# (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 a 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 . . . . . 12

# 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 découpé le projet en 2 parties :

- Web Cryptography API
- WebSocket API

Pour réaliser cela, nous devions réaliser cela avec un seul relais client pour débuter puis ensuite le faire avec 3 afin de bien vérifier l'encryptage des messages et que les messages sont bien redirigés vers le prochain relais. De plus, il fallait gérer le système de clés publiques et privées afin que le message ne puisse pas être décodé par n'importe quel utilisateur lambda.

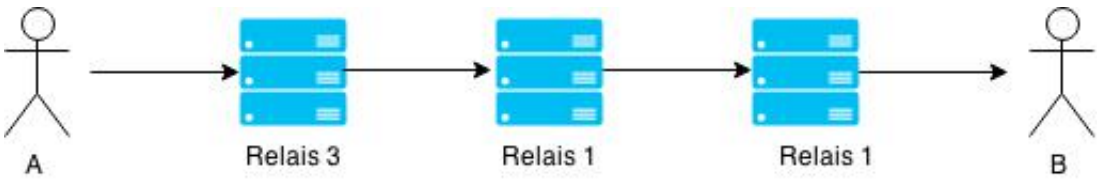


FIGURE 1 – Transfert d'un message de A vers B en passant par 3 relais

## 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 :

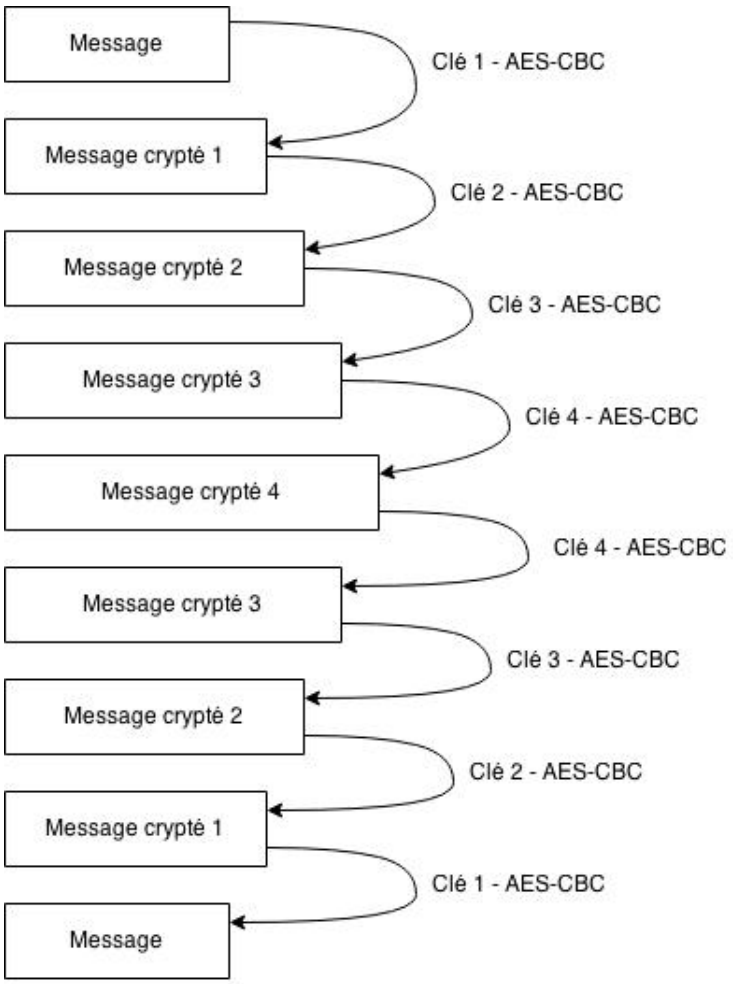


FIGURE 2 – Cryptage d'un message 4 fois de suite en AES-CBC

Pour la solution optimale, il est plus préférable d'utiliser un système de clés publiques et de clés privées. Avec l'algorithme RSA-OAEP, celà 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");
```

