Rapport de stage

Conception et réalisation d'une plateforme web d'édition collaborative pair-à-pair

Laporte-Chabasse Quentin

Année 2014–2015

Stage de deuxième année réalisé dans le laboratoire LORIA - équipe COAST

en vue du passage en troisième année à TELECOM Nancy

Maître de stage : Oster Gerald

Encadrant universitaire : Badonnel Rémi

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Laporte-Chabasse, Quentin

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 2014041022

Année universitaire : 2014–2015

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Conception et réalisation d'une plateforme web d'édition collaborative pair-à-pair temps réel sécurisée

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy, le 24 août 2015

Signature :

Rapport de stage

Conception et réalisation d'une plateforme web d'édition collaborative pair-à-pair

Laporte-Chabasse Quentin

Année 2014–2015

Stage de deuxième année réalisé dans le laboratoire LORIA - équipe COAST

en vue du passage en troisième année à TELECOM Nancy

Laporte-Chabasse Quentin
30 rue Fabert
54600, Villers-lès-Nancy
+33 (0)6 09 33 22 85
quentin.laporte-chabasse@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

LORIA - équipe COAST
615 rue du Jardin botanique
54600 Villers-lès-Nancy
+33 (0)3 83 58 17 50

Maître de stage : Oster Gerald

Encadrant universitaire : Badonnel Rémi

Remerciements

Je tiens à remercier l'ensemble de l'équipe COAST pour son accueil et l'aide précieuse qu'ils ont pu m'apporter tout au long de mon stage. Je remercie tout particulièrement Matthieu NICOLAS, Gerald OSTER et Luc ANDRÉ qui ont su m'intégrer au projet et m'apporter leur soutien.

Table des matières

1 Introduction

L'avènement du web 2.0 a permis de dynamiser les sites web et d'accroitre l'interactivité avec l'utilisateur. L'internaute n'est plus seulement spectateur, mais aussi acteur du web. Le web n'est donc plus seulement un média, mais une plateforme d'échange de l'information. Une dimension collaborative sous forme de blog, wiki, réseaux sociaux, s'est ainsi ajoutée au web traditionnel.

La multiplication des périphériques connectés tels que les tablettes et les smartphones a permis de démocratiser l'accès au web. Les applications "Desktops" principalement utilisées sur les ordinateurs n'étaient alors plus suffisantes pour répondre aux contraintes d'un environnement numérique pluriel. Le partage des différents contenus multimédia avec tous ces appareils se révélait être une nécessité.L'augmentation de la puissance des serveurs hébergeant ces applications web 2.0 et le développement de nouvelles technologies côté client, ont permis progressivement l'apparition d'applications plus complexes étant susceptibles de remplacer ces applications "Desktops". De nouvelles suites bureautiques online ont alors vu le jour, s'inspirant des applications "Desktops" et y incorporant la dimension collaborative inspirée par le web 2.0.

L'exemple le plus marquant étant bien évidemment celui de Google qui propose une suite d'applications : Google Docs, Google Sheets, Google Slides et Google Forms. Toutes ces applications permettent de travailler en collaboration avec plusieurs acteurs sur un même document. De plus, elles fournissent une interface utilisateur similaire à celles proposées par les applications "Desktops". L'utilisateur n'est alors pas déstabilisé et retrouve aisément ses repères. La figure 1.1 présente ainsi l'interface de Google Doc, une application permettant de faire du traitement de texte en édition collaborative.

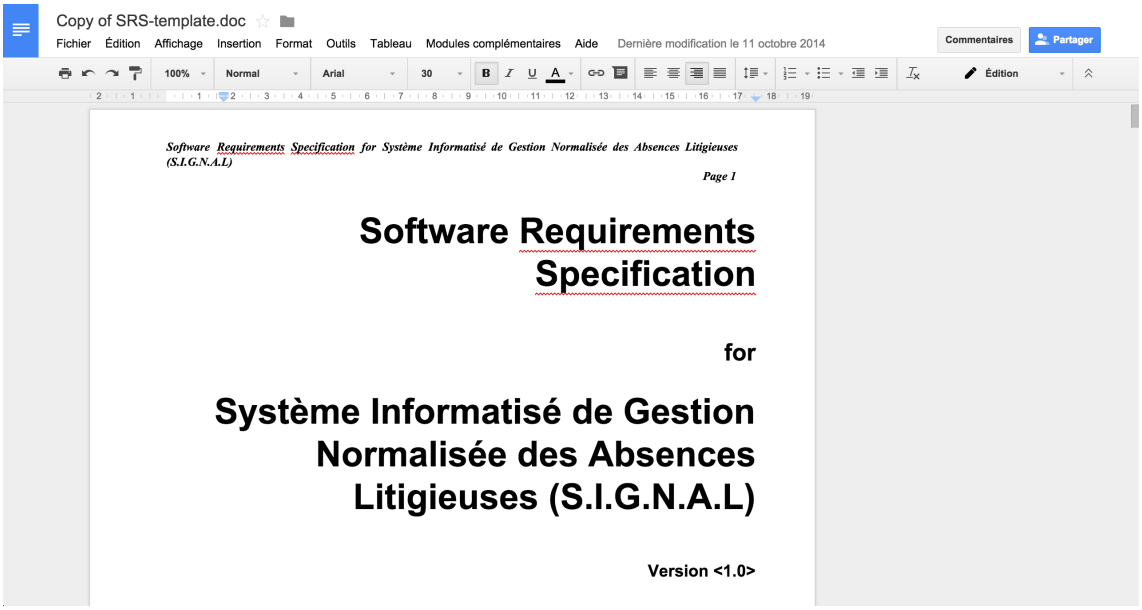

FIGURE 1.1 – Interface de l'application Google Doc

Toutes ces applications ont permis de rendre accessible le travail collaboratif. La présence physique de tous les contributeurs dans un même lieu n'est alors plus requise. Le travail sur des projets interdisciplinaires s'en trouve alors plus aisé. Chaque contributeur au projet peut ainsi composer tout en visualisant en temps réel la contribution des autres protagonistes. Cet aspect temps réel est primordial dans le domaine de l'édition collaborative, sans cela les différents contributeurs du projet ne peuvent constater l'évolution immédiate du document. Ce qui pourrait conduire à d'importantes erreurs dans la réalisation de ce dernier. L'application se doit donc d'être réactive pour répercuter le plus rapidement possible les différentes modifications du document. Se pose ainsi le problème du passage à l'échelle : si l'application offre des temps de réaction satisfaisants pour un nombre réduit de collaborateurs, en est-il de même pour un nombre plus important ?

L'architecture client/serveur très largement utilisée pour ce type d'application peut montrer certaines faiblesses en période de montée en charge. En effet, le serveur devant gérer les communications entre tous les clients et maintenir une copie cohérente du document peut se retrouver débordé si le nombre de collaborateurs augmente de manière importante. Les performances de l'application se verraient alors dégradées, et l'expérience utilisateur entachée par des temps de latence importants. Un des moyens de répondre à cette problématique est de développer un réseau pair-à-pair entre tous les clients d'un même document. Les données ne transiteraient plus par le serveur, mais seraient directement émises d'un client vers les autres clients. Cette communication directe entre chaque client permet de conserver des performances correctes même en cas de montée en charge.

J'ai ainsi pour mission d'adapter un éditeur de texte collaboratif nommé MUTE [1], de façon à y incorporer un mécanisme d'échange pair-à-pair. Je dois pour cela utiliser la technologie webRTC, qui permet d'établir des connexions pair-à-pair entre deux clients par le biais du navigateur web. Mon travail consiste donc à choisir les outils de développement les plus adaptés pour utiliser la technologie webRTC et de développer un module de communication pair-à-pair pour l'application MUTE.

1. Pour : Multi-User Text Editor, l'acronyme MUTE sera utilisé tout au long du rapport

2 Présentation de l'entreprise

2.1 Présentation du Loria

J'ai effectué mon stage au sein de l'équipe COAST qui appartient au LORIA, Laboratoire Lorrain de Recherche en Informatique et ses Applications. Ce Laboratoire situé sur le campus scientifique de l'Université de Lorraine a été fondé en 1997, et ce, dans le but de mener des investigations dans le domaine de la recherche fondamentale et appliquée en sciences informatiques. Cette unité mixte regroupant plusieurs établissements, à savoir : l'Inria, le CNRS et l'Université de Lorraine est composée de 30 équipes regroupées en cinq départements.

Chaque département à sa propre thématique de recherche :

1. Alogrithme, calcul, image et géométrie
2. Méthodes Formelles
3. Réseaux, Systèmes et Services
4. Traitement des langues et des connaissances
5. Systèmes complexes et intelligence artificielle

Néanmois, pour les besoins de la recherche, il n'est pas rare que des collaborations s'établissent entre des équipes de différents départements. De plus, le Loria conserve un lien fort avec le monde de l'industrie au travers de collaborations avec des industrielles à l'échelle nationale ou internationale.

