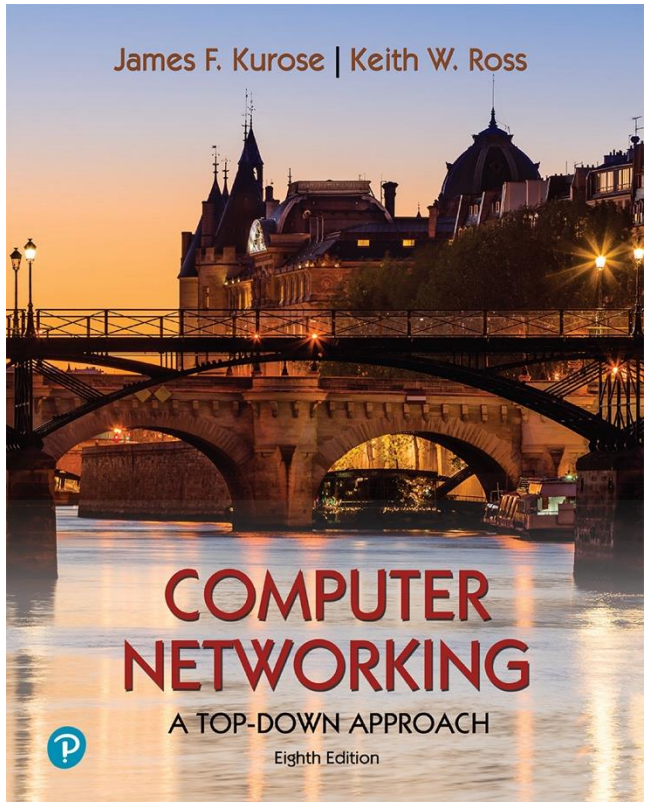
Chapter 3
Transport Layer

Yaxiong Xie

Department of Computer Science and Engineering
University at Buffalo, SUNY



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Adapted from the slides of the book's authors

Transport layer: overview

Our goal:

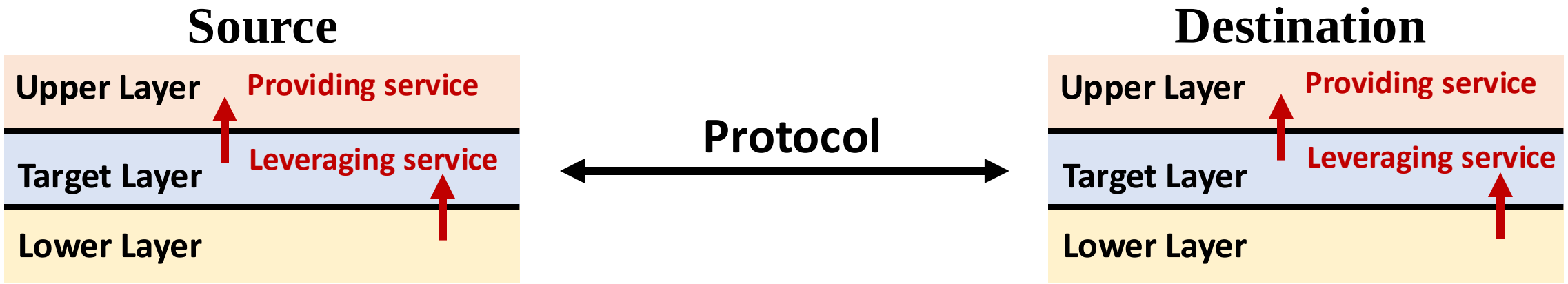
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion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

Transport layer: roadmap

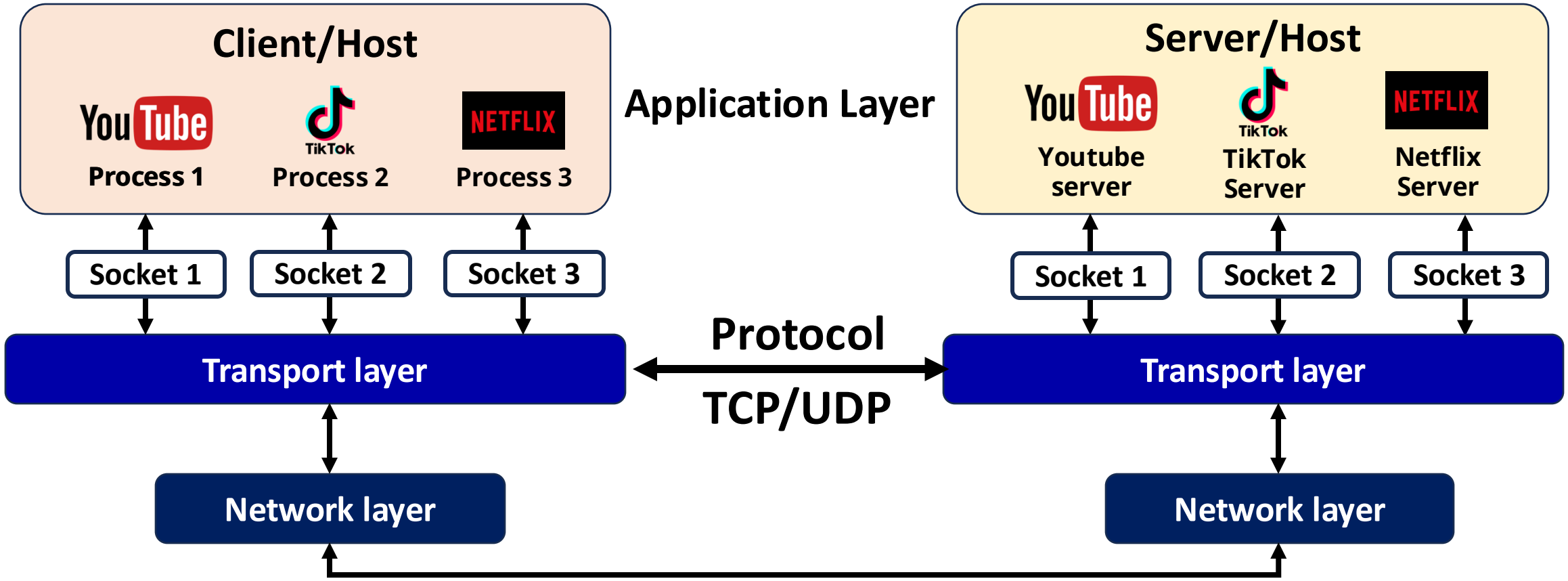
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Transport services and protocols

- Providing service for upper layers
- Leveraging service provided by lower layers
- Communicating using protocols



