Introduction aux Websockets

Intervenant : Chouki TIBERMACINE

Courriel : Chouki.Tibermacine@lirmm.fr

Web : http://www.lirmm.fr/~tibermacin/ens/ws/

Plan du cours

- HTTP et ses limites en prog. d’appli. distribuées
- Le protocole Websockets
- API Websockets en JavaScript

Plan du cours

- HTTP et ses limites en prog. d’appli. distribuées
- Le protocole Websockets
- API Websockets en JavaScript

Le protocole HTTP

- HTTP est un protocole de la couche « Application », couche la plus haute des modèles OSI ou Internet (TCP/IP)

- Il s'agit d'un protocole client-serveur (requête/réponse) :
  - Un client demande l'accès à une ressource (ça peut être aussi l'exécution d'un programme) en indiquant entre autres un URL
  - Le serveur répond en renvoyant la ressource demandée (ou les résultats de l'exécution du programme)

- Parmi les clients les plus connus : les navigateurs Web et les robots d'indexation de pages Web

- Parmi les serveurs les plus connus : Apache (C), Tomcat (Java) IIS (C# & ASP.Net) et serveurs écrits avec Node.js (JavaScript)

Applications utilisant HTTP

- Toute application Web s'appuie sur HTTP :
  - Des programmes exécutés par le processus serveur (httpd sous Unix) produisent dans des réponses HTTP
    1. des interfaces utilisateurs (HTML, CSS, ...)
    2. + des programmes (JS) destinés aux « programmes » clients

  - Les programmes clients (processus du navigateur) :
    1. produisent le rendu graphique de l'interface utilisateur
    2. attendent les actions de l'utilisateur sur cette interface
    3. peuvent envoyer des requêtes HTTP et attendre les réponses

- L'interaction entre les programmes se fait toujours selon le modèle requête/réponse. **Elle est toujours initiée par le client**

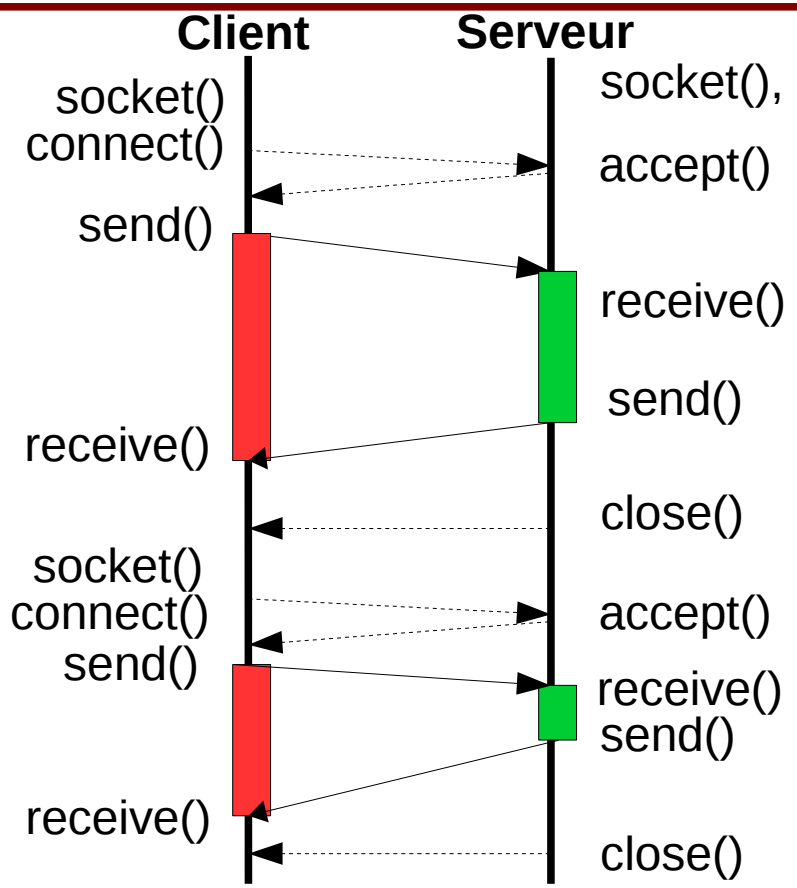
En dessous de HTTP

- HTTP peut théoriquement fonctionner sur n'importe quel protocole de (la couche) Transport

