# CSCC09F
# Programming on the Web

## Protocols and HTTP,
## Web Browsers and Servers,
## Web Caching

first some background on network-based applications …

# Client-Server Paradigm

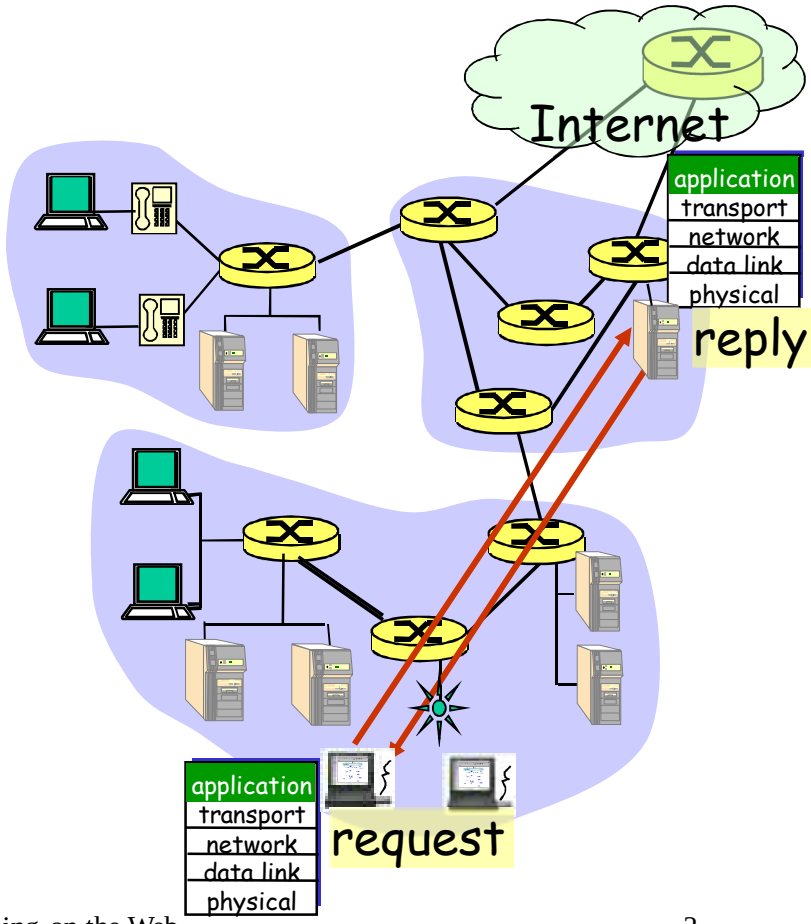
Client:

- initiates contact with server ("speaks first")
- typically requests service from server
- for Web, client is implemented in browser; for e-mail, in mail user agent

Server:

- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail

Typical network app has two pieces: client and server



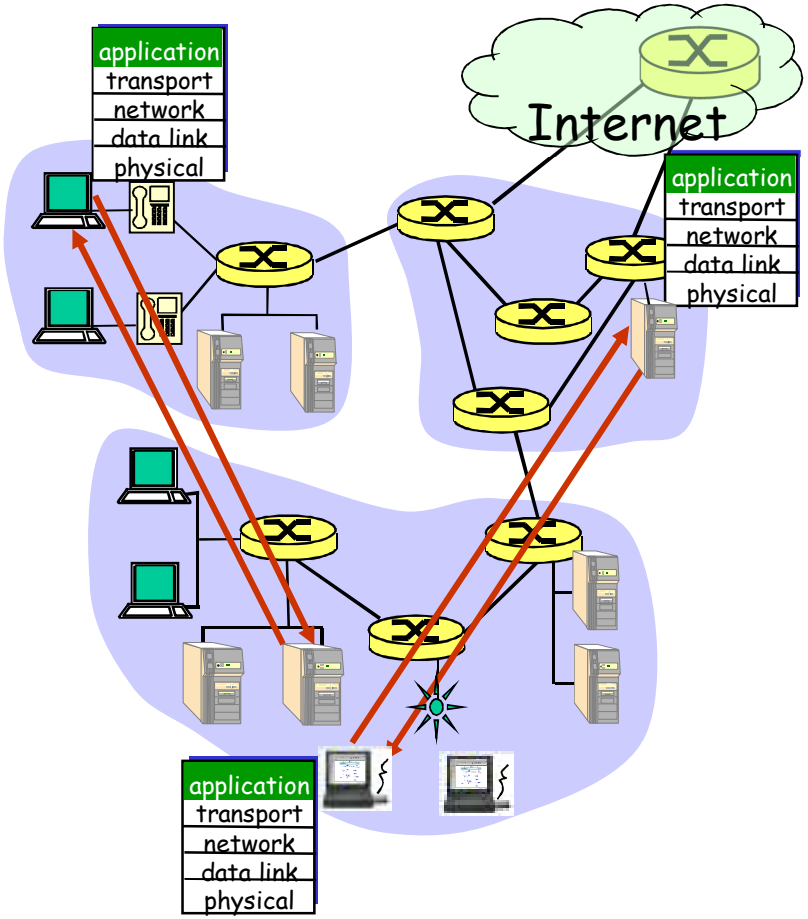
# Applications and application-layer protocols

Application: communicating, distributed processes

- running in network hosts in "user space"
- exchange messages to implement application
- e.g., email, file transfer, the Web, network news