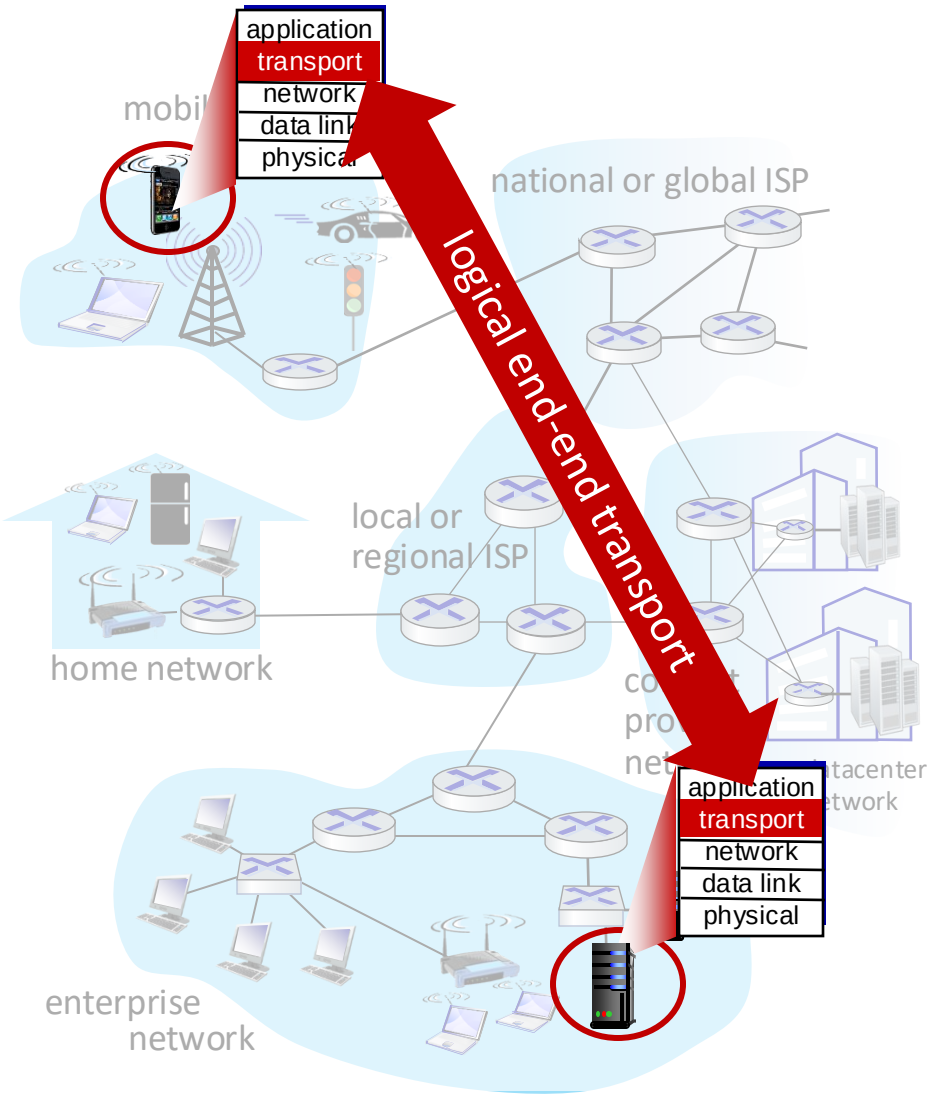
Transport vs. network layer services and protocols



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

Two principal Internet transport protocols

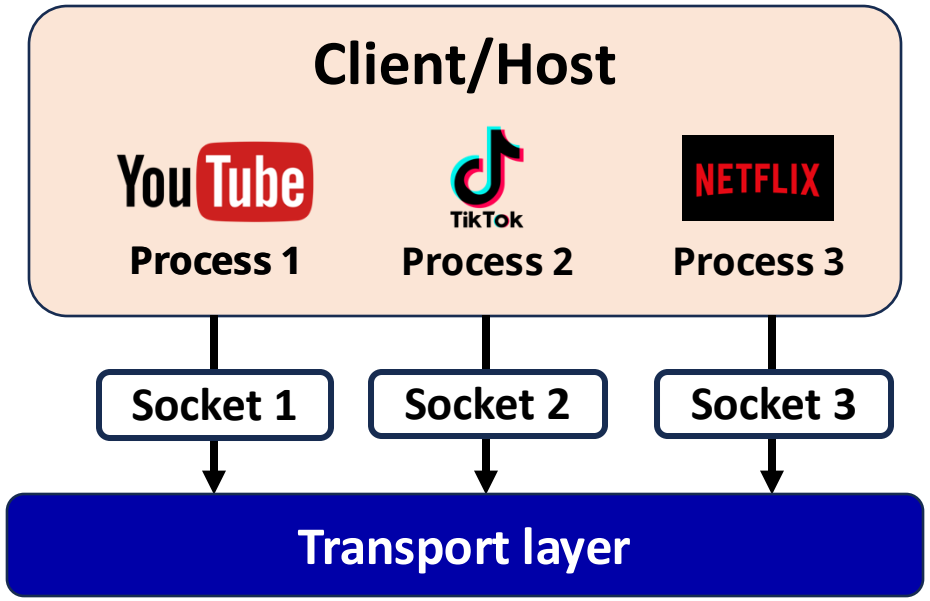
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services *not* available:
  - delay guarantees
  - bandwidth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

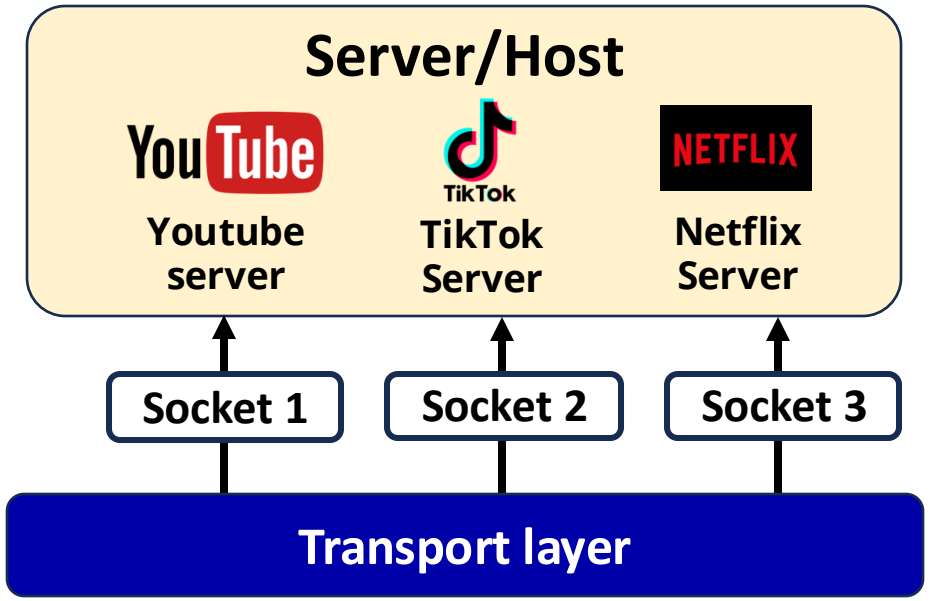


Chapter 3: roadmap

- 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

Multiplexing/demultiplexing





multiplexing as sender:

