RFC 7540 : Hypertext Transfer Protocol version 2

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 16 mai 2015. Dernière mise à jour le 10 octobre 2021

Date de publication du RFC : Mai 2015

https://www.bortzmeyer.org/7540.html

Le protocole HTTP, à la base des échanges sur le Web, a connu plusieurs versions successives, la première était 0.9, la suivante 1.0 et la plus courante aujourd'hui est la 1.1. L'évolution entre ces versions était progressive. Mais HTTP/2, dont la norme vient de sortir, marque un saut plus important : s'il garde la sémantique de HTTP (ses en-têtes, ses codes de retour, etc), il a désormais un encodage binaire et un transport spécifique (binaire, multiplexé, avec possibilité de "*push*"). Fini de déboguer des serveurs HTTP avec telnet. En échange, le nouveau protocole promet d'être plus rapide, notamment en diminuant la latence <https://www.bortzmeyer.org/latence.html> lors des échanges. La norme de cette version a depuis été mise à jour dans le RFC 9113[1] . Oubliez donc le RFC 7540.

C'est un groupe de travail IETF, nommé httpbis <https://tools.ietf.org/wg/httpbis>, qui a mis au point cette nouvelle version (voir son site Web <http://http2.github.io/>). Ce groupe httpbis travaillait sur deux projets parallèles : d'abord réorganiser les normes sur HTTP 1.1 pour corriger les innombrables bogues, et mieux organiser la norme, sans changer le protocole (projet conclus en juin 2014 <https://www.bortzmeyer.org/http-11-reecrit.html>). Et un deuxième projet, HTTP version 2, connu aussi sous le nom de code de SPDY (initialement lancé par Google). Si SPDY a changé de nom, on peut toutefois prévoir que cet acronyme apparaitra pendant encore un certain temps, dans les API, codes source, messages d'erreur...

La section 1 de notre nouveau RFC résume les griefs qu'il y avait contre HTTP 1.1, normalisé dans le RFC 7230 :

— Une seule requête au plus en attente sur une connexion TCP donnée. Cela veut dire que, si on a deux requêtes à envoyer au même serveur, et que l'une est lente et l'autre rapide, il faudra faire deux connexions TCP (la solution la plus courante), ou bien se résigner au risque que la lente bloque la rapide. (La section 6.3.2 du RFC 7230 permettait d'envoyer la seconde requête tout de suite mais les réponses devaient être dans l'ordre des requêtes, donc pas moyen pour une requête rapide de doubler une lente.)

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9113.txt

— Un encodage des en-têtes inefficace, trop bavard et trop redondant.

Au contraire, HTTP/2 n'utilisera toujours qu'une seule connexion TCP, de longue durée (ce qui sera plus sympa pour le réseau). L'encodage étant entièrement binaire, le traitement par le récepteur sera plus rapide.

La section 2 de notre RFC résume l'essentiel de ce qu'il faut savoir sur HTTP/2. Il garde la sémantique de HTTP 1 (donc, par exemple, un `GET` de `/cetteressourcenexistepas` qui faisait un 404 avant le fera toujours après) mais avec une représentation sur le réseau très différente. On peut résumer en disant que HTTP/2 ne change que le transport des messages, pas les messages ou leurs réponses.

Avec HTTP/2, l'unité de base de la communication est la **trame** ("*frame*", et, dans HTTP 2, vous pouvez oublier la définition traditionnelle qui en fait l'équivalent du paquet, mais pour la couche 2). Chaque trame a un type et, par exemple, les échanges HTTP traditionnels se feront avec simplement une trame `HEADERS` en requête et une `DATA` en réponse. Certains types sont spécifiques aux nouvelles fonctions de HTTP/2, comme `SETTINGS` ou `PUSH_PROMISE`.

