WWW - Hypertext and Hypermedia

- The Web is a ***distributed hypermedia*** system that supports interactive access to Hypermedia documents (aka ***Resources)***.
- *Hypermedia* (as opposed to *Hypertext*) Resources potentially contain:
  - Different types of information including: text, pictures, graphics, audio etc. Examples include: HTML files, image files, query results etc.
  - *Hyperlinks* to other Resources,
- From a Network Programming perspective these Resources are treated as ***Data***.

WWW and the Client-server paradigm

- The distributed nature of the Web means that the Resources/Data are potentially spread across a number of computers across the Internet.
- This lends itself well to the Client-server paradigm as follows:
  - ***Client*:** The consumer of the Resources/Data are the end-users whom typically interact with a *client* application known as a Web Browser,
  - ***Server*:** Resource repositories are typically located on remote server-class machines and access to these Resources is typically controlled by a Web *server*.

Problems to be addressed

- However, this distribution of Resources also introduces a number of potential problems:
  - The Resources may be updated, moved or removed without notification to the client applications,
  - Likewise, links between Resources may be updated, moved or removed without notification to the client applications,
  - Accessing Resources on remote server-class machines has implications for network bandwidth usage.
- These problems can affect the end-user experience and the network.

Client-server interaction - HTTP

- Web browsers and servers interact with each other using the *HyperText Transfer Protocol* (HTTP).
- This is a *network protocol* used to deliver virtually all Resources on the Web:
  - The Web Browser client sends ***HTTP Request*** messages to Web Servers. These messages typically (but not always) contain requests for a Resource,
  - The Web server returns ***HTTP Response*** messages to the clients. These messages typically contain Resources/Data (but not always).

