Network layer: “control plane” roadmap

- introduction
- routing protocols
- intra-ISP routing: OSPF
- routing among ISPs: BGP
- SDN control plane
- Internet Control Message Protocol

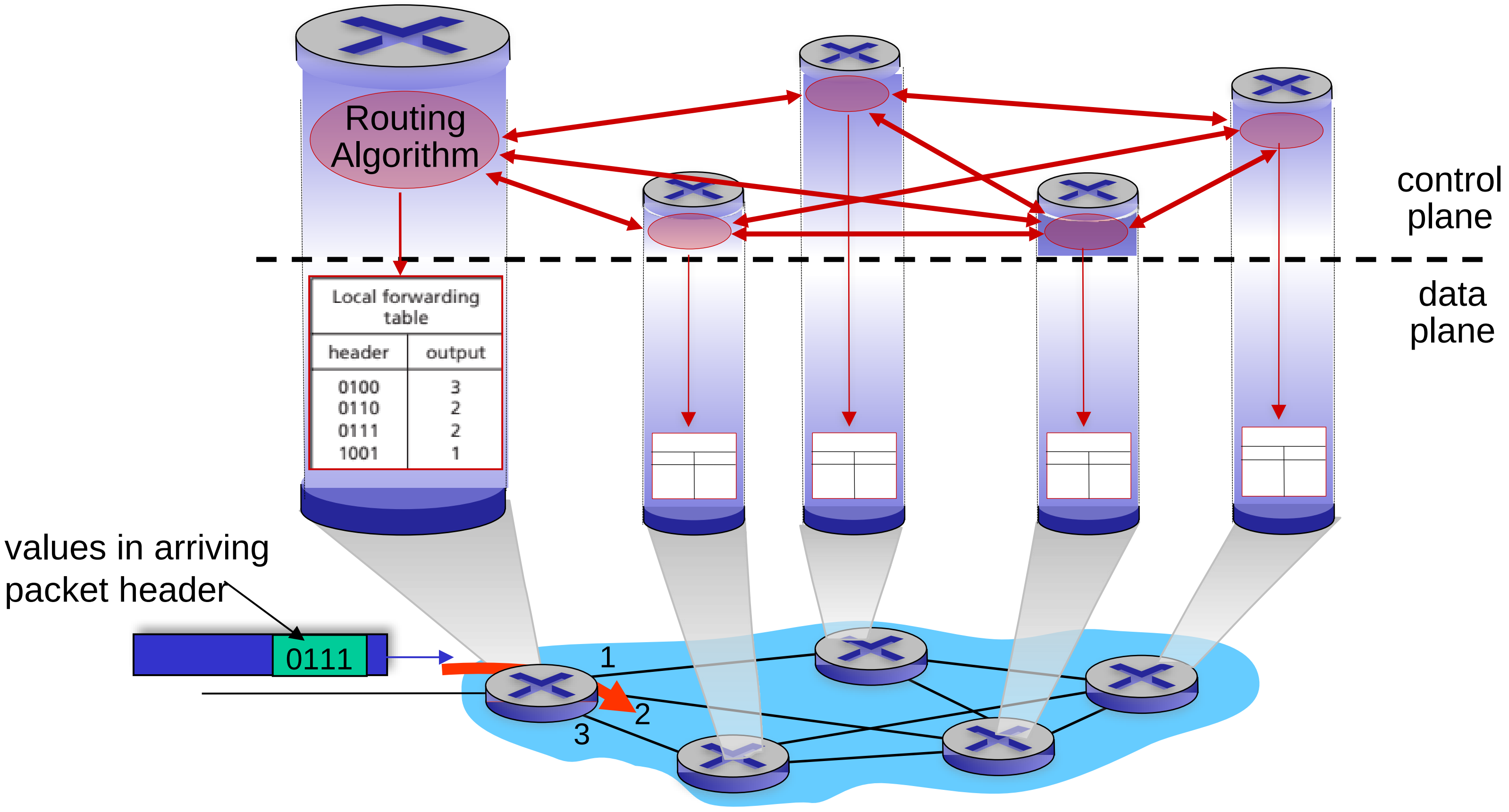
- network management, configuration
  - SNMP
  - NETCONF/YANG

Software defined networking (SDN)

- Internet network layer: historically implemented via distributed, per-router control approach:
  - *monolithic* router contains switching hardware, runs proprietary implementation of Internet standard protocols (IP, RIP, IS-IS, OSPF, BGP) in proprietary router OS (e.g., Cisco IOS)
  - different "middleboxes" for different network layer functions: firewalls, load balancers, NAT boxes, ..
- ~2005: renewed interest in rethinking network control plane

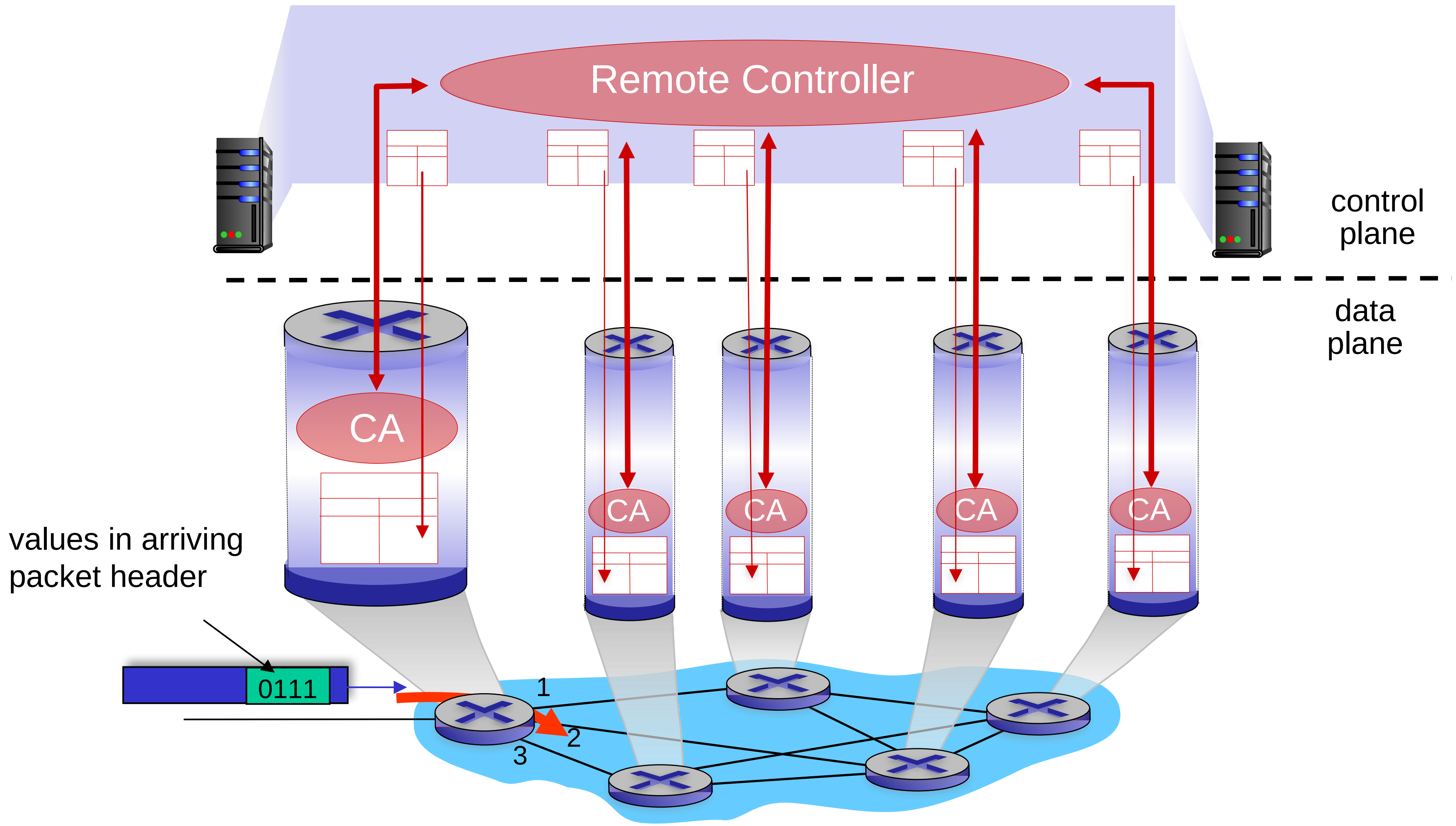
Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane to computer forwarding tables



Software-Defined Networking (SDN) control plane

Remote controller computes, installs forwarding tables in routers



Software defined networking (SDN)

Why a *logically centralized* control plane?

- easier network management: avoid router misconfigurations, greater flexibility of traffic flows
- table-based forwarding (recall OpenFlow API) allows "programming" routers
  - centralized "programming" easier: compute tables centrally and distribute
  - distributed "programming" more difficult: compute tables as result of distributed algorithm (protocol) implemented in each-and-every router
- open (non-proprietary) implementation of control plane
  - foster innovation: let 1000 flowers bloom

SDN analogy: mainframe to PC revolution

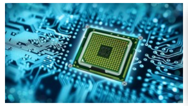
Vertically integrated
Closed, proprietary
Slow innovation
Small industry

App

Open Interface

Windows or Linux or MAC OS

Open Interface

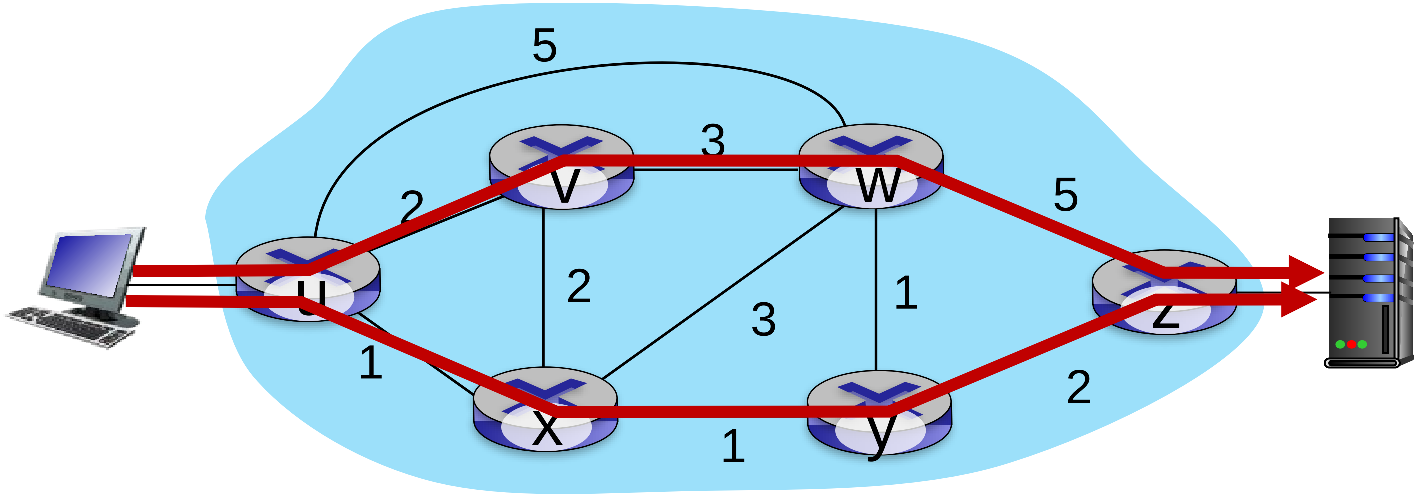


Microprocessor

Horizontal
Open interfaces
Rapid innovation
Huge industry

* Slide courtesy: N. McKeown

Traffic engineering: difficult with traditional routing

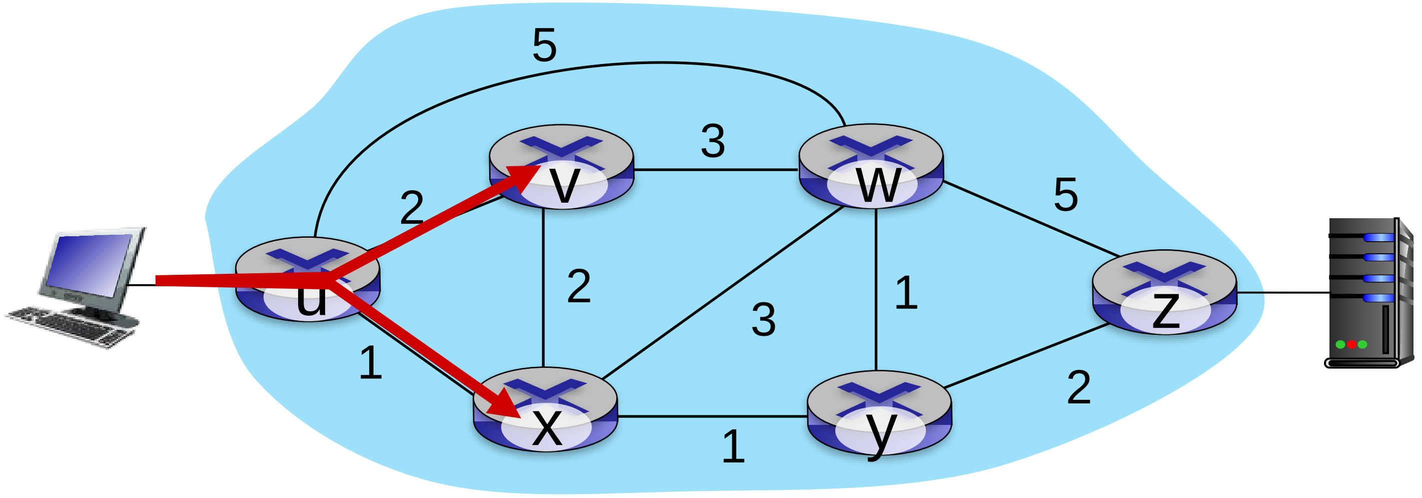


Q: what if network operator wants u-to-z traffic to flow along *uvwz*, rather than *uxyz*?

