Application layer: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Application layer

our goals:

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- creating network applications
  - socket API

Some network apps

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube,Youku, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- Search
- Location- and context-sensitive apps
- …
- …

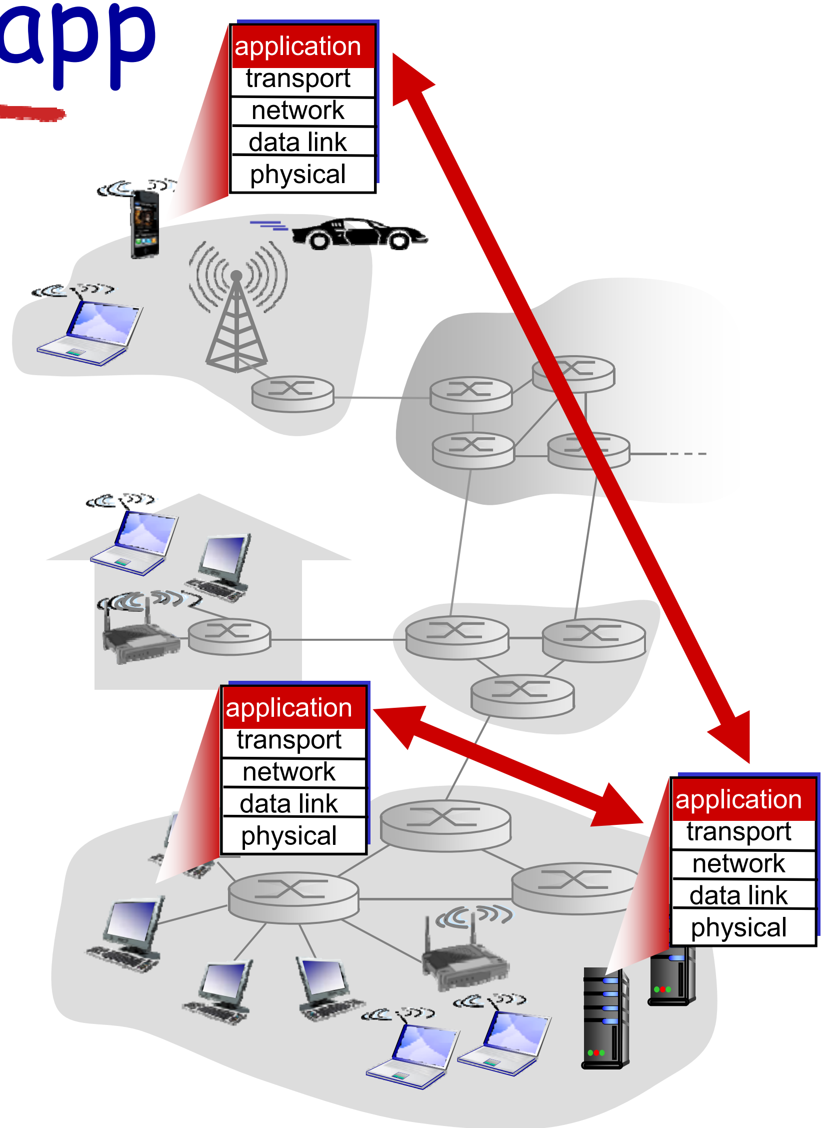
Creating a network app

write programs that:

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

no need to write software for network-core devices

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

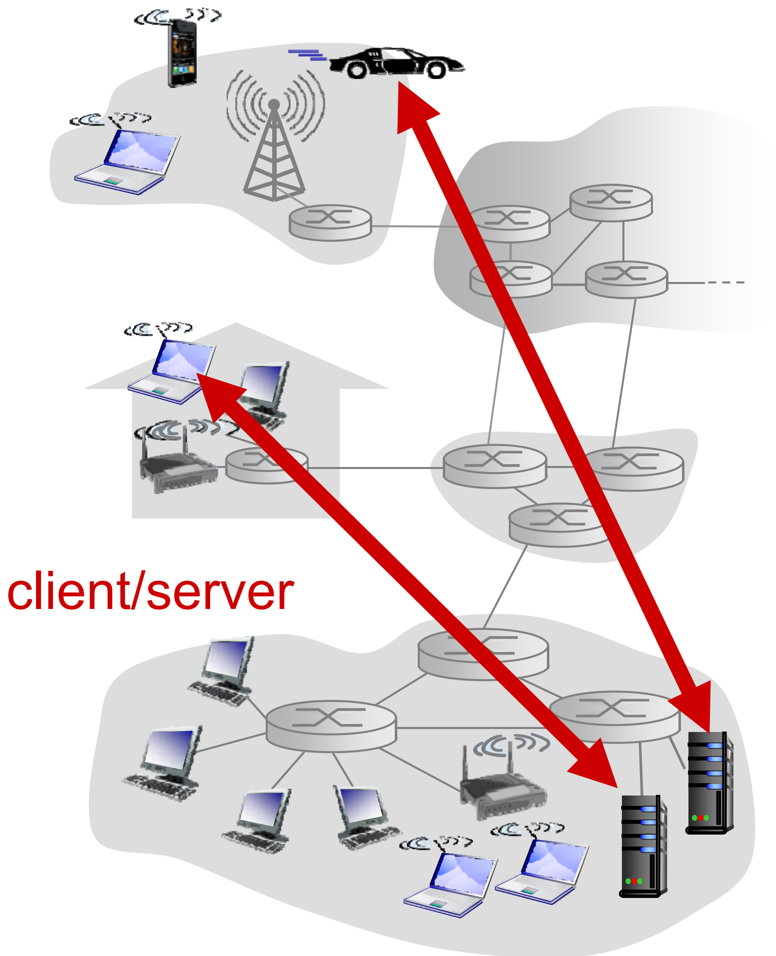


Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)

Client-server architecture

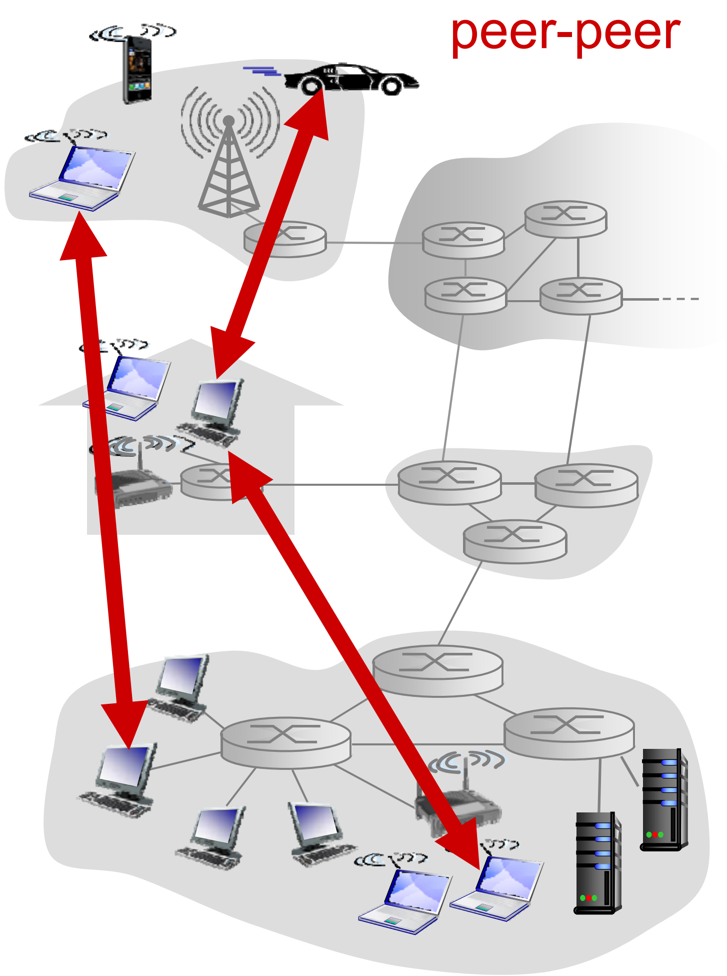


server:
- always-on host
- permanent IP address
- data centers for scaling

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

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, as well as new service demands*
- peers are intermittently connected and change IP addresses
  - complex management



Processes communicating

process: program running within a host

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

clients, servers

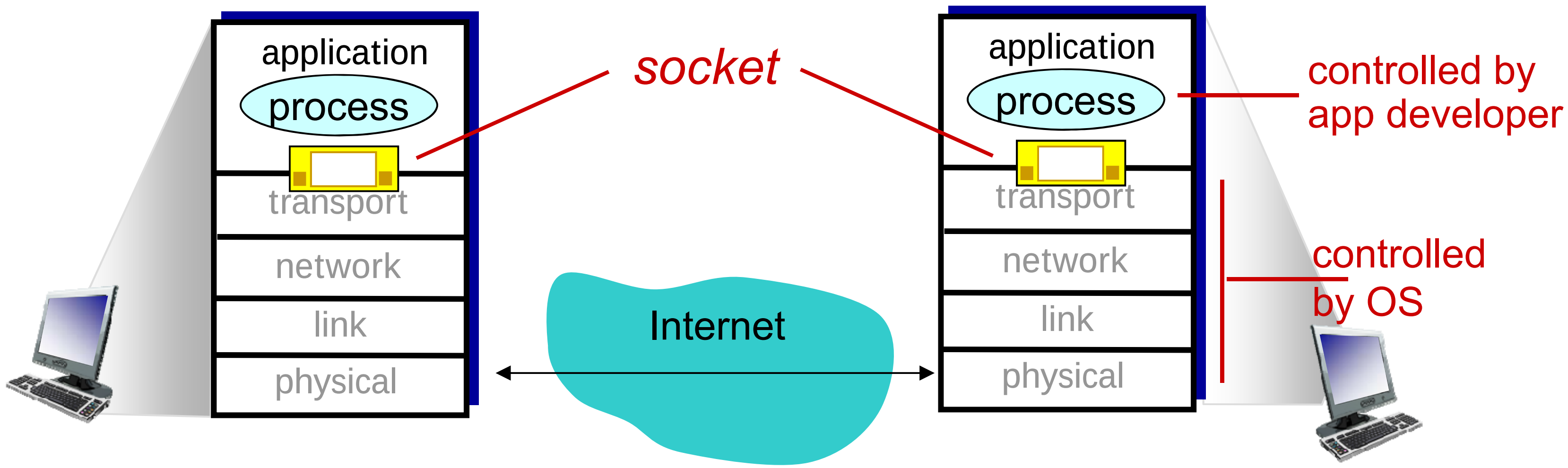
client process: process that initiates communication

server process: process that waits to be contacted

- aside: applications with P2P architectures have **both** client processes & server processes

Sockets

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



Addressing processes

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

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to "www.fudan.edu.cn" web server:
  - IP address: 202.120.224.5
  - port number: 80
- more shortly…

App-layer protocol defines

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

open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

What transport service does an app need?

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

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

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

security
- encryption, data integrity, …

Transport service requirements: common apps

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100's
			msec
			yes and no

Internet transport protocols services

TCP service:

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

UDP service:

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

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

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

Securing TCP

TCP & UDP

- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

SSL

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

SSL is at app layer

- Apps use SSL libraries, which "talk" to TCP

SSL socket API

- cleartext passwds sent into socket traverse Internet encrypted

Application layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Web and HTTP

First, a review…

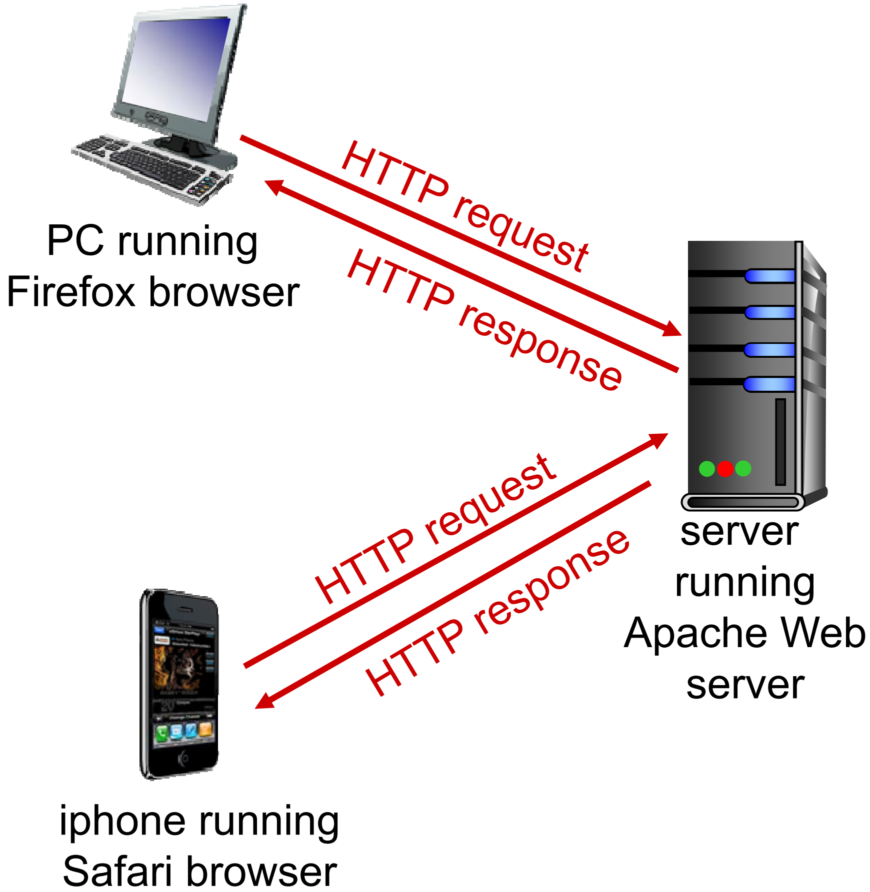
- *web page* consists of *objects*
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL,* e.g.,

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)

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

1b. HTTP server at host www.someSchool.edu 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 someDepartment/home.index

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

time

Non-persistent HTTP (cont.)

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

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

time

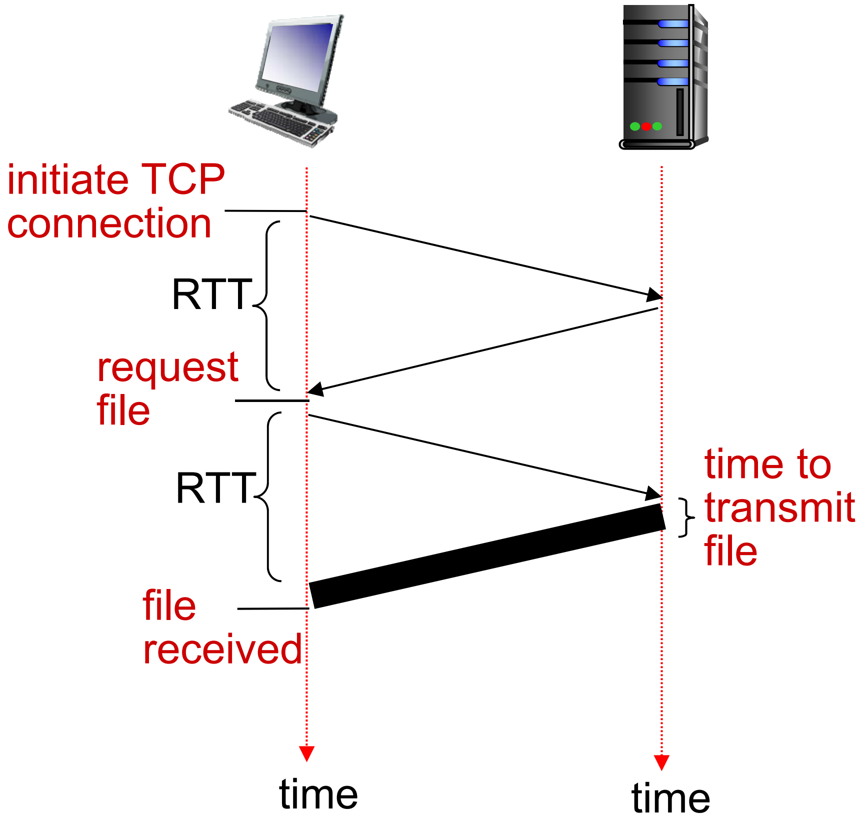
Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and 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



File transmission time = Transfersize/Bandwidth

Persistent HTTP

non-persistent HTTP issues:

- requires 2 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 request message

- two types of HTTP messages: *request, response*
- *HTTP request message:*
  - ASCII (human-readable format)

carriage return character

line-feed character

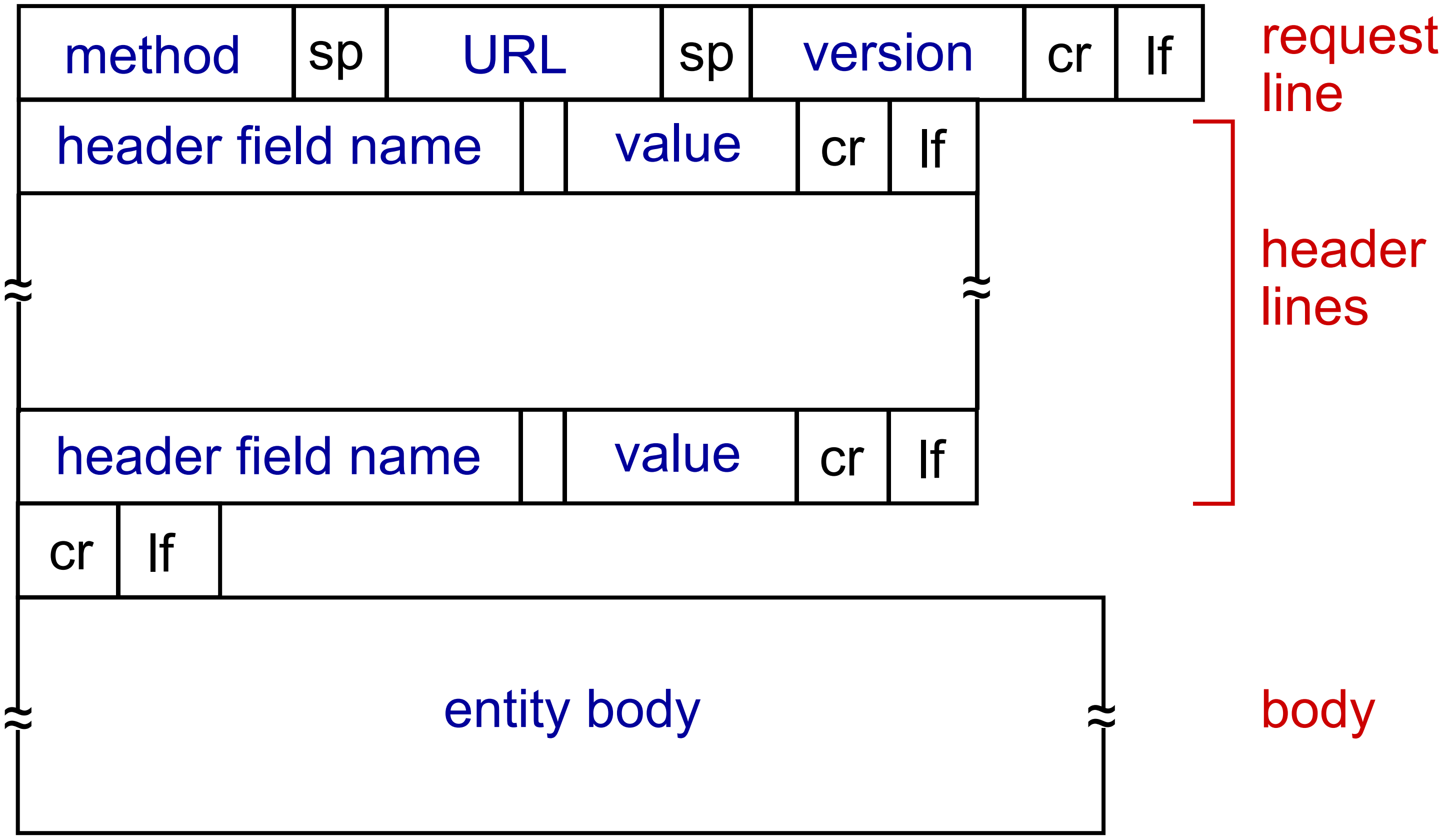
request line (GET, POST, HEAD commands)

header lines

carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP request message: general format



Uploading form input

POST method:

- web page often includes form input
- input is uploaded to server in entity body

URL method:

- uses GET method
- input is uploaded in URL field of request line:

`www.somesite.com/animalsearch?monkeys&banana`

Method types

HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

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 response message

status line
(protocol
status code
status phrase)

header
lines

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

data, e.g.,
requested
HTML file

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

- requested object moved, new location specified later in this msg (Location:)

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. anything typed in sent to port 80 at cis.poly.edu

