



THE UNIVERSITY OF WINNIPEG

# ACS-3911-050 Computer Network

## Chapter 2 Application Layer

# ACS-3911-050 – Slides Used In The Course

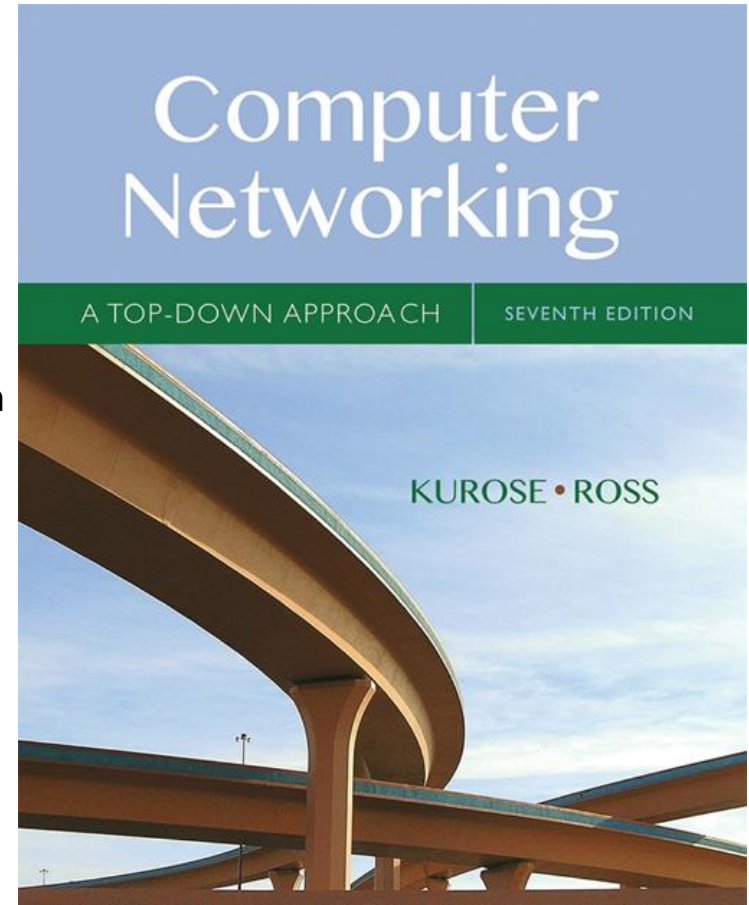
## A note on the use of these PowerPoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2016  
J.F Kurose and K.W. Ross, All Rights Reserved



## 2.1 principles of network applications

## 2.2 Web and HTTP

## 2.3 electronic mail

- SMTP, POP3, IMAP

## 2.4 DNS

## 2.5 P2P applications

## 2.6 video streaming and content distribution networks

## 2.7 socket programming with UDP and TCP

# Goal and Overview

---

- ❖ 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

## 2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

## Some Network Applications

- ❖ *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*
- ❖ ...
- ❖ ...