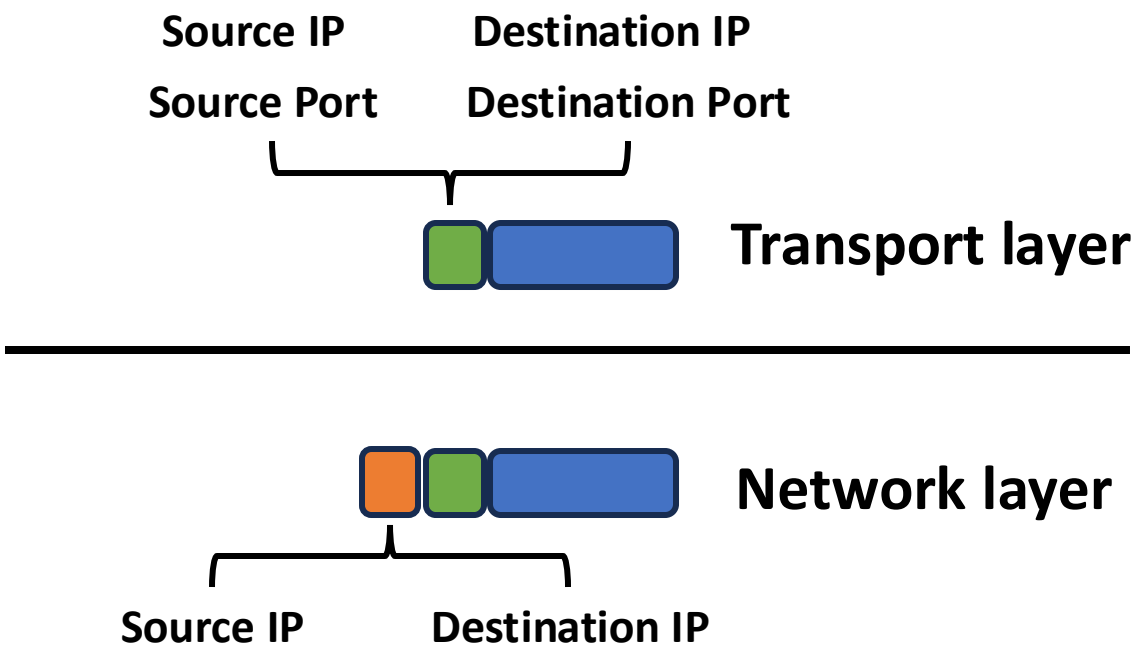
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing as receiver:

use header info to deliver received segments to correct socket

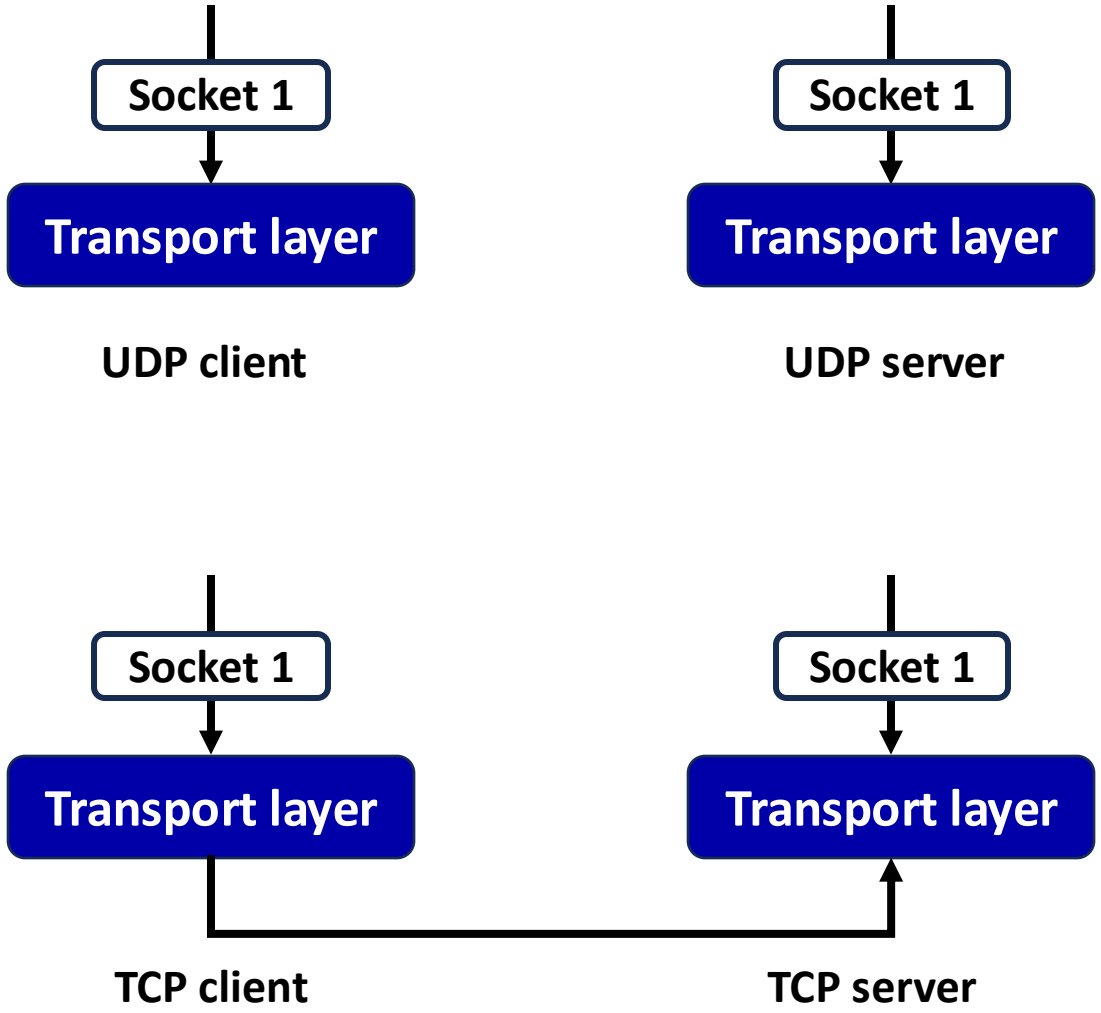
How demultiplexing works

- Each network layer datagram has source and destination IP address
- Each transport layer segment has source and destination port number
- Host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP VS UDP: connection VS connectionless

