

---

# (73) PJI Architecture TOR sur navigateur web

---

Franquenouille Kevin  
Cornette Damien

Git Repository : <https://github.com/kfranquenouille/PJI-TOR.git>

Suivi par : Julien Iguchi-Cartigny

Année universitaire : 2014-2015

# Introduction

Tor est un projet pour le support de l'anonymat en dissimulant les communications entre un client web et un serveur. Le mécanisme s'appuie sur un ensemble de relais de confiance disposés sur des serveurs.

Le nombre limité de relais (moins de 5000) et la confiance dans leur non-compromission peut être considéré comme une limitation de l'approche de Tor. Un moyen d'outrepasser cette limitation serait de limiter le rôle du serveur à un simple relais et de laisser les clients gérer l'anonymat dans le système.

Afin de rendre le système le plus pratique et adoptable pour multiplier les chemins possibles, une idée serait de ne s'appuyer que sur des technologies webs (http, javascript) pour que chaque navigateur puisse devenir un client.

Le but de ce projet est d'étudier la faisabilité de l'approche à l'aide de technologie comme websocket. Le cas d'étude sera limité à un simple chat (saisie limitée à 140 caractères).

# Table des matières

1	Description générale du projet . . . . .	3
1.1	Problématique . . . . .	3
1.2	Web Cryptography API . . . . .	3
1.3	WebSocket API . . . . .	7
1.4	Architecture de l'application . . . . .	7
2	Mode d'emploi . . . . .	8
2.1	Installer un relais . . . . .	8
2.2	Configurer un relais . . . . .	8
2.3	Configurer un client . . . . .	8
2.4	Lancer l'application . . . . .	8
3	Tests . . . . .	9
3.1	Docker . . . . .	9
3.2	Cryptographie . . . . .	9
3.3	WebSocket . . . . .	9

# 1 Description générale du projet

## 1.1 Problématique

Le but de ce projet était de mettre en place une "simulation" du réseau TOR via un navigateur web. Pour cela, nous avons pu voir avec Julien Iguchi-Cartigni les technologies que l'on pouvait utiliser. De ce fait, on a pu découper le projet en 2 parties :

- Web Cryptography API
- WebSocket API

Pour réaliser cela, nous devions réaliser cela avec un seul relais client pour debuter puis ensuite le faire avec 3 afin de bien vérifier l'encryption des messages et que les messages sont bien redirigés vers le prochain relais.

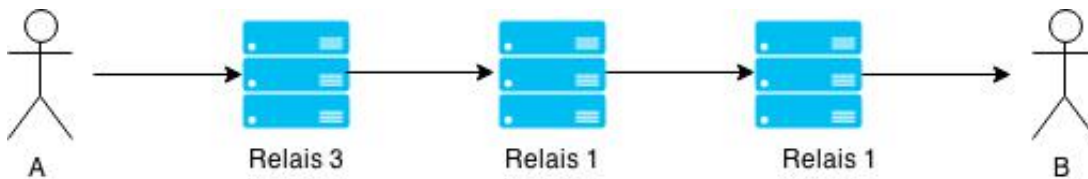


FIGURE 1 – Transfert d'un message de A vers B en passant par 3 relais

## 1.2 Web Cryptography API

Afin de bien crypter les messages envoyés et d'utiliser une technologies assez fiable et récente, nous avons privilégié cette API. En effet, elle est compatible sur presque tous les navigateurs (inclus de base). En revanche, il nous a fallu trouver un bon algorithme de chiffrement.

Dans un premier temps, nous avons utilisé le chiffrement AES-CBC afin de tester et découvrir Webcrypto API et réaliser un premier jet de ce que nous voulions faire. Une fois le messages encrypté 4 fois. Voici un exemple du cryptage :

