



# Introduction to Computer Networks (SWE3022)

Jaehoon (Paul) Jeong

Department of Computer Science and Engineering

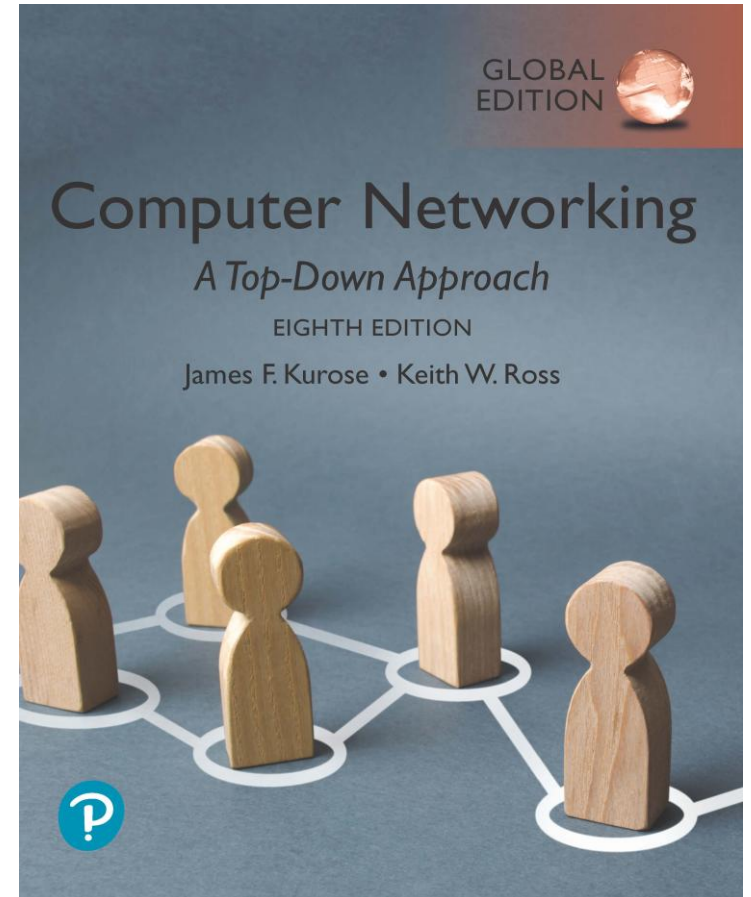
Sungkyunkwan University

Email: [pauljeong@skku.edu](mailto:pauljeong@skku.edu)

Note: The slides are adapted from the slides of Pearson Education Ltd.

# Chapter 2.

## Application Layer: Part 2



### *Computer Networking: A Top-Down Approach*

8<sup>th</sup> Edition, Global Edition

Jim Kurose, Keith Ross

Copyright © 2022 Pearson Education  
Ltd

# 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



# DNS: Domain Name System

*people:* many identifiers:

- SSN (Social Security Number), name, passport #

*Internet hosts, routers:*

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., cs.umass.edu and www.skku.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

*Domain Name System (DNS):*

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol*: hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, *implemented as application-layer protocol*
  - complexity at network’s “edge”

# DNS: services, structure

## DNS services

- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one DNS name

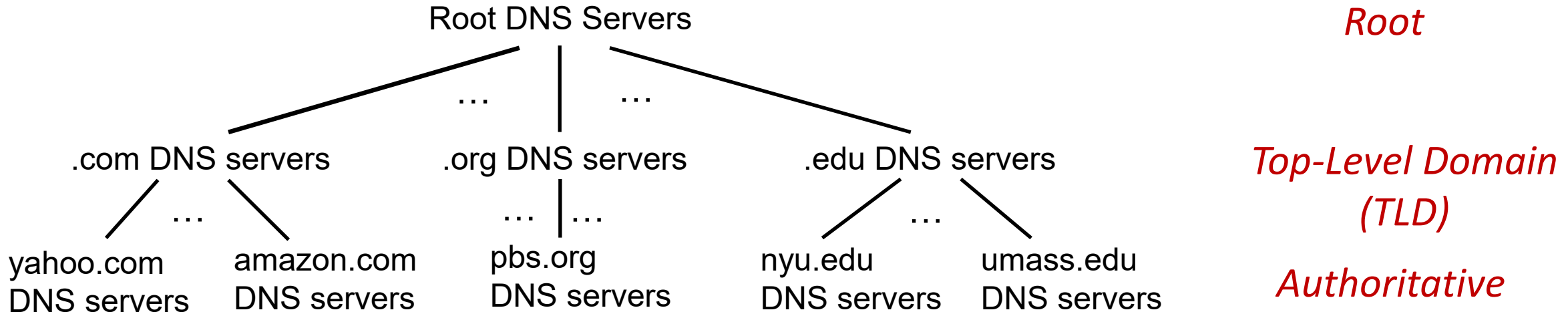
## *Q: Why not centralize DNS?*

- single point of failure
- traffic volume
- distant centralized database
- maintenance

## *A: doesn't scale!*

- Comcast DNS servers alone: 600B DNS queries per day

# DNS: a distributed, hierarchical database



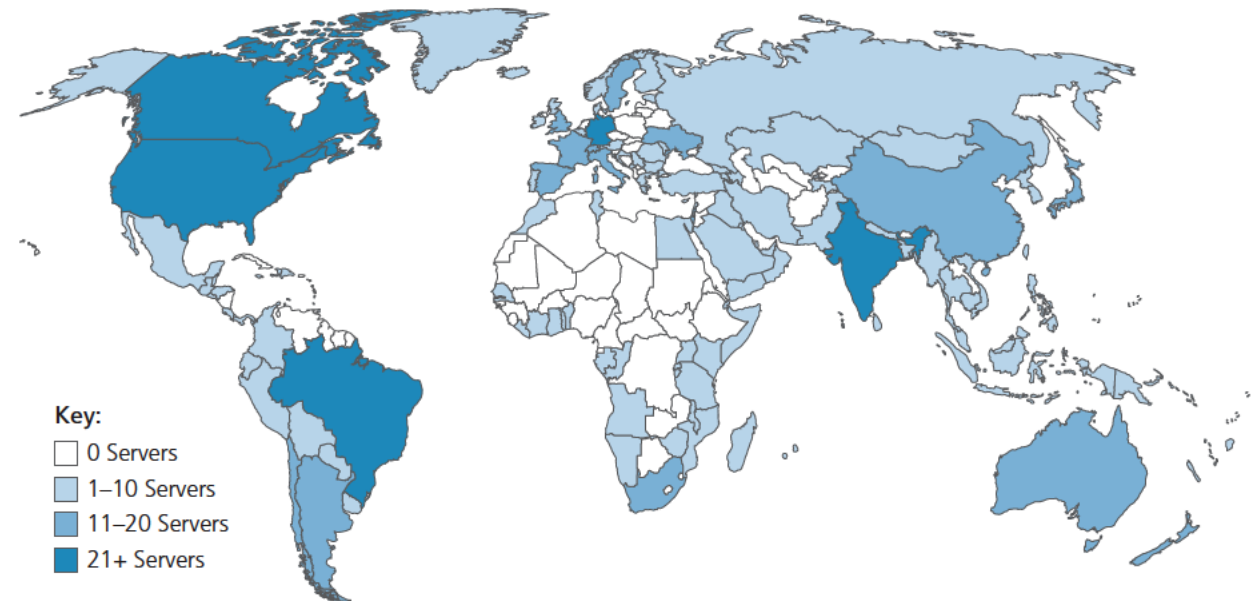
Client wants IP address for `www.amazon.com`; 1<sup>st</sup> approximation:

- client queries root server to find `.com` DNS server
- client queries `.com` DNS server to get `amazon.com` DNS server
- client queries `amazon.com` DNS server to get IP address for `www.amazon.com`

# DNS: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name “servers”:  
worldwide each “server” is replicated  
many times (~200 servers in US, ~1000  
servers over the world)



# TLD: authoritative servers

## Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp, .kr
- Network Solutions: authoritative registry for .com, .net TLD
- Education Organizations: .edu TLD

## Authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider



# Local DNS name servers

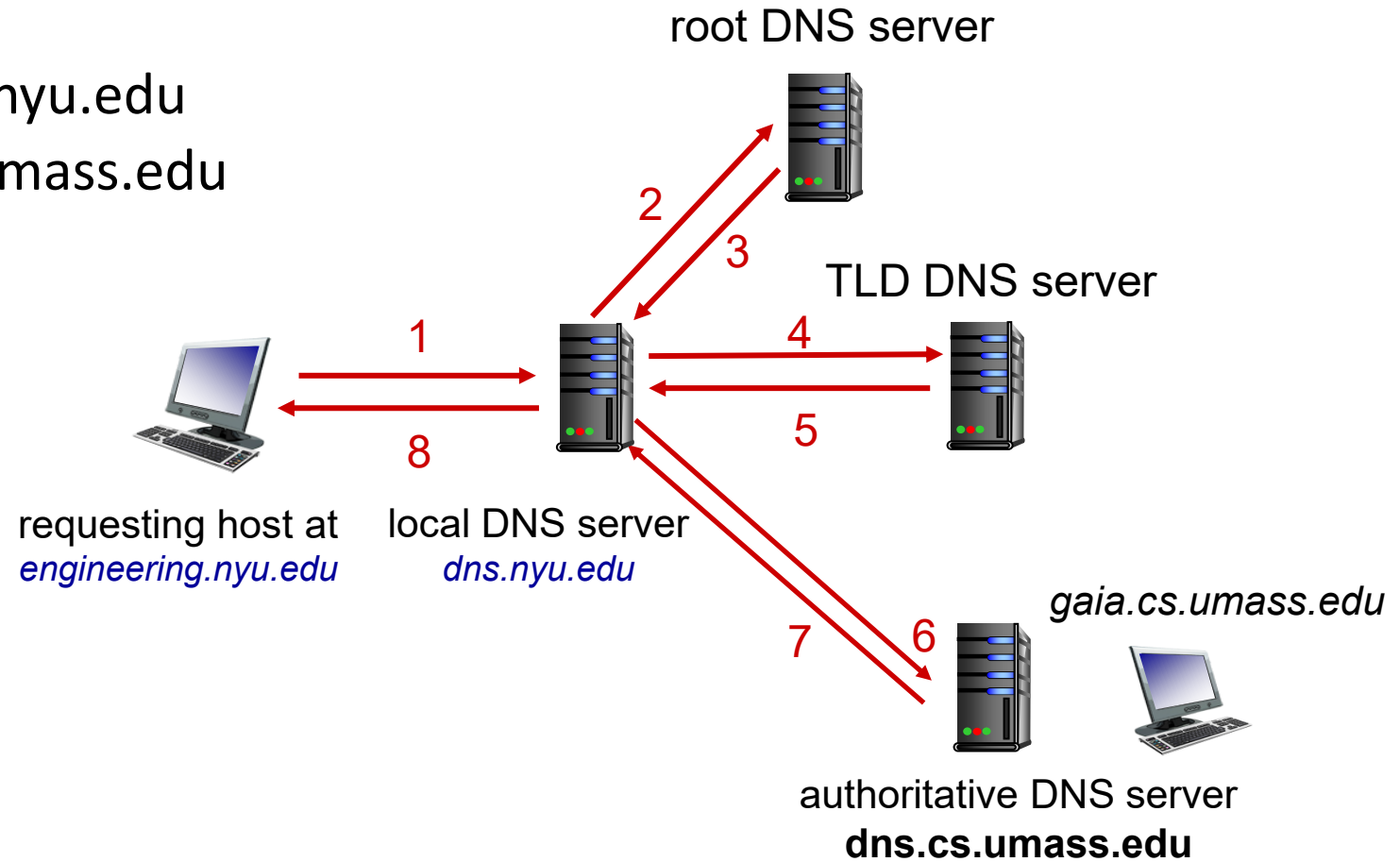
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

