PFA
Rapport
Informatique Deuxième Année

Partage d'écran en réseau local

Élèves :
Antoine Pringalle (apringalle@enseirb-matmeca.fr)
Benjamin Upton (bupton@enseirb-matmeca.fr)
Jalal Izekki (jizekki@enseirb-matmeca.fr)
Saad Margoum (smargoum@enseirb-matmeca.fr)
Souhail Nadir (snadir@enseirb-matmeca.fr)
Saad Zoubairi (szoubairi@enseirb-matmeca.fr)
Théo Lelasseux (tlelasseux@enseirb-matmeca.fr)
Zaid Zerrad (zzerrad@enseirb-matmeca.fr)

Enseignant :
David Renault
(renault@labri.fr)

12 / 05 / 2021

Introduction

De nos jours, **le partage de flux vidéo est devenu un domaine très utilisé et demandé**. En effet, plusieurs entreprises, établissements et organismes s'intéressent au télétravail et ainsi les cours, les entretiens et les présentations en distanciel sont devenus un besoin primordial. Par conséquent, et grâce au développement que connaît ce domaine, la transmission de flux est devenue facile et est réalisée par différents moyens. La communication en temps réel pour le web (Web Real-Time Communication - **WebRTC**), se présente comme étant l'une des méthodes les plus pratiques permettant cette communication. En effet, son but est la mise en communication de deux pairs qui désirent échanger des données ou des flux de médias.

Intégrer ce système sur **un réseau local** se présente comme étant très utile, puisqu'un réseau local permettra l'échange de données **sans nécessité d'accès à internet**. Par suite, cet outil permet de palier à plusieurs difficultés qui se posent notamment au niveau de l'enseignement puisqu'un tel système faciliterait la réalisation des cours au niveau des établissements en simplifiant la discussion entre étudiants et aidant les encadrants à avoir plus de fluidité lors de la réalisation de leurs cours en affichant leur flux vidéo et ayant accès aux flux vidéo des étudiants. Ainsi, le but de ce projet a été d'**utiliser les propriétés liées au WebRTC afin de mettre en place un système de partage d'écran sur un réseau local**. Cette mise en place a consisté en la définition d'un **serveur** qui a permis la communication entre plusieurs **clients**. Notons aussi que la communication ne nécessite que **peu de débit**. Ainsi, ce paramétrage aura pour but de limiter le transfert de données coûteuses.

C'est aussi notre premier long projet qui s'étend sur quasiment la totalité de l'année scolaire, et c'est aussi notre premier projet avec un client. Notre groupe avait comme particularité d'avoir l'enseignant qui jouait aussi le rôle du client. Cela nous a tout de même permis de faire une sorte de jeu de rôle en demandant à notre client ses attentes pour rédiger le dossier de spécification ou pour lui proposer des solutions avant de les mettre en place.

Nous présenterons dans ce rapport tout d'abord l'**organisation et la gestion du projet** en détaillant les différents outils qui nous ont permis la réalisation du projet. Nous essayerons aussi de détailler l'**architecture** mise en place et les différents liens entre composants ainsi que les **principes d'implémentation ou de développement** de notre outil. Ensuite, nous tâcherons de présenter les étapes qui permettent le **fonctionnement du système**. Enfin, nous finirons par présenter **les différentes fonctionnalités** du système ainsi que les **expérimentations** réalisées tout au long du projet.

Table des matières

1 Organisation

Nous présentons au niveau de cette partie **les différentes méthodes** qui nous ont permis de **travailler ensemble**. En effet, plusieurs outils ont permis de **structurer notre travail** surtout que nous n'avions pas l'habitude de travailler en groupe de 6-7 personnes, ce qui rajoutait une difficulté de plus.

1.1 Travail individuel

Au début du projet et lors de la **période dédiée à la documentation**, nous avons décidé de travailler **individuellement** en essayant, chacun de sa part, de comprendre le sujet, d'assimiler les problématiques liées au sujet ainsi que d'acquérir des bases sur les différents modules indispensables au développement du projet. En effet, il s'agissait d'un travail individuel, mais nous avons essayé de faire en sorte que tous les membres du groupe aient une idée claire du sujet en réalisant parfois des réunions qui permettaient d'échanger nos points de vue.

