WebSocket

Jérôme Bousquié
IUT de Rodez – Université de Toulouse 1 Capitole
50, avenue de Bordeaux
12000 Rodez

Résumé

Le protocole Websocket a fait l'objet d'un RFC (RFC 6455[1]) publié fin 2011 et est actuellement en cours de standardisation au W3C à l'état de Candidate Recommendation[2] depuis septembre 2012.

Ce protocole permet d'établir sur les ports web standards une connexion permanente et bidirectionnelle entre le client et le serveur distant.
Il devient alors possible de procéder à l'échange de données instantanément, hors du mode requête/réponse classique de http, et même à du « push » de données du serveur vers le client, sans que ce dernier n'ait émis la moindre requête.

Les dernières versions des navigateurs qui se targuent d'implémenter la spécification en cours de HTML5[3] supportent d'ores et déjà le protocole Websocket.

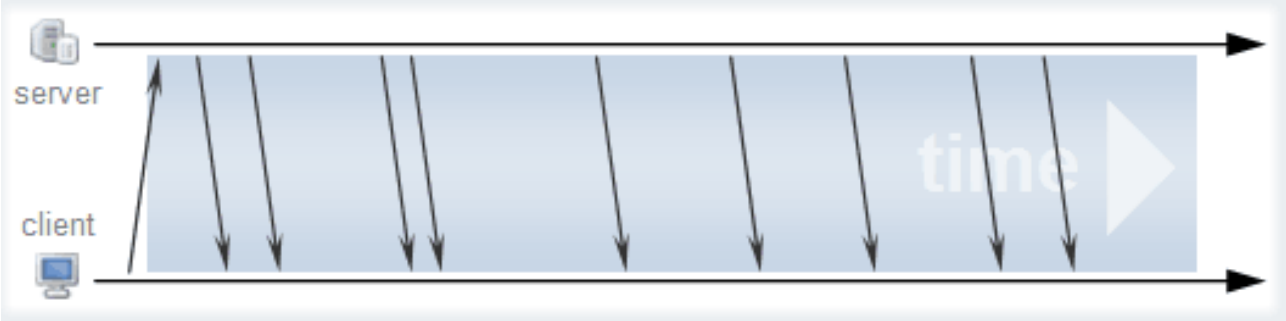
Mots-clefs

Web, bidirectionnel, push, HTML5.

1 Problématique

Le navigateur Web est devenu au fil du temps, bien plus qu'un simple client d'affichage de sites Web, un environnement d'exécution de services ou d'applications connectées de plus en plus riches fonctionnellement. Le mode requête/réponse propre au protocole http devient alors parfois très contraignant quand une application cherche à simuler le temps réel ou à simplement simuler le « push », c'est-à-dire l'envoi d'informations du serveur vers le client sans que ce dernier n'ait émis de requête. Quelques techniques de développement existent pour parvenir à ces fins, mais elles se heurtent à plusieurs obstacles.

1.1 Le streaming

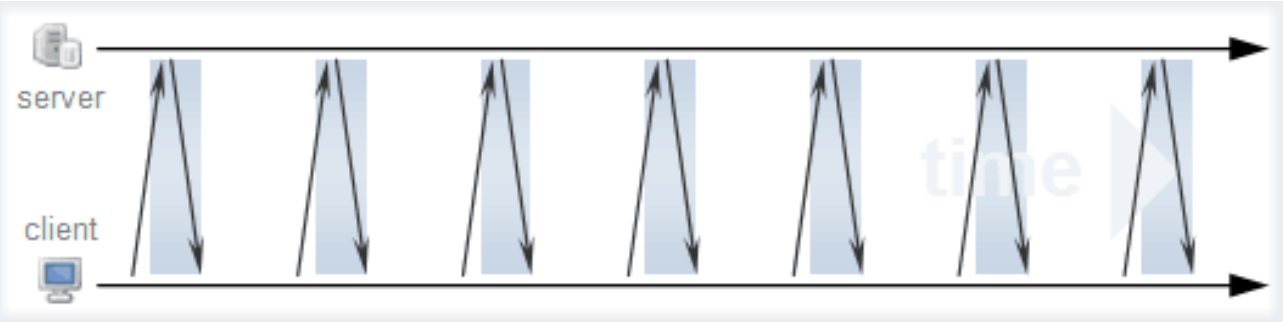


[4]

Le client émet une requête et le serveur de streaming commence à répondre immédiatement et continue indéfiniment jusqu'au moment où le client décide d'interrompre la connexion. Ce procédé semble idéal. En effet, les données parviennent au client sur une connexion pré-établie une fois pour toute quand le serveur le décide.

