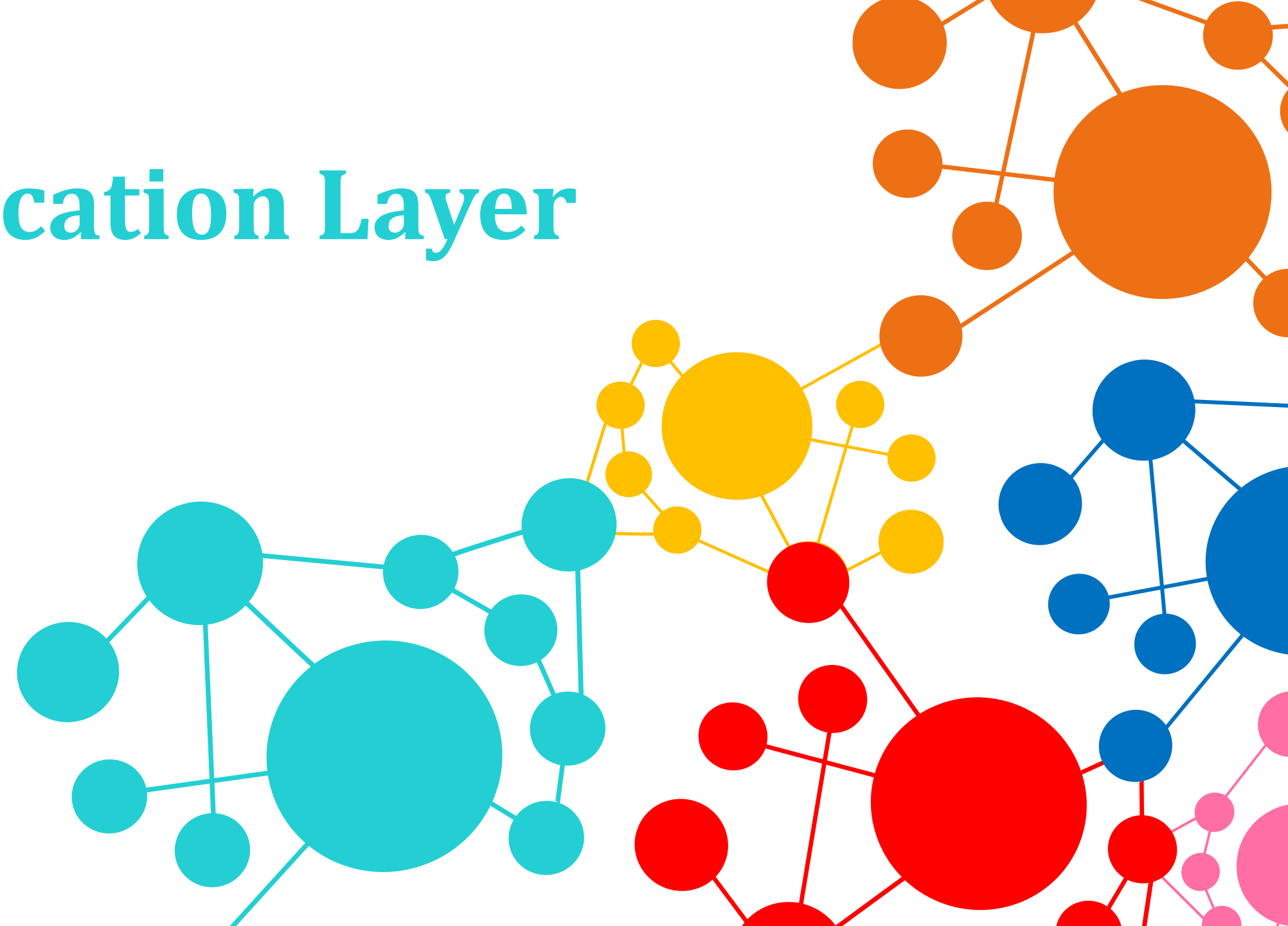
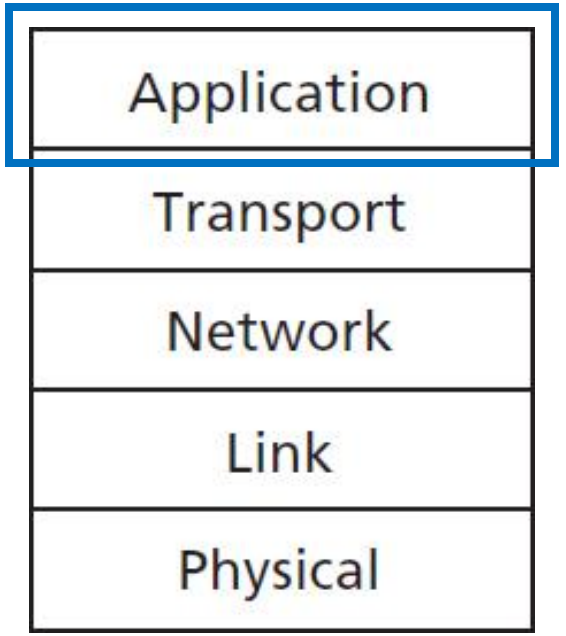
Application Layer



Application Layer

- Network Applications
- Web & HTTP
- E-mail (SMTP, POP3, IMAP)
- Domain Name System (DNS)
- Peer-to-Peer (P2P) Applications
- Video Streaming and Content Distribution Networks (CDN)
- Socket Programming (UDP & TCP)



Network Applications

Network Applications

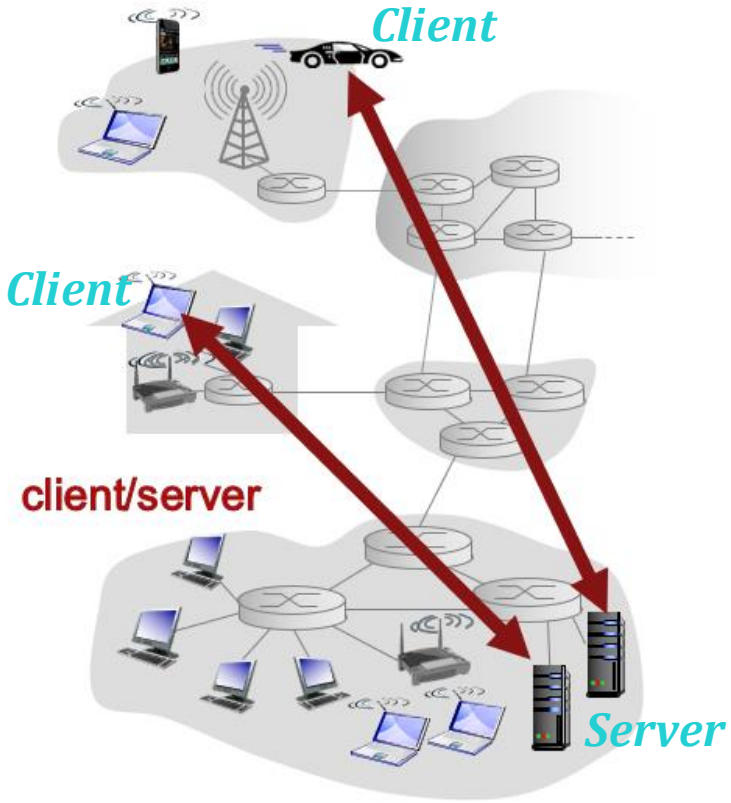
- Example Network Applications
  - E-mail
  - Web
  - Messaging
  - Remote Login
  - P2P File Sharing
  - Multi-user Network Games
  - Video Streaming (YouTube, Hulu, Netflix)
  - Voice Over IP (Skype)
  - Real-Time Video Conferencing
  - Social Networking
  - Search

- Programs that
  - Run on (different) end systems
  - Communicate over network

- Application Architectures
  - Client-Server
  - Peer-to-Peer (P2P)

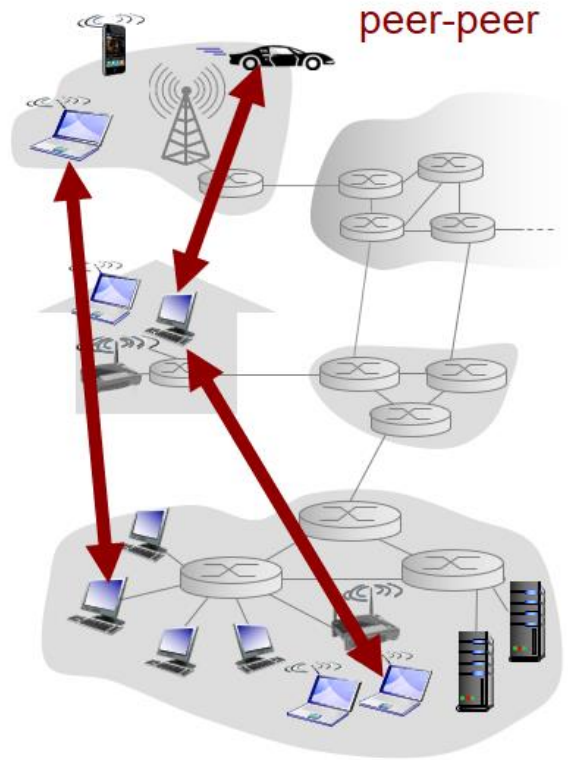
Network Applications: Client-Server

- Server
  - Always-on host
  - Permanent IP address
  - Data centers for scaling
- Clients
  - Communicate with Server
  - May be intermittently connected
  - May have dynamic IP addresses
  - Do not communicate directly with each other



Network Applications: P2P

- No always-on server: Arbitrary end systems directly communicate
  - Peers request service from other peers
  - Peer provide service in return to other peers
- Self Scalability
  - New peers bring new service capacity
  - New peers bring new service demands
- Peers are intermittently connected & change IP addresses
  - Complex Management



Network Applications: Processes

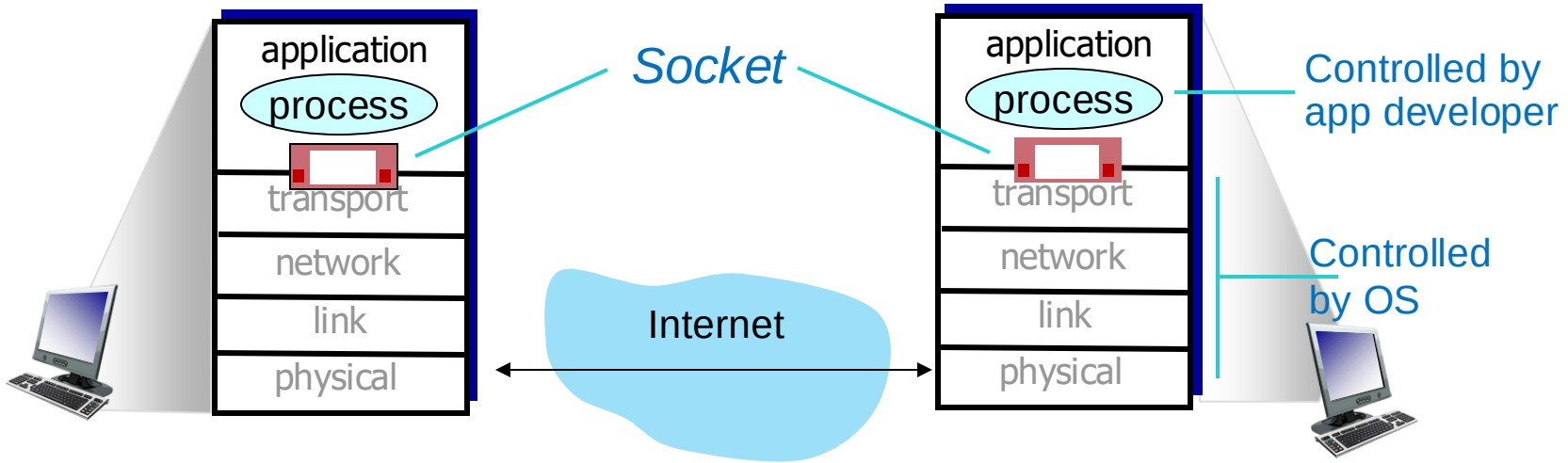
- Process: Program running on a host

- Processes Communicating
  - Processes running on the same host
    - Two processes communicate using inter-process communication (Defined by OS)

  - Processes running on different hosts
    - Two processes communicate by message exchange (using **Socket**) over the network

Network Applications: Socket

- Processes send & receive messages using Sockets



- Socket analogous to door
  - Sending process sends message out the door
  - Sending process relies on transport infrastructure on the other side of the door to deliver message to socket at the receiving process

Network Applications: Process Addressing

- Address (identifier) needed for processes to receive messages
  - IP Address (Unique 32-bit address)
  - Is IP address enough? ***No***
    - Processes running on the same host → IP address + port number

- Example
  - To send HTTP messages to `cs.sfu.ca` web server
    - **IP Address:** 142.58.102.68
    - **Port:** 80

Network Applications: Protocols

- Application Layer Protocol defines
  - Types of messages exchanged (e.g. request, response,…)
  - Message Syntax (fields, and field delineated)
  - Message Semantics (Meaning of fields)
  - Rules (When & how processes send & respond to messages)

- Protocols
  - Open
    - Examples: HTTP, SMTP
    - Refined in RFCs
    - Allows for interoperability
  - Proprietary
    - Examples: Skype

Network Applications: Needs

- Throughput
  - Minimum throughput guarantees
  - Make use of available throughput
- Data Integrity
  - Total reliability
  - Some loss acceptable
- Timing
  - Low delay

    Internet telephony

    interactive games
- Security
  - Encryption
  - Data Integrity

Application	*Data Loss*	*Throughput*	*Time-Sensitive*
File transfer	*no loss*	*elastic*	*no*
E-mail	*no loss*	*elastic*	*no*
Web documents	*no loss*	*elastic*	*no*
Text messaging	*no loss*	*elastic*	*yes and no*
Real-time audio/video	*loss-tolerant*	*audio: 5kbps-1Mbps video:10kbps-25Mbps*	*yes, 100's msec*
Interactive games	*loss-tolerant*	*few kbps up*	*yes, 100's msec*
Stored audio/video	*loss-tolerant*	*same as above*	*yes, few secs*

Network Applications: Transport Protocol Services

- Transport Layer services to Application Layer for delivery of messages
  - Transport Layer Protocols
    - **TCP**
      - Reliable Data Transfer
      - Flow Control
      - Congestion Control
      - **Does not provide** timing, throughput guarantee, and security
      - Connection-oriented
    - **UDP**
      - **Does not provide** reliable data transfer, flow control, congestion control, timing, throughput guarantee, security, or connection setup
      - Connection-less

Covered in detail in Transport Layer

Network Applications: Transport Protocol Services

Application	*Application layer protocol*	*Underlying transport protocol*
E-mail	*SMTP [RFC 2821]*	*TCP*
Remote terminal access	*Telnet [RFC 854]*	*TCP*
Web	*HTTP [RFC 2616]*	*TCP*
File transfer	*FTP [RFC 959]*	*TCP*
Streaming multimedia	*HTTP (e.g., YouTube), RTP [RFC 1889]*	*TCP or UDP*
Internet telephony	*SIP, RTP, proprietary (e.g., Skype)*	*TCP or UDP*

Web & HTTP

Web & HTTP

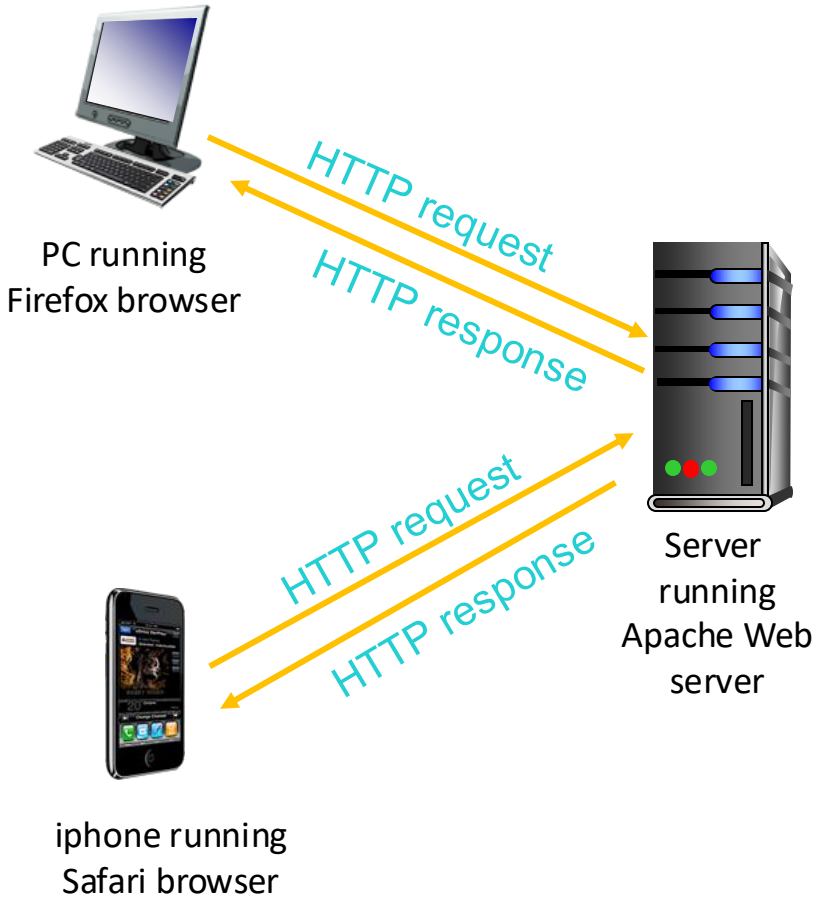
- Web pages consist of a base HTML (Hypertext Markup Language) file, including several references objects including
    - HTML file
    - JPEG image
    - Java Applet
    - Audio file
  - Each object **addressed** by a **URL** (Uniform Resource Locator)