La figure 2.1 représente ainsi l'organisation du LORIA

2.2 Présentation de l'équipe COAST

J'ai ainsi intégré l'équipe COAST, dirigée par François CHAROY. L'équipe est composée de 23 membres, dont 11 permanents. Les recherches de cette équipe se concentrent autour du développement de services pour l'hébergement d'équipes distribuées sur internet. Ces services sont variés et s'articulent principalement autour de la co-conception et de la co-ingéieurie.

Le travail de l'équipe s'organise plus particulièrement autour de trois axes de recherche :

— Systèmes collaboratifs distribués
— Gestion des processus "business" et service informatique
— Interopérabilité et modélisation d'entreprise

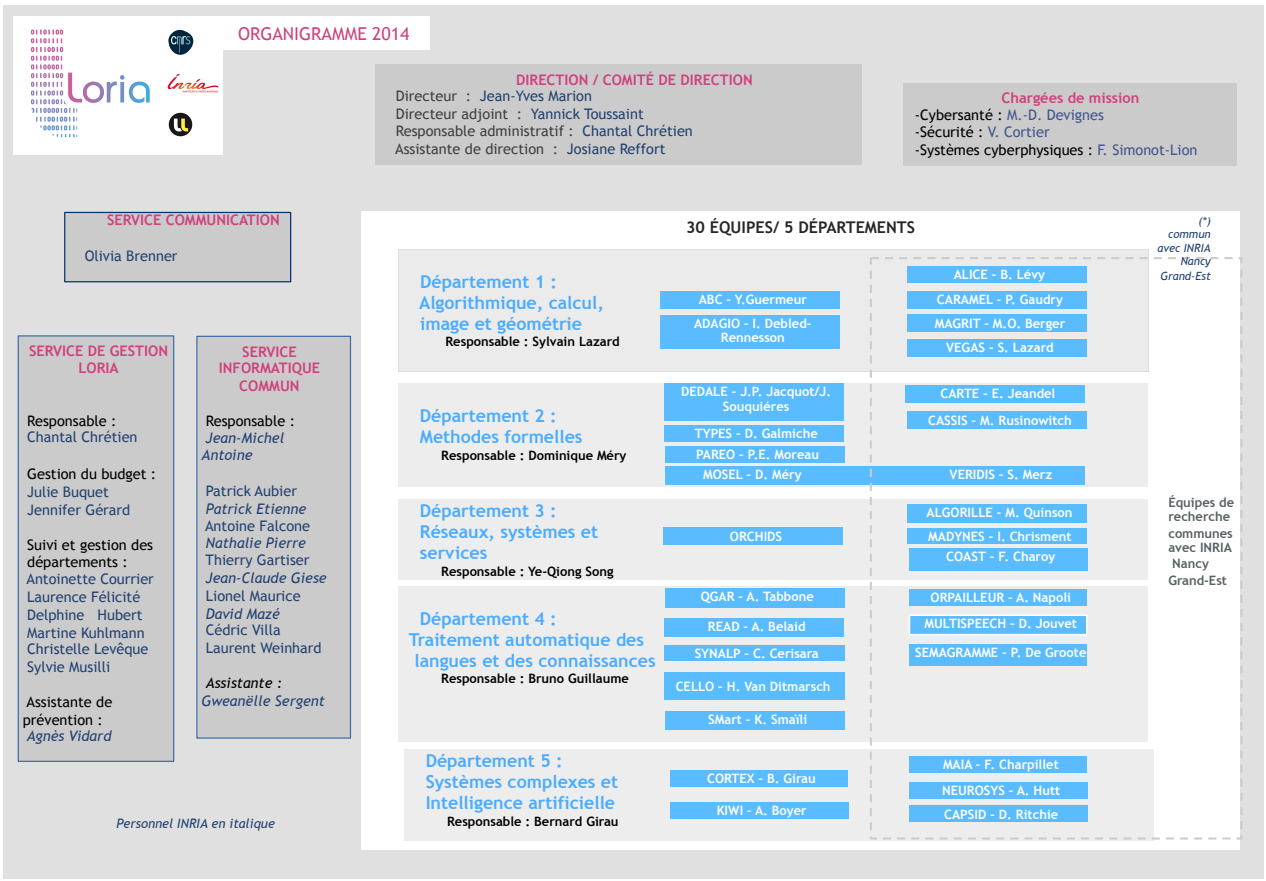

FIGURE 2.1 – Organisation du Loria

Mon travail au sein de l'équipe s'intégrait principalement dans le domaine des systèmes collaboratifs distribués. J'ai travaillé sous la tutelle de Gerald OSTER, enseignant chercheur au LORIA et à TELECOM Nancy. Ses recherches portent sur la réplication optimiste et cohérente dans les environnements distribués collaboratifs. Par ailleurs, j'ai travaillé en collaboration avec Mathieu NICOLAS, ingénieur de recherche dans l'équipe VERIDIS, qui a développé l'éditeur collaboratif sur lequel j'ai été amené à travailler.

3 Etat de l'art

3.1 Présentation de MUTE

L'application MUTE est un éditeur de texte collaboratif développé par Matthieu NICOLAS dans le cadre de la thèse de Luc ANDRÉ. La Figure 3.1 représente l'interface proposée par l'application. Il se démarque des autres éditeurs par l'algorithme employé pour le traitement des opérations texte [1]. En effet, il existe deux grandes familles d'algorithme d'édition collaborative adoptant chacune une approche différente.

L'approche la plus commune se nomme l'approche OT [2], elle est employée dans de nombreux éditeurs de texte collaboratif, le plus connu étant Google Doc. Pour gérer les opérations concurrentes, l'algorithme utilise la transformée opérationnelle, qui résout les conflits en transformant une opération en fonction des autres opérations concurrentes. Cette méthode est relativement couteuse et n'est pas adaptée à la solution pair-à-pair choisie.

Une seconde approche plus appropriée consiste à introduire une nouvelle structure de données permettant l'application des opérations texte concurrentes de façons désordonnées. L'ordre d'arrivée des opérations n'a donc plus d'importance dans la résolution des conflits. Cette approche se nomme approche CRDT, pour Conflict-free Replicated Data Type. L'application MUTE utilise l'algorithme LogootSplit qui est issu de cette famille. Il a été proposé par Luc ANDRÉ dans le cadre de sa thèse.

3.1.1 Architecture et fonctionnement de MUTE

L'application MUTE adopte initialement une architecture client/serveur. Tous les collaborateurs d'un même document sont connectés à un serveur qui se charge de relayer les différentes opérations texte et de maintenir une copie du document. Chaque client possède une copie du document ainsi que des informations relatives à l'utilisateur, comme son nom et la position de son curseur. Il possède également ces informations pour les autres collaborateurs du document. Tous les clients sont connectés au serveur au travers d'une webSocket [3]. Comme le montre la Figure 3.2, le serveur est l'élément central de toutes les communications. Il conserve une trace de tous les documents et associe à chaque document les contributeurs de ce dernier et une copie du document. De cette façon, lorsqu'un nouveau client se connecte, ce dernier a la possibilité de récupérer la dernière copie du document en cours d'édition.

1. Une opération texte correspond à l'ajout ou la suppression d'un caractère dans le document
2. Signifiant : Transformée Opérationnelles
3. Connexion client/serveur persistante

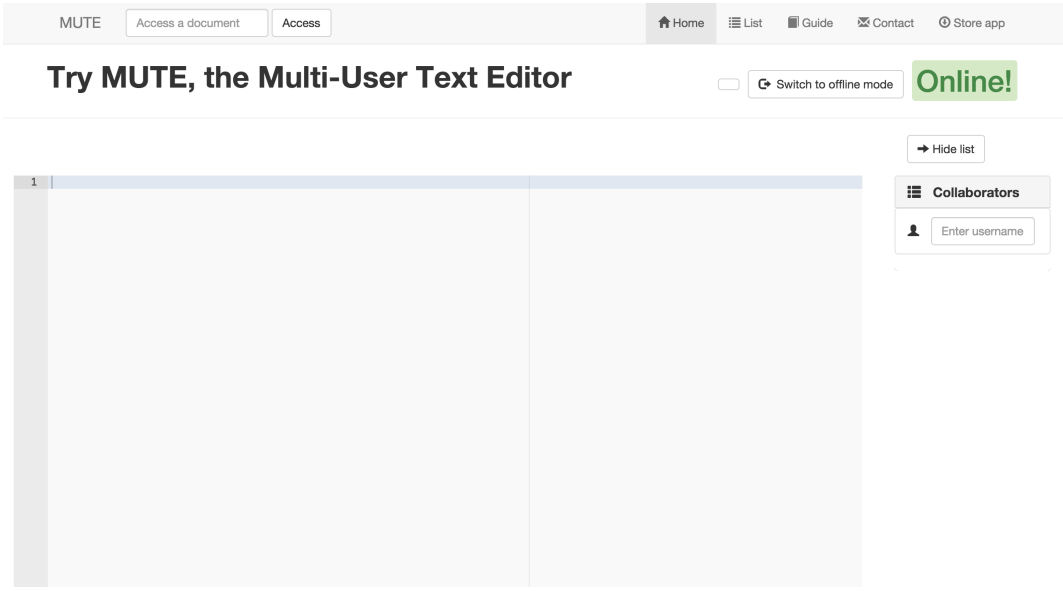

FIGURE 3.1 – Interface de l'application MUTE

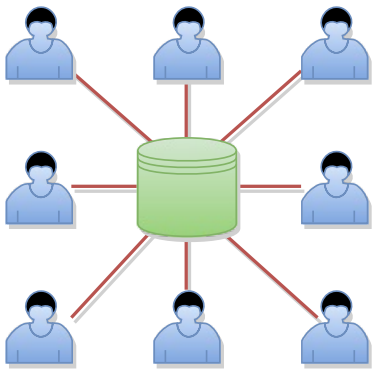

FIGURE 3.2 – Architecture client/serveur - le serveur au centre de toutes les communications

Architecture du client

Le client est composé de quatre modules :

- Ace-editor [4] : librairie utilisée pour créer l'éditeur de texte qui apparait à l'écran du client
- InfoUser : module permettant de gérer les informations de l'utilisateur et des autres contributeurs
- Coordinator : module permettant de gérer le modèle [5] et utilisant l'algorithme LogootSplit pour l'application des opérations texte
- Socket-io-adapter : module en charge de la communication avec le serveur

L'organisation de ces modules est décrite dans la Figure 3.3 Chaque collaborateur est identifié de manière unique grâce à un *numéro de site* attribué par le serveur.