Les trames voyagent ensuite dans des ruisseaux ("*streams*"), chaque ruisseau hébergeant un et un seul échange requête/réponse. On crée donc un ruisseau à chaque fois qu'on a un nouveau `GET` ou `POST` à faire. Les petits ruisseaux sont ensuite multiplexés dans une grande rivière, l'unique connexion TCP entre un client HTTP et un serveur. Les ruisseaux ont des mécanismes de contrôle du trafic et de prioritisation entre eux.

Les en-têtes sont comprimés, en favorisant le cas le plus courant, de manière à s'assurer, par exemple, que la plupart des requêtes HTTP tiennent dans un seul paquet de la taille des paquets Ethernet.

Bon, maintenant, les détails pratiques (le RFC fait 92 pages). D'abord, l'établissement de la connexion. HTTP/2 tourne au-dessus de TCP. Comment on fait pour commencer du HTTP/2 ? On utilise un nouveau port, succédant au 80 de HTTP ? (Non. Les ports sont les mêmes, 80 et 443.) On regarde dans le DNS ou ailleurs si le serveur sait faire du HTTP/2 ? (Non plus.) Il y a deux méthodes utilisées par les clients HTTP/2. Tout dépend de si on fait du TLS ou pas. Si on fait du TLS, on va utiliser ALPN (RFC 7301), en indiquant l'identificateur `h2` (HTTP/2 sur TLS). Le serveur, recevant l'extension ALPN avec `h2`, saura ainsi qu'on fait du HTTP/2 et on pourra tout de suite commencer l'échange de trames HTTP/2. (`h2` est désormais dans le registre IANA `<https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids>`.) Si on ne fait pas de TLS (identificateur `h2c` pour « "*HTTP/2 in clear*" » mais il est juste réservé pour information puisqu'on ne peut pas utiliser ALPN si on n'utilise pas TLS), on va passer par l'en-tête HTTP `Upgrade:` (section 6.7 du RFC 7230) :

```
GET / HTTP/1.1
Host: www.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: [base64url encoding of HTTP/2 SETTINGS payload]
```

Si le serveur est un vieux serveur qui ne gère pas HTTP/2, il ignorera le `Upgrade:` :

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
```

Si le serveur est d'accord pour faire de l'HTTP/2, il répondra 101 ("*Switching Protocols*" `<https://flic.kr/p/aXXExP>`) :

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[Tout ce qui suit est en HTTP/2]
```

Dans ce dernier cas, la première trame envoyée par le serveur sera de type `SETTINGS`. Rappelez-vous que chaque couple requête/réponse HTTP aura lieu dans un ruisseau séparé. Ici, le ruisseau numéro 1 aura été créé implicitement, et servira à la requête `Upgrade:`.

Une fois qu'un client aura réussi à établir une connexion avec un serveur en HTTP/2, il sait que le serveur gère ce protocole. Il peut s'en souvenir, pour les futures connexions (mais attention, ce n'est pas une indication parfaite : un serveur peut abandonner HTTP/2, par exemple).

Maintenant, c'est parti, on s'envoie des trames. À quoi ressemblent-elles (section 4)? Elles commencent par un en-tête indiquant leur longueur, leur type (comme `SETTINGS`, `HEADERS`, `DATA`... cf. section 6), des options (comme `ACK` qui sert aux trames de type `PING` à distinguer requête et réponse) et l'identificateur du ruisseau auquel la trame appartient (un nombre sur 31 bits). Le format complet est en section 4.1.

Les en-têtes HTTP sont comprimés selon la méthode normalisée dans le RFC 7541.

Les ruisseaux ("*streams*"), maintenant. Ce sont donc des suites **ordonnées** de trames, bi-directionnelles, à l'intérieur d'une connexion HTTP/2. Une connexion peut comporter plusieurs ruisseaux, chacun identifié par un "*stream ID*" (un entier de quatre octets, pair si le ruisseau a été créé par le serveur et impair autrement). Les ruisseaux sont ouverts et fermés dynamiquement et leur durée de vie n'est donc pas celle de la connexion HTTP 2. Contrairement à TCP, il n'y a pas de « triple poignée de mains » : l'ouverture d'un ruisseau est unilatérale et peut donc se faire très vite (rappelez-vous que chaque échange HTTP requête/réponse nécessite un ruisseau qui lui est propre ; pour vraiment diminuer la latence, il faut que leur création soit rapide).

