RFC 9110 : HTTP Semantics

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 7 juin 2022

Date de publication du RFC : Juin 2022

https://www.bortzmeyer.org/9110.html

Que voilà une épaisse lecture (252 pages). Mais c'est parce qu'il s'agit de réécrire complètement la totalité des normes de HTTP. Pas le protocole lui-même, je vous rassure, HTTP ne change pas. Mais la rédaction de ses normes est profondément réorganisée, avec un RFC (notre RFC 9110[1]) qui décrit une vision de haut niveau de HTTP, puis un autre RFC par version majeure de HTTP, décrivant les détails de syntaxe de chaque version.

Par exemple, HTTP/1 (RFC 9112) a un encodage en texte alors que HTTP/2 (RFC 9113) a un encodage binaire. Pourtant, tous les deux suivent les mêmes principes, décrits dans ce RFC 9110 (méthodes comme `GET`, en-têtes de la requête et de la réponse, codes de retour à trois chiffres…) mais avec des encodages différents, chacun dans son propre RFC. Notre RFC 9110 est donc la vision de haut niveau de HTTP, commune à toutes les versions, et d'autres RFC vous donneront les détails. Inutile de dire que cette réorganisation a été un gros travail, commencé en 2018.

Les trois versions de HTTP actuellement en large usage (1.1, 2 et 3) reposent toutes sur des concepts communs. Par exemple, les codes d'erreur (comme le fameux 404) sont les mêmes. Il n'est pas prévu, même à moyen terme, que les versions les plus anciennes soient abandonnées (HTTP/1.1 reste d'un usage très courant, et souvent pour de bonnes raisons). D'où cette réorganisations des normes HTTP, avec notre RFC 9110 qui décrit ce qui est commun aux trois versions, d'autres RFC communs aux trois versions, et un RFC par version :

— La mémorisation des données (RFC 9111),
— HTTP/1.1 (RFC 9112),
— HTTP/2 (RFC 9113),
— HTTP/3 (RFC 9114).

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

Vous connaissez certainement déjà HTTP, mais notre RFC ne présuppose pas de connaissances préalables et explique tout en partant du début, ce que je fais donc également ici. Donc, HTTP est un protocole applicatif, client/serveur, sans état, qui permet l'accès et la modification de ressources distantes (une ressource pouvant être du texte, une image, et étant générée dynamiquement ou pas, le protocole est indépendant du format de la ressource ou de son mode de création, le RFC insiste bien sur ce point). Le client se connecte, envoie une requête, le serveur répond. HTTP ne fonctionne pas forcément de bout en bout, il peut y avoir des relais sur le trajet, et leur présence contribue beaucoup à certaines complexités de la norme.

S'il fallait résumer HTTP rapidement, on pourrait dire qu'il décrit un moyen d'interagir avec une **ressource** distante (la ressource peut être un fichier, un programme…). Il repose sur l'échange de messages, avec une requête du client vers le serveur et une réponse en sens inverse. Outre la **méthode** qui indique ce que le client veut faire avec la ressource, HTTP permet de transporter des métadonnées.

La section 3 du RFC décrit les concepts centraux de ce protocole, comme celui de ressource présenté plus haut. (Qui est parfois appelé « page » ou « fichier » mais ces termes ne sont pas assez génériques. Une ressource n'est pas forcément une page HTML !) HTTP identifie les ressources par des URI. Une **représentation** est la forme concrète d'une ressource, les bits qu'on reçoit ou envoie. (Du fait de la négociation de contenu et d'autres facteurs, récupérer une ressource en utilisant le même URI ne donnera pas forcément les mêmes bits, même s'ils sont censés être sémantiquement équivalents.) La ressource n'est pas non plus forcément un fichier, pensez à une ressource qui indique l'heure qu'il est, ou le temps qu'il fait, par exemple. Ou à l'URI `https://www.bortzmeyer.org/apps/random` qui vous renvoie une page choisie aléatoirement de ce blog. HTTP agit sur une ressource (dont le type n'est pas forcément connu) via une méthode qui va peut-être retourner une représentation de cette ressource. C'est ce qu'on nomme le principe REST et de nombreuses API se réclament de ce principe.

HTTP est un protocole client/serveur. Le serveur attend le client. (Le client est parfois appelé "*user agent*".) Entre les deux, HTTP utilisera un protocole de transport fiable, comme TCP (HTTP/1 et 2) ou QUIC <https://www.bortzmeyer.org/quic.html> (HTTP/3). Par défaut, HTTP est sans état : une fois une requête servie, le serveur oublie tout. Les clients sont très variés : il y a bien sûr les navigateurs Web, mais aussi les robots, des outils en ligne de commande comme wget, des objets connectés, des programmes vite faits en utilisant une des zillions de bibliothèques qui permettent de développer rapidement un client HTTP, des applications sur un ordiphone, etc. Notamment, il n'y a pas forcément un utilisateur humain derrière le client HTTP. (Pensez à cela si vous mettez des éléments d'interfaces qui demandent qu'un humain y réponde ; le client ne peut pas forcément faire de l'interactivité.)

Les messages envoyés par le client au serveur sont des **requêtes** et ceux envoyés par le serveur des **réponses**.

La section 2 du RFC explique ce qu'on attend d'un client ou d'un serveur HTTP conforme. Un point important et souvent ignoré est que HTTP ne donne pas de limites quantitatives à beaucoup de ses éléments. Par exemple, la longueur maximale de la première ligne de la requête (celle qui contient le chemin de la ressource) n'est pas spécifiée, car il serait trop difficile de définir une limite qui convienne à tous les cas, HTTP étant utilisé dans des contextes très différents. Comme les programmes ont forcément des limites, cela veut dire qu'on ne peut pas toujours compter sur une limite bien connue.

Une mise en œuvre conforme pour HTTP doit notamment bien gérer la notion de version de HTTP. Cette version s'exprime par deux chiffres séparés par un point, le premier chiffre étant la version majeure (1, 2 ou 3) et le second la mineure (il est optionnel, valant 0 par défaut, donc HTTP/2 veut dire la même chose que HTTP/2.0). Normalement, au sein d'une même version majeure, on doit pouvoir interopérer sans trop de problème alors qu'entre deux versions majeures, il peut y avoir incompatibilité

totale. La sémantique est forcément la même (c'est du HTTP, après tout) mais la syntaxe peut être radicalement différente (pensez à l'encodage texte de HTTP/1 vs. le binaire de HTTP/2 et 3). Donc, être conforme à HTTP/1.1 veut dire lire ce RFC 9110 mais aussi le RFC 9112, qui décrit la syntaxe spécifique de HTTP/1.1.

