Basics of Socket Programming

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Outline

- OSI and TCP/IP Models
- Services
- Encapsulation
- Socket Programing
- UDP Socket
- TCP Socket
- Q & A

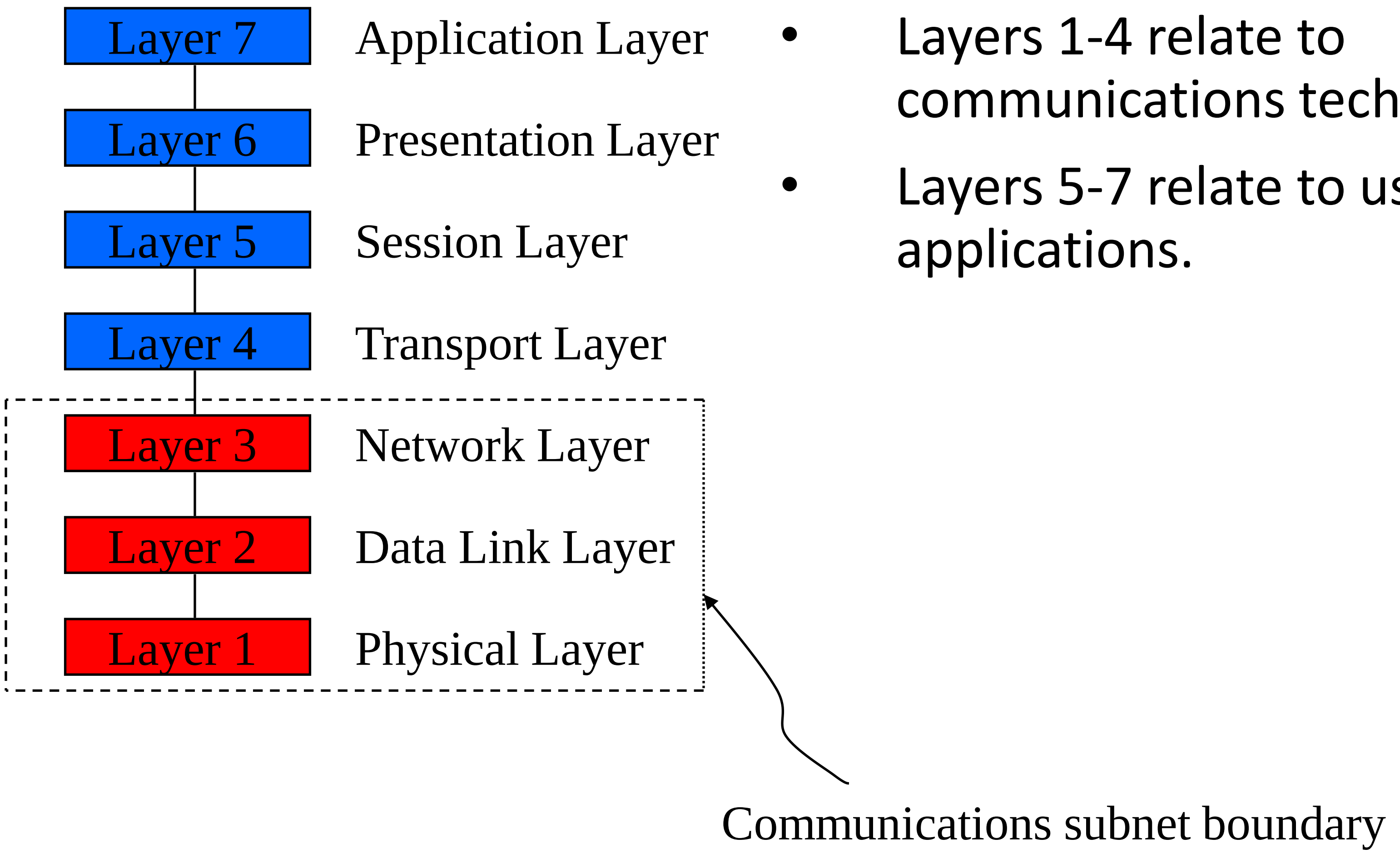
OSI Reference Model

- OSI Reference Model - internationally standardised network architecture.
- OSI = *Open Systems Interconnection*: deals with *open systems*, i.e. systems open for communications with other systems.
- Specified in ISO 7498.
- Model has 7 layers.

OSI History

- In 1978, the International Standards Organization (ISO) began to develop its OSI framework architecture.
- OSI has two major components:
  - an abstract model of networking, called the Basic Reference Model or seven-layer model,
  - a set of specific protocols.
- The concept of a 7-layer model was provided by the work of Charles Bachman, then of Honeywell.
- Various aspects of OSI design evolved from experiences with the Advanced Research Projects Agency Network (ARPANET) and the fledgling Internet.

7-Layer OSI Model



- Layers 1-4 relate to communications technology.
- Layers 5-7 relate to user applications.

Layer 7: Application Layer

- Level at which applications access network services.
  - Represents services that directly support software applications for file transfers, database access, and electronic mail etc.

Layer 6: Presentation Layer

- Related to representation of transmitted data
  - Translates different data representations from the Application layer into uniform standard format
- Providing services for secure efficient data transmission
  - e.g. data encryption, and data compression.

Layer 5: Session Layer

- Allows two applications on different computers to establish, use, and end a session.
  - e.g. file transfer, remote login
- Establishes dialog control
  - Regulates which side transmits, plus when and how long it transmits.
- **Performs *token management* and *synchronization*.**

Layer 4: Transport Layer

- Manages transmission packets
  - Repackages long messages when necessary into small packets for transmission
  - Reassembles packets in correct order to get the original message.
- Handles error recognition and recovery.
  - Transport layer at receiving acknowledges packet delivery.
  - Resends missing packets

Layer 3: Network Layer

- Manages addressing/routing of data within the subnet
    - Addresses messages and translates logical addresses and names into physical addresses.
    - Determines the route from the source to the destination computer
    - Manages traffic problems, such as switching, routing, and controlling the congestion of data packets.
- Routing can be:
    - Based on static tables
    - determined at start of each session
    - Individually determined for each packet, reflecting the current network load.

Layer 2: Data Link Layer

- Packages raw bits from the Physical layer into frames (logical, structured packets for data).
- Provides reliable transmission of frames
  - It waits for an acknowledgment from the receiving computer.
  - Retransmits frames for which acknowledgement not received

Layer 1: Physical Layer

- Transmits bits from one computer to another
- Regulates the transmission of a stream of bits over a physical medium.
- Defines how the cable is attached to the network adapter and what transmission technique is used to send data over the cable. Deals with issues like
  - The definition of 0 and 1, e.g. how many volts represents a 1, and how long a bit lasts?
  - Whether the channel is simplex or duplex?
  - How many pins a connector has, and what the function of each pin is?

Internet Protocols vs OSI

OSI	Internet Protocols
Application	Application
Presentation	
Session	
Transport	TCP
Network	IP
	Network Interface
Data Link	
Physical	Hardware

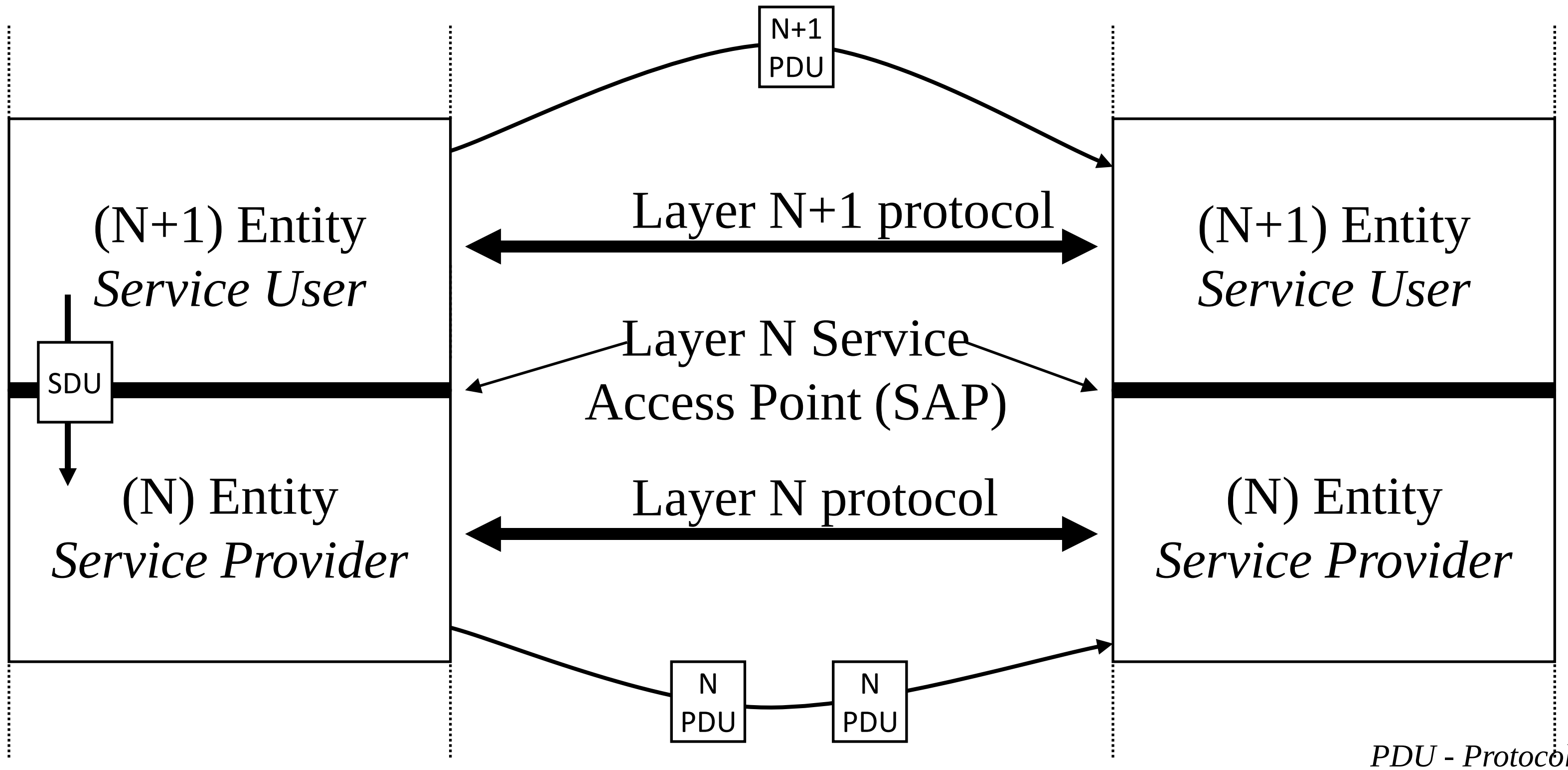
- Explicit Presentation and session layers missing in Internet Protocols
- Data Link and Network Layers redesigned

