Proxys et caches

Juliusz Chroboczek

16 octobre 2022

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, un serveur TURN, un tunnel ssh sont des exemples de *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.

Une application d'un *proxy* traditionnel est de traverser les pare-feu restrictifs. Dans les années 1990, il était habituel que le réseau client ne soit pas interconnecté à l'Internet (à la couche 3), ou alors uniquement à travers un pare-feu très restrictif. L'accès au *web* était fourni par un *proxy* qui avait le droit d'accéder aussi bien au réseau interne qu'à l'Internet Global, et pouvait donc faire suivre les requêtes des clients et les réponses des serveurs. Cette façon de faire est fort heureusement obsolète, et il est normal aujourd'hui de connecter les clients à l'Internet.

Les *proxies* habituels ont d'autres applications, par exemple mettre les données partagées en cache (voir paragraphe 2 ci-dessous) ou garder un *log* de tous les accès externes faits par les clients (par exemple dans les banques).

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* à la couche 3 (par un routeur) : 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 explicite de la part du client, ce qui rend les problèmes difficiles à débogguer. 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 à proximité 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.

Outre le fait de mettre les données en cache (paragraphe 2), les proxies inverses permettent de faire de la distribution de charge : un seul *proxy* inverse peut être mis devant tout un troupeau de serveurs, et le *proxy* décide vers quel serveur envoyer chaque requête (par exemple envoyer chaque URL à un serveur spécifique, ou, si les serveurs sont tous équivalents, vers le serveur le moins chargé).

2 Caches

La copie originale des données se trouve souvent loin du client, 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 client. 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.

1. Glacialement lente.

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 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 plus rarement utilisés de nos jours.

Cache dans un proxy inverse Un *proxy* inverse 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.

Par ailleurs, beaucoup de sites desservent des données identiques. Par exemple, beaucoup de pages *web* incluent des polices décoratives, qui sont exactement les mêmes pour plusieurs sites. De même, les sites utilisent souvent des bibliothèques standard de code *Javascript*. Pour éviter de télécharger ces données plusieurs fois et de manière inefficace, il est utile de les mettre dans un CDN.

L'utilisation pervasive des CDN soulève un problème de vie privée : un fournisseur de CDN a des informations détaillées sur le comportement d'un vaste nombre de clients, et cela à l'insu de ces derniers. Par exemple, si votre site de *memes* favori utilise le CDN de *Cloudflare*, alors *Cloudflare* a des informations précises sur votre goût en images drôles, et cela sans que vous en ayez été informé.

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 Validateur

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 différentes. 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.

Le type principal d'*ETag* est le *ETag fort*. Un *ETag* fort a la syntaxe d'une chaîne entre guillemets doubles. Un serveur qui génère un *ETag* fort garantit que si la donnée change, alors le *ETag* changera. En d'autres termes, si pour la même URL le serveur retourne le même *ETag*, alors les données sont identiques.

Comment générer un *ETag* est une décision privée du serveur. 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 (et pas d'une seconde, comme avec l'entête `Last-Modified`). Si le résultat dépend des entêtes de la requête, le serveur peut utiliser une valeur (par exemple un *hash*) qui combine la date de dernière modification avec les valeurs des entêtes.

2.3 Entêtes de contrôle de cache

Par défaut, les réponses à `GET` sont cachables pendant un temps déterminé heuristiquement par le cache : le cache a le droit de ne pas recontacter le serveur pour retourner une réponse à une requête si la valeur stockée dans le cache est suffisamment récente. Lorsque le cache contacte le serveur pour confirmer la fraîcheur d'une donnée, on dit que le cache *revalide* la donnée.

Le client et le serveur peuvent inclure des *entêtes de contrôle de caches* qui indiquent explicitement aux caches quand une révalidation est nécessaire.

Contrôle de cache HTTP/1.0 HTTP/1.0 incluait l'entête de réponse `Expires`, qui indiquait une date après laquelle le cache devrait forcément revalider la donnée avant de retourner une réponse au client. En particulier, une valeur dans le passé permettait de forcer une revalidation systématique. Le principal problème de `Expires` est qu'il contient une date absolue, ce qui cause des problèmes si les horloges du serveur et du cache sont désynchronisées.

Par ailleurs, HTTP/1.0 incluait l'entête de requête `Pragma`, dont l'utilité n'a jamais été bien définie. La seule valeur de cet entête qui a été implémentée est la valeur `no-cache`, qui demandait aux caches de ne pas retourner une valeur antérieure, mais de revalider la donnée.

Contrôle de cache moderne HTTP/1.1 a rendu les entêtes `Expires` et `Pragma` obsolètes en introduisant un seul entête `Cache-Control`. À la différence de `Expires`, cet entête contient des temps relatifs ; à la différence de `Pragma`, sa syntaxe et le comportement qu'il indique sont normalisés et bien définis.

Dans une reqête, l'entête `Cache-Control` peut entre autres avoir la valeur suivante :

- `Cache-Control: no-cache` demande au cache de ne pas répondre par une donnée ancienne, mais de contacter le serveur pour une revalidation ; par exemple, un navigateur envoie cet entête lorsque l'utilisateur clique sur *Shift-Reload*.

Dans une réponse, `Cache-Control` peut entre autres avoir l'une des valeurs suivantes :

- `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.

Il est généralement utile d'envoyer une réponse dynamique avec une directive `no-cache`, et une réponse statique avec une directive `max-age` bien choisie, afin d'éviter les problèmes difficiles à débogguer dûs à l'expiration heuristique. La directive `no-store` n'est utile que lorsque les données sont de nature confidentielle.

2.4 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.