4. Pour plus d'information : http://ace.c9.io/
5. Sous la forme de structures LogootSplit

Architecture du serveur

L'architecture du serveur décrite par la Figure 3.3 est semblable à celle du client, à la différence qu'elle n'utilise pas Ace-editor. Elle se compose donc de trois modules :

- InfoUser : module permettant de gérer les informations de tous les utilisateurs d'un document
- Coordinator : module permettant de gérer le modèle et utilisant l'algorithme LogootSplit pour l'application des opérations texte
- Socket-io-adapter : module en charge de la communication avec les clients

Fonctionnement de MUTE

La Figure 3.3 représente le fonctionnement de l'architecture MUTE. Deux clients et le serveur y sont représentés, un client est en train de rédiger, pendant que le second visualise les modifications. Le cheminement des données est représenté par des flèches et chaque étape est décrite dans les encadrés bleus.

Lorsqu'un client écrit dans l'éditeur, le module Ace-editor génère des opérations texte qui vont être transmises au coordinateur [6]. Le coordinateur va en suite générer les opérations Logoot pour les opérateurs distants [7] et va transmettre ces opérations au module socket-io-adapter. Les opérations vont être ainsi transmises au serveur.

À la réception des opérations Logoot par le serveur, ce dernier va transmettre ces opérations à son coordinateur et dans le même temps, les enverra aux autres contributeurs du document. Le coordinateur du serveur va appliquer les opérations Logoot de façon à conserver une copie à jour du document. Les autres contributeurs vont faire de même et vont répercuter les modifications sur leur modèle local. Une fois les opérations appliquées, la vue de l'éditeur va se mettre à jour pour que les autres utilisateurs puissent visualiser les modifications.

Avantages et limites de cette architecture

Cette architecture est adaptée pour le développement rapide d'un prototype fonctionnel. Elle permet une gestion centralisée des documents et les nombreuses librairies disponibles permettent de gérer de façon fiable les connexions client-serveur. Cependant cette architecture rencontre des problèmes de passage à l'échelle. En effet, le serveur devant gérer l'ensemble des documents et toutes les communications avec les clients peut se retrouver rapidement débordé si le nombre de documents et/ou de collaborateurs venait à augmenter. Une solution pourrait être d'augmenter les moyens matériels, à savoir la puissance des serveurs, cependant cette solution est relativement couteuse et non viable sur le long terme.

Il est alors nécessaire de s'affranchir de cette architecture centralisée du moins pour les échanges d'opérations texte, de façon à soulager le serveur. Une solution est donc de créer des connexions directes entre les clients. Une technologie nommée webRTC permet de créer des connexions pair-à-pair entre clients par le biais d'un navigateur web. Cette solution a été retenue pour répondre au problème de scalabilité du système.

6. Ou coordinator en anglais
7. Les opérateurs distants correspondent aux autres clients et au serveur

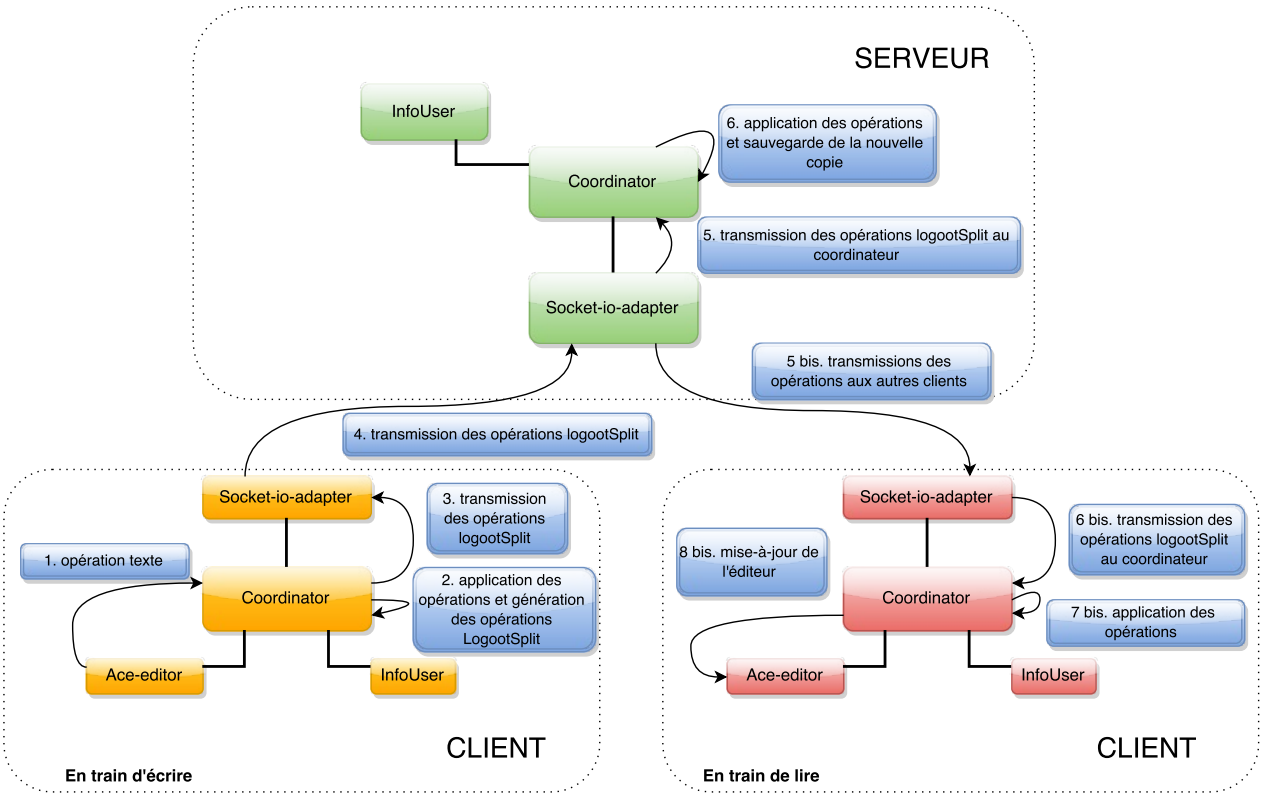

FIGURE 3.3 – Architecture et fonctionnement de l'application MUTE

3.2 Présentation des technologies webRTC

WebRTC pour *Real Time Communication* est un standard de communication élaboré par le W3C [8]. Ce standard encore à l'étude aujourd'hui a été initié en mai 2011 et a pour principal objectif de permettre la communication en temps réel de différents médias [9]. WebRTC doit donc permettre d'établir une connexion pair-à-pair entre deux clients web, mais doit également permettre la gestion des flux de données.

Pour ce faire, webRTC propose l'implémentation de trois APIs JavaScript :

- MediaStream : En charge de synchroniser les flux de données pouvant par exemple provenir de la webcam ou du micro de l'ordinateur
- RTCPeerConnection : En charge de gérer la bande passante et le chiffrement des données
- RTCDataChannel : En charge de gérer la connexion pair-à-pair entre deux clients

Tous les navigateurs n'implémentent pas encore ces APIs, la technologie webRTC est principalement disponible sur les dernières versions de :

- Mozilla Firefox
- Google Chrome
- Opera
- Android Browser et Chrome for Android

En somme, grâce à ces APIs, il est possible d'utiliser les flux de données provenant de la webcam ou du micro et de le diffuser via une connexion pair-à-pair à d'autres clients. Cette technologie a

8. RFC de webRTC : http://www.w3.org/TR/webrtc/
9. À savoir : vidéo, audio et tout autre type de données

permis le développement de nombreuses applications de messagerie instantanée et de visioconférence. Le dernier exemple connu se nomme Firefox Hello [10].

3.2.1 Les mécanismes réseau utilisés

Outre la gestion des flux de média, certains mécanismes ont été introduits au niveau du réseau permettant l'établissement d'une connexion pair-à-pair entre deux clients distants, ainsi que la résolution de certaines difficultés induites par les pare-feu [11] et par les NAT [12].