Un mécanisme de contrôle du flot s'assure que les ruisseaux se partagent pacifiquement la connexion. C'est donc une sorte de TCP dans le TCP, réinventé pour les besoins de HTTP/2 (section 5.2 et relire aussi le RFC 1323). Le récepteur indique (dans une trame `WINDOWS_UPDATE`) combien d'octets il est prêt à recevoir (64 Kio par défaut) et l'émetteur s'arrête dès qu'il a rempli cette fenêtre d'envoi. (Plus exactement, s'arrête d'envoyer des trames `DATA` : les autres, les trames de contrôle, ne sont pas soumises au contrôle du flot).

Comme si ce système des connexions dans les connexions n'était pas assez compliqué comme cela, il y a aussi des dépendances entre ruisseaux. Un ruisseau peut indiquer qu'il dépend d'un autre et, dans ce cas, les ressources seront allouées d'abord au ruisseau dont on dépend. Par exemple, le code JavaScript ne peut en général commencer à s'exécuter que quand toute la page est chargée, et on peut donc le demander dans un ruisseau dépendant de celle qui sert à charger la page. On peut dépendre d'un ruisseau dépendant, formant ainsi un arbre de dépendances.

Il peut bien sûr y avoir des erreurs dans la communication. Certaines affectent toute la connexion, qui devra être abandonnée, mais d'autres ne concernent qu'un seul ruisseau. Dans le premier cas, celui qui détecte l'erreur envoie une trame `GOAWAY` (dont on ne peut pas garantir qu'elle sera reçue, puisqu'il y a une erreur) puis coupe la connexion TCP. Dans le second cas, si le problème ne concerne qu'un seul ruisseau, on envoie la trame `RST_STREAM` qui arrête le traitement du ruisseau.

Notre section 5 se termine avec des règles qui indiquent comment gérer des choses inconnues dans le dialogue. Ces règles permettent d'étendre HTTP/2, en s'assurant que les vieilles mises en œuvre ne pousseront pas des hurlements devant les nouveaux éléments qui circulent. Par exemple, les trames d'un type inconnu doivent être ignorées et mises à la poubelle directement, sans protestation.

On a déjà parlé plusieurs fois des trames, la section 6 du RFC détaille leur définition. Ce sont aux ruisseaux ce que les paquets sont à IP et les segments à TCP. Les trames ont un type (un entier d'un octet). Les types possibles sont enregistrés à l'IANA <https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml#frame-type>. Les principaux types actuels sont :

— DATA (type 0), les trames les plus nombreuses, celles qui portent les données, comme les pages HTML,
— HEADERS (type 1), qui portent les en-têtes HTTP, dûment comprimés selon le RFC 7541,
— PRIORITY (type 2) indique la priorité que l'émetteur donne au ruisseau qui porte cette trame,
— RST_STREAM (type 3), dont j'ai parlé plus haut à propos des erreurs, permet de terminer un ruisseau (filant la métaphore, on pourrait dire que cela assèche le ruisseau ?),
— SETTINGS (type 4), permet d'envoyer des paramètres, comme SETTINGS_HEADER_TABLE_SIZE, la taille de la table utilisée pour la compression des en-têtes, SETTINGS_MAX_CONCURRENT_STREAMS pour indiquer combien de ruisseaux est-on prêt à gérer, etc (la liste des paramètres est dans un registre IANA <https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml#settings>),
— PUSH_PROMISE (type 5) qui indique qu'on va transmettre des données non sollicitées ("*push*"), du moins si le paramètre SETTINGS_ENABLE_PUSH est à 1,
— PING (type 6) qui permet de tester le ruisseau (le partenaire va répondre avec une autre trame PING, ayant l'option ACK à 1),
— GOAWAY (type 7) que nous avons déjà vu plus haut, sert à mettre fin proprement (le pair est informé de ce qui va se passer) à une connexion,
— WINDOW_UPDATE (type 8) sert à faire varier la taille de la fenêtre (le nombre d'octets qu'on peut encore accepter, cf. section 6.9.1),
— CONTINUATION (type 9), indique la suite d'une trame précédente. Cela n'a de sens que pour certains types comme HEADERS (ils peuvent ne pas tenir dans une seule trame) ou CONTINUATION lui-même. Mais une trame CONTINUATION ne peut pas être précédée de DATA ou de PING, par exemple.

