Designing Web Applications

HTTP

Francesco Marcelloni

Department of Information Engineering
University of Pisa
ITALY

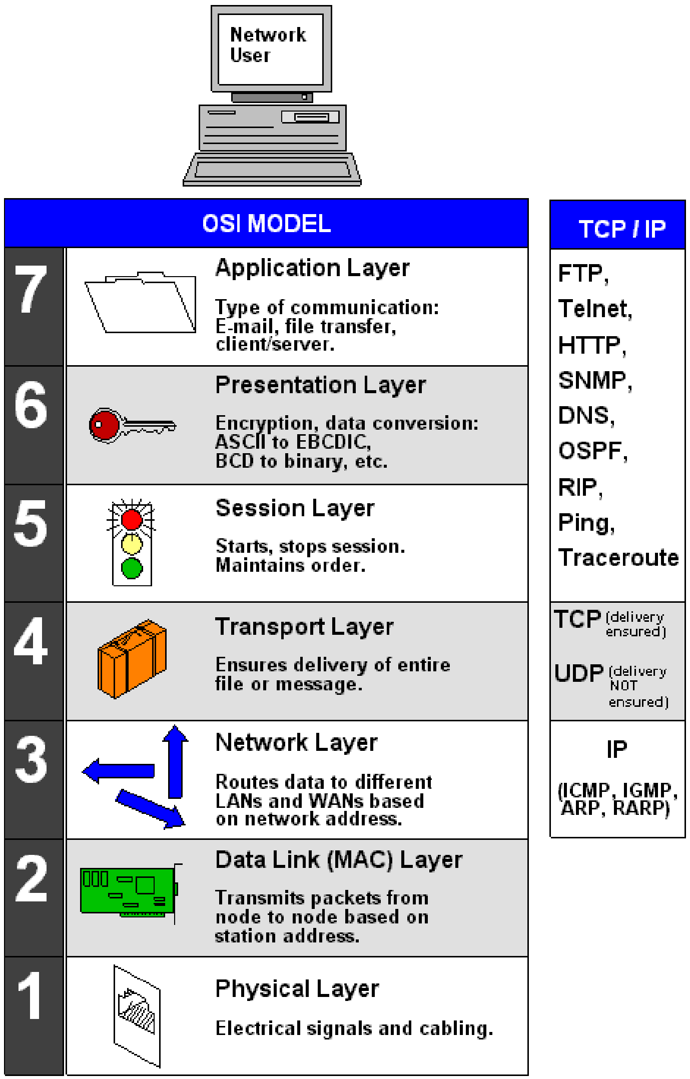
Acknowledgement

I acknowledge that some slides have been adapted from (or are just identical to) the slides provided with the following book:

Thee Computer Networking: A Top Down Approach , 5th edition. Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

TCP/IP Stack

From Computer Desktop Encyclopedia
© 2003 The Computer Language Co. Inc.



OSI MODEL		TCP / IP
7	**Application Layer** Type of communication: E-mail, file transfer, client/server.	FTP, Telnet, HTTP, SNMP, DNS, OSPF, RIP, Ping, Traceroute
6	**Presentation Layer** Encryption, data conversion: ASCII to EBCDIC, BCD to binary, etc.	
5	**Session Layer** Starts, stops session. Maintains order.	
4	**Transport Layer** Ensures delivery of entire file or message.	TCP (delivery ensured) UDP (delivery NOT ensured)
3	**Network Layer** Routes data to different LANs and WANs based on network address.	IP (ICMP, IGMP, ARP, RARP)
2	**Data Link (MAC) Layer** Transmits packets from node to node based on station address.	
1	**Physical Layer** Electrical signals and cabling.	

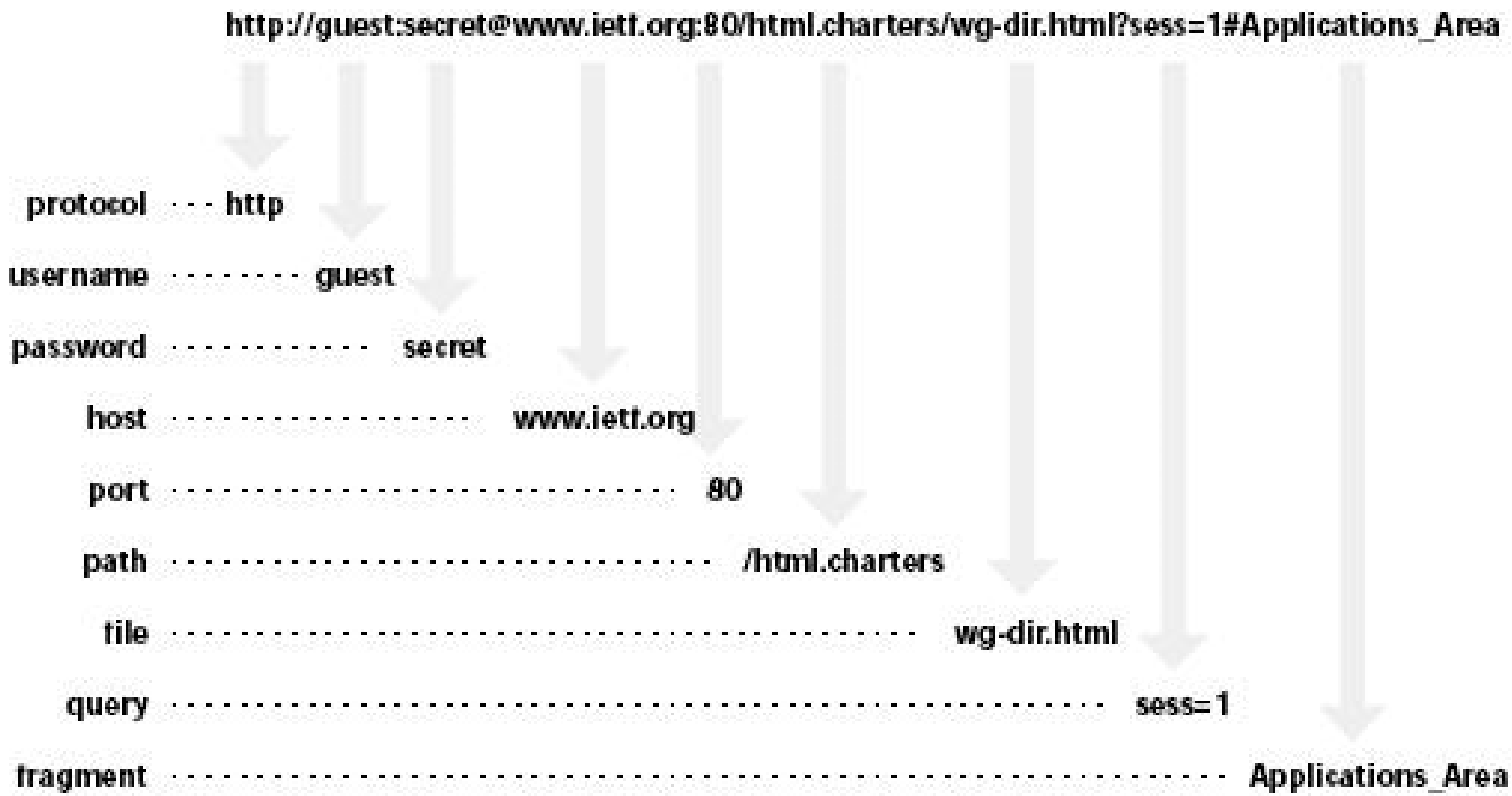
HTTP Overview

First some jargon

- Web page consists of objects
- Objects can be HTML files, JPEG images, Java applets, audio files,…
- Web page consists of base HTML-file which includes several referenced objects
- Each object is addressable by a URI
- Example URI:

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

HTTP Overview



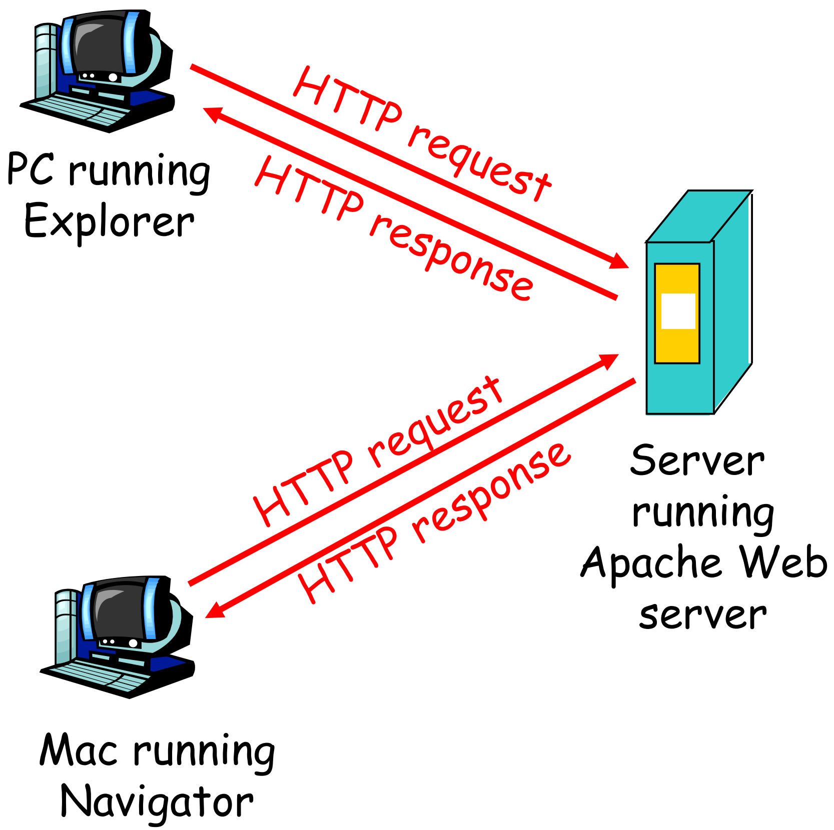
http://guest:secret@www.ietf.org:80/html.charters/wg-dir.html?sess=1#Applications_Area

protocol	http
username	guest
password	secret
host	www.ietf.org
port	80
path	/html.charters
file	wg-dir.html
query	sess=1
fragment	Applications_Area

HTTP Overview

HTTP: hypertext transfer protocol

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



HTTP Overview

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

Nonpersistent HTTP

- At most one object is sent over a TCP connection.

Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.

Nonpersistent HTTP

Suppose user enters URL

(contains text, references to 10 jpeg images)

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

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

Nonpersistent HTTP (continued)

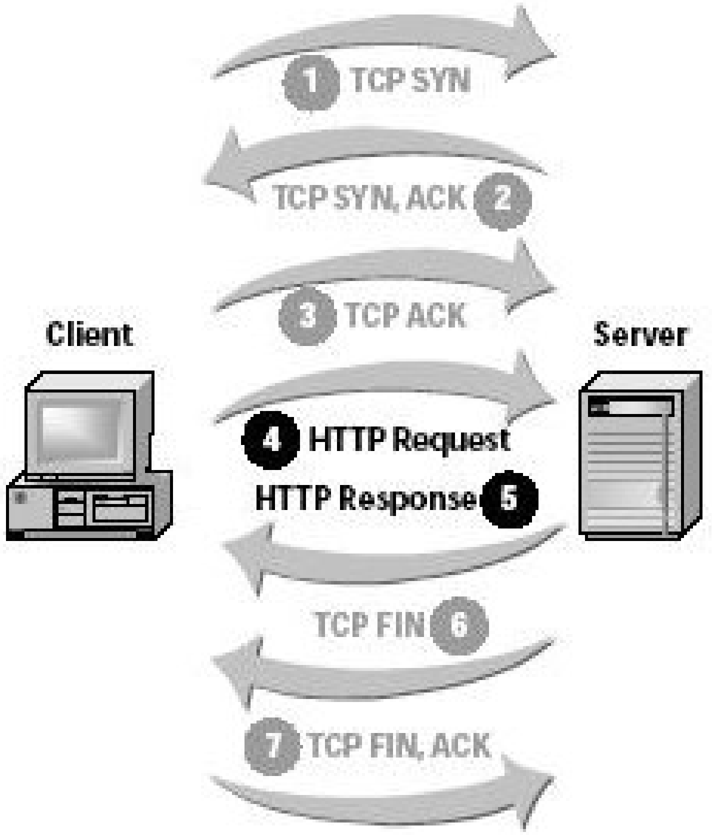
4. HTTP server closes TCP connection.

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

Nonpersistent HTTP (continued)



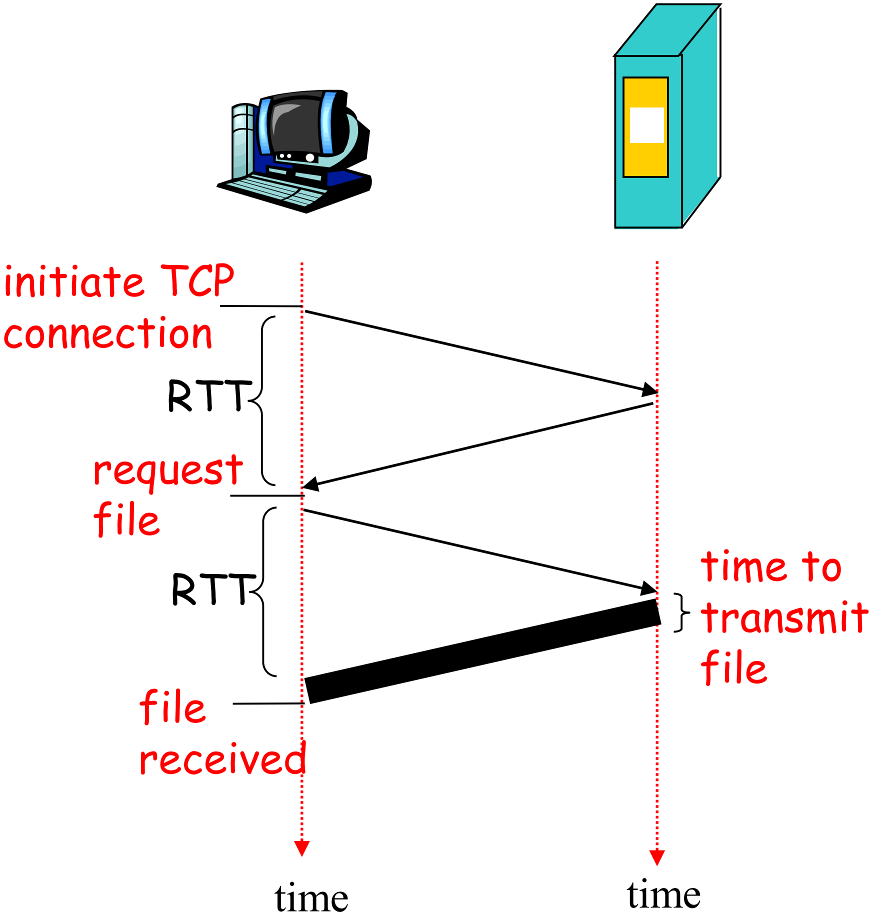
Nonpersistent HTTP: Response Time

Definition of Round-Trip Time (RTT): time for a small packet to travel from client to server and back.

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

total = 2RTT+transmit time



Persistent HTTP

Nonpersistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection (for each connection, TCP buffers and TCP variables have to be allocated in the client and in the server)
- 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
- Pipelining version:
  - client sends requests as soon as it encounters a referenced object
  - as little as one RTT for all the referenced objects

Persistent HTTP

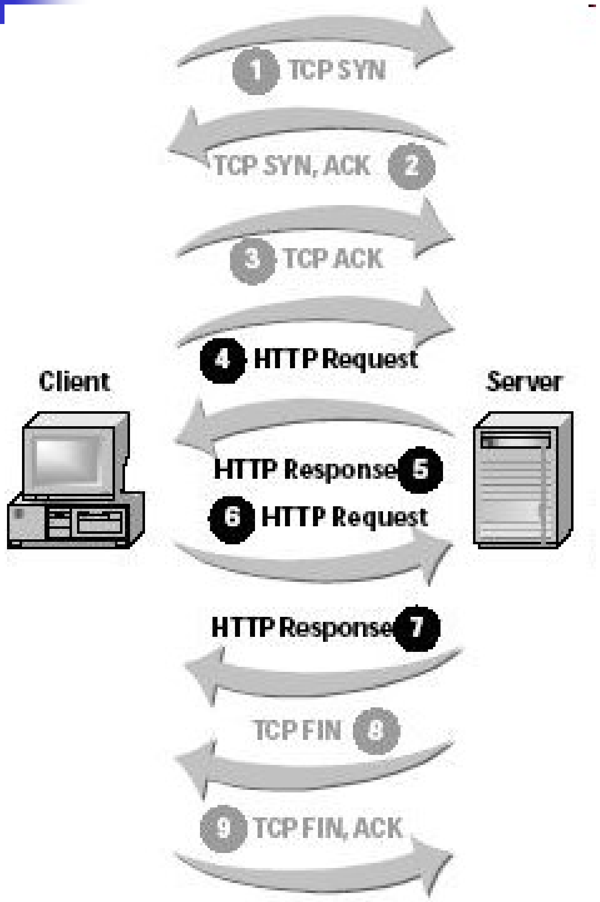
Two versions of persistent connections

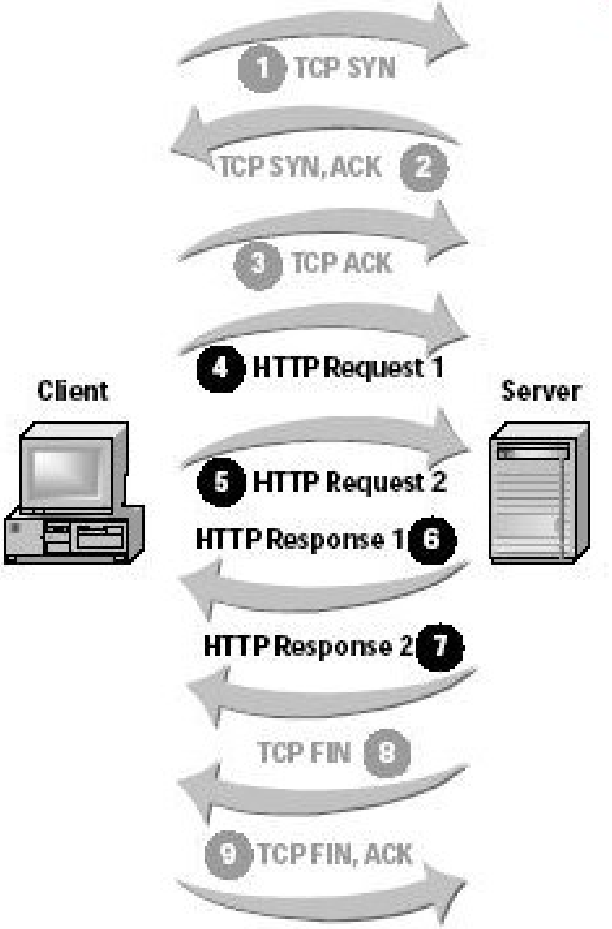
- Without pipelining
  - The client issues a new request only when the previous response has been received
  - The client experiences one RTT in order to request and receive each of the referenced objects
  - After the server sends an object over the persistent TCP connection, the connection idles while it waits for another request

Persistent HTTP

- With pipelining (default mode in HTTP/1.1)
  - The client issues a new request as soon as it encounters a reference.
  - It is possible for only one RTT to be expended for all the referenced objects
  - The pipelined TCP connection remains idle for a smaller fraction of time