- En pratique, HTTP utilise TCP, de la façon suivante :
  - Le client établit une connexion TCP avec le serveur (TCP handshake)
  - Le client envoie une requête en utilisant cette connexion et attend la réponse
  - Le serveur se limite à répondre aux requêtes. Il n'envoie pas de façon proactive des messages (dans l'autre sens)

- HTTP utilise une seule connexion TCP par requête
  - On a une connexion fiable du client vers le serveur
  - Mais le serveur ne cherche pas à avoir une connexion fiable avec un client (le client re-demande la ressource si problème)

En dessous de HTTP -vue simplifiée-



Le serveur fait un send suite à un receive (send réactif)

Il ne fait jamais de send (proactif)

Applications nécessitant un modèle d'interaction différent

- Beaucoup d'applications distribuées nécessitent des interactions initiées par les deux parties (client et serveur) :
  - Applications multi-utilisateurs : messagerie instantanée, éditeurs collaboratifs avec édition « simultanée », visio-conf, jeux, ...
  - Applications dépendant de données à jour sur le serveur

- Le client envoie des requêtes au serveur, mais le serveur envoie aussi des ressources au client, sans que le client n'en fasse explicitement la demande (*server push*)

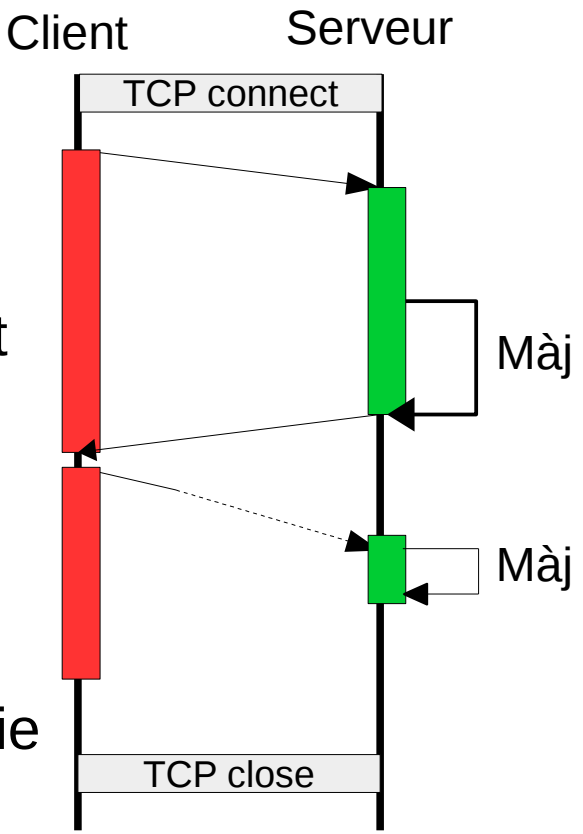
- Exemple : Messagerie instantanée (*chat*) sur le Web
  - Le serveur gère deux clients à un moment donné
  - Lorsque le serveur reçoit un message du client 1, comment fait-il pour transmettre ce message au client 2 ?

Techniques courantes pour du HTTP bidirectionnel

- Périodiquement, le client sollicite le serveur pour des mises à jour des ressources (*periodic polling*)
- Mais solution consommatrice en ressources (réseau et serveur, à cause des requêtes inutiles, si fréquence importante du polling)
- Les deux techniques les plus courantes :
  - HTTP long polling
  - HTTP streaming

