Chapter 2
Application layer

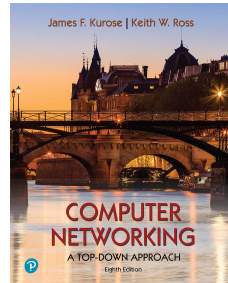*Computer Networking: A Top-Down Approach*
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Chapter 2: 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, IMAP
  - ▪ DNS
  - ▪ video streaming systems, CDNs
- ❖ creating network applications
  - ▪ socket API

Chapter 2: outline

2.1. Principles of network applications
　2.1.1. Network application architectures
　2.1.2. Communicating between processes
　2.1.3. Transport services
2.2. The Web and HTTP
2.3. FTP
2.4. Electronic mail

2.5. DNS (Domain Name Systems)
2.6. Peer-to-peer applications
2.7. Video streaming and content distribution networks
2.8. Socket programming with UDP and TCP

Some network apps

- ❖ social networking
- ❖ web
- ❖ text messaging
- ❖ e-mail
- ❖ multi-user network games
- ❖ streaming stored video (YouTube, Hulu, Netflix)
- ❖ P2P file sharing

- ❖ voice over IP (e.g., Skype)
- ❖ real-time video conferencing
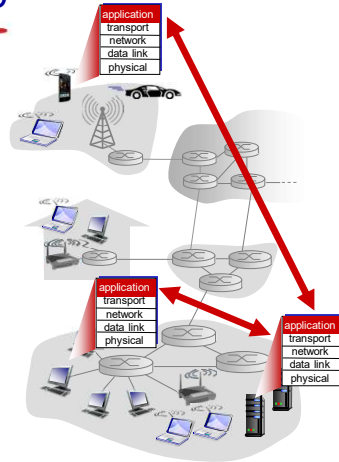- ❖ Internet search
- ❖ remote login
- ❖ …

Q: *your* favorites?

Creating a network app

write programs that:

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

no need to write software for network-core devices

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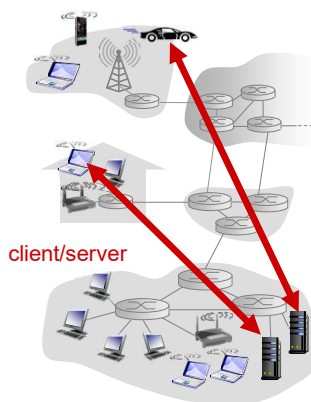- ❖ applications on end systems allows for rapid app development, propagation

application
transport
network
data link
physical

application
transport
network
data link
physical

application
transport
network
data link
physical

2-5

Application architectures

possible structure of applications:

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

2-6

Client-server architecture

server:

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

clients:

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

examples:

- ❖ HTTP, IMAP, FTP

client/server

2-7

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
- ❖ example: P2P file sharing [BitTorrent]

peer-peer

2-8

2

Processes communicating

process: program running within a host

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

clients, servers
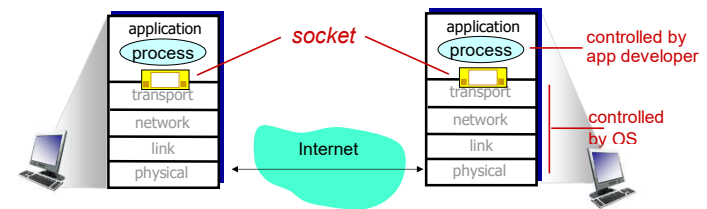
client process: process that initiates communication

server process: process that waits to be contacted

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

Sockets

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

Addressing processes

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

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

An application-layer protocol defines

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

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

proprietary protocols:
- e.g., Skype, Zoom

What transport service does an app need?

data integrity

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

timing

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

throughput

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

security

❖ encryption, data integrity, …

2-13

Transport service requirements: common apps

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

2-14

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?

2-15

Internet applications and transport protocols

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

2-16

4

Securing TCP

Vanilla TCP & UDP sockets:
- no encryption
- Clear-text passwords sent into socket traverse Internet in clear-text

Transport Layer Security (TLS)
- provides encrypted TCP connection
- data integrity
- end-point authentication

TLS implemented in application layer
- Apps use TSL libraries, that use TCP in turn
- Clear-text passwords sent into socket traverse Internet encrypted

Chapter 2: outline

2.1. Principles of network applications
  2.1.1. Network application architectures
  2.1.2. Communicating between processes
  2.1.3. Transport services
2.2. The Web and HTTP
2.3. FTP
2.4. Electronic mail

2.5. DNS (Domain Name Systems)
2.6. Peer-to-peer applications
2.7. Video streaming and content distribution networks
2.8. Socket programming with UDP and TCP

Web and HTTP

First, a quick 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.,

```
www.someschool.edu/someDept/pic.gif
```

host name   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

PC running Firefox browser

HTTP request
HTTP response

HTTP request
HTTP response

server running Apache Web server

iphone running Safari browser

HTTP overview (cont.)

HTTP 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

2-21

HTTP connections: two types

non-persistent HTTP

- TCP connection opened
- at most one object sent over TCP connection
- TCP connection then closed

➔ downloading multiple objects required multiple connections

persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over single TCP connection between client, and that server
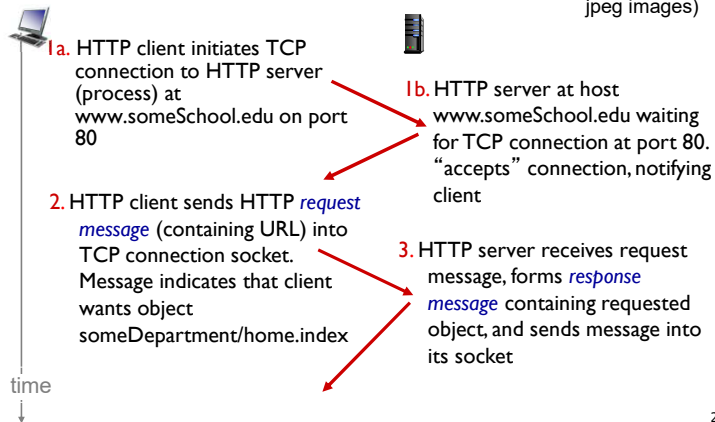- TCP connection then closed