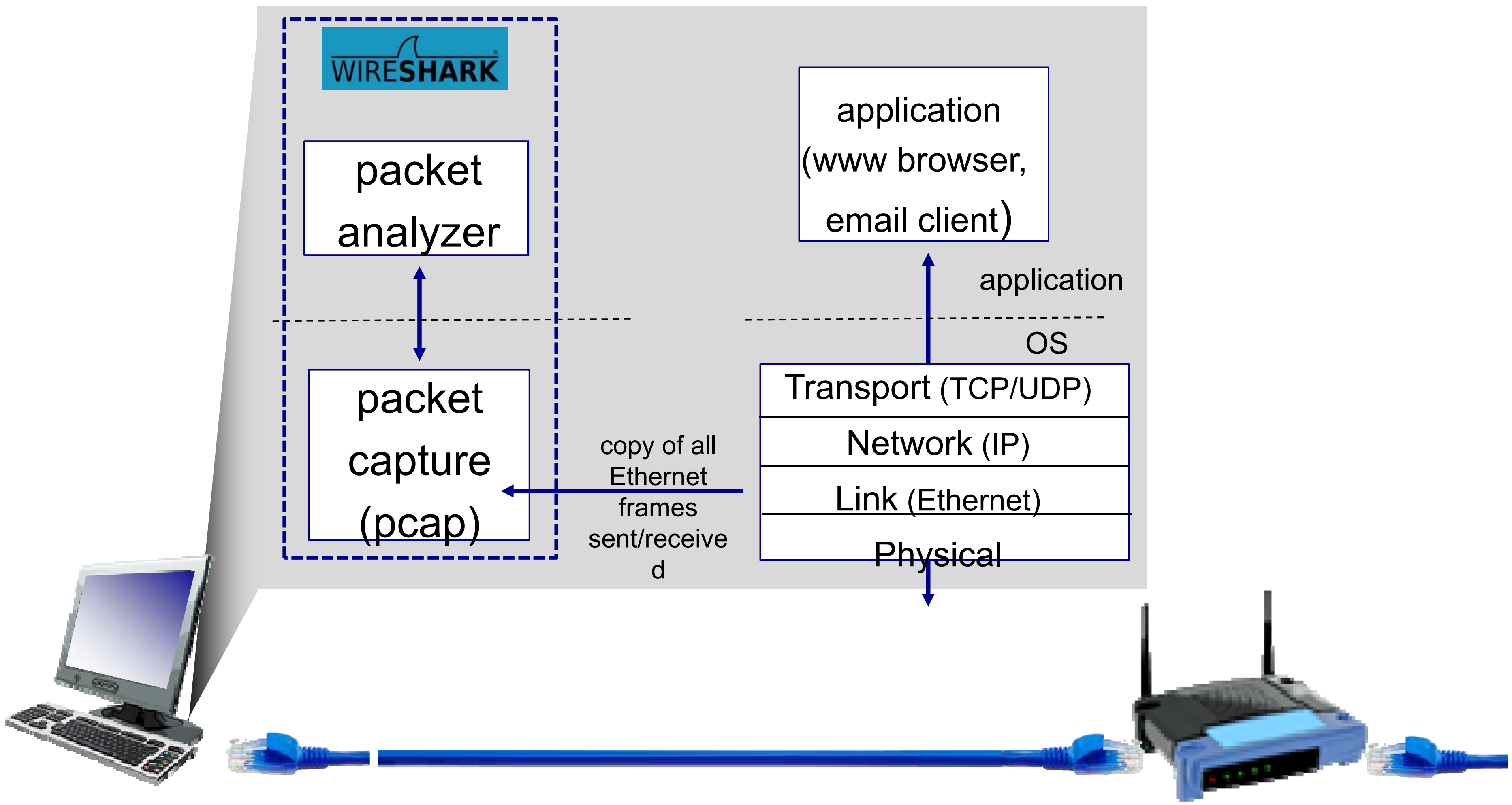
2. type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1
Host: cis.poly.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!

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


WIRESHARK
packet analyzer
packet capture (pcap)
copy of all Ethernet frames sent/received
application (www browser, email client)
application
OS
Transport (TCP/UDP)
Network (IP)
Link (Ethernet)
Physical

User-server state: cookies

many Web sites use cookies

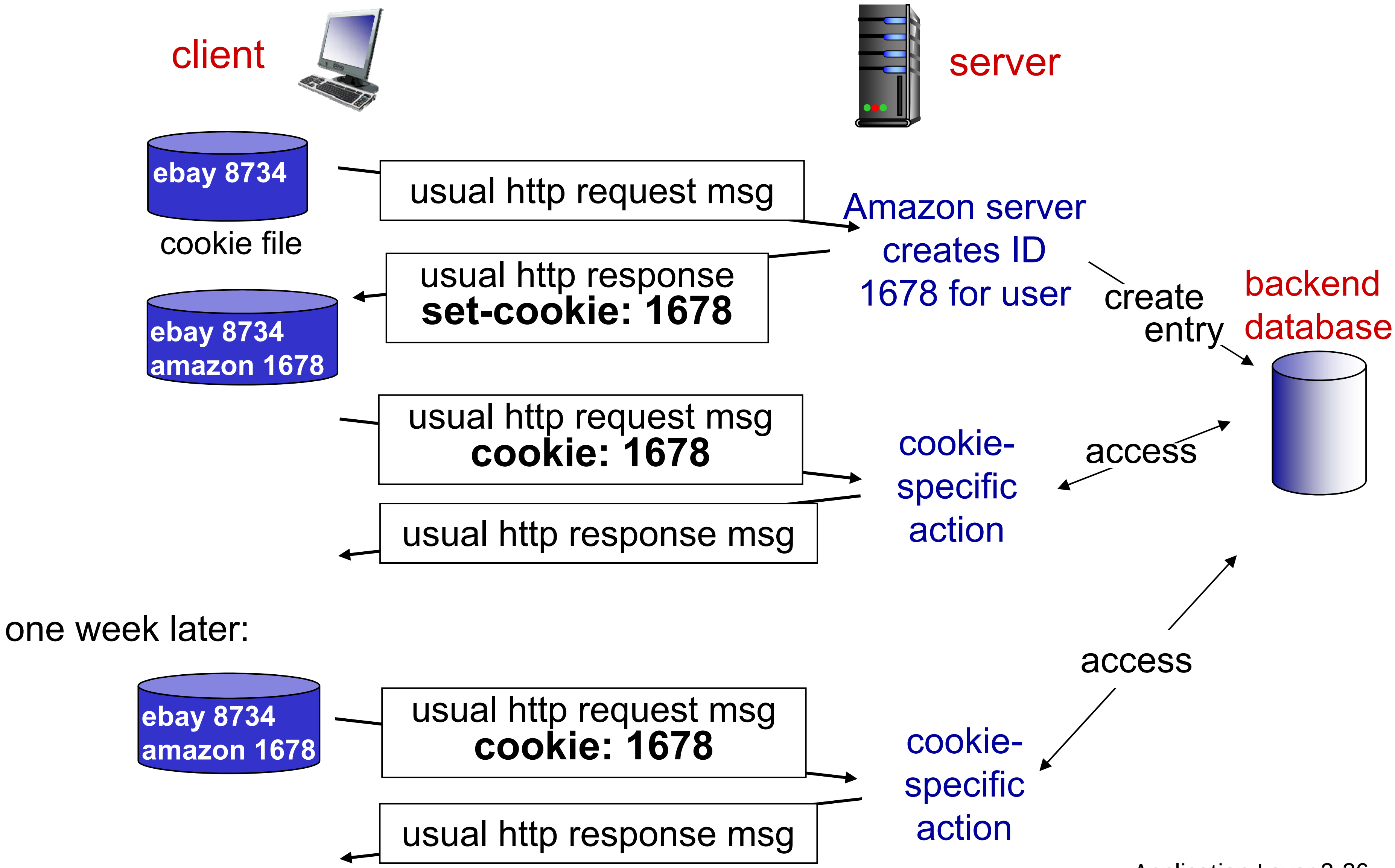
four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

example:

- Susan always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping “state” (cont.)



Cookies (continued)

what cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

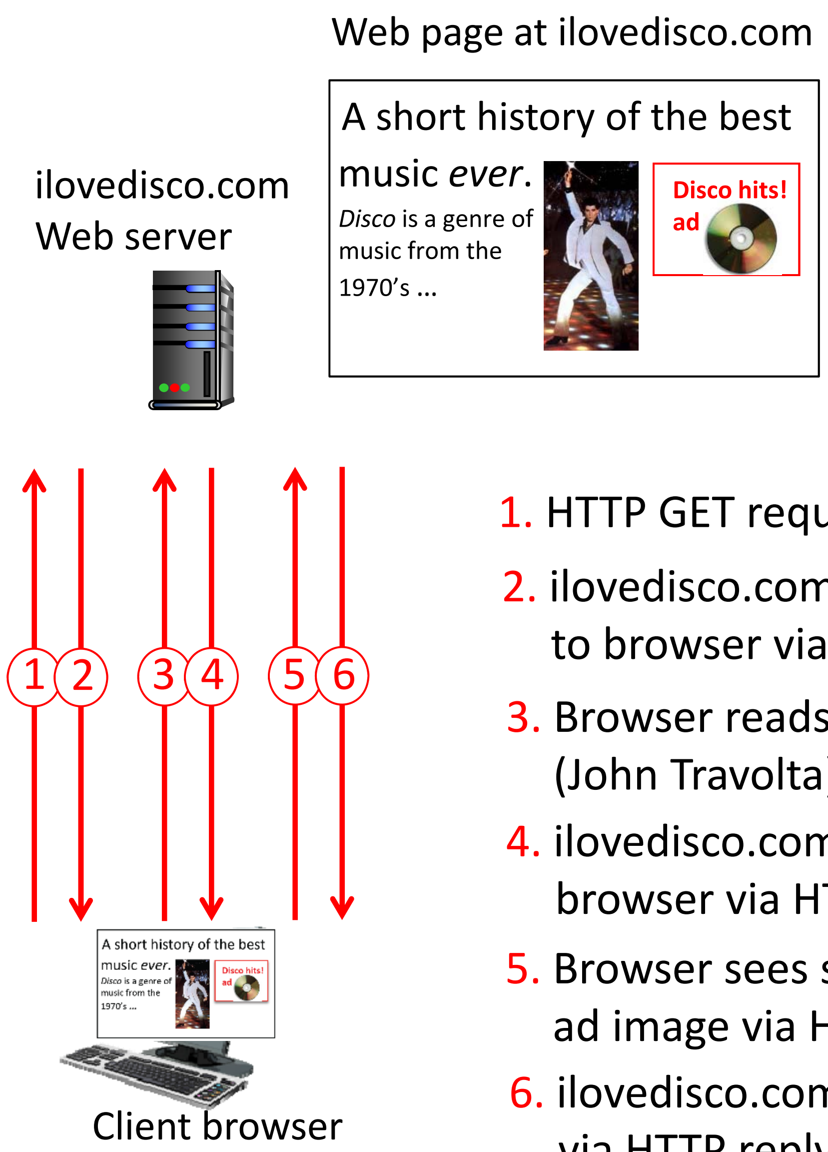
how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Aside - HTTP: cookies and advertising

- third-party cookies: ad network server tracking user web page accesses across multiple sites

HTTP: homepage, image, ad (v1)

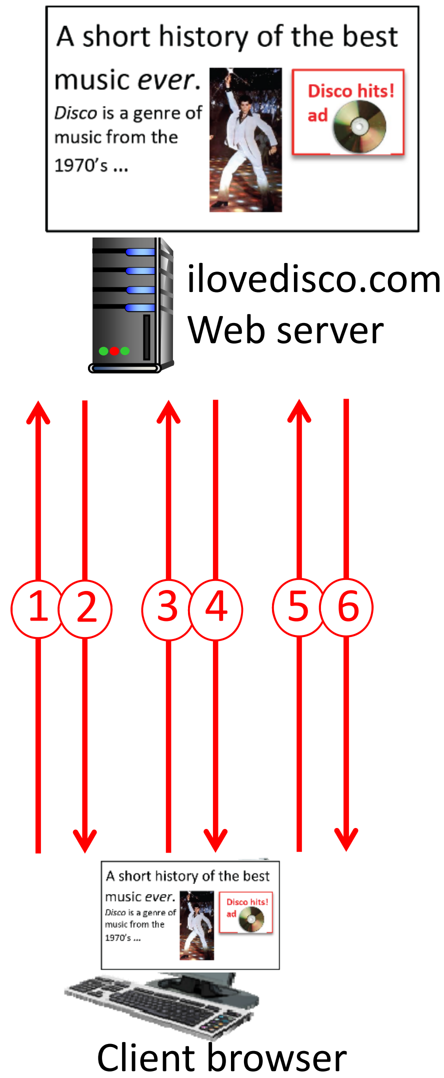


HTML file contains text, and two <IMG> tags. Both images are stored on ilovedisco.com

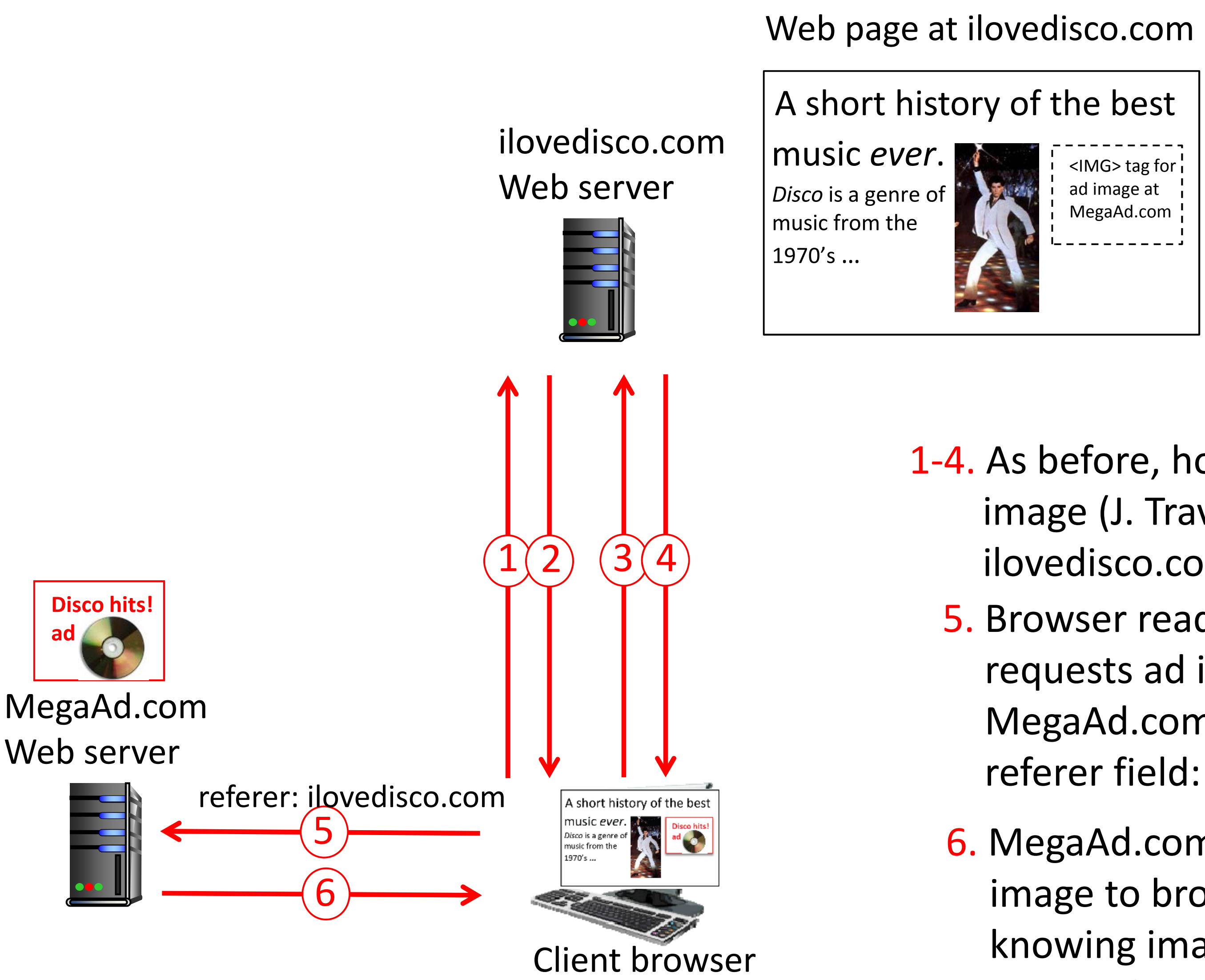
1. HTTP GET request to ilovedisco.com for homepage
2. ilovedisco.com server send homepage HTML file to browser via HTTP reply
3. Browser reads homepage HTML file, sees <img> tag (John Travolta), requests first image via HTTP GET
4. ilovedisco.com server sends first image to browser via HTTP reply
5. Browser sees second <img> tag (the ad) and requests ad image via HTTP GET
6. ilovedisco.com server sends ad image to browser via HTTP reply

HTTP: homepage, image, ad (v1): observations

- all web page content at ilovedisco.com
- HTML file, Travolta image, ad are *separate* files on server - *composed into webpage at client*
- same content would be served to all browsers
- ilovedisco.com would sell ad space directly to Disco Hits



HTTP: homepage, image, ad (v2)



HTML file contains text, and two <IMG> tags. Travolta images stored on ilovedisco.com; ad image stored at MegaAd.com

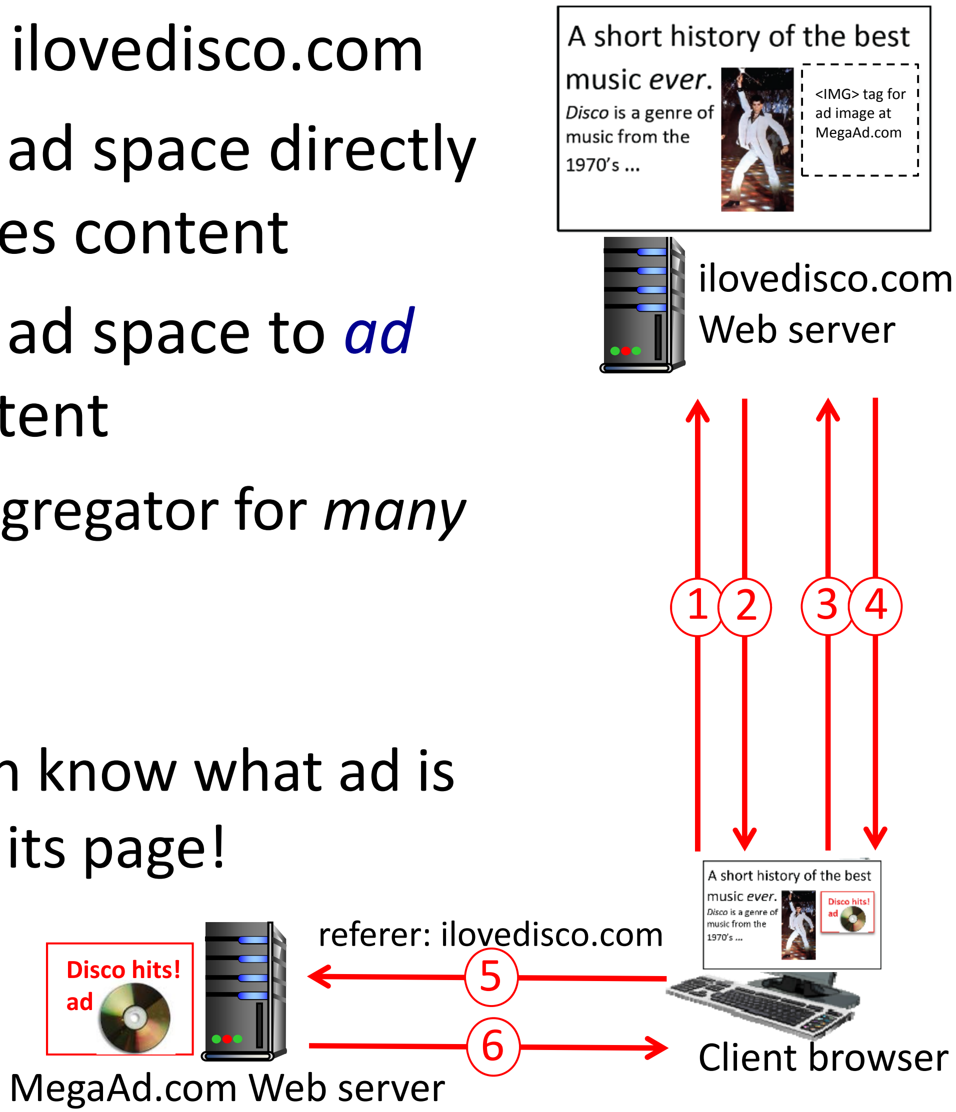
1-4. As before, home page, and first image (J. Travolta) downloaded from ilovedisco.com

5. Browser reads second <img> tag (ad) requests ad image from MegaAd.com via HTTP GET, with referer field: ilovesdisco.com

