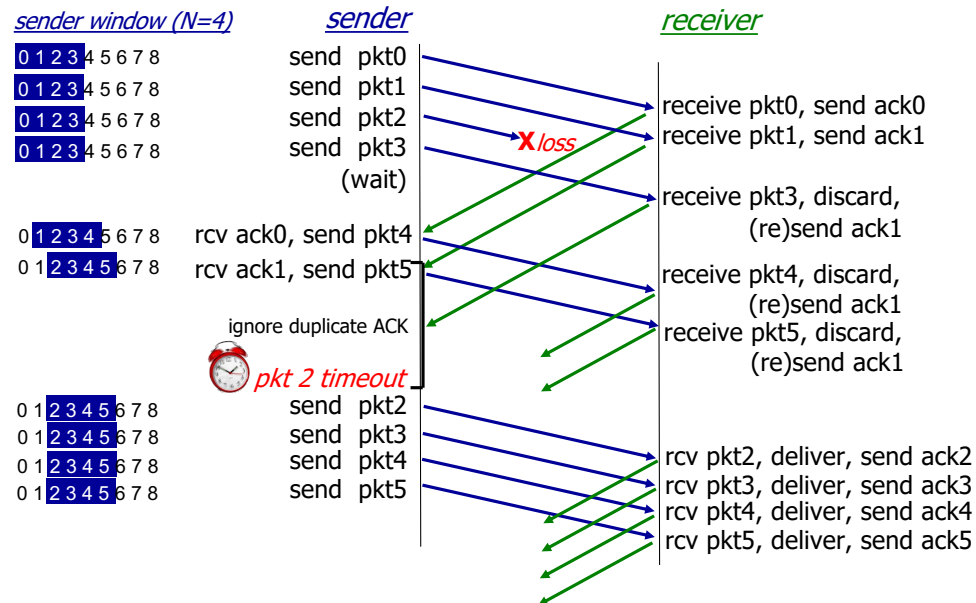
- ACK-only: always send ACK for correctly-received packet so far, with highest **in-order** seq #
  - may generate duplicate ACKs
  - need only remember  $rcv\_base$
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



74

# Go-Back-N in action



75

## Selective repeat

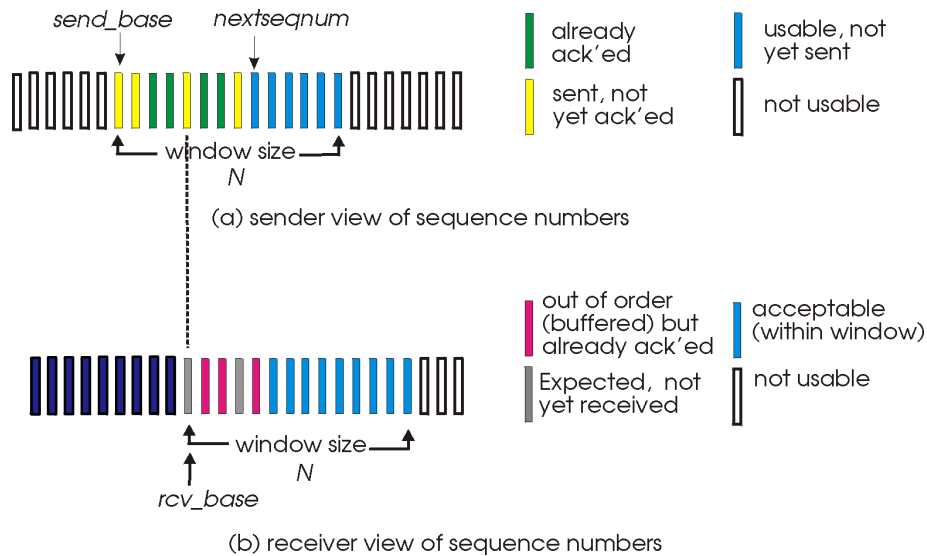
- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - N consecutive seq #s
  - limits seq #s of sent, unACKed packets

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)

[http://www.ccs-labs.org/teaching/rn/animations/gbn\\_sr/](http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/)