Serveur de signaling

Une des premières difficultés à laquelle il est nécessaire de répondre est l'établissement d'une connexion pair à paire entre deux clients distants. Il est en effet nécessaire d'avoir un mécanisme de contrôle permettant d'initialiser la connexion, d'échanger les IPs et les ports applicatifs de chaque machine et d'assurer que les deux clients soient bien compatibles. Un serveur de signaling se charge donc de l'échange de ces différents messages de contrôle.

Ainsi, quand un client souhaite entamer une connexion pair-à-pair avec un autre client, il va tout d'abord contacter le serveur de signaling. Une fois le second client disposé à recevoir la demande, le serveur va la lui transmettre. Une succession d'échange va alors s'effectuer pour que la connexion pair-à-pair puisse s'établir correctement.

Trois types d'information sont échangés par l'intermédiaire du serveur de signaling :
- Des messages de contrôle, pour l'initialisation de la connexion et la gestion des erreurs
- Des messages liés à la configuration du réseau [13]
- Des messages liés à la compatibilité des médias [14]

Une fois ces différentes informations échangées et la compatibilité assurée, la connexion pair-à-pair peut être établie comme le montre la Figure 3.4

Il est important de préciser que le serveur de signaling ne fait pas l'objet de spécification dans le standard webRTC. Les développeurs ne sont donc pas contraints ni dans le choix d'un protocole de communication ni dans le choix d'une technologie pour la communication client/serveur.

Serveur STUN et TURN

Pour créer une connexion pair-à-pair, il est également nécessaire d'obtenir une adresse IP publique et un numéro de port sur lequel se connecter. De nombreux éléments intermédiaires comme les NAT peuvent empêcher l'obtention de ces informations. Pour pallier à ce problème webRTC spécifie l'utilisation de serveurs STUN [15] utilisant le protocole ICE [16]. Ce protocole [17]

10. Pour plus d'information : https://www.mozilla.org/fr/firefox/hello/
11. Ou firewall en anglais
12. Signifiant : Network Address Translation
13. Adressage IP et numéro de port
14. Codecs vidéo et résolution
15. Pour : Simple Traversal of UDP through NATs
16. Pour : Interactive Connectivity Establishement
17. RFC de ICE : https://tools.ietf.org/html/rfc5245

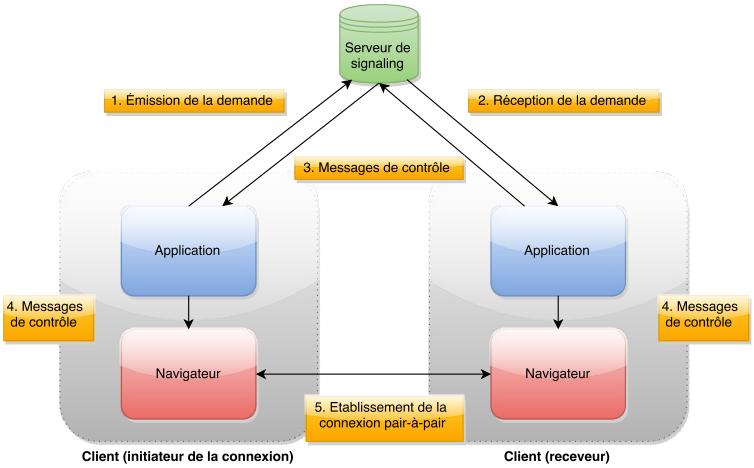

FIGURE 3.4 – Mécanisme de "signaling" pour l'initialisation d'une connexion pair-à-pair *(inspiré de la référence bibliographique [2])*

permet donc de répurérer une adresse IP publique et un numéro de port même en cas de présence de NAT.

La Figure 3.5 représente de cette manière la place des serveurs STUN dans le contexte d'une connexion pair-à-pair. Le nuage aux contours bleus représente le serveur de signaling qui se charge de l'échange des messages de contrôle (notamment des adresses publiques), tandis que les serveurs STUN attribuent à chaque client une adresse IP publique est un numéro de port.

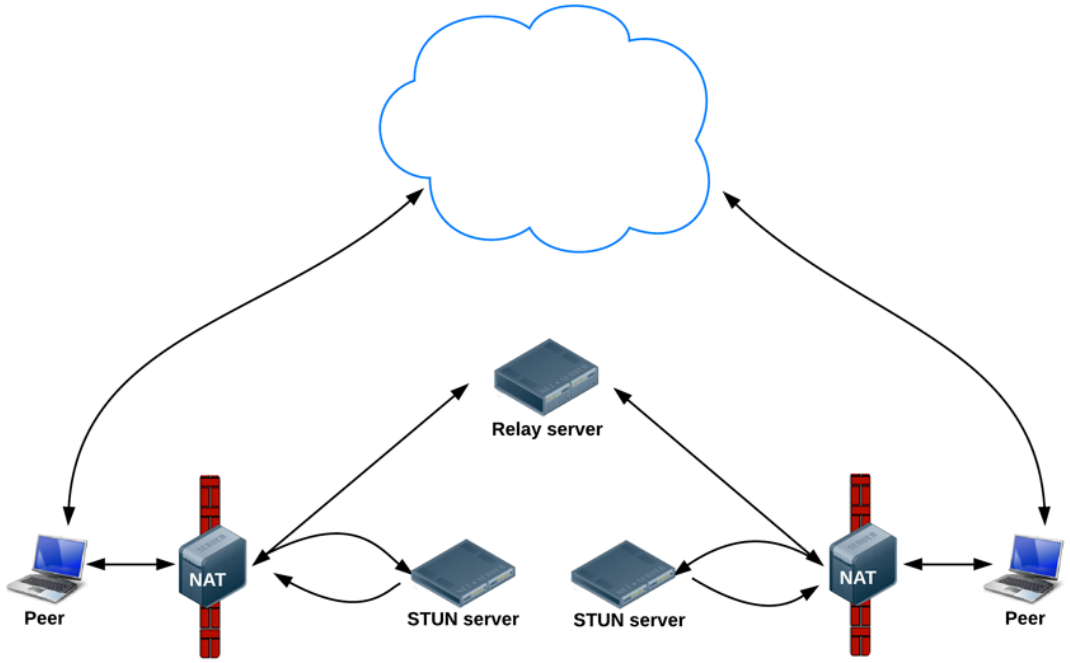

FIGURE 3.5 – Établissement d'une connexion avec un serveur STUN *(figure issue de la référence bibliographique [2])*

Le problème d'attribution d'adresses publiques étant résolu, les pairs vont alors tenter de s'interconnecter directement. La présence d'un serveur proxy ou d'un pare-feu peut cependant poser problème et empêcher l'établissement de la connexion. Dans ce cas-là, l'information doit transiter par un serveur TURN [18]. La Figure 3.6 représente la place occupée par un serveur TURN dans

18. Pour : Traversal Using Relays around NAT

le contexte d'une connexion pair-à-pair bloquée parla présence d'un pare-feu ou d'un serveur proxy. Les serveurs TURN jouent alors le rôle de relais.

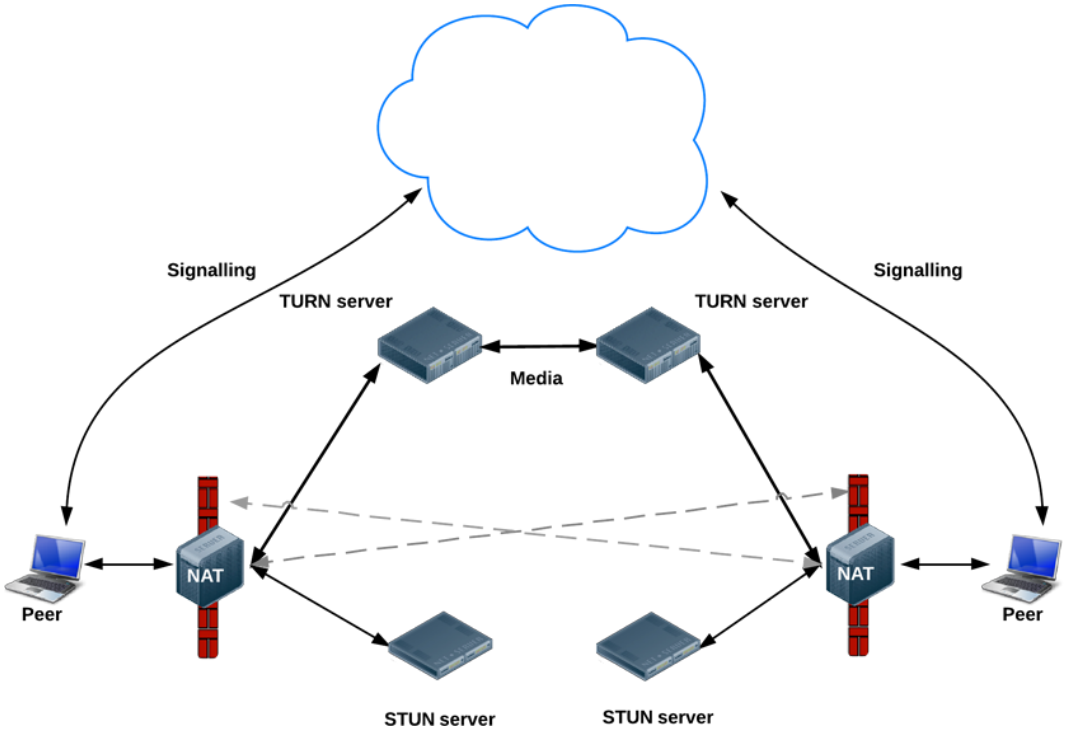

FIGURE 3.6 – Fonctionnement d'un serveur TURN *(figure issue de la référence bibliographique [2])*

Ces différents mécanismes permettent de répondre aux contraintes imposées par internet et garantissent l'établissement d'une connexion RTC.