Application-layer protocols

- one "piece" of an app
- define messages exchanged by apps and actions taken
- uses services provided by lower layer protocols



# HyperText Transfer Protocol (HTTP)

Part of Tim Berners-Lee's innovation

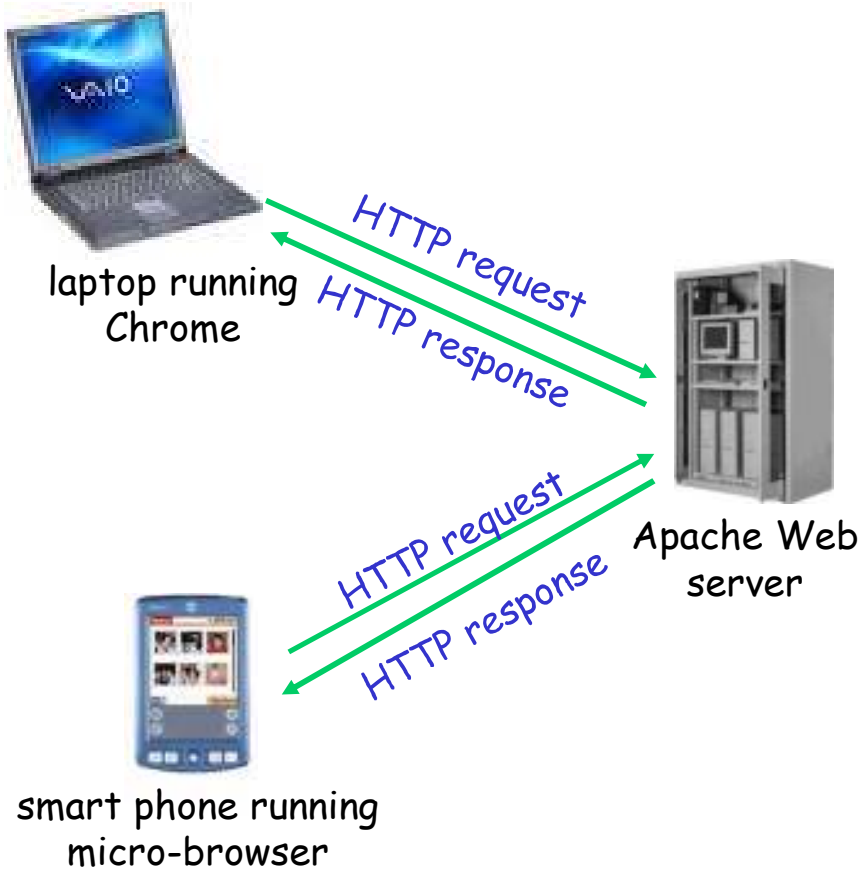
- Share documents across the network
- Defined 1989-1991

- Standardized and much expanded by the IETF
- Rides on top of "TCP" transport protocol
  - TCP provides: reliable, bi-directional, in-order byte stream (exists to compensate for unreliable Internet Protocol IP)
- Goal: transfer objects between systems
  - Do not confuse with other WWW concepts:
    - HTTP is not a page-layout language (that is HTML)
    - HTTP is not an object-naming scheme (that is URLs)

# The Web: HTTP protocol

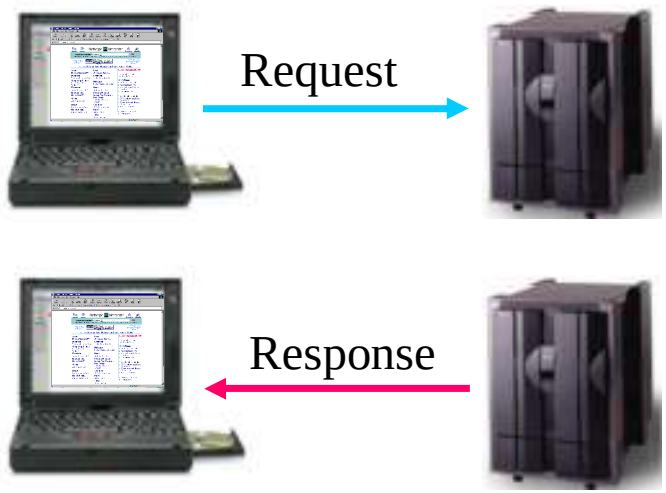
## HTTP: HyperText Transfer Protocol

- Web's application-layer protocol
- client/server model
  - client: (active) browser that requests, receives, "displays" Web objects
  - server: (passive) Web server sends objects in response to requests
- http1.0: RFC 1945
- http1.1: RFC 2068



# HTTP 1.0

- Interaction between Web client (browser) and Web server occurs in two phases:
  - Request phase:
    Browser requests page from Web server
  - Response phase:
    Server sends back requested page or code




- Protocol for each phase consists of:
  - Header (2 parts: request line or response status (single line), header fields (possibly multiple lines)
  - Body (request or response)

# Try out HTTP (client side) for yourself

1. Telnet to your favorite Web server, e.g.:

```
telnet www.utsc.utoronto.ca 80
```

Open TCP connection to port 80 (default http server port) at **www.utsc.utoronto.ca**. Anything typed is sent to port 80 at www.utsc.utoronto.ca

2. Type in a GET HTTP request:

```
GET /~rosselet/cscc09/ HTTP/1.0
```

Type this at the prompt (followed by two carriage returns) to send this minimal GET request to the HTTP server