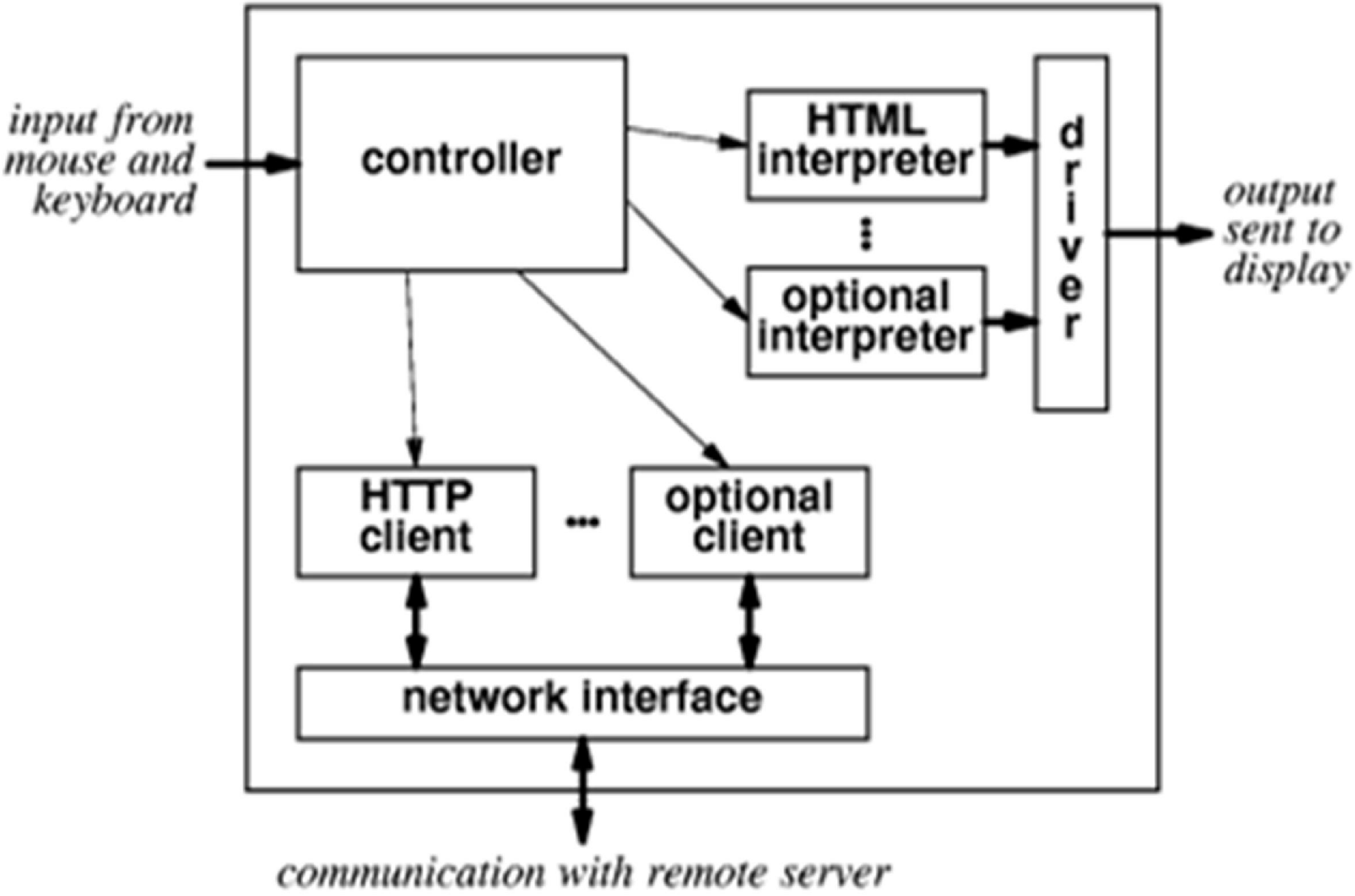
Operation of Web Browsers and Servers

- In order to appreciate the potential problems to be addressed when developing Web Browsers and Web servers it is important to understand their operation.
- Web Servers perform a straightforward task over and over again:
  - They accept HTTP Requests from clients,
  - They return HTTP Responses to clients indicating success or failure in dealing with the request.

Browser Architecture

- Web browsers are much more complex in their operation. This can best seen from their architecture (see next slide).
- The functions of a Browser include:
  - Rendering and displaying disparate (different types) resources to the user,
  - User interaction the user,
  - Initiating interaction with Web servers to retrieve resources or in some instances to upload resources.
- The Browser provides these services seamlessly using a number of software components.

Browser Architecture



Browser Architecture – contd.

- Specifically, a browser consists of the following components:
  - A set of *clients* for uploading/retrieving Resources*,*
  - A set of *interpreters* for displaying/rendering Resources,
  - A *Controller* to manage them all. The Controller is responsible for: Interpreting user input via the keyboard and mouse clicks <u>and</u> invoking interpreter and client components at the appropriate time.
- All browsers minimally contain a basic HTTP client, a basic HTML interpreter and a Controller.
  - Modern Browsers contain much more.

Interpreter and Client components

- An example *interpreter* is the ***HTML interpreter***:
  - It parses documents that contain standard HTML code and renders the content to the local screen,
  - Other interpreters are needed for displaying pictures, video, audio etc.
- An example *client* is the ***HTTP client***:
  - It is used to interact with HTTP servers for the purpose of retrieving/uploading Resources,
  - Other clients are needed for sending/receiving e-mail messages, file transfer using FTP etc.
  - The 1st field in the destination URL is used to determine which client component to invoke.

Document Transfer and HTTP

- From a Network Programming perspective we are interested in the HTTP client and its interaction with HTTP servers.
- This interaction consists of an exchange of HTTP Requests and Responses:
  - These are typically sent as plain-text encoded in ASCII i.e. in English,
  - This means that when viewed with a protocol analyzer such as Wireshark the Application Data field can be easily read and understood.

HTTP Requests

- HTTP Requests originate from the HTTP client.
- They support a number of operations through a set of ***methods***:
  - GET, HEAD, POST, OPTIONS, PUT, DELETE, TRACE and CONNECT.
  - For the purposes of this module we shall restrict ourselves to the GET and HEAD methods,
  - These two methods should adequately demonstrate the problems to be addressed.

HTTP Responses

- HTTP Responses originate from the HTTP server.
- Recall the problems previously outlined in relation to broken links, re-located Resources and Bandwidth limitations:
  - HTTP includes a lot of functionality to address these problems,
  - The server typically sends additional information with each transfer of data,
  - From this additional information the HTTP client can: call an interpreter to display/render the Resource data, infer an error condition etc.

Structure of HTTP Messages

- Recall that each layer of the Protocol Hierarchy specifies a “framing-type” structure known as a *Protocol Data Unit (PDU)*:
  - Examples include: a Data Link Frame, an IP Datagram/Packet, a TCP segment etc.
- HTTP is also a protocol:
  - It exists in the Application layer,
  - There are many other protocols that exist in the Application layer.
- When talking about Application layer protocols the term PDU has no real meaning.

Structure of HTTP Messages

- This is because Application layer protocols typically follow a *request-response* or *command-response* model of interaction:
  - Clients typically request something from the server or issue a command to the server,
  - Servers typically respond to the Client with an indication of success or failure,
  - However, sometimes servers issue requests and commands, but more on that later.
- Application layer protocols are more usefully described in terms of ***Syntax*** and ***Semantics***.

