RFC 7230 : Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 14 juin 2014

Date de publication du RFC : Juin 2014

https://www.bortzmeyer.org/7230.html

Grande révision de la norme HTTP 1.1, désormais éclatée en huit RFC différents <https://www.bortzmeyer.org/http-11-reecrit.html>. Celui-ci décrit l'architecture générale du principal protocole du Web, HTTP, le format des URI `http:` et `https:`, et la syntaxe générale des messages. (Il a depuis été largement remplacé par les RFC 9110[1] et RFC 9112.)

HTTP, un des protocoles les plus célèbres de l'Internet, permet à des **clients** d'accéder à des **ressources** situées sur des **serveurs**. (Le terme de « ressource » a été choisi car il est abstrait : les ressources peuvent être des fichiers mais ce n'est pas forcément le cas.) HTTP est sans état, chaque requête est indépendante des autres et un serveur peut répondre à une requête sans forcément connaître la séquence des requêtes précédentes. Comme il est très générique, et ne suppose pas grand'chose sur les clients et les serveurs, HTTP peut être utilisé dans un grand nombre de contextes différents. Son utilisation par les navigateurs Web n'est donc qu'une seule possibilité. HTTP est utilisé, côté client, par des "*appliances*", des programmes non-interactifs (mise à jour du logiciel, par exemple), des applications tournant sur mobile et récupérant des données sans que l'utilisateur le voit, etc. De même, le modèle du serveur HTTP Apache tournant sur un serveur Unix dans un "*data center*" n'est qu'un seul modèle de serveur HTTP. On trouve de tels serveurs dans les caméras de vidéo-surveillance, les imprimantes, et bien d'autres systèmes. Il faut notamment se souvenir qu'il n'y a pas forcément un humain dans la boucle. C'est pourquoi certaines propositions d'évolution de HTTP qui nécessitaient une interaction avec un utilisateur humain, par exemple pour désambiguïser des noms de domaine, sont absurdes. Même chose pour les décisions de sécurité.

Il existe de nombreuses passerelles vers d'autres systèmes d'information. Un client HTTP peut donc, via une passerelle, accéder à des sources non-HTTP. D'une manière générale, HTTP étant un protocole,

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc9110.txt`

et pas une implémentation, le client ne sait pas comment le serveur a obtenu la ressource et où. Au tout début du Web, le seul mécanisme pour le serveur était de lire un fichier, mais ce n'est plus le cas depuis bien longtemps (d'où l'utilisation du terme « ressource » et pas « fichier » dans la norme). HTTP spécifie donc un comportement extérieur, pas ce qui se passe à l'intérieur de chaque machine.

La section 2 de notre RFC décrit l'architecture du World-Wide Web et notamment de HTTP. Ce dernier, on l'a vu, est un protocole requête/réponse, sans état. Un client interroge un serveur, au-dessus d'un protocole de transport fiable, TCP. Comme dans tout protocole client/serveur, le serveur attend passivement des requêtes et les traite lorsqu'elles arrivent. Les ressources sont identifiées par un URI (normalisés dans le RFC 3986). Le format des messages HTTP est du texte, comme avec bien d'autres protocoles TCP/IP, par exemple SMTP. Cela facilite l'écriture des programmes, et surtout leur débogage (messages tapés à la main, lecture des communications). À noter que la prochaine version de HTTP, HTTP 2, utilisera au contraire un encodage binaire. Ce format texte ressemble à bien des égards à l'IMF du RFC 5322, notamment pour la syntaxe des en-têtes (`Name: value`). HTTP emprunte aussi à MIME par exemple pour indiquer le type des ressources (texte, image, etc).