1.2 Séparation des tâches par équipes

Dès la fin de cette période de documentation, nous nous sommes intéressés à **séparer les tâches** et créer des petites équipes qui s'intéressaient à une partie précise du sujet. En effet, notre sujet peut se décomposer en deux grandes parties, une première liée au développement du serveur responsable de l'initiation de la connexion, ainsi qu'une deuxième partie client, contenant les différentes fonctionnalités liées au sujet. Ainsi, deux équipes ont été créées et les membres de ces équipes pouvaient se répartir les tâches de leur partie plus facilement.

Ce travail entre plusieurs membres nous a poussé à utiliser l'outil **Git** qui permettait une gestion du code plus efficace puisqu'il permet la fusion des travaux, le développement en parallèle ainsi que l'accessibilité aux travaux des autres membres du groupe. Au début de notre implémentation, nous avons choisi une méthode basée sur le **stockage des sources extérieures sur des dossiers** ce qui nous permettait d'y accéder facilement. Aussi, différents membres travaillaient sur ces dossiers dans le but d'extraire des informations utiles. Après avoir réussi à implémenter une version de base de notre projet (une messagerie + un partage de vidéo entre deux utilisateurs), nous avons constaté que l'utilisation excessive des dossiers créait un **désordre au niveau de notre dépôt**. Ainsi, en utilisant cette méthode, nous avons dû résoudre plusieurs conflits et notre dépôt était rempli de fichiers inutiles. Par la suite, un **nettoyage du code** s'est imposé, et l'**utilisation des branches** s'est présenté comme le moyen le plus efficace. En effet, les branches évitaient la génération des conflits et des fonctionnalités ont pu être implémentées dans des branches indépendantes. Cependant, cette méthode pouvait créer parfois des sources

de problèmes surtout lors de la partie de fusion et réalisation du lien entre les des deux parties client et serveur.

1.3 Communication

La partie la plus importante de notre organisation au niveau du projet était la communication. En effet, les deux parties du sujet étaient bien liées et nécessitaient une communication continue afin de s'organiser et assurer l'avancement de notre implémentation. Ainsi, au début du projet, nous nous sommes contentés de réaliser des **réunions en distanciel** à cause de la crise sanitaire actuelle. Cette méthode s'est vite montrée **inefficace** puisque la majorité des membres essayaient de produire une implémentation individuelle, ce qui a rendu le choix du code de base un peu délicat. Afin de résoudre ces problèmes, nous avons décidé d'organiser des **réunions en présentiel** qui ont mené vers des résultats satisfaisants puisque :

— nous avons pu discuter de l'état de notre implémentation

— nous avons pu réaliser du **peer programming**

— nous avons aussi pu prendre une décision concernant la version de base à utiliser

Aussi, nous avons pu améliorer notre communication, au cours du projet, en utilisant le système des **issues** de **GitHub** qui nous a permis aussi de gérer les différentes tâches à réaliser.

Outil issues de GitHub

Le manque d'organisation que nous avons ressenti au début du projet nous a poussé à chercher une alternative efficace qui pouvait maintenir la communication tout en continuant le développement de notre projet. En effet, le système des **issues** existant dans **GitHub** était la solution utilisée afin de remédier à ces problèmes. Cet outil de gestion en ligne permet la séparation des tâches entre les différents membres du groupe en gardant une trace écrite de :

— ce qui a été réalisé

— ce qui est en cours de réalisation

— et ce qui devra être fait

À travers ce système, nous avions une **représentation de l'état d'avancement des tâches** ce qui permettait ainsi de représenter l'avancement du projet. Comme le montre la figure 1, une action réalisée est caractérisée par **un titre**, **les membres la réalisant**, l'état actuel de son **avancement** ainsi que des **commentaires** qui illustrent le travail réalisé ou les problèmes à régler. Ainsi tous les membres peuvent interagir à propos du contenu proposé au niveau de la section commentaires.