Dans le cas vu plus haut d'erreur entrainant la fin d'un ruisseau ou d'une connexion entière, il est nécessaire d'indiquer à son partenaire en quoi consistait l'erreur en question. C'est le rôle des codes d'erreur de la section 7. Stockés sur quatre octets (et enregistrés dans un registre IANA <https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml#error-code>), ils sont transportés par les trames RST_STREAM ou GOAWAY qui terminent, respectivement, ruisseaux et connexions. Parmi ces codes :

— NO_ERROR (code 0), pour les cas de terminaison normale,
— PROTOCOL_ERROR (code 1) pour ceux où le pair a violé une des règles de HTTP/2, par exemple en envoyant une trame CONTINUATION qui n'était pas précédée de HEADERS, PUSH_PROMISE ou CONTINUATION,
— INTERNAL_ERROR (code 2), un malheur est arrivé,
— ENHANCE_YOUR_CALM (code 11), qui ravira les amateurs de spam et de Viagra, demande au partenaire en face de se calmer un peu, et d'envoyer moins de requêtes.

Toute cette histoire de ruisseaux, de trames, d'en-têtes comprimés et autres choses qui n'existaient pas en HTTP 1 est bien jolie mais HTTP/2 n'a pas été conçu comme un remplacement de TCP, mais comme un moyen de faire passer des dialogues HTTP. Comment met-on les traditionnelles requêtes/réponses HTTP sur une connexion HTTP/2 ? La section 8 répond à cette question. D'abord, il faut se rappeler que HTTP/2 reste du HTTP. L'essentiel des RFC « sémantiques » HTTP, à savoir les RFC 7231, RFC 7232 et les suivants s'applique toujours. Le RFC 7230 reste partiellement applicable : la sémantique est la même mais son expression change. Certains en-têtes disparaissent comme Connection: qui n'est plus utile en HTTP/2.

HTTP reste requête/réponse. Pour envoyer une requête, on utilise un nouveau ruisseau (envoi d'une trame avec un numéro de ruisseau non utilisé), sur laquelle on lira la réponse (les ruisseaux ne sont pas persistents). Dans le cas le plus fréquent, la requête sera composée d'une trame `HEADERS` contenant les en-têtes (comme `User-Agent:` ou `Host:`, cf. RFC 7230, section 3.2) et les « pseudo-en-têtes » comme la méthode (`GET`, `POST`, etc), avec parfois des trames `DATA` (cas d'un `POST`). La réponse comprendra une trame `HEADERS` avec les en-têtes (comme `Content-Length:`) et les pseudo-en-têtes comme le code de retour HTTP (200, 403, 500, etc) suivie de plusieurs trames `DATA` contenant les données (HTML, CSS, images, etc). Des variantes sont possibles (par exemple, les trames `HEADERS` peuvent être suivies de trames `CONTINUATION`). Les en-têtes ne sont pas transportés sous forme texte (ce qui était le cas en HTTP 1, où on pouvait utiliser telnet comme client HTTP) mais encodés selon le RFC 7541. À noter que cet encodage implique une mise du nom de l'en-tête en minuscules.

