HTTP/1.0

The successor of HTTP/0.9

What improvements does HTTP/1.0 bring with regard to HTTP/0.9? It first eases web surfing: it is able to work with **cache systems** (even if the mechanism stays quite simple). It is also possible to send data to a server (since the new **POST** method). HTTP/1.0 is also able to recignize when a request did not work (the famous "404 not Found" message). Finally it allows users to authenticate, for instance to access a hidden part of a web site.

Example of HTTP/1.0 request

Regarding HTTP/0.9, HTTP/1.0 brings a real innovation regarding the form of a request, and especially the form of the reply:

```
$ telnet www2.themanualpage.org 80
Trying...
Connected to www2.themanualpage.org.
Escape character is '^]'.
GET http://www2.themanualpage.org/http/hello.txt HTTP/1.0
User-Agent: Mozilla/4.03 [fr]

HTTP/1.1 200 OK
Date: Thu, 20 Jul 2000 06:43:02 GMT
Server: Apache/1.3.12 (Unix) PHP/3.0.9
Last-Modified: Mon, 17 Jul 2000 15:55:03 GMT
Content-Type: text/plain

Hello

Connection closed by foreign host.
```

We immediately notice that there is much more information in this request than in a HTTP/0.9 request. Let's first notice a "HTTP/1.0" at the end of the first line. This is used to tell the server that we would like to speak HTTP/1.0 with in for this request. It will be same with **HTTP/1.1** and also in all likelihood with any other new version of HTTP. In this example the server will actually reply in HTTP/1.1. This can happen...

The second (and very important) difference is that we also say what web browser we are using... We are adding data in the request! We will see later what other kind of data with what we call the **request's header**.

The third difference is also very important: the server says a lot of things (with **directives**) before saying what we are waiting for. Another header!

Finally, (at last!) we get the file (the so-called **entity**). Just straigh after the server

cuts the connection.

Requests

Presentation

An HTTP/1.0 request is made up of 3 elements: a method (to say what we are doing), immediately followed by headers (to specify the request and the possible data), an extra line break and finaly the possible entity's body (for example the content of a form). We will see later that the headers are also made up of 3 parts (the last part is optional and specifies the possible following data). In any case, we can sum up all this by the following POST request (the most complete type of request):

Methods

There are 3 kinds of requests in HTTP/1.0 (3 different methods): the **GET**, **HEAD** and **POST** methods.

The **GET** method is the same as in HTTP/0.9: it is used to retrieve the document specified by the URI.

The role of the **HEAD** method will be explain in the part dealing with **caches**. Shortly speaking, this method is used to only get the header part of a complete reply.

The **POST** method is actually much more interesting because it is the one used to send data to a server. It is now possible to use huge and complete forms in web pages. We send a certain amount of data to the program specified in the URL. This data is sent at the same time as the request in what we call the **body of the entity** (or of the request).

Let's speak now about a problem with HTTP/1.0: it is not able to handle URLs on a virtual server configuration. In concrete terms if we do a HTTP/1.0 request on a machine that hosts www.bar.com and foo.bar.com (this could happen) to simply retrieve the file called /index.html (the request is something like: `GET /index.html HTTP/1.0`), the server is not able to know if it should send /index.html from www.bar.com or foo.bar.com... Paths must be complete with HTTP/1.0 (`GET http://foo.bar.com/index.html HTTP/1.0`).

Headers

When we send a request to a server, one must at first glance send a header to specify what we are requesting. This header is actually optional: for backward compatibility reasons, we can directly finish a request by striking twice the enter

key just after the first line. Anyway, a header is divided into 3 specific parts:

- **generic header** that concerns the request or the reply,
- **request's header** that directly concerns the request itself,
- **entity's header** that concerns the possible data we send (meta information).

The **generic header** is mainly made up of the date and time at which we do the request (the directive "Date:").

