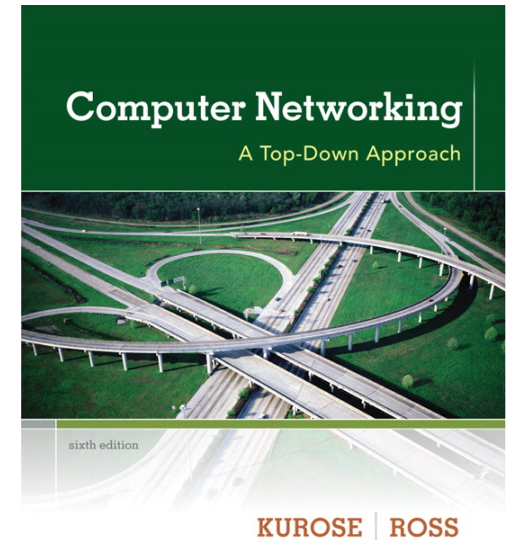


# Chapter 2

# Application Layer



*Computer  
Networking: A Top  
Down Approach*  
6<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Addison-Wesley  
March 2012

All material copyright 1996-2012  
J.F Kurose and K.W. Ross, All Rights Reserved

©

# 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`

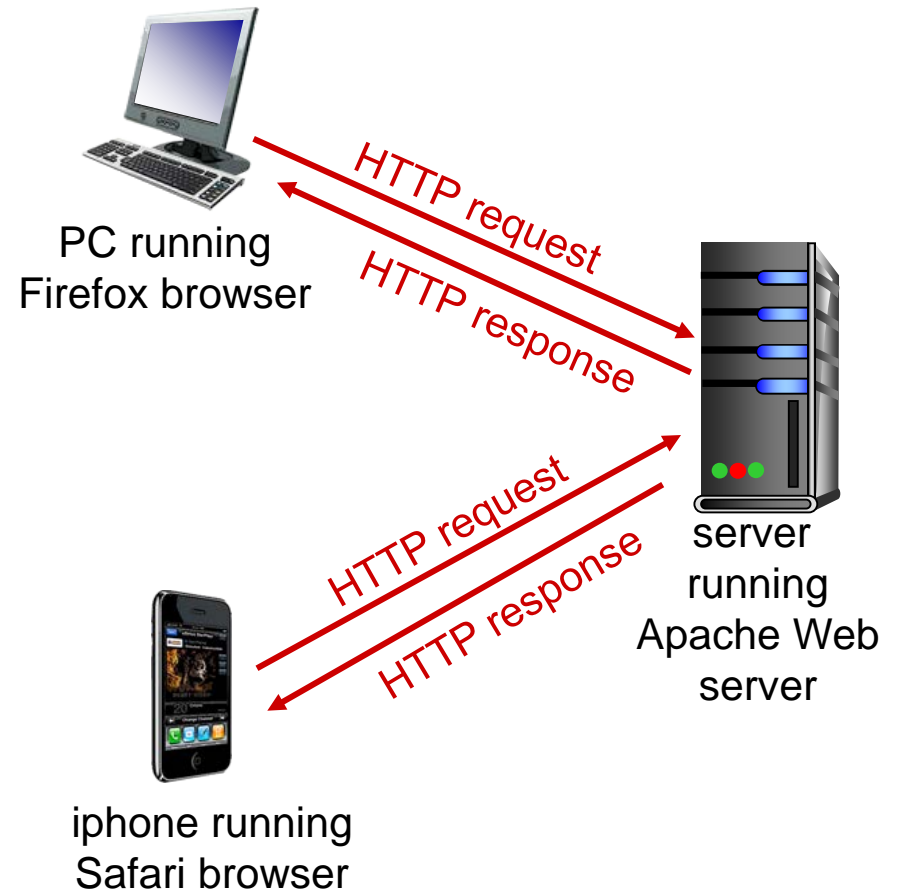
host name

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 is "stateless"*

- ❖ server maintains no information about past client requests

*aside*  
protocols that maintain "state" are complex!

