Secure P2P Data Transfer using WebRTC

Une application web permettant d'envoyer des fichiers volumineux rapidement grâce à une connexion peer-to-peer.

Travail de Bachelor

David Baschung

Juin 2021

Supervisé par :

Prof. Dr. Jacques Pasquier–Rocha
et
Arnaud Durand

Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Abstract

Le contexte actuel pour la transmission de fichiers entre particuliers permet d'échanger des informations de taille limitée à l'aide d'e-mails ou de services de messageries, via une simple adresse ou un numéro d'identification. Bien que rapides, ces services ne permettent pas d'envoyer des fichiers volumineux sans passer par une solution d'hébergement (Cloud), ou réduisent automatiquement la taille des fichiers, notamment celle des images et vidéos.

Ce projet vise donc à créer une application web gratuite, accessible rapidement par tout le monde et qui ne nécessite pas d'installation, d'appartenance à une organisation, ni d'hébergement des fichiers sur des serveurs appartenant à des tiers.

Mots-clés : Secure, Secured, P2P, Peer-to-peer, Send, Sending, Data, File, Transfer, WebRTC, Thèse, Travail de Bachelor

Remerciements

Je tiens en premier lieu à remercier ma famille pour l'entraide dont j'ai pu bénéficier, autant sur les plans logistiques, financiers, que pour leur appui moral constant tout au long de ce projet, de mes récentes années d'études et de la période de confinement.

Je tiens ensuite à remercier les membres du *Software Engineering Group*, à savoir Arnaud Durand pour les clarifications qu'il a su m'apporter afin de finaliser mon projet et le Professeur Jacques Pasquier pour la supervision de la thèse.

Préambule

Notations et Conventions

Les mots-clés suivants seront appliqués de manière interchangeable dans l'intégralité de ce rapport, ils possèdent la même signification :

- *Envoyeur, Expéditeur, Emetteur* : la personne envoyant les fichiers
- *Receveur* : la personne recevant les fichiers
- *Pair, Peer, Client, Partenaire, Alice, Bob, Destinataire, Utilisateur* : l'envoyeur ou le receveur
- *P2P, Pair-à-pair, Peer-to-peer* : connexion directe entre deux pairs distants
- *Fichier, Données, Document* : fichiers transférés
- *Métadonnées, Liste de fichiers* : noms et propriétés des fichiers transférés
- *Hacker, Homme du milieu* : un intermédiaire malveillant

Deux codes seront ressassés de manière récurrente. A mémoriser :

- L'envoyeur introduit manuellement le *code envoyeur* dans sa page web. Il reçoit ce code de la part du receveur.
- Le receveur introduit manuellement le *code receveur* dans sa page web. Il reçoit ce code de la part de l'envoyeur.

Table des matières

Liste des Figures

Liste des Tables

Liste du Code Source

1 Introduction

1.1 Motivations

Au cours du 21e siècle, les réseaux se sont considérablement développés et les prix des connexions internet à haut débit se sont démocratisées, à tel point que chaque ménage en possède au moins une. Parallèlement, les services de messagerie et de communication entre pairs se sont développés, si bien que l'envoi d'e-mails est devenu un standard. Ces services, à la base conçus pour envoyer du texte, ne permettaient pas d'envoyer des fichiers volumineux, menant ainsi au développement des clouds, des serveurs de données et VPNs. Ces méthodes, bien qu'efficace pour une organisation, ne constituent pas un accès libre et direct pour un internaute lambda désirant partager des documents avec une tierce personne.

Le but de ce projet est de créer une application web accessible de manière facile et rapide à l'aide d'un navigateur web. Toute personne possédant un navigateur moderne peut ainsi envoyer des fichiers de taille illimitée à l'aide d'une connexion peer-to-peer, sans passer par un quelconque relais intermédiaire. Il doit correspondre aux caractéristiques suivantes :

- Etablir un canal de transfert sécurisé entre un envoyeur et un receveur, sans que les données des fichiers envoyés ne soient lisibles par des personnes intermédiaires sur le réseau.
- Répondre aux principes des piliers de la cyber-sécurité lors du transfert, à savoir, la confidentialité et l'intégrité des données reçues par le receveur, la disponibilité de l'envoyeur et l'authentification de l'envoyeur et du receveur.
- Être accessible facilement et par n'importe qui. En l'occurrence, l'application est accessible via une adresse URL à l'aide d'un navigateur web.
- Ne pas faire appel à l'utilisation de Clouds, stocker les fichiers sur des serveurs externes ou compresser les données pour optimiser leur consommation en mémoire sur ces serveurs.

1.2 Mode d'emploi

Cette section présente les moyens d'accès à l'application et la manière de l'utiliser. Elle expose l'écosystème de l'application du point de vue de l'utilisateur final. Elle présente la phase dite de *découverte*, au cours de laquelle deux personnes établissent manuellement un lien réseau au travers de leur navigateur. Cette phase est suivie d'une seconde période dite de *signalement*, plus technique, que nous examinerons dans le chapitre deux.

1.2.1 Cas d'utilisation

L'application peut être accédée de deux manière par le client. La méthode la plus accessible consiste à passer par le site web, via un navigateur supportant la technologie peer-to-peer. La seconde méthode consiste à utiliser une version pour bureau installable à l'avance (desktop-app). Dans les deux cas, l'application établira une liaison avec le serveur web pour réunir l'envoyeur et le receveur, afin qu'ils créent une connexion peer-to-peer.

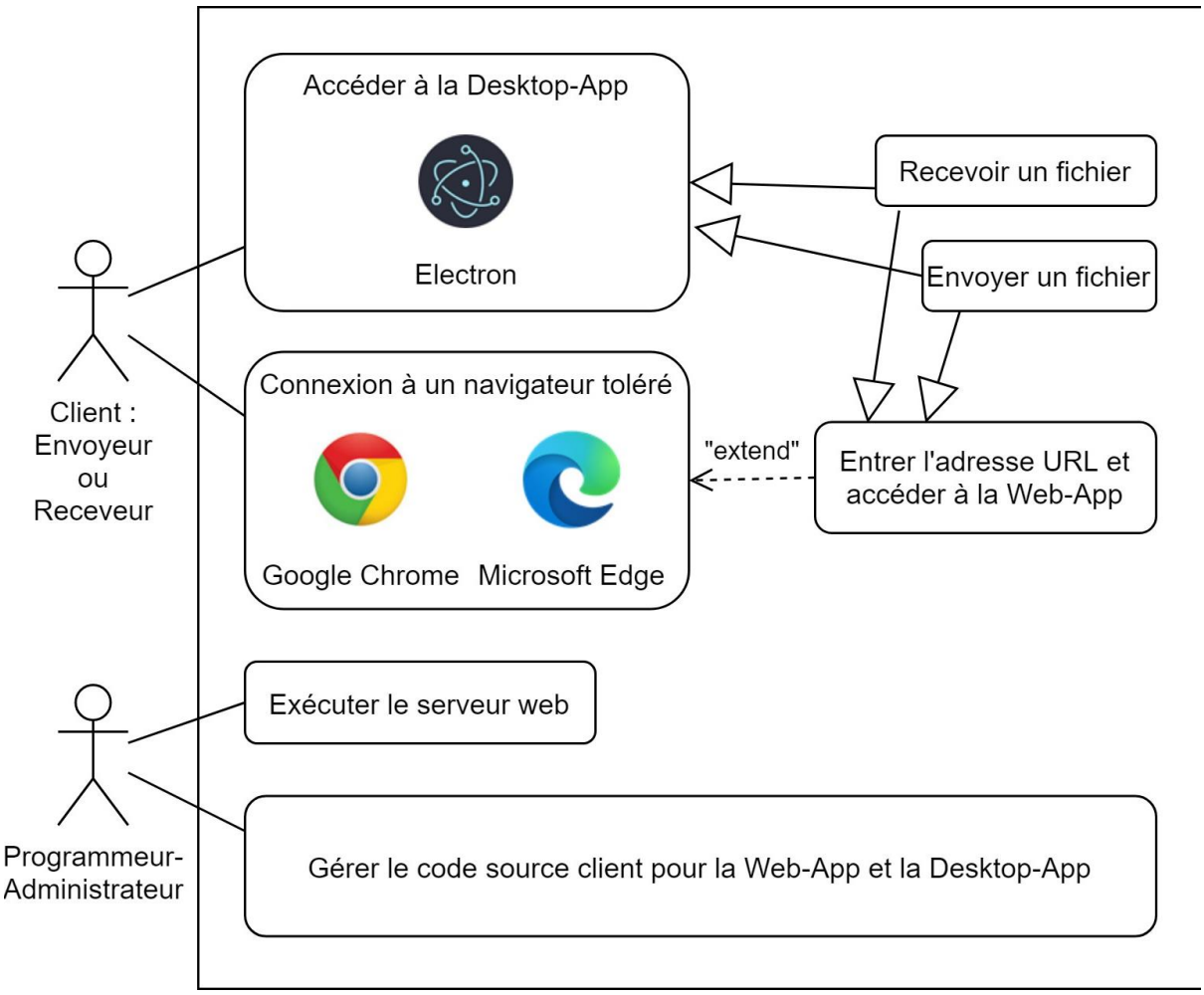

Figure 1 – *Cas d'utilisation : application P2P File Transfer*

1.2.2 Démonstration

Pour accéder à l'application web, il suffit d'entrer l'adresse URL du site web :

https://www.p2psecurefiletransfer.com
ou
https://travail-de-bachelor.herokuapp.com/

Bien entendu, il faut que les deux pairs se connectent simultanément au site web, celui-ci servira alors de canal de signalement pour établir la connexion peer-to-peer entre l'envoyeur et le receveur. Voici ce que chaque utilisateur voit lors de son arrivée sur la page d'accueil :

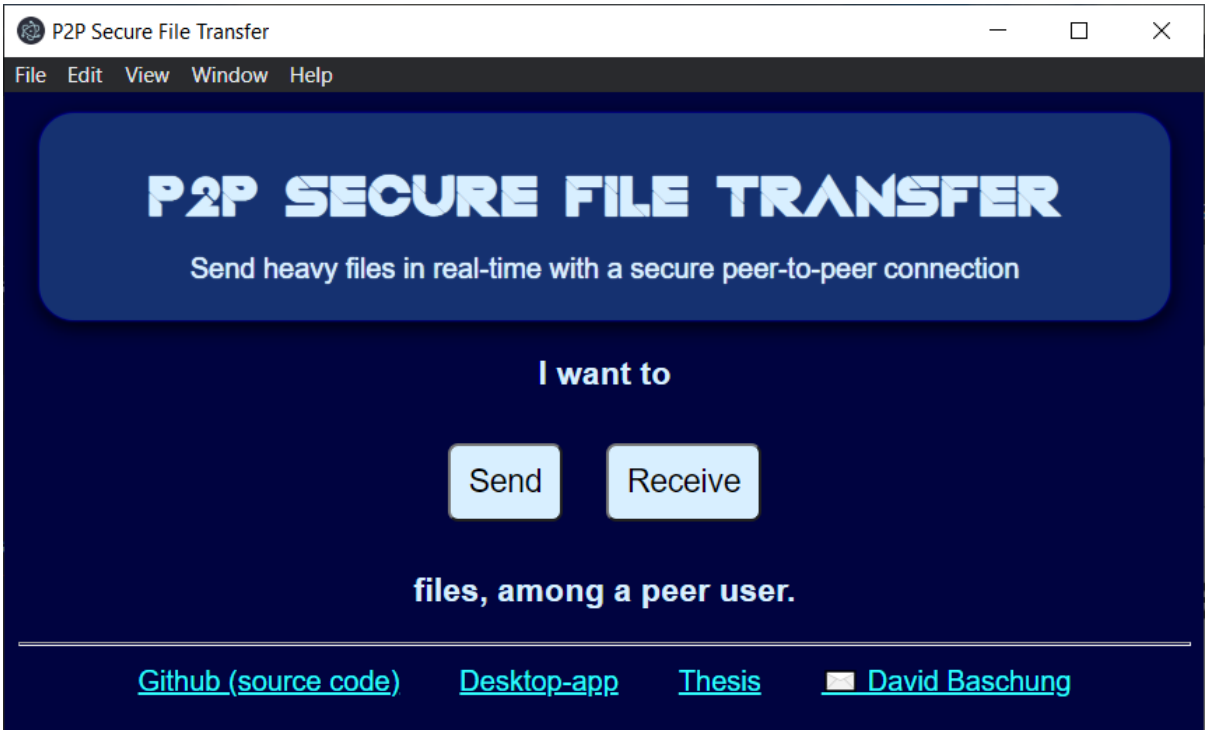

Figure 2 - *Page web d'accueil du client*

Le client va alors choisir le bouton *Send* ou *Receive* en fonction de la position qu'il prétend jouer tout au long du processus. La page se modifie pour afficher les fonctionnalités dont il dispose. S'il décide changer de rôle, il peut revenir à la page principale en cliquant sur le titre.

La page de l'envoyeur s'affichera comme suit après activation du bouton *Send* :

Figure 3 - *Page web d'accueil de l'envoyeur*

La page du receveur s'affichera différemment :

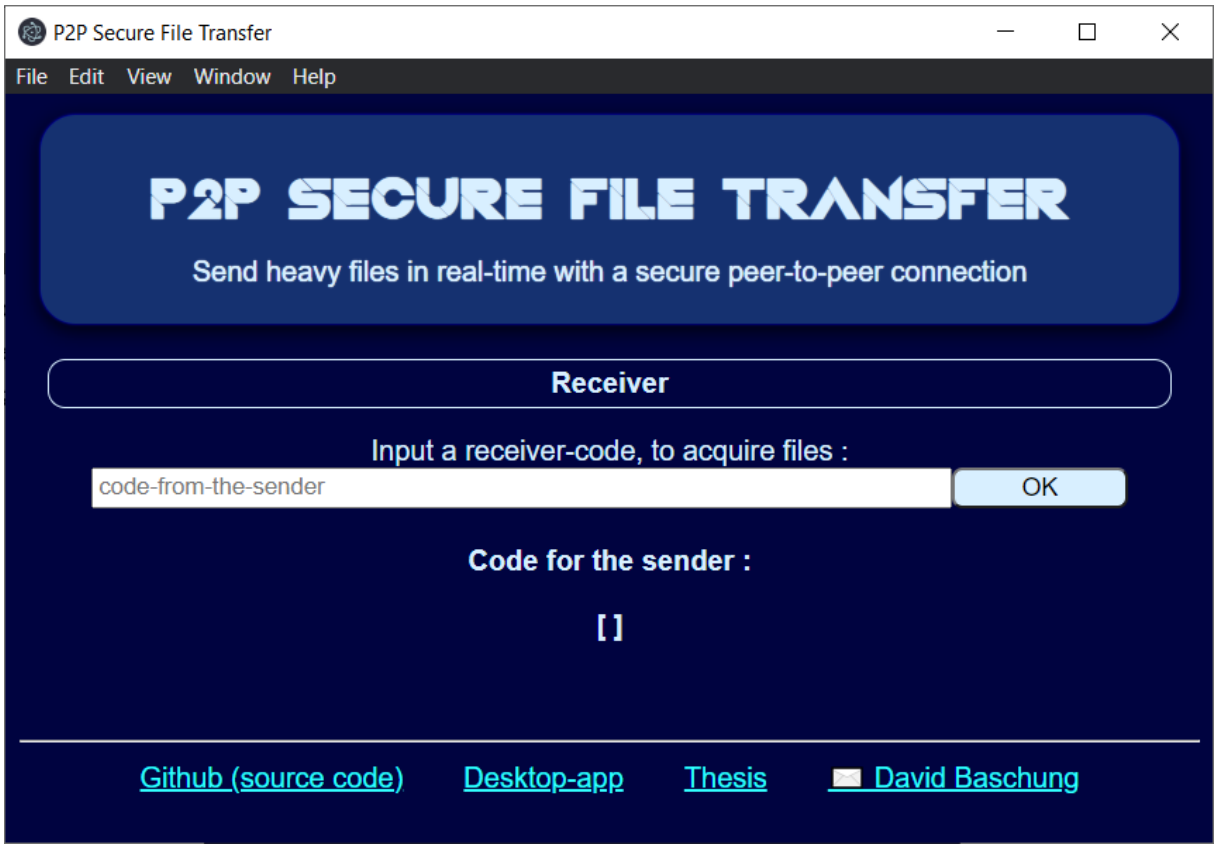

Figure 4 - *Page web d'accueil du receveur*

Prenons la situation de l'envoyeur. Pour sélectionner les fichiers, l'envoyeur doit simplement cliquer sur la boîte verte, qui devient bleue dès qu'elle est survolée. S'il le désire toutefois, il peut faire usage de la fonctionnalité Drag-and-drop en sélectionnant les fichiers à la souris, puis en faisant un glisser-déposer sur la boîte, qui devient jaune :

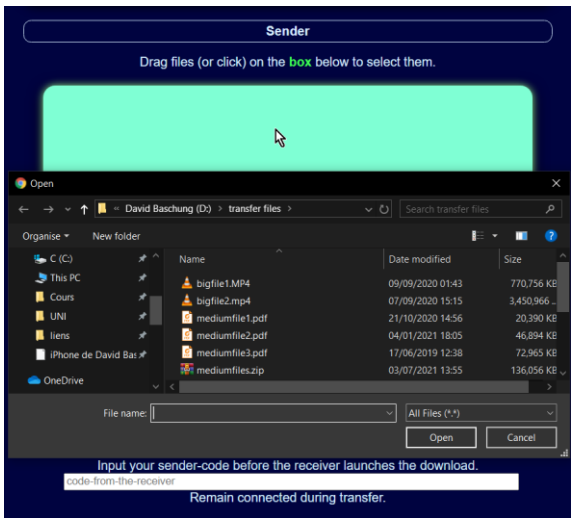

Figure 5 - *Clic sur la boîte*

Figure 6 - *Drag-and-drop sur la boîte*

Les noms des fichiers à transférer sont alors importés et listés dans la boîte verte. Il est ensuite possible de rajouter d'autres fichiers en répétant l'opération. Si l'utilisateur rajoute par mégarde le même fichier à plusieurs reprises, celui-ci sera alors considéré comme un doublon et n'apparaîtra qu'une fois sur la liste.