4 Présentation du travail réalisée

J'ai pour mission d'intégrer un module de communication pair-à-pair dans l'application MUTE. Mon travail consiste donc à comprendre et appréhender l'architecture de MUTE, d'étudier et de choisir la librairie la plus adaptée pour utiliser la technologie webRTC et enfin de développer un module de communication pair-à-pair opérationnel.

4.1 Présentation du contexte et de la problématique détaillée

MUTE est une application développée en JavaScript, l'application côté serveur a été développée en NodeJS, l'application côté client utilise certains modules NodeJS, mais également des librairies JavaScript externes [1].

Le module de communication pair-à-pair sera intégré au même niveau que le module de communication client/serveur [2]. Les fonctionnalités initialement proposées seront conservées et l'utilisateur pourra choisir ou non d'utiliser la communication pair-à-pair. L'objectif n'est donc pas de modifier l'existant, mais d'adapter le module de communication pair-à-pair aux mécanismes déjà utilisés. Dans le cas où des modifications venaient à être effectuées sur les modules existants, elles ne devraient en rien altérer le comportement des fonctionnalités existantes.

L'objectif est d'aboutir à une topologie identique à celle décrite à la Figure 4.1. On remarque ainsi que tous les contributeurs sont interconnectés. De plus, une connexion persistante est établie entre le serveur et les clients de façon à ce que le serveur ait connaissance des contributeurs, et ce pour chaque document.

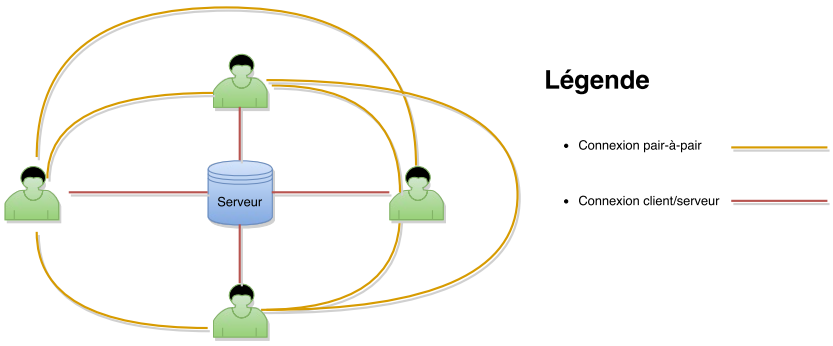

FIGURE 4.1 – Topologie du réseau pour un document avec quatre contributeurs

1. JQuey, Ace-editor , etc.
2. Socket-io-adapter

4.2 Réalisation d'un système de communication pair-à-pair pour l'outil MUTE

La réalisation du système de communication pair-à-pair se décompose en deux étapes :

1. Étude, recherche et validation d'une librairie exploitant la technologie webRTC
2. Développement du module de communication paire à pair

À la suite de cela l'ensemble des fonctionnalités proposées par MUTE ont été testées et validées.

4.2.1 Choix de la librairie et développement d'un prototype

Le choix de librairie est crucial pour la suite du développement, elle doit être en mesure de répondre aux contraintes imposées par le projet. Dans le cadre du projet, il est question de développer un prototype fonctionnel, maintenable et évolutif.

La librairie doit donc respecter certaines contraintes, à savoir :

— Offrir un niveau d'abstraction suffisant
— Proposer un support et une communauté de développeurs active [3]
— Proposer une documentation claire et complète sur l'utilisation de son API

De cette manière, un niveau d'abstraction suffisant permet de concentrer le développement autour de la réalisation du module et non sur des mécanismes plus «bas-niveau». Une communauté active permet quant à elle de proposer un support et une aide sur les bugs et les problèmes rencontrés. Enfin, une documentation claire et concise permet d'appréhender plus facilement et plus rapidement la librairie.

Mes recherches ont retenu quatre librairies exploitant la technologie webRTC :

— SimpleWebRTC [4]
— y-webrtc [5]
— rtc-scamp [6]
— PeerJS [7]

En m'appuyant sur la documentation de l'API, l'activité des dépôts et le nombre de tickets encore ouverts, j'ai pu établir un comparatif. Ce dernier décrit par la Figure 4.2 m'a ainsi permis de sélectionner la librairie qui à mon sens était la plus adaptée. La librairie PeerJS m'a donc semblé être un choix judicieux, répondant aux besoins du projet. C'est la seule librairie qui offre en plus de son API, la possibilité d'utiliser un serveur de signaling distant.

Le fonctionnement de l'API est relativement simple. Chaque client se voit attribuer par le serveur de signaling un identifiant unique, nommé peerId. Ainsi, si un client prend connaissance du peerId d'un autre pair, il pourra entamer une connexion avec lui. Ce peerId est donc un moyen d'identifier de manière unique les pairs, mais également de se connecter avec un pair distant. L'API est

3. Concernant la correction des bugs et l'ajout de nouvelles fonctionnalités
4. Pour plus d'information : https://simplewebrtc.com/
5. Pour plus d'information : https://github.com/y-js/y-webrtc
6. Pour plus d'information : https://github.com/Chat-Wane/rtc-SCAMP
7. Pour plus d'information : http://peerjs.com/

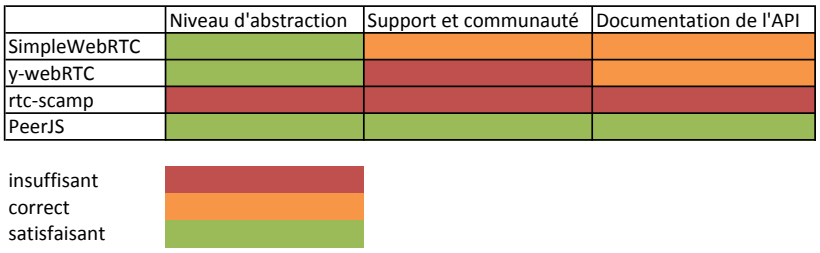

FIGURE 4.2 – Comparatif des librairies étudiées

en JavaScript et repose donc sur de la programmation événementielle. Il est ainsi possible d'intercepter des évènements comme l'ouverture ou la fermeture d'une connexion et la réception de données.

Avant d'entamer le développement du module, j'ai voulu valider mon choix en développant un prototype simple. J'ai donc développé une application NodeJS où chaque client envoie à intervalle de temps réguliers des messages aux autres pairs [8] avec qui, il est connecté. Les messages contiennent le peerId du pair émetteur, l'heure d'envoi et le numéro du message. La Figure 4.3 représente l'interface du client, cette dernière permet de visualiser le peerId du client, la liste des messages reçus et la liste des autres pairs avec qui le client est connecté.

Ce prototype m'a permis de tester et d'utiliser les fonctionnalités d'envoi et de réception des messages. J'ai également pu m'assurer que l'échange des messages respectait une certaine forme de causalité, l'ordre d'arrivée étant similaire à l'ordre d'émission.

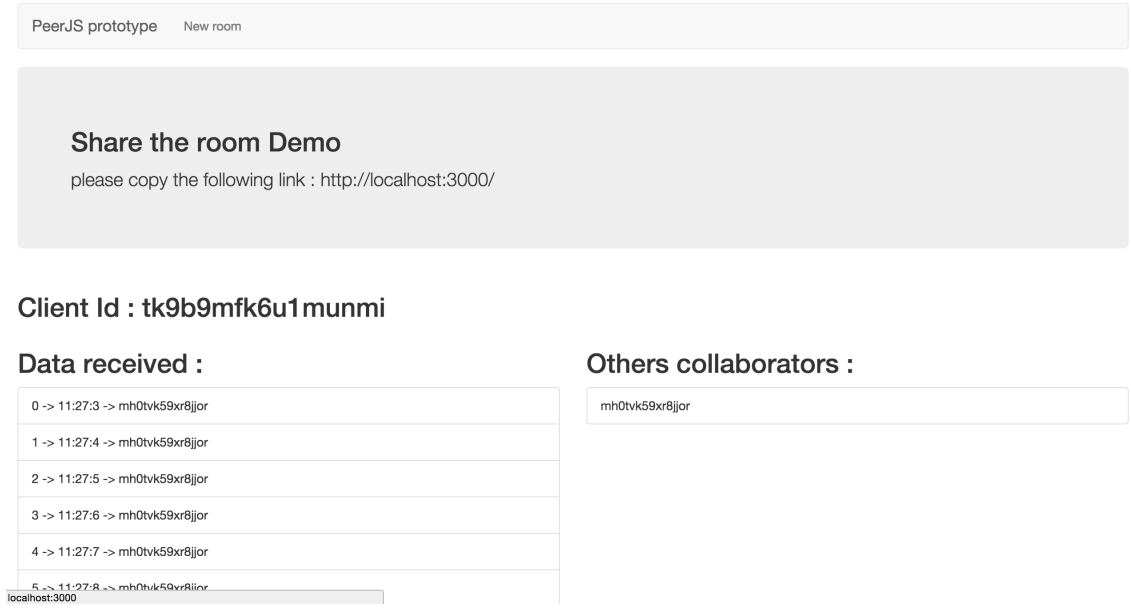

FIGURE 4.3 – Interface du prototype

4.2.2 Implémentation du système pair-à-pair dans MUTE

Après avoir sélectionné la librairie, je me suis attelé au développement du module pour l'application MUTE. J'ai ainsi mené une réflexion sur les responsabilités respectives du client et du serveur

8. Un client et un pair correspondent à la même notion, à savoir : une page web ouverte dans un navigateur

et sur les mécanismes de gestion des contributeurs (ajout/suppression d'un contributeur, gestion des messages, etc.)