# DNS name resolution: iterated query

**Example:** host at `engineering.nyu.edu` wants IP address for `gaia.cs.umass.edu`

## Iterated query:

- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”

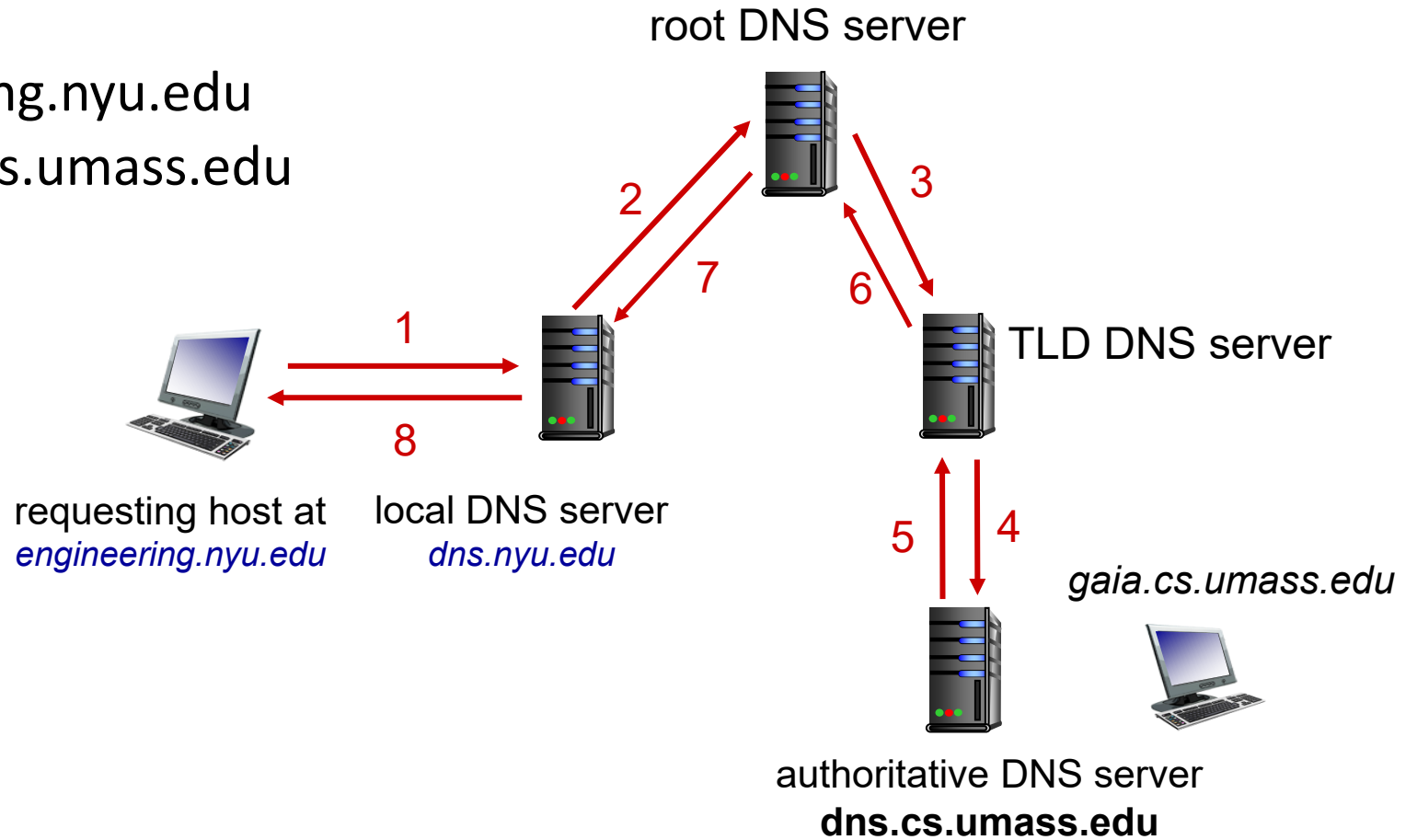


# DNS name resolution: recursive query

**Example:** host at `engineering.nyu.edu` wants IP address for `gaia.cs.umass.edu`

## Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



# Caching, Updating DNS Records

- once (any) name server learns mapping, it *cached* mapping
  - cache entries timeout (disappear) after some time (TTL: Time-To-Live)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best-effort name-to-address translation!)
  - if name host changes IP address, it may not be known Internet-wide until all TTLs expire!
- update/notify mechanisms proposed IETF standard
  - RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE)
    - <https://datatracker.ietf.org/doc/html/rfc2136>

# DNS records

**DNS:** distributed database storing resource records (**RR**)

RR format: (name, value, type, ttl)

## **type=A** (IP Address)

- name is hostname
- value is IP address

## **type=NS** (Name Server)

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

## **type=CNAME** (Canonical Name)

- name is alias name for some “canonical” (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

## **type=MX** (Mailserver's Name)

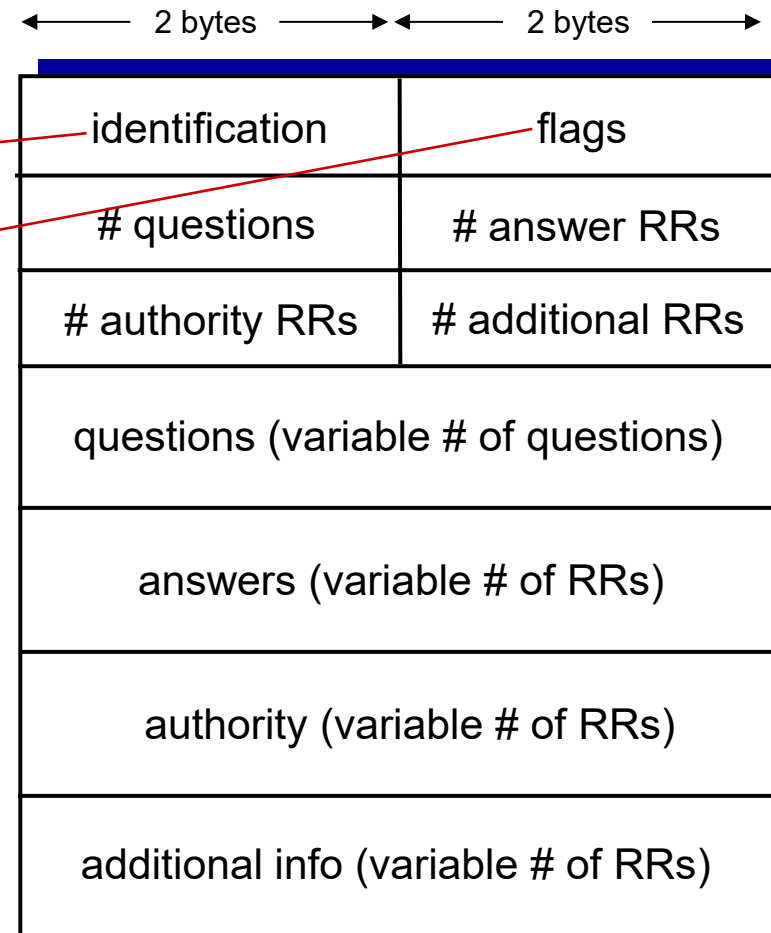
- value is name of mailserver associated with name

# DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:

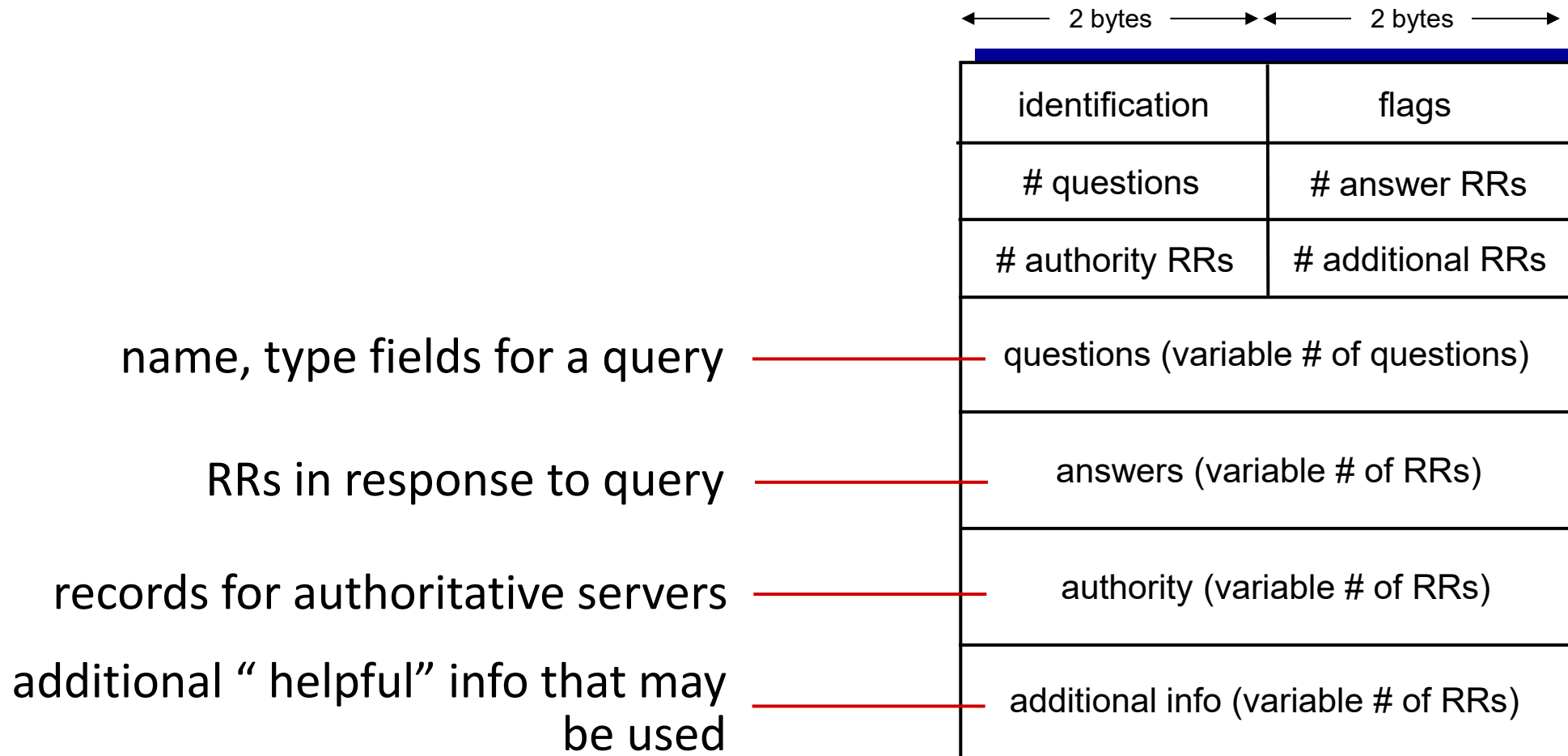
message header:

- **identification**: 16 bit # for query, reply to query uses same #
- **flags**:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



# DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:



# Inserting records into DNS

Example: new startup “Network Utopia”

- register name networkutopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:  
`(networkutopia.com, dns1.networkutopia.com, NS)`  
`(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server locally with IP address `212.212.212.1`
  - type A record for `www.networkutopia.com`
  - type MX record for `networkutopia.com`



# DNS security

## DDoS attacks

- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous

## Redirect attacks

- man-in-middle
  - intercept DNS queries
- DNS poisoning
  - send bogus replies to DNS server, which caches

DNSSEC  
[RFC 4033]

## Exploit DNS for DDoS

- send queries with spoofed source address: target IP
- requires amplification

- RFC 4033: DNS Security Introduction and Requirements (DNSSEC)
  - <https://datatracker.ietf.org/doc/html/rfc4033>

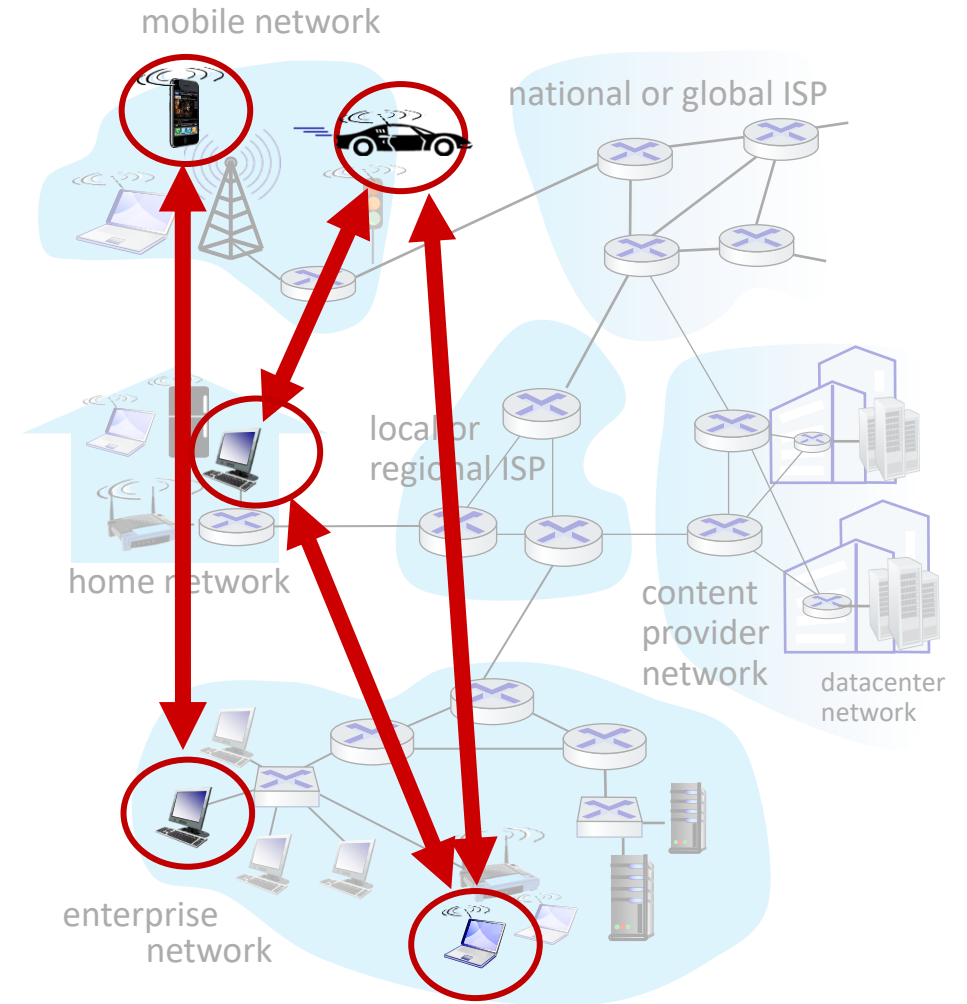
# Application Layer: Overview

- 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



# Peer-to-peer (P2P) architecture

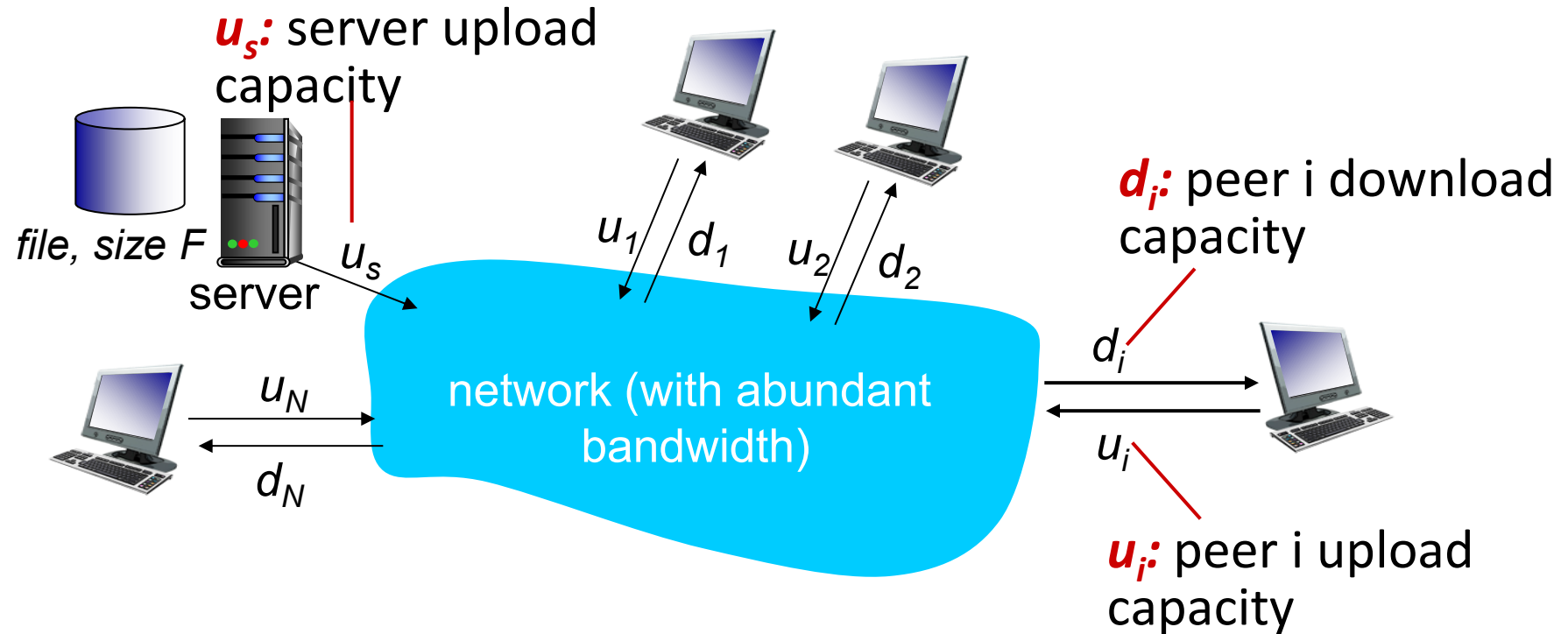
- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



# File distribution: client-server vs P2P

Q: how much time to distribute file (size  $F$ ) from one server to  $N$  peers?

- peer upload/download capacity is limited resource



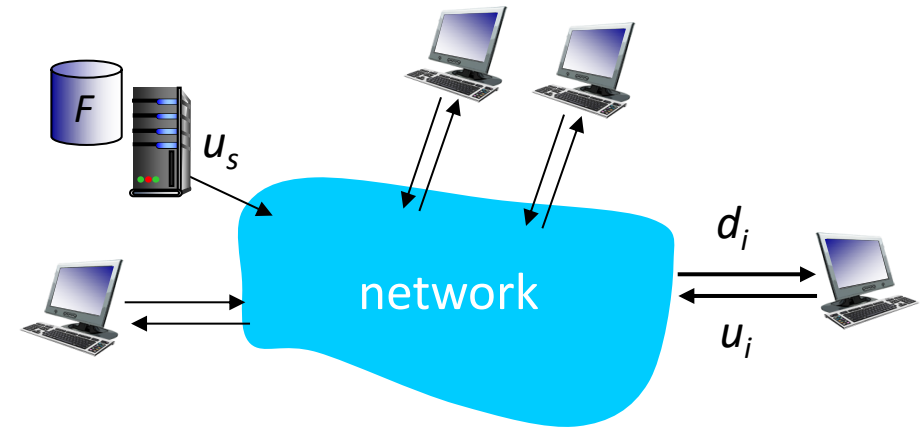
# File distribution time: client-server

- **server transmission:** must sequentially send (upload)  $N$  file copies:

- time to send one copy:  $F/u_s$
- time to send  $N$  copies:  $NF/u_s$

- **client:** each client must download file copy

- $d_{min} = \min \text{ client download rate} = \min\{d_1, d_2, \dots, d_N\}$
- min client download time:  $F/d_{min}$



A lower bound on the minimum distribution time for the **client-server architecture**