Le cas le plus simple en HTTP est la récupération d'une ressource par une requête `GET`. En voici un exemple, affiché par le client HTTP curl dont l'option `-v` permet de visualiser les requêtes et les réponses. Le client envoie la ligne `GET` suivie du chemin de la ressource sur le serveur, le serveur répond par une ligne de statut, commençant par le fameux code à trois chiffres (ici, 200). Client et serveur peuvent et, dans certains cas, doivent, ajouter des en-têtes précisant leur message :

```
% curl -v http://www.bortzmeyer.org/files/exemple-de-contenu.txt
...
> GET /files/exemple-de-contenu.txt HTTP/1.1
> User-Agent: curl/7.26.0
> Host: www.bortzmeyer.org
> Accept: */*
>
[Fin de la requête. La réponse suit]

< HTTP/1.1 200 OK
< Date: Thu, 29 May 2014 16:35:44 GMT
< Server: Apache/2.2.22 (Debian)
< Last-Modified: Fri, 11 Nov 2011 18:05:17 GMT
< ETag: "4149d-88-4b1795d0af140"
< Accept-Ranges: bytes
< Content-Length: 136
< Vary: Accept-Encoding
< Link: rel="license"; title="GFDL"; href="http://www.gnu.org/copyleft/fdl.html"
< Content-Type: text/plain; charset=UTF-8

[Fin des en-têtes, le contenu de la ressource suit]

C'est juste un exemple de texte ("contenu"), rien de particulier. Il
est uniquement en ASCII, pour contourner les histoires d'encodage.
```

Ceci était le cas le plus simple : HTTP permet des choses bien plus compliquées. Ici, pour une page en HTML avec davantage de champs dans la réponse :

```
% curl -v http://www.hackersrepublic.org/
...
> GET / HTTP/1.1
> User-Agent: curl/7.26.0
> Host: www.hackersrepublic.org
```

```
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: Apache/2.4.6
< X-Powered-By: PHP/5.4.4-14+deb7u9
< X-Drupal-Cache: HIT
< Content-Language: french
< X-Generator: Drupal 7 (http://drupal.org)
< Cache-Control: public, max-age=0
< Expires: Sun, 19 Nov 1978 05:00:00 GMT
< Etag: "1401374100-0-gzip"
< Last-Modified: Thu, 29 May 2014 14:35:00 GMT
< Content-Type: text/html; charset=utf-8
< Vary: Cookie,Accept-Encoding
< Transfer-Encoding: chunked
< Date: Thu, 29 May 2014 16:37:15 GMT
< Connection: keep-alive
< Via: 1.1 varnish
< Age: 0
<
...
<!DOCTYPE html>
<head>
<meta http-equiv="X-UA-Compatible" content="IE=Edge" />
<meta charset="utf-8" />
<link rel="apple-touch-icon-precomposed" href="http://www.hackersrepublic.org/sites/all/modules/touch_icons/appl
<link rel="apple-touch-icon" href="http://www.hackersrepublic.org/sites/all/modules/touch_icons/apple-touch-icon
<meta name="viewport" content="width=device-width" />
<meta name="Generator" content="Drupal 7 (http://drupal.org)" />
...
```

Une des complications possibles est la présence d'intermédiaires. HTTP permet des relais des passerelles et des tunnels. Le relais ("*proxy*") est du côté du client, souvent choisi par lui, et transmet les requêtes, après avoir appliqué certains traitements, comme le filtrage de la publicité, la censure, ou bien la mise en cache (cf. RFC 7234) des ressources souvent demandées, pour accélérer les requêtes suivantes (c'est par exemple la principale fonction de l'excellent logiciel Squid et c'est un excellent moyen d'économiser de la capacité réseau, particulièrement lorsqu'on est connecté par des lignes lentes). Lorsque le relais n'est pas explicitement choisi par le client, on parle de "*transparent proxy*" (RFC 1919 et RFC 3040). Ils servent typiquement à restreindre les services auquel un utilisateur captif peut accéder. La passerelle ("*gateway*", également nommée "*reverse proxy*", et qu'il ne faut pas confondre avec celle décrite plus haut qui fait la conversion entre HTTP et un autre protocole) est, au contraire, proche du serveur, choisie par lui, et fournit des services comme la répartition de charge ou comme la mémorisation des réponses, pour aller plus vite la prochaine fois (c'est par exemple le rôle du logiciel Varnish dont vous avez vu la présence signalée par l'en-tête `Via:` dans l'exemple précédent). Enfin, le tunnel assure juste une transmission des octets d'un point à un autre. Il est surtout utilisé pour le cas où la communication est chiffrée par TLS mais que le client et le serveur ne peuvent pas se parler directement. Pour tout intermédiaire, il est important de se rappeler que HTTP est **sans état** : deux requêtes, même venant de la même adresse IP, ne sont pas forcément liées (le RFC 4559 faisait l'erreur de violer cette règle).