```
  console.log("Data:");
  console.log("value: ceci est un test")
  console.log(data);
  var algorithm = key.algorithm;
  algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));

  window.crypto.subtle.encrypt(algorithm, key, data).then(function (ct) {
    console.log("AES-CBC encrypt 1:");
    console.log(new Uint8Array(ct));

    window.crypto.subtle.decrypt(algorithm, key, ct).then(function (pt) {
      console.log("AES-CBC decrypt 1:");
      console.log(new Uint8Array(pt));
      console.log(uint8ArrayToAscii(new Uint8Array(pt)));
    }, handle_error);
  }, handle_error);
}, 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 avions 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 par 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 :

```
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) {
  var data = asciiToUint8Array("ceci est un test");
  console.log("Data:");
  console.log(uint8ArrayToAscii(data));
  console.log("Public Key");
  console.log(key.publicKey);
  console.log("Private Key");
  console.log(key.privateKey);

  window.crypto.subtle.encrypt({name: "RSA-OAEP"}, key.publicKey, data).then
      (function (encrypted) {
    console.log(uint8ArrayToAscii(new Uint8Array(encrypted)));

    window.crypto.subtle.decrypt({name: "RSA-OAEP"}, key.privateKey,
        encrypted).then(function (decrypted) {
      console.log(uint8ArrayToAscii(new Uint8Array(decrypted)));
    }, handle_error);
  }, handle_error);
}, 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 |
| RSAES-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 avions 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 :

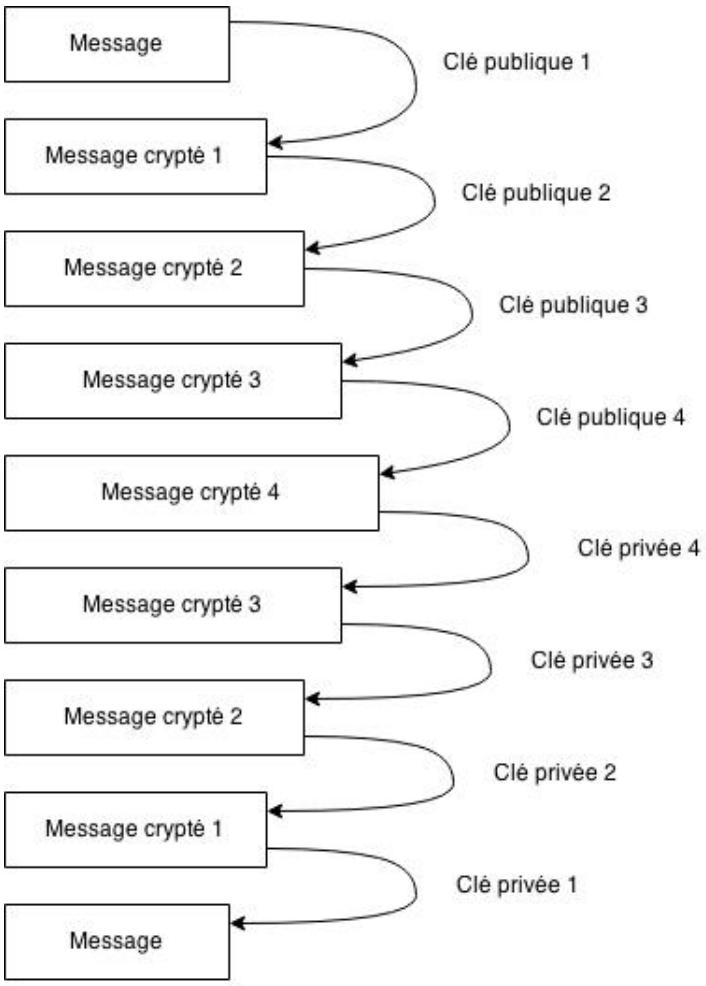


FIGURE 3 – Cryptage d'un message 4 fois de suite en RSA-OAEP

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

Pour la cryptographie avec WebCrypto API, il fallait tester des cas de base. Tout d'abord pour découvrir le fonctionnement, nous avons pris l'algorithme AES-CBC. Puis nous avons généré une clé et encrypté un message de test. Une fois ce message encrypté, nous avons affiché le résultat sous la forme d'une chaîne de caractères. Ensuite, pour vérifier que l'on puisse bien décrypter cela, nous l'avons décrypter avec la clé et vérifié que le message décrypté correspondait bien à celui de départ.
Voici le code qui a permis de réaliser cela :

