SGBLE

Zeste de savoir

Le problème de HTTP/2

25 mars 2021

Table des matières

1. Introduction : une commande au bistro

Vous entrez dans un bistro. Un déjeuner simple et rapide suffira, vous êtes pressé! Vous vous installez, lisez la carte et rapidement vous faites votre choix. Ça sera un sandwich, un verre de jus d'agrumes frais et un cookie fait maison pour le dessert.

Le serveur arrive, prend la commande pour le sandwich… et s'en va aussitôt le chercher! Pour passer le reste de votre commande, vous appelez donc un deuxième serveur. Vous demandez votre jus et encore une fois, le deuxième serveur s'en va aussitôt. Même chose pour le troisième serveur qui sera chargé de prendre en charge le cookie.

Vous en conviendrez, cet établissement a un curieux mode de fonctionnement. Pourtant, nous procédons de façon analogue quand nous naviguons sur le Web avec le protocole HTTP/1.1.

2. Le problème de HTTP/1

HTTP/1.1 fonctionne sur le principe une connexion TCP par requête/réponse. Quand on récupère une ressource, tout va bien: on ouvre une connexion TCP, on envoie une requête HTTP dedans (pour demander la ressource `/index.html` par exemple) et on obtient la réponse du serveur en retour sur cette même connexion TCP.

Mais une page d'un site Web moderne se compose rarement du seul document HTML. Il nous faut en général au moins les choses suivante pour une seule page:

— le document HTML;
— le fichier favicon;
— la (ou les) feuille(s) de style CSS;
— le (ou les) script(s) JavaScript;
— des images.

Dans ce cas, le navigateur va essayer d'optimiser son travail pour afficher la page le plus vite possible. Il peut pour cela ouvrir plusieurs connexions TCP en parallèle pour y faire les requêtes en même temps. Voici un exemple typique des requêtes qu'on pourrait faire en parallèle:

```
GET /index.html
GET /favicon.ico
GET /assets/css/framework.css
GET /assets/css/style.css
GET /assets/js/framework.js
GET /assets/js/script.js
GET /static/img/header.webp
GET /static/img/logo.svg
```

Évidemment, certaines ressources sont dépendantes des autres, un certain ordre doit donc être respecté pour afficher une page au plus vite. Les frameworks JS ou CSS sont un cas typique de ressources prioritaires car ils conditionnent l'affichage de la page. *A contrario*, les images peuvent être récupérées à la fin car une latence d'affichage est admissible.

Tout cela n'est pas très optimal, il faut ouvrir beaucoup de connexions en parallèle et y passer une requête dans chacune d'entre elles et récupérer les multiples réponses.

Dans la vie réelle, un seul serveur prend commande des trois articles et les ramène d'un coup sur un seul plateau. Nous appelons cela le multiplexage et c'est l'une des idées majeures introduite par HTTP/2.

3. Le multiplexage de HTTP/2

Imaginons le cas où nous tenterions de multiplexer des requêtes avec HTTP/1.1. Nous demanderions sur une même connexion TCP les ressources suivantes et nous attendrions les trois réponses à la fin:

```
GET /script.js
GET /bootstrap.css
GET /style.css
```

Les trois réponses obtenues:

```
HTTP/1.1 200 OK
Date: Wed, 16 Dec 2020 10:35:54 GMT
Content-Type: text/css; charset=utf-8
Content-Length: 420
[...]

HTTP/1.1 200 OK
Date: Wed, 16 Dec 2020 10:35:54 GMT
Content-Type: text/css; charset=utf-8
Content-Length: 420042
```

```
[...]

HTTP/1.1 200 OK
Date: Wed, 16 Dec 2020 10:35:54 GMT
Content-Type: application/javascript; charset=utf-8
Content-Length: 1337
[...]
```

Comment savoir quelle réponse correspond à quelle requête? Nous pouvons certes nous douter que la réponses contenant `Content-Type: application/javascript` correspond à notre requête pour `GET /script.js` car c'est la seule réponse JS. Mais comment pourrions-nous l'inférer pour les deux autres?

Il s'agit là de tout le problème de HTTP/1.1: il ne permet pas de multiplexer les transmissions entre le client et le serveur car il ne propose aucun moyen standardisé ou fiable de faire correspondre une réponse à une requête.

HTTP/2 introduit la notion de *stream* pour gérer l'entremêlement des réponses. À chaque requête nous faisons correspondre un *stream ID*. Les réponses du serveur n'arrivent pas nécessairement dans l'ordre des requêtes mais elles sont estampillées (*taguée*) du *stream ID* qui correspond à la requête à laquelle elle font écho. Nous pouvons donc associer correctement chaque réponse à sa requête, et ainsi éviter d'essayer d'afficher un script JavaScript comme si c'était une image.