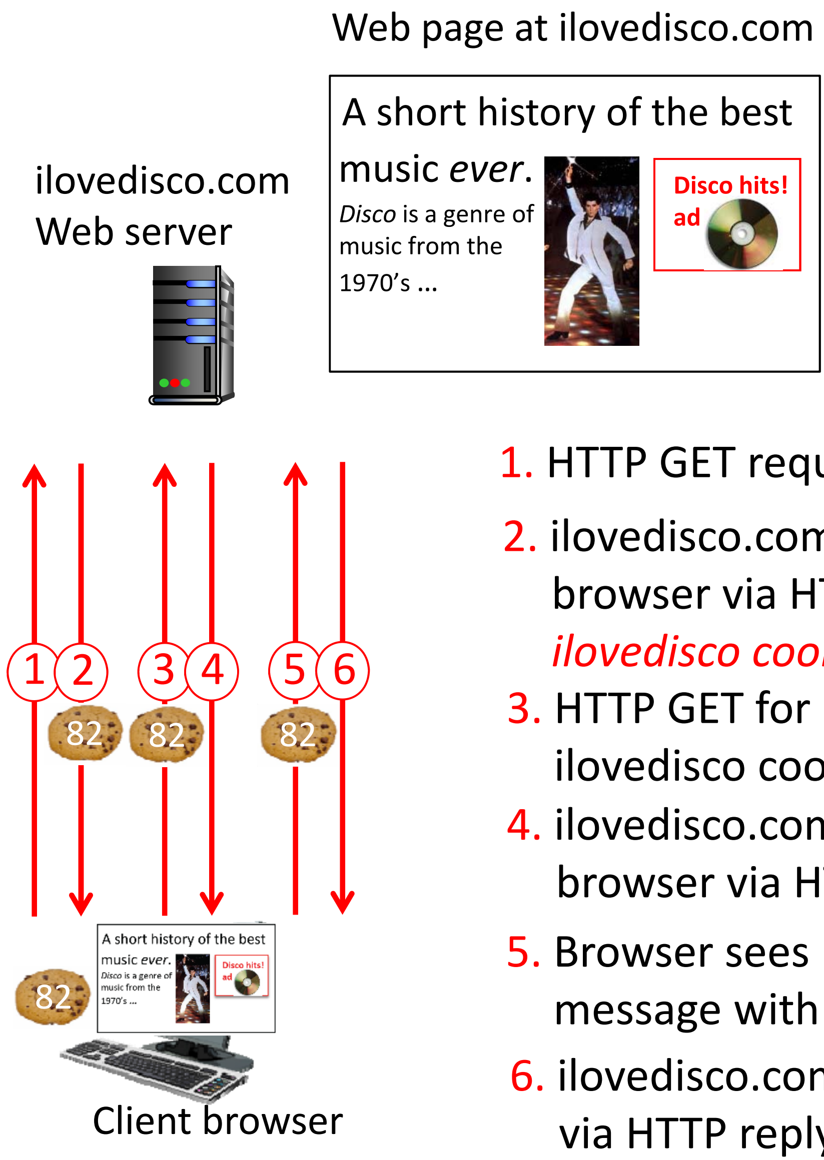
6. MegaAd.com server sends ad image to browser via HTTP reply, knowing image is to be embedded in page from ilovedisc.com

HTTP: homepage, image, ad (v2): observations

- ad content *not* served by ilovedisco.com
- ilovedisco.com could sell ad space directly to Disco Hits who provides content
- ilovedisco.com could sell ad space to *ad network*, who serves content
  - ad network serves as aggregator for *many* products/companies,
  - knows “referer”
  - ilovedisco wouldn’t even know what ad is going to be displayed in its page!



HTTP: homepage, image, ad (v3): cookies



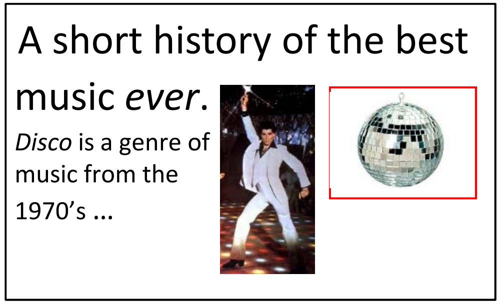
HTML file contains text, and two <IMG> tags. Both images are stored on ilovedisco.com

1. HTTP GET request to ilovedisco.com for homepage
2. ilovedisco.com server send homepage HTML file to browser via HTTP reply, with ilovedisco cookie: 82.
   ilovedisco cookie: 82 Cookie stored at client.
3. HTTP GET for image, GET message contains with ilovedisco cookie: 82
4. ilovedisco.com server sends first image to browser via HTTP reply
5. Browser sees requests ad image via HTTP GET message with ilovedisco cookie: 82
6. ilovedisco.com server sends ad image to browser via HTTP reply

HTTP: homepage, image, ad (v3): cookies

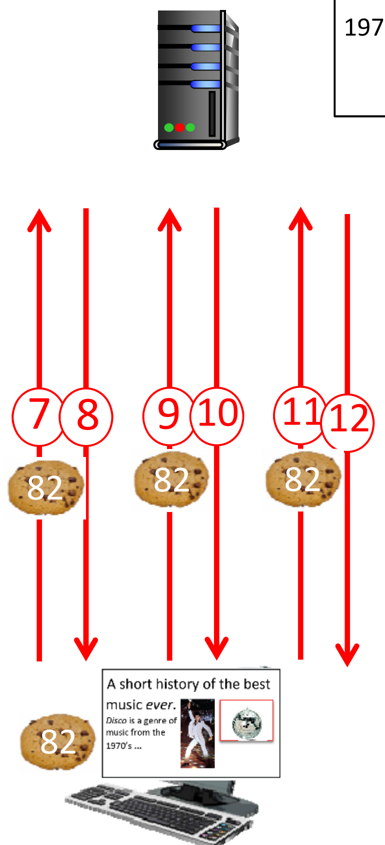
One week later

Web page at ilovedisco.com



HTML file contains text, and two <IMG> tags. All images are stored on ilovedisco.com. The second (ad) image will be chosen based on cookie

ilovedisco.com
Web server

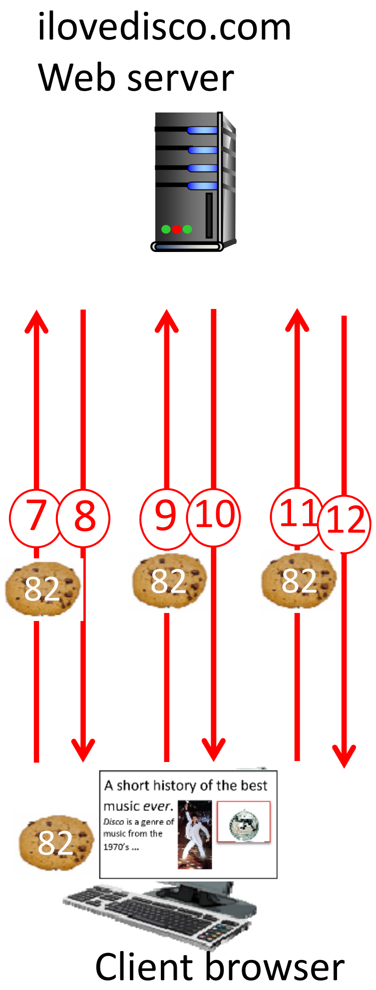


Client browser

7. HTTP GET request to ilovedisco.com for homepage with ilovedisco cookie: 82 from last week
8. ilovedisco.com server sees cookie in GET msg, sends homepage HTML file to browser via HTTP reply containing *DIFFERENT AD IMAGE* from last time
9. HTTP GET for Travolta image, GET contains with ilovedisco cookie: 82
10. ilovedisco.com server sends Travolta image
11. Browser requests new ad image via HTTP GET with ilovedisco cookie: 82
12. ilovedisco.com server sends *new ad image* to browser via HTTP reply

HTTP: homepage, image, ad (v3): observations

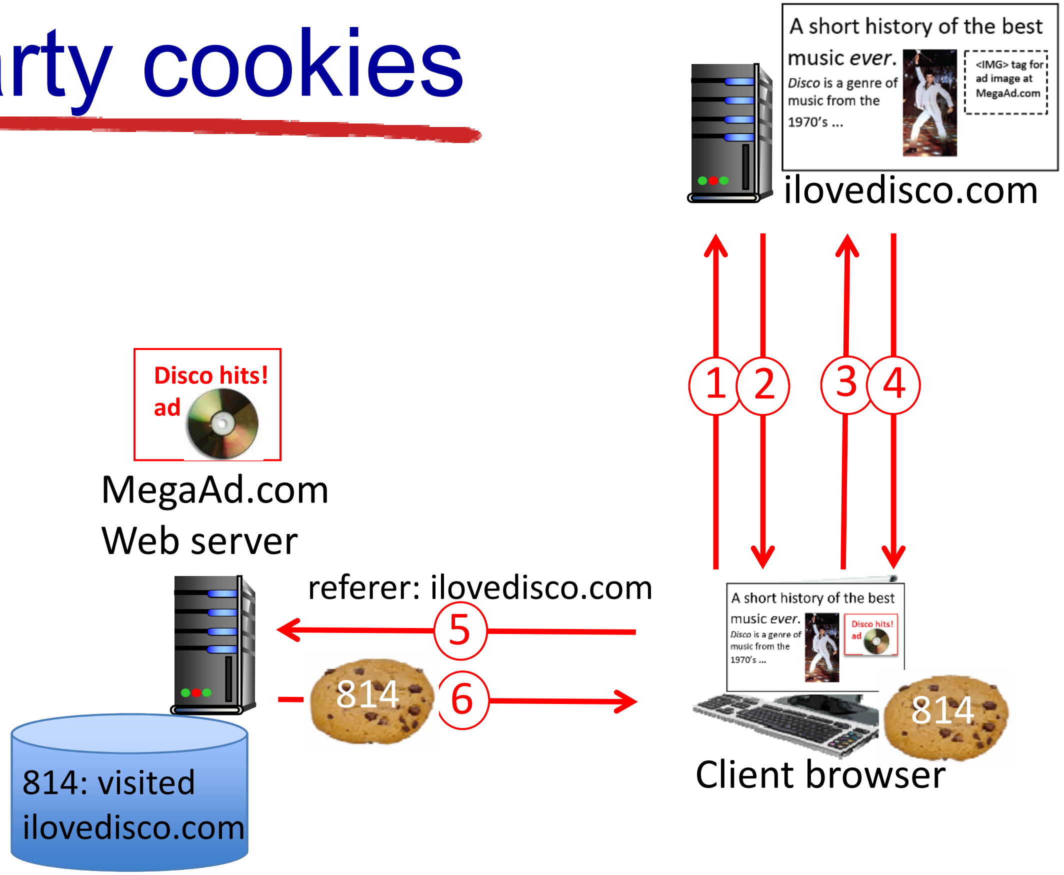
- cookies can be used to personalize (target) content (e.g., ads) to client based on past interaction with this server
  - web server can dynamically generate content depending on what client has done/seen in past



HTTP: Third party cookies

1-5. As before, home page, and first image (J. Travolta) downloaded from ilovedisco.com, request made for ad image from MegaAd.com via HTTP GET, with referer field: ilovesdisco.com

6. MegaAd.com server sends ad image to browser via HTTP reply, knowing image is to be embedded in page from ilovedisc.com, adds its own cookie MegaAd: 814. Remembers that cookie #814 owner had visited ilovedisco.com



Third party cookie: when you visit a web page, a third website is able to put a cookie on your browser (as shown here).

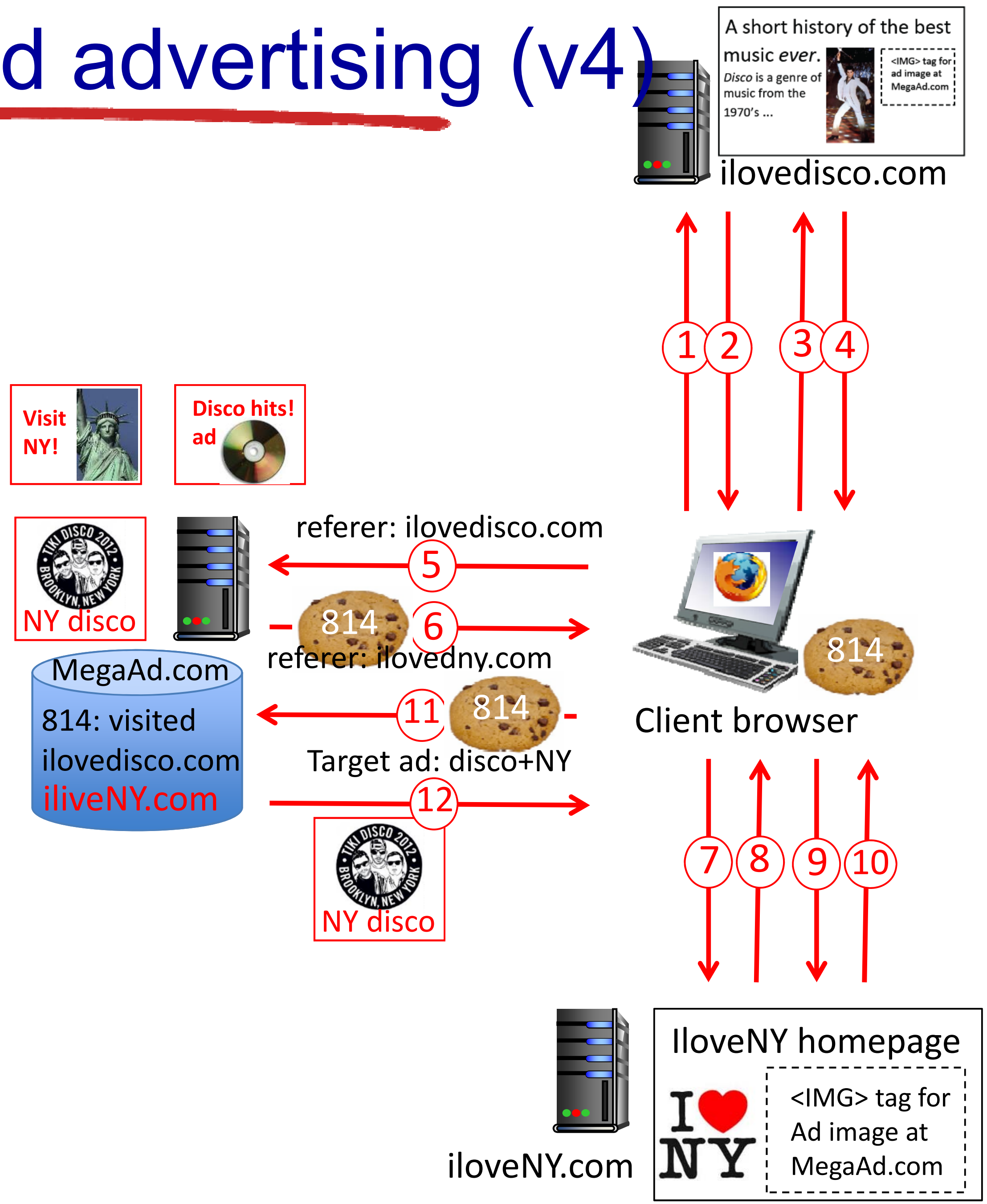
HTTP: Targeted advertising (v4)

1-6 client visits ilovedisco.com, disco ad served by MegaAd.com

7-10 client visits iloveNY.com, HTML text and image served by iloveNY.com

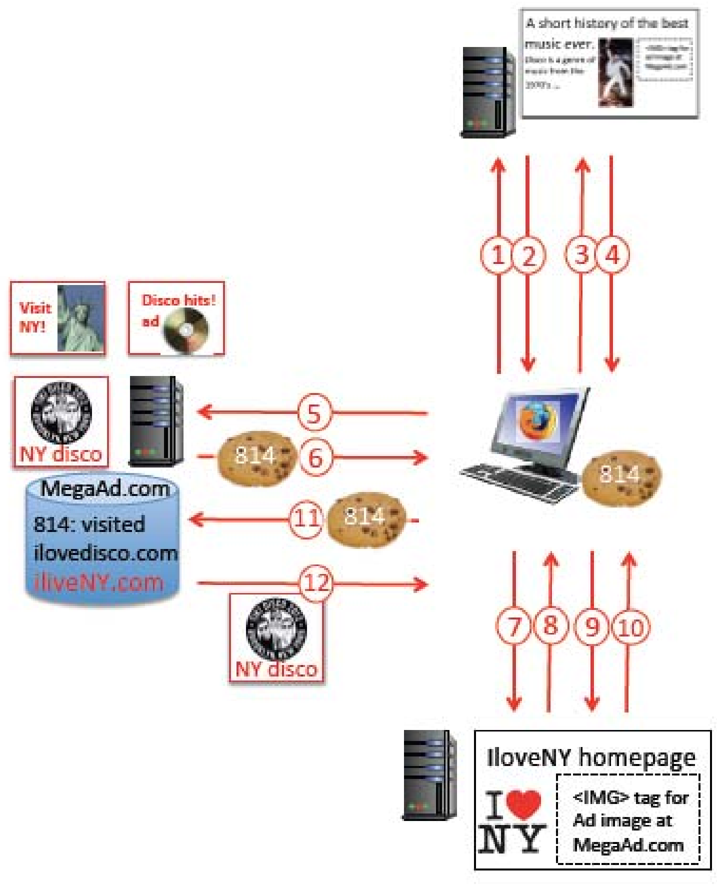
11 client contacts MegaAd.com to get ad to display, includes MegaAd cookie # 814

12 MegaAd.com sees refered request from iloveNY.com, sees cookie 814, knows client visited disco site earlier, serves targeted content ad: disco + NY



HTTP: Targeted advertising - observations

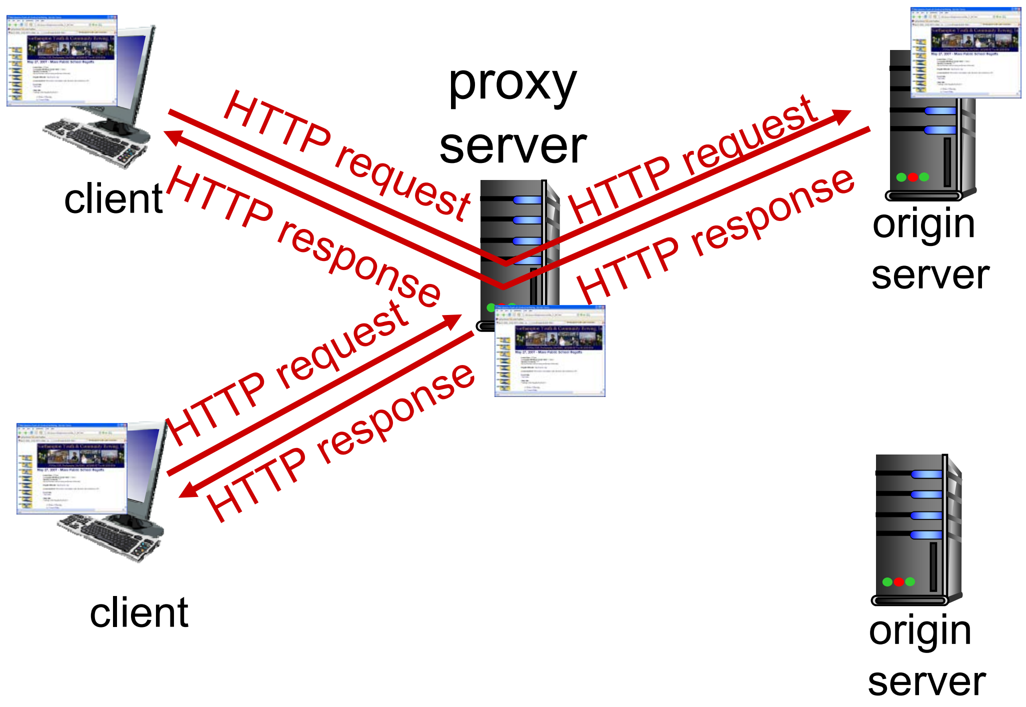
- *third party cookies* allow third party (e.g., MegaAds.com) to track user access over multiple web sites (any site with MegaAc link)
- MegaAd uses past user activity to *micro-target specific a*ds to specific users
  - MegaAd can charge ad creators more to place their ads in micro-targeted manner (since user is more likely to be interested in ad)
- users not aware of third party cookies and tracking
  - invasion of privacy ????



