

CMPE 150/L : Introduction to Computer Networks

Chen Qian

Computer Engineering
UCSC Baskin Engineering

Lecture 4

Reschedule office hour

- ❑ As stated in an earlier email, we will have an office hour 2-3pm tomorrow.
- ❑ If that doesn't work for you, email me.
- ❑ Only this week!

Chapter 1: roadmap

1.1 what *is* the Internet?

1.2 network edge

- end systems, access networks, links

1.3 network core

- packet switching, circuit switching, network structure

1.4 delay, loss, throughput in networks

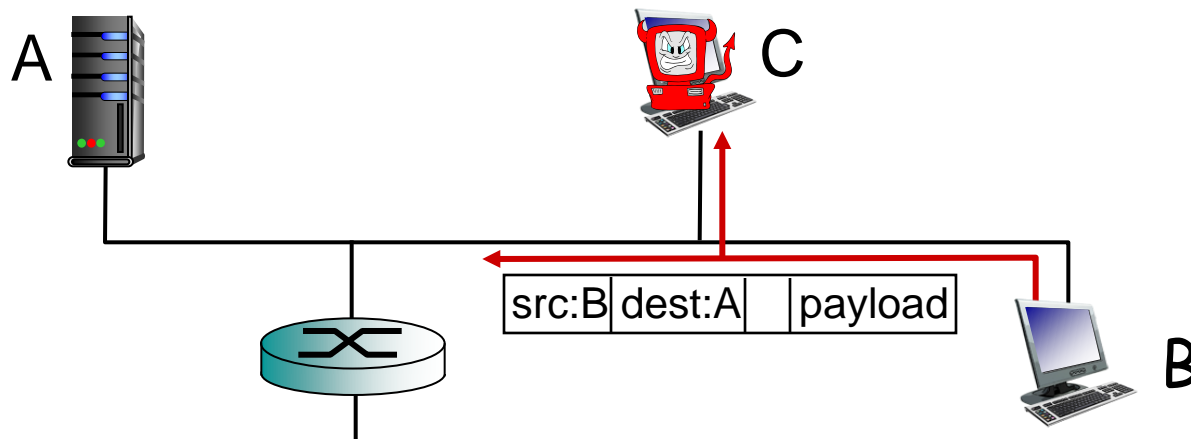
1.5 protocol layers, service models

1.6 networks under attack: security

Bad guys can sniff packets

packet “sniffing”:

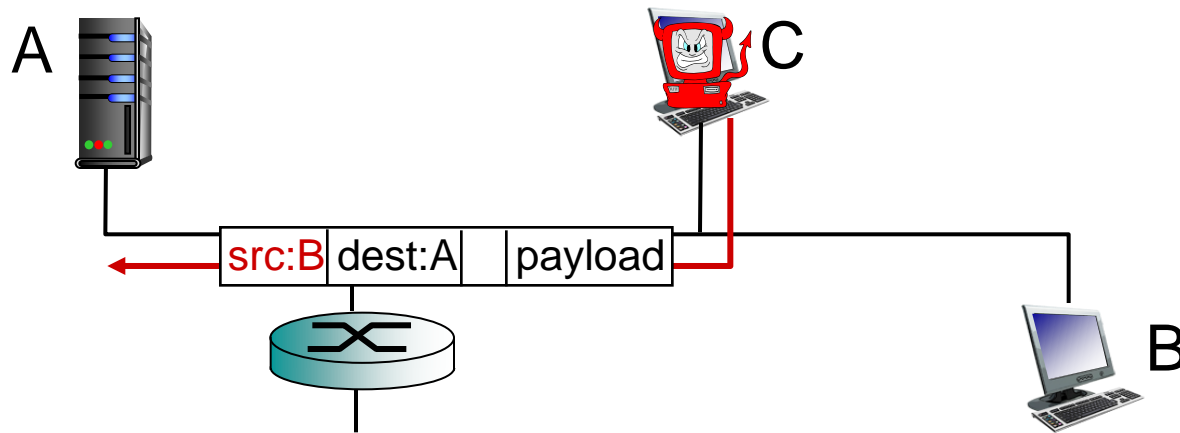
- broadcast media (shared ethernet, wireless)
- promiscuous network interface reads/records all packets (e.g., including passwords!) passing by



- ❖ wireshark software used for labs is a (free) packet-sniffer

Bad guys can use fake addresses

IP spoofing: send packet with false source address



... lots more on security (Chapter 8 and CMPE 253)

Chapter 2 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

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 / 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, Hulu, Netflix)
- ❖ voice over IP (e.g., Skype)
- ❖ real-time video conferencing
- ❖ social networking
- ❖ search
- ❖ ...
- ❖ ...