La connexion peut cependant traverser des relais (ex proxy) qui ne veulent pas jouer le jeu et attendent une réponse complète avant de la relayer. Le client lui même peut aussi attendre la réponse complète avant de la traiter. Enfin le serveur peut être confronté à la multiplication de processus zombies de clients n'écoutant plus mais mal déconnectés.

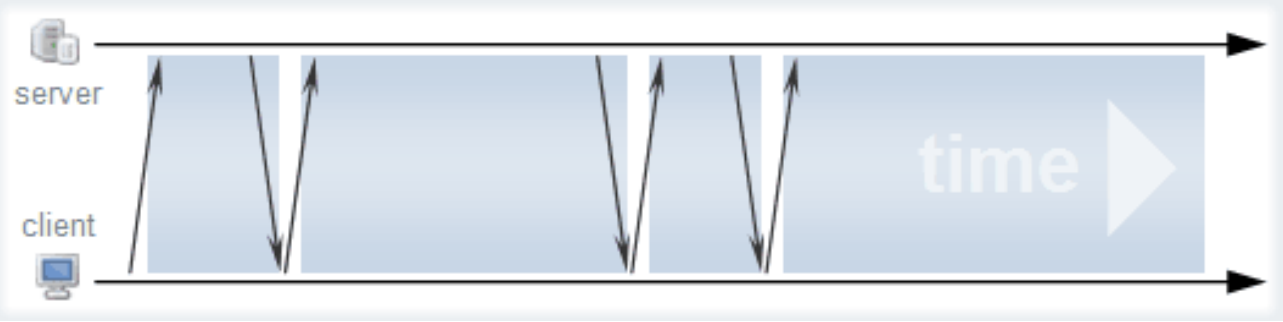
1.2 Le short polling



[4]

Le client recharge la page, ou une partie de celle-ci, à une fréquence régulière, fréquence d'autant plus élevée qu'on souhaite simuler le « temps réel ». Cette méthode extrêmement simple à mettre en œuvre pose un problème de consommation de ressources flagrant dès que l'on multiplie les clients. En effet, même pour recevoir une réponse vide du serveur, chaque client va émettre quelques centaines d'octets d'en-têtes http. Si plusieurs centaines de clients émettent plusieurs dizaines de requêtes par seconde simultanément, la consommation de bande passante va vite s'en ressentir ainsi que l'utilisation du serveur si celui-ci réalise, par exemple, à chaque fois un accès à une base de données pour *in fine* signifier au client qu'aucune donnée n'a été modifiée depuis la requête précédente.

1.3 Le long-polling

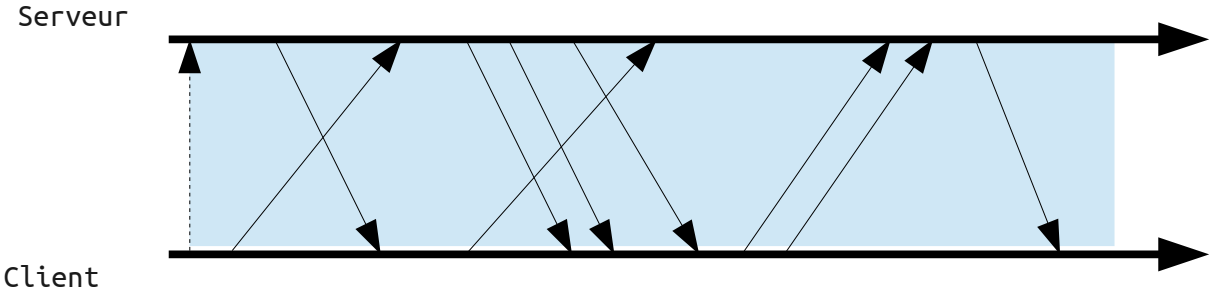


[4]

Le client émet une requête en arrière-plan (ajax) depuis une page html téléchargée et attend la réponse. Le serveur ne répond que lorsqu'un événement choisi (mise à jour d'une donnée par exemple) se produit et ferme la connexion. Le client reçoit la réponse à traiter et ré-émet immédiatement une nouvelle requête. Le serveur attend à nouveau un événement déclencheur pour répondre. Le processus est ensuite réitéré.