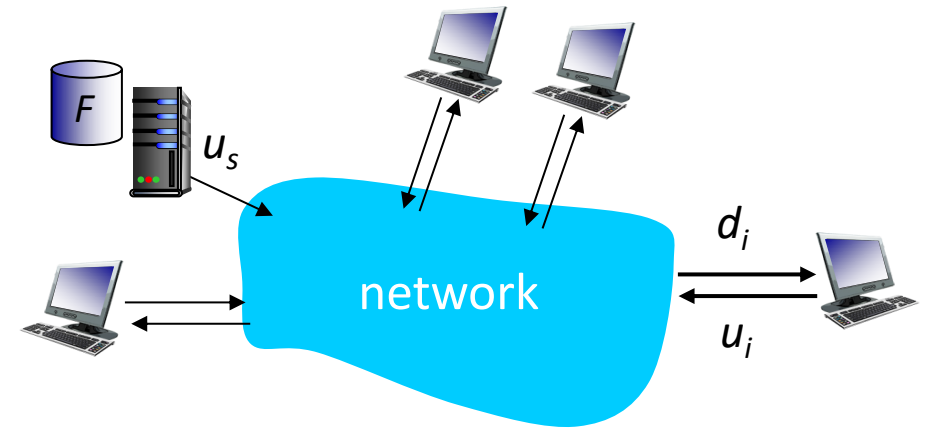
*time to distribute  $F$  to  $N$  clients using client-server approach*

$$D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}$$

increases linearly in  $N$

# File distribution time: P2P

- **server transmission:** must upload at least one copy:
  - time to send one copy:  $F/u_s$
- **client:** each client must download file copy:
  - min client download time:  $F/d_{min}$
- **clients:** as aggregate, the system of server and  $N$  clients uploads  $NF$  bits:
  - max upload rate (limiting max download rate) is  $u_s + \sum u_i$



A lower bound on the minimum distribution time for the **P2P architecture**

*time to distribute  $F$  to  $N$  clients using P2P approach*

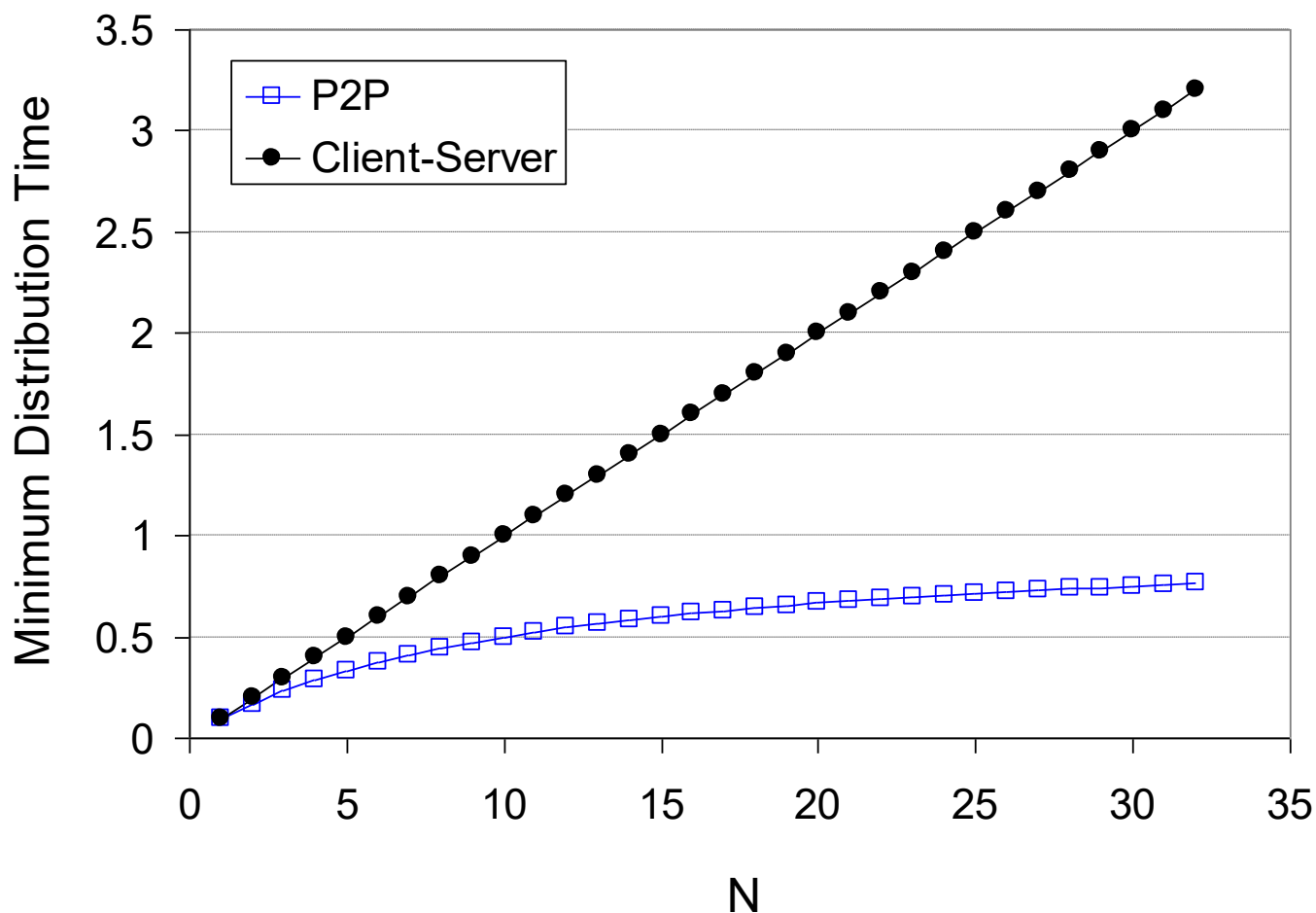
$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

increases linearly in  $N$  ...  
... but so does this, as each peer brings service capacity

# Client-server vs. P2P: example

client upload rate =  $u$ ,  $F/u = 1$  hour,  $u_s = 10u$ ,  $d_{min} \geq u_s$

Note: Peer download rates (i.e.,  $d_{min}$ ) are set large enough so as not to have an effect on minimum distribution time in P2P architecture.

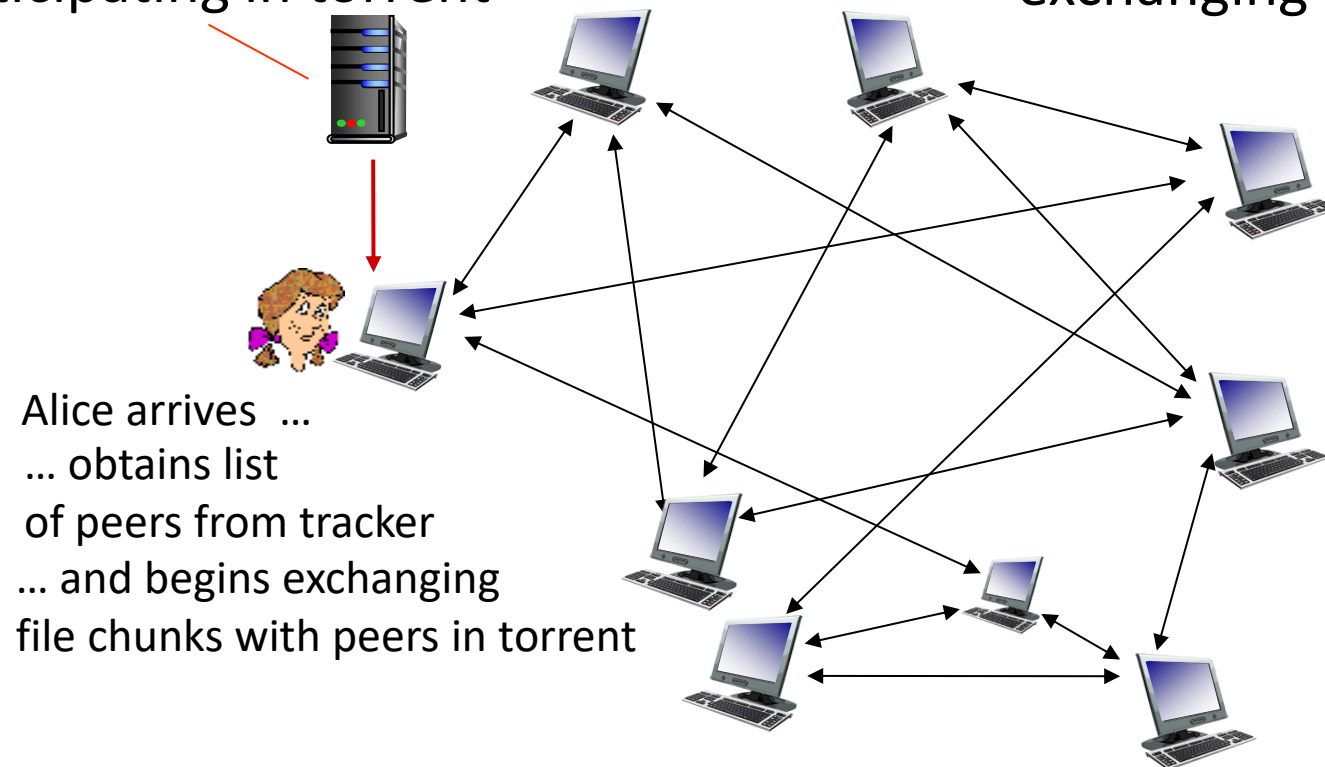


# P2P file distribution: BitTorrent

- BitTorrent is a popular P2P protocol for file distribution.
- file is divided into 256Kb chunks.
- peers in torrent send/receive file chunks.