J'ai parlé plus haut des pseudo-en-têtes : c'est le mécanisme HTTP 2 pour traiter des informations qui ne sont pas des en-têtes HTTP 1. Ces informations sont mises dans les `HEADERS` HTTP/2, précédés d'un deux-points. C'est le cas de la méthode (RFC 7231, section 4), donc `GET` sera encodé `:method get`. L'URL sera éclaté dans les pseudo-en-têtes `:scheme`, `:path`, etc. Idem pour la réponse HTTP, le fameux code à trois lettres est désormais un pseudo-en-tête, `:status`.

Voici un exemple (mais vous ne le verrez pas ainsi si vous espionnez le réseau, en raison de la compression du RFC 7541) :

```
### HTTP 1, pas de corps dans la requête ###
GET /resource HTTP/1.1
Host: example.org
Accept: image/jpeg

### HTTP/2 (une trame HEADERS)
:method = GET
:scheme = https
:path = /resource
host = example.org
accept = image/jpeg
```

Puis une réponse qui n'a pas de corps :

```
### HTTP 1 ###
HTTP/1.1 304 Not Modified
ETag: "xyzzy"
Expires: Thu, 23 Jan ...

### HTTP/2, une trame HEADERS ###
:status = 304
etag = "xyzzy"
expires = Thu, 23 Jan ...
```

Une réponse plus traditionnelle, qui inclut un corps :

```
### HTTP 1 ###
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 123

{binary data}
```

```
### HTTP/2 ###
# trame HEADERS
:status = 200
content-type = image/jpeg
content-length = 123

# trame DATA
{binary data}
```

Plus compliqué, un cas où les en-têtes de la requête ont été mis dans deux trames, et où il y avait un corps dans la requête :

```
### HTTP 1 ###
POST /resource HTTP/1.1
Host: example.org
Content-Type: image/jpeg
Content-Length: 123

{binary data}

### HTTP/2 ###
# trame HEADERS
:method = POST
:path = /resource
:scheme = https

# trame CONTINUATION
content-type = image/jpeg
host = example.org
content-length = 123

# trame DATA
{binary data}
```

Nouveauté de HTTP/2, la possibilité pour le serveur de pousser (”*push*”, section 8.2 de notre RFC) du contenu non sollicité vers le client (sauf si cette possibilité a été coupée par le paramètre `SETTINGS_ENABLE_PUSH`). Pour cela, le serveur (et lui seul) envoie une trame de type `PUSH_PROMISE` au client, en utilisant le ruisseau où le client avait fait une demande originale (donc, la sémantique de `PUSH_PROMISE` est « je te promets que lorsque le moment sera venu, je répondrai plus longuement à ta question »). Cette trame contient une requête HTTP. Plus tard, lorsque le temps sera venu, le serveur tiendra sa promesse en envoyant la « réponse » de cette « requête » sur le ruisseau qu'il avait indiqué dans le `PUSH_PROMISE`.

Et enfin, à propos des méthodes HTTP 1 et de leur équivalent en HTTP/2, est-ce que `CONNECT` (RFC 7231, section 4.3.6) fonctionne toujours ? Oui, on peut l'utiliser pour un tunnel sur un ruisseau. (Un tunnel sur un ruisseau... Beau défi pour le génie civil.)

La section 9 de notre RFC rassemble quelques points divers. Elle rappelle que, contrairement à HTTP 1, toutes les connexions sont persistentes et que le client n'est pas censé les fermer avant d'être certain qu'il n'en a plus besoin. Tout doit passer à travers une connexion vers le serveur et les clients ne doivent plus utiliser le truc d'ouvrir plusieurs connexions HTTP avec le serveur. De même, le serveur laisse les connexions ouvertes le plus longtemps possible, mais a le droit de les fermer s'il doit économiser des ressources.

À noter qu'on peut utiliser une connexion prévue pour un autre nom, du moment que cela arrive au même serveur (même adresse IP). Le pseudo-en-tête `:authority` sert à départager les requêtes allant à

chacun des serveurs. Mais attention si la session utilise TLS ! L'utilisation d'une connexion avec un autre `:authority` ("*host*" + port) n'est possible que si le certificat serveur qui a été utilisé est valable pour tous (par le biais des `subjectAltName`, ou bien d'un joker).