Un point important pour les logiciels HTTP : la norme ne met pas de limites quantitatives dans bien des cas. C'est le cas par exemple de la longueur des URI. Il y a donc potentiellement problèmes d'interopérabilité. Au minimum, notre RFC demande qu'une mise en œuvre de HTTP sache lire des éléments aussi longs que ceux qu'elle génère elle-même, ce qui semble du bon sens.

On l'a dit, cette version de HTTP est la même que celle du RFC 2616 (et, avant, celle du RFC 2068), la version 1.1. L'idée est que la syntaxe des messages échangés dépend du numéro majeur (1, ici). C'est

pour cela que le passage à un encodage binaire (et non plus texte) des messages va nécessiter un passage à la version majeure numéro 2. Par contre, des nouveaux messages ou des extensions des messages précédents peuvent être ajoutés en incrémentant juste le numéro mineur (1, à l'heure actuelle). En général, clients et serveurs HTTP 1.1 et HTTP 1.0 (normalisé dans le RFC 1945) peuvent ainsi interagir.

Le World-Wide Web repose sur trois piliers, le protocole HTTP, présenté ici, le langage HTML, et les adresses des ressources, les URI, normalisées dans le RFC 3986. HTTP utilise deux plans ("*scheme*") d'URI, `http:` et `https:`. `http:` est spécifique à TCP, bien que HTTP ait juste besoin d'un canal fiable et ne se serve pas des autres fonctions de TCP. Porter HTTP sur, par exemple, SCTP, serait trivial, mais nécessiterait des URI différents (autrement un client bilingue ne saurait pas a priori s'il doit essayer d'établir la connexion en TCP ou en SCTP). Le plan est suivi des deux barres obliques et du champ « nom de machine ». L'adresse IP de la (ou des) machine(s) est typiquement trouvée dans le DNS. Ainsi, ce blog est en `http://www.bortzmeyer.org/` ce qui veut dire qu'il faudra faire une requête DNS pour le nom `www.bortzmeyer.org` (`http://www.bortzmeyer.org/` est un URI, `www.bortzmeyer.org` est un nom de domaine). Le port par défaut est le bien connu 80. Malheureusement, HTTP n'utilise pas de mécanisme d'indirection comme les MX du courrier électronique ou comme les plus modernes SRV du RFC 2782, utilisés par presque tous les autres protocoles Internet. Résultat, il n'est pas trivial de mettre un nom de domaine « court » (juste le nom enregistré, comme `example.org`, sans préfixe devant) dans un URI. Cela ne peut se faire qu'en mettant directement une adresse IP au nom enregistré, empêchant ainsi d'autres services sur le nom court. Cela rend également très difficile la répartition de charge côté client. C'est un des manques les plus sérieux de HTTP.

Le plan `https:` est pour les connexions HTTP sécurisées avec TLS (le petit cadenas du navigateur Web...) Le port est alors le 443. TLS est normalisé dans le RFC 5246.

La section 3 de notre RFC décrit le format des messages. Bon, HTTP est bien connu, il faut vraiment que je le répète ? Une ligne de départ, puis une syntaxe inspirée de l'IMF du RFC 5322, avec ses champs « Nom : valeur », puis une ligne vide puis un corps optionnel. Le récepteur va en général lire la ligne de départ, puis lire les en-têtes en les mettant dans un dictionnaire, puis, si l'analyse de ces données montre qu'un corps peut être présent, le récepteur va lire le corps pour la quantité d'octets indiquée, ou bien jusqu'à la coupure de la connexion. La ligne de départ est la seule dont la syntaxe est différente entre les requêtes et les réponses. Pour une requête, on trouve une méthode (la liste des méthodes possibles est dans le RFC 7231), une cible, puis la version HTTP. Pour la réponse, on a la version HTTP, le code de retour (les fameux trois chiffres), et une raison exprimée en langue naturelle. Voici un exemple avec curl, où on récupère une ressource existante, avec la méthode GET et on a le code de retour 200 (succès) :

```
% curl -v http://www.afnic.fr/
...
> GET / HTTP/1.1
> User-Agent: curl/7.32.0
> Host: www.afnic.fr
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 22 Apr 2014 16:47:34 GMT
< Server: Apache/2.2.3 (Red Hat) DAV/2 mod_ssl/2.2.3 OpenSSL/0.9.8e-fips-rhel5
< Expires: Thu, 19 Nov 1981 08:52:00 GMT
< Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
< Pragma: no-cache
< Content-Type: text/html; charset=utf-8
< Set-Cookie: afnic-prod=m3nc4r1oivltbdkd9qbh6emvr5; path=/
< Transfer-Encoding: chunked
<
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1
-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
...
```