3. Observe response message sent by http server

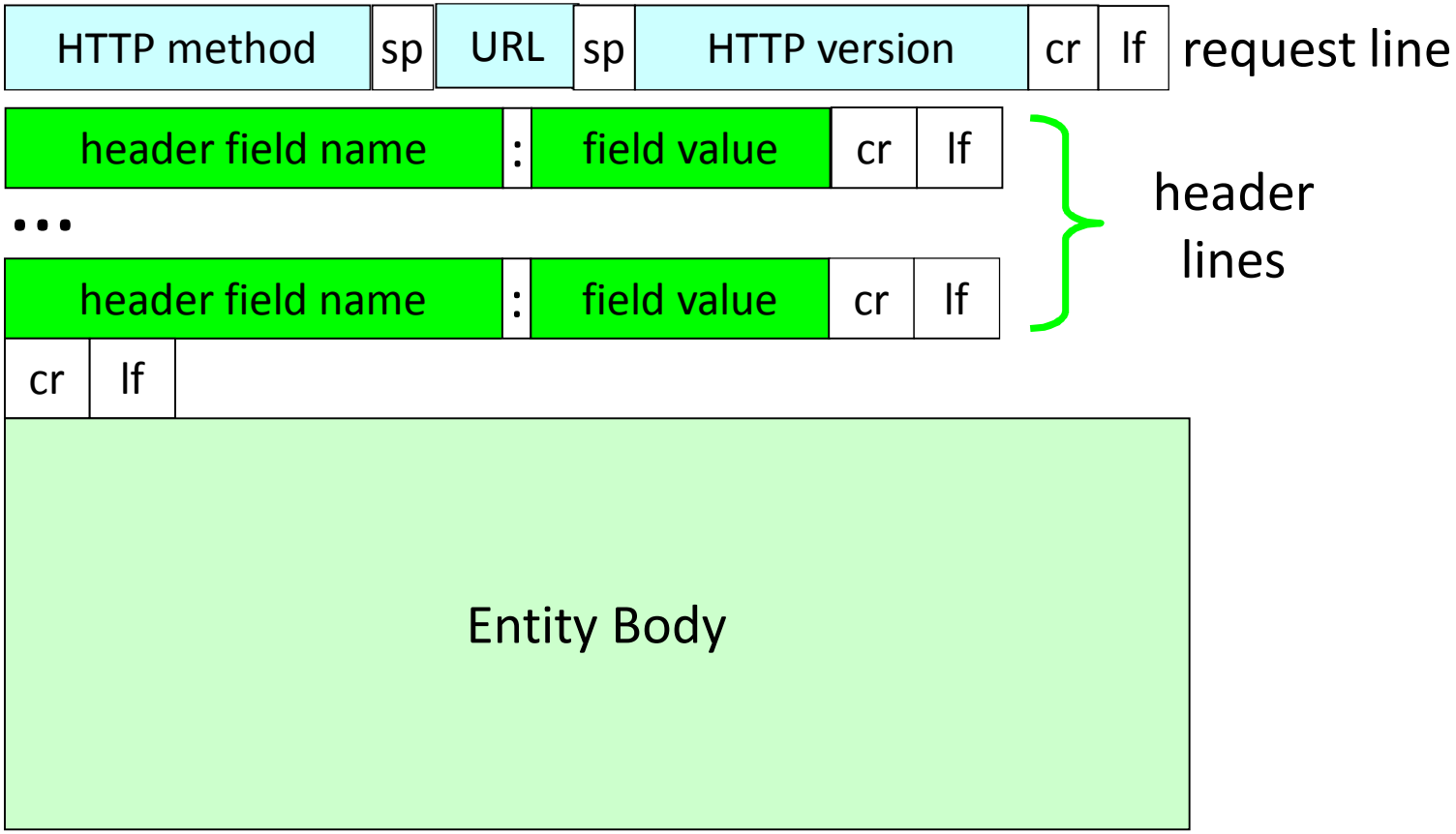
# Example: command line HTTP request

```
mathlab:~> telnet www.tdcanadatrust.com 80
Trying 184.86.36.40...
Connected to e9726.b.akamaiedge.net.
Escape character is '^]'.
GET /index.html HTTP/1.0
Host: www.tdcanadatrust.com

HTTP/1.0 301 Moved Permanently
Location: http://www.tdcanadatrust.com/products-
   services/banking/index-banking.jsp
Content-Length: 280
Content-Type: text/html; charset=iso-8859-1
Date: Wed, 22 Oct 2014 12:03:45 GMT
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head><title>301 Moved Permanently</title></head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a
   href="http://www.tdcanadatrust.com/products-
   services/banking/index-banking.jsp">here</a>.</p>
</body></html>
Connection closed by foreign host.
```

# HTTP request message format



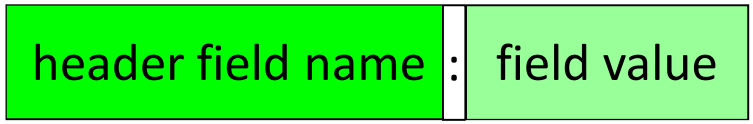
# HTTP 1.0 Request Phase

- Request line


  - HTTP method
    - GET – return content of specified document
    - HEAD – return only headers of GET response
    - POST – execute specified doc with enclosed data
  - URL
    - Usually expressed as: /path (different for HTTP/1.1)
      - e.g. /index.html
  - HTTP version
    - e.g. HTTP/1.0