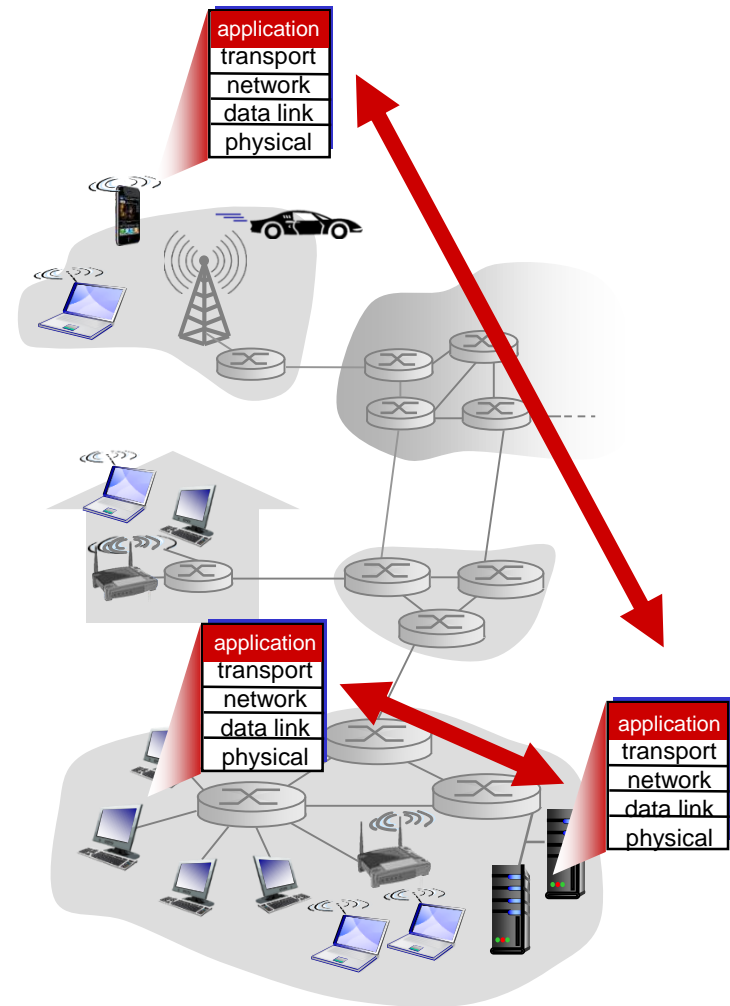
# Creating a Network Application

## 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



## Possible structure of applications:

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



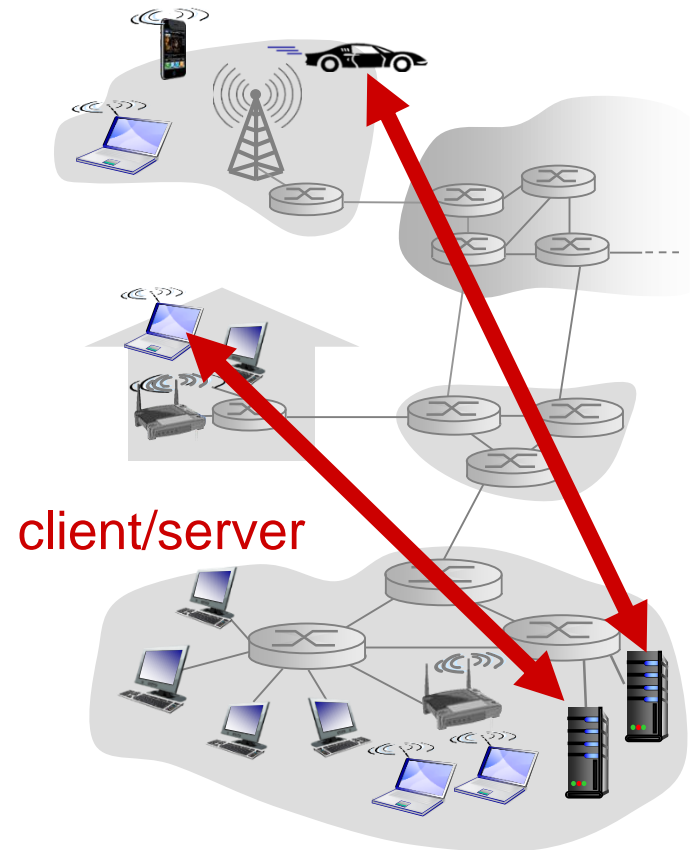
# Client-Server Architecture

## server:

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

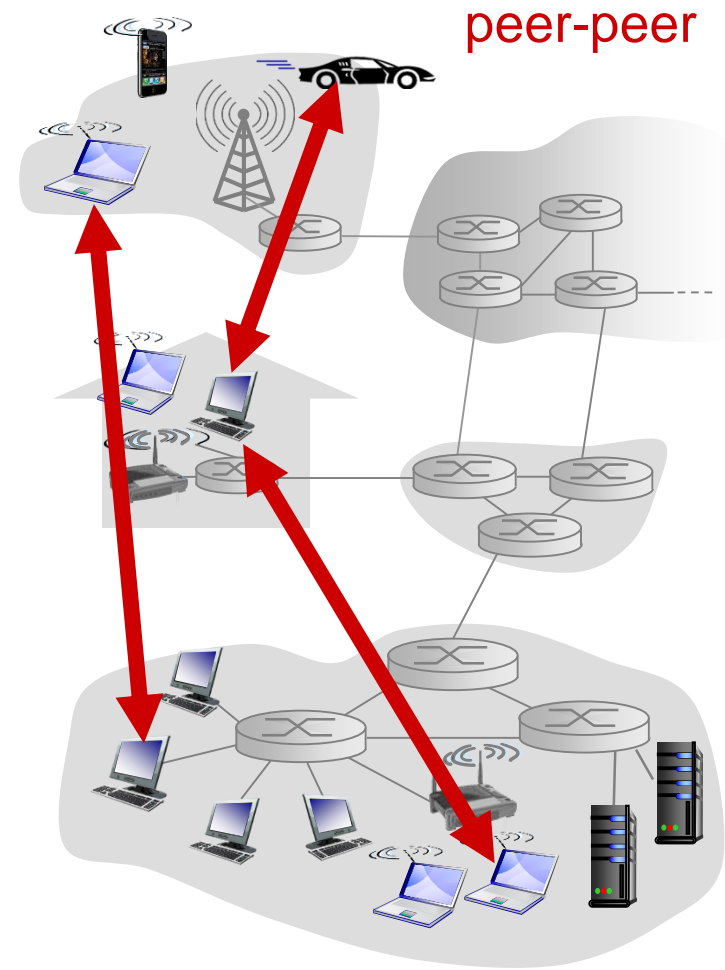
## clients:

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



# P2P Architecture

- *not* 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



- 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

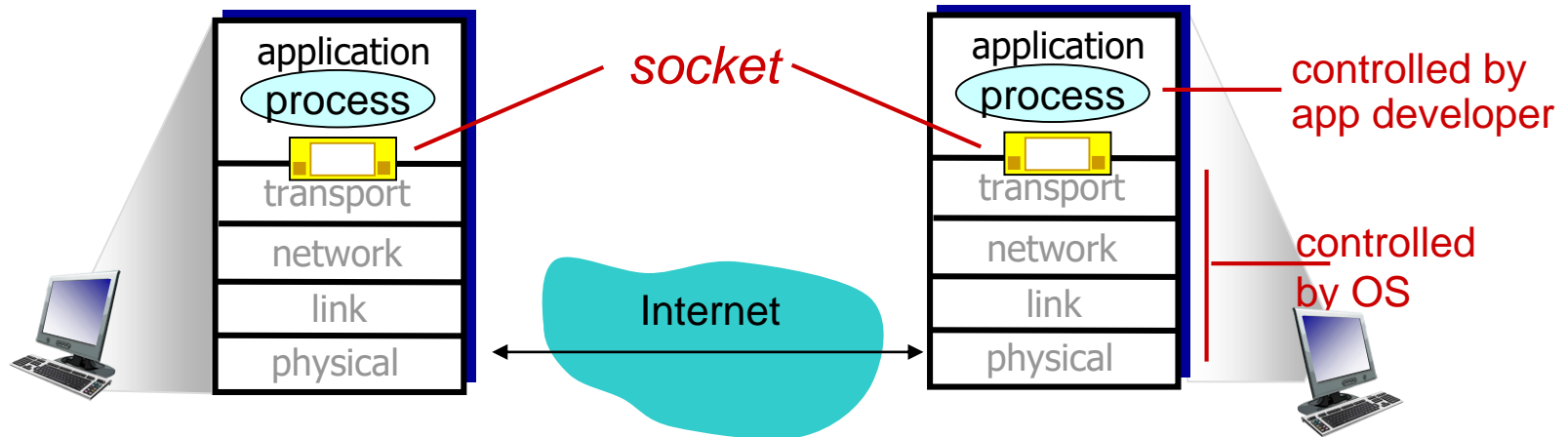
**client process**: process that initiates communication

**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
- ❖ more shortly...

# 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
  - allows for interoperability
  - e.g., HTTP, SMTP
- proprietary protocols:**
- e.g., Skype

## 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 Applications



<b>application</b>	<b>data loss</b>	<b>throughput</b>	<b>time sensitive</b>
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



## *TCP (Transmission Control Protocol) 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 (User Datagram Protocol) service:*

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

Q: why bother? Why is there a UDP?

<b>application</b>	<b>application layer protocol</b>	<b>underlying transport protocol</b>
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

## TCP & UDP

- no encryption
- Clear text passwords sent into socket traverse Internet in clear text

## SSL (Secure Sockets Layer)

- 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

- Clear text passwords sent into socket traverse Internet encrypted
- See Chapter 8

2.1 principles of network applications

**2.2 Web and HTTP**

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

## *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`

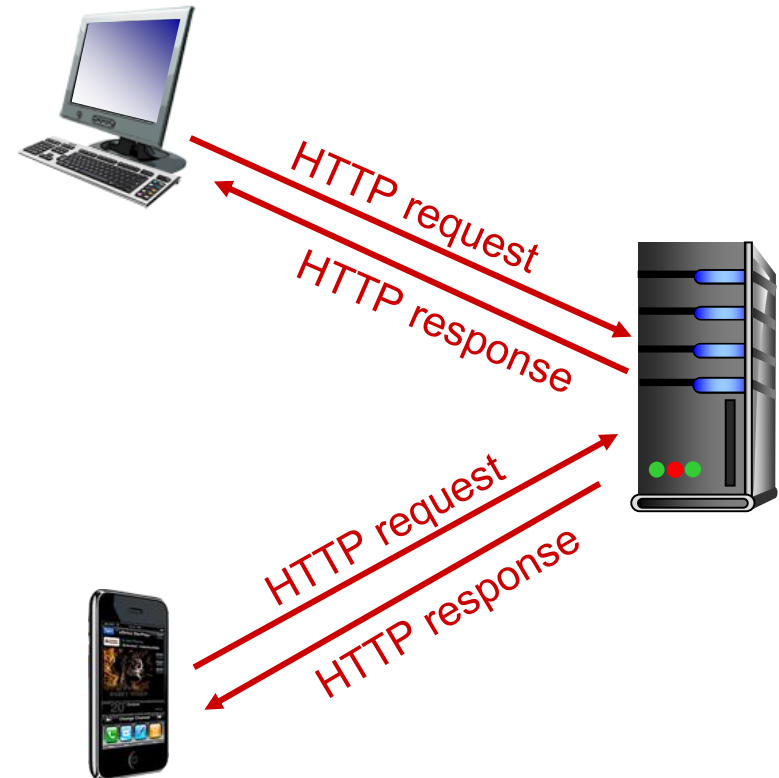
host name

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 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

