CSC/CPE 138

COMPUTER NETWORK FUNDAMENTALS

SACRAMENTO STATE

Lecture 2_2: Application Layer

California State University, Sacramento
Fall 2024

Slide Courtesy: Computer Networking: A Top-Down Approach, Kurose Ross, 8th Edition

Review of Lecture 2-1

- Application Layer Overview
- Client – Server Architecture
- Peer-to-peer Architecture
- Transport Layer Services
- Cookies

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection

- Server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- With FCFS, small object may have to wait for transmission (head-of-line (HOL) blocking) behind large object(s)
- Loss recovery (retransmitting lost TCP segments) stalls object transmission

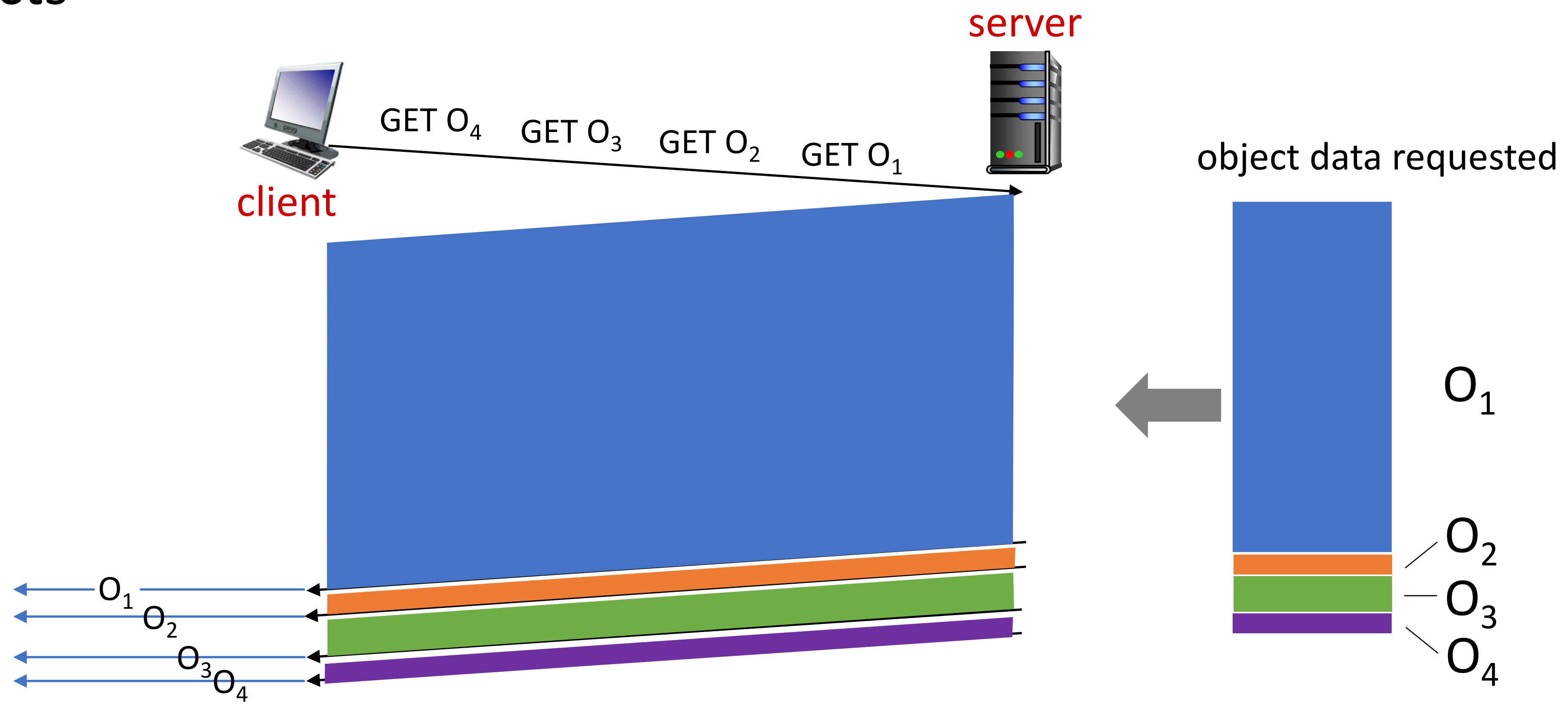
HTTP/2

Key goal: decreased delay in multi-object HTTP requests

- *HTTP/2:* [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:
- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

HTTP/2: mitigating HOL blocking

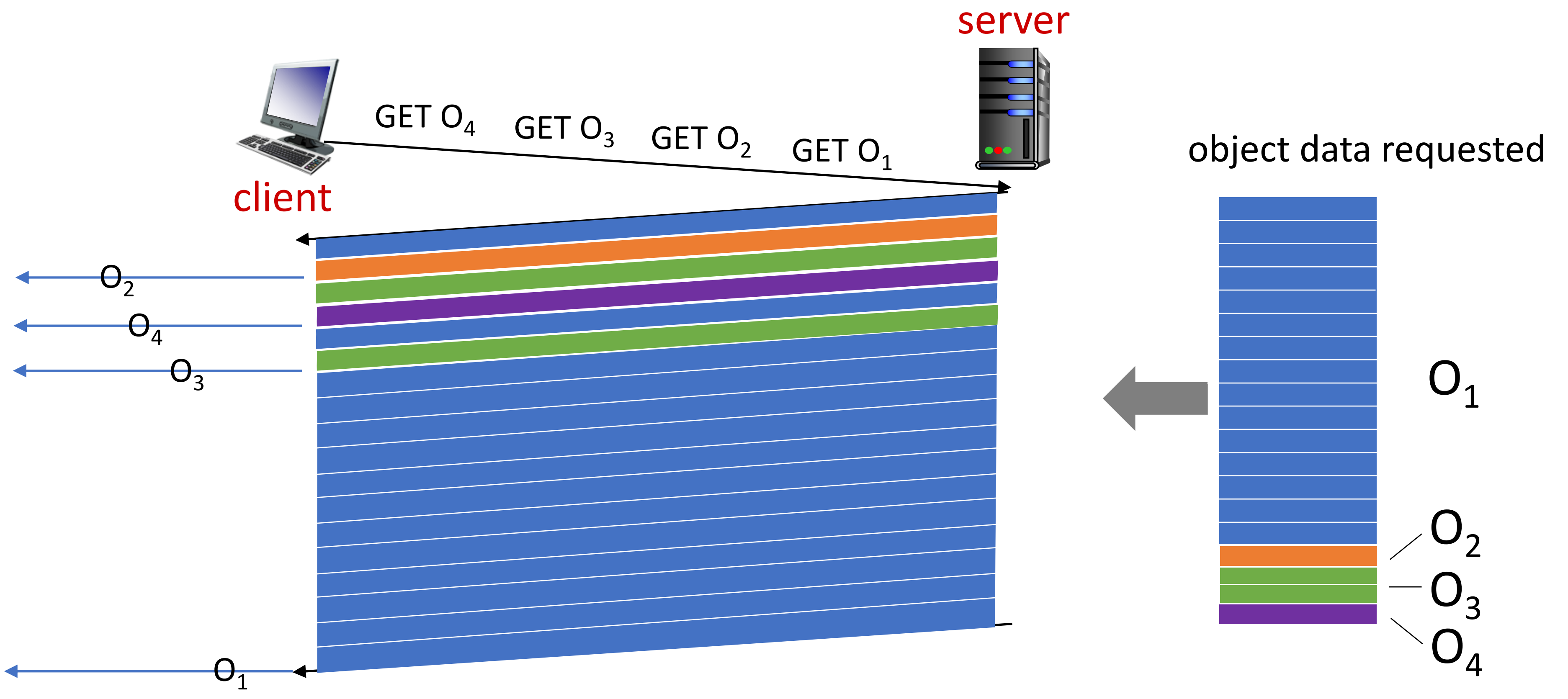
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/2 to HTTP/3

HTTP/2 over single TCP connection means:

- Recovery from packet loss still stalls all object transmissions
  - as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- No security over vanilla TCP connection
- **HTTP/3**: adds security, per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer

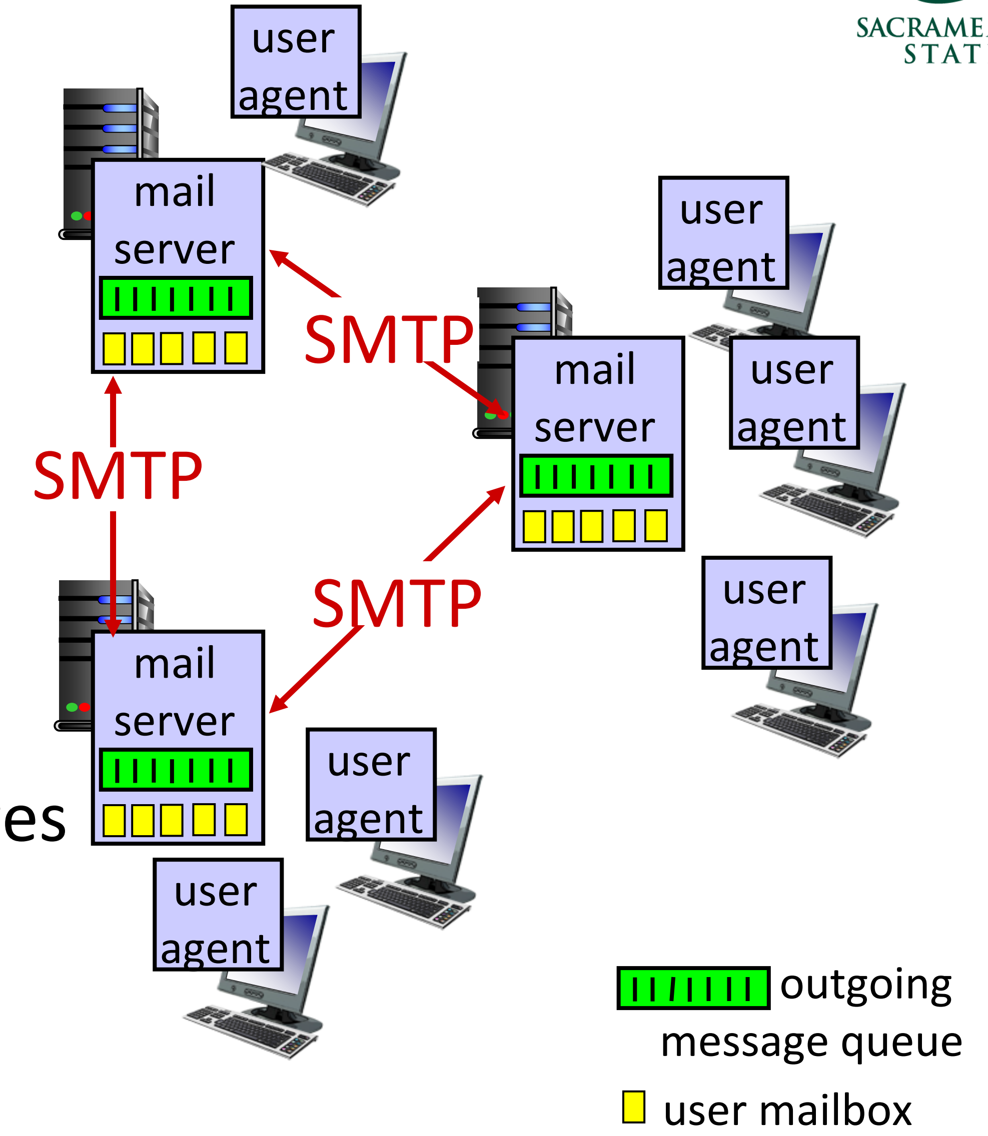
E-mail

Three major components:

- User agents
- Mail servers
- Simple mail transfer protocol: SMTP

User Agent

- a.k.a. “mail reader”
- Composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- Outgoing, incoming messages stored on server



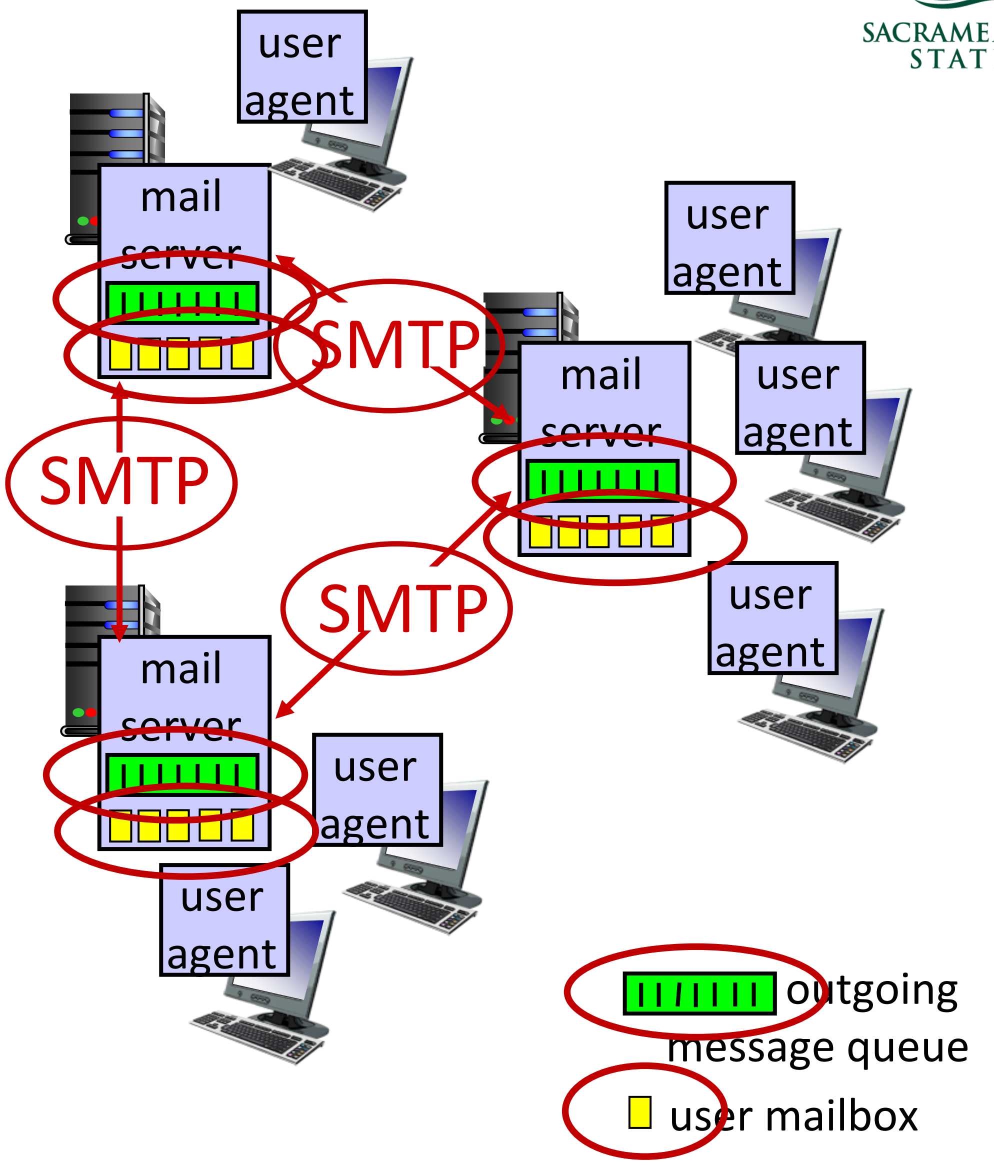
E-mail: mail servers

Mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages

SMTP protocol between mail servers to send email messages

- client: sending mail server
- "server": receiving mail server



Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message "to" bob@someschool.edu

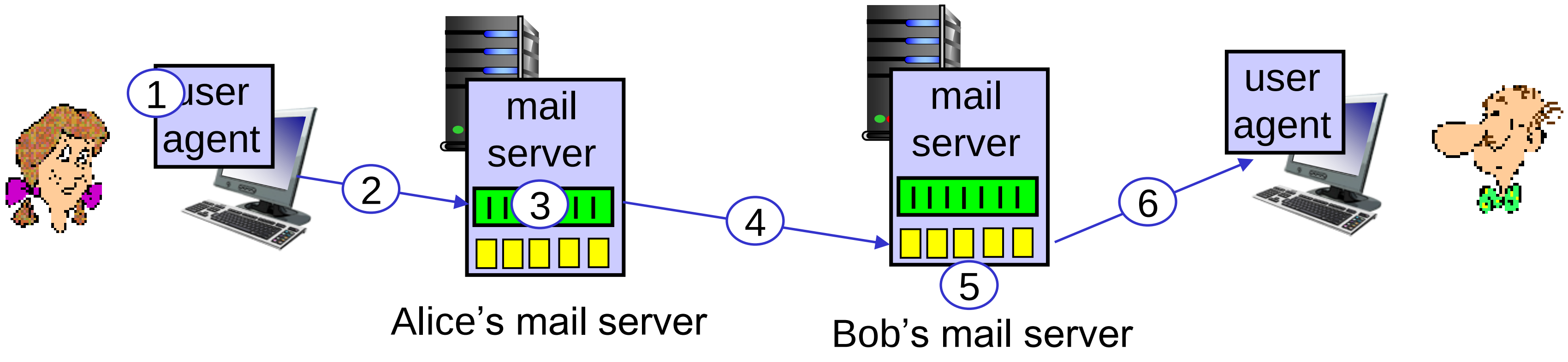
2) Alice's UA sends message to her mail server using SMTP; message placed in message queue

3) Client side of SMTP at mail server opens TCP connection with Bob's mail server