# HTTP 1.0 Request Phase (cont)

- Header fields, sequence of:




- Examples:
  - **`Accept: text/html`**
  - **`Accept: image/jpg`**
  - **`Accept-language: en; fr`**
  - **`If-modified-since: 20 Oct 2014`**
  - **`Content-Length: 7814`**

# HTTP request message example

- http request message:
  - Note all HTTP messages are in ASCII (text format)

```
GET /appdir/page.html HTTP/1.1
User-agent: Chrome/37.0.2062.124
Accept: text/html,image/gif,image/jpeg
Accept-language:fr

data data data data data ...
data data data data data ...
data data data data data ...
```

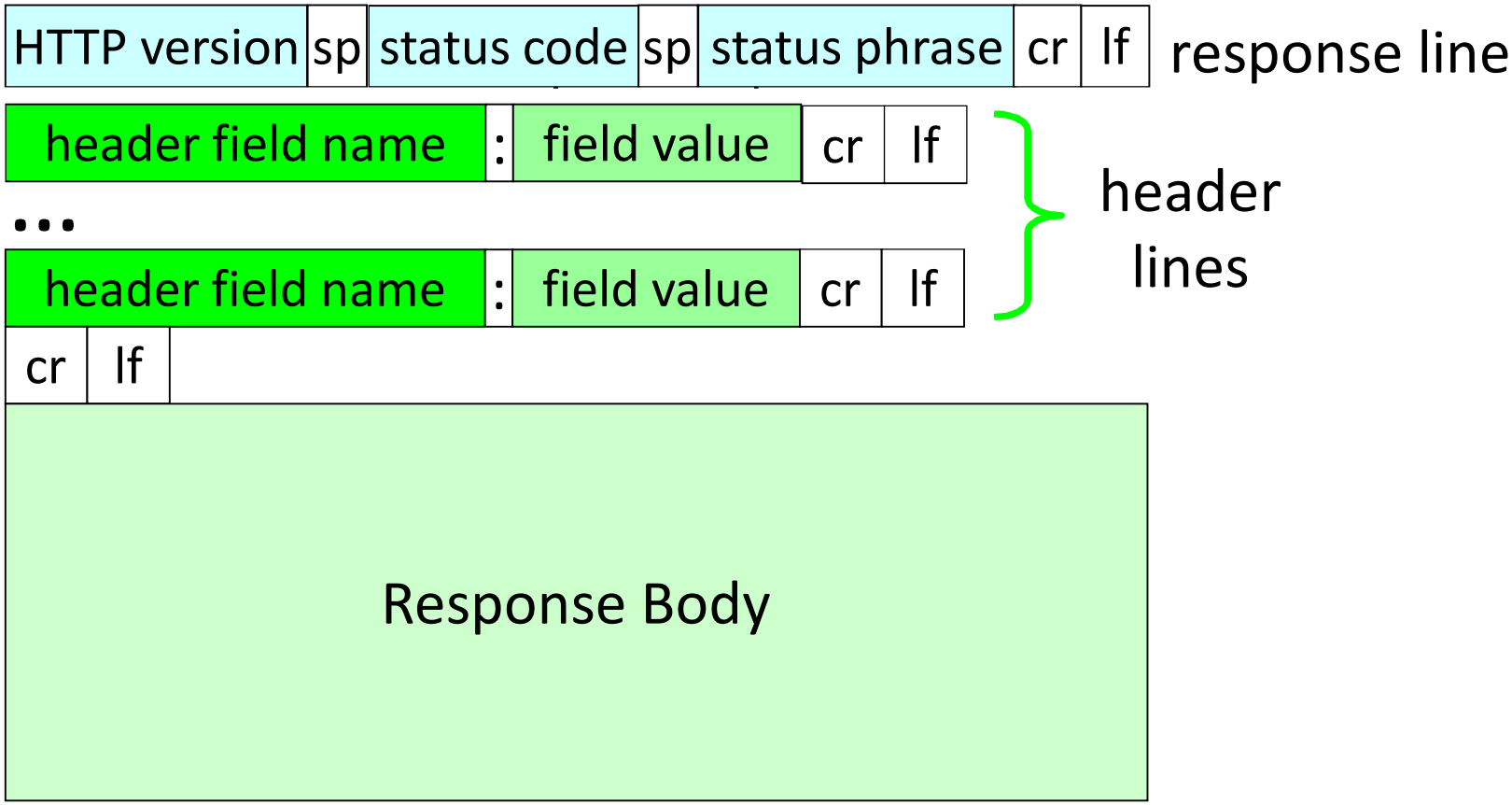
request line (GET, POST, HEAD, etc., commands)

header lines

blank line indicates end of header

Post method data, e.g., form field values

# HTTP response message format



# HTTP 1.0 Response Phase

- Response consists of
  - Status line:
    - HTTP version
    - 3-digit status code (success, error, redirection, etc)
    - Brief text explanation of status code (e.g. OK)
  - Response Header fields:
    - Other page attributes (content type, content length, expiration, last modified, server type, etc)
    - Additional information (if redirection, other location)
  - empty line (delimiter between header and body)
  - Response body (e.g. html for requested page)

# Response Status Codes


- 3 digit response code
  - 1XX – informational
  - 2XX – success (return requested document)
  - 3XX – redirection (send request elsewhere)
  - 4XX – client error (e.g. document not found)
  - 5XX – server error (e.g. Tomcat setup problem)
- status phrase – brief translation of the numeric response code (e.g. 404 – File not found)

