Chapter 4: The Hypertext Transfer Protocol (HTTP)

References:

- Erik Wilde: World Wide Web — Technische Grundlagen. Springer, 1999, ISBN 3-540-64700-7, 641 Seiten.
- T. Berners-Lee, R. Fielding, H. Frystyk: Hypertext Transfer Protocol — HTTP/1.0. RFC 1945, May 1996, 60 pages.
- R. Fielding et al.: Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, June 1999, 176 pages.
- J. Franks et al.: HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, June 1999, 34 pages.
- David H. Crocker (Ed.): Standard for the Format of ARPA Internet Text Messages. RFC 822, August 1982, 47 pages.
- P. Wainwright: Professional Apache. Wrox Press, 1999, ISBN 1-861003-02-1, 617 pages.
- D. Kristol, L. Montulli: HTTP State Management Mechanism. RFC 2965, Oct. 2000, 26 pages.
- K. Moore, N. Freed: Use of HTTP State Management. RFC 2964, Oct. 2000, 8 pages.
- David M. Kristol: HTTP Cookies: Standards, privacy, and politics. ACM Transactions on Internet Technology (TOIT), Volume 1 , Issue 2 (November 2001), Pages: 151 - 198
- S. Brass: Cookies. In R. Flynn (Ed.): Macmillan Computer Sciences. Macmillan, 2002.

Objectives

After completing this chapter, you should be able to:

- explain what exactly happens when you click on a link in a web page.

  You should be able to write HTTP requests and interpret HTTP responses. Why it is good to keep the TCP connection open for a short time after the response?

- explain how language and format are selected.
- explain authentication for protected pages.
- explain cookies including privacy problems.
- understand many of the configuration options for a web server.

Overview

1. Requests and Responses
2. Content Negotiation
3. Access Control/Password-Protected Pages
4. Caching (Proxies)
5. State Management (Cookies)

HTTP Communication (1)

- Example: Suppose the following URL is requested:

  `http://www.informatik.uni-giessen.de/index.html`

- The basic HTTP communication model has four steps:
  - ◇ Opening a TCP connection to the web server.
  - ◇ Sending a request to the web server.
  - ◇ Receiving a response from the web server (which includes the data of the requested web page).
  - ◇ Closing the connection (optional).

HTTP Communication (2)

First Step: Opening a TCP Connection

- The URL (web address) contains the name of the web server: `www.informatik.uni-giessen.de`.
- The Browser asks a DNS server for the IP address.

  If IP address lookup fails: "Netscape is unable to locate the server".
- The Client (Browser) opens a TCP-connection to port 80 of this machine (134.176.28.61).

  80 is the default port number of HTTP. One can specify another port number explicitly in the URL. If no process is listening on that port, Netscape prints the error message "Connection refused". If the machine is switched off or not reachable via the network, it prints "There was no response".

HTTP Communication (3)

Second Step: Request

- The client (browser) requests an object (file) from the server.
- This is done with a human-readable message, e.g.

```
GET /index.html HTTP/1.0
(Empty Line)
```

- One can open a TCP connection with

```
telnet www.informatik.uni-giessen.de 80
```

  and enter the request manually.

HTTP Communication (4)

- Between the `GET`-line and the empty line many options ("Headers") can be specified, see below.

  The empty line is needed to mark the end of the `GET`-request. The client does not immediately close the connection after it has sent the request, therefore the server needs another means to know when the request is complete. `POST`-requests contain data after the empty line, but there a header specifies how many bytes the server must still read after the empty line.

- In HTTP 0.9 the request was simply

  ```
  GET ⟨Filename⟩
  ```

  (without further options).

HTTP Communication (5)

- Example for a request sent by Netscape 4.76:

```
GET /index.html HTTP/1.0
Referer: http://www.informatik.uni-giessen.de/.../c3.html
Connection: Keep-Alive
User-Agent: Mozilla/4.76 [en] (X11; U; SunOS 5.8 sun4u)
Host: wega.informatik.uni-giessen.de:8080
Accept: image/gif, image/x-xbitmap, image/jpeg,
        image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
(Empty Line)
```

HTTP Communication (6)

- Example for a request sent by Internet Explorer:

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg,
        image/pjpeg, application/vnd.ms-powerpoint,
        application/vnd.ms-excel,
        application/msword, */*
Referer: http://www.informatik.uni-giessen.de/.../c3.html
Accept-Language: en-us,de;q=0.5
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
        (compatible; MSIE 5.5; Windows 98; Win 9x 4.90)
Host: wega.informatik.uni-giessen.de:8080
Connection: Keep-Alive
(Empty Line)
```

HTTP Communication (7)

- Request of the web robot (spider, crawler):

```
GET /robots.txt HTTP/1.0
Host: www.informatik.uni-giessen.de
Accept: text/*
User-Agent: Slurp/si (slurp@inktomi.com;
        http://www.inktomi.com/slurp.html)
From: slurp@inktomi.com
(Empty Line)
```

- Web robots are programs that “surf” on the web and try to download as many as possible web pages, e.g. for entering them into a search engine index.

HTTP Communication (8)

- The request can contain data (e.g. from a form):

```
POST /db-cgi/grades?view HTTP/1.0
Referer: http://.../staff/brass/grades/view.html
Connection: Keep-Alive
User-Agent: Mozilla/4.73 [en] (X11; U; SunOS 5.7 sun4m)
Host: www.informatik.uni-giessen.de
Accept: image/gif, image/x-xbitmap, image/jpeg,
        image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 46

first_name=Stefan&last_name=Brass&password=abc
```

HTTP Communication (9)

Third Step: Response

- The server answers with the requested document (object, entity), e.g. it transfers an HTML file.

  The contents of the HTML file is normally sent including comments etc., i.e. it is not interpreted.

- Before the real data, a status code is sent (e.g. 200 "OK"), as well as information about the document (meta data) and about the server.
- These headers and the data are again separated by an empty line.

  In HTTP 0.9 only the document was sent.

HTTP Communication (10)

- Example for a response from the Apache Server:

```
HTTP/1.1 200 OK
Date: Thu, 16 Nov 2000 18:52:10 GMT
Server: Apache/1.3.12 (Unix)
Last-Modified: Mon, 08 May 2000 09:22:58 GMT
ETag: "60304-46b-39168772"
Accept-Ranges: bytes
Content-Length: 1131
Connection: close
Content-Type: text/html

... HTML Document ...
```

HTTP Communication (11)

- Example for a response from the Microsoft Internet Information Server (IIS):

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Content-Location: http://136.142.116.25/Default.htm
Date: Thu, 16 Nov 2000 19:00:39 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Thu, 02 Mar 2000 23:41:04 GMT
ETag: "fca66ec6a084bf1:abe"
Content-Length: 4263

... HTML Document ...
```

HTTP Communication (12)

- In this way, arbitrary files can be transfered, not only HTML:

```
HTTP/1.1 200 OK
Date: Fri, 17 Nov 2000 07:35:20 GMT
Server: Apache/1.3.12 (Unix)
Last-Modified: Tue, 18 Jul 2000 12:11:48 GMT
ETag: "9ac90-d2f-39744984"
Accept-Ranges: bytes
Content-Length: 3375
Connection: close
Content-Type: image/jpeg

... Binary Data of the JPEG-File (3375 Bytes) ...
```

HTTP Communication (13)

- Often, the data that the server sends are simply the contents of a file stored on the server (“static contents”).
- However, it is also possible that the data are computed by an arbitrary program that runs on the server (“dynamic contents”).

  The WWW server communicates with this program via CGI (“Common gateway Interface”). Alternative: “Servlets” written in Java.

HTTP Communication (14)

- It is also possible that an HTML file contains commands or program pieces that are interpreted by the server (Server Side Includes, Active Server Pages, etc.).
- A program (e.g. a DBMS) can also directly have an HTTP interface.
- Even my printer can be controlled with a brower via a built-in HTTP interface.

HTTP Communication (15)

Fourth Step: Closing the Connection

- Earlier, the server always closed the connection after it had transfered the requested data.
- But this turned out to be inefficient, since often further files (images, frame contents, more web pages) must be fetched from the same server.

  TCP needs a three-way handshake for opening a connection, and for closing a connection, even four packets must be sent. In addition, the operating system must keep the data about the connection for a short time in case the final acknowledgement was lost and the other side resends a packet.

HTTP Communication (16)

- Therefore, client and server can agree to keep the TCP connection alive for a short time.