Services in the OSI Model

BENNETT UNIVERSITY
THE TIMES GROUP

- In OSI model, *each layer provide services to layer above, and ‘consumes’ services provided by layer below.*
- Active elements in a layer called ***entities.***
- Entities in same layer in different machines called ***peer entities.***

Layering Principles



PDU - Protocol Data Unit
SDU - Service Data Unit

- Layer N provides service to layer N+1

Connections

- Layers can offer ***connection-oriented* or *connectionless* services.**
- Connection-oriented like telephone system.
- Connectionless like postal system.
- Each service has an associated ***Quality-of-service*** (e.g. reliable or unreliable).

Reliability

- Reliable services ***never lose/corrupt data.***
- Reliable service ***costs more.***
- Typical application for reliable service is file transfer.
- Typical application not needing reliable service is voice traffic.
- Not all applications need connections.

Topics

- Service = set of primitives provided by one layer to layer above.
- Service defines what layer can do (but not how it does it).
- Protocol = *set of rules governing data communication between peer entities, i.e. format and meaning of frames/packets.*
- Service/protocol decoupling very important.

TCP/IP Model

- Protocol Layers
- Services
- Encapsulation

Protocol “layers” and reference models

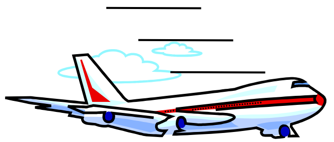
Networks are complex, with many “pieces”:

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

Question: is there any hope of *organizing* structure of network?

- and/or our *discussion* of networks?

Example: organization of air travel



end-to-end transfer of person plus baggage

ticket (purchase)	ticket (complain)
baggage (check)	baggage (claim)
gates (load)	gates (unload)
runway takeoff	runway landing
airplane routing	airplane routing

airplane routing

How would you *define/discuss* the *system* of airline travel?

- a series of steps, involving many services

Example: organization of air travel

layers: each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

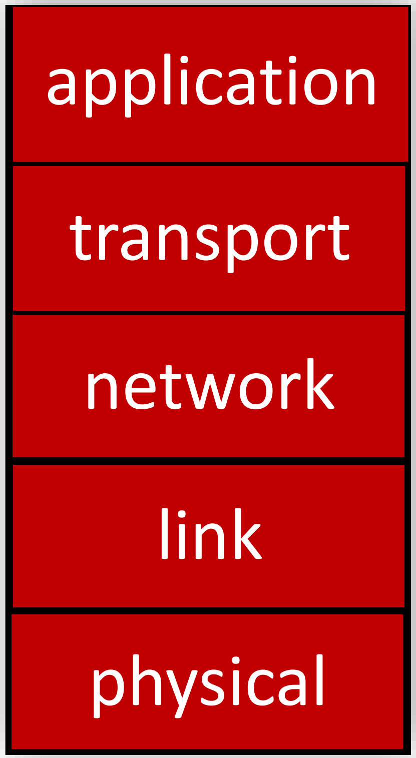
Why layering?

Approach to designing/discussing complex systems:

- explicit structure allows identification, relationship of system’s pieces
  - layered *reference model* for discussion
- modularization eases maintenance, updating of system
  - change in layer's service *implementation*: transparent to rest of system
  - e.g., change in gate procedure doesn’t affect rest of system

Layered Internet protocol stack

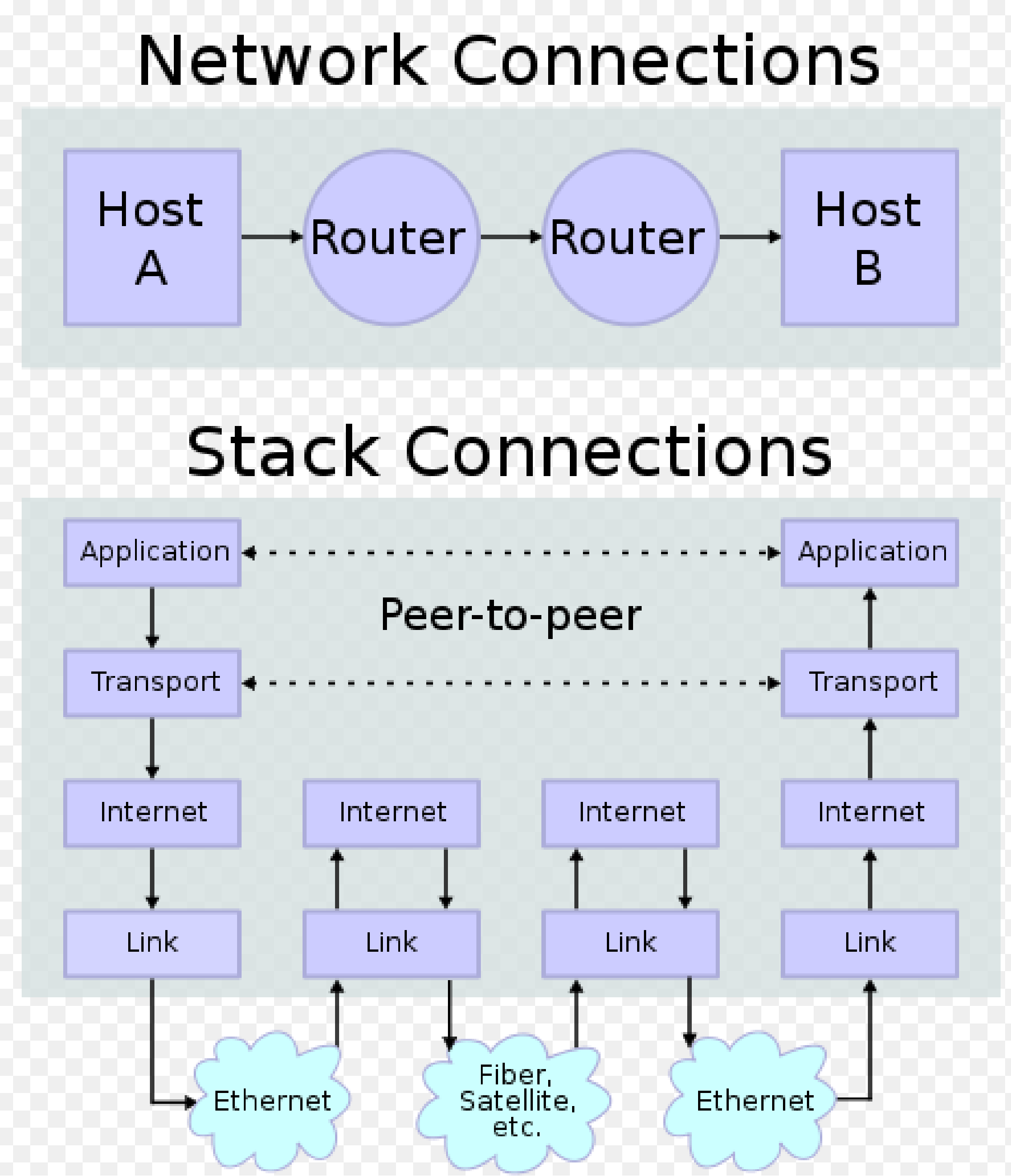
- *application:* supporting network applications
  - HTTP, IMAP, SMTP, DNS
- *transport:* process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.11 (WiFi), PPP
- *physical:* bits “on the wire”



