RFC 9112 : HTTP/1.1

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/9112.html

Ce nouveau RFC normalise HTTP/1.1, la plus ancienne version de HTTP encore en service. Il décrit les détails de comment les messages sont représentés sur le réseau, la sémantique de haut niveau étant désormais dans un document séparé, le RFC 9110 [1]. Ensemble, ces deux RFC remplacent le RFC 7230.

HTTP est certainement le protocole Internet le plus connu. Il en existe plusieurs versions, ayant toutes en commun la sémantique normalisée dans le RFC 9110. Les versions les plus récentes, HTTP/2 et HTTP/3 sont loin d'avoir remplacé la version 1, plus précisément 1.1, objet de notre RFC et toujours largement répandue. Un serveur HTTP actuel doit donc gérer au moins cette version. (Par exemple, en octobre 2021, les ramasseurs de Google et Baidu utilisaient toujours exclusivement HTTP/1.1.)

Un des avantages de HTTP/1, et qui explique sa longévité, est que c'est un protocole simple, fondé sur du texte et qu'il est donc relativement facile d'écrire clients et serveurs. D'ailleurs, pour illustrer cet article, je vais prendre exemple sur un simple serveur HTTP/1 que j'ai écrit (le code source complet est disponible ici (en ligne sur https://www.bortzmeyer.org/files/indian.tar.gz)). Le serveur ne gère que HTTP/1 (les autres versions sont plus complexes) et ne vise pas l'utilisation en production : c'est une simple démonstration. Il est écrit en Elixir. Bien sûr, Elixir, comme tous les langages de programmation sérieux, dispose de bibliothèques pour créer des serveurs HTTP (notamment Cowboy <https://www.bortzmeyer.org/cowboy-elixir.html>). Le programme que j'ai écrit ne vise pas à les concurrencer : si on veut un serveur HTTP pour Elixir, Cowboy est un bien meilleur choix ! C'est en référence à Cowboy que mon modeste serveur se nomme Indian.

Commençons par le commencement, la section 1 de notre RFC rappelle les bases de HTTP (décrites plus en détail dans le RFC 9110).

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

La section 2 attaque ce qui est spécifique à la version 1 de HTTP. Avec les URL de plan `http:`, on commence par établir une connexion TCP avec le serveur. Ensuite, un message en HTTP/1 commence par une ligne de démarrage, suivie d'un CRLF (fin de ligne sous la forme des deux octets "*Carriage Return*" et "*Line Feed*"), d'une série d'en-têtes ressemblant à celui de l'IMF du RFC 5322 (par exemple `Accept: text/*`), d'une ligne vide et peut-être d'un corps du message. Les requêtes du client au serveur et les réponses du serveur au client sont toutes les deux des messages, la seule différence étant que, pour la requête, la ligne de démarrage est une ligne de requête et, pour la réponse, c'est une ligne d'état. (Le RFC note qu'on pourrait réaliser un logiciel HTTP qui soit à la fois serveur et client, distinguant requêtes et réponses d'après les formats distincts de ces deux lignes. En pratique, personne ne semble l'avoir fait.)

Pour une première démonstration de HTTP, on va utiliser le module http.server <`https://docs.python.org/3/library/http.server.html#module-http.server`> du langage Python, qui permet d'avoir un serveur HTTP opérationnel facilement :

```
% python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Hop, nous avons un serveur HTTP qui tourne sur le port 8000. On va utiliser curl et son option `-v`, qui permet de voir le dialogue (le ¿ indique ce qu'envoie curl, le ¡ ce qu'il reçoit du serveur en Python) :