Définition des responsabilités du client et du serveur

J'ai donc énuméré les responsabilités respectives du client et du serveur.

Le serveur doit être en mesure de :
- conserver et maintenir une liste des documents créés
- conserver et maintenir une liste à jour des contributeurs pour chaque document [9]
- fournir la liste des contributeurs à un client souhaitant rejoindre le document.

Le client doit être en mesure de :
- contacter le serveur au moment de l'initialisation
- initialiser, établir et maintenir les connexions pair-à-pair
- gérer les ouvertures et fermetures de connexion pair-à-pair distantes
- gérer l'émission et la réception de messages
- conserver et maintenir une liste à jour des contributeurs pour chaque document
- conserver le dernier état connu du document [10]

Le client et le serveur doivent être en mesure de maintenir une connexion persistante sous la forme d'une webSocket.

Comme le montre la Figure 4.4, deux modules vont ainsi être implémentés, un côté serveur et l'autre côté client.

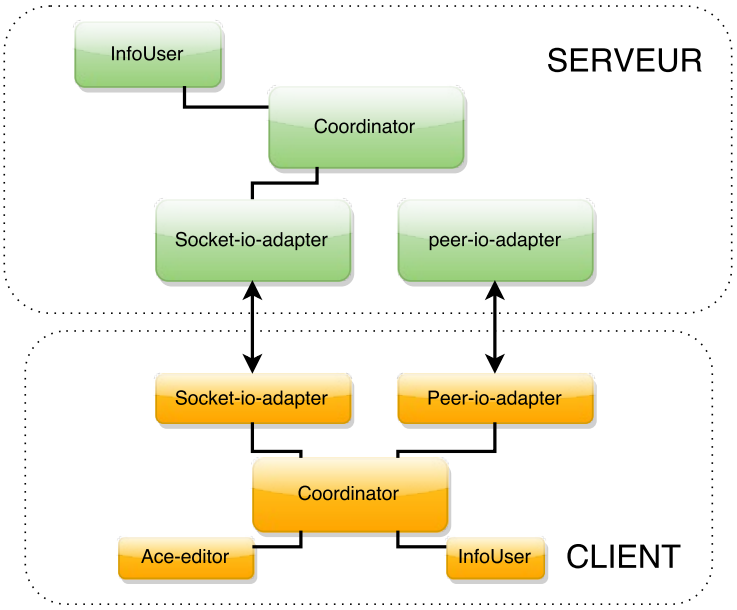

FIGURE 4.4 – Comparatif des librairies étudiées

Il est important de remarquer que même si le module côté serveur porte le nom de "Peer-io-adapter", ce dernier n'utilisera pas la technologie webRTC et n'instanciera jamais de connexion

9. Ce qui implique l'ajout ou la suppression d'un contributeur qui se connecte ou se déconnecte
10. Cette fonctionnalité a déjà été implémentée, tous les clients maintiennent une base de données locale

pair-à-pair. Le serveur se charge de maintenir une liste des collaborateurs à jour, et ce pour chaque document.

Mécanismes implémentés

Le premier mécanisme implémenté et celui de l'ajout d'un contributeur. La Figure 4.5 explicite les différentes phases par lesquelles le client doit passer pour rejoindre un document. Ainsi, quand un nouveau client tente de rejoindre à un document, il va tout d'abord contacter le serveur et lui donner son peerId. le serveur va en retour l'ajouter à la liste des contributeurs, lister l'ensemble des contributeurs actuellement présent et lui renvoyer cette liste accompagnée du numéro de site. Le client va donc créer une connexion [11] vers chacun de ces pairs. Une fois toutes ces connexions établies, le client peut communiquer directement avec tous les contributeurs du projet. Le client va demander une copie du document au premier pair de la liste que le serveur lui aura communiquée.

Si c'est le client est le premier à rejoindre le document et que le document n'a pas encore était créé, le serveur instanciera alors la structure de données qui permettra de conserver la liste de contributeurs et ajoutera le client à cette liste. Dans le cas où le document est déjà créé, le serveur se chargera uniquement d'ajouter ce client à la liste des collaborateurs du document.

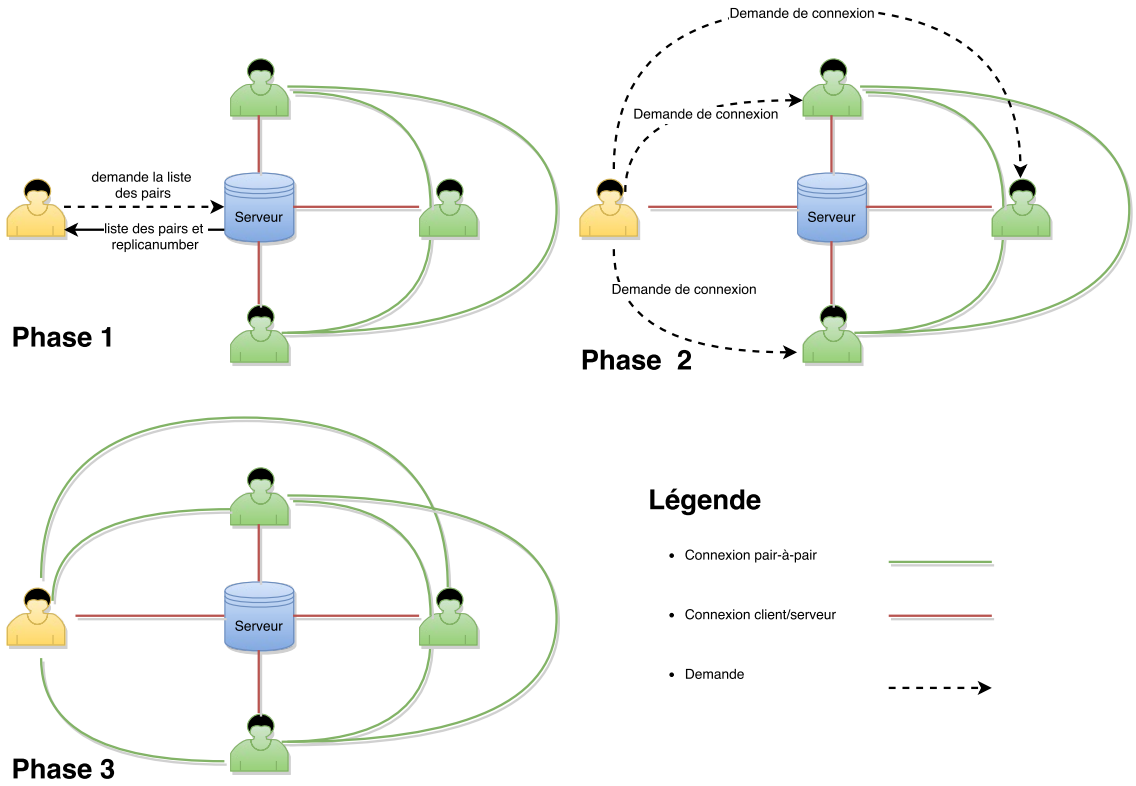

FIGURE 4.5 – Inscription d'un contributeur

Les messages envoyés à travers la connexion pair-à-pair sont formatés en JSON, ils sont convertis en chaîne de caractères à l'émission et parser à la réception. Ils comportent deux attributs : event et data. L'attribut event correspond au contexte du message, tandis que data correspond à la donnée envoyée. J'ai dû établir une liste des différents contextes pouvant se présenter de façon à adapter le comportement du client en fonction du type de message reçu. Ainsi, à la réception d'un message, le client est en mesure de savoir s'il s'agit de l'émission d'une opération texte,

11. Implicitement, une connexion pair-à-pair

d'information sur l'utilisateur ou d'autres demandes particulières. Une liste exhaustive de ces contextes est disponible en Annexe B.1.

Dans une majorité de cas, les messages sont envoyés à tous les pairs, seuls quelques messages comme la demande d'une copie du document ou la demande d'informations sur un utilisateur sont destinés à un client particulier. Quand un utilisateur génère des opérations texte, il va ainsi les envoyer à tous les clients de sa liste.