Syntax and Semantics

- **Syntax** describes the structure of the *request-response* messages.
- **Semantics** describes the interaction between the client and the server:
  - Essentially this relates to the sequence of request/response messages,
  - More usefully this can be described as "*who talks first*" i.e. which side issues the first message.

Syntax of HTTP Messages

- HTTP Messages have a particular structure or format.
- The format of the ***Requests*** and ***Responses*** messages are similar. Both consist of:

An initial line,

Zero or more Header Lines,

A blank line, and

An optional Message Body typically containing Resource data from a file, or a query output etc.

Syntax of HTTP Messages

- Specifically, the format of an HTTP message is:

  <initial line, different for *request* and *response*>

  Header1: value1

  Header2: value2

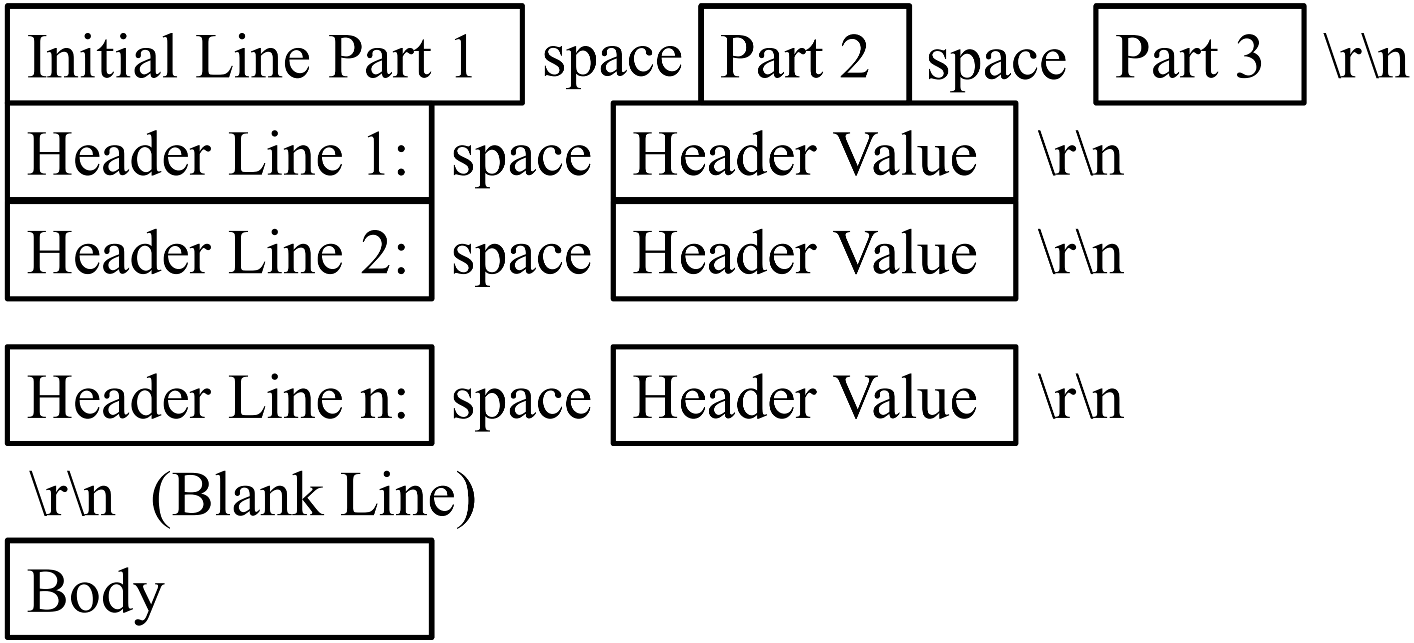
  Header3: value3

  <optional message body goes here, like file contents or query data>

- Initial lines and headers should end in CRLF
  - Specifically CR and LF here mean ASCII values 13 and 10 respectively.

Syntax of HTTP Messages

- This structure can more usefully described in terms of a *Protocol Box Diagram*:



Initial Line Part 1 space Part 2 space Part 3 \r\n

Header Line 1: space Header Value \r\n

Header Line 2: space Header Value \r\n

Header Line n: space Header Value \r\n

\r\n (Blank Line)

Body

Initial ***Request*** Line

- The initial line for a Request line has three parts, separated by spaces:
  - a method name
  - the local path of the requested resource
  - the version of HTTP being used.