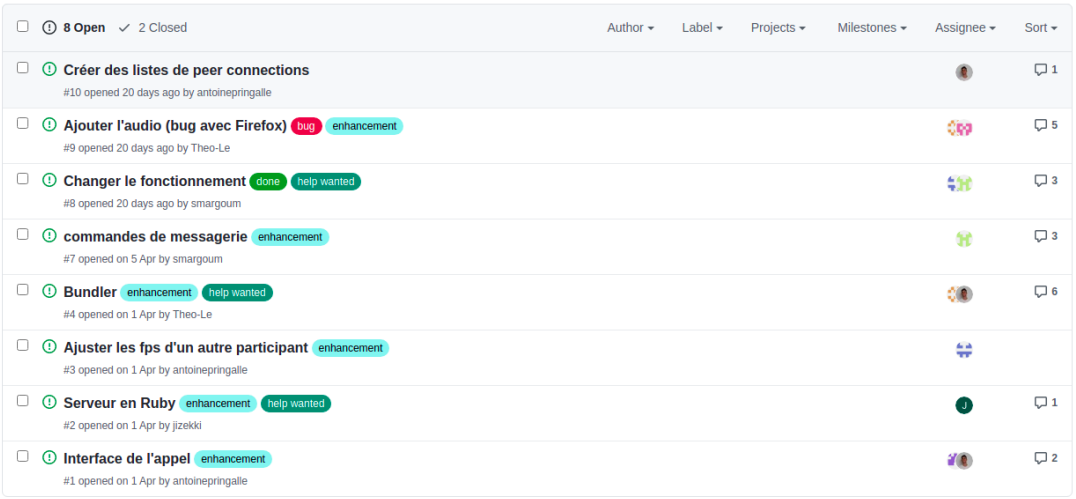

FIGURE 1 – *État actuel des issues du projet*

Une fonctionnalité très utile au niveau de ce système est l'étiquette qui définit l'état actuel de la tâche. En effet, il existe plusieurs types d'étiquettes et le comportement du groupe devait s'adapter à chaque type, par exemple lorsqu'une tâche passe à l'état de test de son fonctionnement, quelques membres qui n'étaient pas chargés de la tâche devaient tester son bon fonctionnement et donner leur accord sur le travail réalisé pour passer en **étiquette bug** s'il existe des problèmes ou en **done** si aucun problème n'a été détecté. La figure 2 suivante liste les différentes étiquettes choisies par les membres du groupe.

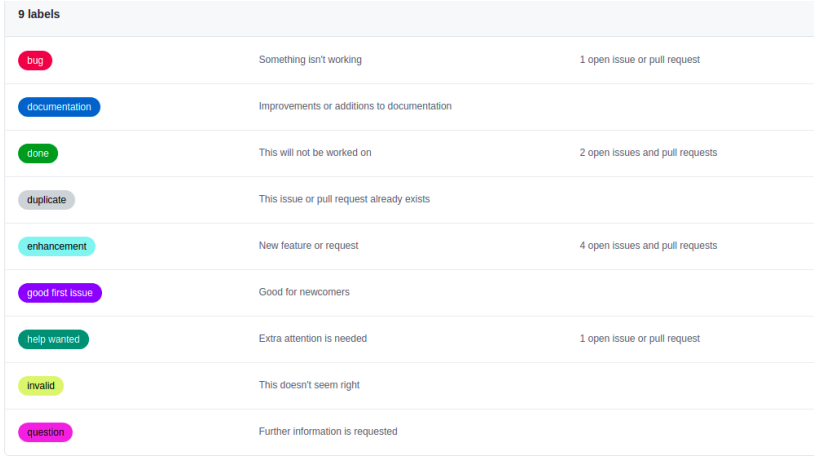

FIGURE 2 – *Les étiquettes définies au niveau du système "issues"*

Notons qu'il existe d'**autres méthodes** pour la répartition des tâches telles que **Trello** ou **Kanboard**, qui ont des interfaces plus adaptées en affichant les tâches à réaliser, en cours de réalisation et déjà réalisées et aussi des colonnes qui peuvent être spéciales définies par les utilisateurs, ces outils peuvent contenir le **nombre d'heures** d'une tâche ou même sa **complexité** et permettent aussi d'ajouter des pièces jointes ainsi que des listes de tests (check-lists).

Il existe aussi des outils bien plus complexes contenant des fonctionnalités plus développées, mais **le système des issues nous a parfaitement convenu** puisque notre but n'était que de répartir les tâches et avoir une vision sur l'avancement et une description de nos actions.