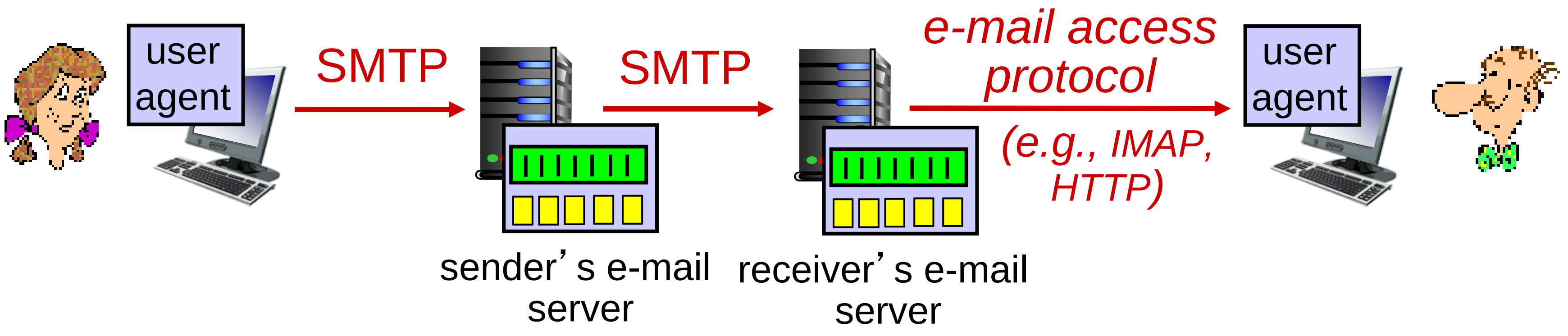
4) SMTP client sends Alice's message over the TCP connection

5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message



Retrieving email: mail access protocols



- SMTP: delivery/storage of e-mail messages to receiver’s server
- Mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

DNS: Domain Name System

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., cs.umass.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, DNS 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 name

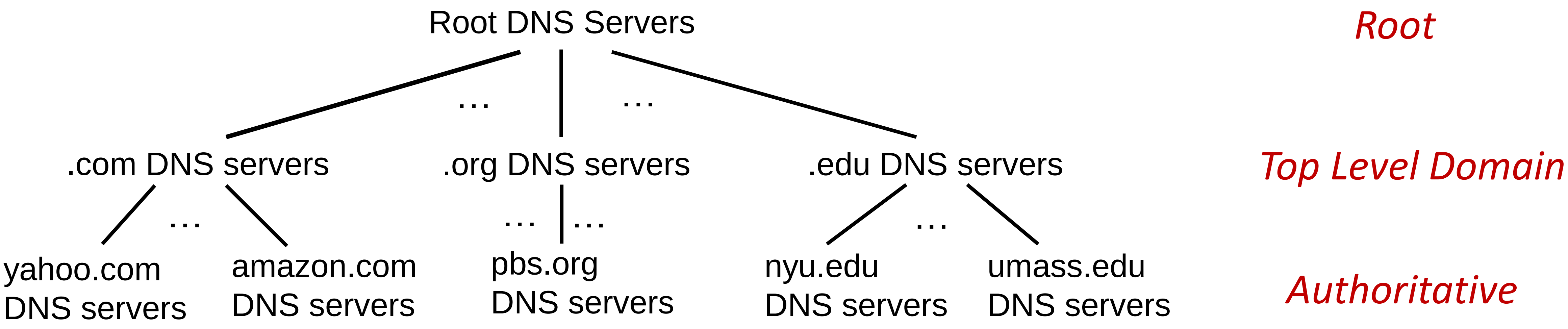
Q: Why not centralize DNS?

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

A: doesn't scale!

- Comcast DNS servers alone: 600B DNS queries/day
- Akamai DNS servers alone: 2.2T DNS queries/day

DNS: a distributed, hierarchical database



Client wants IP address for www.amazon.com; 1st 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

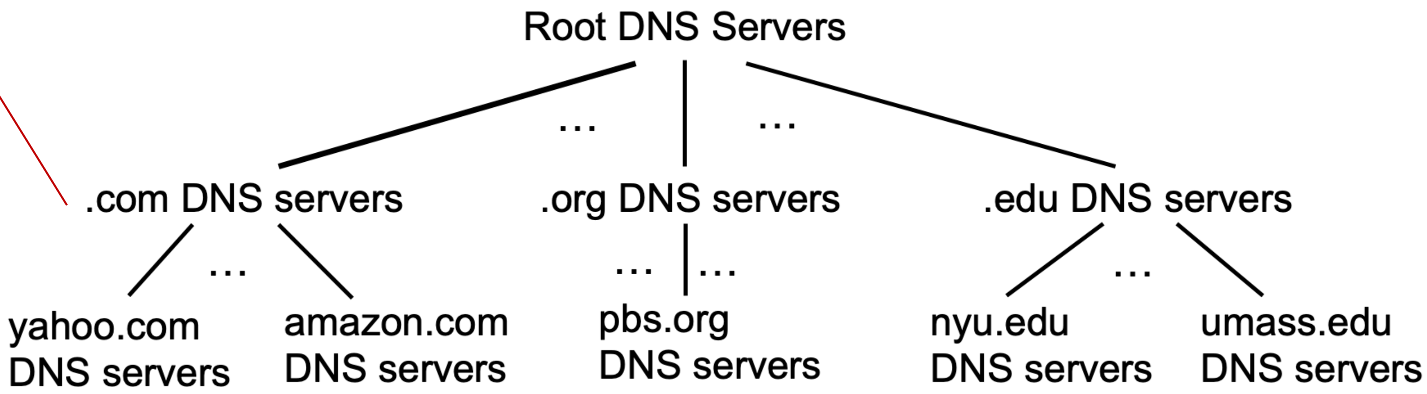
DNS: Servers

Root name servers:

- Official, contact-of-last-resort by name servers that can not resolve name

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
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .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

- When host makes DNS query, it is sent to its *local* DNS server
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (possibly out of date!)
    - forwarding request into DNS hierarchy for resolution
  - Each ISP has local DNS name server; to find yours:
    - MacOS: `% scutil --dns`
    - Windows: `>ipconfig /all`
- Local DNS server doesn't strictly belong to the hierarchy
- Caching mapped servers

DNS records

DNS: distributed database storing resource records (RR)

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

type=A

- `name` is hostname
- `value` is IP address

type=NS

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

type=CNAME

- `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

- `value` is name of SMTP mail server associated with `name`

How to setup a DNS record

Example: new startup "Network Utopia"

- Tegister name networkuptopia.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.networkuptopia.com
  - type MX record for networkutopia.com

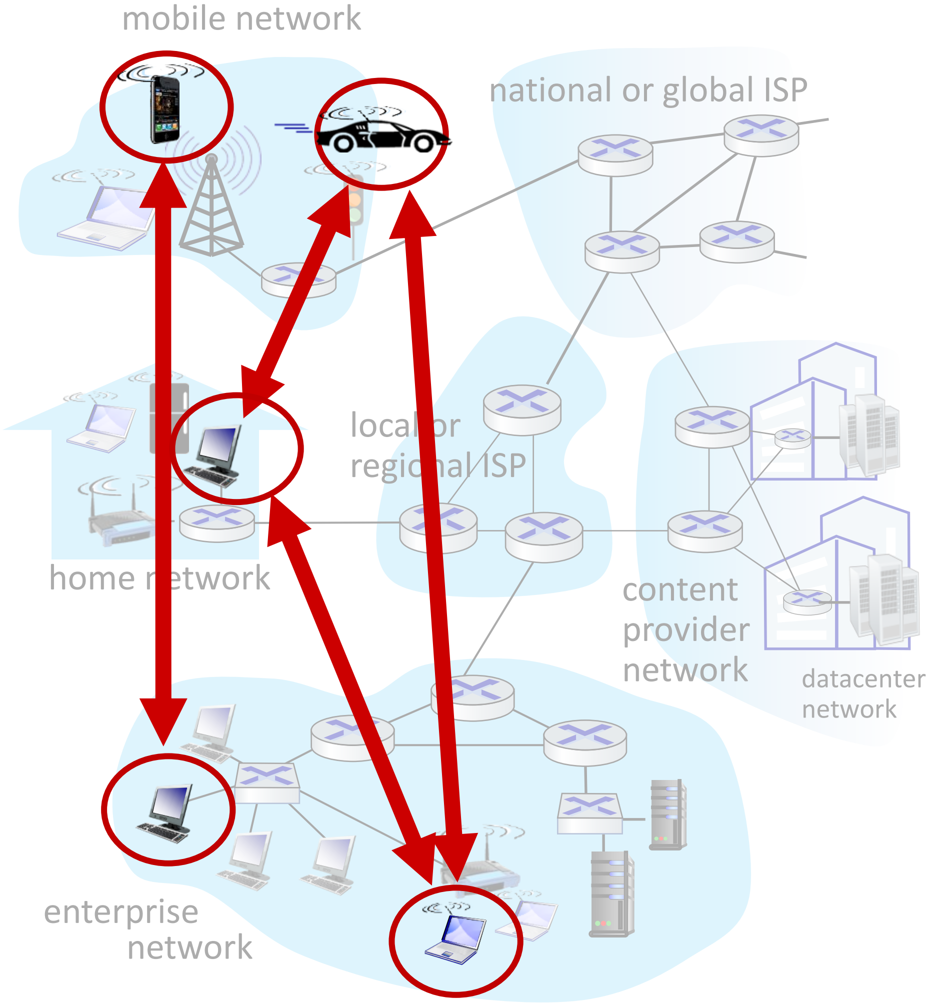
Review

- Question 1: You are hired as a network engineer in a XYZ company. The CEO asks you to upgrade the company's web server1 as customers experience a delay in loading the website objects. The website downloads objects sequentially and experiences HOL blocking; what do you think might be the problem? Do you have any recommendations?
  Answer: Webserver 1 uses HTTP1.1 and requires upgradation to HTTP 2