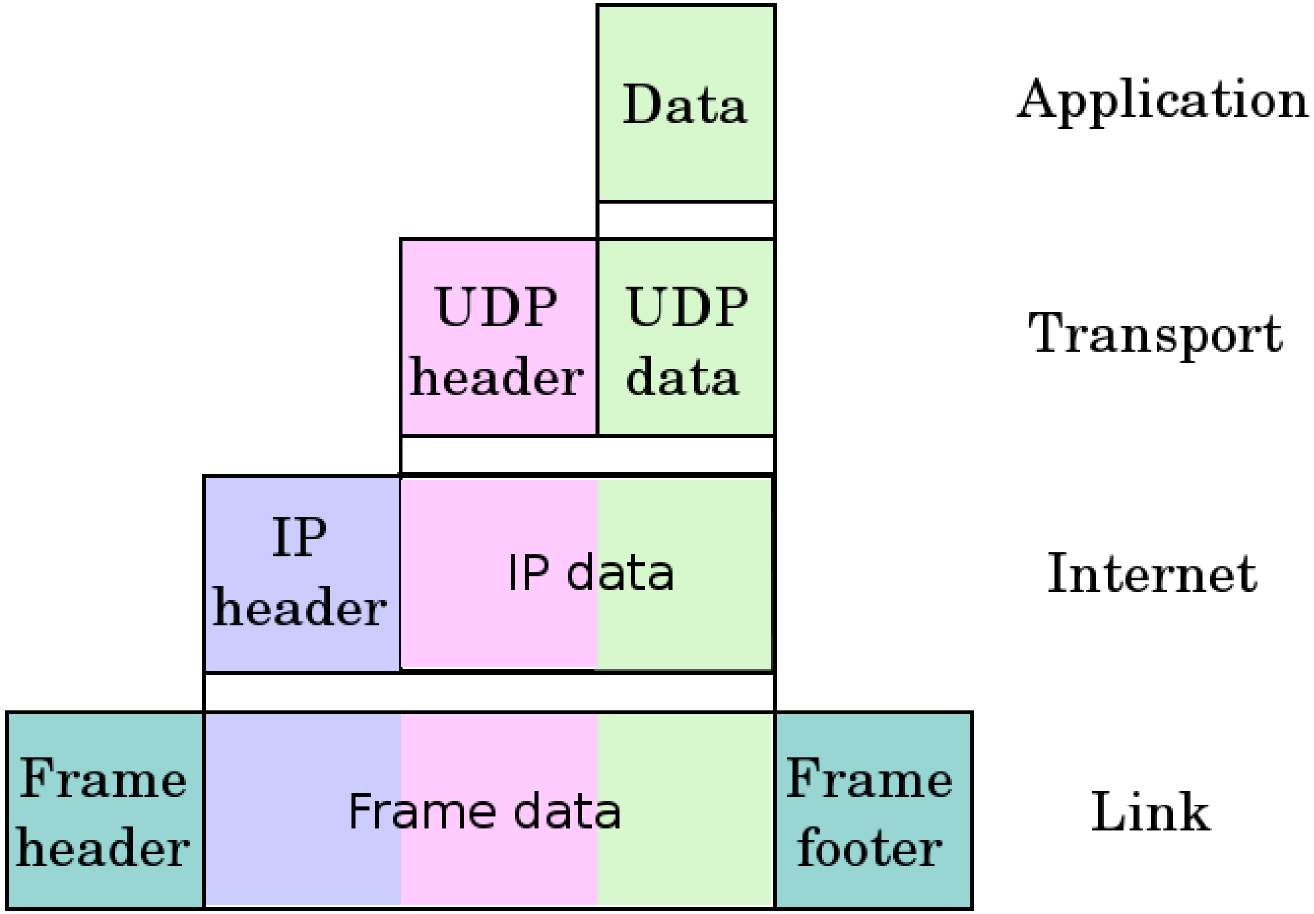
TCP/IP Layers

OSI	TCP/IP
Application Layer	Application Layer TELNET, FTP, SMTP, POP3, SNMP, NNTP, DNS,NIS, NFS, HTTP, ...
Presentation Layer	
Session Layer	
Transport Layer	Transport Layer TCP , UDP , ...
Network Layer	Internet Layer IP , ICMP, ARP, RARP, ...
Data Link Layer	Link Layer FDDI, Ethernet, ISDN, X.25,...
Physical Layer	

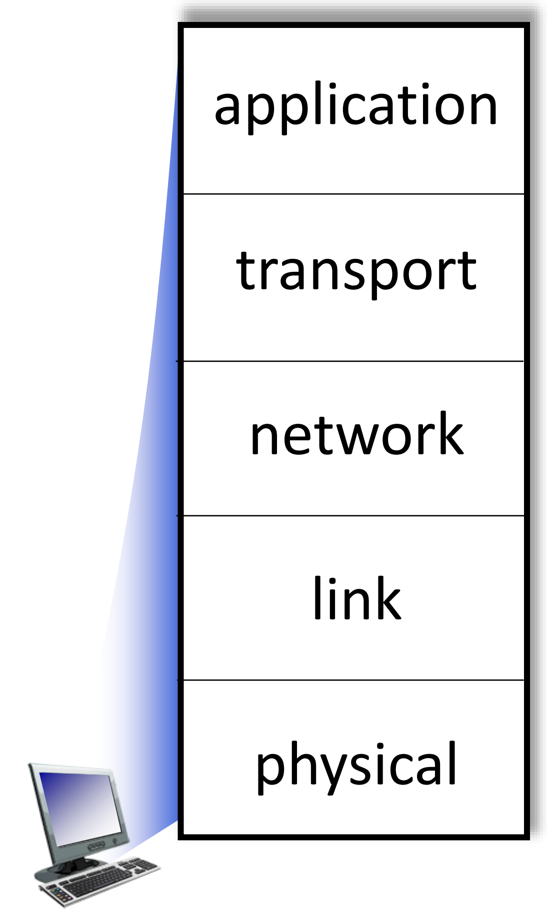
TCP/IP Stack


Network Connections
Host A
Router
Router
Host B
Stack Connections
Application
Application
Peer-to-peer
Transport
Transport
Internet
Internet
Internet
Internet
Link
Link
Link
Link
Ethernet
Fiber, Satellite, etc.
Ethernet

TCP/IP Encapsulation



Services, Layering and Encapsulation



source

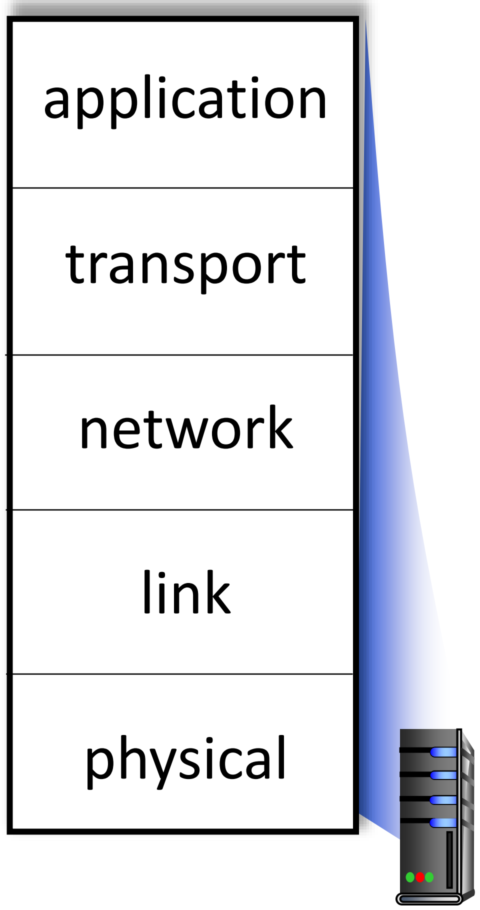
M

Application exchanges messages to implement some application service using *services* of transport layer

H_t M

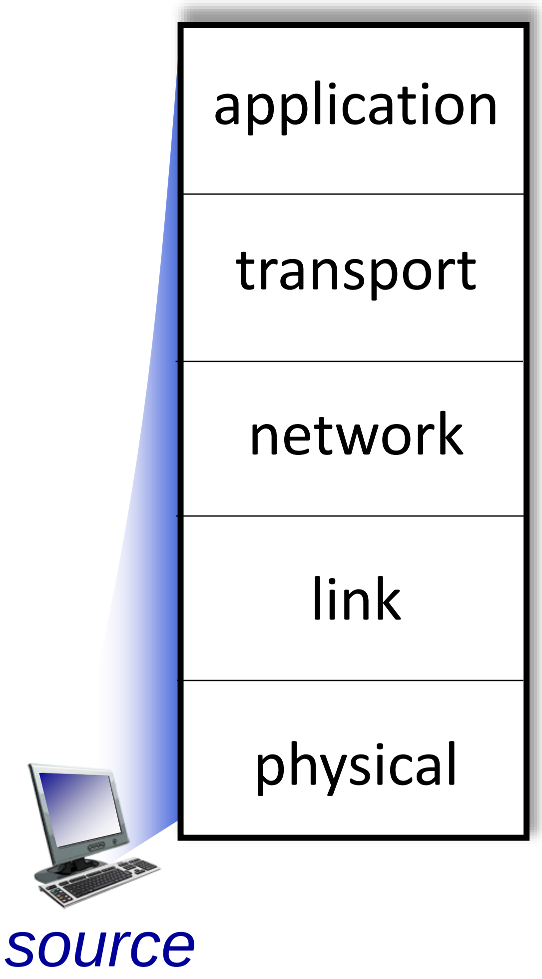
Transport-layer protocol transfers M (e.g., reliably) from one *process* to another, using services of network layer