Mais ce n'est pas tout. Les réponses peuvent être envoyées de façon non séquentielle, découpée. Chaque bout de réponse indique son *stream ID*, ce qui permet de la recomposer correctement à la fin (démultiplexage).

Pour comparer avec HTTP/1, le schéma suivant représente les trois requêtes parallèles.

HTTP/2 apporte d'autres améliorations substantielles par rapport à son prédécesseur, mais le multiplexage semble être la plus remarquable d'entre elles.

4. Les problèmes de HTTP/2

HTTP/2 ne vient pas sans son lot de difficultés. S'il est vrai que la technique vue permet d'économiser de la bande passante et des ressources réseau, il revient à l'applicatif (code client ou code serveur) d'en payer le prix en complexité d'implémentation et en ressource de calcul car les étapes de mux/demux ne sont pas gratuites.

La version HTTP/1.1 pourra donc être privilégiée dans certains cas sans un réel besoin des forces de HTTP/2. Nous penserons notamment aux interactions avec une API: il s'agit généralement de requêtes séquentielles et synchrones entre le programme et l'API, mettant ainsi en question la pertinence de HTTP/2.

Autre problème: dans un déploiement Web réel, il est commun de mettre en place un serveur frontal comme *reverse-proxy* devant un ou des serveurs applicatifs. Le but est généralement de terminer une connexion en TLS, de faire du *load-balancing* ou encore du *fail-over*. Ce rôle de *serveur frontal* (*frontend server*) peut être endossé par exemple par un serveur classique comme Nginx ou d'autres programmes plus spécialisés comme HAProxy.

Le souci étant que le serveur qui fait office de terminaison doit vraisemblablement par la suite communiquer en HTTP/1 avec un serveur applicatif comme Gunicorn, voire utiliser un protocole qui n'est pas HTTP du tout (FastCGI ou uwsgi par exemple). Cela met donc potentiellement en échec tous les efforts précédemment accomplis par l'usage de HTTP/2.

La grande question est donc la suivante: est-il si intéressant de déployer HTTP/2 entre le client et le serveur frontal comme cela?

Nous l'avons vu, cela implique donc une étape de mux/demux au niveau de l'échange entre le serveur frontal et le serveur applicatif derrière. Ces opérations ne sont pas gratuites en CPU et il vaudrait mieux qu'elle se rentabilisent en se traduisant par un gain pour le client: c'est le but de HTTP/2, après tout, n'est-ce pas?

Pour palier à cela, on peut donc privilégier la mise en place de bout en bout quand c'est possible. Mais ce n'est pas si simple: il faut que les protocoles s'y prêtent bien et que les briques soient assez intelligentes entre elles pour éviter le travail inutile.

Présentons sans plus tarder un tel exemple: il s'agit d'une configuration entre un HAProxy frontal et un Nginx derrière lui. Depuis sa version 2.0, HAProxy gère bien les *backend* HTTP/2 et est assez intelligent pour éviter les étape de mux/demux si son *frontend* est lui aussi utilisé en HTTP/2. (cf. https://www.haproxy.com/fr/blog/haproxy-2–0-and-beyond/#end-to-end-http-2)

Dans l'exemple qui suit, Nginx sert les requêtes en local en utilisant HTTP/2 en clair avec HAProxy. Ce qui entre dans HAProxy entre aussi dans Nginx *verbatim*, sans étape intermédiaire de mux/demux. Idem dans l'autre sens.

Contrairement à une croyance populaire, HTTP/2 n'a techniquement pas besoin de reposer sur une couche TLS. Ce sont les navigateurs qui imposent cela de fait. Ici, c'est HAProxy qui prend en charge TLS avec le client, mais la couche TLS n'a aucunement besoin d'être en place de bout en bout.

Côté HAProxy 2.0+:

```
HTTP/1.1 200 OK
Date: Wed, 16 Dec 2020 10:35:54 GMT
Content-Type: text/css; charset=utf-8
Content-Length: 420
[...]

HTTP/1.1 200 OK
Date: Wed, 16 Dec 2020 10:35:54 GMT
```

```
Content-Type: text/css; charset=utf-8
Content-Length: 420042
[...]

HTTP/1.1 200 OK
Date: Wed, 16 Dec 2020 10:35:54 GMT
Content-Type: application/javascript; charset=utf-8
Content-Length: 1337
[...]
```

Côté Nginx (1.18):

```
server {
  # Pas de gestion de TLS/ALPN ici

    listen 127.0.0.1:444 http2 default_server;

    root /var/www/html;

    location / {
        gzip_static on;
        try_files $uri $uri/ =404;
    }
```

Dans une configuration comme cela, nous avons donc ce genre de liaison en images:

Pensez donc bien à vérifier si vous avez la possibilité de mettre en place HTTP/2 de bout en bout! 🍊