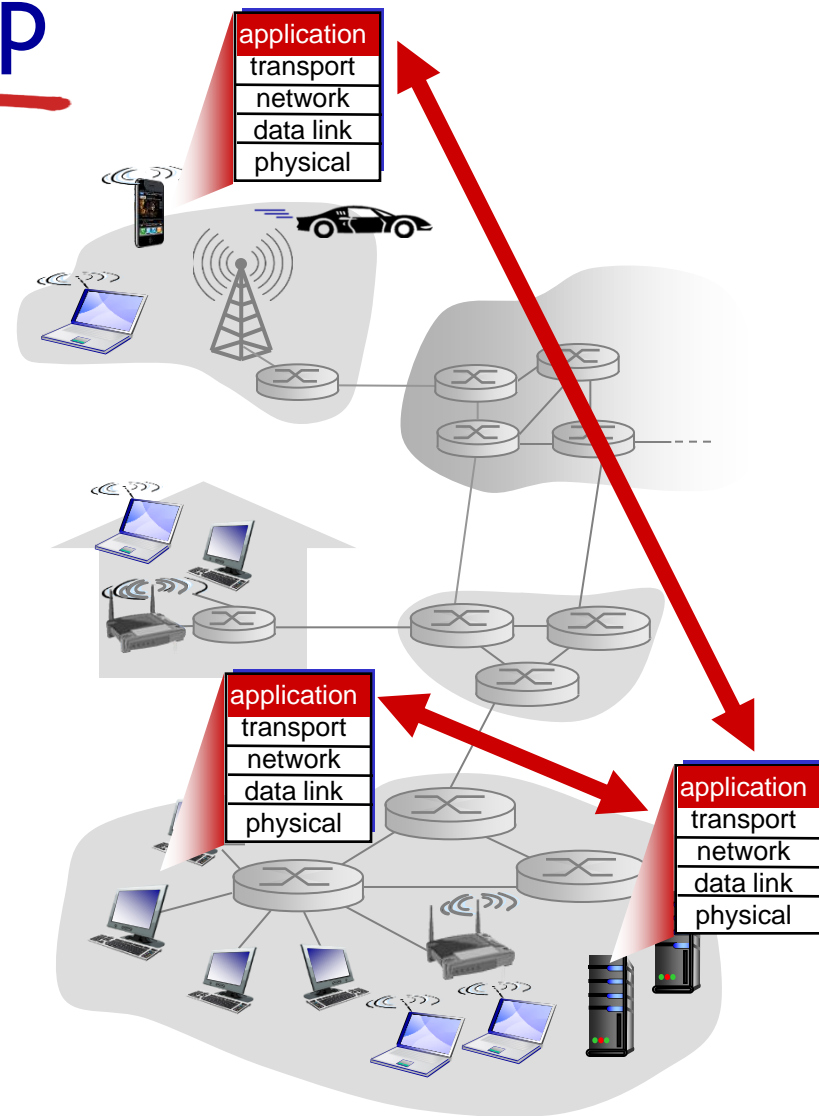
Creating a network app

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

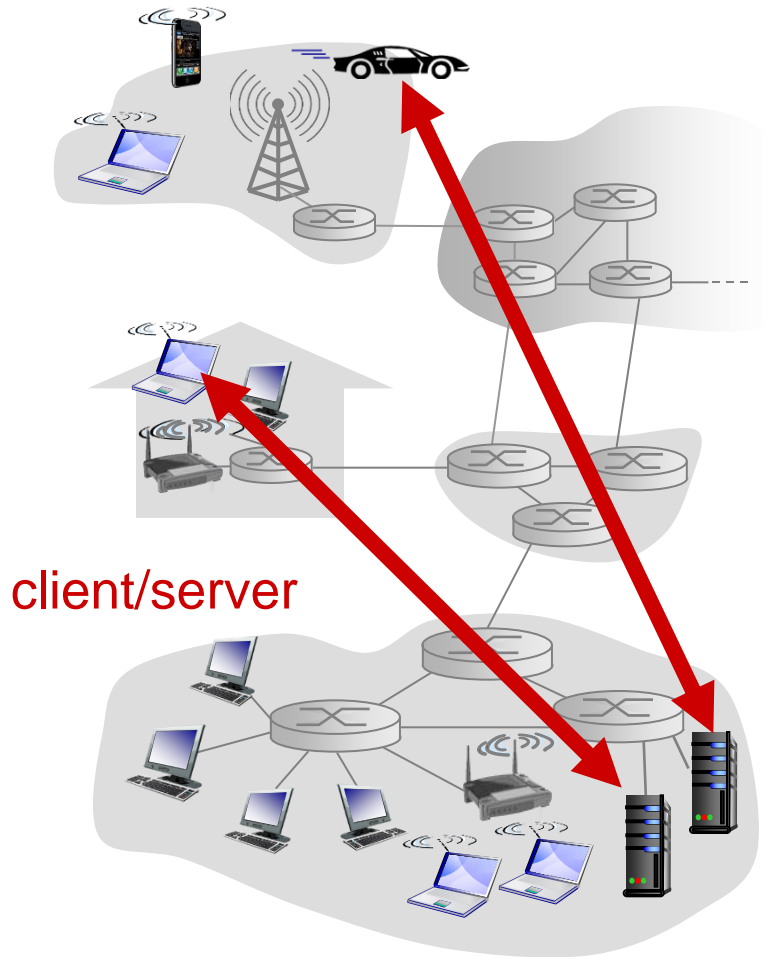


Application architectures

possible structure of applications:

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

Client-server architecture



server:

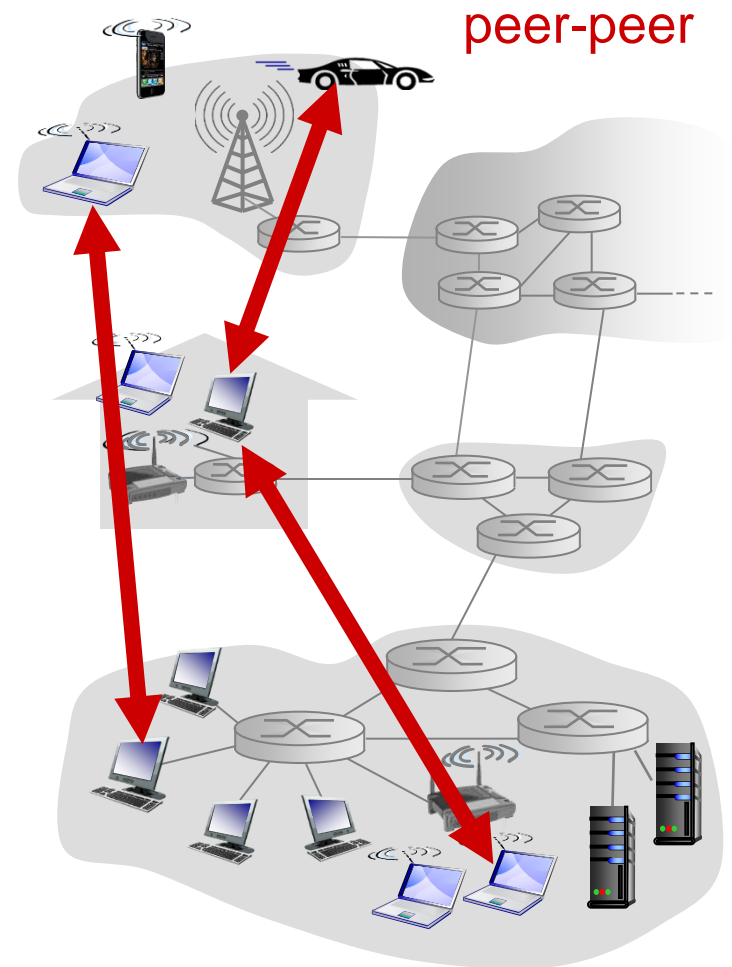
- ❖ always-on host
- ❖ permanent IP address
- ❖ data centers for scaling