2-22

Non-persistent HTTP: example

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
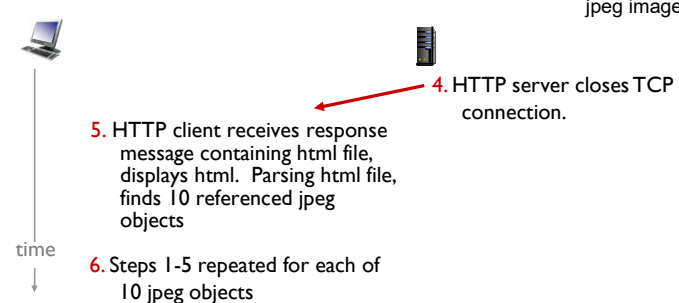
time

2-23

Non-persistent HTTP: example (cont.)

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

(contains text, references to 10 jpeg images)

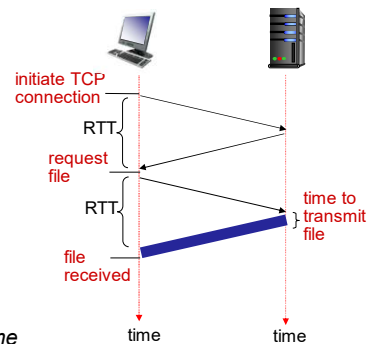4. HTTP server closes TCP connection.

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

2-24

6

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
- object/file transmission time

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

initiate TCP connection

RTT

request file

RTT

file received

time to transmit file

time   time

Persistent HTTP

non-persistent HTTP issues:
- requires 2 RTTs per object
- OS overhead for *each* TCP connection
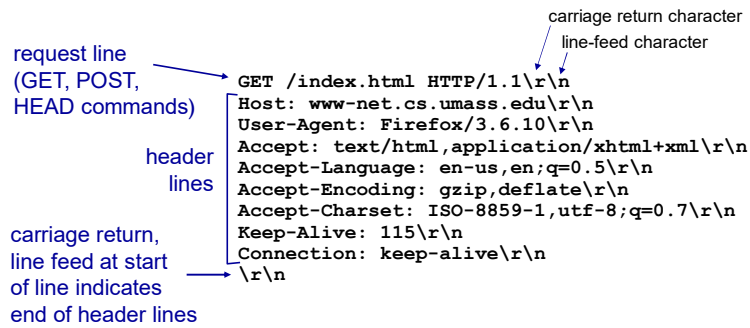- browsers often open parallel TCP connections to fetch referenced objects

persistent HTTP (HTTP 1.1):
- 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
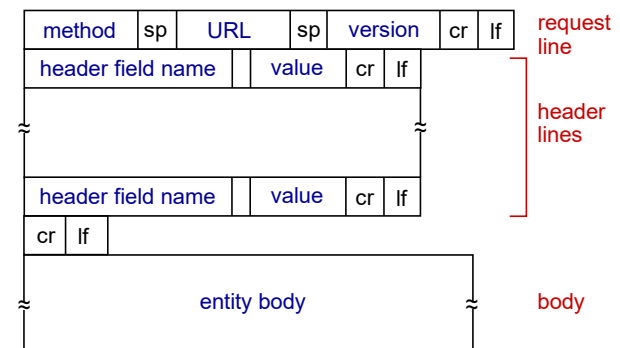- as little as one RTT for all the referenced objects (cutting response time in half)

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)

```
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
```

header lines

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

HTTP request message: general format

| method | sp | URL | sp | version | cr | lf | request line |
| header field name | | value | cr | lf | | |

| header field name | | value | cr | lf | | |

| cr | lf |

| entity body | body

header lines

Other HTTP request messages

POST method:
- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):
- include user data in URL field of HTTP GET request message (following a '?'):

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

HEAD method:
- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

PUT method:
- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

HTTP response message

status line
(protocol
status code
status phrase)

```
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 ...
```

header lines

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 message

301 Moved Permanently
- requested object moved, new location specified later in this message (Location: field)

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 out HTTP (client side) for yourself

I. 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 /kurose_ross/interactive/index.php 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!

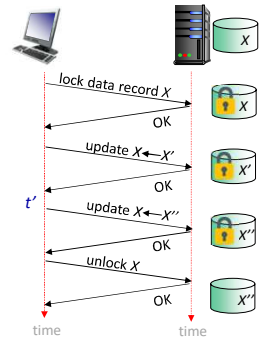(or use Wireshark to look at captured HTTP request/response)

Maintaining user-server state: cookies

Recall: HTTP GET/response interaction is *stateless*

❖ no notion of multi-step exchanges of HTTP messages to complete a Web "transaction"
- no need for client/server to track "state" of multi-step exchange
- all HTTP requests are independent of each other
- no need for client/server to "recover" from a partially-completed-but-never-completely-completed transaction

a stateful protocol: client makes two changes to X, or none at all

lock data record *X*

OK

update *X* ← *X'*

OK

t'

update *X* ← *X''*

OK

unlock *X*

OK

time   time

Q: what happens if network connection or client crashes at *t'* ?