Comme dans le premier cas, les proxies peuvent devenir problématiques avec cette technique. En effet, un proxy peut considérer qu'une connexion http trop longue est une connexion qui a échoué faute d'un serveur capable de répondre. Il est à noter que ce problème concerne aussi les WebSockets et est abordé au 5. Cette technique limite néanmoins grandement l'envoi de requêtes inutiles et est largement utilisée dans les applications web existantes : chat, éditeurs ou tableurs en ligne, jeux, etc.

1.4 WebSocket



Le client émet une unique requête http de connexion auprès du serveur de Websockets par un script javascript embarqué dans une page HTML. Dès que le serveur a accepté la connexion par une mise à jour (upgrade) du protocole, un canal birectionnel est ouvert entre ce dernier et le client. Dès lors, le client comme le serveur peuvent envoyer des messages dans ce canal quand ils le souhaitent.

Un proxy intermédiaire incapable de traiter la méthode http CONNECT ou n'autorisant pas le tunneling http peut ici encore empêcher l'établissement de la connexion (cf 5).

2 Connexion : la poignée de main

La demande de connexion au serveur de Websockets est typiquement réalisée par du code javascript exécuté dans une page html. Il s'agit de l'envoi d'une requête http GET classique à laquelle sont ajoutés quelques en-têtes. Cette requête particulière et sa réponse constituent la poignée de main initiale avant l'ouverture de la connexion websocket proprement dite. L'URI Websocket utilise les plans *ws :* et *wss :* enregistrés à l'IANA[5] . exemple : *wss://my.server.net/xmpp*

```
GET /socket HTTP/1.1
Host: websocket.server.org
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Protocol: soap, xmpp, stomp
Sec-WebSocket-Version: 13
Origin: http://mon.siteweb.net
```

L'en-tête Upgrade demande au serveur une mise à jour du protocole HTTP vers le protocole Websocket.

L'en-tête Sec-WebSocket-Protocol précise quels protocoles applicatifs le client aimerait utiliser au travers de Websocket. Le RFC recommande d'utiliser, par souci d'unicité, les noms enregistrés auprès de l'IANA, voire de les suffixer par le nom de domaine du serveur (ex : xmpp.server.org).

L'en-tête Sec-WebSocket-Version précise la version du protocole demandée, la version 13 étant la version définitive du procole décrite dans le RFC.

L'en-tête Origin précise l'origine Web du script ayant émis la requête de poignée de main.

La Sec-WebSocket-Key est une valeur permettant au client de vérifier qu'il dialogue bien ensuite avec le serveur avec lequel il a échangé la poignée de main. Le serveur combine cette valeur avec un identifiant global unique (GUI), 258EAFA5-E914-47DA-95CA-C5AB0DC85B11, hache le résultat avec SHA-1 et encode ce hachage en base64 avant de le retourner au client dans l'en-tête Sec-WebSocket-Accept de la réponse.

La poignée de main en réponse est beaucoup plus simple. Si la requête est acceptée, le serveur répond par un code HTTP 101 d'acceptation de changement de protocole et en ajoutant les en-têtes WebSocket déjà mentionnées dans la requête.

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

Une fois cette poignée de main échangée, la connexion bidirectionnelle est établie entre le client et le serveur.