# Response Status Code Examples

- Some commonly occurring status codes:

**200 OK**
- request succeeded, requested object appears 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 (malformed HTTP request)

**404 Not Found**
- requested document not found on this server

**501 Internal Server Error**
- something wrong on server side – e.g. CGI script error

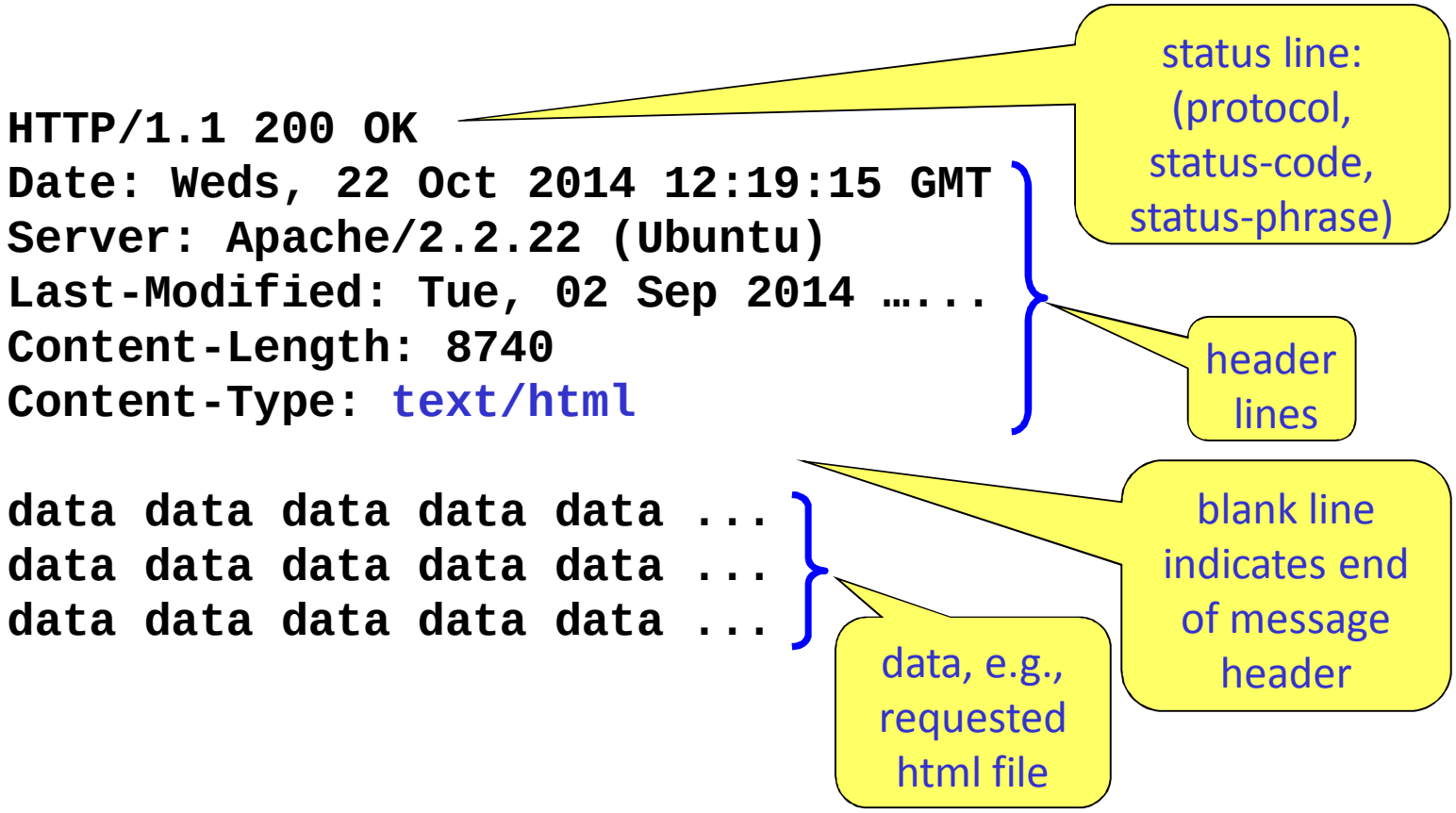
# HTTP response header

- A server may send a set of responses to a given client within a browsing session, e.g.: plain text, html, images, pdf files, etc.
- How does the browser know what to do when a message from the server shows up?
- content-type header is the key to identifying the message data-type and thus how the browser should handle it.

»

- based on the MIME type-codes originally defined to allow attachments to be bundled with e-mail (RFC 1341)

# HTTP response message example



# HTTP other features

- POST method
  - Client can send information to server, e.g. form field data, file-upload content
- Conditional GET **`if-modified-since`** request header
  - Client tells server it already has a copy of the requested data, and asks server whether this copy is stale version and thus fresh copy needs to be re-fetched from server
  - Why bother?
    - Efficiency: avoid (re)sending data if client already has up-to-date copy (cached)