Maintaining user-server state: cookies

Web sites and client browser use *cookies* to maintain some state between transactions

four components:

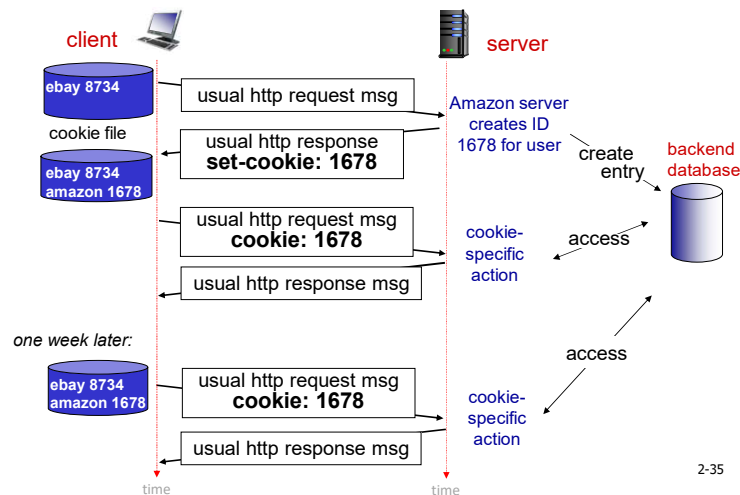1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

Example:

❖ Susan uses browser on laptop, visits specific e-commerce site for first time
❖ when initial HTTP requests arrives at site, site creates:
- unique ID (aka "cookie")
- entry in backend database for ID
❖ subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

2-34

Maintaining user-server state: cookies

client

ebay 8734
cookie file

ebay 8734
amazon 1678

usual http request msg

usual http response
set-cookie: 1678

usual http request msg
cookie: 1678

usual http response msg

one week later:

ebay 8734
amazon 1678

usual http request msg
cookie: 1678

usual http response msg

time   time

server

Amazon server creates ID 1678 for user

create entry

cookie-specific action

access

cookie-specific action

access

backend database

2-35

HTTP cookies: comments

What cookies can be used for:

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

Challenge: How to keep state?

❖ *At protocol endpoints:* maintain state at sender/receiver over multiple transactions
❖ *in messages:* cookies in HTTP messages carry state
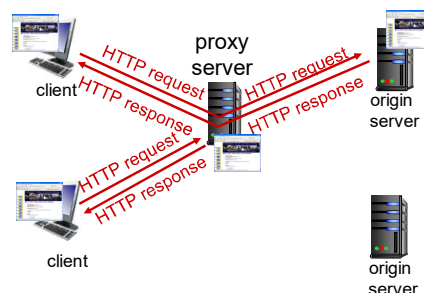
— aside —

cookies and privacy:

- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

2-36

Web caches

goal: satisfy client request without involving origin server

- user configures browser to a point to a (local) *Web cache*
- browser sends all HTTP requests to cache
  - *If* object in cache: cache returns object
  - *else* cache requests object from origin server, caches received object, then returns object to client

Web caches (aka proxy servers)

- Web 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
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches:
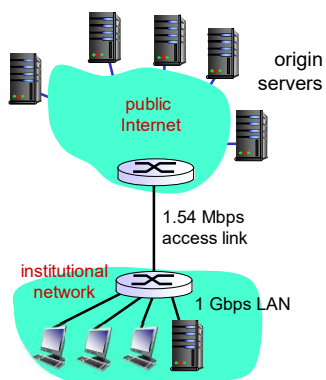 - enables "poor" content providers to more effectively deliver content

Caching example

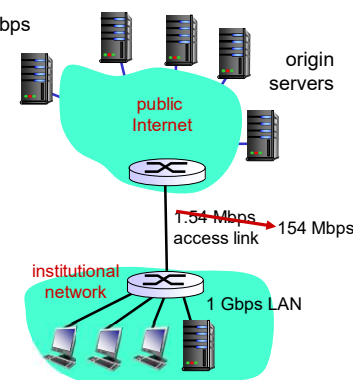*Scenario:*

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

Performance:

- access link utilization = .97
- LAN utilization: .0015
- end-end delay
  = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

problem: large queueing delays at high utilization!

1.54 Mbps access link

institutional network

1 Gbps LAN

Option 1: buy a faster access link

Scenario:

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

Performance:

- access link utilization = 0.97 → 0.0097
- LAN utilization: 0.0015
- end-end delay
  = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs → msecs

Cost: faster access link (expensive!)

1.54 Mbps access link → 154 Mbps

institutional network

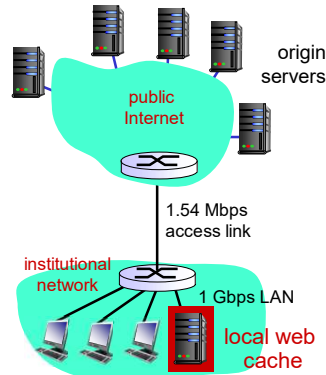1 Gbps LAN

Option 2: install a web cache

Scenario:

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

Cost: web cache (cheap!)

Performance:

- ❖ LAN utilization: .?
- ❖ access link utilization = ?
- ❖ average end-end delay = ?

How to compute link utilization, delay?

origin servers

public Internet

1.54 Mbps access link

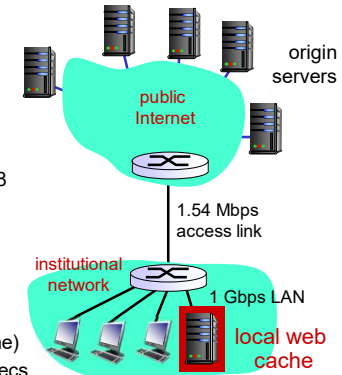institutional network

1 Gbps LAN

local web cache

Calculating access link utilization, end-end delay with cache

suppose cache hit rate is 0.4:

- ❖ 40% requests served by cache, with low (msec) delay
- ❖ 60% requests satisfied at origin
  - • rate to browsers over access link
    = 0.6 * 1.50 Mbps = 0.9 Mbps
  - • access link utilization = 0.9/1.54 = 0.58 means low (msec) queueing delay at access link
- ❖ average end-end delay:
  = 0.6 * (delay from origin servers)
    + 0.4 * (delay when satisfied at cache)
  = 0.6 * (2.01) + 0.4 (~msecs) = ~ 1.2 secs

lower average end-end delay than with 154 Mbps link (and cheaper too!)