Comme, dans la nature, des programmes ne sont pas corrects, le RFC autorise du bout des lèvres à utiliser le contenu des champs `User-Agent:` ou `Server:` de l'en-tête pour s'ajuster à des bogues connues (mais, normalement, ce doit être uniquement pour contourner des bogues, pas pour servir un contenu différent).

De même qu'un client HTTP n'est pas forcément un navigateur Web, un serveur HTTP n'est pas forcément une grosse machine dans un centre de données chez un GAFA. Le serveur HTTP peut parfaitement être une imprimante, un petit objet connecté, une caméra de vidéosurveillance, un Raspberry Pi dans son coin…Le RFC parle de « serveur d'origine » pour le serveur qui va faire autorité pour les données servies. Pourquoi ce concept ? Parce que HTTP permet également l'insertion d'un certain nombre d'intermédiaires, les relais ("*proxy*" ou "*gateway*" en anglais), entre le client et le serveur d'origine. Leurs buts sont très variés. Par exemple, un relais ("*proxy*", pour le RFC) dans le réseau local où se trouve le client HTTP peut servir à mémoriser les ressources Web les plus souvent demandées, pour améliorer les performances. Un relais ("*gateway*" ou "*reverse proxy*", pour le RFC) qui est au contraire proche du serveur d'origine peut servir à répartir la charge entre diverses instances. Revenons à la mémorisation des ressources ("*caching*" en anglais). La mémoire ("*cache*" en anglais) est un stockage de ressources Web déjà visitées, prêtes à être envoyées aux clients locaux pour diminuer la latence <`https://www.bortzmeyer.org/latence.html`>. La mémorisation est un sujet suffisamment fréquent et important pour avoir son propre RFC, le RFC 9111.

On a vu que HTTP servait à agir sur des ressources distantes. Des ressources, il y en a beaucoup. Comment les identifier ? Le Web va utiliser des URI comme **identificateurs**. Ces URI sont normalisés dans le RFC 3986, mais qui ne spécifie qu'une syntaxe générique. Chaque **plan** d'URI (la chaine de caractères avant le deux-points, souvent appelée à tort protocole) doit spécifier un certain nombre de détails spécifique à ce plan. Pour les plans `http` et `https`, cette spécification est la section 4 de notre RFC. (Tous les plans sont dans un registre IANA <`https://www.iana.org/assignments/uri-schemes/uri-schemes.xml#uri-schemes-1`>.) Un URI de plan `http` ou `https` indique forcément une autorité (un identificateur du serveur d'origine, en pratique un nom de machine) et un chemin (identificateur de la ressource à l'intérieur d'un autorité. Ainsi, dans `https://www.afnic.fr/observatoire-ressources/consul` le plan est `https`, l'autorité `www.afnic.fr` et le chemin `/observatoire-ressources/consultations-publiques` Le port par défaut est 80 pour `http` et 443 pour `https` (tous les deux sont enregistrés à l'IANA <`https://www.iana.org/assignments/http-status-codes/http-status-codes.xml#http-status-codes-1`>). La différence entre les deux plans est que `https` implique l'utilisation du protocole de sécurité TLS (RFC 8446), pour assurer notamment la confidentialité des requêtes.

En théorie, un URI de plan `http` et un autre identique, sauf pour l'utilisation de `https`, sont complètement distincts. Ils ne représentent pas la même origine (l'origine est un triplet {plan, machine, port}) et les deux ressources peuvent être complètement différentes. Mais notre RFC note que certaines normes violent ce principe, notamment celle sur les "*cookies*" (RFC 6265), avec parfois des conséquences fâcheuses pour la sécurité.

En HTTPS, puisque ce protocole s'appuie sur TLS, le serveur présente un certificat, que le client doit vérifier (section 4.3.4), en suivant les règles du RFC 6125.

Pour expliquer plusieurs des propriétés de HTTP, je vais beaucoup utiliser le logiciel curl, un client HTTP en ligne de commande, dont l'option `-v` permet d'afficher tout le dialogue HTTP. Si vous voulez faire des essais vous aussi, interrompez momentanément votre lecture pour installer curl. […] C'est fait ? On peut reprendre ?

```
%  curl -v http://www.hambers.mairie53.fr/
...
> GET / HTTP/1.1
> Host: www.hambers.mairie53.fr
> User-Agent: curl/7.68.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Wed, 02 Mar 2022 16:25:02 GMT
< Server: Apache
...
< Content-Length: 61516
< Content-Type: text/html; charset=UTF-8
<

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-trans
<html xmlns="http://www.w3.org/1999/xhtml">
  ...
```

Nous avons vu que la requête et la réponse HTTP contenaient des métadonnées dans un en-tête composé de champs. (Il peut aussi y avoir un pied, une sorte de post-scriptum, mais c'est peu utilisé.) Chaque champ a un nom et une valeur. La section 5 du RFC détaille cet important concept. Les noms de champs sont insensibles à la casse. Ils sont enregistrés dans un registre IANA <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names> spécifique à HTTP (ils étaient avant dans le même registre que les champs du courrier électronique). Un client, un serveur ou un relais HTTP doivent ignorer les champs qu'ils ne connaissent pas, ce qui permet d'introduire de nouveaux champs sans tout casser. Le même champ peut apparaitre plusieurs fois. Comme pour d'autres éléments du protocole HTTP, la norme ne fixe pas de limite de taille. La valeur d'un champ peut donc être très grande.

La valeur d'un champ obéit à des règles qui dépendent du champ. Les caractères doivent être de l'ASCII, une limite très pénible de HTTP. Si on veut utiliser Unicode (ou un autre jeu de caractères), il faut l'encoder comme indiqué dans le RFC 8187. Le RFC rappelle qu'autrefois Latin-1 était autorisé (avec l'encodage du RFC 2047 pour les autres jeux) mais cela ne devrait normalement plus être le cas (mais ça se rencontre parfois encore). Si une valeur comprend plusieurs termes, ils doivent normalement être séparés par des virgules (et on met entre guillemets les valeurs qui comprennent des virgules). Les valeurs peuvent inclure des paramètres, écrits sous la forme nom=valeur. Certaines valeurs ont leur propre structure (RFC 8941). Ainsi, plusieurs champs peuvent inclure une estampille temporelle. La syntaxe pour celles-ci n'est hélas pas celle du RFC 3339 mais celle de l'IMF (RFC 5322), plus complexe et plus ambigüe. (Sans compter, vu l'âge de HTTP, qu'on rencontre parfois de vieux formats comme celui du RFC 850.) Voici des exemples de champs vus avec curl :

```
% curl -v  http://confiance-numerique.clermont-universite.fr/
...
> GET / HTTP/1.1
> Host: confiance-numerique.clermont-universite.fr
> User-Agent: curl/7.68.0
> Accept: */*

< HTTP/1.1 200 OK
< Date: Fri, 28 Jan 2022 17:21:38 GMT
< Server: Apache/2.4.6 (CentOS)
< Last-Modified: Tue, 01 Sep 2020 13:29:41 GMT
< ETag: "1fbc1-5ae4082ec730d"
< Accept-Ranges: bytes
< Content-Length: 129985
```

```
< Content-Type: text/html; charset=UTF-8
<

<!DOCTYPE HTML SYSTEM>
<html>

<head>
<title>S&eacute;minaire Confiance Num&eacute;rique</title>
```