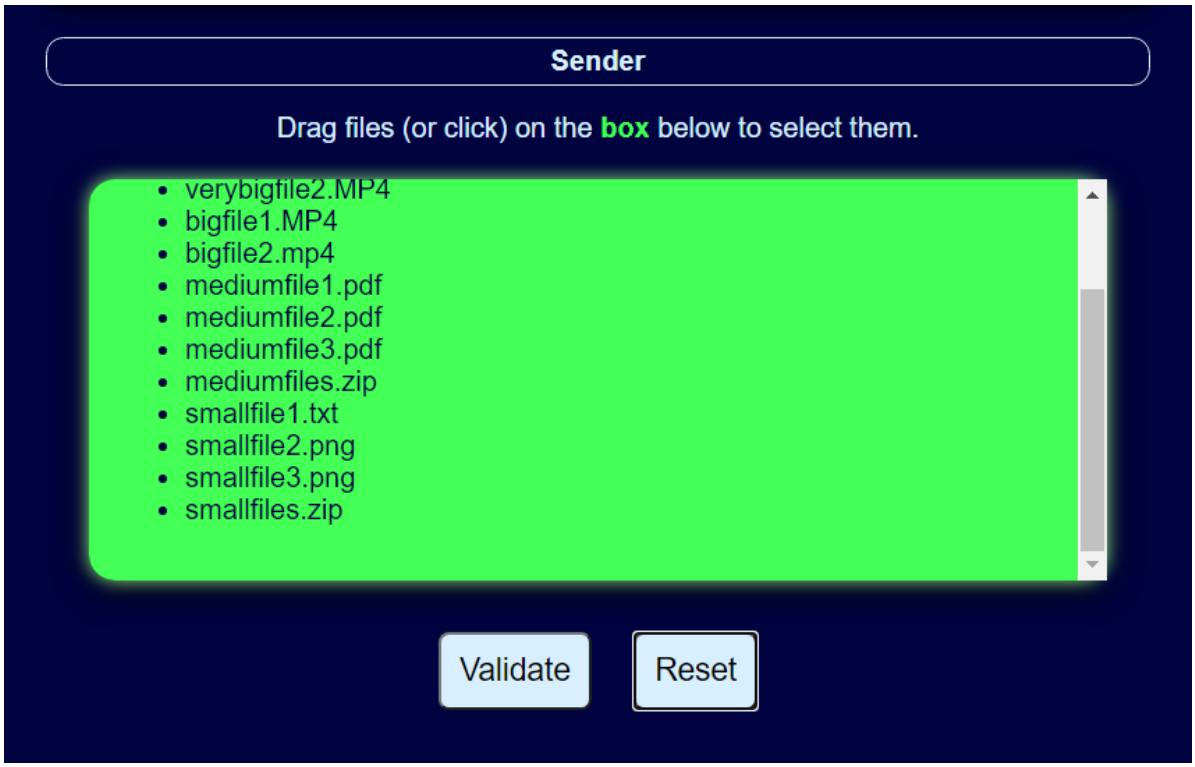

Figure 7 - *Liste des fichiers dans la boîte*

Une fois que l'envoyeur a retenu son choix, il valide la liste en cliquant sur le bouton *Validate*. Cette action crée et affiche un *code receveur* de 4 mots, y associe la liste des noms de nos fichiers, puis transmet le tout vers le serveur web. Par conséquent, ce code fournit au serveur un identifiant qui l'habilite à décrire les métadonnées (nom et taille) des fichiers et permettra plus tard au receveur de décider s'il accepte de télécharger les fichiers proposés ou non.

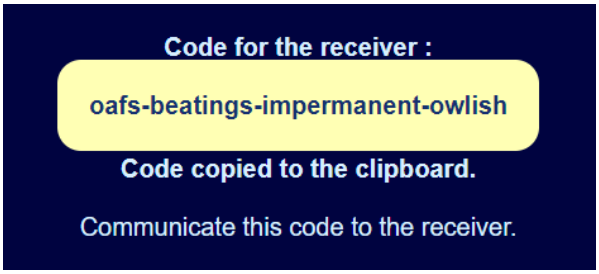

Figure 8 - *Code receveur*

La liste devient ensuite immuable sur la page web de l'envoyeur, nous garantissant ainsi l'exactitude de la liste et l'intégrité des données dès que le transfert aura démarré. L'application étant conçue pour le transfert de gros fichiers, ceux-ci doivent être envoyés séquentiellement et non pas simultanément. Cela permet de cautionner la successivité des téléchargements et leur vérification par la personne les recevant.

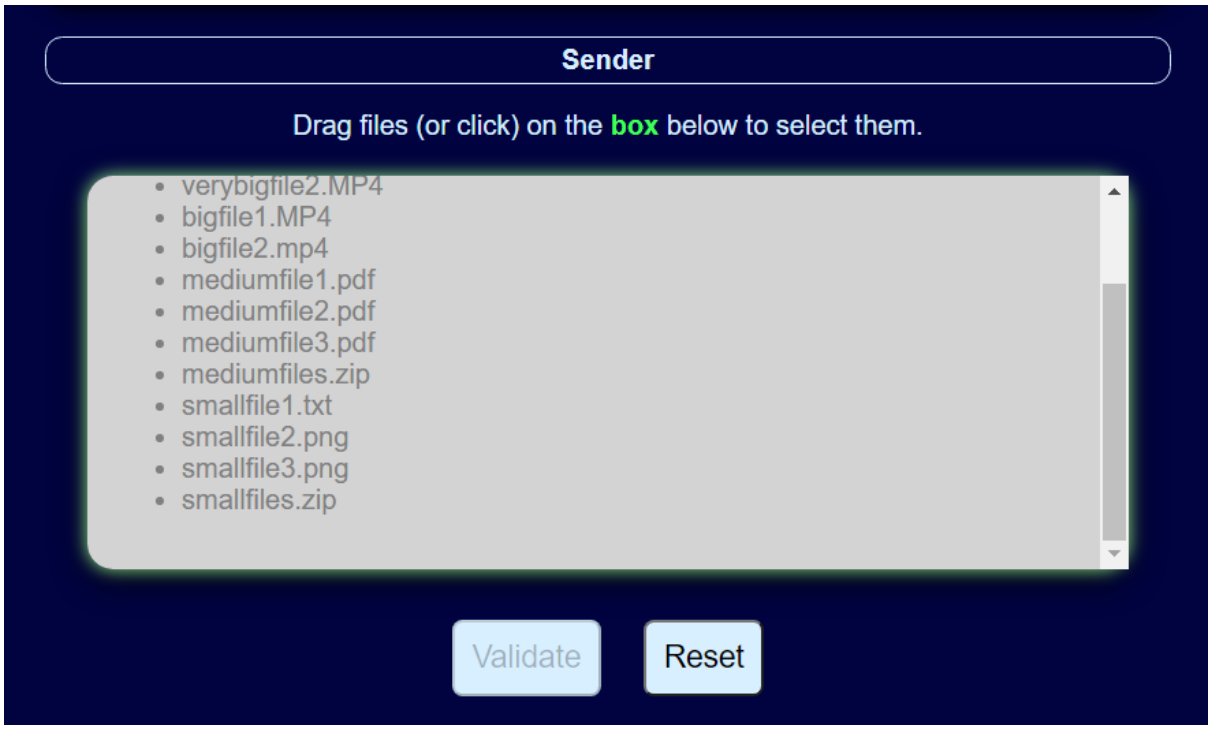

Figure 9 - *Liste des fichiers validée*

Le bouton *Reset* permet également de modifier cette liste si l'envoyeur veut annuler sa sélection. Ce bouton aura également un effet rétroactif après le commencement du transfert, afin de dispenser un moyen d'interrompre la transmission. Son inscription sera donc remplacée par "*Cancel download*".

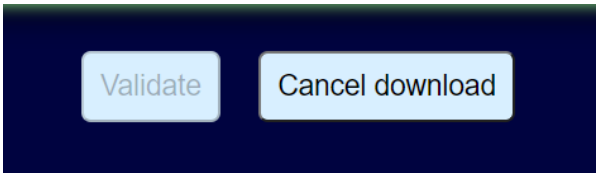

Figure 10 – *Bouton d'annulation du transfert*

Une fois la liste validée, envoyée et le code créé, l'envoyeur doit réaliser une opération sortant du cadre d'utilisation du site web et qui permet au receveur de s'authentifier : l'envoyeur doit communiquer son *code receveur* au receveur à l'aide d'un canal externe de son choix (messagerie, téléphone...). Dans ce but, le code est automatiquement copié dans le presse-papier.

Le receveur, qui s'est préalablement aussi connecté, entre alors en action et doit insérer ledit code dans le champ prévu sur sa page web, puis cliquer *OK* ou appuyer sur la touche *Entrée*.

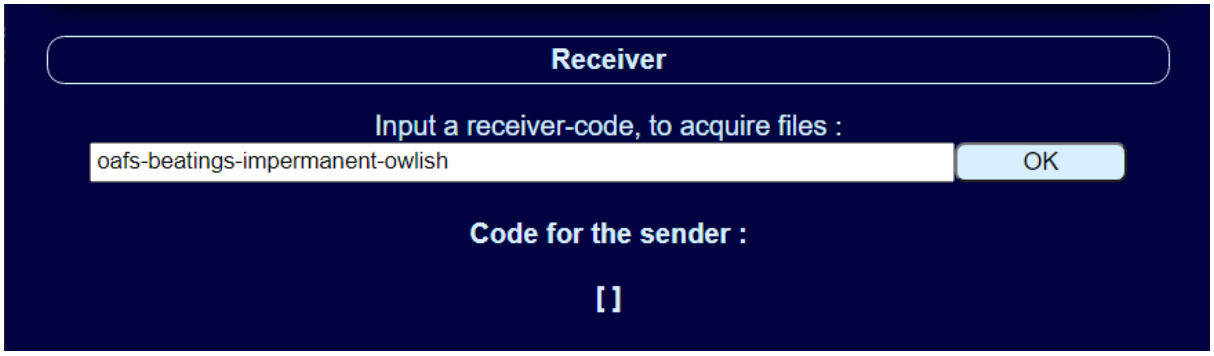

Figure 11 - *Insertion du code receveur*

En validant le code, le receveur émet une requête auprès du serveur web : il demande ainsi les métadonnées qui y sont associées. Si le code est une clé correcte reconnue par le serveur, le bouton *OK* se mue en *Download*. Les métadonnées liées au *code receveur* entré s'affichent en vert en dessous, puis un *code envoyeur* est créé (en jaune, il s'agit donc d'un second code à communiquer) :

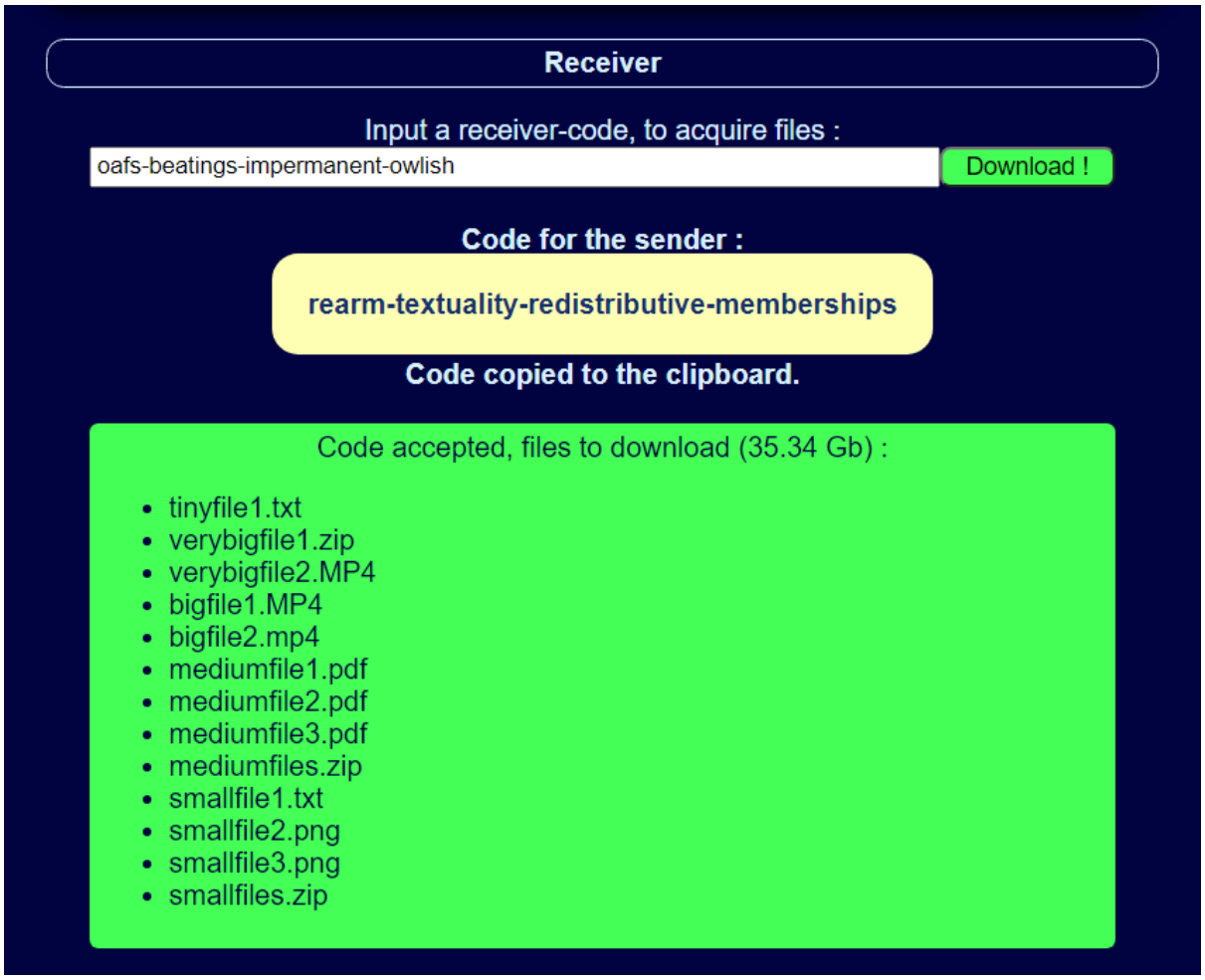

Figure 12 - *Contrôle de la liste de métadonnées par le receveur*

Si le *code receveur* est erroné, un message d'erreur s'affiche et l'utilisateur peut toujours corriger son entrée :

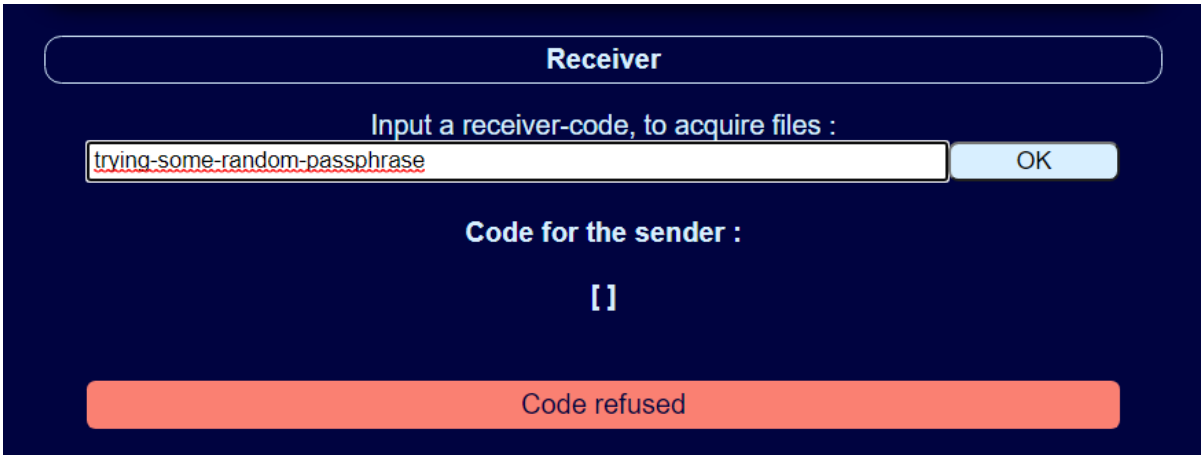

Figure 13 - *Code receveur refusé*

Avant de démarrer le téléchargement, le receveur doit aussi s'authentifier auprès de l'envoyeur. Il lui donne son *code envoyeur*, nouvellement créé, de la même manière qu'il a reçu le sien, par un moyen externe. L'envoyeur insère alors le *code envoyeur* dans son champ et clôt son devoir.