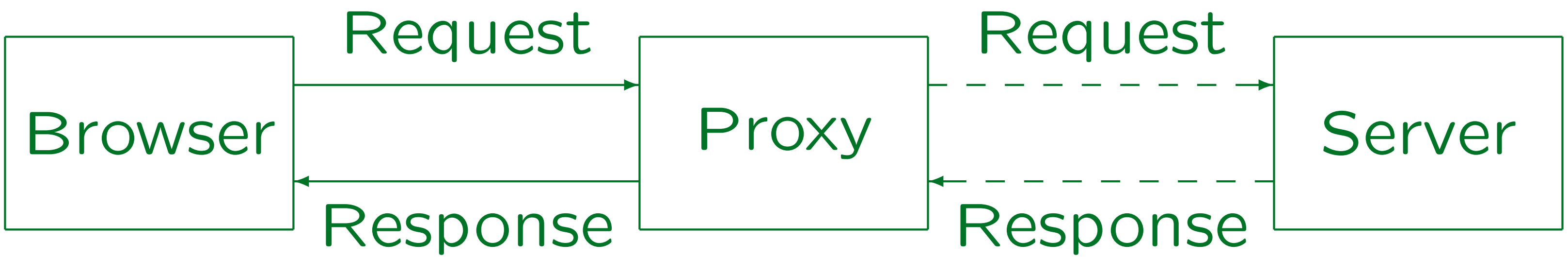
  This is done via the header "`Connection:`".

- If the client knows that it needs several documents from the server, it can send the requests one after the other without waiting for the response.

  This is called "pipelining". Earlier, browsers often opened many concurrent connections to the same server, but that creates an unnecessary load on the server. Today, the rule is that client should not open more than two concurrent connections to the same server.

Proxies (1)

- Sometimes client (browser) and server communicate via one or more proxy servers (caches):



- Browsers can be configured in such a way that they send all requests to a fixed proxy server (e.g. operated by the ISP) instead of the real server.

Proxies (2)

- The proxy then checks whether the requested page is in its cache.

  It also tries to check whether the page is still current, see below.

- If yes, the proxy answers the request from its cache.
- If not, the proxy sends the request to the real server (“Origin Server”) or another proxy.
- It forwards the response that it gets to the client, but in addition it saves the response in its cache (for future requests to the same URL).

Syntax of a Request (1)

- A request consists of
  - ◇ a command line,
  - ◇ zero or more headers,
  - ◇ an empty line,
  - ◇ a body (entity, data) (optional).
- A request command line consists of
  - ◇ A method, e.g. `GET`.
  - ◇ An identification of the resource to which the method should be applied (e.g. absolute path).
  - ◇ The HTTP-version of the request, e.g. `HTTP/1.1`.

Syntax of a Request (2)

- The most common resource identifications are:
  - ◇ An absolute path.
  - ◇ An absolute URI (Uniform Resource Identifier).

    A URI can be a URL ("Uniform Resource Locator", "web address") or a URN ("Uniform Resource Name", not yet used, see Chapter 3).

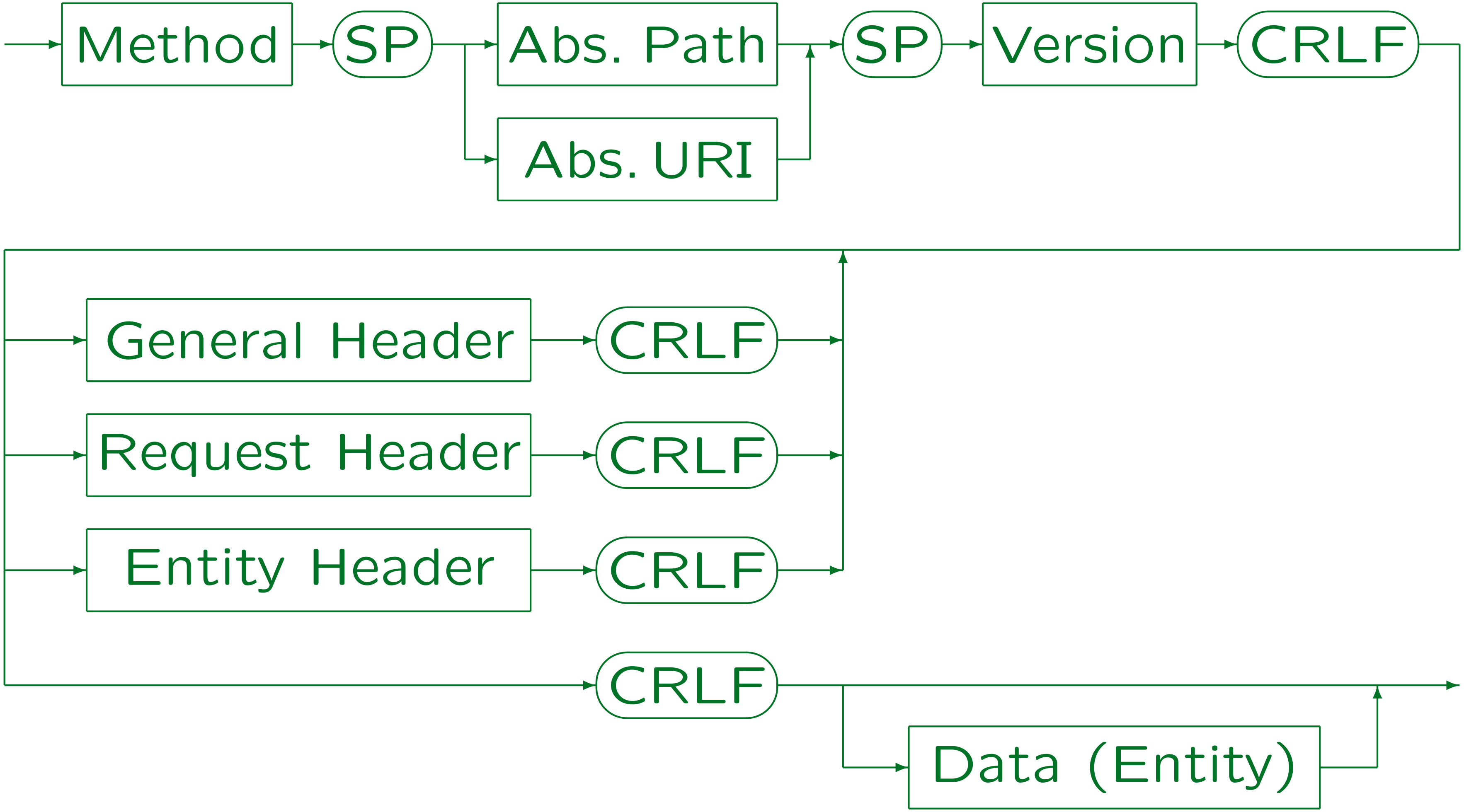
- Proxy servers require an absolute URI, for normal web servers the absolute path suffices.

  In HTTP/1.0, normal web servers only accepted an absolute path.

Syntax of a Request (3)

- There are four classes of headers:
  - ◇ General Header: In request and response, no matter whether it contains data or not.
  - ◇ Entity Header: In request and response, but only if it contains data (an entity).
  - ◇ Request Header: Only in a request.
  - ◇ Response Header: Only in a response.

Syntax of a Request (4)



Syntax of Headers (1)

- The syntax of headers is the same as in Emails (e.g. "From:"), see RFC 822.
- A header consists of:
  - ◇ The name of the header (field).
  - ◇ A colon ":".
  - ◇ The value of the header.
  - ◇ Carriage Return and Linefeed (CRLF).
- On both sides of colon and after the value white space (e.g. blanks, tabs) is permitted, but the name of the header must start in the first column.

Syntax of Headers (2)

- A header can be distributed over multiple lines. Continuation lines must start with a blank or tab.

  The syntactic analysis merges such white space to a single space.

- The sequence of headers is not important.

  Exception: If the same header is repeated, the values are concatenated (separated by commas). Then the sequence is important for the result value. Headers can only be repeated if their value must be a comma-separated list (e.g. `Accept`).

Methods (1)

- GET: The data stored under the given path/URI are requested.

  This can be the contents of a file on the server, but the path/URI can also identify a program that computes the data. This depends on the server configuration, and even a simple URL that looks like a file name can actually be computed. Arguments/Parameters for the program can be appended after a "?" to the path. For a GET request, the program should not perform state changes on the server (GET requests can be cached in a proxy, not all actually reach the server).

- HEAD: Like GET, but only the headers should be delivered, not the data (body).

  E.g. in this way one gets the date of last change, the file size, the media type (MIME type), etc. (meta data).

Methods (2)

- POST: Data are transfered from the client to the server which should be assigned to the given URI.
  - ◇ Most often this is applied for data the user entered into a form. The URI then names a program that should process the data.

    Also the GET method can be used for transfering form data to the server. But if the form data are stored on the server, and not only used for computing a result web page (e.g. query forms), POST is preferable.
  - ◇ However, the URI could also name a newsgroup in which the data/message should be posted.