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



Client/server versus P2P

- ❖ **Throughput and Scalability:**
- ❖ P2P wins!
- ❖ Because a server can only serve limited number of clients
- ❖ P2P allows clients exchange data among them.
- ❖ **That's why P2P became popular in early 2000**
- ❖ **Management**
- ❖ C/S wins!
- ❖ Because users in P2P are HIGHLY unreliable.
- ❖ In the recent years, throughput are not a big problem, management became the main issue.
- ❖ **That's why we now switch back to C/S**

Hybrid of client-server and P2P

Skype

- voice-over-IP P2P application
- centralized server: finding address of remote party:
- client-client connection: direct (not through server)

Instant messaging

- chatting between two users is (can be) P2P
- centralized service: client presence detection/location
 - user registers its IP address with central server when it comes online
 - user contacts central server to find IP addresses of buddies

- ❖ Interview with Bram Cohen, inventor of BitTorrent
- ❖ <https://www.youtube.com/watch?v=u0xngxfbKAE>
- ❖ 2:25 – 6:25

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

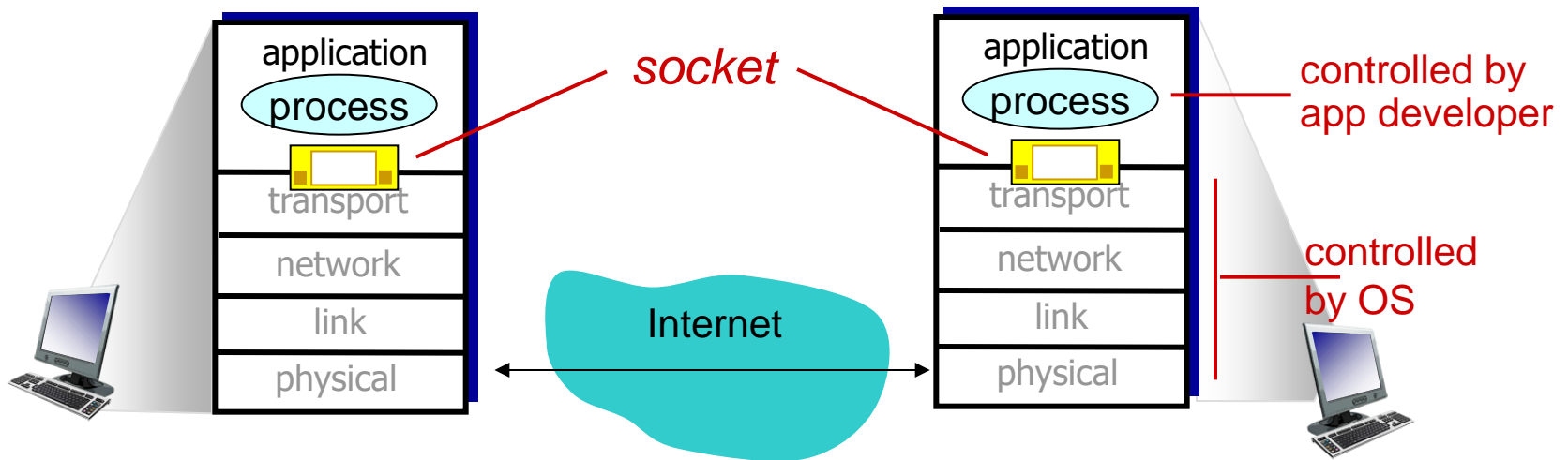
client process: process that initiates communication

server process: process that waits to be contacted

- ❖ aside: applications with P2P architectures have 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 gaia.cs.umass.edu web server:
 - **IP address**: 128.119.245.12
 - **port number**: 80

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	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100' s msec
text messaging	no loss	elastic	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, or connection 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