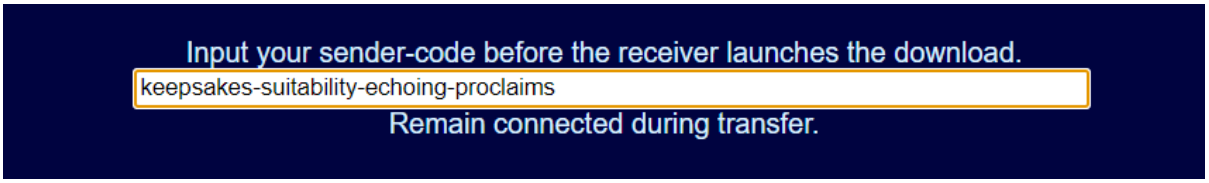

Figure 14 - *Insertion du code envoyeur*

Le receveur démarre alors le téléchargement.
Les fichiers téléchargés s'enregistrent successivement dans le dossier par défaut du système d'exploitation. Ils sont enregistrés progressivement, au fur et à mesure que l'application reçoit des fragments de données (streaming). Dès qu'un fichier est complètement enregistré, son titre change d'apparence dans la liste. Si sa taille est supérieure à 100Mb, il apparaît en vert. Si elle est inférieure à 100Mb, il apparaît en bleu et est muni d'un lien de téléchargement. Ce lien pointe vers une copie du fichier dans la mémoire vive et permet au receveur d'ouvrir et prévisualiser les documents. Pour éviter que l'utilisateur ne quitte la page, cette fonctionnalité peut nécessiter de faire un clic droit sur lien et de l'ouvrir dans un nouvel onglet, selon les réglages du navigateur.

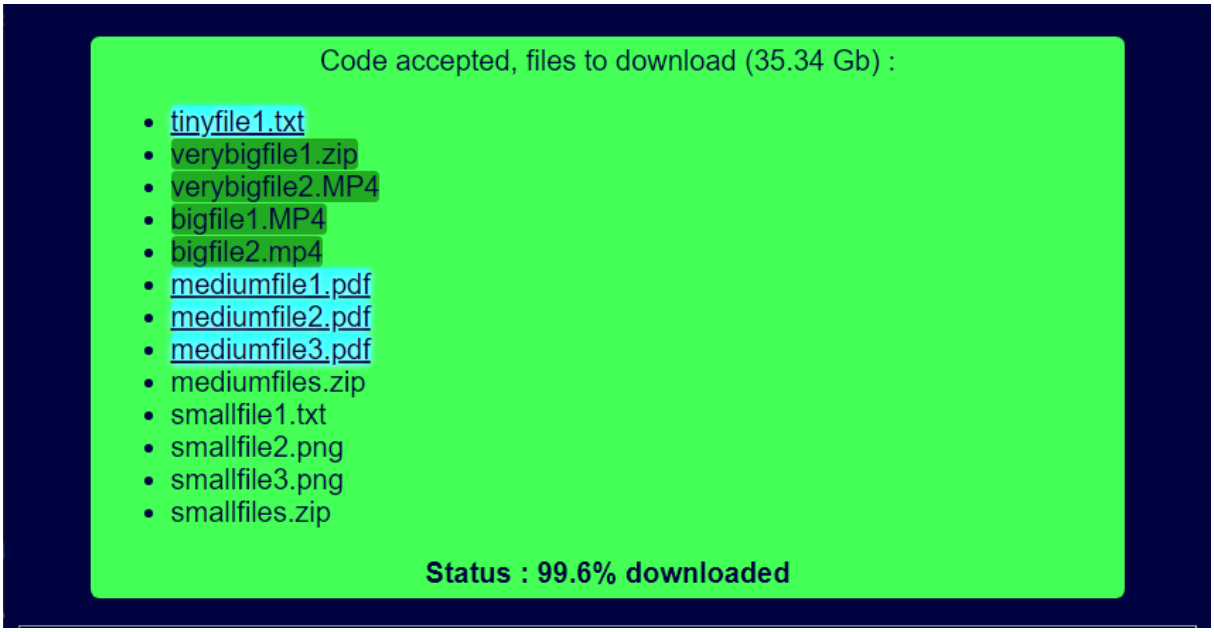

Figure 15 - *Liens de prévisualisation des fichiers téléchargés*

Au bas du panneau d'informations, un statut sur le pourcentage de téléchargement total s'affiche, il se met à jour toutes les demi-secondes.
Ce statut correspond à la quantité exacte de données reçues, indépendamment des aléas du réseau et de la mise en mémoire tampon des paquets de données envoyés. C'est pourquoi il est communiqué à l'envoyeur et indiqué au bas de sa page.

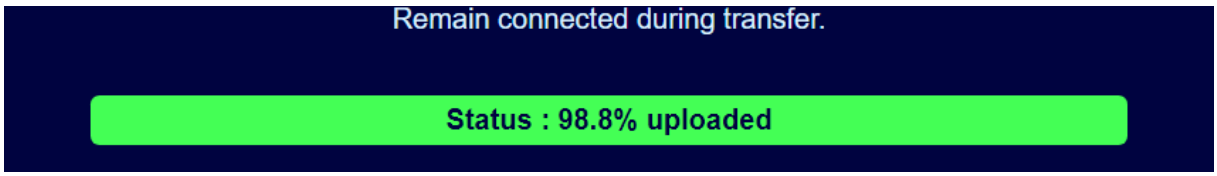

Figure 16 – *Statut de téléchargement retransmis à l'envoyeur.*

Le diagramme de séquence suivant résume le processus pour démarrer le transfert :

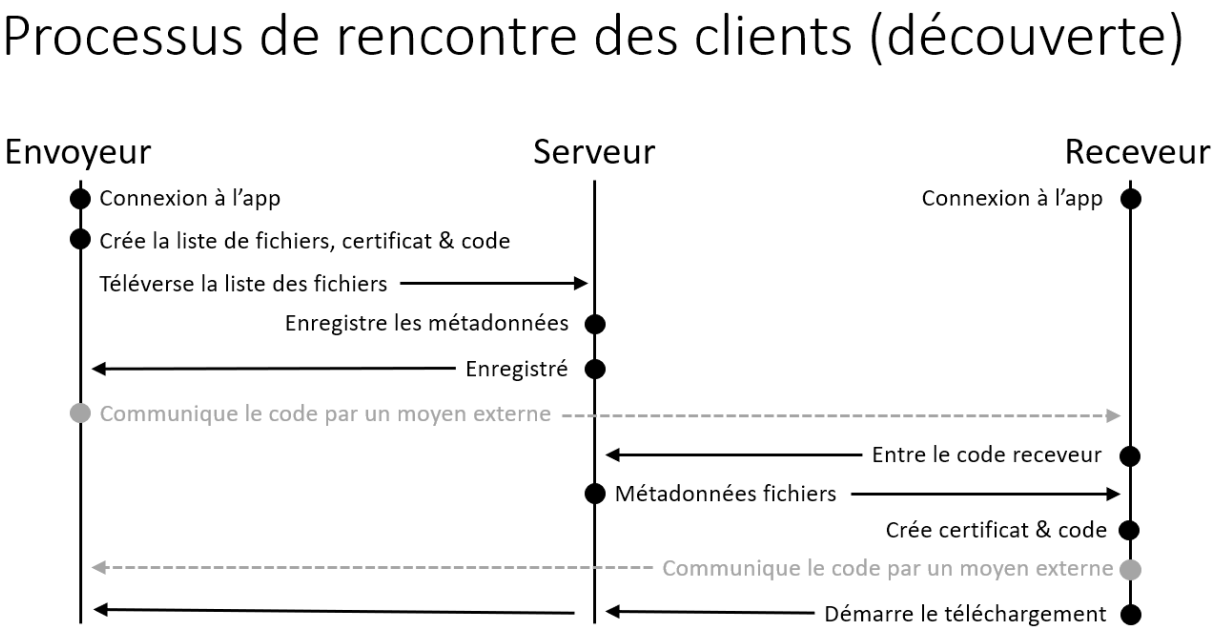

Figure 17 – *Séquence : processus de rencontre des clients*

2 Structure de l'application

2.1 Aperçu de la structure

L'application client constitue une seule unité accessible par l'utilisateur, mais l'ensemble de l'architecture est composé de plusieurs éléments séparés et nécessaires à son exécution. Nous trouvons notamment :

- L'application client, accessible par le site web ou dans sa version installée sur le bureau.
- Un serveur Web répondant aux requêtes, dont le rôle est de fournir les fichiers exécutables de l'application web au client. Ce serveur remplit également une seconde fonction lors de la phase de découverte entre les pairs car il établit la liaison entre le client receveur et l'envoyeur, pour qu'ils bâtissent une connexion peer-to-peer définitive.
- Un serveur TURN faisant office de relais au milieu de la connexion peer-to-peer. Celui-ci est indispensable en raison de la nature de NAT, des pare-feux et pour cacher l'adresse IP privée des clients.
- Des serveurs STUN pour révéler aux 2 pairs leur adresse IP publique.

Voici un exemple de la structure d'une communication établie entre deux pairs :

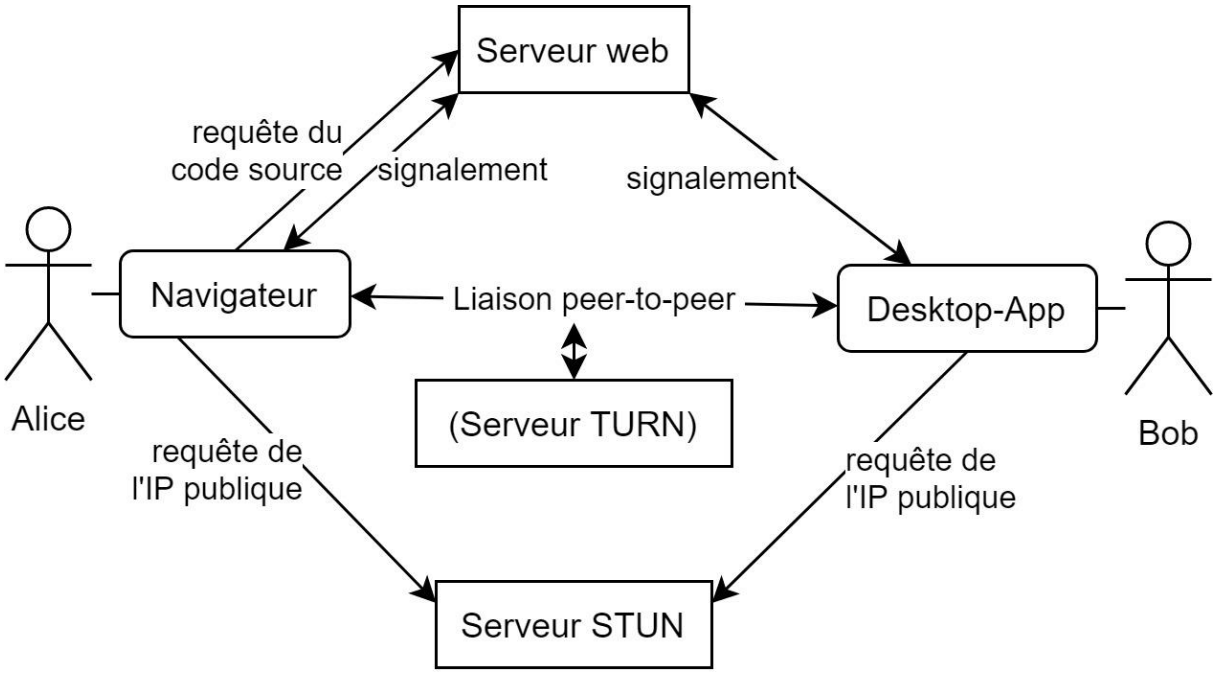

Figure 18 - *Exemple de structure de communication entre pairs et composants intermédiaires*

La constitution des composants mentionnés est précisée ci-dessous.

2.2 Technologies utilisées

2.2.1 Dispositifs du client

Le client utilise les dispositifs d'affichage basés sur *HTML*, *CSS* et *JAVASCRIPT*.

Les moyens d'accès fiables et reconnus sont :

- *Google Chrome* et *Microsoft* Edge pour les navigateurs, car tous deux implémentent l'API *WebRTC* d'une manière suffisamment sécurisée. L'utilisation d'un autre navigateur fera apparaître une alerte chez le client.

- L'application bureau qui utilise *Electron* et embarque les dispositifs d'affichage avec *Chromium*, qui est la base pour *Google Chrome*.

La communication mutuelle avec le serveur est établie à l'aide de *web-sockets*.

La communication peer-to-peer entre l'envoyeur et le receveur utilise *WebRTC*, une API nativement intégrée dans plusieurs navigateurs.

2.2.2 Dispositifs du serveur web

Le serveur est, comme pour le client, programmé en javascript. Il a été conçu pour fonctionner avec l'environnement d'exécution *NodeJS* ainsi que les librairies *express.io* et *socket.io*, de manière à être apte à répondre aux requêtes *https* des clients, présenter la page web et pouvoir communiquer avec eux, le tout à l'aide des mêmes fichiers écrits en javascript. Le serveur joue donc un rôle de point d'accès lorsque les clients accèdent au site, puis sert de médiateur lors de l'échange initial entre les pairs qui tentent de créer un couplage peer-to-peer.

Ce serveur web est hébergé sur la plateforme pour applications cloud *heroku.com*, créée par Salesforce. Le choix de cet hébergeur a été fait en fonction de son prix, de l'affinité qu'il possède avec les applications de type *NodeJS* et pour l'efficacité du déploiement du code source. En effet, il a été possible de créer un pipeline passant par *github.com*, qui informe immédiatement l'hébergeur web lorsqu'une modification du code source a été faite. Un redémarrage automatique du serveur est la plupart du temps imminent, mais peut aussi être forcé.

2.2.3 Dispositifs du serveur TURN

Le serveur TURN sert de relais pour le transfert de données entre les deux pairs lorsqu'ils sont distants et qu'une connexion peer-to-peer directe n'est donc pas possible sur le réseau, Il peut constituer un point d'accès nécessaire pour communiquer avec des pairs se trouvant hors du réseau local LAN car en le traversant avec NAT, l'adresse IPv4 privée du client est muée en adresse IP publique du routeur, plus un port d'accès. Cela implique que si un deuxième pair tente d'envoyer un message au premier, il ne peut pas cibler le pair par son adresse IP privée. Le serveur TURN, en revanche, met à disposition une adresse IP fixe et accessible par les deux pairs. D'autres raisons motivent l'utilisation d'un tel serveur, comme le fait de pouvoir justement cacher l'adresse IP des pairs ou la possibilité d'outrepasser la rigidité des pare-feux sur les réseaux locaux.

Le serveur TURN a été implémenté à l'aide de CoTURN. Ce serveur est déployé sur un Virtual Private Server (VPS) appartenant à un autre hébergeur et indépendant du serveur web. Cette dissociation existe, car les données des clients qui transitent sur le serveur TURN (contrôlé par un tiers), sont sensibles et pourraient être espionnées sur ce serveur. Le chiffrement des informations est accompli par les clients et ces données sont en principe illisible par le serveur TURN. Comme les clients utilisant le navigateur web reçoivent leur code source (non obfusqué et donc modifiable) directement à partir du serveur web, il ne serait pas judicieux de fusionner le rôle de ces deux serveurs. En effet, un serveur unique pourrait donner un contrôle total du chiffrement à l'hébergeur.

2.2.4 Définition des serveurs STUN

Le serveur STUN est un serveur externe au réseau local LAN qui permet aux pairs d'obtenir leur adresse IP publique, telle qu'elle est vue depuis l'ensemble du réseau internet. Il appartient à une entité externe telle que Google ou Mozilla et n'est donc pas contrôlé par nos soins. Lorsqu'un client accède à l'un de ces serveurs, ce dernier retourne l'adresse IP publique à partir de laquelle il vient d'être contacté. Son utilisation permet alors aux pairs de fournir une adresse à laquelle répondre lorsqu'ils envoient un message à l'autre pair.

