(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 web (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

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 :

- WebSocket
- Web Cryptography

Pour réaliser cela, nous devions réaliser cela avec un seul serveur relais 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.

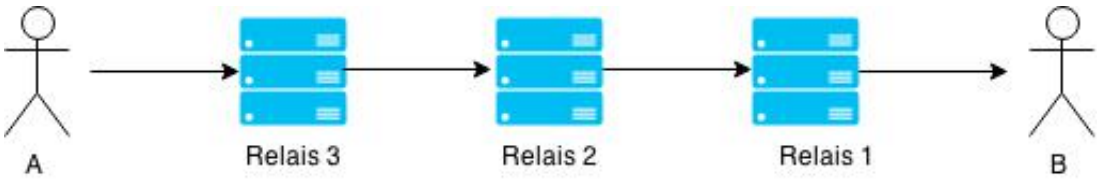

FIGURE 1 – Transfert d'un message de A vers B en passant par 3 relais

1.2 WebSocket

La première partie de notre PJI fut l'implémentation d'un réseau TOR à base de WebSocket entre des clients sous forme de navigateur web et un certains nombre de relais placé sur des serveurs web.

Qu'est-ce que les WebSocket ?

WebSocket est une technologie très récente encore en cours de standardisation par le W3C et n'est disponible que sur les navigateurs les plus récents. Selon la définition officielle, je cite :

> WebSocket est une technologie évoluée qui permet d'ouvrir un canal de communication interactif entre un navigateur (côté client) et un serveur.
> Avec cette API vous pouvez envoyer des messages à un serveur et recevoir ses réponses de manière événementielle sans avoir à aller consulter le serveur pour obtenir une réponse.

Cette technologie correspond donc parfaitement à notre problématique.

A partir de la, il nous a fallu choisir quelle librairie de websocket utiliser car il en existe une infinité qui fleurisse chaque jour sur le net. Mais parmi toutes ces librairies, deux ont retenu notre attention tout particulièrement :

- WebSocket API
- Socket.IO

1.2.1 WebSocket API

WebSocket API est tout simplement la librairie officielle supportée et en cours de standardisation par le W3C. Il nous était donc tout naturel de choisir cette librairie, ce que nous avons fait dans un premier temps.

Implémentation côté client

La première instruction est l'instanciation d'un objet WebSocket afin d'ouvrir une connexion avec le serveur.

```
1 var websocket = new WebSocket("ws://127.0.0.1:9000");
```

Remarquez le protocole utilisé qui n'est pas *http* mais *ws*. On ne remarque pas vraiment la différence entre ces deux protocoles côté client mais c'est côté serveur que tout va changer. En effet, voila à quoi ressemble l'entête d'un paquet envoyé du protocole *ws* :

```
1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
6 Sec-WebSocket-Protocol: chat, superchat
7 Sec-WebSocket-Version: 13
8 Origin: http://example.com
```

On appelle ce paquet en anglais un *WebSocket handshake request*, ce qui se traduit en français par une requête *poignée de main*, c'est à dire que le client tend la main au serveur (fais une demande de connexion) et attend un retour positif du serveur afin d'établir une connexion correcte.

Voici le paquet attendu par le client du serveur :

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
5 Sec-WebSocket-Protocol: chat
```

Une fois ce paquet du serveur réceptionné, le client est connecté de manière persistante avec le serveur.

La suite du code est l'implémentation d'une série d'événements.

Ouverture de connexion

```
1   websocket.onopen = function(ev) { // connection is open
2     $('#message_box').append("<div class=\"system_msg\">Connected!</div>");
          //notify user
3   }
```

Fermeture de connexion

```
websocket.onclose    = function(ev){
  $('#message_box').append("<div class=\"system_msg\">Connection Closed</
     div>");
};
```

Erreur de connexion

```
websocket.onerror = function(ev){
  $('#message_box').append("<div class=\"system_error\">Error Occurred - "
     +ev.data+"</div>");
};
```

Réception d'un message du serveur

