RFC 6455 : The WebSocket protocol

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 12 décembre 2011

Date de publication du RFC : Décembre 2011

https://www.bortzmeyer.org/6455.html

Ce nouveau protocole, WebSocket, vise à résoudre un problème embêtant pour les développeurs d'applications réseau. L'architecture de l'Internet était conçue pour que le seul point commun réellement obligatoire soit le protocole de couche 3, IP. Ce protocole est simple et fournit peu de services. Les développeurs qui voulaient des choses en plus les mettaient dans leurs applications ou, la plupart du temps, comptaient sur les protocoles de transport comme TCP. Si aucun des protocoles de transport existant ne satisfaisaient leurs besoins, ils avaient parfaitement le droit d'en inventer un autre et de le déployer sans rien demander à personne (principe dit « de bout en bout »). Ce modèle ne marche plus guère aujourd'hui. Il est devenu très difficile de faire passer un autre protocole de transport que TCP à travers les innombrables obstacles du réseau (NAT et pare-feux, notamment), et même un protocole applicatif nouveau, tournant sur un port TCP à lui, risque d'avoir le plus grand mal à passer. D'où l'idée de base de WebSocket : faire un protocole de transport au dessus de HTTP, qui va être le seul à passer à peu près partout. WebSocket est donc l'aboutissement d'un processus, qui a mené à ce que le protocole d'unification ne soit plus IP mais HTTP. Bienvenue dans l'Internet d'aujourd'hui « Tout sur le port 80 ».

WebSocket n'est toutefois pas un protocole généraliste, il est conçu pour fonctionner essentiellement dans le cadre du navigateur Web qui charge du code inconnu (typiquement en JavaScript) et qui va ensuite l'exécuter. Ce code pourra utiliser le réseau, dans une certaine limite (vers la même **origine**, cf. RFC 6454 [1]) mais les possibilités de communication offertes précédemment étaient limitées. WebSocket donne à ce code JavaScript (ou écrit dans d'autres langages) les possibilités d'une vraie communication réseau bidirectionnelle.

Avant, les applications s'exécutant sur le navigateur n'avaient pas de moyen simple de faire de telles communications, équivalentes à ce qu'on fait en TCP. Des applications comme la messagerie instantanée, le partage d'un document en cours d'édition ou bien comme des jeux en commun souhaitaient un

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6454.txt

modèle d'interaction plus riche que le traditionnel `GET` du client vers le serveur. Bien sûr, elles auraient pu être extérieures au navigateur, ce qui était certainement plus propre du point de vue architectural. Mais elles se heurtent alors au problème de filtrage décrit plus haut. Et, dans le navigateur, elles dépendaient de XMLHttpRequest ou bien de `<iframe>` et de "*polling*". Par exemple, un code tournant sur le navigateur qui voulait simplement se mettre en attente de donnés émises de manière asynchrone par le serveur n'avait pas d'autres solutions que d'interroger ce dernier de temps en temps. Ce problème est décrit en détail dans le RFC 6202. Il avait plusieurs conséquences fâcheuses comme un surcoût en octets (tout envoi de données nécessitait les en-têtes HTTP complets) ou comme un modèle de programmation peu naturel.

WebSocket vise à résoudre ce problème en transformant HTTP en un protocole de transport. Il réutilise toute l'infrastructure HTTP (par exemple les relais ou bien l'authentification). Passant sur les mêmes ports 80 et 443, on espère qu'il réussira à passer partout. Comme le note un observateur, « WebSocket est un protocole totalement alambiqué pour contourner la stupidité du monde ».

Le résultat n'est donc pas parfait (rappelez-vous que HTTP n'avait pas été conçu pour cela) et le RFC note qu'on verra peut-être un jour les services de WebSocket fonctionner directement sur TCP (personnellement, j'ai des doutes, puisqu'on pourrait aussi bien dans ce cas utiliser des protocoles de transport qui fournissent les mêmes services, comme SCTP - RFC 4960).

Une petite note avant d'attaquer le RFC : si vous avez l'habitude de lire des RFC, vous noterez que celui-ci a des notations originales (section 2.1) comme d'utiliser les tirets bas pour souligner les définitions, les barres verticales pour encadrer les noms d'en-têtes ou de variables et les barres obliques pour les valeurs des variables.