- A typical request line is:

```
GET  /path/to/file/index.html  HTTP/1.0
```

Initial ***Request*** Line

- Notes:
  - GET is the most common HTTP *method*
    - it says “fetch me this resource"
    - *Method* names are always uppercase.
  - The *path* is the part of the URL after the host name
  - The HTTP *version* always takes the form "**HTTP/x.x**", uppercase.

Initial ***Response*** Line

- The initial response line also has three parts separated by spaces:
  - The HTTP version,
  - A response *status code* that gives the result of the request,
  - An English *reason phrase* describing the status code.

- Typical status lines are:

```
HTTP/1.0 200 OK
HTTP/1.0 404 Not Found
```

Initial ***Response*** Line

- Notes:
  - The HTTP *version* is of format "**HTTP/x.x**".
  - The *status code* is meant to be *computer-readable*
    - It comprises a three-digit integer, and the first digit identifies the general category of response
  - The *reason phrase* is meant to be *human-readable*, and may vary.

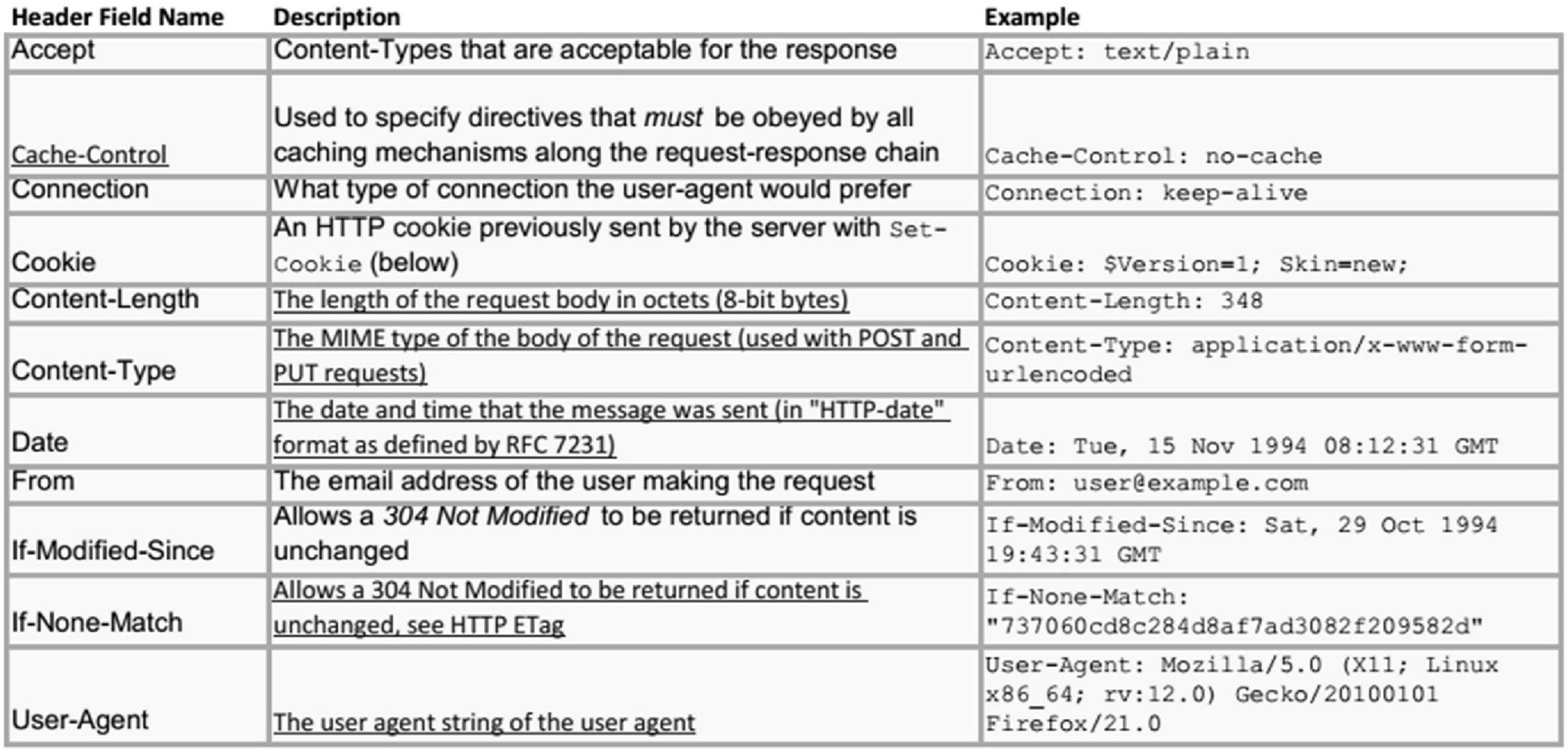
Header Lines

- Header lines provide information about the *request* or *response*, or about the Resource sent in the message body.
- The header lines are in a particular format
  - One line per header, of the form "Header-Name: value", ending with CRLF.
  - This is a similar format used for email and is defined in RFC 822.
  - The header name is not case-sensitive (though the value may be).

