RFC 9113 : HTTP/2

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/9113.html

Le protocole HTTP, à la base des échanges sur le Web, a plusieurs versions, 1, 2 et 3. Toutes ont en commun la même sémantique, décrite dans le RFC 9110 [1]. Mais l'encodage sur le câble est différent. HTTP/2 a un encodage binaire et un transport spécifique (binaire, multiplexé, avec possibilité de "*push*"). Fini de déboguer des serveurs HTTP avec telnet. En échange, cette version promet d'être plus rapide, notamment en diminuant la latence <https://www.bortzmeyer.org/latence.html> lors des échanges. Ce RFC remplace l'ancienne norme HTTP/2 (RFC 7540), mais le protocole ne change pas, il s'agit surtout d'une réécriture pour suivre le nouveau cadre de normalisation, où un RFC générique, le RFC 9110 spécifie la sémantique de HTTP et où il y a un RFC spécifique par version.

La section 1 de notre RFC résume les motivations derrère cette version 2, et notamment les limites de HTTP/1.1, normalisé dans le RFC 9112 :

— 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 9.3.2 du RFC 9112 permet d'envoyer la seconde requête tout de suite mais les réponses doivent être dans l'ordre des requêtes, donc pas moyen pour une requête rapide de doubler une lente.)
— Un encodage des en-têtes inefficace, trop bavard et trop redondant.

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

Au contraire, HTTP/2 n'utilise 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 est normalement plus rapide. Le RFC note toutefois que HTTP/2, qui dépend de TCP et n'utilise qu'une seule connexion TCP, ne résout pas le problème du "*head-of-line blocking*". (Pour cela, il faudrait HTTP/3, normalisé dans le RFC 9114.)

La section 2 de notre RFC résume l'essentiel de ce qu'il faut savoir sur HTTP/2. Il garde la sémantique générale de HTTP (donc, par exemple, un `GET` de `/cetteressourcenexistepas` fait un 404). La norme HTTP/2 ne normalise que le transport des messages, pas les messages ou leurs réponses (qui sont décrits par le RFC 9110). Notez que HTTP/2 s'appelait il y a très longtemps SPDY (initialement lancé par Google).

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 (ils avaient aussi un mécanisme de prioritisation entre eux, que notre RFC abandonne).

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 90 pages). D'abord, l'établissement de la connexion. HTTP/2 tourne au-dessus de TCP. Comment on fait pour savoir si le serveur accepte HTTP/2 ? C'est marqué dans l'URL ? Non, les URL sont les mêmes, avec les plans `http:` et `https:`. 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 ? Pas encore, bien que le futur type de données DNS `<https://www.afnic.fr/observatoire-ressources/papier-expert/de-nouveaux-types-dns-pour` `> HTTPS` permettra cela. Aujourd'hui, les méthodes pour savoir si le client doit tenter HTTP/2 sont :

— Une configuration explicite (comme l'option `--http2` de curl dans les exemples plus loin),
— L'en-tête `Alt-Svc:` du RFC 7838, qui nécessite de tenter en HTTP/1 d'abord,
— Si on fait du HTTPS et donc du TLS, on peut 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 dans le registre IANA `<https://www.iana.org/assignments/tls-extensiontype-values/tls-extensio` `xml#alpn-protocol-ids>`. Dans le RFC 7540, il y avait aussi un `h2c` dont l'usage est maintenant abandonné.)
— L'en-tête HTTP `Upgrade:` qui était dans la section 6.7 du RFC 7230 est désormais abandonné.

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 (il y a d'abord une préface, un nombre magique qui permet de s'assurer que tout le monde comprend bien HTTP/2, mais je n'en parlerai pas davantage). À quoi ressemblent ces trames (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). Les identificateurs ne sont jamais réutilisés (si on tombe à cours, la seule solution est de fermer la connexion TCP et d'en ouvrir une autre).

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.

HTTP/2 avait (RFC 7540, section 5.3) un mécanisme de priorité entre trames, qui permettait d'éviter, par exemple, que la récupération d'une grosse image ne ralentisse le chargement d'une feuille de style. Mais il était trop complexe, et a été peu mis en œuvre, la plupart des serveurs ignoraient les demandes de priorité des clients. Un nouveau mécanisme est décrit dans le RFC 9218.

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.xml#frame-type>. Les principaux types actuels sont :

— DATA (type 0), les trames les plus nombreuses, celles qui portent les données, comme les pages HTML (elles peuvent aussi contenir du remplissage, pour éviter qu'un observateur ne déduise de la taille des réponses la page qu'on regardait, cf. section 10.7),
— HEADERS (type 1), qui portent les en-têtes HTTP, dûment comprimés selon le RFC 7541,
— PRIORITY (type 2) indiquait la priorité que l'émetteur donne au ruisseau qui porte cette trame, mais ce type de trame n'est désormais plus utilisé,
— 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.xml#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.xml#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 est du HTTP. La sémantique est donc celle du RFC 9110. Il y a quelques différences comme le fait que certains en-têtes disparaissent, par exemple Connection: (section 8.2.2) qui n'est plus utile en HTTP/2 ou Upgrade: (section 8.6).

HTTP est 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 9110, section 10.1) 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 en binaire, et comprimés 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 en HTTP (section 8.3). Ces informations sont mises dans les `HEADERS` HTTP/2, précédés d'un deux-points. C'est le cas de la méthode (RFC 9110, section 9.3), 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`.