A: need to re-define link weights so traffic routing algorithm computes routes accordingly (or need a new routing algorithm)!

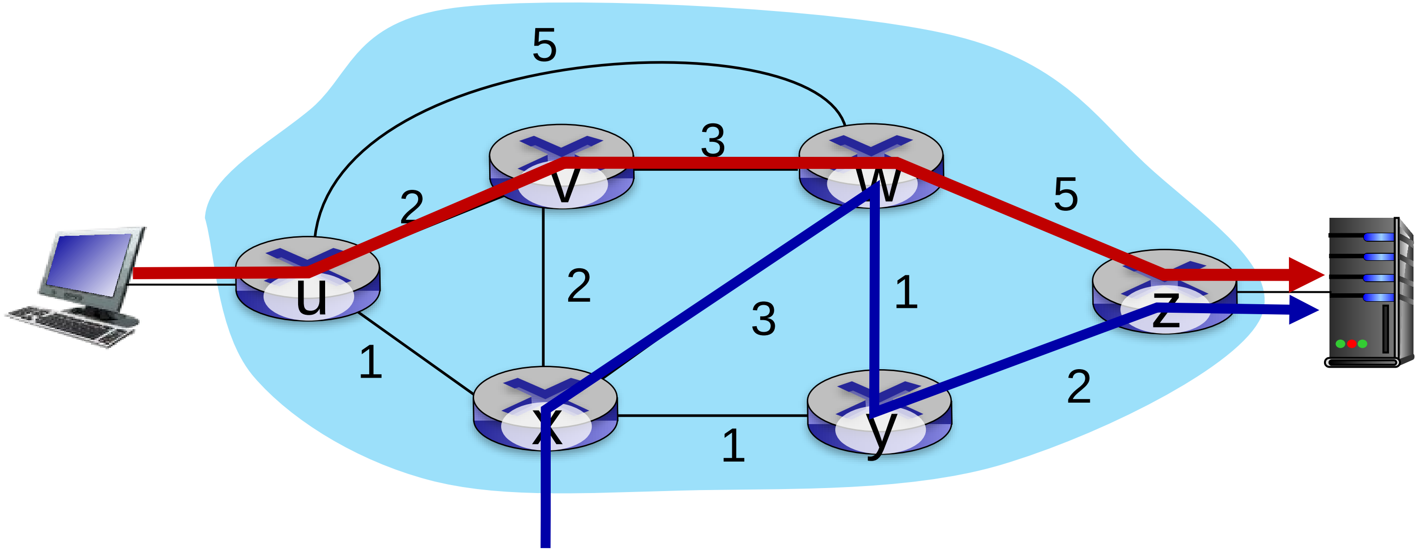
link weights are only control "knobs": not much control!

Traffic engineering: difficult with traditional routing



Q: what if network operator wants to split u-to-z traffic along uvwz *and* uxyz (load balancing)?
A: can't do it (or need a new routing algorithm)

Traffic engineering: difficult with traditional routing



Q: what if w wants to route blue and red traffic differently from w to z?

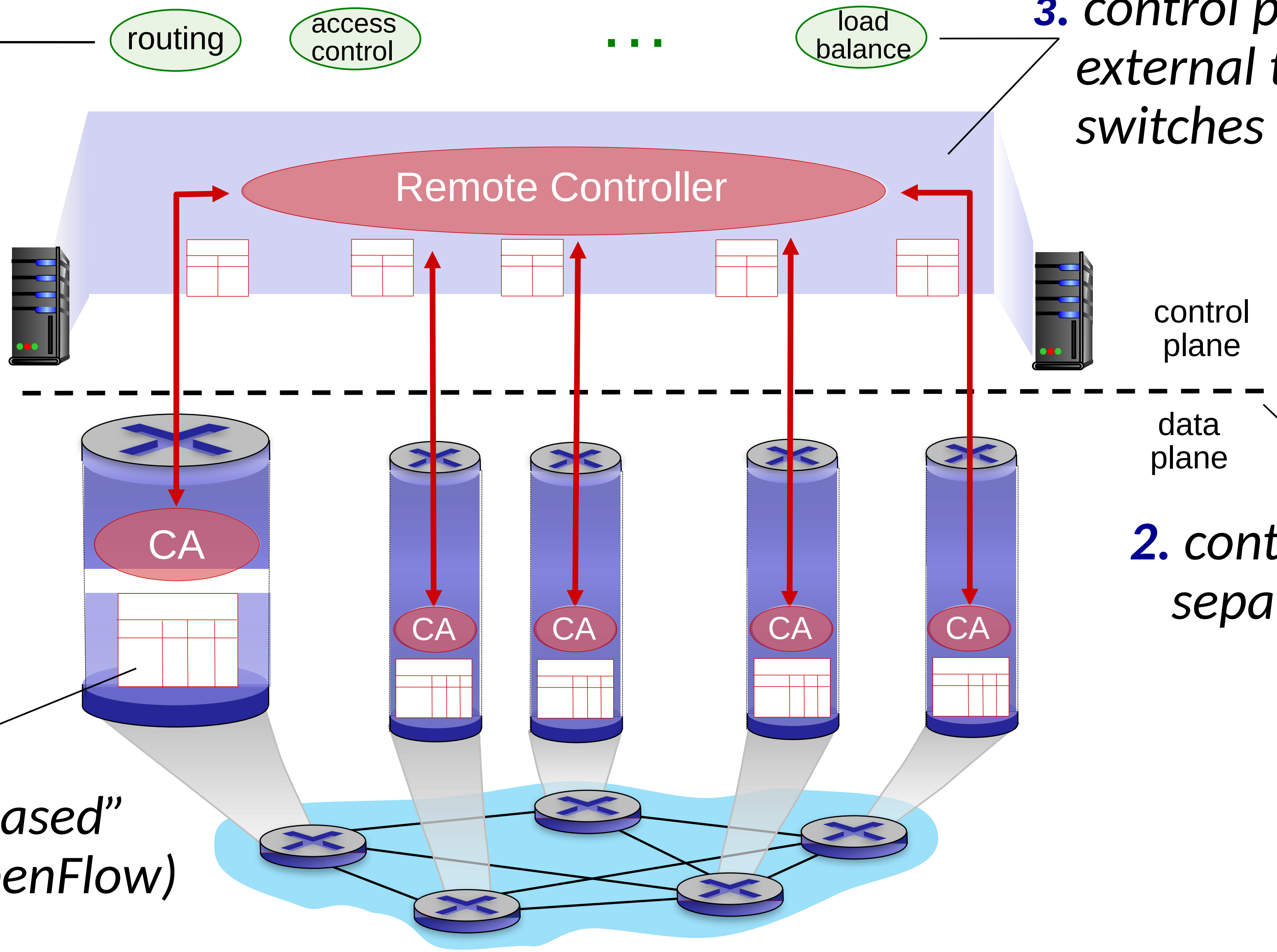
A: can't do it (with destination-based forwarding, and LS, DV routing)

We learned in Chapter 4 that generalized forwarding and SDN can be used to achieve *any* routing desired

Software defined networking (SDN)

4. programmable control applications

3. control plane functions external to data-plane switches



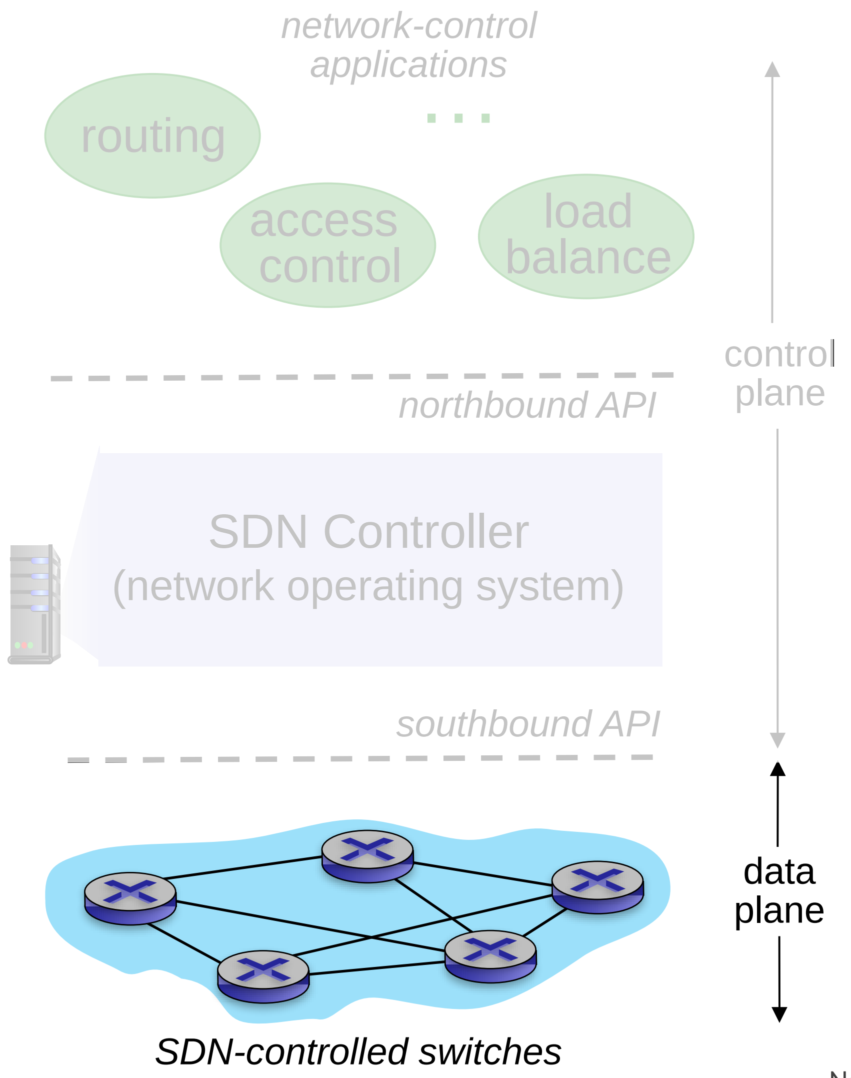
2. control, data plane separation

1: generalized "flow-based" forwarding (e.g., OpenFlow)

Software defined networking (SDN)

Data-plane switches:

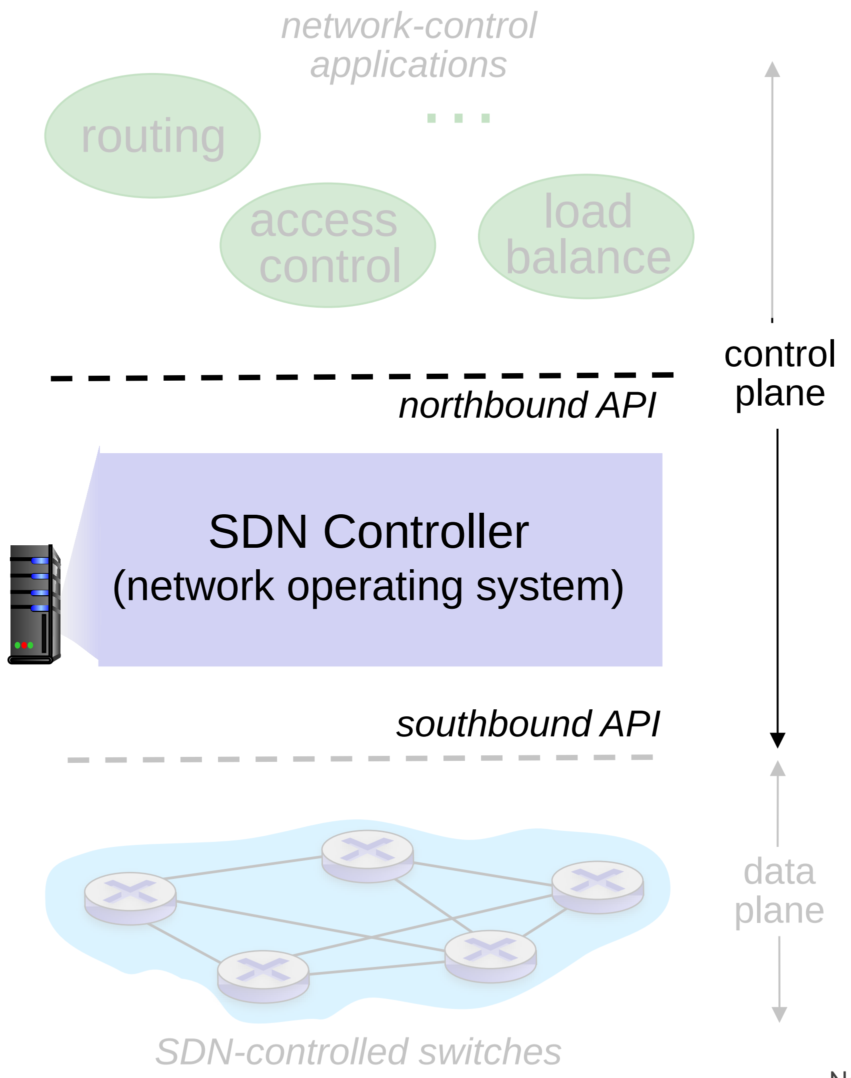
- fast, simple, commodity switches implementing generalized data-plane forwarding (Section 4.4) in hardware
- flow (forwarding) table computed, installed under controller supervision
- API for table-based switch control (e.g., OpenFlow)
  - defines what is controllable, what is not
- protocol for communicating with controller (e.g., OpenFlow)



Software defined networking (SDN)

SDN controller (network OS):