Quand un client quitte le document, les connexions pair-à-pair avec les autres collaborateurs vont se fermer et la websocket partagée avec le serveur, va elle aussi se fermer. À la fermeture de la webSocket, le serveur va retirer le contributeur de la liste des collaborateurs et sur le même principe, tous les clients vont retirer le contributeur de leur liste au moment où la connexion pair-à-pair se rompt.

La Figure 4.6 représente ainsi la nouvelle architecture et le mécanisme d'échange d'opérations texte, dans le même contexte que celui présenté par la Figure 3.3. Il est important de remarquer que le serveur n'a plus aucun rôle à jouer dans la communication des opérations textes, toutes ses fonctionnalités ont été reprises par le module Peer-io-adapter côté client. Les connexions webSocket sont uniquement maintenues pour que le serveur puisse répercuter la déconnexion d'un pair dans la liste des collaborateurs.

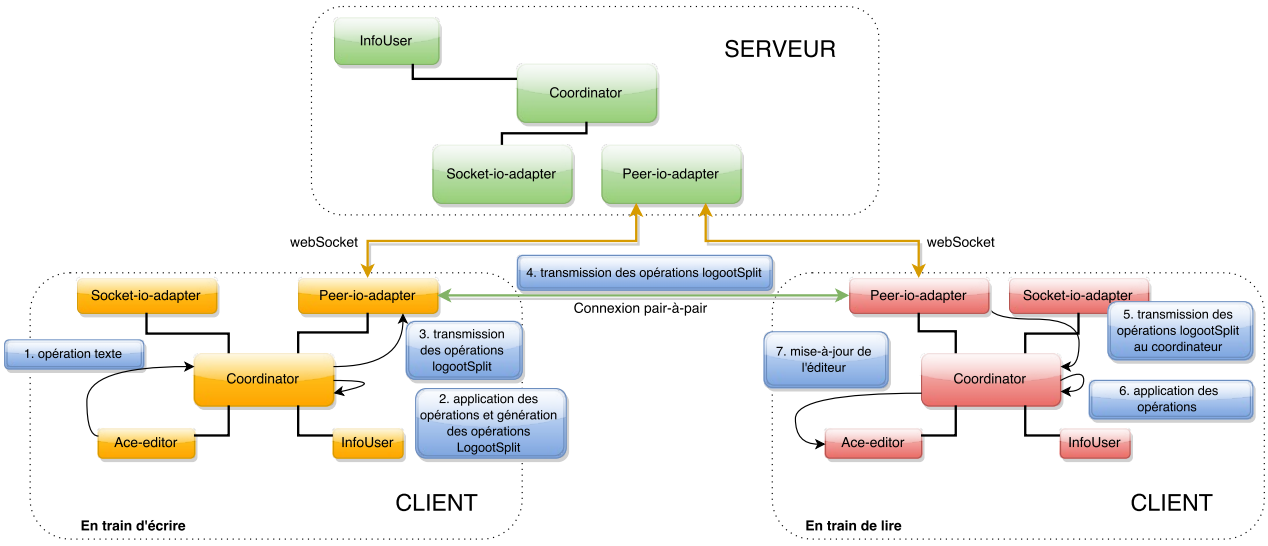

FIGURE 4.6 – Nouvelle architecture de MUTE

J'ai ainsi développé deux modules implémentant ces mécanismes, un côté serveur permettant de maintenir une liste à jour des collaborateurs, et un autre côté client implémentant le système d'insciprtion et communication avec le serveur et celui d'échange pair-à-pair avec les autres clients. En ce qui concerne le module développé côté serveur, ma réflexion a principalement porté sur la structure de données permettant de conserver la liste des collaborateurs ainsi que sur les processus d'inscription et de déconnexion des collaborateurs. Le travail autour du module côté client s'est concentré autour de l'étude du module existant, à savoir : Socket-io-adapter et de son adaptation pour un schéma pair-à-pair. Je devais en effet adapter le module Peer-io-adapter de façon à ce qu'il respecte les mécanismes de communications existants avec les autres modules.

4.2.3 Présentation du résultat

L'ensemble des fonctionnalités ont été testé et l'expérience utilisateur n'a pas mis en évidence de problème particulier. En effet, les fonctionnalités existantes décrites ci-dessous conservaient leur comportement initial.

Fonctionnalités existantes supportées par le modèle pair-à-pair :
- Partage des opérations textes avec tous les collaborateurs d'un même document
- Partage de la position du curseur avec tous les collaborateurs d'un même document
- Partage de la sélection d'un utilisateur avec tous les collaborateurs d'un même document
- Partage du nom de l'utilisateur tous les collaborateurs d'un même document

Le prototype est totalement fonctionnel et correspond donc aux attentes. Cependant nous avons estimé que la topologie du réseau pair-à-pair n'était pas utilisée de façon suffisamment efficace pour être pleinement "scalable". En effet, dans la topologie résultante tous les pairs d'un même réseau se connaissent et sont tous directement connectés les uns aux autres. Cette topologie rencontre donc certaines limites si le nombre de collaborateurs venait à augmenter. Un des moyens de répondre à cette problématique et d'utiliser un algorithme de gossip pour la construction du réseau.

4.3 Implémentation d'un système de gossip pour MUTE

Les algorithmes de gossip constituent un domaine vaste de la recherche en informatique. L'adaptation et l'intégration d'un tel algorithme pour MUTE n'est qu'une piste de réflexion et a abouti à un prototype partiellement fonctionnel. Ce travail ne faisait pas partie des objectifs initiaux du stage, mais reste néanmoins dans la continuité du travail réalisé.

4.3.1 Présentation des algorithmes de gossip

Les algorithmes de gossip ou algorithme probabilistes de dissémination d'informations sont des algorithmes inspirés par la propagation des maladies au sein d'une population. Dans le cadre de notre réseau pair-à-pair, tous les noeuds ont une certaine probabilité d'être directement interconnectés. Tous les noeuds ne se connaissent plus directement entre eux, le réseau s'en trouve alors plus soulagé. Ces algorithmes s'intègrent dans de réseau de grande envergure [12]. La Figure 4.7 présente les deux formes de topologie.

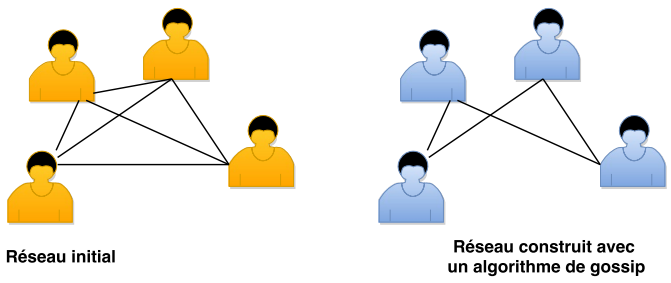

FIGURE 4.7 – Présentation des deux formes de topologie

12. Réseau pouvant contenir plusieurs centaines de milliers de noeuds

Les algorithmes de gossip assurent l'équilibrage du réseau et la fiabilité des transmissions. De nombreux algorithmes existent et adoptent chacun une approche différente. L'étude de l'ensemble de ces algorithmes n'était pas réalisable dans le temps imparti et demandait des connaissances et un recul que je n'avais pas forcément. Pour me faciliter la tâche, j'ai repris les travaux de recherche [13] portant également sur le développement d'un éditeur de texte collaboratif pair-à-pair implémentant différents algorithmes de gossip. Ces travaux présentent sur trois algorithmes : SCAMP, CYCLON et SCAMPLON. J'ai choisi d'implémenter le premier algorithme qui me paraissait être le plus simple et le plus adapté à MUTE.

4.3.2 Présentation de l'algorithme SCAMP

L'algorithme SCAMP [14] appartient à la famille des algorithmes à échantillonnage réactif [15]. L'objectif de cet algorithme est donc d'adapter la vue partielle de chaque noeud [16] à la taille du réseau, d'équilibrer et d'homogénéiser ce dernier. L'algorithme décrit également des mécanismes permettant d'éviter l'isolement d'un noeud.

On considérera un réseau comportant n noeud et des noeuds de ce réseau, numérotés i, j, k. Une valeur c sera également considérée, et correspond à une constante liée au taux de perte et à la fiabilité du réseau.

L'intégration d'un nouveau noeud dans le réseau s'effectue de la manière suivante :

1. Quand un nouveau noeud k rejoint le réseau, il envoie une demande au noeud i [17]
2. À la réception de la demande, le noeud i transfert à tous ses voisins et crée c copies de la demande et les envoie à certains de ses voisins choisis de manière aléatoire (appartenant à Ni, le nombre de voisins de i)
3. Le noeud j ayant reçu la demande va intégrer cette demande avec une probabilité de p dépendant de la taille de Nj (Nj nombre de voisins de j). Si le noeud j n'intègre pas k il transfère la demande à un de ses voisins choisi aléatoirement

Un système de bail permet de supprimer les noeuds défaillants du réseau. Une fois le bail expiré, il est de la responsabilité du noeud de s'inscrire de nouveau.