origin servers

public Internet

1.54 Mbps access link

institutional network

1 Gbps LAN

local web cache

Browser caching: Conditional GET

- ❖ *Goal:* don't send object if browser has up-to-date cached version
  - ▪ no object transmission delay (or use of network resources)
- ❖ *client:* specify date of cached copy in HTTP request
  `If-modified-since: <date>`
- ❖ *server:* response contains no object if browser-cached copy is up-to-date:
  `HTTP/1.0 304 Not Modified`

client   server

HTTP request msg
If-modified-since: <date>

object not modified before <date>

HTTP response
HTTP/1.0 304 Not Modified

- - - - - - - - - - - - - - - - - - - -

HTTP request msg
If-modified-since: <date>

object modified after <date>

HTTP response
HTTP/1.0 200 OK
<data>

Chapter 2: outline

2.1. Principles of network applications

    2.1.1. Network application architectures

    2.1.2. Communicating between processes

    2.1.3. Transport services

2.2. The Web and HTTP
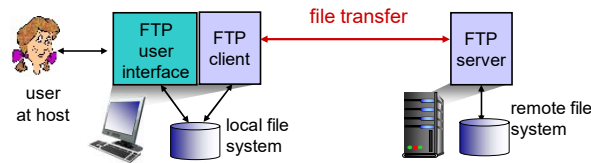
2.3. FTP

2.4. Electronic mail

2.5. DNS (Domain Name Systems)

2.6. Peer-to-peer applications

2.7. Video streaming and content distribution networks

2.8. Socket programming with UDP and TCP

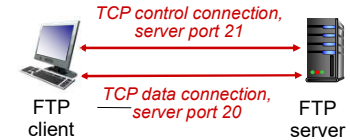## FTP: the file transfer protocol

file transfer

- ❖ 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

TCP control connection, server port 21

TCP data connection, server port 20

- ❖ 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**
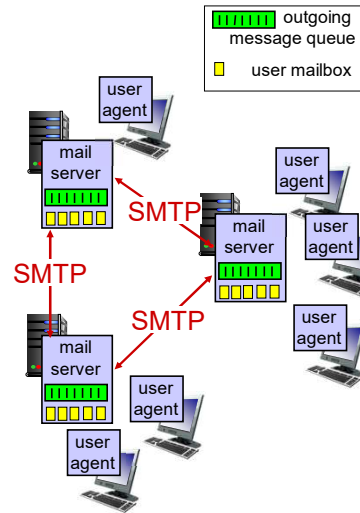- ❖ **452 Error writing file**

Chapter 2: outline

Electronic mail

Three major components:

* user agents
* mail servers
* simple mail transfer protocol: SMTP

User Agent

* a.k.a. "mail reader"
* composing, editing, reading mail messages
* e.g., Outlook, iPhone mail client
* outgoing, incoming messages stored on server
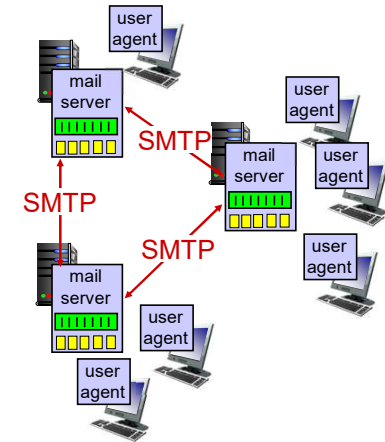
outgoing message queue

user mailbox

SMTP

SMTP

SMTP

Electronic mail: mail servers

mail servers:

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

SMTP protocol between mail servers to send email messages

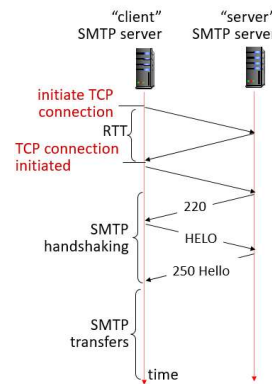* client: sending mail server
* "server": receiving mail server

SMTP

SMTP

SMTP

Electronic Mail: SMTP [RFC 5321]
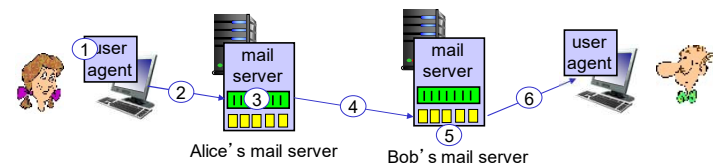
* uses TCP to reliably transfer email message from client to server, port 25
* direct transfer: sending server to receiving server
* three phases of transfer
  * SMTP handshaking (greeting)
  * SMTP transfer of messages
  * SMTP closure
* command/response interaction (like HTTP)
  * commands: ASCII text
  * response: status code and phrase

"client" SMTP server

"server" SMTP server

initiate TCP connection

RTT

TCP connection initiated

220

SMTP handshaking

HELO

250 Hello

SMTP transfers

time

Scenario: Alice sends e-mail to Bob

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

Alice's mail server

Bob's mail server

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 interaction for yourself:

❖ **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)

SMTP: observations

❖ 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:

❖ HTTP: pull
❖ SMTP: push

❖ both have ASCII command/response interaction, status codes

▪ HTTP: each object encapsulated in its own response message
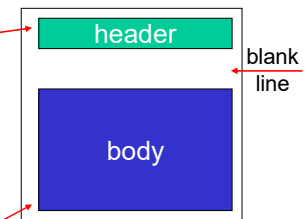▪ SMTP: multiple objects sent in multipart message

Mail message format
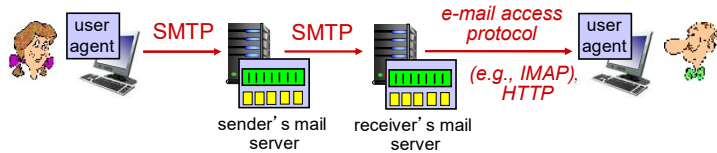
SMTP: protocol for exchanging e-mail messages
RFC 2822: defines *syntax* for e-mail message:
❖ header lines, e.g.,
  ▪ To:
  ▪ From:
  ▪ Subject:
  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
❖ Body: the "message"
  ▪ ASCII characters only

header

blank line

body

Retrieving email: mail access protocols

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

Chapter 2: outline

DNS: domain name system

people: many identifiers:
- ▪ SSN (Social Security number), name, passport #

Internet hosts, routers:
- ▪ IP address (32 bit) - used for addressing datagrams
- ▪ "name", e.g., 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