```
% curl -v http://localhost:8000/
> GET / HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.68.0
> Accept: */*
>
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/3.8.10
< Date: Thu, 06 Jan 2022 17:24:13 GMT
< Content-type: text/html; charset=utf-8
< Content-Length: 660
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
...
```

La ligne qui commence par `GET` est la ligne de démarrage, ici une requête, curl a envoyé trois lignes d'en-tête. La ligne qui commence par `HTTP/1.0` est la ligne de démarrage de la réponse, et elle est suivie par quatre lignes d'en-tête. La requête n'avait pas de corps, mais la réponse en a un (il commence par `<!DOCTYPE HTML PUBLIC`), ici au format HTML. En dépit du H de son nom, HTTP n'a pas grand'chose de spécifiquement lié à l'hypertexte, et peut être utilisé pour tout type de données (le serveur Indian ne renvoie que du texte brut).

Pour le corps des messages, HTTP utilise certains concepts de MIME (RFC 2045). Mais HTTP n'est pas MIME : l'annexe B détaille les différences.

Le client est censé lire la réponse, commençant par la ligne d'état, puis tout l'en-tête jusqu'à une ligne vide, puis le corps, dont la taille est indiquée par le champ `Content-Length:`, ici 660 octets. (Sans ce

champ, le client va lire jusqu'à la fin de la connexion TCP sous-jacente.) Notez qu'Indian ne fait pas cela bien : il fait une seule opération de lecture et analyse ensuite le résultat (alors qu'il faudra peut-être plusieurs opérations, et que, si on utilise les connexions persistentes, on ne peut découvrir la fin du corps que si on tient compte de `Content-Length:`, ou des délimiteurs de `Transfer-Encxoding: chunked`). Ce choix a été fait pour simplifier l'analyse syntaxique (qui devrait normalement être incrémentale, contrairement à ce que fait Indian, mais la bibliothèque utilisée <https://github.com/dashbitco/nimble_parsec> ne le permet pas, contrairement à, par exemple tree-sitter <https://tree-sitter.github.io/>). Rappelez-vous que ce n'est qu'un programme de démonstration.

Quand la réponse est du texte, le client ne doit pas supposer un encodage particulier, il doit lire des octets, quitte à les convertir dans des concepts de plus haut niveau (comme les caractères) plus tard.

Notez tout de suite qu'on trouve de tout dans le monde HTTP, et que beaucoup de clients et de serveurs ne suivent pas forcément rigoureusement la norme dans ses moindres détails. En général, Indian est plutôt strict et colle à la norme, sauf dans les cas où il était absolument nécessaire d'être plus tolérant pour pouvoir être testé avec les clients que j'ai utilisé. Comme souvent sur l'Internet <https://www.bortzmeyer.org/principe-robustesse.html>, ces déviations par rapport à la norme permettent des attaques rigolotes comme le "*request smuggling*" (section 11.2 du RFC) ou le "*response splitting*" (section 11.1).

La réponse du serveur indique un numéro de version, sous la forme de deux chiffres séparés par un point. Ce RFC spécifie la version 1.1 de HTTP (Indian peut aussi gérer la version 1.0).

Commençons par la requête (section 3 du RFC). Elle commence par une ligne qui comprend la méthode, le chemin et la version de HTTP. Elles sont séparées par un espace. Pour analyser les requêtes, Indian utilise la combinaison d'analyseurs syntaxiques avec NimbleParsec <https://www.bortzmeyer.org/combi-analyseurs-elixir.html>, l'analyseur de la requête est donc : `method |> ignore(string(" ")) |> concat(path) |> ignore(string(" ")) |> concat(version)`. (La norme ne prévoit qu'un seul espace, autrement, on aurait pu prévoir une répétition de `string(" ")`. Le RFC suggère que cette version plus laxiste est acceptable mais peut être dangereuse.) La méthode indique ce que le client veut faire à la ressource désignée. La plus connue des méthodes est `GET` (récupérer la ressource) mais il en existe d'autres, et la liste <https://www.iana.org/assignments/http-methods/http-methods.xml#methods> peut changer. Indian ne met donc pas un choix limitatif mais accepte tout nom de méthode (`method = ascii_string([not: ?\ ], min: 1)`), quitte à vérifier plus tard. La ressource sur laquelle le client veut agir est indiquée par un chemin (ou, dans certains cas par l'URL complet). Ainsi, un client qui veut récupérer `http://www.example.org/truc?machin` va envoyer au serveur au moins :

```
GET /truc?machin HTTP/1.1
Host: www.example.org
```

Il existe d'autres formes pour la requête mais je ne les présente pas ici (lisez le RFC).

La première ligne de la requête est suivie de l'en-tête, composée de plusieurs champs (cf. section 5). Voici la requête que génère wget pour récupérer `https://cis.cnrs.fr/a-travers-les-infrastructures-c-est`