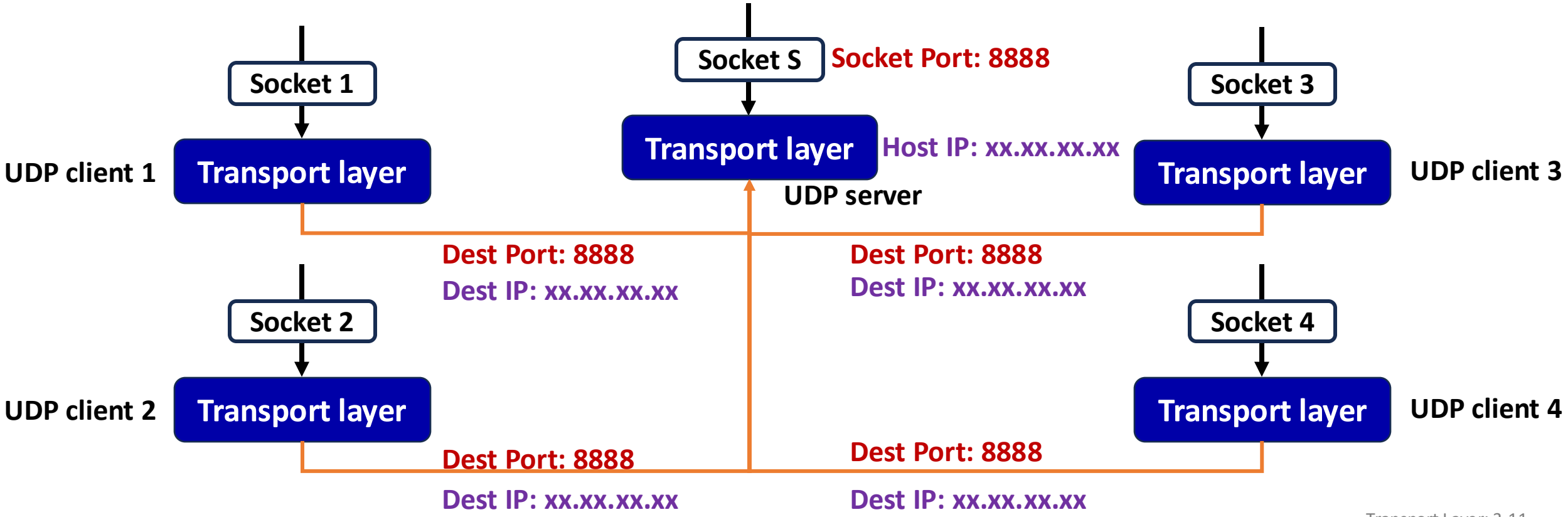
- UDP the socket are local and independent
- TCP two sockets are linked together



Connectionless demultiplexing

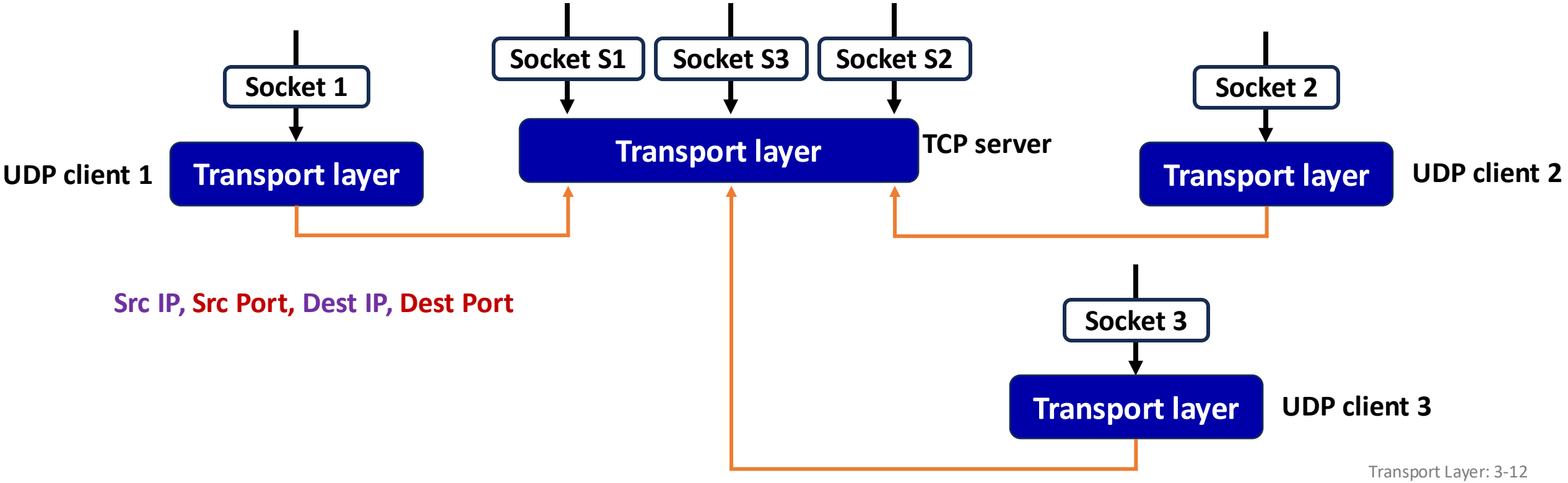
- UDP the socket are local and independent
- Demultiplexing based on destination port

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

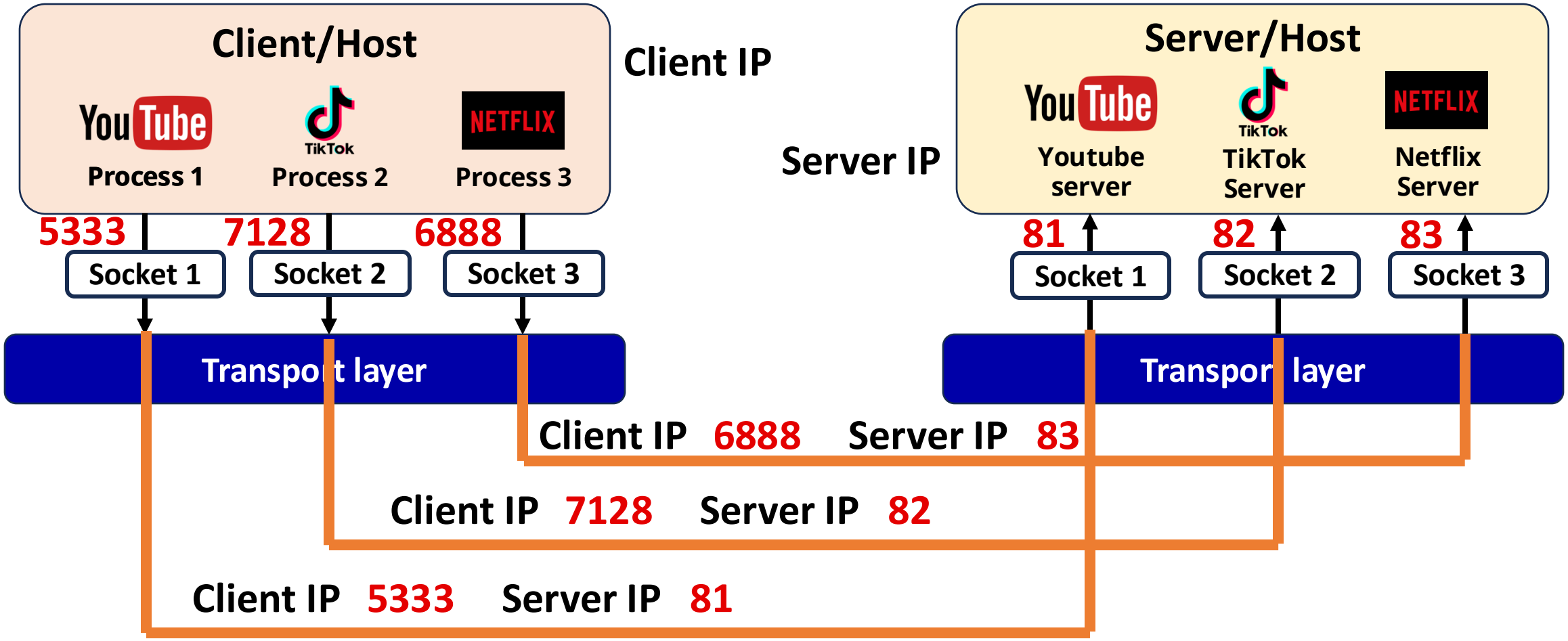


Connection-oriented demultiplexing

- TCP two sockets are linked together
- We need to ID the connection for demultiplexing



Connection-oriented demultiplexing



4-tuple matters here!