*tracker*: tracks peers  
participating in torrent

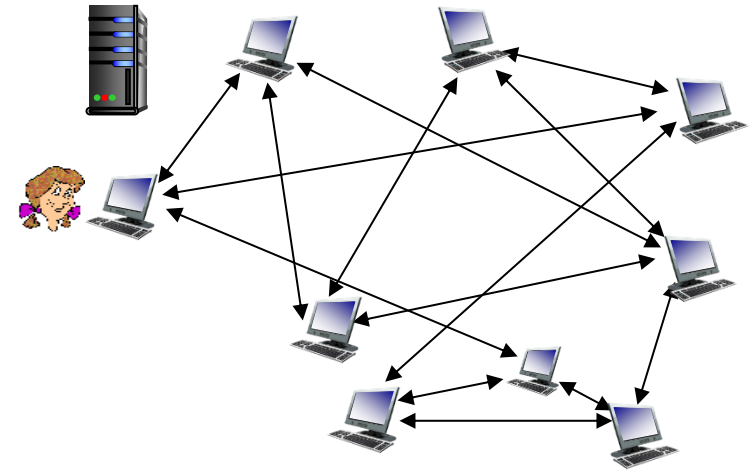
*torrent*: group of peers  
exchanging chunks of a file





# P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn*: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent



# BitTorrent: requesting, sending file chunks

## Requesting chunks: equalizing the number of chunks in torrent

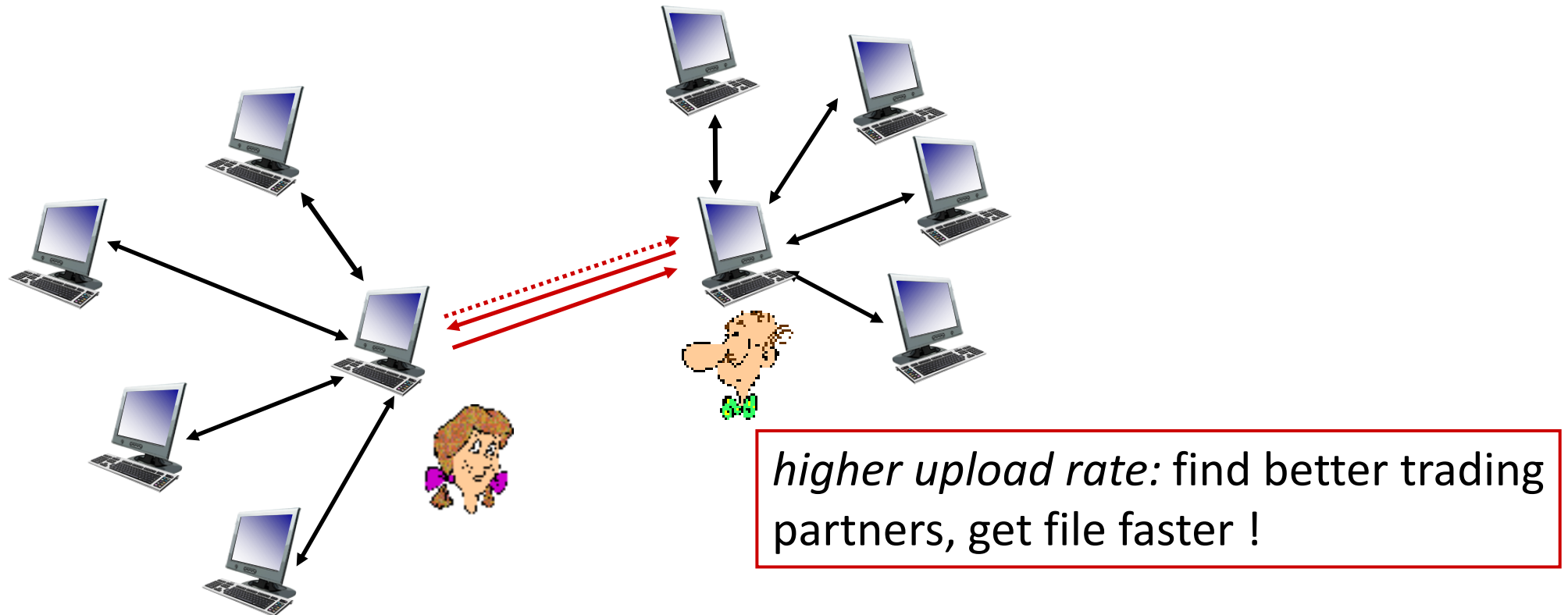
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, “rarest first”
- “Rarest first” is to request the rarest chunks first from her neighbors for quick redistribution of the rare chunks

## Sending chunks: tit-for-tat (trading incentive mechanism)

- Alice sends chunks to those four peers currently sending her chunks *at highest rate* (as clever trading algorithm)
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer as a *new trading partner*, starts sending it chunks
  - “optimistically unchoke” this peer
  - newly chosen peer may join top 4

# BitTorrent: tit-for-tat

- (1) Alice “optimistically unchokes” Bob
- (2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
- (3) Bob becomes one of Alice’s top-four providers



# 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



# Video Streaming and CDNs: context

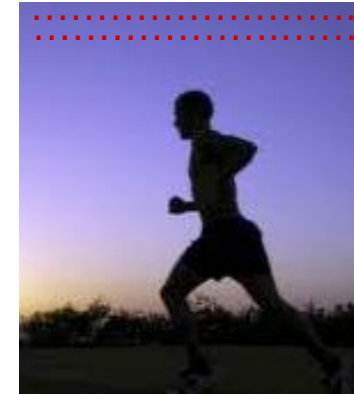
- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution: distributed, application-level infrastructure*



# Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

*spatial coding example:* instead of sending  $N$  values of same color (all purple), send only two values: color value (*purple*) and number of repeated values ( $N$ )



frame  $i$

*temporal coding example:* instead of sending complete frame at  $i+1$ , send only differences from frame  $i$

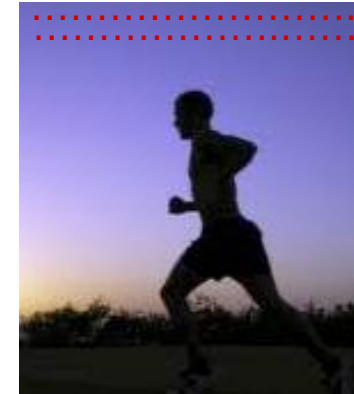


frame  $i+1$

# Multimedia: video

- **CBR: (constant bit rate):** video encoding rate fixed
- **VBR: (variable bit rate):** video encoding rate changes as amount of spatial, temporal coding changes
- **examples:**
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, 64Kbps – 12 Mbps)

*spatial coding example:* instead of sending  $N$  values of same color (all purple), send only two values: color value (*purple*) and number of repeated values ( $N$ )



frame  $i$

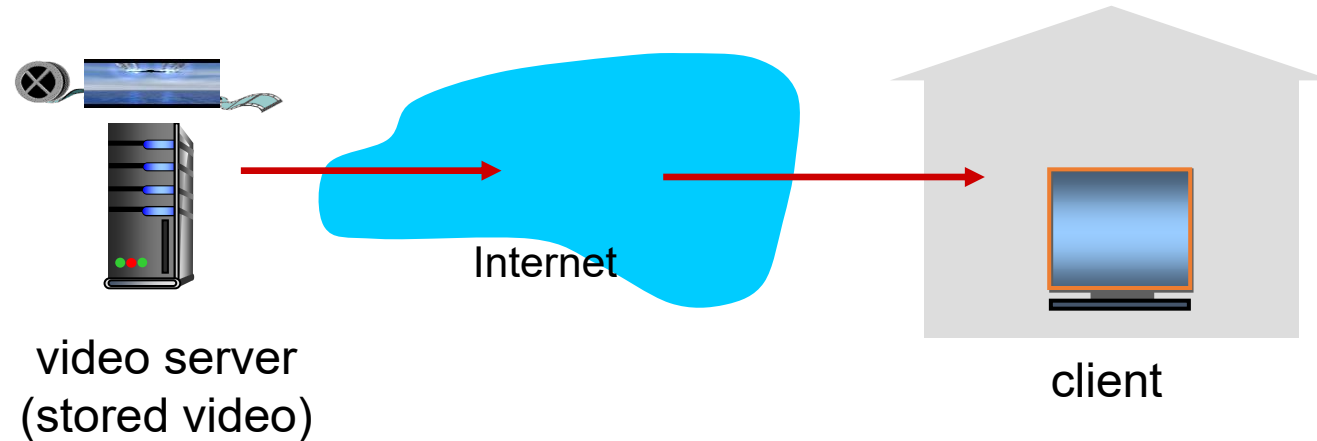
*temporal coding example:* instead of sending complete frame at  $i+1$ , send only differences from frame  $i$



frame  $i+1$

# Streaming stored video

simple scenario:

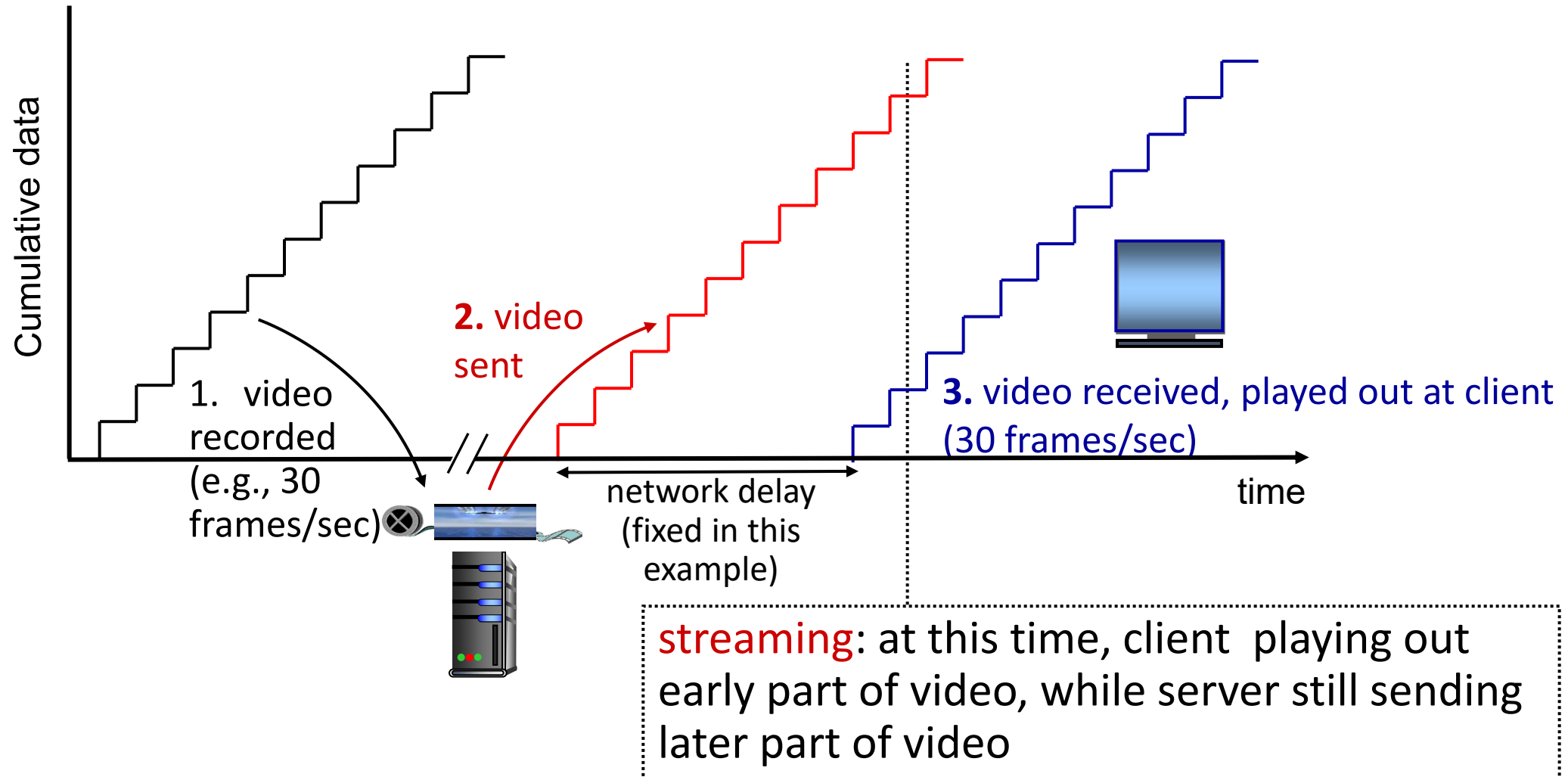


## Main challenges:

- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, in access network, in network core, at video server)
- packet loss and delay due to congestion will delay playout, or result in poor video quality