Persistent HTTP



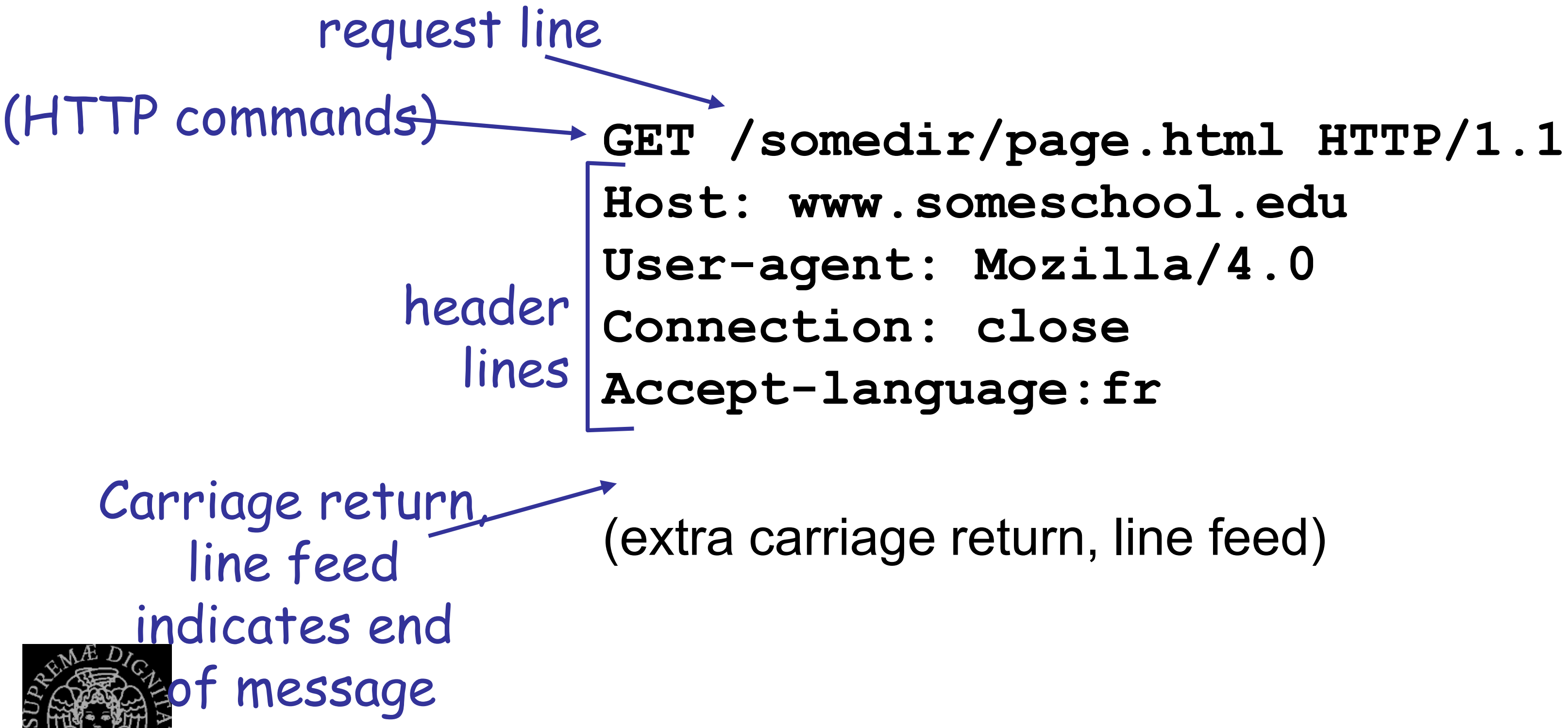


Without Pipelining

With Pipelining

HTTP Request Message

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



request line

(HTTP commands)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr
```

header lines

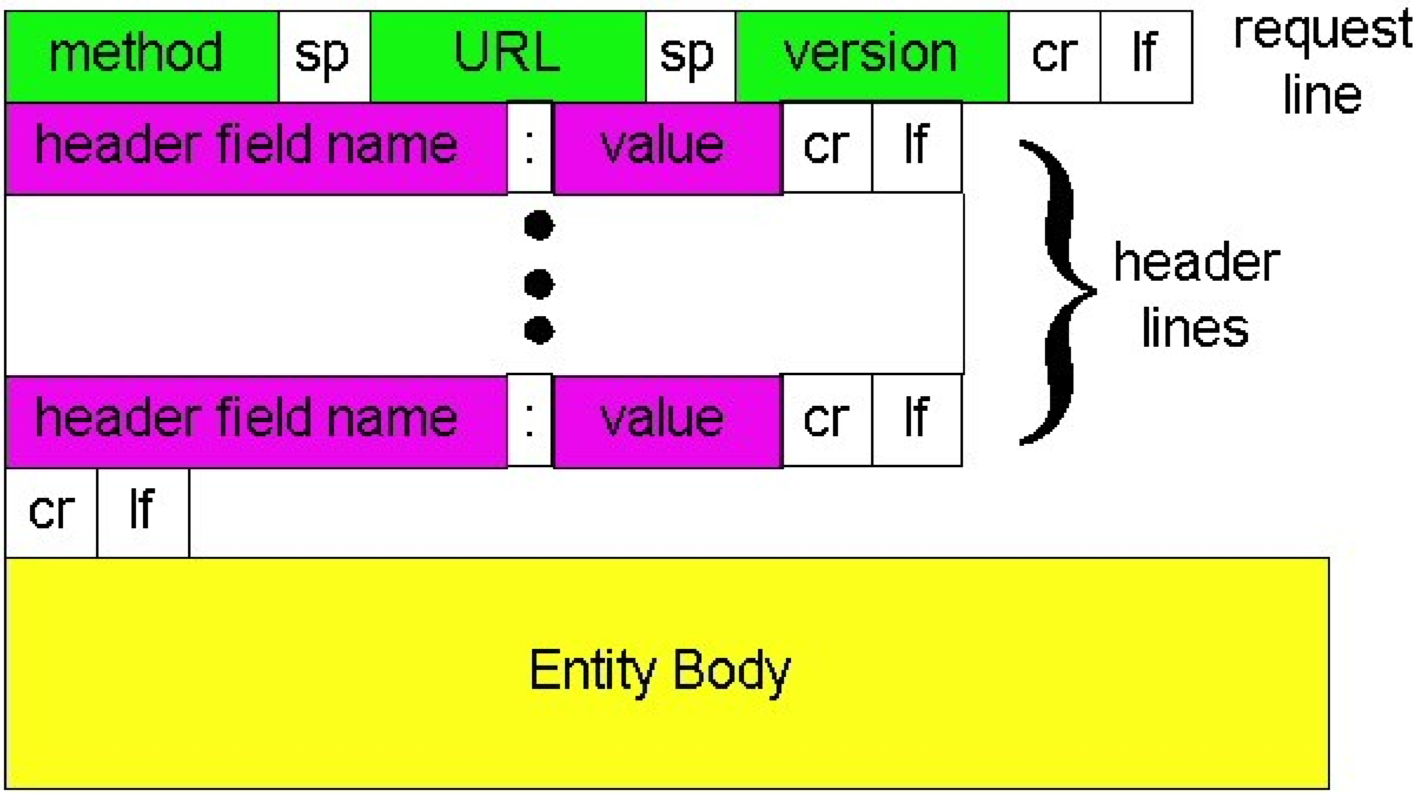
Carriage return, line feed indicates end of message

(extra carriage return, line feed)

HTTP Request Message

- Connection: close
  - The browser is telling the server that it does not want to use persistent connections (when it wants, it uses Keep-Alive)
- User-agent
  - Specifies the user agent, that is, the browser that is making the request to the server
  - Useful: the server could send different versions of the same object to different types of user agent
- Accept-language
  - The user prefers to receive a French version of the object, if such an object exists on the server; otherwise the server should send its default version

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

header lines

cr | lf

Entity Body

HTTP Request Message

- Web page often include form input

Post method:

- Input is uploaded to server in entity body

GET method:

- Input is uploaded in URL field of request line:

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

HTTP Request Message

HTTP/1.0

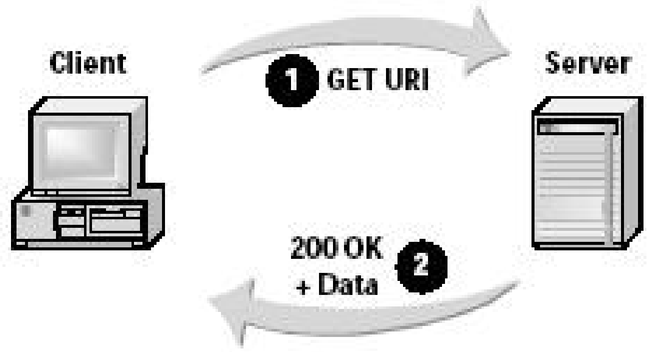
- GET
- POST
- HEAD
  - asks server to leave requested object out of response

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
- TRACE
  - invoke a remote, application-layer loop- back of the request message

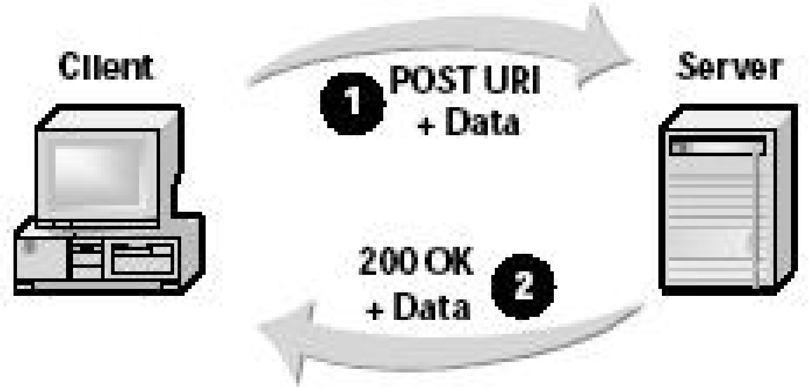
HTTP Request Message

- GET
  - Input in the URL field. No entity body
  - Sure and idempotent (the side-effects of N > 0 identical requests is the same as for a single request)
  - Only ASCII characters
  - Browser and proxy can cache the responses of the server



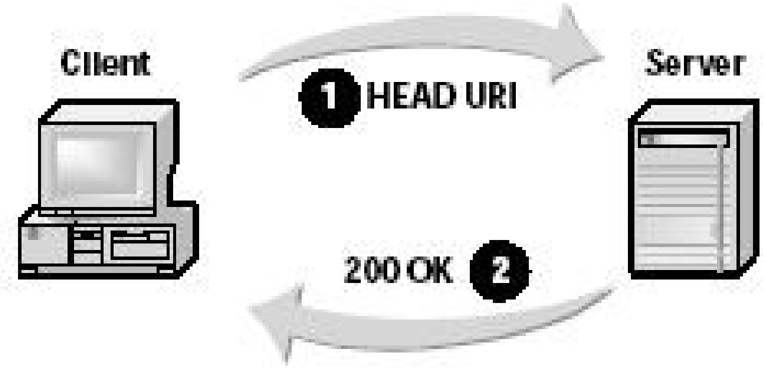