- URL example: `www.sfu.ca/computing/research.html`

Host Name *Path Name*

HTTP

- Hypertext Transport Protocol (HTTP)
  - Web's application layer protocol
  - **Client-Server**
    - **Client:** Browser that requests, receives, and displays web objects
    - **Server**: Web server sends object in response to requests
  - **Stateless**: No information about past client requests
  - Uses **TCP**
    - Client initiates TCP connection to server port 80
    - Server accepts TCP connection
    - HTTP messages exchange
    - TCP connection close



HTTP Connections

- Non-Persistent HTTP
  - At most one object sent over TCP connection
  - Downloading multiple objects required multiple connections

- Persistent HTTP
  - Multiple objects can be sent over single TCP connection between client and server

Non-Persistent HTTP

- User enters URL:

`www.sfu.ca/computing/index.html`

1a. HTTP client initiates TCP connection to HTTP server (process) at www.sfu.ca on port 80

1b. HTTP server at host www.sfu.ca waiting for TCP connection at port 80 "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `www.sfu.ca/computing/index.html`

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

time

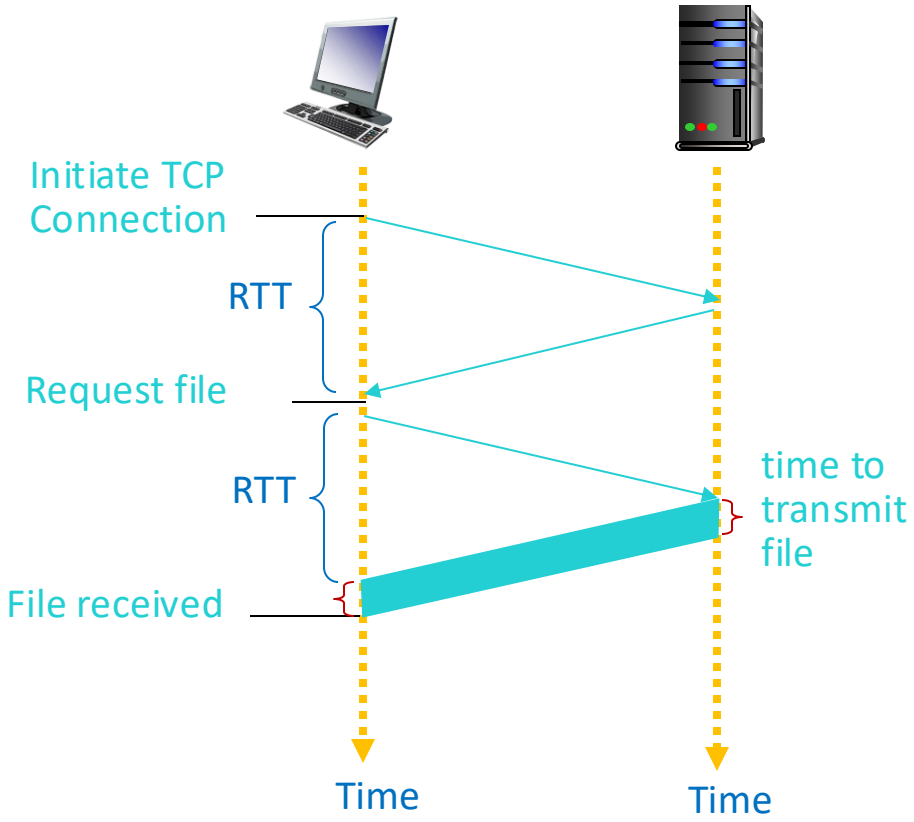
6. Steps 1-5 repeated for each of 10 jpeg objects

Non-Persistent HTTP

- **RTT:** Time for a small packet to travel from client to server & back
- **HTTP response time:**
  - One RTT to initiate TCP connection
  - One RTT for HTTP request and first few bytes of HTTP response to return
  - File transmission time

Non-persistent HTTP response time = 2RTT + file transmission time

- **Issue**
  - Two RTTs per object
  - OS Overhead for each TCP connection
  - Browsers often open parallel TCP connections to fetch referenced objects

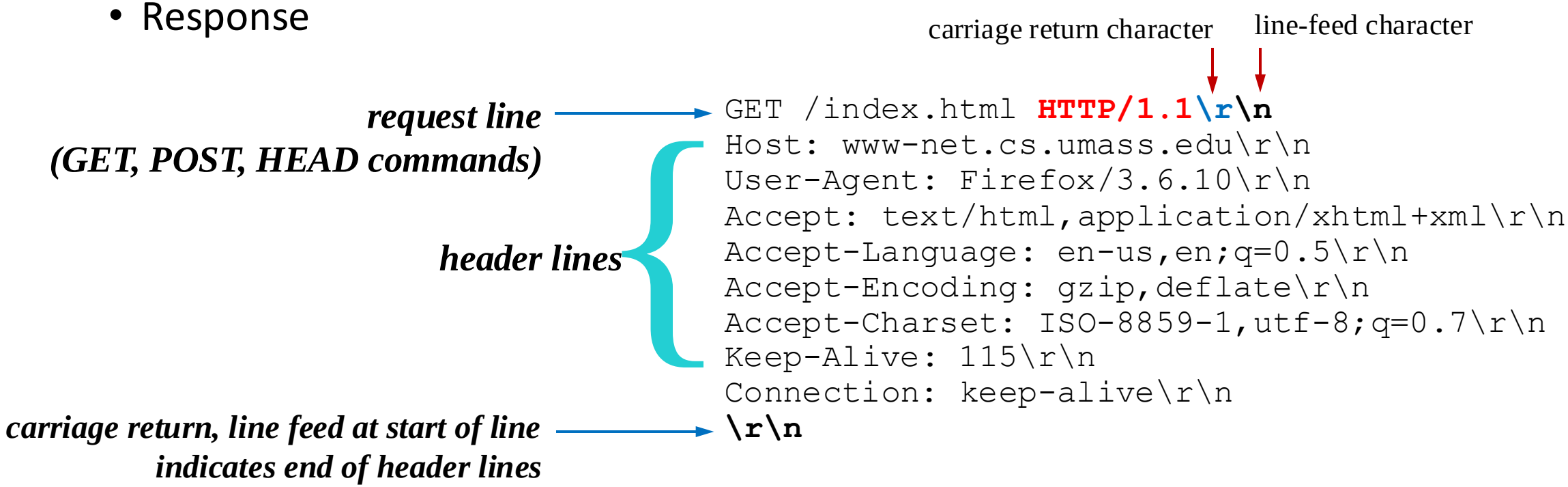


Persistent HTTP

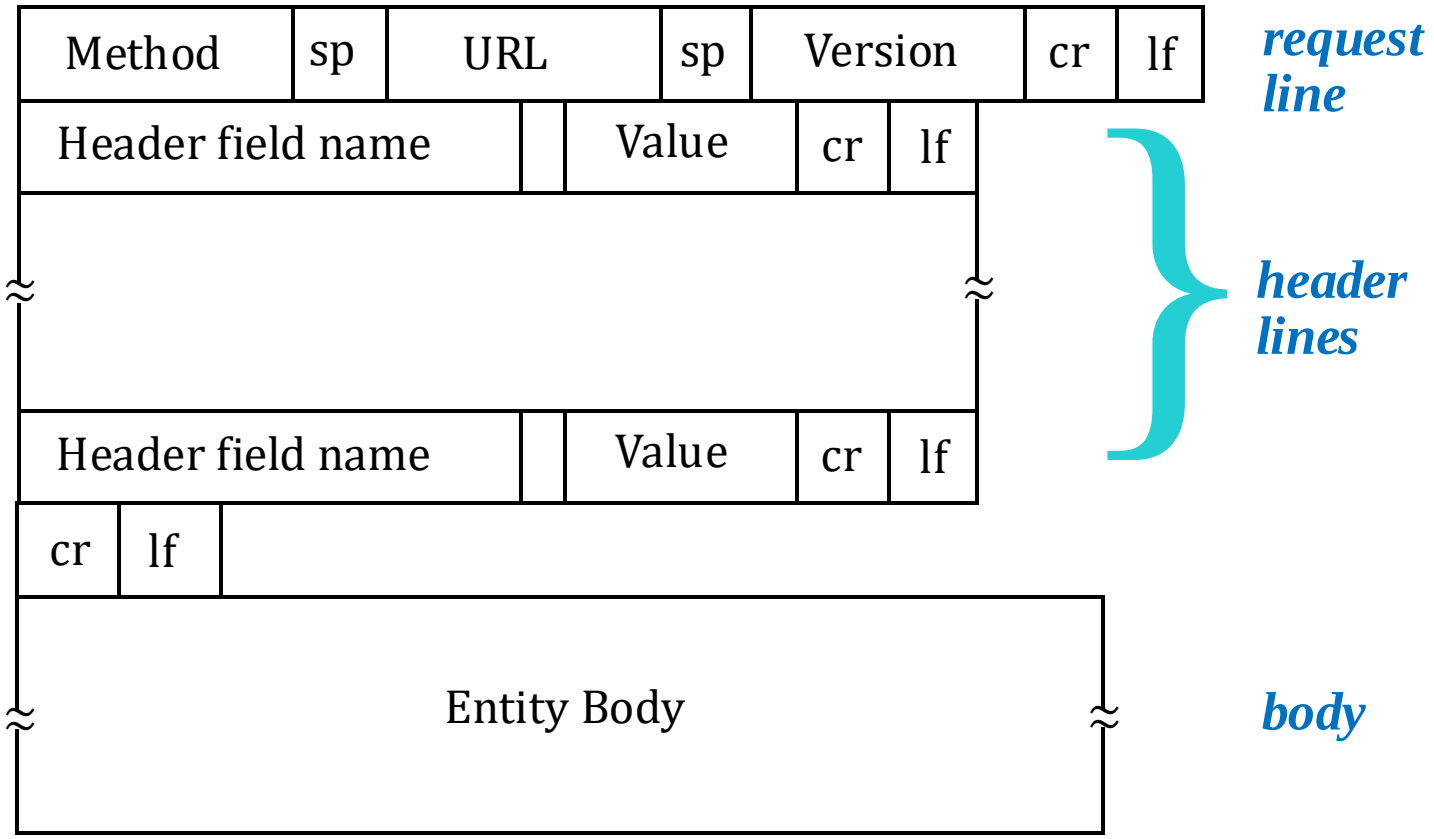
- Server leaves connection open after sending response
- Subsequent HTTP messages between same client server sent over open connection
- Client sends requests as soon as it encounters a referenced object
- As little as one RTT for all the referenced objects

HTTP Messages

- Two types of HTTP messages
  - **Request (ASCII, human-readable)**
  - Response



HTTP Request: General



Method Types

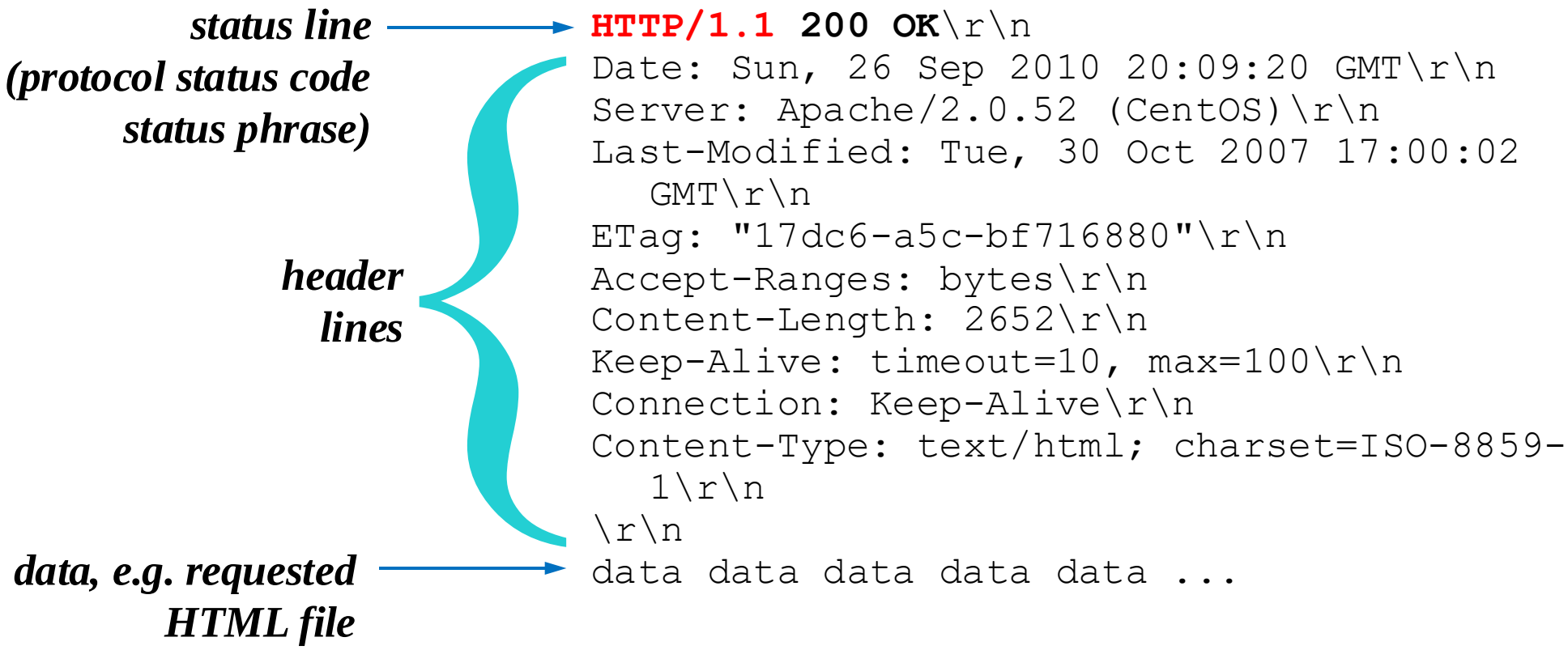
HTTP/1.1:

- GET
- POST
- HEAD
- PUT
  - Uploads file in entity body to path specified in URL field
- DELETE
  - Deletes file specified in the URL field

HTTP/1.0:

- GET
- POST
- HEAD
  - Asks server to leave requested object out of response
  - Used to get metadata about an object without transferring it

HTTP Response



HTTP Response

- **Status Code** appear in the first line in the server to client response message
- **Example Codes**

`200 OK` Request succeeded, requested object later in this message

`301 Moved Permanently`
Requested object moved, new location specified later in this message (Location:)

`400 Bad Request`
Request message not understood by server

`404 Not Found`
Requested document not found on this server

`505 HTTP Version Not Supported`

Trying HTTP (Client-Side)

1. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass.edu. Anything typed in will be sent to port 80 at gaia.cs.umass.edu