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

# HTTP connections

## *non-persistent HTTP*

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

## *persistent HTTP*

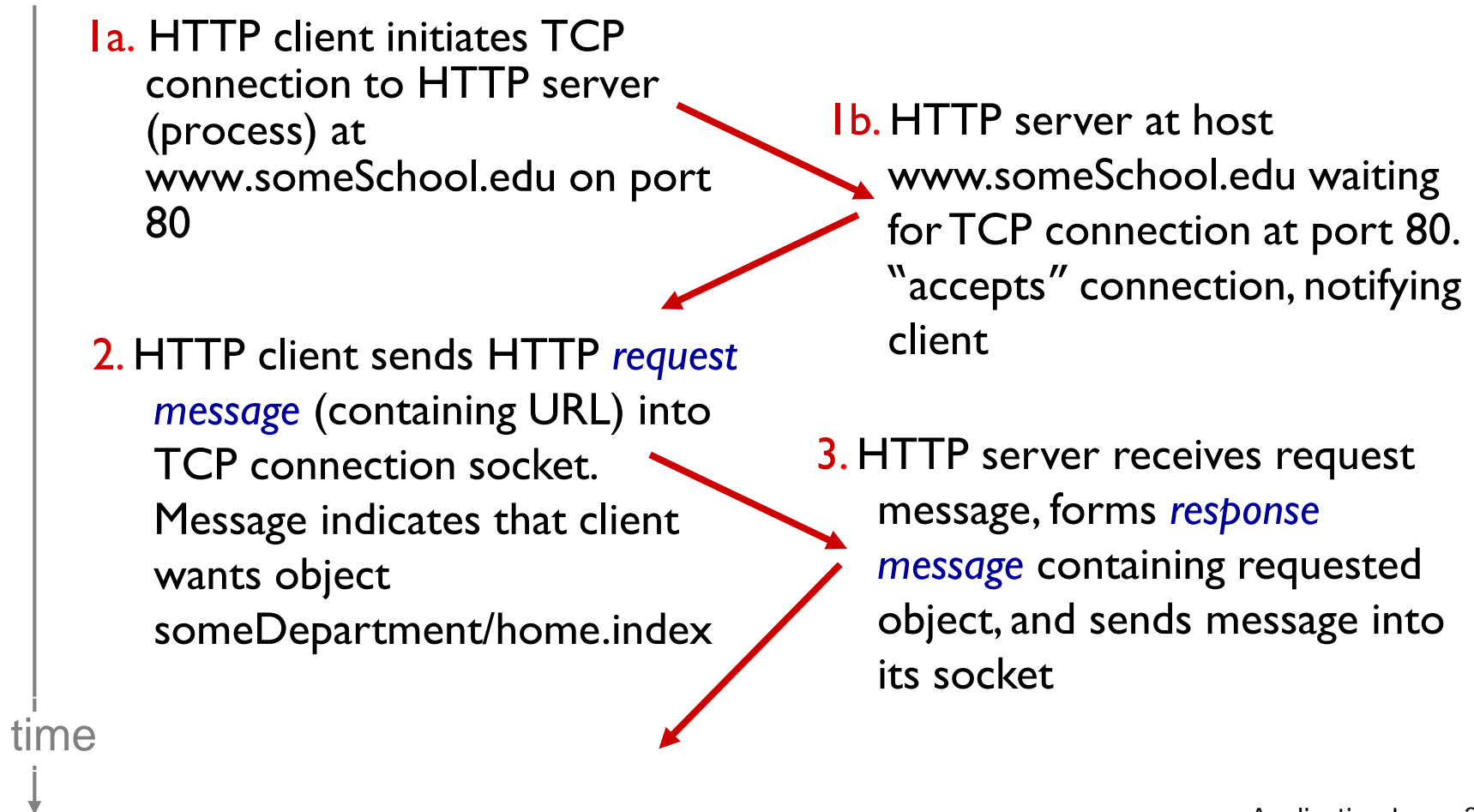
- ❖ multiple objects can be sent over single TCP connection between client, server

