Proxys et caches

Juliusz Chroboczek

14 octobre 2021

1 Proxies

Lors du cours de réseau, nous avons vu plusieurs exemples d'interconnexion de réseaux aux couches basses :

- à la couche 1, les *hubs* permettent d'interconnecter des câbles Ethernet ;
- à la couche 2, les *switches* recopient les trames d'un segment à un autre ;
- à la couche 3, les *routeurs* recopient les paquets d'un lien à un autre.

Un *proxy* de couche transport est un processus qui recopie les données d'une connexion à une autre ; par exemple, un serveurs SOCKS ou un tunnel ssh sont des *proxies* de couche transport. *Tor* est un réseau de *proxies* de couche transport.

Un *proxy* de couche application est un processus qui recopie les données d'une session de couche application à une autre. Par exemple, un serveur de courrier accepte les courriers et les fait suivre à un serveur distant : il agit simultanément comme serveur et client. Un *proxy HTTP* est un processus qui agit comme serveur HTTP et fait suivre chaque requête qu'il reçoit à un serveur distant.

1.1 Proxies traditionnels

Un *proxy* HTTP traditionnel est explicitement configuré dans le navigateur. Le navigateur fait une requête (avec un format légèrement différent) au *proxy*, qui fait suivre la requête au serveur, puis fait suivre la réponse de celui-ci.

Un *proxy* traditionnel permet de traverser les pare-feu restrictifs, de mettre en cache les données partagées par plusieurs utilisateurs, de filtrer le trafic en interdisant l'accès à certains sites, ou encore d'espionner les utilisateurs (pour leur bien, naturellement).

1.2 Proxies transparents

Un *proxy* transparent fonctionne comme un *proxy* traditionnel, mais le trafic des ports 80 et 443 est redirigé vers le *proxy* par un NAT : le client fait une requête au serveur de destination, mais le routeur redirige le trafic vers le *proxy*.

Les *proxies* transparents ont les mêmes applications que les *proxies* traditionnels, mais ne demandent pas de configuration particulière de la part du client. Ils sont généralement déployés à

l'insu des utilisateurs, par des administrateurs qui seront les premiers contre le mur lorsque la révolution viendra.

1.3 Proxies inverses

À la différence des *proxies* traditionnels, qui se trouvent généralement dans le réseau du client, les *proxies* inverses sont devant le serveur : le client se connecte au *proxy* en faisant une requête HTTP ordinaire, que le *proxy* fait ensuite suivre au serveur d'application.

Les *proxies* inverses ont souvent des fonctionnalités supplémentaires. Ils servent les pages statiques sans intervention du *backend server*, ils combinent plusieurs serveurs en un seul site, et ils mettent en cache les réponses aux requêtes qui ne changent pas souvent.

2 Caches

La copie originale des données se trouve souvent loin du consommateur, ce qui peut causer des latences, incompressibles du fait de la vitesse de la lumière [1]. Pour diminuer cette latence, il est désirable de maintenir une copie des données située plus près du consommateur. La structure de données qui contient ces copies des données s'appelle un *cache*.

L'utilisation des caches est pervasive en informatique, et pas seulement en réseau. En système, vous avez vu les ncaches du processeur, placés entre le processeur et la mémoire vive, et les caches du système de fichiers, placés entre le processeur et le dique. En protocoles réseau, vous avez vu le cache ARP, qui contenait une copie locale de l'association entre adresses IP et adresses de couche lien (adresses MAC).

2.1 Types de caches HTTP

Un *cache HTTP* est un cache qui est situé entre un client et un serveur HTTP. Il existe plusieurs endroits où l'on peut placer un cache HTTP, et plusieurs caches peuvent être utilisés simultanément.

Cache du navigateur Chaque navigateur *web* contient un cache local, partiellement stocké en mémoire vive et partiellement stocké sur disque. Un accès à une donnée déjà dans le cache du navigateur ne recquiert aucun transfert à travers le réseau.

Le cache du navigateur est particulièrement utile lorsque la même donnée est référencée par plusieurs pages *web* auxquelles accède le même utilisateur. Par exemple, plusieurs pages d'un site peuvent contenir le même logo, et le cache du navigateur évite de le recharger sur chaque page.

Cache dans un *proxy* côté client Un cache peut aussi être présent dans un *proxy* côté client, classique ou transparent. Si le proxy dessert une population de clients uniforme, il permet de n'envoyer qu'une requête au serveur d'origine pour servir une donnée à toute la population. Par exemple, si tous les utilisateurs d'un cache accèdent à la même page approximativement au même

1. Glacialement lente.

moment (pensez à une salle de TP où tous les étudiants accèdent à la documentation de la même classe *Java*), alors le cache peut servir localement tous les utilisateurs.

Les *caching proxies* côté client étaient très populaires dans les années 2000. Ils sont rarement utilisés de nos jours, sauf dans certaines entreprises ou dans les salles de TP de certaines universités.

