

RFC 2616 : Hypertext Transfer Protocol – HTTP/1.1

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 6 Juin 2007

Date de publication du RFC : Juin 1999

<http://www.bortzmeyer.org/2616.html>

Pas besoin de présenter HTTP, un des protocoles les plus importants de l'Internet, en raison du succès du Web. Notre RFC, malgré son âge, est la norme officielle de HTTP.

Le Web a beaucoup évolué depuis la sortie du RFC mais son protocole principal reste le même, HTTP 1.1. Contrairement aux normes portant sur le format des données, qui sont en général réalisées par le W3C, HTTP est une norme IETF. Le RFC est très long car, si HTTP est très simple dans son principe (ce qui a beaucoup aidé à son succès), il contient de nombreux détails (par exemple de nombreux en-têtes peuvent être transmis entre serveurs et clients pour des options variées) et je parie que peu de logiciels mettent en œuvre la totalité du RFC.

On notera aussi que, si HTTP est surtout connu pour permettre à un navigateur Web d'accéder à une page HTML, il peut aussi servir de protocole pour permettre à deux applications de communiquer, notamment en suivant l'architecture REST (voir <http://www.bortzmeyer.org/programmation-rest.html> et <http://www.bortzmeyer.org/signaler-a-signal-spam.html> pour des exemples REST). Ainsi, des protocoles comme IPP (RFC 2911¹) ou XCAP, "XML Configuration Access Protocol" (RFC 4827), utilisent HTTP (pas forcément en REST). Un RFC a même été écrit pour essayer de décrire dans quels cas il était souhaitable de réutiliser HTTP pour un protocole Internet, le RFC 3205.

Le protocole repose sur un mécanisme **requête / réponse** très simple (section 1.4). Le client se connecte en TCP au serveur, envoie une requête sur une seule ligne, la requête incluant un URI, éventuellement des options sous forme d'en-têtes analogues à ceux du courrier électronique (Accept: text/*), éventuellement un contenu, et reçoit une réponse sous forme d'un code numérique indiquant de manière non ambiguë le résultat de la requête, d'en-têtes optionnels et d'un corps, la ressource demandée.

1. Pour voir le RFC de numéro NNN, <http://www.ietf.org/rfc/rfcNNN.txt>, par exemple <http://www.ietf.org/rfc/rfc2911.txt>

Le terme de **ressource** est défini dans la section 1.3. Il est plus général que celui de page. Une ressource n'est pas forcément une page HTML, ce n'est même pas toujours un fichier.

Il existe plusieurs requêtes, définies en section 5.1.1. Les plus connues sont **GET** (récupérer une ressource, sans rien modifier), **POST** (modifier une ressource), **PUT** (créer une nouvelle ressource) et **DELETE** (comme son nom l'indique). Mais il faut noter que, si les applications REST, par définition, utilisent les différentes requêtes pour les différentes actions qu'elles peuvent effectuer, le navigateur Web typique ne connaît que GET et POST. Et beaucoup d'applications ne sont pas REST, utilisant par exemple le POST comme un fourre-tout, l'action étant exprimée par un paramètre du POST et pas par la requête. C'est dommage, car la sémantique de chaque requête est rigoureusement définie par le RFC (voir particulièrement la section 9.1), notamment pour ce qui concerne les caches, l'idempotence et les invariants comme $GET(PUT(x)) = x$ (en français : après un PUT, un GET du même URI doit donner la ressource transmise lors du PUT).

Les codes numériques de retour sont à trois chiffres, le premier chiffre indiquant la catégorie (2 est un succès, 4 une erreur dûe au client, 5 une erreur dûe au serveur, etc), comme avec SMTP. L'erreur la plus célèbre est sans doute la 404, « ressource non trouvée », décrite dans la section 10.4.5, qui se produit par exemple lorsqu'un webmestre n'a pas lu l'article "*Cool URIs don't change*" <<http://www.w3.org/Provider/Style/URI>> et a changé ses fichiers de place.

Il existe évidemment une pléthore de logiciels clients et serveurs HTTP, puisque les navigateurs Web et les serveurs comme Apache, parlent tous HTTP. Mais, pour apprendre le protocole, les clients en ligne de commande sont très pratiques. Ici, curl avec l'option `--verbose` permet de voir les requêtes envoyées par le client au serveur (elles sont précédées d'un signe `;`) et les réponses du serveur (précédées d'un signe `;`) :

```
% curl --verbose http://www.bortzmeyer.org/2616.html
* About to connect () to www.bortzmeyer.org port 80
*   Trying 80.67.170.20... connected
* Connected to www.bortzmeyer.org (80.67.170.20) port 80
> GET /2616.html HTTP/1.1
> User-Agent: curl/7.15.5 (i486-pc-linux-gnu) libcurl/7.15.5 OpenSSL/0.9.8c zlib/1.2.3 libidn/0.6.5
> Host: www.bortzmeyer.org
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Date: Wed, 06 Jun 2007 13:20:03 GMT
< Server: Apache/1.3.26 Ben-SSL/1.48 (Unix) Debian GNU/Linux AuthMySQL/3.1 mod_throttle/3.1.2 mod_watch/3.1
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=iso-8859-1
...
```