Header Lines

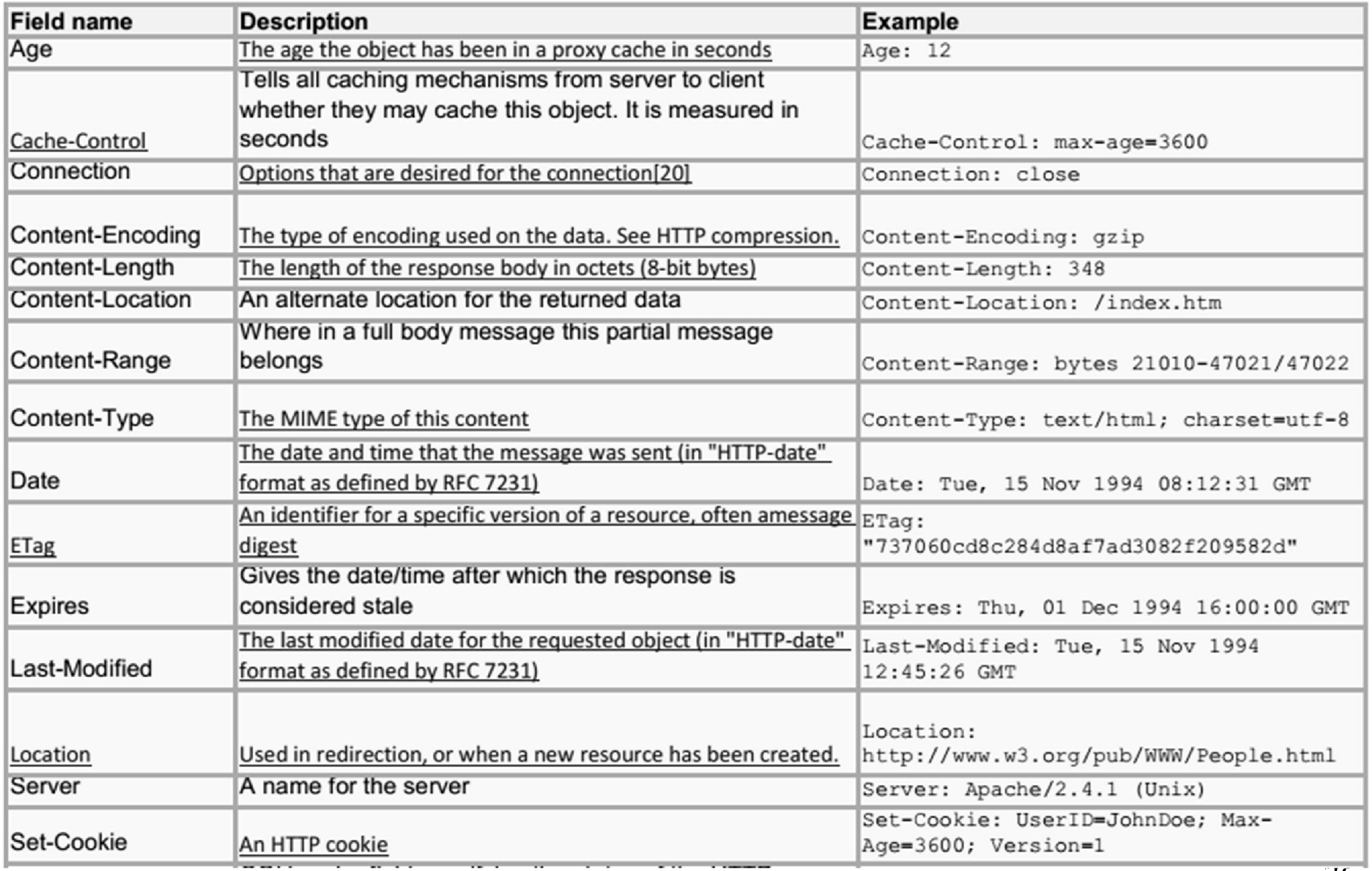
- There are two versions of HTTP:
  - HTTP 1.0 is older and defines 16 headers, although none are required.
  - HTTP 1.1 is newer and defines 46 headers, and one (Host:) is required in requests.