2.2.5 Communication client-serveur

Pour la version web, l'application est accessible sur le site via l'adresse URL : https://www.p2psecurefiletransfer.com, un domaine de représentation utilisant DNS, ou via http://travail-de-bachelor.herokuapp.com, adresse de base pour l'hébergement de l'application. Si l'utilisateur utilise une URL précédée de "http", il est instantanément redirigé pour utiliser la page sécurisée par https.

Le serveur répond à la requête https *GET* en dispensant la page d'accueil *index.html* et les fichiers de type *CSS* et JS associés.

Pour la version bureau, ces fichiers sont déjà intégrés à l'application, l'utilisateur détient alors une version précise du logiciel et dispose d'une meilleure accessibilité, bien qu'il faille mettre à jour le programme en cas de changement.

Dans les deux cas, au chargement de la page principale, le client établit tout de même une connexion au serveur web à l'aide de web-sockets. Grâce à ce canal de communication bidirectionnel, le serveur et les clients pourront s'envoyer des messages et des données. Ensuite, le serveur constituera ainsi un point d'accès obligatoire pour les deux pairs voulant communiquer entre eux, mais qui n'ont pas connaissance de l'adresse *IP* de leur congénère. Ce processus de reconnaissance entre les pairs est appelé *signalement* et nécessite de passer par le serveur.

2.3 Fonctionnement de WebRTC et intégration à l'app

2.3.1 Aperçu

L'API WebRTC (Web Real Time Communication) comprend un ensemble de méthodes accessible en javascript, qui défèrent au programmeur un moyen simple pour établir des connexions en peer-to-peer. Elle est intégrée au native code package de la plupart des navigateurs modernes et ne nécessite pas d'installation ni d'importation dans le code source d'une page web. Cela permet au client d'établir des connexions en peer-to-peer avec d'autres pairs, puis d'effectuer des actions de réseautage sous forme de datagrammes UDP ou de connexions fiables TCP. Ainsi, cette technologie ouvre un champ suffisamment large pour concevoir des applications de diverses natures, telles que le streaming vidéo, les appels, les chats, l'envoi massif de données, etc. le tout en minimisant le rôle de médiateur que jouerait un serveur entre les pairs. De plus, les procédés mis à disposition ne nécessitent qu'une faible planification dans l'interaction avec le réseau et en matière de cryptographie, car toutes les couches du modèle OSI sont automatiquement gérées. En conséquence, la seule partie du framework à respecter lors du développement est l'établissement interactif de connectivité (processus ICE), qui correspond au déroulement de la procédure de signalement.

2.3.2 Découverte et signalement à l'aide des websockets

Le processus de signalement a lieu entre les deux clients et utilise le serveur de signalement (serveur web aussi en l'occurrence) comme moyen intermédiaire de

communication. Les clients et le serveur s'envoient des messages à l'aide des web-sockets, pour décrire chaque étape d'une connexion en cours de construction.

Ce processus est composé d'une première partie *découverte* servant à identifier les deux pairs et que nous avons vu précédemment dans la démonstration. Il est ensuite suivi d'une deuxième partie *Interactivity Connection Establishment (ICE)* qui consiste à élaborer le lien peer-to-peer définitif.

Web-sockets dans le processus ICE

Lors du processus *ICE*, les deux pairs ont déjà terminé la phase de découverte, Ils se sont distingués à l'aide de leur *code receveur* et *code envoyeur* et connaissent l'identifiant du web-socket de leur acolyte. Ils peuvent donc échanger des messages en passant par le serveur. Ces messages sont associés à un mot-clé, qui décrit l'étape, puis incluent les données à transporter, ainsi qu'un destinataire.

Pour exemple, un client (l'envoyeur des fichiers) écrira le message avec le label *"offerSDP"* au serveur de la manière suivante :

```
1     socket.emit("offerSDP", offerSDP, currentReceiverID);
```

Code 1 – *Exemple d'émission d'un message étiquetté avec socket.emit()*

Le message contient ici une offre et l'identifiant du socket du receveur. Lorsque le serveur interceptera l'envoi marqué du mot-clé en question, un code conçu pour relayer le message au receveur s'exécutera :

```
1     socket.on("offerSDP", function (offerSDP, receiverID) {//réception
2         socket.to(receiverID).emit(           //ré-émission au receveur
3             "offerSDP", offerSDP, senderID=socket.id //contenu ré-émis
4         );
5     }
```

Code 2 – *Exemple de transition d'un message par le serveur de signalement*

Dans ce code, le serveur retransmet l'offre en émettant un nouveau message. Il utilise l'identifiant donné par l'envoyeur.

Finalement, le receveur utilisera le même procédé de capture pour faire usage des données dans l'offre proposée :

```
6     socket.on("offerSDP", function (offerSDP, senderID) {      //etc.
```

Code 3 – *Exemple de réception d'un message de signalement*

2.3.3 Interactive Connectivity Establishment (ICE)

WebRTC définit la classe *RTCPeerConnection*. Elle joue le rôle d'interface pour le programmeur, car elle englobe toutes les informations de la connexion peer-to-peer et contient des classes internes possédant toutes les informations relatives au processus de connexion ICE. Nous verrons que cette classe propose, tout au long du signalement, des mécanismes réagissant à des événements déclenchés par le navigateur du client. En tant que développeur, nous devons passer des méthodes à cette classe, pour qu'elles interceptent les événements en réagissant de la manière à laquelle nous nous attendons.

Voici enfin le procédé de signalement *ICE* dans son intégralité. Le schéma suivant le dépeint dans sa version adaptée à notre application, en respectant les principes requis par WebRTC :

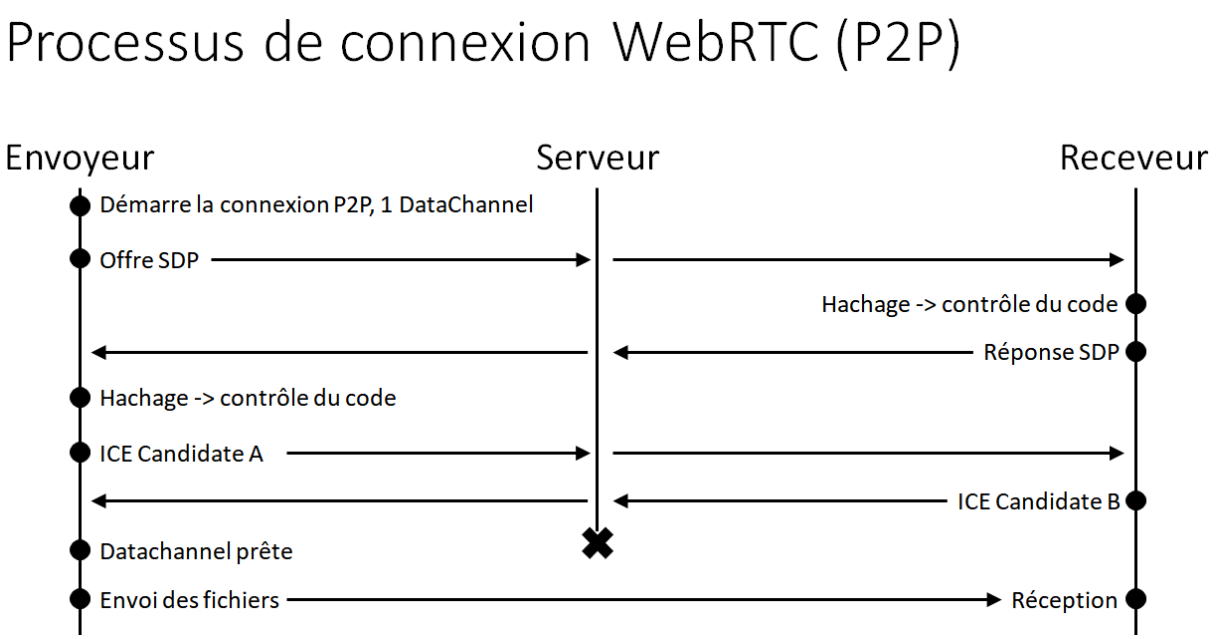

Figure 19 - *Etablissement de la connexion peer-to-peer*

Les étapes de ce schéma se succèdent ainsi :

Initialisation

L'envoyeur reçoit la demande de transfert du receveur et démarre le protocole. Il créée sa connexion peer-to-peer et un canal de données (DataChannel) sur laquelle des données pourront être envoyées de manière fiable, ordonnée, de telle manière qu'un flux continu ait lieu jusqu'au receveur.

De plus, l'envoyeur crée également un certificat attestant de son authenticité. Ce certificat permet d'éviter qu'une attaque de type man-in-the-middle ait lieu, s'il est contrôlé par l'utilisateur. Dans le cas inverse, toute personne se trouvant sur le chemin réseau et pouvant espionner les messages pourrait se faire passer pour l'envoyeur et donc fausser les données, brisant ainsi les principes de confidentialité et d'intégrité.

Ce certificat est ensuite dérivé de manière à obtenir le *code receveur*. De base, la création d'un certificat produit automatiquement un hachage qui permet de l'identifier rapidement. Dans notre cas, ce hash est ensuite redérivé de manière à obtenir le *code receveur* qui doit être communiqué par un moyen externe entre les deux pairs.

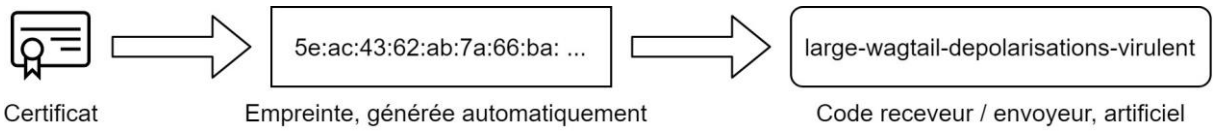

Figure 20 - Génération des codes receveur et envoyeur

Voici un extrait de la console du navigateur, après création du certificat :

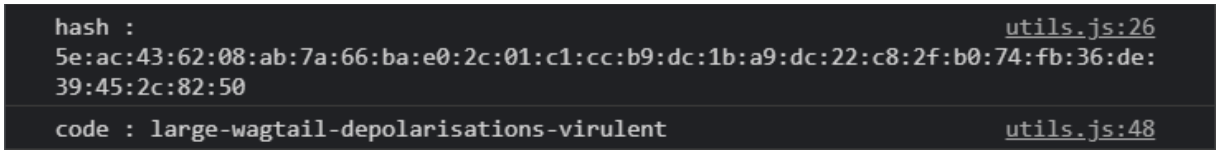

Figure 21 - Création d'une connexion et d'un certificat d'authentification, puis dérivation en code receveur

Offre SDP (Session Description Protocol)

L'envoyeur demande une Offre SDP à son navigateur pour pouvoir l'envoyer à l'autre pair. Cette offre contient des informations quant-à l'adresse IP et au port du receveur, sa connectivité, le type de données transférées (de la vidéo par exemple), et d'autres informations nécessaires ou facultatives. Elle est enregistrée comme description locale et sera directement envoyée au receveur.

Cette offre contient un élément tout particulièrement important : elle communique l'empreinte du certificat de l'envoyeur (*fingerprint*) :

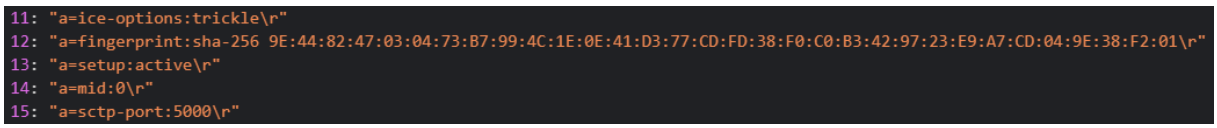

Figure 22 - Extrait de l'offre SDP, qui contient le hash

Parmi l'empreinte et le *code receveur*, seule l'empreinte (hash) est incluse dans l'offre SDP. Le code, affiché sur la page web, servira à contrôler cette empreinte.

Lorsque le receveur reçoit l'offre, il contrôle d'abord l'empreinte du certificat. Pour cela, il transforme aussi l'empreinte en code avec le même algorithme que l'envoyeur avait utilisé. Il contrôle ensuite ce code avec le *code receveur* qu'il a reçu par message, téléphone ou autre de la part de l'envoyeur selon le script suivant :

```
1     socket.on("offerSDP", function (offerSDP, senderID) {
2         console.log("Socket : Received SDP offer");
3         if ( hashToPassphrase(getSDPFingerprint(offerSDP))
4             != inputedReceiverCode ) {
5                     setFeedback(true,
6                           "The sender's authentication certificate
7                            is not valid.", "red");
8             return;
9         }
10    // etc. utiliser l'offre et la DataChannel, créer la connexion
```

Code 4 – Réception de l'offre SDP et contrôle du certificat

Si le code dérivé du hachage par le receveur ne correspond pas au code reçu de l'envoyeur par le canal externe, un panneau d'alerte, rouge, apparaît et bloque instantanément le téléchargement :

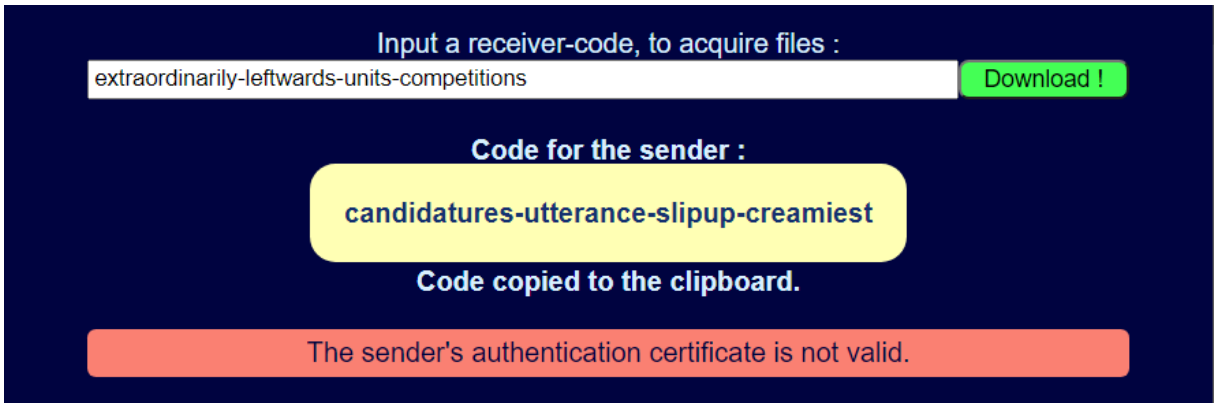

Figure 23 - *Echec de l'authentification de l'envoyeur*

Réponse SDP

La réponse SDP fonctionne de manière analogue à l'offre. Lorsque le receveur reçoit l'offre, il l'enregistre comme description du pair à distance. Ensuite, il crée à son tour une réponse SDP basée sur l'offre puis l'enregistre comme description locale et l'envoie à l'envoyeur.

L'envoyeur hache aussi l'empreinte du receveur en code et en contrôle la correspondance avec celui reçu par le canal externe. Si les codes ne correspondent pas, un panneau d'alerte s'affiche également. Il en est de même si le code transmis par le receveur est faux, ce qui reste possible puisque le *code envoyeur* n'a pas déjà servi à obtenir les métadonnées enregistrées sur le serveur.

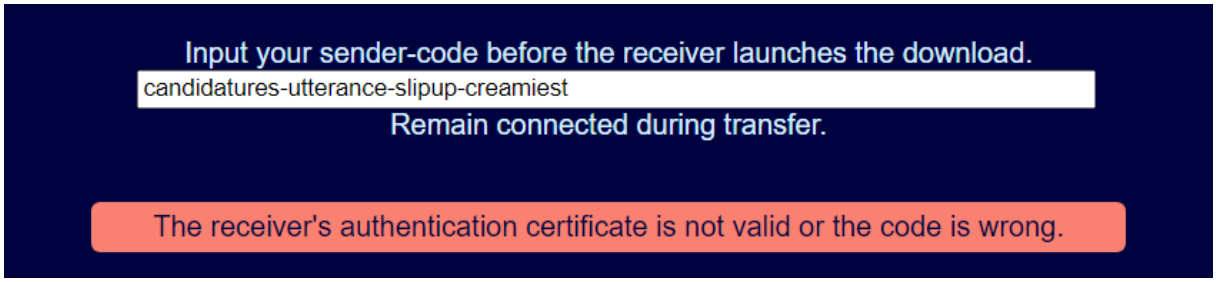

Figure 24 - *Echec de l'authentification du receveur*

STUN & TURN

L'étape suivante du processus ICE est de négocier un chemin réseau définitif pour la connexion peer-to-peer. Elle sera établie en transmettant les candidats ICE. Une étape préliminaire est néanmoins nécessaire pour identifier clairement les deux pairs et le chemin réseau, ceci à l'aide de serveurs externes au réseau local.