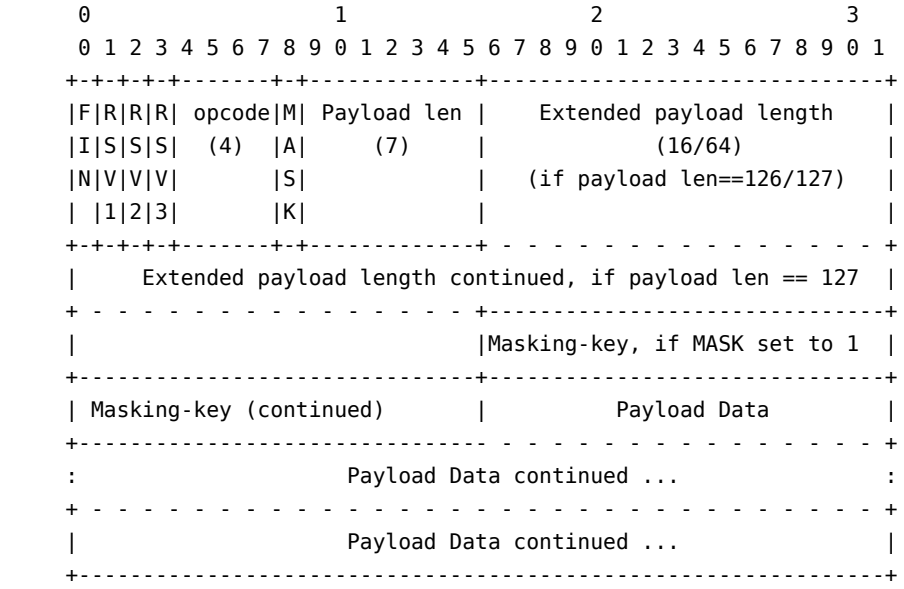
3 Les trames

Les données échangées dans le canal bidirectionel sont structurées en trames de trois types : les trames de contrôle permettant d'échanger des commandes du protocole même, les trames « texte » transportant des caractères encodés en UTF-8 et les trames de données binaires.

La structure de chaque type de trame ne peut pas être détaillée dans cet article pour des raisons de concision. On retiendra pour l'essentiel que la longueur de chaque trame peut différer (*extended, or not, payload length*), qu'une clé de masque (*masking-key*) peut être intégrée dans la trame pour masquer les données transportées (*payload data*) et qu'elle commence toujours par une suite de bits de contrôle suivi du type de trame (*opcode*). Le RFC précise que le non-respect de la construction de la structure ou d'une des valeurs de contrôle entraîne *de facto* la rupture de la connexion.

À noter : il existe une trame de contrôle demandant la fermeture explicite de la connexion Websocket, la fermeture de(s) la connexion(s) TCP sous-jacente(s) ne suffisant pas en présence de relais (i.e proxy)

Ci-dessous la structure d'une trame décrite dans le RFC.



4 Sécurité

Le protocole étant initialement conçu pour étendre le comportement des navigateurs, WebSocket se base sur l'implémentation dans le browser des contraintes relatives à l'origine Web décrite dans le RFC 6454[6] , implémentation encore nommée *Same Origin Policy ou SOP*. Pour faire simple, le script initialisant la connexion Websocket avec un serveur n'aura accès qu'aux ressources ayant la même origine que lui dans une page Web.

Évidemment si la connexion n'est pas initiée depuis un navigateur, mais depuis un programme externe (exemple : librairie cliente), la *SOP* ne pourra pas être appliquée. La chaîne de sécurité se poursuit donc d'une part par l'échange correct et la vérification des clés Sec-WebSocket-Key dans la poignée de main sans quoi la connexion n'est pas établie, ce qui limite la possibilité de réception de messages d'un programme malicieux.

Cette chaîne de sécurité continue ensuite par le contrôle des trames et le masquage de ces dernières. Une seule incohérence dans la validité des données par rapport au modèle décrit dans le RFC entraine *ipso facto* l'interruption de la connexion ce qui limite ici encore les possibilités d'injection de données malveillantes.

Enfin la connexion elle-même peut utiliser TLS de bout en bout comme n'importe quelle connexion https.

5 Proxies et firewalls

Les ports standards utilisés par le protocole WebSocket sont les ports standards de http, à savoir les ports 80 et 443. Aussi les mêmes règles de filtrage ou de passage du trafic http s'appliqueront à WebSocket sur les firewalls rencontrés sur la connexion.

Contrairement au cycle requête/réponse de http, la connexion WebSocket peut rester ouverte longtemps. Les éventuels proxies sur le chemin peuvent alors, selon leur comportement, aussi autoriser ce type de connexion longue ou y mettre fin sans préavis.

Le protocole WebSocket ne gère pas la présence de proxies[7] . Il se contente d'émettre la poignée de main http demandant la mise à jour puis de gérer les émissions et réceptions de trames. Selon que client tente une connexion chiffrée (*wss://)* ou non chiffrée (*ws://*) et qu'il aura connaissance de la présence d'un proxy (proxy explicite) ou non (proxy transparent) sur le chemin, quatre cas sont donc envisageables.

