Protocoles des Services Internet II

Juliusz Chroboczek

5 octobre 2021

Dans ce cours, nous voyons la structure des protocoles construits au-dessus de HTTP.

1 Le protocole HTTP

Le *Web* est un hypertexte distribué à grande échelle. Les deux éléments essentiels du *Web* sont le protocole de communication HTTP et le format de texte enrichi HTML.

1.1 HTTP/0.9

La première version de HTTP, développée en 1989, s'appelle aujourd'hui HTTP/0.9. C'était un protocole requête réponse très simple. Le client se connecte au serveur puis envoie une ligne de texte consistant de deux lexèmes :

- le mot `GET` ;
- le chemin du fichier demandé.

Le serveur envoie le contenu du fichier demandé, qui est forcément du HTML, puis ferme la connexion TCP.

1.2 HTTP/1.0

HTTP/0.9 est un protocole très limité. HTTP/1.0, documenté en 1996 (mais développé beaucoup plus tôt) ajoute plusieurs fonctionnalités importantes :

- la possibilité de négocier la version du protocole ;
- la possibilité de transmettre des formats de fichier autres que HTML (par exemple des images) ;
- la possibilité d'envoyer des données du client vers le serveur.

La première ligne d'une requête HTTP/1.0 consiste de trois lexèmes :

- la méthode, par exemple `GET`, `PUT` ou `POST` ;
- l'URL (qui n'est pas forcément juste un chemin, mais qui peut contenir des paramètres) ;
- la version du protocole `HTTP/1.0`.

La requête est ensuite suivie d'un nombre arbitraire d'*entêtes*, par exemple

```
Accept-Language: fr, en
```

Si un entête est inconnu, il est ignoré par le serveur, ce qui rend le protocole extensible. Les entêtes sont suivis d'une ligne blanche suivie, si la méthode n'est pas `GET` ou `HEAD`, par le contenu de la requête.

La réponse du serveur commence par une ligne indiquant le statut, par exemple

```
HTTP/1.0 200 OK
```

Elle est ensuite suivie d'un nombre arbitraire d'entêtes, dont le plus important est `Content-Type`, qui indique le type du fichier envoyé. Une ligne blanche sépare les entêtes du contenu du fichier, qui, comme en HTTP/0.9, est indiqué par la fin de la connexion TCP.

1.3 HTTP/1.1

HTTP/1.1, normalisé en 1997, résoud la plupart des problèmes de HTTP/1.0 tout en restant compatible avec ce dernier. Tout d'abord, HTTP/1.0 utilise une connexion par « entité » transférée, ce qui utilise TCP d'une façon inefficace ; HTTP/1.1 permet de réutiliser la même connexion pour plusieurs entités. Ensuite, HTTP/1.0 indique la fin du transfert par une fin de connexion, ce qui n'est pas fiable à la couche application ; HTTP/1.1 indique explicitement la fin du transfert. Enfin, HTTP/1.1 ajoute un protocole relativement riche d'invalidation et d'intorrogation des caches, ce dont nous parlerons au cours suivant.

1.4 HTTP/2

HTTP/2 conserve la sémantique de HTTP/1.1 (toute requête HTTP/2 peut être convertie en HTTP/1.1 sans perte d'informations), mais change la syntaxe : les entêtes sont envoyés en un format binaire et compressé, et plusieurs flots de données peuvent être multiplexés sur une seule connexion TCP. Comme TCP n'est pas adapté au multiplexage, HTTP/2 peut parfois être très inefficace.

Malgré ses problèmes, HTTP/2 est largement déployé aujourd'hui.

1.5 HTTP/3

HTTP/3 conserve encore la sémantique de HTTP/1.1, mais n'utilise plus TCP : il est basé sur le protocole de couche de transport QUIC, qui est lui-même basé sur UDP. À la différence de TCP, QUIC intègre le chiffrage au protocole de couche de transport lui-même, et implémente un multiplexage efficace.

Au moment où ce document est écrit, HTTP/3 est implémenté par 70% des navigateurs, mais n'est à ma connaissance déployé du côté serveur que par Google.

2 Applications *web*

HTTP a initialement été conçu pour le transfert de documents HTML. Cependant, il a rapidement été utilisé pour implémenter des applications dont l'interface utilisateur s'affiche dans un navigateur *web* : ce sont les *applications web*.

L'avantage principal des applications *web* est qu'elles ne demandent aucune installation de logiciel supplémentaire du côté du client, ce qui est pratique aussi bien pour l'utilisateur que pour le distributeur de l'application.

2.1 Génération côté serveur

Les premières applications *web* étaient implémentées entièrement du côté serveur : un programme (souvent implémenté en PHP) s'exécutait sur le serveur et produisait un document HTML ordinaire, qui était envoyé au navigateur qui le traitait comme une page *web*. Cette approche a permis de rapidement déployer des applications sans demander aucun changement des clients.