Le client HTTP (curl) a envoyé trois champs, `Host:` (le serveur qu'on veut contacter), `User-Agent:` (une chaine de caractères décrivant le client) et `Accept:` (les formats acceptés, ici tous). Le serveur a répondu avec divers champs comme `Server:` (l'équivalent du `User-Agent:`) et `Content-Type:` (le format utilisé, ici HTML). Et voici ce qu'envoie le navigateur Firefox :

```
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:96.0) Gecko/20100101 Firefox/96.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

On notera surtout un `Accept:` plus complexe (curl accepte tout car il ne s'occupe pas de l'affichage).

Maintenant, les messages (requêtes et réponses). La façon exacte dont ils sont transmis dépend de la version de HTTP. Par exemple, la version 1 les encode en texte alors que les versions 2 et 3 préfèrent le binaire. Autre exemple, la version 3 ne prévoit pas de mécanisme de début et de fin d'un message car chaque ruisseau QUIC <https://www.bortzmeyer.org/quic.html> ne porte qu'un seul message, un peu comme les versions 0 de HTTP, avec le ruisseau QUIC au lieu de la connexion TCP (notez qu'avec TCP sans TLS, le client peut ne pas savoir s'il a bien reçu toutes les données). La section 6 de notre RFC ne donne donc qu'une description abstraite. Un message comprend donc une information de contrôle (la première ligne, dans le cas de HTTP/1, un « pseudo en-tête » avec des noms de champs commençant par un deux-points pour les autres versions), un en-tête, un corps (optionnel) et un pied (également optionnel). L'information de contrôle donne plusieurs informations nécessaires pour la suite, comme la version de HTTP utilisée. Le contenu (le corps) est juste une suite d'octets, que HTTP transporte sans l'interpréter (ce n'est pas forcément de l'HTML). Dans la réponse, l'information de contrôle comprend notamment un code numérique de trois chiffres, qui indique comment la requête a été traitée (ou pas).

Beaucoup moins connu que l'en-tête, un message peut aussi comporter un pied, également composé de champs « nom : valeur ». Il est nécessaire de les utiliser dans les cas où l'information est générée dynamiquement et que certaines choses ne peuvent être déterminées qu'après coup (une signature numérique, par exemple).

Dans le cas le plus simple, le client HTTP parle directement au serveur d'origine et il n'y a pas de complications de routage du message. Le serveur traite le message reçu, point. Mais HTTP permet d'autres cas, par exemple avec un relais qui reçoit le message avant de le transmettre au « vrai » serveur

(section 7 du RFC). Ainsi, dans une requête, l'information de contrôle n'est pas forcément un simple chemin (`/publications/cahiers-soutenabilites`) mais peut être un URL complet (`https://www.strategie.`
C'est ce que fait le client HTTP s'il est configuré pour utiliser un relais, par exemple pour mémoriser les réponses des requêtes (RFC 9111), ou bien parce que l'accès direct aux ports 80 et 443 est bloqué et qu'on est obligé d'utiliser un relais. Dans le cas où la ressource demandées est identifiée par un URL complet, le relais doit alors se transformer en client HTTP et faire une requête vers le serveur d'origine (ou bien vers un autre relais…).

La section 8 de notre RFC s'attaque à une notion cruciale en HTTP, celle de **représentation**. La représentation d'une ressource est la suite d'octets qu'on obtient en réponse à une requête HTTP (« représentation » est donc plus concret que « ressource »). Une même ressource peut avoir plusieurs représentations, par exemple selon les métadonnées que le client a indiqué dans sa requête. Le type de la représentation est indiqué par le champ `Content-Type:` de l'en-tête (et aussi par `Content-Encoding:`). Sa valeur est un type MIME (RFC 2046). Voici par exemple le type de la page que vous êtes en train de lire :

```
Content-Type: text/html; charset=UTF-8
```

(Notez que le paramètre `charset` est mal nommé, c'est en fait un encodage, pas un jeu de caractères. L'erreur vient du fait que dans les vieilles normes comme ISO-8859-1, les deux concepts étaient confondus.) Normalement, du fait de ce `Content-Type:`, le client HTTP n'a pas à deviner le type de la représentation, il se fie à ce que le serveur raconte. Ceci dit, certains clients ont la mauvaise idée de chercher à deviner le type <`https://mimesniff.spec.whatwg.org/`>. Cette divination est toujours incertaine (plusieurs types de données peuvent se ressembler) et ouvre même la possibilité de failles de sécurité.

Un autre champ, `Content-Language:`, indique la langue de la représentation récupérée. Sa valeur est une étiquette de langue, au sens du RFC 5646. Si le texte est multilingue, ce champ peut prendre plusieurs valeurs. Le RFC illustre cela avec le traité de Waitangi, qui est en maori et en anglais :

```
Content-Language: mi, en
```

Attention, la seule présence de différentes langues ne signifie pas qu'il faut mettre plusieurs étiquettes de langue. Un cours d'introduction à l'arabe écrit en français, pour un public francophone, sera :

```
Content-Language: fr
```

Les étiquettes de langue peuvent être plus complexes que l'indication de la seule langue, mais il me semble que c'est rarement utilisé sur le Web.

La taille de la représentation, elle, est exprimée avec `Content-Length:`, un champ très pratique pour le client HTTP qui sait ainsi combien d'octets il va devoir lire (avant HTTP/1, c'était facile, on lisait jusqu'à la fin de la connexion TCP ; mais ça ne marche plus depuis qu'il y a des connexions persistentes et, de toute façon, en l'absence de TLS, cela ne permettait pas de détecter des coupures prématurées). Évidemment, le client doit rester paranoïaque et supposer que l'information puisse être fausse. curl (avec `-v`) avertit ainsi, si la taille indiquée est trop faible :