# Non-persistent HTTP

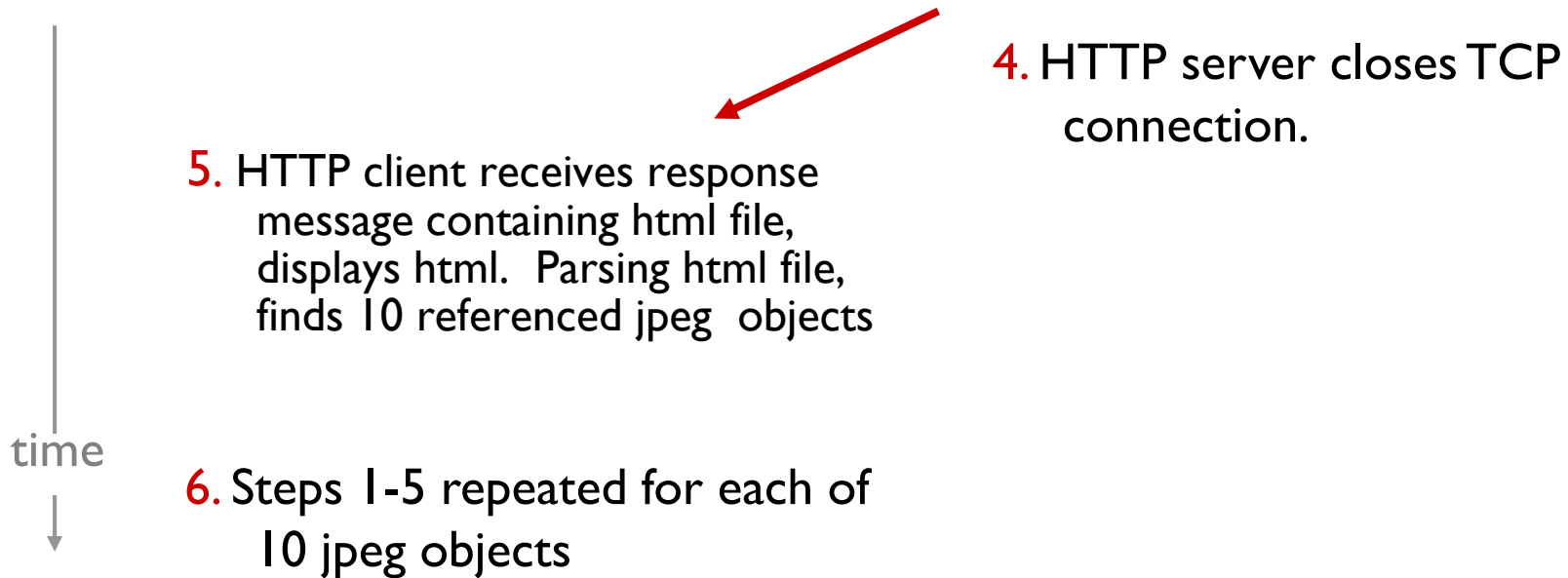
suppose user enters URL:

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

(contains text,  
references to 10  
jpeg images)