Le RFC met en garde les programmeur·ses : certains caractères peuvent être dangereux car profitant des faiblesses de certains analyseurs ou bien utilisant le fait que HTTP n'est pas toujours de bout en bout et qu'un message peut être traduit de HTTP/1 en HTTP/2 (ou réciproquement). Un deux-points dans le nom d'un champ, par exemple, pourrait produire un message dont l'interprétation ne serait pas celle attendue (ce qu'on nomme le "*request smuggling*").

Voici des exemples de requêtes HTTP (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 introduite par HTTP/2, la possibilité pour le serveur de pousser ("*push*", section 8.4 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 9110, section 9.3.6) 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, 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 !

Puisqu'on parle de sécurité, la section 10 traite un certain nombre de problèmes de sécurité de HTTP/2. Elle rappelle que TLS est fortement recommandé (mais il n'est devenu obligatoire qu'avec HTTP/3, HTTP/2 permettant toutefois une utilisation « opportuniste », cf. RFC 8164). Parmi les problèmes 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.

Comme on le voit, HTTP/2 est bien plus complexe que HTTP/1. On ne peut pas espérer programmer un client ou un serveur en quelques heures, comme on le fait avec HTTP/1. C'est en partie pour cela que personne ne prévoit un abandon de HTTP/1, qui continuera à coexister avec HTTP/2 (et HTTP/3 !) pendant très longtemps encore.

Question mises en œuvre, HTTP/2 est désormais présent dans la quasi-totalité des clients, serveurs et bibliothèques HTTP. Ici, avec curl, en forçant l'utilisation de HTTP/2 dès le début :

```
% curl -v --http2 https://www.bortzmeyer.org/7540.html
*   Trying 2001:4b98:dc0:41:216:3eff:fe27:3d3f:443...
* Connected to www.bortzmeyer.org (2001:4b98:dc0:41:216:3eff:fe27:3d3f) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
...
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=www.bortzmeyer.org
...
* Using HTTP2, server supports multiplexing
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* h2h3 [:method: GET]
* h2h3 [:path: /7540.html]
* h2h3 [:scheme: https]
* h2h3 [:authority: www.bortzmeyer.org]
* h2h3 [user-agent: curl/7.82.0]
* h2h3 [accept: */*]
```

```
* Using Stream ID: 1 (easy handle 0x5639a5d10cf0)
> GET /7540.html HTTP/2
> Host: www.bortzmeyer.org
> user-agent: curl/7.82.0
> accept: */*
...
< HTTP/2 200
...
< etag: "b4b0-5de09d5830d11"
< content-type: text/html; charset=UTF-8
< date: Thu, 05 May 2022 13:07:08 GMT
< server: Apache/2.4.53 (Debian)
<
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
<html xml:lang="fr" lang="fr" xmlns="http://www.w3.org/1999/xhtml">
<head>
...
```

Vous voulez voir un joli pcap de HTTP/2? La plupart des sites accessibles en HTTP/2 (et parfois des clients) imposent TLS. Il faut donc, si on veut voir « à l'intérieur » des paquets, utiliser la technique classique d'exportation de la clé :

```
% export SSLKEYLOGFILE=/tmp/http2.key
% curl --http2 https://www.bortzmeyer.org/7540.html
```

Puis vérifiez que dans les préférences de Wireshark, on a cette clé (par exemple dans `/.config/wireshark/pre` une ligne `tls.keylog_file: /tmp/http2.key`). On peut alors regarder le pcap en détail. Voici un tel pcap (en ligne sur `https://www.bortzmeyer.org/files/http2.pcap`) et la clé correspondante (en ligne sur `https://www.bortzmeyer.org/files/http2.key`). Cela permet de regarder le contenu des messages avec Wireshark :

Cela permet aussi, avec une commande comme `tshark -V -r http2.pcap > http2.txt`, de produire un joli fichier d'analyse (en ligne sur `https://www.bortzmeyer.org/files/http2.txt`). Notez les identificateurs de ruisseaux ("*Stream ID*") : il n'y en a que deux, 0 et 1, car on n'a chargé qu'une ressource (il y a un ruisseau dans chaque direction). Je vous laisse faire vous-même l'opération pour le cas de deux ressources, avec une commande comme `curl -v --http2 https://www.bortzmeyer.org/7540.` `https://www.bortzmeyer.org/9116.html`. Vous verrez alors le parallélisme de HTTP/2 et les multiples ruisseaux.

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 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 :

L'annexe B liste les principaux changements depuis le RFC 7540, notamment :

— L'abandon du mécanisme de priorité entre ruisseaux, trop complexe, remplacé par celui du RFC 9218.
— L'abandon de l'utilisation de `Upgrade:` pour passer de HTTP/1 en HTTP/2,
— Obligation plus strictes de valider la syntaxe des en-têtes,
— Meilleure description des en-têtes spécifiques à la gestion de la connexion, et qui ne doivent plus être utilisés,
— Et bien sûr un certain nombre de changements pour s'aligner avec le nouveau cadre générique du RFC 9110 et notamment sa terminologie.

Et, sinon, si vous voulez vous instruire sur HTTP/2 sans lire tout le RFC, il y a évidemment le livre de Daniel Stenberg <https://daniel.haxx.se/http2/>.