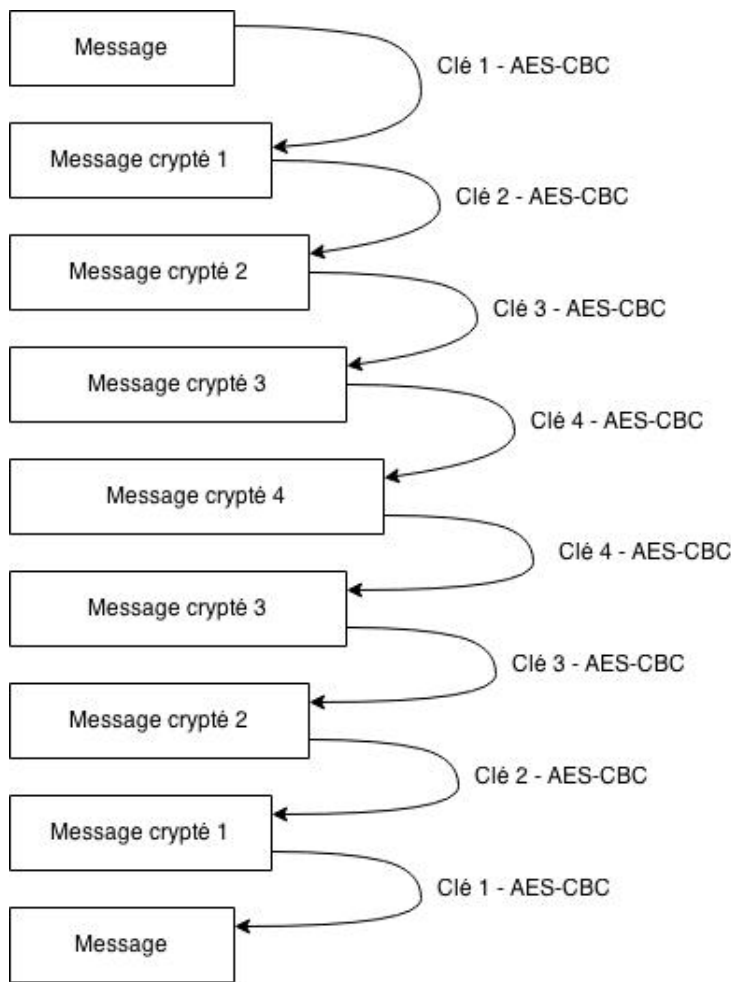


FIGURE 2 – Cryptage d'un message 4 fois de suite en AES-CBC

Pour la solution optimale, il est plus préférable d'utiliser un système de clés publiques et de clés privées. Avec l'algorithme RSA-OAEP, cela est possible. Chaque clé générée (l'objet `KeyPromise`) contient une clé publique et une clé privée. De ce fait, cette solution est plus attendu car elle ressemble très fortement à l'architecture du réseau TOR. Bien entendu, le client doit connaître le chemin afin qu'il puisse créer son message crypté avec les clés publiques des relais correspondants.

Ce qu'il faut comprendre en utilisant WebCrypto API, c'est le fait que pour générer une clé, encrypter ou décrypter, nous jouons avec des *Promises*. En Javascript, des *Promises* ou "promesses" en français, sont utilisées pour réaliser des actions asynchrones. En effet, elles représentent des intermédiaires vers des valeurs qui ne sont nécessairement connues lors de leurs créations. Cela permet de gérer des actions asynchrones et de leur associer des gestionnaires d'erreurs.

La principale difficulté qui réside dans le fait d'utiliser des *Promises* est de bien gérer les erreurs mais savoir exactement où le code à échouer. Le fait d'encrypter un message avec une clé publique puis d'encrypter le résultat avec une autre clé publique nécessite obligatoirement une imbrication du code. Voici un exemple :

```

1 window.crypto.subtle.generateKey({name: "aes-cbc", length: 128}, true, ["
  encrypt", "decrypt"]).then(function (key) {
2   var data = asciiToUint8Array("ceci est un test");

```

```

3 console.log("Data:");
4 console.log("value: ceci est un test");
5 console.log(data);
6 var algorithm = key.algorithm;
7 algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));
8
9 window.crypto.subtle.encrypt(algorithm, key, data).then(function (ct) {
10     console.log("AES-CBC encrypt 1:");
11     console.log(new Uint8Array(ct));
12
13     window.crypto.subtle.decrypt(algorithm, key, ct).then(function (pt) {
14         console.log("AES-CBC decrypt 1:");
15         console.log(new Uint8Array(pt));
16         console.log(uint8ArrayToAscii(new Uint8Array(pt)));
17     }, handle_error);
18 }, handle_error);
19 }, handle_error);

```

Une autre difficulté réside dans le fait de trouver un bon algorithme qui respecte ce que l'on souhaite faire. Par exemple, pour découvrir WebCrypto API, nous avons testé avec la cryptographie AES-CBC comme dans l'exemple précédent. Dans ce cas, on constate que l'on a bien une clé qui est générée. En revanche, elle ne respecte pas ce que l'architecture TOR exige. La clé n'est pas publique ni privée mais permet juste d'encrypter un message et de le décrypter. Nous avons regardé les différents cryptages disponibles et compatibles et nous avons trouvé RSA-OAEP. Ce dernier gère les clés publiques et privées et permet d'encrypter et de décrypter. D'autres cryptographies RSA possèdent juste les fonctions de signature et de vérification de la signature. Egalement, la cryptographie RSAES-PKCS1-v1\_5 aurait pu être exploitée mais cette dernière n'est plus disponible dans les navigateurs. C'est donc pour cela que nous avons choisi RSA-OAEP.