```
window.crypto.subtle.generateKey({name: "aes-cbc", length: 128}, true, ["encrypt", "decrypt"]).then(function (key) {
  var data = asciiToUint8Array("ceci est un test");
  console.log("Data:");
  console.log("value: ceci est un test")
  console.log(data);
  var algorithm = key.algorithm;
  algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));

  window.crypto.subtle.encrypt(algorithm, key, data).then(function (ct) {
    console.log("AES-CBC encrypt 1:");
    console.log(new Uint8Array(ct));

    window.crypto.subtle.decrypt(algorithm, key, ct).then(function (pt) {
      console.log("AES-CBC decrypt 1:");
      console.log(new Uint8Array(pt));
      console.log(uint8ArrayToAscii(new Uint8Array(pt)));
    }, handle_error);
  }, handle_error);
}, handle_error);
```

Une fois ceci réalisé, nous nous sommes posés la question de ce qu'il en est si nous cryptons 4 fois de suite un message. Pour cela, il nous a fallu adapter notre code de manière basique et de tester ce que cela donnait.
Voici l'enchaînement du code :

```
function startCrypto(input_content){
  window.crypto.subtle.generateKey({name: "aes-cbc", length: 128}, true, ["encrypt", "decrypt"]).then(function (key1) {
    var data = asciiToUint8Array(input_content);
    var algorithm = key1.algorithm;
    algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));

    window.crypto.subtle.generateKey({name: "aes-cbc", length: 128}, true, ["encrypt", "decrypt"]).then(function (key2) {
      var algorithm = key2.algorithm;
      algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));

      window.crypto.subtle.generateKey({name: "aes-cbc", length: 128}, true, ["encrypt", "decrypt"]).then(function (key3) {
        var algorithm = key3.algorithm;
        algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));

        window.crypto.subtle.generateKey({name: "aes-cbc", length: 128}, true, ["encrypt", "decrypt"]).then(function (key4) {
          var algorithm = key4.algorithm;
```

```
17          algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));
18
19          window.crypto.subtle.encrypt(algorithm, key1, data).then(function
                (ct1) {
20            console.log("AES-CBC encrypt 1:");
21            console.log(uint8ArrayToAscii(new Uint8Array(ct1)));
22
23            window.crypto.subtle.encrypt(algorithm, key2, new Uint8Array(ct1
                  )).then(function (ct2) {
24              console.log("AES-CBC encrypt 2:");
25              console.log(uint8ArrayToAscii(new Uint8Array(ct2)));
26
27              window.crypto.subtle.encrypt(algorithm, key3, new Uint8Array(
                    ct2)).then(function (ct3) {
28                console.log("AES-CBC encrypt 3:");
29                console.log(uint8ArrayToAscii(new Uint8Array(ct3)));
30
31                window.crypto.subtle.encrypt(algorithm, key4, new Uint8Array
                      (ct3)).then(function (ct4) {
32                  console.log("AES-CBC encrypt 4:");
33                  console.log(uint8ArrayToAscii(new Uint8Array(ct4)));
34
35                  window.crypto.subtle.decrypt(algorithm, key4, ct4).then(
                        function (pt4) {
36                    console.log("AES-CBC decrypt 4:");
37                    console.log(uint8ArrayToAscii(new Uint8Array(pt4)));
38
39                    window.crypto.subtle.decrypt(algorithm, key3, pt4).then(
                          function (pt3) {
40                      console.log("AES-CBC decrypt 3:");
41                      console.log(uint8ArrayToAscii(new Uint8Array(pt3)));
42
43                      window.crypto.subtle.decrypt(algorithm, key2, pt3).
                            then(function (pt2) {
44                        console.log("AES-CBC decrypt 2:");
45                        console.log(uint8ArrayToAscii(new Uint8Array(pt2)));
46
47                        window.crypto.subtle.decrypt(algorithm, key1, pt2).
                              then(function (pt) {
48                          console.log("AES-CBC decrypt 1:");
49                          console.log(uint8ArrayToAscii(new Uint8Array(pt)))
                                ;
50                        }, handle_error);
51                      }, handle_error);
52                    }, handle_error);
53                  }, handle_error);
54                }, handle_error);
55              }, handle_error);
56            }, handle_error);
57          }, handle_error);
58        }, handle_error);
59      }, handle_error);
60    }, handle_error);
61  }, handle_error);
62  }
```

Comme on peut le constater le code devient de plus en plus en illisible si on l'imbrique. Cependant, étant juste le test de découverte, cela avait peut d'importance mais il s'agissait juste de la manière de procéder qui nous intéressait. De ce fait, nous avons appliqué ceci avec la bon cryptage qui est RSA-OAEP. pour le cas d'un seul encryptage, il s'agit a peu de chose

près le même cas que AES-CBC. En revanche, pour un encryptage multiple, cela dépend de la taille des données. En effet, nous avons voulu faire varier le *modulusLength*. Cela nous a permis d'avoir un message crypté plus ou moins grand.
L'exemple suivant nous a permis de réaliser cela :