Chapter 2: 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.,

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

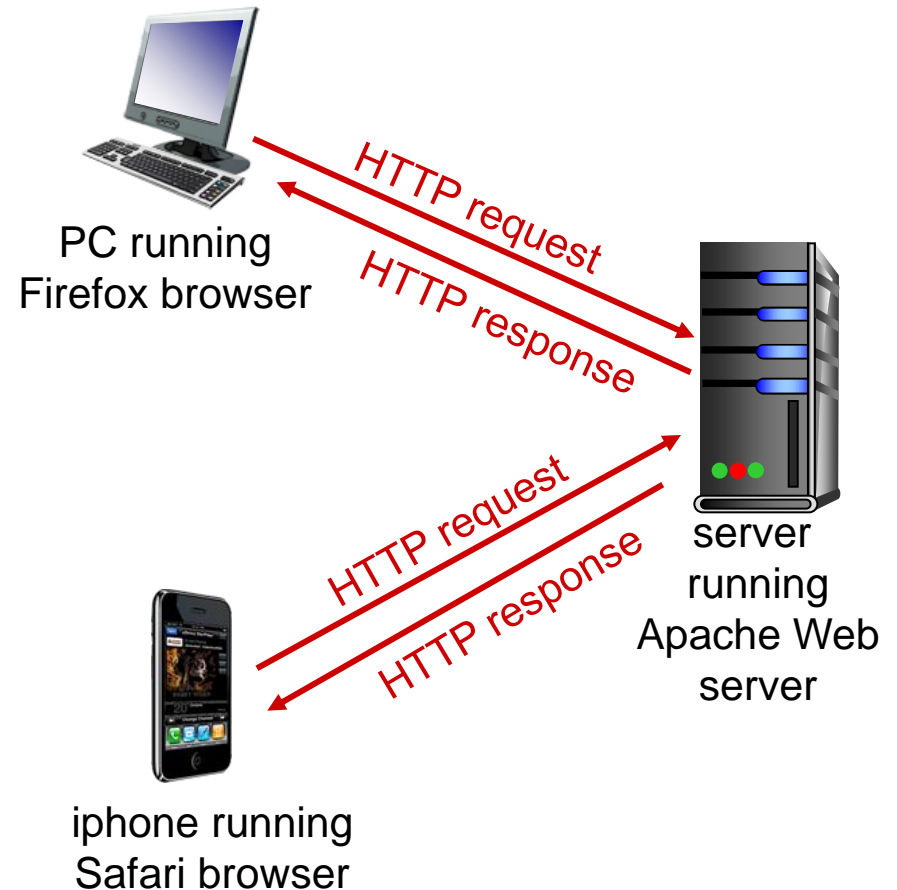
host name

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



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 connections

non-persistent HTTP

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

persistent HTTP

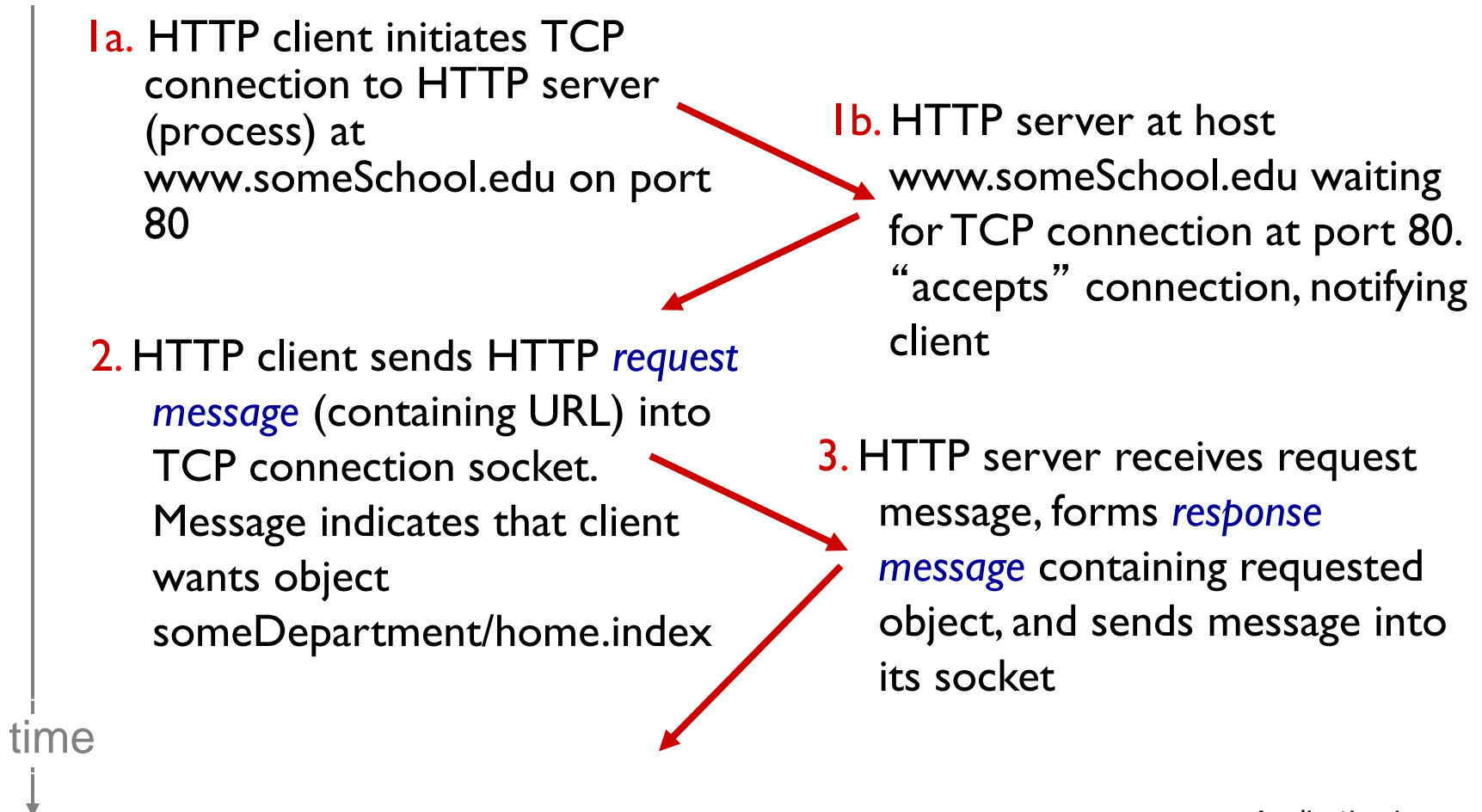
- ❖ multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP

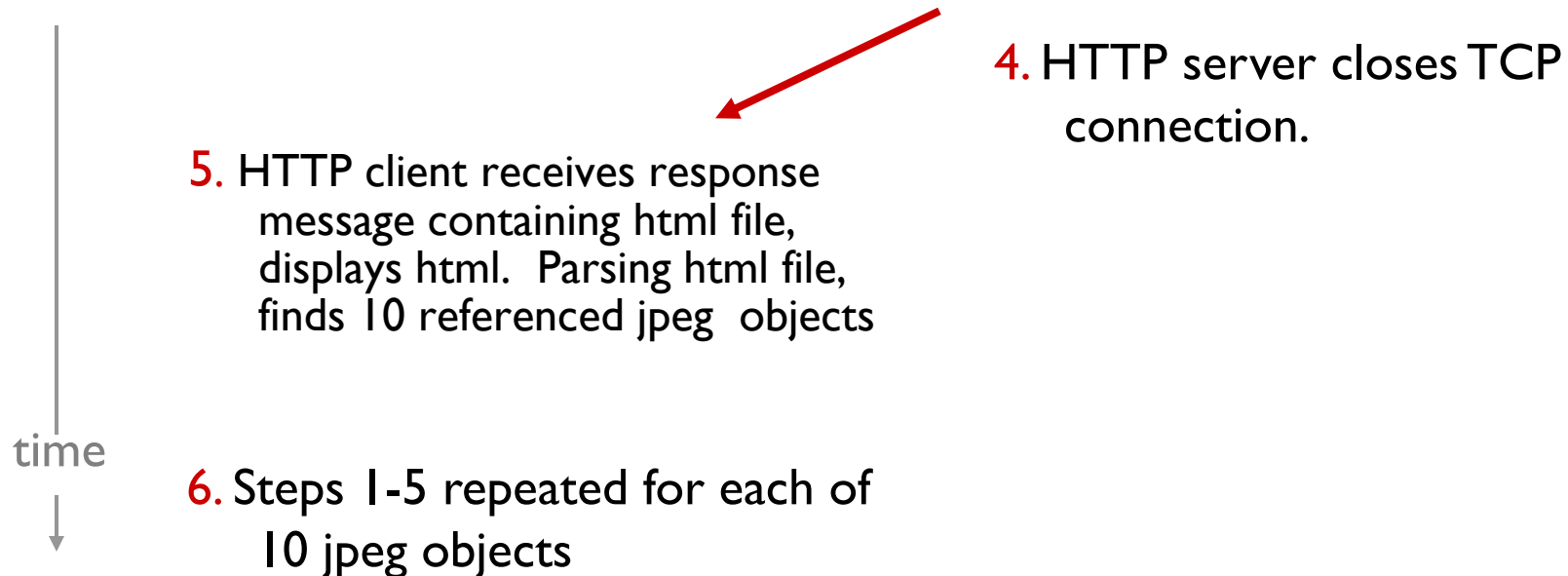
suppose user enters URL:

`www.someSchool.edu/someDepartment/home.index`

(contains text,
references to 10
jpeg images)



Non-persistent HTTP (cont.)



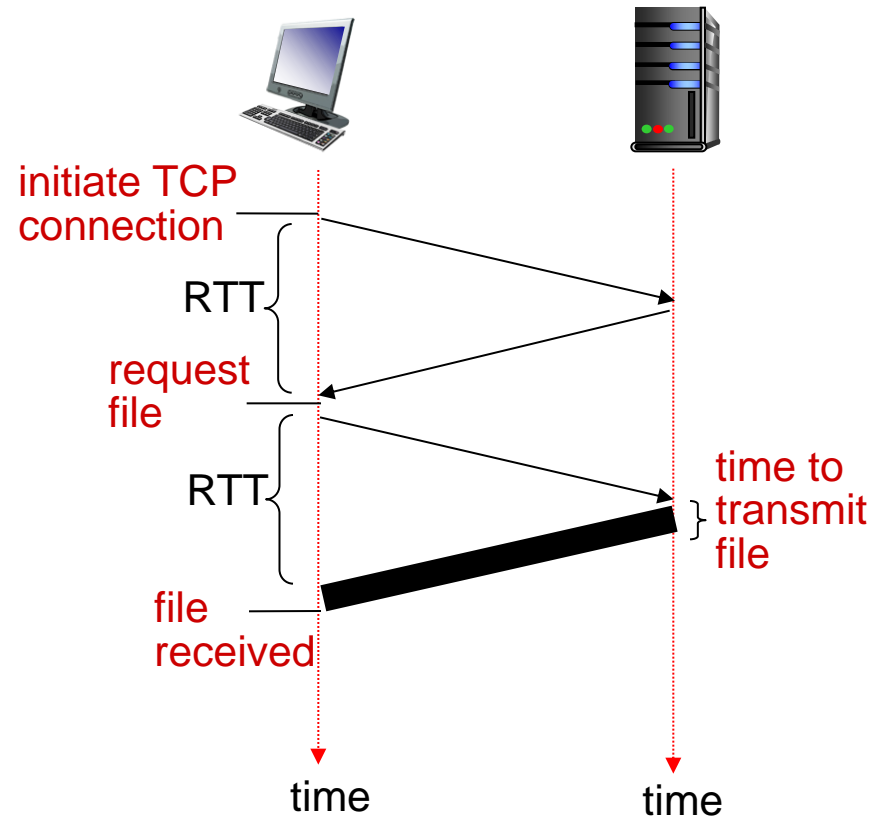
Non-persistent HTTP: response time

Round Trip 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 =
 $2\text{RTT} + \text{file transmission time}$



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:

- ❖ 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)

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
```

carriage return character
line-feed character

HTTP response message

status line
(protocol
status code
status phrase)

header
lines

data, e.g.,
requested
HTML file