```
* Excess found in a read: excess = 1, size = 12, maxdownload = 12, bytecount = 0
```

Si la taille indiquée est trop grande, curl attend pour essayer de lire davantage sur le connexion qui reste ouverte. Autre raison d'être paranoïaque, la taille indiquée peut être énorme, menant par exemple un client imprudent, qui allouerait la mémoire demandée à épuiser celle-ci. Sans compter l'éventualité d'un débordement d'entier si la taille ne peut pas être représentée dans les entiers utilisés par le client HTTP.

Ensuite vient un autre point pas forcément très connu : les validateurs. HTTP permet d'indiquer des pré-conditions à la récupération d'une ressource, pour épargner le réseau. Un client HTTP peut ainsi demander « donne-moi cette ressource, si elle n'a pas changé ». Pour cela, HTTP repose sur ces validateurs, qui sont des métadonnées qui accompagnent la requête (avec des champs qui expriment la requête conditionnelle, comme `If-Modified-Since:`, et qui sont détaillés en section 13) et que le serveur vérifiera. Il existe deux sortes de validateurs, les forts et les faibles. Les faibles sont faciles à générer mais ne garantissent pas une comparaison réussie, les forts sont plus difficiles à faire mais sont plus fiables. Par exemple, un condensat du contenu est fort. Il changera forcément (sauf malchance inouïe) dès qu'on changera un seul bit du contenu. Si le contenu est géré par un VCS, celui-ci fournit également des validateurs forts : l'identificateur de "*commit*". Au contraire, une estampille temporelle est un validateur faible. Si sa résolution est d'une seconde, deux modifications dans la même seconde ne seront pas détectées et le serveur croira à tort que le contenu n'a pas changé.

Pour connaitre la valeur actuelle d'un futur validateur, le client HTTP dispose de champs comme `Last-Modified:` (une estampille temporelle) ou `ETag:` ("*Entity Tag*", l'étiquette de la ressource, une valeur opaque, qui peut s'utiliser avec des requêtes conditionnelles comme `If-None-Match:`). Voici un exemple :

```
Last-Modified: Mon, 07 Feb 2022 12:20:20 GMT
ETag: "5278-5d76c9fc1c9f4"
```

(Le serveur utilisé était un Apache. Par défaut, Apache génère des étiquettes qui sont un condensat de divers attributs du fichier comme l'inœud, la taille et la date de modification. Apache permet de configurer cet algorithme <https://httpd.apache.org/docs/2.4/mod/core.html#fileetag>. Rappelez-vous que l'étiquette est opaque, le serveur peut donc la générer comme il veut, il doit juste s'assurer qu'elle change à chaque modification de la ressource. Le serveur peut par exemple utiliser un SHA-1 du contenu de la ressource.) A priori, l'étiquette de la ressource est un validateur fort, autrement, le serveur doit la préfixer par `W/` (W pour "*Weak*").

Passons maintenant aux **méthodes** (section 9 du RFC). Il y a très longtemps, HTTP n'avait qu'une seule méthode pour agir sur les ressources, la méthode `GET`. Désormais, il y a nettement plus de méthodes, chacune agissant sur la ressource indiquée d'une manière différente et ayant donc une sémantique différente. Par exemple, `GET` va récupérer une représentation de la ressource, alors que `PUT` va au contraire écrire le contenu envoyé, remplaçant celui de la ressource et que `DELETE` va…détruire la ressource. La liste complète des méthodes figure dans un registre IANA <https://www.iana.org/assignments/http-methods/http-methods.xml#methods>.

Certaines des méthodes sont dites sûres car elles ne modifient pas la ressource et ne casseront donc rien. Bien sûr, une méthode sûre peut avoir des effets de bord (comme d'écrire une ligne dans le journal du serveur, mais ce n'est pas la faute du client). `GET`, `HEAD` et les moins connues `OPTIONS` et `TRACE`

sont sûres. Du fait de cette garantie de sûreté, un programme qui ne fait que des requêtes sûres a moins d'inquiétudes à avoir, notamment s'il agit sur la base d'informations qu'il ne contrôle pas. Ainsi, le ramasseur d'un moteur de recherche ne fait a priori que des requêtes sûres, pour éviter qu'une page Web malveillante ne l'entraine à effectuer des opérations qui peuvent changer le contenu des sites Web visités.

Une autre propriété importante d'une méthode est d'être idempotente ou pas. Une méthode idempotente a le même effet qu'on l'exécute une ou N fois. Les méthodes sûres sont toutes idempotentes mais l'inverse n'est pas vrai : `PUT` et `DELETE` sont idempotentes (qu'on détruise une ressource une ou N fois donnera le même résultat : la ressource est supprimée) mais pas sûres. L'intérêt de cette propriété d'idempotence est qu'elles peuvent être répétées sans risque, par exemple si le réseau a eu un problème et qu'on n'est pas certain que la requête ait été exécutée. Les méthodes non-idempotentes ne doivent pas, par contre, être répétées aveuglément.

La méthode la plus connue et sans doute la plus utilisée, `GET`, permet de récupérer une représentation d'une ressource. La syntaxe avec laquelle s'exprime le chemin de cette ressource fait penser à l'arborescence d'un système de fichiers et c'est en effet souvent ainsi que c'est mis en œuvre dans les serveurs (par exemple dans Apache, où le chemin, mettons `/foo/bar`, est ajouté à la fin de la variable de configuration `DocumentRoot`, avant d'être récupéré sur le système de fichiers : si `DocumentRoot` vaut `/var/www`, le fichier demandé sera `/var/www/foo/bar`). Mais ce n'est pas une obligation de HTTP, qui ne normalise que le protocole entre le client et le serveur, pas la façon dont le serveur obtient les ressources.

La méthode `HEAD` fait la même chose que `GET` mais sans renvoyer la représentation de la ressource.

`POST` est plus compliquée. Contrairement à `GET`, la requête contient des données qui vont être envoyés au serveur. Celui-ci va les traiter. `POST` est souvent utilisé pour soumettre le contenu d'un formulaire Web, par exemple pour envoyer un texte qui sera le contenu d'un commentaire lors d'une discussion sur un forum Web. Avec `GET`, `POST` est probablement la méthode la plus souvent vue sur le Web.

`PUT`, lui, est également accompagné de données qui vont être écrites à la place de la ressource désignée. On peut donc mettre en œuvre un serveur de fichiers distant avec des `PUT` et des `GET`. On peut y ajouter `DELETE` pour supprimer les ressources devenues inutiles.

La méthode `CONNECT` est plus complexe. Elle n'agit pas sur une ressource mais permet d'établir une connexion avec un service distant. Sa principale utilité est de permettre d'établir un tunnel au-dessus de HTTP. Ainsi :

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com
```

va établir une connexion avec `server.example.com` et les octets envoyés par la suite sur cette connexion HTTP seront relayés aveuglément vers `server.example.com`.