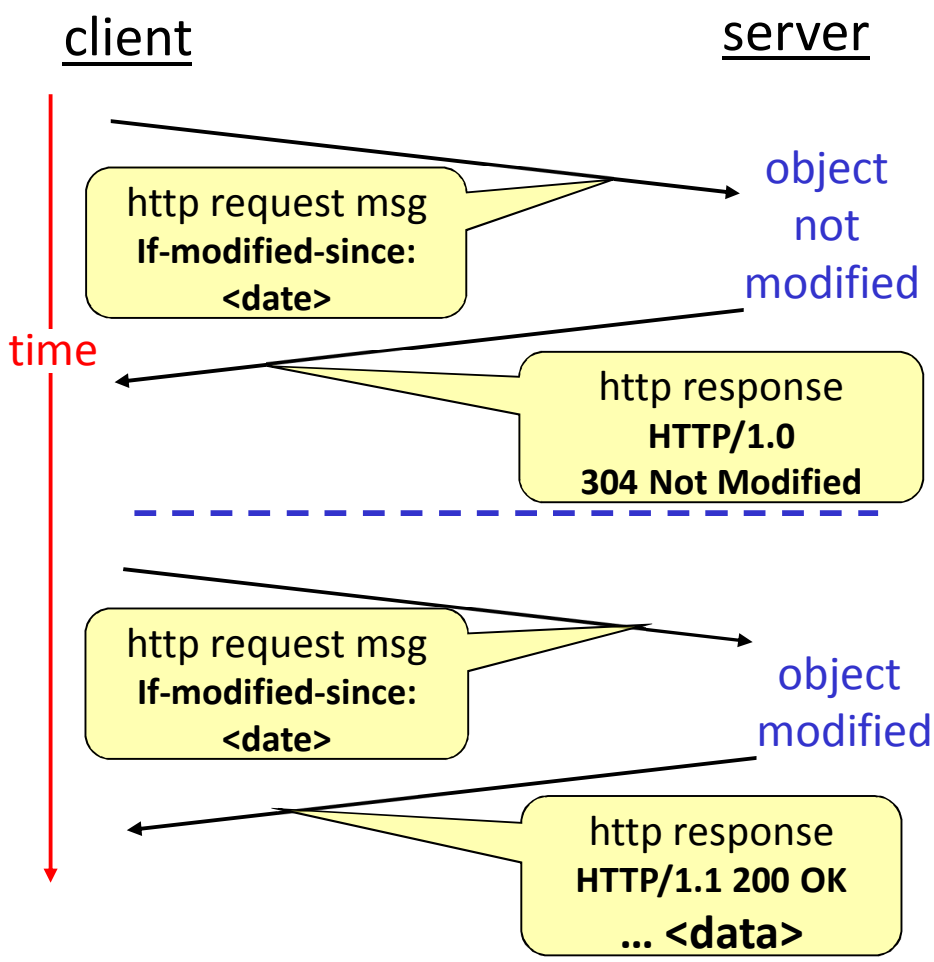
# User-server interaction: conditional GET

- client: specifies date of cached copy in request

  **`If-modified-since: <date>`**

- server: response contains no body if cached copy is up-to-date:

  **`HTTP/1.0 304 Not Modified`**



# HTTP Authentication

Example:
```
private.html
(user: cscc09f14,
 pass: topsecret)
```

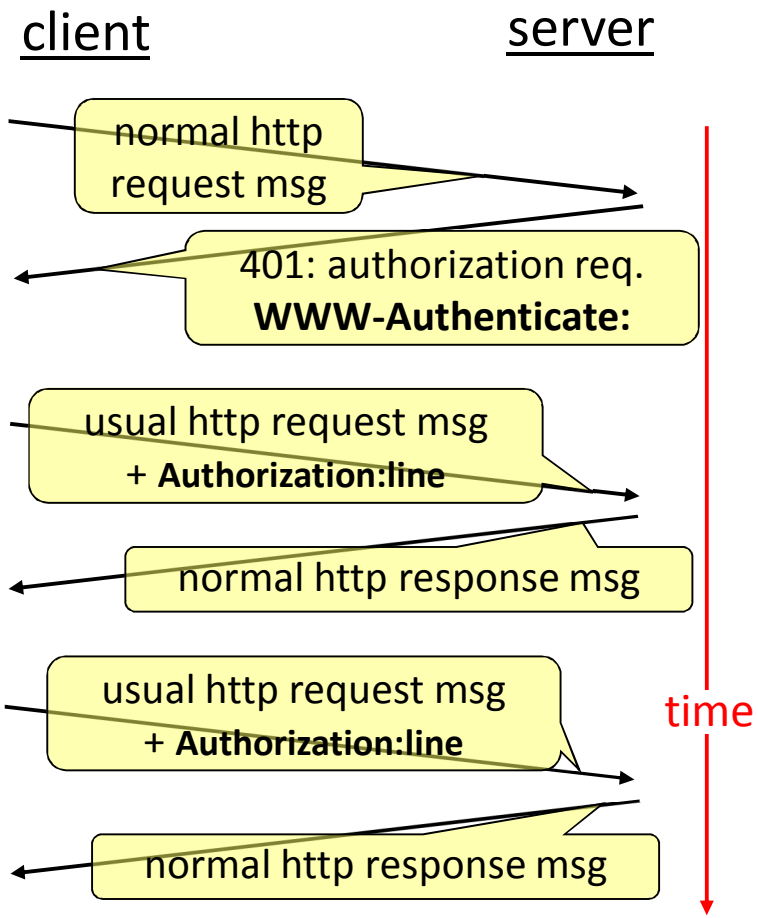
- Goal: controlled access to documents
- Basic authentication
  - When challenged, client sends user-id and password in clear to server
  - Not secure enough (snooping is easy) but useful for restricting access to low-value targets
- Protocol is stateless: client must present credentials in each request
  - but, browser caches user-id + password so user does not have to enter them for each request