76

# Selective repeat: sender, receiver windows



77

## Selective repeat: sender and receiver

### sender

#### data from above:

- 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 $[rcvbase-N, rcvbase-1]$

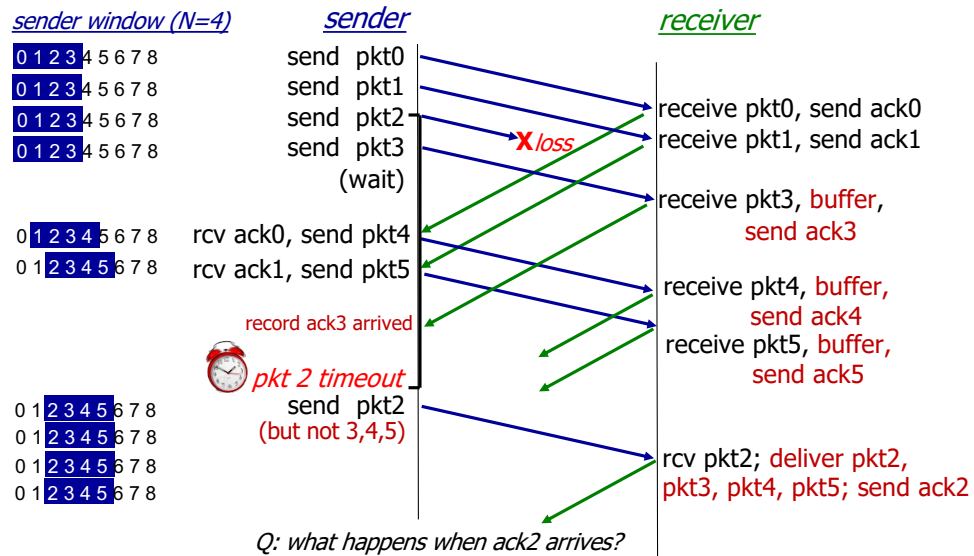
- ACK( $n$ )

#### otherwise:

- ignore

78

# Selective Repeat in action

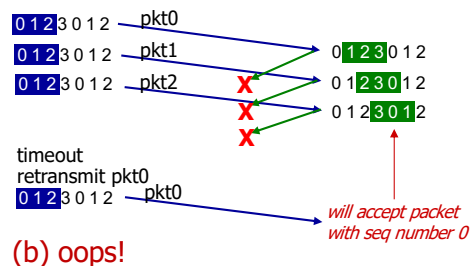
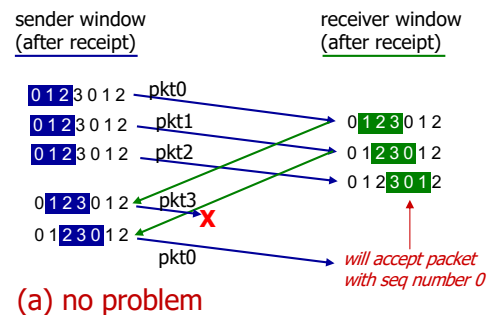


79

## Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



80



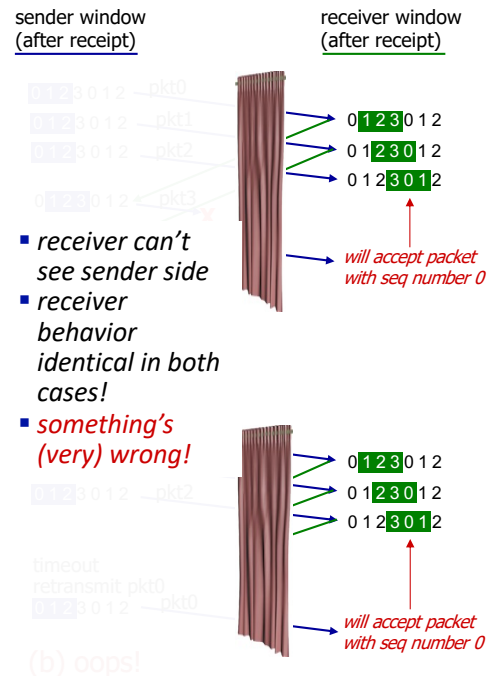
# Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