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
  - else cache requests object from origin server, then returns object to client



More about Web caching

- cache acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

Caching example:

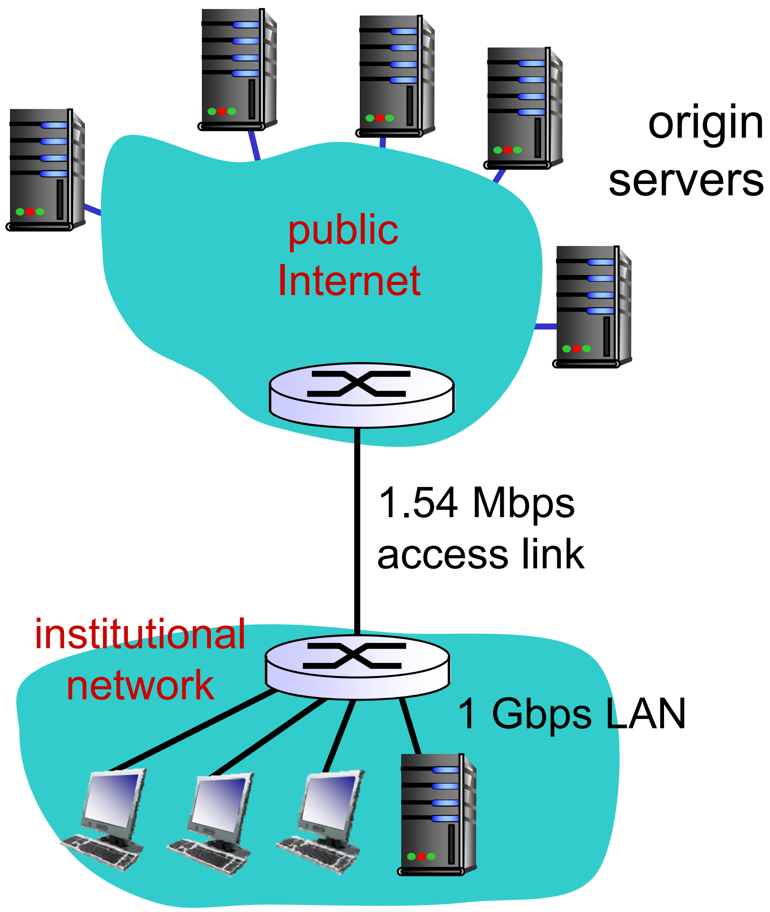
assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

problem!

- LAN utilization: 15%
- access link utilization = 99%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + msecs



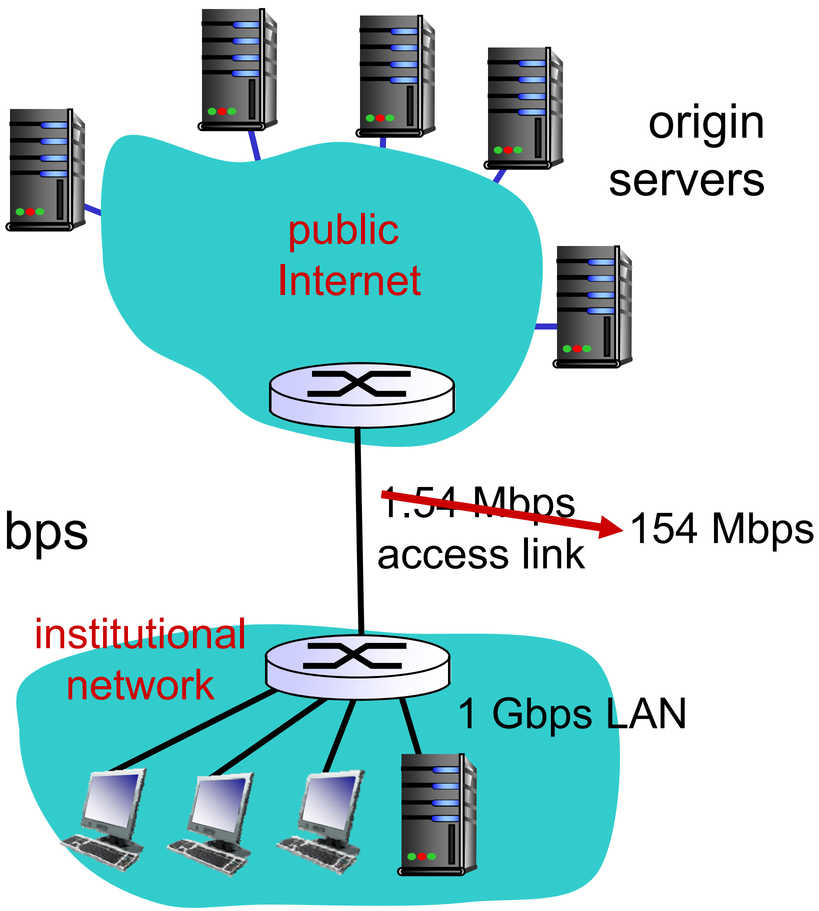
Caching example: fatter access link

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 ~~Mbps~~ 154 Mbps

consequences:

- LAN utilization: 15%
- access link utilization = ~~99%~~ 9.9%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + ~~minutes~~ + msecs
  msecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

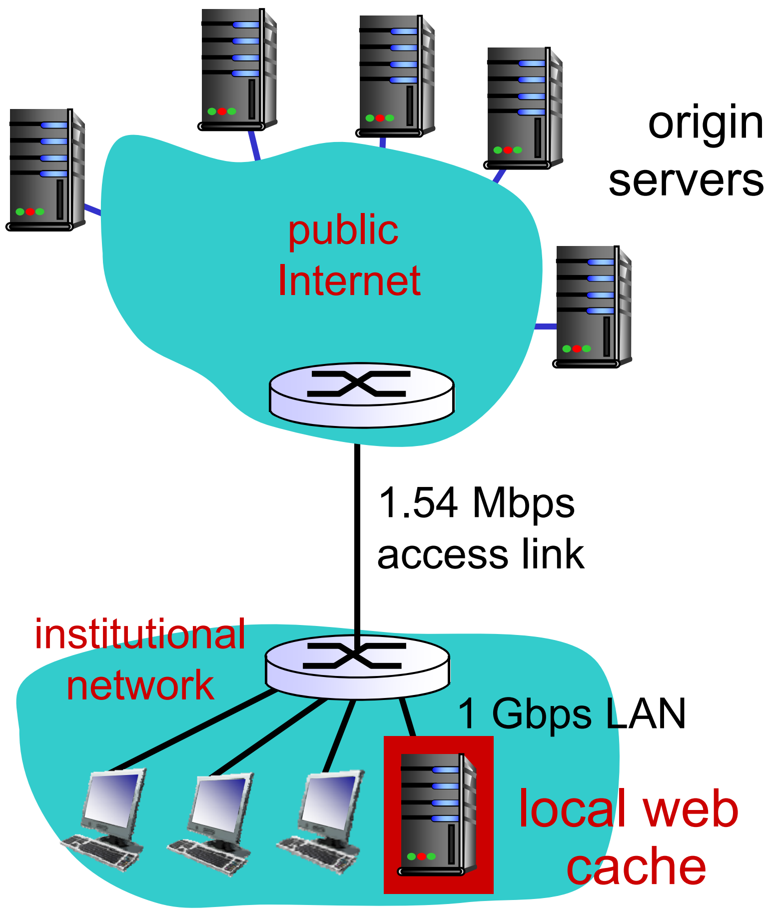
- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

- LAN utilization: 15% ?
- access link ut ?
- *How to compute link utilization, delay?*

= 2 sec + xx + msecs ?

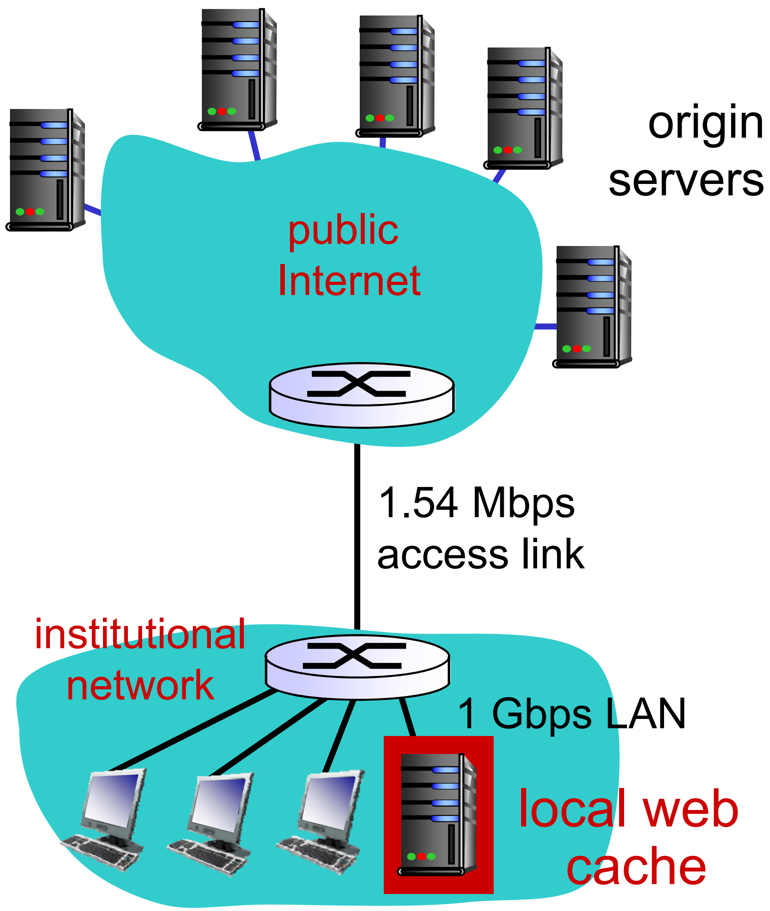
Cost: web cache (cheap!)



Caching example: install local cache

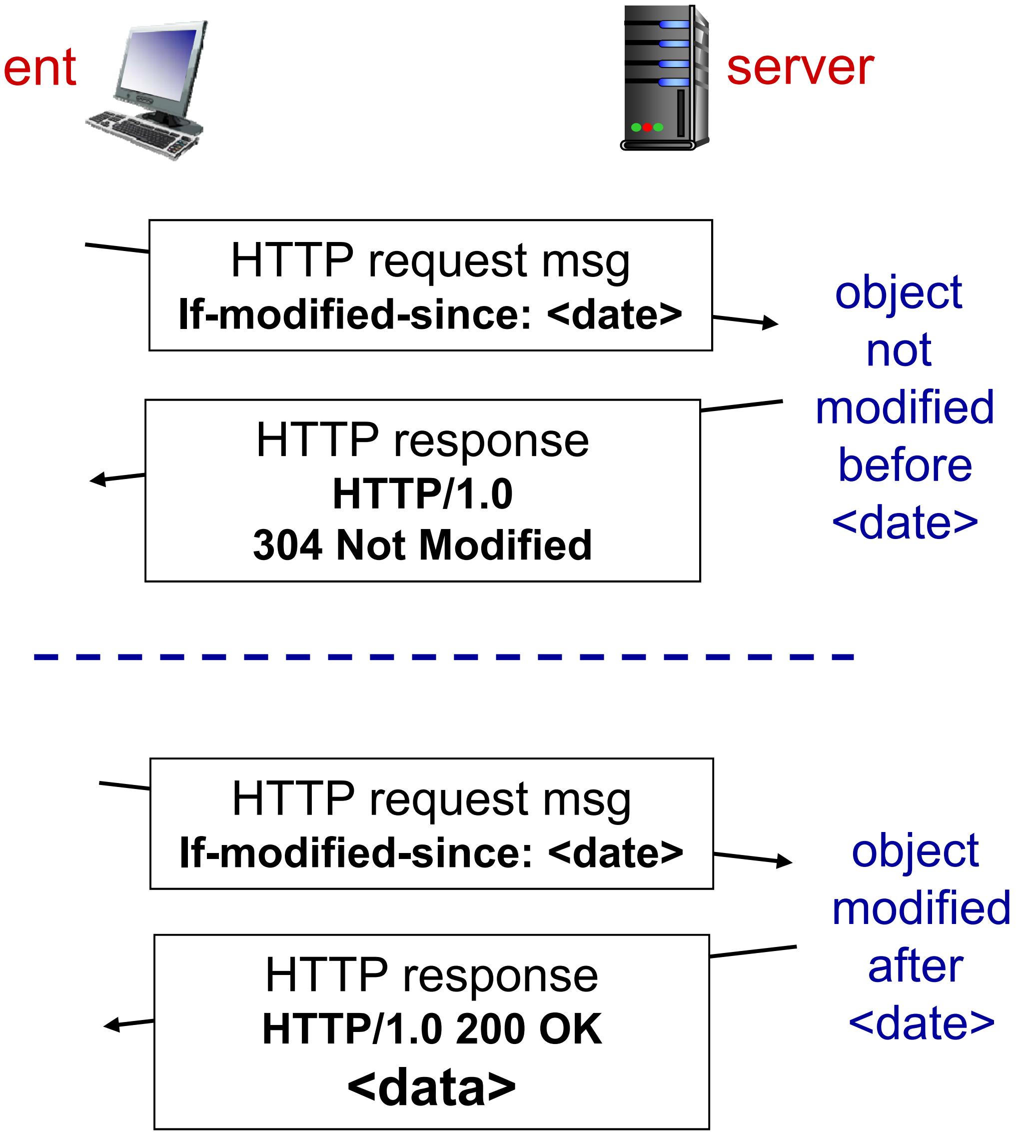
Calculating access link utilization, delay with cache:

- 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
- data rate to browsers over access link = 0.6*1.50 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs)
  - = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)



Conditional GET

- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *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



Application layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

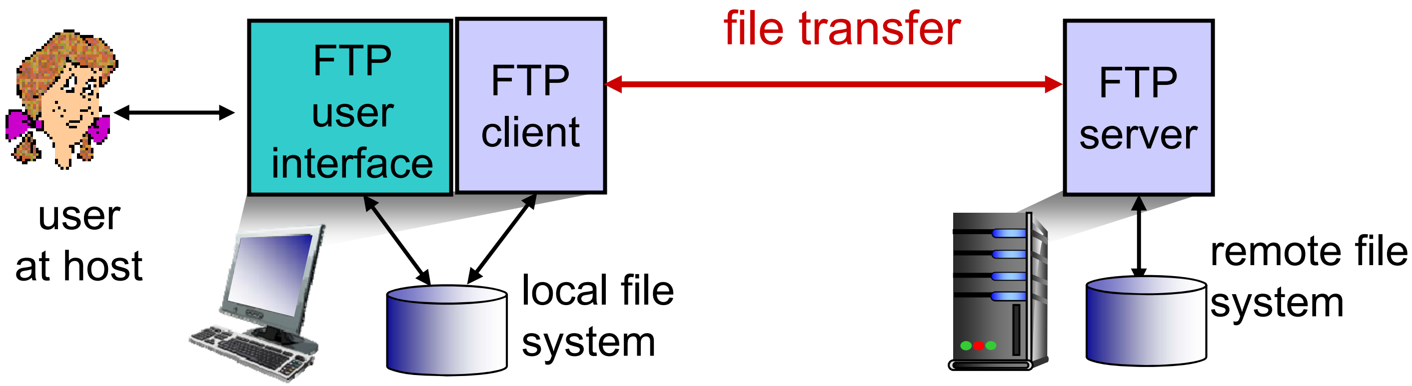
2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

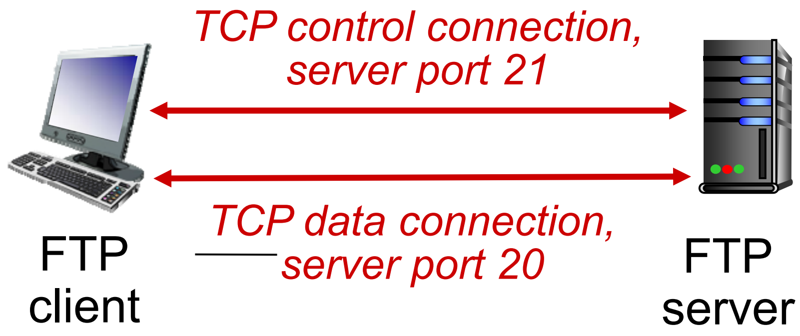
FTP: the file transfer protocol



- transfer file to/from remote host
- client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959
- ftp server: port 21

FTP: separate control, data connections

- FTP client contacts FTP server at port 21, using TCP
- client authorized over control connection
- client browses remote directory, sends commands over control connection
- when server receives file transfer command, *server* opens 2nd TCP data connection (for file) to client
- after transferring one file, server closes data connection



- server opens another TCP data connection to transfer another file
- control connection: *"out of band"*
- FTP server maintains "state": current directory, earlier authentication

FTP commands, responses

sample commands:

- sent as ASCII text over control channel
- **`USER username`**
- **`PASS password`**
- **`LIST`** return list of file in current directory
- **`RETR filename`** retrieves (gets) file
- **`STOR filename`** stores (puts) file onto remote host

sample return codes

- status code and phrase (as in HTTP)
- **`331 Username OK, password required`**
- **`125 data connection already open; transfer starting`**
- **`425 Can't open data connection`**
- **`452 Error writing file`**

Application layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Application layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

DNS: domain name system

people: many identifiers:
- name, passport #

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.yahoo.com - used by humans

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

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)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's "edge"

DNS: services, structure

DNS services

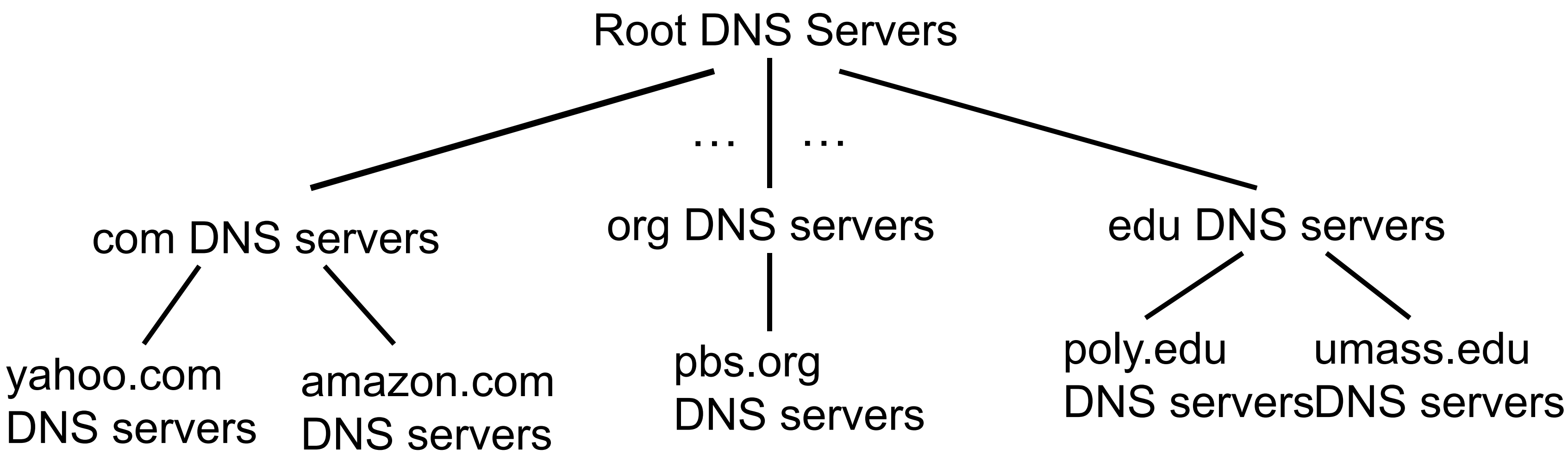
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

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

A: *doesn't scale!*

DNS: a distributed, hierarchical database

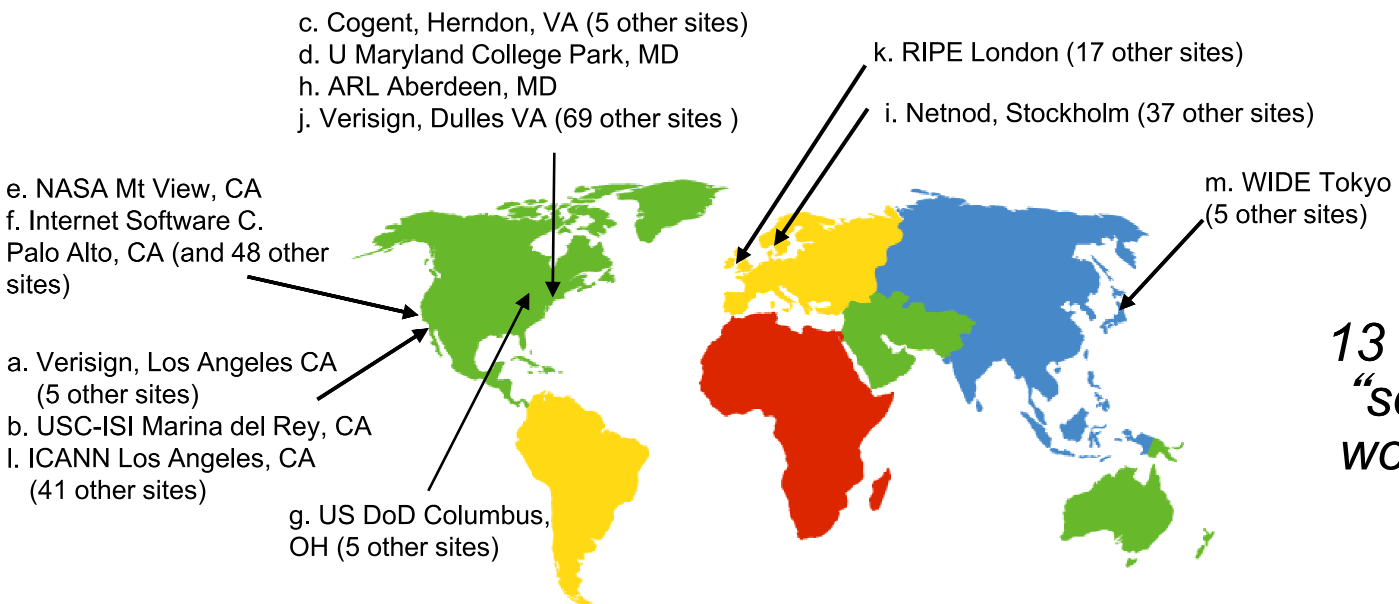


client wants IP for www.amazon.com; 1st approx:

- 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

- 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



13 root name "servers" worldwide

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.: uk, fr, ca, cn
- Network Solutions maintains servers for .com TLD
- Educause for .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 server

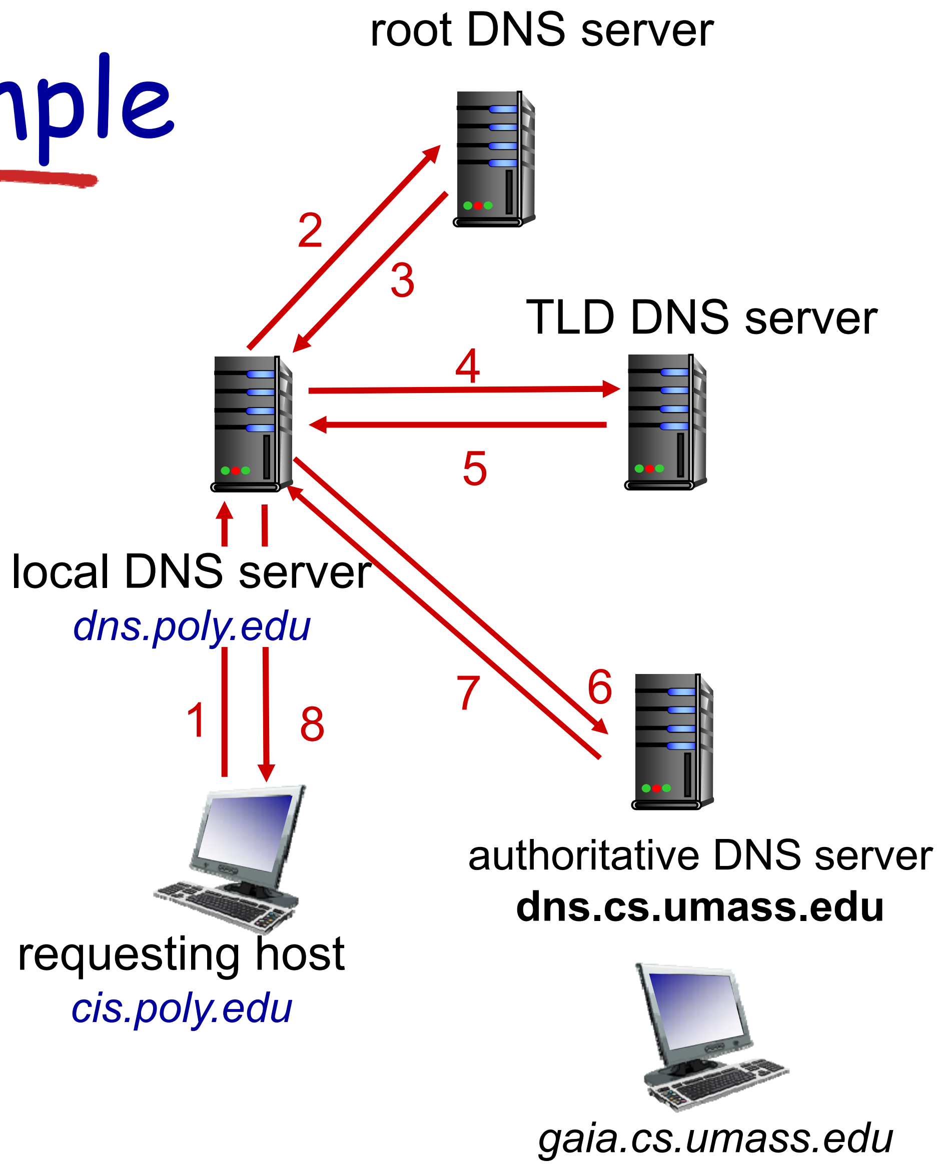
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

DNS name resolution example

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

iterated query:

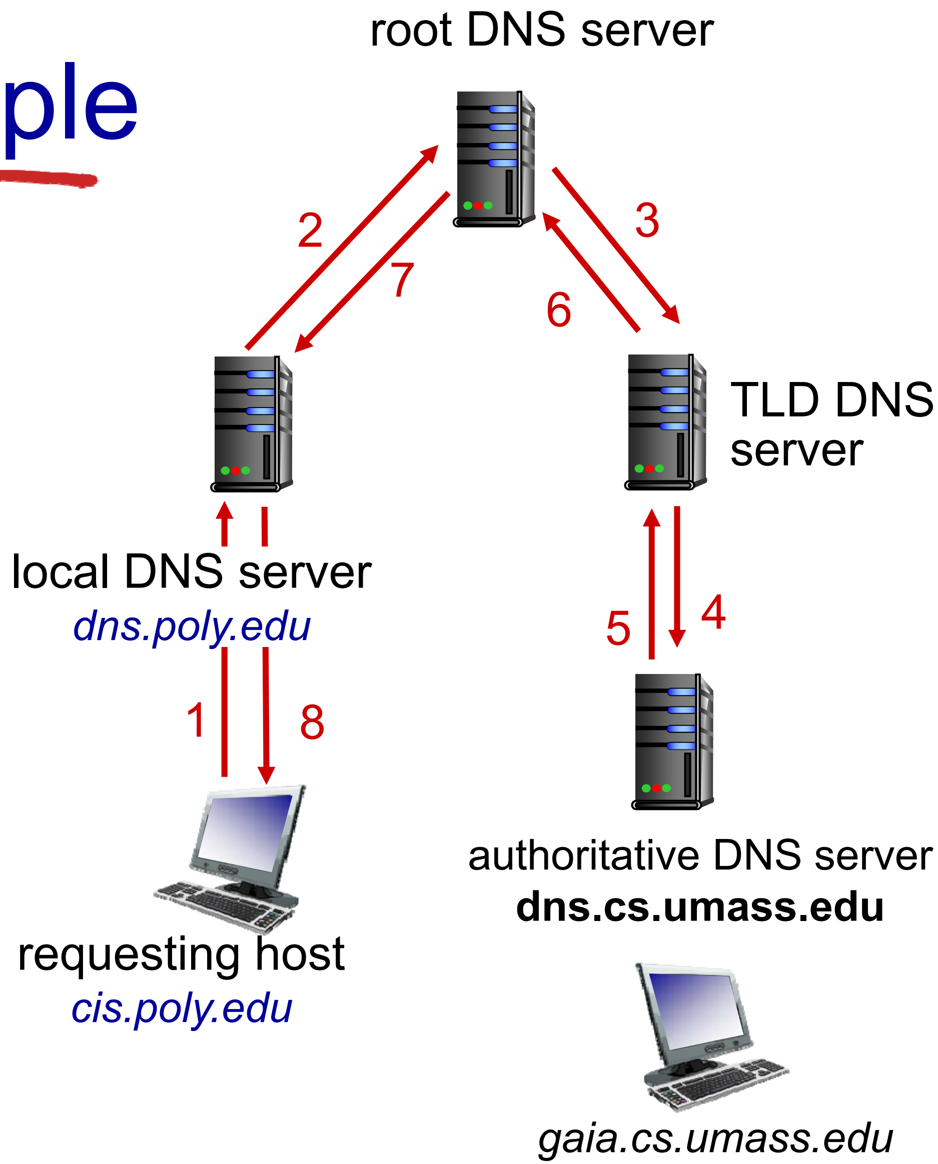
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

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 db 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 mailserver associated with **name**

DNS protocol, messages

- *query* and *reply* messages, both with same *message format*

msg header

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

2 bytes	2 bytes
identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

QR	opcode	AA	TC	RD	RA	(zero)	rcode
1	4	1	1	1	1	3	4

QR(1比特）：查询/响应的标志位，1为响应，0为查询。

opcode（4比特）：定义查询或响应的类型（若为0则表示是标准的，若为1则是反向的，若为2则是服务器状态请求）。

AA（1比特）：授权回答的标志位。该位在响应报文中有效，1表示名字服务器是权限服务器（关于权限服务器以后再讨论）

TC（1比特）：截断标志位。1表示响应已超过512字节并已被截断（依稀好像记得哪里提过这个截断和UDP有关，先记着）

RD（1比特）：该位为1表示客户端希望得到递归回答（递归以后再讨论）

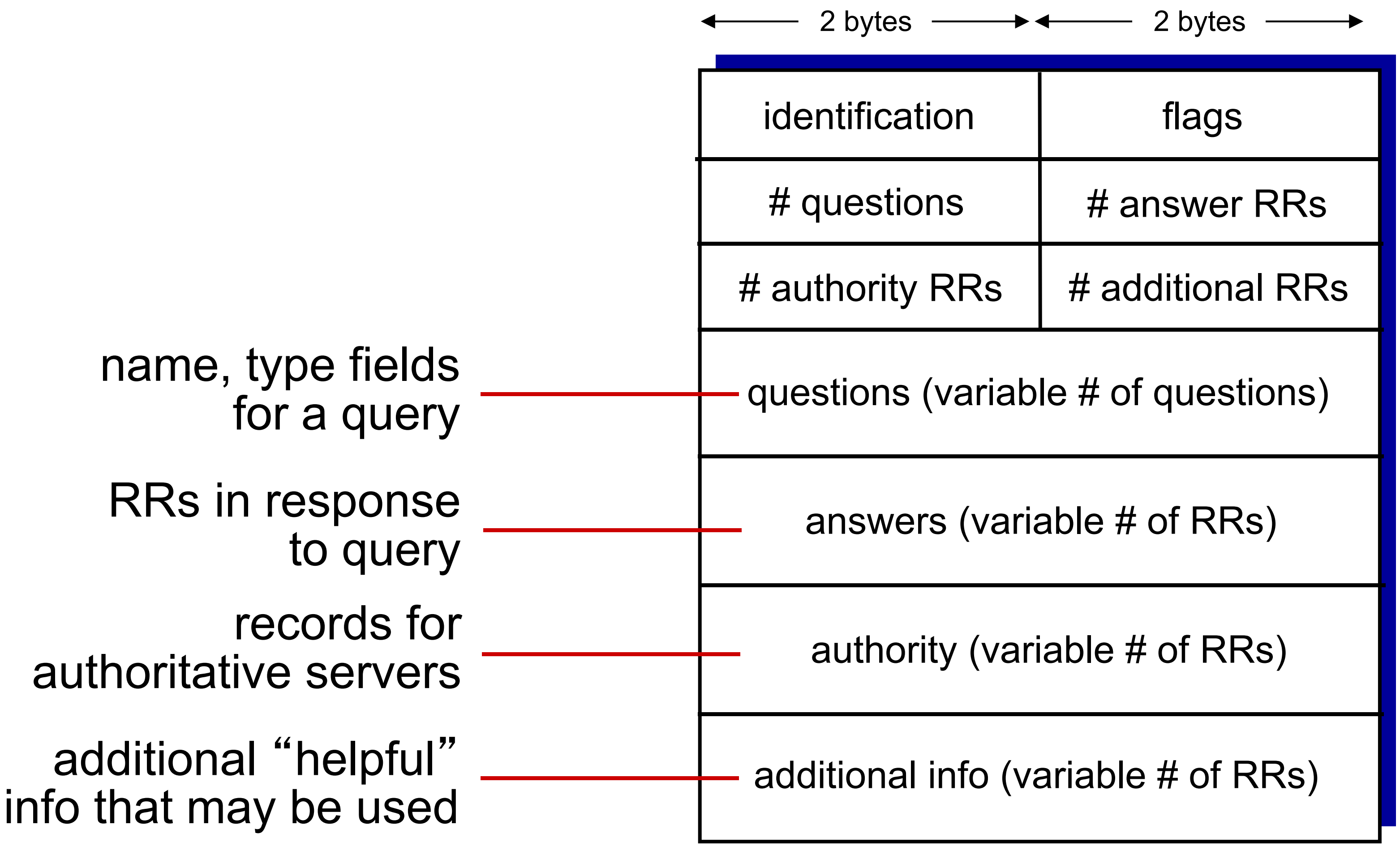
RA（1比特）：只能在响应报文中置为1，表示可以得到递归响应。

zero（3比特）：不说也知道都是0了，保留字段。

rcode（4比特）：返回码，表示响应的差错状态，通常为0和3，各取值含义如下：

0	无差错
1	格式差错
2	问题在域名服务器上
3	域参照问题
4	查询类型不支持
5	在管理上被禁止
6	-- 15 保留

DNS protocol, messages



Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two RRs into .com TLD server:
    (networkutopia.com, dns1.networkutopia.com, NS)

    (dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server type A record for www.networkuptopia.com; type MX record for networkutopia.com

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

Redirect attacks

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

Exploit DNS for DDoS

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

Application layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Defintion of P2P

1) Significant autonomy from central servers

2) Exploits resources at the edges of the Internet

   - storage and content
   - CPU cycles
   - human presence

3) Resources at edge have intermittent connectivity, being added & removed

It's a broad definition:

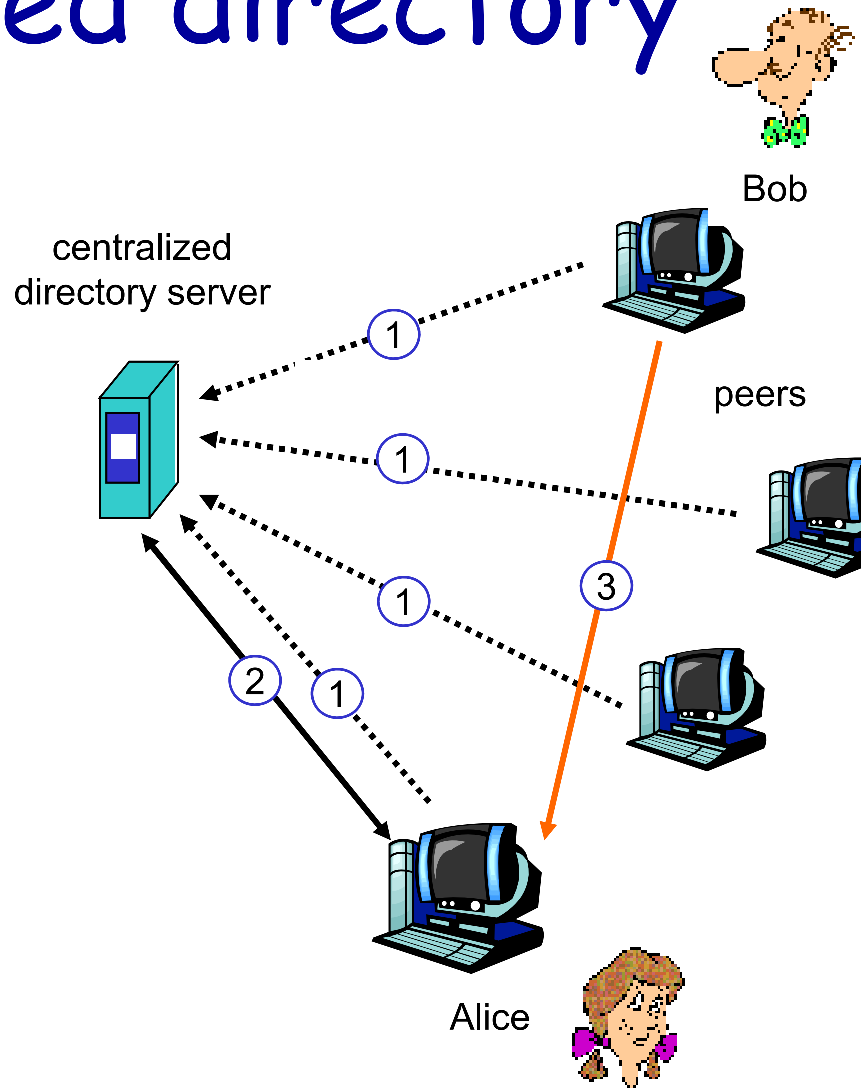
- P2P file sharing
  - Napster, Gnutella, KaZaA, eDonkey, etc
- P2P communication
  - Instant messaging
  - Voice-over-IP: Skype
- P2P computation
  - seti@home
- DHTs & their apps
  - Chord, CAN, Pastry, Tapestry
- P2P apps built over emerging overlays
  - PlanetLab

Wireless ad-hoc networking not covered here

P2P: centralized directory

original "Napster" design

1) when peer connects, it informs central server:
   - IP address
   - content
2) Alice queries for "Hey Jude"
3) Alice requests file from Bob



P2P: problems with centralized directory

- Single point of failure
- Performance bottleneck
- Copyright infringement

file transfer is decentralized, but locating content is highly centralized

Query flooding: Gnutella

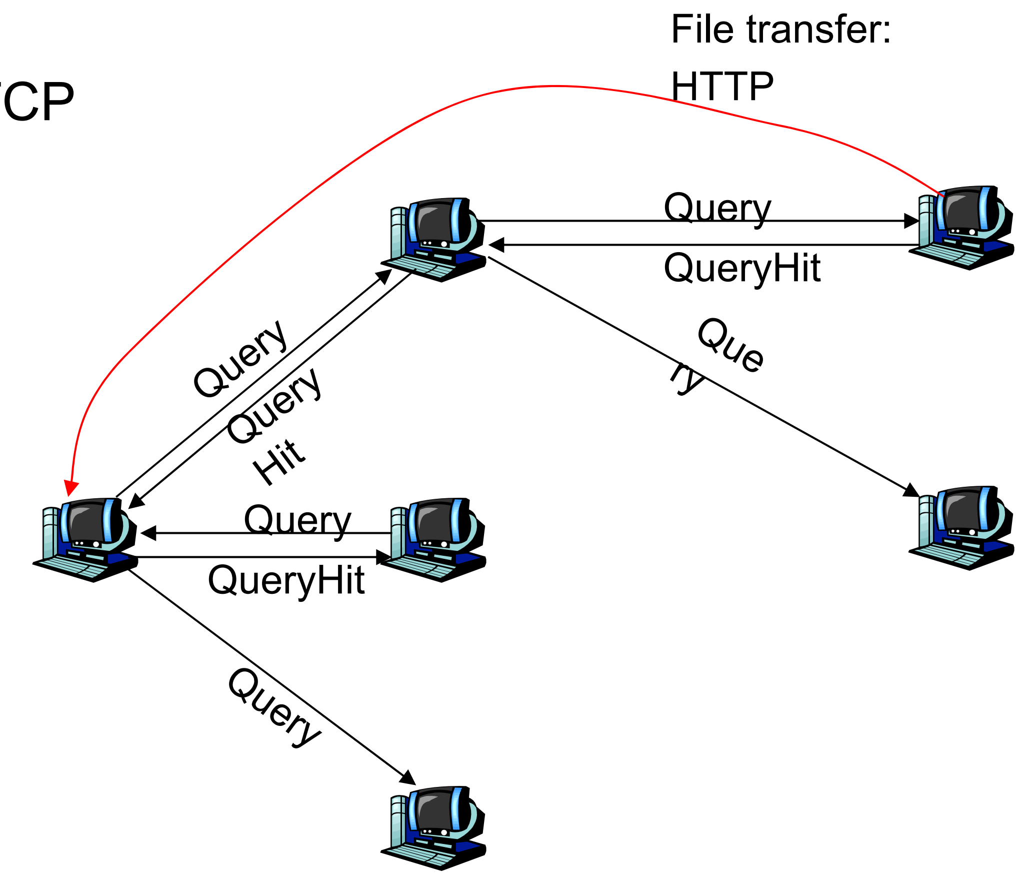
- fully distributed
  - no central server
- public domain protocol
- many Gnutella clients implementing protocol

overlay network: graph

- edge between peer X and Y if there's a TCP connection
- all active peers and edges is overlay net
- Edge is not a physical link
- Given peer will typically be connected with < 10 overlay neighbors