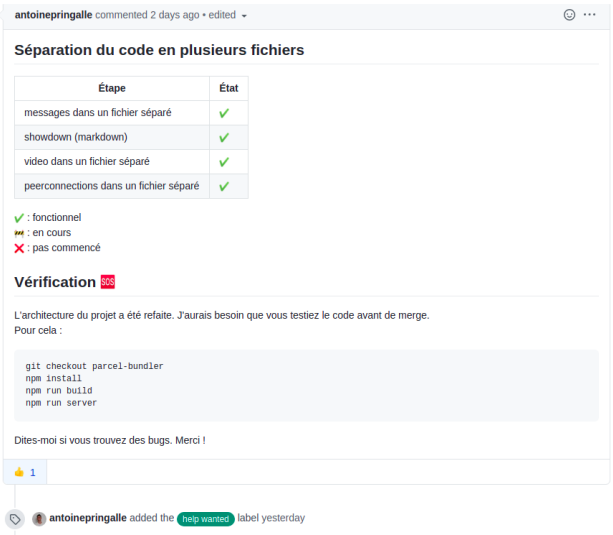

FIGURE 3 – *Exemple d'une tâche sur GitHub*

La figure 3 présente un exemple de construction d'une tâche, la manière de présentation de l'implémentation réalisée, le changement d'état de la tâche ainsi que les réactions des membres du groupe. Ainsi, l'étape à réaliser ensuite serait de tester les fonctionnalités rajoutées et affirmer la validité de l'implémentation. Notons que ce système contient une catégorie contenant les tâches fermées et qui n'ont plus besoin d'être traitées (la catégorie `closed`).

Enfin, d'autres moyens de communication ont été utilisés tel que **Messenger** et **Discord** qui nous ont fourni un moyen rapide et facile de **communication instantanée**. Grâce à ces applications, nous avons pu :

— organiser des réunions

— vérifier la disponibilité des membres du groupe

— ainsi que travailler en petites équipes

1.4 Problèmes rencontrés

D'une part, comme nous l'avons déjà présenté, nous avons parfois rencontré des difficultés qui ont impacté notre avancement au niveau du projet, ces difficultés ont généralement été **organisationnelles**. Notons que le travail devait être réalisé en dehors des heures de cours, et donc dans des périodes où la charge d'étude au niveau de l'établissement était importante, nous avons donc eu des difficultés à nous organiser et programmer des créneaux de travail. Du fait du contenu de notre projet, **nous avions à utiliser de nouvelles technologies**, ce qui rendait notre durée de documentation un peu plus longue, ainsi nous avons un peu tardé à commencer

notre implémentation par rapport à ce qui était prévu. Notons que le sujet du projet ne donnait pas la possibilité de fixer une version intermédiaire puisque son développement est continu et à chaque ajout de fonctionnalité différents problèmes ou idées d'amélioration se présentaient.

D'autre part, nous avons également rencontré différents problèmes en ce qui concerne les **technologies utilisées**. En effet, la consigne était de produire un serveur en **Ruby** mais nous n'avons pas pu parfaitement transformer notre serveur **javascript** en **Ruby**. Les problèmes liés à la documentation ont été source d'une grande difficulté, puisque quelques défauts d'implémentation et ce manque de documentation limitaient notre avancement.

De plus, **le seul moyen de tester notre implémentation était de se réunir** et installer un serveur local, mais une réunion ne se faisait qu'une fois par semaine ce qui ralentissait l'avancement du travail. Notons aussi, que les tests réalisés ne sont pas exhaustifs et qu'il est donc possible que parfois nous validions une fonctionnalité sans tester tous les cas possibles.

Par suite, ces différents obstacles, ont impacté le projet. Il est intéressant de regarder la réalité du produit et le comparer avec les prévisions faites au niveau du document de spécification, disponible en annexe 9.2. En effet, malgré les anticipations que nous avions faites au début du projet, il était certain que d'autres problèmes allaient surgir et impacter notre travail. Cet aspect particulier a été clairement présent au niveau de ce projet, puisque **la cadence de travail n'a pas toujours été régulière**. Par conséquent, tout au long du projet nous avons connu des périodes de résolution de problèmes et des périodes d'ajout de fonctionnalités. Ce qui explique le fait que notre **rendement était différent de ce qui était prévu au début**.

