Étude technique

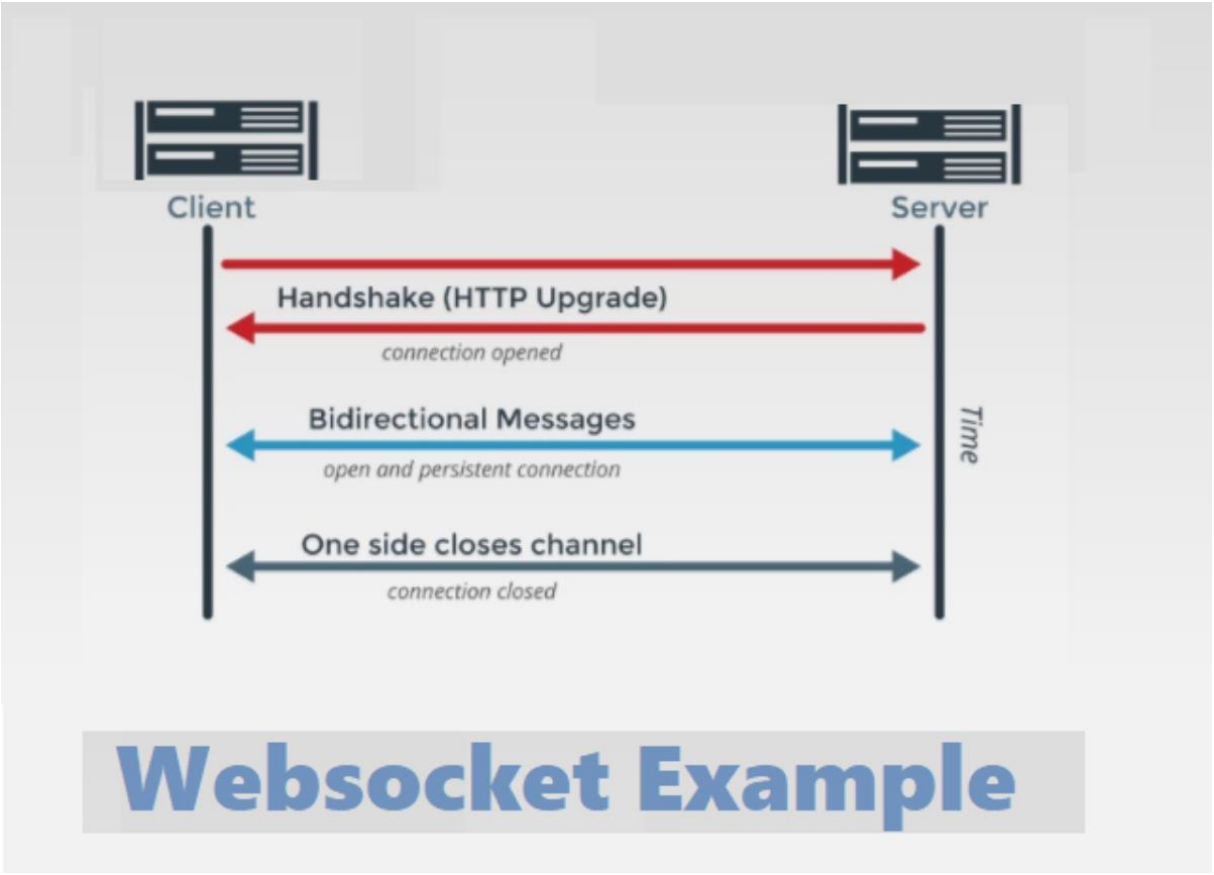
Les Web-sockets

Le protocole Web-Socket vise à développer un canal de communication sur un socket TCP entre les navigateurs et les serveurs web.

Contrairement aux requêtes HTTP(S), qui sont une demande du client vers le serveur ces canaux de communications sont utilisables dans les deux sens.

Ils permettent donc :

- La notification au client d'un changement d'état du serveur
- L'envoi de données en mode « pousser » (méthode Push) du serveur vers le client (sans que ce dernier ait à effectuer une requête)



L'interactivité croissante des applications web avec leurs serveurs, ainsi que l'amélioration des performances des navigateurs, a rendu nécessaire le développement de techniques de communications bidirectionnelles.