Quant à la méthode `OPTIONS`, elle permet d'obtenir des informations sur les options gérées par le serveur. curl permet d'indiquer une méthode avec son option `--request` (ou `-X`) :

```
% curl -v --request OPTIONS https://www.bortzmeyer.org/
...
> OPTIONS / HTTP/2
> Host: www.bortzmeyer.org
> user-agent: curl/7.68.0
> accept: */*
>
...
< HTTP/2 200
< permissions-policy: interest-cohort=()
< allow: POST,OPTIONS,HEAD,GET
...
```

La section 10 du RFC est ensuite une longue section qui décrit le contexte des messages HTTP, c'est-à-dire les métadonnées qui accompagnent requêtes et réponses. Je ne vais évidemment pas en reprendre toute la liste <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names> ici. Juste quelques exemples de champs intéressants :

— `From` : permet d'indiquer l'adresse de courrier du responsable du logiciel. Il est surtout utilisé par les bots (par exemple ceux qui ramassent les pages pour le compte d'un moteur de recherche) pour indiquer qui contacter si le bot se comporte mal, par exemple en faisant trop de requêtes. Comme le rappelle le RFC, un navigateur ordinaire ne doit évidemment pas transmettre une telle donnée personnelle à tous les sites Web visités !

— `Referer` : (oui, avec une faute d'orthographe <https://en.wiktionary.org/wiki/referrer#Noun>) sert à indiquer l'URL d'où vient le client HTTP. Le Web étant fondé sur l'idée d'hypertexte, l'utilisateur est peut-être venu ici en suivant un lien, et il peut ainsi indiquer où il a trouvé ce lien, ce qui peut permettre au webmestre de voir d'où viennent ses visiteurs. Lui aussi pose des problèmes de vie privée, et j'ai toujours été surpris que le Tor Browser l'envoie.

— `User-Agent` : indique le type du client HTTP. À part s'amuser en regardant le genre de visiteurs qu'on a, il n'a pas de vraie utilité, le Web reposant sur des normes, et précisant une structure et pas une présentation, il ne devrait pas y avoir besoin de changer une ressource en fonction du logiciel du visiteur. Mais c'est quand même ce que font certains serveurs HTTP, poussant les clients à mentir pour obtenir un certain résultat, ce qui donne des champs `User-Agent` : ridicules comme (vu sur ce blog) `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.75 Safari/537.36` (probablement le navigateur Safari indiquant autant de logiciels que possible ; le RFC dit qu'il ne faut pas le faire mais c'est courant, pour tenir compte de serveurs qui interprètent ce champ). Là encore, on a une métadonnée qui contribue puissamment à la fuite d'information si commune sur le Web (votre client HTTP est certainement trop bavard). Le `User-Agent` : est très utile pour le "*fingerprinting*", l'identification d'un visiteur particulier, comme le démontre le Panopticlick <https://coveryourtracks.eff.org/>.

— `Server` : est l'équivalent de `User-Agent` : mais pour le serveur.

Jusqu'à présent, on a supposé que les ressources servies étaient accessibles à tous et toutes. Mais en pratique, on souhaite parfois servir du contenu à accès restreint et on veut donc n'autoriser que certains visiteurs. Il faut donc disposer de mécanismes d'authentification, exposés dans la section 11 du RFC. HTTP n'a pas un mécanisme unique d'authentification. Chaque mécanisme est identifié par un nom (et les possibilités sont dans un registre IANA <https://www.iana.org/assignments/http-authschemes/http-authschemes.xml#authschemes>). Le serveur indique le mécanisme à utiliser dans un champ `WWW-Authenticate` : de sa première réponse. Par exemple, `basic`, normalisé dans le RFC 7617, est un mécanisme simple de mot de passe, alors que `digest` (normalisé dans le RFC 7616) permet de s'authentifier via un défi/réponse. Le mécanisme est spécifique à un royaume, une information donnée par le serveur pour le cas où le même serveur gérerait des types d'authentification différents selon la ressource.

Voici un exemple d'authentification (avec le service d'administration d'un serveur dnsdist <https://dnsdist.org/>, celui utilisé pour mon résolveur public <https://doh.bortzmeyer.fr/policy>) où l'identificateur est `admin` et le mot de passe `2e12` :

```
% curl -v --user admin:2e12 https://doh.bortzmeyer.fr:8080/
> GET / HTTP/1.1
> Host: doh.bortzmeyer.fr:8080
> Authorization: Basic YWRtbW...OTcyM=
> User-Agent: curl/7.68.0
> Accept: */*
>
...
< HTTP/1.1 200 OK
...
```

La représentation renvoyée peut dépendre du client, c'est ce qu'on nomme la négociation de contenu (section 12 du RFC). La méthode officielle est que le client annonce avec le champ `Accept:` les types de données qu'il accepte, et le serveur lui envoie de préférence ce qu'il a demandé. (C'est utilisé sur ce blog pour les images. En pratique, ça ne se passe pas toujours bien <https://www.bortzmeyer.org/negociation-contenu-http.html>.) La demande du client n'est pas strictement binaire « je veux du format WebP ». Elle peut s'exprimer de manière plus nuancée, via le système de qualité. Ainsi :

```
Accept: text/plain; q=0.5, text/html
```

signifie que le client comprend le texte brut et l'HTML mais préfère ce dernier (le poids par défaut est 1, supérieur, donc, au 0,5 du texte brut).

La négociation de contenu ne s'applique pas qu'au format des représentations, elle peut aussi s'appliquer à la langue, avec le champ `Accept-Language:`. Ainsi, en disant :

```
Accept-Language: da, en;q=0.8
```

veut dire « je préfère le danois (poids de 1 par défaut), mais j'accepte l'anglais ». En pratique, ce n'est pas très utile sur le Web car cela ne permet pas d'indiquer la qualité de la traduction. Si on indique qu'on préfère le français, mais qu'on peut lire l'anglais, en visitant des sites Web d'organisations internationales, on se retrouve avec un texte français mal traduit, alors qu'on aurait préféré la version originale <https://www.bortzmeyer.org/web-et-version-originale.html>. En outre, comme beaucoup de champs de l'en-tête de la requête, il contribue à identifier le client (*"fingerprinting"*). C'est d'autant plus gênant que l'indication des langues préférées peut vous signaler à l'attention de gens peu sympathiques, si ces langues sont celles d'une minorité opprimée.

Comme la représentation envoyée peut dépendre de ces demandes du client, le serveur doit indiquer dans sa réponse s'il a effectivement tenu compte de la négociation de contenu. C'est notamment important pour les relais Web qui mémorisent le contenu des réponses (RFC 9111). Le champ `Vary:` permet d'indiquer de quoi a réellement dépendu la réponse. Ainsi :

```
Vary: accept-language
```

indique que la réponse ne dépendait que de la langue. Si un client d'un relais Web demande la même ressource mais avec une autre langue, il ne faut pas lui donner le contenu mémorisé.

HTTP permet d'exprimer des requêtes conditionnelles. Un client HTTP peut par exemple demander une ressource « sauf si elle n'a pas été modifiée depuis 08 :00 ». Cela permet d'économiser des ressources, notamment dans les relais qui mémorisent RFC 9111. C'est également utile aux clients de syndication, qui récupèrent régulièrement ("*polling*") le flux Atom pour voir s'il a changé. Les requêtes conditionnelles permettent, la plupart du temps, d'éviter tout téléchargement de ce flux.