Methods (3)

- POST, continued:
  - ◇ The URI can name also a database relation, in which the data should be inserted as new row.
  - ◇ Another possibility is that the URI names a document, to which the data should be attached as annotation.
  - ◇ What exactly happens, depends on the configuration of the server (and the URI). HTTP does not prescribe a specific action.

Methods (4)

Further Methods (not always implemented):

- `PUT`: The data sent in the request body should be saved on the server under the specified URI.

  If a document exists under this path, it is overwritten. It depends on the configuration of the server and the access rights of the client whether the server actually performs the request. Not every server understands "`PUT`". The specification states that all methods except `GET` and `HEAD` are optional. But `POST` is also very common.

- `DELETE`: The document stored under the given URI should be deleted.

  With `PUT` and `DELETE`, remote administration of the web server contents is possible. Of course, only authenticated users with special access rights should be allowed to do this.

Methods (5)

- `TRACE`: The server sends the request back as data.

  This can be interesting if one or more proxies are on the way from the client to the server, which possibily modify the request. They also add their address in a `Via:` header, which can be queried in this way.

- `OPTIONS`: The server sends back the methods that would be acceptable for the given URI.

  This is done in the `Allow:` Header. Instead of a path/URI, one can also specify "`*`" in order to get all methods supported by the server.

- `CONNECT`: For SSL connections via a proxy.
- One can also define one's own methods.

General Headers (1)

- `Cache-Control`: Information for proxies, see below.
- `Connection`: Client/server can state whether they want to keep the TCP-connection after sending the request/resonse (`Keep-Alive`) or not (`close`).
- `Date`: Date and time when the request or response was constructed.
- `Pragma`: Was used for proxy information, see below.

General Headers (2)

- `Trailer`: Used for chunked encoding to specify headers that will be sent after the body, see below.
- `Transfer-Encoding`: Encoding of the body in order to safely transfer it (e.g. `chunked`), see below.
- `Upgrade`: For changing to a different protocol.
- `Via`: Proxies between client and server add this header with their address to the request.
- `Warning`: Warning generated by a proxy.

Request Headers (1)

- Headers for content negotiation (see below):
  - ◇ `Accept`: Acceptable media types.
  - ◇ `Accept-Encoding`: Acceptable encodings.

    E.g. compression methods.
  - ◇ `TE`: Acceptable transfer encodings.
  - ◇ `Accept-Charset`: Acceptable character sets.
  - ◇ `Accept-Language`: Acceptable languages.
- `Authorization`: For password-protected pages.

  See below.

Request Headers (2)

- **Expect**: The client uses the value “`100-continue`” to state that after sending the headers it waits for an acknowledgement before it will send the data.
- **From**: Email-address of the user who is responsible for the request.

  Earlier, browsers sent the users email address quite freely. Now the specification recommends that the email address should only be sent if the user explicitly agreed, which usually means that it is not sent.
- **Host**: Name of the web server.

  Important for virtual servers. HTTP/1.1 requires that this header is specified if the command line does not contain an absolute URI.

Request Headers (3)

- Headers for conditional requests (important for proxies and search engines which already have an old version of the web page, see below):
  - ◇ `If-Match`
  - ◇ `If-Modified-Since`
  - ◇ `If-None-Match`
  - ◇ `If-Range`
  - ◇ `If-Unmodified-Since`

Request Headers (4)

- `Max-Forwards`: Number of proxies that can still relay the request on its way to the server.

  Each proxy must decrement this value by 1. If it is 0, it must not forward the request, but answer the client (normally with an error message).

- `Proxy-Authorization`: If the proxy supports only selected clients, this can be used to prove one's identity.

- `Range`: Only a part of the entity is requested.

  E.g. "`Range: bytes=0-1023`" returns only the first KB.

Request Headers (5)

- `Referer`: URL of the web page that contains a reference to the requested URI.

  This is useful for statistical evaluations of the web traffic: E.g. which web pages contain links to my web page, and from where most users find my page? It also helps to find broken links: If the requested path/URI does not exist, the referencing page is known and it can either be repaired (it it is on my web server) or I can try to find out who is responsible for the web page and send him/her an email.

- `User-Agent`: Information about the browser.

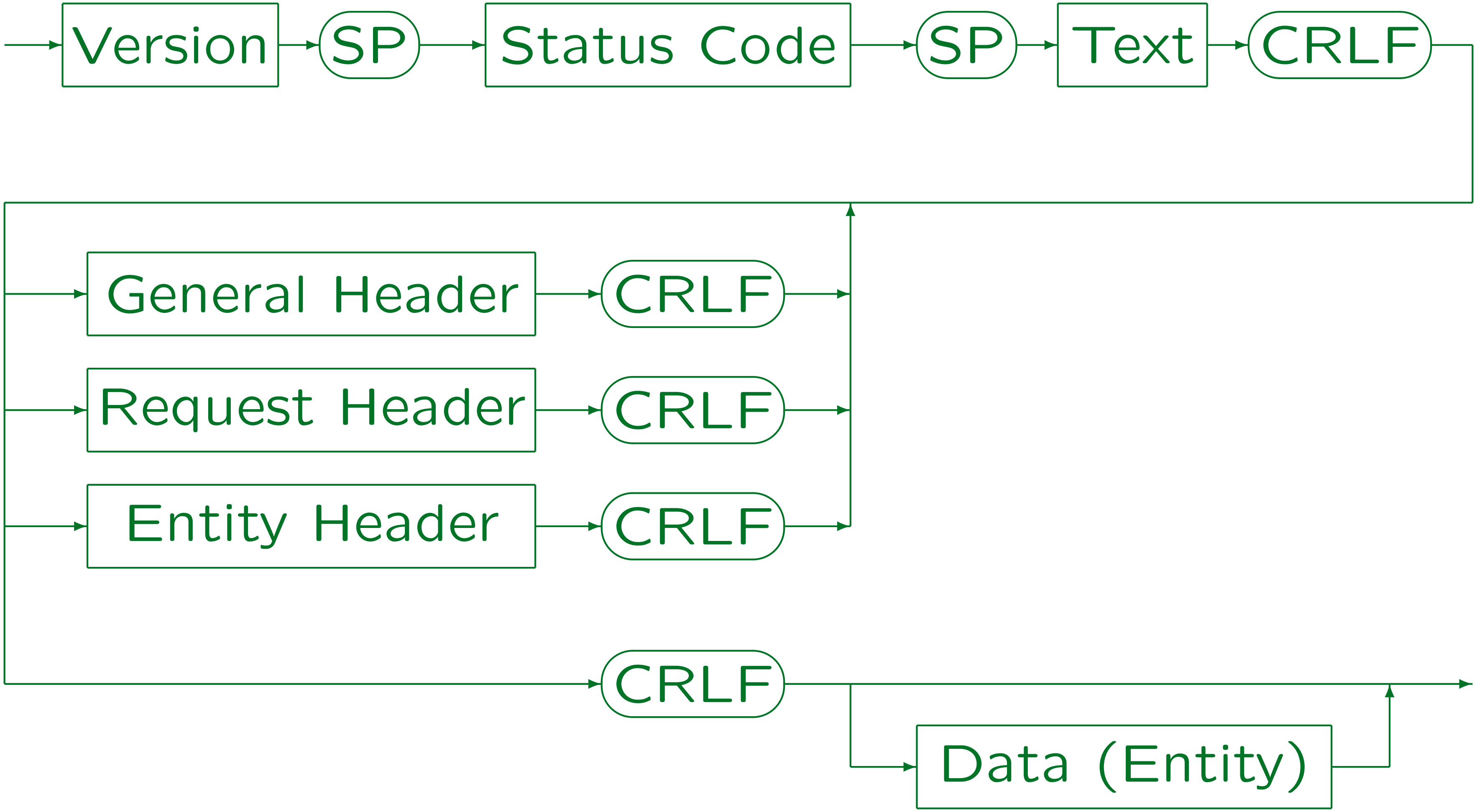
  E.g. the server can deliver different versions based on which browser requests the web page. The server can also collect statistics how often which browser is used.

Syntax of a Response (1)

- A response consists of:
  - ◇ a status line,
  - ◇ zero or more headers,
  - ◇ an empty line,
  - ◇ a body (entity, data, document) (optional).
- The status line consists of:
  - ◇ the HTTP-version,
  - ◇ a status code (three digits) (e.g. error number),
  - ◇ a text that explains the status code.

    Status code: for the computer, Text: for human user.

Syntax of a Response (2)



Status Codes (1)

- Status codes consist of three digits. The first digit specifies the general class of the status code.

  If the client does not know the specific status code, it can treat the situation like the code 00 of the class.

- 1xx: Intermediate reply, additional response follows.
  - ◇ 100: Continue (i.e. client should send data).