A savoir, l'utilisation réseau est minimale seules les informations nécessaires circulent.

Il n'y que très peu de latence quand une donnée est changée car la page entière n'est pas rechargée, seule la donnée en question.

L'utilisation du HTTPS est grandement recommandée pour la requête « d'ouverture » du socket. Car c'est déjà plus sécurisé mais aussi que les sockets peuvent être bloqués par les différents proxys.

Les utilisations possibles

Le domaine d'utilisation le plus courant est celui d'une messagerie en direct un « chat ».

Si des collègues gestionnaires ont besoin de données en temps réel pour éviter les conflits et d'être à jour en permanence avec le serveur, une implémentation dans : Rapid, Base Auto, Arès et pourquoi pas Coffre.

Cela pourrait s'appliquer aussi à PCIO ou à Fiches Réflexes, s'il y a des informations à faire parvenir rapidement.

Problématiques de sécurité

Contrairement aux requêtes standards, les Web sockets ne sont pas soumis à la politique SOP, c'est à dire qu'une origine B peut lire la réponse (ou plus exactement les données poussées par l'origine A) d'une connexion établie entre ces deux origines.

Cela peut conduire à des attaques de type CSRF ce qui rend possible à l'attaquant de récupérer des données potentiellement sensibles de la victime.

Par exemple, admettons qu'un site malicieux (malicious.com) tente de récupérer les données de profile de sa victime via la page malicious.html.

Lorsque la victime visite la page malicieuse (en s'étant authentifié au préalable sur le site cybersecurity.com), son cookie de session est envoyé dans la requête de connexion au Web socket.

Le serveur légitime approuve la connexion et envoi par Web socket le profil de l'utilisateur (récupéré donc par l'attaquant)

Performances

Tout d'abord, lorsqu'on utilise du polling ou toute autre technique sur HTTP, on utilise forcément une connexion dédiée pour recevoir des données poussées du serveur vers le client. Puis une autre connexion pour envoyer des données depuis le client (via une requête HTTP classique).

Une connexion Web-socket, quant à elle, est full duplex. C'est à dire que la même connexion est utilisée pour recevoir et envoyer des données. Le serveur gère donc moins de connexions simultanément et moins d'ouvertures et fermetures de connexions.

Il faut cependant souligner que les implémentations serveur doivent être adaptées à la gestion d'un grand nombre de connexions simultanées et continues (utilisation d'un pool de threads plutôt qu'un thread par socket ouverte, entrées-sorties non bloquantes, notifications par l'OS de l'activité des sockets, …), pour une bonne utilisation des Web-sockets.

Ensuite, en termes de bande passante, la taille du message minimum du protocole Web-socket est de 2 octets (0x00 pour le début du message et 0xFF pour la fin). Avec HTTP, il faut compter au moins 200 octets pour un header.
Quand il s'agit d'envoyer des messages de maintien de connexions, avec un nombre important de clients connectés, l'occupation de la bande passante devient non négligeable.

Les Web-sockets sont un vrai plus en termes d'optimisation de temps de réponses et de réception par le client. Une fois bien maîtrisés (sécurisés) et utilisés dans les bonnes applications ils sont efficaces.

En PHP il existe :

- Ratchet - Ratchet est une bibliothèque PHP à couplage libre qui fournit aux développeurs des outils pour créer des applications bidirectionnelles en temps réel entre les clients et les serveurs sur Web sockets.
- Php-websocket - Implémentation simple de PHP Web socket pour PHP 5.3.
- Phpws - PHP Web Socket server.
- Grès - Micro-Framework pour construire une API en temps réel.

Exemple d'un serveur Web-socket de chat avec Docker

Nous allons étudier un exemple complet avec la librairie PHP « Ratchet »

Ratchet est entièrement compatible PSR-4, donc il joue naturellement bien avec les autres. Tirant parti des composants de Symfony2 il ne pose pas soucis avec ceux qui y sont familiers

Prérequis

- Il vous faut la version 5.4.2 de PHP au minimum
- Ensuite, il vous faut Composer, le "package manager" de PHP.