Un autre scénario d'utilisation des requêtes conditionnelles est le cas de la mise à jour perdue ("*lost update*"). Prenons un client qui met à jour une ressource, en récupérant d'abord son état actuel (avec un `GET`), en la modifiant, puis en téléversant la version modifiée (avec par exemple un `PUT`). Si deux clients font l'opération à peu près en même temps, il y a un risque d'une séquence :

— Client 1 fait un `GET`,
— Client 2 fait pareil,
— Client 1 fait un `PUT` avec sa version modifiée,
— Client 2 en fait autant,
— La mise à jour de Client 1 est perdue

Les mises à jour conditionnelles résoudraient ce problème : si Client 2 fait sa mise à jour en ajoutant « seulement si la ressource n'a pas changé », son `PUT` sera refusé, il devra refaire un `GET` et il récupérera alors les changeements de Client 1 avant d'appliquer les siens.

En pratique, les requêtes conditionnelles se font avec des champs comme `If-Modified-Since:` dont la valeur est une date. Par exemple, le client de syndication qui a récupéré une page le 24 février à 18 :11, va envoyer un `If-Modified-Since: Thu, 24 Feb 2022 18:11:00 GMT` et le serveur ne lui enverra la ressource Atom ou RSS que si elle est plus récente (il recevra un code de retour 304 dans le cas contraire). Autre exemple de champ d'en-tête pour les requêtes conditionnelles, `If-Match:`. Ce champ demande que la requête ne soit exécutée que si la ressource correspond à la valeur du `If-Match:`. La valeur est un validateur, comme expliqué plus haut. `If-Match:` permet ainsi de résoudre le problème de la mise à jour perdue.

Les ressources chargées en HTTP peuvent être de grande taille. Parfois, le réseau a un hoquet et le transfert s'arrête. Il serait agréable de pouvoir ensuite reprendre là où on s'était arrêté, au lieu de tout reprendre à zéro. HTTP le permet via les requêtes d'un intervalle (section 14). Le client indique par le champ `Range:` quel intervalle de la ressource il souhaite. Cet intervalle peut se formuler en plusieurs unités, le champ `Accept-Ranges:` permettant au serveur d'indiquer qu'il gère ces demandes, et dans quelles unités. En pratique, seuls les octets marchent réellement. Par exemple, ici, j'utilise curl pour récupérer 50 octets d'un article :

```
% curl -v --range 3100-3150 https://www.bortzmeyer.org/1.html
...
> GET /1.html HTTP/2
> Host: www.bortzmeyer.org
> range: bytes=3100-3150
...
ocuments, qui forme l'ossature de
la <b><a class="
```

Passons maintenant aux codes de retour HTTP (section 15). Il y en a au moins un qui est célèbre, 404, qui indique que la ressource demandée n'a pas été trouvée sur ce serveur et qui est souvent directement visible par l'utilisateur humain (pensez aux « pages 404 » d'erreur). Ces codes sont composés de trois chiffres, le premier indiquant la classe :

— 1 indique que l'opération n'est pas terminée,
— 2 indique que l'opération a été finie et avec succès,
— 3 indique que l'opération n'a pas été faite mais il ne s'agit pas pour autant d'une erreur (contrairement à ce que dit le RFC, ce n'est pas forcément une redirection, plutôt « la vérité est ailleurs », ainsi 304 signifie, en réponse à une requête conditionnelle, que la ressource n'a pas été modifiée),
— 4 indique une erreur due au client, et qu'il n'est donc pas utile que celui-ci réessaie à l'identique,
— 5 indique une erreur due au serveur et le client peut donc avoir intérêt à réessayer plus tard.

Les deux autres chiffres fournissent des détails mais un client HTTP simple peut se contenter de comprendre la classe et d'ignorer les deux autres chiffres. Par exemple, 100 signifie que le serveur a compris la requête mais qu'il faut encore attendre pour la vraie réponse, 200 veut dire qu'il n'y a rien à dire, que tout s'est bien passé comme demandé, 308 indique qu'il faut aller voir à un autre URL, 404, comme signalé plus haut, indique que le serveur n'a pas trouvé la ressource, 500 est une erreur générique du serveur, en général renvoyée quand le serveur a eu un problème imprévu. Peut-être connaissez-vous également :

— 201 qui indique que la ressource a été créée (en réponse à un `PUT`),
— 304, pour répondre à une requête conditionnelle, qui veut dire que le contenu n'ayant pas été modifié, le serveur n'envoie pas de réponse (si votre serveur a un flux de syndication, vous trouverez souvent cette réponse dans vos journaux),
— 400 qui signifie que la requête était syntaxiquement incorrecte,
— 401, pour demander que le client s'authentifie avant d'accéder à cette ressource, et 403 qui lui refuse purement et simplement l'accès,
— 410 qui veut dire la même chose que 404 mais en ajoutant que la ressource a bien existé dans le passé mais a été retirée (utilisé par exemple dans un article de ce blog <https://www.bortzmeyer.org/dinos-partis.html>),
— 409 indique un conflit entre l'état de la ressource et une modification demandée par le client (lorsqu'on se sert de HTTP pour éditer des ressources),
— 414 pour un URL trop long pour ce serveur (la norme HTTP ne met pas de limite quantitative à la taille des URL mais chaque serveur, lui, a une telle limite),
— 418 n'est pas réellement dans le registre IANA <https://www.iana.org/assignments/http-status-codes/http-status-codes.xml#http-status-codes-1> mais il avait été utilisé dans un RFC du premier avril et il apparait souvent dans les blagues <https://save418.com/>,
— 503 que le serveur utilise quand il est temporairement à bout de moyens (surcharge, par exemple),

De nouveaux codes sont créés de temps en temps, et mis dans le registre IANA <https://www.iana.org/assignments/http-status-codes/http-status-codes.xml#http-status-codes-1>.

`curl -v` vous affichera entre autres ce code de retour. Si vous ne voulez que le code, et pas tous les messages que l'utilisation de `-v` entrainera, l'option `--write-out` est bien pratique :

```
% curl --silent --write-out "%{http_code}\n" --output /dev/null https://www.bortzmeyer.org/1.html
200

% curl --silent --write-out "%{http_code}\n" --output /dev/null https://www.bortzmeyer.org/2.html
404
```

Sinon, pour rire avec les codes de statut HTTP, il existe des photos de chats <https://http.cat/> et une proposition d'illustrer ces codes par des émojis <https://www.bortzmeyer.org/http-code-emoji.html>.