## *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

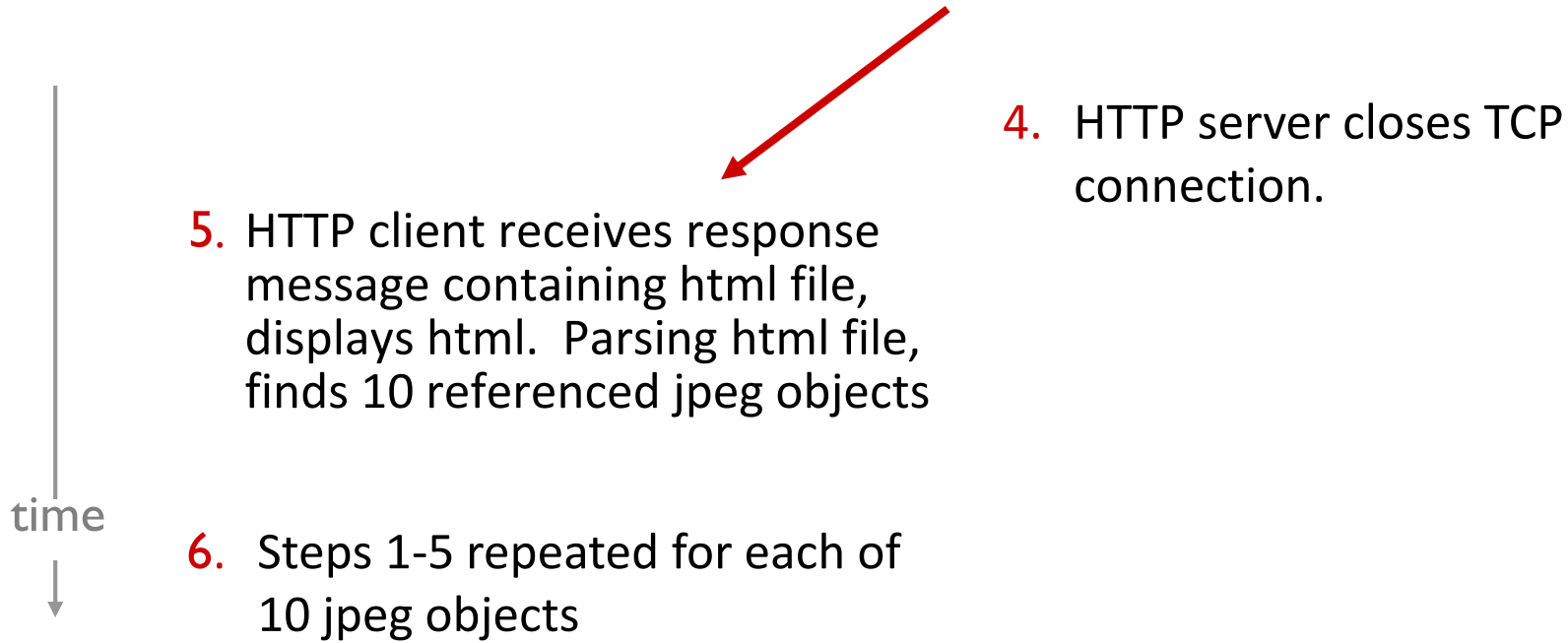
2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`

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

time



# Non-persistent HTTP (cont.)



# Non-persistent HTTP: Response Time

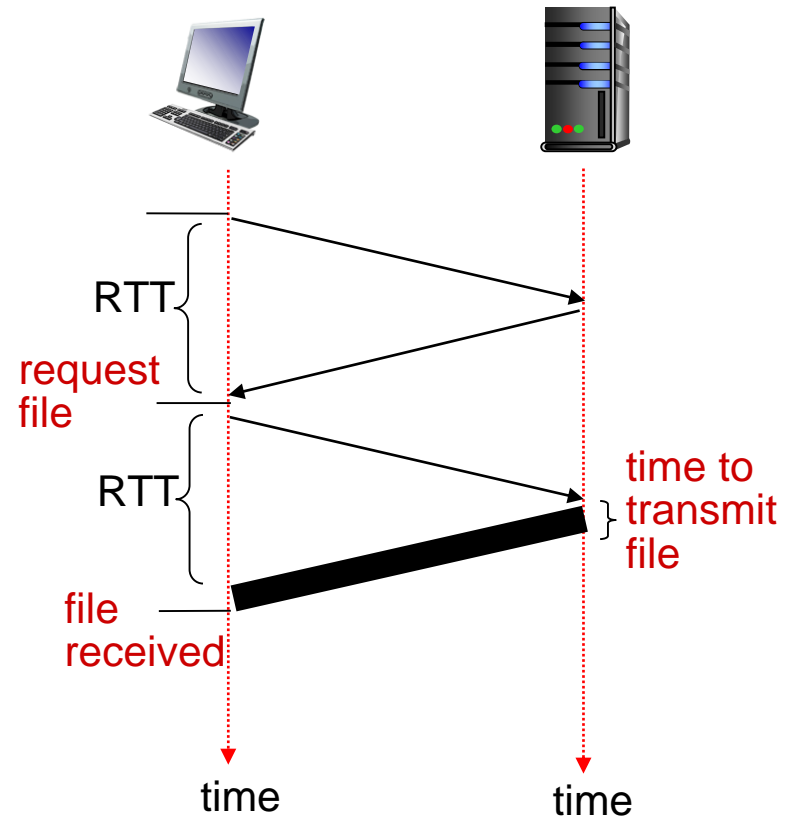
## RTT ((Round Trip Time) definition):