# Non-persistent HTTP (cont.)



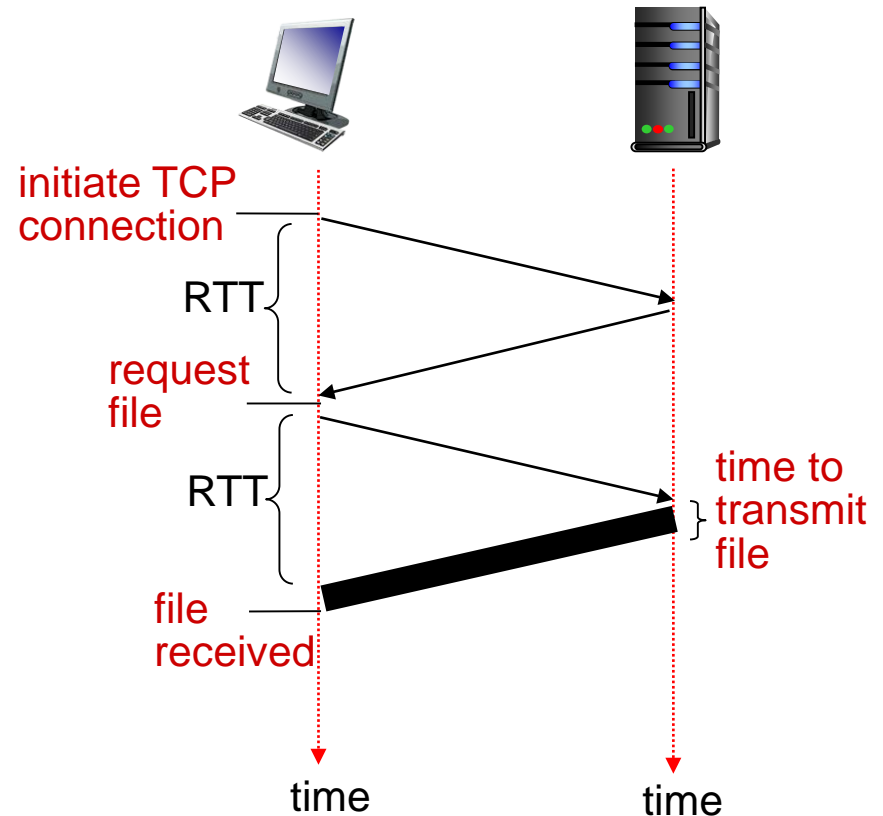


# Non-persistent HTTP: response time

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

**HTTP response time:**

- ❖ one RTT to initiate TCP connection
- ❖ one RTT for HTTP request and first few bytes of HTTP response to return
- ❖ file transmission time
- ❖ non-persistent HTTP response time =  
 $2\text{RTT} + \text{file transmission time}$



# 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

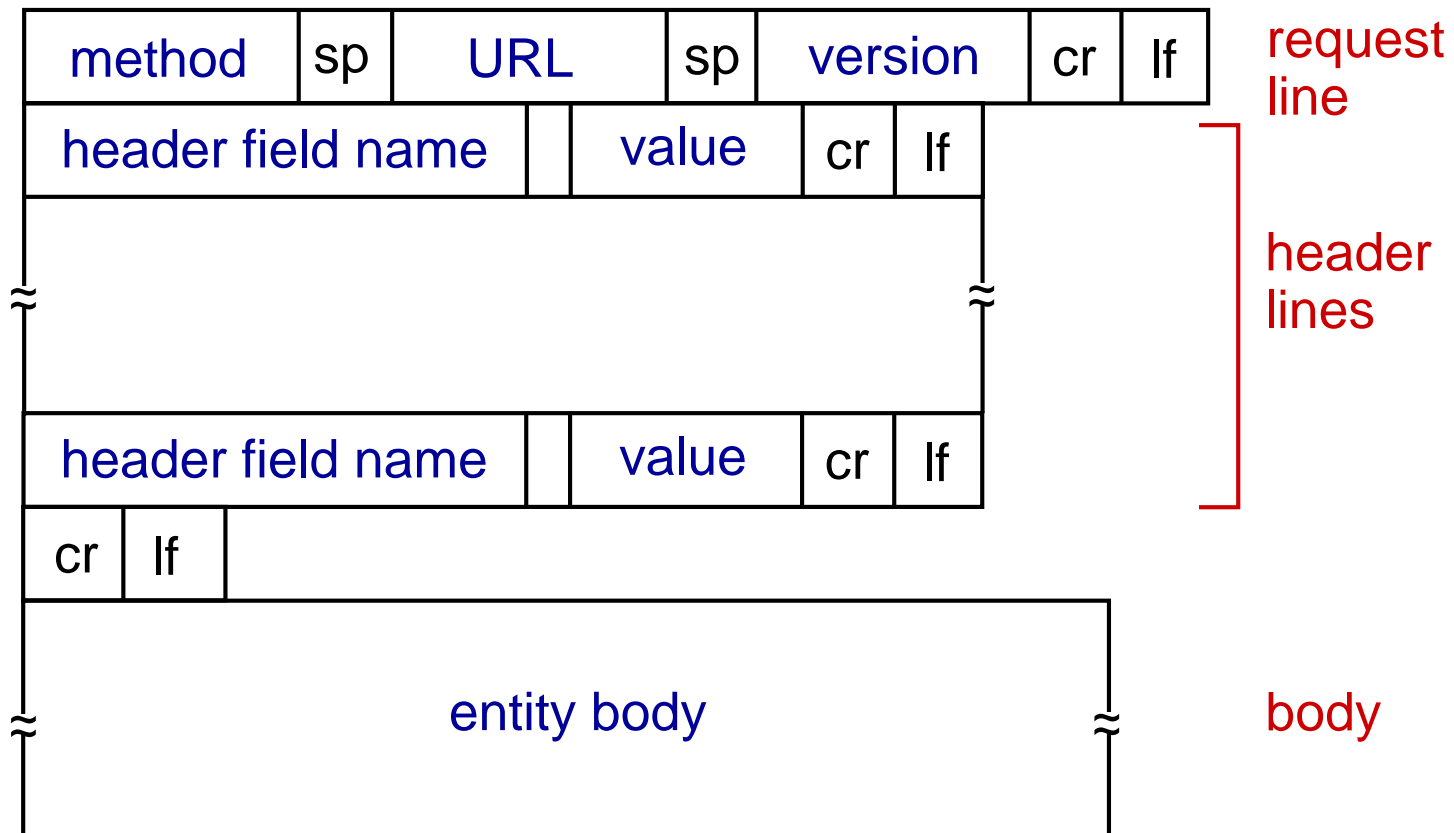
carriage return,  
line feed at start  
of line indicates  
end of header lines

```
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

The diagram shows an example of an HTTP request message in ASCII format. It is divided into three main sections: the request line, header lines, and the end of the header section. The request line is 'GET /index.html HTTP/1.1\r\n'. The header lines are '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', and 'Connection: keep-alive\r\n'. The final line is '\r\n'. Annotations with arrows point to these sections: 'request line (GET, POST, HEAD commands)' points to the first line; 'header lines' points to the block of lines between the request line and the final '\r\n'; 'carriage return, line feed at start of line indicates end of header lines' points to the final '\r\n' line. Additionally, two arrows point to the '\r\n' at the end of the first line, labeled 'carriage return character' and '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