We can specify 5 different things in the **request's header**:

directive	meaning
From:	says the e-mail address of the person who is using the web browser. This could pose problems of private life respect.
Referer:	URI of the object that refers to the current request (for instance it is the URL of the page that contains the link you have just clicked on)
User-Agent:	the web browser ID. Is used to adapt a reply to the kind of web browser.
Authorization:	used for authentification
If-Modified-Since:	used to make **conditional GET** requests

Finally, the optional **entity's header** deals with the optional data. The web browser adds it only when it sends data to a server with the POST method. The main directives used with it are the same as those used by the server:

directive	meaning
Content-Type:	the type of the data (MIME type: text/html, image/jpeg...)
Content-Length:	length (in byte) of the data
Content-Encoding:	used if the data has been coded or compressed (x-gzip for instance)

What is interesting is that we can send data different from the one contained in a form, files for example (this is how we can send attached document with web-based e-mail clients, like Hotmail).

The entity's body

There is a body only when one sends data with the **POST** method (see HTML forms to learn how to use it). In this case we must add the Content-Type and Content-Length directives in the headers of the request. Once all this is provided, we fill in the entity's body with the message to send. Just as a server does, a line break must stand between the request's header and the entity's body (see below).

Server's reply

Presentation

The HTTP/1.0 response of a server is similar to the HTTP/1.0 request showed above, except that the very first line is a **status**. The rest (headers and entity's body) works more or less the same way.

The status

Now, with HTTP/1.0, the web client is aware of the type of the reply, thanks to the reply's header. The very first line of the header part of the request says how the request was treated. The reply starts with something like:

```
HTTP/1.0 200 OK
```

There are 3 parts in this (small) line:

- the HTTP version used by the server for its reply,
- the numerical status of the reply,
- the human readable status of the reply.

The status exactly says how the request was treated. The most famous one is "404 Not Found". There are 5 classes of status:

- 1xx: not used in HTTP/1.0
- 2xx: request successfully treated (hence the "200 OK" status)
- 3xx: redirection. The request was not treated but the server knows where it could be fulfilled
- 4xx: wrong request (wrong syntax for instance, hencec the "404 Not Found" that means "you should learn how to type a correct URI!")
- 5xx: correct request but not satisfied (problem with the server, not implemented yet...)

HTTP/1.0 statuts: very useful to understand what happened.

Reply's header

Just like the web browser, the server says things quite useful things in this part. The reply's header is also splitted into 3 parts, the same 3 parts as those used in the client's request.

The **generic header** consists of these 2 fields:

directive	meaning
Date:	gives the date when the reply was sent
Pragma:	defines specific behaviours. The only behaviour defined on the RFC1945 is "no-cache", which means that the replied document must not be cached (see **cache management**).

The **reply's header** consists of 3 different fields:

directive	meaning
Location:	the absolute URI of a resource
Server:	contains information about the HTTP server that is replying (for instance: Apache/1.3.12)
WWW-Authenticate:	asks for the user to authenticate

The **entity's header** can contain these fields:

directive	meaning
Content-Type:	like the client, specifies the type of data the server is sending
Content-Length:	the length of the entity (in byte)

Content-Encoding:	tells if the data is encoded and how
Expires:	expiration date of the sent resource. Useful with **caches**.
Last-Modified:	last update date and time. Also used with caches.
Allow:	list of usable methods to access the resource (typically HEAD and GET)

It is also possible to use directives not defined by the RFC1945 (extension-header).

The entity's body

It is simply the result of the request (generally the HTML page). From time to time, there is no body, for instance when we have done a HEAD request. It is also the case when there is a "404 Not Found".

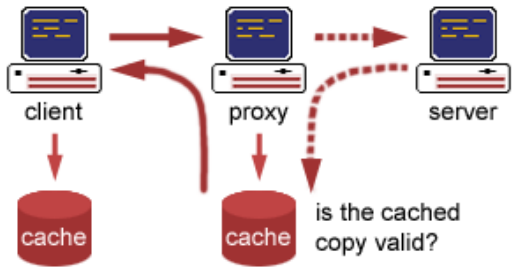
Only important point: the server must send a Content-Length if it sends a document.