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 =

$2RTT + \text{file transmission time}$



# 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)

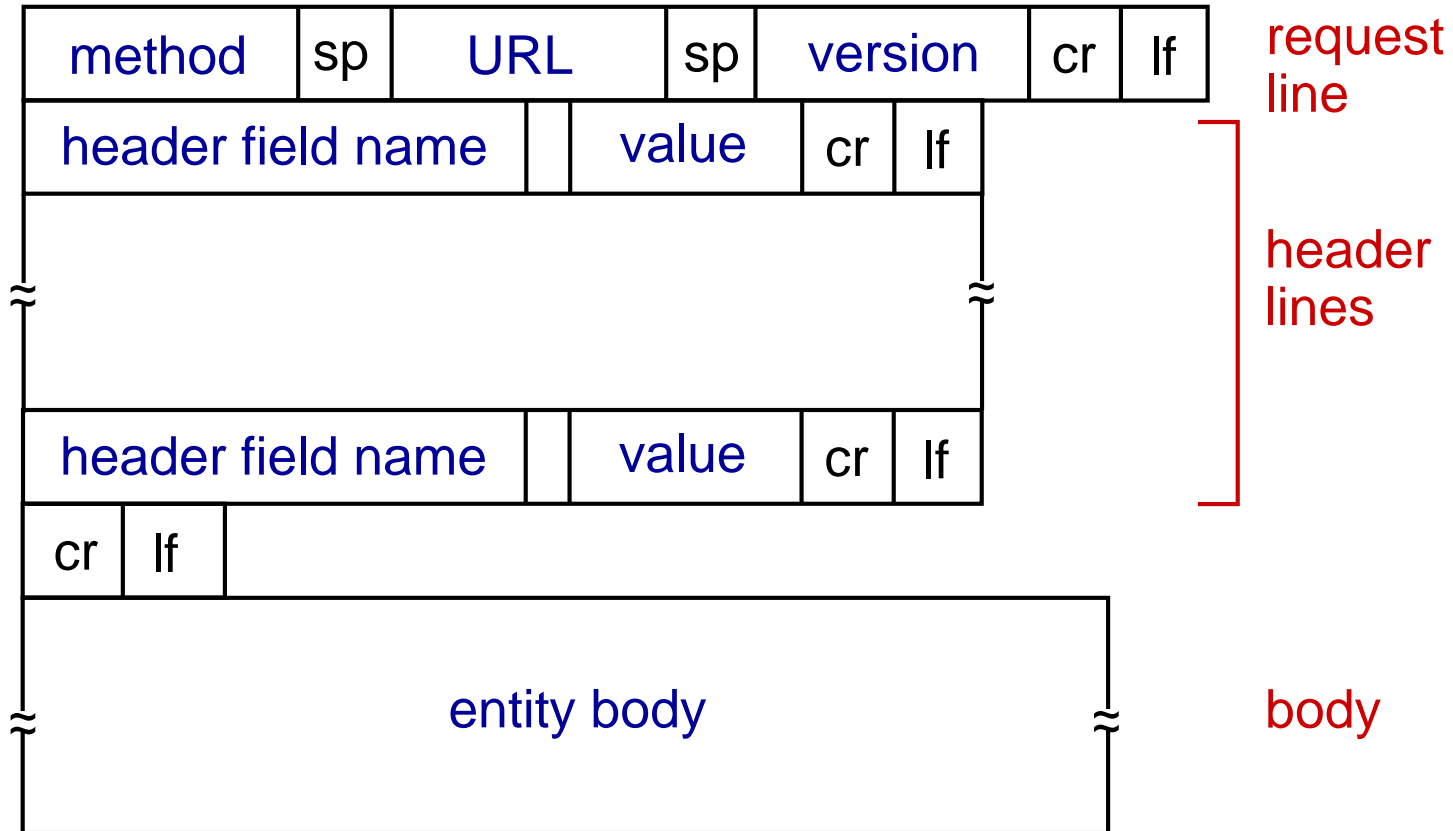
header  
lines

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 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

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 message (Location:)

