Notifications asynchrones

Juliusz Chroboczek

27 octobre 2023

1 Requêtes et réponses

Commes nous l'avons vu, HTTP est strictement un protocole *requête-réponse* : chaque interaction est initiée par une requête du client, à laquelle le serveur répond par exactement une réponse. Cette discipline empêche certains modes d'interaction.

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 ;
- 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-like 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 élevée, le traffic généré sera trop important.

Il y a d'autres situations semblables. Une application de courrier électronique un un *chat*, par exemple, a besoin de mettre à jour l'affichage lorsqu'un nouveau message est arrivé. Pour ce genre d'applications, on aurait besoin de *notifications asynchrones* — de messages envoyés par le serveur à un moment arbitraire, sans avoir été sollicitées par le client.

2 *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*.

Server-Sent Events Comme le *slow polling* est difficile à implémenter correctement, une forme standard de *slow polling* a été définie en 2004 sous le nom de *Server-Sent Events* (SSE). SSE définit un format standard pour les réponses au *slow polling*, et permet d'envoyer plusieurs réponses à une seule requête. SSE est implémenté dans tous les navigateurs modernes. Je pense cependant que cette technologie a été rendue obsolète par le déploiement du protocole *WebSocket*.

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

- le client se connecte à une URL plutôt qu'une adresse de *socket* (adresse IP et numéro de port), ce qui s'intègre beaucoup mieux aux applications *web* ;
- *WebSocket* implémente une sémantique par flots de messages de taille arbitraire, ce qui est beaucoup plus utile que les flots d'octets sans structure implémentés par TCP.

WebSocket n'impose aucune structure aux messages transmis dans les TLV. Généralement, les messages sont codés en JSON, mais rien n'empêche par exemple de coder les messages en XML ou même d'envoyer des données binaires brutes (par exemple des images au format PNG).

Comme *WebSocket* est un protocole connecté avec peu de structure, 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).

4 Autres technologies

***Data channels* SCTP** SCTP est un protocole de couche transport originellement conçu pour les procoles de contrôle téléphonique (établissement de connexion, routage et facturation dans le réseau téléphonique commuté). Une version légèrement simplifiée de SCTP a été intégrée aux navigateurs sous le nom de *data channels*. Les *data channels* héritent donc de la plupart des fonctionnalités de SCTP :

- flots multiples sur une seule connection (ce qui évite le blocage de tête) ;
- sémantique par flots de messages ;
- sémantique fiable ou non-fiable, au choix de l'application.

Les *data channels* sont conçus pour permettre la communication pair-à-pair, et l'API est donc assez complexe ; elle demande notamment l'utilisation d'un canal de signalisation pour négocier

les paramètres de la connexion. Les *data channels* sont donc plus compliqués à utiliser que les technologies qui ne supportent que la communication client-serveur.

WebTransport Le protocole *WebTransport* est promu par *Google* comme le successeur à *WebSocket*. *WebTransport* implémente (proprement) les flots multiples sur une seule connexion, et inclut un *hack* spécifique pour envoyer des datagrammes non-fiables. Par contre, la sémantique des flots fiables est une sémantique par flots d'octets, comme au temps de grand'père.

WebTransport dépend de HTTP/3, et n'est donc pour le moment pas largement déployé en dehors des applications *Google*.

***Server push* propriétaire** En plus des protocoles ouverts décrits ci-dessus, il existe des protocoles propriétaires promus par les producteurs d'OS pour téléphones mobiles. Le plus notable est *FireBase server push*, dont *Google* fait une promotion éhontée en incluant un support spécifique dans l'OS mobile *Android* : une application *Android* qui utilise *FireBase* est exemptée des restrictions auxquelles sont soumises les autres applications *Android* [1].

Apple fournit quelque chose de semblable pour iOS (« *Apple Push Notification Service* »), mais je ne connais pas les détails.

Ces solutions sont propriétaires, et dépendent de la bonne volonté d'un seul fournisseur, il me semble donc bon de les éviter.

1. Je ne comprends pas comment cela peut être légal.