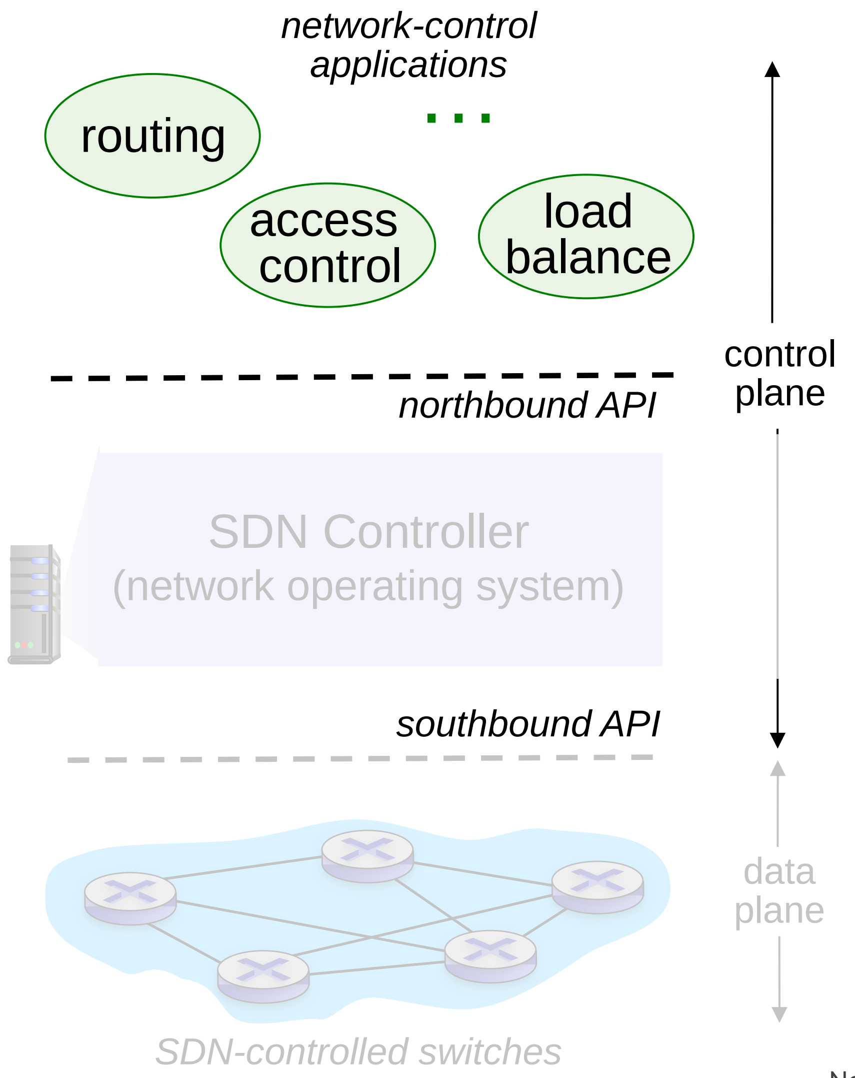
- maintain network state information
- interacts with network control applications "above" via northbound API
- interacts with network switches "below" via southbound API
- implemented as distributed system for performance, scalability, fault-tolerance, robustness



Software defined networking (SDN)

network-control apps:

- “brains” of control: implement control functions using lower-level services, API provided by SDN controller
- *unbundled*: can be provided by 3rd party: distinct from routing vendor, or SDN controller

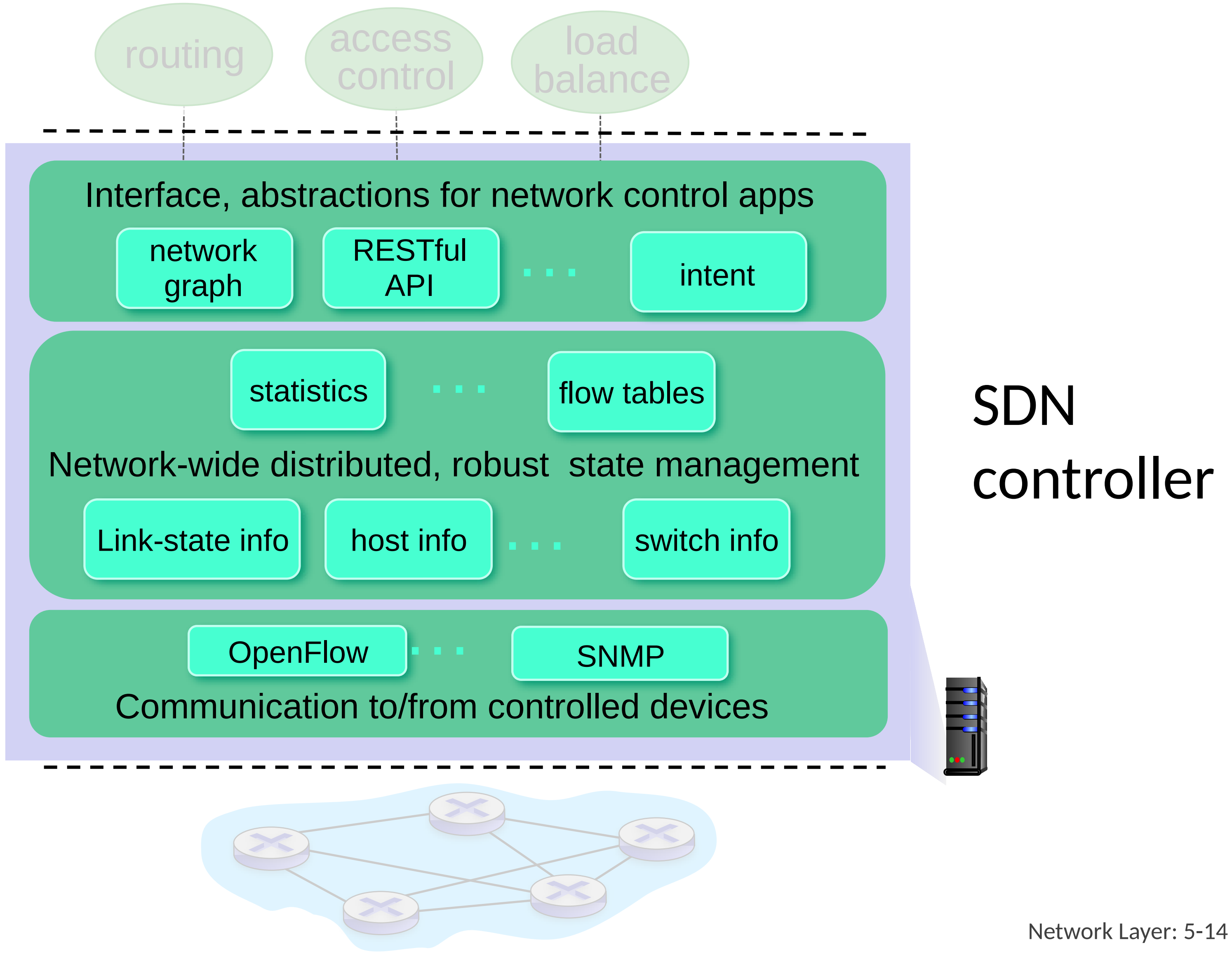


Components of SDN controller

interface layer to network control apps: abstractions API

network-wide state management : state of networks links, switches, services: a *distributed database*

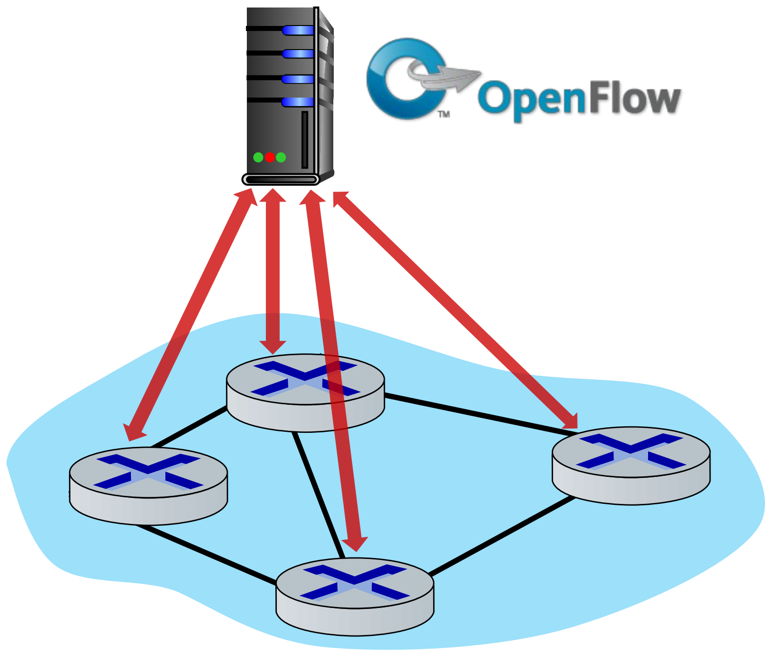
communication: communicate between SDN controller and controlled switches



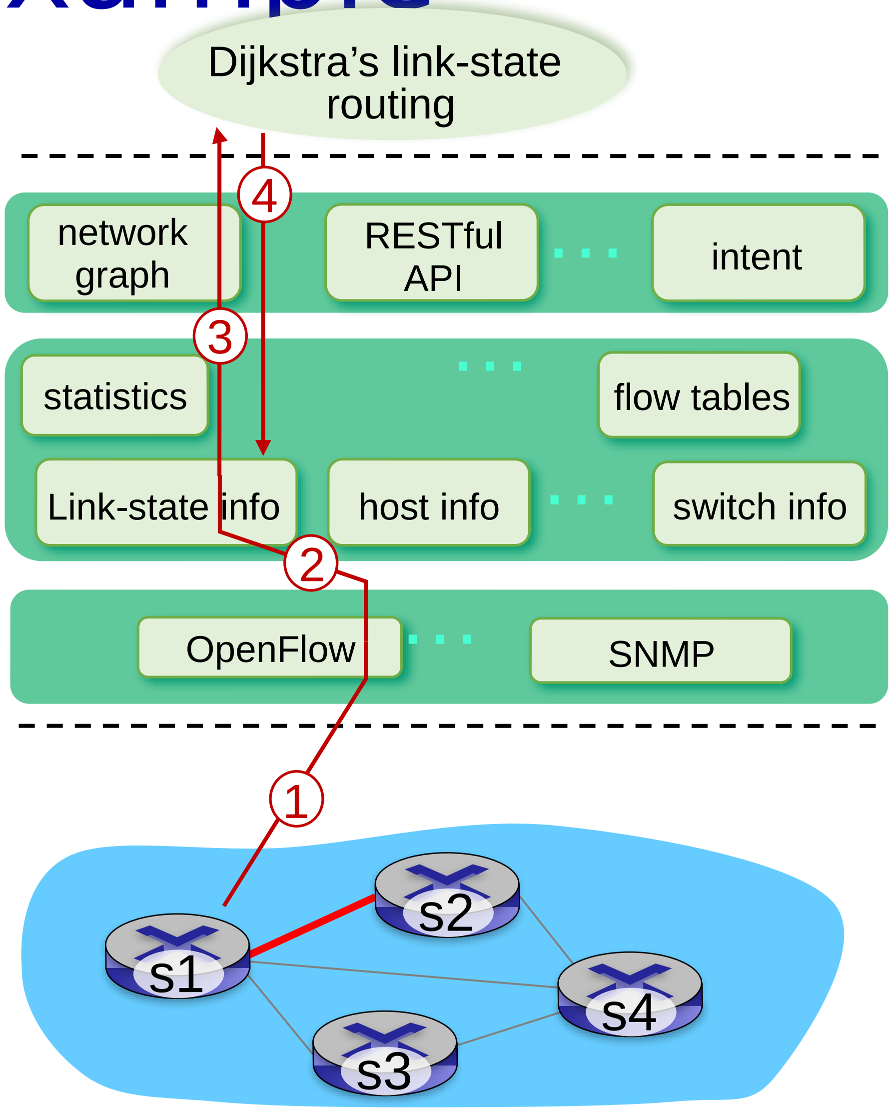
OpenFlow protocol

- operates between controller, switch
- TCP used to exchange messages
  - optional encryption
- three classes of OpenFlow messages:
  - controller-to-switch
  - asynchronous (switch to controller)
  - symmetric (misc.)
- distinct from OpenFlow API
  - API used to specify generalized forwarding actions