```
websocket.onmessage = function(ev) {
  var msg = JSON.parse(ev.data); //PHP sends Json data
  var type = msg.type; //message type
  var umsg = msg.message; //message text
  var uname = msg.name; //user name
  var ucolor = msg.color; //color

  if(type == 'usermsg')
  {
    $('#message_box').append("<div><span class=\"user_name\" style=\"color
       :#"+ucolor+"\">"+uname+"</span> : <span class=\"user_message\">"+
       umsg+"</span></div>");
  }
  if(type == 'system')
  {
    $('#message_box').append("<div class=\"system_msg\">"+umsg+"</div>");
  }

  $('#message').val(''); //reset text
  $("#message_box").scrollTop($("#message_box")[0].scrollHeight);
};
```

Puis dernier bout de code, l'envoi d'un message par le biais d'un simple tchat

```
$('#send-btn').click(function(){ //use clicks message send button
  var mymessage = $('#message').val(); //get message text
  var myname = $('#name').val(); //get user name

  if(myname == ""){ //empty name?
    alert("Enter your Name please!");
    return;
  }
  if(mymessage == ""){ //emtpy message?
    alert("Enter Some message Please!");
    return;
  }

  //prepare json data
  var msg = {
    message: mymessage,
    name: myname,
    color : '<?php echo $colours[$user_colour]; ?>',
    nodes : ["192.168.33.10:9002", "192.168.33.10:9001"] // node path to
       the client
```

```
20      };
21      //convert and send data to server
22      websocket.send(JSON.stringify(msg));
23    });
```

Implémentation côté client

La librairie officielle des WebSocket n'est à ce jour implémentée que pour le côté client.

Nous avons choisi PHP pour développé le serveur mais PHP ne propose aucune librairie officielle pour communiquer correctement avec le protocole *ws*.

Nous avons donc pris l'initiative de développer nous même une implémentation du protocole *ws* en PHP.

Nous détaillerons pas ici le code de cette implémentation car il est long et complexe et il nous permet pas de nous concentrer sur le cœur de notre PJI.

Au final, nous avons pris la décision de passer sous NodeJS et d'utiliser la librairie *Socket.IO* qui est bien plus simple à utiliser et nous permet de nous concentrer sur le cœur du projet initial.

1.2.2 Socket.IO

Socket.IO est une librairie pour le serveur JavaScript NodeJS.
L'avantage de cette librairie est qu'en plus de sa simplicité déconcertante à utiliser, le code est le même que ce soit côté client ou côté serveur puisque le langage utilisé est le même (JavaScript).

Implémentation côté client

Ouverture de connexion avec le serveur

```
1  var socket = io.connect('http://127.0.0.1:9000');
```

Gestion des événements

```
1   socket.on('connect', function() {
2     ...
3   });
4
5   socket.on('disconnect', function() {
6     ...
7   });
8
9   socket.on('error', function() {
10    ...
11  });
12
13  socket.on('msg', function(msg) {
14    ...
15  });
```

Envoi d'un message

```
  $('#send-btn').click(function() { //use clicks message send button
    ...
        //prepare json data
      var msg = {
        message: encryptedMsg,
        name: myname,
        color : user_colour,
        nodes : nodes_list // node path to the client
      };

      //send data to server
      socket.emit('msg', msg);
    ...
  });
```

Implémentation côté serveur

Création d'un serveur web

```
var http = require('http');

httpServer = http.createServer(function(req, res) {
  res.end('Hello World');
});

httpServer.listen(port);
```

Création d'une socket serveur écoutant sur le serveur web

```
var io = require('socket.io').listen(httpServer);
```

Réception des connexions clientes

```
io.sockets.on('connection', function(socket) {
  console.log('new user');
  ...
});
```

Réception des messages

```
  socket.on('msg', function(msg) {
    msg.type = 'usermsg';

    // Envoi le message a tous les clients connectes
    io.sockets.emit('msg', msg);