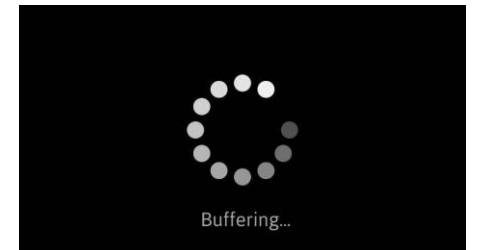


# Streaming stored video

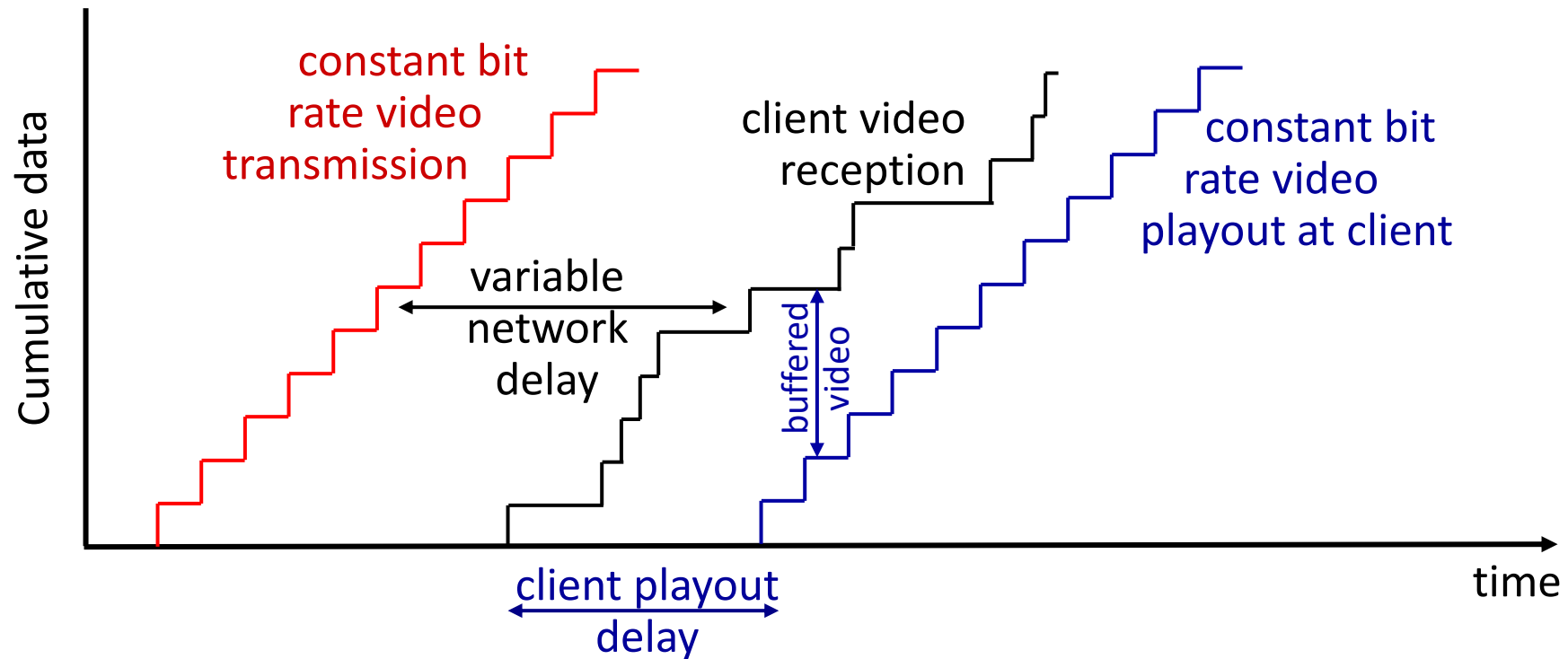


# Streaming stored video: challenges

- **continuous playout constraint**: once client playout begins, playback must match original timing
  - ... but **network delays are variable** (jitter), so will need **client-side buffer** to match playout requirements
- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted



# Streaming stored video: playout buffering



- *client-side buffering and playout delay*: compensate for network-added delay, delay jitter

# Streaming multimedia: DASH

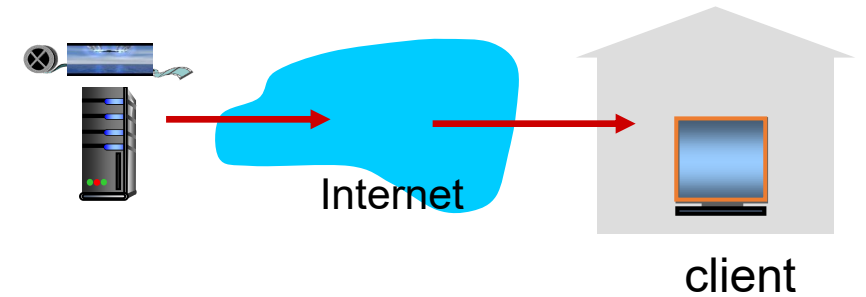
■ *DASH*: *D*ynamic *A*daptive *S*teaming over *H*TTP

■ *server*:

- divides video file into multiple chunks
- each chunk stored, encoded at different rates
- *manifest file*: provides URLs for different chunks

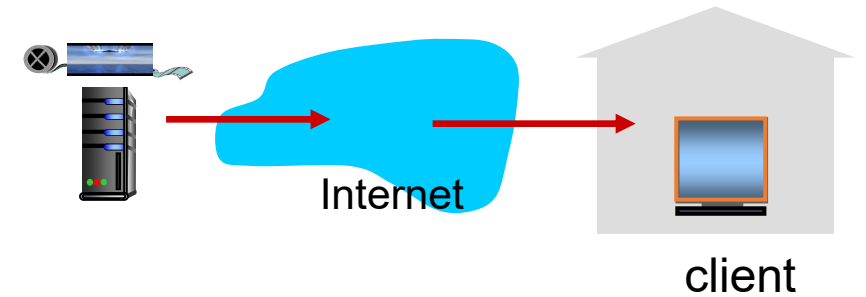
■ *client*:

- periodically measures server-to-client bandwidth
- consulting manifest, requests one chunk at a time
  - chooses maximum coding rate sustainable given current bandwidth
  - can choose different coding rates at different points in time (depending on available bandwidth at time)
- Thus, DASH allows the client to freely switch among different quality levels.



# Streaming multimedia: DASH

- “*intelligence*” at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is “close” to client or has high available bandwidth)



**Streaming video** = encoding + DASH + playout buffering

# Content distribution networks (CDNs)

- *challenge*: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- *option 1*: single, large “mega-server”
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

....quite simply: this solution *doesn't scale*

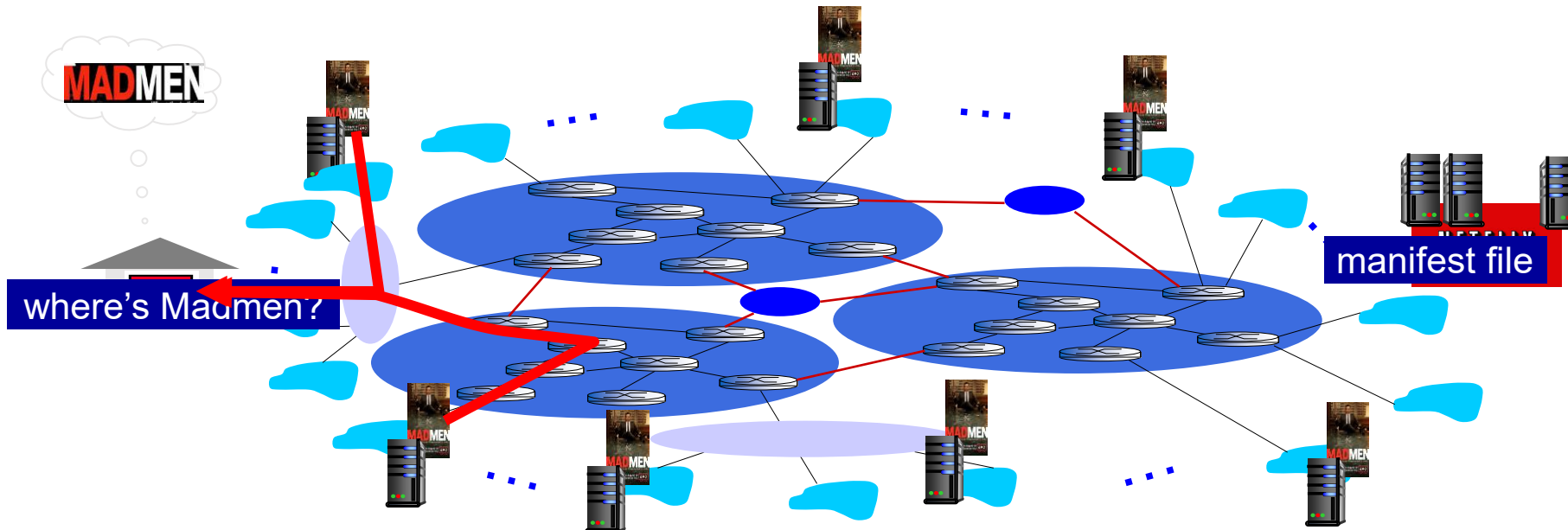
# Content distribution networks (CDNs)

- *challenge*: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- *option 2*: store/serve multiple copies of videos at multiple geographically distributed sites (*CDN*)
  - *enter deep*: push CDN servers deep into many access networks
    - close to users
    - Akamai: 240,000 servers deployed in more than 120 countries (2015)
  - *bring home*: smaller number (10's) of larger clusters in POPs near (but not within) access networks
    - used by Limelight