Example Request Header Lines



Header Field Name	Description	Example
Accept	Content-Types that are acceptable for the response	`Accept: text/plain`
Cache-Control	Used to specify directives that *must* be obeyed by all caching mechanisms along the request-response chain	`Cache-Control: no-cache`
Connection	What type of connection the user-agent would prefer	`Connection: keep-alive`
Cookie	An HTTP cookie previously sent by the server with `Set-Cookie` (below)	`Cookie: $Version=1; Skin=new;`
Content-Length	The length of the request body in octets (8-bit bytes)	`Content-Length: 348`
Content-Type	The MIME type of the body of the request (used with POST and PUT requests)	`Content-Type: application/x-www-form-urlencoded`
Date	The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231)	`Date: Tue, 15 Nov 1994 08:12:31 GMT`
From	The email address of the user making the request	`From: user@example.com`
If-Modified-Since	Allows a *304 Not Modified* to be returned if content is unchanged	`If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT`
If-None-Match	Allows a 304 Not Modified to be returned if content is unchanged, see HTTP ETag	`If-None-Match: "737060cd8c284d8af7ad3082f209582d"`
User-Agent	The user agent string of the user agent	`User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/21.0`

Example Response Header Lines



Field name	Description	Example
Age	The age the object has been in a proxy cache in seconds	`Age: 12`
Cache-Control	Tells all caching mechanisms from server to client whether they may cache this object. It is measured in seconds	`Cache-Control: max-age=3600`
Connection	Options that are desired for the connection[20]	`Connection: close`
Content-Encoding	The type of encoding used on the data. See HTTP compression.	`Content-Encoding: gzip`
Content-Length	The length of the response body in octets (8-bit bytes)	`Content-Length: 348`
Content-Location	An alternate location for the returned data	`Content-Location: /index.htm`
Content-Range	Where in a full body message this partial message belongs	`Content-Range: bytes 21010-47021/47022`
Content-Type	The MIME type of this content	`Content-Type: text/html; charset=utf-8`
Date	The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231)	`Date: Tue, 15 Nov 1994 08:12:31 GMT`
ETag	An identifier for a specific version of a resource, often amessage digest	`ETag: "737060cd8c284d8af7ad3082f209582d"`
Expires	Gives the date/time after which the response is considered stale	`Expires: Thu, 01 Dec 1994 16:00:00 GMT`
Last-Modified	The last modified date for the requested object (in "HTTP-date" format as defined by RFC 7231)	`Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT`
Location	Used in redirection, or when a new resource has been created.	`Location: http://www.w3.org/pub/WWW/People.html`
Server	A name for the server	`Server: Apache/2.4.1 (Unix)`
Set-Cookie	An HTTP cookie	`Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1`

Header Lines - Net-politeness

- Consider including the following headers in *client* requests:
  - A **From:** header gives the email address of whoever's making the request, or running the program doing so. (This must be user-configurable, for privacy concerns.)
  - A **User-Agent:** header identifies the program that's making the request, in the form "**Program-name/x.xx**", where x.xx is the (mostly) alphanumeric version of the program.
  - e.g. Netscape 3.0 sends the header "User-agent: Mozilla/3.0Gold".

Header Lines - Net-politeness

- Consider including the following headers in *server* responses:
  - A **Server:** header. Similar to the **User-Agent:** header: it identifies the server software in the form "**Program-name/x.xx**".
  - e.g. An Apache server might return "Server: Apache/1.2b3-dev".
  - The **Last-Modified:** header gives the modification date (in GMT) of the resource that's being returned. It is used in caching.
  - e.g. **Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT**

The Message Body

- A HTTP *message* may have a body of data sent after the header lines.
- In a *response:*
  - This is where the requested resource is returned to the client (the most common use of the message body),
  - Or some explanatory text for an error condition.
- In a *request*:
  - This is where form data or uploaded files are sent to the server.

The Message Body

- If a HTTP message includes a body, there are usually header lines in the message that describe the body. In particular,
  - The **Content-Type:** header gives the MIME-type of the data in the body, such as text/html or image/gif.
  - The **Content-Length:** header gives the number of bytes in the body.

Other HTTP Methods - HEAD and POST

- Two other commonly used methods are HEAD and POST.
- The HEAD Method
  - Similar to a GET request, except it asks the server to return the response headers only, not the actual resource (i.e. no message body).
  - Useful for checking characteristics of a resource without actually downloading it.
  - The response to a HEAD request must never contain a *message body*.