Ici, par contre, on essaie de détruire (méthode DELETE) une ressource qui n'existe pas. On a le code de retour 404 (ressource inexistante) :

```
% curl -v -X DELETE http://www.afnic.fr/test
...
> DELETE /test HTTP/1.1
> User-Agent: curl/7.32.0
> Host: www.afnic.fr
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Date: Tue, 22 Apr 2014 16:50:16 GMT
< Server: Apache/2.2.3 (Red Hat) DAV/2 mod_ssl/2.2.3 OpenSSL/0.9.8e-fips-rhel5
< Expires: Thu, 19 Nov 1981 08:52:00 GMT
< Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
< Pragma: no-cache
...
```

Les codes de retour possibles sont décrits en détail dans le RFC 7231. En attendant, vous trouvez des jolies photos de chats illustrant ces codes chez les HTTP status cats <https://www.flickr.com/photos/girliemac/sets/72157628409467125/> et, pour les chiens, voyez par ici <http://httpstatusdogs.com/>.

Il n'y a pas de limite imposée par la norme pour des choses comme la longueur des lignes de début ou comme la longueur d'un champ dans l'en-tête. En pratique, il est recommandé aux logiciels HTTP d'accepter au moins 8 000 octets.

Les en-têtes des requêtes et réponses comprennent un nom de champ (comme `User-Agent` ou `Expires`), deux-points et la valeur de l'en-tête. Des nouveaux champs sont introduits régulièrement, HTTP n'impose pas un jeu fixe de noms de champs. Ils sont enregistrés dans un registre IANA <https://www.iana.org/assignments/message-headers/message-headers.xhtml> qui est le même que pour les champs du courrier électronique (certains en-têtes, comme `Date:`, sont communs à plusieurs protocoles/formats.)

L'ordre des champs n'est pas significatif. Normalement, les champs sont présents une seule fois au maximum. Mais il y a des exceptions, si le contenu d'un champ est une liste, on peut avoir plusieurs occurrences du champ, la valeur étant alors la concaténation de toutes les valeurs en une liste. Au moins un champ, `Set-Cookie:` (RFC 6265), n'obéit pas à cette règle, pour des raisons historiques, et doit donc être traité à part.

Important changement par rapport à la norme précédente, le RFC 2616, la grammaire des en-têtes. Il n'y a plus une règle spécifique par champ mais une grammaire générique pour les champs, avec une partie spécifique pour la valeur.

Il n'y a **pas** d'espace entre le nom de champ et le deux-points et le RFC impose, principe de robustesse <https://www.bortzmeyer.org/principe-robustesse.html> ou pas, de rejeter les messages ayant de tels espaces. (Autrement, cela permettrait d'intéressantes attaques <https://bugzilla.mozilla.org/show_bug.cgi?id=329746>.)

Autre changement important depuis le précédent RFC, l'encodage par défaut. La norme autorisait explicitement ISO 8859-1 dans les en-têtes, les autres encodages devaient passer par la technique du RFC

2047. Cette règle est désormais abandonnée, les valeurs des en-têtes devant rester en ASCII, ou bien être traitées comme du binaire, sans être interprété comme du ISO 8859-1.

Au cours de son transfert, la ressource à laquelle on accède en HTTP peut subir des transformations, par exemple pour en réduire la taille <https://www.bortzmeyer.org/gzip-compression-apache.html>. La section 4 de notre RFC décrit ces « codages pendant le transfert » : compression mais aussi transfert en plusieurs morceaux.

Maintenant, en section 5 du RFC, un autre point important de HTTP, le routage des requêtes. Lorsqu'un client HTTP reçoit un URL, qu'en fait-il ? Il va regarder si la ressource correspondant à cet URL est déjà dans sa mémoire et est réutilisable. Si non, il va regarder s'il doit faire appel à un relais (cela dépend de la configuration dudit client). Si oui, il se connecte au relais et fait une requête HTTP où l'identificateur de ressource est l'URL complet ("*absolute form*" dans le RFC). Si non, il extrait le nom du serveur HTTP de l'URL, se connecte à ce serveur, et fait une requête HTTP où l'identificateur de ressource est juste la partie « chemin ». Le champ `Host:` de l'en-tête HTTP vaut le nom du serveur. Le port par défaut (s'il n'est pas indiqué dans l'URL) est, comme chacun le sait, 80 (et 443 pour HTTPS). Le nom de serveur donné dans l'URL est directement utilisé pour une requête de résolution de noms pour avoir l'adresse. Malheureusement, comme indiqué plus haut, HTTP n'utilise pas les SRV du RFC 2782, d'où le fait qu'on voit souvent des adresses IP mises directement à l'apex du domaine enregistré.