- transport-layer protocol encapsulates application-layer message, M, with *transport* layer-layer header H_t to create a transport-layer segment
  - H_t used by transport layer protocol to implement its service



destination

Services, Layering and Encapsulation



M

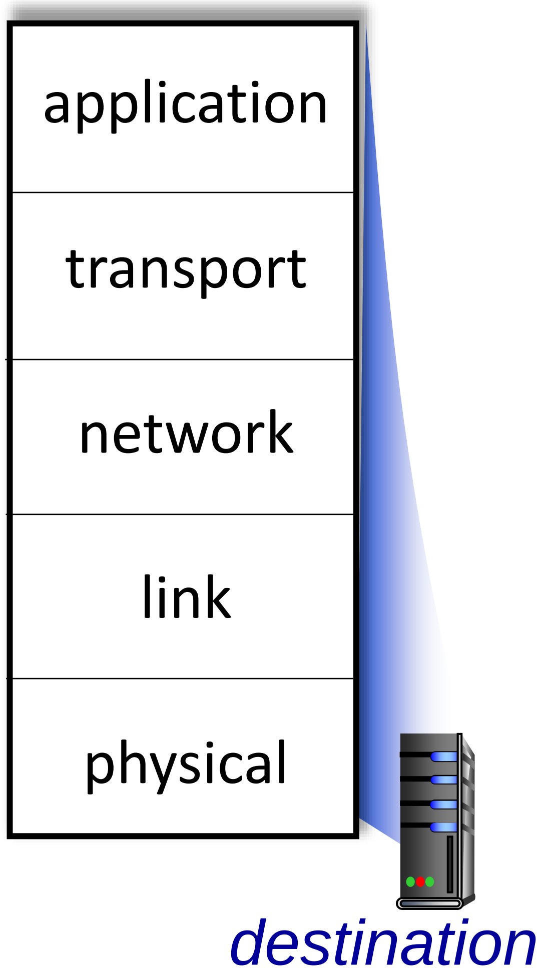
H_t M

Transport-layer protocol transfers M (e.g., reliably) from one *process* to another, using services of network layer

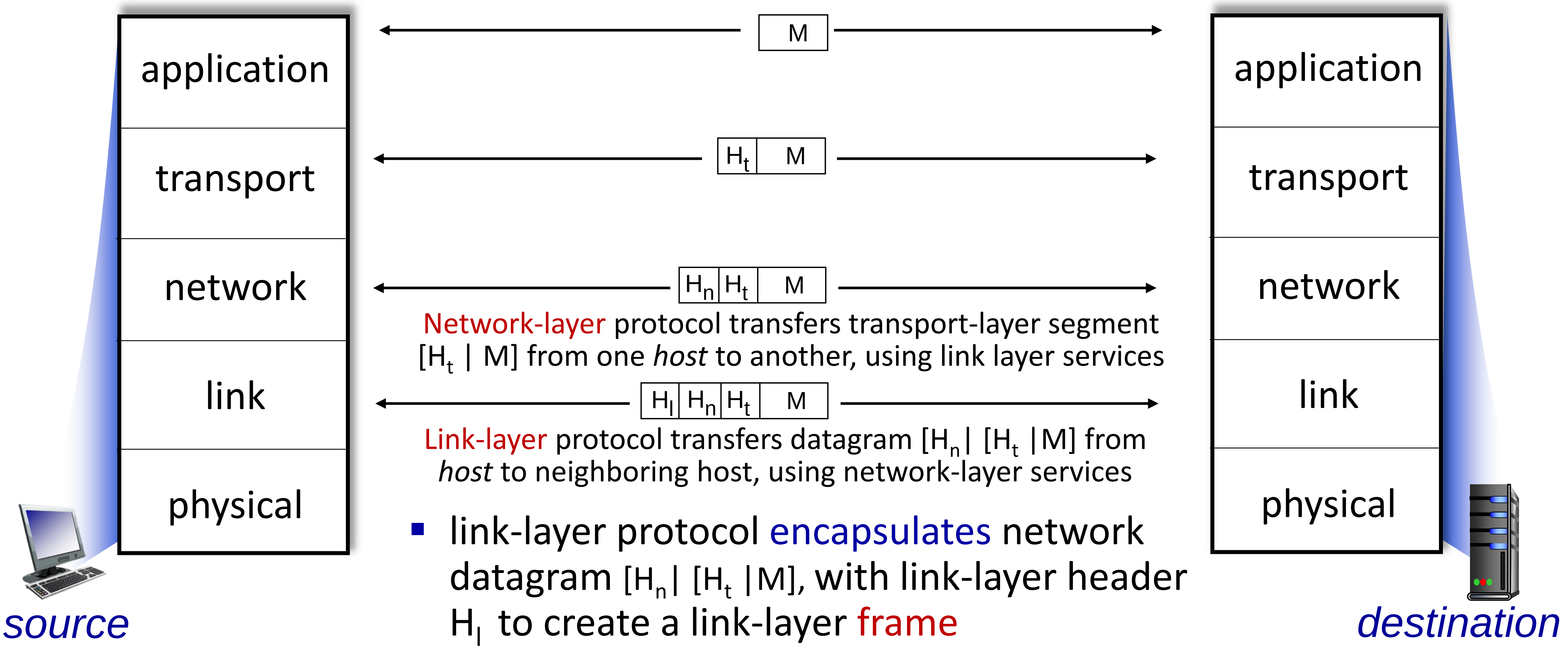
H_n H_t M

Network-layer protocol transfers transport-layer segment [H_t | M] from one *host* to another, using link layer services

- network-layer protocol encapsulates transport-layer segment [H_t | M] with network layer-layer header H_n to create a network-layer datagram
  - H_n used by network layer protocol to implement its service



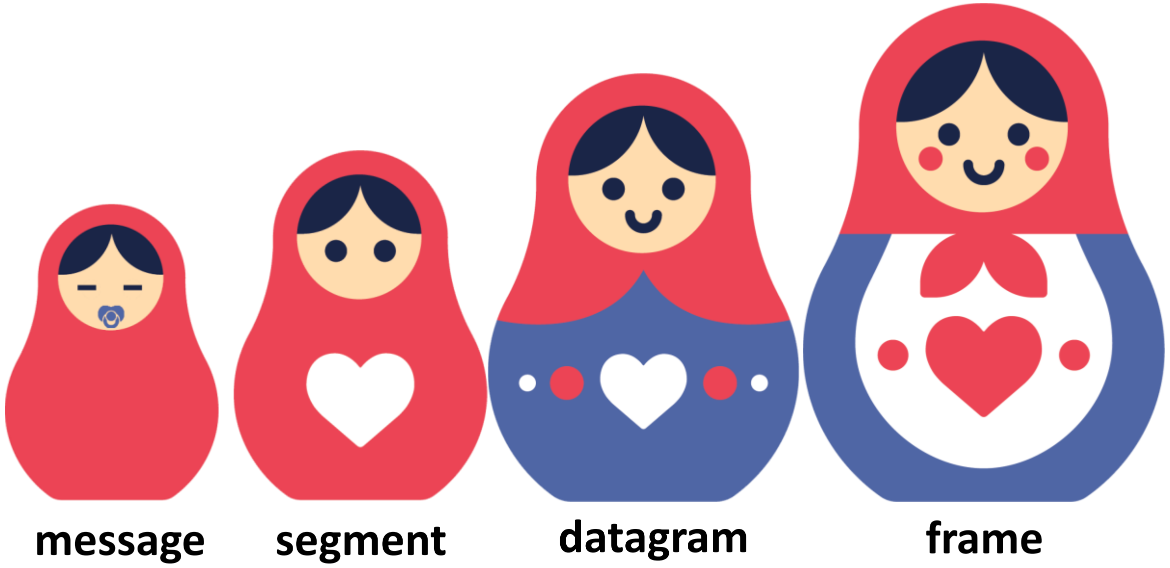
Services, Layering and Encapsulation



- link-layer protocol encapsulates network datagram [H_n| [H_t |M], with link-layer header H_l to create a link-layer frame