2. Type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

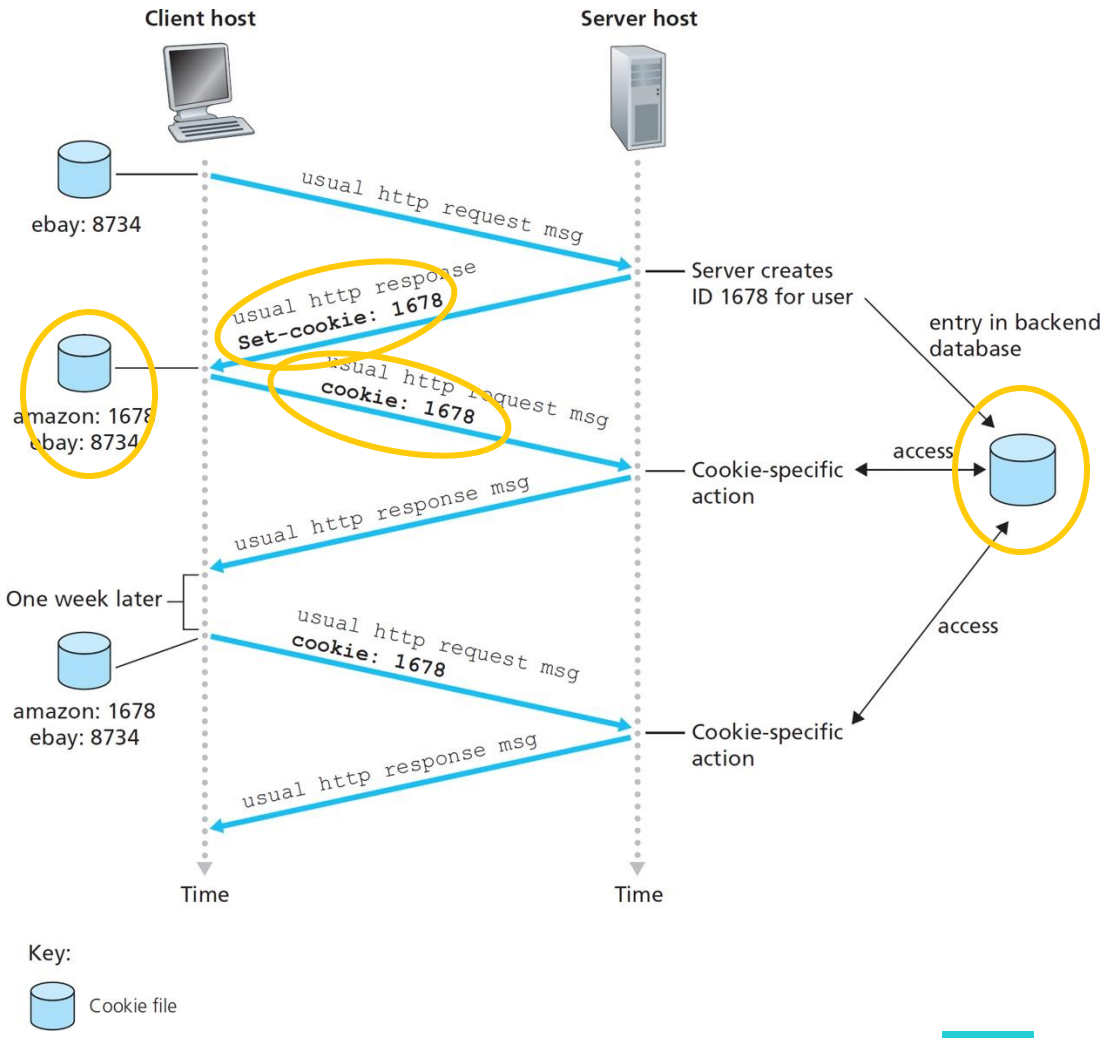
By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. Look at response message sent by HTTP server!

(Alternatively, use Wireshark to look at captured HTTP request/response)

HTTP: User-Server State (Cookies)

- **Cookies** have **four components:**
  - Cookie header line of HTTP **response** message
  - Cookie header line in **next** HTTP **request** message
  - Cookie file kept on user's host, managed by user's browser
  - Back-end database at Web site



HTTP State: Cookies

- How to keep **state**?
  - Protocol endpoints: Maintain state at sender/receiver over multiple transactions
  - Cookies: HTTP messages carry state

- Cookies can be used for
  - Authorization
  - Shopping carts
  - Recommendations
  - User sessions state (e.g. web email)

- Cookies permit sites to learn a lot about you – Privacy Concerns?!

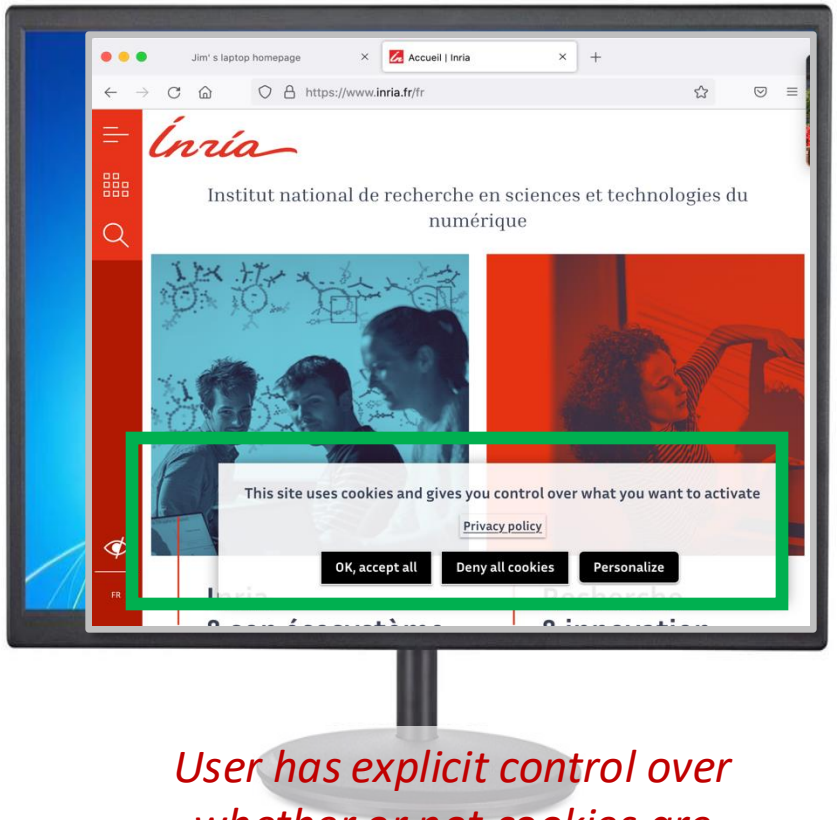
GDPR (EU General Data Protection Regulation) and cookies

"Natural persons may be associated with online identifiers [...] such as internet protocol addresses, cookie identifiers or other identifiers [...].

This may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them."

GDPR, recital 30 (May 2018)

When cookies can identify an individual, cookies are considered personal data, subject to GDPR personal data regulations.

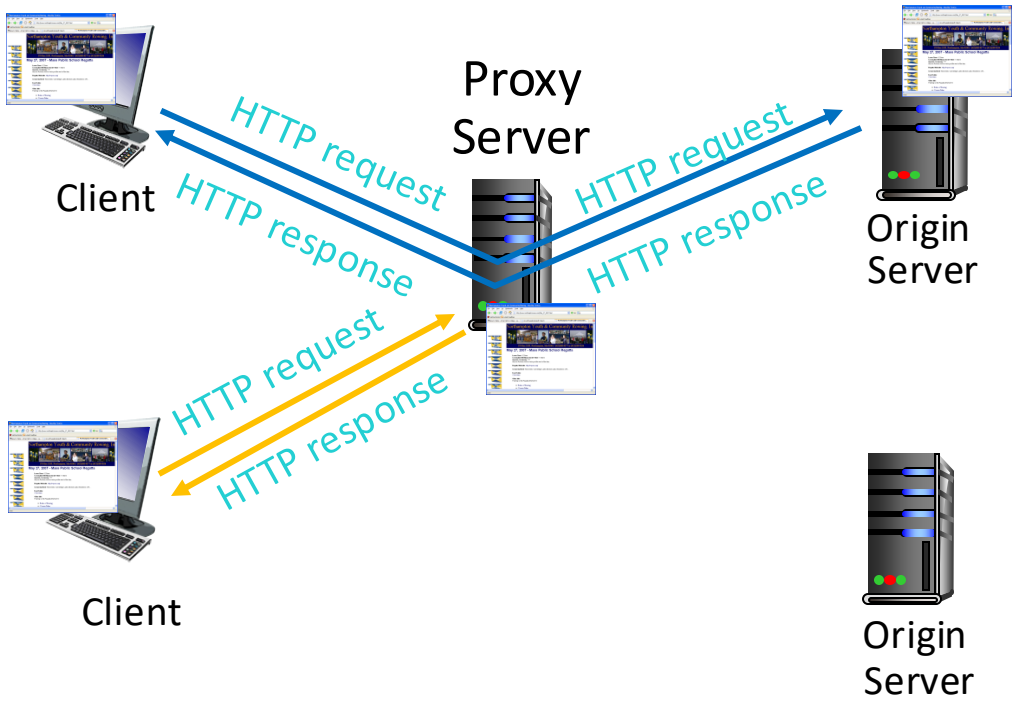


User has explicit control over whether or not cookies are allowed

Web Caches (Proxy Server)

Goal: Satisfy client request without involving origin server

- **User sets browser:** Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: Cache returns object
  - Otherwise cache requests object from origin server, then returns object to client



- Typically cache is installed by ISP (University, Company, Residential ISP)

Web Caching

- Cache acts as both client and server
  - Server for original requesting client
  - Client to origin server

Why Web caching?

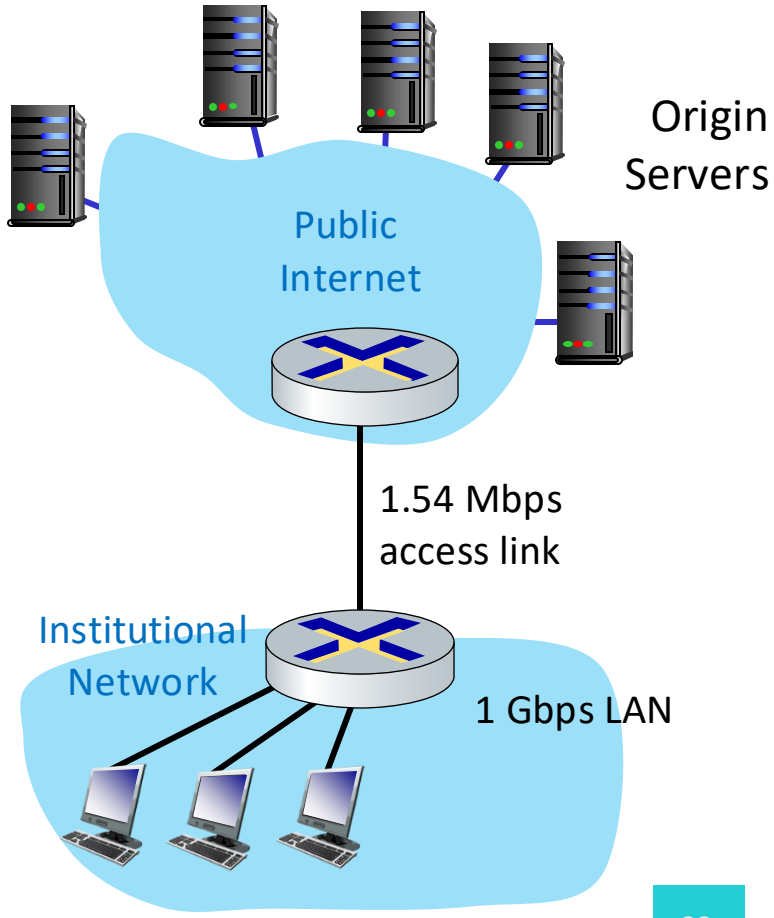
- Reduce response time for client request
- Reduce traffic on an institution's access link
- Enables content providers to cost-effectively deliver content (same for P2P file sharing)

Web Caching Example

Assumptions:
- Access link rate: **1.54 Mbps**
- Average object size: **100K bits**
- Average request rate from browsers to origin servers: **15/sec**
- RTT from institutional router to any origin server: **2 sec**

Consequences:
- Average data rate to browsers: **1.50 Mbps**
  - LAN utilization: 0.0015 ✓
  - Access link utilization: 0.97 → Problem
- Total delay = Internet delay + access delay + LAN delay

  = 2 sec + minutes + uSecs



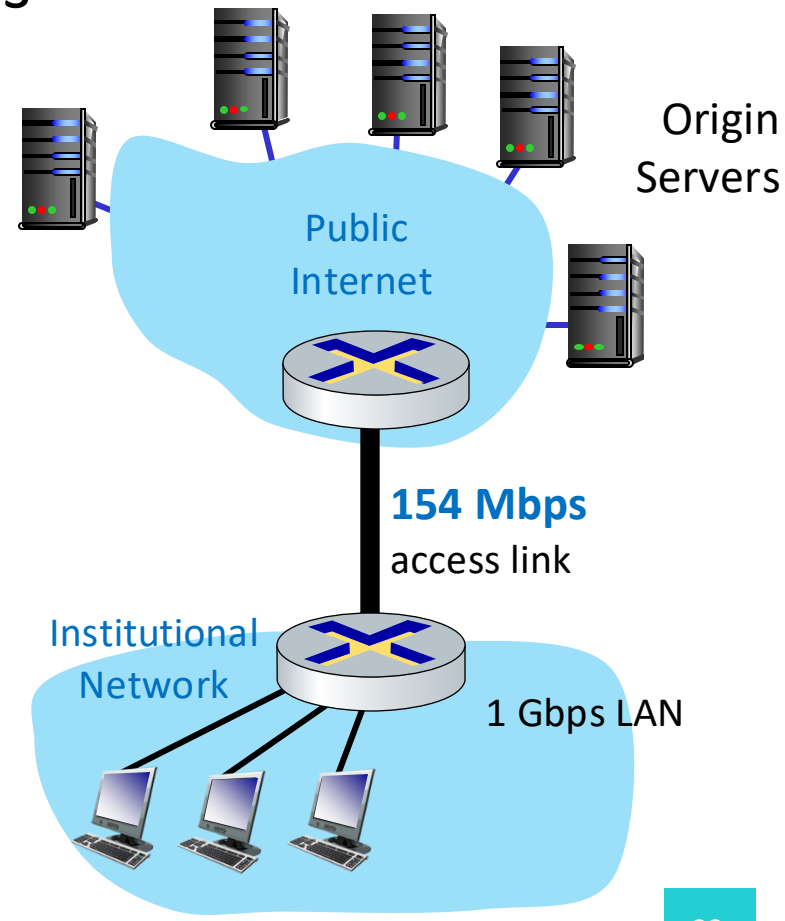
Web Caching Example

Solution (1): Access link rate change to 154Mbps

Consequences:

- LAN utilization: 0.0015
- Access link utilization: 0.0097
- Total delay = Internet delay + access delay + LAN delay

= 2 sec + ~~minutes~~ + usecs
msecs



Cost: Increased access link speed is not cheap!

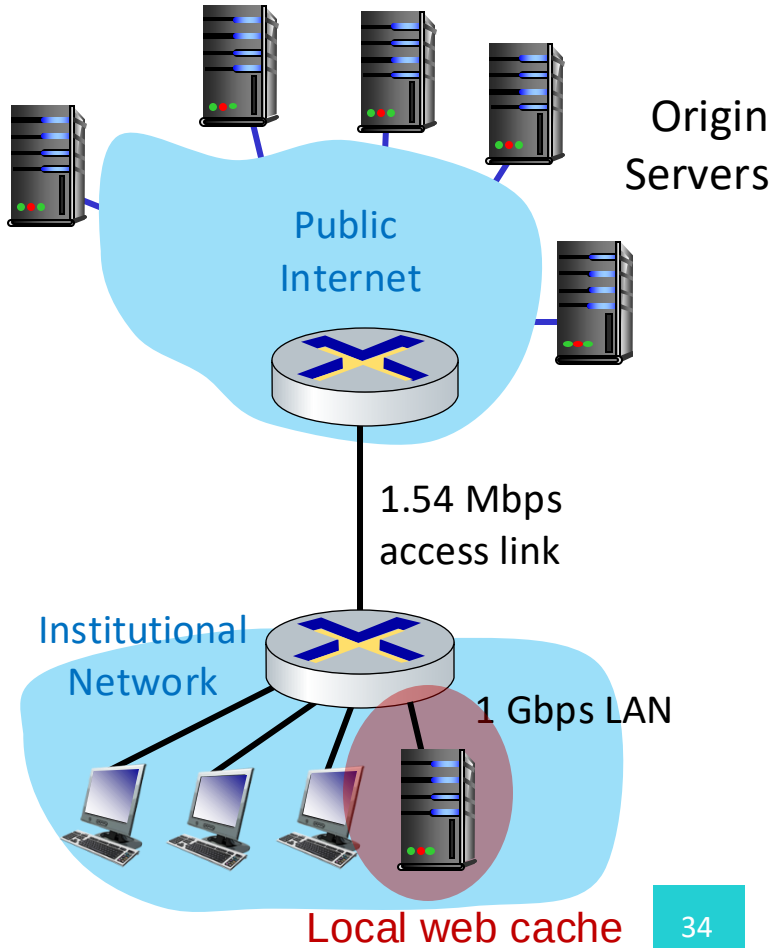
Web Caching Example

Solution (2) : Install local cache

Consequences

- Suppose cache hit rate is 0.4
  40% requests satisfied at cache
  60% requests satisfied at origin
- Access link utilization: 60% of requests use access link
  = 0.6 x 1.5 / 1.54 =0.58
- Data rate to browsers over access link
  Total delay = 0.6 (2.0) + 0.4 (~msecs) = ~ 1.2 secs

Cost: Web cache is not expensive!