HTTP Request Message

- POST
  - The URI specifies the script which should process the information
  - The entity body contains what the user entered into the form fields
  - The script is called at each form submission (no cache)



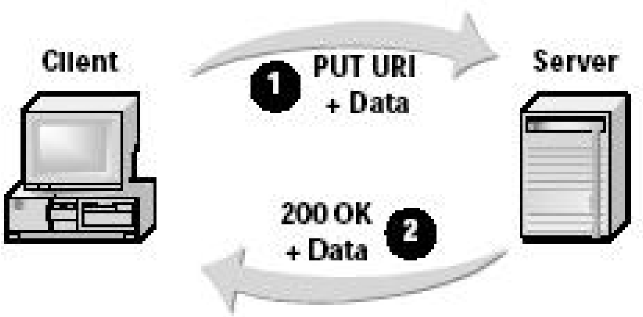
HTTP Request Message

- HEAD
  - Similar to the GET method
  - When a server receives a request with the HEAD method, it responds with an HTTP message but it leaves out the requested object
  - Often used for debugging



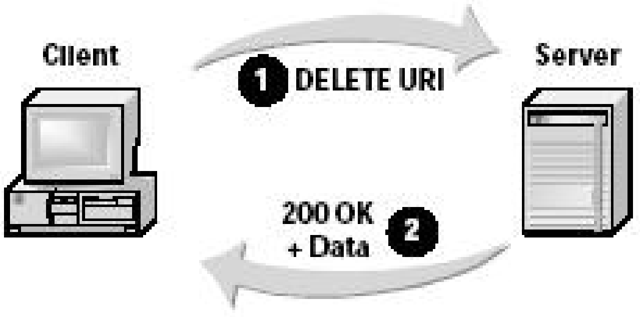
HTTP Request Message

- PUT
  - Used in conjunction with Web publishing tools
  - Allows a user to upload an object to a specific path (directory) on a specific Web server (typically the object is a file)
  - If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.
  - Used by applications which need to upload objects to Web servers



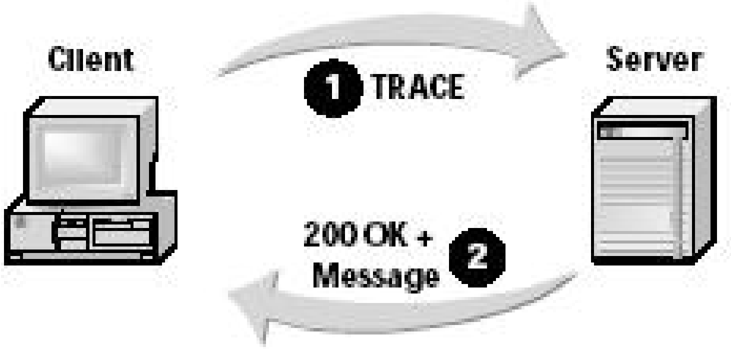
HTTP Request Message

- DELETE
  - Allows a user, or an application, to delete an object on a Web server



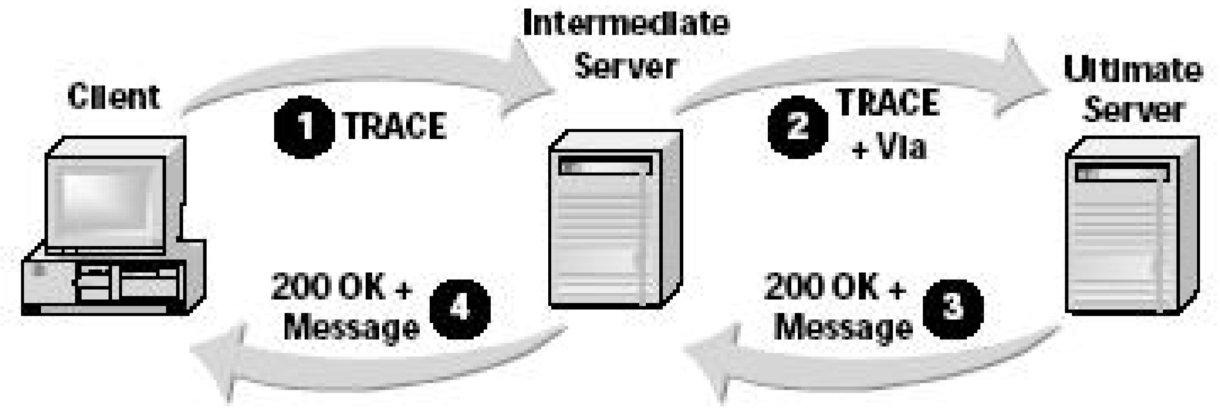
HTTP Request Message

- TRACE
  - The TRACE method is used to invoke a remote, application-layer loop- back of the request message.
  - The final recipient (origin server or proxy) of the request SHOULD reflect the message received back to the client as the entity-body of a 200 (OK) response.
  - TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information.



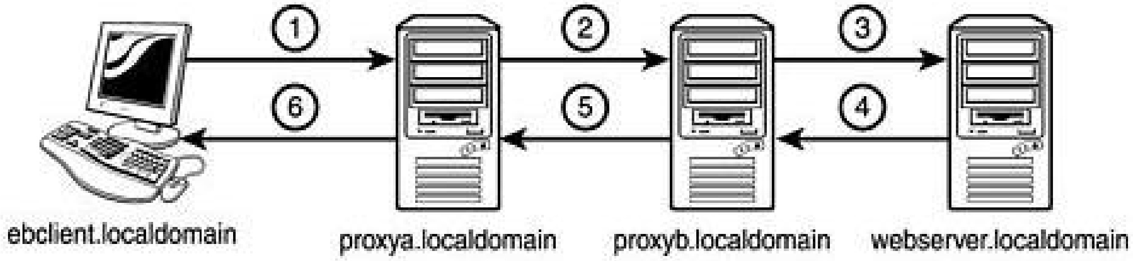
HTTP Request Message

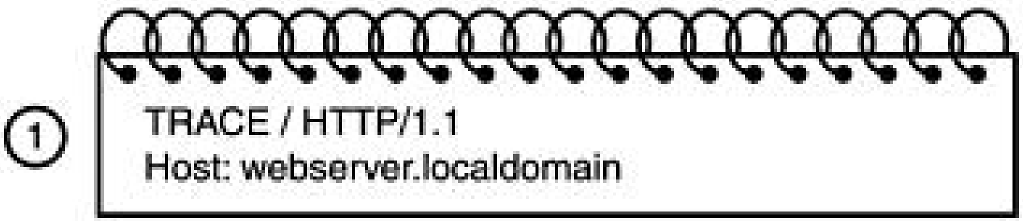
- TRACE (example)



HTTP Request Message

- TRACE (example)

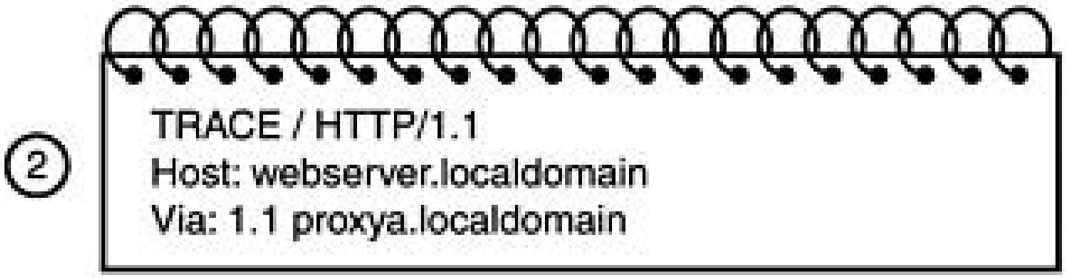




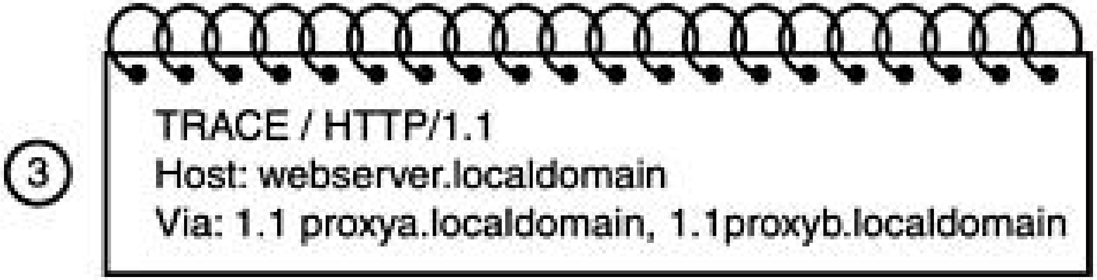
```
TRACE / HTTP/1.1
Host: webserver.localdomain
```

HTTP Request Message

- TRACE (example)



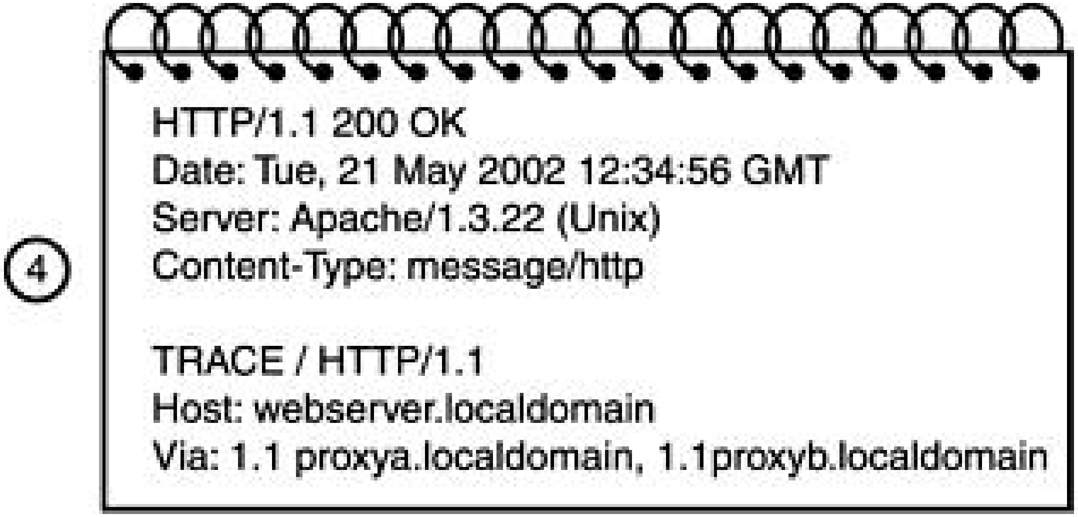
```
TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain
```