# Content distribution networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g., Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested





# Content distribution networks (CDNs)



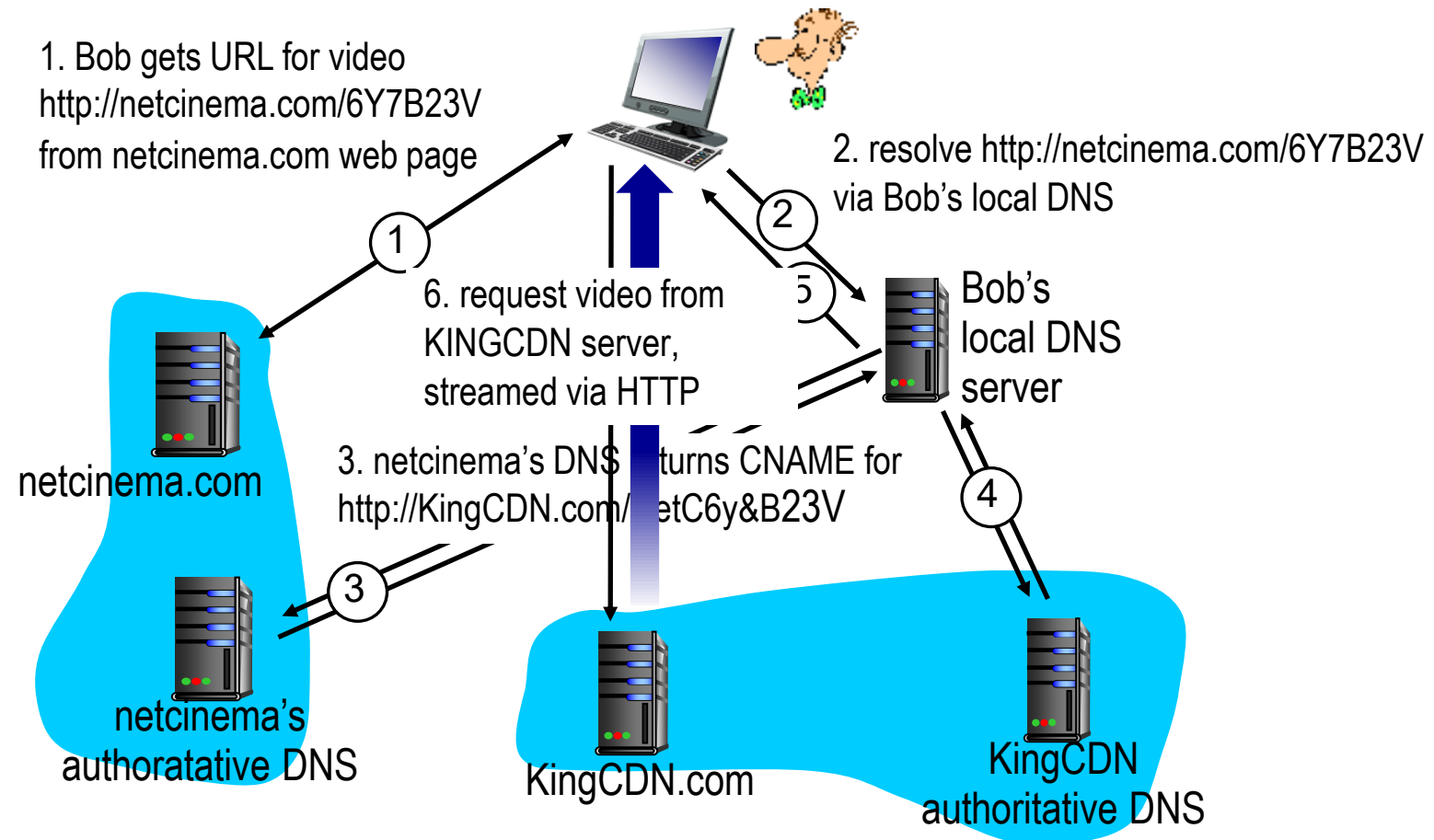
*OTT challenges:* coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

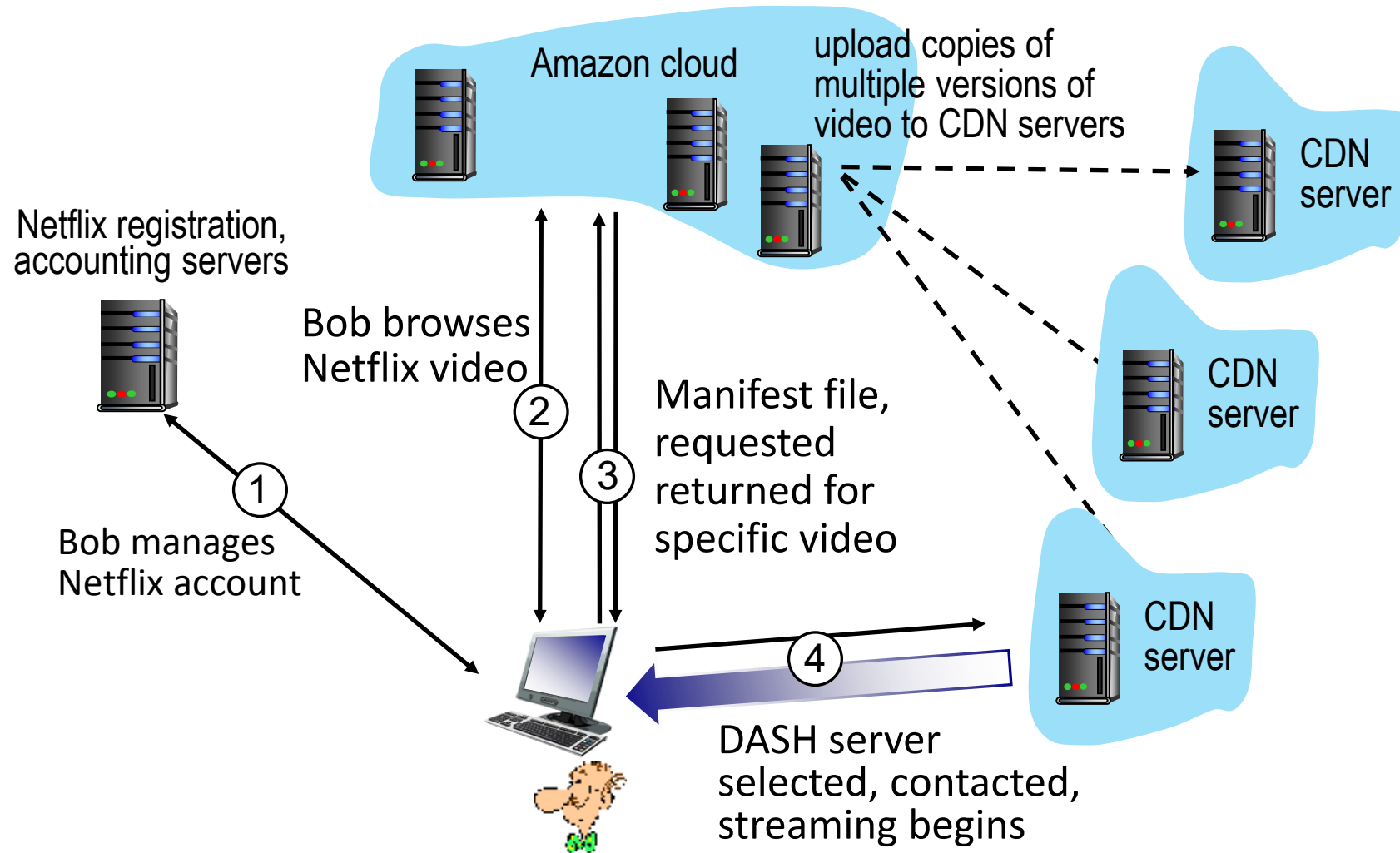
# CDN content access: a closer look

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

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



# Case study: Netflix



# Application Layer: Overview

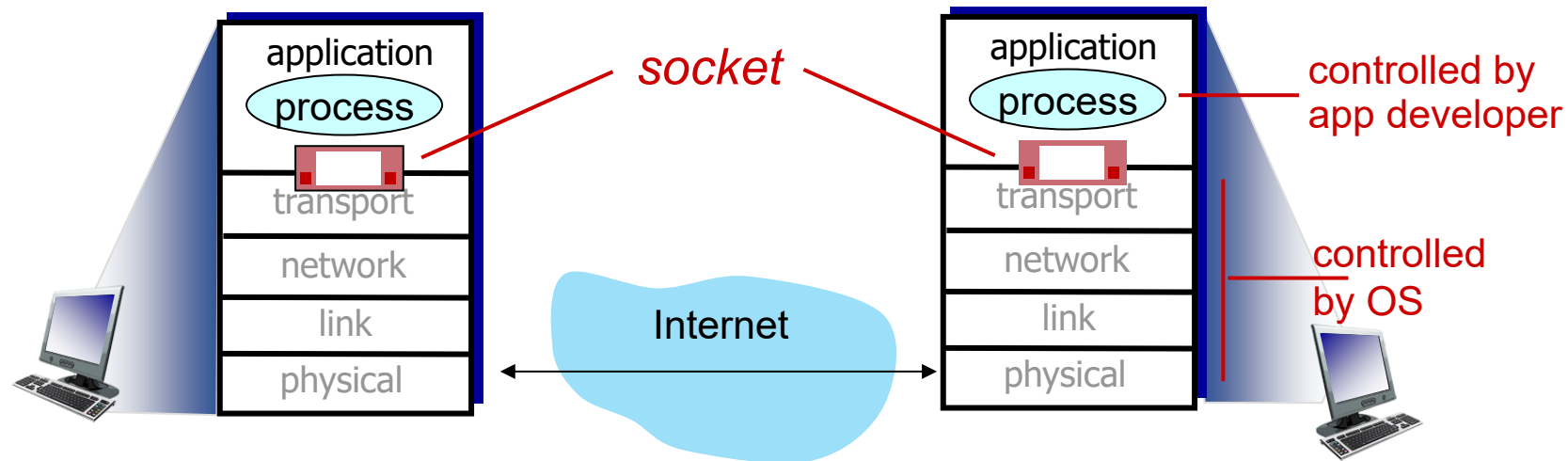
- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- **socket programming with UDP and TCP**



# 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 & 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

# 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 server IP 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 *
		serverName = 'hostname'
		serverPort = 12000
create UDP socket for server	→	clientSocket = socket(AF_INET, SOCK_DGRAM)
get user keyboard input	→	message = raw_input('Input lowercase sentence:')
attach server name, port to message; send into socket	→	clientSocket.sendto(message.encode(), (serverName, serverPort))
read reply characters from socket into string	→	modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print out received string and close socket	→	print modifiedMessage.decode() clientSocket.close()