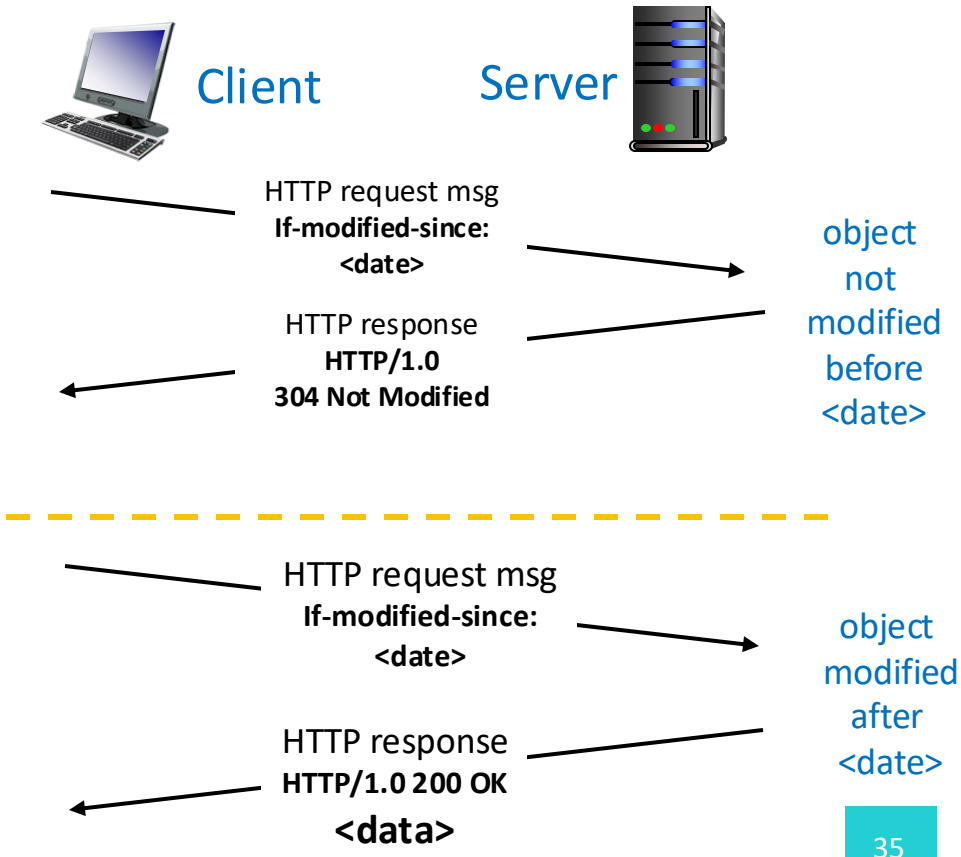


Conditional Get

*Goal: Don't send object if cache has up-to-date **cached** version*

- **Cache:** Specify date of cached copy in HTTP request
  If-modified-since: <date>

- **Server:** Response contains no object if cached copy is up-to-date:
  HTTP/1.0 304 Not Modified



HTTP/2

Goal: *Decreased delay in multi-object HTTP requests*

HTTP1.0 [RFC 1945, 1996]: Uses non-persistent HTTP

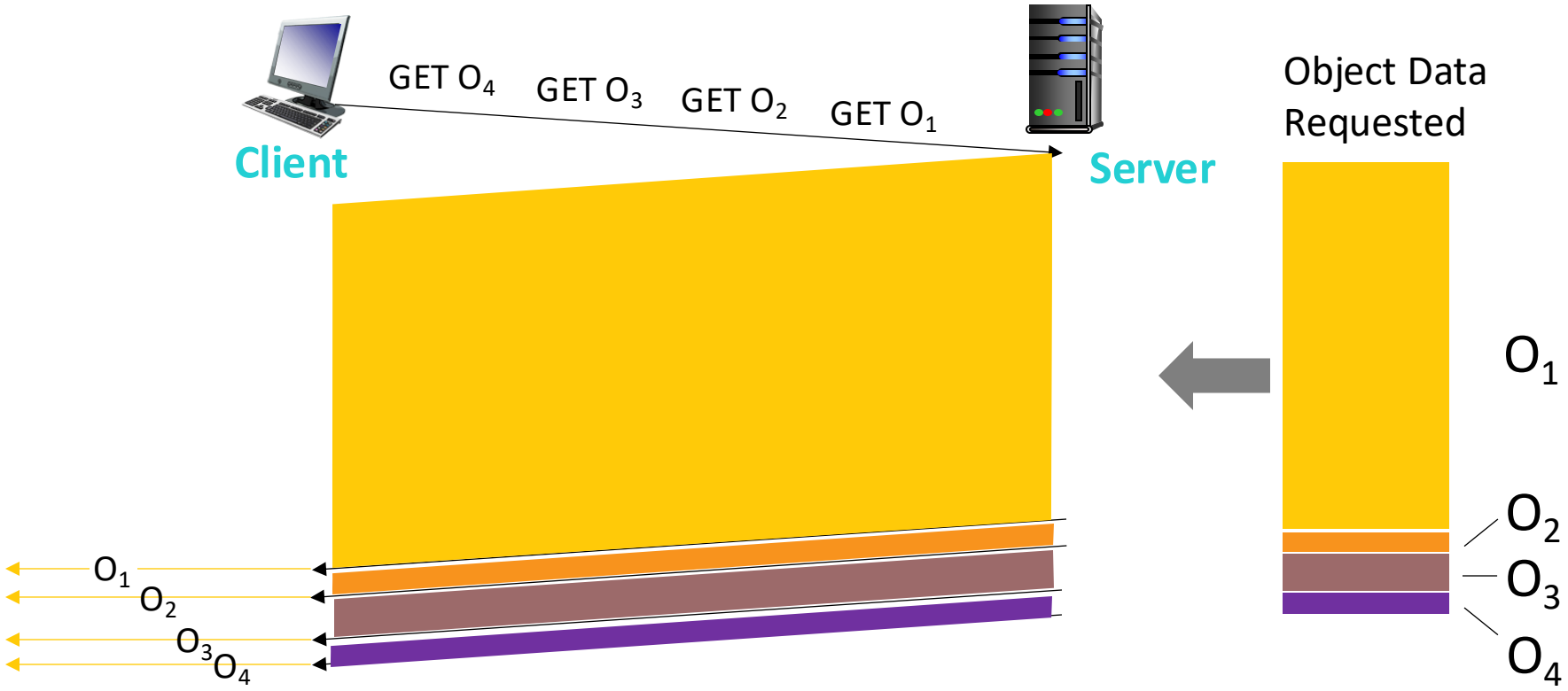
HTTP1.1 [RFC 2616, 1997]: Introduced multiple, pipelined GETs over **single TCP connection**

- Server responds First-Come-First-Served (FCFS) to GET requests
- With FCFS, small object may have to wait for transmission behind large object(s)
  This is called Head-Of-Line (HOL) Blocking
- Retransmitting lost TCP segments stalls object transmission

HTTP/2 [RFC 7540, 2015]: Increased flexibility at *server* in sending objects to client:

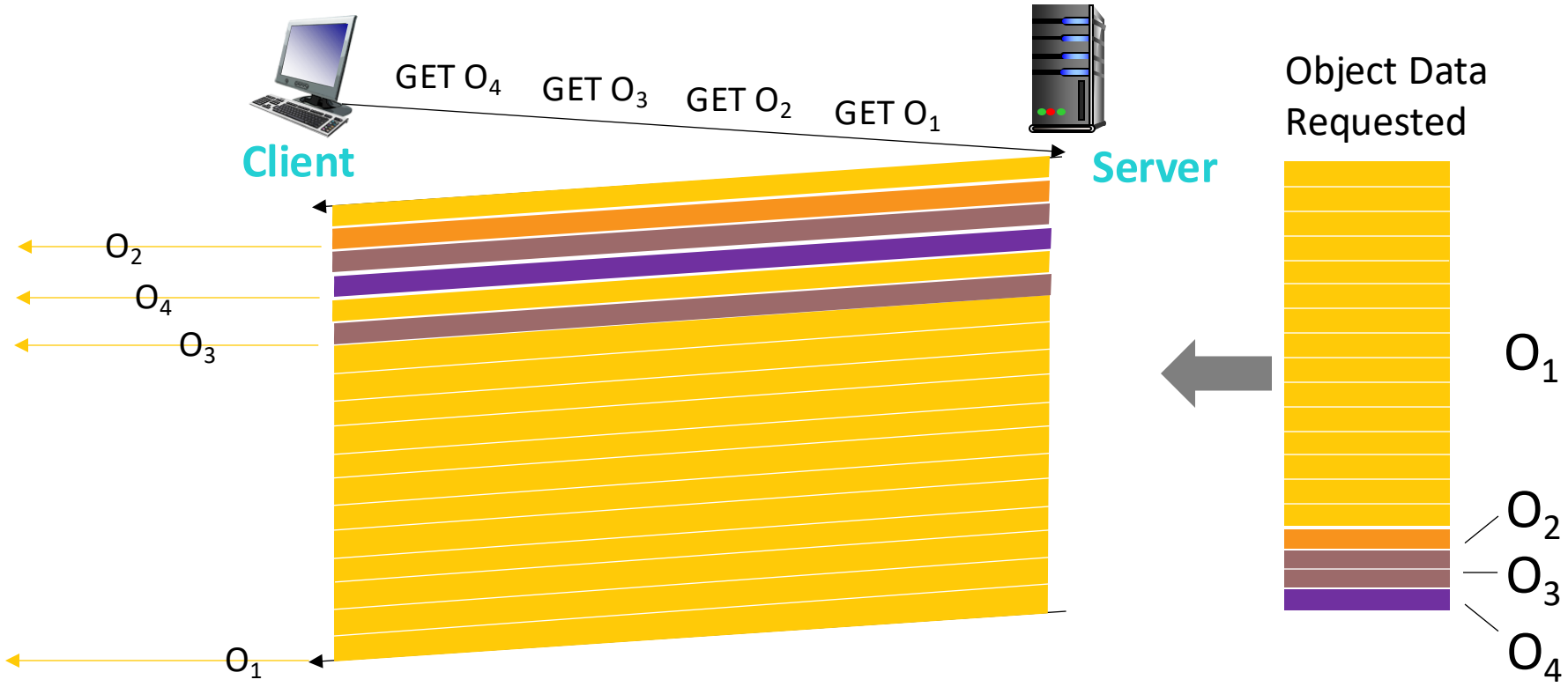
- Methods, status codes, most header fields unchanged from HTTP 1.1
- Transmission order of requested objects based on client-specified priority (not necessarily FCFS)
- Push unrequested objects to client
- Divide objects into frames, schedule frames to mitigate HOL blocking

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 & congestion-control (more pipelining) over UDP
- HTTP & **QUIC**

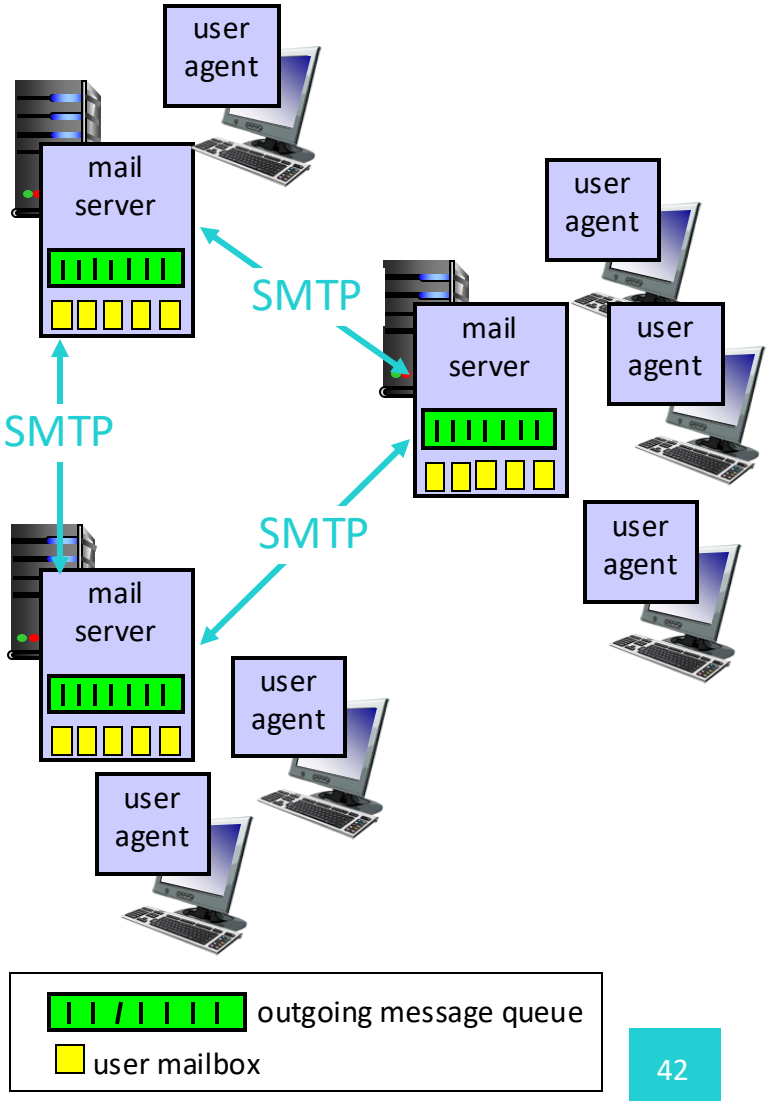
Web & HTTP Summary

- Messages: Request, Response
  - Methods: GET, POST, HEAD, PUT, DELETE
  - Status Codes and Phrases
- Persistent versus Non-Persistent Connection
- Cookies and State Information
- Caching and Proxy Servers
- Head Of Line Blocking and HTTP/2
- HTTP/2 to HTTP/3

Email

Electronic Mail

- Major components
  - User agents
    - Reading, composing, editing mail messages
    - Mail client examples: Outlook, Thunderbird, iPhone mail client
    - Outgoing and incoming messages stored on server
  - Mail servers
    - Mailbox: Contains incoming mails for user
    - Mail Queue: Contains outgoing (to be sent) mails
  - Simple Mail Transfer Protocol: SMTP
    - Client: Sending mail server
    - Server: Receiving mail server

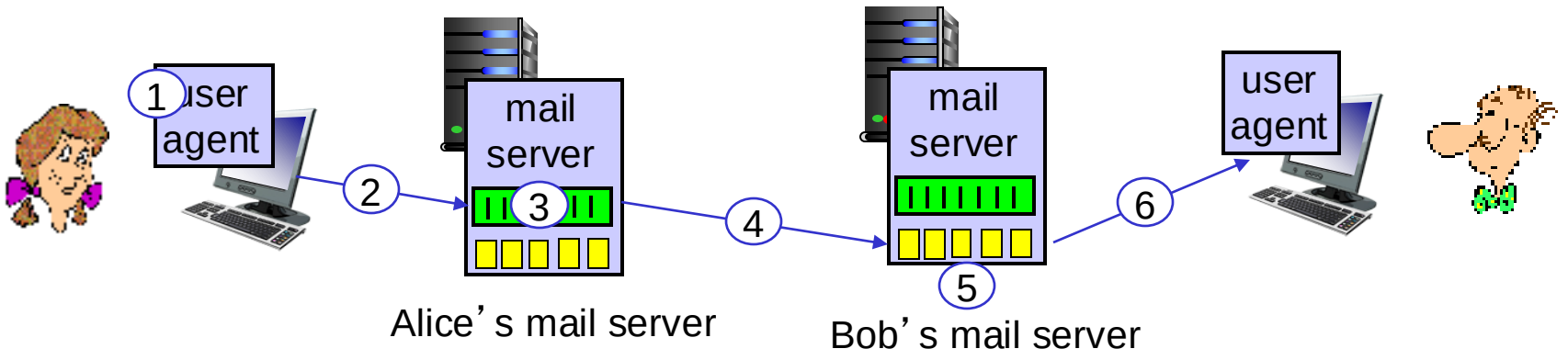


Electronic Mail: SMTP (RFC 5321)

- Uses TCP to reliably transfer email message from client to server (port 25)
- Direct transfer: Sending server to receiving server
- Three phases of transfer
  - Handshaking (greeting)
  - Transfer of messages
  - Closure
- Command/response interaction (like HTTP)
  - **Commands:** ASCII text
  - **Response:** Status code and phrase
- Messages must be in 7-bit ASCII

Electronic Mail

1. Alice uses UA to compose message to bob@someschool.edu
2. Alice's UA sends message to her mail server. message placed in message queue.
3. Client side of SMTP 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