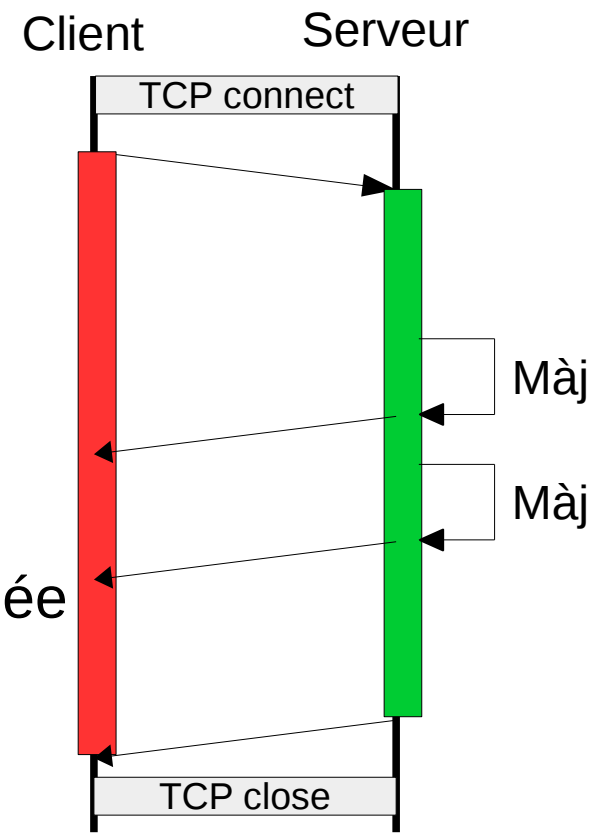
HTTP long polling

- Le client envoie une requête au serveur
- Le serveur ne répond pas tout de suite
- Il attend que les ressources demandées aient été mises à jour
- Dès qu'il y a une mise à jour (ou un timeout), le serveur répond au client
- Dès la réception de la réponse, le client envoie une autre requête, et ainsi de suite



HTTP streaming

- Le client envoie une requête au serveur
- Le serveur ne répond pas tout de suite
- Il attend que les ressources demandées aient été mises à jour
- Le serveur répond au client, mais indique que c'est une réponse partielle (le reste est à suivre). La connexion n'est pas terminée
- Le client attend la suite jusqu'à ce que le serveur décide de la fin (ressources envoyées entièrement)



Mise en œuvre de ces techniques

- Server-Sent Events : implémentation de HTTP streaming

  - **Côté client :** JS produit une seule requête GET XHR

```
var evtSource = new EventSource("script_test_sse.php");
evtSource.onmessage = function(e) { … e.data ... }
```

  - **Côté serveur :** on retourne des réponses de type *event-stream*

```
header("Content-Type: text/event-stream\n\n");
$messages = array("Hello World","Salut à tous","Hola Mundo");
while(true) {
  print("data: ".$messages[array_rand($messages)]."\n\n");
  ob_flush(); flush(); // Réponse partielle au client
  $r = rand(1,5); sleep($r); // Attendre qlq secondes et itérer
}
```

DEMO

Limites de ces techniques

- Ces techniques fonctionnent mal ou pas, en présence d'entités intermédiaires, ex. proxies, qui peuvent utiliser des caches (bufferiser réponses partielles du long polling, par exemple)
- Manque d'efficacité parce que : surcoût lié aux multiples messages échangés (messages avec headers HTTP), le serveur doit attendre, des ressources OS sont allouées pour des connexions TCP/ requêtes HTTP ouvertes dans le long polling, ...
- Support faible côté client et aucune obligation à les implémenter
- Détails dans la RFC 6202 de l'IETF : https://tools.ietf.org/html/rfc6202

Plan du cours

- HTTP et ses limites en prog. d’appli. distribuées
- Le protocole Websockets
- API Websockets en JavaScript

But du protocole

- Le but est de remplacer les techniques existantes qui exploitent l'infrastructure HTTP pour mettre en place des interactions bidirectionnelles entre clients et serveurs
- Le protocole est conçu pour exploiter toute l'infrastructure HTTP existante (ports, proxies, authentification, …)
- Mais dans le futur, il est probable qu'il soit implémenté sans utiliser HTTP

Aperçu du protocole

- Protocole à deux étapes :
  1. Handshake (poignée de main) **HTTP**
  Requête HTTP GET *upgrade* du protocole, puis réponse HTTP

