Protocoles des Services Internet II

Juliusz Chroboczek

16 octobre 2023

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

1 Le protocole HTTP

Le *Web* est une des applications tournant sur l'Internet. C'est un hypertexte distribué à grande échelle :

- c'est un *hypertexte*, il consiste de documents liés par des *hyperliens* (ou simplement *liens*) ;
- il est distribué, le *web* s'exécute sur plusieurs hôtes différents ;
- il est à grande échelle.

Le *web* ne garantit pas sa cohérence : lorsqu'une entité est supprimée ou déplacée, les liens qui la ciblent ne sont pas automatiquement mis à jour. Les erreurs (*404 not found*, ou simplement destination incorrecte) sont donc à prévoir.

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 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`, `POST` ou `PATCH`;
– 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 document envoyé.

L'entête `Content-Length` indique la longueur en octets du corps de la requête ou de la réponse. En présence de cet entête, la fin du corps n'est plus indiquée par la fin de la connexion, ce qui permet de distinguer une transaction complète d'une rupture de connexion, et rend donc le protocole fiable [1].

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/1.1 indique explicitement la fin du transfert, ce qui rend le protocole fiable à la couche application. 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/1.1 est beaucoup plus efficace que HTTP/1.0. Cependant, les pages pleines d'images et les applications de type REST (voir ci-dessos) causent de très nombreux transferts, ce qui force le client à ouvrir plusieurs connexions (ou alors à subir le coût d'un RTT par transfert).

Afin de diminuer la charge du serveur, les ingénieurs de Google ont développé HTTP/2. HTTP/2 conserve en grande la sémantique de HTTP/1.1 (presque toute transaction 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 (communication ordonnée menant au bloquage de tête, contrôle de congestion par connexion), HTTP/2 peut parfois être inefficace. Malgré ses problèmes, HTTP/2 est largement déployé aujourd'hui.

1. Si la taille de la réponse ne peut pas être déterminée à l'avance, la fin de la transaction est encore signalisée par la fin de la connexion. Ce problème est résolu en HTTP/1.1 (transferts « *chunked* »).

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, ce qui diminue le temps nécessaire à l'établissement d'une connexion, et implémente le multiplexage de manière efficace (au niveau de la couche transport).

En octobre 2023, HTTP/3 était activé par 75% des navigateurs (tous les navigateurs importants sauf ceux d'Apple), mais n'était déployé que sur 25% des serveurs.

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. Leur défaut, c'est qu'elles sont limitées aux fonctionnalités disponibles dans les navigateurs.

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 page contenait normalement des formulaires (élément `form` en HTML), qui pouvaient eux-même provoquer le chargement d'une autre page *web*.

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, par exemple *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 qu'il soutient lui avoir pris 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* ou *Web 2.0*, où un patron statique est envoyé au client accompagnée d'un *script* Javascript. Le *script* fait des requêtes au serveur, qui retourne des structures de données (et pas des documents) qui vont servir à peupler l'interface utilisateur. Le *script* génère donc les données affichées à l'utilisateur, et gère localement l'interaction, ce qui permet de faire des applications interactives.

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 généralement désiré 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*.

Pour cela, il faut s'intéresser à deux problèmes : comment coder les données, et comment structurer l'API.

3.1 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.

3.1.1 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`).

3.1.2 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 est très complexe, et cette complexité n'est pas utile lorsqu'il est utilisé pour transporter des structures de données. Les protocoles récents ont tendance à lui préférer JSON.

3.1.3 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.

3.1.4 Formats binaires

Il existe de nombreaux formats binaires génériques, plus compacts que JSON. Ils ne sont pas beaucoup utilisés, sauf dans le domaine des systèmes embarqués (« IoT »). CBOR me semble très bien fait, je n'ai pas regardé en détail BSON et *MessagePack*.

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. Les protocoles générés automatiquement ne préservent généralement pas la compatibilité entre versions : une modification même mineure des structures de données cause un changement complet du protocole.

3.2 Structure des API *web*

Historiquement, la plupart des applications *web* utilisaient 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.

Il y a eu plusieurs efforts pour standardiser la structure des applications *web* dans le but de les rendre plus compréhensibles et de permettre de les faire évoluer sans casser la compatibilité avec les applications natives.

3.2.1 SOAP

À la toute fin des années 1998, un consortium industriel mené par Microsoft a défini une manière de structurer les API *web* en des ensembles de requêtes complexes codées en XML. Initialement appelé XML-RPC, cette technologie a été renommée en SOAP *(Simple Object Access Protocol)*.

SOAP était très complexe, et la plupart des développeurs *web* ont migré vers des mécanismes plus simples. À ma connaissance, SOAP n'est plus utilisé que par les logiciels de *Microsoft* (par exemple *Outlook*).

3.2.2 REST

L'approche REST vise à simplifier les protocoles *web*, et est probablement issue de l'opposition à la complexité toujours croissante des techniques de type SOAP. L'approche REST consiste de plusieurs principes de conception, dont les plus importants sont selon moi :

- toutes les requêtes manipulent des objets côté serveur ; par exemple, dans un serveur d'impression, au lieu d'implémenter une requête *imprimer*, il y aura une requête *créer une tâche d'impression* qui retourne une référence à la tâche ;
- 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 côté serveur indiquées explicitement par la requête ;
- les objets auxquels sont appliquées les actions sont indiqués dans l'URL (et pas dans le corps de la requête) ;
- les actions sont codées dans la méthode HTTP.

Le second 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 déjà dans le protocole NFS (publié en 1984).

Le troisième principe (objets indiqués par l'URL) a une conséquence importante : chaque requête REST ne peut agir que sur un seul objet. Du coup, une transaction qui agit sur plusieurs objets doit faire intervenir plusieurs requêtes, ce qui peut facilement mener à des protocoles très inefficaces [2].

REST-like En pratique, les API décrites en REST n'obéissent pas strictement aux principes de REST. Les API utilisées en pratique sont parfois décrites comme étant *RESTful* ou *REST-like*.

2. Un ami me fait savoir que son téléphone mobile génère 4.5 Mo de traffic par heure lorsqu'il est en veille.