- Question 2: You have solved the web-server1 problem. Now your CEO asks you to host another web-server (web-server 2) for hosting a new application. The web server 2 needs to be accessed from the internet with a url www.web-server2.com. What service/protocol do you need to configure to allow the url access? Additionally, what records do you need to declare?
- Answer: DNS, A record

Evaluating File Distribution Time

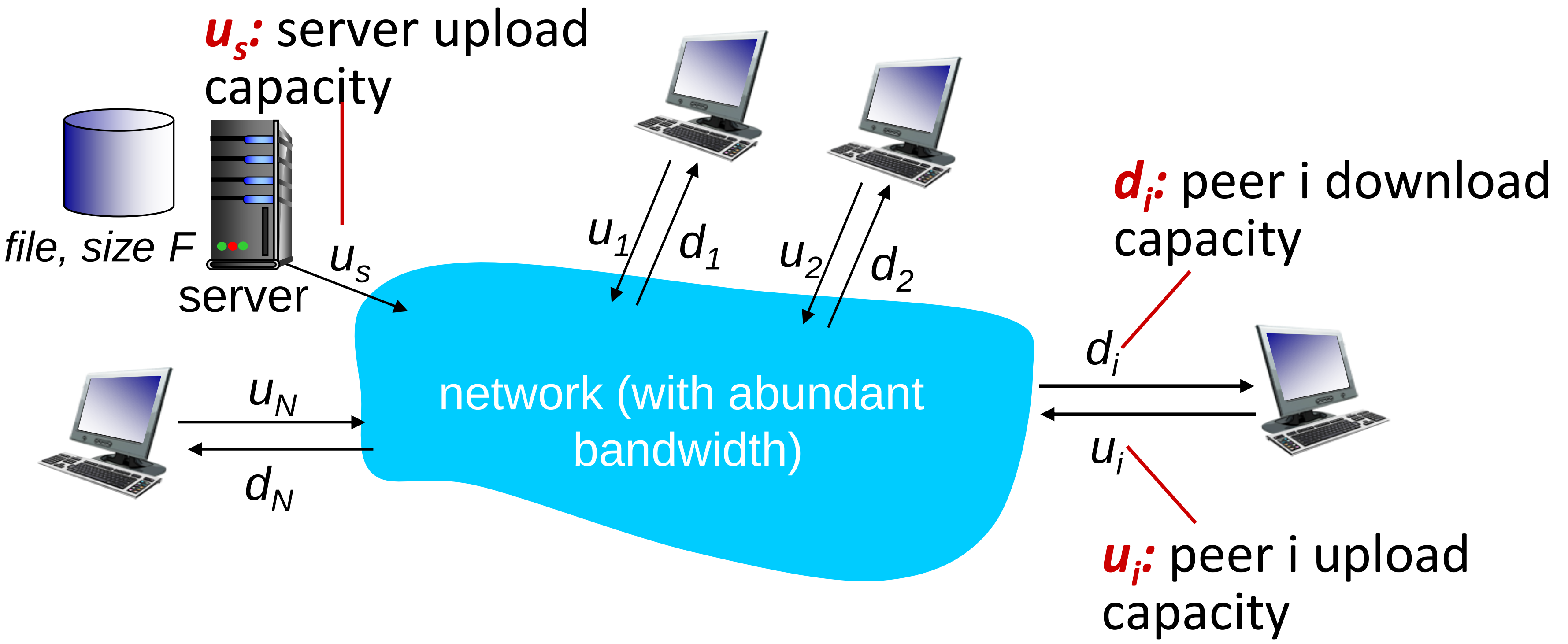
- *Review P2P Architecture*
  - *No* always-on server
  - Arbitrary end systems directly communicate
  - Peers request service from other peers, provide service in return to other peers
    - *self scalability* – new peers bring new service capacity, and new service demands
  - Peers are intermittently connected and change IP addresses



File distribution: client-server vs P2P

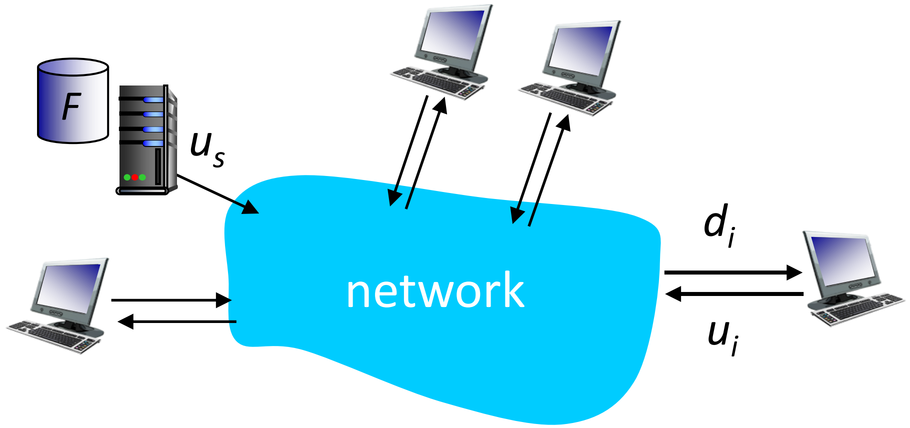
Q: how much time to distribute file (size F) from one server to *N peers*?

- 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 client download rate
  - min client download time: F/d_{min}

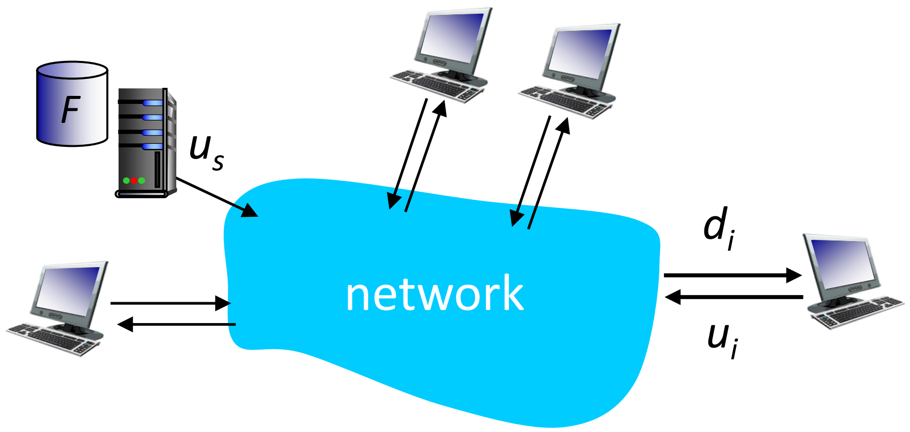
time to distribute F to N clients using client-server approach

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

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 must download *NF* bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$