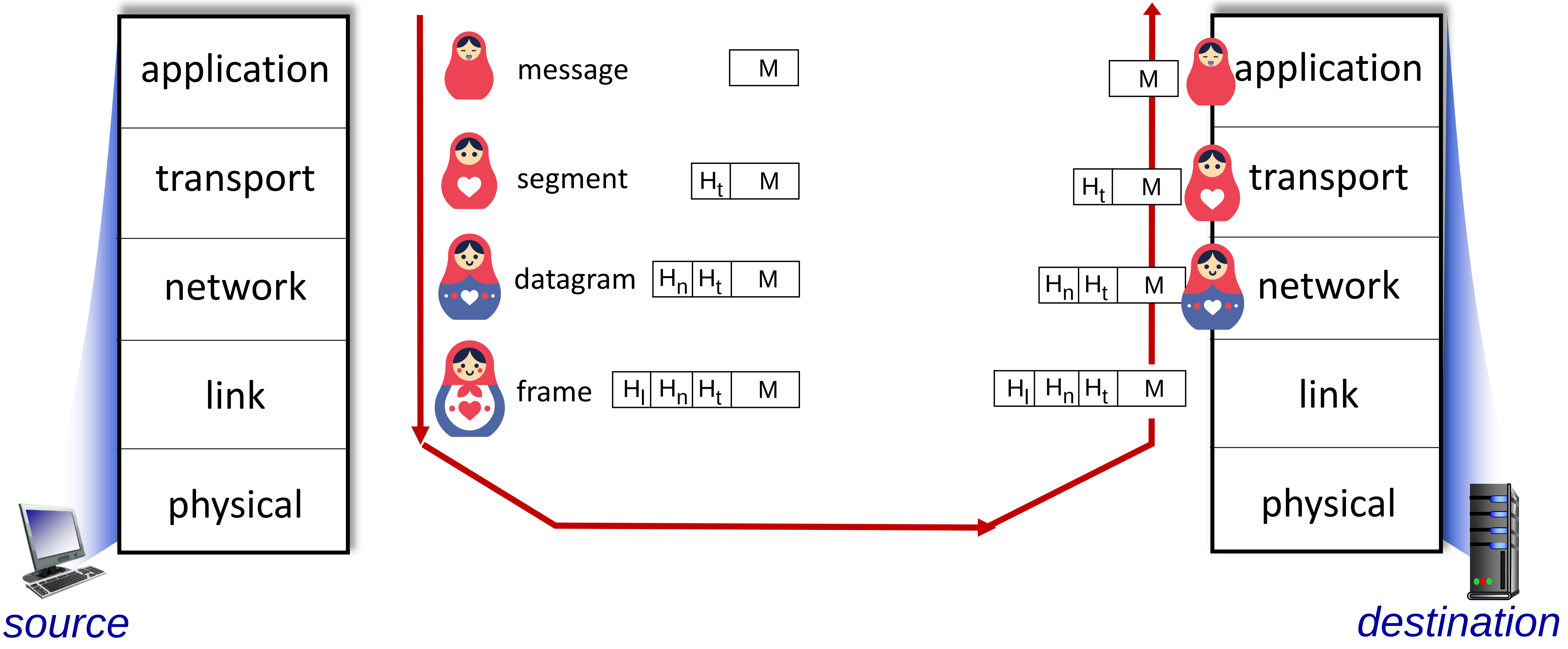
Encapsulation

Matryoshka dolls (stacking dolls)

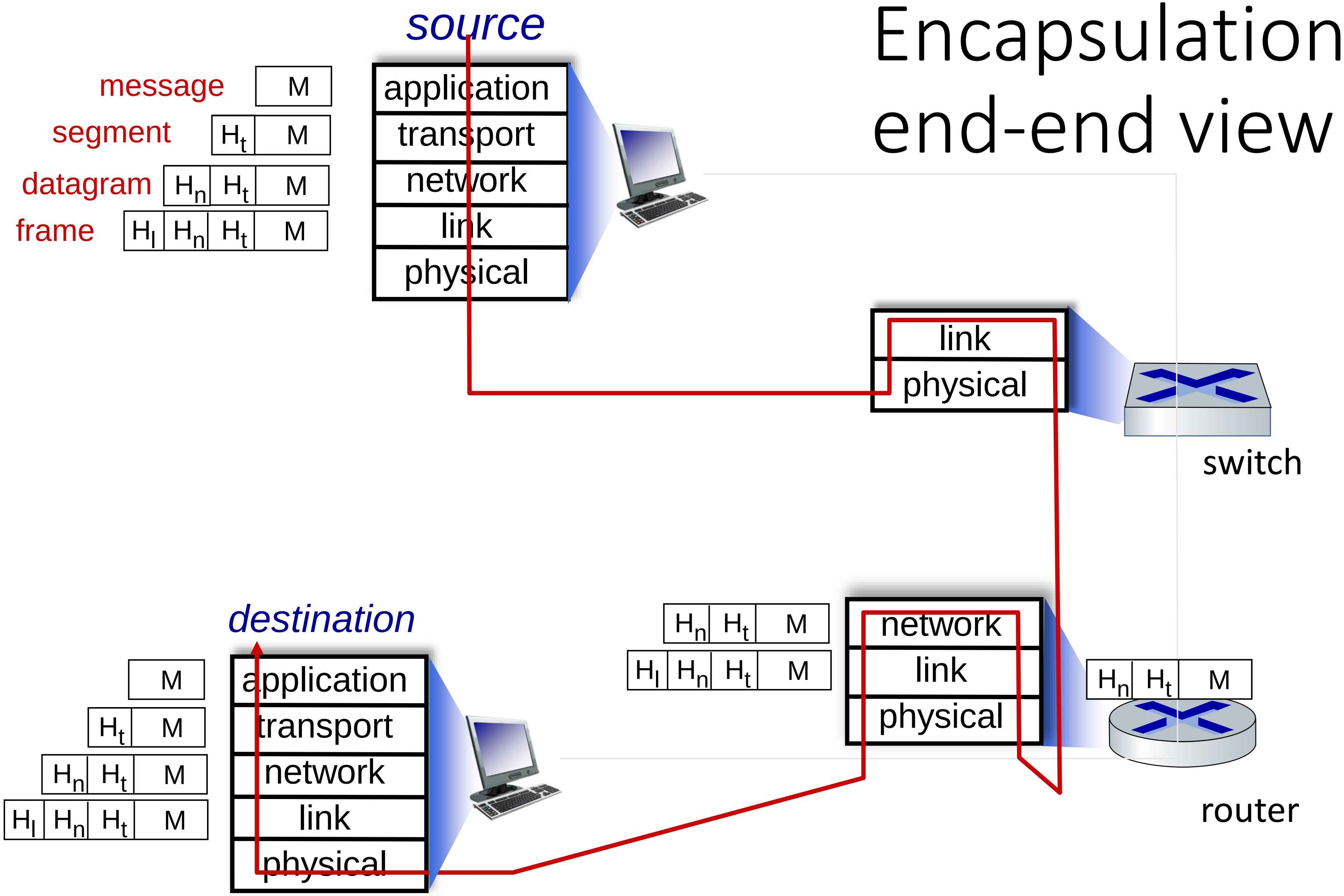


Credit: https://dribbble.com/shots/7182188-Babushka-Boi

Services, Layering and Encapsulation



Encapsulation: an end-end view



TCP/IP Some Protocol

Layer	Protocol
Application	DNS, TFTP, TLS/SSL, FTP, Gopher, HTTP, IMAP, IRC, NNTP, POP3, SIP, SMTP, SMPP, SNMP, SSH, Telnet, Echo, RTP, PNRP, rlogin, ENRP
	Routing protocols like BGP and RIP which run over TCP/UDP, may also be considered part of the Internet Layer.
Transport	TCP, UDP, DCCP, SCTP, IL, RUDP, RSVP
Internet	IP (IPv4, IPv6), ICMP, IGMP, and ICMPv6
	OSPF for IPv4 was initially considered IP layer protocol since it runs per IP-subnet, but has been placed on the Link since RFC 2740.
Link	ARP, RARP, OSPF (IPv4/IPv6), IS-IS, NDP

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm

- learn about protocols by examining popular application-layer protocols and infrastructure
  - HTTP
  - SMTP, IMAP
  - DNS
  - video streaming systems, CDNs
- programming network applications
  - socket API

Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing (e.g., Zoom)
- Internet search
- remote login
- …

Q: *your* favorites?

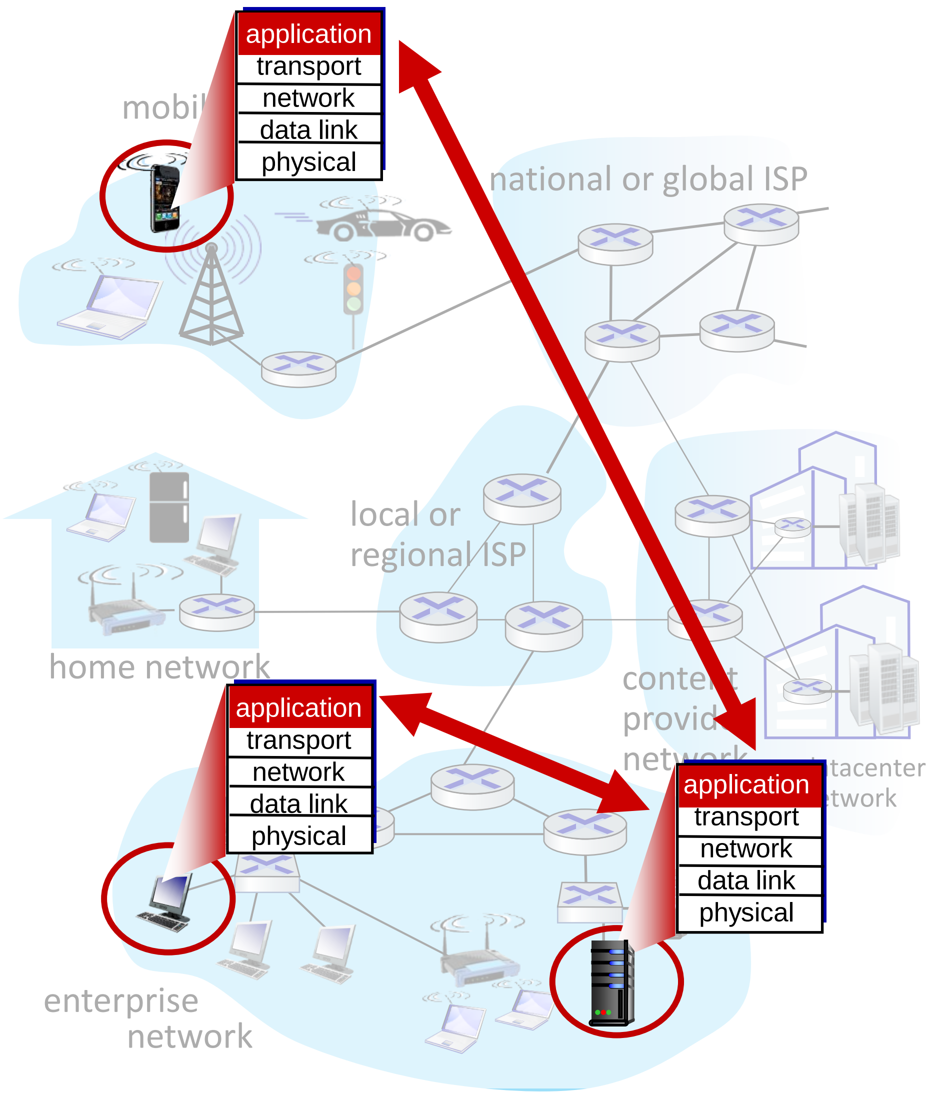
Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