Beaucoup de choses dans HTTP peuvent être étendues (section 16). On peut créer de nouvelles méthodes, de nouveaux codes de retour, etc. Un logiciel client ou serveur ne doit donc pas s'étonner de voir apparaitre des questions ou des réponses qui n'existaient pas quand il a été programmé.

Ainsi, des nouvelles méthodes, en sus des traditionnelles `GET`, `POST`, `PUT`, etc , peuvent être créées, comme l'avait été `PATCH` par le RFC 5789. La liste à jour des méthodes est dans un registre IANA <https://www.iana.org/assignments/http-methods/http-methods.xml#methods>. Si vous programmez côté serveur, et que vous utilisez l'interface CGI, la méthode est indiquée dans la variable `REQUEST_METHOD` et vous pouvez la tester, par exemple ici en Python :

```
if environ["REQUEST_METHOD"] == "FOOBAR":
   ... Do something useful
else:
   return unsupported(start_response, environ["REQUEST_METHOD"])
```

Des codes de retour peuvent également être ajoutés, comme le 451 (censure) du RFC 7725. Là aussi, la liste faisant autorité est le registre IANA <https://www.iana.org/assignments/http-status-codes/http-status-codes.xml#http-status-codes-1>.

Bien sûr, le registre le plus dynamique, celui qui voit le plus d'ajouts, est celui des champs de l'en-tête <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names>. Mais attention : du fait qu'il bouge beaucoup, les nouveaux champs ne seront pas compris et utilisés par une bonne partie des logiciels. (Au passage, notre RFC rappelle que l'ancienne convention de préfixer les noms de champs non officiels par un `X-` a été abandonnée, par le RFC 6648.)

Et enfin, on peut étendre les listes de mécanismes d'authentification <https://www.iana.org/assignments/http-authschemes/http-authschemes.xml#authschemes>, et plusieurs autres.

Très utilisé, HTTP a évidemment connu sa part de problèmes de sécurité. La section 17 du RFC analyse les principaux risques. (Certains risques spécifiques sont traités dans d'autres RFC. Ainsi, les problèmes posés par l'analyse de l'encodage textuel de HTTP/1 sont étudiés dans le RFC 9112. Ceux liés aux URL sont dans le RFC 3986.) D'autre part, beaucoup de problème de sécurité du Web viennent :

— du code qui tourne sur le serveur ; notre RFC ne va pas parler de la sécurité de WordPress,
— des failles du client (par exemple si en exécutant du code JavaScript, il permet un accès excessif à la machine cliente),
— de l'Internet lui-même (comme la divulgation de l'adresse IP source).

Cette section 17 se concentre sur les problèmes de sécurité de HTTP, ce qui est déjà pas mal. Le RFC recommande la lecture des documents OWASP pour le reste.

Bon, premier problème, la notion d'autorité. Une réponse fait autorité si elle vient de l'**origine**, telle qu'indiquée dans l'URL. Le client HTTP va donc dépendre du mécanisme de résolution de nom. Si, par exemple, la machine du client utilise un résolveur DNS <https://www.bortzmeyer.org/resolveur-dns.html> menteur, tout est fichu. On croit aller sur `http://pornhub.com/` et on se retrouve sur une page Web de l'ARCOM. Il est donc crucial que cette résolution de noms soit sécurisée, par exemple en utilisant un résolveur DNS de confiance, et qui valide les réponses avec DNSSEC. HTTPS protège partiellement. Une des raisons pour lesquelles sa protection n'est pas parfaite est qu'il est compliqué de valider proprement (cf. RFC 7525). Et puis les problèmes sont souvent non techniques, par exemple la plupart des tentatives d'hameçonnage ne vont pas viser l'autorité mais la perception que l'utilisateur en a. Une page Web copiée sur celle d'une banque peut être prise pour celle de la banque même si, techniquement, il n'y a eu aucune subversion des techniques de sécurité. Le RFC recommande qu'au minimum, les navigateurs Web permettent d'examiner facilement l'URL vers lequel va un lien, et de l'analyser (beaucoup d'utilisateurs vont croire, en voyant `https://nimportequoi.example/banque-de-confian` que le nom de domaine est `banque-de-confiance.com`...).

On a vu qu'HTTP n'est pas forcément de bout en bout et qu'il est même fréquent que des intermédiaires se trouvent sur le trajet. Évidemment, un tel intermédiaire est idéalement placé pour certaines attaques. Bref, il ne faut utiliser que des intermédiaires de confiance et bien gérés. (De nombreuses

organisations placent sur le trajet de leurs requêtes HTTP des boites noires au logiciel privateur qui espionnent le trafic et font Dieu sait quoi avec les données récoltées.)

HTTP est juste un protocole entre le client et le serveur. Le client demande une ressource, le serveur lui envoie. D'où le serveur a-t-il tiré cette ressource ? Ce n'est pas l'affaire de HTTP. En pratique, il est fréquent que le serveur ait simplement lu un fichier pré-existant sur ses disques et, en outre, que le chemin menant à ce fichier vienne d'une simple transformation de l'URL. Par exemple, Apache, avec la directive `DocumentRoot` valant `/var/doc/mon-beau-site` et une requête HTTP `GET /toto/tata.html` va chercher un fichier `/var/www/mon-beau-site/toto/tata.html`. Dans ce cas, attention, certaines manipulations sur le chemin donné en paramètre à `GET` peuvent donner au client davantage d'accès que ce qui était voulu. Ainsi, sans précautions particulières, une requête `GET /../toto/tata.html` serait traduite en `/var/www/mon-beau-site/../toto/tata.html`, ce qui, sur Unix, équivaudra à `/var/www/toto/tata.html`, où il n'était peut-être pas prévu que le client puisse se promener. Les auteurs de serveurs doivent donc être vigilants : ce qui vient du client n'est pas digne de confiance.

Autre risque lorsqu'on fait une confiance aveugle aux données envoyées par le client, l'injection <https://www.bortzmeyer.org/sql-injection.html>. Ces données, par exemple le chemin dans l'URL, sont traitées par des langages qui ont des règles spéciales pour certains caractères. Si un de ces caractères se retrouve dans l'URL, et qe le programme, côté serveur, n'est pas prudent avec les données extérieures, le ou les caractères spéciaux seront interprétés, avec parfois d'intéressantes failles de sécurité à la clé. (Mais, attention, tester la présence de « caractères dangereux <https://www.bortzmeyer.org/caracteres-dangereux.html> » n'est en général pas une bonne idée.)

La liste des questions de sécurité liées à HTTP ne s'arrête pas là. On a vu que HTTP ne mettait pas de limite de taille à des éléments comme l'URL. Un analyseur imprudent, côté serveur, peut se faire attaquer par un client qui enverrait un chemin d'URL très long, déclenchant par exemple un débordement de tableau.