Cache dans un proxy inverse Un *proxy* est placé entre l'Internet et une application *web*. Comme l'application est souvent écrite dans un langage lent, tel que PHP ou Python, il est utile de cacher sa sortie dans le *proxy* inverse, qui peut alors servir des requêtes dupliquées localement, sans faire intervenir l'application.

CDN Un *CDN (Content Delivery Network)* est un vaste ensemble de *caching proxies* déployés à l'échelle globale. Par exemple, *Netflix* maintient des caches chez la plupart des fournisseurs de services, qui contiennent des copies locales des films les plus populaires.

Il existe aussi des fournisseurs de CDN, qui fournissent des réseaux de caches à des tierces parties. Par exemple, les serveurs qui distribuent les mises à jours des systèmes d'exploitation subissent un trafic très irrégulier, très élevé juste après la mise à disposition d'une nouvelle version, et assez faible autrement. Ils sont généralement implémentés par des fournisseurs de CDN qui desservent plusieurs distributeurs, ce qui permet de mieux distribuer la charge.

2.2 Cohérence des caches

Dès qu'on a plusieurs copies des données, il faut s'assurer que les différentes copies sont identiques ou tout au moins semblables : c'est le problème de la *cohérence* des caches. HTTP inclut des mécanismes qui permettent au serveur à l'origine d'une donnée d'indiquer aux caches au bout de combien de temps une donnée ne peut plus être servie sans consulter le serveur : c'est l'*invalidation*. HTTP inclut aussi des mécanismes qui permettent à un cache de rafraîchir efficacement une donnée qui a été invalidée : c'est la *revalidation*.

2.2.1 Validateurs HTTP

Un *validateur* est une étiquette qui est attachée à une donnée et qui permet de vérifier efficacement si deux copies de la donnée sont identiques. HTTP/1.0 utilisait la date de dernière modification comme validateur, ce qui n'est pas toujours fiable. HTTP/1.1 utilise des validateurs opaques, les *entity tags (ETag)*.

Date de dernière modification Une réponse HTTP peut contenir la date de dernière modification de la ressource dans l'entête `Last-Modified`. Les caches HTTP/1.0 utilisaient cette date comme validateur.

La date de dernière modification n'est pas un validateur fiable. Tout d'abord, une réponse peut dépendre d'entêtes de la requête (par exemple l'entête `Accept-Language`), qui font que deux données ayant la même date de dernière modification peuvent être indépendantes ; HTTP/1.1 ajoute l'entête `Varies`, qui permet de déclarer les entêtes dont dépend la réponse. Ensuite, les

dates HTTP ont une granularité d'une seconde, et la date de dernière modification ne permet donc pas de détecter qu'une ressource a changé deux fois durant la même seconde.

ETag Dans HTTP/1.1, une donnée peut être étiquetée par le serveur avec un *entity tag*, un validateur qui est transmis dans l'entête `ETag`. Le *ETag* est opaque pour le client, qui n'en connaît pas la structure : les seules opérations autorisées sur les *ETags* sont la comparaison. Comment le serveur génère le *ETag* est une affaire privée, la seule contrainte est que deux données distinctes aient des *ETags* distincts.

Par exemple, le serveur peut utiliser un *hash* cryptographique de la donnée entière, ce qui est inefficace mais fiable et facile à implémenter. Dans les cas où la donnée ne dépend pas de la requête, le serveur peut simplement utiliser la date de dernière modification, qui peut alors être envoyée avec une granularité arbitraire. Selon l'application, le serveur peut aussi utiliser un *hash* combinant la date de dernière modification avec les entêtes de la requête.

2.2.2 Entêtes de contrôle de cache

Par défaut, les réponses à `GET` sont cachables pendant un temps déterminé heuristiquement par le cache, tandis que les réponses aux autres requêtes ne sont pas cachables. Le serveur peut inclure des *entêtes de contrôle de caches* qui indiquent explicitement au cache comment se comporter.

HTTP/1.0 incluait un seul entête de contrôle de cache, l'entête `Expires`, qui indiquait la date à laquelle la donnée allait devenir invalide. HTTP/1.1 a remplacé cela par l'entête `Cache-Control`, qui utilise des temps relatifs (et évite donc les problèmes liés aux horloges non-synchronisées) et permet de définir les comportements plus précisément :

- `Cache-Control: no-cache` indique au cache que la donnée peut varier à tout moment, et qu'il faut donc la revalider systématiquement ;
- `Cache-Control: no-store` indique qu'il ne faut jamais stocker la donnée dans le cache (`no-store` implique `no-cache`) ;
- `Cache-Control: max-age=3600` indique que la donnée peut être mise en cache pendant au plus 3600 secondes depuis le moment où la requête a été envoyée au serveur.

2.2.3 Entêtes de revalidation

Après qu'une donnée a été invalidée (de façon heuristique ou en obéissant à un entête de contrôle de cache), elle peut encore être présente dans le cache, mais ne doit pas être utilisée sans consulter le serveur. C'est la *revalidation* de la donnée.