Ou bien, avec `wget`, son option `--debug`, et une ressource qui existe (dans le cas ci-dessus, le fichier n'avait pas encore été copié sur le serveur, d'où le résultat 404) :

```
% wget --debug http://www.bortzmeyer.org/1035.html
...
Connecting to www.bortzmeyer.org|80.67.170.20|:80... connected.
...
---request begin---
GET /1035.html HTTP/1.0
User-Agent: Wget/1.10.2
Accept: */*
```

<http://www.bortzmeyer.org/2616.html>

```

Host: www.bortzmeyer.org
Connection: Keep-Alive
---request end---
...
---response begin---
HTTP/1.1 200 OK
Date: Wed, 06 Jun 2007 13:23:48 GMT
Server: Apache/1.3.26 Ben-SSL/1.48 (Unix) Debian GNU/Linux AuthMySQL/3.1 mod_throttle/3.1.2 mod_watch/3.17
Last-Modified: Wed, 30 May 2007 18:21:58 GMT
ETag: "2e6e69-19a6-465dc0c6"
Accept-Ranges: bytes
Content-Length: 6566
Connection: close
Content-Type: text/html; charset=utf-8

```

Afficher les en-têtes du dialogue entre le client et le serveur peut aussi se faire, si on utilise le navigateur Firefox, avec son extension « HTTP headers <<http://livehttpheaders.mozdev.org/>> ».

Pour un exemple concret des options qui peuvent être passées au serveur via les en-têtes, prenons `If-Modified-Since` (section 14.25) qui permet au client d'annoncer au serveur « J'ai déjà une version de cette ressource, obtenue à telle date, ne me l'envoie que si tu as une version plus récente ». Par exemple, sur le site des étiquettes de langue <<http://www.langtag.net/>>, on a le registre dans divers formats. Ces formats sont produits localement, après avoir transféré le registre original. Celui-ci étant un gros fichier, on l'analyse pour trouver sa date et on utilise l'option `--time-cond` de `curl` pour générer un en-tête `If-Modified-Since`. Voici le script :

```

#!/bin/sh

MYURL=http://www.langtag.net/
LTR_URL=http://www.iana.org/assignments/language-subtag-registry
LTR_LOCAL=language-subtag-registry

if [ -e ${LTR_LOCAL} ]; then
    ltr_date='head -n 1 ${LTR_LOCAL} | cut -d" " -f2\'
    current_date='date +%Y%m%d" --date="${ltr_date} +1 day"\'
else
    # Trick to force a downloading
    current_date="19700101"
fi
curl --verbose --output ${LTR_LOCAL}.TMP \
    --compressed \
    --referer ${MYURL} \
    --proxy "" \
    --time-cond "${current_date}" \
    ${LTR_URL}
...

```

Cela donne ceci, si le fichier n'existe pas (on a pris une date arbitraire dans le passé) :

```

...
> If-Modified-Since: Thu, 01 Jan 1970 00:00:00 GMT
>
< HTTP/1.1 200 OK
< Last-Modified: Fri, 04 May 2007 18:36:16 GMT
< Content-Length: 80572

```

Si par contre le fichier existe et on a la version du 5 mars 2007, `curl` affiche :

<http://www.bortzmeyer.org/2616.html>

```
...  
> If-Modified-Since: Sat, 05 May 2007 00:00:00 GMT  
>  
< HTTP/1.1 304 Not Modified  
...
```

et le fichier n'est pas transféré.

Naturellement, si on veut développer un client HTTP, au delà de ce que permet curl (qui permet déjà beaucoup de choses), on dispose de nombreuses bibliothèques toutes faites comme `httplib` <<http://docs.python.org/lib/module-httplib.html>> pour Python, `libcurl` (avec le programme du même nom) ou `neon` <<http://www.webdav.org/neon/>> pour C, etc. HTTP est suffisamment simple pour qu'on puisse écrire rapidement un client, même sans ces bibliothèques comme le montre cet exemple en Python <<http://docs.python.org/lib/socket-example.html>>.

À noter que notre RFC utilise, pour décrire le protocole, une variante de l'ABNF standard, décrite en section 2.1, ce qui a suscité un certain nombre de problèmes.

Un travail de révision du RFC a commencé. La liste des questions ouverts est longue : voici celle maintenue par le W3C <<http://www.w3.org/Protocols/HTTP/1.1/rfc2616bis/issues/>> et celle de l'IETF <http://skrb.org/ietf/http_errata.html>. La charte du futur groupe de travail est en cours de discussion <<http://www1.ietf.org/mail-archive/web/discuss/current/msg00614.html>>, il semble qu'une bonne partie de l'insatisfaction vienne d'un autre document, le RFC 2617, qui spécifié l'authentification. L'autre grand débat étant le classique « Faut-il écrire un document en partant de zéro ou bien se contenter de corriger les erreurs les plus évidentes du RFC? ».