Création du serveur en PHP

Installation des prérequis

La première étape consiste à installer la librairie "Ratchet" depuis votre terminal :

```
composer require cboden/ratchet
```

Ou depuis votre composer.json en ajoutant les lignes suivantes :

```
"autoload": {
        "psr-4": {
        "": "src/"
        },
"require": {
        "cboden/ratchet": "^0.4.2"
        },
```

Ensuite lancez l'installation des différentes librairies PHP grâce à la commande

```
composer install
```

Mise en place du serveur

Attention - Notez que le fichier PHP que l'on va créer ici n'est pas destiné à être appelé via un navigateur tiers mais à être exécuté directement depuis votre serveur.

Avec l'aide de la documentation de Ratchet le code minimum de votre serveur de chat doit ressembler à ça :

```
<?php
require __DIR__.'/vendor/autoload.php';

use Ratchet\Server\IoServer;
```

```php
use Ratchet\Http\HttpServer;
use Ratchet\WebSocket\WsServer;
use Ratchet\MessageComponentInterface;
use Ratchet\ConnectionInterface;

define('APP_PORT', 8080);

class ServerImpl implements MessageComponentInterface {
    protected $clients;

    public function __construct() {
        $this->clients = new \SplObjectStorage;
    }

    public function onOpen(ConnectionInterface $conn) {
        $this->clients->attach($conn);
        echo "Nouvelle connection ({$conn->resourceId}).\n";
    }

    public function onMessage(ConnectionInterface $conn, $msg) {
        echo sprintf("Nouveau message de  '%s': %s\n\n\n", $conn->resourceId, $msg);
        foreach ($this->clients as $client) { // BROADCAST
            $message = json_decode($msg, true);
            //if ($conn !== $client) {   // Permet d'envoyer a tous les client sauf l'émetteur
                $client->send($msg);
            //}
        }
    }

    public function onClose(ConnectionInterface $conn) {
        $this->clients->detach($conn);
        echo "Connection a {$conn->resourceId} perdue.\n";
    }

    public function onError(ConnectionInterface $conn, \Exception $e) {
        echo "Une erreur est survenue lors de la connection {$conn->resourceId}: {$e-
>getMessage()}\n\n\n";
        $conn->close();
    }
}

$server = IoServer::factory(
    new HttpServer(
        new WsServer(
            new ServerImpl()
```

```
        )
    ),
    APP_PORT
);
echo "Serveur crée sur le port : " . APP_PORT . "\n\n";
$server->run();
```

Nous avons donc une classe (que vous pouvez déclarer dans un fichier à part) qui prend en charge :

- Les nouvelles connexions (dans la méthode "onOpen")
- Les messages envoyés par les clients (dans la méthode "onMessage")
- Les connexions fermées (dans la méthode "onClose")
- Les erreurs (dans la méthode "onError")

La construction est finalement assez similaire à une appli en NodeJS, puisque quand un nouveau client arrive, on le "stocke" dans notre serveur.

Cela nous permet de pouvoir envoyer des messages en broadcast (à tout le monde) quand c'est nécessaire. Sauf à la personne qui a envoyé le message en question, à moins de vouloir afficher le fil de discussion dans ce cas-là, il faut modifier le broadcast afin qu'il l'envoie à tous les clients actuellement connectés sur le serveur de Web-sockets.

Coté client en JQuery

On doit donc ajouter la partie client ainsi que l'interface pour pouvoir discuter.

La page HTML suivante comprend donc un champ texte afin d'y rentrer son pseudo lors de la première saisie et le contenu des messages par la suite. Mais aussi un bouton qui nous permet de gérer l'envoi des messages en surveillant l'événement de « click » sur ce dernier.