CLIENT : URI de la Websocket : ws://server.example.com/chat

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

SERVEUR :

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

2.Transfert de données TCP : bidirectionnel (chaque côté peut envoyer des données de façon délibérée)
La fermeture de la connexion se fait par envoi de frames de contrôle

Philosophie du protocole

- Le protocole Websockets a été conçu pour être une couche « fine » au dessus de TCP

- Le protocole :
  - est indépendant de HTTP, sauf pour le handshake pour l'instant
  - exploite les capacités de HTTP, comme CORS (modèle de sécurité basé sur les origines Web utilisé par les navigateurs) et l'adressage Web (URL : port HTTP + plusieurs noms de hôtes sur une seule adresse IP)
    Par défaut : ports 80 (HTTP) et 443 (HTTPS)
  - simplifie la communication client/serveur et le format d'échange de messages (il est très proche de TCP)

Transfert de données

- Les données peuvent être transmises par le client ou par le serveur à n'importe quel moment, tant que la connexion est ouverte
- Le client/serveur peut faire plusieurs envois successifs
- Transfert sous la forme de frames (trames) de données textuelles (en UTF-8) ou binaires
- Une trame de contrôle transmise des deux côtés ferme la connexion Websocket (qui sera suivie de la fermeture de la connexion TCP sous-jacente)
- Au delà, aucun transfert de données n'est possible

États d'une connexion Websocket

- CONNECTING : Le client a envoyé la requête correspondant au handshake et attend la réponse du serveur
- OPEN : Le serveur a répondu avec succès à la requête du client La connexion est maintenant établie entre le client et le serveur
- CLOSING : L'une des extrémités (endpoints) a demandé la fermeture de la connexion et attend la confirmation de l'autre extrémité
- CLOSED : état final

Plan du cours

- HTTP et ses limites en prog. d’appli. distribuées
- Le protocole Websockets
- API Websockets en JavaScript

Clients de Websockets

- Théoriquement, un client peut être écrit dans n'importe quel langage
- Usuellement, on écrit des programmes JavaScript, embarqués dans des pages HTML, interprétés par le navigateur Web
- L'API côté client du protocole Websockets est maintenant standardisée
- Elle est implémentée par tous les navigateurs *mainstream* (Chrome, Firefox, Edge, IE, Safari, Opera, ...)

Écrire un client de Websockets en JS

- Il faudra en premier créer un objet de la « classe » WebSocket
- Le constructeur de cette classe établira la connexion (handshake) avec le serveur
- Le constructeur prend en paramètre l'URL du serveur de Websockets

```
var maSocket = new WebSocket("ws://server.example.com/chat");
```

- Il est possible de passer plus d'arguments au constructeur (valeurs des headers HTTP du handshake)
- maSocket.readyState contient CONNECTING
  → OPEN (dès la réception de la réponse du serveur)

Envoyer des données au serveur

- L'objet Websocket possède une méthode send, qui sert à envoyer des données au serveur :

```
maSocket.send("Hello Server");
```

- Utiliser send de façon asychrone (demander son exécution lorsque l'état de la connexion devient OPEN) :

```
maSocket.onopen = function(e) {
    maSocket.send("Hello Server");
} ;
```

- Les données envoyées peuvent être du :
  - Text : objets String
  - Binaire : des objets Blob (données de grande taille) ou ArrayBuffer

Envoyer des données JSON au serveur

- Il est recommandé d'envoyer autant que possible des données au format JSON au serveur, si données semi-structurées :

```
// Une application de chat :
var msg = {
    type: "message",
    text: document.getElementById("moi").value,
    id: clientID,
    date: Date.now()
};

maSocket.send(JSON.stringify(msg));
```

- La donnée est envoyée comme String au format JSON

Recevoir des données du serveur et fermer la connexion

- Il suffit de définir un écouteur d'événement onmessage

```
// Une application de chat :
maSocket.onmessage = function (event) {
  document.getElementById("autre").value = event.data;
} ;
```

event (paramètre de la callback) est un objet MessageEvent

- Les données sont reçues dans la propriété data de l'objet MessageEvent

- Recevoir des données JSON :

```
var msg = JSON.parse(event.data);
var time = new Date(msg.date);
```