    This is an answer to the `Expect` header.
  - ◇ 101: Switching Protocols

    The server accepts the protocol that the client suggested in the `Upgrade` header.

Status Codes (2)

- 2xx: Successful:
  - ◇ 200: OK

    The requested operation was executed successfully. E.g. one requested a web page which is successfully returned in this response.
  - ◇ 201: Created.

    The operation created a new resource (e.g. answer to a PUT request). The location of the resource is returned in the body, and the most specific URI also in a Location header.
  - ◇ 202: Accepted.

    The requested operation will be executed later.

Status Codes (3)

- 2xx: Successful, continued:
  - ◇ 203: Non-Authoritive Information.

    Meta information was changed by the proxy.
  - ◇ 204: No Content

    The requested operation was executed, but answer is empty. The browser should not change the document in the browser window if it gets a 204 answer.
  - ◇ 205: Reset Content.

    The submitted form data were successfully processed. The form should now be emptied so that the user can enter the next record.
  - ◇ 206: Partial Content.

    This is an answer to a `Range` request.

Status Codes (4)

- 3xx: Redirection (requires another request).
  - ◇ 300: Multiple Choices.

    There are several variants for the resource. The body of the response contains information about the variants. The browser may choose automatically (e.g. GIF vs. JPEG).
  - ◇ 301: Moved Permanently.

    The web address of the resource has changed. The response contains a `Location` header with the new URI. Hyperlinks and bookmarks should be updated.
  - ◇ 302: Found (temporary redirect).

    The resource was moved temporarily to a new URI which is given in the `Location` header. 302 means officially the same as 307, but many browsers interpret it like 303 (see below).

Status Codes (5)

- 3xx: Redirection, continued:
  - ◇ 303: See Other.

    This is also a temporary redirect, the browser should automatically load the web page at the URI given in the Location header. The new URI must be accessed with the method GET. E.g. this is sometimes used when form data submitted with POST, and the CGI program does not want to return the data of the result page, but ask the browser instead to fetch it.

  - ◇ 304: Not Modified.

    This is an answer to conditional requests like If-Modified-Since.

Status Codes (6)

- 3xx: Redirection, continued:
  - ◇ 305: Use Proxy.

    The resource may only be accessed via a proxy server. The address of the proxy server is contained in the `Location` header.

  - ◇ 307: Temporary Redirect.

    This actually means the same as 302: The browser should automatically access a different URI given in the `Location` header. In contrast to 303, the new URI must be accessed with the same method as the original request. Since many browsers interpreted 302 like 303, this status code was introduced to emphasize the difference. E.g. if the address of a CGI program for processing form data has changed, this status code should be used.

Status Codes (7)

- 4xx: Client Error (error of browser/user).
  - ◇ 400: Bad Request.

    The request is syntactically invalid. E.g. HTTP/1.1 was specified as protocol version, but the required Host header is missing.
  - ◇ 401: Unauthorized.

    E.g. this page is password protected. The normal reaction of the browser is to ask the user for a user name and password, and then try it again with a request that includes this data, see below.
  - ◇ 402: Payment Required.

    This is reserved for future use. There are already web pages that can only be accessed for paying money. Today the user must explicitly register, in future the browser may manage a small purse.

Status Codes (8)

- 4xx: Client Error, continued:
  - ◇ 403: Forbidden.

    The server refuses to deliver the data. E.g. the file access rights on the server might be wrong (Under UNIX, files normally must be readable by everybody, otherwise the web server process cannot access them. Depending on the "umask" that defines rights for new files, the file access rights must be explicitly changed for the web pages. If that was forgotten, this error occurs.). However, it is also possible to configure the web server such that certain pages can only be accessed from the local net.
  - ◇ 404: Not Found.

    The path specified in the request was wrong: There is no such web page.

Status Codes (9)

- 4xx: Client Error, continued:
  - ◇ 405: Method not allowed.

    The request method cannot be applied to this URI.
  - ◇ 406: Not Acceptable.

    There was no variant of the resource that could fulfill the constraints specified in the Accept-headers of the request. E.g. the browser specified that it only understands the GIF and JPEG picture formats, but the image exists only in a PNG version. However, the server may simply deliver a different media type than requested (and not give the 406 error).
  - ◇ 407: Proxy Authentication Required.
  - ◇ 408: Request Timeout.

Status Codes (10)

- 4xx: Client Error, continued:
  - ◇ 409: Conflict.

    This might e.g. happen with PUT requests if two users independently edited the same file.
  - ◇ 410: Gone.

    The web page was deleted, and is no longer offered (at least, no forwarding address is known). Bookmarks to this URI should be deleted.
  - ◇ 411: Length Required.

    A Content-Length header is required for this request.
  - ◇ 412: Precondition Failed.

Status Codes (11)

- 4xx: Client Error, continued:
  - ◇ 413: Request Entity Too Large.
  - ◇ 414: Request-URI Too Long.
  - ◇ 415: Unsupported Media Type.

    The server does not understand the format of the request body as specified in the Content-Type header.
  - ◇ 416: Requested Range Not Satisfiable.

    A byte-range was requested with a start address that is larger than the current size of the file.
  - ◇ 417: Expectation Failed.

Status Codes (12)

- 5xx: Server Error.
  - ◇ 500: Internal Server Error.

    This error code is e.g. returned when the CGI-program crashed (that was supposed to compute the response).

  - ◇ 501: Not Implemented.

    The request method is not known to the server.

  - ◇ 502: Bad Gateway.

    This is an error message generated by a proxy server. It got an invalid response from the original server.

Status Codes (13)

- 5xx: Server Error, continued:
  - ◇ 503: Service Unavailable.

    E.g. the server is overloaded or currently not available because of maintainance work. Status code 503 means that one can try the request again after some time. A Retry-After header might contain a suggestion when to try it again.
  - ◇ 504: Gateway timed out.

    This is an error message from a proxy server. It got no answer from the original server. (or the DNS lookup timed out, etc.).
  - ◇ 505: HTTP Version not supported.

Response Headers (1)

- `Accept-Ranges`: The server may use this in order to show whether it can process requests for partial entities.

  The value of this header can be `bytes` or `none`. However, clients can request a partial entity with the `Range` header even if they did not yet receive an `Accept-Ranges` from the server.

- `Age`: Age of the response in seconds.

  A proxy should add this header to show how old the response is.

- `ETag`: Unique identifer of this version.

  E.g. a proxy may use this to decide whether the version of the web page it has in its cache is still current.

Response Headers (2)

- `Location`: URI of the entity (e.g. in case of a redirection).
- `Proxy-Autheticate`: The proxy requires a password etc.

  The header contains information about the method of authentication.
- `Retry-After`: Information when this request should be tried again.

  The contents of this header field can contain an integer (number of seconds) or a date and time.

Response Headers (3)

- `Server`: Information about the server software.
- `Vary`: Criteria used in the content negotiation.

  If there are several variants for the same URI, a proxy must know which `Accept`-headers were used in the selection, see below.

- `WWW-Authenticate`: Authentication method for protected pages (see below).

  This method must be included in a response with status code `401` ("Unauthorized").

Entity Headers (1)

- `Allow`: Methods that are applicable to this entity.

  Required for resonses with status code 405 “Method Not Allowed”.

- `Content-Encoding`: Encoding (compression) of the delivered entity, e.g. “`gzip`” (media type modifier).
- `Content-Language`: Language the intended audience of the document should speak, e.g. “`en`” (English).
- `Content-Type`: Media type of the delivered entity, e.g. “`text/html`”.
- `Content-Length`: Length of the data in bytes.

Entity Headers (2)

- Content-Location: URI of the delivered entity.

  E.g. if the requested URI denotes a set of variants (GIF, JPEG, etc.), this might be the URI of the selected variant. The URI specified in the Content-Location header is also used for the completion of relative URIs in the document.

- Content-MD5: Checksum of the data.
- Content-Range: For transmission of partial entities.
- Expires: Date and time until which this request can be used by proxies (see below).
- Last-Modified: Date and time of the last modification of the document.

Transmission of Entities (1)

- Normally the header "`Content-Length`" is used to tell the recipient the number of data bytes in the body of the request or response.

  In this way the recipient knows when the body is complete.

- However, it might happen that the sender does not know the size of the data beforehand (e.g. output of a program) and does not want to buffer them.

  In HTTP/1.0, the only possibility was to compute the complete response, store it in a temporary file, and transmit the data only after the file size was known. This is inefficient and must be used with great care (can the disk get full?).

Transmission of Entities (2)

- Therefore, the “chunked encoding” was introduced in HTTP/1.1: The data is sent as a sequence of pieces (chunks).
- Each piece begins with a line that contains the size of the piece in bytes (in hexadecimal notation).
- After this line (i.e. after carriage return and line feed) this number of data bytes follow.
- Then the next piece follows (length, data) etc.
- The end is marked with a piece of length 0.