Q: why not centralize DNS?
- ❖ single point of failure
- ❖ traffic volume
- ❖ distant centralized database
- ❖ maintenance

A: doesn't scale!
- ▪ Comcast DNS servers alone: 600B DNS queries/day
- ▪ Akamai DNS servers alone: 2.2T DNS queries/day

Thinking about the DNS

humongous distributed database:
- ~ billion records, each simple

handles many *trillions* of queries/day:
- *many* more reads than writes
- *performance matters:* almost every Internet transaction interacts with DNS - msecs count!

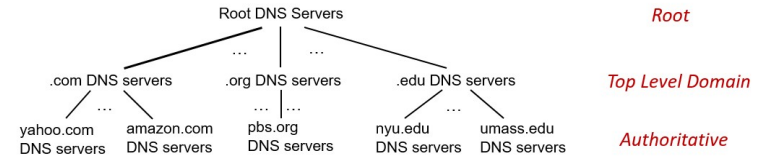organizationally, physically decentralized:
- millions of different organizations responsible for their records

"bulletproof": reliability, security

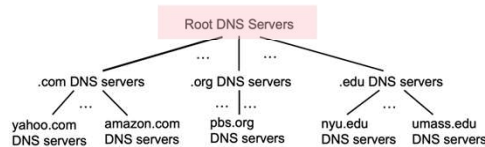NOT SO EASY

2-61

DNS: a distributed, hierarchical database

Root DNS Servers — *Root*

.com DNS servers   .org DNS servers   .edu DNS servers — *Top Level Domain*

yahoo.com DNS servers   amazon.com DNS servers   pbs.org DNS servers   nyu.edu DNS servers   umass.edu DNS servers — *Authoritative*

Client wants IP address for www.amazon.com; 1st approximation:
- client queries root server to find .com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

2-62

DNS: root name servers

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

Root DNS Servers

.com DNS servers   .org DNS servers   .edu DNS servers

yahoo.com DNS servers   amazon.com DNS servers   pbs.org DNS servers   nyu.edu DNS servers   umass.edu DNS servers

2-63

DNS: root name servers

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

13 logical root name "servers" worldwide each "server" replicated many times (~200 servers in US)

- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication, message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

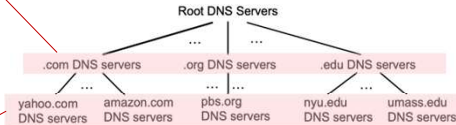Key:
0 Servers
1–10 Servers
11–20 Servers
21+ Servers

2-64

16

Top-Level Domain, and authoritative servers

top-level domain (TLD) servers:

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

Root DNS Servers

```
                    ...        ...
  .com DNS servers      .org DNS servers      .edu DNS servers
       ...                   ...                   ...
 yahoo.com  amazon.com    pbs.org          nyu.edu   umass.edu
 DNS servers DNS servers  DNS servers      DNS servers DNS servers
```

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

2-65

Local DNS name server

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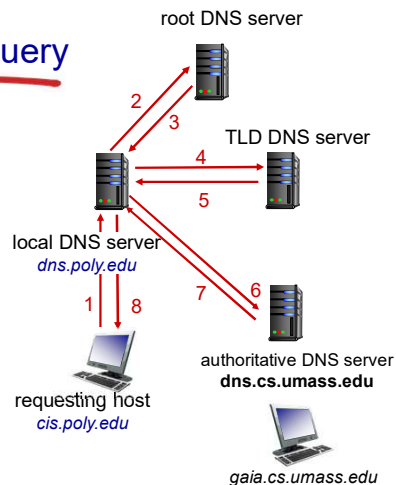- Local DNS server doesn't strictly belong to hierarchy

2-66

DNS name resolution: iterated query

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"

root DNS server

TLD DNS server

local DNS server
dns.poly.edu

authoritative DNS server
dns.cs.umass.edu

requesting host
cis.poly.edu

gaia.cs.umass.edu

2-67

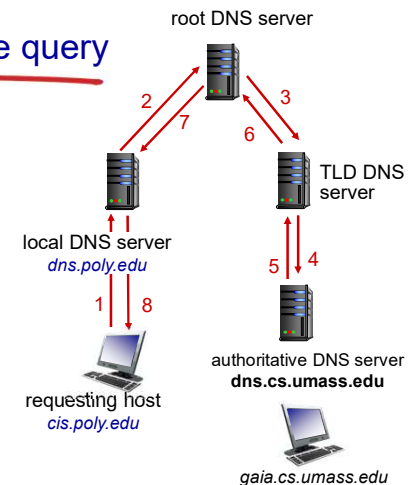## DNS name resolution: recursive query

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?

root DNS server

TLD DNS server

local DNS server
dns.poly.edu

authoritative DNS server
dns.cs.umass.edu

requesting host
cis.poly.edu

gaia.cs.umass.edu

2-68

Caching DNS Information

- once (any) name server learns mapping, it *caches* mapping, and *immediately* returns a cached mapping in response to a query
  - caching improves response time
  - 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*
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
  - best effort name-to-address translation!