HTTP est un protocole très bavard, et un client HTTP possède beaucoup d'informations sur l'utilisateur humain qui est derrière. Le client doit donc faire très attention à ne pas envoyer ces données. Le RFC ne donne pas d'exemple précis mais on peut par exemple penser au champ `Referer` qui indique l'URL d'où on vient. Si le client l'envoie systématiquement, et que l'utilisateur visitait un site Web interne de l'organisation avant de cliquer vers un lien externe, son navigateur enverra des détails sur le site Web interne. Autre cas important, un champ comme `Accept-Language`, qu'on peut estimer utile dans certains cas, est dangereux pour la vie privée, transmettant une information qui peut être sensible, par exemple si on a indiqué une langue minoritaire et mal vue dans son pays. Et `User-Agent` facilite le ciblage d'éventuelles attaques du serveur contre le client.

Du fait de ce caractère bavard, et aussi parce que, sur l'Internet, il y a des choses qu'on ne peut pas dissimuler facilement (comme l'adresse IP source), ce que le serveur stocke dans ses journaux est donc sensible du point de vue de la vie privée. Des lois comme la loi Informatique & Libertés encadrent la gestion de telles bases de données personnelles. Le contenu de ces journaux doit donc être protégé contre les accès illégitimes.

Comme HTTP est bavard et que le client envoie beaucoup de choses (comme les `Accept-Language` et `User-Agent` cités plus haut), le serveur peut relativement facilement faire du "*fingerprinting*", c'est-à-dire reconnaitre un client HTTP parmi des dizaines ou des centaines de milliers d'aures. (Vous ne me croyez pas ? Regardez le Panopticlick <https://coveryourtracks.eff.org/>.) Un serveur peut ainsi suivre un client à la trace, même sans "*cookies*" (voir « "*A Survey on Web Tracking : Mechanisms, Implications, and Defenses*" <https://doi.org/10.1109/JPROC.2016.2637878> »).

Voilà, et encore je n'ai présenté ici qu'une partie des questions de sécurité liées à l'utilisation de HTTP. Lisez le RFC pour en savoir plus. Passons maintenant aux différents registres IANA qui servent à stocker les différents éléments du protocole HTTP. Je les ai présenté (au moins une partie d'entre eux !) plus haut mais je n'ai pas parlé de la politique d'enregistrement de nouvaux éléments. En suivant la terminologie du RFC 8126, il y a entre autres le registre des méthodes <https://www.iana.org/assignments/http-methods/http-methods.xml#methods> (pour ajouter une nouvelle méthode, il faut suivre la politique « Examen par l'IETF », une des plus lourdes), le registre des codes de retour <https://www.iana.org/assignments/http-status-codes/http-status-codes.xml#http-status-codes-1> (même politique), le registre des champs <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names> (désormais séparé de celui des champs du courrier, politique « Spécification nécessaire »), etc.

Ah, et si vous voulez la syntaxe complète de HTTP sous forme d'une grammaire formelle, lisez l'ABNF en annexe A.

Et avec un langage de programmation ? Vu le succès de HTTP et sa présence partout, il n'est pas étonnant que tous les langages de programmation permettent facilement de faire des requêtes HTTP. HTTP, en tout cas HTTP/1, est suffisamment simple pour qu'on puisse le programmer soi-même en appelant les fonctions réseau de bas niveau, mais pourquoi s'embêter ? Utilisons les bibliothèques prévues à cet effet et commençons par le langage Python. D'abord avec la bibliothèque standard http.client <https://docs.python.org/3/library/http.client.html> :

```
conn = http.client.HTTPConnection(HOST)
conn.request("GET", PATH)
result = conn.getresponse()
body = result.read().decode()
```

Et hop, la variable `body` contient une représentation de la ressource demandée (le programme complet est en (en ligne sur `https://www.bortzmeyer.org/files/sample-http-client.py`)). En pratique, la plupart des programmeurs Python utiliseront sans doute une autre bibliothèque standard <https://docs.python.org/3/library/urllib.request.html>, qui n'est pas spécifique à HTTP et permet de traiter des URL quelconques (cela donne le programme (en ligne sur `https://www.bortzmeyer.org/files/sample-http-urllib.py`)). D'encore plus haut niveau (mais pas incluse dans la bibliothèque standard, ce qui ajoute une dépendance à votre programme) est la bibliothèque Requests <https://docs.python-requests.org/>, souvent utilisée (voir par exemple (en ligne sur `https://www.bortzmeyer.org/files/sample-http-requests.py`)).

Ensuite, avec le langage Go. Là aussi, il dispose de HTTP dans sa bibliothèque standard <https://pkg.go.dev/net/http> :

```
response, err := http.Get(Url)
defer response.Body.Close()
body, err := ioutil.ReadAll(response.Body)
```

Le programme complet est (en ligne sur `https://www.bortzmeyer.org/files/sample-http.go`).

Et ici un client en Elixir, utilisant la bibliothèque HTTPoison <https://hexdocs.pm/httpoison/> :

```
HTTPoison.start()
{:ok, result} = HTTPoison.get(@url)
```

La version longue est en .

L'annexe B de notre RFC fait la liste des principaux changements depuis les précédents RFC. Je l'ai dit, le protocole ne change pas réellement mais il y a quand même quelques modifications, notamment des clarifications de textes trop ambigus (par exemple la définition des intervalles). Et bien sûr le gros changement est qu'il y a désormais une définition abstraite de ce qu'est un message HTTP, séparée des définitions concrètes pour les trois versions de HTTP en service. En outre, il y a désormais des recommandations explicites de taille minimale à accepter pour certains élements (par exemple 8 000 octets pour les URI).

HTTP est, comme vous le savez, un immense succès, dû à la place prise par le Web, dont il est le protocole de référence. Le RFC résume l'histoire de HTTP :

— Publié pour la première fois en 1990, au début un protocole ultra-simple, réduit à la méthode `GET`, et qui n'a été documenté qu'après,
— HTTP/1.0 (RFC 1945 en 1996 mais le protocole existait bien avant le RFC) en a fait un protocole bien plus riche (en-tête, plusieurs méthodes), avec indication du type des ressources, et "*virtual hosting*",
— HTTP/1.1 en 1995 (normalisé dans le RFC 2068 à l'origine, et aujourd'hui RFC 9112), au moment de l'explosion de l'usage du Web, a apporté beaucoup d'améliorations, comme le fait que les connexions sont persistentes par défaut,
— HTTP/2 a marqué plusieurs ruptures en 2015 (RFC 7540), avec son parallélisme (les réponses n'arrivent plus forcément dans l'ordre des requêtes) et son encodage binaire, rompant avec la tradition textuelle de beaucoup de protocoles IETF,
— Et enfin HTTP/3 (RFC 9114, en 2022) a abandonné TCP pour QUIC `<https://www.bortzmeyer.org/quic.html>`,