Les serveurs STUN et TURN ont été indiqués dès la création de la connexion au début du protocole. Dorénavant, ils sont contactés pour obtenir l'adresse IP publique du client (STUN) et assurer un chemin de connexion fiable (TURN). Ils ont été définis par une adresse pour STUN, suppléée d'un nom d'utilisateur et mot de passe pour TURN :

```
1     var iceServers = [
2         { urls: "stun:stun.services.mozilla.com" },
3         { urls: "stun:stun.l.google.com:19302" },
4         {
5             "iceTransportPolicy": "relay",
6             "urls": "turn:51.15.228.16:3478",
7             "username": "p2psecurefiletransfer",
8             "credential": "quickerthancloudtransfer"
9         }
10    ];
11
```

Code 5 – *Configuration des serveurs STUN et TURN*

La présence de deux serveurs appartenant à des fournisseurs différents garantit de manière très fiable l'accès à un serveur disponible. La présence d'un trop grand nombre de serveurs STUN et TURN (ensemble) peut toutefois causer un ralentissement de l'application.

Le serveur TURN est celui édifié par nos soins. Il forme un relais avec une adresse IP invariable et est accessible à l'aide d'un nom d'utilisateur et d'un mot de passe, comme défini avec CoTURN et selon le standard.

ICE Candidates

La dernière étape du signalement ICE consiste à s'échanger des candidats permettant d'établir la connectivité. Le chemin réseau étant automatiquement déterminé par le navigateur, aucune action n'est entreprise par le programmeur pour en diriger la découverte (contrairement à la phase de découverte du signalement). Le seul rôle restant à jouer est dans la gestion des états de la connexion.

Pour transmettre les candidats proposés par le navigateur, il suffit d'intercepter les événements qu'il produit en attribuant une fonction adaptée au champ *onicecandidate* de la *RTCPeerConnection* :

```
1     async function onIceCandidateRTC_A(event) {
2         while (senderConnection.remoteDescription==undefined)
3             await sleep(50);
4         console.log("RTC : IceCandidateA created, it will be sent");
5         if (event.candidate) {
6             socket.emit("IceCandidateA",
7                         event.candidate, currentReceiverID);
8         } else {
9             console.log ("RTC : End-of-candidates");
```

Code 6 – *Interception des candidats ICE de l'envoyeur et transmission au receveur*

Ces candidats peuvent alors être produits en une certaine quantité et doivent encore être envoyés via les web-sockets. Les candidats proposés sont envoyés de L'envoyeur au receveur puis enregistrés, et vice versa. La recherche continue jusqu'à ce qu'un accord soit trouvé et que le chemin réseau le plus court soit sélectionné. Si un candidat proposé n'est pas complet, cela indique qu'il s'agit du dernier candidat que le navigateur propose.

2.3.4 Datachannels

Lorsque le signalement est terminé, le canal de données (dataChannel) que nous avons façonné au début du signalement chez l'envoyeur sera déclenché par un événement du navigateur. La connexion peer-to-peer s'ouvre et le transfert des données démarre.

Les canaux de données possèdent différentes options que nous devons configurer en fonction du type de données que l'on désire transférer. Voici les options de configuration de notre canal :

```
1     var senderDataChannelOptions = {
2             ordered:true,
3             binaryType:"arraybuffer",
4         };
```

Code 7 – *Options de la dataChannel utilisée lors de l'envoi des fichiers*

L'option *ordered:true* indique que les paquets réseau doivent être transmis dans l'ordre et que leur arrivée chez le receveur doit être certaine. Etant donné que nous envoyons des fichiers, cette option est indispensable pour éviter leur corruption.

L'option *arraybuffer* indique la manière dont sont enregistrées les données dans la mémoire du navigateur. Comme nous n'effectuons aucune modification à l'intérieur-même de ces données, ce type d'enregistrement convient.

2.3.5 Sécurité

WebRTC fournit un environnement propice pour la gestion de la sécurité. Comme la définition de l'API est directement intégrée par le navigateur, il n'y a pas besoin d'utiliser de plugins ou de faire appel à une tierce partie. De plus, comme la version pour navigateur fournit le code source uniquement lorsque le client accède au site web, il n'est pas aisé pour un logiciel malveillant interne de le modifier. En revanche, il est nécessaire de pouvoir faire confiance au serveur web. En conséquence, s'il reste peu probable que ce dernier modifie le code source du client, la possibilité qu'il écoute le trafic réseau le traversant est réelle. C'est pourquoi il est nécessaire que les certificats d'authentification soient générés par les clients et qu'il est judicieux d'utiliser un serveur TURN séparé.

Les composantes corroborées par WebRTC permettent de garantir la confidentialité et l'intégrité. Elles fournissent également des moyens de vérifier l'authenticité, mais ceux-ci sont beaucoup plus rudimentaires.

Dans notre cas, ces trois piliers doivent être garantis pour sécuriser le transport des données. Les contraintes de disponibilité et de ne non-répudiation restent toutefois hors-sujet pour le développement de notre projet.

Le schéma suivant décrit les protocoles et moyens de sécurité utilisés par WebRTC sur les différentes couches réseau du modèle OSI :

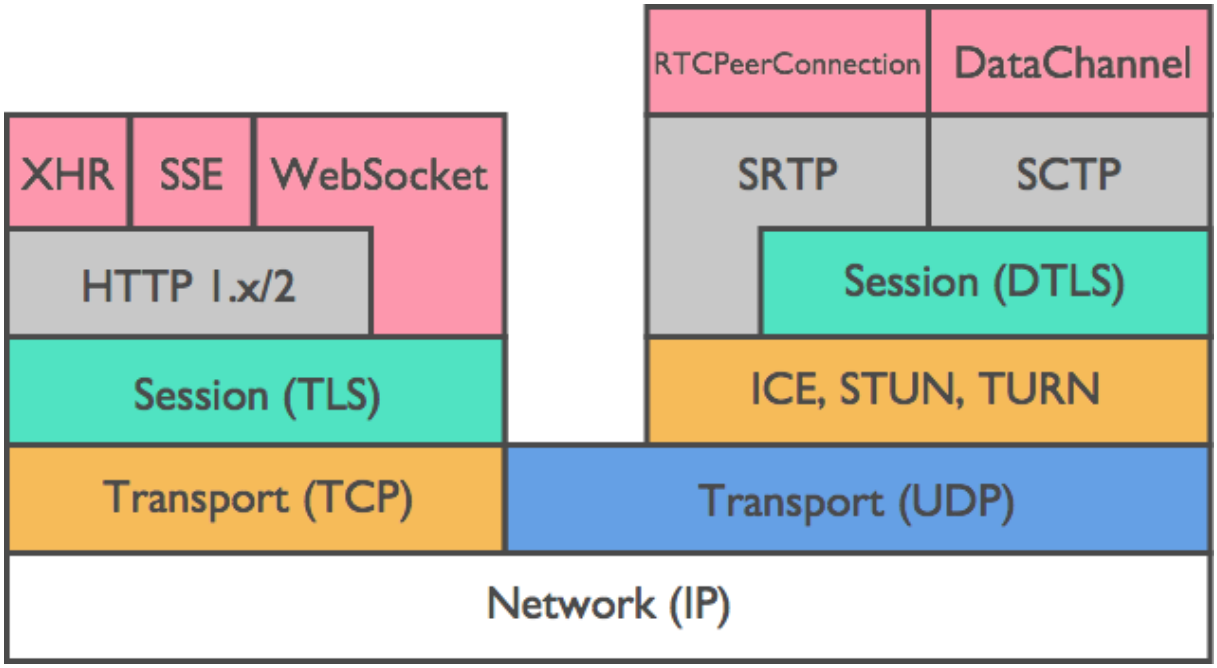

Figure 25 - *Couches des protocoles réseau et moyens de sécurité*

Authentification mutuelle

L'authentification des pairs constitue le point faible de WebRTC en matière de sécurité. Elle est la cause de l'existence du *code receveur* et *code envoyeur* dans l'application.

D'une manière générale en matière de réseaux, une entité (pair, serveur ou autre) peut s'authentifier auprès de quelqu'un d'autre à l'aide d'un certificat. Celui-ci peut être envoyé au début d'une communication et est utilisé pour toute sa durée. Il contient une clé publique, une clé privée correspondante reste conservée secrètement chez cette entité. La clé privée sert à chiffrer les messages et la clé publique, incluse dans le certificat, sert à les déchiffrer.

L'avantage et le problème de ces certificats réside dans le fait qu'ils peuvent être générés par n'importe qui. Ainsi, l'envoyeur peut prouver qu'il est bien l'auteur des données en créant un certificat. Cependant, comme celui-ci est retransmis à l'aide du serveur web lors du signalement, il peut être intercepté et échangé contre un

autre certificat produit par le serveur si celui-ci est malveillant. Pour cette raison, un mécanisme d'authentification additionnel est requis.

Code receveur et code envoyeur

Pour contourner cette faiblesse, une possibilité serait d'utiliser des certificats reconnus par une autorité de certification. Toutefois, ces certificats sont payants, produits en nombre limité et ne peuvent pas être établis à la volée par l'utilisateur. Une alternative consiste à donner un certificat auto-signé directement au pair avec lequel nous voulons rentrer en communication, en dehors de l'environnement d'utilisation du site web. C'est l'équivalent de ce que ne nous accomplissons en partageant le *code receveur* et *code envoyeur*, qui sont des dérivés du certificat.

Confidentialité, intégrité

Dans notre application, la confidentialité et l'intégrité sont garanties par un chiffrement symétrique. Les fonctionnalités pré-implémentées par WebRTC offrent une protection suffisante pour les circonstances de notre modèle de transfert.

Dans notre cas, les données transitent à l'aide d'un canal de données. De manière généralisée, les canaux négocient automatiquement une clé symétrique et basent leur utilisation sur le protocole *Datagram Transport Layer Security (DTLS)*. La négociation s'effectue à l'aide d'un échange *Diffie-Hellman* et établit une clé utilisée tout au long de l'échange, si bien que la confidentialité est forcée dès l'instanciation d'un canal.

De plus, les canaux utilisent une signature numérique. Ils chiffrent l'empreinte d'un message à l'aide de la clé privée du pair l'émettant, si bien que l'autre pair peut déchiffrer l'empreinte à l'aide du certificat et le comparer à son propre hachage du message, il peut ainsi contrôler l'origine d'un message. Si Alice est émettrice d'un tel message, l'illustration suivante démontre le déroulement de la signature :

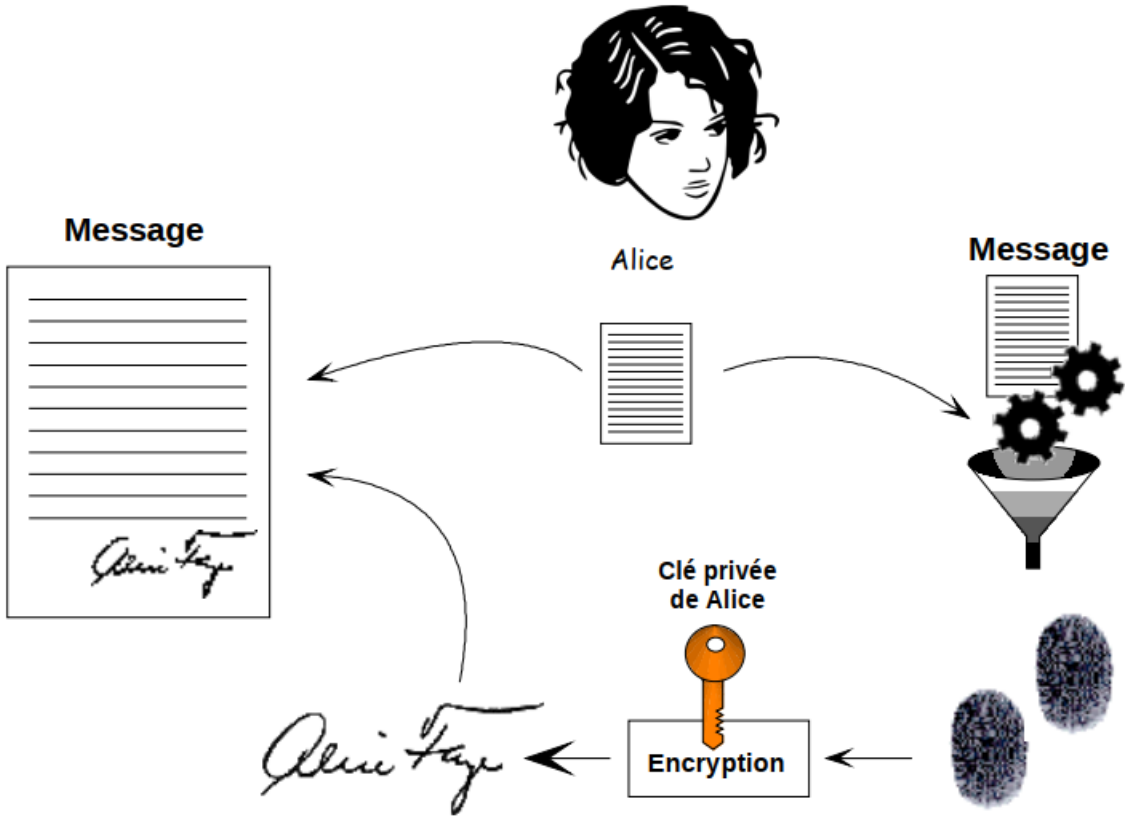

Figure 26 - *Alice participe à l'intégrité en signant un message à l'aide du certificat négocié mutuellement*

Ces certificats ayant été mutuellement authentifiés par nos soins, ils assurent que des paquets falsifiés ne puissent pas être interprétés par le receveur. Finalement, nous avons défini notre canal pour être fiable sur la quatrième couche du modèle OSI (couche de transport), cela grâce à *SCTP*. Il s'ensuit que l'intégrité des données est aussi garantie.

3 Adaptation, code source

Le code source du projet est accessible publiquement sur un répertoire Github à l'adresse https://github.com/davidbaschung/travail-de-bachelor . Il s'agit du pipeline intermédiaire pour le déploiement, dont Heroku.com dispose directement pour l'exploitation du serveur web.

Le contenu est organisé en 3 sous-dossiers contenus dans le dossier *src*. Le dossier *server* contient le code pour le serveur web / de signalement, tandis que le dossier *client* contient l'intégralité des ressources qu'un client téléchargerait depuis le web ou qu'il utilise en exécutant l'application pour le bureau. Le dossier *electron* permet de mettre en œuvre cette application uniquement.

La configuration du serveur TURN n'est pas intégrée dans ce répertoire. Elle correspond simplement à une implémentation standard basée sur CoTURN, grâce à laquelle nous aurons pu définir un nom d'utilisateur et mot de passe (tels que mentionnés dans les serveurs ICE du code client (p.22)).

Les autres fichiers et dossiers dans le niveau principal contiennent essentiellement les exécutables. Des commandes sont disponibles dans la partie *"scripts"* de *package.json*, elles permettent notamment d'exécuter le serveur et compiler le client pour l'application de bureau, respectivement, avec les instructions *npm run server* et *npm run make*, en lorsqu'elles sont exécutées depuis le dossier principal d'un UNIX Shell.

```
1     "scripts": {
2         "server": "node src/server/main.js",
3         "start": "electron-forge start",
4         "package": "electron-forge package",
5         "make": "electron-forge make",
6         "publish": "electron-forge publish",
7         "lint": "echo \"No linting configured\""
8       },
```

Code 8 – *Commandes de lancement, dont celle du serveur, dans la partie "scripts" de package.json*

Une autre forme d'exécutable est disponible avec l'extension *.exe* pour l'application de bureau. De plus, le fichier *Procfile* est requis pas Heroku.com pour lancer automatiquement le serveur.

Finalement, certaines composantes ont été volontairement omises dans ce répertoire. Parmi elles, le dossier *out* intègre une version portable, en plus de l'application de bureau avec l'extension *.exe*. Enfin, les *node_modules* utilisés par NodeJS servent uniquement à la programmation et ne nécessitent pas de mise en ligne, car l'hébergeur emploie ses propres librairies.

3.1 Serveur web

Le serveur web peut être démarré à l'aide de la commande *npm run server*. Elle exécute **main.js** puis imprime une ligne de bienvenue. Le nom de ce fichier représente une simple convention, tandis que l'impression dans la console indique clairement le déclenchement du programme.

La prochaine instruction requiert **server.js**. Les modules *express* et *socket.io* sont chargés, puis un port d'écoute est défini par Heroku.com . Autrement, l'hôte local 4001 le remplace. (Le port est choisi arbitrairement et supérieur aux 1024 premiers ports réservés). Ensuite, le serveur démarre et écoute les sockets des clients requérant une action.

Lorsque le serveur reçoit une nouvelle connexion avec *io.on()* , il reçoit le socket qui l'identifie parmi les (potentiellement nombreux) clients connectés. Il définit alors plusieurs actions possibles pour ce socket, en fonction de la balise au début du message reçu.

```
1     io.on("connection", function (socket) {
2         console.log("User Connected. Socket ID : " + socket.id);
3         socket.on('requestNewRoom', function(files, receiverCode) {
```

Code 9 – *Connexion d'un nouveau client et définition de réponses aux messages balisés*

Dans l'exemple ci-dessus, nous définissons une réponse à la balise *"requestNewRoom"* émissible par le client envoyeur. Une liste de fichiers et un *code receveur* sont fournis en paramètres, puis la fonction que nous définissons permettra de stocker la liste et de l'associer audit code sur notre serveur.