DNS records

DNS: distributed database storing resource records (RR)

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

type=A
- **name** is hostname
- **value** is IP address

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

type=CNAME
- **name** is alias name for some "canonical" (the real) name
- **www.ibm.com** is really **servereast.backup2.ibm.com**
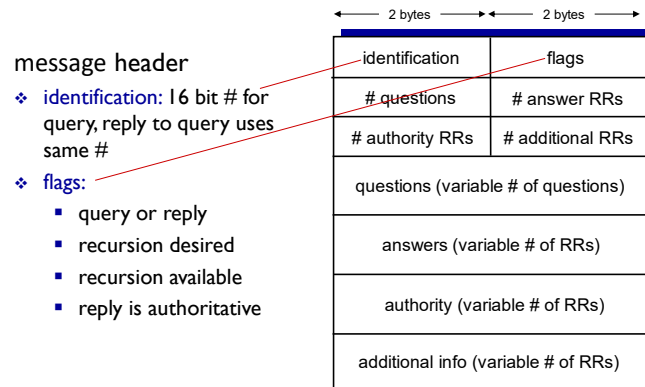- **value** is canonical name

type=MX
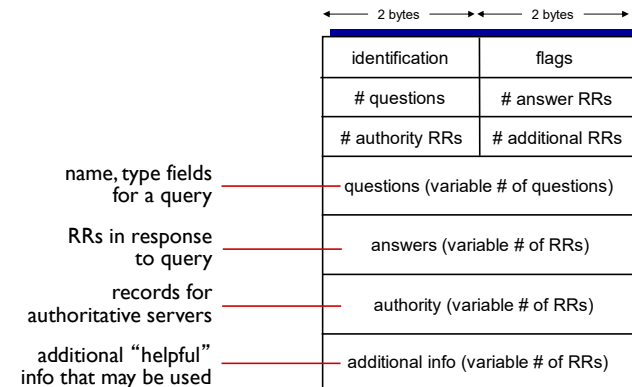- **value** is name of mailserver associated with **name**

DNS protocol messages

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

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

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)	

DNS protocol messages

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

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)	

name, type fields for a query

RRs in response to query

records for authoritative servers

additional "helpful" info that may be used

Getting your info into the 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 NS, A RRs into .com TLD server:
    ```
    (networkutopia.com, dns1.networkutopia.com, NS)
    (dns1.networkutopia.com, 212.212.212.1, A)
    ```
- ❖ Create authoritative server locally with IP address `212.212.212.1`
  - ▪ type A record for `www.networkuptopia.com`
  - ▪ type MX record for `networkutopia.com`

DNS security

DDoS attacks

- ❖ Bombard root servers with traffic
  - ▪ Not successful to date
  - ▪ Traffic Filtering
  - ▪ Local DNS servers cache IPs of TLD servers, allowing root server bypass
- ❖ Bombard TLD servers
  - ▪ Potentially more dangerous

Spoofing attacks

- ❖ intercept DNS queries, returning bogus replies
  - ▪ DNS cache poisoning
  - ▪ RFC 4033: DNSSEC authentication services

Chapter 2: outline

2.1. Principles of network applications
  2.1.1. Network application architectures
  2.1.2. Communicating between processes
  2.1.3. Transport services
2.2. The Web and HTTP
2.3. FTP
2.4. Electronic mail

2.5. DNS (Domain Name Systems)
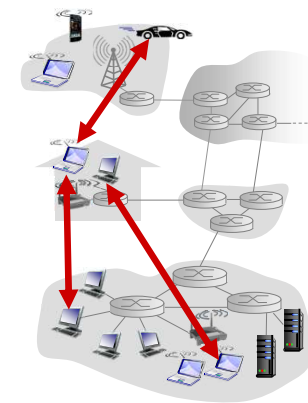2.6. Peer-to-peer applications
2.7. Video streaming and content distribution networks
2.8. Socket programming with UDP and TCP

Peer-to-peer (P2P) architecture

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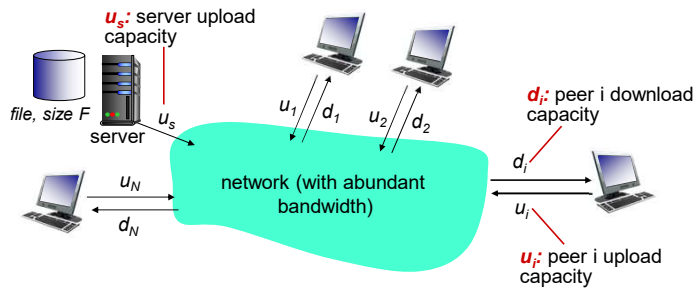- ❖ *examples:* file distribution (BitTorrent); Streaming (KanKan); VoIP (Skype)

File distribution: client-server vs P2P

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

- peer upload/download capacity is limited resource

u_s: server upload capacity

d_i: peer i download capacity

u_i: peer i upload capacity

file, size F

server

u_s

u_1 / d_1 u_2 / d_2

u_N

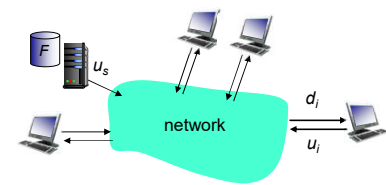d_N

network (with abundant bandwidth)

d_i

u_i

2-77

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}

F u_s

network

d_i

u_i

> time to distribute F to N clients using client-server approach
> $$D_{c-s} \geq max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

2-78

File distribution time: P2P
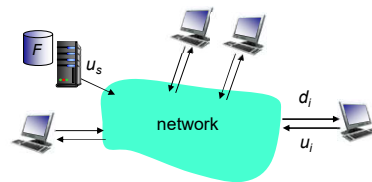
- *server transmission:* must upload at least one copy
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* as aggregate must download NF bits
  - max upload rate (limting max download rate) is $u_s + \Sigma u_i$

F u_s

network

d_i

u_i

> time to distribute F to N clients using P2P approach
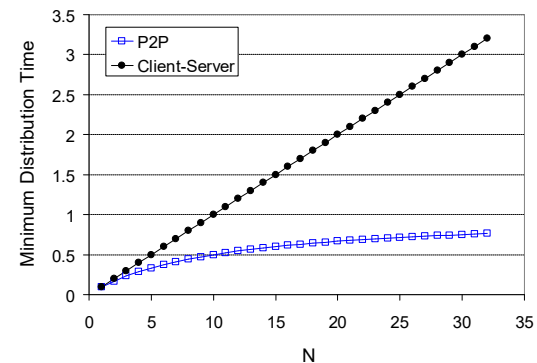> $$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