```
% wget -d https://cis.cnrs.fr/a-travers-les-infrastructures-c-est-la-souverainete-numerique-des-etats-qui-se-jou
...
GET /a-travers-les-infrastructures-c-est-la-souverainete-numerique-des-etats-qui-se-joue/ HTTP/1.1
User-Agent: Wget/1.20.3 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: cis.cnrs.fr
Connection: Keep-Alive
```

Une particularité souvent oubliée de HTTP est qu'il n'y a pas de limite de taille à la plupart des éléments du protocole. Les programmeurs se demandent souvent « quelle place dois-je réserver pour tel élément ? » et la réponse est souvent qu'il n'y a pas de limite, juste des indications. Par exemple, notre RFC dit juste qu'il faut accepter des lignes de requête de 8 000 octets au moins.

Le serveur répond avec une ligne d'état et un autre en-tête (section 4). La ligne d'état comprend la version de HTTP, un code de retour formé de trois chiffres, et un message facultatif (là encore, avec un espace comme séparateur). Voici par exemple la réponse d'Indian :

```
HTTP/1.1 200
Content-Type: text/plain
Content-Length: 18
Server: myBeautifulServerWrittenInElixir
```

Le message est d'autant plus facultatif (Indian n'en met pas) qu'il n'est pas forcément dans la langue du destinataire et qu'il n'est pas structuré, donc pas analysable. Le RFC recommande de l'ignorer.

Beaucoup plus important est le code de retour. Ces trois chiffres indiquent si tout s'est bien passé ou pas. Ils sont décrits en détail dans le RFC 9110, section 15. Bien qu'il s'agisse normalement d'éléments de protocole, certains sont bien connus des utilisatrices et utilisateurs, comme le célèbre 404. Et ils ont une représentation en chats <https://http.cat/> et on a proposé de les remplacer par des émojis <https://www.bortzmeyer.org/http-code-emoji.html>.

L'en-tête, maintenant (section 5 du RFC). Il se compose de plusieurs lignes, chacune comportant le nom du champ, un deux-points (pas d'espace avant ce deux-points, insiste le RFC), puis la valeur du champ. Cela s'analyse dans Indian avec `header_line = header_name |> ignore(string(":")) |> ignore(repeat(string(" "))) |> concat(header_value) |> ignore(eol)`. Les noms de champs possibles sont dans un registre IANA <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names> (on peut noter qu'avant ce RFC, ils étaient mêlés aux champs du courrier électronique dans un même registre).

Après les en-têtes, le corps. Il est en général absent des requêtes faites avec la méthode `GET` mais il est souvent présent pour les autres méthodes, et il est en général dans les réponses. Ici, une réponse d'un serveur avec le corps en JSON :

```
% curl -v https://atlas.ripe.net/api/v2/measurements/34762605/results/
...
< HTTP/1.1 200 OK
< Server: nginx
< Date: Tue, 11 Jan 2022 20:19:31 GMT
< Content-Type: application/json
< Transfer-Encoding: chunked
...
[{"fw":5020,"mver":"2.2.0","lts":4,"resultset":[{"time":1641657433,"lts":4,"subid":1,"submax":1,"dst_addr":"
```

Le champ `Content-Length:` est normalement obligatoire dans la réponse, sauf s'il y a un champ `Transfer-Encoding:`, comme ici. Il permet au client de gérer sa mémoire, et de savoir s'il a bien tout récupéré. (Avec TLS, si on reçoit un signal de fin de l'application, on sait qu'on a toute les données mais, sans TLS, on ne pourrait pas être sûr, s'il n'y avait ce `Content-Length:`.)

HTTP/1.1 est un protocole simple (quoiqu'il y ait un certain nombre de pièges pour une mise en œuvre réelle) et on peut donc se contenter de telnet comme client HTTP :

```
% telnet evil.com 80
Trying 66.96.146.129...
Connected to evil.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: evil.com