Connection-oriented demultiplexing

- 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

Summary

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

Chapter 3: 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

UDP: User Datagram Protocol

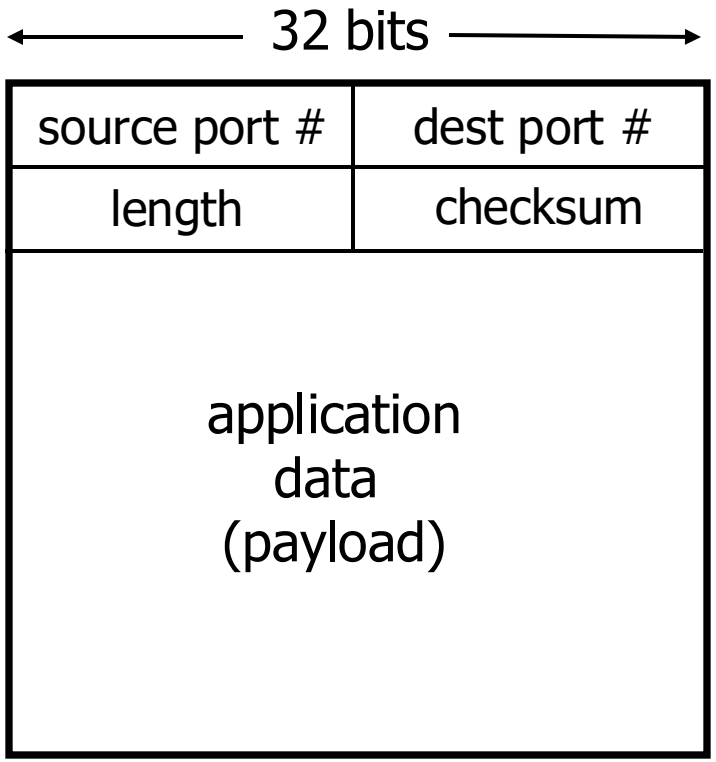
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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

UDP: User Datagram Protocol

- Lightweight communication between processes:
  - Send and receive messages
- Avoid overhead of ordered, reliable delivery:
  - No connection setup delay, no in-kernel connection state



UDP segment format

UDP: Advantage

- Fine-grain control:
  - UDP sends as soon as the application writes
- No connection set-up delay
  - UDP sends without establishing a connection
- No connection state in host OS
  - No buffers, parameters, sequence #s, etc
- Small header overhead
  - UDP header is only eight-bytes long

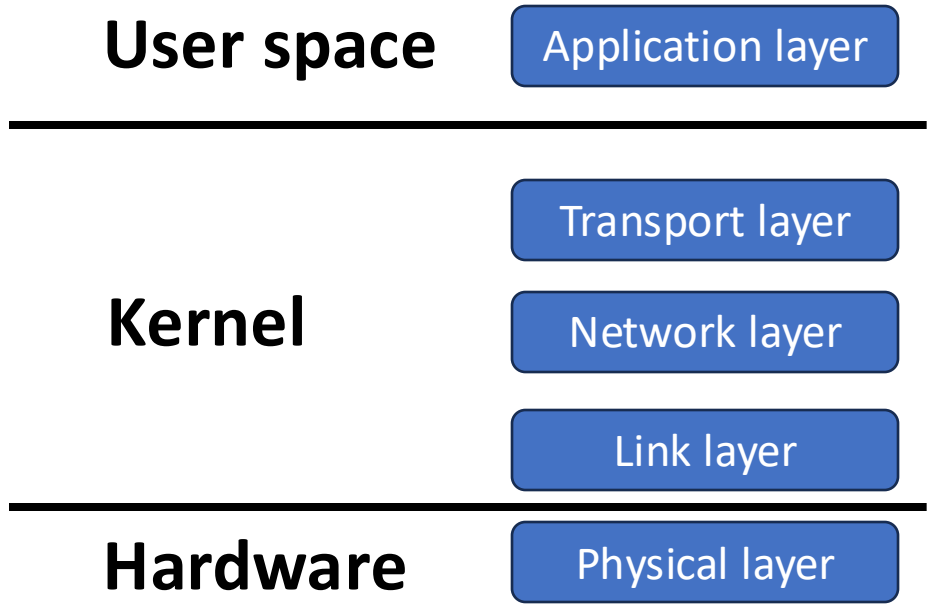
UDP: User Datagram Protocol

- UDP use:
  - 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

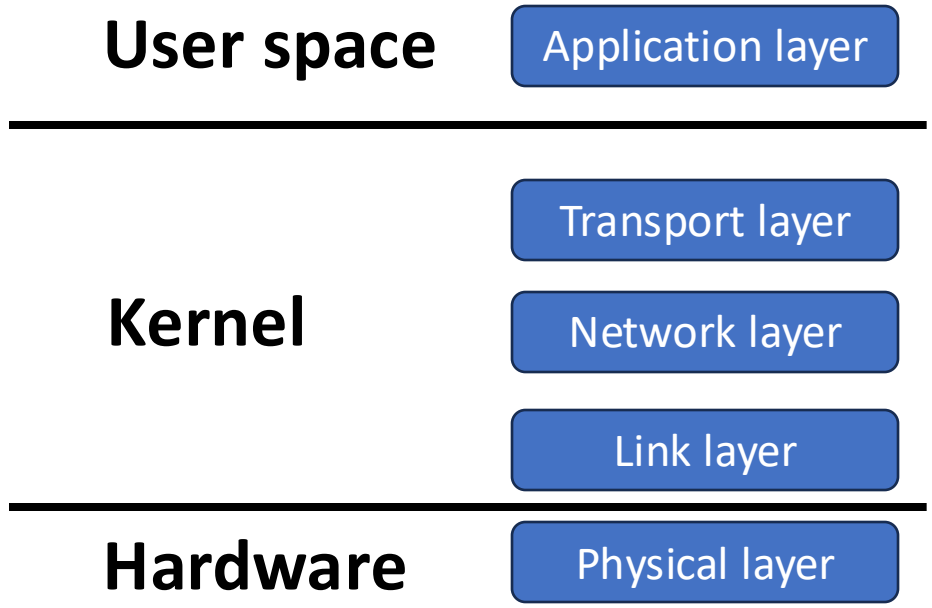
UDP and TCP are implemented inside Kernel

- A typical OS includes userspace and kernel space
- Kernel Space Program:
  - Has **full control** over the hardware and manages system resources like memory, CPU scheduling, and I/O operations.
  - Runs in **privileged mode**
- User Space Program:
  - has no direct access to hardware and must communicate with the kernel for resource management.