À propos de TLS (certainement un des points les plus chaudement disputés à l'IETF dans les discussions sur HTTP/2), la section 9.2 prévoit quelques règles qui n'existaient pas en HTTP 1 (et dont la violation peut entrainer la coupure de la connexion avec l'erreur `INADEQUATE_SECURITY`) :

— TLS 1.2 (RFC 5246), minimum,
— Gestion de SNI (RFC 6066) obligatoire,
— Compression coupée (RFC 3749), comme indiqué dans le RFC 7525 (permettre la compression de données qui mêlent informations d'authentification comme les "*cookies*", et données contrôlées par l'attaquant, permet certaines attaques comme BREACH) ce qui n'est pas grave puisque HTTP a de meilleures capacités de compression (voir aussi la section 10.6),
— Renégociation <https://www.bortzmeyer.org/tls-renego.html> coupée, ce qui empêche de faire une renégociation en réponse à une certaine requête (pas de solution dans ce cas) et peut conduire à couper une connexion si le mécanisme de chiffrement sous-jacent ne permet pas d'encoder plus de N octets sans commencer à faire fuiter de l'information.
— Très sérieuse limitation du nombre d'algorithmes de chiffrement acceptés (voir l'annexe A pour une liste complète), en éliminant les algorithmes trop faibles cryptographiquement (comme les algorithmes « d'exportation » utilisés dans la faille FREAK). Peu d'algorithmes restent utilisables après avoir retiré cette liste !

TLS avait été au cœur d'un vigoureux débat : faut-il essayer systématiquement TLS, même sans authentification, même si l'URL est de plan `http:` et pas `https:` ? Ce « chiffrement opportuniste » avait été proposé à l'IETF, pour diminuer l'efficacité de la surveillance massive (RFC 7258). C'était le ticket #314 <https://github.com/http2/http2-spec/issues/314> du groupe de travail, et ce fut un des plus disputés. Quelques articles sur ce sujet du « tentons TLS à tout hasard » : <http://bitsup.blogspot.fr/2013/08/ssl-everywhere-for-http2-new-hope.html>,<http://it.slashdot.org/story/13/11/13/1938207/http-20-may-be-ssl-only>,<http://www.pcworld.com/article/2061189/next-gen-http-2-0-protocol-will-require-https-encryption-most-of-the-time.html>,<https://www.tbray.org/ongoing/When/201x/2013/11/05/IETF-88-HTTP-Security>, <http://www.mnot.net/blog/2014/01/04/strengthening_http_a_personal_view> ou <http://lists.w3.org/Archives/Public/ietf-http-wg/2013OctDec/0625.html>. Il faut préciser que l'IETF n'a pas de pouvoirs de police : même si le RFC sur HTTP/2 avait écrit en gros « le chiffrement est obligatoire », rien ne garantit que cela aurait été effectif. Finalement, la décision a été de ne pas imposer ce passage en TLS, mais la question reste en discussion à l'IETF pour le plus grand déplaisir de ceux qui voudraient espionner le trafic plus facilement.

Puisqu'on parle de sécurité, la section 10 traite un certain nombre de problèmes de sécurité de HTTP/2. Parmi ceux qui sont spécifiques à HTTP/2, on note que ce protocole demande plus de ressources que HTTP 1, ne serait-ce que parce qu'il faut maintenir un état pour la compression. Il y a donc potentiellement un risque d'attaque par déni de service. Une mise en œuvre prudente veillera donc à limiter les ressources allouées à chaque connexion.

Enfin, il y a la question de la vie privée, un sujet chaud dans le monde HTTP depuis longtemps. Les options spécifiques à HTTP/2 (changement de paramètres, gestion du contrôle de flot, traitement des innombrables variantes du protocole) peuvent permettre d'identifier une machine donnée par son comportement. HTTP/2 facilite donc le "*fingerprinting*".