À noter que ce RFC ne couvre pas l'autre partie du « routage », le fait, pour le serveur, de trouver, pour une cible donnée, la localisation de la ressource demandée. Les premiers serveurs HTTP avaient un routage très simple : la cible était préfixée par un nom de répertoire configuré dans le serveur, et le tout était interprété comme le chemin d'un fichier sur le serveur. Ainsi, `GET /toto.html` sur un serveur où le nom de départ était `/var/web`, servait le fichier `/var/web/toto.html`. Aujourd'hui, ce mécanisme de routage existe toujours mais il est accompagné de nombreux autres. À noter que, depuis la création du concept de "*virtual host*", le serveur HTTP commence par chercher le "*virtual host*", en utilisant le champ `Host:` pour le routage.

La section 6 de notre RFC couvre la gestion des connexions. HTTP n'a pas besoin de grand'chose de la part du protocole de transport sous-jacent : juste une connexion fiable, où les octets sont reçus dans l'ordre envoyé. TCP convient à ce cahier des charges et c'est le protocole de transport utilisé lorsque l'URL est de plan `http:` ou `https:`. On pourrait parfaitement faire du HTTP sur, par exemple, SCTP (RFC 4960), mais il faudrait un nouveau plan d'URL. HTTP, pour l'instant, utilise forcément TCP, et le client HTTP doit gérer les connexions TCP nécessaires (les créer, les supprimer, etc).

Le modèle le plus simple (et le modèle historique de HTTP mais qui n'est plus celui par défaut) est celui où chaque couple requête/réponse HTTP se fait sur une connexion TCP différente, établie avant l'envoi de la requête, et fermée une fois la réponse reçue. Mais d'autres modèles sont possibles. Pour indiquer ses préférences, le client utilise l'en-tête `Connection:`. Par défaut, une connexion TCP persiste après la fin de l'échange, et peut servir à envoyer d'autres requêtes. Si le client veut fermer la connexion TCP immédiatement, il envoie :

```
Connection: close
```

L'établissement d'une connexion TCP prenant un certain temps (la fameuse triple poignée de mains), il est logique que les connexions soient persistentes et réutilisables.

Un client HTTP peut aussi avoir plusieurs connexions TCP ouvertes simultanément vers le même serveur mais le RFC lui impose de limiter leur nombre. (Ce parallélisme est utile pour éviter qu'une

courte requête, par exemple pour une feuille de style soit bloquée par un gros téléchargement.) Les versions précédentes de la norme donnaient des valeurs précises (deux, dans le RFC 2616) mais notre nouveau RFC ne donne plus de chiffre, demandant simplement aux clients d'être raisonnables.

La section 9 est l'obligatoire section de sécurité. D'abord la question de l'autorité que fait (ou pas) la réponse. Les problèmes de sécurité surviennent souvent lorsque l'idée que se fait l'utilisateur ne correspond pas à la réalité : c'est le cas par exemple du hameçonnage où la réponse qui fait autorité, pour HTTP, n'est pas celle que croit l'utilisateur. Le RFC donne quelques conseils comme de permettre aux utilisateurs d'inspecter facilement l'URI (ce que ne font pas les utilisateurs et que les navigateurs Web ne facilitent pas, trop occupés à noyer la barre d'adresses, jugée trop technique, au milieu d'autres fonctions). Mais il peut aussi y avoir des cas où HTTP lui-même est trompé, par exemple si un empoisonnement DNS ou bien une attaque contre le routage IP a envoyé le navigateur vers un autre serveur que celui demandé. HTTPS vise à résoudre ces problèmes mais, avec l'expérience qu'on a maintenant de ce service, on peut voir que ce n'est pas si simple en pratique (attaques contre les AC, bogues dans les mises en œuvre de TLS, etc). Et cela ne résout pas le problème de l'utilisateur qui suit aveuglément un lien dans un courrier reçu... À noter que HTTP n'a aucun mécanisme d'intégrité, pour se protéger contre une modification du message. Il dépend entièrement des services sous-jacents, TLS dans le cas de HTTPS. Ces services protègent le canal de communication mais pas les messages eux-mêmes, pour lesquels il n'y a pas de sécurité de bout en bout, encore une sérieuse limite de HTTPS. Même chose pour la confidentialité (le groupe de travail, après de longues discussions n'a pas réussi à se mettre d'accord sur un texte à inclure au sujet de l'interception des communications HTTP.)