Le problème principal de ces applications de première génération était leur faible interactivité : chaque interaction demandait un aller-retour client-serveur et le chargement d'une nouvelle page, ce qui pouvait prendre plusieurs secondes. Malgré cela, certaines applications de première génération sont encore déployées aujourd'hui, notamment *Wordpress*.

2.2 Javascript

En 1995, Brendan Eich, alors chez Netscape, ajouta un interpréteur *Scheme* au navigateur *Netscape Navigator*, et exporta les structures de données internes du navigateur au langage de script : c'est l'ancêtre du DOM. Comme Java était alors une technologie à la mode, on lui demanda d'ajouter des accolades au langage, ce qui lui arait pris (d'après lui) moins de deux semaines : le résultat s'appelle *Javascript*.

Avec Javascript, une réponse à une requête HTTP peut télécharger un *script* Javascript qui s'exécute dans le navigateur. Initialement, Javascript servait principalement à faire des animations et à valider les entrées de l'utilisateur sans consulter le serveur, ce qui ne changeait pas fondamentalement le modèle du *web*.

2.3 AJAX

En Mars 1999, les développeurs d'Internet Explorer 5.0 ont ajouté à leur version de Javascript la fonction `XMLHttpRequest`, qui permet à un *script* Javascript de faire une requête HTTP au serveur. Cette addition aparemment innocente change complètement la structure du *web* : au lieu d'être un protocole requête-réponse tout bête, HTTP est maintenant un protocole où une réponse peut être à l'origine de nouvelles requêtes, qui elles-mêmes vont causer des réponses, *ad nauseam*.

`XMLHttpRequest` a permis un nouveau type d'applications *web*, les applications *AJAX*, où un patron statique est envoyé au client accompagnée d'un *script* Javascript qui va faire des requêtes pour mettre à jour la page, et gérer localement les interactions avec l'utilisateur. Les gens qui savent vendre les choses parlent de *Web 2.0*.

La fonction `XMLHttpRequest` est aujourd'hui obsolète : elle a été remplacée par la fonction `fetch`, plus puissante, plus élégante, et, surtout, ne souffrant pas de capitalisation incohérente.

3 Protocoles au-dessus de HTTP : « API »

Une application AJAX consiste d'un *script* Javascript qui fait des requêtes à un serveur HTTP. L'ensemble des requêtes que peut faire le *script* s'appelle une *API* (par analogie à une API en programmation orientée objet) ; il s'agit en fait d'un protocole requête-réponse basé sur HTTP.

Il est typiquement désirable qu'une application *web* soit aussi utilisable par des applications non-Javascript (par exemple des applications natives, pour machine de bureau ou pour téléphone mobile). Il est naturellement possible d'implémenter deux protocoles distincts dans le même serveur, une API *web* et un protocole plus classique : par exemple, de nombreux serveurs de courrier électronique implémentent le protocole classique IMAP et une API *web* utilisée par le *webmail*. Cependant, pour éviter la duplication de code, il est parfois désirable d'implémenter un seul protocole qui sert aussi bien à l'application *web* qu'aux clients natifs. Ce protocole est alors l'API *web*, qui doit être suffisamment bien structurée pour être utilisée par des applications natives, qui ne sont pas forcément aussi faciles à mettre à jour qu'une application *web*.

3.1 API ad hoc

Historiquement, la plupart des applications *web* utilisent une API ad hoc, dont la structure reflète la structure interne de l'application sans se préocupper d'être cohérente ou facile à faire évoluer. Le code client Javascript est mis à jour à chaque mise à jour de l'application, et tant pis pour les applications natives qui deviennent du coup obsolètes.

Or, il y a plusieurs façons naturelles de représenter une API donnée en HTML. Considérons par exemple une API qui donne accès à une base de données. Une telle API pourrait considérer HTTP comme un simple transport, et transférer une requête SQL dans le corps d'une requête POST :

```
POST /sql.php HTTP/1.1
Host: sql.example.com
Content-Type: application/sql
Content-Length: 18

DROP TABLE USERS;
```

Alternativement, l'API pourrait demander que la requête soit codée dans l'URL :

```
POST /sql.php?query=DROP%20TABLE%20USERS%3B HTTP/1.1
```

En transportant SQL dans les requêtes, ces API sont fortement liées à l'implémentation de la base de données, et même des détails de son dialecte de SQL. Pour éviter cela, on pourrait tenter de réfléter l'intention de la requête dans les paramètres de l'URL, et laisser le soin de traduire la requête dans le langage de l'implémentation de la base de données utilisée au serveur :

```
POST /db.php?op=DELETE&kind=TABLE&target=USERS HTTP/1.1
```

Allant encore plus loin, on peut essayer de traduire la sémantique de la requête dans les différents champts de HTTP :

```
DELETE /table/users HTTP/1.1
```

3.2 REST

Dans les années 2000, la mode était à l'encapsulation de protocoles toujours plus complexes dans les corps des requêtes `POST` (on peut citer notamment le protocole SOAP). L'approche REST a été développée par réaction à cette tendance.