La section 1.2 résume le fonctionnement du protocole (le lecteur pressé du RFC peut d'ailleurs se contenter de la section 1, non normative mais qui contient l'essentiel sur WebSocket). Le principe de base est d'utiliser du HTTP normal (RFC 7230) mais le client ajoute un en-tête `Upgrade:` à une requête `GET`, pour indiquer sa volonté de faire du WebSocket :

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Le serveur répond alors par un code 101 et en indiquant `upgrade` dans l'en-tête `Connection:` et en ajoutant des en-têtes spécifiques à WebSocket :

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Bien sûr, d'autres en-têtes HTTP sont possibles comme les petits gâteaux du RFC 6265. Une fois que le client a fait sa demande, et que le serveur l'a acceptée, il existe une connexion bidirectionnelle entre ces deux acteurs et on peut faire passer les données.

Contrairement à TCP (mais comme dans SCTP), la communication n'est pas un flot d'octets sans structure ; c'est une suite de **messages**, chacun composé d'une ou plusieurs **trames**. Les trames sont typées et toutes les trames d'un même message ont le même type. Par exemple, il existe un type texte, où les trames contiennent des caractères Unicode encodés en UTF-8. Il existe évidemment un type binaire. Et il existe des trames de contrôle, conçues pour l'usage du protocole lui-même.

La section 1.3 se focalise sur la poignée de main entre client et serveur qui lance le protocole (les détails complets étant en section 4). On l'a vu, le client doit ajouter l'en-tête `Upgrade: websocket` pour demander au serveur de basculer en WebSocket (RFC 7230, section 6.7 ; la valeur `websocket` pour cet en-tête est enregistrée à l'IANA <https://www.iana.org/assignments/http-upgrade-tokens/http-upgrade-tokens.xml>, cf. section 11.2, et aussi le RFC 2817, section 7.2). Le client indique également des en-têtes spécifiques à WebSocket comme `Sec-WebSocket-Protocol:` qui permet d'indiquer un protocole applicatif au dessus de WebSocket (`chat` dans l'exemple plus haut). Ces protocoles applicatifs (section 1.9) sont normalement enregistrés à l'IANA pour assurer l'unicité de leurs noms. L'en-tête `Origin:` du RFC 6454 sert à indiquer quelle était l'origine de la page Web qui a chargé le script client. Quand à `Sec-WebSocket-Key:`, son rôle est de permettre de vérifier que la connexion était bien prévue pour être du WebSocket et pas des jeux faits par un programme malveillant qui enverrait des données ressemblant à du WebSocket sur le port 80, sans être passé par la poignée de main normale. Le serveur doit combiner la valeur de l'en-tête `Sec-WebSocket-Key:` avec un GUID (RFC 9562) fixe, `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`. Il passe le résultat par SHA-1 puis par Base64 et retourne ce résultat au client (dans un en-tête `Sec-WebSocket-Accept:`), qui peut alors être sûr que c'est bien ce serveur qui a reçu sa poignée de main. (Voir la section 1.6 sur ce point. Les en-têtes commençant par `Sec` ne peuvent pas être ajoutés via du code XMLHttpRequest normal et, donc, un client JavaScript ordinaire ne peut pas se comporter en client WebSocket.)

À noter que les en-têtes spécifiques de WebSocket ont été ajoutés au registre des en-têtes <https://www.iana.org/assignments/message-headers/perm-headers.html>.