Si la connexion n'est pas chiffrée et que le proxy est déclaré explicitement dans le client, ce dernier va émettre une requête http CONNECT. Il suffit alors que le proxy autorise la méthode http CONNECT vers le port 80 pour que la connexion soit alors établie avec le serveur de WebSocket. Par ailleurs, non seulement la réalisation de cette condition est à réitérer autant de fois que de proxies sont présents sur le chemin, mais chacun d'eux devra aussi participer au mécanisme de mise à jour du protocole en ré-émettant l'en-tête *Connection : Upgrade* vers le relais suivant.

Si la connexion n'est pas chiffrée et que le proxy n'est pas connu du client, celle-ci risque fortement d'échouer. En effet, le client n'émet alors aucune requête http CONNECT, le proxy transparent s'attend alors à voir passer du trafic http classique. Dans ce cas, il risque fortement de supprimer l'en-tête *Connection : Upgrade* dans la requête relayée. Quand bien même l'en-tête serait conservé sur un proxy transparent doté d'une configuration particulière, il ne saurait pas relayer correctement les trames WebSocket qui viendraient ensuite car elles ne s'apparentent en rien à du trafic http régulier.

Si la connexion est chiffrée et que le proxy est connu du client, on se retrouve exactement dans le même cas qu'une connexion https sur le port 443 avec l'envoi initial classique d'une requête http CONNECT suivie de la négociation TLS. Le reste (poignée de main WebSocket et circulation des trames) passe ensuite dans le tunnel TLS ouvert. Cette configuration fonctionne donc dans tous les cas si https est déjà correctement relayé par le proxy.

Enfin, si la connexion est chiffrée et que le proxy n'est pas connu du client, il est très probable que le proxy transparent laisse déjà passer le trafic chiffré vers le port 443 (https) et le protocole WebSocket passera alors lui aussi sans encombre de la même manière.

En résumé, à moins d'avoir la connaissance complète du réseau entre les clients et le serveur de WebSocket, chose parfois possible sur le réseau interne de l'organisation, mais jamais sur l'Internet, il est recommandé dans la pratique de présenter les services WebSocket avec TLS sur le port standard 443 pour permettre à la majorité des clients d'y accéder sans encombre.

cf schéma décisionnel en annexe.

6 Côté client

S'il existe des librairies clientes WebSocket pour la majorité des langages de programmation actuels, la destination principale du protocole reste le navigateur pour lequel il a été conçu initialement. Le W3C propose donc une recommandation, encore à l'état de Candidate Recommandation au moment de la publication de cet article, qui détaille l'API [2] permettant à une page Web d'utiliser WebSocket.

Cette API est relativement simple pour le développeur : un nouvel objet Javascript seulement, nommé WebSocket, est implémenté. Cet objet dispose de méthodes et de propriétés lui permettant de traiter les nouveaux évènements liés par

exemple à l'ouverture ou à la fermeture d'une connexion, à la réception d'un message, ou d'émettre directement des messages.

Ci-dessous quelques exemples commentés :

```
// Création d'un objet WebSocket et connexion au serveur ws://ws.server.net
var ws = new WebSocket('ws://ws.server.net');

// Envoi de données au serveur : data peut être une variable texte ou binaire
ws.send(data) ;

// Réception d'un message : la fonction affiche le contenu du message
ws.onmessage = function(event) { console.log('message reçu : ' + event.data) ; } ;

// Traitement de la connexion : la fonction envoie un message au serveur
ws.onopen = function(event) { ws.send('mon browser est maintenant connecté') ; } ;

// Traitement de la déconnexion : cette fonction dit « bye bye » au serveur
ws.onclose = function(event) { ws.send('bye bye, baby') ; } ;

// Traitement des erreurs : cette fonction affiche le message d'erreur
ws.onerror = function(event) { console.log('erreur : ' +event.data) ; } ;
```

Dans la pratique, le développeur échange habituellement des données structurées dans le format JSON nativement connu par Javascript.

7 Côté serveur

Ici encore, il existe des librairies serveurs correspondantes aux librairies clientes dans la plupart des langages de programmation courants. Cependant pour un usage en production, on pourra préférer déployer un serveur spécialisé.