time to distribute F to N clients using P2P approach

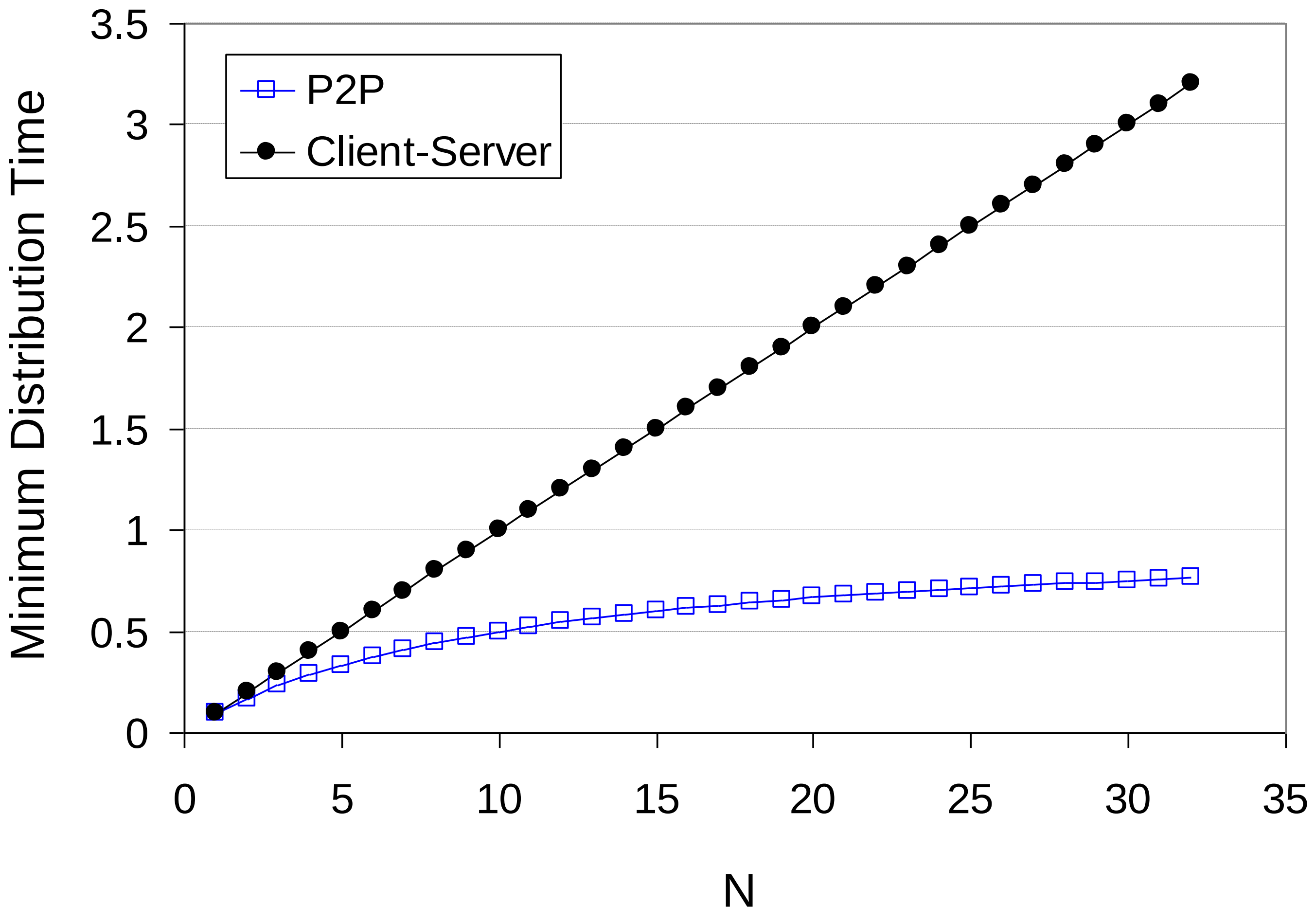
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

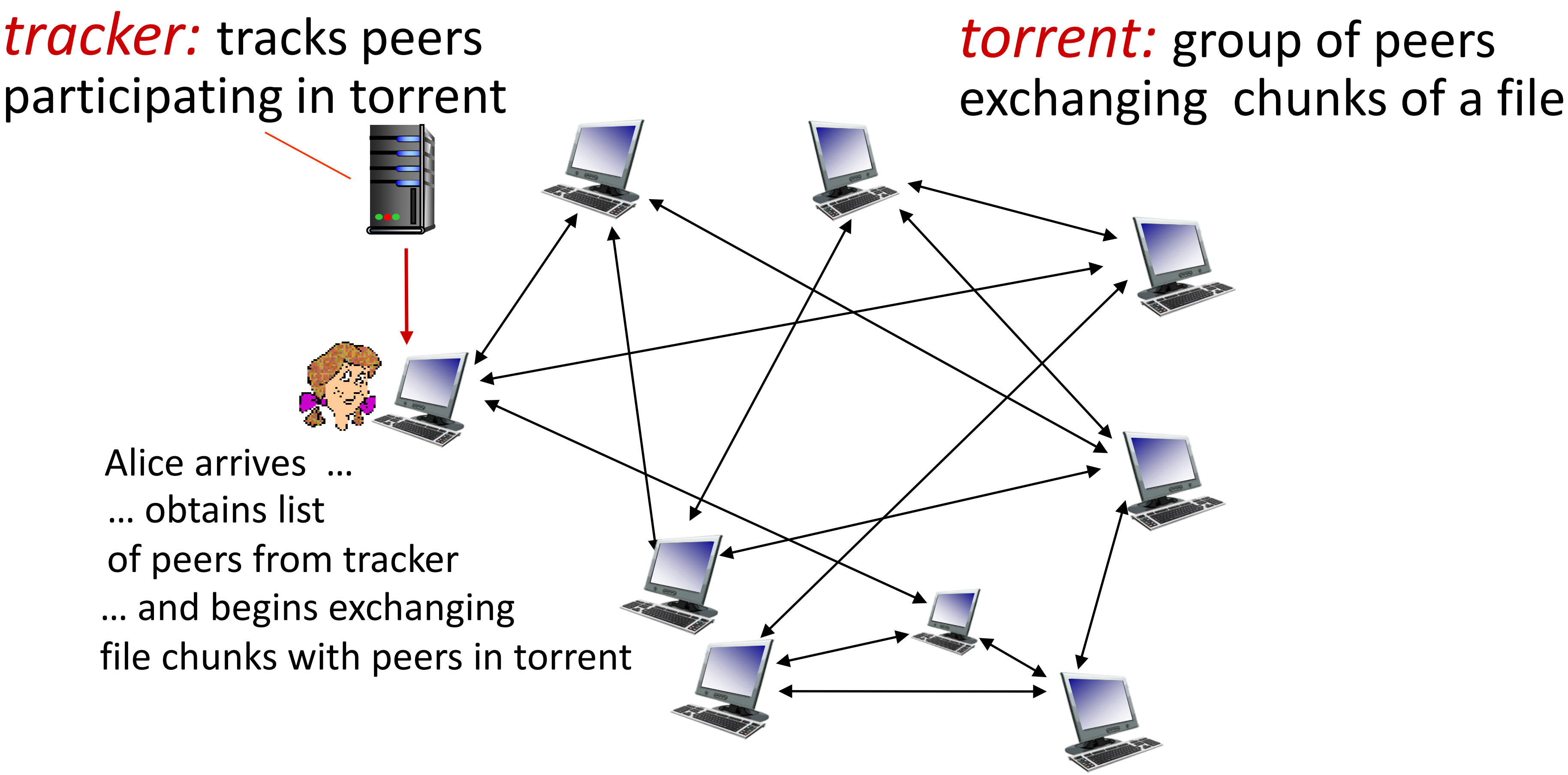
Client-server vs. P2P: example

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

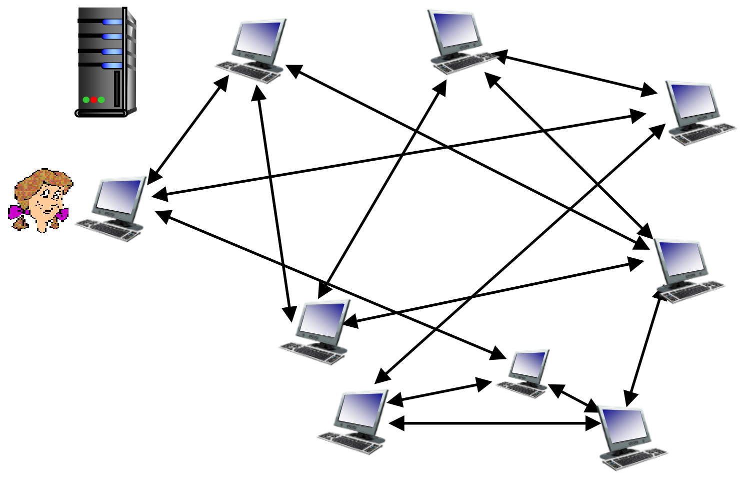


P2P file distribution: BitTorrent

- File divided into 256Kb chunks
- Peers in torrent send/receive file chunks



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:

- 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

Sending chunks: tit-for-tat

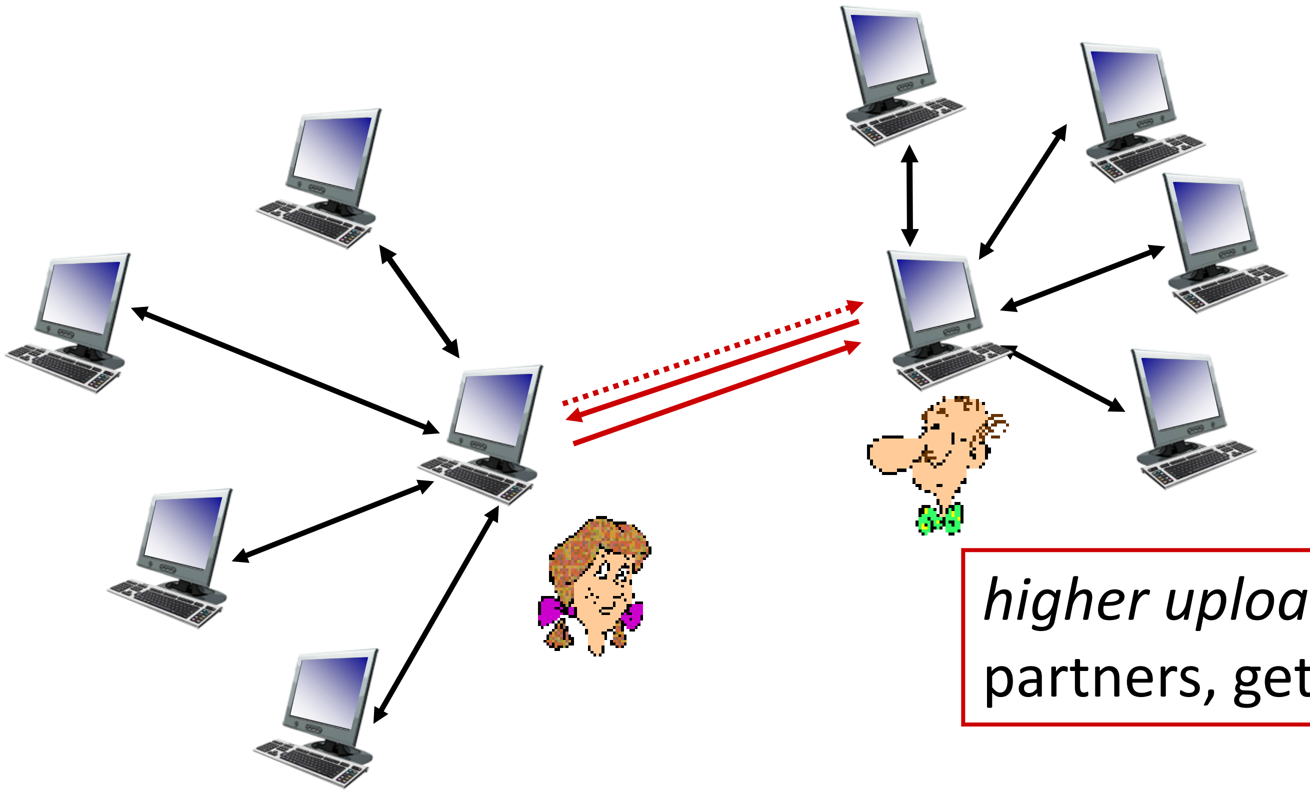
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - “optimistically unchoke” this peer
  - 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



higher upload rate: find better trading partners, get file faster !

Video Streaming and Content Delivery Networks (CDN)

- 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?
- *Challenge:* heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *Solution:* distributed, application-level infrastructure

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 (and possibly congested) path to distant clients

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

Content distribution networks (CDNs)

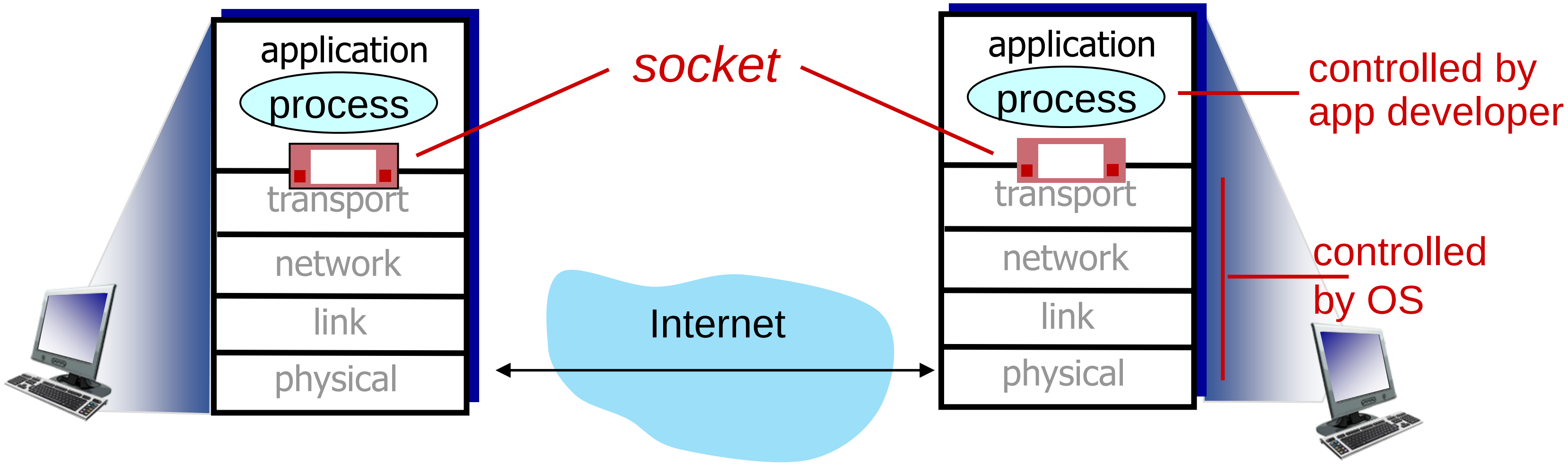
Challenge: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *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 > 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in POPs near access nets
    - used by Limelight

Socket programming (Low level networking)

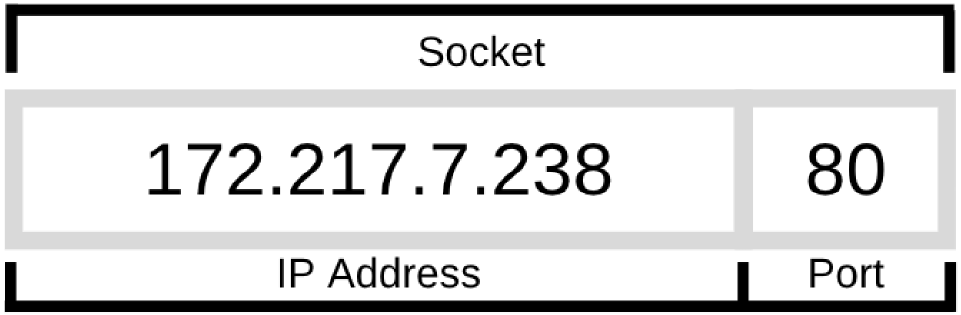
Goal: learn how to build client/server applications that communicate using sockets

Socket: door between application process and end-end-transport protocol

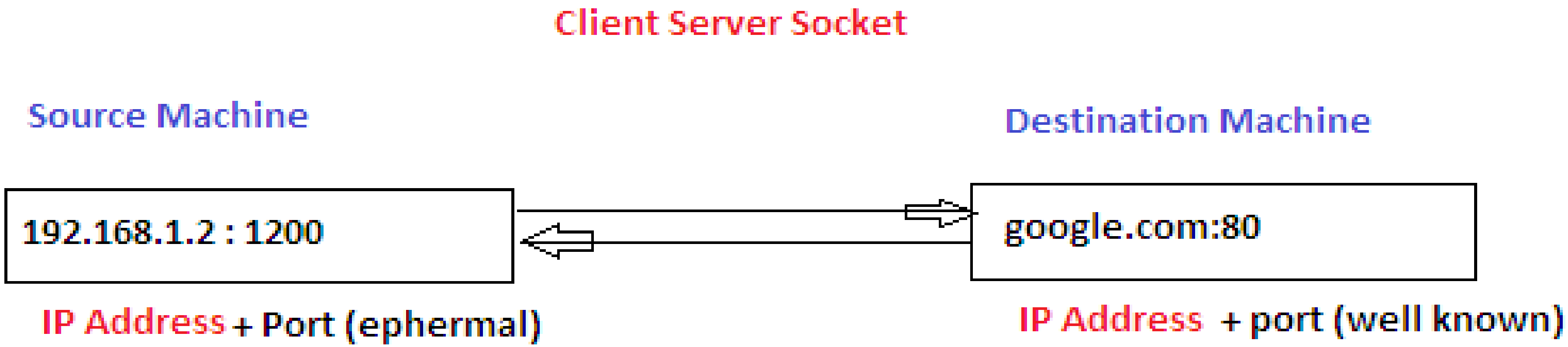


Similar to telephone example where you need a connection

How do sockets communicate : Example



Socket is defined as a class with **IP address** and a **port number**



Communication example of a Socket

Socket programming

Two socket types for two transport services:

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

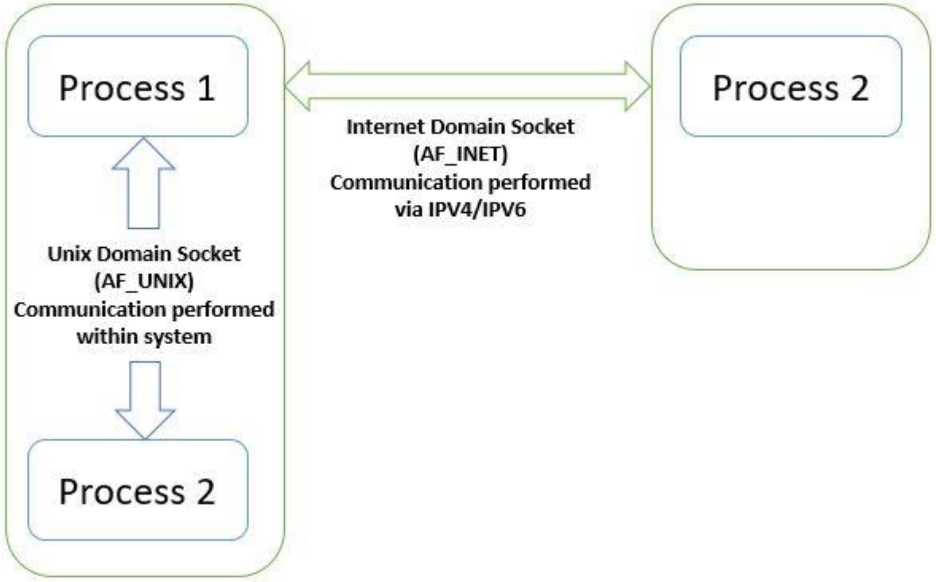
Application Example:

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 Attributes

- Sockets are characterized by three attributes
  - Domain
  - Type
  - Protocol

- For communication between processes, sockets can be implemented in the following domains
  - UNIX (e.g. : AF_UNIX)
    - Processes are on the same machine
  - INTET(e.g. : AF_INET)
    - Each process is on different machine



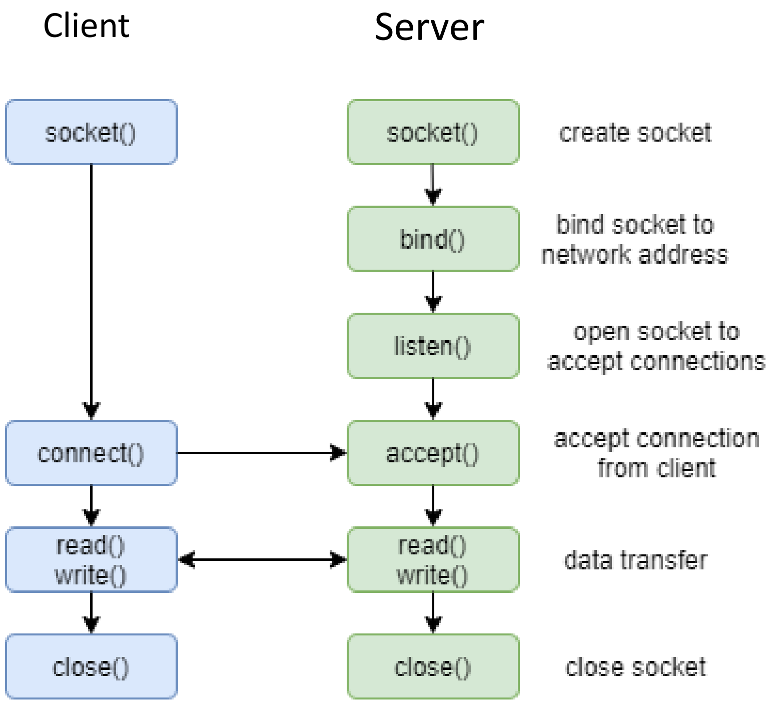
Example of Socket Domain

Socket Attributes

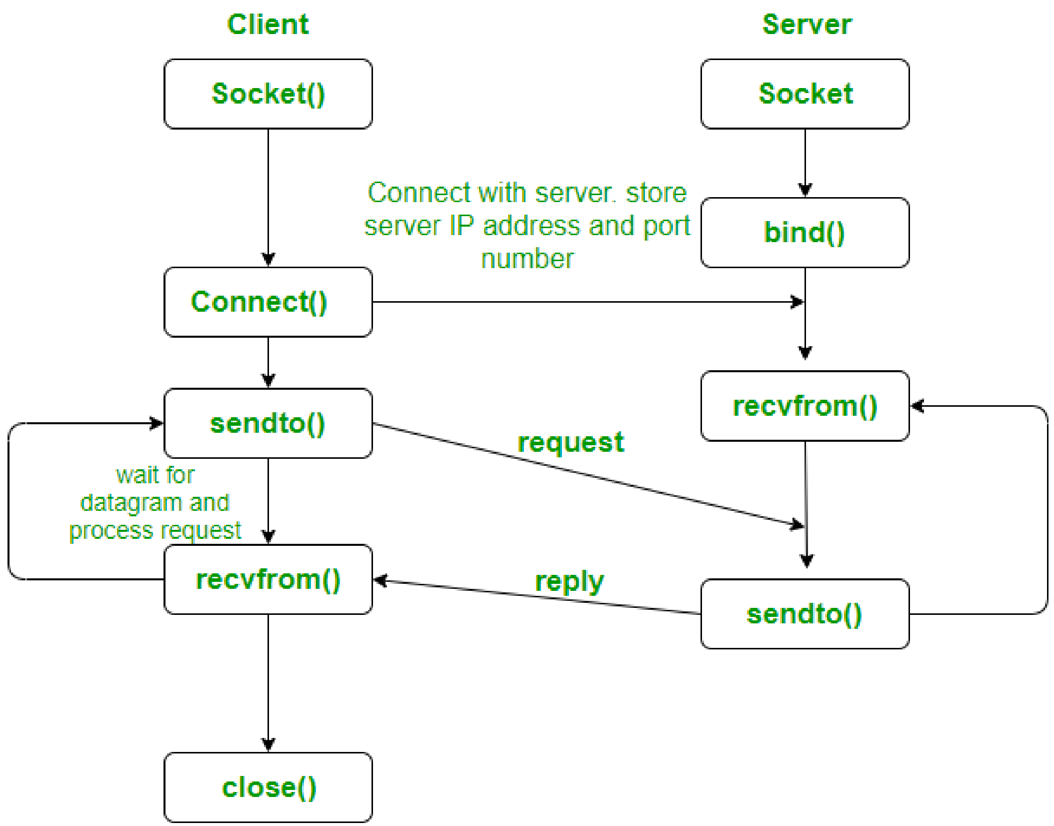
SOCK_STREAM	• Provides a connection oriented, sequenced, reliable and bidirectional network communication, (e.g: TCP)
SOCK_DGRAM	• Provides a connectionless, unreliable, best-effort network communication service (e.g.: UDP)
SOCK_RAW	• Allows direct access to other layer protocols such as IP, ICMP or IGMP

Types of Sockets

Socket Flows



TCP Socket Flow



UDP Socket Flow

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

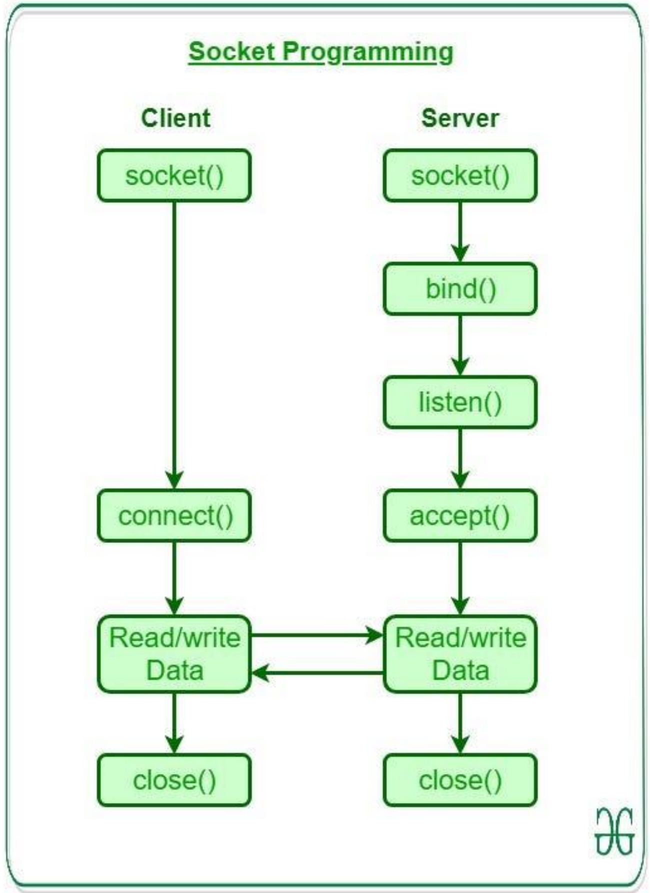
- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - client source port # and IP address used to distinguish clients (more in Chap 3)

Application viewpoint

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

Fundamental logic - TCP

- Endpoints
- Address of both ends
- Initiation from one end (client)
- Other end waits for connection (server)
- Once connection established, send messages
- Once messages are sent over, terminate