    // Envoi le message au relais suivant
    if(msg.nodes.length > 0) {
      ...
    }
  });
```

1.3 Web Cryptography

Afin de bien crypter les messages envoyés et d'utiliser une technologies assez fiable et récente, nous avons privilégié la Web Cryptography 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 :

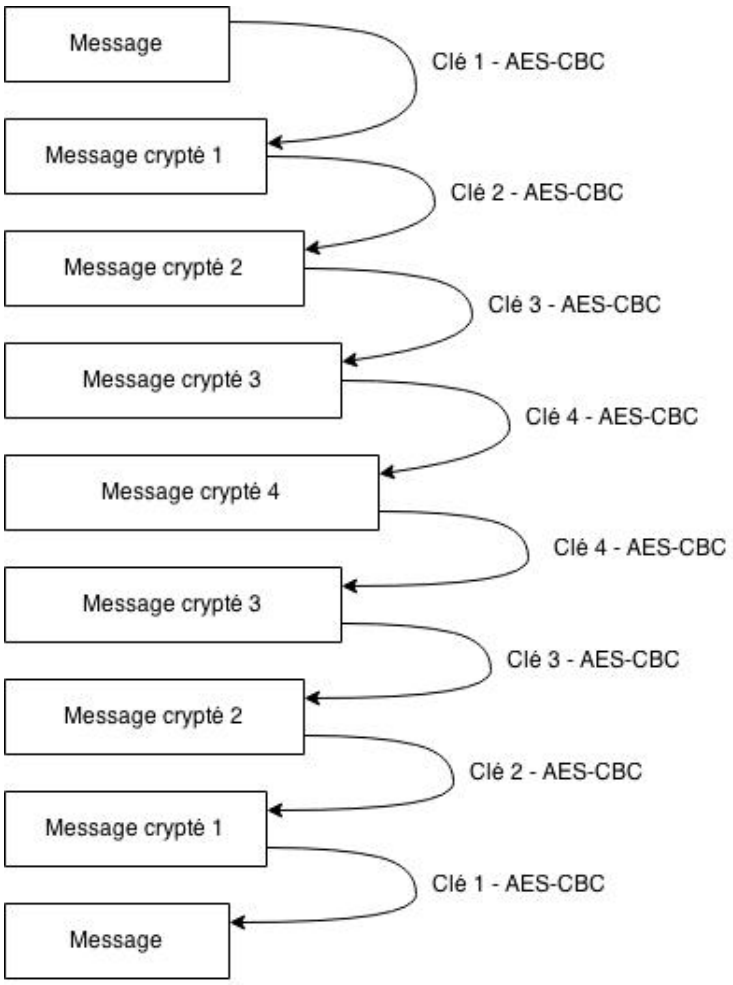

FIGURE 2 – Cryptage d'un message 4 fois de suite en AES-CBC

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");
```

```
  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. Également, 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);
```

Également, 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 :