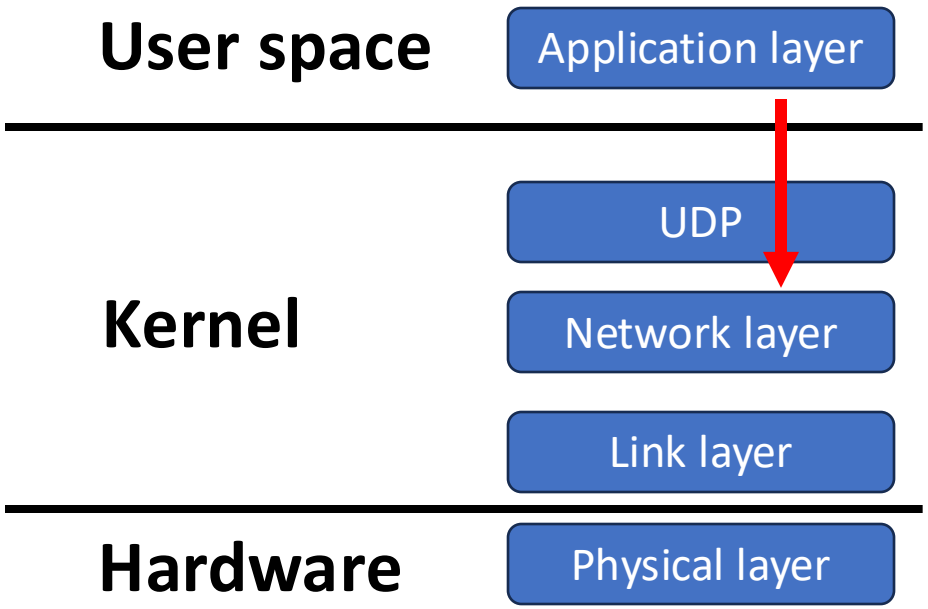
UDP and TCP are implemented inside Kernel

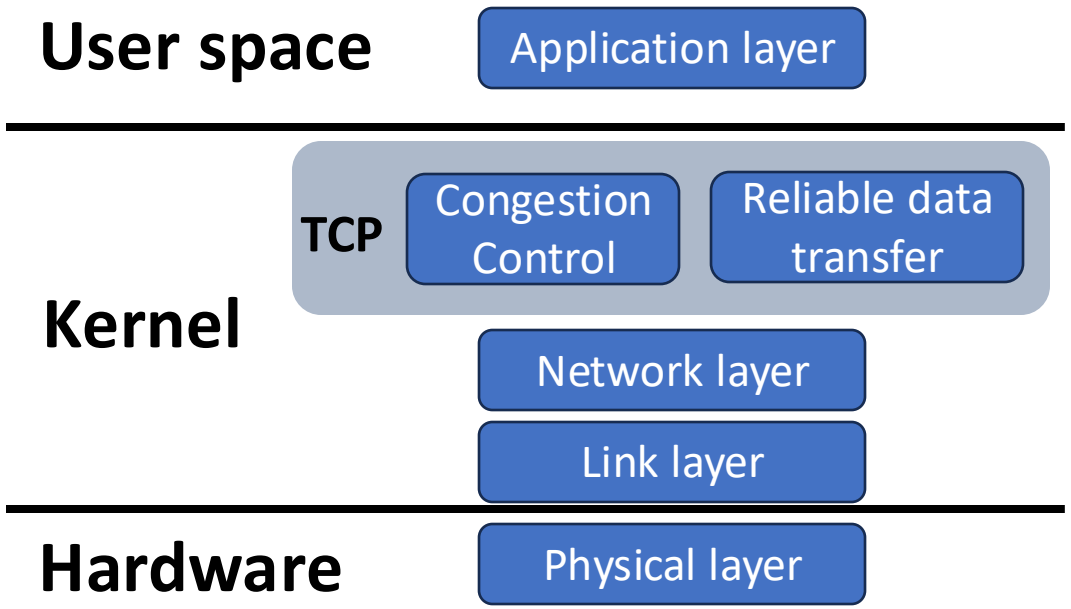
- Network Stack inside OS:
  - Physical layer is hardware
  - Application layer is implemented in userspace
  - Transport, network and link layer is implemented inside kernel



UDP and TCP are implemented inside Kernel

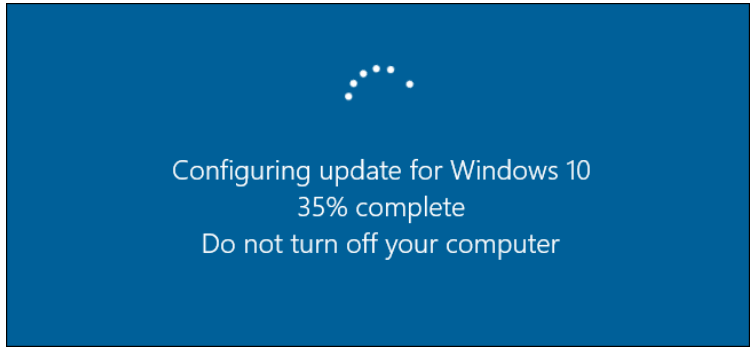
- TCP and UDP inside OS:

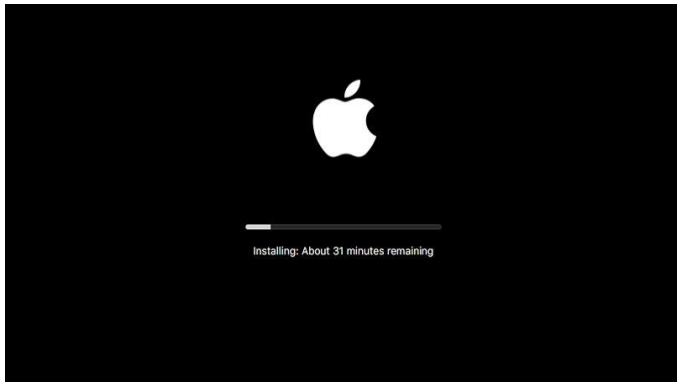


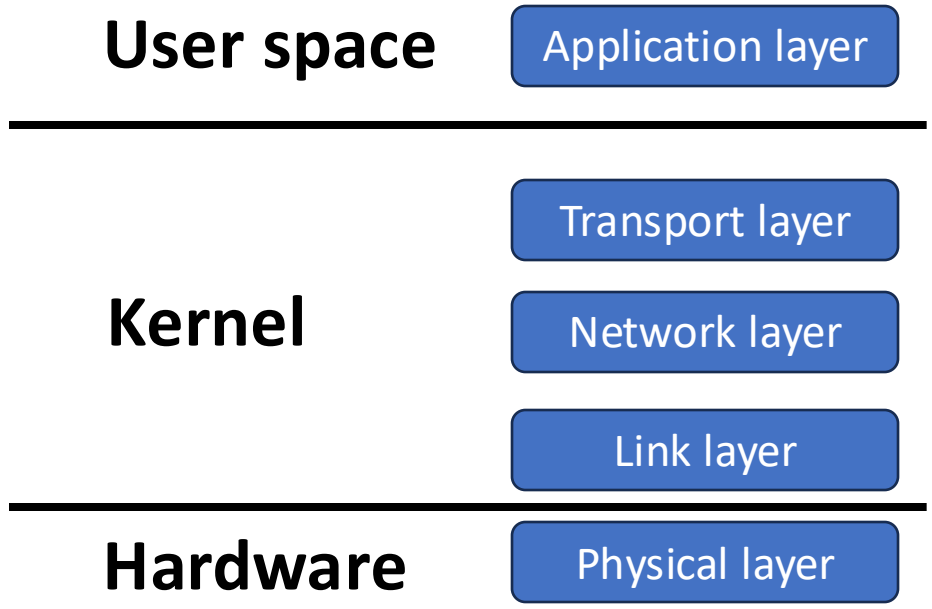


Updating kernel is hard!

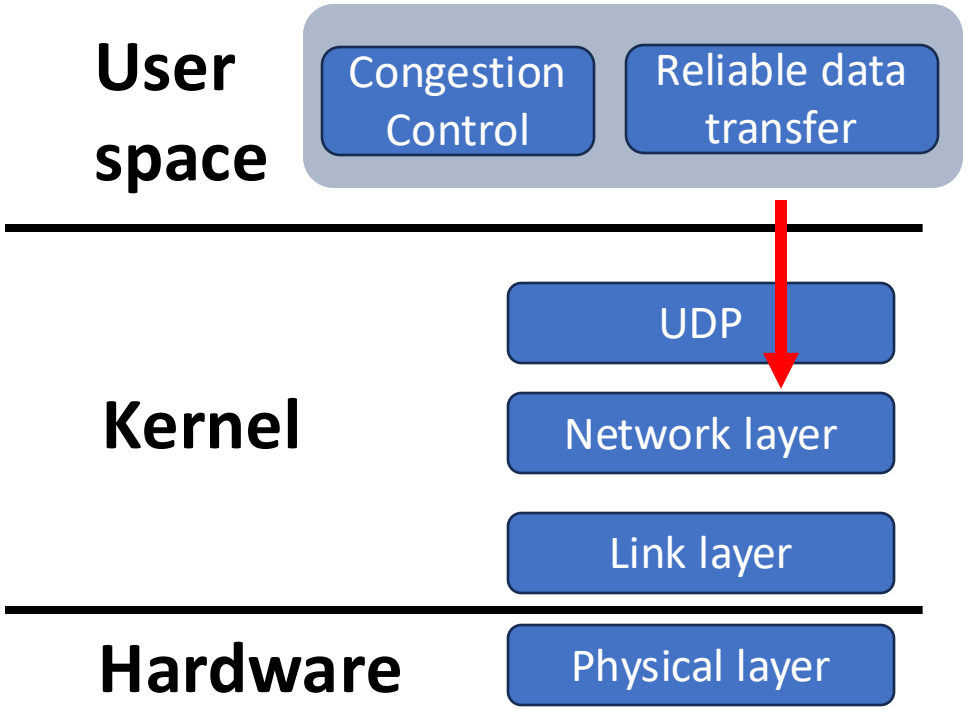
- We only have three main OS:
  - Linux, MAC OS and Windows
- Any updates must be approved by those three OS
  - Kernel affects billions of machines
- We need to push the update of system to billions of machines