Voici un exemple basique avec RSA-OAEP :

```

1 window.crypto.subtle.generateKey({name: "RSA-OAEP", modulusLength: 2048,
    publicExponent: new Uint8Array([0x01, 0x00, 0x01]), hash: {name: "SHA-256"}}, true, ["encrypt", "decrypt"]).then(function (key) {
2     var data = asciiToUint8Array("ceci est un test");
3     console.log("Data:");
4     console.log(uint8ArrayToAscii(data));
5     console.log("Public Key");
6     console.log(key.publicKey);
7     console.log("Private Key");
8     console.log(key.privateKey);
9
10    window.crypto.subtle.encrypt({name: "RSA-OAEP"}, key.publicKey, data).then
        (function (encrypted) {
11        console.log(uint8ArrayToAscii(new Uint8Array(encrypted)));
12
13        window.crypto.subtle.decrypt({name: "RSA-OAEP"}, key.privateKey,
            encrypted).then(function (decrypted) {
14            console.log(uint8ArrayToAscii(new Uint8Array(decrypted)));
15        }, handle_error);
16    }, handle_error);
17 }, handle_error);

```

Egalement, nous avons pu regarder quels sont les algorithmes supportés par la plupart des navigateurs. Nous en avons donc dressé le tableau suivant :

Algorithme	Supporté
RSASSA-PKCS1-v1_5	Oui
RSASSA-PKCS1-v1_5	Non
RSA-PSS	Oui
RSA-OAEP	Oui
AES-CTR	Oui
AES-CBC	Oui
AES-CMAC	Non
AES-GCM	Oui
AES-CFB	Oui
AES-KW	Oui
SHA-1	Oui
SHA-256	Oui
SHA-384	Oui
SHA-512	Oui

Les algorithmes SHA sont des *digest*. Cela sert principalement à créer des table de hachage crypter un message ou une clé par exemple. Dans le code précédent, on peut remarquer que la fonction de hachage mise en place est bien le SHA-256.

Une fois que nous avons trouvé l'algorithme et trouver son fonctionnement, nous avons du mettre en place une maquette. Voici donc une image représentative de ce que l'on a testé :

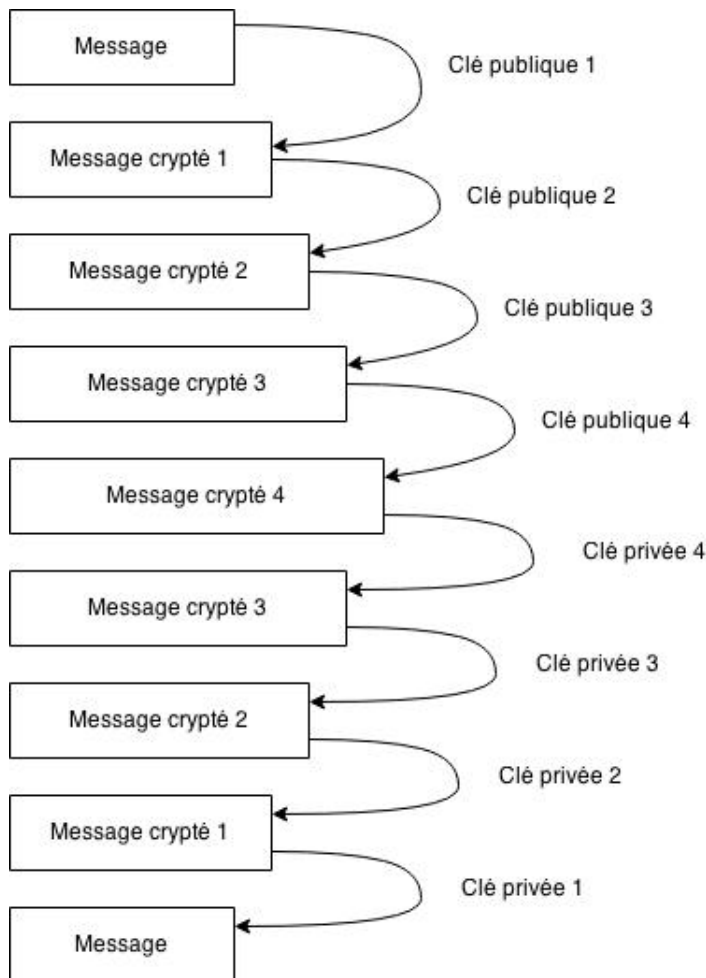


FIGURE 3 – Cryptage d'un message 4 fois de suite en RSA-OAEP

Dans cet exemple, il faut crypter obligatoirement avec la clé publique et décrypter avec la clé privée correspondante. En effet, les clés publiques sont accessibles à tout le monde alors que les clés privées sont uniquement restreintes au point relais correspondant.

### **1.3 WebSocket API**

### **1.4 Architecture de l'application**



## 2 Mode d'emploi

### 2.1 Installer un relais

### 2.2 Configurer un relais

### 2.3 Configurer un client

### 2.4 Lancer l'application

## 3 Tests

### 3.1 Docker

### 3.2 Cryptographie

### 3.3 WebSocket

## Conclusion