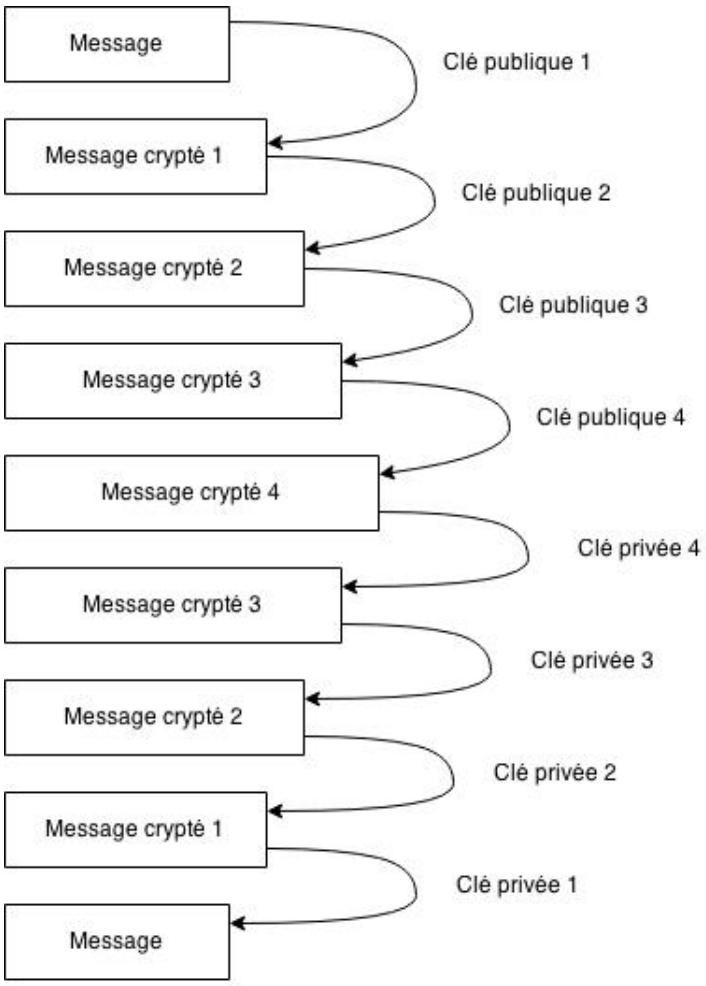

FIGURE 3 – Cryptage d'un message 4 fois de suite en RSA-OAEP

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.

1.4 Architecture de l'application

L'application se compose en 3 parties :

- WebCrypto
- Websocket en PHP
- Websocket en NodeJS

Dans chacune des parties, un README est fourni précisant comment tester la partie concernée. Ces derniers sont écrits en markdown.

La partie Webcrypto comprends une page html ainsi que plusieurs fichiers JavaScript dans le dossier js/. Dans ce dernier dossier, on y retrouve les tests de base en AES-CBC puis en RSA-OAEP.
Voici les fichiers dans le dossier Webcrypto part :

- index.html
- js/
  - jquery-1.11.2.min.js
  - short-cbc.js
  - short-rsa.js
  - test-cbc.js
  - test-rsa.js

Par défaut, la page index.html utilise le fichier test-rsa.js.

Pour la partie utilisant NodeJS, nous avons plusieurs dossiers et fichiers Voici la composition :

- lib/ dossier contenant les librairies
- js/ dossier contenant le code JavaScript pour le client et Socket.io
- node_modules/ dossier contenant les modules pour NodeJS
- css/
- client1.html
- client2.html
- server.js
- package.json

Pour la partie PHP, il y a juste 2 fichiers :

- index.html
- server.php

Comme leurs noms l'indique, la page index.html correspond au client et server.php au server.

2 Mode d'emploi

2.1 Installer et configurer les relais

Dans un premier temps, nous supposons que NodeJS est installé et fonctionnel en ligne de commande.

La première chose à faire est d'installer toutes les dépendances au bon fonctionnement du relais, c'est à dire tous les modules NodeJS dont le programme à besoin pour fonctionner.

Grâce à un fichier *package.json* qui décrit toutes les dépendances à installer, il suffit de taper la commande : (dans le même dossier au se trouve le fichier *package.json*)

```
1 npm install
```

Un dossier *node_ modules* sera créé contenant les dépendances.

Nous pouvons maintenant démarrer autant de relais que nous le souhaitons via la commande :

```
1 node server.js [<num_port>]
```

Évidemment, nous mettons un numéro de port différent pour chaque relais.

2.2 Installer et configurer les clients

Les clients sont sous forme de simple fichiers html. (ex : client1.html)

Cependant, nous devons quand même regarder le code pour les configurer.
En effet, il faut définir l'adresse des relais à parcourir pour envoyer un message.

Ouvrons le fichier *client1.html* et regardons le code.

```
1 var node_addr = 'http://127.0.0.1:9000';
```

Cette ligne décrit le relais qui aura une connexion permanente avec le client.
C'est également le relais qui recevra en premier tous les messages du client.

```
1 var nodes_list = ["127.0.0.1:9001", "127.0.0.1:9002"];
```

Cette ligne décrit la liste des relais suivant à parcourir.
Les clients qui recevront les messages doivent être connectés au dernier relais de la liste.