- Pour fermer la connexion, il suffit d'écrire : maSocket.close();

Traiter les erreurs

- Si un problème de connexion survient, des événements sont produits par le navigateur Web

- D'abord un événement error est émis, qu'on peut écouter en affectant une callback à la propriété onerror de l'objet Websocket

```
maSocket.onerror = function(e) {
    // ...
} ;
```

- Ensuite, un événement close (onclose) est émis

APIs Websockets côté serveur

- Tout dépend de l'implémentation du protocole côté serveur
- Node.js est un interprète de JavaScript indépendant de tout navigateur Web. Il permet d'écrire des programmes JS serveurs
- Son but : fournir un modèle de programmation événementielle à des programmes serveurs afin d'améliorer leurs performances
- Il fournit plusieurs modules. Exemple, 'http' permet de démarrer un serveur Web, sans passer par Apache
- ws, websockets et socket.io sont des exemples de modules (non fournis par défaut). Ils permettent d'écrire des serveurs WS

Programme serveur HTTP avec Node.js

- Créer et démarrer un serveur Web : -fichier **server.js**-

```
// Création d'un serveur avec une fonction callback qui attend
// des requêtes HTTP et qui renvoie une réponse HTTP
var serveur = require('http');// Charger le module http de Node.js
serveur.createServer(function (req, rep) { // Callback
    rep.writeHead(200,{"Content-Type":"text/html; charset=utf-8"});
    rep.write("Bonjour à tous");
    rep.end();
}).listen(8080);
```

- Sur un invité de commandes : (commande node ou nodejs)

```
$ nodejs server.js
```

- Sur le navigateur : localhost:8080 → page avec Bonjour à tous

DEMO

Programme Websocket serveur avec le module ws

- Utiliser le module ws (http://websockets.github.io/ws/) :

```
var WebSocketServer = require("ws").Server;// Charger le module ws
var serveur = new WebSocketServer( { port: 8100 } );
console.log("Serveur démarré...");
var clients = [];
serveur.on("connection", function (ws) {
    clients.push(ws);
    clients[clients.length-1].on("message", function (str) {
        console.log("Client connecté ...");
        console.log(str);
        ws.send("Hello client");
    });
    clients[clients.length-1].on("close", function() {
        console.log("Client déconnecté.") ;
    });
});
```

DEMO

- Module simple et pratique (il en existe d'autres)

Sous-protocoles de Websockets

- Il est possible de définir des sous-protocoles côté serveur et ensuite les utiliser côté client
- Ces sous-protocoles peuvent par exemple imposer un certain format de données à échanger entre les 2 parties
- Exemples de sous-protocoles : soap (services Web XML), wamp (Web Application Messaging Protocol : RPC et publish/ subscribe), ...
- Sous-protocoles enregistrés : http://www.iana.org/assignments/websocket/websocket.xml

Sous-protocoles de Websockets

- Choisir un sous-protocole côté client : (requête handshake)
  GET /chat HTTP/1.1
  ...
  Sec-WebSocket-Protocol: soap, wamp

- Si l'un des sous-protocoles est supporté par le serveur, il renvoie dans la réponse le même entête avec le sous-protocole choisi
  Sec-WebSocket-Protocol: soap

- Si plusieurs sous-protocoles sont supportés par le serveur, il en choisira un, selon l'ordre indiqué par le client (soap, puis wamp)

- Si aucun des sous-protocoles n'est supporté par le serveur, la réponse du serveur ne comportera pas cet entête

Quelques références

- RFC du protocole Websocket de l'IETF : https://tools.ietf.org/html/rfc6455
- Spécification de l'API Websockets du W3C : https://www.w3.org/TR/websockets/
- Tutorial sur les Websockets du MDN : https://developer.mozilla.org/fr/docs/WebSockets

Questions