OpenFlow Controller

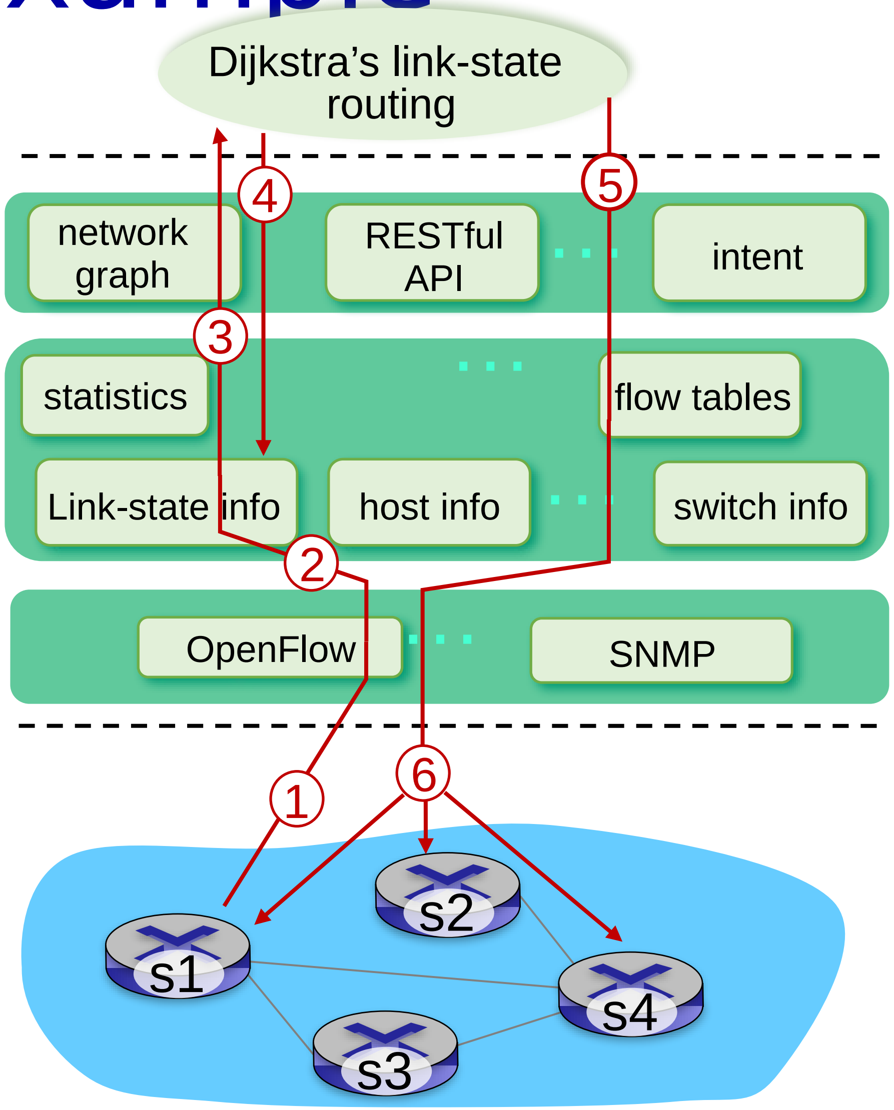


SDN: control/data plane interaction example



1. S1, experiencing link failure uses OpenFlow port status message to notify controller
2. SDN controller receives OpenFlow message, updates link status info
3. Dijkstra's routing algorithm application has previously registered to be called when ever link status changes. It is called.
4. Dijkstra's routing algorithm access network graph info, link state info in controller, computes new routes

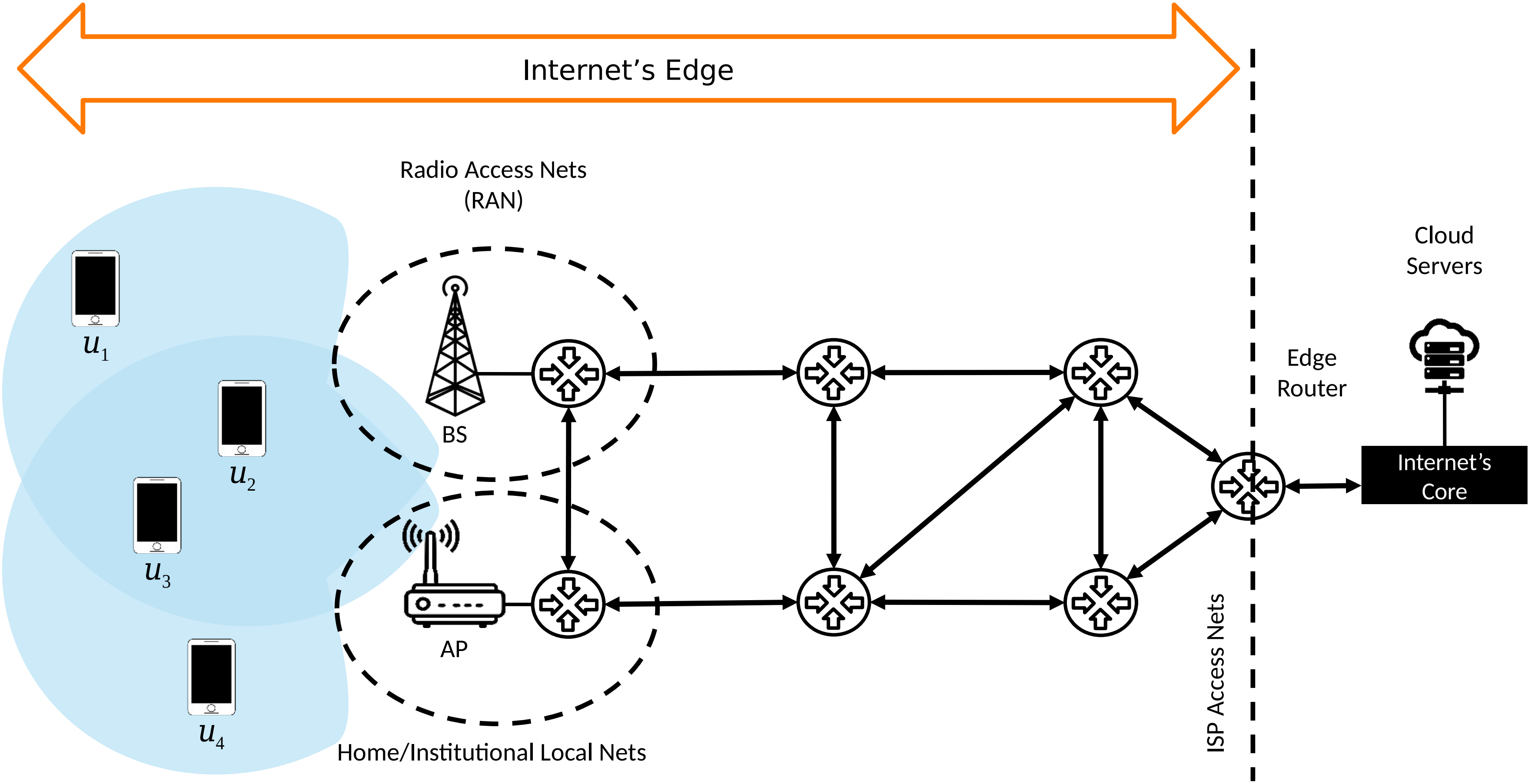
SDN: control/data plane interaction example



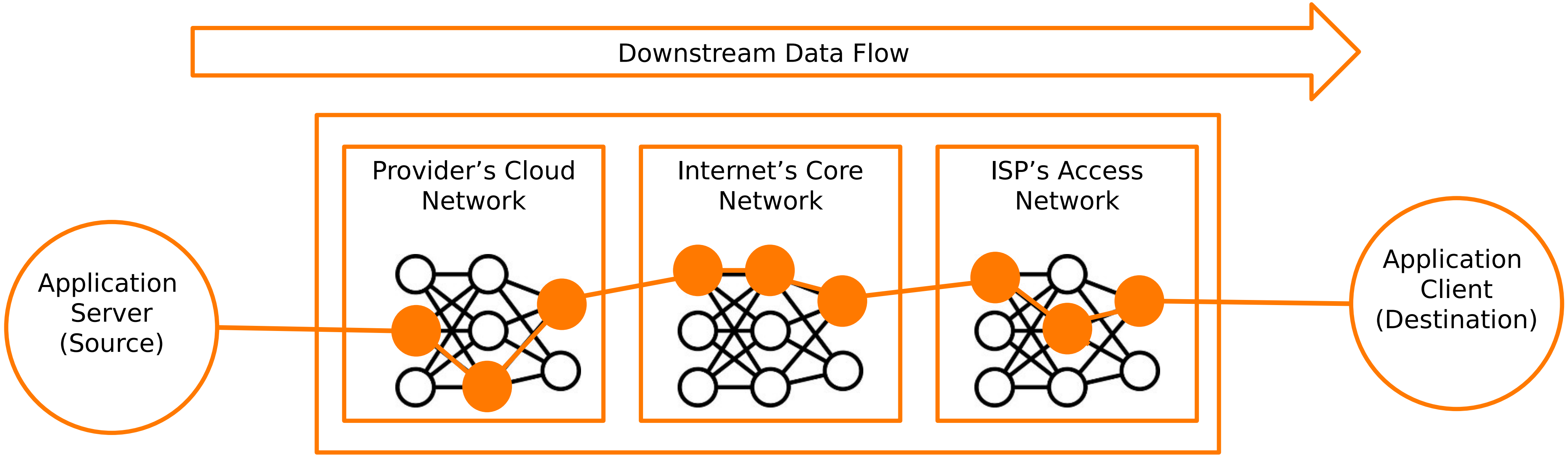
⑤ link state routing app interacts with flow-table-computation component in SDN controller, which computes new flow tables needed

⑥ controller uses OpenFlow to install new tables in switches that need updating

Edge Computing to the rescue!



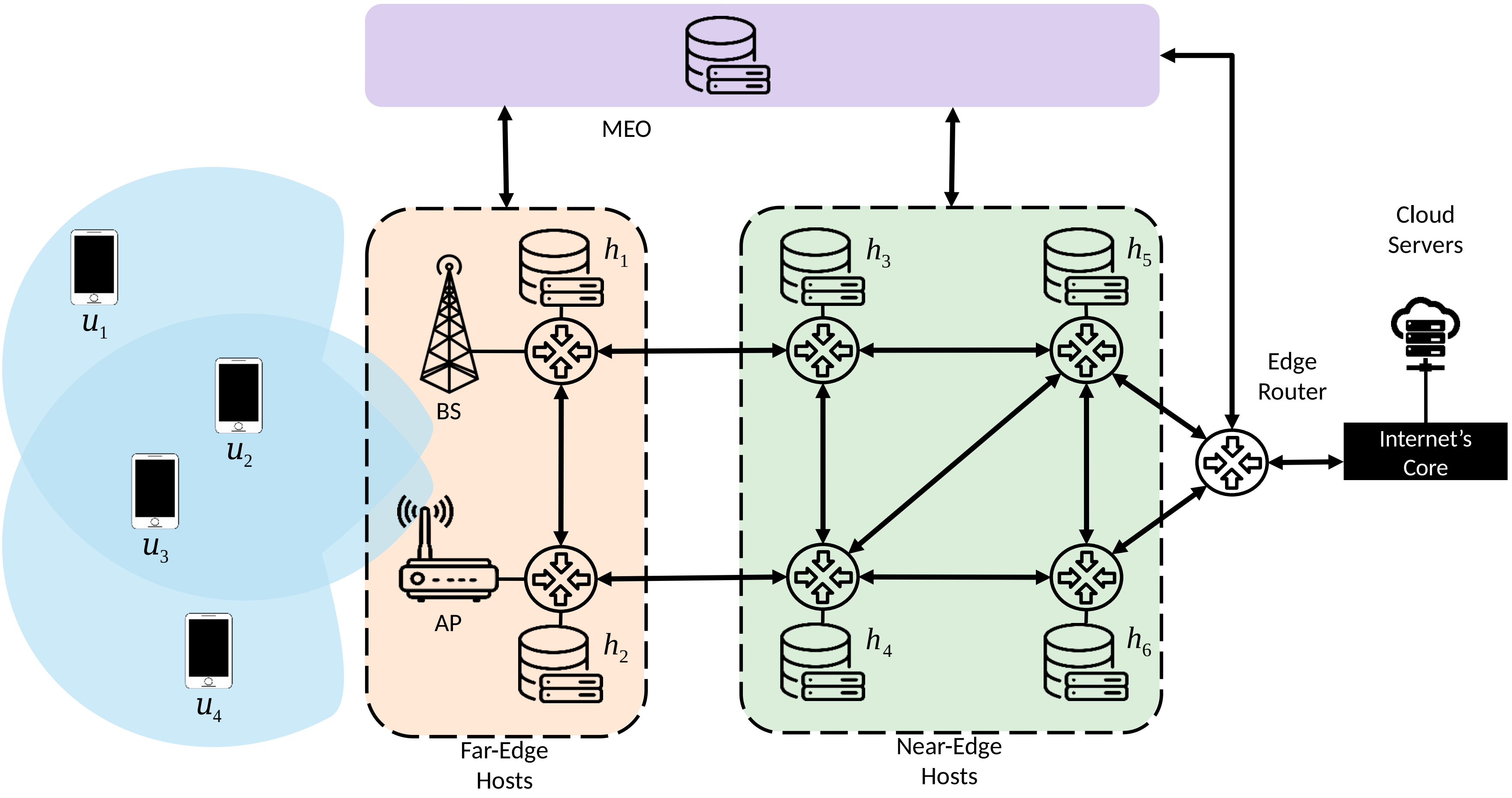
How Edge Computing can help meet strict latency requirements?
Definition of Latency



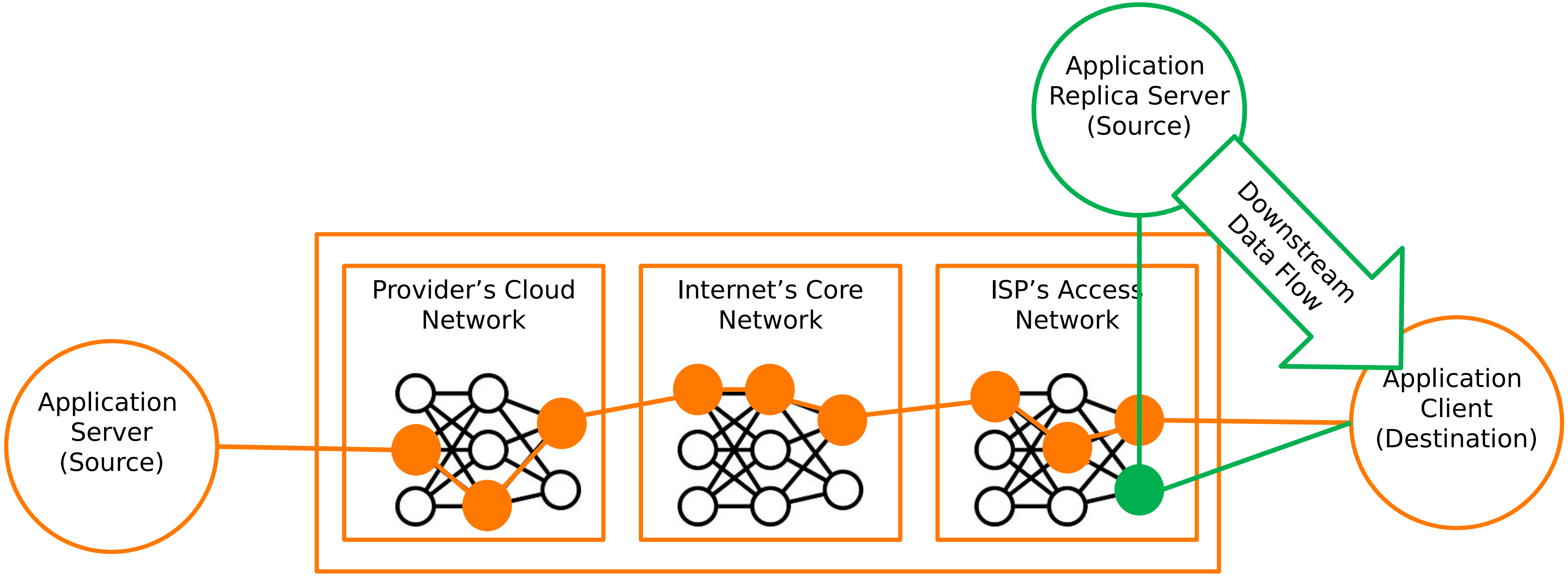
Communication Latency = End-to-End Delay

$$d_{E2E} = \sum_{n \in N_{Internet}} d^{n}_{Nodal}$$

How Edge Computing can help meet strict latency requirements of 5G+ applications?
Definition of Mobile Edge Computing (MEC) Network



How Edge Computing can help meet strict latency requirements?
Definition of Latency (Revisited)



Communication Latency = End-to-End Delay

$$d_{E2E} = \sum_{n \in N_{MEC}} c \quad \boxed{N_{MEC} \vee \ll \vee N_{Internet} \vee} \quad g$$

Network layer, control plane: Done!