## **400 Bad Request**

- request message not understood by server

## **404 Not Found**

- requested document not found on this server

## **505 HTTP Version Not Supported**

# Trying 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)

# User-server State: Cookie

---

Many Web sites use cookies

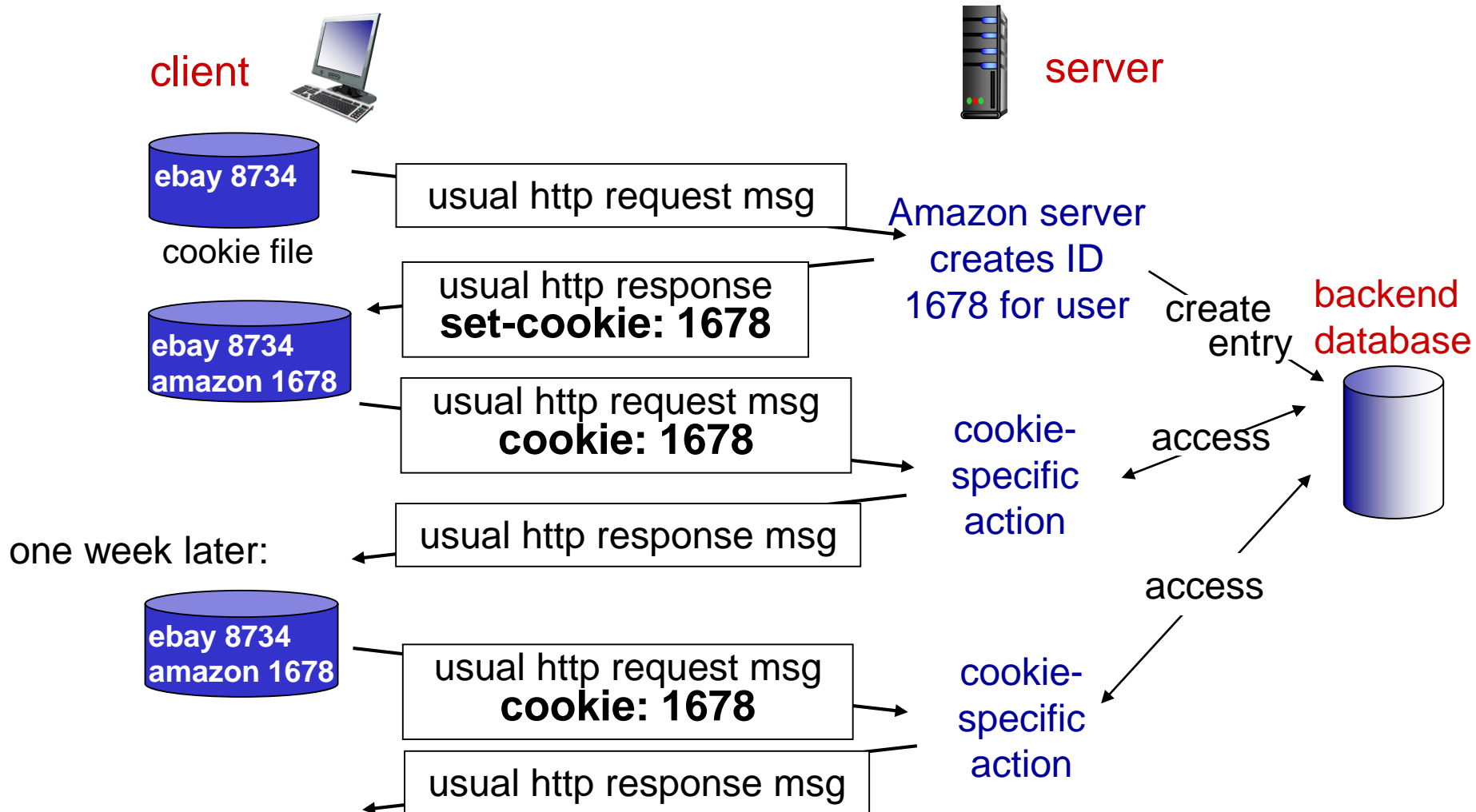
## *Four components of Cookie:*

- 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 (cont.)