HTTP/1.1 200 OK
Date: Sun, 16 Jan 2022 11:31:05 GMT
Content-Type: text/html
Content-Length: 4166
Connection: keep-alive
Server: Apache/2
Last-Modified: Sat, 15 Jan 2022 23:21:33 GMT
Accept-Ranges: bytes
Cache-Control: max-age=3600
Etag: "1046-5d5a72e24309e"
Expires: Sun, 16 Jan 2022 12:14:45 GMT
Age: 980

<HTML>
<HEAD>
   <meta content="Microsoft FrontPage 6.0" name="GENERATOR">
   <meta content="FrontPage.Editor.Document" name="ProgId">
```

Les lignes `GET / HTTP/1.1` et `Host: evil.com` ont été tapées à la main, une fois telnet connecté. HTTP/1.1 (contrairement aux versions 2 et 3) fait partie de ces protocoles en texte, qu'on peut déboguer à la main avec telnet.

En plus perfectionné que telnet, il y a netcat :

```
% echo   -n "GET /hello HTTP/1.1\r\nConnection: close\r\n\r\n" | nc ip6-localhost 8080
HTTP/1.1 200
Content-Type: text/plain
Content-Length: 12
Server: myBeautifulServerWrittenInElixir

Hello, ::1!
```

On a dit plus haut que HTTP/1.1 fonctionnait au-dessus d'une connexion TCP. La section 9 de notre RFC décrit la gestion de cette connexion. (En HTTP 0.9, c'était simple, une transaction = une connexion, mais ça a changé avec HTTP 1.) HTTP n'a pas forcément besoin de TCP (d'ailleurs, HTTP/3 fonctionne sur QUIC <https://www.bortzmeyer.org/quic.html>), il lui faut juste une liaison fiable faisant passer les octets dans l'ordre et sans perte. Dans HTTP/1.1, c'est TCP qui fournit ce service. (Avec TLS si on fait du HTTPS.) L'établissement d'une connexion TCP prend du temps, et la latence <https://www.bortzmeyer.org/latence.html> est un des plus gros ennemis de HTTP. Il est donc recommandé de ne pas établir une connexion TCP par transaction HTTP, mais de réutiliser les connexions. Le problème est délicat car le serveur peut avoir envie de supprimer des connexions pour récupérer des ressources. Clients et serveurs doivent donc s'attendre à des comportements variés de la part de leur partenaire.

HTTP/1 n'a pas d'identificateur de requête (comme a, par exemple, le DNS). Les transactions doivent donc se faire dans l'ordre : si on envoie une requête A puis une requête B sur la même connexion TCP, on recevra forcément la réponse A puis la B. (HTTP/2 et encore plus HTTP/3 ont par contre une certaine dose de parallélisme.) Les connexions sont persistentes par défaut dans HTTP/1.1 (ce n'était pas le cas

en HTTP/1.0) et des champs de l'en-tête servent à contrôler cette persistence (`Connection: close` indique qu'on ne gardera pas la connexion ouverte, et un client poli qui ne fait qu'une requête doit envoyer ce champ). Dans le code source d'Indian, les accès à `context["persistent-connection"]` vous montreront la gestion de connexion.

Si le client et le serveur gère les connexions persistentes, le client peut aussi envoyer plusieurs requêtes à la suite, sans attendre les réponses (ce qu'on nomme le "*pipelining*"). Les réponses doivent parvenir dans le même ordre (puisqu'il n'y a pas d'identificateur de requête, qui permettrait de les associer à une requête), donc HTTP/1.1 ne permet pas un vrai parallélisme.

Pour économiser les ressources du serveur, un client ne devrait pas ouvrir « trop » de connexions vers un même serveur. (Le RFC 2616, section 8.1.4, mettait une limite de 2 connexions mais cette règle a disparu par la suite.)

Jusqu'à présent, on a parlé de HTTP tournant directement sur TCP. Mais cela fait passer toutes les données en clair, ce qui est inacceptable du point de vue sécurité, dans un monde de surveillance massive. Aujourd'hui, la grande majorité des connexions HTTP passent sur TLS, un mécanisme cryptographique qui assure notamment la confidentialité et l'authentification du serveur. HTTPS (HTTP sur TLS) était autrefois normalisé dans le RFC 2818 mais qui a désormais été abandonné au profit du RFC 9110 et de notre RFC 9112. Le principe pour HTTP/1.1 est simple : une fois la connexion TCP établie, le client HTTP démarre une session TLS (RFC 8446) par dessus et voilà. (L'ALPN à utiliser est `http/1.1`.) Lors de la fermeture de la connexion, TLS envoie normalement un message qui permet de différencier les coupures volontaires et les pannes (`close_notify`, RFC 8446, section 6.1). (Indian ne gère pas TLS, si on veut le sécuriser - mais ce n'est qu'un programme de démonstration, il faut le faire tourner derrière stunnel ou équivalent.)

Pour tester HTTPS à la main, on peut utiliser un programme distribué avec GnuTLS, ici pour récupérer `https://fr.wikipedia.org/wiki/Hunga_Tonga` :