Contrairement à un serveur Web qui, par défaut, sert des ressources (fichiers HTML, images, etc) au client, un serveur de WebSocket ne fait rien d'autre que d'assurer la connexion. Il faut donc implémenter son comportement par du code. Le choix du langage de programmation devient alors non négligeable, que ce soit pour des raisons de performances, d'intégration au SI existant, ou tout simplement de compétence des développeurs.

Ainsi, s'il existe un module apache-websocket[8] pour Apache qui permet au développeur système d'implémenter directement en C/C++ les services fournis par son serveur de WebSockets, les organisations ont tendance à préférer des solutions comme Jetty[9] ou Kaazing[10] (Java) pour des raisons d'intégration avec les autres briques de leur système d'information en général majoritairement codées en Java.

Néanmoins, comme WebSocket est un protocole dédié au navigateur et que l'API cliente de celui-ci impose l'usage du Javascript, il est pertinent d'envisager le déploiement de nouveaux types de serveurs proposant l'emploi de ce langage comme Node.js[11] (Javascript) ou le polyglotte Vert.x[12] qui comprend aussi bien Java que Javascript, CoffeeScript, Ruby, Python ou Groovy.

Le choix d'un serveur Javascript permet au développeur du code client de programmer le code côté serveur avec la même philosophie, comme le montre ci-dessous l'exemple commenté sur un serveur Node.js incluant la librairie WebSocket-Node[13] .

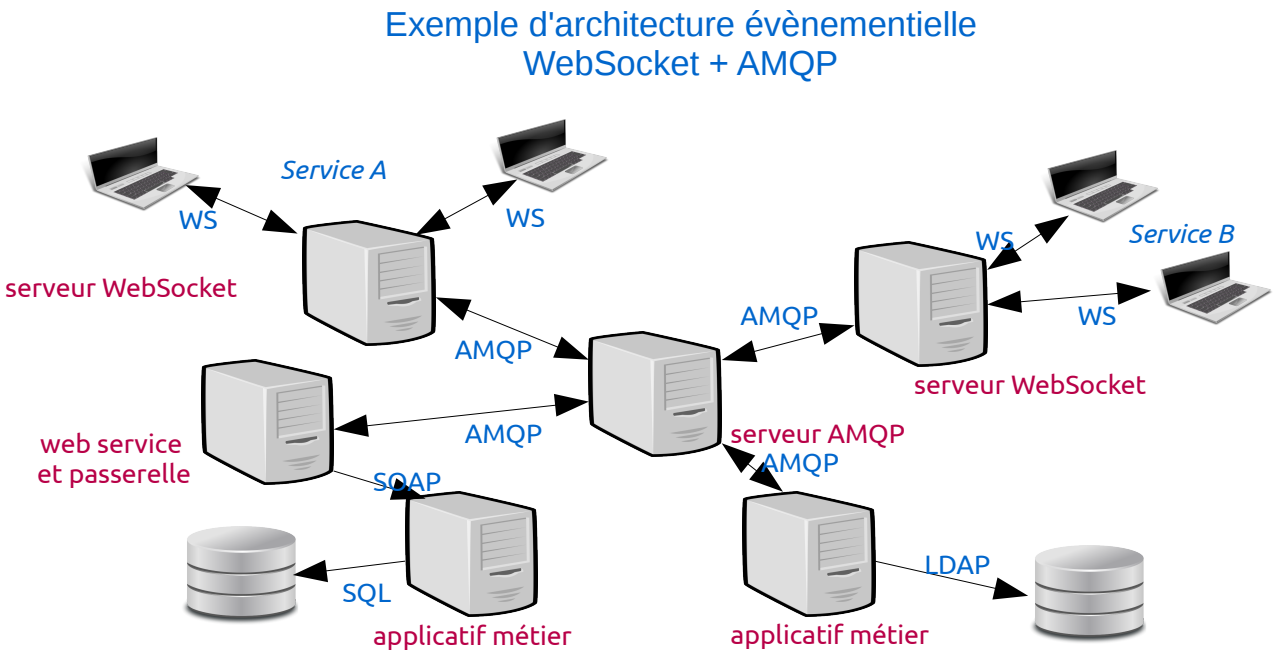
```
// Déclaration d'un objet connexion d'un client
var connection = request.accept(null, request.origin);

// Traitement des messages reçus d'un client
connection.on('message', function(msg) {
```

```
// si message texte, on appelle traiteTexte() définie ailleurs dans le code
if (msg.type === 'utf8') {
  var reponseTexte = traiteTexte(msg.utf8Data) ;

  // on renvoie au client la résultat de traiteText()
  connection.sendUTF(reponseTexte);     }

// sinon, si message binaire, on appelle traiteBinaire() définie ailleurs
else if (msg.type === 'binary') {

  // et on retourne son résultat aussitôt au client
  var reponseBinaire = traiteBinaire(msg.binaryData) ;
  connection.sendBytes(reponseBinaire); }      };
```