```
TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```

HTTP Request Message

- TRACE (example)

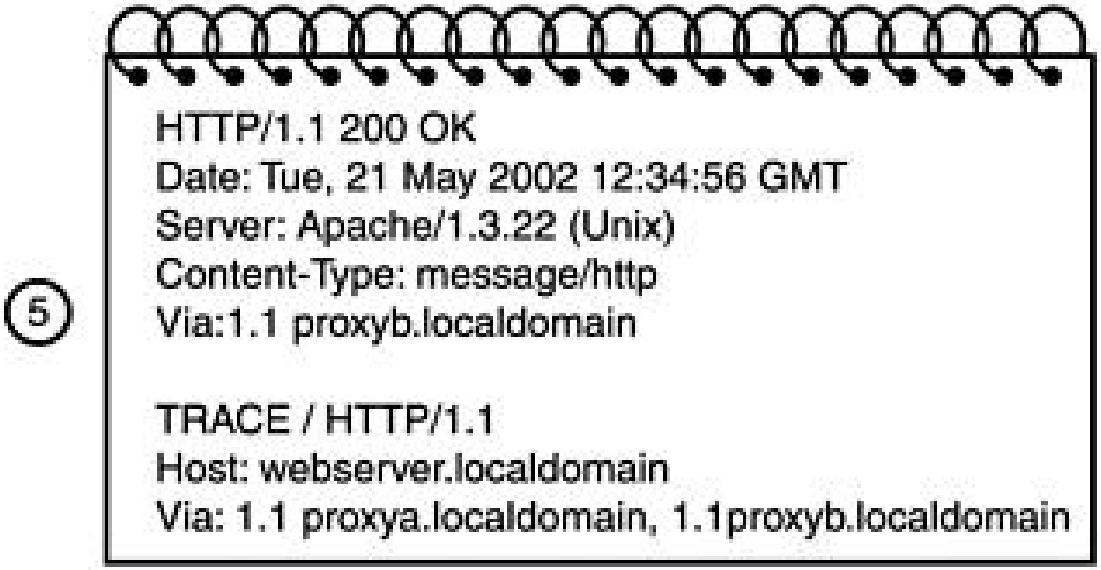


```
HTTP/1.1 200 OK
Date: Tue, 21 May 2002 12:34:56 GMT
Server: Apache/1.3.22 (Unix)
Content-Type: message/http

TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```

HTTP Request Message

- TRACE (example)

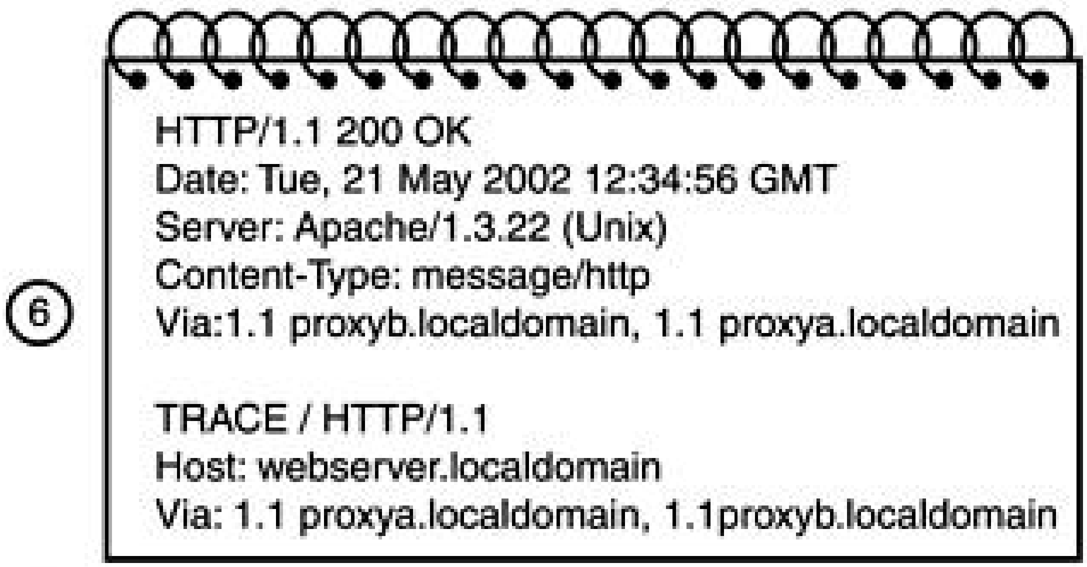


```
HTTP/1.1 200 OK
Date: Tue, 21 May 2002 12:34:56 GMT
Server: Apache/1.3.22 (Unix)
Content-Type: message/http
Via:1.1 proxyb.localdomain

TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```

HTTP Request Message

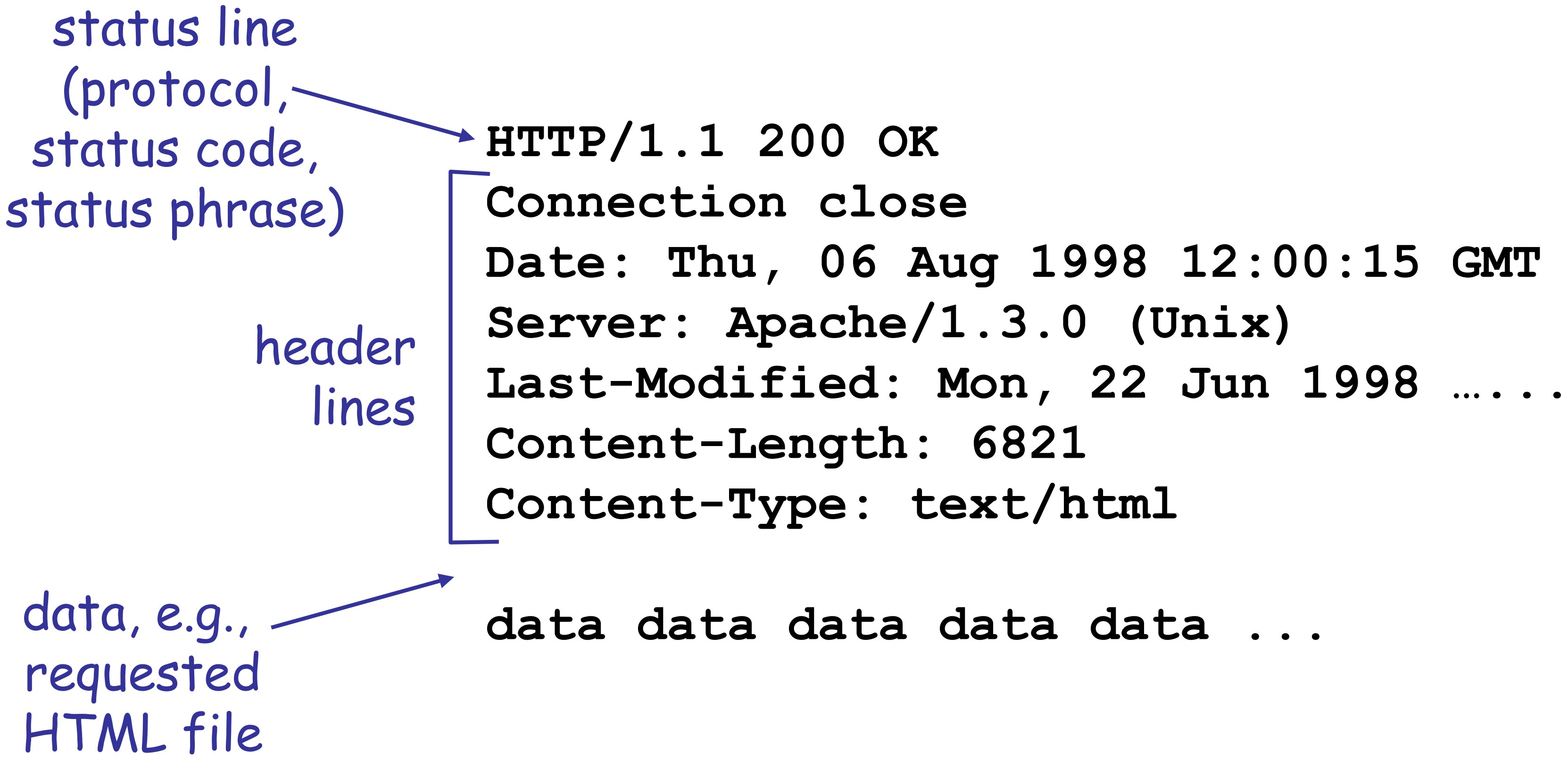
- TRACE (example)



```
HTTP/1.1 200 OK
Date: Tue, 21 May 2002 12:34:56 GMT
Server: Apache/1.3.22 (Unix)
Content-Type: message/http
Via:1.1 proxyb.localdomain, 1.1 proxya.localdomain

TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```

HTTP Response Message



status line (protocol, status code, status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

HTTP Response Status Codes

In first line in server->client response message.

A few 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:)

`400 Bad Request`

- request message not understood by server

`404 Not Found`

- requested document not found on this server

`505 HTTP Version Not Supported`

HTTP Response Message Header Lines

- Date: indicates the time and the date when the HTTP response was created and sent by the server
- Content-Type: indicates that the object in the entity body is HTML text (The object type is officially indicated by the Content-Type: header and not by the file extension)

Trying HTTP

1. Telnet to your favorite Web server:

```
telnet localhost 80
```

Opens TCP connection to port 80 (default HTTP server port) at localhost. Anything typed in sent to port 80 at localhost

2. Type in a GET HTTP request:

```
GET /form.html HTTP/1.1
Host: localhost
```

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!

Trying HTTP (get.php)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
  <meta http-equiv="Content-type" content="text/html;
  charset=ISO-8859-1">
    <title>send data using the GET method without forms and
  opening a new page</title>
            </head>
            <body>
```

Trying HTTP

```
/* Redirect to a different page in the current directory that was requested */
<?php
        $host  = $_SERVER['HTTP_HOST'];
        $uri   = rtrim(dirname($_SERVER['PHP_SELF']), '/\\');
        //rtrim — Strip whitespace (or other characters) from the end of a string
        $extra = 'receiveGET.php?name=Bob&age=21';
        header("Location: http://$host$uri/$extra");
        //Send a raw HTTP header
        exit;
        ?>
        </body>
</html>
```

Trying HTTP (receiveGet.php)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1">
    <title>Receive data and forward to the client</title>
  </head>
  <body>
    <p> Hi <?php echo htmlspecialchars($_GET['name']); ?>,<br>
       You are <?php echo (int)$_GET['age']; ?> old.
    </p>
  </body>
</html>
```

htmlspecialchars — Convert special characters to HTML entities

Trying HTTP (post.php)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
 <head>
 <meta http-equiv="Content-type" content="text/html;
 charset=ISO-8859-1">
  <title>send data using the POST method without forms and
 opening a new page</title>
   </head>
        <body>
        <p>
```

Trying HTTP (post.php)

```
<?php
  $data =
  sendPost('/receivePOST.php','name=Bob&age=21','localhost');
  echo $data;

  function sendPost($uri,$postdata,$host){
          $flow = fsockopen($host, 80, $errno, $errstr);
          //Initiates a socket connection to the resource specified by
  hostname.
          if (!$flow) {
                  echo "$errstr ($errno)<br>\n";
                  echo $flow;      }
          else {
```

Trying HTTP (post.php)

```
$requestHTTP ="POST $uri  HTTP/1.1\r\n";
$requestHTTP.="Host: $host\r\n";
$requestHTTP.="User-Agent: PHP Script\r\n";
$requestHTTP.= "Content-Type: application/x-www-form-urlencoded\r\n";
$requestHTTP.="Content-Length: ".strlen($postdata)."\r\n";
$requestHTTP.="Connection: close\r\n\r\n";
$requestHTTP.=$postdata;
$replyHTTP = NULL;
fwrite($flow, $requestHTTP);
while (!feof($flow))
    $replyHTTP.=fgets($flow, 128);
$replyHTTP = explode("\r\n\r\n",$replyHTTP);
$headerReplyHTTP = $replyHTTP[0];
$contentsReplyHTTP     = $replyHTTP[1];
```

Trying HTTP (post.php)

```
if(!(strpos($headerReplyHTTP,"Transfer-Encoding:
  chunked")===false))
{  $temp=explode("\r\n",$contentsReplyHTTP);
  for($i=0;$i<count($temp);$i++)
  if($i==0 || ($i%2==0))
     $temp[$i]="";
  $contentsReplyHTTP=implode("",$temp);
  //implode — Join array elements with a string
   }
   fclose($flow);
return rtrim($contentsReplyHTTP);
   }
 } ?>
</p>
</body>
</html>
```

Trying HTTP (receivePost.php)

Example of chunked transfer-encoding

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked
25\r\n
This is the data in the first chunk\r\n
1C\r\n
and this is the second one\r\n
3\r\n
Con\r\n
8\r\n
Sequence\r\n
0\r\n
```

Trying HTTP (receivePost.php)