Client-server paradigm

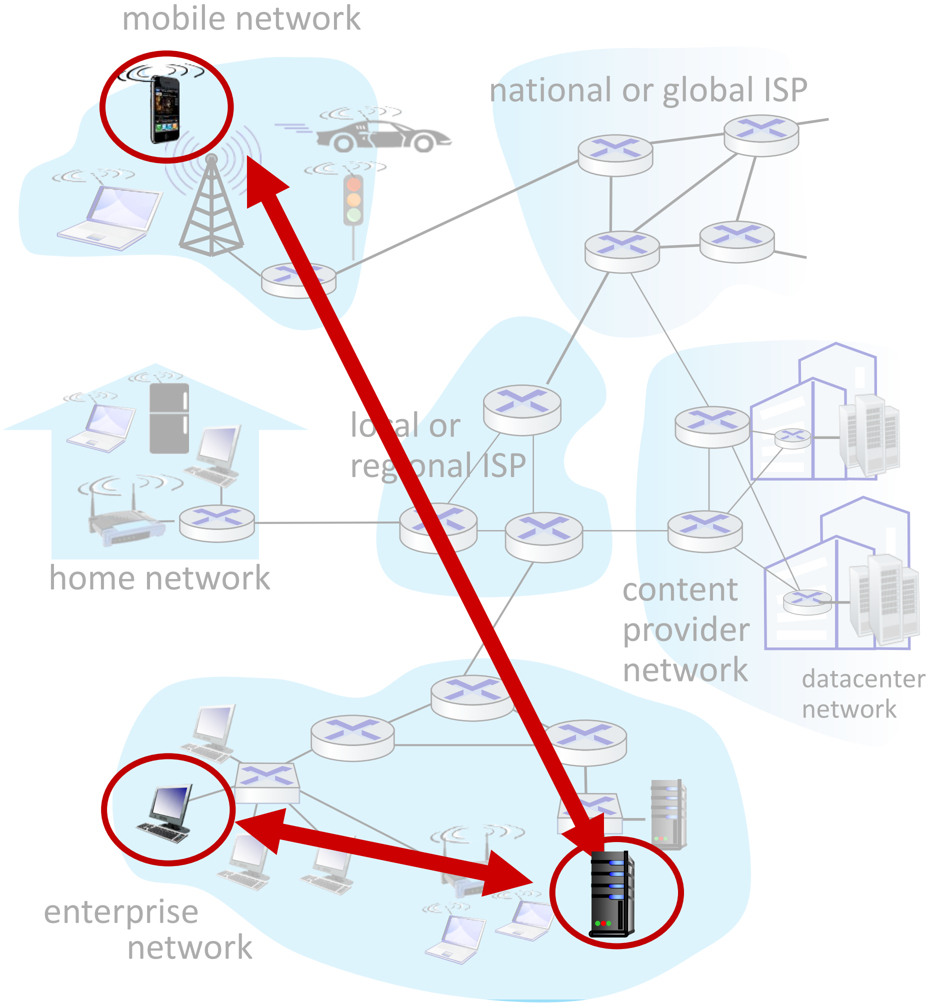
server:

- always-on host
- permanent IP address
- often in data centers, for scaling

clients:

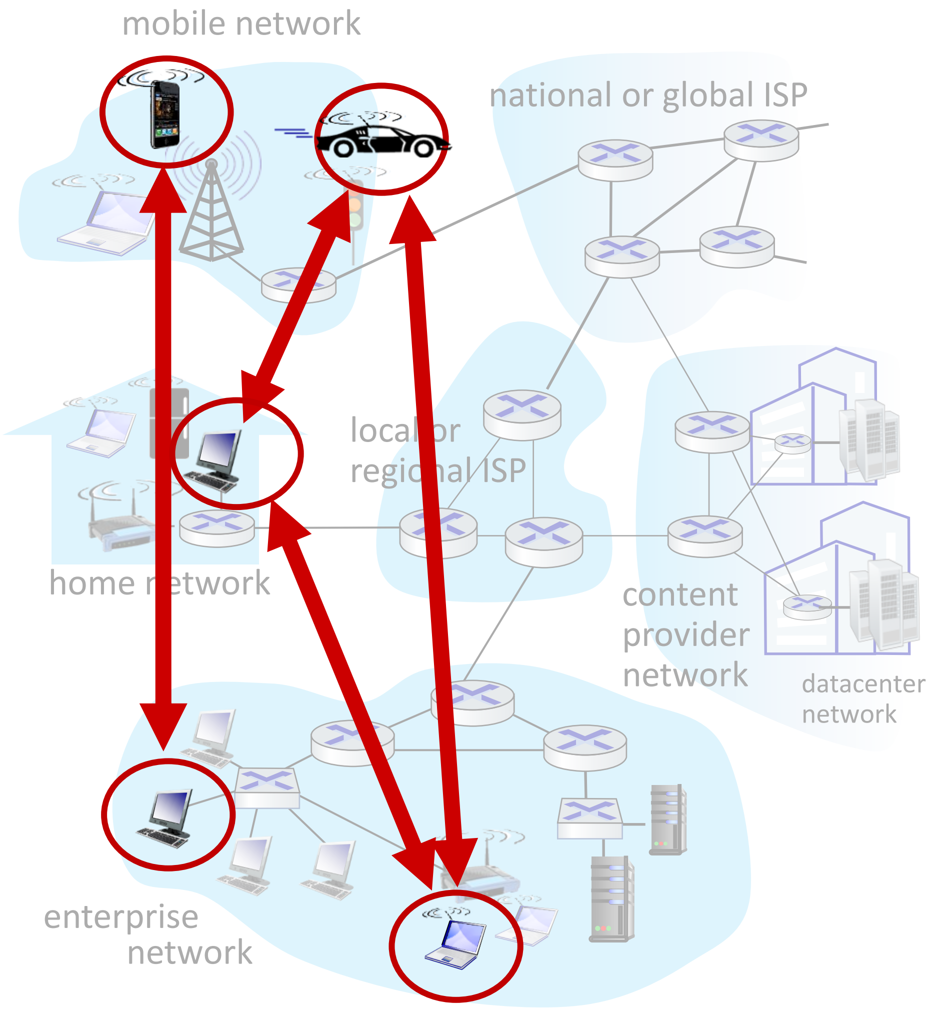
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-peer architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing [BitTorrent]



Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

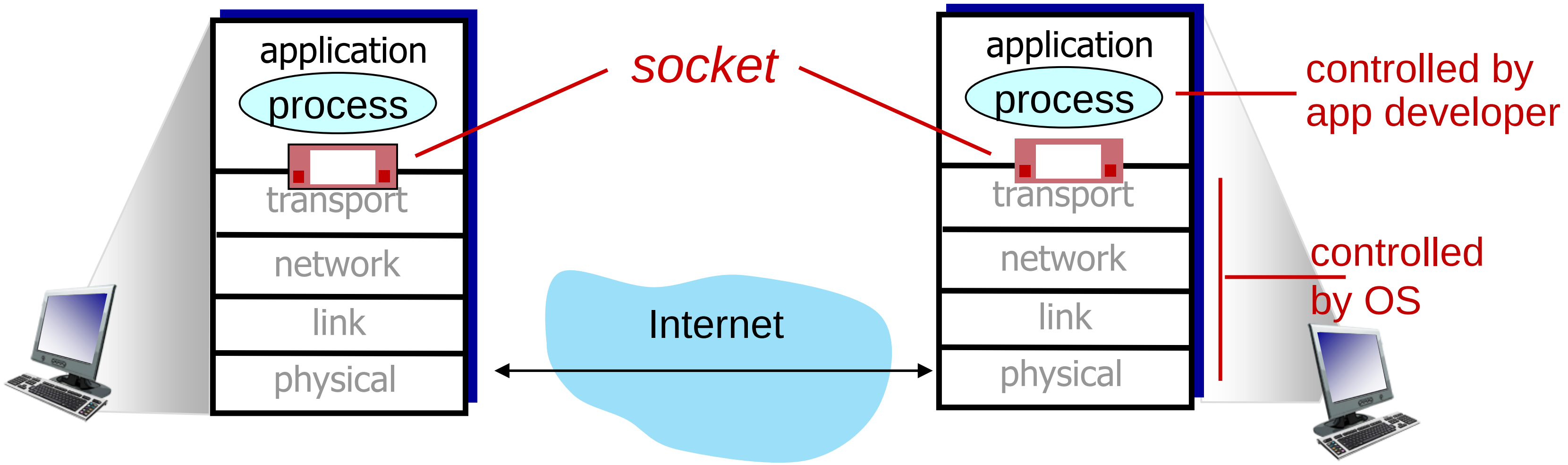
client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