Gnutella: protocol

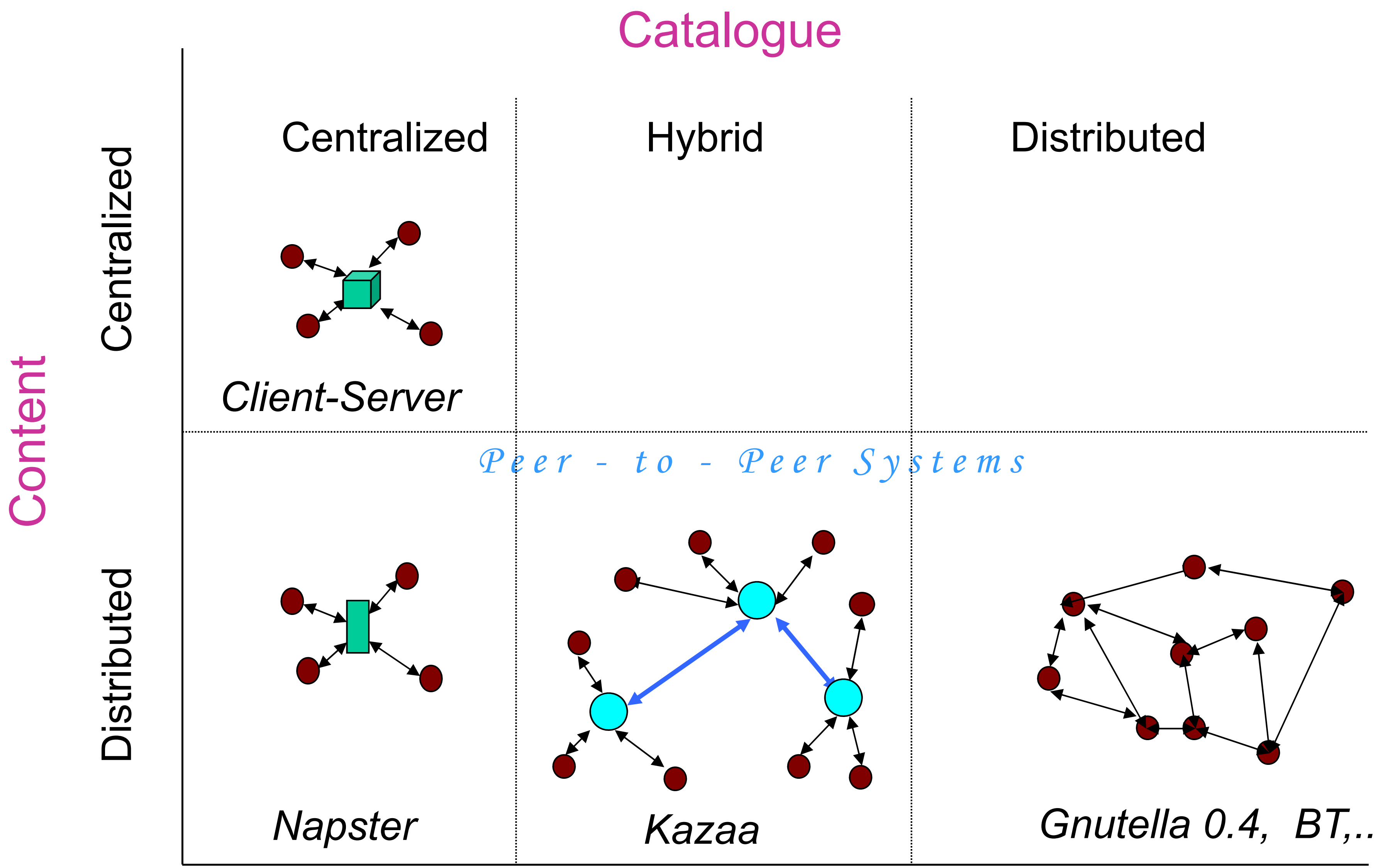
- Query message sent over existing TCP connections
- peers forward Query message
- QueryHit sent over reverse path



Scalability:

limited scope flooding

P2P Architectures

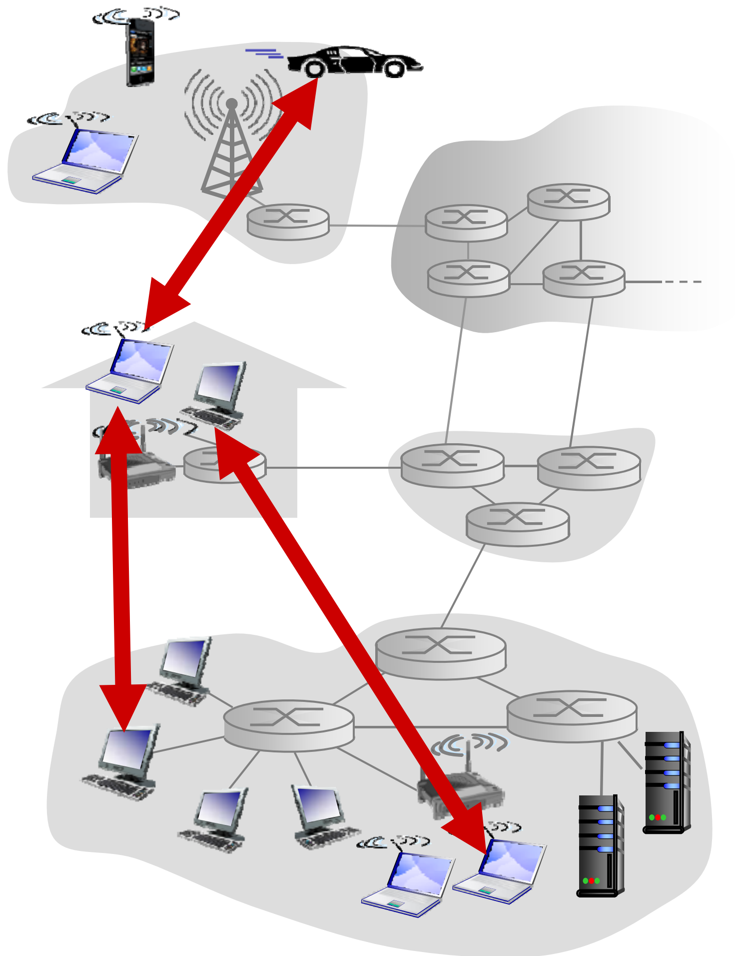


Pure P2P architecture

- *no always-on server*
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

examples:

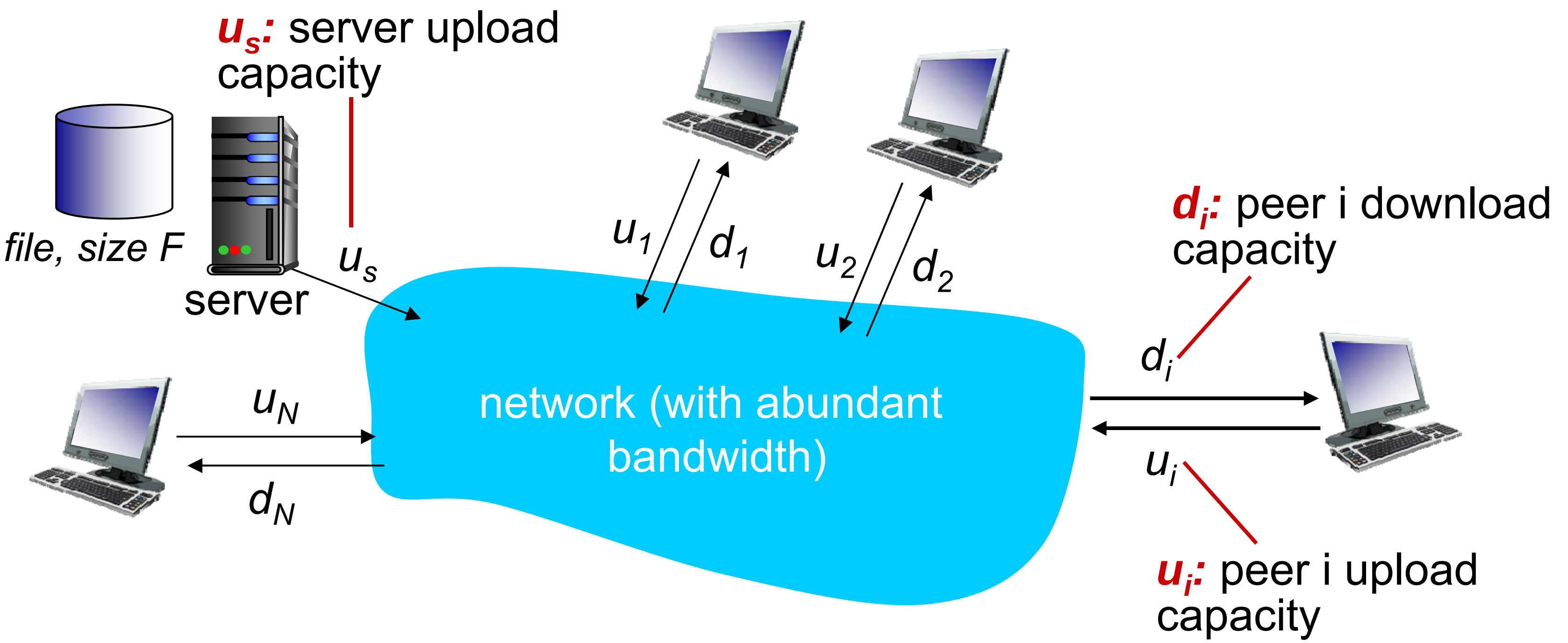
- file distribution (BitTorrent)
- Streaming (KanKan)
  - End-to-end multicast
- VoIP (Skype)



File distribution: client-server vs P2P

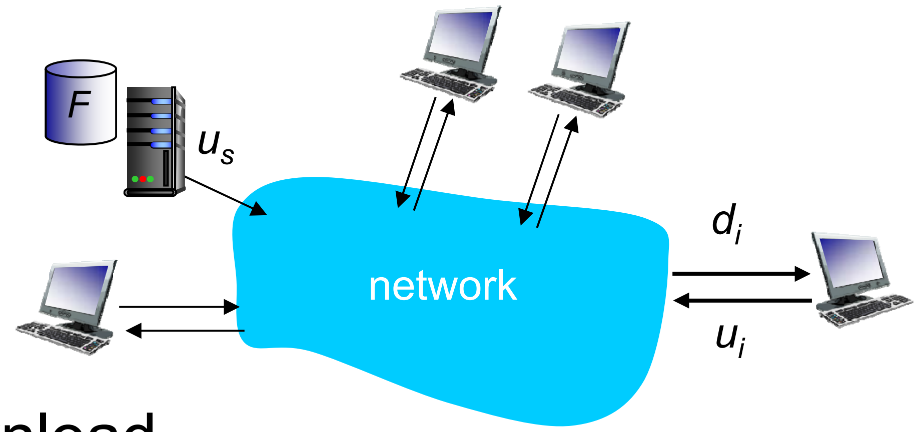
Question: how much time to distribute file (size *F*) from one server to *N* *peers*?

- 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}

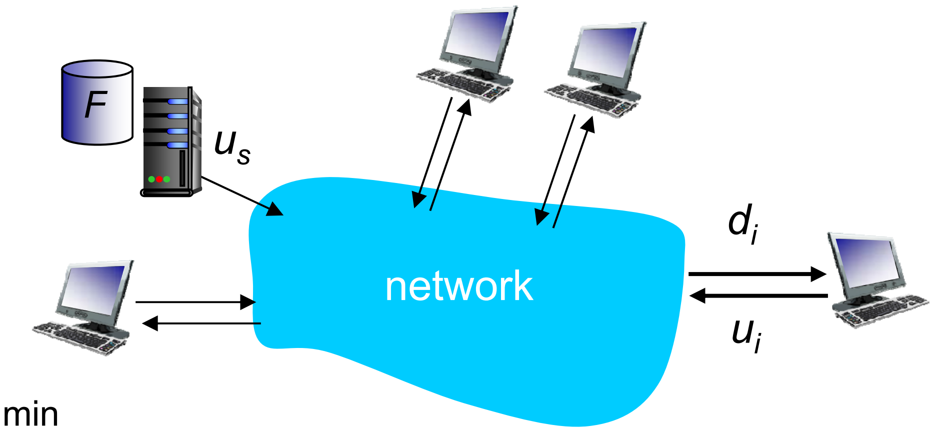
time to distribute F to N clients using client-server approach

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

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 (limting max download rate) is $u_s + \Sigma u_i$

time to distribute F to N clients using P2P approach

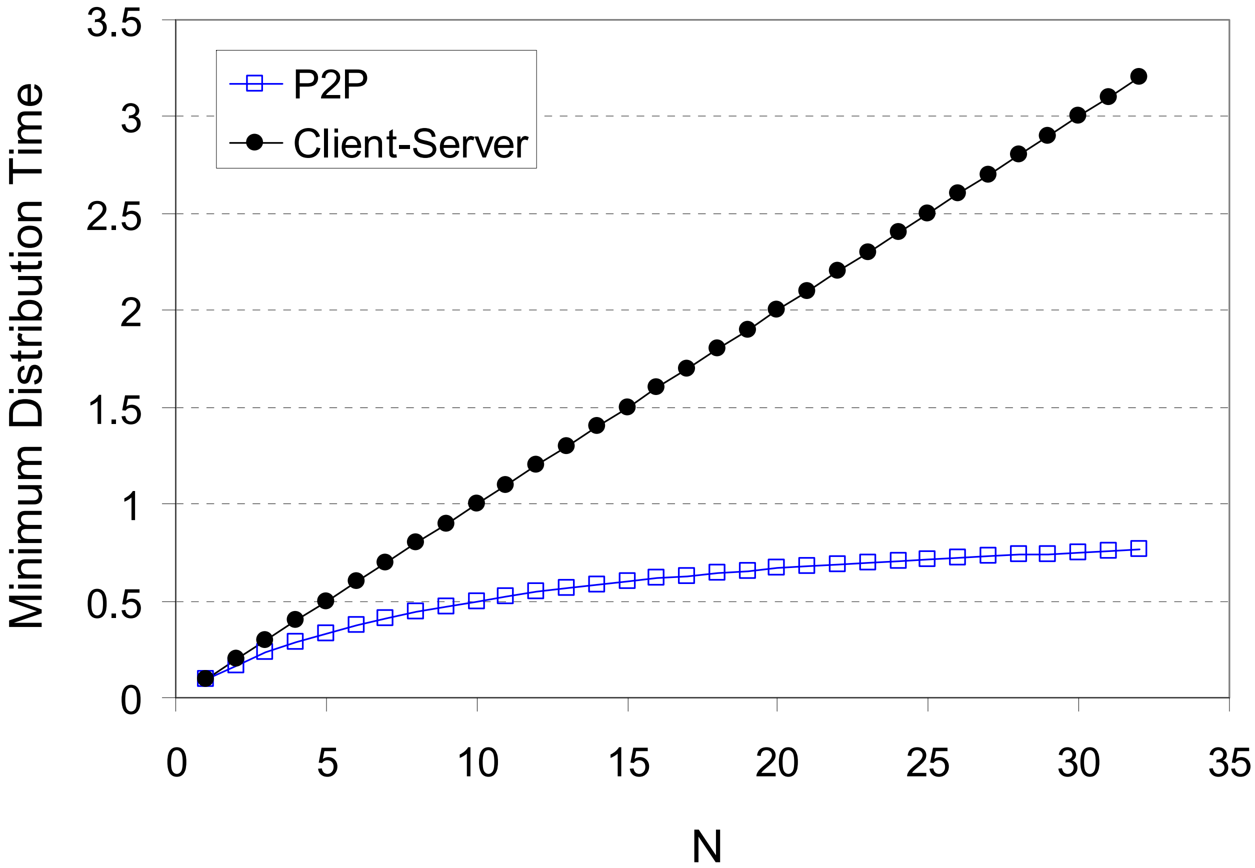
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

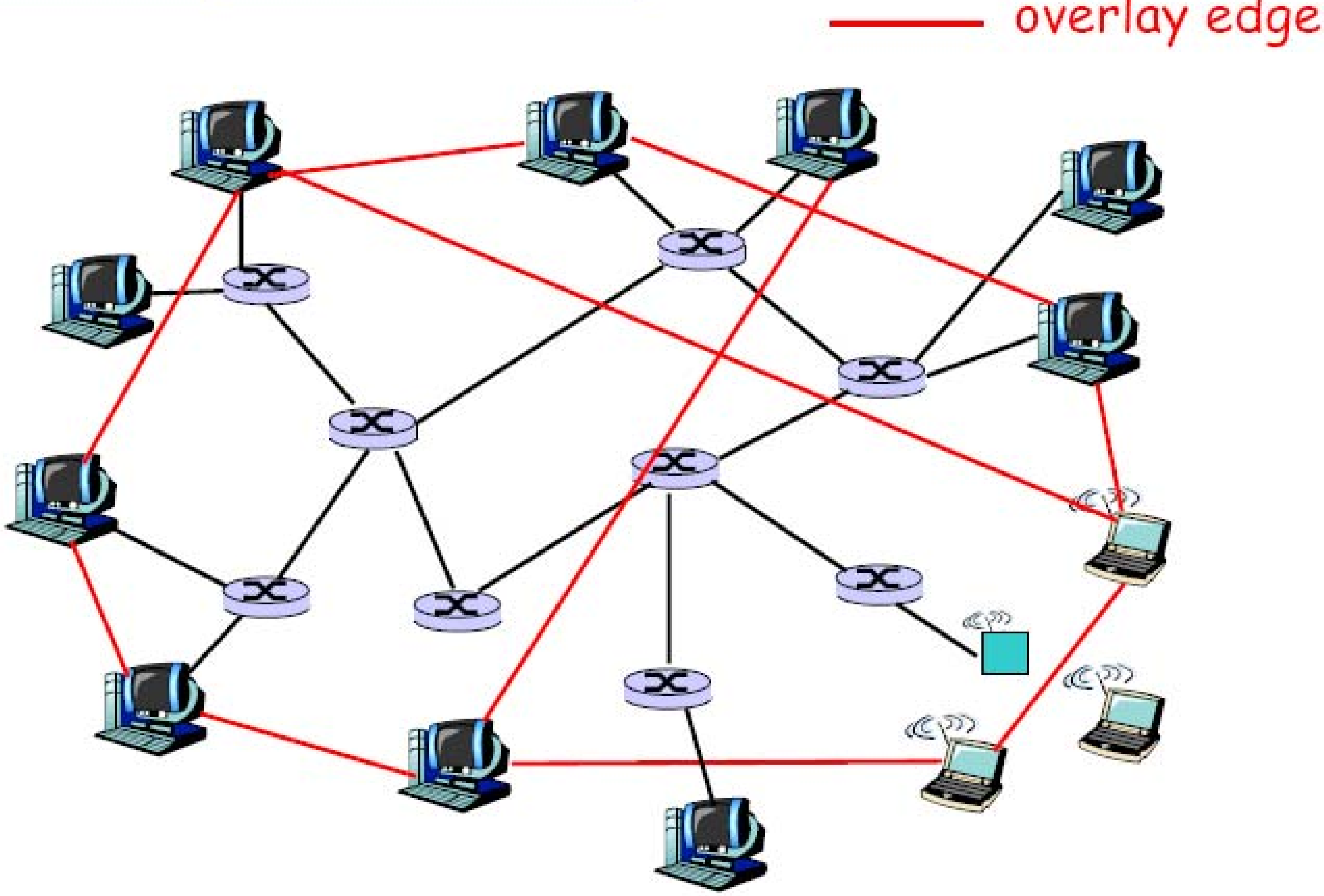
… but so does this, as each peer brings service capacity

Client-server vs. P2P: example

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



Overlay networks



Overlay graph

Virtual edge

- TCP connection
- or simply a pointer to an IP address

Overlay maintenance

- Periodically ping to make sure neighbor is still alive
- Or verify liveness while messaging
- If neighbor goes down, may want to establish new edge
- New node needs to bootstrap

More about overlays

Unstructured overlays

- e.g., new node randomly chooses three existing nodes as neighbors

Structured overlays

- e.g., edges arranged in restrictive structure

Proximity

- Not necessarily taken into account

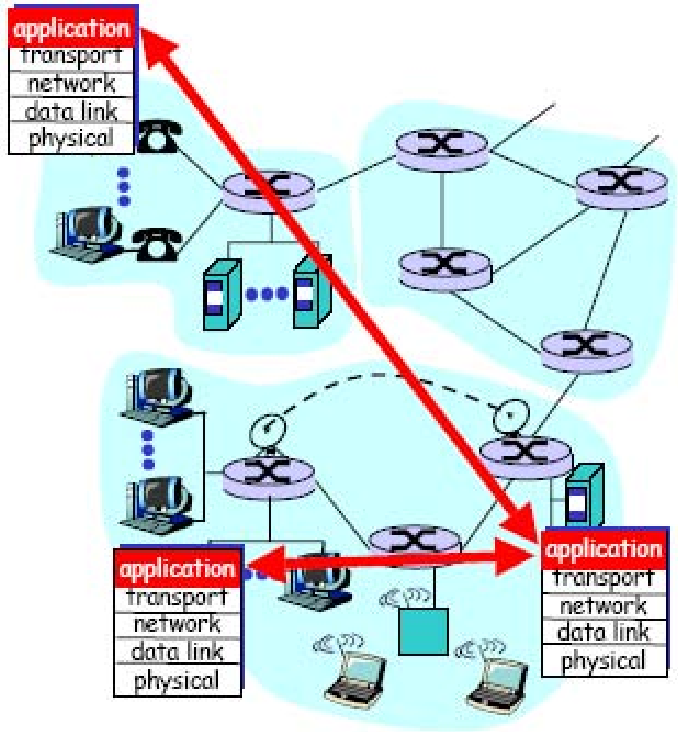
Overlays: all in the application layer

Tremendous design flexibility

- Topology, maintenance
- Message types
- Protocol
- Messaging over TCP or UDP

Underlying physical net is transparent to developer

- But some overlays exploit proximity



10

Examples of overlays

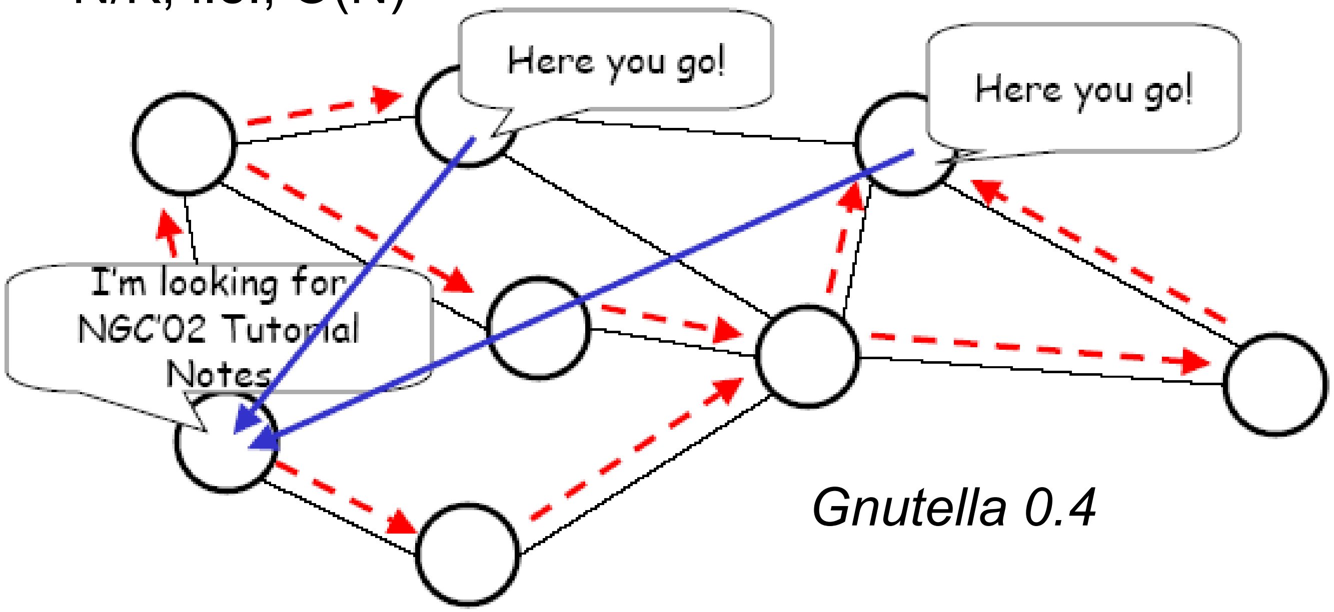
- DNS
- BGP routers and their peering relationships
- Content distribution networks (CDNs)
- Application-level multicast
  - economical way around barriers to IP multicast

- And P2P apps !