- introduction
- routing protocols
  - link state
  - distance vector
- intra-ISP routing: OSPF
- routing among ISPs: BGP
- SDN control plane
- Internet Control Message Protocol

- network management, configuration
  - SNMP
  - NETCONF/YANG

Transport layer: overview

Our goal:

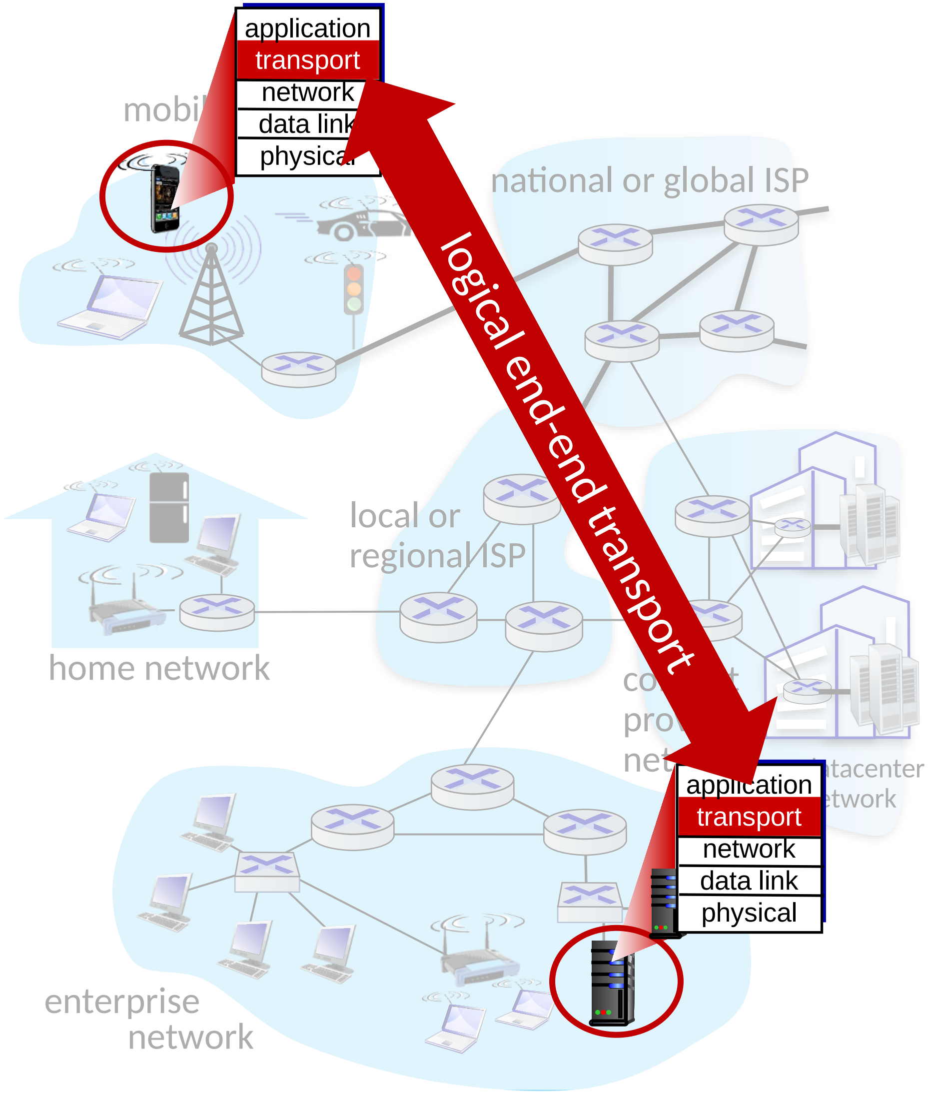
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion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

- 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



Transport vs. network layer services and protocols

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

- network layer: communication between *hosts*

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

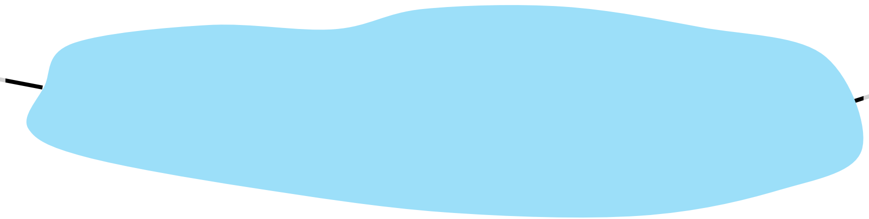
- hosts = houses
- processes = kids
- app messages = letters in envelopes

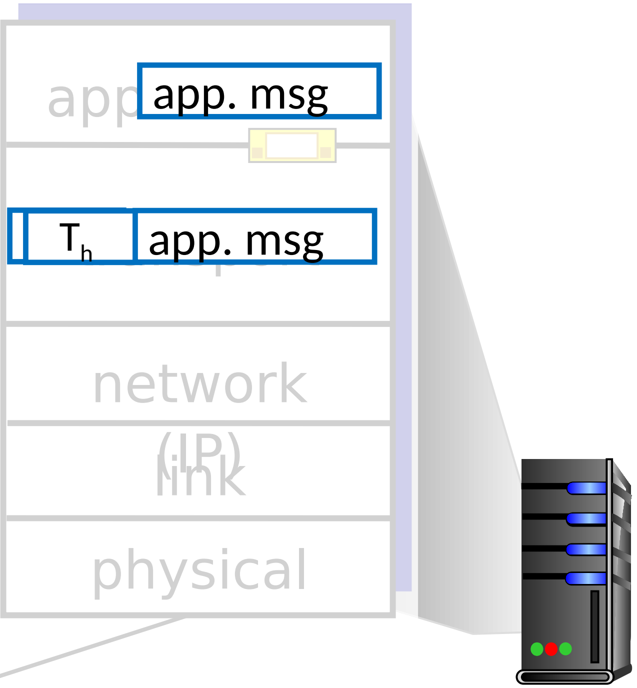
service

Transport Layer Actions

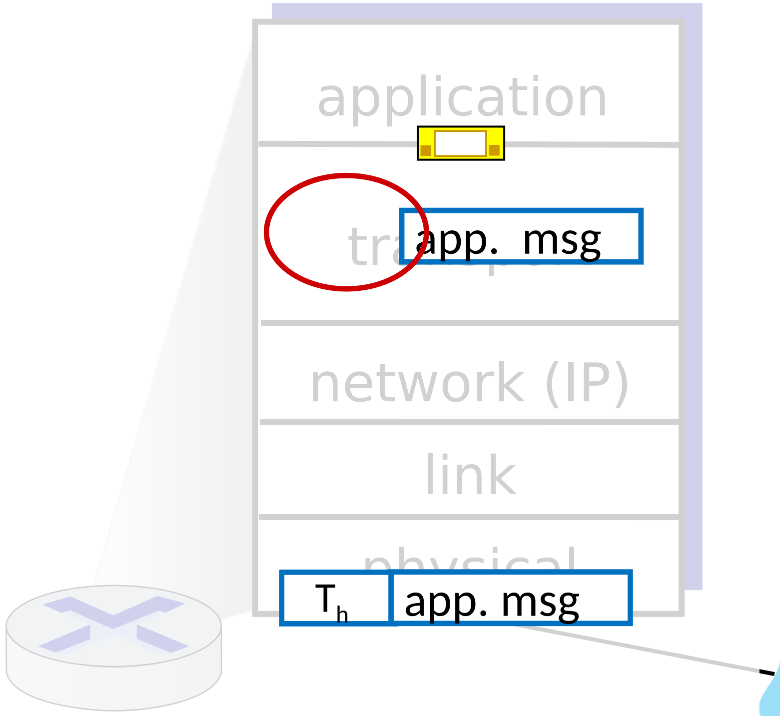
Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP



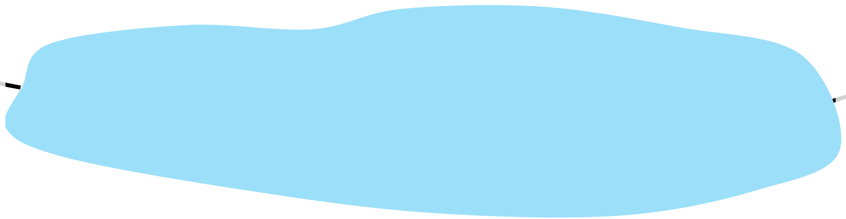


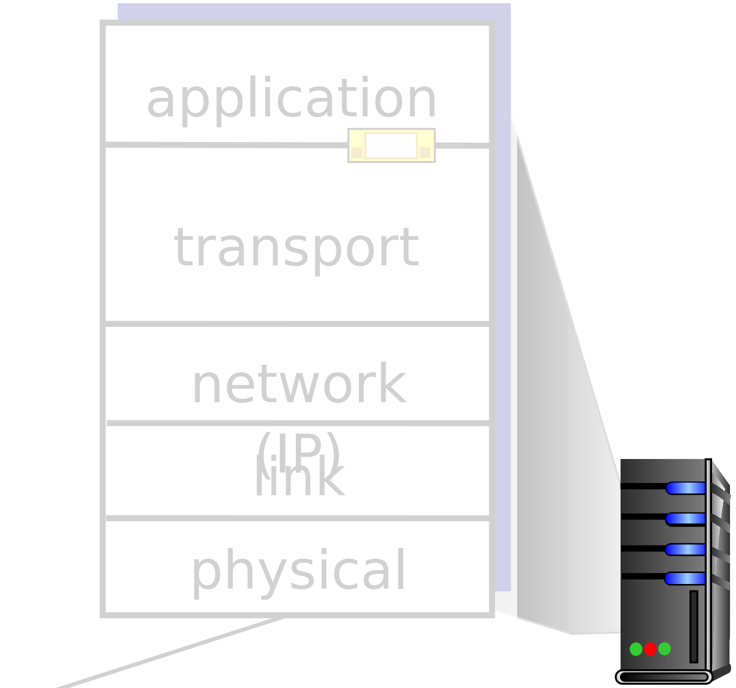
Transport Layer Actions



Receiver:

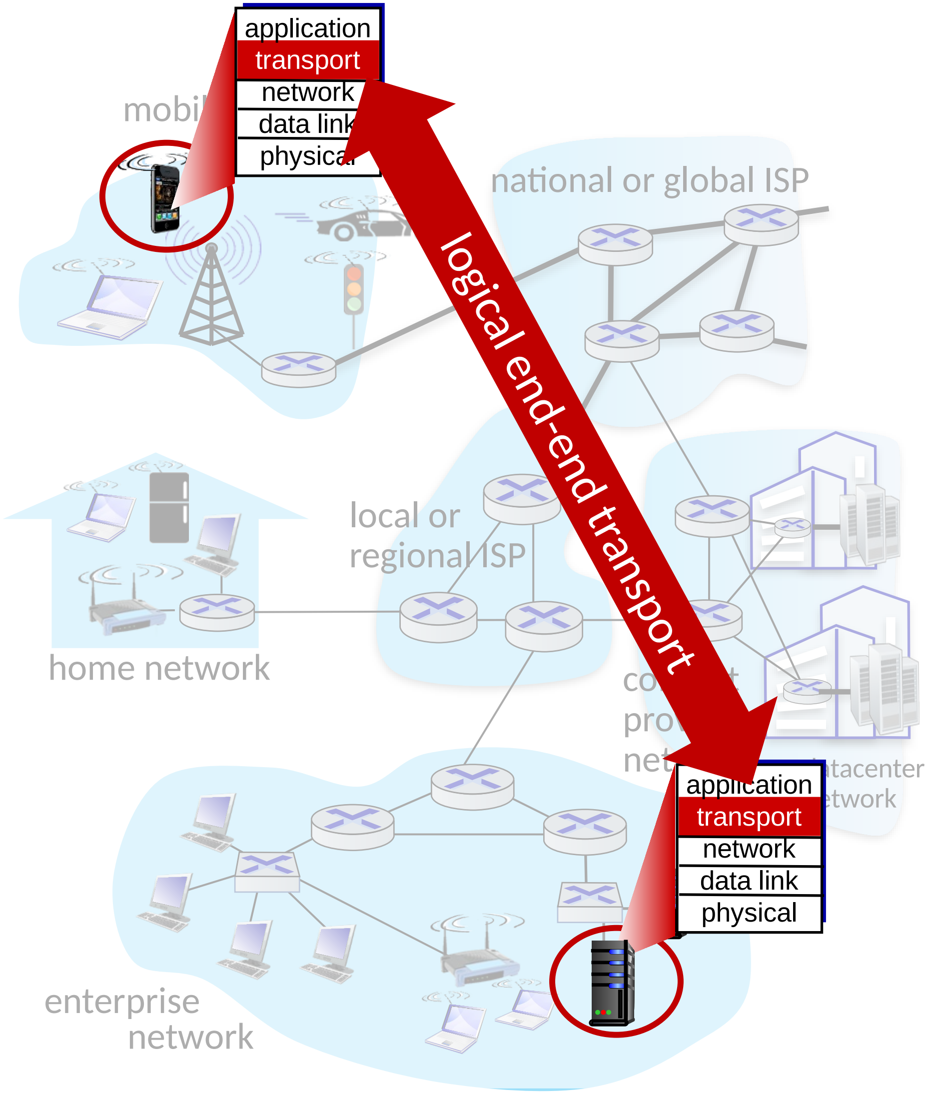
- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket





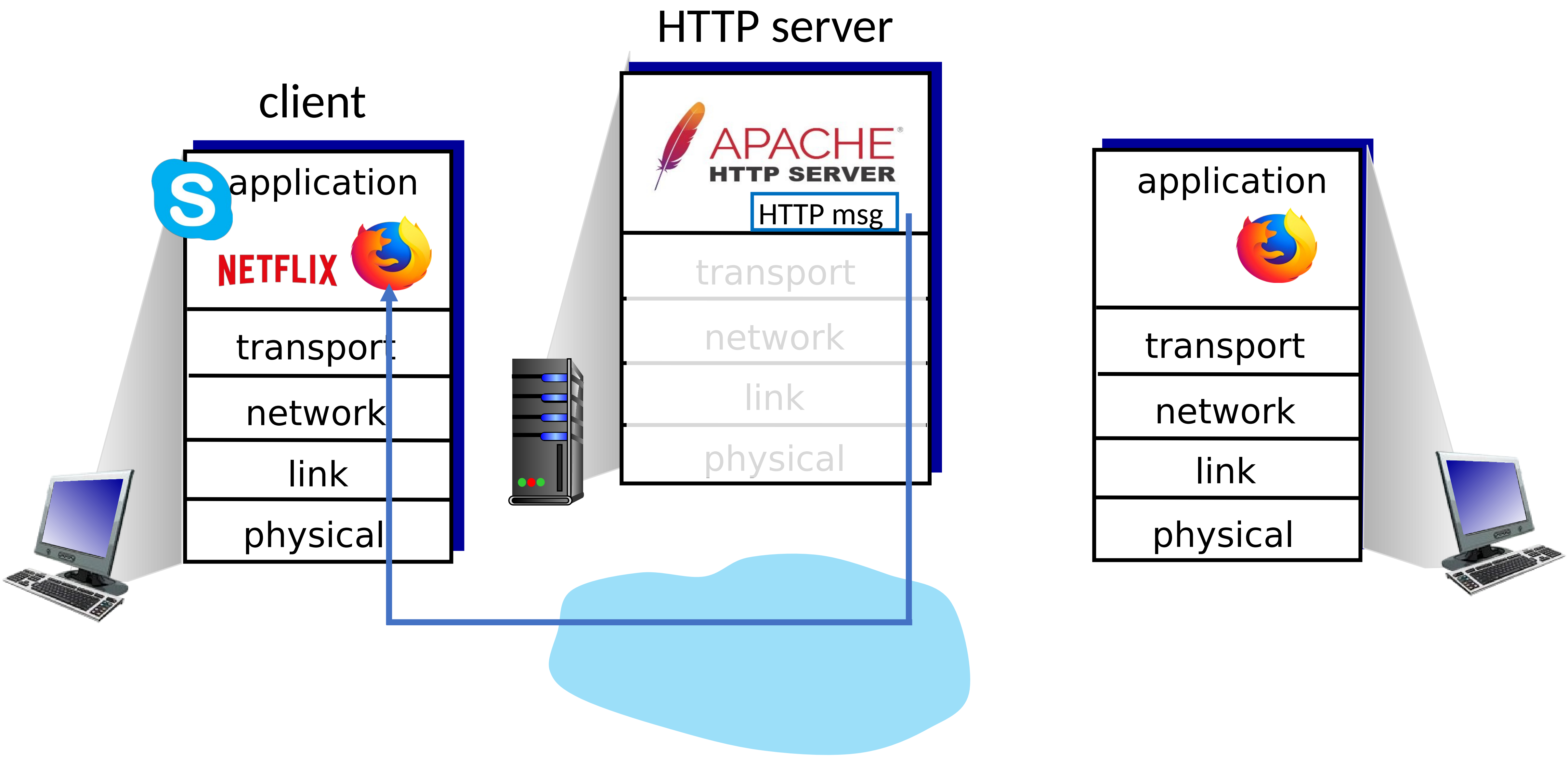
Two principal Internet transport protocols

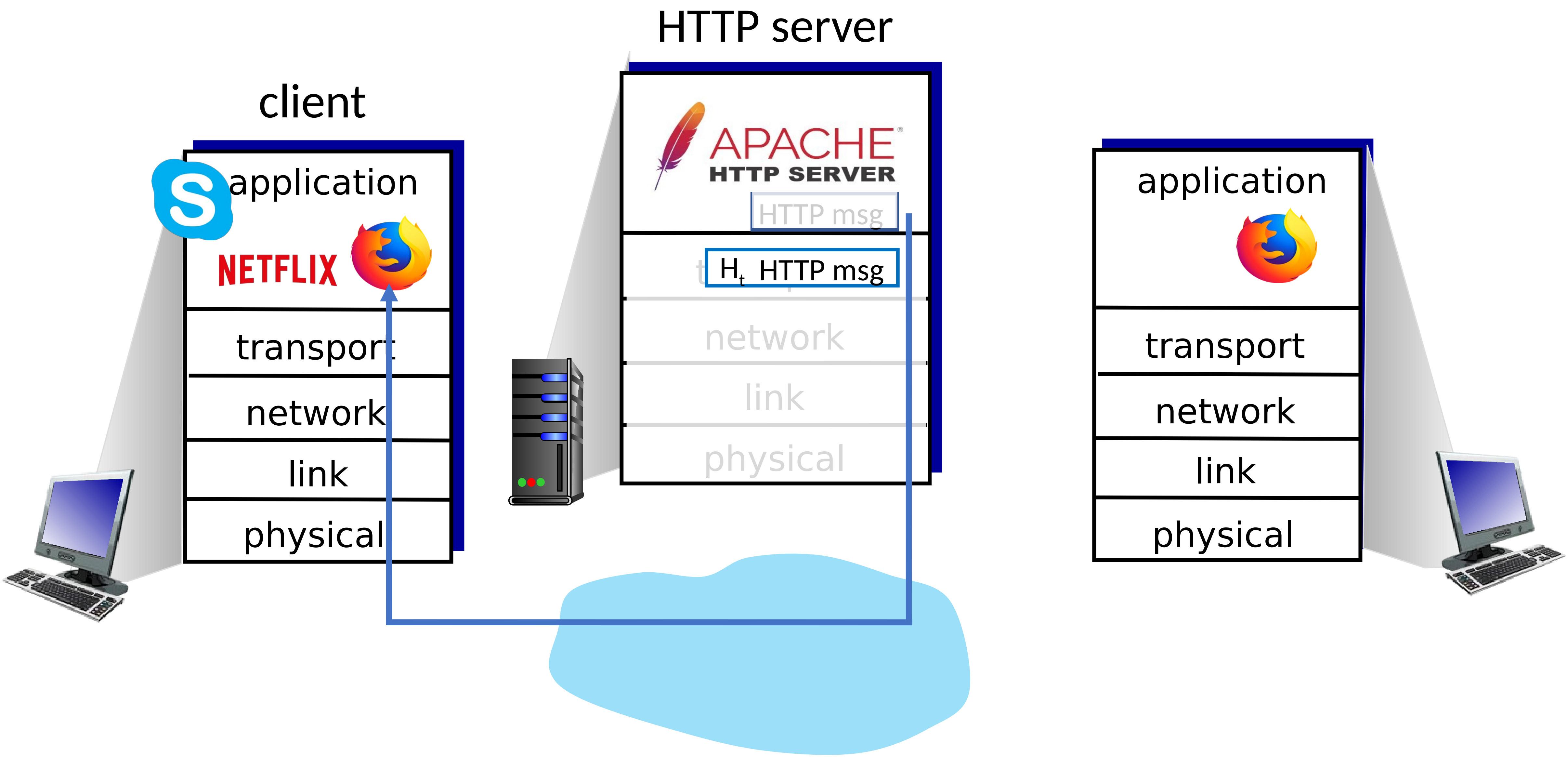
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth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