Si le cache utilise `Last-Modified` comme validateur, il envoie au serveur une requête équippée de l'entête `If-Modified-Since` contenant la date de dernière modification de la donnée en cache. Si la donnée a été modifiée depuis, le serveur répond avec la nouvelle version ; dans le cas contraire, il répond avec le code 206 *(« Not Modified »)*, et un corps vide.

Si le cache uilise un *ETag*, il utilise l'entête `If-None-Match` avec le *ETag* de la donnée en cache. Si la donnée sur le serveur a le même *ETag*, le serveur répond avec un code 206 et un corps vide.

3 Notifications asynchrones

Les approches REST et REST-like, qui transmettent les données de l'application dans des messages HTTP, permettent à l'application de profiter des avantages du *web* : disponibilité pervasive, distribution de la charge à travers plusieur serveurs, utilisation des CDN. Cependant, elles imposent des coûts importants :

- *overhead* important, car chaque message HTTP contient quelques centaines d'octets d'entêtes ;
- obligation de coder l'état de l'application par une structure de données explicite, qu'il faut transmettre avec chaque requête ;
- impossibilité de faire des notifications asynchrones, une réponse HTTP ne peut être générée qu'en réponse à une requête du client.

L'*overhead* est gênant, mais pas prohibitif en pratique, et a été sensiblement diminué par HTTP/2 et /3, qui compressent les entêtes. L'obligation de coder l'état est une bonne idée, et il existe des techniques qui permettent de l'éviter (par exemple en utilisant un *token* de session, stocké par exemple dans un *cookie*). Par contre, l'impossibilité de faire des notifications asynchrones est parfois prohibitive.

Considérons par exemple un serveur de jeu d'échecs. Pour éviter la triche, il est nécessaire de conserver l'état de la partie sur le serveur. Un protocole REST pourrait être constitué de quatre requêtes :

- une requête `POST` qui crée une nouvelle partie et retourne l'URL décrivant cette dernière ;
- une requête `POST` qui joue un coup ;
- une requête `GET` qui retourne l'état de la partie (par exemple après que le client a été relancé) ;
- une requête `DELETE` qui détruit la partie.

Mais comment notifier le client que l'opposant a joué un coup et qu'il faut mettre à jour l'affichage ? Une solution REST-ful pourrait consister à faire des `GET` périodiques (par exemple équipés d'un entête `If-None-Match` codant l'état précédent). Mais à quelle fréquence faut-il faire ces requêtes ? Si la fréquence est trop basse, l'application sera peu responsive, si elle est trop haute, le traffic généré sera trop important.

Pour cette application, comme pour beaucoup d'autres, on a besoin de *notifications asynchrones* — de messages envoyés par le serveur à un moment arbitraire.

3.1 *Slow polling*

Le *slow polling* (ou *Comet*) est une technique permettant de simuler les notifications asynchrones sans sortir du cadre de HTTP. Le client envoie une requête demandant l'état de la ressource obervée ; tant que rien ne change, le serveur n'envoie pas de réponse et laisse la réponse en suspens. Lorsque la ressource change, le serveur envoie la réponse à la requête suspendue, le client met sa vue à jour, et envoie immédiatement une nouvelle requête au serveur.

Pour éviter que les *proxies* interrompent une réponse, le serveur envoie une réponse indiquant que rien n'a changé au bout d'un certain temps (typiquement 30 s), et le client envoie alors immédiatement une nouvelle requête.

Essentiellement, le *slow polling* préserve la sémantique requête-réponse de HTTP, mais découple la requête de la réponse en permettant un temps arbitraire entre les deux. C'est une solution élégante au problème des notifications asynchrones, qui permet de préserver en partie les capacités de passage à l'échelle de HTTP, mais qui hérite de son d'*overhead*.

3.2 *WebSockets*

Les *WebSockets* sont une approche diamétralement opposée à l'approche REST : ce sont essentiellement des connexions TCP brutes qui permettent au client et au serveur de communiquer de manière traditionnelle, sans aucune structure de type requête-réponse. Je les ai souvent vues critiquées dans la communeauté REST, mais du fait de leur relative efficacité, elles sont très souvent utilisées pour les applications interactives.

Le protocole *WebSocket* est construit au-dessus de TCP, mais ajoute deux améliorations à ce dernier :

- un entête HTTP au début de la connexion, ce qui permet d'identifier la destionation d'une connexion par une URL, ce qui s'intègre beaucoup mieux aux applications *web* qu'un numéro de port ;
- une structure de type *TLV* normalisée, qui structure le traffic comme une suite de messages de taille arbitraire, ce qui est beaucoup plus utile que le flot d'octets sans structure transmis par TCP.

WebSocket n'impose aucune structure aux messages transmis dans les TLV. Il est souvent utilisé avec des messages codés en JSON.

Comme *WebSocket* est un protocole connecté, il n'a aucun des avantages de HTTP : les messages *WebSocket* ne sont pas cachables, et une connection *WebSocket* ne peut pas facilement migrer d'un serveur surchargé à un autre, moins chargé. Par contre, *WebSocket* permet de facilement transporter des protocoles bidirectionnels arbitraires, et fait cela avec un *overhead* minimal (2 octets par message dans le meilleur cas).