2.3 Lancer l'application

Prenons pour exemple cette figure :

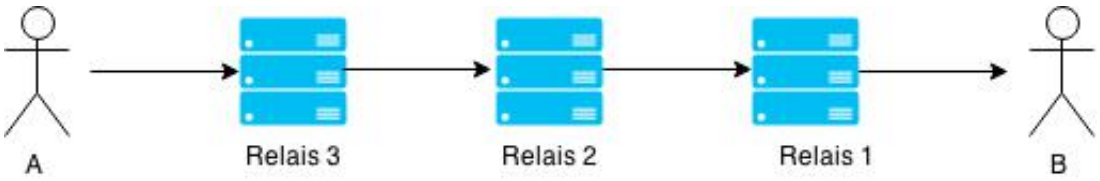

FIGURE 4 – Transfert d'un message de A vers B en passant par 3 relais

Nous sommes un client A et nous voulons envoyer un message au client B en passant par 3 relais d'adresse :

- Relais 3 : 127.0.0.1 :9000
- Relais 2 : 127.0.0.1 :9001
- Relais 1 : 127.0.0.1 :9002

En revanche le client B n'envoie pas de message au client A.

Tout d'abord nous configurons les relais ainsi que les clients comme décrit ci-dessus.
Les relais se lancent avec les commandes :

```
node server.js 9000
node server.js 9001
node server.js 9002
```

Le client A se configure ainsi :

```
var node_addr = 'http://127.0.0.1:9000';
var nodes_list = ["127.0.0.1:9001", "127.0.0.1:9002"];
```

Le client B se configure ainsi :

```
var node_addr = 'http://127.0.0.1:9002';
var nodes_list = [];
```

Maintenant, nous ouvrons le tchat et arrivons sur cette page :

Un message système nous informe que la connexion s'est effectué correctement au premier relais.
Nous pouvons alors envoyer des messages au client B.

FIGURE 5 – Le client A est bien connecté au relais 3

FIGURE 6 – Le relais nous renvoie le message envoyé

3 Tests

3.1 WebSocket

Ici, nous nous intéresserons à la partie NodeJS seulement car c'est la partie finale qui a été choisi pour ce PJI.

Le premier test à effectuer avec NodeJS est si le serveur est bien démarré ou non.

FIGURE 7 – Le serveur est démarré correctement

Sur cette figure, nous pouvons visualiser un *Hello World* lorsque nous tapons l'adresse du relais dans le navigateur. Le serveur est donc bien démarré.

Le code se rapportant au démarrage du serveur est le suivant :

```
1 httpServer = http.createServer(function(req, res) {
2   res.end('Hello World');
3 });
4
5 httpServer.listen(port);
```

Nous voulons tester que le relais reçoit bien la connexion d'un client
Pour cela, il faut d'abord créer une instance d'une socket serveur comme ceci :

```
1 var io = require('socket.io').listen(httpServer);
```

Cette ligne importe un nouveau module NodeJS, il faut donc que ce module soit bien installé au projet.
Si aucune erreur apparaît lors du démarrage du relais alors le module est installé correctement.

Voici le code qui écoute la connexion des socket clientes :

```
1 io.sockets.on('connection', function(socket) {
2   console.log('new user');
3
4   ...
5 });
```

A chaque connexion d'un client au relais, la console affiche donc *new user*.

Nous voulons tester que le relais reçoit bien un message client et le renvoi correctement à tous les clients connectés.

```
1   socket.on('msg', function(msg) {
2     ...
3     io.sockets.emit('msg', msg);
4
5     ...
6   });
```

Si le code fonctionne, tous les clients connectés reçoivent le message.