Structured P2P systems

- In Unstructured P2P
  - Simplest strategy: expanding ring search
  - Need many cached copies to keep search overhead small

If K of N nodes have copy, expected search cost at least: N/K, i.e., O(N)



Gnutella 0.4

Structured P2P systems

- Directed Searches

Idea:

- assign particular nodes to hold particular content (or pointers to it, like an information booth)
- when a node wants that content, go to the node that is supposed to have or know about it

Challenges:

- Distributed: want to distribute responsibilities among existing nodes in the overlay
- Adaptive: nodes join and leave the P2P overlay
  distribute knowledge responsibility to joining nodes
  redistribute responsibility knowledge from leaving nodes

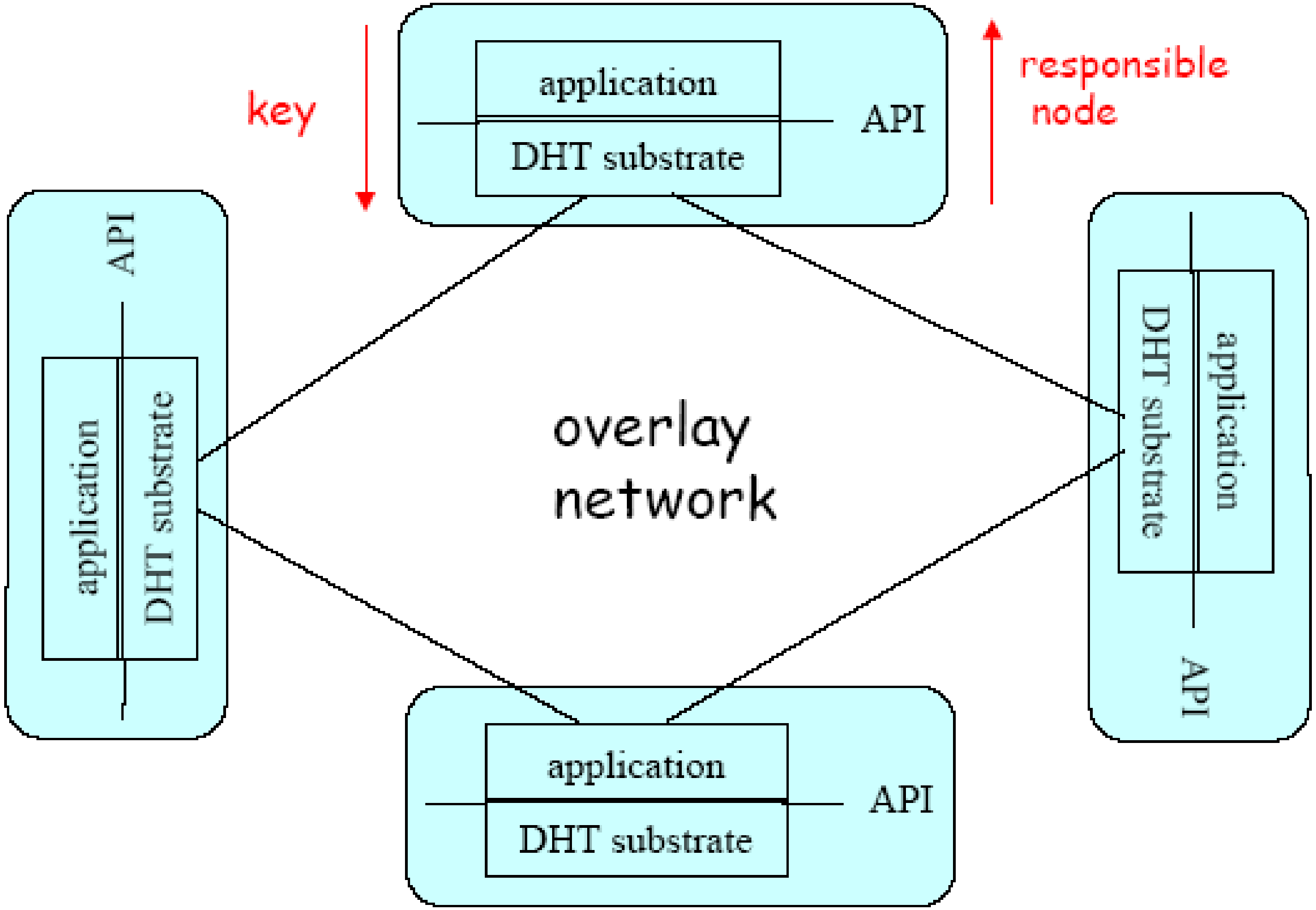
Often using DHT(Distributed Hash Table)

DHT API

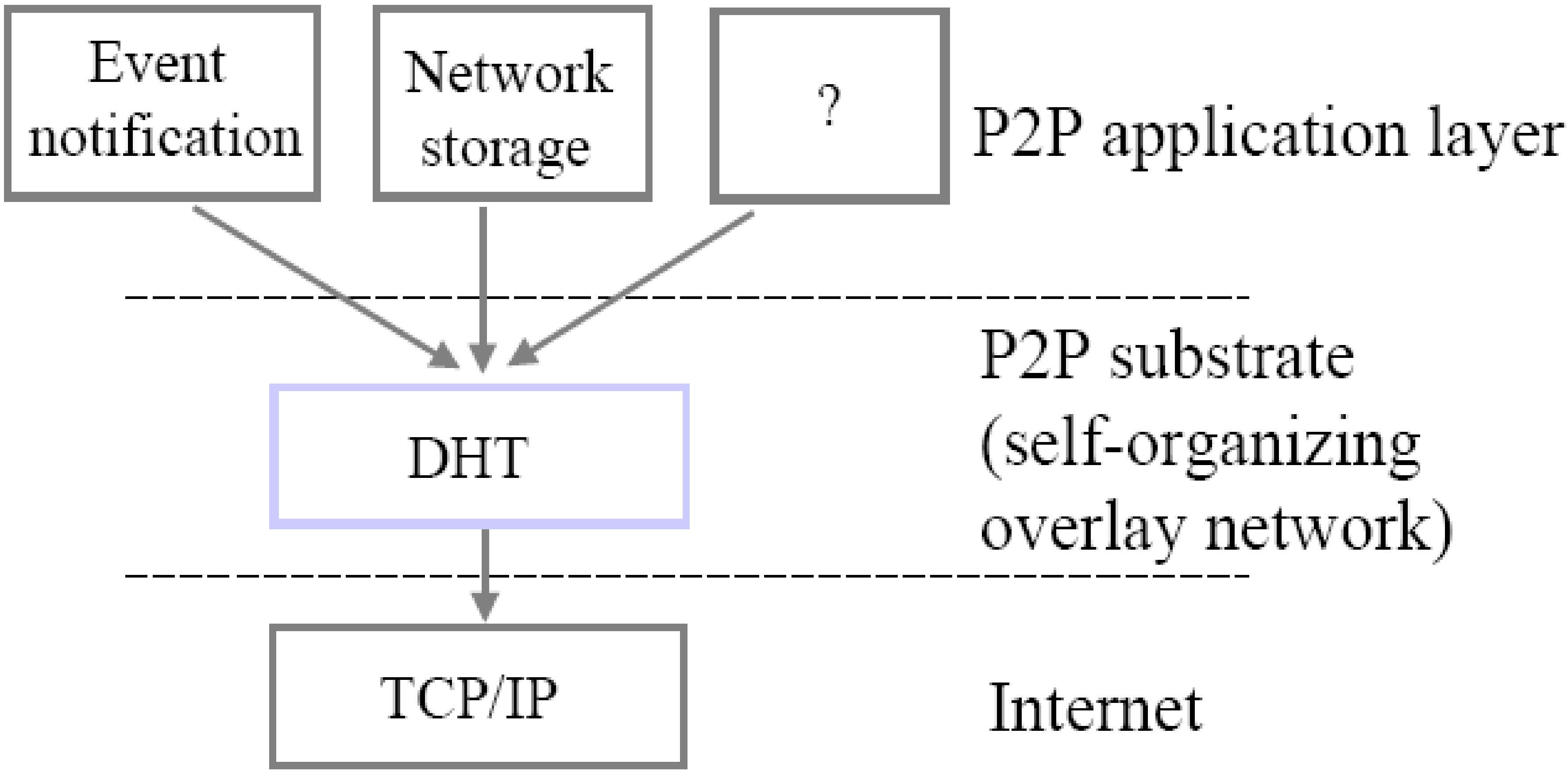
- each data item (e.g., file or metadata containing pointers) has a key in some ID space
- In each node, DHT software provides API:
  - Application gives API key k
  - API returns IP address of node that is responsible for k
- API is implemented with an underlying DHT overlay and distributed algorithms

DHT API

each data item (e.g., file or metadata pointing to file copies) has a key

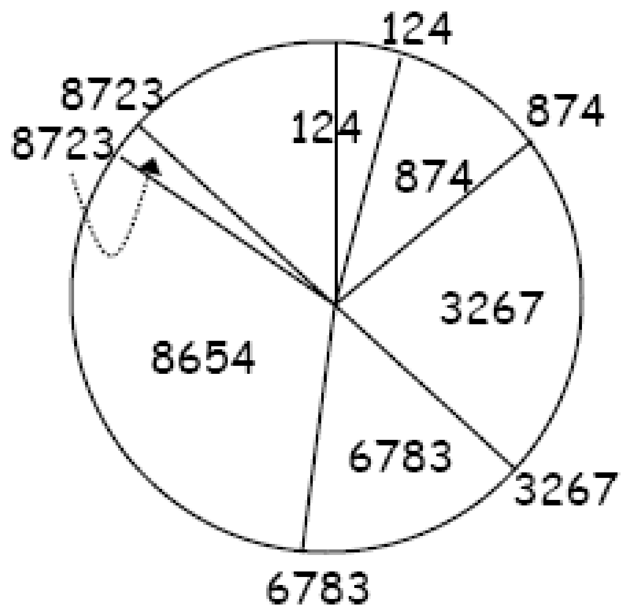


DHT Layered Architecture



Chord

- Nodes assigned 1-dimensional IDs in hash space at random (e.g., hash on IP address)
- Consistent hashing: Range covered by node is from previous ID up to its own ID (modulo the ID space)



Chord Routing

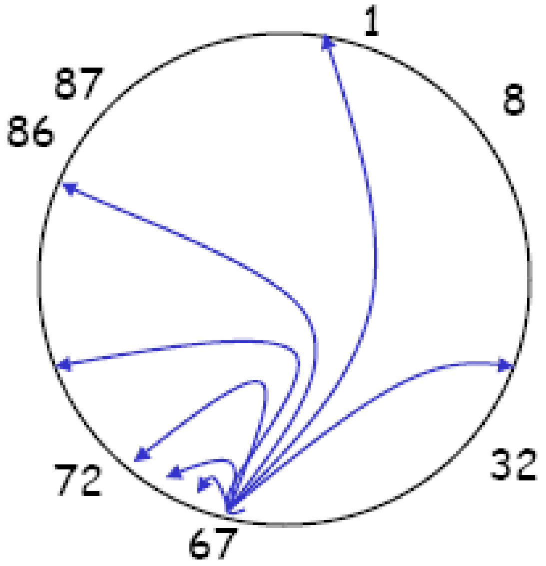
- A node s's i^{th} neighbor has the ID that is equal to $s+2^i$ or is the next largest ID (mod ID space), $i \geq 0$
- To reach the node handling ID t, send the message to neighbor $\#\log_2(t-s)$
- Requirement: each node s must know about the next node that exists clockwise on the Chord (0^{th} neighbor)
- Set of known neighbors called a finger table

Chord Routing (cont'd)

- A node s is node t's neighbor if s is the closest node to $t+2^i \bmod H$ for some i. Thus,
  - each node has at most $\log_2 N$ neighbors
  - for any object, the node whose range contains the object is reachable from any node in no more than $\log_2 N$ overlay hops

    (each step can always traverse at least half the distance to the ID)

- When a new node joins or leaves the overlay, O(K / N) objects move between nodes



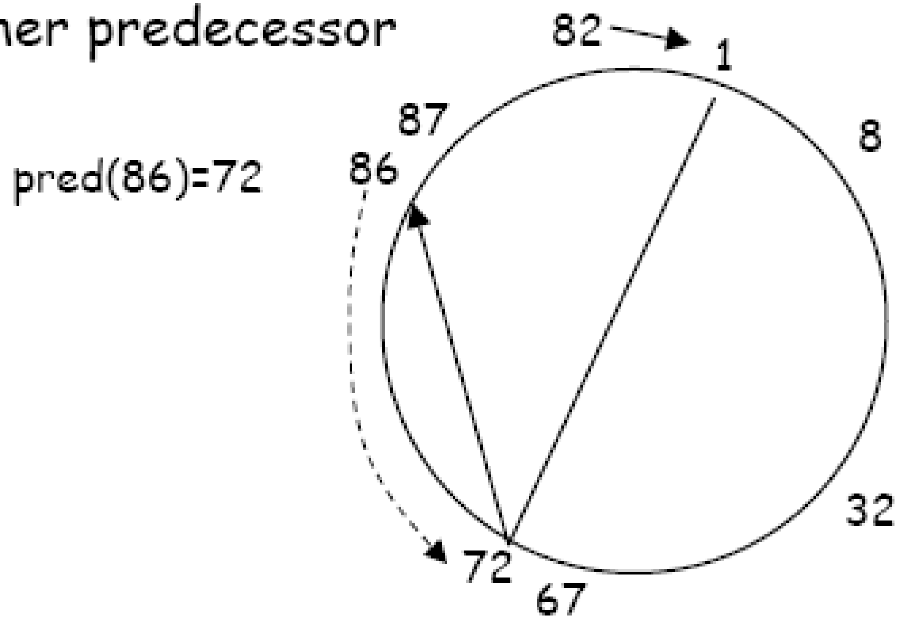
i	Finger table for node 67
0	72
1	72
2	72
3	86
4	86
5	1
6	32

Closest node clockwise to

$67+2^i$ mod 100

Chord Node Insertion

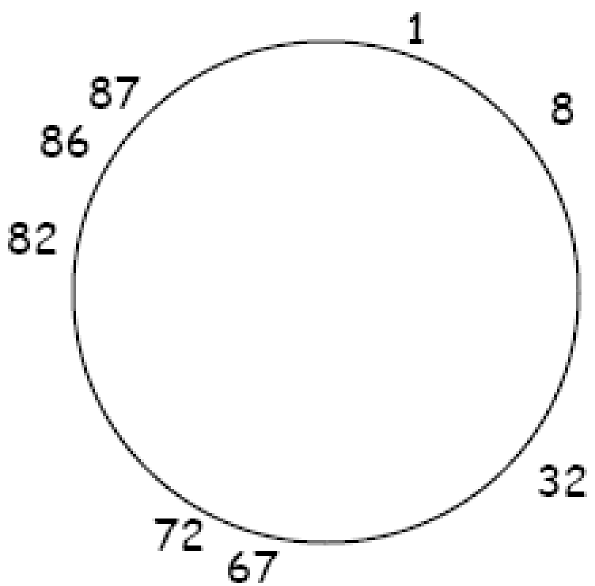
- One protocol addition: each node knows its closest counter-clockwise neighbor
- A node selects its unique (pseudo-random) ID and uses a bootstrapping process to find some node in the Chord
- Using Chord, the node identifies its successor in the clockwise direction
- An newly inserted node's predecessor is its successor's former predecessor



Example: Insert 82

Chord Node Insertion (cont'd)

- First: set added node s's fingers correctly
  - s's predecessor t does the lookup for each distance of 2^i from s



Lookups from node 72

Lookup(83) = 86

Lookup(84) = 86

Lookup(86) = 86

Lookup(90) = 1

Lookup(98) = 1

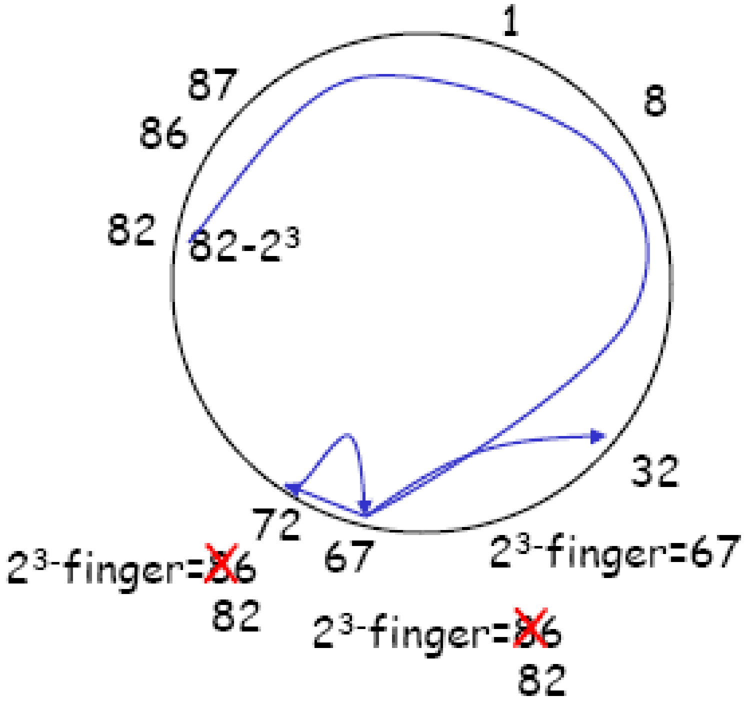
Lookup(14) = 32

Lookup(46) = 67

i	Finger table for node 82
0	86
1	86
2	86
3	1
4	1
5	32
6	67

Chord Node Insertion (cont'd)