Pour rappel, notre serveur possède un rôle de relais lors de la phase de signalement. Cette mission nécessite au préalable d'établir le lien entre les bons clients grâce au *code receveur* pendant la phase de découverte. D'abord, le client envoyeur émet sa liste, puis, plus tard le receveur entrera le code pour vérifier l'existence de cette liste en ligne. Ces données sont stockées dans la classe *TransferMetaData* de ***server.js*** :

```
1     class TransferMetaData {
2
3         constructor(roomHostSocket, files) {
4             this.roomHostSocket = roomHostSocket;
5             this.hostReconnected = false;
6             this.files = files;
7             var s = 0;
8             for (var f of files) {
9                 s += f.size;
10                this.size = s;
11            }
12            this.hostResponded = false;
13            this.noResponseCount = 0;
14        }
15    }
```

Code 10 – *La classe TransferMetaData contient les métadonnées de la liste des fichiers de l'envoyeur*

Cette classe contient l'identifiant *roomHostSocket* du socket de l'hôte qui a procuré une liste, le recensement des fichiers correspondants dans *files* et la taille sur le disque *size* pour l'ensemble de ceux-ci. Lorsqu'une nouvelle instance de cette classe est construite, la taille est calculée immédiatement.

Le reste de *server.js* consiste essentiellement à gérer la retransmission des messages reçus par les sockets clients pour la découverte et le signalement. Il intervient donc dans toutes les étapes d'établissement de la connexion, raison pour laquelle nous la présentons en premier ici. Les scripts de l'envoyeur et du receveur sont globalement basés sur l'émission ou la réception de ces même messages (avec les mêmes balises), en fonction du rôle. Voici la liste exhaustive des balises et de leur effet lorsque leur fonction respective est appelée, elles sont présentées par ordre chronologique :

Table 1 - Liste des balises des messages et de leur(s) action(s)

```
io.on("connection", function (socket) {
```

Connecter un nouveau client. La fonction contient toutes les réponses de type *socket.on("someLabel", function(...) {* , celles-ci sont activées lorsque le client utilise *socket.emit("someLabel", ...)*

```
socket.on('requestNewRoom', function(files, receiverCode) {
```

Requérir une nouvelle salle pour une liste de fichiers et un *code receveur*. Une salle correspond au point de rencontre entre les deux pairs. Elle est concrétisée par une nouvelle instance de *TransferMetaData()*. A sa création, l'objet est rajouté à un dictionnaire en associant le *code receveur* comme clé, et les métadonnées comme valeur.

```
socket.on("abortUpload", function(receiverCode, receiverID) {
```

Annuler un transfert enregistré. Cette action est lancée par l'envoyeur dès qu'il appuie sur le bouton Reset. Elle efface l'instance de *TransferMetaData* associée au *receiverCode* dans le dictionnaire, puis elle envoie une confirmation de suppression et informe le receveur de l'annulation.

```
socket.on('joinRoom', function(inputedCode) {
```

Rejoindre une salle. Cette demande est faite par le receveur, lorsqu'il entre le *code receveur* sur sa page et tente d'obtenir les métadonnées des fichiers. Le serveur utilise le code comme clé de recherche dans le dictionnaire *transferMetaDataMap*. Si un code existe et est associé à des données, elles seront retournées au receveur avec une mention d'acceptation et un second message informe l'envoyeur. Si le code a été refusé, une simple notation de refus est envoyée au receveur.

```
socket.on("initDownload", function(inputedCode) {
```

Une demande pour initier le téléchargement est sollicitée par le receveur. Après vérification du code, la demande est transmise à l'hôte (envoyeur) pour lancer le processus de signalement.

```
socket.on("offerSDP", function (offerSDP, receiverID) {
```

L'offre SDP est relayée au receveur. Il s'agit de la première étape dans le processus de signalement *ICE* tel que formulé par WebRTC. L'envoyeur vient de créer sa *RTCPeerConnection*, son certificat et son canal de données, il envoie donc l'offre SDP pour configurer le lien pair-à-pair.

```
socket.on("answerSDP", function (answerSDP, senderID) {
```

La réponse SDP est retransmise à l'envoyeur. Dès cette seconde étape, le receveur a également créé une connexion et un certificat. Il réagit donc en remettant ses propres informations de configuration.

```
socket.on("receiverAuthenticationFailed", function (receiverID) {
```

Lorsque l'authentification du receveur a échoué, l'information est retransmise.

```
socket.on("IceCandidateA", function (IceCandidateA, receiverID) {
```

Un candidat ICE est remis au receveur pour chercher un chemin de transmission. Ces candidats sont initiés par l'envoyeur et peuvent être potentiellement produits en grand nombre avant qu'une route définitive ne soit découverte.

```
socket.on("IceCandidateB", function (IceCandidateB, senderID) {
```

Un candidat ICE est retourné à l'envoyeur. De manière similaire aux événements chez celui-ci, il est déclenché par le navigateur du receveur et sera rajouté à la *RTCPeerConnection* de l'envoyeur.

```
socket.on("transferStatus", function (newStatus, senderID) {
```

Communique le statut en pourcentage du téléchargement d'après le receveur.

```
socket.on("restoreConnection", function (receiverCode, isHost) {
```

Reconnecte les deux pairs après une perte de la connexion peer-to-peer. Cette fonction ne sert pas de relais, mais de point de connexion. L'hôte (envoyeur) doit d'abord se reconnecter avec le *code receveur* à une salle pour y mettre à jour l'identifiant réseau de son socket. Le receveur fait de même avec ce code pour vérifier périodiquement la bonne réactualisation de l'hôte (max. 10 minutes), ensuite il redémarre le téléchargement.

```
socket.on("pong", function (receiverCode) {
```

L'hôte d'une salle confirme qu'il est toujours connecté.

```
async function cleanUnusedRooms() {
```

Fonction expérimentale, avec socket.io. Vérifie l'existence de la connexion de l'hôte dans toutes les salles existantes sur le serveur. Chaque hôte est contacté avec un message "ping" et doit pouvoir y répondre. S'il n'y parvient pas, un compteur s'enclenche pour supprimer la salle après dix minutes d'échec.

3.2 Client web

Ce sous-chapitre explique le comportement de la page client sur un navigateur web. Nous verrons plus tard que le client pour bureau s'applique de manière considérablement analogue.

Les fichiers javascript du client sont séparés selon les usages suivants :

- page.js : interactions et modifications de la page web
- client.js : contenu commun pour les deux pairs
- sender.js : processus de signalement pour l'envoyeur
- sendFiles.js : lecture et envoi des fichiers de l'envoyeur
- receiver.js : processus de signalement pour le receveur
- receiveFiles.js : réception et enregistrement des fichiers chez le receveur
- utils.js : contenu utilitaire en dehors de ces catégories mais relatif au projet.

L'arrangement des fichiers est donc conçu pour séparer le modèle WebRTC, l'affichage et les exécutions provoquées par l'utilisateur.

La séparation entre le signalement et le transfert permet de clarifier l'organisation globale. La séparation entre envoi et réception est cohérente dans le cas de notre projet, même si la forme standardisée des clients apparaît généralement unifiée dans d'autres projets, tels que ceux intégrant de la communication vidéo.

3.2.1 Communication avec le serveur, web-sockets

L'intégralité de la procédure de découverte, de signalement, de création et d'abolition du lien peer-to-peer a lieu dans les scripts sender.js et receiver.js. Les éléments communs pour les deux acteurs apparaissent dans client.js. Le processus se fait parallèlement au serveur, en même temps que les actions décrites dans server.js, lorsque notre client envoie des messages avec *socket.emit(...)* et en reçoit avec *socket.on(...)*.

Les méthodes de cette section définissent, emploient et ferment la connexion peer-to-peer à l'aide de l'objet de connexion *RTCPeerConnection,* qui contient un objet de transfert *DataChannel.* Cette connexion est configurée en lui attribuant des données (par ex. *STUN/TURN*, définition d'une *DataChannel()*) et des fonctions/méthodes

indispensables pour préparer l'environnement (par ex. envoyer un candidat *ICE* en passant par notre serveur, quand la connexion en crée).

Une fois la fin du signalement atteinte, la tâche du séquençage et de l'envoi des fichiers sous forme de paquets sera décernée à **sendFiles.js** et **receiveFiles.js**.

client.js

Les éléments communs aux deux pairs sont définis ici en matière de communication avec le serveur et en peer-to-peer. Ils comportent notamment :

- Un web-socket unique par client, pour contacter le serveur.
- l'adresse *URL* du serveur. Pour le client web, elle est disponible dans la barre d'adresse, mais elle doit être enregistrée ("codée en dur") pour l'application de bureau.
- l'algorithme de chiffrement asymétrique utilisé pour chiffrer les certificats.

```
1     var encryptionAlgorithm = {
2         name: 'RSASSA-PKCS1-v1_5',
3         hash: 'SHA-256',
4         modulusLength: 2048,
5         publicExponent: new Uint8Array([1, 0, 1])
6     }
7
```

Code 11 – Définition des algorithmes de chiffrement asymétrique et de hachage

- Les adresses des serveurs STUN et TURN (vues dans le chapitre précédent (p.22))
- Une fonction *getSDPFingerprint(sdpObject)* permettant d'extraire l'empreinte de l'offre ou de la réponse SDP reçue de l'autre pair. Nous en avons besoin en raison du formatage de cette offre sous forme de chaîne de caractères.
- D'autres éléments communs existent dans le script ***utils.js***. Ceux-ci sont nécessaires, mais ils ne sont pas considérablement spécifiques à notre projet.

sender.js

Notre pair émetteur définit en premier un ensemble de variables qu'il va fréquemment utiliser. Elles se nomment :

- *senderConnection* : correspond à l'instance de *RTCPeerConnection*
- *senderCertificate* : certificat d'authentification
- *currentReceiverID* : identifiant du socket du receveur, livré par le serveur. Cet identifiant sert uniquement à gérer la correspondance entre le serveur et le receveur.

Pour notre envoyeur, il servira à indiquer au serveur le destinataire d'un message, lorsqu'il doit le relayer. Exemple d'utilisation :

```
socket.emit("offerSDP", offerSDP, currentReceiverID);
```

- *senderDataChannel* : le canal de données, préparé pour utilisation dans ***sendFile.js***
- *readyForSending;* : Switch permettant d'interrompre l'envoi des données en cas de perte de connexion. Cela évite au buffer de la DataChannel de se remplir car il sera enregistré avant la reconnexion.

Les méthodes de l'envoyeur, les réactions du *socket* et les fonctions pour la *senderConnection* sont définies comme suit :

Table 2 - Liste des méthodes utilisées par l'envoyeur

```
function launchClientSender() {
```

Lance la procédure de découverte. Nous élaborons d'abord la liste de fichiers (nom et taille de chaque fichier), puis nous créons le certificat d'authenticité, en hachons l'empreinte pour créer un code et émettons ce code vers le serveur avec la liste. Le code est affiché dans un *codeLabel*, en jaune sur la page web.

```
socket.on('newRoomCreated', function() {
```

Le serveur confirme la création d'une salle à laquelle le receveur pourra se connecter en entrant le code.

```
function abortUpload(receiverCode) {
```

L'envoyeur peut abroger l'envoi en cliquant sur le bouton *Reset*, Il recevra ensuite un message du serveur pour confirmer l'annulation.

```
socket.on("receiverJoined", function (receiverID) {
```

Le serveur nous informe qu'un receveur a entré le bon code et a obtenu les métadonnées requises avec succès.

```
1  socket.on("initDownload", function() {
2      console.log("Socket : initializing download, receiver
                        ID : ", receiverID);
3      senderConnection = new RTCPeerConnection({
4          iceServers: iceServers,
5          certificates: [senderCertificate]
6      });
7      senderConnection.onicecandidate = onIceCandidateRTC_A;
8      senderConnection.oniceconnectionstatechange =
9                                  iceConnectionStateChange_A;
10     var senderDataChannelOptions = {
11         ordered:true,
12         binaryType:"arraybuffer",
13     };
14     senderDataChannel = senderConnection.createDataChannel(
                             socket.id, senderDataChannelOptions);
15     senderDataChannel.binaryType = "arraybuffer"; //Firefox
16     senderDataChannel.onopen = openSendingDC;
17     senderDataChannel.onclose = closeSendingDC;
18     senderDataChannel.onmessage = (message) =>
           {console.log("DataChannel:message:",message.data)};
19     senderDataChannel.onerror = (error) =>
           {console.log("DataChannel : ERROR : ", error); };
20     startSignaling();
21 });
```

Code 12 – Création du point de connexion peer-to-peer et configuration

Dans *"socket.on(initDownload ..."*, le serveur nous remet une demande d'initialisation du téléchargement. Cette requête intervient lorsque le receveur clique sur le bouton *download*.

L'identifiant *receiverID* reçu en paramètre est enregistré dans *currentReceiverID*. Ainsi, nous pourrons adresser des messages au receveur lors des étapes de signalement ultérieures.

La majorité des objets et méthodes nécessaires à la liaison peer-to-peer sont définis ici. Nous attribuons à *senderConnection* une nouvelle instance de *RTCPeerConnection*, adjointe du certificat déjà créé pour le code ainsi que de la liste des serveurs STUN/TURN enregistrés dans **client.js**

Nous complétons ensuite les méthodes nécessaires à la *senderConnection* pour l'interception des événements lancés par le navigateur. Elle contient deux champs à combler :

1. *senderConnection.onicecandidate* , indispensable pour établir une route
2. *senderConnection.oniceconnectionstatechange* , facultatif. En attribuant une méthode à ce champ, nous pouvons gérer les changements d'état de la connexion. En l'occurrence, une méthode restée expérimentale a été attribuée pour reconnecter les deux pairs en cas d'échec.

Subséquemment, la sender*DataChannel* servant de support à l'expédition des données va être créée. Elle reçoit un paramètre l'identifiant (*socket.id* ici) puis est configurée de manière à envoyer des messages fiables et ordonnés. Le canal contient aussi des champs à combler :

1. *onopen* est requis. Lorsque la connexion sera prête et que le canal sera ouvert, un événement sera déclenché. Nous pourrons donc y réagir en commençant l'expédition des données.
2. *onclose*, *onmessage* et *onerror* sont facultatifs et aident au débogage.

Finalement, *startSignaling()* est appelée. Tous les comportements possibles ont été définis pour notre *senderConnection*. Il est désormais possible d'entamer le processus de signalement pour de bon.

```
1   function startSignaling() {
2      senderConnection.createOffer(
3         function (offerSDP) {
4             senderConnection.setLocalDescription(offerSDP);
5             socket.emit("offerSDP", offerSDP, currentReceiverID);
6          },
```

Code 13 – Fonction pour créer l'offre SDP

startSignaling() Démarre la procédure de signalement. Nous définissons une méthode supplémentaire pour compléter notre connexion. L'offre SDP est créée par la connexion puis nous l'enregistrons comme description locale. Ensuite, nous l'envoyons au serveur, en identifiant le destinataire

```
socket.on("answerSDP", function (answerSDP) {
```

Lorsque le receveur a reçu l'offre, il nous répond avec une réponse SDP. Nous prélevons alors l'empreinte de l'offre et la transformons en un code à 4 mots, dont nous contrôlons la correspondance avec le *code envoyeur* reçu par un moyen de communication externe.

```
async function onIceCandidateRTC_A(event) {
```

Définit une réaction à un événement de création de candidat *ICE*. Cette fonction complète *RTCPeerConnection.onicecandidate*. Lorsque le navigateur propose un candidat, celui-ci est immédiatement envoyé au receveur.

```
socket.on("IceCandidateB", function (IceCandidateB) {
```

Le serveur nous remet un candidat ICE de la part du receveur. Nous définissons une réaction pour ajouter ce candidat à la connexion, dans une fonction interne complétant *RTCPeerConnection.addIceCandidate*.

```
function openSendingDC() {
```

Définit une réaction à l'événement d'ouverture du canal. Cette fonction complète *RTCPeerConnection.onopen*, elle démarre l'envoi des données en appelant *sendFilesAsync()* de sendFiles.js.

```
function closeSendingDC() {
```

Définit une réaction à l'événement de fermeture du canal. Cette fonction complète *RTCPeerConnection.onclose*. Elle ferme le canal d'envoi des données, la connexion peer-to-peer et en vide les variables globales.

```
socket.on("transferStatus", function (newStatus) {
```

Met à jour le statut de téléchargement reçu du receveur, au bas de la page.

```
function iceConnectionStateChange_A(event) {
```

Gère les changements d'état de la connectivité peer-to-peer. Avec socket.io, cette fonction est restée au stade expérimental. Elle démarre un compteur limitant la reconnexion à 10 minutes dès qu'elle est perdue. Pendant cette période, le pair tente des reconnexions à répétition toutes les 10 secondes

```
socket.on("ping", function (id) {
```

L'envoyeur, en tant qu'hôte d'une salle, doit pouvoir répondre aux messages du serveur pour garantir qu'il est toujours connecté.

receiver.js

Ce script définit des méthodes propres au receveur pour la découverte. Pour le signalement, il fonctionne de manière relativement similaire à sender.js mais comporte essentiellement des étapes plus simples et quelques différences. Les variables globales *receiverConnection, receiverCertificate, currentSenderID* et *receiverDataChannel* sont

applicables de manière analogue. Voici les méthodes utilisées :

Table 3 - Liste des méthodes utilisées par le receveur

```
function requireFilesMetada(inputedCode) {
```

Envoie un *code receveur* au serveur après que l'utilisateur ait entré le code et confirmé la requête. Une réponse "code accepté" ou "code refusé" sera reçue plus tard.

```
function download(inputedCode) {
```

Envoie une demande d'initialisation du téléchargement au serveur à l'aide du même code, lorsque le receveur clique sur le bouton *download*. L'envoyeur recevra la requête (*"initDownload",...*) et démarrera le signalement.

```
socket.on("codeRefused", function() {
```

Le serveur a refusé le code envoyé par le receveur. Un panneau d'information exprimant le refus est affiché sur la page web.

```
socket.on("codeAccepted", function(transferMetaData) {
```

Le serveur a accepté le code envoyé par le receveur. Un panneau d'information est affiché sur la page web, il contient la taille du téléchargement affiché en Kb/Mb/Gb/+ et la liste des fichiers.