```
function generateKeys(){
  window.crypto.subtle.generateKey({name: "RSA-OAEP", modulusLength: 512,
     publicExponent: new Uint8Array([0x01, 0x00, 0x01]), hash: {name: "SHA-1
     "}}, true, ["encrypt", "decrypt"]).then(function (key) {
    keyGenerate1 = key;
    console.log(keyGenerate1);
    window.crypto.subtle.generateKey({name: "RSA-OAEP", modulusLength: 1024,
       publicExponent: new Uint8Array([0x01, 0x00, 0x01]), hash: {name: "
      SHA-1"}}, true, ["encrypt", "decrypt"]).then(function (key2) {
      keyGenerate2 = key2;
      console.log(key2);
      window.crypto.subtle.generateKey({name: "RSA-OAEP", modulusLength:
         2048, publicExponent: new Uint8Array([0x01, 0x00, 0x01]), hash: {
         name: "SHA-1"}}, true, ["encrypt", "decrypt"]).then(function (key3)
          {
        keyGenerate3 = key3;
        console.log(key3);
        window.crypto.subtle.generateKey({name: "RSA-OAEP", modulusLength:
           4096, publicExponent: new Uint8Array([0x01, 0x00, 0x01]), hash: {
           name: "SHA-1"}}, true, ["encrypt", "decrypt"]).then(function (
           key4) {
          keyGenerate4 = key4;
          console.log(key4);
          alert("Cles generees");
        }, handle_error);
      }, handle_error);
    }, handle_error);
  }, handle_error);
}
```

Une fois les clés générées, il ne nous restait plus qu'à les stocker afin de pouvoir les réutiliser par la suite. De plus, nous avons factoriser le code de l'encryptage et du décryptage afin d'éviter l'imbrication vu comme précédemment.

```
function encryptData(key, data, jquerySelector){
  var temp;
  compteur += 1;
  window.crypto.subtle.encrypt({name: "RSA-OAEP"}, key.publicKey, data).then
     (function (ct1) {
    console.log("RSA-OAEP encrypt "+compteur+" :");
    temp = new Uint8Array(ct1);
    encryptedMsg = new Uint8Array(ct1);
    $( jquerySelector ).text($( jquerySelector ).text()+uint8ArrayToAscii(
       temp));
    console.log(uint8ArrayToAscii(temp));
  }, handle_error);
  return temp;
}

function decryptData(key, data, jquerySelector){
  window.crypto.subtle.decrypt({name: "RSA-OAEP"}, key.privateKey, data).
     then(function (pt) {
    console.log("RSA-OAEP decrypt "+compteur+" :");
    encryptedMsg = new Uint8Array(pt);
    $( jquerySelector ).text($( jquerySelector ).text()+uint8ArrayToAscii(
       encryptedMsg));
```

```
19      console.log(uint8ArrayToAscii(encryptedMsg));
20      compteur -= 1;
21      return pt;
22    }, handle_error);
23  }
```

L'utilisation de *console.log()* nous a été très utile afin de debugger notre code mais aussi servir de trace pour voir l'utilisation de ce dernier. Une fois ceci fait, l'enchaînement des cryptages et décryptages se fait séquentiellement.
Un exemple est disponible dans la partie ./Webcrypto part/ en affichant la page index.html.

## 3.3 WebSocket

# Conclusion