La ligne http-equiv permet de contourner la règle de même origine, une sécurité des navigateurs, qui empêche l'utilisation de différents domaines (dans notre cas le port 80 pour notre application et 8080 pour notre serveur de Web-sockets)

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Security-Policy" content="default-src 'self' data: ga
      p: ws: ssl.gstatic.com 'unsafe-inline';">
    <meta charset="UTF-8">
    <title>CHAT</title>
```

```
</head>
<body>
    <center>
        <div id="erreur"></div>
        <input type="text" id="message" placeholder="Saisir votre pseudo">
        <input type="submit" value="Valider" id="valider">
        <div id="content"></div>
    </center>

    <script src="../component/jquery-3.2.0/jquery-3.2.0.min.js"></script>
    <script>
        $(document).ready(function()
        {
            var conn;
            var pseudo;
            $("#valider").click(function()
            {
                if (conn == undefined)
                {
                    pseudo = $("#message").val();
                    if (pseudo.length < 2)
                    {
                        $("#erreur").html("Pseudo inférieur a 2 caractères veuillez réessay
                    er");
                    }
                    else
                    {
                        $("#erreur").html(""); //.remove
                        $("#message").attr('placeholder', 'Votre message').val("");
                        $("#valider").val('Envoyer');

                        conn = new WebSocket('ws://localhost:8080');

                        conn.onopen = function(e)
                        {
                            $("#content").html("Vous êtes bien connecté " + pseudo + " !");
                        };
                        $('title').append(" " + pseudo);

                        conn.onmessage = function(e)
                        {
                            var msg = JSON.parse(e.data);
                            $("#content").append("<br>" + msg.chat_user + " : " + ms
                    g.chat_message);
                        };
```

```
                }
            }
            else
            {
                var messageJSON = {
                    chat_user: pseudo,
                    chat_message: $("#message").val()
                };
                conn.send(JSON.stringify(messageJSON));
                $("#message").val("");
            }
        });
    });
    </script>
  </body>
</html>
```

La partie script de la page va :

- Créer et ouvrir le canal d'échange en Web-sockets seulement si le pseudo de l'utilisateur est validé. Sinon un message d'erreur s'affiche.

- Déclarer une fonction qui sera appelée quand la connexion sera établie. Cette fonction va afficher « Vous êtes bien connecté pseudochoisi ! » une fois la connexion établie avec le serveur.

- Déclarer une fonction qui sera appelée quand un message proviendra du serveur et qui l'affichera ainsi que le pseudo de l'émetteur.

- Une gestion de l'envoi du message via un JSON afin de pouvoir récupérer le pseudo et le message sur tous les clients.

Docker et démarrage du serveur

Avant de pouvoir lancer notre serveur il nous faut paramétrer le conteneur Web qui s'occupera de faire tourner la partie serveur de notre Chat.

Pour cela il suffit de modifier un docker-compose.yml habituel et de préciser dans la partie du conteneur Web une seconde ouverture de ports correspondant au port précisé dans le script PHP du serveur de Web-socket.

```
ports:
```

```
        - "85:80"           // Pour apache
        - "8080:8080"       // Pour notre serveur Web-socket
```

Pour le lancement de notre serveur pour le moment la seule solution que j'ai pu trouver est de se connecter au conteneur Web via la commande (bien entendu il faut avoir lancé ses conteneurs avant) :

```
docker exec -ti --user $UID lenomdevotreconteneurweb bash
```

Et ensuite de lancer notre script serveur :

```
php Chat.php
```

Si le serveur se lance il va prendre « contrôle » de votre terminal et vous afficher :

```
Serveur crée sur le port : 8080
```

Il reste juste à vous rendre avec votre navigateur sur la page HTML d'interface client et à tester le Chat.

Si vous maîtrisez les notions de PHP, Composer, Apache et du JS/JQuery abordées dans cet exemple, il faut peu de temps pour mettre ce système en place.

À vous les serveurs de notifications fait maison, les chats réactifs, les éditeurs de contenus synchronisés et toutes sortes d'échanges sans avoir besoin de se soucier des en-têtes HTTP !

Sources :

https://www.xul.fr/html5/websocket.php

https://www.twilio.com/blog/create-php-websocket-server-build-real-time-even-driven-application

https://stackoverflow.com/questions/14512182/how-to-create-websockets-server-in-php

https://c-mh.fr/posts/websockets-en-php-plus-simple-qu-il-n-y-parait

https://blog.octo.com/pourquoi-les-websockets/

http://socketo.me/docs/

https://afsy.fr/avent/2017/21-symfony-et-websockets

https://doc.ubuntu-fr.org/creer_un_service_avec_systemd