Other HTTP Methods - HEAD and POST

- The POST Method
  - A POST request is used to send data to the server to be processed in some way, like by a CGI script.
  - It differs from a GET request in the following ways:
    - There's a block of data sent with the request, in the message body. There are usually extra headers to describe this message body, like **Content-Type:** and **Content-Length:**.
    - The request URI is not a resource to retrieve; it's usually a program to handle the data you're sending.

Sample Document Transfer with HTTP

```
GET http://www.comp.dit.ie/dbourke/ HTTP/1.1
Accept-Language: en-us,en;q=0.5
Accept:text/xml,application/xml,application/xhtml+x
    ml,text/html;q=0.9,text/plain;q=0.8,image/png.
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Host: www.comp.dit.ie
Accept-Encoding: gzip,deflate
User-Agent: Mozilla/5.0 (Windows; U; Windows NT
    5.1; en-US; rv:1.8.0.1) Gecko/20060111
    Firefox/1.5.0.1
Cookie: PHPSESSID=13ceaac67329048c4
Keep-Alive: 300
Proxy-Connection: keep-alive

HTTP/1.1 200 OK
Pragma: no-cache
Cache-Control: no-cache
MicrosoftOfficeWebServer: 5.0_Pub
ETag: "e21ceefa6e28df"
Accept-Ranges: bytes
Content-Type: text/html
Connection: close
Date: Wed, 22 Oct 2008 14:20:12 GMT
Server: Microsoft-IIS/6.0
Content-Location:
    http://www.comp.dit.ie/dbourke/index.htm
Last-Modified: Thu, 02 Oct 2008 09:12:23 GMT
Content-Length: 1837
X-Powered-By: ASP.NET
```

Data derived from Muffin

HTTP Versions 1.0 and 1.1

- As mentioned there are two versions of HTTP: **HTTP/1.0** and **HTTP/1.1**
- There are some key differences between the two in the following areas (not all are listed):
  - Persistent connections,
  - Multi-hosting
  - More efficient cache control.

HTTP/1.0 Connections

- With HTTP 1.0 the connection between the server and the client is closed by the server immediately after the HTTP Response is transmitted.
- Consider a simple HTML page containing five image tags:
  - Downloading and rendering this page requires **six** connection establishments and cessations,
  - One for the HTML page and one for each of the images. The images are downloaded separately on a separate connection.

HTTP/1.0 Connections

- This behaviour has implications for:
  - **Network bandwidth** due to the amount of extra segments required to establish and terminate connections,
  - **Internal TCP resources**. Each connection consumes resources within the TCP memory space.

HTTP/1.1 Persistent Connections

- With HTTP 1.1 the connection between the server and the client remains open by default:
  - This allows for multiple HTTP Requests to be submitted across a single connection,
  - This default behaviour is not always required and it can be overridden using the ***Connection:*** header as follows:

    Connection: close\r\n

  - This is an instruction to the server to close the connection after the Resource has been returned.

Multi-hosting

- HTTP/1.1 facilitates *Multi-hosting*:
  - Multiple webservers sharing a single IP address,
  - The problem is that each server is listening on Port 80,
  - This causes problems for TCP determining which server should receive an incoming HTTP Request,
  - The problem is solved using the ***Host:*** header as follows:

    Host: www.tudublin.ie
  - Its inclusion is mandatory in all HTTP/1.1 Requests.

Caching

- The *locality of reference* principal suggests:
  - Once a pair of hosts communicate with other they are most likely to continue communicating with each other frequently (*temporal* locality of reference),
  - Hosts tend to communicate with other hosts nearby (*physical* locality of reference).
- Browsers tend to adhere to the *temporal locality of reference* rather than the *physical locality of reference*:
  - Users tend to access Web pages on remote machines.
- To improve performance, browsers use a ***cache***.