```
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, e.g.,  
requested  
HTML file

```
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:*

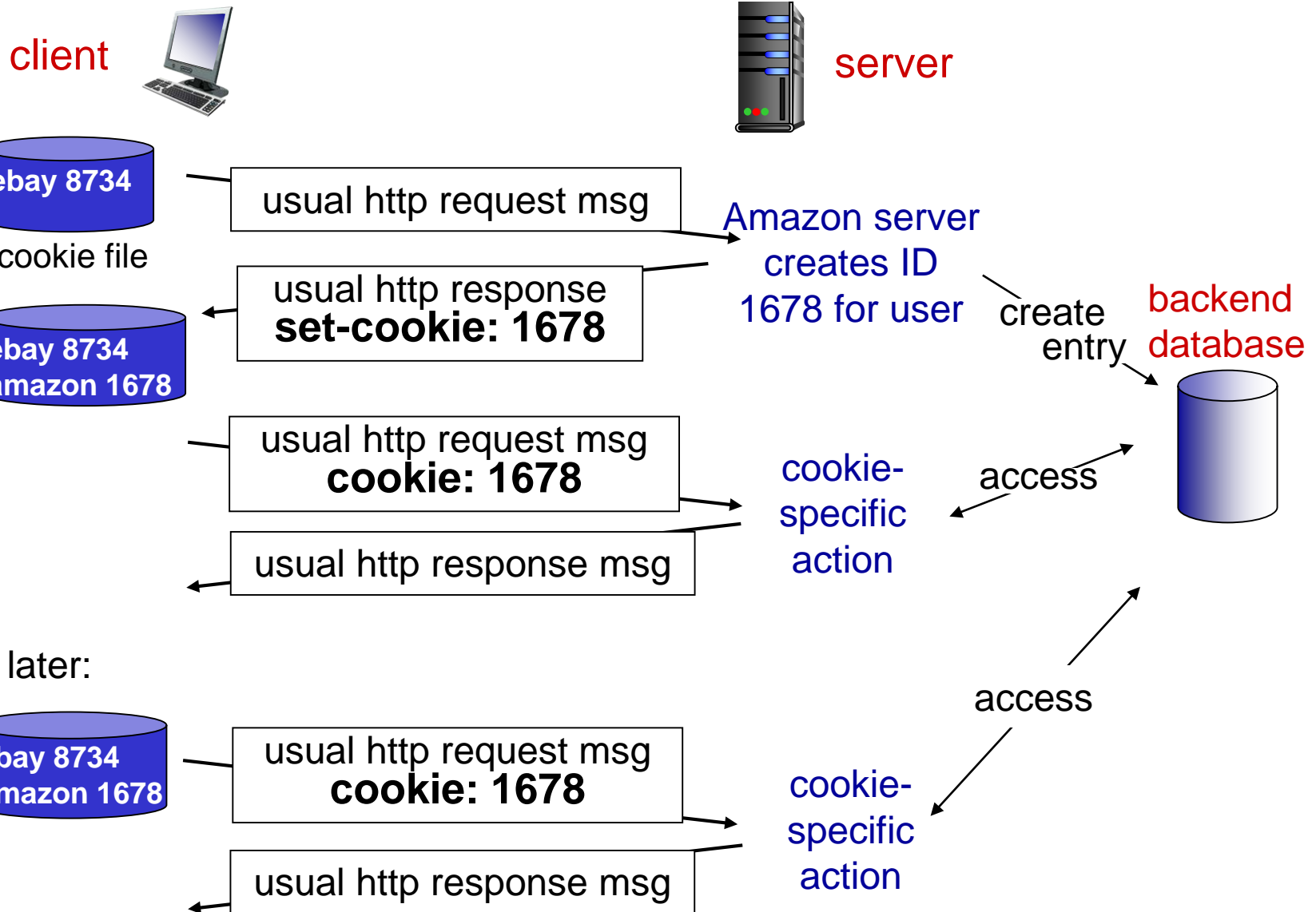
- 1) cookie header line of HTTP *response* message
- 2) cookie header line in next HTTP *request* message
- 3) cookie file kept on user's host, managed by user's browser
- 4) back-end database at Web site

**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

View cookies in Firefox