C'est à cette étape de découverte que le certificat est créé pour le receveur. Comme pour l'envoyeur, il en extrait l'empreinte et la transforme en *code envoyeur*, qu'il doit lui faire parvenir par un un moyen de communication externe. L'utilisateur pourra ensuite cliquer sur le bouton *download* après avoir examiné la liste et livré le code à son partenaire.

```
socket.on('abortDownload', function() {
```

Le serveur nous informe de l'annulation volontaire du transfert, sur commande de l'expéditeur qui a cliqué sur le bouton *Reset*.

```
socket.on("offerSDP", function (offerSDP, senderID) {
```

Le serveur transmet l'offre SDP de l'envoyeur. La demande d'initialisation a été accordée par l'envoyeur et il s'agit de la première étape du signalement pour le receveur. Il applique donc des étapes similaires à celles de l'expéditeur créant sa connexion dans *socket.on("initDownload",...)* (p.22) .

L'empreinte contenue dans l'offre est extraite, transformée en code, puis son identicité est vérifiée par rapport au code reçu par un moyen externe. En cas de contradiction, le message *"authentication failed"* s'affiche dans le panneau d'information.

Le point de connexion *RTCPeerConnection* est créé et attribué à la variable *receiverConnection*. Les champs *onicecandidate, oniceconnectionstatechange* sont complétés. L'offre SDP y est enregistrée, elle contient les informations liées à la *DataChannel* et le navigateur émettra un événement pour nous informer que le canal a été établi avec succès.

Pour l'envoyeur, un champ supplémentaire *receiverConnection.ondatachannel* est utilisé pour recueillir le canal de données créé par l'envoyeur. En lui attribuant notre propre méthode *receiveDataChannelRTC*, nous pourrons obtenir le canal, sa configuration, puis lui attribuer une méthode (*onmessage*) pour gérer la réception des données.

Une fois l'environnement peer-to-peer établi, nous ordonnons à la *receiverConnection* de créer une réponse SDP. Celle-ci est émise vers le serveur avec l'étiquette *"answerSDP"*.

```
socket.on("receiverAuthenticationFailed", function() {
```

Un échec d'authentification du receveur sera communiqué en retour et indiqué en rouge, car ce dernier pourrait appuyer sur le bouton *download* avant d'avoir transmis son code, s'il connaît encore mal l'application.

```
socket.on("IceCandidateA", function (IceCandidateA) {
```

Le serveur nous transmet un candidat *ICE* proposé par l'envoyeur. Il est rajouté à la *receiverConnection* et testé par le navigateur.

```
function onIceCandidateRTC_B(event) {
```

Le navigateur propose un candidat pour une route, que nous transmettons à l'envoyeur.

```
function receiveDataChannelRTC(event) {
```

Le canal de données reçu a été démarré par le navigateur et est donné en paramètre par *event.channel*. Nous l'enregistrons dans la variable *receiverDataChannel.* Cette méthode complète le champ *receiverConnection.ondataChannel.*

Des méthodes peuvent encore être attribuées aux champs du canal. Celui qui nous intéresse réellement est *receiverDataChannel.onmessage*. Nous pouvons y définir une méthode qui s'exécute dès que l'on reçoit un paquet de données du receveur. Nous lui attribuons donc *receiveMessageDC(),* expliquée plus bas.

```
function openReceivingDC() {
```

Le navigateur nous informe de l'ouverture du canal. Cette fonction complète *receiverConnection.onopen.*

```
function receiveMessageDC(message) {
```

Le navigateur nous transmet un paquet de données. Cette fonction complète *receiverConnection.onmessage* et gère la réception d'un paquet de données en peer-to-peer. Elle nous redirige avec la méthode *receiveChunk(message.data)*, définie dans receiveFiles.js. Nous sommes alors prêts pour récupérer et enregistrer les données des fichiers.

```
function closeReceivingDC() {
```

Ferme la DataChannel et la connexion à la fin du téléchargement.

```
function iceConnectionStateChange_B(event) {
```

Gère les changements d'état de la connectivité peer-to-peer. Avec socket.io, cette fonction est restée au stade expérimental. Elle démarre un compteur de 10 minutes. période durant laquelle le pair vérifie toutes les 10 secondes si l'hôte s'est reconnecté à la salle et s'il faut continuer le téléchargement.

```
socket.on("socketsReconnected", function(senderID) {
```

Le serveur nous informe du succès de la reconnexion à la salle avec de nouveaux identifiants réseau, après échec. Le téléchargement peut être repris.

sendFiles.js

Ce script gère l'importation, le chargement, le séquençage et l'envoi des fichiers à transférer. L'ensemble des actions qui y sont décrites sont automatiquement activées et ne nécessitent aucune action de la part de l'utilisateur de la page web.

Les données seront séquencées en petits morceaux de 16'000 bytes contenus dans un *Arraybuffer*. Ce buffer est limité à 16 Mb, car un envoi excessif de données provoque presque systématiquement la corruption des fichiers chez le receveur.

La première méthode activée est *sendFilesAsync()*. Elle est appelée à l'ouverture d'un canal de données dans **sendFiles**. Voici le procédé qui a lieu lors de son lancement :

Table 4 - Liste des méthodes automatiquement activées dans sendFiles.js

```
function sendFilesAsync() {   // (pluriel)
```

Envoie tous les fichiers à l'aide de la connexion peer-to-peer créée dans sender.js. Cette méthode est asynchrone et les sous-fonctions qu'elle exécutera tout au long font usage du *callback*. Soit, les opérations gourmandes en temps sont exécutées sur un *thread* séparé du processus principal. Quand une opération se termine, elle appelle une fonction de type callback qui va informer le processus principal de son achèvement, de manière similaire aux événements déclenchés par le navigateur pendant le signalement. Ainsi, il est possible de charger des fichiers volumineux sans provoquer le gel de l'exécution de la page web.

Pour chaque fichier de la liste d'envoi enregistrée dans *filesToSend*, cette méthode provoque itérativement son envoi avec *sendFileAsync(file)* .

```
function sendFileAsync(file) {   // (singulier)
```

Envoie le fichier *file*. Un nouveau lecteur de données asynchrone (*FileReader*) est créé et une fonction de chargement est attribuée à son champ *onload*. Cette fonction est appelée à chaque fois qu'un morceau de données de 16'000 bytes a été chargé. Elle l'ajoute au buffer du canal avec *senderDataChannel.send(result)* pour autant que la taille maximale de ce dernier ne soit pas dépassée. Elle ordonne ensuite le chargement du prochain morceau, sauf si la taille totale du fichier a été atteinte, auquel cas le callback *sendFilesAsyncCallback(file)* est activé pour charger le prochain fichier.

```
reader.onload = async function(event) {
```

Cette fonction interne à *SendFileAsync(file)* est activée à chaque fois que le lecteur de fichier a chargé un morceau de données. Elle est asynchrone et est le résultat de l'utilisation de *reader.readAsArrayBuffer(slice)*. Les données chargées sont envoyées à la DataChannel. Si une perte de connexion a eu lieu, le buffer est mémorisé et réenvoyé après reconnexion.

```
function sendFilesAsyncCallback(file) {
```

Le chargement et l'envoi d'un fichier a abouti. Le prochain fichier est donc envoyé en répétant *sendFileAsync*. Si tous les fichiers ont été envoyés, le compteur des fichiers est remis à zéro avec *resetFilesSending()*.

```
function resetFilesSending() {
```

Remet à zéro le compteur des fichiers.

```
function restoreDataChannel() {
```

Après un échec de connexion, une copie des données envoyées sur la DataChannel a été faite. Lorsque la situation est rétablie cette fonction permet de ré-envoyer les données qui restaient après l'interruption de la transmission.

receiveFiles.js

Ce script gère la réception et l'enregistrement des fichiers. Comme pour **sendFiles.js**, le déroulement des actions est automatisé.

Les morceaux de données sont reçus sous forme de rassemblements par paquets (*chunks*) à partir du canal de données. Les morceaux sont successivement rajoutés à une liste *currentReceiveBuffer* temporaire réinitialisée à chaque nouveau fichier reçu. Ce buffer peut contenir un grand nombre de morceaux mais est limité à 100Mb car il est essentiellement conservé dans la mémoire vive et sert à la prévisualisation des fichiers sur la page web. Les données réellement enregistrées sur le disque dur ne sont pas mises en mémoire tampon mais écrites directement à l'aide d'un flux (streaming) au fil de l'évolution du transfert. Grâce à cette méthode, aucun temps d'attente n'est engendré une fois le téléchargement terminé.

Deux méthodes ont lieu dans ce script :

Table 5 - Liste des méthodes automatiquement activées dans sendFiles.js

```
function receiveChunk(chunk) {
```

Reçoit un paquet de morceaux d'un fichier. Cette méthode est appelé à partir de la méthode de callback définie dans *receiverDataChannel.onmessage* de receiver.js.

Au début de la fonction, le statut est calculé et les 3 premières conditions *if* initialisent le graveur de fichier.

Dans la quatrième, si la taille totale du fichier à télécharger actuellement n'est pas atteinte, le paquet de données reçu est écrit sur le disque dur et éventuellement rajouté à *currentReceiveBuffer*.

Dans le cas contraire nous pourrions incrémenter le compteur de fichier et considérer les nouveaux morceaux arrivés comme appartenant au prochain fichier. Toutefois, il est encore possible que nous ayons reçu trop de morceaux dans un même paquet et qu'une partie appartienne encore au fichier actuel. Il faut alors faire la différence entre la taille des données reçues et la taille du fichier actuel pour rajouter les morceaux restants et isoler ceux du prochain fichier. (Chez l'envoyeur, il n'est malheureusement pas possible de forcer la *DataChannel* à envoyer les morceaux restants d'un seul fichier).

Une fois que le buffer est plein, un nouvel objet de type Binary Large Object est créé. Cet objet est ensuite passé à un nouvel objet URL avec un lien. Le nom du fichier apparaît en bleu dans la liste et l'utilisateur peut visualiser le fichier en faisant un clic droit dessus.

Le buffer est ensuite réinitialisé. Si tous les fichiers sont téléchargés, la fonction *resetFilesReceiving* est appelée et le canal de données est fermé.

```
function resetFilesReceiving() {
```

Réinitialise le compteur de fichiers, le buffer et la taille des morceaux reçus.

3.2.2 Page Web

index.html

Par convention, la page *index.html* est la première page accédée lorsque le client accède à l'aide de l'URL. Dans notre cas, il s'agira également de la seule page nécessaire. En effet, les changements de page sont en réalité des modifications effectuées avec javascript. De cette manière, il est possible de garantir qu'un client arrive toujours sur la même page lors de son chargement.

sender.html & receiver.html

Ces pages sont des éléments partiels de la page index.html, ils remplacent les pages normalement accédées sur un site en cliquant sur de nouveaux liens URL. Ces portions sont chargées dynamiquement à l'aide de requêtes Ajax auprès du serveur, lorsque l'utilisateur choisit le bouton correspondant à son rôle sur la page d'accueil.

stylesheet.css

Les styles utilisés au chargement de la page et lors de sa modification sont prédéfinis ici. Généralement, lorsqu'une modification de l'affichage a lieu dans page.js, les attributs de style sont modifiés pour une balise HTML visée. Ces attributs sont rassemblés dans une seule feuille de style source et peuvent être modifiés facilement.

page.js

Les fonctions dans page.js sont chargées d'effectuer les modifications sur la page web. Elles peuvent être facilement appelées depuis les autres scripts, de telle manière que ceux-ci n'aient pas à gérer les noms des variables et des balises dans le contenu *html*.

Deux listes globales importantes sont définies au début :

- *filesToSend* : elle contient la liste des fichiers à transmettre et affichés dans la box verte. L'envoyeur la valide avec le bouton *Validate*, elle peut ensuite être transmise au serveur de signalement.
- *filesToReceive* : elle contient la liste des fichiers à recevoir. Cette liste est remplie dès que l'envoyeur entre le bon *code receveur* et le panneau vert les énumérant apparaît sur la page web.

D'autres variables sont ensuite définies, notamment pour identifier la *box* verte et confirmer l'état de validation de la liste.

Les fonctions sont définies comme suit :

Table 6 - Liste des méthodes permettant de modifier la page web

```
function create(nodeString, content) {
```

Créer un nouvel élément sur la page. Le premier paramètre doit correspondre à un type d'élément *html*, le second insère une valeur (texte ou html) à l'intérieur de la balise.

```
function setRole(role) {
```

Charge la page référant au rôle d'envoyeur ou receveur désigné par le client lors de son arrivée.

```
function clickReceiverField() {
```

Vide le champ du receveur pour entrer son code, lorsqu'il clique dessus.

```
function inputInReceiverField (event) {
```

Surveille les entrées d'un utilisateur dans le champ prévu pour son code. S'il appuie sur la touche *Entrée*, une action de clic est simulée sur le bouton *OK*. Si d'autres caractères sont insérés, le bouton est réinitialisé, de même que l'éventuel panneau d'information.

```
function receiverCodeButton(event, action) {
```

Exécute l'action du bouton *OK/download* pour le champ du *code receveur*. Pour la première action, le code est envoyé vers le serveur de signalement. Pour la seconde action, la demande d'initiation du téléchargement est faite.

```
function setReceiverCodeButtonAction(action) {
```

Met à jour l'action du bouton *OK/download* et modifie son apparence. Les deux premiers cas de '*action*' adaptent le bouton en fonction de leur effet respectif, le troisième cas désactive le bouton si le téléchargement est lancé.

```
function getInput(isReceiver) {
```

Retourne le *code envoyeur* marqué en jaune sur la page, pour un client.

```
function setFeedback(isReceiver, message, highlightColor) {
```

Affiche un feedback au pair. Le premier paramètre est une variable booléenne.

Si elle est vraie, le panneau d'information du receveur sera affecté, autrement, celui de l'envoyeur. Le second paramètre contient le message texte ou *html* à afficher. Le dernier paramètre est une étiquette de style pour le choix de la couleur du panneau.

```
function getFeedback(isReceiver) {
```

Retourne le feedback de l'envoyeur / receveur. De telles fonctions sont rudimentaires, mais permettent d'isoler l'accès à des noms de tags spécifiques dans le seul script page.js.

```
function createLink(file, index, virtual) {
```

Crée un lien de téléchargement bleu ou vert (virtuel), sans objet URL.

```
function updateTransferStatus(isReceiver, text, instantaneous) {
```

Met à jour le statut de téléchargement, autant pour le conteneur du receveur que pour celui de l'envoyeur. La variable booléenne *instantaneous* permet de mettre à jour le statut immédiatement ou non. Ainsi, un statut ayant atteint les 100% peut être forcé.

```
function dragOverAction(event) {
```

La *box* cataloguant les documents devient jaune lors du survol de la souris pendant un drag-and-drop. Ce cas s'applique lorsque des documents à importer sont sélectionnés.
La *box* peut aussi devenir bleue lors d'un survol de la souris simple (sans drag-and-drop). Il s'agit cependant d'une simple propriété *hover* dans la feuille de style et non d'une fonction.

```
function dragLeaveAction(event)  {
```

La *box* cataloguant les documents redevient verte quand la souris quitte la zone.

```
function dropAction(event) {
```

La *box* cataloguant les documents enregistre et affiche les fichiers importés lors d'un relâchement de la souris, puis redevient verte.

```
function browseFiles(event) {
```

Ouvre une fenêtre pour sélectionner des fichiers, lorsque l'on clique sur la *box*.

```
function addFiles(files) {
```

Traite l'importation des fichiers au sens pratique, après un drag-and-drop ou browsing. Le paramètre *files* contient la liste des fichiers sélectionnés. La liste est itérée pour chaque fichier est effectue ce qui suit :

- contrôle l'existence d'un doublon avec le même nom
- rajoute le fichier à la variable *filesToSend*
- crée une ligne avec le nom du fichier dans la *box*.
- réhabilite le bouton *Activate*.

```
function validateButton(event) {
```

Valide la liste des fichiers et l'envoie vers le serveur. Cette étape débute le signalement dans client.js en appelant *launchClientSender()*. Désactive aussi le bouton *Validate* et l'importation d'autres fichiers.

```
function resetButton(event) {
```

Active le bouton *Reset* ou *Cancel download* selon le stade du transfert. Réinitialise la liste de fichiers, l'affichage de la *box*, le panneau d'information et active le bouton *Validate*.

```
function setResetButtonLabel(label) {
```

Modifie l'apparence du bouton Reset

```
function setCodeLabel(code, containerID) {
```

Crée un *label* contenant le *code envoyeur* ou *receveur*. Le premier paramètre décrit le code à afficher, tandis que le second identifie l'élément du bon conteneur (de l'envoyeur/receveur) dans *index.html*.