La réponse du serveur utilise le code HTTP 101 (qui avait été prévu de longue date par le RFC 7231, section 6.2.2), qui signifie que le serveur accepte le changement de protocole. Tout autre code indique que le serveur n'accepte pas WebSocket et que le client doit donc continuer en HTTP normal. Ainsi, un serveur HTTP normal refusera l'en-tête `Upgrade:` :

```
% telnet www.bortzmeyer.org http
Trying 2605:4500:2:245b::bad:dcaf...
Connected to www.bortzmeyer.org.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.bortzmeyer.org
Upgrade: websocket

HTTP/1.1 400 Bad Request
Date: Tue, 29 Nov 2011 21:15:34 GMT
Server: Apache/2.2.16 (Debian)
Vary: Accept-Encoding
Content-Length: 310
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

La section 1.4, elle, décrit la fermeture de la connexion (détails en section 7). Elle se fait par l'envoi d'une trame de contrôle ad hoc. Notez que la simple fermeture de la connexion TCP sous-jacente ne suffit pas forcément : en présence d'intermédiaires, par exemple les relais, elle peut être insuffisante.

Un peu de philosophie après ces détails ? La section 1.5 décrit les concepts à la base de WebSocket. Par exemple, l'un des buts était de garder au strict minimum le nombre de bits de tramage. La structure des données que permet WebSocket est donc réduite (séparation des trames, et typage de celles-ci) et toute structuration plus sophistiquée (par exemple pour indiquer des métadonnées) doit être faite par l'application, au dessus de WebSocket.

Dans cette optique « ne pas trop en ajouter », cette section note que WebSocket ajoute à TCP uniquement :

— Le modèle de sécurité du Web, fondé sur la notion d'origine du RFC 6454,
— Un mécanisme de nommage permettant de mettre plusieurs applications sur le même port (c'est le chemin donné en argument de la commande `GET`),
— Un système de tramage, que n'a pas TCP, qui ne connait que les flots d'octets sans structure,
— Un mécanisme de fermeture explicite de la connexion (on l'a dit, TCP seul ne suffit pas, dans la jungle qu'est le Web d'aujourd'hui, truffé de "*middleboxes*").

Et c'est tout. Le reste doit être fait par les applications. Compte-tenu des contraintes spécifiques du Web, WebSocket offre donc pratiquement le même service aux applications que TCP. Sans ces contraintes Web (de sécurité, de fonctionnement à travers les "*middleboxes*"), du TCP brut suffirait.

Voila, vous connaissez maintenant l'essentiel de WebSocket. Le reste du RFC précise les détails. La section 3 décrit les URI WebSocket. Ils utilisent le plan `ws:` (non chiffré, port 80 par défaut) ou le `wss:` (chiffré avec TLS, port 443 par défaut). La section 11 décrit l'enregistrement de ces plans dans le registre IANA <https://www.iana.org/assignments/uri-schemes.html>. Par exemple `ws://example.com/cha` est un URI WebSocket (pour la connexion donnée en exemple au début de cet article), comme `ws://www.3kbo.com:` ou `wss://foobar.example/newsfeed`.

Comment fonctionne le tramage, le découpage du flot de données en trames bien délimitées ? La section 5 le normalise avec précision. Une trame a un type, une longueur et des données. On trouve également quelques bits comme `FIN` qui indique la dernière trame d'un message, ou comme `RSV1`, `RSV2` et `RSV3`, réservés pour de futures extensions du protocole. Une grammaire complète est donnée en section 5.2, en utilisant ABNF (RFC 5234. Les fanas d'ABNF noteront que cette grammaire ne décrit pas des caractères mais des bits, ce qui représente une utilisation originale de la norme.

Le type est nommé "*opcode*" et occupe quatre bits. Les valeurs de 0x0 à 0x7 indiquent une trame de données, les autres sont des trames de contrôle. 0x1 indique une trame de texte, 0x2 du binaire, 0x8 est une trame de contrôle signalant la fin de la connexion, 0x9 une demande d'écho ("*ping*"), 0xA une réponse ("*pong*"), etc. La sémantique des trames de contrôle figure en section 5.5. On y apprend par exemple que des échos non sollicités ("*pong*" non précédé d'un "*ping*") sont légaux et peuvent servir à indiquer qu'une machine est toujours en vie.

On le sait, l'insécurité est une des plaies du Web, on trouve tout le temps de nouvelles manières de pirater les utilisateurs. Les problèmes viennent souvent de nouveaux services ou de nouvelles fonctions qui semblent super-cool sur le moment mais dont on découvre après qu'elles offrent plein de pièges. Il n'est donc pas étonnant que la section 10, sur la sécurité, soit longue.

D'abord, le serveur WebSocket doit se rappeler qu'il peut avoir deux sortes de clients (section 10.1) : du code « embarqué » par exemple du JavaScript exécuté par un navigateur et dont l'environnement

d'exécution contraint sérieusement les possibilités. Par exemple, l'en-tête `Origin:` est mis par ce dernier, pas par le code Javascript, qui ne peut donc pas mentir sur sa provenance. Mais un serveur WebSocket peut aussi être appelé par un client plus capable, par exemple un programe autonome. Celui-ci peut alors raconter ce qu'il veut. Le serveur ne doit donc pas faire confiance (par exemple, il ne doit pas supposer que les données sont valides : il serait très imprudent de faire une confiance aveugle au champ "*Payload length*", qu'un client malveillant a pu mettre à une valeur plus élevée que la taille de la trame, pour tenter un débordement de tampon).