```
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 ...
```

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

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.)

client



server



cookie file



ebay 8734
amazon 1678

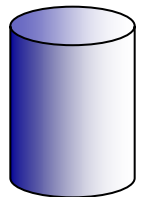
usual http request msg

Amazon server
creates ID
1678 for user

usual http response
set-cookie: 1678

create
entry

backend
database



usual http request msg
cookie: 1678

cookie-
specific
action

access

usual http response msg

access

cookie-
specific
action

one week later:



ebay 8734
amazon 1678

usual http request msg
cookie: 1678

usual http response msg

Cookies (continued)

what cookies can be used for:

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

cookies and privacy: aside

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

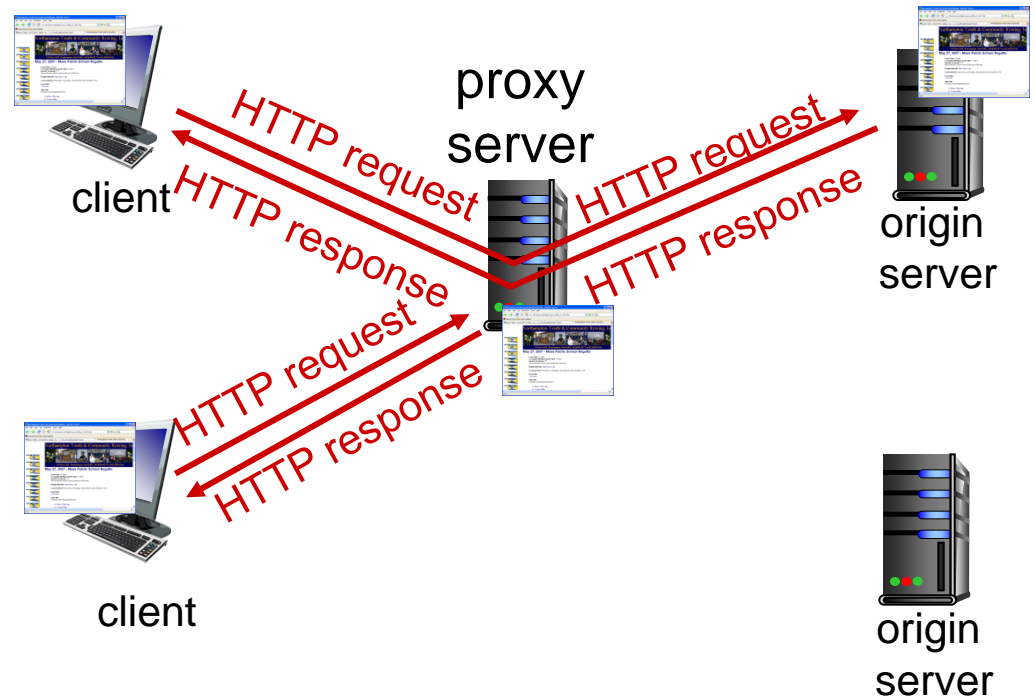
how to keep “state”:

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

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

When is cache not good?

- ❖ Every client of the ISP requests different content.
 - Waste time on visiting cache server

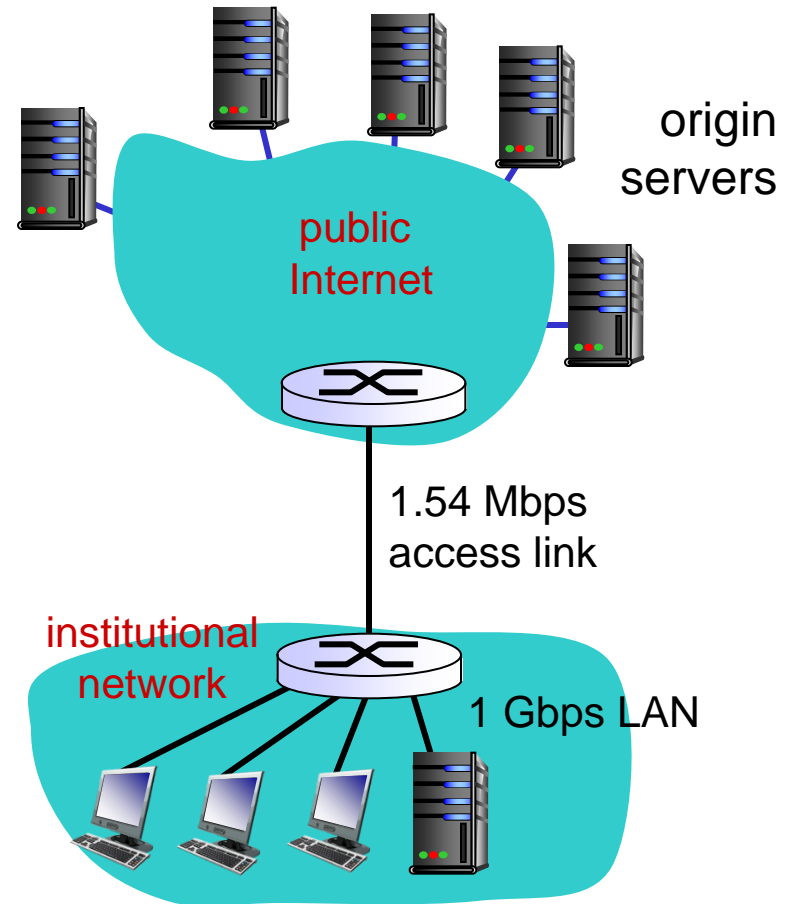
Caching example:

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:

- ❖ LAN utilization: 15%
- ❖ access link utilization = **99%** *problem!*
- ❖ total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs



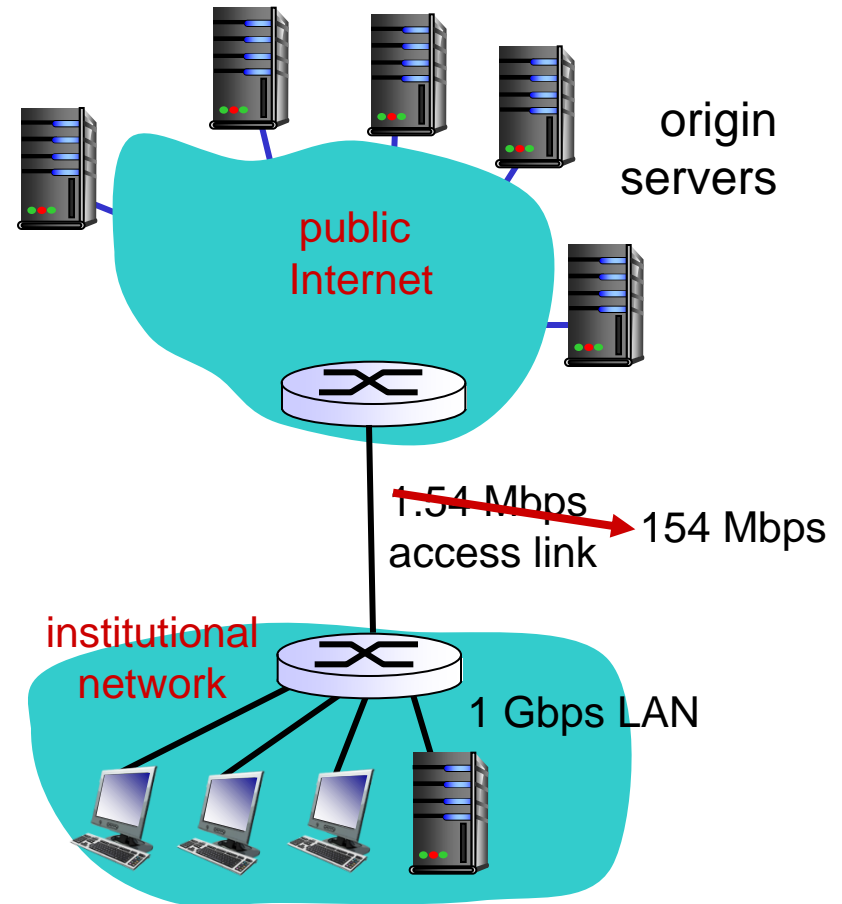
Caching example: fatter access link

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~~ → msec



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

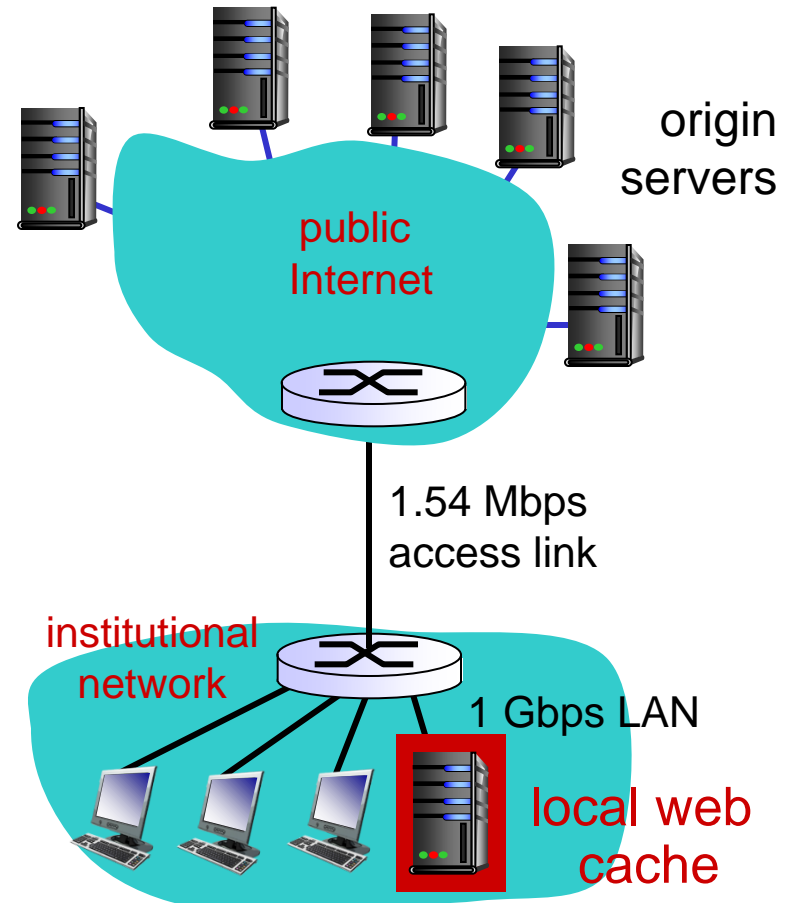
- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

consequences:

- ❖ LAN utilization: 15%
- ❖ access link utilization = ?
- ❖ total delay = ?

How to compute link utilization, delay?

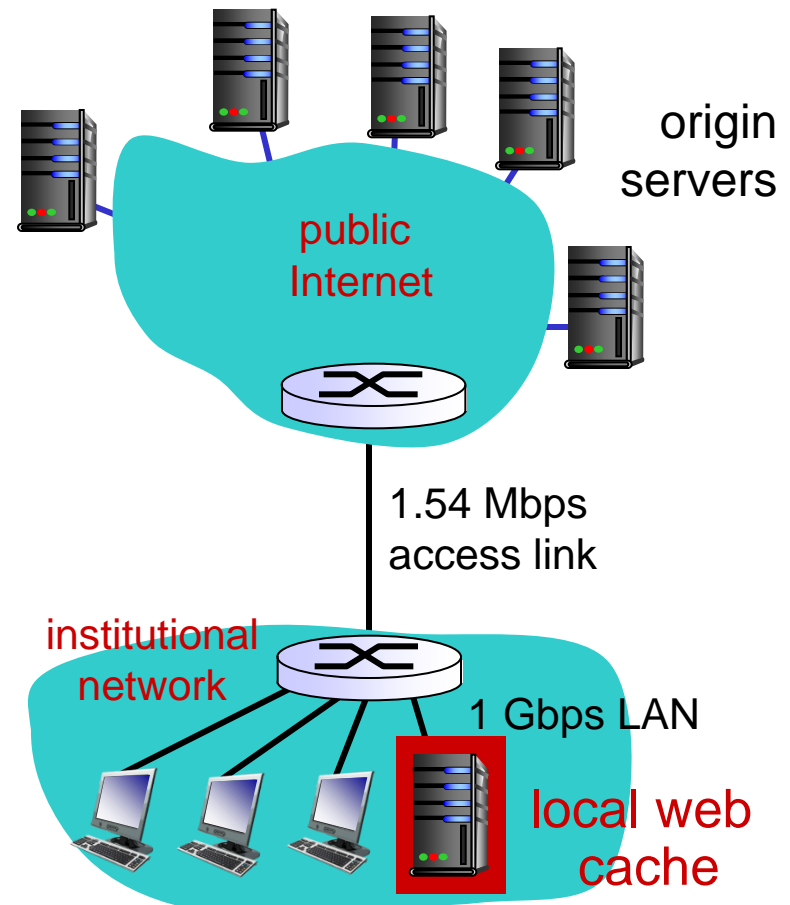
Cost: web cache (cheap!)



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
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
 - utilization $= 0.9 / 1.54 = .58$
- ❖ total delay
 - $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
 - $= 0.6 (2.01) + 0.4 (\sim \text{msecs})$
 - $= \sim 1.2 \text{ secs}$
 - less than with 154 Mbps link (and cheaper too!)



- ❖ Interview with early developers of Web Browsers
- ❖ https://www.youtube.com/watch?v=_L3Y2_YiT-A

Marc Andreessen