```
% gnutls-cli fr.wikipedia.org
...
Connecting to '2620:0:862:ed1a::1:443'...
 - subject `CN=*.wikipedia.org,O=Wikimedia Foundation\, Inc.,L=San Francisco,ST=California,C=US', issuer `CN
...
- Simple Client Mode:

GET /wiki/Hunga_Tonga HTTP/1.1
Host: fr.wikipedia.org
Connection: close

HTTP/1.1 200 OK
Date: Sun, 16 Jan 2022 20:41:56 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 79568
...

<!DOCTYPE html>
<html class="client-nojs" lang="fr" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Hunga Tonga | Wikipédia</title>
...
```

Les trois lignes commençant par `GET` ont été tapées à la main par l'utilisateur.

La section 10 de notre RFC traite d'une fonction plus rare : l'inclusion d'un message HTTP comme donnée d'un protocole (qui peut être HTTP ou un autre). Un tel message est étiqueté avec le type MIME `application/http`.

Quelques mots sur la sécurité pour finir (section 11) : en raison de la complexité du protocole (qui est moins simple qu'il n'en a l'air !) et des mauvaises mises en œuvre qu'il faut quand même gérer car elles sont largement présentes sur le Web, deux programmes peuvent interpréter la même session HTTP différemment. Cela permet par exemple l'attaque de "*response splitting*" (cf. l'article de Klein « "*Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics*" <https://packetstormsecurity.com/papers/general/whitepaper_httpresponse.pdf> »). Autre attaque possible, le "*request smuggling*" (cf. l'article de Linhart, Klein, Heled et Orrin, « "*HTTP Request Smuggling*" <https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf> »).

Notre section 11 rappelle aussi que HTTP tout seul ne fournit pas de mécanisme pour assurer l'intégrité et la confidentialité des communications. Il dépend pour cela d'un protocole sous-jacent, en pratique TLS (HTTP+TLS étant appelé HTTPS).

L'annexe C décrit les changements de HTTP jusqu'à cette version 1.1. Ainsi, HTTP/1.0 a introduit la notion d'en-têtes, qui a permis, entre autres, le "*virtual hosting*", grâce au champ `Host:`. HTTP/1.1 a notamment changé la persistence par défaut des connexions (de non-persistente à désormais persistente). Et notre RFC, par rapport à la précédente norme de HTTP/1.1, le RFC 7230 ? Le plus gros changement est éditorial, toutes les parties indépendantes du numéro de version de HTTP ont été déplacées vers le RFC 9110, notre RFC ne gardant que ce qui est spécifique à HTTP/1.1. S'il y a beaucoup de changements de détail, le protocole n'est pas modifié, un client ou un serveur HTTP/1.1 reste compatible.

Vous noterez que j'ai fait un cours HTTP au CNAM, dont les supports et la vidéo sont disponibles <https://www.bortzmeyer.org/cours-http-cnam.html>. HTTP/1 est un protocole simple, très simple, et trivial à programmer. Cela en fait un favori des enseignants en informatique car on peut écrire un client (ou même un serveur) HTTP très facilement, et il peut être utilisé contre des serveurs (ou des clients) réels, ce qui est motivant pour les étudiant[Caractère Unicode non montré [2]]es.

2. Car trop difficile à faire afficher par LaTeX