Sample SMTP Interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

Try SMTP

- **`telnet servername 25`**
- See 220 reply from server
- Enter `HELO`, `MAIL FROM`, `RCPT TO`, `DATA`, `QUIT` commands

  above lets you send email without using email client (reader)

Set up and test that for an exchange server:
https://learn.microsoft.com/en-us/exchange/mail-flow/test-smtp-telnet

SMTP

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

Comparison with HTTP:

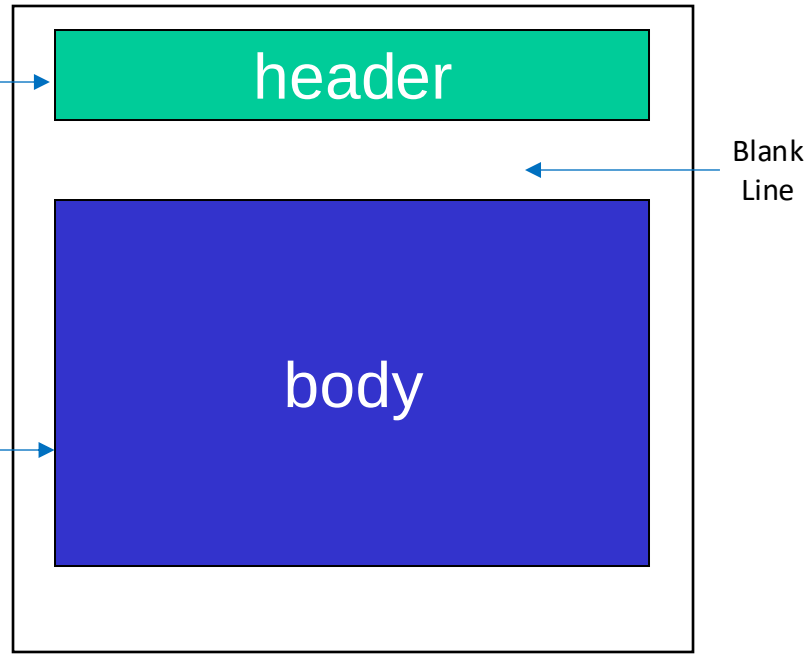
SMTP	*HTTP*
Push	*Pull*
Multiple Objects sent in multipart message	*Each object encapsulated in its own response message*
Both have ASCII command/response interaction & status codes	

Mail Message Format

SMTP [RFC 5321]: Protocol for exchanging e-mail messages

RFC 5322 defines syntax for e-mail message itself

- Header lines, e.g.,
  - To:
  - From:
  - Subject:
  - These lines, within the body of the email message area, are different from `SMTP MAIL FROM:`, `RCPT TO:` commands
- Body: The "message" in ASCII characters only

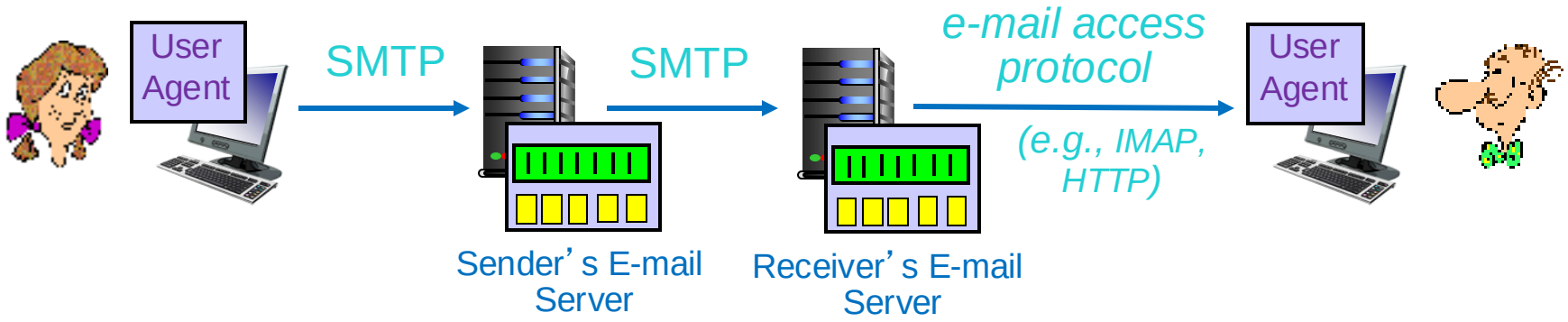


Mail Access Protocols

SMTP: Delivery/storage to receiver's server

Mail access protocol: Retrieval from server

- **POP:** Post Office Protocol [RFC 1939]: Authorization, download
- **IMAP:** Internet Mail Access Protocol [RFC 3501]: More features, including manipulation of stored messages on server
- **HTTP:** Gmail, Hotmail, Yahoo! Mail, etc.



POP3

Authorization phase

- Client commands
  - User: Declare username
  - Pass: Password
- Server Responses
  - OK
  - ERR

Transaction Phase

- Client
  - list: list message numbers
  - retr: retrieve messages by number
  - dele: delete
  - quit

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 & IMAP

POP3 (RFC 1939)	*IMAP (RFC 3501)*
Download & delete mode: *User cannot re-read e-mail if he changes client* *Download & keep mode:* *Copies of messages on different clients*	*Keeps all messages in one place: at server*
No mail folder organization	*Allows user to organize messages in folders*
POP3 is stateless across sessions	*Keeps user state across sessions:* *Names of folders* *Mappings between message IDs & folder name*
Both Mail Access Protocols	

Mail Security Issues

- Basic electronic mail is not secure
  - No encryption (Plain Text)
    - Eavesdropping and interception is easy
  - Spamming
  - Source of Malware Delivery
    - Viruses, worms, Trojan horses, documents with destructive macros, and malicious code
  - No Source Verification
    - Spoofing sender address

Learn More:

Certificates & Digital Signatures
PGP: Pretty Good Privacy
SPF: Sender Policy Framework
DKIM: DomainKeys Identified Mail
DMARC: Domain-based Message Authentication Reporting and Conformance

DNS

DNS

Internet hosts and routers:

- IP address used for addressing datagrams
- Name: e.g., www.yahoo.com

Analogy: Name and ID number for people

Question: How to map between the IP address and name?

Domain Name System:

- **Distributed database** implemented in hierarchy of many **name servers**
- **Application-layer protocol:** Hosts, name servers communicate to **resolve** names (address/name translation)
- Core Internet function, implemented as application-layer protocol
- Complexity at network's edge

DNS Services

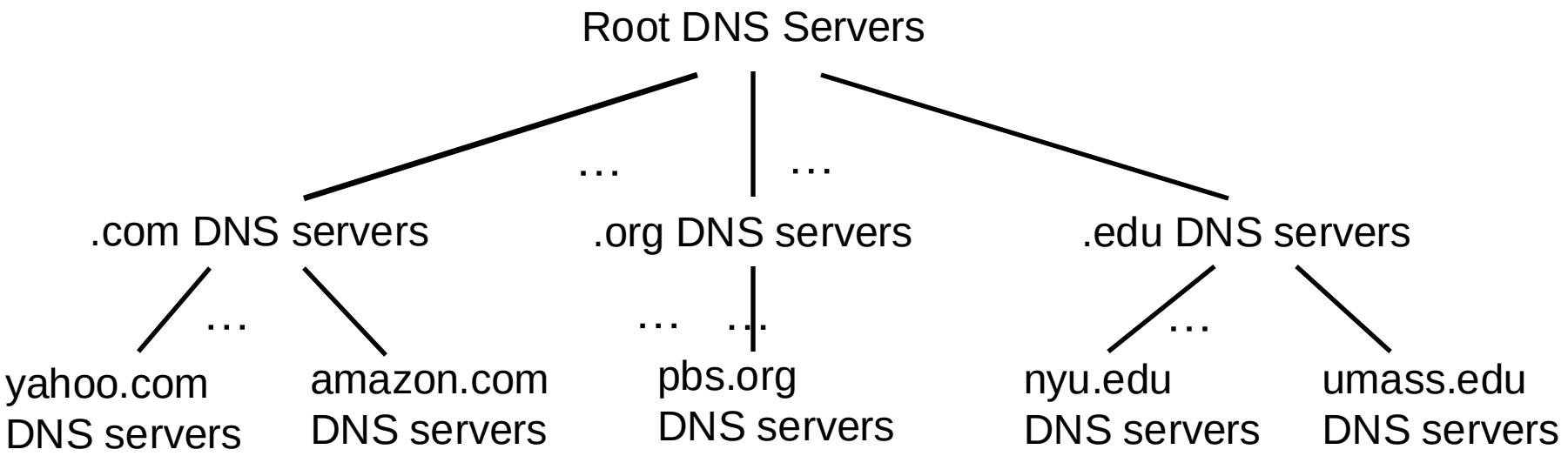
- Hostname to IP address translation
- Host aliasing (canonical, alias names)
- Mail server aliasing
- Load distribution (Replicated Web Servers: many IP addresses correspond to one name)

DNS: Why Not Centralized?

- Single point of failure
- Traffic volume
- Distant centralized database
- Maintenance

Does not Scale!

DNS: A Distributed Hierarchical Database



Root

Top Level Domain

Authoritative

DNS

Local DNS Name Server

- Does not strictly belong to hierarchy

- Each ISP (residential ISP, company, university) has one
  - also called **default name server**

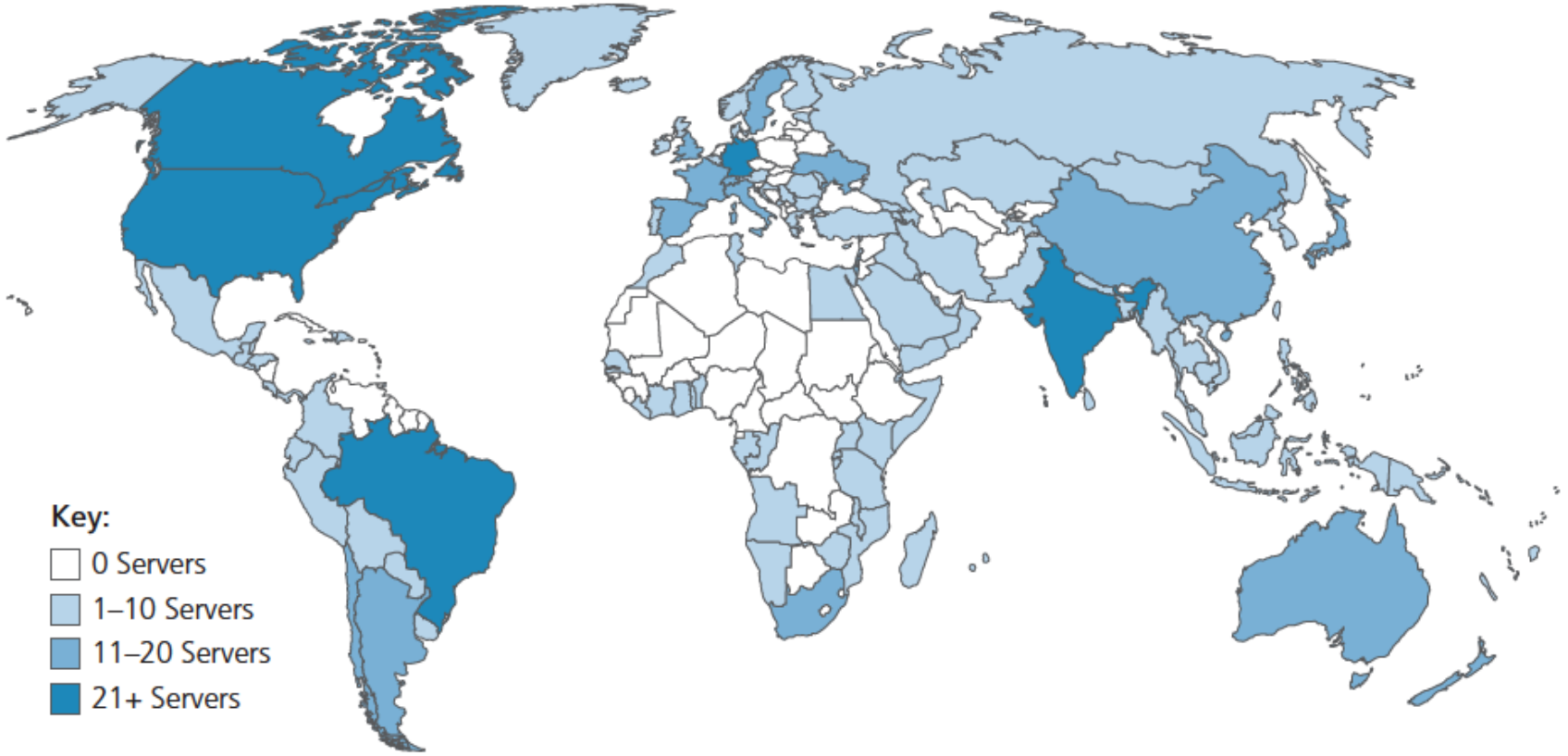
- When host makes DNS query, query is sent to its local DNS server
  - Has local cache of recent name-to-address translation pairs (but may be out of date!)
  - Acts as proxy & forwards query into hierarchy

DNS

Root Name Servers

- Contacted by local name server that can not resolve name
- Root name server:
  - Contacts authoritative name server if name mapping not known
  - Gets mapping
  - Returns mapping to local name server

DNS



13 logical root name servers worldwide each server replicated many times

DNS

Top-level domain (TLD) servers

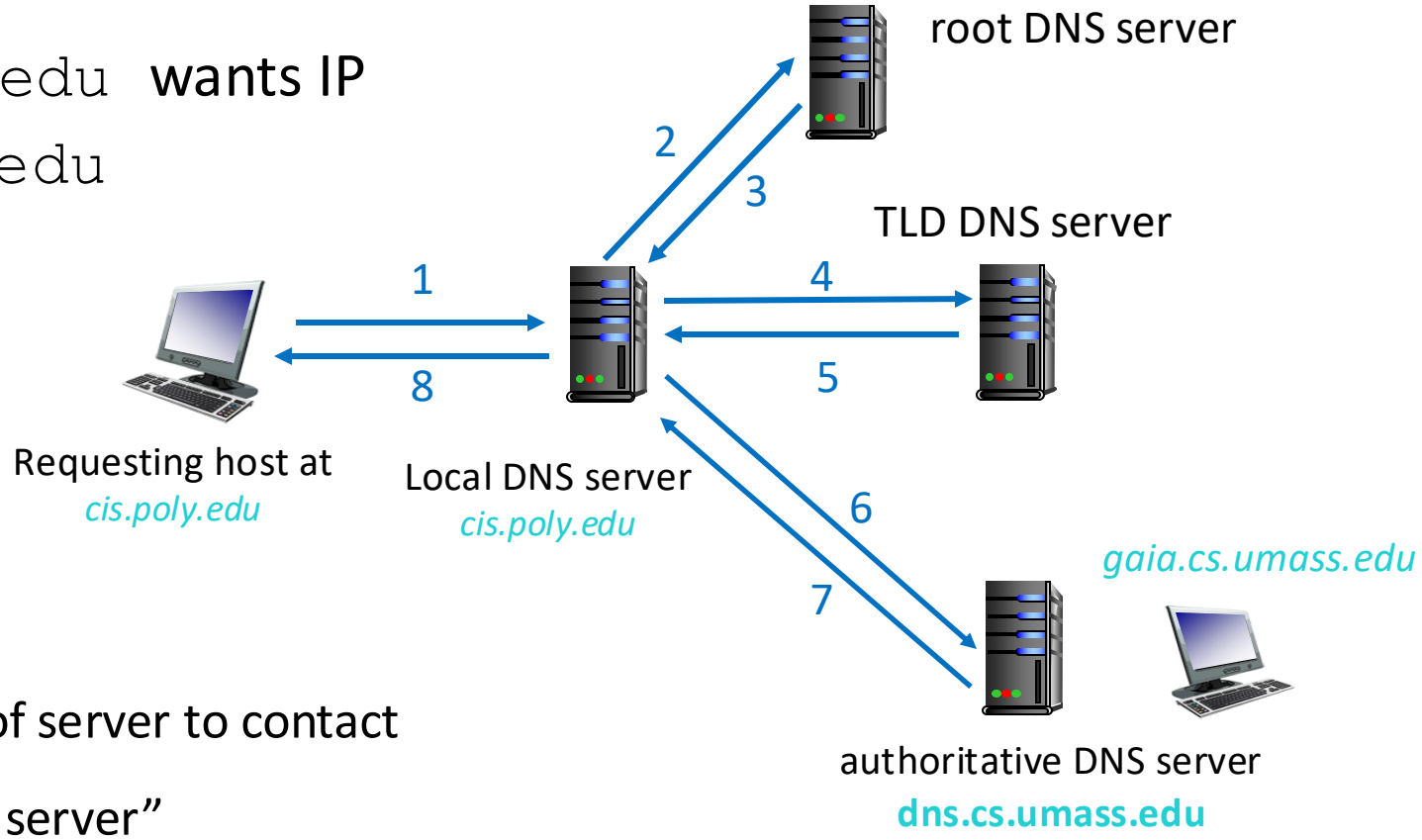
- Responsible for `com`, `org`, `net`, `edu`, `aero` and all top-level country domains like `uk`, `fr`, `ca`