```
Hi <?php echo htmlspecialchars($_POST['name']); ?>,
  <br> You are <?php echo (int)$_POST['age']; ?> old
```

Header Lines

- How does a browser decide which header lines to include in a request message?
  - Browser type and version
  - User configuration of the browser
  - If the browser has a cached, but possibly out-of-date version of the object
- How does a Web server decide which header lines to include in a response message?
  - Server type and version
  - Server configuration

Header Lines

- HTTP/1.0 defines 16 different header lines (no mandatory)
- HTTP/1.1 defines 51 different header lines (only one Host is mandatory) in the request messages to avoid that the server replies 400 Bad Request.

Authorization and Cookies

- HTTP server is stateless
  - Simplifies server design
  - High performance Web servers that can handle thousands of simultaneous TCP connections
- Sometimes it is desirable for a Web site to identify users
  - To restrict user access
  - To serve a content as a function of the user identity

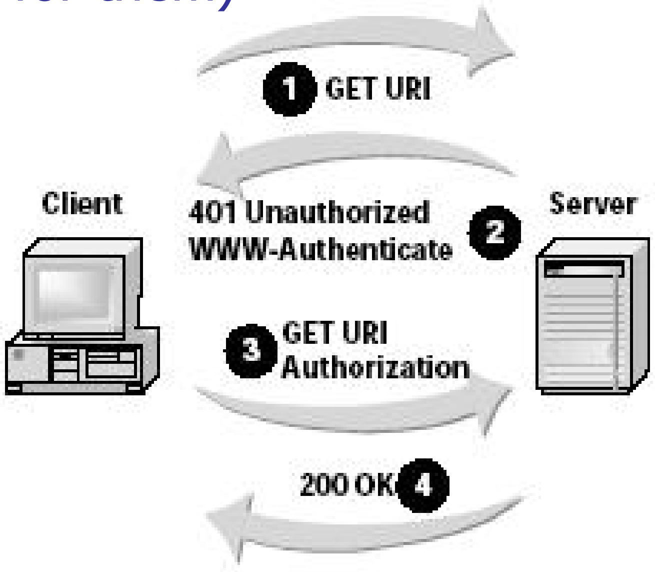
Authorization

- Requesting and receiving authorizations is often done by using special HTTP headers and status codes
- Scenario
  1. Client requests an object from a server and the server requires an authorization
  2. Client sends an ordinary request message with no special header lines
  3. The server responds with an empty entity body and 401 Authorization Required status code. In the response message the server includes the WWW-Authenticate: header

Authorization

4. The client prompts the user for a username and passwd
5. The client resends the request message including an Authorization: header line
6. After obtaining the first object, the client continues to send username and password in subsequent requests for objects on the server (username and passwords are cached and therefore the user is not prompted for them)

- Note: this is a very weak form of authorization
  - One can sniff (read and store) all the packets and therefore steal the login password.



Cookies

Many major Web sites use cookies

Four components:

1) cookie header line of HTTP *response* message

2) cookie header line in HTTP *request* message

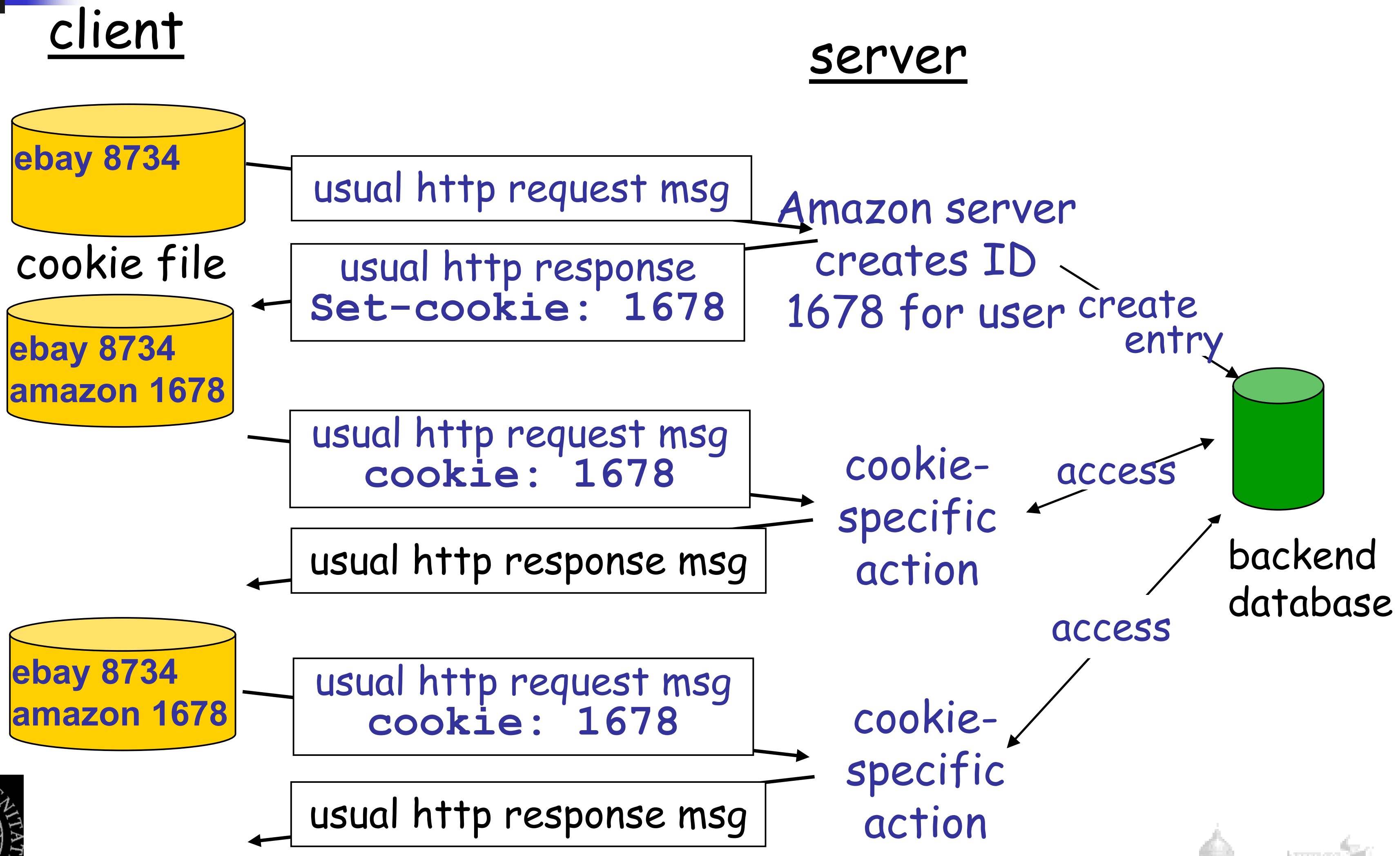
3) cookie file kept on user's host, managed by user's browser

4) back-end database at Web site

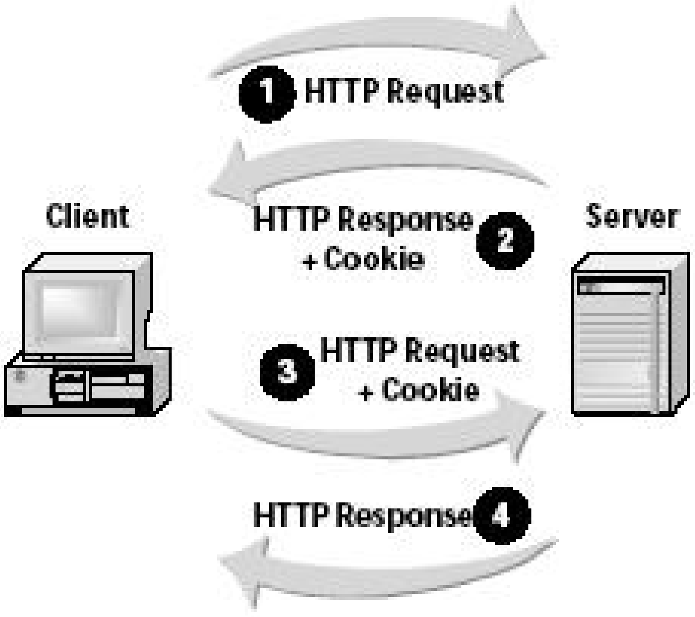
Example:

- Susan always access Internet always 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


client
server
ebay 8734
cookie file
usual http request msg
Amazon server
creates ID
1678 for user
create
entry
usual http response
Set-cookie: 1678
ebay 8734
amazon 1678
usual http request msg
cookie: 1678
cookie-
specific
action
access
usual http response msg
backend
database
access
ebay 8734
amazon 1678
usual http request msg
cookie: 1678
cookie-
specific
action
usual http response msg

Cookies


1 HTTP Request
Client
HTTP Response + Cookie 2