Transmission of Entities (3)

- After the last piece, additional headers can be sent in a trailer. The trailer ends with an empty line.

  The empty line at the end is required in the chunked encoding, the use of headers in the trailer is optional. The header "`Trailer`" contains the names of the headers that will be contained in the trailer. The client can tell the server with "`TE: trailers`" that it can process headers in the trailer. Not all HTTP/1.1 clients must be able to process non-empty trailers. If the server does not know whether the client understands headers in the trailer, it may still send them, but the client is allowed to ignore them. Otherwise, all HTTP/1.1 clients must understand the chunked encoding (only trailers are optional).

- If the data are sent piecewise in this way, the header "`Transfer-Encoding: chunked`" must be specified.

Overview

1. Requests and Responses

2. Content Negotiation

3. Access Control/Password-Protected Pages

4. Caching (Proxies)

5. State Management (Cookies)

Media Types (1)

- HTTP can not only be used for transmitting HTML documents, but also for arbitrary binary data.
- However, the browser must know what to do with the data (how to interpret/display them).
- Therefore, the header `Content-Type` contains the media type of the data sent in the body.

  The standard specifies that any HTTP/1.1 message that contains a body should contain a `Content-Type` header. If this header is missing, the client is allowed to guess or treat the body as unknown binary data (`application/octet-stream`). Often the file extension in the URL helps, but e.g. `.pl` can be Perl or Prolog. Also, the URL might denote a program that computes the data, the data can then be of any type.

Media Types (2)

- Media types were introduced in the MIME standards (“Multipurpose Internet Mail Extensions”).

  RFC 1590: Media Types Registration Procedure.
  RFC 2045: MIME, Part One: Format of Internet Mail Bodies.
  RFC 2046: MIME, Part Two: Media Types.
  See also RFC 2047 to 2049.

- Media types consist of a general class and a subtype, e.g. `image/gif`.
- If the client does not know the subtype, it might guess from the class what to do with the data.

Media Types (3)

- E.g., all `text/*` types should be such that the client can show them directly to the user if it does not understand the subtype.

  E.g. `text/postscript` is wrong, it must be `application/postscript`.

- Besides class and subtype, also optional parameters can be specified (separated by ";"), e.g.

```
text/html; charset=ISO-8859-4.
```

Media Types (4)

- The currently defined classes are:
  - ◇ text, e.g. text/plain, text/html, text/xml.
  - ◇ multipart, e.g. multipart/mixed.
  - ◇ message, e.g. message/rfc822, message/news.
  - ◇ application, e.g. application/octet-stream, application/postscript, application/pdf.
  - ◇ image, e.g. image/jpeg, image/gif, image/png.
  - ◇ audio, e.g. audio/basic, audio/mpeg.
  - ◇ video, e.g. video/mpeg, video/quicktime.
  - ◇ model, e.g. model/vrml.

Media Types (5)

- Media types are registered by the IANA (Internet Assigned Numbers Authority) [http://www.iana.org].
- The current list of media types is available at [ftp://ftp.isi.edu/in-notes/iana/assignments/media-types]
- Non-registered media types should start with "x-".
- In Netscape (under UNIX), one can specify under

  Edit → Preferences → Navigator → Applications

  what to do with the different media types.

Media Types (6)

- E.g. one can specify that if a postscript file is received, it is stored in a temporary file and the ghostscript viewer is automatically started.
- It is also possible to extend the list of media types and to specify rules for guessing the media type from the file extension.
- Internet Explorer (under Windows) uses the Windows settings for file types.

  See `Tools` → `Folder Options` → `File Types` in any Windows Explorer window. Internet Explorer is also available under UNIX, there it is specified in `Tools` → `Internet Options` → `Associations`.

Alternative Versions (1)

- The document denoted by the URL/URI might exist on the server in different formats.
- E.g. Plain ASCII, HTML, LaTeX, Postscript, PDF.

  A URI does not necessarily denote a unique file. Normally, the format can be derived from the file extension in the URL. However, the file does not necessarily have an extension, or its meaning may be not unique. Only the `Content-Type` header clearly specifies the data format.

- The server might also do certain transformations "on the fly" when specific formats are requested.

  E.g. the file is stored compressed (`gzip`) on the server. If the client does not understand this compression, the server can uncompress it and send the uncompressed version.

Alternative Versions (2)

- It is also possible that the document exists in several different languages (e.g. German and English), but the URI only denotes the document in general.

  E.g. it would be possible that the university homepage under the URI `http://www.uni-giessen.de/` is delivered in English or in German depending on the preferences of the client.

- The user of the HTTP client (browser) normally has preferences for certain languages. These can be specified in the request that is sent to the server.

Alternative Versions (3)

- Preferences can also be defined on the server, based e.g. on the relative quality of the different versions.

  E.g. the original of the document is in French, the translation to English is also good, but for German there is only an automatic translation done by a program. If the client has no preferences between French and English, he/she gets the French original. The German version is only delivered if the user understands neither French nor English.

- E.g. in the Apache server (with option `Multiviews`), one can store files `doc.html.en` and `doc.html.de`.
- If the client requests the non-existent file `doc.html`, Apache selects one of the two language versions.

Alternative Versions (4)

- If a document exists in different versions, there is normally
  - ◇ one general URI where the content negotiation between client and server is done, and
  - ◇ one URI for each specific format/version (so that the client can do the selection manually).
- When content negotiation is done, normally the header `Content-Location` should be sent which contains the URI of the selected version.

  This is also important for proxies, see below.

Alternative Versions (5)

- Apache also has "Type Maps".

  The URI points to a file that decribes the different versions.

- E.g. under `doc.var` the following can be stored:

```
URI: doc.html.en
Content-Type: text/html; qs=1
Content-Language: en
Description: "English Original"

URI: doc.html.de
Content-Type: text/html; qs=0.8
Content-Language: de
Description: "Deutsche Übersetzung"
```

Alternative Versions (6)

- The version selection described above is called “Server-driven Content Negotiation”.
- It has certain disadvantages:
  - ◇ There are seldom several variants, but the client must always send its preferences.
  - ◇ The selection criteria of the client can not always be described with the HTTP headers.
  - ◇ Caching is getting more complicated.

Alternative Versions (7)

- Disadvantages of server-driven content negotiation, continued:
  - ◇ It might be considered a privacy violation that the user must tell the server his/her selection criteria and display possibilities.
- Therefore, work is done on
  - ◇ "Agent-driven Negotiation" (at the Client)
  - ◇ "Transparent Negotiation" (on a proxy).
- However, the proposals are not yet complete.

Media Type Selection (1)

- The client can define which media types it can process. This is done with the request header "`Accept`".

  If the requested resource is not available in one of these media types, the server can answer with the error code 406, but it may also ignore the media type specification and send whatever it has.

- The value of the `Accept`-header is a list of media types (separated with commas ",").

  One can also specify parameters for the media types, e.g. `text/html; level=2`. Note that the semicolon binds more strongly in this case than the comma.

- One can also use wildcards, e.g. `text/*` or `*/*`.

Media Type Selection (2)

- The client can specify preferences between the possible media types (in case the resource is available in different formats).
- This is done by means of quality factors which can be between 0 (not usable) and 1 (perfect).

  The numbers can have up to three digits after the decimal point. If a media type is written without quality factor, the default 1 is assumed.
- E.g. this header would specify that the client prefers HTML to PDF:

```
Accept: text/html;q=1, application/pdf;q=0.5
```

Media Type Selection (3)

- The syntax for parameters of a media type and for quality factors is the same, but media type parameters must be specified first:

```
Accept: text/html ;level=1 ;q=1,
        text/html ;level=2 ;q=0.9
```

- For each media type, the most specific case is used:

```
Accept: text/html;level=1;q=1, text/html;q=0.9,
        text/*;q=0.8, */*;q=0.7
```

- Then e.g. `text/plain` has the quality factor `0.8` and `image/gif` has the quality factor `0.7`.

Media Type Selection (4)

- The standard does not specify how the server selects a variant of the resource based on the quality factors.
- E.g. in the Apache type maps, one can specify server-side quality factors for the variants of a resource (`qs`).
- If e.g. the PDF version contains more formatting than the HTML version, the PDF version might have a slightly higher quality factor on the server.

Media Type Selection (5)

- The Apache server multiplies its own quality factors with those of the client and delivers the variant with the highest product.

  E.g. if the client has a strong preference for HTML, it gets HTML, otherwise PDF.

- Example:

Variant	Client	Server	Product	Selected
HTML	1.0	0.9	0.9	yes
PDF	0.5	1.0	0.5	no

Language Selection (1)

- The client can specify preferences for languages:

```
Accept-Language: de, en-US;q=0.8, en;q=0.7,
                 fr;q=0.3, *;q=0.1
```

- The syntax is similar to the media types, but languages have no parameters.