2 Architecture

Notre projet étant **expérimental**, l'architecture de notre projet ne se résume pas seulement à sa version finale. On y retrouve également des petits dossiers qui nous ont permis de réaliser nos tests avant d'implémenter sur la version principale.

À la racine de notre projet, on peut retrouver 3 grands répertoires.

— `local-screen-sharing`

— `essais_javascript`

— `essais_ruby`

Dans la suite nous présentons à quoi servent les dossiers expérimentaux et ce que l'on peut trouver, puis nous détaillerons l'architecture du dossier principal.

2.1 Dossiers à la racine

Le premier dossier, `local-screen-share` est le dossier principal sur lequel nous travaillons ensemble. C'est dans ce dossier que nous avons développé le serveur et les clients de notre application. Nous en parlerons davantage dans la section qui suit.

Le second dossier, `essais_javascript`, contient une petite dizaine d'autres répertoires. Chacun de ces répertoires sont des **petits environnements** sur lesquels nous avons pu réaliser des **tests d'implémentation en javascript avant de s'en servir sur le dossier principal**. Il y a notamment des serveurs implémentant juste un **chat**, un autre gérant des **appels vidéo**, d'autres mettant en relation des **peerconnections**.

Le dernier dossier, `essais_ruby`, comporte 3 **serveurs en Ruby**. Nous avions pour but d'implémenter le serveur de notre application en Ruby. Le serveur en Ruby aurait pu alors être disponible sur la forge de l'école, elle même implémentée en Ruby. Nous allons vous présenter dans les grandes lignes ces répertoires, mais nous y reviendrons dans la section 7 Expérimentation.

- `ruby-server-version` est une traduction de l'un de nos serveurs javascript en Ruby.
- `WebRTC-ruby` est un application en Ruby jouant avec le WebRTC. L'application a été trouvée sur le GitHub de ethlocker, nous l'avons un peu adaptée pour faire nos expériences personnelles sur cette base.
- `ruby-tiny-https` est un simple serveur **https** en ruby utilisant les gems **webrick** et **sinatra**.

2.2 Le dossier principal : local-screen-sharing

2.2.1 Le bundler Parcel

Parcel.js est un bundler générant de façon assez autonome les fichiers que sert notre serveur. Il nous permet notamment de **séparer notre code en plusieurs petits fichiers** ce qui constitue pour nous un grand avantage. Premièrement **on gagne en clareté**, car on peut ranger nos fonctions selon leur utilité (on regroupe les fonctions liées au partage de vidéo ensemble, celles liées au chat dans un autre fichier, etc.). Deuxièmement, pour le **développement** il est **plus pratique** que chacun puisse travailler sur le fichier correspondant à la fonctionnalité qu'il souhaite améliorer sans risquer d'entrer en conflit avec le travail des autres. Si tout le code se trouve dans un grand fichier, les chances que nos modifications impactent une fonctionnalité sur laquelle on ne travaille pas sont plus grandes que quand on sépare nos fonctionnalités dans différents fichiers.

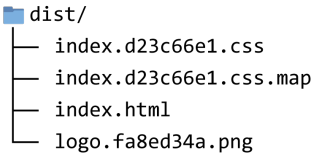

FIGURE 4 – *fichiers générés par Parcel*

Ce n'est pas tout, grâce au bundler, le **code est optimisé**. C'est à dire qu'il est réécrit pour qu'il soit plus rapide à charger pour le client. Quand on observe les fichiers générés par Parcel, disponibles en figure 4, on ne retrouve qu'un seul fichier javascript. Ce dernier a été condensé en une seule ligne et va de paire avec son fichier .map associé. L'association de ces deux fichiers permet au navigateur de reconstruire le code source.

Il est très facile de rajouter une autre fonctionnalité, il suffit de rajouter un fichier et de configurer les fonctions à utiles aux clients dans `imports.js`. Ce dernier fichier appelé par `index.html` sert, comme son nom l'indique, à importer toutes les fonctions dont aura besoin le client et les autres fichiers de l'application. Par exemple, on retrouve les fonctions liées au boutons qui sont appelées directement par le client et les fonctions implémentées dans les différents fichiers dont `chatclient.js` se sert.