Authoritative DNS servers

- Providing authoritative hostname to IP mappings for named hosts
- Can be maintained by organization or service provider

DNS: Name Resolution

Example: Host at `cis.poly.edu` wants IP address for `gaia.cs.umass.edu`

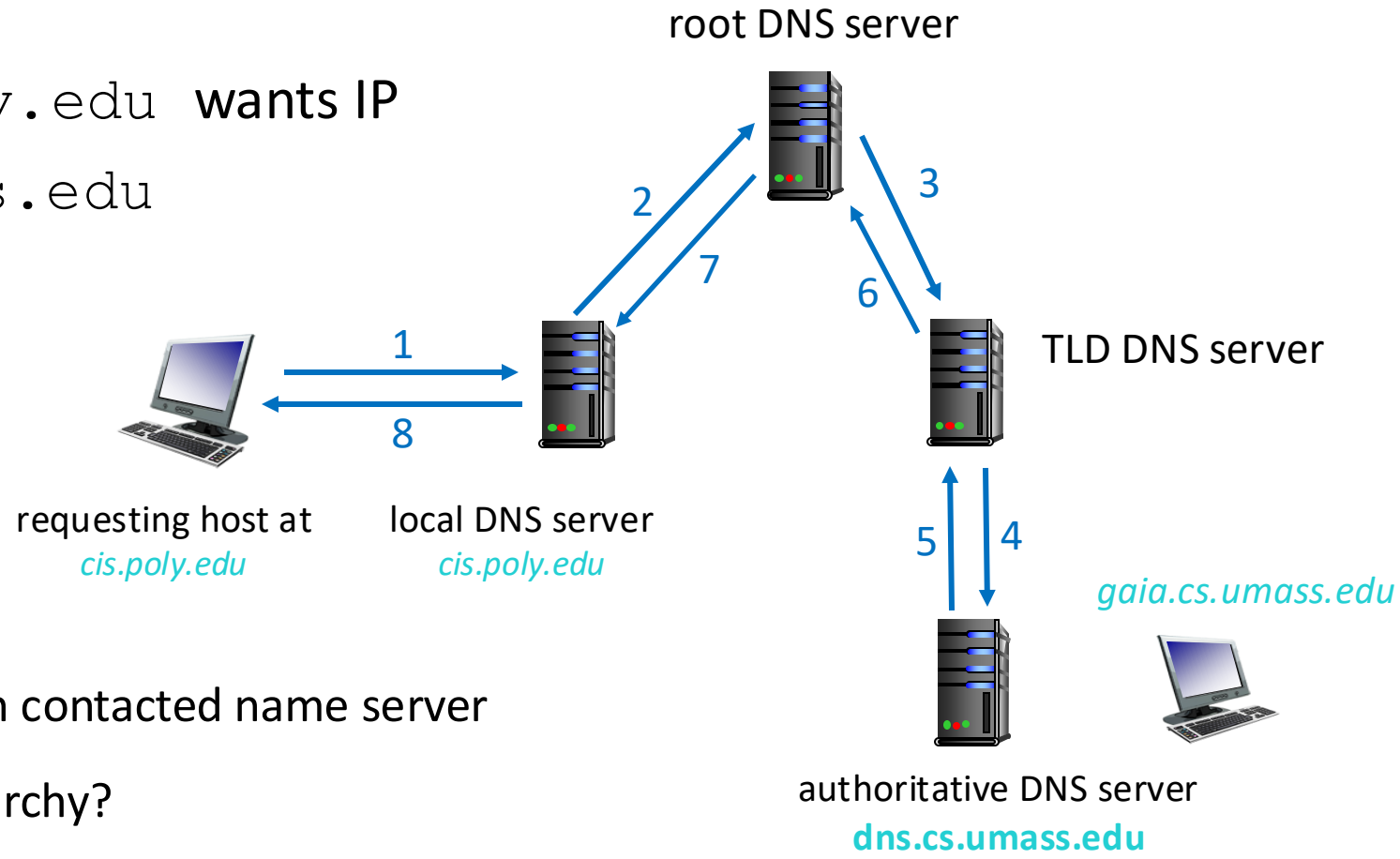


Iterated query

- Contacted server replies with name of server to contact
- "I don't know this name, but ask this server"

DNS: Name Resolution

Example: Host at `cis.poly.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?

DNS: Caching & Updating Records

- Once (any) name server learns mapping, it **caches** mapping
  - Cache entries timeout (disappear) after some time (TTL)
  - 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, may not be known Internet-wide until all TTLs expire

- Update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS Records

- **DNS:** Distributed database storing resource records **(RR)**

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

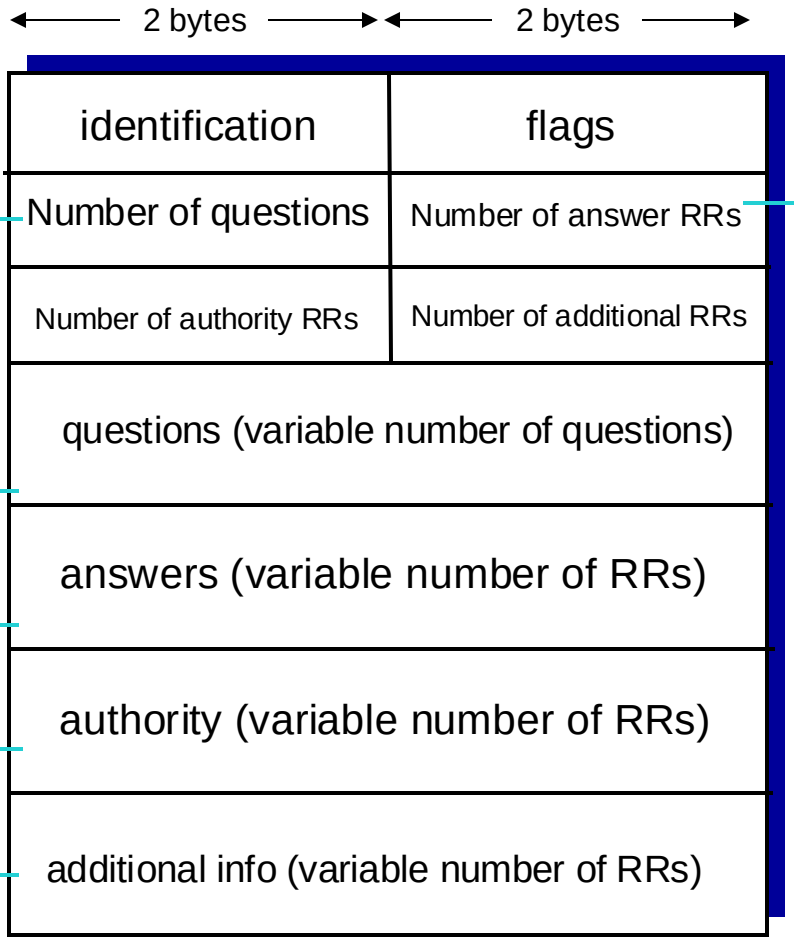
type=A : `name` is hostname
`value` is IP address

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
E.g. `www.ibm.com` is really `servereast.backup2.ibm.com`
`value` is canonical name

type=MX
`value` is the name of mailserver associated with **`name`**

DNS Messages



Identification: 16 bit # for query, reply to query uses same #

Name, type fields for a query

RRs in response to query

Records for authoritative servers

Additional helpful info that maybe used

Flags:

- Query for reply
- Recursion desired
- Recursion available
- Reply is authoritative

Query and **reply** messages both have the same **message format**

Inserting Records into DNS

Example: New startup Network Utopia

- Owner
  - Create authoritative server type A record for server and type MX record for mail server
  - Register name networkuptopia.com at **DNS registrar**
    - Owner provides names and IP addresses of authoritative name servers (primary and secondary)

- DNS **registrar** (list available at `internic.net`)
  - Inserts NS and A type RRs into .com **TLD server**:

```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)

(networkutopia.com, dns2.networkutopia.com, NS)
(dns2.networkutopia.com, 212.212.212.2, A)
```

Attacking DNS

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

Exploit DNS for DDoS (DNS Amplification)

- Send queries with spoofed source address: target IP
- Requires amplification

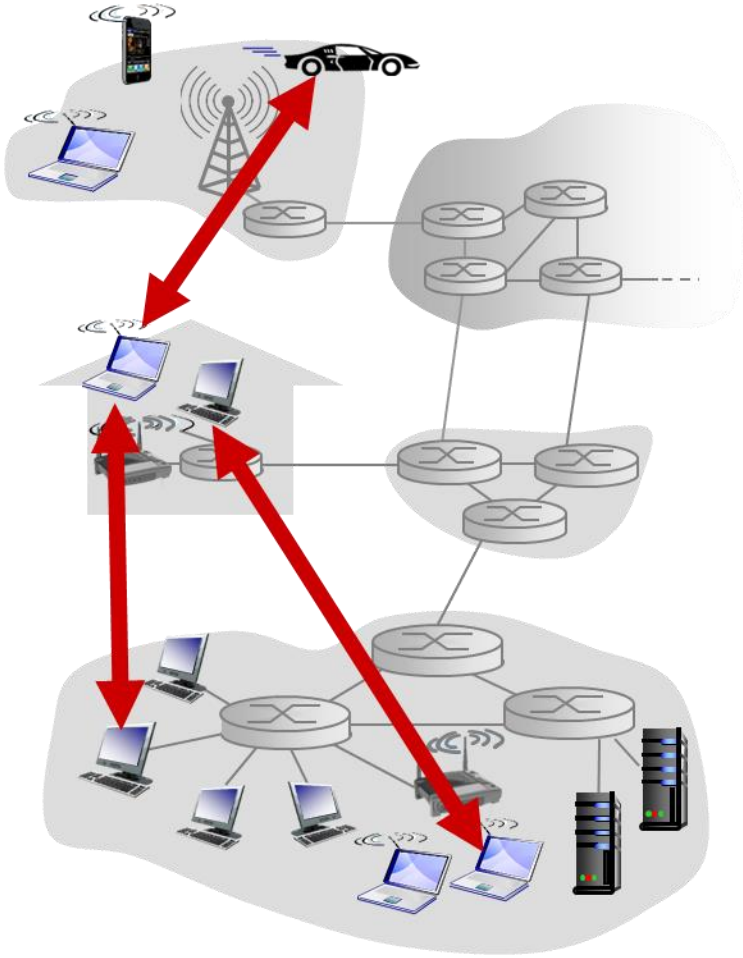
Redirect or Hijacking attacks

- Man-in-middle: Intercept queries
- DNS poisoning: Send bogus relies to DNS server, which caches

P2P

Pure P2P Architecture

- **No** always-on server
- Arbitrary end systems **directly** communicate
- Peers are **intermittently connected** and **change IP addresses**

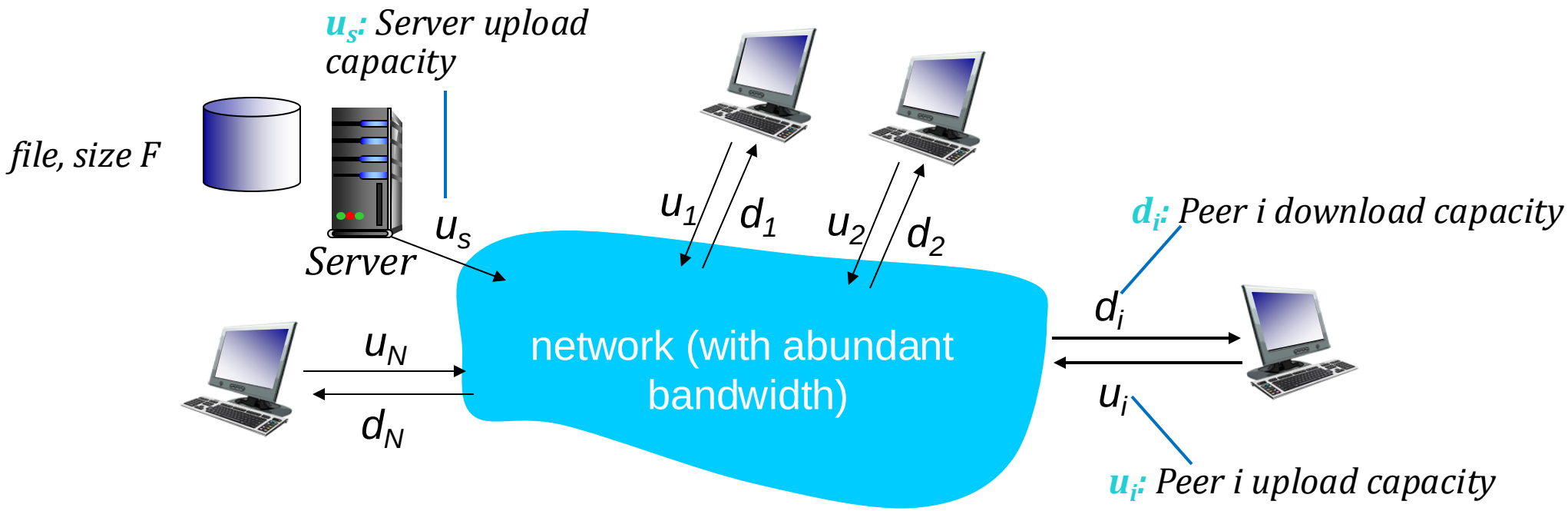


Examples:

- File distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

File Distribution

Question: How much time to distribute file (size *F*) from one server to *N* **receivers**?



File Distribution: Client-Server

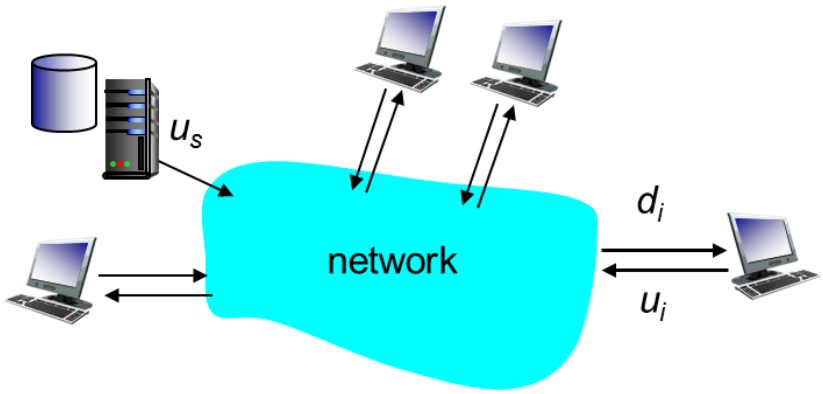
- **Server transmission:** Must sequentially send (upload) *N* file copies
  - Time to send one copy: F/u_s
  - Time to send *N* copies: $N \times F/u_s$

- **Client:** Each client must download file copy
  - Client download time: F/d_i
  - d_{min} = Min client download rate
  - Min client download time: F/ d_{min}