Le dernier test est de vérifier que le relais traite bien le message qu'il a reçu et renvoi le message au relais suivant le chemin donné dans le message.

```
    if(msg.nodes.length > 0) {
      var next_node = 'http://' + msg.nodes[0];
      msg.nodes = msg.nodes.slice(1);

      if(!io_client_connected) {
        io_client = require("socket.io-client")(next_node);
      } else {
        io_client.emit('msg', msg);
      }

      io_client.on('connect', function() {
        io_client_connected = true;
        io_client.emit('msg', msg);
      });

      io_client.on('disconnect', function() {
        io_client_connected = false;
      });
    }
```

Ici, il faut avoir installé le module *socket.io-client* car pour se connecter au prochain relais de la liste, le relais devient client.

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;
          algorithm.iv = window.crypto.getRandomValues(new Uint8Array(16));

          window.crypto.subtle.encrypt(algorithm, key1, data).then(function (ct1) {
            console.log("AES-CBC encrypt 1:");
            console.log(uint8ArrayToAscii(new Uint8Array(ct1)));

            window.crypto.subtle.encrypt(algorithm, key2, new Uint8Array(ct1)).then(function (ct2) {
              console.log("AES-CBC encrypt 2:");
              console.log(uint8ArrayToAscii(new Uint8Array(ct2)));

              window.crypto.subtle.encrypt(algorithm, key3, new Uint8Array(ct2)).then(function (ct3) {
                console.log("AES-CBC encrypt 3:");
                console.log(uint8ArrayToAscii(new Uint8Array(ct3)));

                window.crypto.subtle.encrypt(algorithm, key4, new Uint8Array(ct3)).then(function (ct4) {
                  console.log("AES-CBC encrypt 4:");
                  console.log(uint8ArrayToAscii(new Uint8Array(ct4)));

                  window.crypto.subtle.decrypt(algorithm, key4, ct4).then(function (pt4) {
                    console.log("AES-CBC decrypt 4:");
                    console.log(uint8ArrayToAscii(new Uint8Array(pt4)));

                    window.crypto.subtle.decrypt(algorithm, key3, pt4).then(function (pt3) {
                      console.log("AES-CBC decrypt 3:");
                      console.log(uint8ArrayToAscii(new Uint8Array(pt3)));

                      window.crypto.subtle.decrypt(algorithm, key2, pt3).then(function (pt2) {
                        console.log("AES-CBC decrypt 2:");
                        console.log(uint8ArrayToAscii(new Uint8Array(pt2)));

```

```
                        window.crypto.subtle.decrypt(algorithm, key1, pt2).
                            then(function (pt) {
                          console.log("AES-CBC decrypt 1:");
                          console.log(uint8ArrayToAscii(new Uint8Array(pt)))
                              ;
                        }, handle_error);
                      }, handle_error);
                    }, handle_error);
                  }, handle_error);
                }, handle_error);
              }, handle_error);
            }, handle_error);
          }, handle_error);
        }, handle_error);
      }, handle_error);
    }, handle_error);
  }, handle_error);
}
```

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));
    console.log(uint8ArrayToAscii(encryptedMsg));
    compteur -= 1;
    return pt;
  }, handle_error);
}
```

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.

Conclusion

Ce projet est très formateur autant dans l'aspect que dans la complexité des technologies. Nous avons passé beaucoup de temps sur NodeJS mais aussi sur Webcrypto API afin de bien comprendre et bien vérifier nos tests afin que tout soit correct. Avec ce projet, nous avons reçu beaucoup de connaissance sur la cryptographie mais aussi sur l'utilisation et la configuration de NodeJS.

Cependant, étant des technologies encore très récentes et en cours de développement, nous avons pu commencer à utiliser ces technologies qui seront développées et utilisées de plus en plus dans un futur proche. Or, tout n'est pas spécialement développé sur ces technologies.

Pour terminer complètement ce projet l'idéal serait développer un module ou une librairie afin de faire communication entre Webcrypto API et la cryptographie inclue dans NodeJS. Cela consiste en la suite logique et complète du projet.