HTTP soulève aussi plein de questions liées à la vie privée. On sait que le journal d'un serveur HTTP peut révéler beaucoup de choses. Un serveur cache d'un réseau local, notamment, voit tout le trafic et peut le relier à des utilisateurs individuels. Bref, il faut traiter les journaux sérieusement : ils sont souvent soumis à des lois de protection de la vie privée (ils contiennent des informations qui sont souvent nominatives comme l'adresse IP du client HTTP), et ils doivent donc être gérés en accord avec les bonnes pratiques de sécurité (par exemple, lisibles seulement par les administrateurs système). Le RFC recommande qu'on ne journalise pas tout ou que, si on le fait, on « nettoie » les journaux au bout d'un moment (par exemple en retirant l'adresse IP du client ou, tout simplement, en supprimant le journal).

La section 8 de notre RFC résume les enregistrements faits à l'IANA pour HTTP :

— Les champs d'en-tête comme `Connection:` ou `Date:` sont dans le registre des en-têtes <https://www.iana.org/assignments/message-headers/> (qui est partagé avec d'autres services, notamment le courrier électronique),
— Les plans d'URI comme `http:` sont dans le registre des plans <https://www.iana.org/assignments/uri-schemes/>,
— Les types des ressources manipulés, dits aussi « types MIME », comme `text/html` sont dans le registre des types <https://www.iana.org/assignments/media-types>,
— Les codages des données transférées (comme `gzip`) sont dans un registre adhoc <https://www.iana.org/assignments/http-parameters#transfer-coding>,
— etc.

Le travail de développement de HTTP a mobilisé énormément de monde, ce qui reflète l'extrême importance de ce protocole sur l'Internet. La section 10 liste des centaines de noms de personnes ayant participé à ce protocole (dont votre serviteur). L'annexe A résume la longue histoire de HTTP depuis sa création en 1990. HTTP/0.9 (qui n'avait pas encore de numéro de version officiel, il l'a reçu après) était un protocole ultra-simple (d'où son succès, alors que les gourous de l'hypertexte travaillaient tous sur des choses bien plus complexes, et regardaient de haut ce service trop simple) qui n'avait qu'une méthode, `GET`. Sa spécification <http://www.w3.org/Protocols/HTTP/AsImplemented.html> était minimale. Les numéros de versions sont officiellement apparus avec HTTP/1.0, le premier décrit dans un RFC, le RFC 1945. HTTP/1.0 introduisait les en-têtes permettant de varier les requêtes et les

réponses, et notamment d'indiquer le type de la ressource récupérée. Son principal manque était l'absence de toute gestion des "*virtual hosts*", puisqu'il n'avait pas l'en-tête `Host:`. Il fallait donc une adresse IP par site servi...

HTTP/1.1, décrit pour la première fois dans le RFC 2068, introduisait notamment le "*virtual hosting*" et les connexions TCP persistantes. Le RFC 2068 a été remplacé ensuite par le RFC 2616, puis par notre RFC 7230 puis encore, mais partiellement, par le RFC 9112, mais sans changement du numéro de version. Si les changements apportés depuis le RFC 2616 sont très nombreux, ils ne changent pas le protocole. Parmi les principaux changements qu'apporte notre RFC 7230 :

— L'indication du nom de l'utilisateur (et, dans certains cas, du mot de passe !) dans l'URI est abandonnée.
— Les URI `https:` sont désormais officiellement définis dans le RFC HTTP et plus dans un RFC à part (le RFC 2818).
— Les en-têtes s'étendant sur plusieurs lignes sont maintenant découragés.
— Et plein d'autres détails, indispensables au programmeur d'un client ou d'un serveur HTTP mais qui ennuieraient probablement très vite la plupart des lecteurs.