En outre, comme une seule connexion TCP est utilisée pour toute une visite sur un site donné, cela peut rendre explicite une information comme « le temps passé sur un site », information qui était implicite en HTTP 1, et qui devait être reconstruite.

Question mises en œuvre de la nouvelle version de HTTP, un bon point de départ est la liste du groupe de travail <https://github.com/http2/http2-spec/wiki/Implementations>. nginx avait décrit ses plans <http://nginx.com/blog/how-nginx-plans-to-support-http2/> et a désormais une mise en œuvre de HTTP/2 <http://mailman.nginx.org/pipermail/nginx-devel/2015-September/007328.html>. Pour le cas particulier des navigateurs Web, il y a un joli tableau <http://caniuse.com/http2>.

curl a désormais HTTP/2 <https://curl.haxx.se/docs/http2.html> :

```
% curl -v --http2 https://www.ietf.org/
* Connected to www.ietf.org (2400:cb00:2048:1::6814:55) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* ALPN, server accepted to use http/1.1
> GET / HTTP/1.1
> Host: www.ietf.org
> User-Agent: curl/7.52.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 16 May 2017 14:25:47 GMT
< Content-Type: text/html
< Transfer-Encoding: chunked
< Connection: keep-alive
< Set-Cookie: __cfduid=d980f0f518518e9abb9e786531b1cf7ed1494944747; expires=Wed, 16-May-18 14:25:47 GMT; pat
< Last-Modified: Tue, 02 May 2017 20:16:22 GMT
```

Désolé de ne pas vous montrer un joli pcap de HTTP/2 mais la plupart des sites accessibles en HTTP/2 (et parfois des clients <https://wiki.mozilla.org/Networking/http2>, « *"Firefox will only be implementing HTTP/2 over TLS"* ») imposent TLS, ce qui rend un peu plus compliquée l'analyse. Je n'ai pas encore trouvé de pcap HTTP/2 sur pcapr <https://www.bortzmeyer.org/pcapr.html> non plus, mais il y en a à la fin de la page <http://daniel.haxx.se/http2/>.

Et, sinon, si vous voulez activer HTTP/2 sur un serveur Apache, c'est aussi simple que de charger le module `http2` et de configurer :

```
Protocols h2 h2c http/1.1
```

Sur Debian, la commande `a2enmod http2` fait tout cela automatiquement. Pour vérifier que cela a bien été fait, vous pouvez utiliser `curl -v` comme vu plus haut, ou bien un site de test (comme KeyCDN <https://tools.keycdn.com/http2-test>) ou encore la fonction "*Inspect element*" (clic droit sur la page, puis onglet "*Network*" puis sélectionner une des ressources chargées) de Firefox :

Quelques articles pertinents :

— Sur le passé : l'annonce originale de SPDY <http://blog.chromium.org/2009/11/2x-faster-web.html>, un article historique sur les débuts de la normalisation <http://www.journaldunet.com/developpeur/client-web/http-2-0-spdy-de-google-1212.shtml>, et une discussion de l'époque sur LinuxFr <https://linuxfr.org/users/julien04/journaux/avec-spdy-goC3%A9l%C3%A9rer-remplaceracc%C3%A9l%C3%A9rer-http> (et une autre <https://linuxfr.org/users/tankey/journaux/spdy>, pendant qu'on y est).

— Sur l'approbation de la norme : le message de félicitations du W3C <http://www.w3.org/blog/news/archives/4400>, un du président du groupe de travail IETF <https://www.mnot.net/blog/2015/02/18/http2>, un de l'auteur de curl <http://daniel.haxx.se/blog/2015/05/15/rfc-7540-is-http2/> et l'annonce de l'IETF <http://www.ietf.org/blog/2015/02/http2-approved/>.
— Sur le protocole : la bonne FAQ du groupe de travail <http://http2.github.io/faq/> et une description par l'auteur de curl <http://daniel.haxx.se/http2/>.
— Sur le gain en performances : une analyse sceptique <https://ma.ttias.be/sdpy-http2-actually-help/>.