**Q:** what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

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



81

## 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

82

# 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?

**We will answer these questions when we cover TCP**

83

## 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

84

## 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]

85

## 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

86

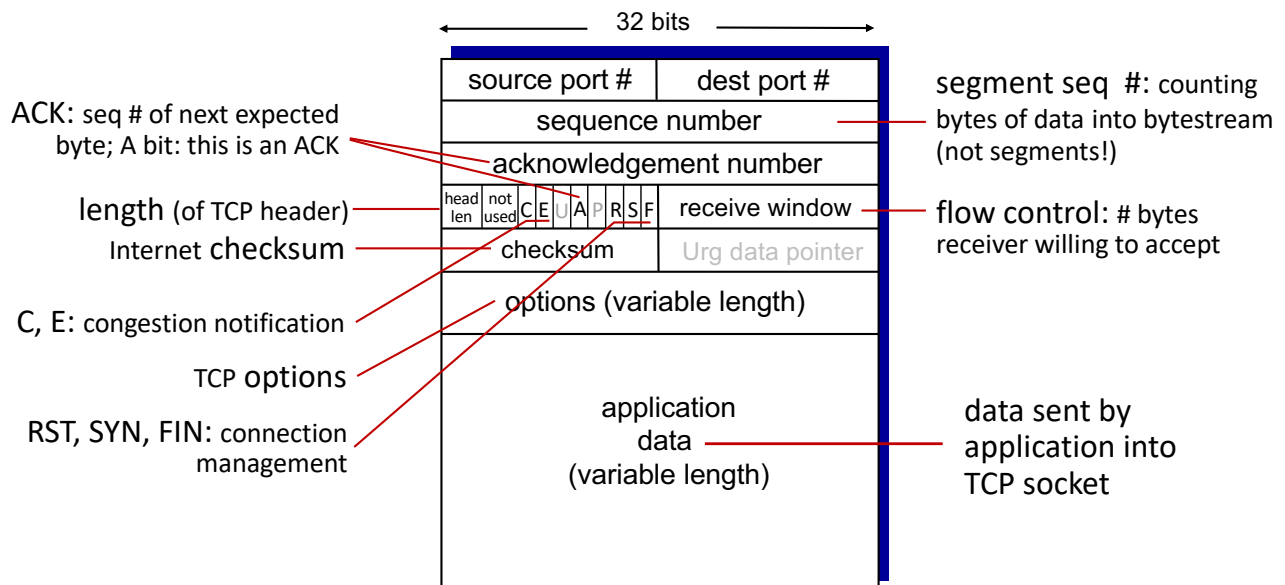
# TCP: overview RFCs: 793, 1122, 2018, 5681, 7323

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



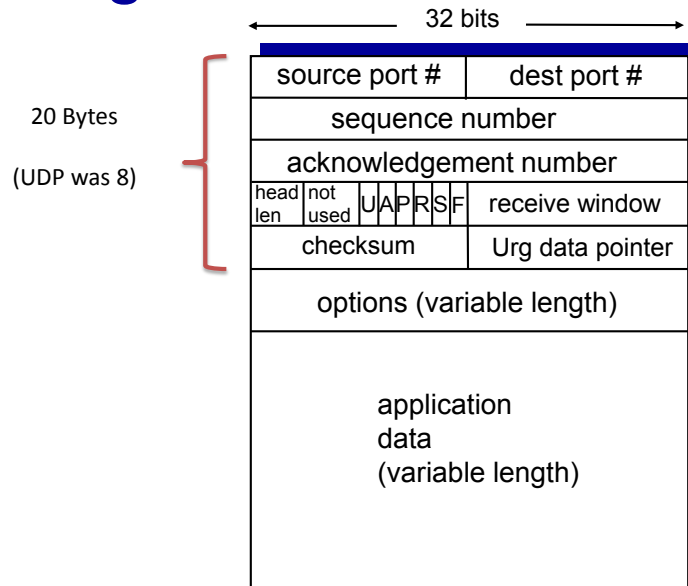
87

## TCP segment structure



88

# TCP segment structure



89

## Summary

- ❖ Multiplexing/Demultiplexing
- ❖ UDP
- ❖ Reliable Data Transfer
  - Stop-and-wait protocols
  - Sliding window protocols
- ❖ TCP - intro
- ❖ Up Next:
  - TCP – continued in more detail
  - Congestion Control

90