# client-server interaction: authentication

- **`WWW-Authenticate`** response header: typically username, password based
  - **`Authorization:`** request header line
  - if no authorization presented, server refuses access, sends another

    **`"WWW authenticate:"`**

    header line in response
  - Set up with **`.htaccess`**, **`.htpasswd`**



# HTTP is "Stateless"

- Server maintains no information about past client requests – each one must be self-contained, why?
- Protocols that maintain "state" are complex – stateless simplifies protocol and server, e.g. server has nothing to recover after crash, smaller and simpler implementation
- Are there any implications for developers?
  - History (state) must be maintained at the application level – by apps themselves

# Stateless? I don't get it.

- OK, so HTTP is stateless, but I don't quite see why that matters to the Web app developer
  - Think about how users interact with a Web application…
  - Typically they perform a sequence of steps, e.g. selecting items, making choices, and the user expects at the end of such a sequence that the application will know what they mean when they say "check out".
  - But what is the server view of these user "steps"?
    - typically these are individual HTTP requests
  - But if the server does not maintain status information across client requests, then how will the server know what to do for "check out"?
  - Short answer, it won't. There's no way for the server to correlate multiple request from same user!

# Cookies, who needs 'em?

- So, what's the answer? How can the server keep track of something when it has no concept of request "state"
- Solution: store a small amount of information about the client state in a cookie
  - Where to store cookies?
    - on client side! so how does the Server get them?
  - client sends cookie(s) with each request
    - what happens if the client is talking to multiple servers or if there are multiple cookies?
  - cookie values may be updated by server with response(s)

# User-server interaction: cookies

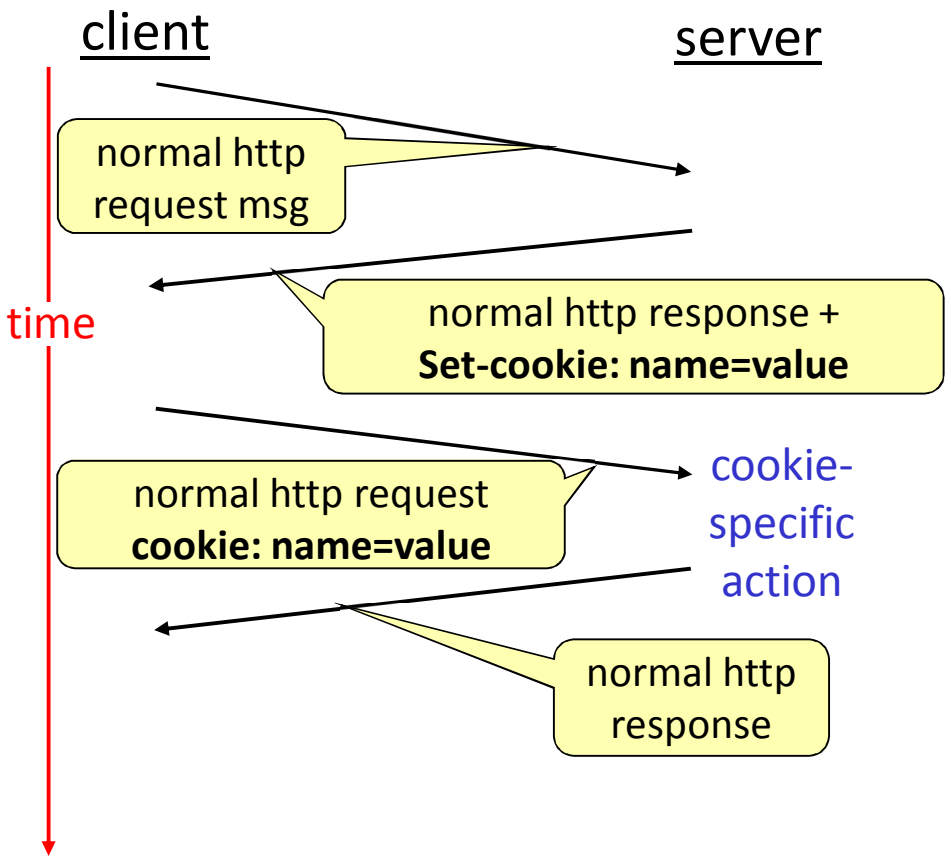
- server sends "cookie" to client in response msg

  **`Set-cookie: name=1678453`**

- client presents cookie in later requests

  **`cookie: name=1678453`**

- server may match presented-cookie with server-stored info.
- uses:
  - authentication
  - remembering user preferences, previous user selections



# Cookies, what's in em?

- How much client state can you store in a cookie?
- Is it a good idea to have client information going back and forth between client and server?
  - What if something gets lost, or changed, does that mean the state changes?
  - What if somebody else can read the cookies from the network, how is user privacy protected?
  - Although cookies can store a modest amount of data (on the order of 4KB, and with up to 20 cookies per server), generally that's not how they are used.
  - Instead, the actual data is stored on the server, e.g. in a DB, and the cookie holds just a random-looking "token" value that allows the server to locate the relevant data.
  - This approach solves both of the above concerns