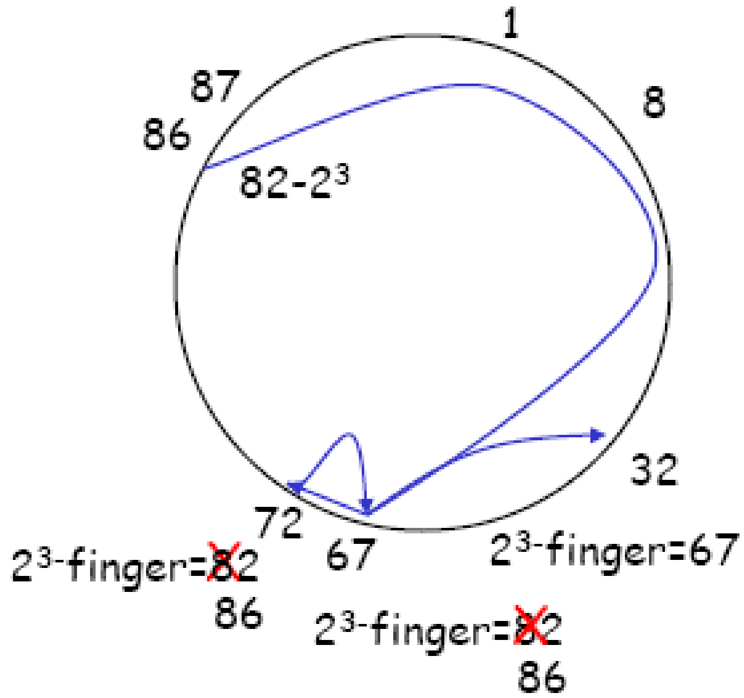
- Next, update other nodes' fingers about the entrance of s (when relevant). For each i:
  - Locate the closest node to s (counter-clockwise) whose 2^i-finger can point to s: largest possible is s - 2^i
  - Use Chord to go (clockwise) to largest node t before or at s - 2^i
    - route to s - 2^i, if arrived at a larger node, select its predecessor as t
  - If t's 2^i-finger routes to a node larger than s
    - change t's 2^i-finger to s
    - set t = predecessor of t and repeat
  - Else i++, repeat from top
- $O(\log^2 N)$ time to find and update nodes



e.g., for i=3

Chord Node Deletion

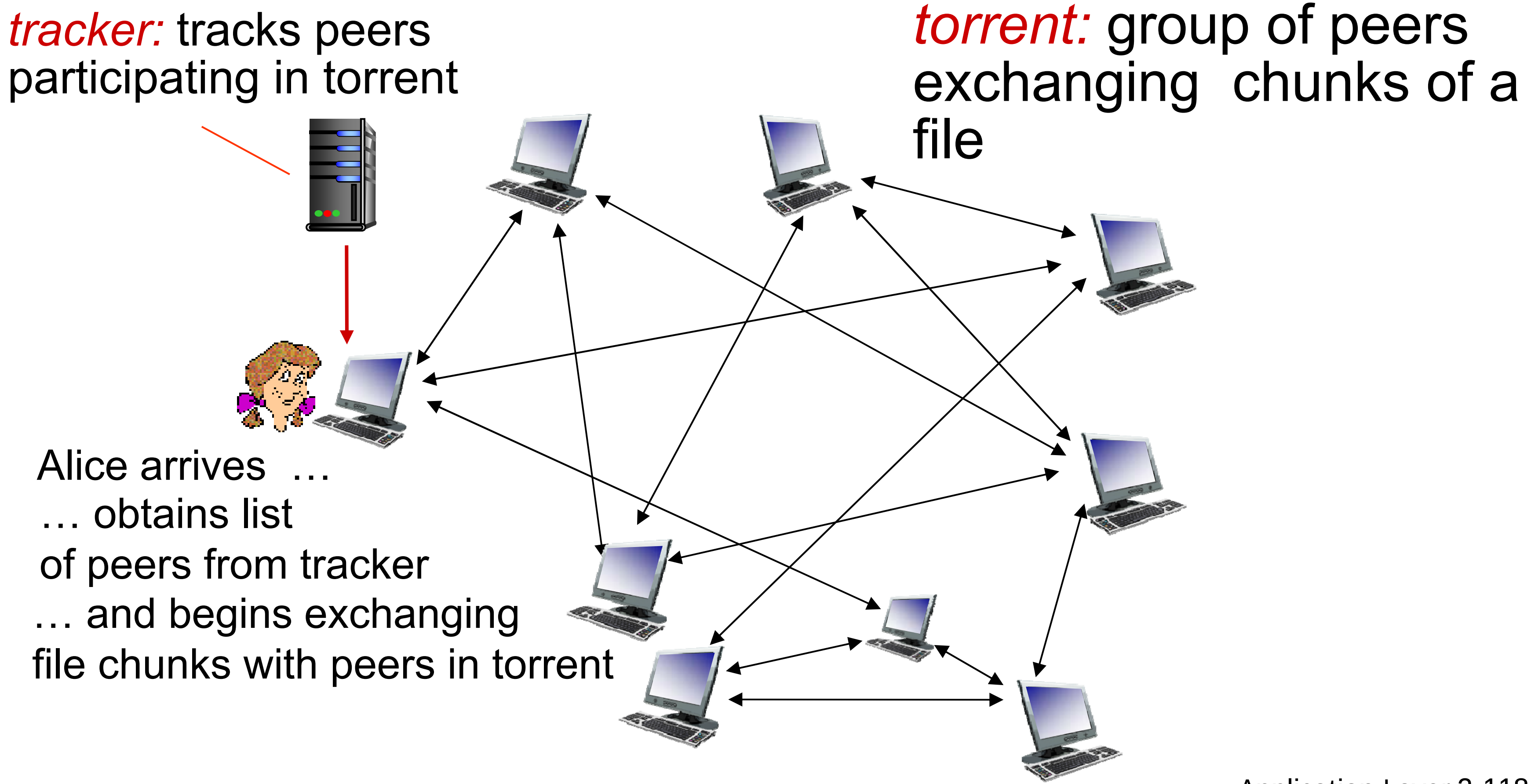
- Similar process can perform deletion



e.g., for i=3

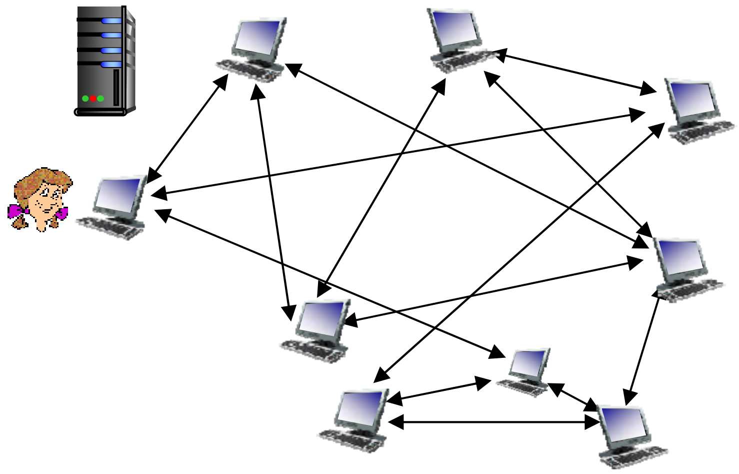
P2P file distribution: BitTorrent

- DHT: Implementing Kademlia
- 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:

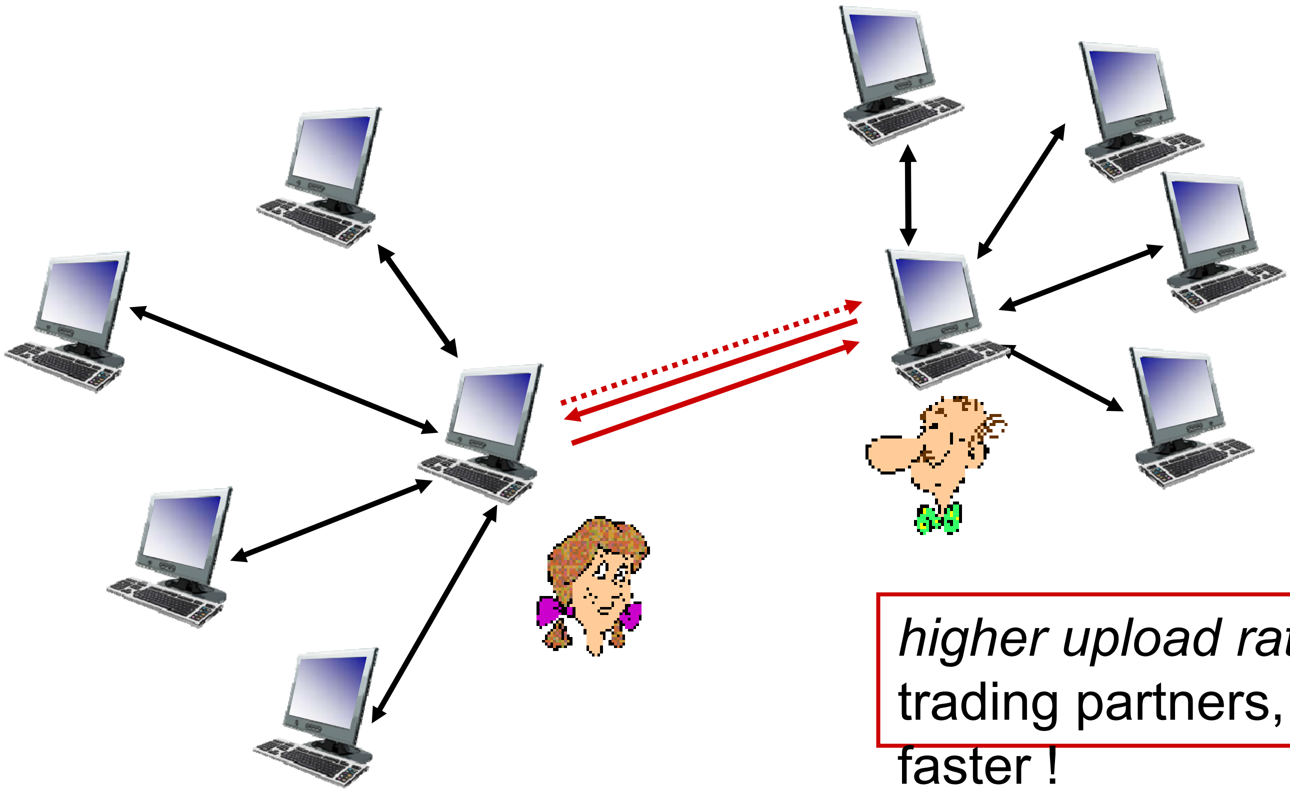
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

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 !

Application layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

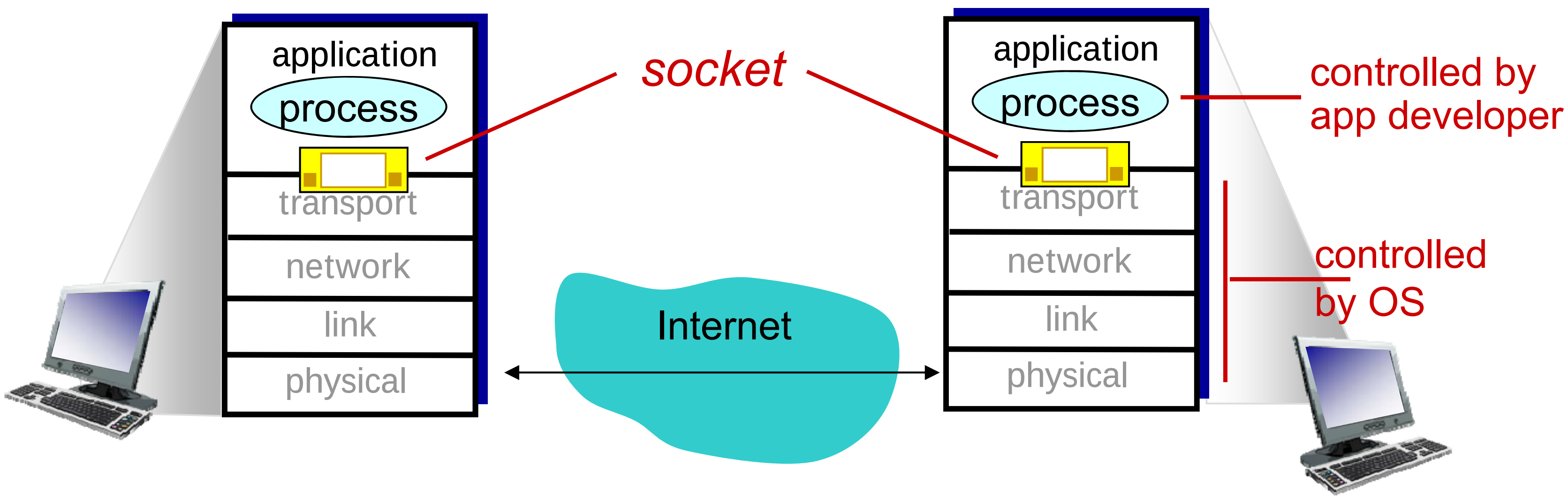
2.6 P2P applications

2.7 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 the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the 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
- rcvr 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(socket.AF_INET,
                      socket.SOCK_DGRAM)
```

get user keyboard input

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message,(serverName, serverPort))
```

read reply characters from socket into string

```
modifiedMessage, serverAddress =
                      clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print modifiedMessage
clientSocket.close()
```

Example app: UDP server

Python UDPServer

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

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, 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

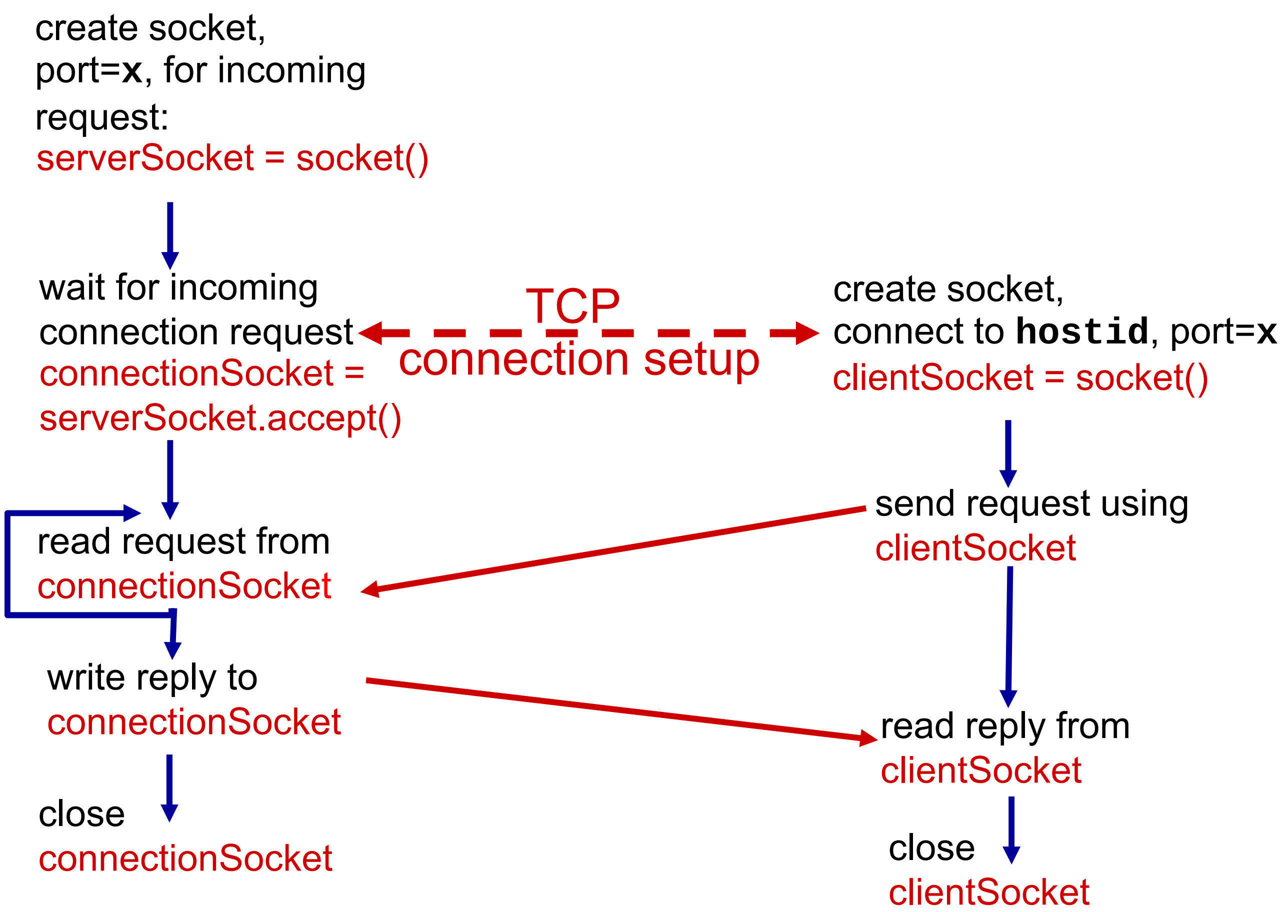
- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

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

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 = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

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 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    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

our study of network apps 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
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, DHT
- socket programming: TCP, UDP sockets

Application layer: summary

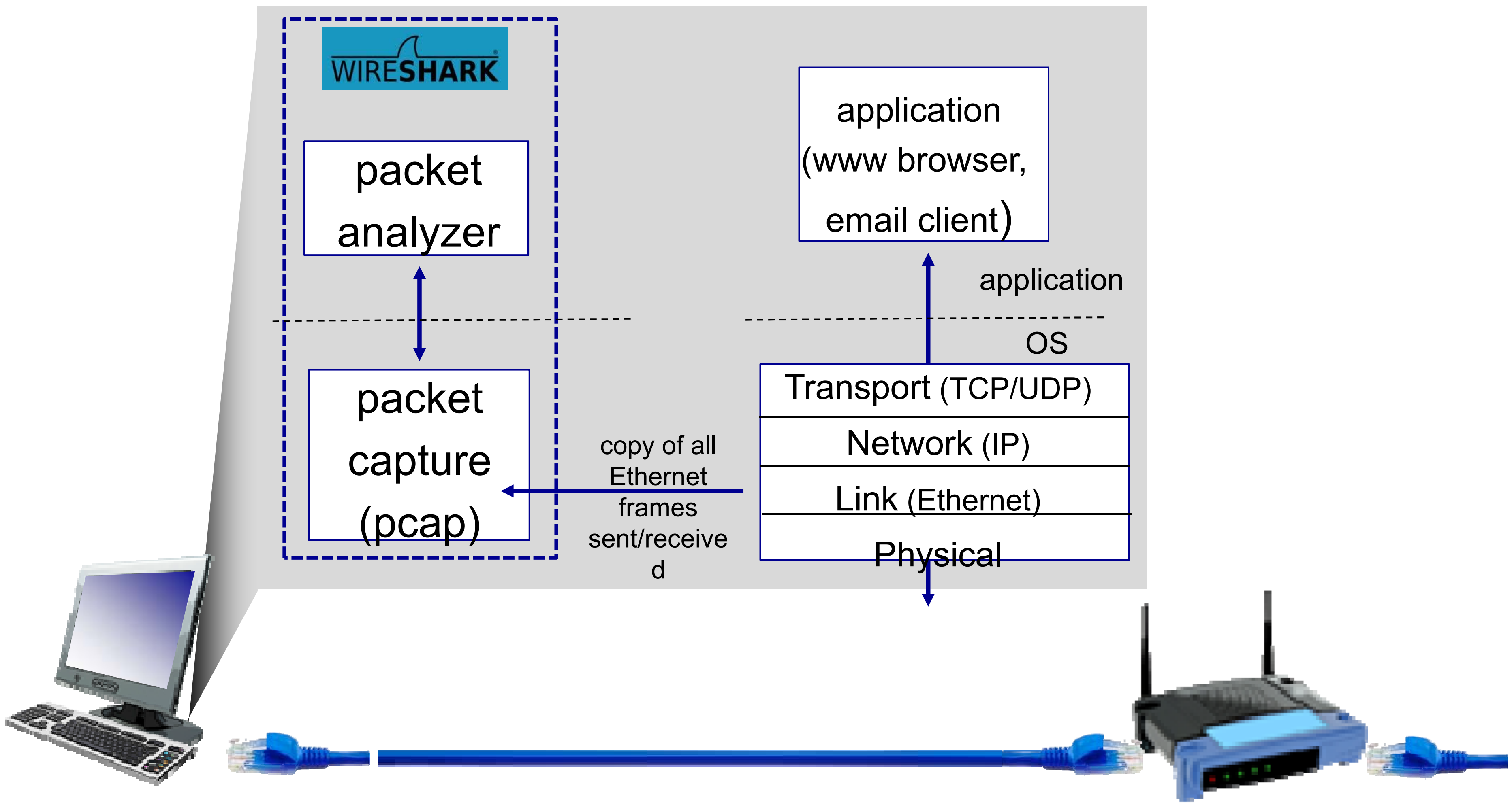
most importantly: learned about protocols!

- 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 being communicated

important themes:

- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"

Chapter 1
Additional Slides



WIRESHARK
packet analyzer
packet capture (pcap)
copy of all Ethernet frames sent/received
application (www browser, email client)
application
OS
Transport (TCP/UDP)
Network (IP)
Link (Ethernet)
Physical