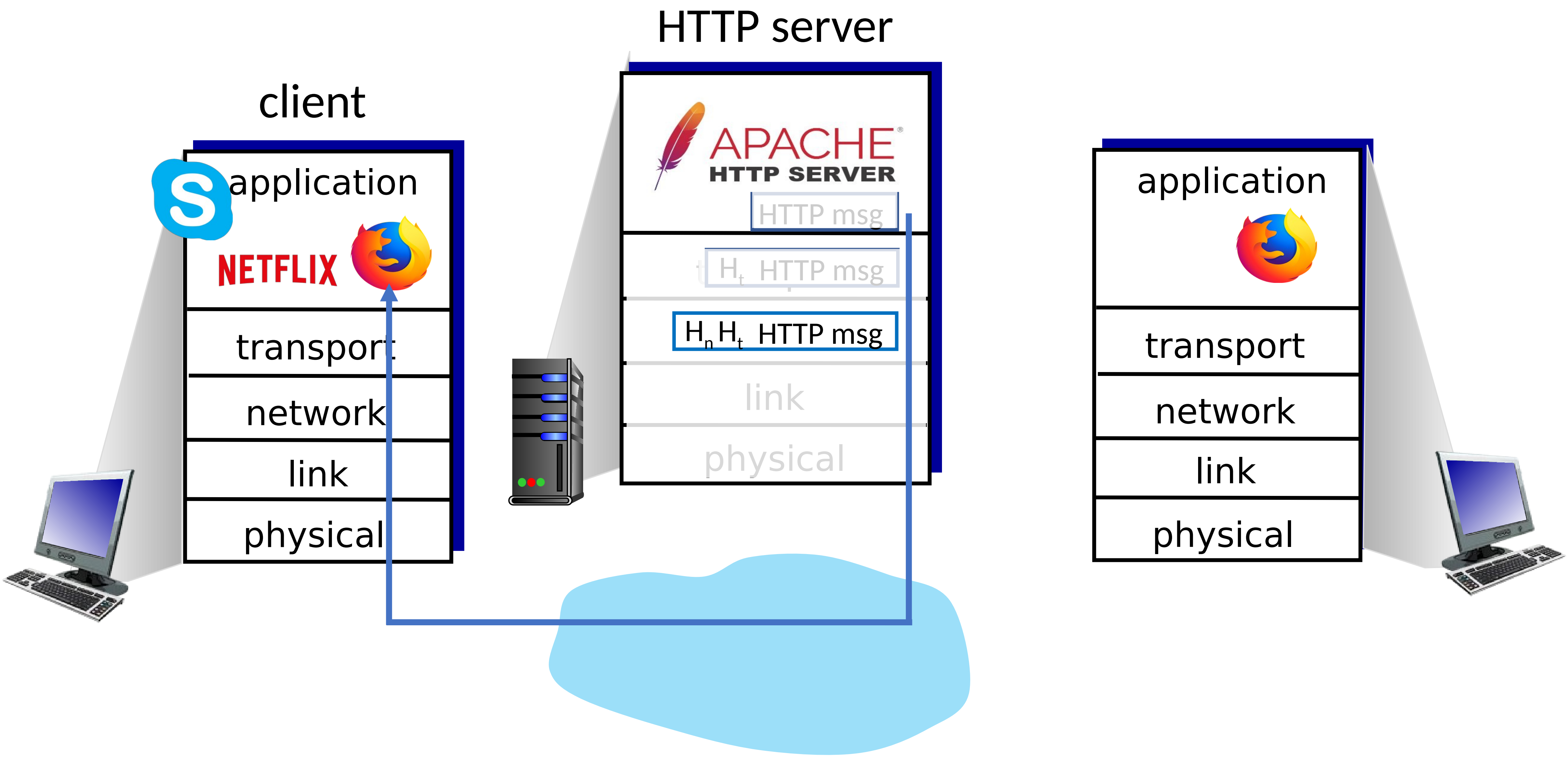


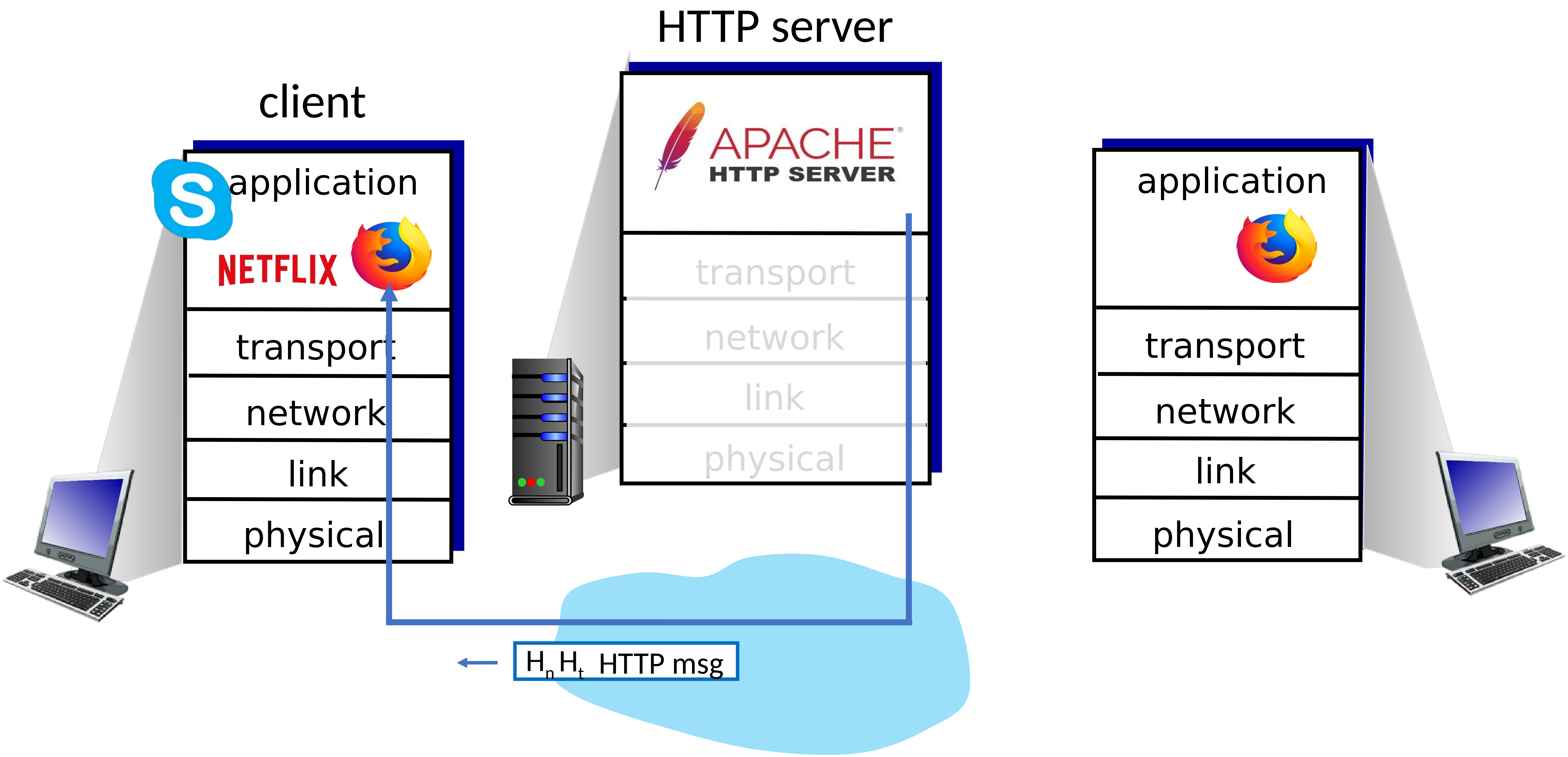
Chapter 3: roadmap

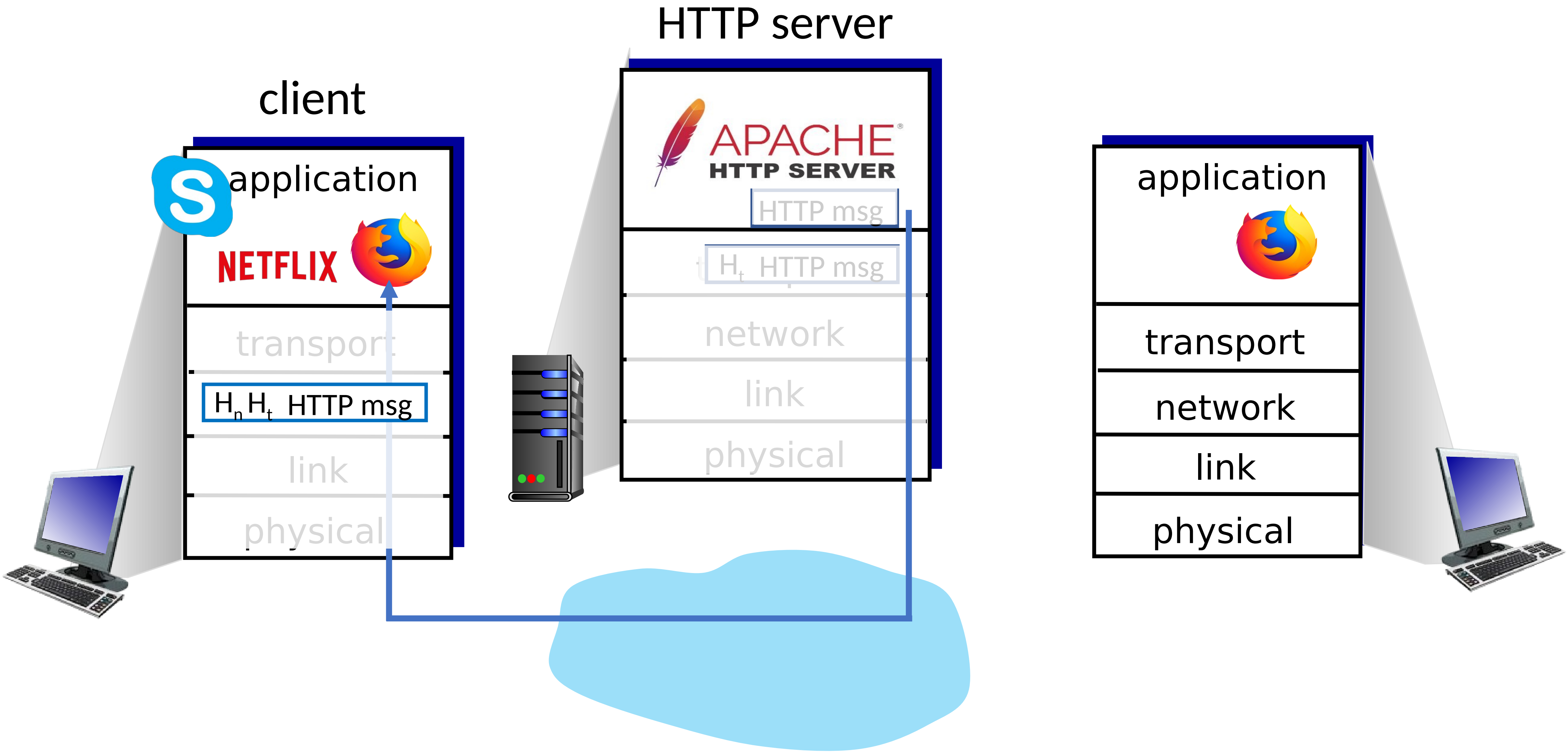
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

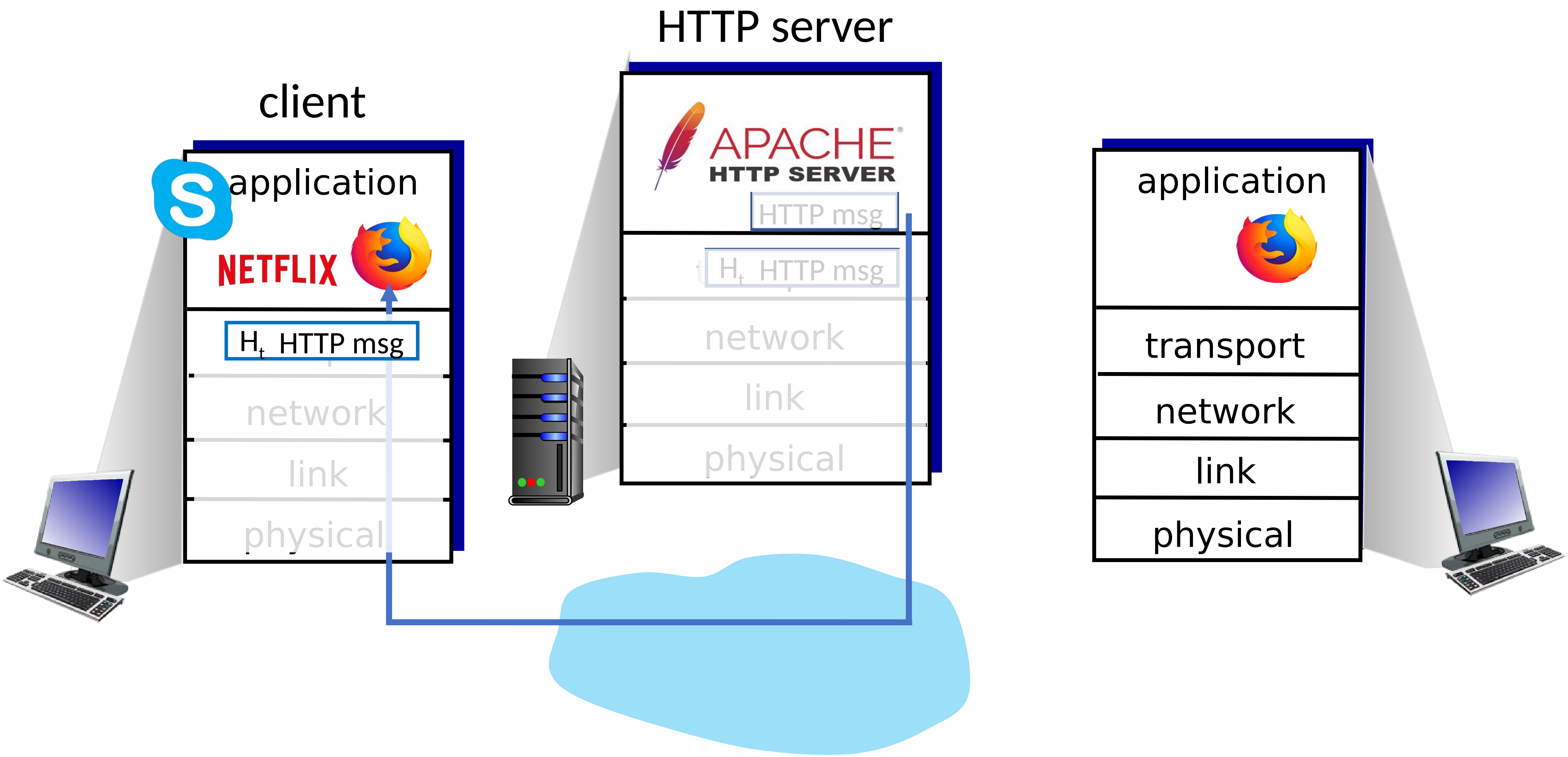

HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H$_t$ HTTP msg
network
link
physical
application
transport
network
link
physical

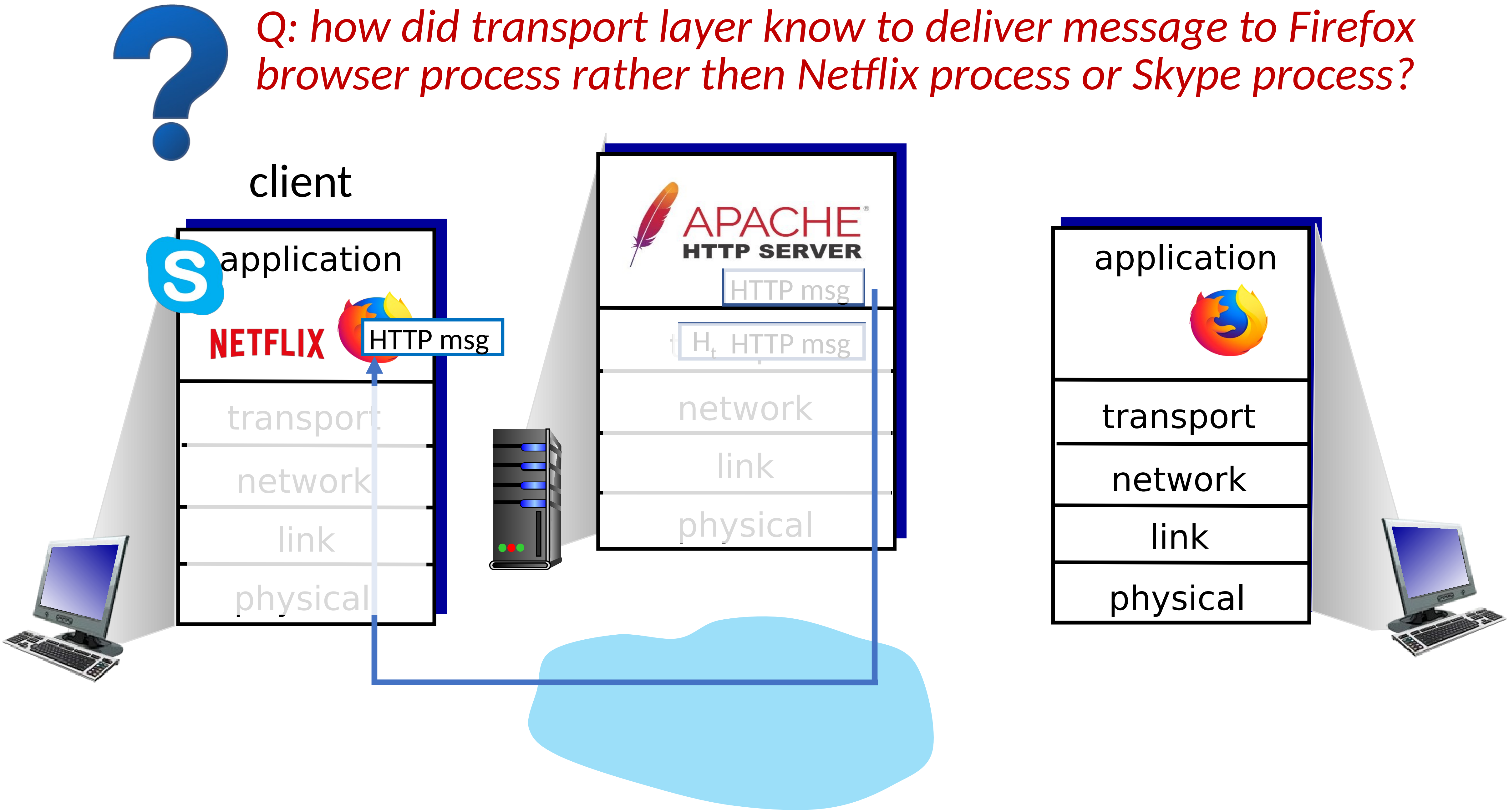

HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H$_t$ HTTP msg
H$_n$ H$_t$ HTTP msg
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
transport
network
link
physical
application
transport
network
link
physical
H_n H_t HTTP msg


HTTP server
client
application
NETFLIX
transport
H_n H_t HTTP msg
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
H$_t$ HTTP msg
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H$_t$ HTTP msg
network
link
physical
application
transport
network
link
physical

Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?