3.2.3 Autres fichiers

dictionary.txt

Importé par une requête *Ajax*, il fournit les mots nécessaires à la construction du code.

StreamSaver.js

Ce script est une libraire que je n'ai pas créée pour le projet, mais qui doit être enregistrée localement pour autoriser l'accès du graveur en streaming au système de fichiers local. Voir référence [StreamSaver.js].

utils.js

Le script **utils.js** catalogue plusieurs fonctions utilisées dans notre projet, mais qui pourraient aussi exister telles quelles pour un autre projet et n'utilisent aucune variable globale issue des autres scripts du client.

Table 7 - Liste des méthodes utilitaires du projet

```
function $(element) {
    return document.getElementById(element);
}
```

Standard pour la sélection d'éléments en javascript

```
function xpath(xpath, context) {
    return document.evaluate(xpath, context, null,
                    XPathResult.singleNodeValue, null).iterateNext();
}
```

Obtient un élément *html* relativement à un autre élément contextualisé.

```
function copyToClipboard(str) {
```

Copie une chaîne de caractères dans le presse-papier.

```
async function asyncSleep(timeMillis) {
```

Cette méthode met en pause l'exécution du programme, de manière asynchrone. Elle est utilisée dans le *FileReader* de **sendFiles.js**

```
function hashToPassphrase(hash) {
```

Convertit une empreinte en passphrase / code de 4 mots. Une requête *Ajax* est d'abord émise vers le serveur pour obtenir le dictionnaire contenant tous les mots. Une fois reçu, il est séparé en éléments distincts dans une *array*. L'empreinte (*hash*) reçue en paramètre est parsée en un nombre hexadécimal. Ensuite un modulo par la taille du dictionnaire (*base*) lui est récursivement appliqué autant de fois qu'il y a de mots dans la passphrase, indiquant ainsi l'index d'un mot à concaténer au code. Le code est finalement retourné.

La construction du code implique des conséquences sur la sécurité de la communication. En effet, bien que les deux pairs soient les auteurs de leur certificat, le serveur ou un autre intermédiaire malveillant sur le réseau pourrait générer un autre certificat dont l'empreinte génère le même code, envoyer sa propre offre / réponse SDP au lieu de relayer celle d'un pair, et

ensuite établir une communication pair-à-pair falsifiée. Nous devons donc calculer la perte de sécurité que la simplification par un code implique.

Le dictionnaire contient 58'110 mots, tandis que l'empreinte de type SHA-256 possède 2^{256} nombres différents. Sachant qu'il y a 4 mots, le dictionnaire propose $58'110^{4}$ nombres différents. Ceci accroît donc le risque de collision des codes par ~ 1.015e+58. Le risque reste néanmoins acceptable pour l'utilisation faite du code, qui devrait être entré en quelques minutes.

En réalité, un hacker voulant créer un certificat qui produit le même code a 58110^{4} possibilités qu'il peut tester par force brute. En admettant qu'il soit équipé pour produire des certificats à la même fréquence que fonctionne un processeur de 3GHz, il lui faudrait 120 ans pour parcourir tous les codes possibles.

3.3 Client installé sur le bureau

L'application installée sur le bureau requiert essentiellement quelques fichiers de configuration supplémentaires. Nous avons vu dans **client.js** qu'un point de connexion avec le serveur devait être enregistré dans notre application. Les seuls autres éléments de programmation concernent le dossier **node_modules** utilisé autant par le serveur que par electron, dans les scripts le fichier package.json et le dossier **electron**, dont les scripts sont pré-générés. Le script **preload.js** existe pour isoler le contexte de lancement de notre application pour des raisons de sécurité. Le script **main.js** nous permet de configurer quelques éléments de base comme la création et la taille de la fenêtre ou le dossier contenant la page racine *index.html*

3.4 Serveur CoTURN

Le serveur CoTURN est lancé sur un Virtual Private Server qui fonctionne sous linux. Le serveur peut être accédé pour configuration en lignes de commandes *bash* avec *openssh* et depuis le répertoire ~/.ssh en écrivant :

```
david@LAPTOP-IDTQVOHN:~/.ssh$ ssh -i "turnkey.ppk" root@51.15.228.16
```

L'option *-i* désigne un fichier pour la clé privée *"turnkey.ppk"*

Cette commande doit être exécutée depuis un système d'opération utilisant linux, le Windows Subsystem for Linux ne suffisant pas pour ce genre d'opérations.

Le serveur VPS a ensuite été configuré pour exécuter le service CoTURN automatiquement à l'aide d'un *démon*, ceci dès l'allumage du serveur VPS.

Nous avons défini les options suivantes dans un fichier */etc/default/coturn* :

```
1  fingerprint
2  user=p2psecurefiletransfer:quickerthancloudtransfer
3  lt-cred-mech
4  realm=kurento.org
5  log-file=/var/log/turnserver/turnserver.log
6  simple-log
7  external-ip=51.15.228.16
```

Code 14 – Options de configuration du serveur TURN

Nous noterons ici quelques propriétés importantes. Le champ *user* définit un nom d'utilisateur et un mot de passe. Le champ *external-ip* est l'adresse IP à laquelle contacter ce serveur.

4 Conclusions

4.1 Achèvement du projet

Nous pouvons estimer que les exigences du projet ont été complètement implémentées avec succès et est qu'il actuellement fonctionnel. Les améliorations encore possibles concernent l'adaptabilité à divers types d'appareils et les fonctions qui sont restées à un stade expérimental utilisant la librairie *socket.io*. En effet, actuellement, deux fonctionnalités ont été désactivées, soit :

1. Le nettoyage des salles inutilisées sur le serveur avec la fonction *cleanUnusedRooms()* dans server.js
2. La reconnexion après un échec lié à un changement ou une perte de réseau avec les fonctions *iceConnectionStateChange_*A et B de sender.js et receiver.js

En effet, la librairie socket.io serait conçue pour faire fonctionner proprement une seule instance de socket à la fois par port et adresse d'un client, dans une idée se rapprochant d'un pattern Singleton. A partir du moment où plus d'une connexion est créée, des défauts empêchent au serveur de répondre à un client avec *socket.emit(...)* et cela de manière passablement aléatoire, ce qui pose un problème conséquent dans le signalement entre pairs.

Pour contrer les déconnexions, WebRTC dispose de son propre mécanisme expérimental *iceRestart*, il est constitué d'un reset (*restartIc*e()) suivi d'une retentative du signalement à partir de l'offre SDP. Malheureusement il s'est révélé plutôt brouillon dans la gestion des événements lors de mes essais et la DataChannel devait tout de même être sauvegardée puisque les données doivent être transmise avec fiabilité (*reliable*). Ce type de redémarrage conviendrait donc au mieux pour un streaming vidéo sur un réseau immuable mais dont la connectivité est peu stable.

4.2 Feedback sur les technologies

Certaines limites propres à WebRTC ont été perçues lors du développement. Notamment, il n'est pas possible de développer un environnement garantissant l'authenticié sur Firefox, car la fonction *getFingerPrint()* des certificats est mal implémentée.

De plus, l'implémentation varie parfois légèrement selon les navigateurs et il faut parfois préciser certains détails. Ces différences peuvent être anodines pour un navigateur particulier mais deviennent perceptibles lorsque les deux pairs utilisent des navigateurs différents. Par exemple, les canaux de données envoient par défaut des objets *Blob* sous Google Chrome et des objets *ArrayBuffer* sous Firefox. Il convient donc de tester les différentes combinaisons de navigateurs possibles.

Ensuite, comme nous l'avons vu avec le code d'authentification, il n'est pas possible de garantir une authentification mutuelle simple uniquement avec WebRTC. Cet outil reste efficace pour garantir l'efficacité d'implémentation en matière de confidentialité, mais n'est pas parfait sur l'intégrité et l'authenticité.

Finalement, la plupart du temps, l'application ne peut pas être utilisée à travers un VPN. Bien que le serveur TURN fonctionne sans problème, le serveur STUN refuse souvent de répondre au fournisseur du service.

4.3 Potentiel de développement

Ce projet a été conçu comme une page web, mais il pourrait aussi être transformé en plugin ou embarqué comme un sous-espace web dans un autre site. Il servirait ainsi de module de transfert dans une autre application.

4.4 Apprentissage personnel

Ce projet m'a permis de découvrir une API qui m'était inconnue auparavant et paraissait compliqué à première vue. L'implémentation s'est finalement révélée relativement simple, mais la documentation, la théorie interminable et les nombreux bugs soulignent l'avantage que constitue l'analyse d'exemples pratiques. De plus, il s'agit de ma première utilisation des web-sockets et l'utilisation de quelques librairies m'a rendu plus confiant en matière de développement web. Finalement, j'ai eu l'occasion de tester la librairie *React,* même si celle-ci ne fait finalement pas partie du projet en raison de la complexité inadaptée du projet pour faire des conversions simples de JSX en HTML.

Je termine donc ce projet avec une grande satisfaction et me réjouis de perfectionner mon apprentissage dans le développement web et logiciel.

Références

[1] Accès au site web du projet : https://www.p2psecurefiletransfer.com/

[2] Accès à l'hôte du site : https://travail-de-bachelor.herokuapp.com/

[3] Code source du projet : https://github.com/davidbaschung/travail-de-bachelor

[4] [Figure 21] A Study of WebRTC Security. https://webrtc-security.github.io/ (accédé le 11 juin 2021)

[5] [Figure 22] Pierre Kuonen & Marcelo Pasin. *Certificats numériques*, cours Info I : Sécurité informatique (f), 2018, UE-EIG.00049

[6] [StreamSaver.js] [Téléchargé le 29 juin 2021] https://github.com/jimmywarting/StreamSaver.js/blob/master/StreamSaver.js

[7] Croc. [Téléchargé le 10 septembre 2020, depuis https://github.com/schollz/croc]

[8] Magic-wormhole. [Téléchargé le 10 septembre 2020, depuis https://github.com/magic-wormhole/magic-wormhole]

[9] GFile. [Téléchargé le 4 novembre 2020, depuis https://github.ycom/Antonito/gfile]

[10] Getting Started With WebRTC. https://christiangomez.me/blog/10/getting-started-with-webrtc/ (accédé le 9 septembre 2020)

[11] Get Started with WebRTC https://www.html5rocks.com/en/tutorials/webrtc/basics/ (accédé le 17 janvier 2021)

[12] Anatomy of a WebRTC SDP. https://webrtchacks.com/sdp-anatomy/ (accédé le 9 septembre 2020)

[13] ShareDrop P2P file transfer. https://www.sharedrop.io/ (accédé le 9 septembre 2020)

[14] Practical WebRTC A Complete WebRTC Bootcamp for Beginners. [Téléchargé le 27 décembre 2020, depuis https://www.udemy.com/course/practical-webrtc-a-complete-webrtc-bootcamp-for-beginners/]

[15] Introduction à l'architecture WebRTC. https://developer.mozilla.org/fr/docs/Web/API/WebRTC_API/Connectivity (accédé le 11 novembre 2020)

[16] WebRTC 1.0: Real-Time Communication Between Browsers. https://www.w3.org/TR/webrtc/ (accédé le 28 janvier 2021)

[17] L'API WebRTC. https://developer.mozilla.org/fr/docs/Web/API/WebRTC_API (accédé le 12 janvier 2021)

[18] WebRTC 1.0: Real-Time Communication Between Browsers https://w3c.github.io/webrtc-pc/#intro (accédé le 13 mars 2021)

[19] Premiers pas avec WebRTC. https://webrtc.org/getting-started/overview (accédé le 15 octobre 2020)

[20] WebRTC, Browser apis and Protocols. https://hpbn.co/webrtc/ (accédé le 4 mars 2021)

[21] WebRTC For The Curious. https://webrtcforthecurious.com/ (accédé le 28 décembre 2020)

[22] Communication en temps réel avec WebRTC. https://codelabs.developers.google.com/codelabs/webrtc-web#5 (accédé le 28 février 2021)

[23] WebRTC samples https://webrtc.github.io/samples/ (accédé le 17 janvier 2021)

[24] Transfer a file. https://webrtc.github.io/samples/src/content/datachannel/filetransfer/ (accédé le 17 janvier 2021)

[25] Generate and transfer data. https://webrtc.github.io/samples/src/content/datachannel/datatransfer/ (accédé le 17 janvier 2021)

[26] Trickle ICE. https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/ (accédé le 22 avril 2021)

[27] What is WebRTC and How to Setup STUN/TURN Server for WebRTC Communication? https://medium.com/av-transcode/what-is-webrtc-and-how-to-setup-stun-turn-server-for-webrtc-communication-63314728b9d0 (accédé le 22 avril 2021)

[28] How to Use SSL/TLS with Node.js. https://www.sitepoint.com/how-to-use-ssltls-with-node-js/ (accédé le 14 février 2021)

[29] How to setup Express.js in Node.js. https://www.robinwieruch.de/node-js-express-tutorial (accédé le 11 février 2021)

[30] Authentication using HTTPS client certificates. .https://medium.com/@sevcsik/authentication-using-https-client-certificates-3c9d270e8326 (accédé le 18 février 2021)

[31] Certificates for localhost. https://letsencrypt.org/docs/certificates-for-localhost/ (accédé le 19 février 2021)

[32] Forge [Téléchargé le 19 février depuis https://github.com/digitalbazaar/forge]

[33] Configuring Your Node.js Server for Mutual TLS https://smallstep.com/hello-mtls/doc/server/nodejs (accédé le 19 février 2021)

[34] Everything you need to know about WebRTC security. https://bloggeek.me/is-webrtc-safe/ (accédé le 12 avril 2021)

[35] The magic of TLS, X509 and mutual authentication explained. https://medium.com/littlemanco/the-magic-of-tls-x509-and-mutual-authentication-explained-b2162dec4401 (accédé le 17 février 2021)

[36] Use of Fingerprints for Identifying Certificates in the Session Description. Protocol (SDP) https://tools.ietf.org/id/draft-thomson-mmusic-fingerprint-00.html (accédé le 12 avril 2021)

[37] WebRTC mutual authentication using certificates. https://stackoverflow.com/questions/60922417/webrtc-mutual-authentication-using-certificates (accédé le 8 avril 2021)

[38] What you really need to know about securing APIs with mutual certificates. https://nevatech.com/blog/post/What-you-need-to-know-about-securing-APIs-with-mutual-certificates (accédé le 17 février 2021)

[39] How to derive a passphrase from a hash? https://crypto.stackexchange.com/questions/80228/how-to-derive-a-passphrase-from-a-hash/80269#80269 (accédé le 14 avril 2021)

[40] 5 Ways To Create & Save Files In Javascript – Simple Examples. https://code-boxx.com/create-save-files-javascript/ (accédé le 20 février 2021)

[41] How to save large files in JavaScript that require client-side processing https://stackoverflow.com/questions/41206733/how-to-save-large-files-in-javascript-that-require-client-side-processing (accédé le 14 avril 2021)

[42] Electron – Démarrage rapide. https://www.electronjs.org/docs/tutorial/quick-start (accédé le 5 mai 2021)

[43] How to detect Safari, Chrome, IE, Firefox and Opera browser? https://stackoverflow.com/questions/9847580/how-to-detect-safari-chrome-ie-firefox-and-opera-browser (accédé le 13 mai 2021)

[44] How to Drag & Drop HTML Elements and Files using Javascript. https://ralzohairi.medium.com/how-to-drag-drop-html-elements-and-files-using-javascript-d31d15279369 (accédé le 20 janvier 2021)

[45] HTML5, JavaScript: Drag and Drop File from External Window (Windows Explorer). https://stackoverflow.com/questions/10261989/html5-javascript-drag-and-drop-file-from-external-window-windows-explorer (accédé le 20 janvier 2021)

[46] File drag and drop. https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API/File_drag_and_drop (accédé le 20 janvier 2021)

[47] Saving a blob https://jimmywarting.github.io/StreamSaver.js/examples/saving-a-blob.html (accédé le 30 juin 2021)