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

2-79

Client-server vs. P2P: example

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

2-80
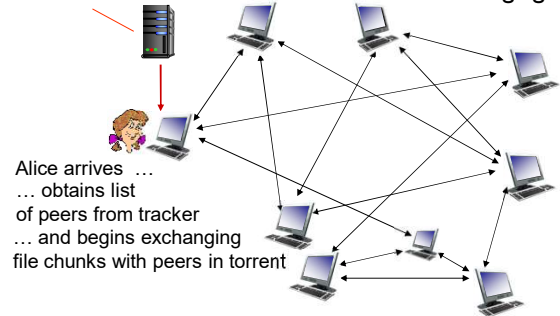
20

P2P file distribution: BitTorrent

❖ file divided into 256Kb chunks
❖ peers in torrent send/receive file chunks

tracker: tracks peers participating in torrent

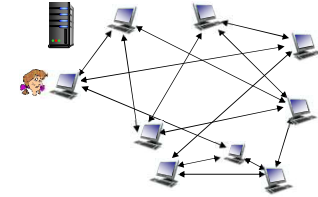*torrent:* group of peers exchanging chunks of a file

Alice arrives …
… obtains list of peers from tracker
… and begins exchanging file chunks with peers in torrent

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

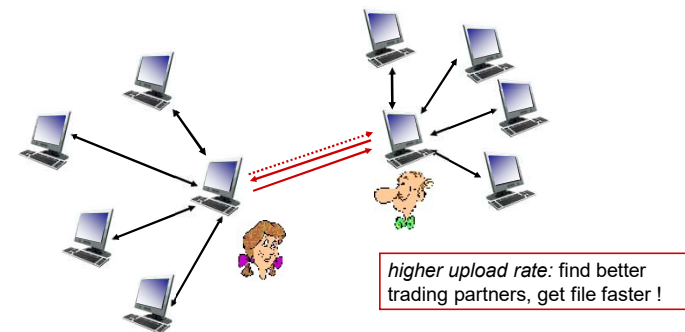*sending chunks: tit-for-tat*

❖ 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 !

Chapter 2: outline
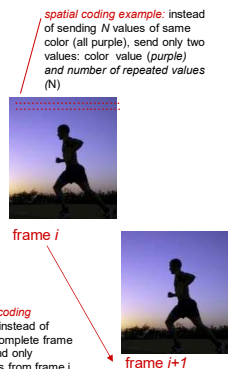
2-85

Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- *challenge:* scale - how to reach ~1B users?

- *challenge:* heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution:* distributed, application-level infrastructure
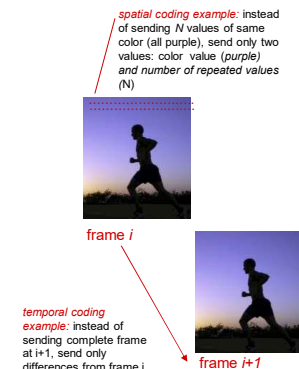
2-82

Multimedia: video

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

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

frame *i*

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

frame *i+1*

2-87

Multimedia: video

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

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

frame *i*

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

frame *i+1*

2-88

Streaming stored video

simple scenario:

video server
(stored video)

Internet

client

Main challenges:

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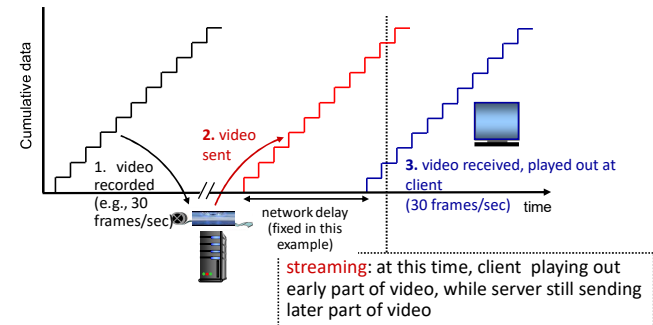
❖ packet loss, delay due to congestion will delay playout, or result in poor video quality

Streaming stored video

Cumulative data

1. video recorded (e.g., 30 frames/sec)

2. video sent

network delay (fixed in this example)

3. video received, played out at client (30 frames/sec) time

streaming: at this time, client playing out early part of video, while server still sending later part of video
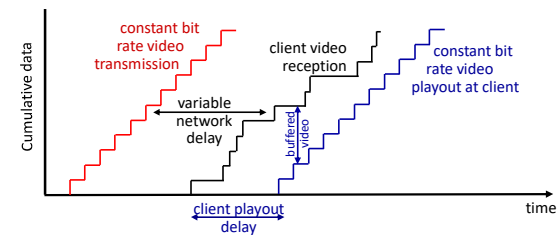
Streaming stored video: challenges

- continuous playout constraint: during client video playout, playout timing must match original timing
  - … but network delays are variable (jitter), so will need client-side buffer to match continuous playout constraint

- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted

Streaming stored video: playout buffering

Cumulative data

constant bit rate video transmission

client video reception

variable network delay

buffered video

constant bit rate video playout at client

client playout delay

time

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

Streaming multimedia: DASH

*D*ynamic, *A*daptive *S*treaming over *H*TTP

server:
- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file:* provides URLs for different chunks

client:
- periodically estimates 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), and from different servers

Streaming multimedia: DASH

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

  Streaming video = encoding + DASH + playout buffering

Content distribution networks (CDNs)

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

- *option 1:* single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long (and possibly congested) path to distant clients