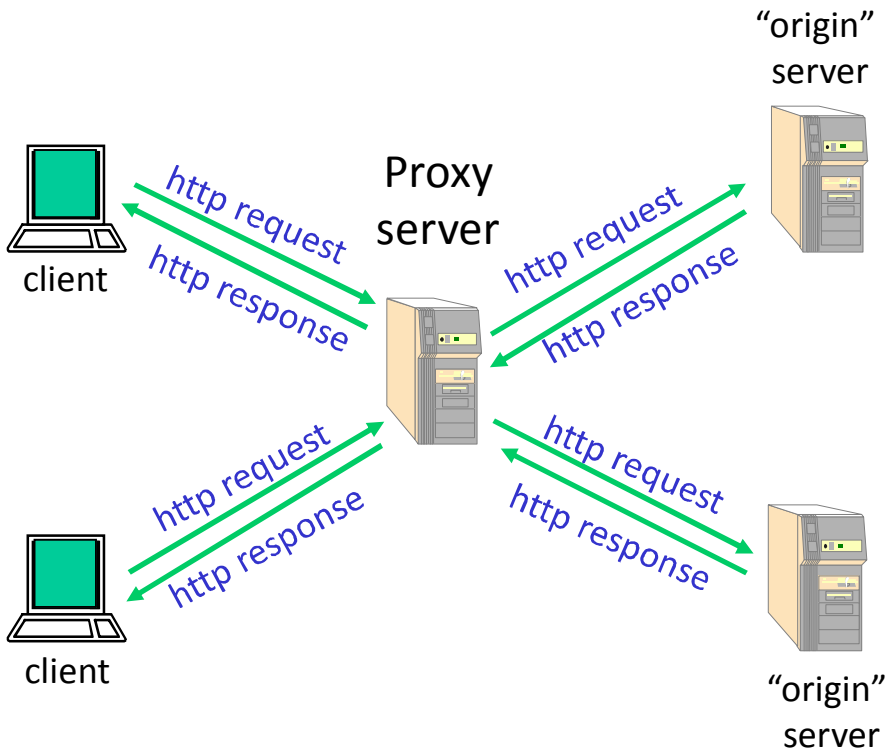
# Web Caching

- Goal: fill client (browser) request without going to originating server. Why bother?
- In early days of Web, the Internet was a lot slower – mask this by retrieving content from local copies.
- Internet faster now, but still pays to be efficient in use of "bandwidth" – network capacity.
- We've already seen one HTTP request header designed for this purpose; which one?
- Client implements its own caching mechanism.
- Important to recognize that a "Web page" may consist of multiple objects, each of which is transferred/cached separately, esp. images.

# Proxy Server Web Caches

Goal: fill client request more efficiently than sending request all the way to originating server

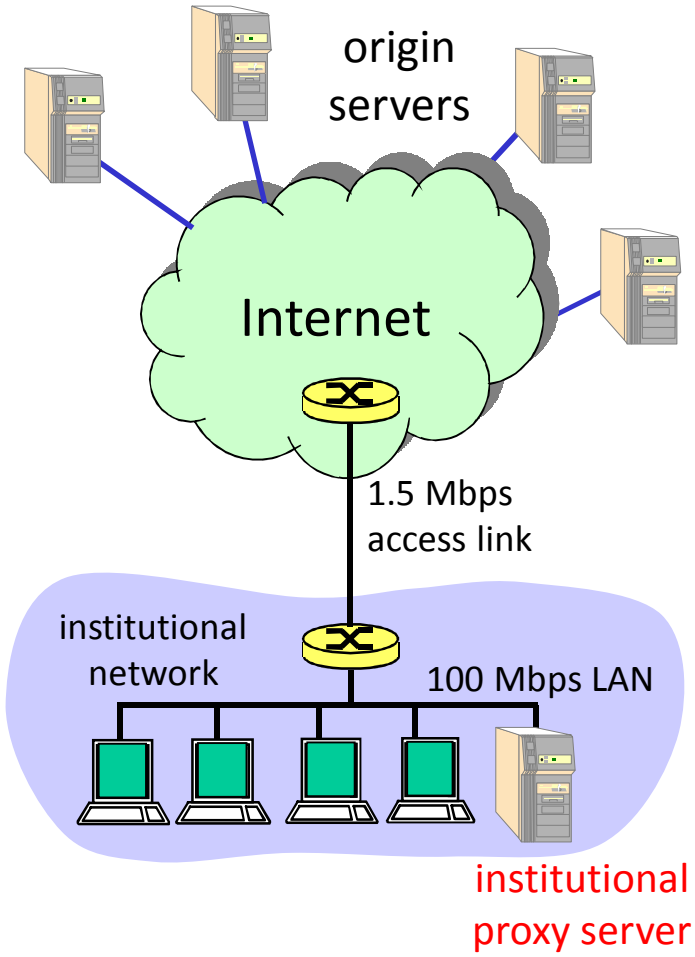
- user configures browser to use proxy-server
- client sends all HTTP requests to proxy server
  - if object exists in proxy's cache, proxy immediately returns object in http response
  - else proxy requests object from origin server, then returns http response to client and retains cached copy



# Benefits of Web Caching

Assume: cache is "close" to client (e.g., in same local area network)

- smaller response time: cache "closer" to client than origin is
- decrease traffic to distant servers
  - link out of institutional/local ISP network often bottleneck



# Caching, why?

- improves performance for users:
  - Response time
  - Availability
  - Scalability
  - Load balancing
- conserves network and server resources
- requirement: logical transparency:
  - the presence of a proxy-cache between client and server should not change the information content received by the client in response to requests, e.g. client should never get stale data