4.3.3 Adaptation de SCAMP

L'algorithme SCAMP est un algorithme initialement développement pour des réseaux de très grand ampleur, il a fallu l'adapter de façon à ce qu'il puisse fonctionner à une échelle moins importante. De plus, il a fallu reconsidérer les différents mécanismes pair-à-pair utilisé dans le module client Peer-io-apdater. L'intégration de l'algorithme SCAMP a fait l'objet du développement module client nommé Scamp-io-adapter comme le montre la Figure 4.8.

Mon travail consistait à modifier les modules Peer-io-adapter et à développer le module Scamp-io-adapter. J'ai dû avant toute chose, élaborer les mécanismes d'inscpription et d'échanges entre les collaborateurs.

13. Référence bibliographique[4]
14. Pour : Scalable probabilistic membership protocol
15. Ou : *Reactive peer smapling*
16. Ou pair
17. Ce noeud est choisi aléatoirement par un élément extérieur au réseau

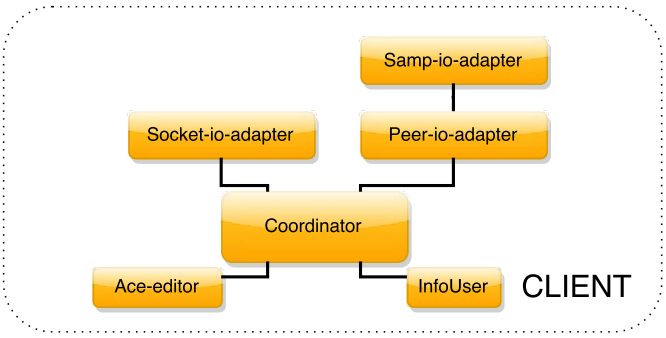

FIGURE 4.8 – Architecture du client avec l'intégration du module SCAMP

Inscription d'un collaborateur

L'inscription d'un nouveau collaborateur s'effectue de la manière suivante :

1. Lorsqu'un nouveau client A chercher à rejoindre le document, il envoie une demande au serveur et joint à cette dernière son peerId
2. En retour, le serveur va lui renvoyer son numéro de cite et le peerId d'un collaborateur selectionné aléatoirement
3. Le client A va alors envoyé une demande d'insciption au collaborateur dont il vient de recevoir le peerId (client B)
4. Le client B va alors accepté la demande ave une probabilité p = $\frac{1}{1+nombreDeVoisins}$
5. Si la demande est acceptée par le client B, la connection entre A et B est maintenue, sinon la demande est transféré à un voisin de B
6. La demande va alors parcourir le réseau jusqu'à ce que la demande soit acceptée

Echanges dans le réseau

Les pairs ne sont plus tous directement interconnectés, la diffusion d'un message à tous les pairs n'est donc plus aussi évidente. Les messages doivent donc être transférés de pair en pair pour que l'information puisse être communiquée à tous. Cependant, il faut veiller à ce que le réseau ne soit pas innondée et le pair ne reçoive pas deux fois la même information.

Pour ce faire, j'ai implémenté un système de hop, c'est un dire : un compteur [18] attaché au message, qui est décrémenté à chaque tranfert, lorsque le hop atteint 0, le message n'est plus transféré. Le réseau ne peut donc plus être innondé par les messages. En revanche, il est toujours possible qu'un pair reçoive deux fois ou plus la même information. Pour pallier à ce problème, j'ai attaché à chaque message un tuple composé du numéro de site et du numéro du message permettant d'identifier de manière unique chaque message. Lorqu'un message est émis ou reçu il est alors stocké dans une liste propre à chaque pair regroupant tous les message déjà connus par ce dernier [19]. A la réception d'un message, si ce dernier est déjà connu du pair, il sera rejeté.

18. Initialisé à une valeur > 0
19. Déjà émis ou reçu par le pair

Bilan

Mon travaillé consistait donc à implémenter ces différents mécanismes et à m'assurer que la communication avec les autres modules était toujours opérationnelle. Je n'ai pas pu mener le développement jusqu'à son terme, j'ai pu néanmoins faire fonctionner le partage des opérations texte entre tous les collaborteurs. La plus grande difficulté a été de d'adapter l'algorithme, j'ai également pris cosncience des difficultés que pouvais générer l'implémentation d'un algorithme à partir d'une description théorique.

5 Conclusion

Ce stage a été une experience enrichanssante où j'ai dû faire preuve d'autonomie et de prise d'initiative. J'ai dû structuré mon travail et dialoguer avec différents acteur pour mener à bien ma mission. L'appréhension d'un projet complex tel que MUTE m'a permis de développer certaines compétences techniques. J'avais auparavant peu développé en JavaScript, ce projet m'a permis d'aborder la programmation évènementielle et sa complexité. En paralléle, j'ai eu l'occasion d'assister à des exposés réalisés par des chercheurs, cela m'a permis d'élergir ma culture scientifique et de prendre conscience des enjeux liés au monde de la recherche.

Bibliographie / Webographie

[1] Fédéric CAVAZZA. Web 2.0 : la révolution par les usages. http://www.journaldunet.com/solutions/0601/060105_tribune-sqli-web-20.shtml.

[2] Sam DUTTON. Getting started with webrtc. http://www.html5rocks.com/en/tutorials/webrtc/basics/. 10, 11, 27

[3] Sam DUTTON. Webrtc in the real world : Stun, turn and signaling. www.html5rocks.com/en/tutorials/webrtc/infrastructure/.

[4] Julian TANKE. Scamplon : A peer sampling service for webrtc. https://github.com/justayak/research_and_Dev_Paper. 20

Liste des illustrations

Glossaire

NAT : *Network Address Translation* Mécanisme permettant de faire correspondre à une adresse publique, une adresse privée appartenant à un intranet

Pare-feu : Mécanisme de sécurité définissant les types de communication utilisés

webSocket : Connection client/serveur persistante

broadcast : Action de diffuser un message vers tous les noeuds d'un même réseau

Annexes

A Première Annexe

A.1 Organisation et rendu du travail réalisé

Le travail réalisé a été rendu sous la forme de pull-request adressées aux dépots de MUTE, accessible à ces adresses :

- MUTE client : https://github.com/coast-team/mute-client
- MUTE server : https://github.com/coast-team/mute-server
- MUTE demo : https://github.com/coast-team/mute-demo

L'organisation de mon travail est décrite par la Figure A.1. Avant de travailler sur la problèmatique proposée par le stage, j'ai réalisé une première mission pour François CHAROY, je devais adapter un framework de crowdsourcing nommmé Pybossa [1] et déployer ce dernier sur un serveur AWS [2]. J'ai eu l'occasion de découvrir les environement virtuels légers tels que Vagrant [3].

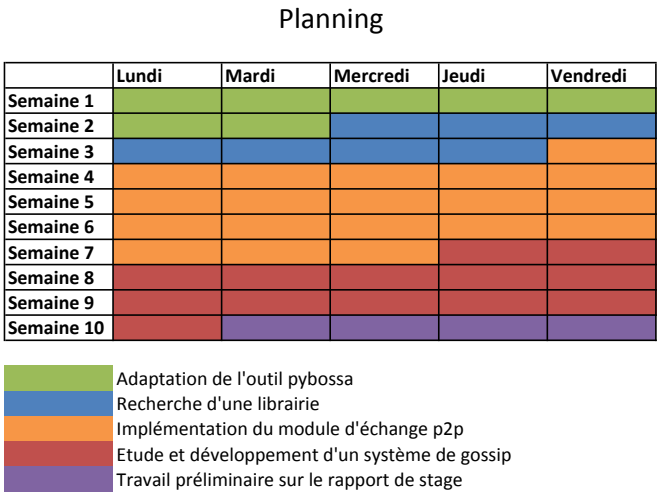

FIGURE A.1 – PLanning de travail

1. Pour plus d'information : http://pybossa.com/
2. Amazon Web Service
3. Pour plus d'informaton : https://www.vagrantup.com/

B Seconde Annexe

B.1 Liste exhaustive des contextes

- *sendOps :* envoi d'opérations textes
- *sendDoc :* envoi d'une copie du document
- *queryUserInfo :* demande d'informations sur l'utilisateur
- *addUser :* demande d'ajout d'un utilisateur
- *broadcastCollaboratorCursorAndSelections :* envoi d'informations sur la position du curseur
- *broadcastCollaboratorUsername :* envoi du nom d'un utilisateur
- *joinDoc :* demande d'une copie du document

Résumé

Le but de ce projet était de développer un module de communication pair-à-pair pour l'éditeur de texte collaboratif MUTE. Je devais ainsi utiliser la technologie webRTC pour instancier les connexions pair-à-pair. Je devais selectionner une librairie exploitant cette technologie, développer et intégrer le module dans l'application MUTE.

Mots-clés : webRTC, p2p, PeerJS, SCAMP, edition collaborative

Abstract

The goal of the project was to implement a peer-to-peer communication module for MUTE, a collaborative text editor. Thus, I had to use the webRTC technology to instanciate peer-to-peer connections. I had to select the right library exploiting webRTC technology, develop and include the communication module into MUTE application.

Keywords : webRTC, p2p, PeerJS, SCAMP, collaborative edition