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

Content distribution networks (CDNs)

challenge: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
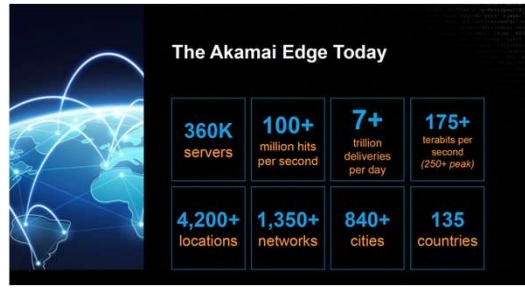
- *option 2:* store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - Akamai: 240,000 servers deployed in > 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in POPs near access nets
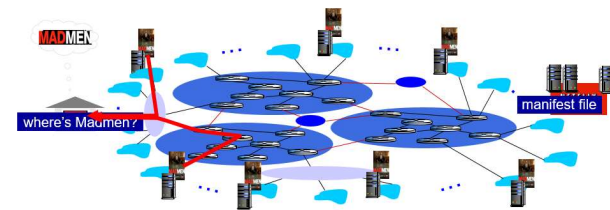   - used by Limelight

Akamai today:

The Akamai Edge Today

360K servers	100+ million hits per second	7+ trillion deliveries per day	175+ terabits per second (250+ peak)
4,200+ locations	1,350+ networks	840+ cities	135 countries

Source: https://networkingchannel.eu/living-on-the-edge-for-a-quarter-century-an-akamai-retrospective-downloads/

How does Netflix work?

- Netflix: stores copies of content (e.g., MADMEN) at its (worldwide) OpenConnect CDN nodes
- subscriber requests content, service provider returns manifest
  - using manifest, client retrieves content at highest supportable rate
  - may choose different rate or copy if network path congested

---

Content distribution networks (CDNs)

OTT: "over the top"

Internet host-host communication as a service

OTT challenges: coping with a congested Internet from the "edge"
- what content to place in which CDN node?
- from which CDN node to retrieve content? At which rate?

Chapter 2: outline

2.1. Principles of network applications
   2.1.1. Network application architectures
   2.1.2. Communicating between processes
   2.1.3. Transport services
2.2. The Web and HTTP
2.3. FTP
2.4. Electronic mail

2.5. DNS (Domain Name Systems)
2.6. Peer-to-peer applications
2.7. Video streaming and content distribution networks
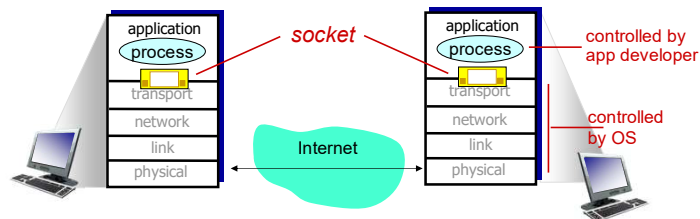2.8. Socket programming with UDP and TCP

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
controlled by app developer
controlled by OS

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
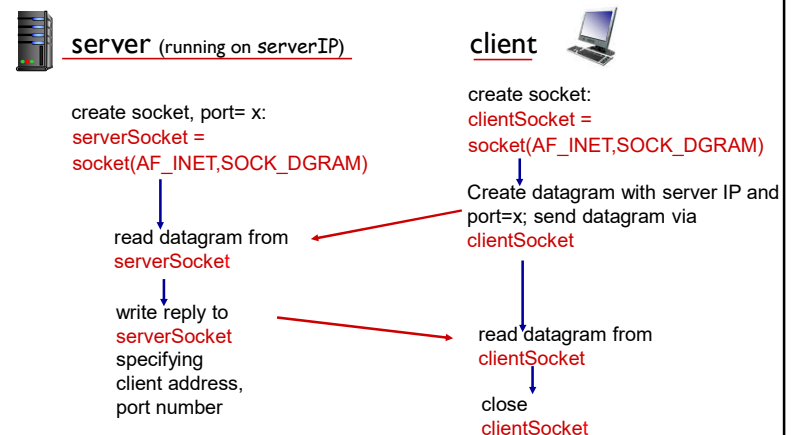- ❖ rcvr extracts sender IP address and port# from received packet

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 processes

---

Client/server socket interaction: UDP

server (running on `serverIP`)   **client**

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

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and port=x; send datagram via clientSocket

read datagram from serverSocket

write reply to serverSocket specifying client address, port number

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

from socket import *

serverPort = 12000

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

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

print "*The server is ready to receive*"

loop forever → while 1:

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

modifiedMessage = message.upper()

send upper case string back to this client → 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
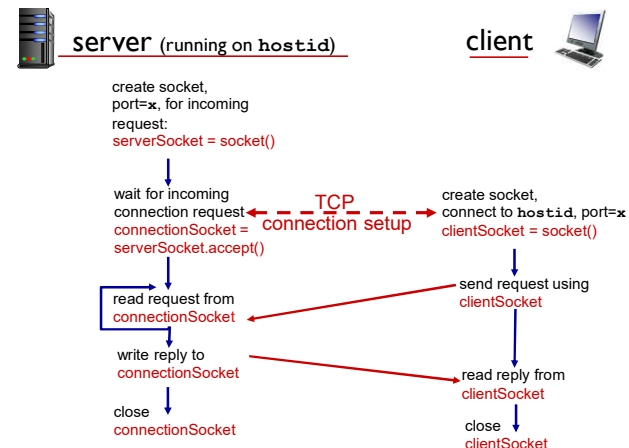- ❖ *when client creates socket:* client TCP establishes connection to server TCP

- ❖ 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

application viewpoint:

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

---

Client/server socket interaction: TCP

Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```python
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()
```

2-109

Example app: TCP server

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

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

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

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

```python
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()
```

2-110

Chapter 2: summary

our study of network application layer is now complete!

- ❖ application architectures
  - client-server
  - P2P
- ❖ application service requirements:
  - reliability, bandwidth, delay
- ❖ Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- ❖ specific protocols:
  - HTTP
  - FTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- ❖ video streaming, CDNs
- ❖ socket programming: TCP, UDP sockets

2-111

Chapter 2: summary

most importantly: learned about protocols!

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

important themes:

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

2-112

References

> Jim Kurose, Keith Ross, "*Computer Networking: A Top-Down Approach*" 8th edition, Pearson, 2020.

1-113