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

$$D_{CS} \geq max\{NF/u_s, F/d_{min}\}$$

*Increases linearly in **N***



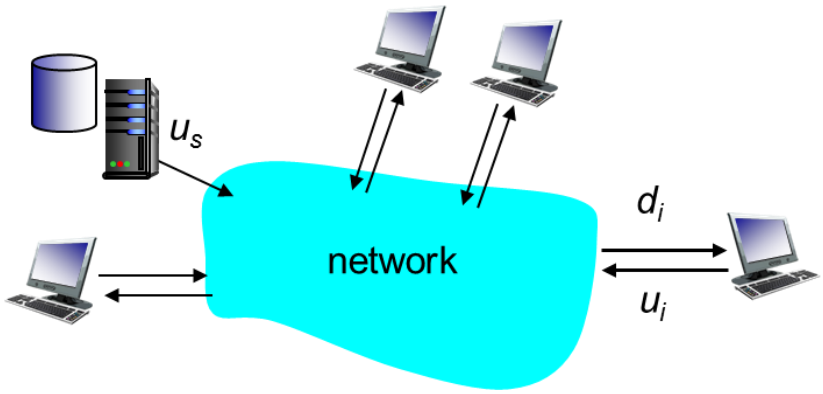
File Distribution: P2P

- **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 $u_s + \sum u_i$

Time to distribute F to N clients using P2P approach

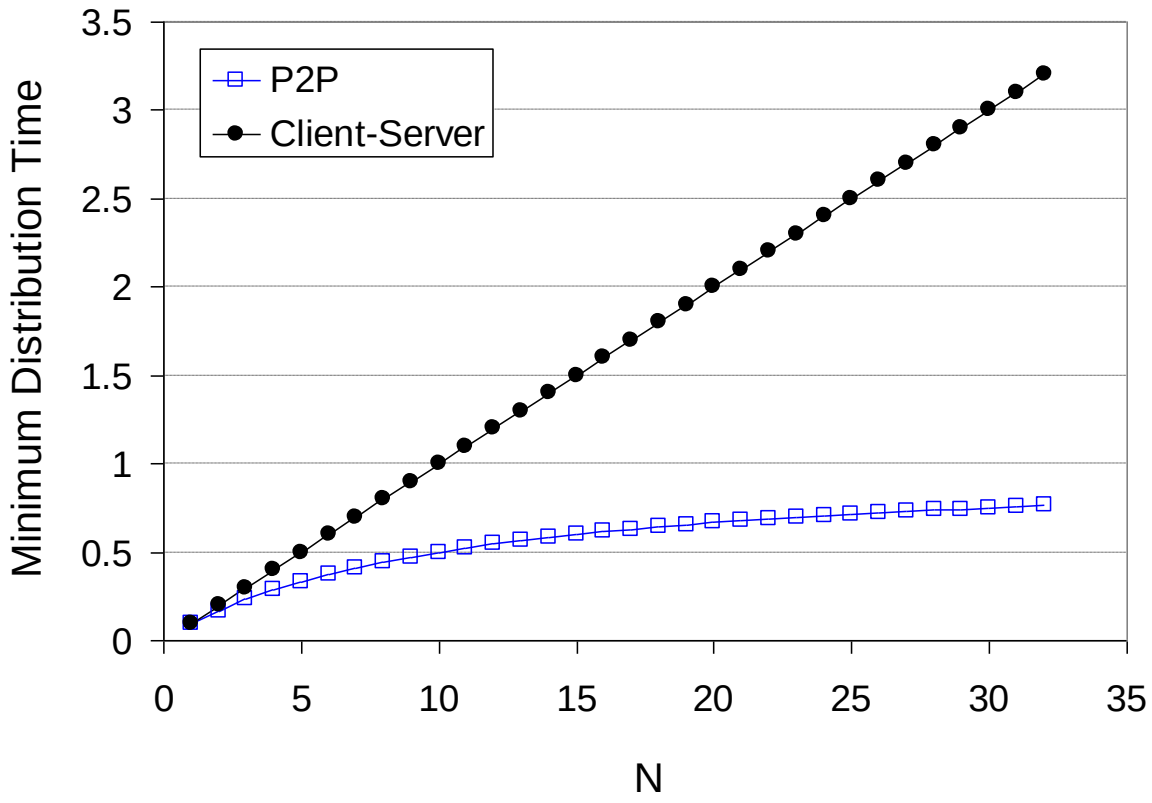
$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

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, $u_s = 10u$, $d_{min} \geq u_s$



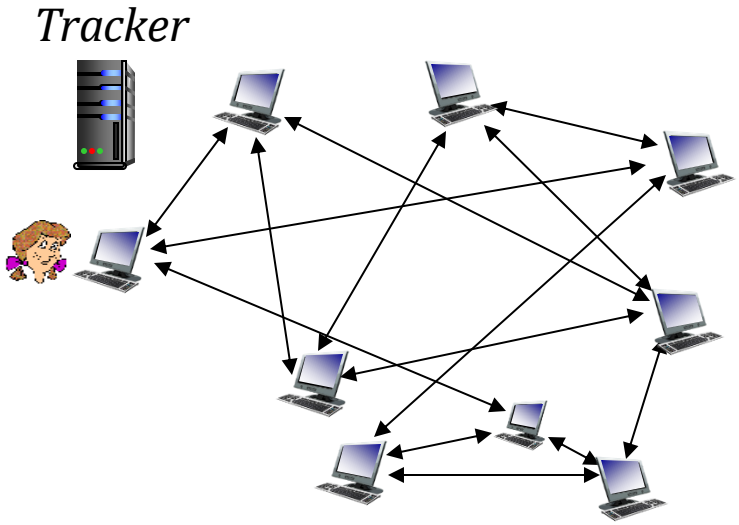
BitTorrent

- File divided into 256KB chunks
  - Peers in torrent send & receive file chunks

- Peer joining torrent
  - Has no chunks, but will accumulate them over time from other peers
  - Connects to subset of peers (neighbors)
  - Registers with **tracker** to get list of peers

- 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 File 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**

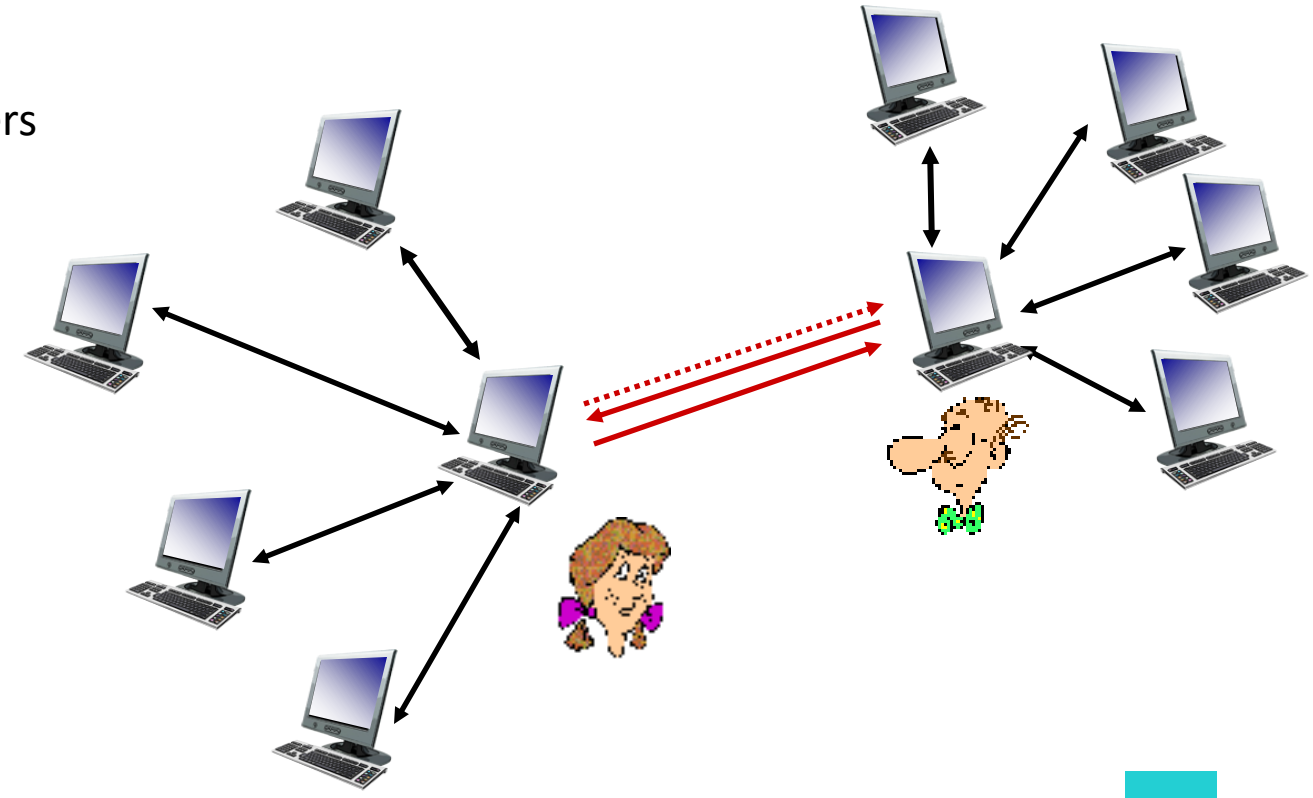
Sending File chunks

- 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 every 10 secs

BitTorrent

Tit-for-tat

- Every 30 secs Alice randomly select another peer (Bob)
- Alice **optimistically unchokes** Bob
- Alice may become one of Bob's top-four providers
- Bob may reciprocate and therefore become one of Alice's top-four providers



Video Steaming & CDN

Video Streaming & CDN

- Video traffic is a major consumer of Internet bandwidth

- Challenges
  - Scale: How to reach ~1B users?
    - Single mega-video server will not work (why?)
  - Heterogeneity
    - Different users have different capabilities
      (e.g. wired vs mobile; bandwidth rich vs bandwidth poor)

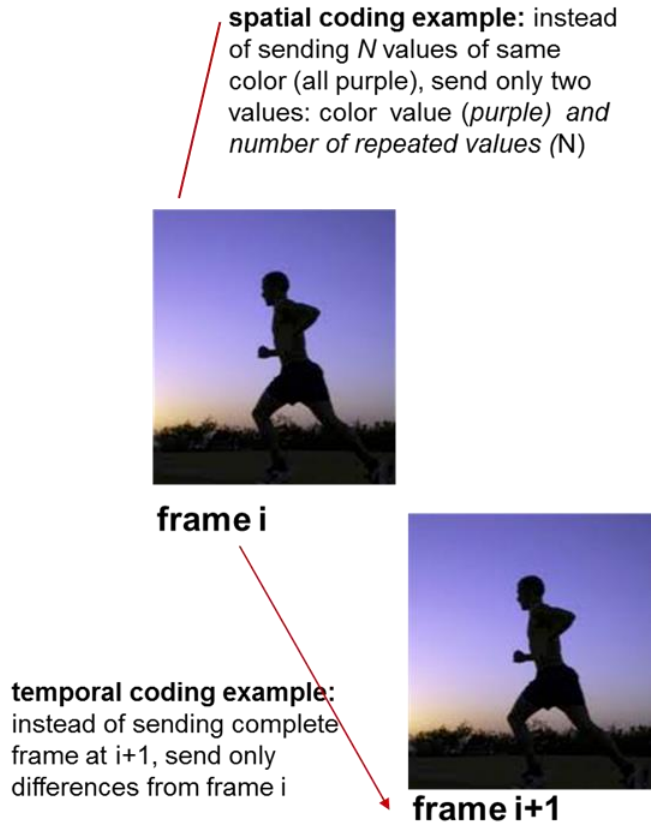
Solution: Distributed Application-Level Infrastructure

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:** Using redundancy **within** and **between** images to decrease number of bits used to encode image
  - **Spatial** (within image)
  - **Temporal** (from one image to next)



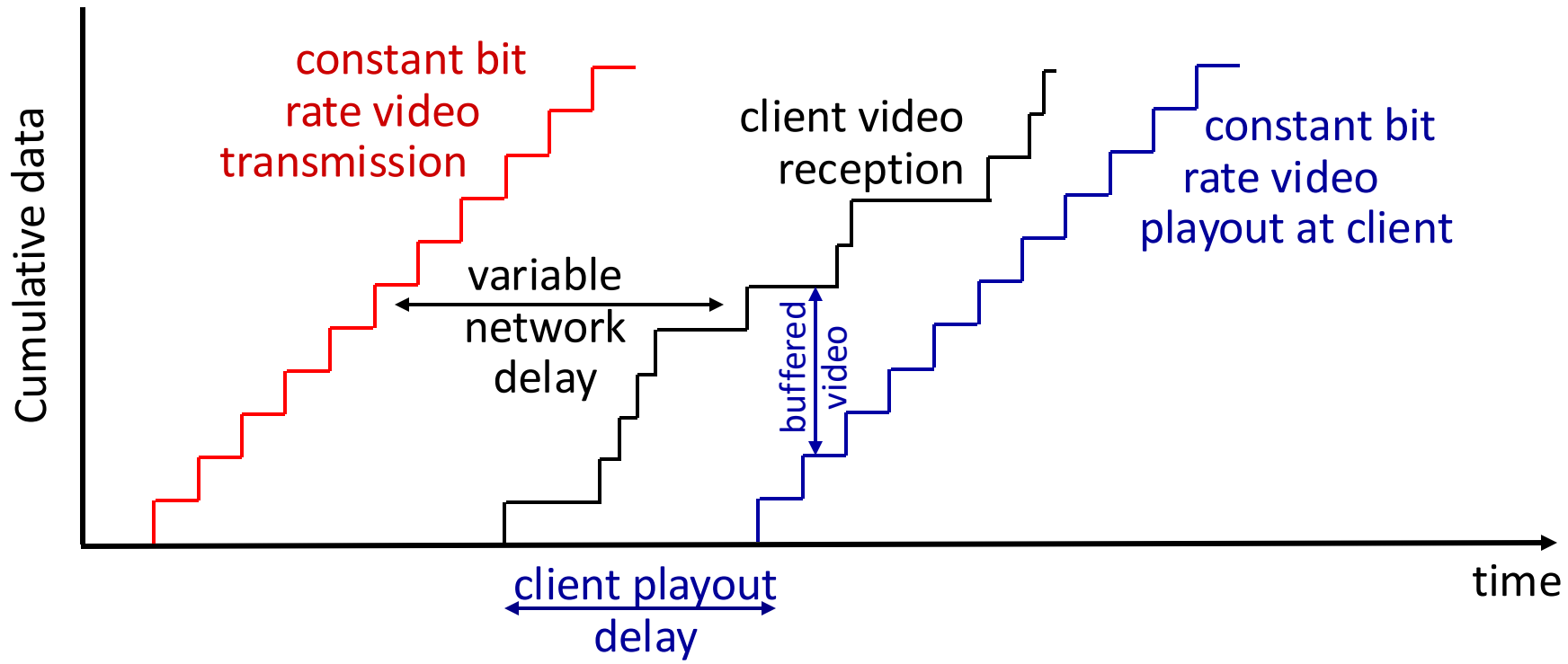
Video

- **CBR: (Constant Bit Rate):** Video encoding rate fixed

- **VBR: (Variable Bit Rate):** Video encoding rate changes as amount of spatial, temporal coding changes

- **Examples:**
  - MPEG1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, < 1 Mbps)

Streaming Video



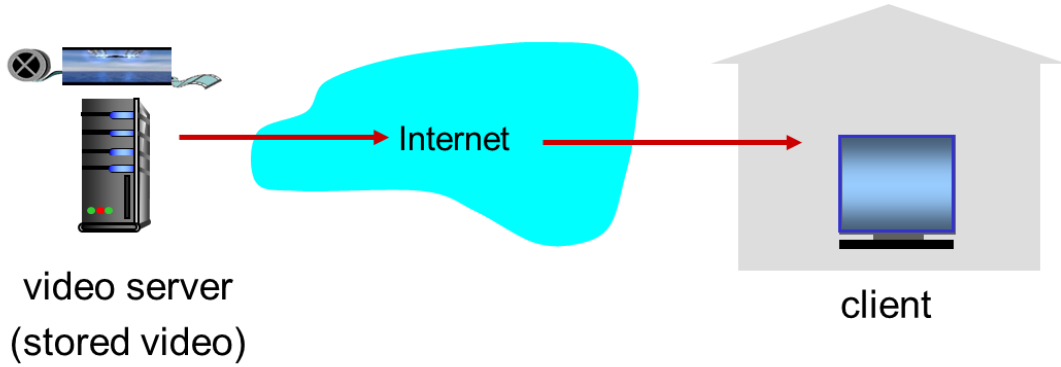
Client-side buffering and playout delay: Compensate for network-added delay and delay jitter

Streaming Stored Video: DASH

DASH: Dynamic **A**daptive **S**treaming 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)
- **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)

CDN: Content Distribution Networks

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

Option 1: A 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

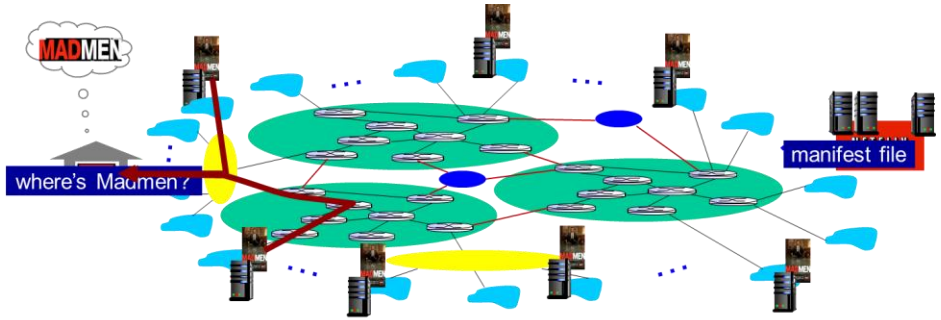
This solution **does not scale!**

Option 2: Store/serve multiple copies of videos at multiple geographically distributed sites **(CDN)**

- **Enter deep: (Access ISPs)** Push CDN servers deep into many access networks
  - Close to users
  - Used by Akamai, 1700 locations
- **Bring home: (IXP)** smaller number (10's) of larger **clusters** near (but not within) access networks
  - Used by Limelight

CDN

- CDN: Stores copies of content at CDN nodes
- Subscriber requests content from CDN
  - Directed to nearby copy, retrieves content
  - May choose different copy if network path congested



Over the top (OTT): Internet host-host communication as a service

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?