*what cookies can be used for:*

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

*how to keep “state”:*

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

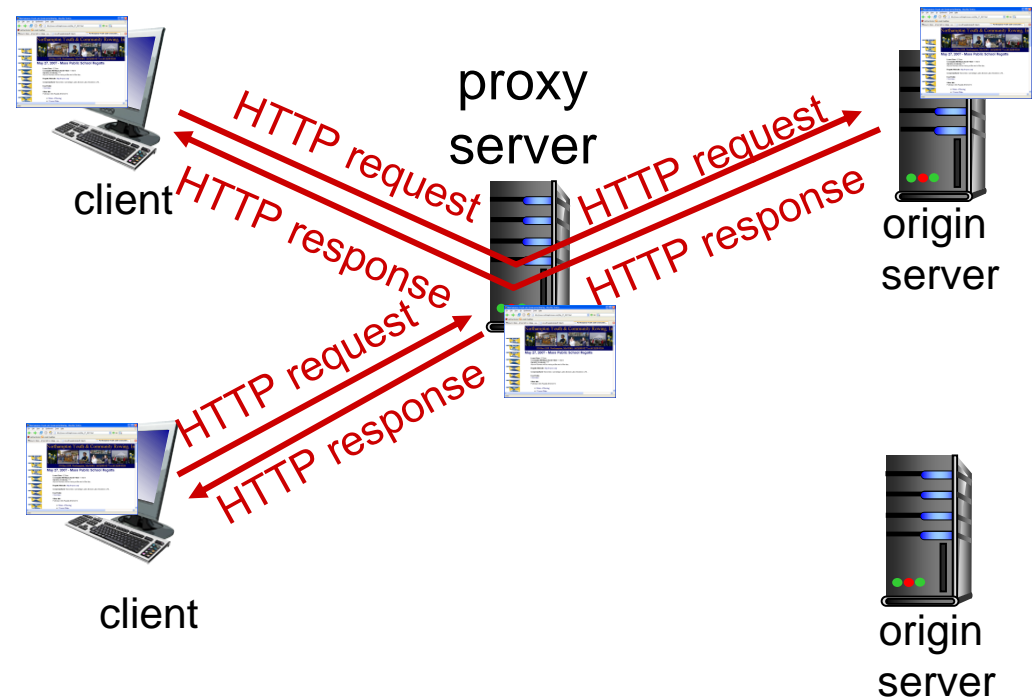
*aside*  
*cookies and privacy:*

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

# 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)



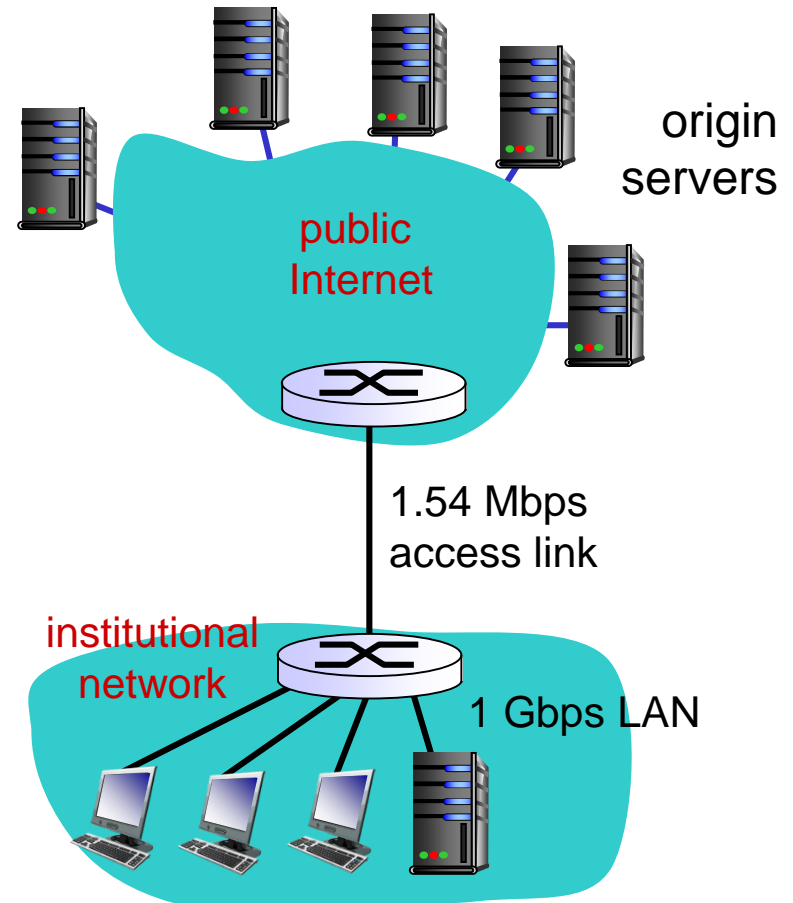
# Caching Example

## *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:*

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



# Caching Example: Faster Access Link

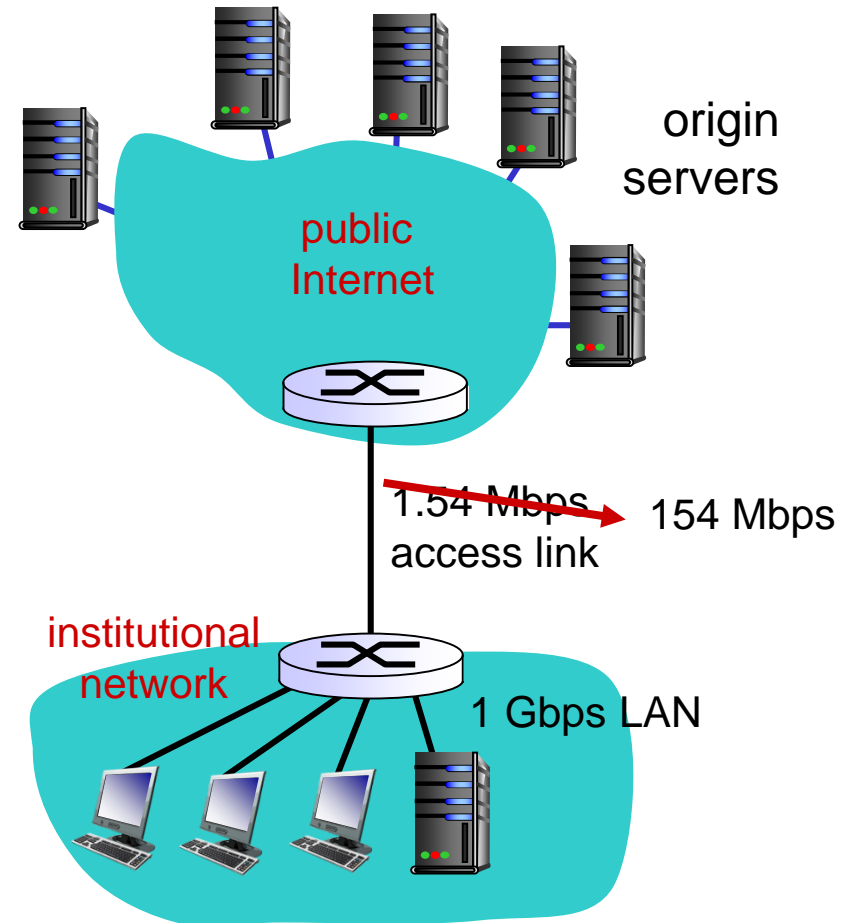
## assumptions:

- ❖ 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~~ → usecs  
msecs

**Cost:** increased access link speed (not cheap!)



# Caching Example: Install Local Cache



## *assumptions:*

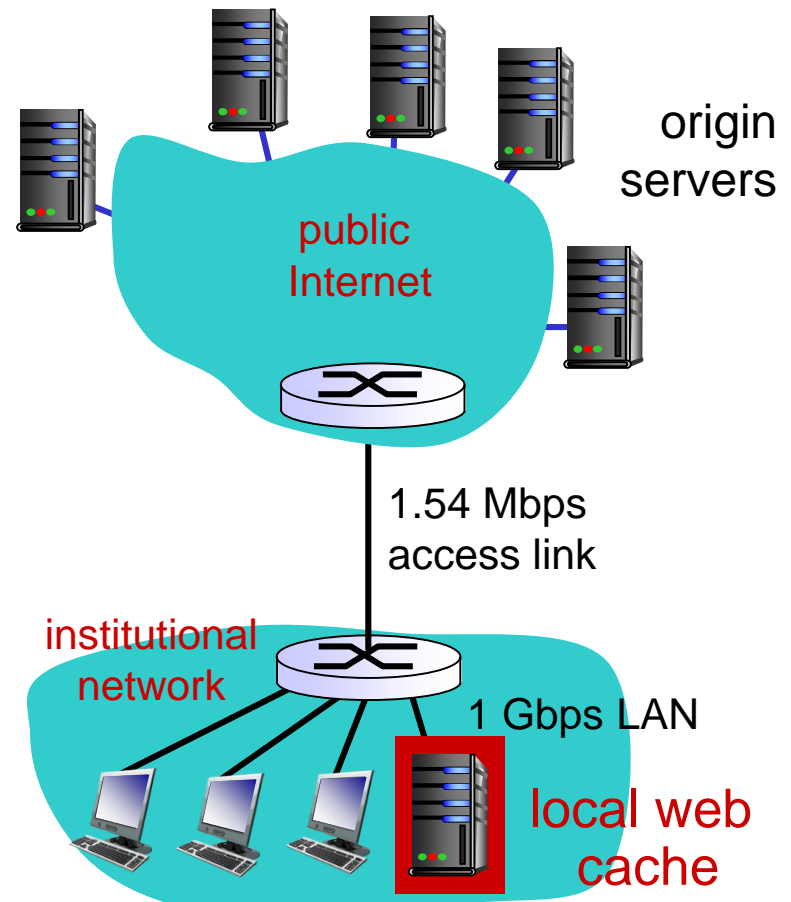
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

## *consequences:*

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

*How to compute link utilization, delay?*

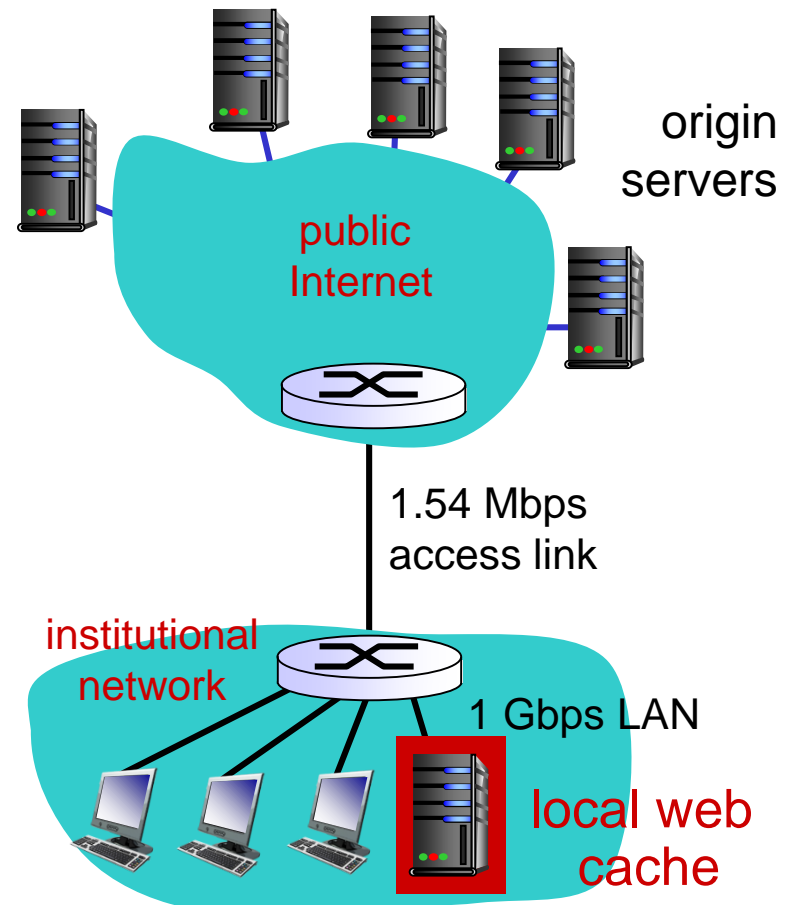
*Cost:* web cache (cheap!)



# 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!)





**QUESTIONS**

**now**