WebSocket ayant été conçu pour fonctionner au dessus de l'infrastructure Web existante, y compris les relais, la section 10.3 décrit les risques que courent ceux-ci. Un exemple est l'envoi, par un client malveillant, de données qui seront du WebSocket pour le serveur mais qui sembleront un `GET` normal pour le relais. Outre le tramage, la principale protection de WebSocket contre ce genre d'attaques est le masquage des données avec une clé contrôlée par l'environnement d'exécution (l'interpréteur JavaScript, par exemple), pas par l'application (section 5.3 pour en savoir plus sur le masquage). Ainsi, un code JavaScript méchant ne pourra pas fabriquer des chaînes de bits de son choix, que WebSocket transmettrait aveuglément.

Un peu d'histoire et de politique, maintenant. WebSocket a une histoire compliquée. L'idée de pousser les informations du serveur vers le client (connue sous le nom de Comet) est ancienne. Le protocole WebSocket (dont l'un des buts est justement cela) a été créé par Ian Hickson (qui a aussi écrit les premiers projets de RFC mais n'apparait plus comme auteur). Le groupe de travail WHATWG a ensuite beaucoup participé. La plupart des mises en œuvre de WebSocket citent ce groupe ou bien les anciens "*Internet-Drafts*", écrits avant la création du groupe de travail IETF HyBi `<http://tools.ietf.org/wg/hybi>` en janvier 2010.

Le principe même de WebSocket a souvent été contesté. Pourquoi passer tant d'efforts à contourner les problèmes de l'Internet aujourd'hui (notamment les "*middleboxes*" abusives) plutôt qu'à les résoudre ? Un intéressant texte de justification a été écrit à ce sujet `<http://www.ietf.org/mail-archive/web/hybi/current/msg02605.html>`. Notez qu'il inclut des exemples de code. Le groupe HyBi a connu de vives discussions, avec menaces de scission (« À WHATWG, c'était mieux, on va quitter l'IETF si ça n'avance pas »). Un des points d'affrontement était par exemple les problèmes de sécurité (résolus par la solution du masquage). Cela s'est ensuite arrangé, début 2011.

Si vous voulez vous lancer dans la programmation d'applications WebSocket tournant dans le navigateur, regardez l'API `<http://dev.w3.org/html5/websockets/>`. Aujourd'hui, on trouve des implémentations de WebSocket pour les différents serveurs (GlassFish, Jetty, Apache). Dans les environnements de développement, on trouve du WebSocket chez Django `<http://pypi.python.org/pypi/django-websocket>` et bien d'autres. Chez les clients, Firefox l'a en théorie depuis la version 6, Chrome et Internet Explorer l'ont annoncé pour une version prochaine. Bon, en pratique, c'est compliqué `<http://caniuse.com/#websockets>`, et la démo `<http://websocket.org/echo.html>` ne donne pas toujours les résultats attendus (elle me refuse mon Firefox 7 mais accepte un Chrome).

Attention si vous lisez les sources, pas mal d'implémentations ont été faites en suivant des vieilles versions de la spécification de WebSocket, antérieures à sa normalisation à l'IETF. Cela se sent par exemple dans les noms des variables utilisés. (Un terme comme « "*opcode*" » pour le type est dans le RFC mais pas toujours dans les programmes.)

En dehors du monde des navigateurs, si vous voulez programmer du WebSocket, vous avez, entre autres :

— En Python, la bibliothèque pywebsocket <http://code.google.com/p/pywebsocket/>, également ws4py <http://www.defuze.org/archives/271-ws4py-websocket-client-and-serve html>, websocket-client <http://pypi.python.org/pypi/websocket-client/> et python-websocket <https://github.com/mtah/python-websocket>. Ne me demandez pas un avis, je n'ai pas eu l'occasion et le temps de les comparer sérieusement. Je crains que, comme souvent en Python, il y ait pléthore d'implémentations plus au moins au point, et aucune « de référence ».
— En Go, il y a un paquetage dans la bibliothèque standard <http://golang.org/pkg/websocket/ >. Un exemple pour un chat est décrit <http://gary.beagledreams.com/page/go-websocket-chat. html>.
— En C++, il y a la bibliothèque websocketpp <https://github.com/zaphoyd/websocketpp>.
— Enfin, il existe un article de Wikipédia de comparaison des mises en œuvres.

Si vous voulez jouer plutôt avec un programme client comme curl, vous avez un bon article qui explique comment faire du WebSocket avec curl <http://blogfranz.blogspot.com/2009/12/hello-websockets-v html>.

Si vous cherchez des fichiers pcap de Websocket, on en trouve sur pcapr <http://www.pcapr. net/browse?q=websocket> mais attention, la plupart concernent des mises en œuvres de versions antérieures du protocole. Mais Wireshark n'a pas l'air encore capable de décoder le "*framing*" Websocket. (Une mise en œuvre existe <http://blog.igut.fr/post/2012/04/20/Impl%C3%A9mentation-de-WebSoc

Les autres registres IANA pour WebSocket sont en <https://www.iana.org/assignments/ websocket/websocket.xml>. Il existe un site de référence sur WebSocket <http://websocket. org/>.

Un exemple de service publiquement accessible et utilisant WebSocket est le service RIS Live <https: //www.bortzmeyer.org/ris-live.html>.

Enfin, si vous voulez d'autres articles sur WebSocket, j'ai beaucoup apprécié celui de Brian Raymor <http://www.sitepoint.com/websockets-stable-and-ready-for-developers/>.