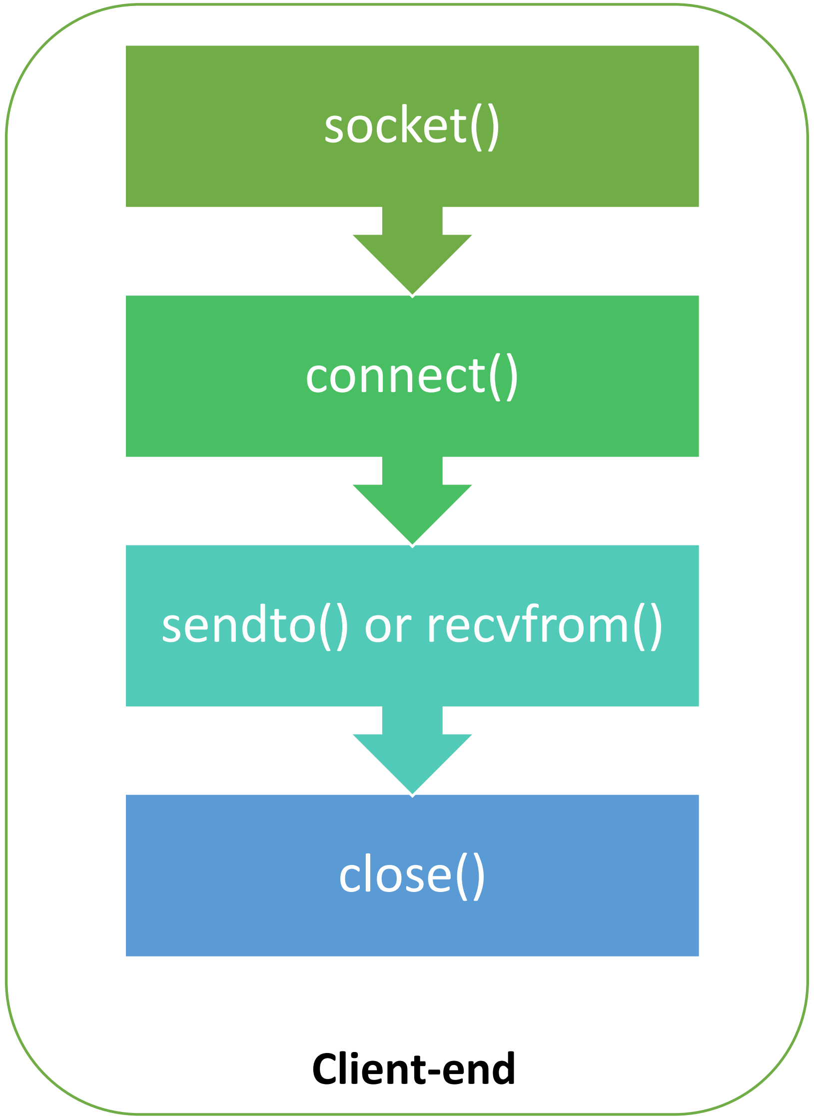


Socket Communication Example - TCP

“Client” Socket Workflow - TCP

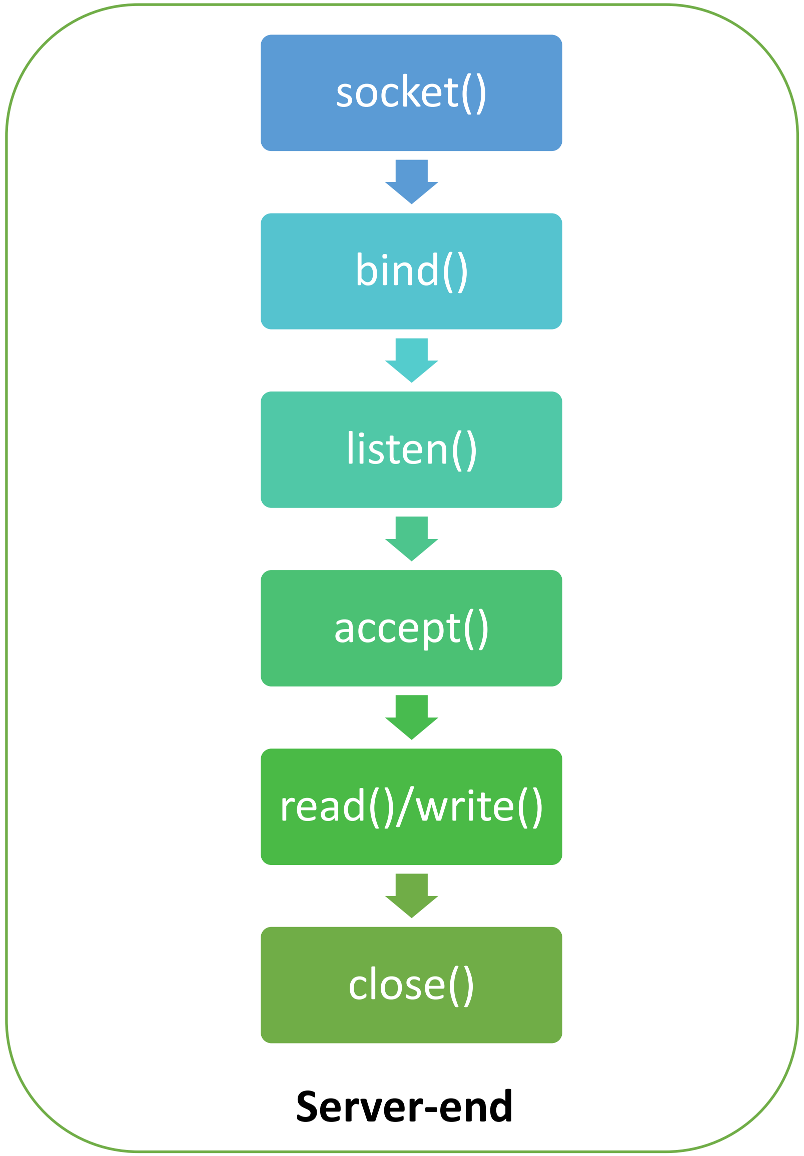
- Socket created with **socket()**
- Connect to a remote address with **connect()**

- Send or receive data with the sendto() or **recvfrom()**
- Close the connection with **close()**

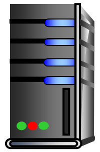


“Server” Socket Workflow - TCP

- Socket created with **socket()**
- Binds the socket to the address and port number with **bind()**
- Waits for the client to approach the server to make a connection with **listen()**
- Creates a new connected socket, and returns a new file descriptor with **accept()**



Client/server socket interaction: TCP



server (running on hostid)

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

Bind address to port
bind(address, port)

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 in Python : 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 in Python: 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)

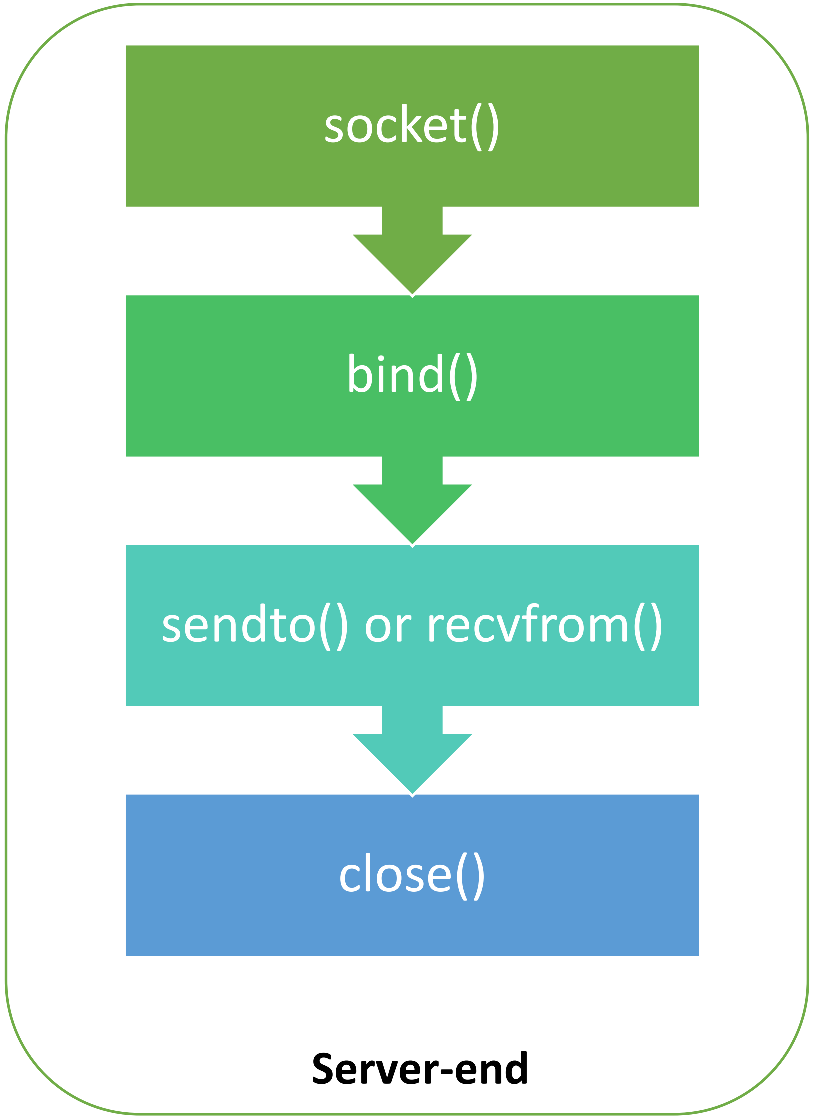
```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()

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

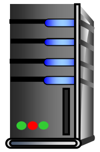
Socket programming with UDP

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
- Transmitted data may be lost or received out-of-order



Client/server socket interaction: UDP



server (running on serverIP)

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

Bind address to port
bind(address, port)

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 in Python: UDP client

Python UDPClient

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

```
serverName = 'hostname'
serverPort = 12000
```

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()
```

Note: this code update (2023) to Python 3

Example in Python: 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

End of Lecture 2_2