Server
3 HTTP Request + Cookie
HTTP Response 4

Cookies

aside

What cookies can bring:

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

Cookies and privacy:

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

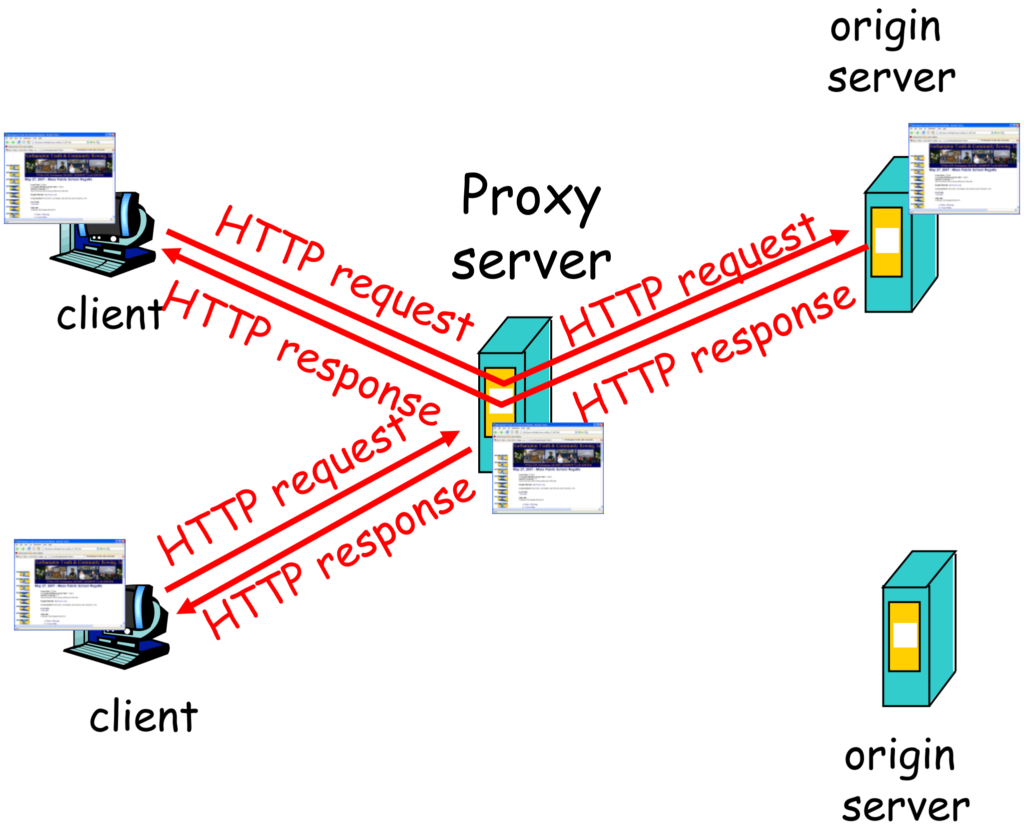
How to keep "state":

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

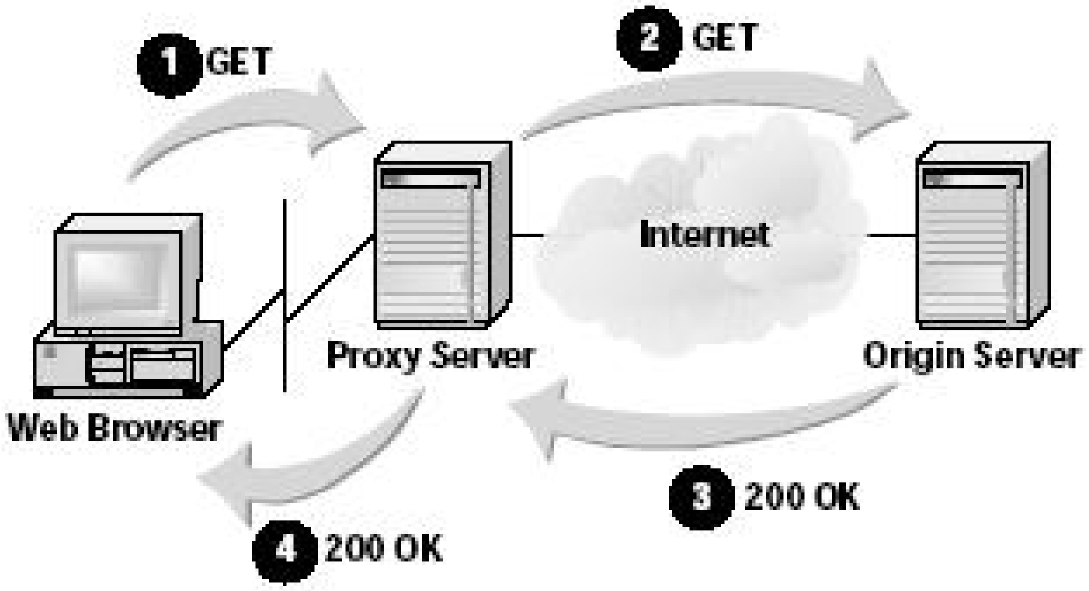
Web caching

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



More about Web caching

- cache acts as both client and server
- typically cache is installed by the Internet Service Providers (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 (but so does P2P file sharing)

Caching Example

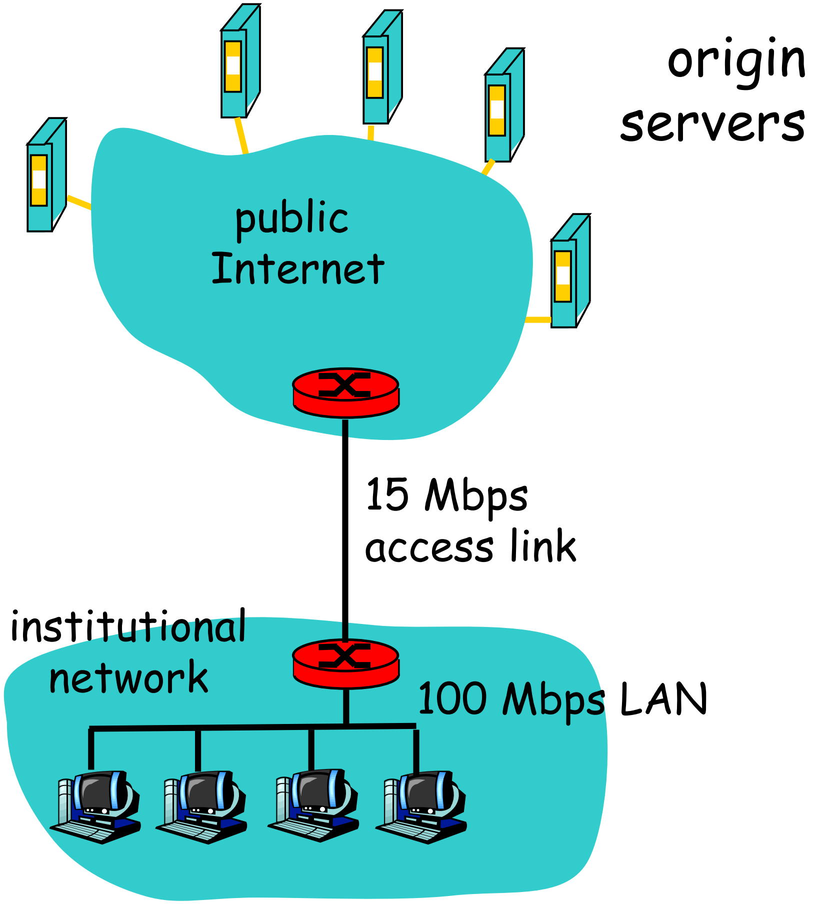
Assumptions

- average object size = 1,000,000 bits
- avg. request rate from institution's browsers to origin servers = 15/sec
- delay from institutional router to any origin server and back to router (Internet delay) = 2 sec

Consequences

- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay

= 2 sec + minutes + milliseconds



institutional cache

Caching Example

Some Computation

Lan Traffic Intensity

(15 requests/sec) * (1Mb/request)/100Mbps = 0.15

An intensity of 0.15 results in, at most, tens of milliseconds of delay

Access Link Intensity

(15 requests/sec) * (1Mb/request)/15Mbps = 1

An intensity of 1 results in a very large delay of the order of minutes

Caching Example (continued)

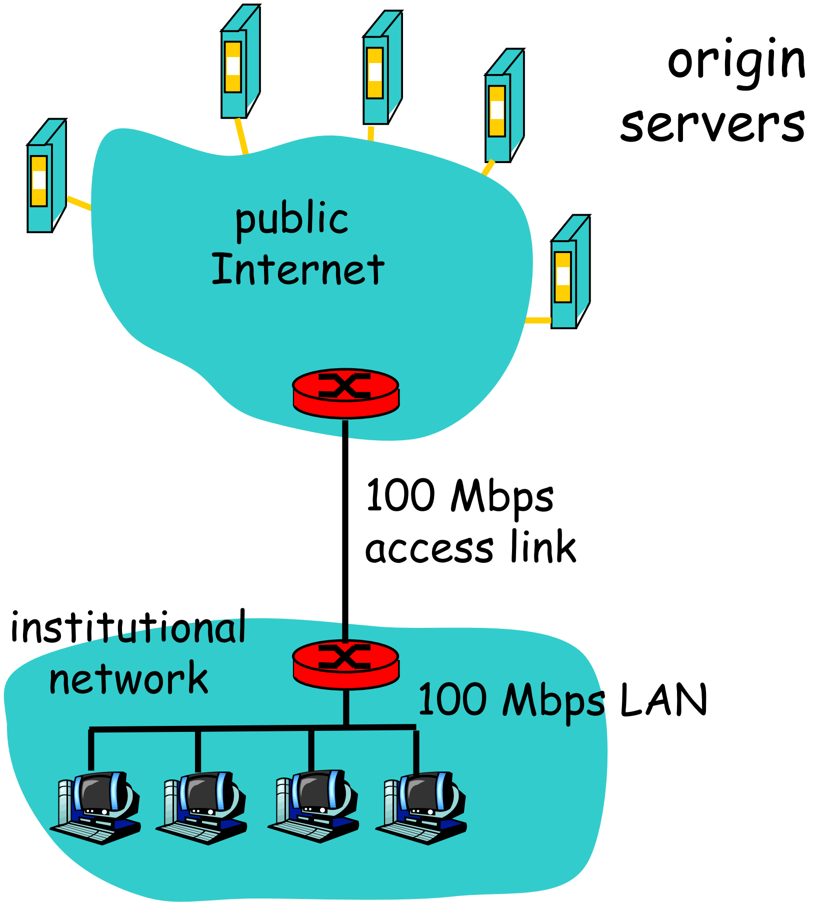
possible solution

- increase bandwidth of access link to, say, 100 Mbps

consequence

- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay

  = 2 sec + msecs + msecs
- often a costly upgrade



institutional cache

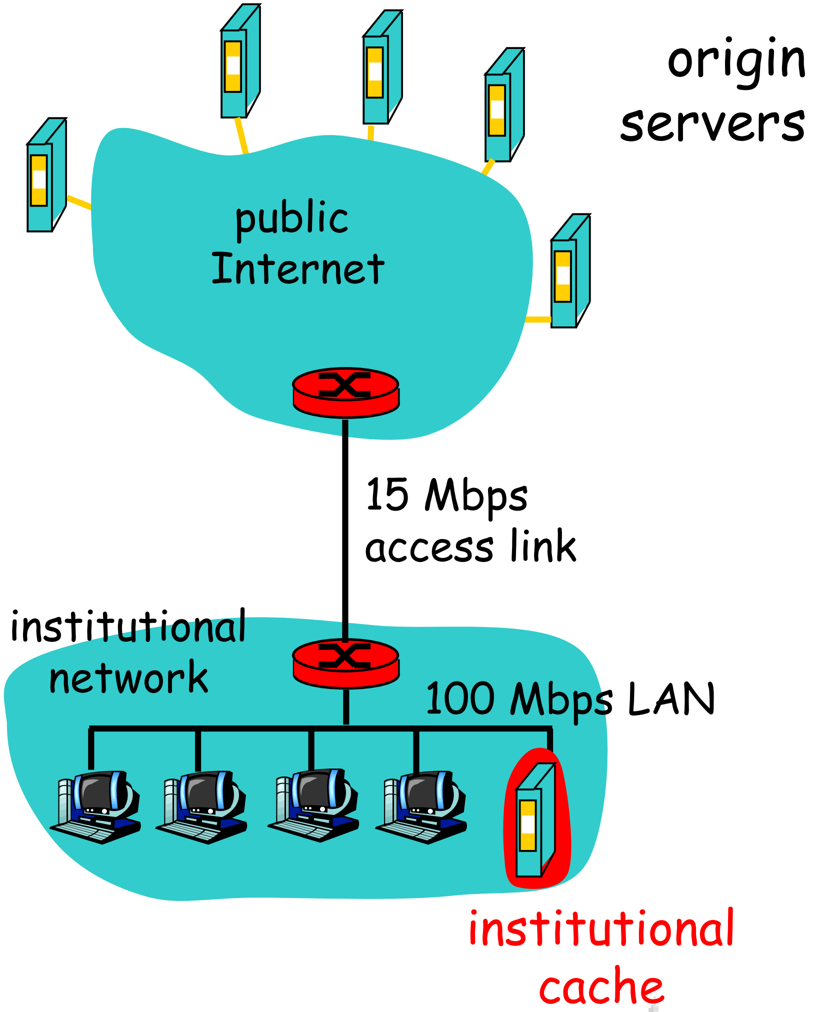
Caching Example (continued)

possible solution: install cache

- suppose hit rate is 0.4

consequence

- 40% requests will be satisfied almost immediately (order of milliseconds)
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
- total avg delay = Internet delay + access delay + LAN delay = .6*(2.01) secs + .4*milliseconds < 1.4 secs

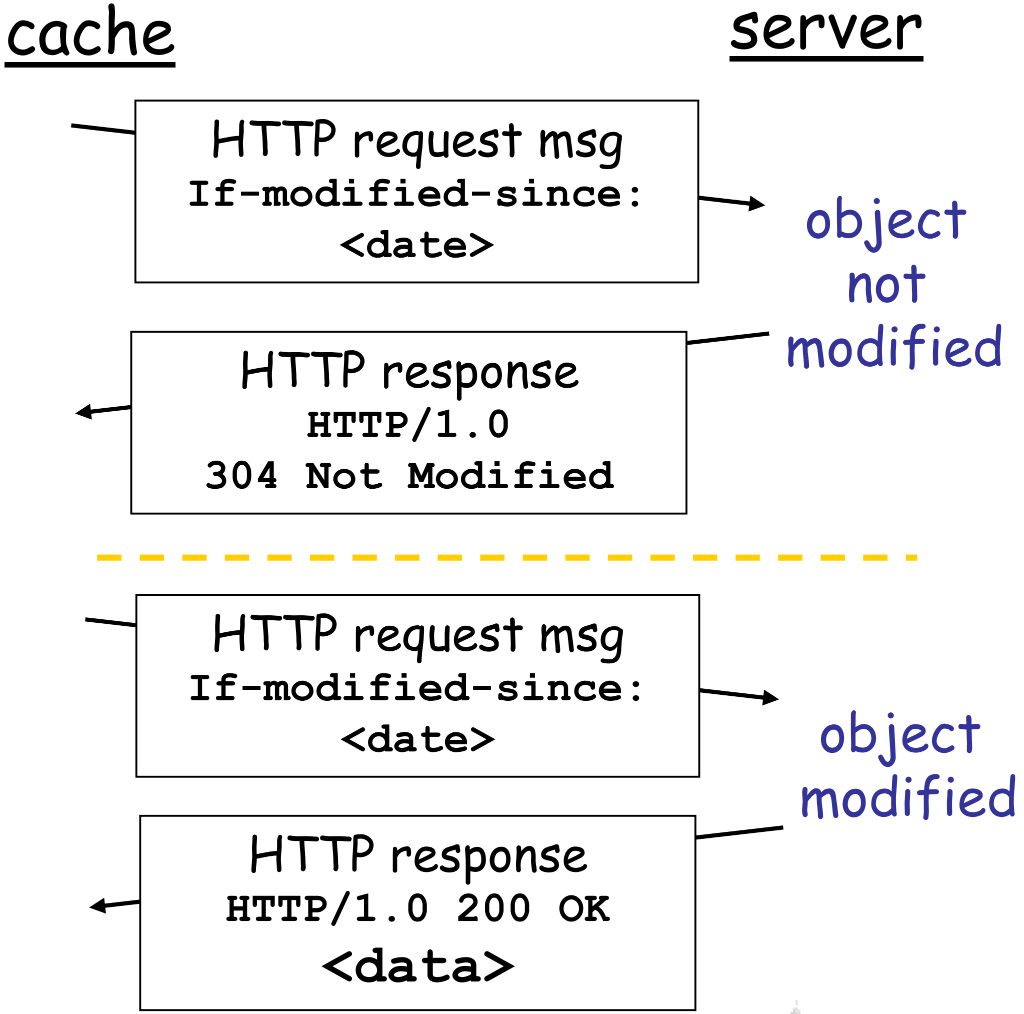


Problem

- Web caches can reside in a client (managed by the user’s browser) or in an intermediate network cache server
- Problem: a copy of an object residing in the cache may be *stale* (object housed in the Web server may have been modified since the copy was cached at the client)
- Solution: the conditional GET
  - The request message uses the GET method
  - The request message includes an If-Modified-Since: header line

Conditional Cache

- Goal: don't send object if cache has up-to-date cached version
- 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`



cache

server

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

object not modified

HTTP response
`HTTP/1.0 304 Not Modified`

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

object modified

HTTP response
`HTTP/1.0 200 OK`
`<data>`