# Example app: UDP server

## *Python UDPServer*

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(("", serverPort))
print ("The server is ready to receive")
loop forever → while True:
    Read from UDP socket into message, getting → message, clientAddress = serverSocket.recvfrom(2048)
    client's address (client IP and port)      modifiedMessage = message.decode().upper()
    send upper case string back to this client → serverSocket.sendto(modifiedMessage.encode(),
                                                                    clientAddress)
```

# 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:

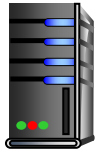
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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
  - source port numbers used to distinguish clients (more in Chap 3)

## Application viewpoint

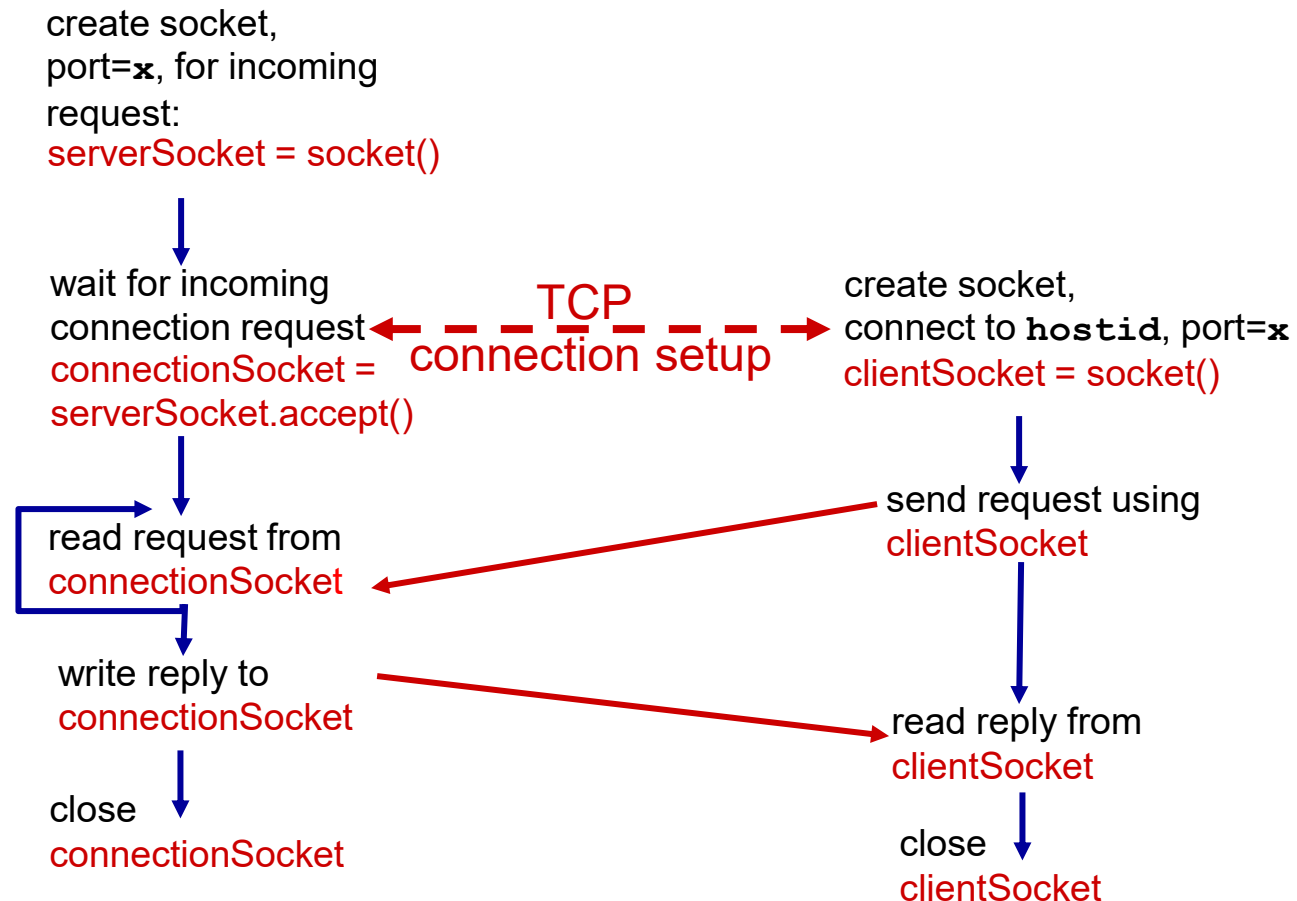
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

# Client/server socket interaction: TCP



server (running on `hostid`)

client



# Example app: TCP client

## *Python TCPClient*

create TCP socket for server,  
remote port 12000

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

No need to attach server name, port

# Example app: TCP server

## *Python TCP Server*

		<pre>from socket import *</pre>
		<pre>serverPort = 12000</pre>
create TCP welcoming socket	→	<pre>serverSocket = socket(AF_INET,SOCK_STREAM)</pre>
		<pre>serverSocket.bind(('',serverPort))</pre>
server begins listening for incoming TCP requests	→	<pre>serverSocket.listen(1)</pre>
		<pre>print 'The server is ready to receive'</pre>
loop forever	→	<pre>while True:</pre>
server waits on accept() for incoming requests, new socket created on return	→	<pre>    connectionSocket, addr = serverSocket.accept()</pre>
		<pre>    sentence = connectionSocket.recv(1024).decode()</pre>
read bytes from socket (but not address as in UDP)	→	<pre>    capitalizedSentence = sentence.upper()</pre>
		<pre>    connectionSocket.send(capitalizedSentence.encode())</pre>
close connection to this client (but <i>not</i> welcoming socket)	→	<pre>    connectionSocket.close()</pre>

# Chapter 2: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming:  
TCP, UDP sockets

# Chapter 2: Summary

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data*: info(payload) being communicated

important themes:

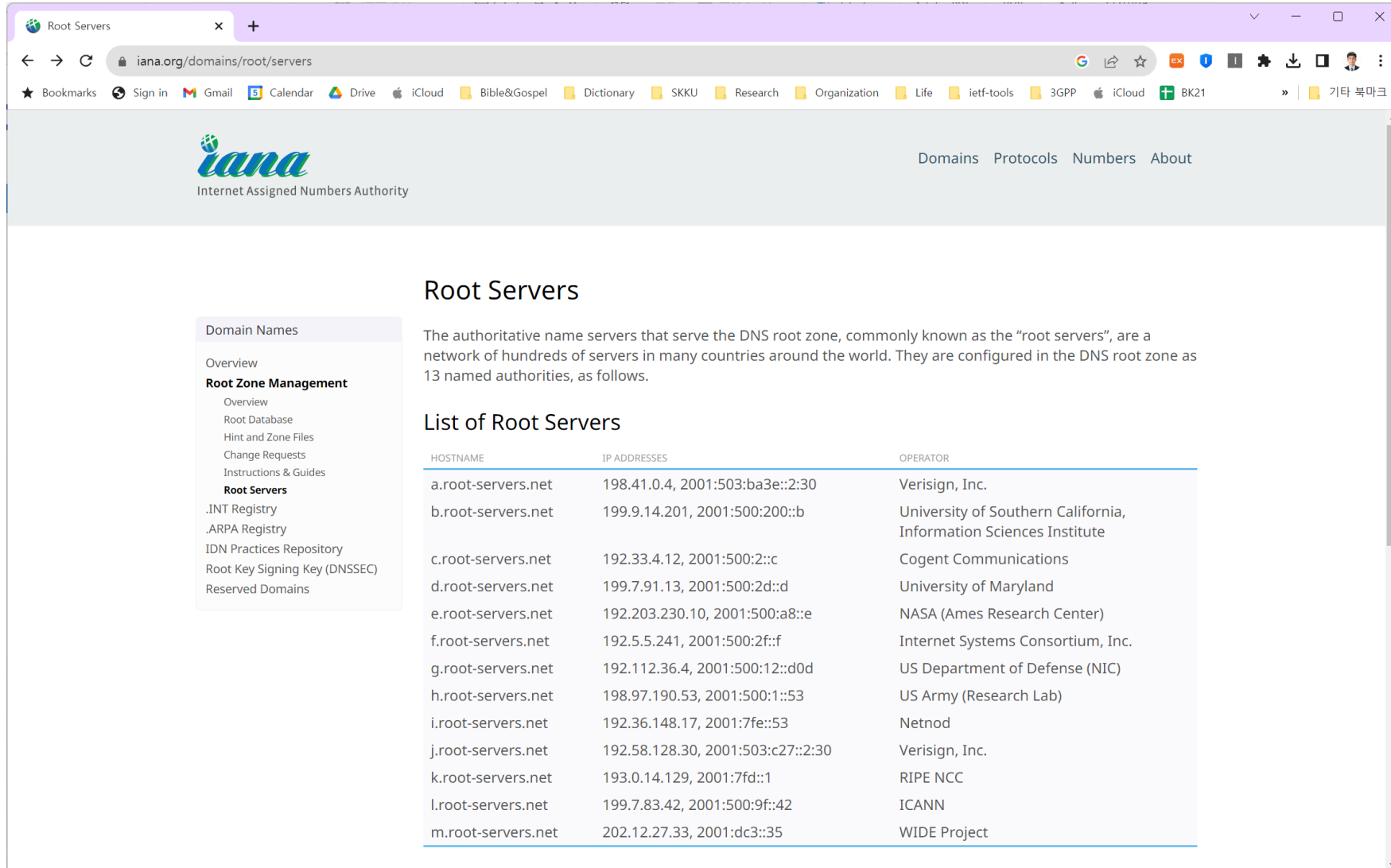
- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”



# Additional Chapter 2 Slides

# DNS Root Servers

URL: <https://www.iana.org/domains/root/servers>



The screenshot shows the IANA Root Servers page. The browser address bar displays the URL <https://www.iana.org/domains/root/servers>. The page header includes the IANA logo and navigation links for Domains, Protocols, Numbers, and About. A left sidebar contains a menu with categories like Domain Names, Root Zone Management, and Root Servers. The main content area features a title 'Root Servers', a descriptive paragraph, and a table titled 'List of Root Servers'.

## Root Servers

The authoritative name servers that serve the DNS root zone, commonly known as the “root servers”, are a network of hundreds of servers in many countries around the world. They are configured in the DNS root zone as 13 named authorities, as follows.

### List of Root Servers

HOSTNAME	IP ADDRESSES	OPERATOR
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	Verisign, Inc.
b.root-servers.net	199.9.14.201, 2001:500:200::b	University of Southern California, Information Sciences Institute
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	Verisign, Inc.
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project

# BitTorrent

URL: <https://www.bittorrent.com/>