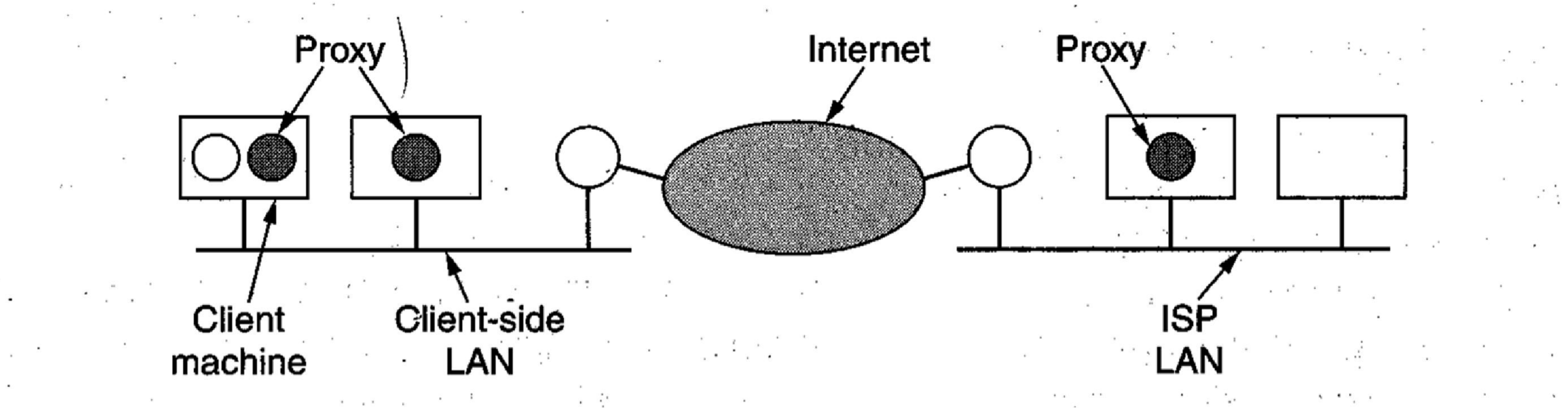
Caching In Web Browsers

- Items in a cache can be read more quickly than over a network connection.
- However, unnecessary caching can:
  - Consume a lot of disk space
  - Degrade performance because the browser has to write the items to disk
- Most browsers have an adjustable caching policy.

HTTP Support For Caching

- HTTP supports caching
  - Servers can specify a cache timeout for a page,
  - Servers can specify non-caching,
  - Clients (Browsers) can revalidate and reuse a cached item using the HTTP HEAD operation.

Example *Hierarchical Caching*



How long should pages be cached?

- *Cacheability* of pages can vary greatly:
  - Some pages should never be cached e.g. Stock Prices,
  - Other pages rarely change and remain valid for hours or days or even years e.g. a list of Greek Gods.
- The main problem to be addressed is the *staleness* of pages.
- Proxy must purge pages from cache, however:
  - If done too quickly it will not be a very effective cache,
  - If pages are kept too long it may return *stale* pages

Caching policies

- Two approaches to dealing with this problem:
  - The first uses *heuristics* to guess how long to keep each page:
    - Often based on the *Last-Modified* header of the page. If it was modified an hour ago, it is held for an hour etc.
    - While this *approach* works well in practice it does, however, return stale pages from time to time.

Caching policies

- The second approach uses special features of HTTP that deal with cache management:
  - The ***If-Modified-Since*** request header, can be sent to a server, specifying the time the cached page was last modified (from the *Last-Modified header*),
  - If the page has not been modified since then, the server sends back a short *Not Modified message* (status code 304),
  - If the page has been modified since then, the new page is returned,

- This approach still requires a connection however, the reply message will be very short if the cached page is still valid

Sample Caching Interaction

- Sample client **request** header for a document (first retrieval):

```
GET /sample.html HTTP/1.1
Host: example.com
```

- A typical server response (including headers) would be:

```
HTTP/1.x 200 OK
Content-Length: 32859
Expires: Tue, 27 Dec 2005 11:25:11 GMT
Date: Tue, 27 Dec 2005 05:25:11 GMT
Server: Apache/1.3.33 (Unix) PHP/4.3.10
Cache-Control: max-age=21600
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Etag: “4135cda4″
```

  - **Cache-Control**: The max. time (seconds) to cache the document
  - **Last-Modified**: The document’s last modified date
  - **Etag**: A unique hash for the document
  - The client caches this document for 21600 seconds

Sample Caching Interaction – Contd.

- Assume that the user calls for the same document /sample.html within the specified cache time-frame. The browser (client) makes a conditional GET request

```
GET /sample.html HTTP/1.1
Host: example.com
If-Modified-Since: Wed, 01 Sep 2004 13:24:52 GMT
If-None-Match: “4135cda4″
```

  - *If-None-Match* specifies the original Etag value

- A typical response to this request would be:

```
HTTP/1.x 304 Not Modified
Expires: Tue, 27 Dec 2005 11:25:19 GMT
Date: Tue, 27 Dec 2005 05:25:19 GMT
Server: Apache/1.3.33 (Unix) PHP/4.3.10
Etag: “4135cda4″
Cache-Control: max-age=21600
```

- The *304* response header allows the client to use the cached document