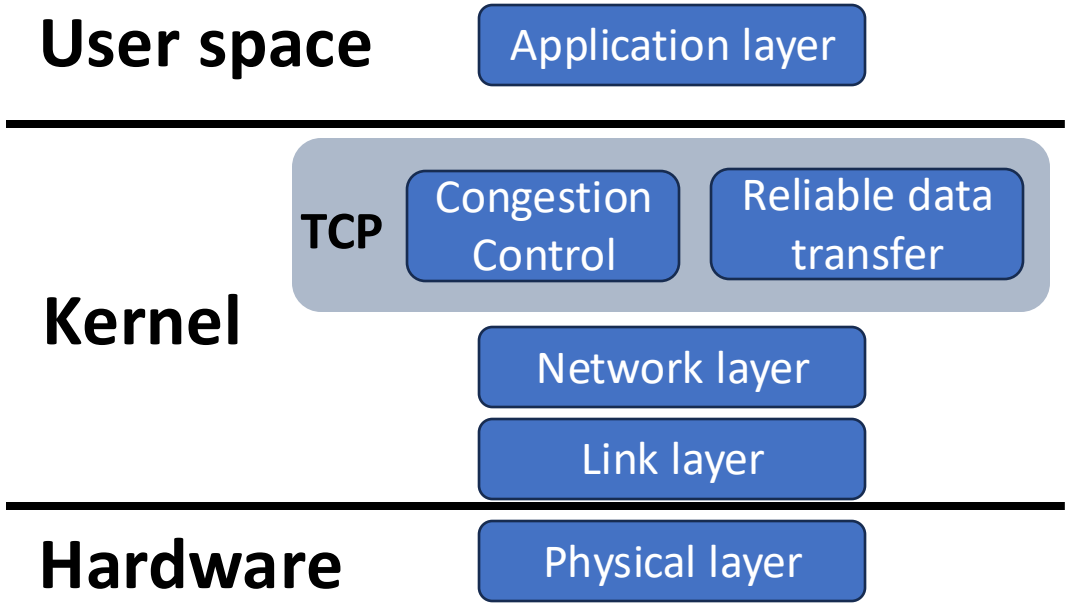




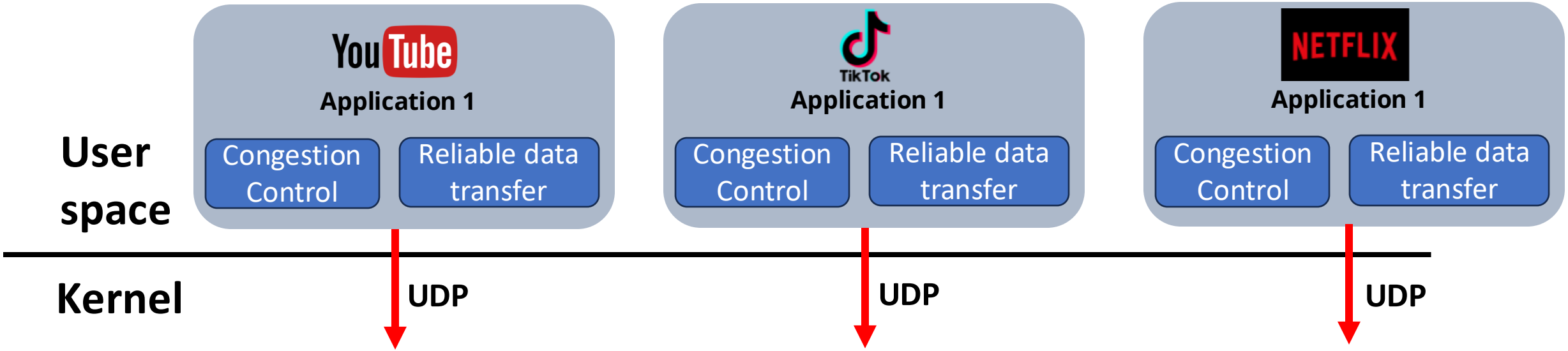


UDP provides the flexibility





UDP provides the flexibility



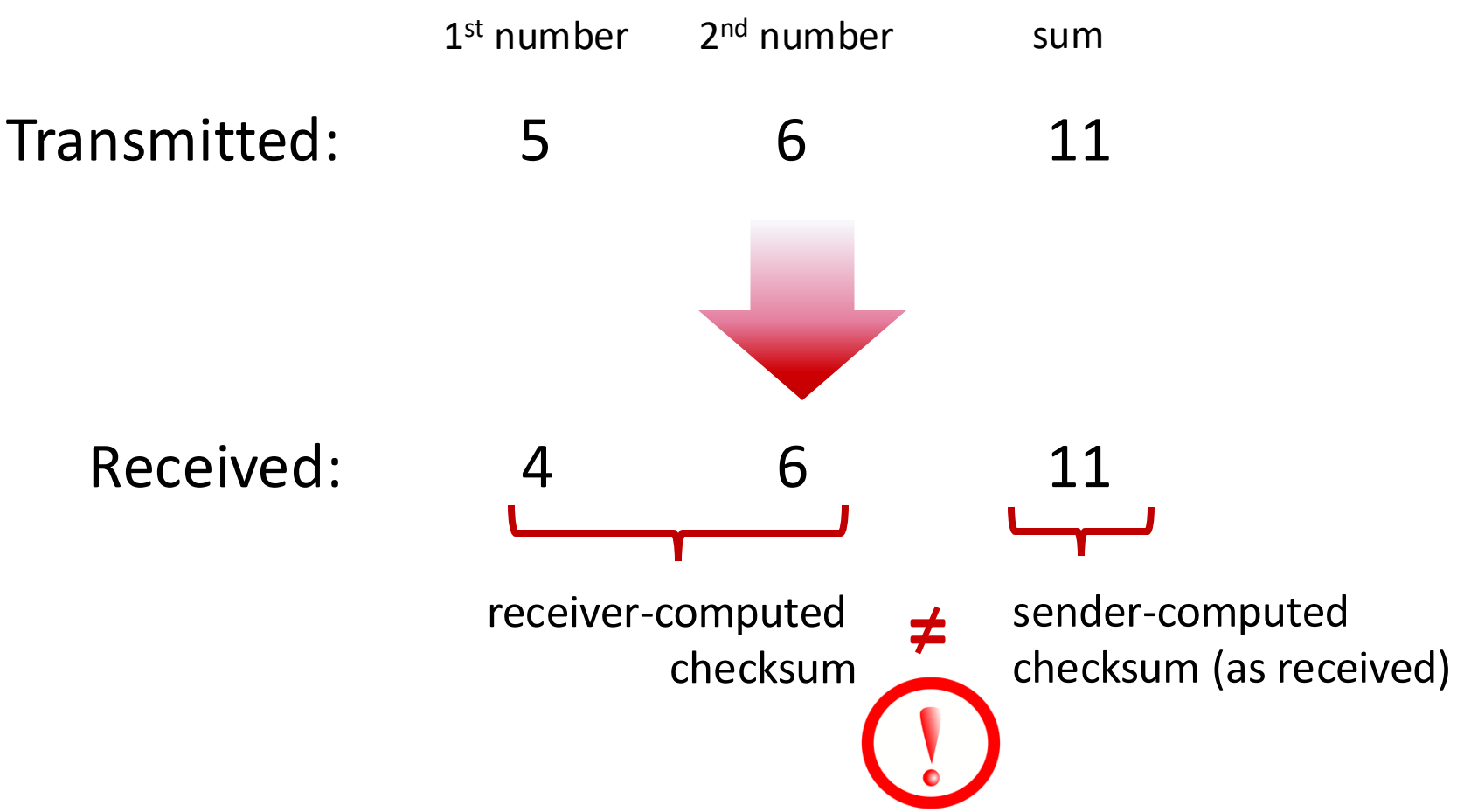
- Each application can implement its own algorithms without the need of approval from the OS
- Updating application is much easier than updating the kernel (OS)
  - Speeding up the development and implementation of new technology

Example: QUIC from Google

- QUIC (Quick UDP Internet Connections) is a transport-layer protocol developed by Google replace TCP by using UDP
  - providing faster connection establishment,
  - improved congestion control, and
  - better performance in mobile and high-latency environments
- Chrome Microsoft Edge, Firefox, and Safari all support it
- In Chrome, QUIC is used by more than half of all connections to Google's servers

UDP checksum

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



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 ….

Internet checksum: an example

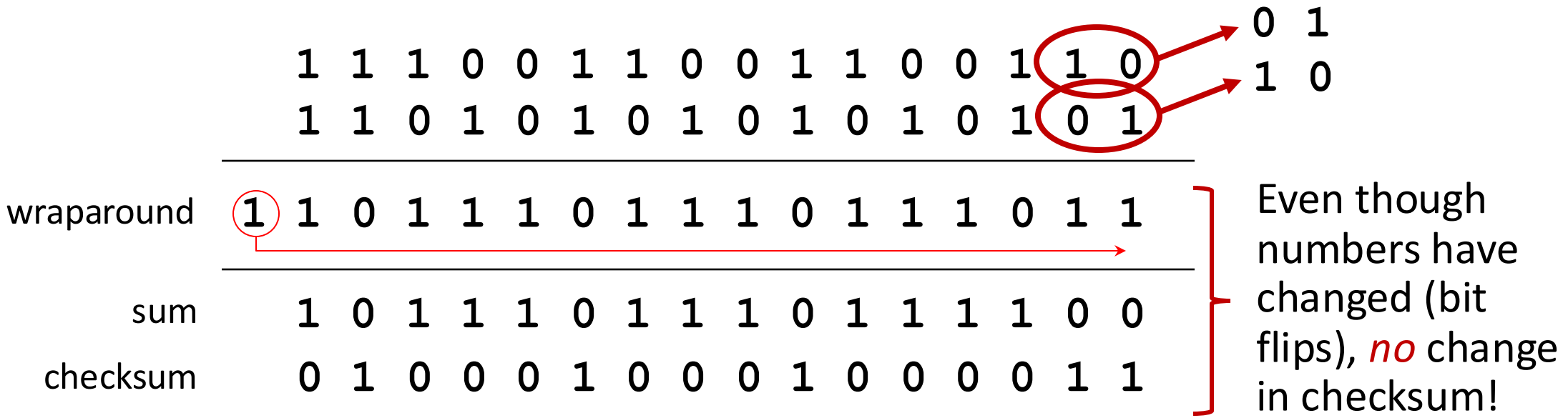
example: add two 16-bit integers

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

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

Internet checksum: weak protection!

example: add two 16-bit integers



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)