8 Exemple d'architecture SI

Le protocole WebSocket est particulièrement bien adapté pour transmettre dans les deux sens de l'information instantanée qui pourrait être relayée entre différents nœuds du SI par le protocole AMQP[14] .

Exemple : des web-services proposés par des briques applicatives du SI ou des scripts de surveillance génèrent, en fonction d'évènements préalablement surveillés, des messages sur une architecture AMQP. Ces messages, bruts ou traités, peuvent simplement être transmis à une interface utilisateur web (un navigateur) via un serveur de WebSockets.



9 Exemple d'architecture applicative

Les applications Web actuellement déployées dans un modèle d'architecture 3-tiers sont généralement structurées de la sorte :

- un serveur de base de données (SGBD) assure le stockage et la persistance des données de l'application,

- un serveur d'application reçoit les demandes du client, assure les traitements « métier », il récupère et sauve des données sur le serveur de base de données, il traite les requêtes http du client et élabore les pages web servies au client,

- un client web se charge uniquement d'afficher l'interface utilisateur et transmettre les saisies utilisateur au serveur d'application.

Le protocole WebSocket, ainsi que les nouvelles fonctionnalités proposées par HTML5, comme la possibilité de stocker des données dans une base côté client ou la capacité à lire ou écrire des fichiers locaux, permet de déplacer le code « métier » vers le client de cette façon :

- le client embarque le code métier ainsi qu'une base de données locale et établit une connexion persistante avec le serveur de WebSockets,

- le serveur de WebSockets notifie le client de changements dans les données partagées et récupère en permanence les données émises par le client que ce soit à des fins de notification ou de sauvegarde,

- le serveur d'application peut être réduit à sa plus simple expression de partage ou notification des données communes à plusieurs clients,

- le serveur de base de données continue d'assurer la persistance et le stockage des données.

10 Conclusion

Le protocole WebSocket, parce qu'il utilise les ports standards du Web généralement accessibles depuis la plupart des réseaux et parce qu'il est simple à utiliser côté client, pourrait voir ses usages se développer rapidement. Pour l'instant, les couches autres que la connexion et le transport (le RFC ne définit d'ailleurs aucun autre service), comme la gestion de l'authentification ou des sessions - comme XMPP -, des queues ou du routage des messages - comme AMQP -, doivent encore être implémentées à la main par du code maison côté serveur. Les serveurs, ou les librairies, qui proposeront ces services au dessus de WebSocket pourraient connaître rapidement un certain succès et accroître encore le déploiement de cette technologie.

Bibliographie

[1] RFC 6455 The WebSocket Protocol : http://tools.ietf.org/html/rfc6455

[2] W3C Candidate Recomendation The WebSocket API : http://www.w3.org/TR/websockets/

[3] W3C Candidate Recomendation HTML5 : http://www.w3.org/TR/html5/

[4] Meteor server documentation, licence GNU GPL : http://meteorserver.org/

[5] S. Bortzmeyer. RFC 6455. http://www.bortzmeyer.org/6455.html

[6] RFC 6454 The Web Origin Concept : http://tools.ietf.org/html/rfc6454

[7] P. Lubbers. How HMTL5 WebSockets Interact With Proxy Server, Mars 2010 : http://www.infoq.com/articles/Web-Sockets-Proxy-Servers

[8] Apache WebSocket module : https://github.com/disconnect/apache-websocket

[9] Jetty server : http://www.eclipse.org/jetty/documentation/current/websockets.html

[10] Kaazing server : http://kaazing.com/products/kaazing-websocket-gateway/websocket-emulation/