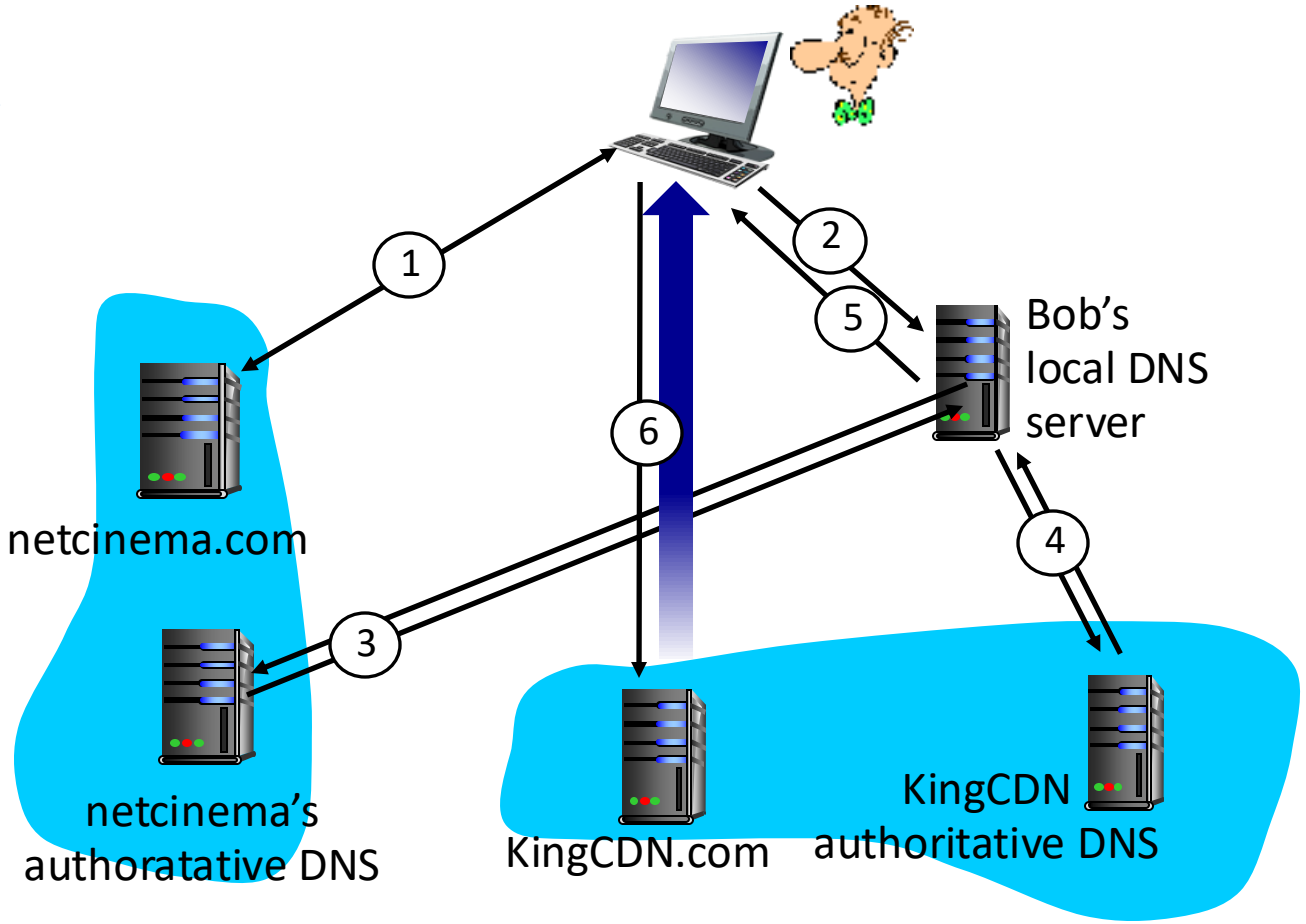
CDN: Example

Bob (client) requests video

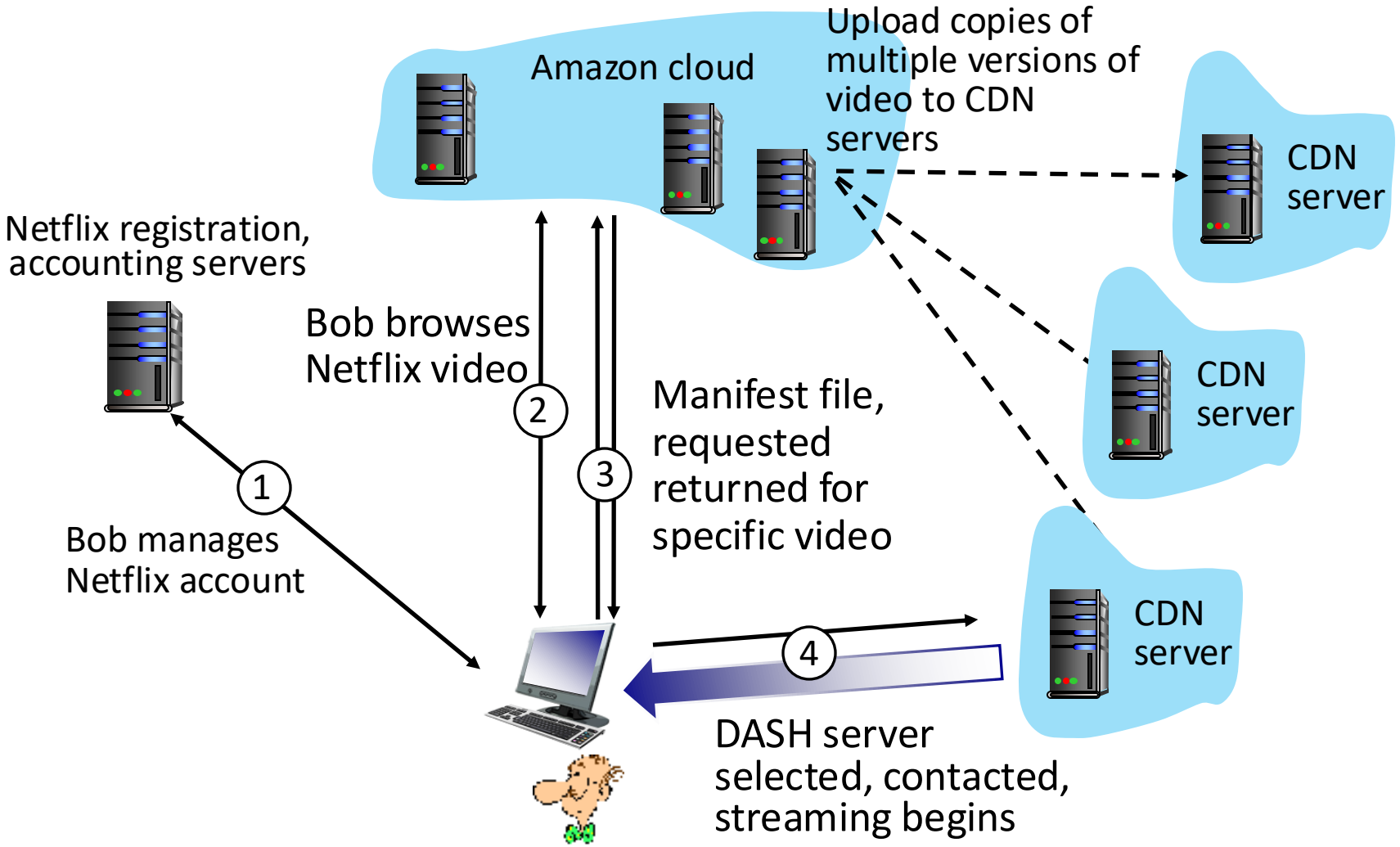
`http://video.netcinema.com/6Y7B23V`

Video stored in CDN at

`http://KingCDN.com/NetC6y&B23V`



Case Study: Netflix

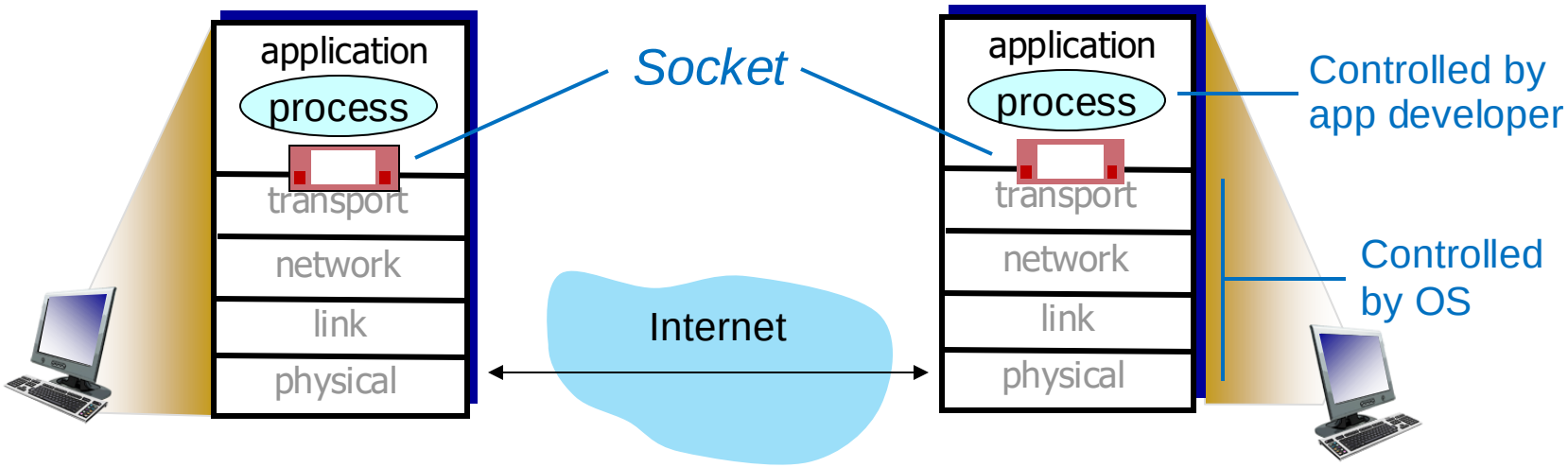


Socket Programming

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 number to each packet
- Receiver extracts sender IP address and port number from received packet
- 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
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)
```

Create UDP socket →

Bind socket to local port number 12000 →

Loop forever →

Read from UDP socket into message, getting client's address (client IP and port) →

Send upper case string back to this client →

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

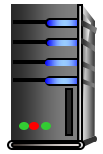
Client contacts server by:

- 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 Transport layer)

Application Viewpoint

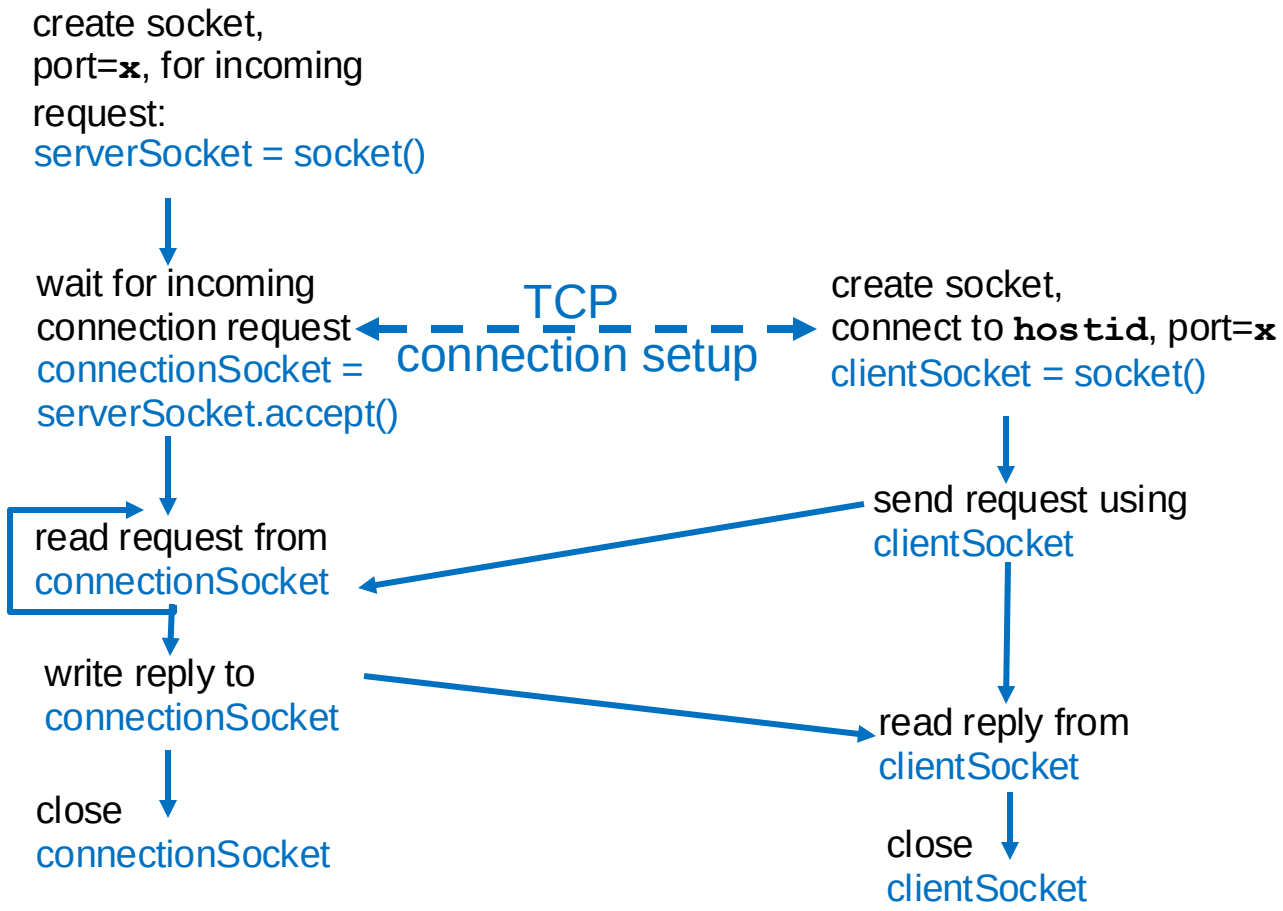
TCP provides reliable, in-order byte-stream transfer (pipe) between client and server

Client/Server Socket Interaction: TCP



Server *(Running on hostid)*

Client



Example App: TCP Client

Python TCPClient

```
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()
```

Create TCP socket for Client →

Connect to Server TCP Socket →

No need to attach server name, port →

Example App: TCP Server

Python TCPServer

```
 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()
```

Create TCP welcoming socket

Server begins listening for incoming TCP requests

loop forever

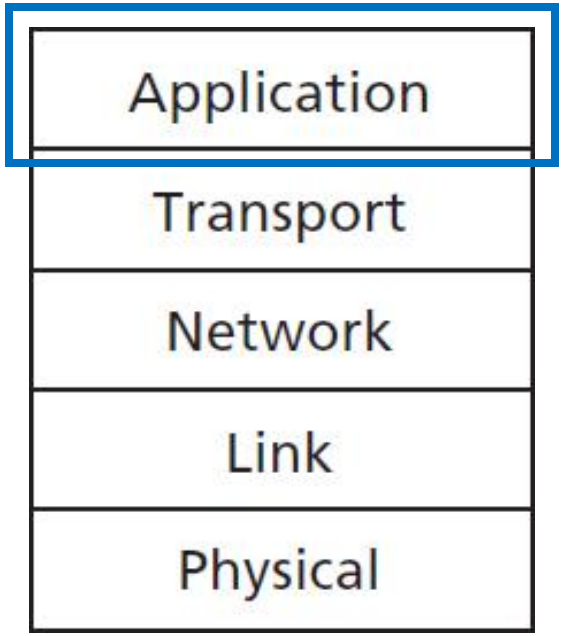
Server waits on accept() for incoming requests, new socket created on return

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

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

Application Layer Summary

- Network Applications
- Web & HTTP
- E-mail (SMTP, POP3, IMAP)
- Domain Name System (DNS)
- Peer-to-Peer (P2P) Applications
- Video Streaming and Content Distribution Networks (CDN)
- Socket Programming (UDP & TCP)



Acknowledgements

- The following materials have been used in preparation of this presentation:

  [1] Textbook and (edited) Slides: Computer Networking: A Top-Down Approach
  James Kurose, Keith Ross
  7th and 8th Edition, Pearson
  http://gaia.cs.umass.edu/kurose_ross/

  [2] Reference: Computer Networks: A Systems Approach
  https://www.systemsapproach.org/book.html

- Recommended Additional Resources:

  [1] Interactive Exercises (Chapter two)
  http://gaia.cs.umass.edu/kurose_ross/interactive/