Cache management

What is this?

A **cache** is a sort of "buffer" or temporary memory: somewhere a machine stores in its memory some documents, so that when a request is asking for an already cached document (document stored in the cache), we would rather send this document instead of making a complete (and long) request to the server. This is very interesting for very requested documents; this can reduce the network traffic.

The trick consists of knowing if the document has been updated. The following diagram shows the route followed by a resource when performing a request on it:



Actually, the client makes its request to a proxy server or any other cache system. Then this proxy is going to guess if the requested document (which is also in its cache) has been updated. To do so, it is going to use the directives it got when it asked for the document for the first time. If it knows nothing about the document, it may perform a **HEAD** request. According to the reply, the proxy is able to know if it should ask for a new version or simply send the version it has in its cache. Fortunately this latter case is the most frequent one, and therefore we save time and network load.

New directives

HTTP/1.0 introduces special directives, such as Date, Expires or Last-Modified to optimize cache systems; caches are then able to know when a cached document has been last updated, calculate how long a document should stay in the cache, or simply know when it must ask for a new version (Expires directive).

The method

The first idea to know is an entity has been update since we last asked for it is to use the new **HEAD** method.

This method is used to get the header part of a complet GET request on a given resource. This is very useful to get the last update date, and then to know if the document has changed or not. The key point is that asking for a header is shorter than asking for the complet document since there is no entity body.

Conditional GET requests

HTTP/1.0 also introduces another nice cache feature: a conditional GET request. This is performed (once again!) with a special directive: If-Modified-Since.

A conditional GET request will ask the server to send the document if and only if this document has changed since the specified date. For the client, it is just a matter of sending the date at which it last got the document. If the resource has not changed, the server just sends a very short "HTTP/1.0 304 Not Modified" and the client simply uses the version it already has.

Non cachable entities

Obviously, some document must not be cached (for instance results of CGI scripts that depend on the data the client sends). To specify a document must not be cached, we use the special directive called Pragma: no-cache that asks intermediate machines not to cache the document. In an HTML document, we can specify this directive in a special meta tag. The other solution is to use a Expires directive with an earlier date.

Authentication

The last innovation of HTTP/1.0 is user's authentication. Indeed it is a very simple mechanism, but it exists. This allows a protection of a specific part of a web site (only an authorized person may access it).

When a client trys to access a protected resource, the server sends a "HTTP/a.0 401 Authorization Required" status with a WWW-Authenticate directive that tells the client how to proceed for the authentication. At this time of the process, the client asks the user to enter a login/password and sends this to the server. Then, the server replys (or not) the requested resource.

Basically HTTP/1.0 offers only one way for authentication. An improvement of HTTP/1.0 and the HTTP/1.1 will offer another one more efficient. The standard authentication process is called **basic** authentication. It simply codes the "login:password" string with a base64 coding. It is so simple that we can say the login/password pair passes in clear on the network.

Restrictions

HTTP/1.0 does not solve all the problems we had with HTTP/0.9. Indeed, we still have the problem of multiple connections (the client opens several connection with the server at the same time), despite the new keep-alive directive (few machines understand it).

Furthermore HTTP/1.0 does not handle very well relative URLs. Indeed we must write the complete (absolute) URL when a server hosts several virtual servers.

The "Basic" authentication method is too basic! A new method was introduced. It is the "Digest" method, but we will speak about it in **HTTP/1.1**.

Fianlly, HTTP/1.0 manages caches in a quite simple way.

Conclusion: welcome to **HTTP/1.1**!

Reference

RFCs:

- **RFC1945**: HTTP/1.0
- **RFC2617**: Basic and Digest authentication methods

 **printable format**