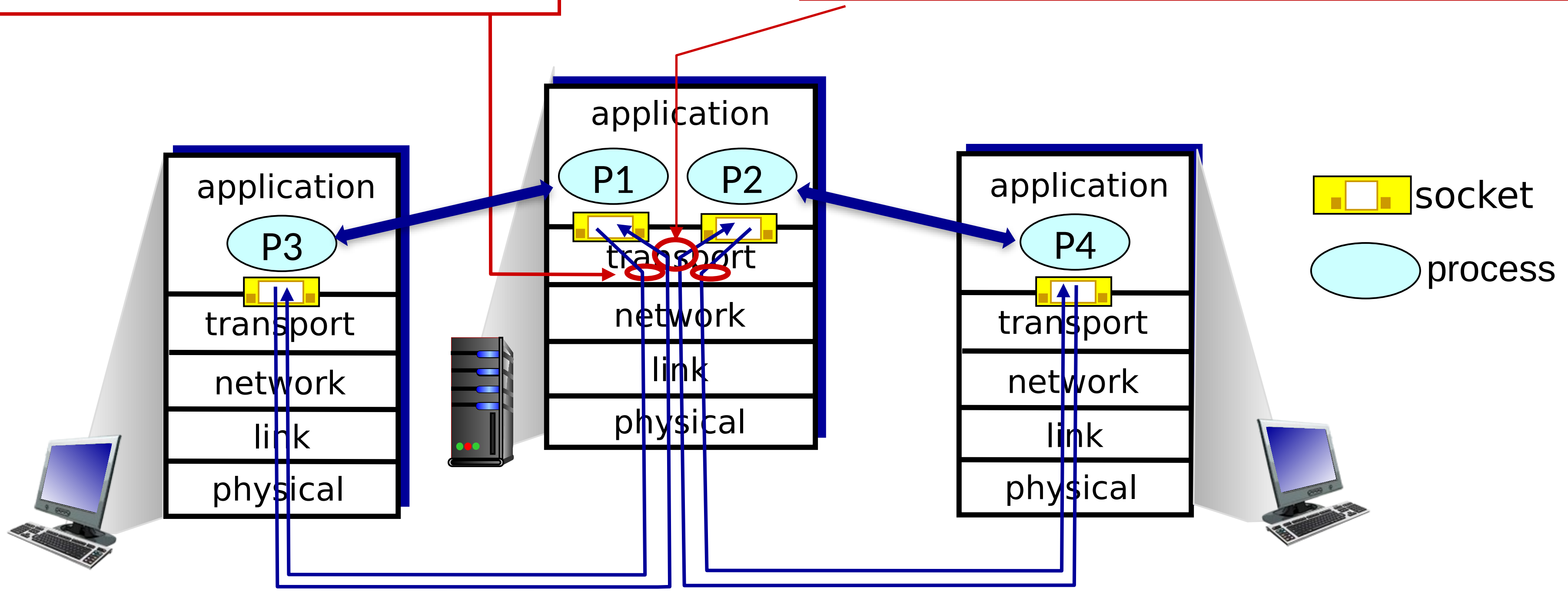
Multiplexing/demultiplexing

multiplexing as sender:

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

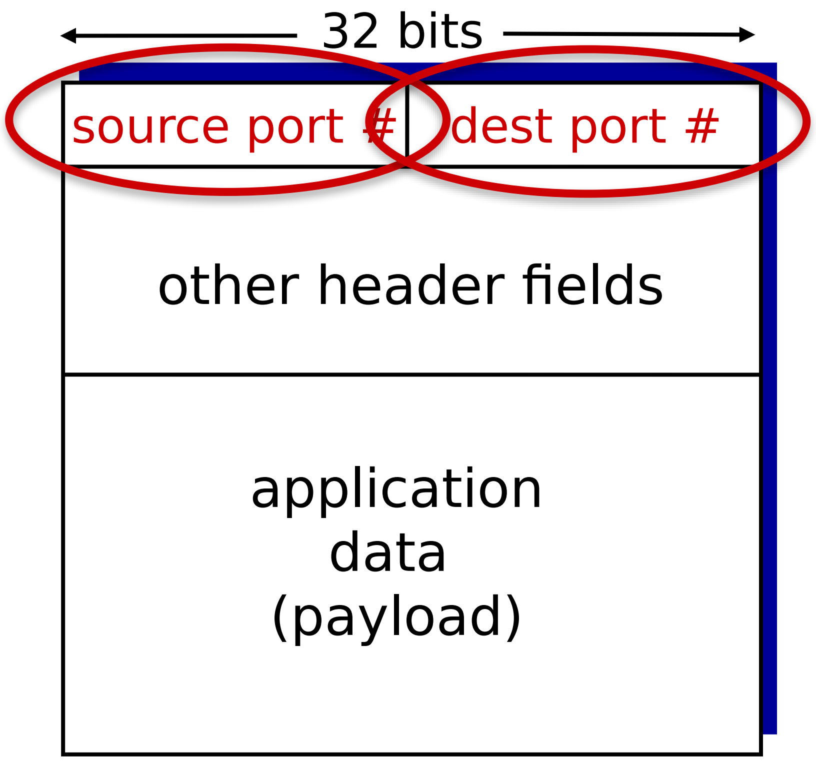
demultiplexing as receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- 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:

- when creating socket, must specify *host-local* port #:

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

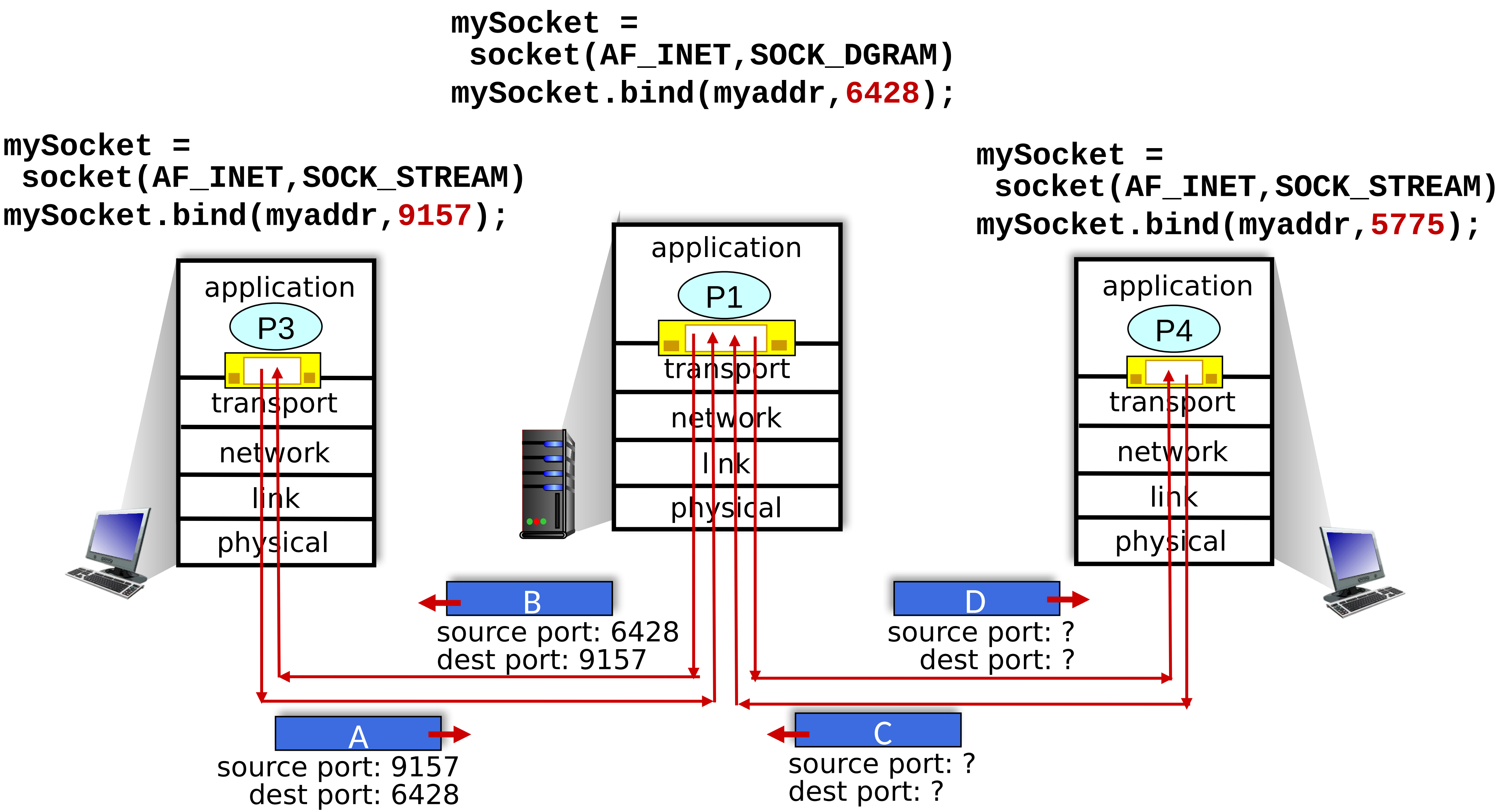
- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

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

Connectionless demultiplexing: an example

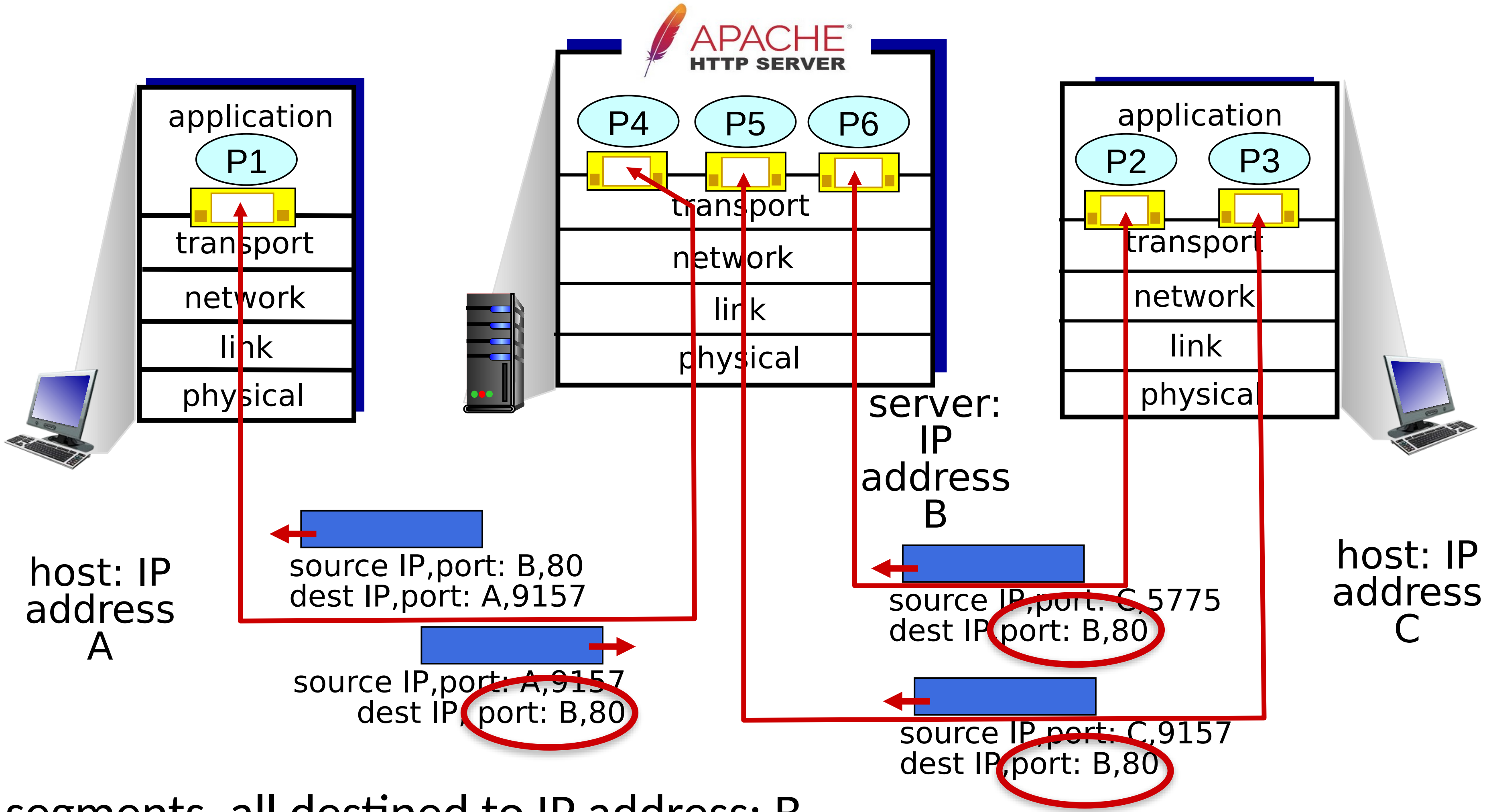


Connection-oriented demultiplexing

- 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

Connection-oriented demultiplexing: example



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

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