  However, there is an optional specification of the region. Capitalization in language names is not important.

- The Apache server first determines the best media type. Only if there are still several variants, the language is considered.

Language Selection (2)

- In the response, the server can define with the `Content-Language` header the language of the document.
- This might e.g. be important for search engines, which try to select only documents of a language the user knows.

  But they must use also other methods for determining the document language, since many servers do not send a `Content-Language` header. There are various ways to specify the language in a HTML document, e.g. some servers might evaluate the `HTTP-EQUIV` meta tag which permits to specify arbitrary HTTP headers in the document itself (see chapter 6).

Character Set Selection (1)

- The client can specify which character sets (encodings) it can display:

```
Accept-Charset: ISO-8859-1, ISO-8859-5;q=0.8
```

- E.g. for cyrillic letters, there are several different possible encodings (`ISO-8859-5`, `windows-1251`).
- The server could in principle translate between different character encodings.

  The file is stored in one specific encoding on the server, but if the client requests a different encoding (that is known to the server), the server can deliver the translated file contents.

Character Set Selection (2)

- In the response, there is no extra header to specify the character set encoding. This information is appended as a parameter to the media type:

```
Content-Type: text/html; charset=windows-1251
```

- The default (no parameter) is `ISO-8859-1`.

  However, the W3C HTML validator now requires that the character set is explicitly specified. This can e.g. be done with

```
<META HTTP-EQUIV="Content-Type"
      CONTENT="text/html; charset=iso-8859-1">
```

  in the document head (see Chapter 6).

Compression Method (1)

- The client can specify which compression methods it understands:

```
Accept-Encoding: gzip;q=1, identity;q=0.5
```

- Formats mentioned in the HTTP specification are:
  - ◇ `gzip` (earlier `x-gzip`): GNU gzip (see RFC 1952).
  - ◇ `compress` (earlier `x-compress`): UNIX compress.
  - ◇ `deflate`: See RFC 1950 and RFC 1951.
  - ◇ `identity`: No compression.

Compression Method (2)

- E.g. `gzip` compresses Postscript files often to less than half of their size, which reduces the download time, the server load, and the required part of the network bandwidth/transfer volume.

  Modems also do a compression, which is not effective on already compressed files. But this is only done over the modem connection. The server load and the data transfer volume over the main part of the internet is not affected by the modem compression.

- Vice versa, the server specifies the compression it has applied to the resource (if not identity):

```
Content-Encoding: gzip
```

Overview

1. Requests and Responses
2. Content Negotiation
3. Access Control/Password-Protected Pages
4. Caching (Proxies)
5. State Management (Cookies)

Restriction by IP-Number (1)

- The web server knows the IP number of the client machine (and then the name can normally be determined via a DNS query).
- Web servers can be configured such that they permit or reject access to certain directories depending on the IP-number of the client.

  E.g. some professors restrict access to course materials to the University net. E.g. the ACM digital library can be accessed from our university net, because the university has paid for it. This does not work for university members connecting from home via a general ISP. In this case, one must use the university modem pool.

Restriction by IP-Number (2)

- E.g. in the Apache server, one can specify in the following way that access is permitted only from the computer science subnet 134.176.28.*:

```
order deny,allow
deny from all
allow from 134.176.28
```

- "order deny,allow" means that allow-specifications have a higher priority than deny-specifications.

  Together with "deny from all", this means that all requests are rejected that do not originate from one of the explicitly allowed machines. "order allow,deny" can be used to exclude only certain machines (e.g. a robot).

Restriction by IP-Number (3)

- One can specify also a list of machines in the `allow` and `deny` commands (or use several commands):

```
allow from 134.176.28.10 134.176.28.11
```

- One can also use symbolic names:

```
allow from informatik.uni-giessen.de
```

- Networks can be defined via the relevant bits in the IP number:

```
allow from 134.176.28.0/24
```

Restriction by IP-Number (4)

- These settings can be included in a central configuration file of the Apache server.
- However, the configuration can also be done in files `.htaccess` that are stored in the directories which contain the web pages to be protected.

  The files `.htaccess` can contain in principle nearly arbitrary configuration settings for the Apache server, not only `allow` and `deny`. However, with the directive `AllowOverride` the administrator can specify in the central configuration files which directives are permitted in the `.htaccess` files. Of course, also the name `.htaccess` is configurable.

Restriction by IP-Number (5)

- The settings in `.htaccess` apply to all files in the directory including all subdirectories in which they are not overridden.
- All the above configuration information is specific to the Apache server.

Restricted Users (1)

- It is also possible to restrict web page access to users that can identify themselves (e.g. with username and password).
- The web server returns the status `401` "Unauthorized" when one tries to access a protected page.
- In addition, it specifies the required authentication method in the header `WWW-Authenticate`.

  The HTTP protocol is not restricted to a specific method for authenticating users. RFC 2617 defines the methods "`Basic`" and "`Digest`". "`Basic`" is most often used. However, it is unsafe because it transfers username and password without encryption.

Restricted Users (2)

- E.g. if one tries to access

  `http://hopper.computer.org/reviews/computer.nsf/$Searchform`

  one gets the response:

```
HTTP/1.1 401 Unauthorized Exception
Server: Lotus-Domino/Release-4.6.4
Date: Thu, 07 Dec 2000 17:57:25 GMT
Connection: close
Content-Type: text/html
Content-Length: 193
WWW-Authenticate: basic realm="/reviews"

<HTML><HEAD><TITLE>Error</TITLE> ...
```

Restricted Users (3)

- The parameter `realm` is used in order to distinguish several protected areas on the same server.
- The browser normally does not show this response but opens instead a dialog box for entering username and password.

  Click on "cancel" in the dialog box to see (the body of) this response.
- Then the browser sends a second request for the same URL/URI, which now includes this header:

```
Authorization: basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Restricted Users (4)

- The data string in the `Authorization` header is the Base-64 encoding of username, “:”, password, e.g.

```
Aladdin:open sesame
```

- The Base-64 encoding translates groups of three input bytes each into four printable characters.
- Each result character encodes 6 data bits (i.e. numbers from 0 to 63) according to the following table:

0	···	25	26	···	51	52	···	61	62	63
A	···	Z	a	···	z	0	···	9	+	/

Restricted Users (5)

- If the last group contains less than three input bytes, zero bytes are added. Characters generated completely from the added bytes are printed as “=”.

  If there are no “=” at the end, the last group was complete.
  If there is one “= at the end, the last group had only two data bytes.
  If there are two “= at the end, the last group had only one data byte.

- This means that from the encoded string, one can easily get back username and password.

  If somebody listens e.g. on an Ethernet, he/she might be able to see all passwords sent from browsers on the local net (or to servers on the local net).

Restricted Users (6)

- The authentication method "Digest" avoids the security problems of the "basic" method.

  It is, however, not yet very often used.

- The server sends in the `WWW-Authenticate` "challenge" a random number.

- In the "`Authorization`" header, the client sends the MD5-checksum of username, password, the random number, the HTTP method, and the URI.

  The password itself is not sent. The client proves only that it knows the password. Because method and URI are included, a hacker with access to a router cannot modify an intercepted request.

Restricted Users (7)

- Often not only a single URI is password-protected, but an entire group of web pages.
- Browsers remember username and password and use it automatically if they get a `WWW-Authenticate` header with the same realm from the same server.

  Normally the browser does not store the password on the disk (that would be an important security problem: Other people might get access to the computer, trojan horses might transmit it). If one starts the browser again, one must again enter username and password.
- The requested data are transfered in clear text.

  For really secret data, use an SSL connection.

Restricted Users (8)

- For the Apache web server, one can store a user list in a file which is managed with the program htpassword. (Alternative: user list in database.)
- The configuration can e.g. look as follows:

```
<Location /confidential>
AuthName "Confidential Documents Realm"
AuthType Basic
AuthUserFile /usr/local/apache/auth/password
AuthGroupFile /usr/local/apache/auth/groups
require user1 user2 group1
AuthAuthoritative on
</Location>
```

Password Fields in Forms

- Also in web forms, often a username and a password is requested. In this case, the data are processed by programs that run on the server.
- This is a different mechanism that the HTTP authentication.

  In this case, HTTP only transfers the requested data. It depends on the application what it does with them.
- The form data should be transfered encrypted (via an SSL connection) if they contain passwords or credit card numbers.

Overview

1. Requests and Responses

2. Content Negotiation

3. Access Control/Password-Protected Pages

4. Caching (Proxies)

5. State Management (Cookies)

Caching: Overview (1)

- Goals:
  - ◇ Reduction of the server load.
  - ◇ Reduction of the network load (used bandwidth).
  - ◇ Faster answers on the client.
- Observation:
  - ◇ Relatively few pages get a relatively large percentage of all requests (uneven distribution).
  - ◇ After a page was visited, it is often visited again after a short time.

Caching: Overview (2)

- A cache accepts the request from the client and answers it. I.e. for the client, it looks like a server.

  However, the complete URI must be specified in the request line.

- If the cache cannot answer the request itself, it forwards it to the server (or another cache). For the server, it looks like a client.
- Proxy servers (see above) are typical HTTP caches.

  Many universities and ISPs have a proxy server in order to reduce the traffic to other networks. Clients should be configured to use it.

- Browsers have a built-in cache.

Caching: Overview (3)

- The cache stores responses it gets from the server for some time.
- If there is another request for the same resource, the cache delivers the stored copy of the response.
- The client should get from the cache the same answer as it would have gotten from the original server ("Semantical Transparency").

  HTTP/1.1 defines the header `Warning`, which can be used by a proxy to state that there is a possible violation of the semantical transparency. E.g. if the origin server is not reachable, but the proxy still has an old copy, it can send it with the warning `111` "Revalidation failed".

Caching: Overview (4)

- HTTP has two important mechanisms to ensure the semantical transparency:
  - ◇ The server should define a minimum lifespan (expiration time) for the response.

    During this time, the cache can assume that the response has not changed and deliver the buffered version without asking the origin server.
  - ◇ The possibility to validate an earlier response with the origin server ("Is this still current?").

    In the positive case, the validation transmits significantly less data than the complete new response would have transmitted.

Exclusion of Caching (1)

- A cache may only store responses with status codes 200, 203, 206, 300, 301, or 410 (unless the server explicitly states that the response is cachable).

  Error messages are normally not buffered.

- Normally, only responses to `GET` and `HEAD` requests may be buffered.

  The information contained in responses to `HEAD` requests can be used for validating cache entries. `POST` requests may only be buffered if the server has explicitly stated that it is cachable. For the methods `OPTIONS`, `PUT`, `DELETE`, and `TRACE` caching is forbidden.

Exclusion of Caching (2)

- HTTP/1.0 servers send the following header to exclude caching of a response:

```
Pragma: no-cache
```

- HTTP/1.1 servers use this header:

```
Cache-Control: no-cache
```

  For compatibility reasons, they often send both headers.

- The header `Cache-Control` contains a list of instructions to the cache, separated by commas.

Exclusion of Caching (3)

- The above headers only forbid to use a cached response without revalidation.

  I.e. the response is immediately expired, but not useless.

- The following header forbids to store the response on any persistent media (e.g. disks):

```
Cache-Control: no-store
```

- This is useful for confidential information.

  E.g. a superuser on the machine might be able to look at the contents of the disk cache, and it might also be stored on backup tapes.
  The header does not prevent the user to explicitly store the document (with "Save As").

Exclusion of Caching (4)

- The server can also state that the response may only be buffered for this single user (e.g. in the browser cache, but not on a public proxy server):

```
Cache-Control: private
```

- The private caching is also the default for responses to requests that contain an `Authorization` header.

Expiration Model (1)

- The server can specify an expiration date in the response:

  ```
  Expires: Sun, 24 Dec 2000 18:00:00 GMT
  ```

- This header defines until which date and time the cache can deliver the response without revalidation with the origin server.

  If the expiration date is in the past (or the `Expires` header and the `Date` header contain the same value), the cache will ask the origin server each time it uses the response whether the page has changed.

- The server guarantees that until this date, the resource (web page) will not significantly change.

Expiration Model (2)

- The `Expires`-header does not mean that the page will certainly change at the given date.
- The `Expires` header also does not mean that the browser should automatically reload the page.

  `Expires` is only an instruction for the cache.
- Expiration dates should not be more than one year in the future.

  The standard suggests that if a resource never expires one should use an expiration date approximately one year in the future.

Expiration Model (3)

- `Expires` was already contained in HTTP/1.0.
- In the HTTP/1.1 header `Cache-Control`, one can alternatively specify a maximal age in seconds:

```
Cache-Control: max-age=86400
```

The variant `s-maxage` applies only to "shared caches" (Proxies). It also implies `proxy-revalidate` (see below).

- HTTP/1.1 caches also must specify the age of the response if it taken from their buffer:

```
Age: 3600
```

Expiration Model (4)

- Caches can be configured such that they use expired responses and clients can request them.
- Therefore the server can explicitly specify that the cache must respect the expiration time:

  `Cache-Control: must-revalidate`

  `proxy-revalidate` applies only to “shared caches”.
- If the server does not specify an expiration time, the cache can heuristically estimate it.

  E.g. if the date of last modification is long in the past, one can assume that the page is not likely to change soon. The guessed expiration time should not be more than 24 hours after the response was generated.

Expiration Model (5)

- In the Apache server, the module `mod_expires` generates the `Expires` header.
- E.g. this would state that web pages normally expire after one day, but GIF images after 1 week, and everything in the archive directory after 4 weeks:

```
ExpiresActive on
ExpiresDefault A86400
ExpiresByType image/gif A604800
<Directory /www/archive>
    ExpiresDefault A2419200
</Directory>
```

Expiration Model (6)

- In the above Apache configuration, the expiration time is defined in seconds from the time of access.

  I.e. from the current time when the response is computed. The letter A means "Access". If one uses the letter M, the time counts from the date of last modification of the file. This is useful e.g. if one knows that the files is modified every 24 hours.

- With the Apache module mod_headers one can set arbitrary headers, e.g.

```
<Directory /www/news>
    Header add Cache-Control: no-cache
</Directory>
```

Client Cache Instructions (1)

- The client can declare that it accepts responses that have already expired (e.g. up to one day ago):

```
Cache-Control: max-stale=86400
```

- Vice versa, the client can request that the response will not expire for a certain time (e.g. 1 hour):

```
Cache-Control: min-fresh=3600
```

- The `Cache-Control` instructions `no-cache`, `no-store`, `max-age=...` can also be used by the client.

Client Cache Instructions (2)

- The client can request that the origin server is asked, and it does not get a cached copy:

  ```
  Cache-Control: max-age=0
  ```

  Or, in HTTP/1.0: `Pragma: no-cache`

  In Netscape, this happens if one presses Shift while clicking on the Reload button.

- The client can also state that it is interested only in responses from the cache:

  ```
  Cache-Control: only-if-cached
  ```

Validating Cache Entries (1)

- If the response has expired according to the date specified with `Expires` or `max-age`, the cache does not have to delete the response.
- However, in order to reuse the response, it must check back with the origin server whether the response is still valid.
- This is usually done by sending a conditional `GET`-request to the server.

  It would be possible to first send a `HEAD` request, and decide based on the result whether the resource has changed. But in this case, two requests are needed. This is avoided with a conditional request.

Validating Cache Entries (2)

- E.g. if the response stored in the cache contains the header

  ```
  Last-Modified: Thu, 14 Dec 2000 13:20:00 GMT
  ```

  the cache can include the following header in the request:

  ```
  If-Modified-Since: Thu, 14 Dec 2000 13:20:00 GMT
  ```

Validating Cache Entries (3)

- This makes the request conditional:
  - ◇ If the files has not changed since the given date and the status code would be 200 (Ok), the server sends the status code 304 “Not Modified”, but does not include the contents of the file.

    The server may include headers, e.g. a new `Expires` header. The cache must replace the corresponding headers in its copy by these new headers.
  - ◇ If the file has changed (or the status code would not be 200 “Ok”), the server sends the complete response.

Validating Cache Entries (4)

- HTTP/1.1 has introduced "Entity Tag Validators", which replace the date of last modification.

  If a resource can change more than once per second, `Last-Modified` is not reliable. In addition, the date of last modification is not always known. If there are several variants of a resource (content negotiation), the date of last modification does not uniquely identify the entity in the response.

- The server now sends in the header `ETag` an arbitrary string (enclosed in quotes):

```
ETag: "60304-46b-39168772"
```

- This tag identifies a unique version of the resource.

Validating Cache Entries (5)

- If the resource changes, its ETag must change.

  The data must be uniquely identified by URI and ETag. The same ETag value may be used for files under different URIs. E.g. the date of last modification can be used as ETag, if there are no variants.

- Again, there are conditional requests:

```
If-None-Match: "60304-46b-39168772"
```

  If one would get this entity tag with the corresponding unconditional GET-request, the server does not execute the request. Instead the server returns 304 “Not Modified” if the method was GET or HEAD. Also requests with other methods can be made conditional in this way, then the server returns 412 “Precondition Failed” if the condition is false (i.e. GET for the same URI would not return this tag).

Validating Cache Entries (6)

- Normal ("strong") entity tags must change if any bit of the resource changes.
- Weak entity tags (`W/"..."`) only have to change if the meaning of the entity changes.

  E.g. the exact value of a counter for the number of access to a web page might not be important, only the order of magnitude. So the server could decide to use a weak entity tag for the image that includes the counter value, and update the entity tag only every day or every 500 accesses. In this way, the image can be delivered from the proxy storage, and does not always have to be fetched from the origin server. Of course, one would declare it as immediately expired, so that the proxy must use a conditional request to the origin server to validate its copy. The origin server can then count these requests.

Further Conditional Requests

- Suppose the client got the first 1000 bytes of the resource, but then the user pressed "Stop" or the connection was terminated. Requesting the remaining bytes is only useful if the resource has not changed in the meantime:

  ```
  Range: bytes 1000-
  If-Range: "60304-46b-39168772"
  ```

  If it has changed, it is completely retransmitted.

- `If-Match`/`If-Unmodified-Since` can e.g. be used for `PUT`-requests, in order to avoid lost updates.

Caching and Variants (1)

- If under one URI several variants are stored, the response cannot always be used for future requests for the same URI.

  The server must tell the cache in some way that content negotiation takes place or make the response immediately expired. Otherwise the cache will deliver the response for requests for the same URI without checking back with the client.

- For HTTP/1.0 clients, the Apache server sends

  ```
  pragma: no-cache
  ```

  if there are several variants for the requested URI.

Caching and Variants (2)

- In HTTP/1.1, a new header "`Vary`" was introduced. The server can now specify which headers were important for the selection of a resource variant:

  `Vary: Accept, Accept-Language`

- Then the cache may use the response for future requests in which the headers `Accept`, `Accept-Language` have the same value as in the current request.

- "`Vary: *`" means that the cache must always ask the origin server.

Caching and Variants (3)

- If a variant selection is done, a cache might have to store several responses for the same URI.

  Since a new response for the same URI does now not always overwrite the existing response, it becomes nontrivial when exactly a response should be deleted from the cache: A cache should delete a response from the cache if it gets a response for the same URI with the same `Content-Location`, a different `ETag`, and a more recent `Date`. If the new response has a different `Content-Location`, the old response does not have to be deleted.

Caching and Variants (4)

- Since the cache now might have several different responses for the same URI, it can include all their entity tags in a conditional request:

```
If-None-Match: "60304-46b-39168772",
               "12345-67c-98765432"
```

- In the answer, the server includes the `ETag` of the selected variant. The data is only sent if it has an `ETag` value that is not included in the above list.

  In this way, the content negotiation can still be done on the server, but the data are delivered from the cache.

Overview

1. Requests and Responses
2. Content Negotiation
3. Access Control/Password-Protected Pages
4. Caching (Proxies)
5. State Management (Cookies)

Stateless Protocol

- HTTP is a stateless protocol: Each request is treated in isolation. There are no "sessions" with "login" and "logout".

  This reduces the server load: After it has answered a request, it can completely forget about it. In contrast, sessions would need some memory on the server for the entire duration of the session (which can be long) in order to store state information.

- But this means that we get back to the times of batch processing: The request must contain all necessary data, there are no "interactive programs".

  Except with Java/Javascript.

Cookies (1)

- However, in many online shops, one can put items into a “shopping cart”, and pay at the exit.
- Obviously, an entire series of requests is linked together on the server and treated like a session.
- This is normally done with “Cookies”, which are pieces of data that
  - ◇ the server sends to the client, and
  - ◇ the client then basically includes with all future requests to the same server.

Cookies (2)

- A cookie can e.g. contain a user number or session number.
- E.g. `telnet www.altavista.com 80`:

```
HTTP/1.0 200 OK
Date: Thu, 14 Dec 2000 16:12:20 GMT
Server: AV/1.0.1
MIME-Version: 1.0
Content-Length: 18713
Content-Type: text/html; CHARSET=ISO-8859-1
Set-Cookie: AV_USERKEY=AVSe36e6eef1b00004b0910ac0008d5f;
            expires=Tuesday, 31-Dec-2013 12:00:00 GMT;
            path=/; domain=.altavista.com;
```

Cookies (3)

- This means that the cookie should be sent to all web servers in the domain `.altavista.com` when accessing arbirary pages (`path=/`).
- The browser then sends the data with the header

  ```
  Cookie: AV_USERKEY=AVSe36e6eef1b00004b0910ac0008d5f;
  ```

- In this way the effort to keep state information is moved from the server to the client.

  But often the contents of a cookie is only a reference to state information that is actually kept on the server.

Cookies (4)

- Netscape stores cookies in `~/.netscape/cookies`.

  This file contains all information about cookies in clear text.

- Internet explorer stores them in `C:\Windows\Cookies`.

  The files in these directory have lines that are terminated only with a linefeed (as under UNIX). But one can look at them e.g. with Wordpad or the MS-DOS edit. They contain the name of the cookie, the contents of the cookie, the domain of the originating web server, and some additional data (e.g. the expiration time). If several cookies are stored in the same file, they are separated by a line containing only an asterisk.

Cookies (5)

- While the contents of a cookie often has a meaning only to the server that processes it, somebody who has access to these files can get a good impression which web pages were visited.

  Today, many servers send cookies, and the domain for which the cookie is intended is contained in the above files.

- Some cookies contain passwords which are then also contained in the cookie files of the browser.

  So somebody who has access to your PC or can copy your cookie files might be able to pretend that he/she is you for certain websites. E.g. "one click" purchases depend on a cookie.

Cookies (6)

- Cookies were invented by Netscape.
- A preliminary specification is contained in

  [http://www.netscape.com/newsref/std/cookie_spec.html]
- A newer specification is contained in RFC 2965. RFC 2964 treats privacy and security aspects.
- It is unclear why this name was chosen.

  One reference says that "cookie" is a computer science term for an opaque piece of data that a client (e.g. of a library) holds and adds to future calls. I have also heard that an Apple operating system had the notion of a cookie jar, originating from a real cookie jar that the programmers kept on top of their computer containing little notes.

Cookies (7)

- Browsers can be configured to ignore cookies.

  Then they do not store them and do not include them in future requests. E.g. under Netscape: Edit → Preferences → Advanced. IE: Tools → Internet Options → Security → Custom Level.

- Some online shops do not work without cookies.

  Often, unique numbers are also appended to URLs. However, this works only as long as the user does not leave the pages of the shop with the "back" button. Cookies are "more persistent". Also the server does not have to compute a different version of the web pages for each user if it uses cookies (which also makes proxies useless).

- One can delete cookies from time to time.

Privacy Problems (1)

- Cookies make it possible to identify a user even when this is not really needed, e.g.
  - ◇ An online bookstore sends a user number in a cookie when a user first accesses the website.
  - ◇ The browser sends this user ID back whenever it accesses pages of the bookstore in future.
  - ◇ When the user buys a book, the bookstore learns name and address for the user ID.
  - ◇ If the user later only looks at offers in the bookstore, he/she is already known by name.

Privacy Problems (2)

- In this way, the bookstore may show each user a different homepage which contains special offers for books similar to books the customer has bought earlier (“Personalization”).
- Search engines might use cookies to count the number of distinct users they have.

  As opposed to the total number of queries. This information is important for getting advertising customers. Also the search engine user number can be linked to all search terms the user has looked at, which can be used for putting advertisements on the web page that are interesting to the user. The search engine normally will not know name and address of the user, only a unique number (but see below).

Privacy Problems (3)

- A web advertizing agency can see in this way which pages with advertisements the user has looked at.

  This is helpful to show the user advertisements in which he/she is interested.

- If an agency has advertisements on many web pages, it can build up a quite detailed profile of interests of the user.

- This might be acceptable as long as the advertising agency does not know the name and the address of the user (only a unique number).

Privacy Problems (4)

- If the pages of the bookstore contain advertisements (maybe of the bookstore itself), the bookstore can pass its user ID to the advertizing agency.
- But if the corresponce between the user IDs of the bookstore and the advertizing agency is known, they can combine their data and know name and address together with a large number of visited web pages (that contain advertisements).

User Sessions (1)

- Although it is possible to implement something that looks like a user session with cookies (or unique numbers in URIs), one must be aware of certain differences.
- The server only "hears something" from the browser when one sends a request.

  It does not help to let a web page with a cheap flight offer in one's browser window open. If the server did not get a request from the user for a certain amount of time (e.g. 20 min), it terminates the "session".

User Sessions (2)

- It is important that session numbers are not assigned sequentially or in another way easily predictable. Then a hacker could easily take over the "session" of a customer.

  The hacker can send arbitrary values for cookies. E.g. he/she can open a session himself/herself, and then increment his/her session number by one.