Marc Andreessen is the co-creator of Mosaic, the Web browser that popularized the World Wide Web in 1993. Mosaic had a clean, easily understood interface and was the first browser to display images in-line with text. In 1994, Marc Andreessen and Jim Clark founded Netscape, whose browser was by far the most popular browser through the mid-1990s. Netscape also developed the Secure Sockets Layer (SSL) protocol and many Internet server products, including mail servers and SSL-based Web servers. He is now a co-founder



Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

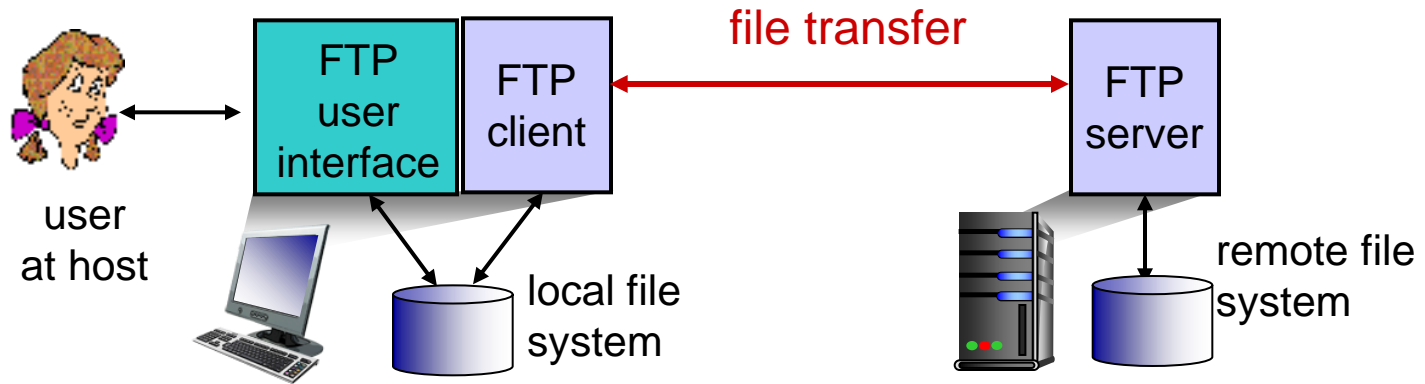
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

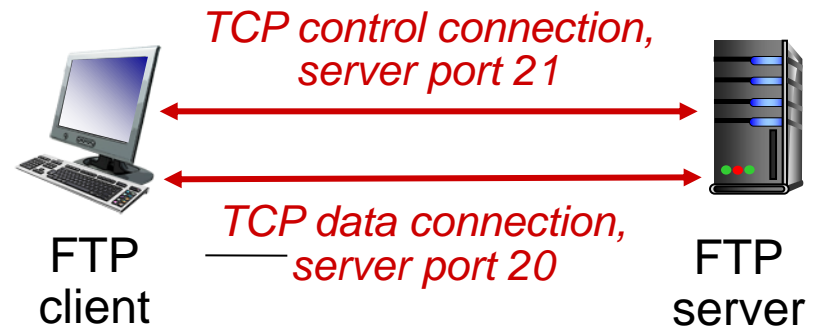
FTP: the file transfer protocol



- ❖ 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
- ❖ FTP server maintains “state”: current directory, earlier authentication

In practice

- ❖ Users do not directly write FTP commands.
- ❖ Instead, the FTP client/server software includes those commands.
 - Such as Filezilla
- ❖ Users operate on FTP through the APIs.
- ❖ Many files transfers today are through HTTP.

Next class

- ❖ Please read Chapter 2.3-2.4 of your textbook
BEFORE Class