An application-layer protocol defines:

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, …

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *connection-oriented:* setup required between client and server processes
- *does not provide:* timing, minimum throughput guarantee, security

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

Internet applications, and transport protocols

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP [RFC 7230, 9110]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Securing TCP

Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

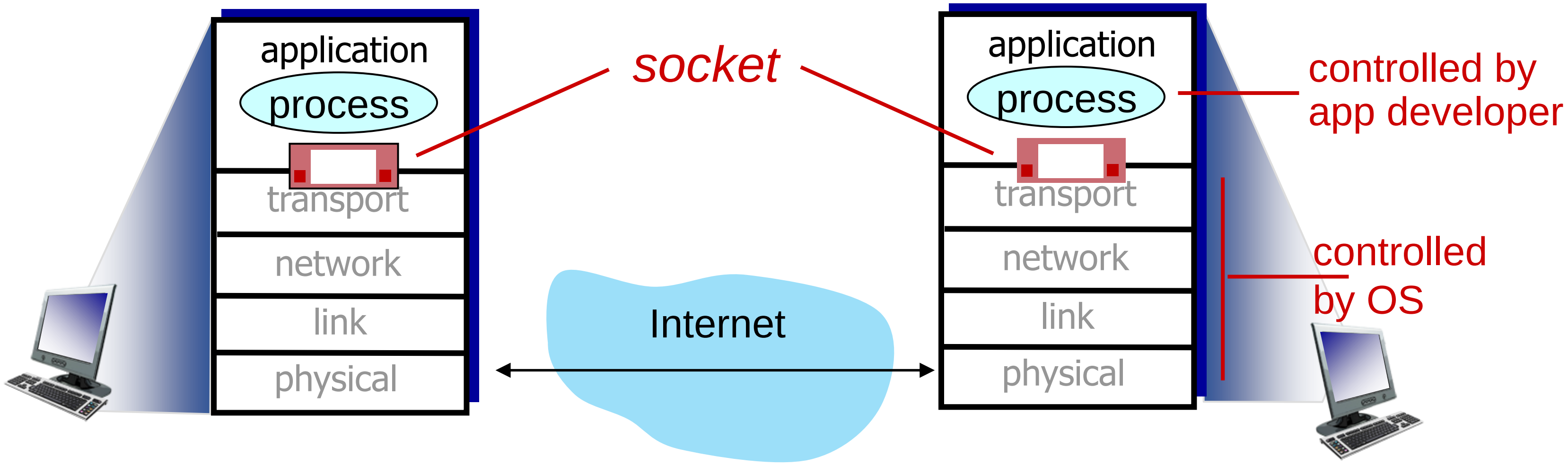
TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn
- cleartext sent into “socket” traverse Internet *encrypted*
- more: Chapter 8

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: ***door between application process and end-end-transport protocol***



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no "connection" between client and server:

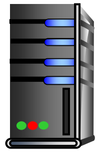
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library → `from socket import *`

create UDP socket → `clientSocket = socket(AF_INET, SOCK_DGRAM)`

get user keyboard input → `message = input('Input lowercase sentence:')`

attach server name, port to message; send into socket → `clientSocket.sendto(message.encode(), (serverName, serverPort))`

read reply data (bytes) from socket → `modifiedMessage, serverAddress = clientSocket.recvfrom(2048)`

print out received string and close socket → `print(modifiedMessage.decode())`, `clientSocket.close()`

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

Note: this code update (2023) to Python 3

Example app: UDP server

Python UDPServer

create UDP socket → `serverSocket = socket(AF_INET, SOCK_DGRAM)`

bind socket to local port number 12000 → `serverSocket.bind(('', serverPort))`

loop forever → `while True:`

Read from UDP socket into message, getting client's address (client IP and port) → `message, clientAddress = serverSocket.recvfrom(2048)`

send upper case string back to this client → `serverSocket.sendto(modifiedMessage.encode(), clientAddress)`

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print('The server is ready to receive')
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                                  clientAddress)
```

Note: this code update (2023) to Python 3

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

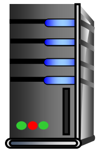
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - client source port # and IP address used to distinguish clients

Application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



server (running on hostid)

client

Server:

create socket, port=**x**, for incoming request:
serverSocket = **socket()**

wait for incoming connection request
connectionSocket = **serverSocket.accept()**

read request from connectionSocket

write reply to connectionSocket

close connectionSocket

TCP connection setup

Client:

create socket, connect to **hostid**, port=**x**
clientSocket = socket()

send request using clientSocket

read reply from clientSocket

close clientSocket

Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Note: this code update (2023) to Python 3

Example app: TCP server

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

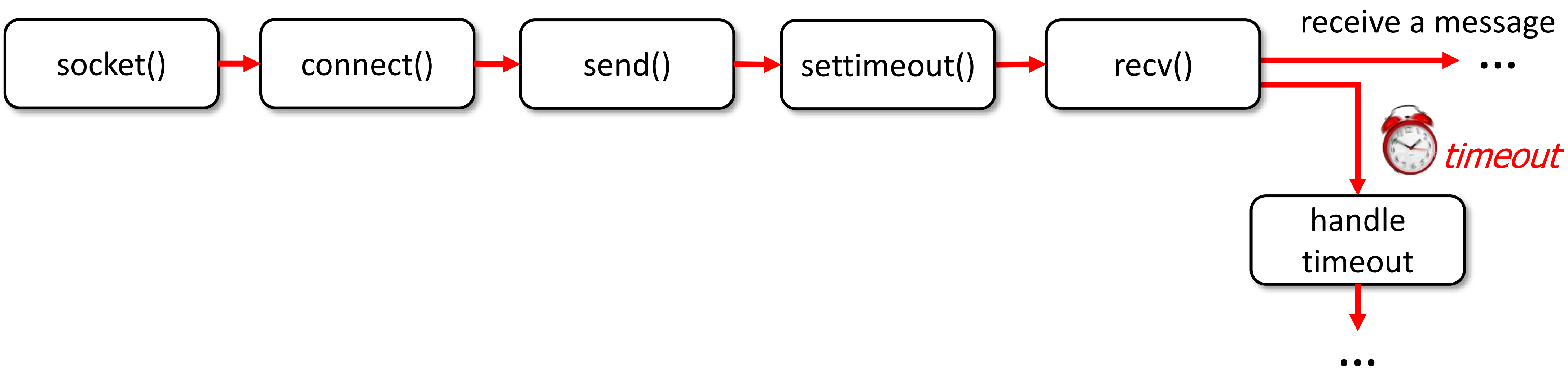
```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                                  encode())
    connectionSocket.close()
```

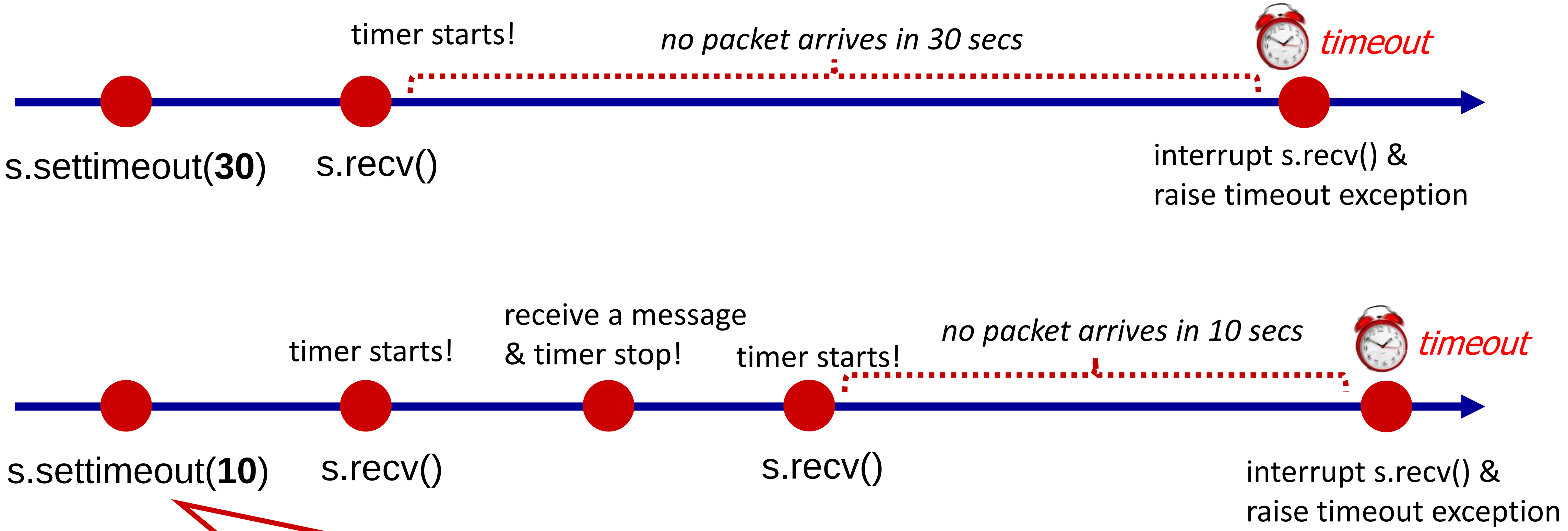
Note: this code update (2023) to Python 3

Socket programming: waiting for multiple events

- sometimes a program must wait for one of several events to happen, e.g.,:
  - wait for either (i) a reply from another end of the socket, or (ii) timeout: timer
  - wait for replies from several different open sockets: select(), multithreading
- timeouts are used extensively in networking
- using timeouts with Python socket:



How Python socket.settimeout() works?



Set a timeout on all future socket operations of that specific socket!

Python try-except block

Execute a block of code, and handle “exceptions” that may occur when executing that block of code

try:

<do something>

Executing this try code block may cause exception(s) to catch. If an exception is raised, execution jumps from jumps directly into except code block

except <exception>:

<handle the exception>

this except code block is only executed *if an <exception> occurred* in the try code block (note: except block is *required* with a try block)

Socket programming: socket timeouts

Toy Example:

- A shepherd boy tends his master's sheep.
- If he sees a wolf, he can send a message to villagers for help using a TCP socket.
- The boy found it fun to connect to the server without sending any messages. But the villagers don't think so.
- And they decided that if the boy connects to the server and doesn't send the wolf location **within 10 seconds for three times**, they will **stop listening** to him forever and ever.

Python TCPServer (Villagers)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
counter = 0
while counter < 3:
    connectionSocket, addr = serverSocket.accept()
    connectionSocket.settimeout(10)
    try:
        wolf_location = connectionSocket.recv(1024).decode()
        send_hunter(wolf_location) # a villager function
        connectionSocket.send('hunter sent')
    except timeout:
        counter += 1
    connectionSocket.close()
```