[11] Node.js : http://nodejs.org/

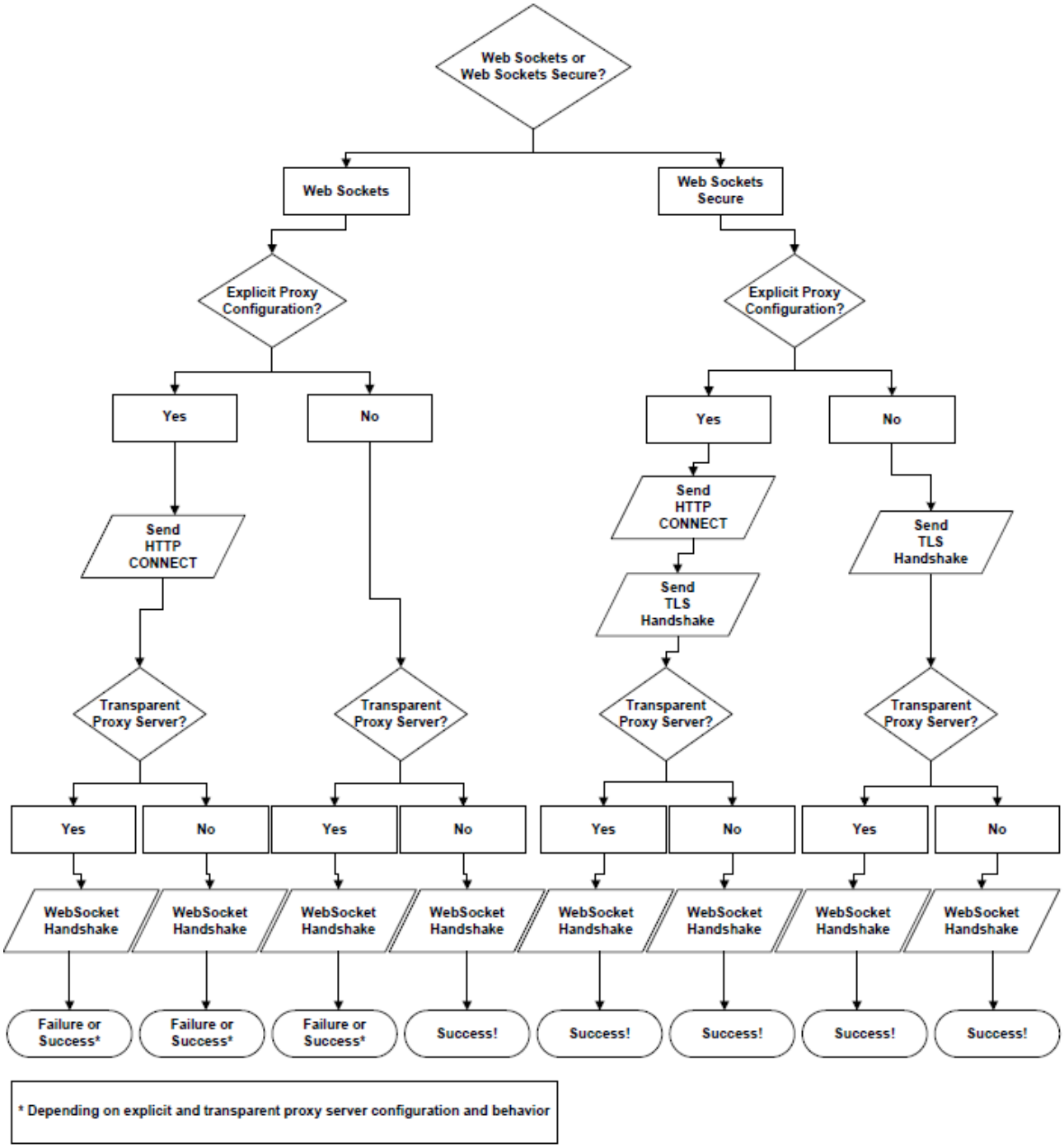
[12] Vert.x : http://vertx.io/core_manual_java.html#web-sockets

[13] WebSocket-Node : https://github.com/Worlize/WebSocket-Node

[14] AMQP Protocol : http://www.amqp.org/

ANNEXE

Schéma décisionnel : passage des proxies



[7]