L'approche REST consiste de plusieurs principes de conception, dont les plus importants sont selon moi :

- il n'y a pas d'état de session du côté serveur — l'état de session doit être maintenu côté client, ou alors stocké dans des structures de données indiquées explicitement par le client (voyez l'exemple de la partie d'échecs donné en cours) ;
- les objets auxquels sont appliquées les actions sont codés sous forme de chemins dans les URL ;
- les actions sont codées dans la méthode HTTP.

Le premier principe (pas d'état de session) a des conséquences importantes, il permet de facilement distribuer l'application sur plusieurs serveurs, de paralléliser les actions, ou encore de survivre au reboot d'un serveur ; c'est un vieux principe de réseau, utilisé notamment dans le protocole NFS (publié en 1984). Le second (codage des objets dans les chemins) est raisonnable, il rend le protocole plus facile à comprendre et plus facile à mettre en cache, mais, appliqué naïvement, cause un nombre important de requêtes, ce qui est inefficace, même avec les versions récentes de HTTP.

Quant au troisième (l'utilisation de méthodes « riches »), son utilité me semble plus marginale. S'il existe une différence importante entre `GET` (pas de corps de requête, utilisé pour les actions idempotentes), et `POST` (corps de requête, non-cachable par défaut), je ne vois pas de différence significative entre `POST` et par exemple `PATCH`.

REST-like Si les principes de REST sont raisonnables, il est parfois difficile de les appliquer trop strictement, notamment lorsqu'une action s'applique à plusieurs objets. La plupart des API décrites comme REST font des concessions à la simplicité ou à l'efficacité, et ne suivent pas strictement les principes REST. Les plus pédants parmi nous parlent parfois d'API *RESTful* ou *REST-like*.

4 Codage des données

Il est parfois possible de concevoir un protocole de telle façon que les données transmises dans le corps des requêtes et des réponses HTTP n'aient pas besoin d'être codées. Le codage est cependant nécessaire lorsque les requêtes ou les réponses transportent des collections de données. Si les structures sont plus complexes, il est généralement désirable d'employer un codage formalisé tel que XML ou JSON.

4.1 Codage ad hoc

Un simple codage ad hoc est souvent suffisant. Par exemple, une liste d' éléments peut souventêtre transmise en séparant les éléments par une virgule (CSV), par un caractère de fin de ligne (`\n`) ou par une ligne vide (`\n\n`).

4.2 XML

XML est un lointain descendant de SGML, un format généralisé de texte enrichi. À la différence de SGML, un document XML peut être analysé *(parsed)* sans en connaître le « schéma » : l'analyse est séparée de la sémantique.

Du fait de son héritage des formats de texte enrichi, XML contient beaucoup de complexité qui n'est pas utile lorsqu'il est utilisé pour transporter des structures de données. S'il est encore utilisé dans beaucoup de protocoles *web*, les protocoles récents ont tendance à préférer JSON.

4.3 JSON

JSON est un sous-ensemble de la syntaxe de Javascript. Il permet de coder les nombres, les chaînes, les tableaux et les dictionnaires dont les clés sont des chaînes (les « objets »). La syntaxe est suffisamment simple pour être codée à la main avec un peu d'habitude.

S'il n'est pas aussi prolixe que XML, JSON n'est pas particulièrement compact. Il est aussi assez fortement lié à Javascript : par exemple, les nombres de JSON sont ceux de Javascript, des flottants capables de coder précisément les entiers de 53 bits ou moins.

4.4 Formats binaires

Il existe de nombreaux formats binaires génériques, plus compacts que JSON, et dont certains visent à remplacer ce dernier. On peut citer CBOR, BSON et MessagePack. Aucun ne me semble beaucoup utilisé en pratique.

Une approche différente est celle de *Protocol Buffers (Protobuf)*. Au lieu d'être un format générique, *Protobuf* définit un processus déterministe de génération d'un codage à partir d'une structure de données. Le compilateur *Protobuf* prend en entrée la définition formelle d'une structure de données, et produit des formateurs et des analyseurs pour un codage spécifique, adapté à cette structure de données. Le résultat est extrêmement compact, mais impossible à décoder sans connaître tous les détails de la structure de données initiale.