set a 10-seconds timeout on all future socket operations

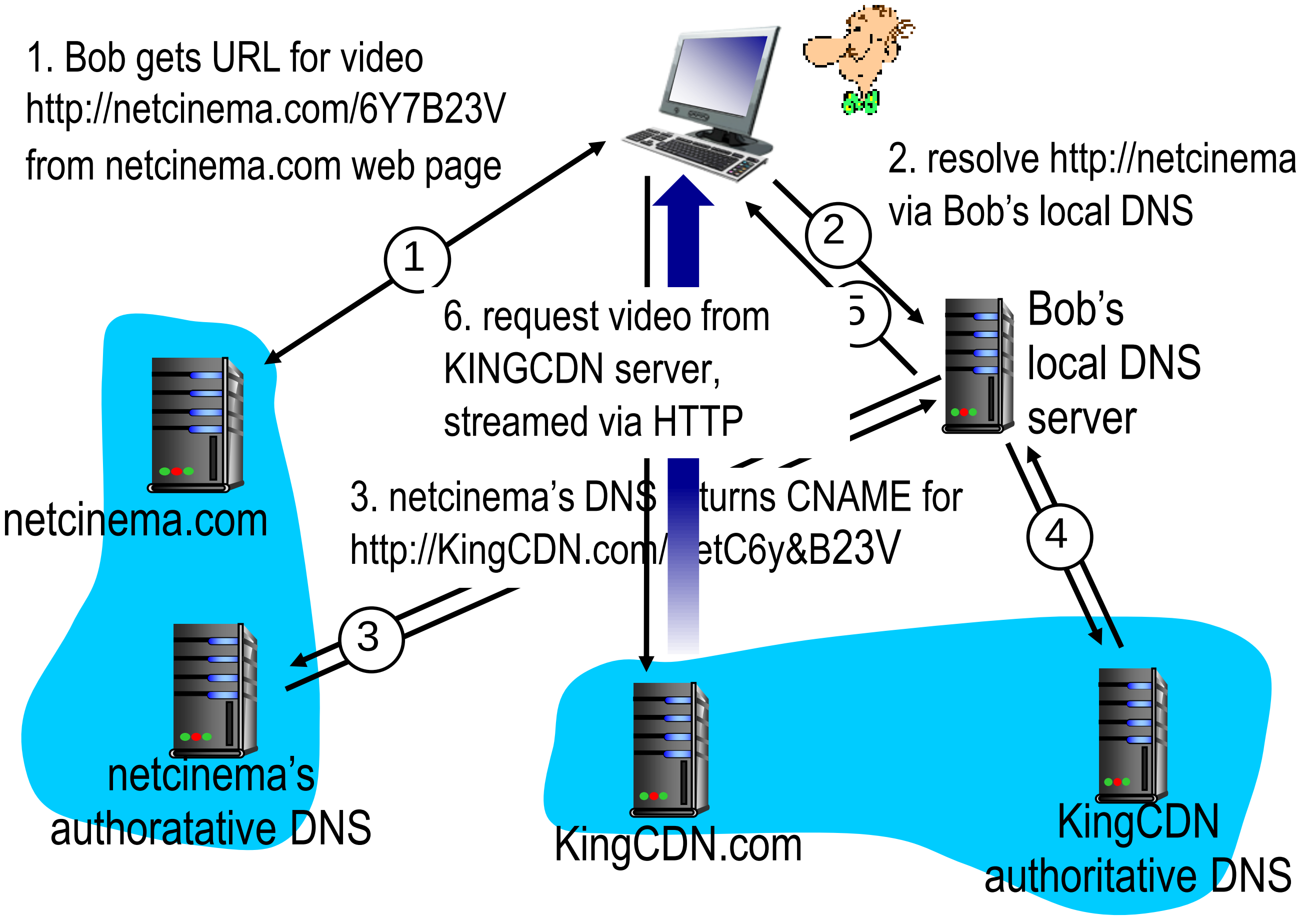
timer starts when recv() is called and will raise timeout exception if there is no message within 10 seconds.

catch socket timeout exception

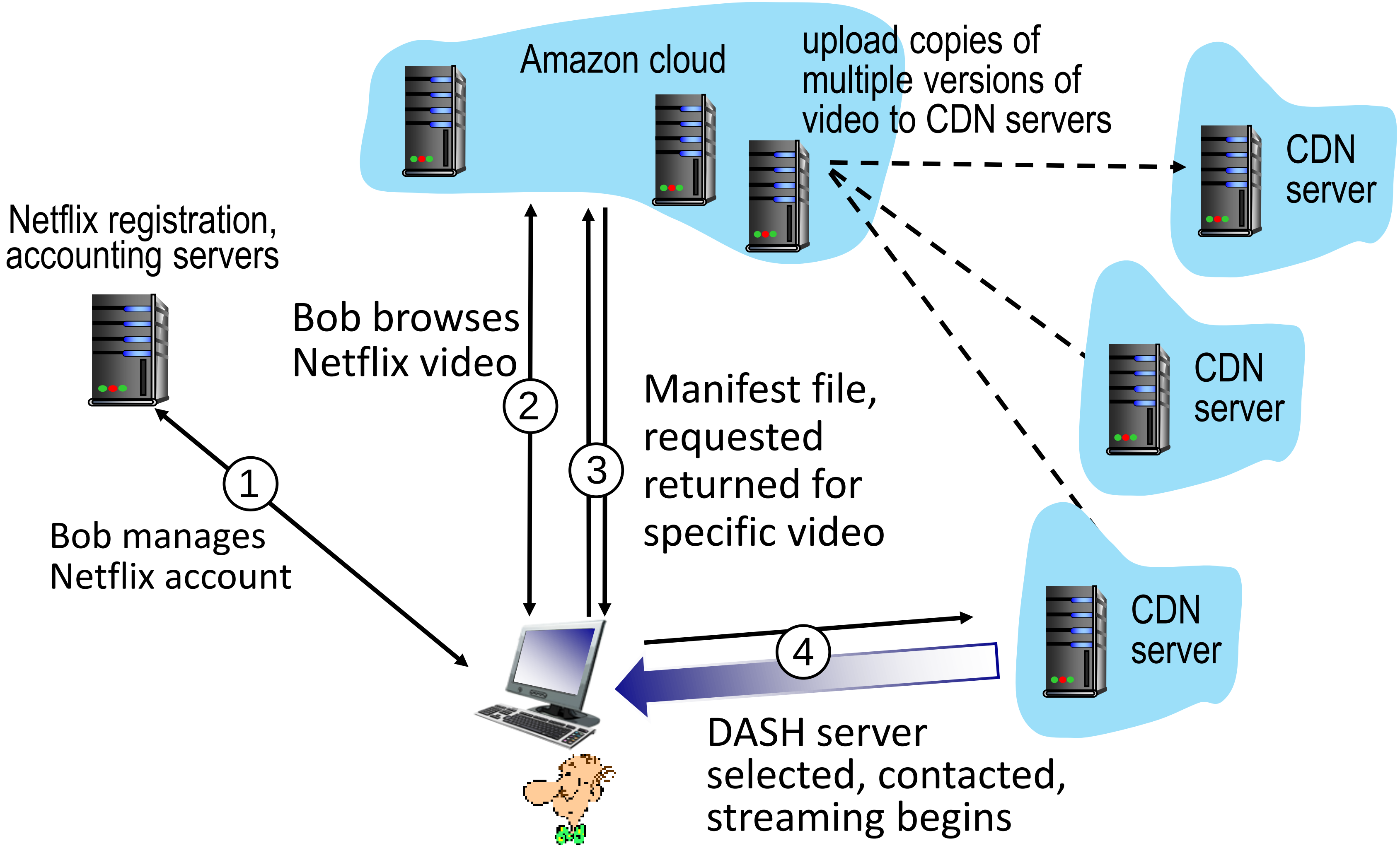
CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V

- video stored in CDN at http://KingCDN.com/NetC6y&B23V



Case study: Netflix



Thanks

Q & A