# Cookies: keeping "state" (cont.)



# 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

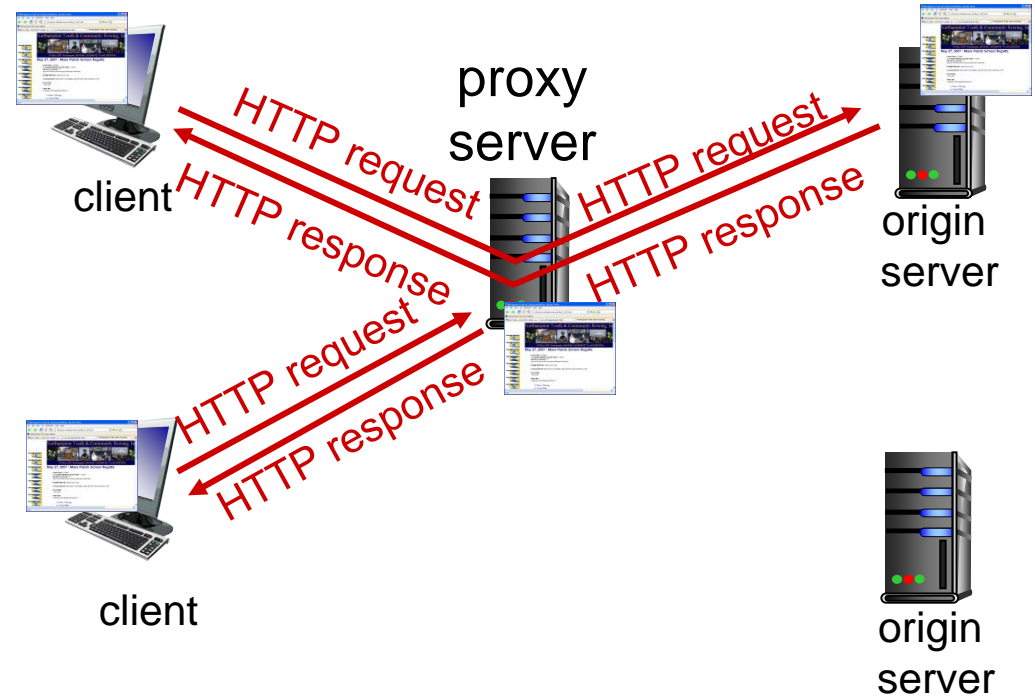
*how to keep "state":*

- ❖ 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
- ❖ Internet dense with caches: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

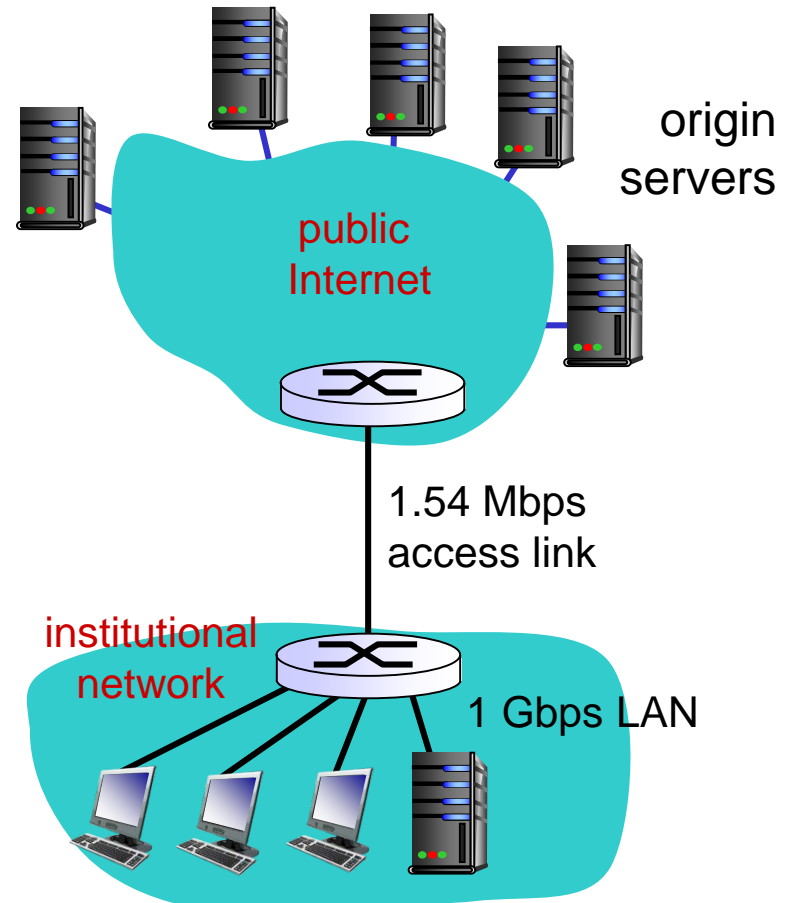
# Caching example:

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



# Caching example: install local cache

## *assumptions:*

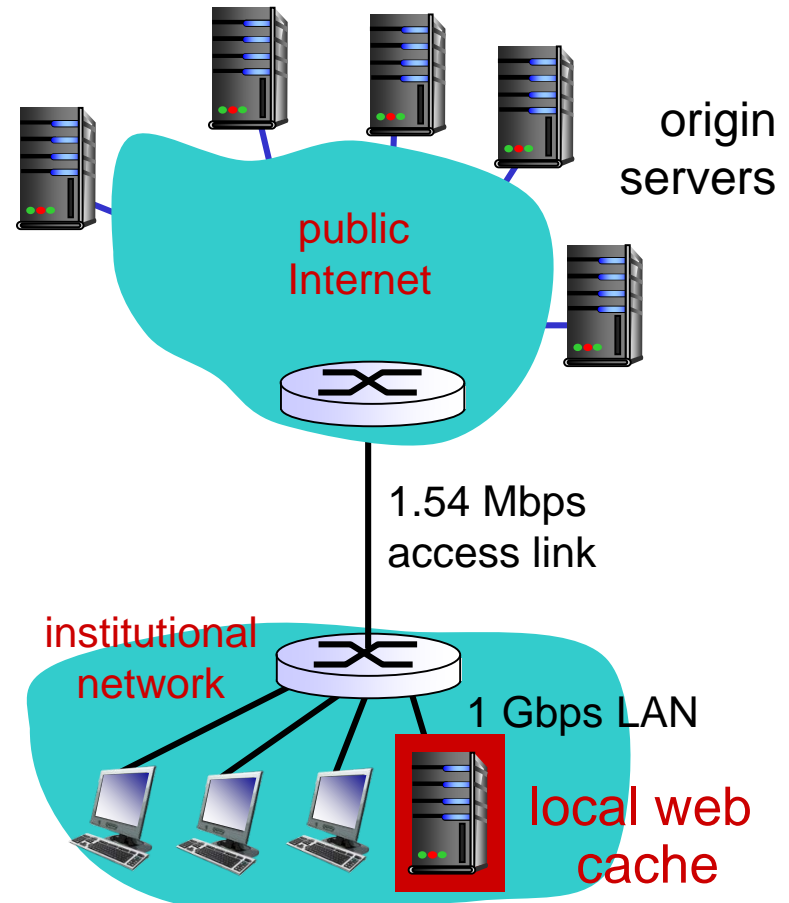
- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

## *consequences:*

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

*How to compute link utilization, delay?*

*Cost:* web cache (cheap!)



# Conditional GET

- ❖ **Goal:** don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- ❖ **cache:** specify date of cached copy in HTTP request

`If-modified-since: <date>`
- ❖ **server:** response contains no object if cached copy is up-to-date:

`HTTP/1.0 304 Not Modified`