En somme, **le bundler présente de nombreux avantages** pour un projet web comme celui-là et n'a pour difficulté que la mise en place des importations à gérer. Si le bundler avait été présent dès le début du projet l'étape d'importation aurait été faite toute au long du projet, et donc plus facile.

2.2.2 Dossiers et fichiers de notre application

Nous vous présentons dans cette partie les différents dossiers et fichiers qui composent notre application, par une courte description. L'arbre de nos fichiers est ci-contre en figure 5.

- `node_modules/` comporte les packages node installés pour le bon fonctionnement de notre application.
- `dist/` est un dossier d'output des fichiers générés par le bundler Parcel.
- `chatclient.js` est le fichier en quelque sorte central pour notre client. Il va piocher dans les autres fichiers javascript les fonctions utiles pour nos clients.
- `video.js` regroupe les fonctions liées au partage d'écran mais aussi celles de l'audio.
- `messages.js` regroupe toutes les fonctions de messagerie.
- `peerconnection.js` regroupe toutes les fonctions relatives aux peerconnections.
- `dark.js` permet de basculer entre l'affichage clair et l'affichage sombre.
- `utils.js` regroupe principalement les fonctions de log, utiles à l'ensemble des autres fichiers javascript.
- `imports.js` est un fichier appelé par index.html ne servant qu'à importer les fonctions des autres fichiers dont dépend le client.
- `chat.css` est le fichier de style appliqué à notre page html.
- `logo.png` est le logo Bordeaux INP qui fait toute la différence.

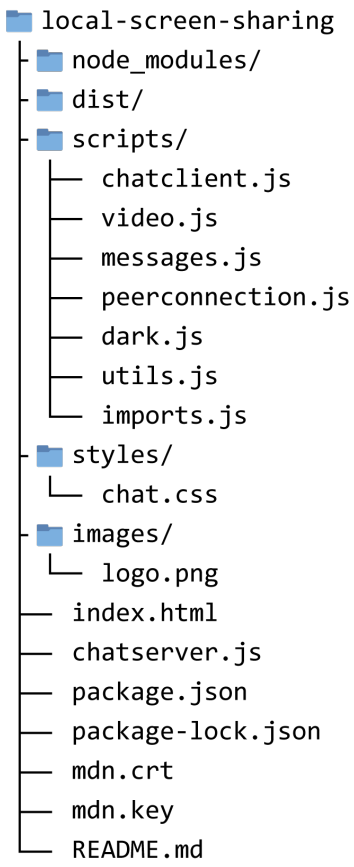

FIGURE 5 – *Architecture de notre application*

- `index.html` est le fichier que l'on sert, c'est la page de notre application.
- `chatserver.js` implémente le serveur HTTPS de notre application.
- `package.json` est le fichier de configuration de npm. Il définit notamment les packages à installer pour le bon fonctionnement du projet. On peut aussi y rajouter des scripts pour lancer rapidement des commandes récurrentes.
- `package-lock.json` est automatiquement généré à chaque fois que npm modifie soit l'arbre node_modules, soit package.json. Il décrit l'arbre exact qui a été généré, de sorte que les installations ultérieures puissent générer des arbres identiques, indépendamment des mises à jour intermédiaires des dépendances.
- `mdn.crt` est le certificat SSL auto-signé que nous avons généré pour lancer le serveur en https.

- `mdn.key` est la clé du certificat SSL auto-signé.
- `README.md` est le document à lire avant de commencer à se servir de l'application, il comporte plusieurs explications sur comment lancer correctement le serveur et s'y connecter.

2.2.3 Architecture

Comme le montre la figure 6, notre architecture (bien que grandement simplifiée sur le graphe) est assez **compliquée au premier abord** puisque nous avons séparé notre implémentation en plusieurs petits fichiers. **Ces dépendances entre les fichiers ne sont en réalité pas compliquées à gérer**. Nous avons un fichier supplémentaire `imports.js` qui s'occupe répertorier et de rendre accessible les fonctions ont dépendent les autres fichiers. Vous retrouverez en Annexe 9.1 l'architecture détaillée des fonctions utilisées par les autres fichiers.

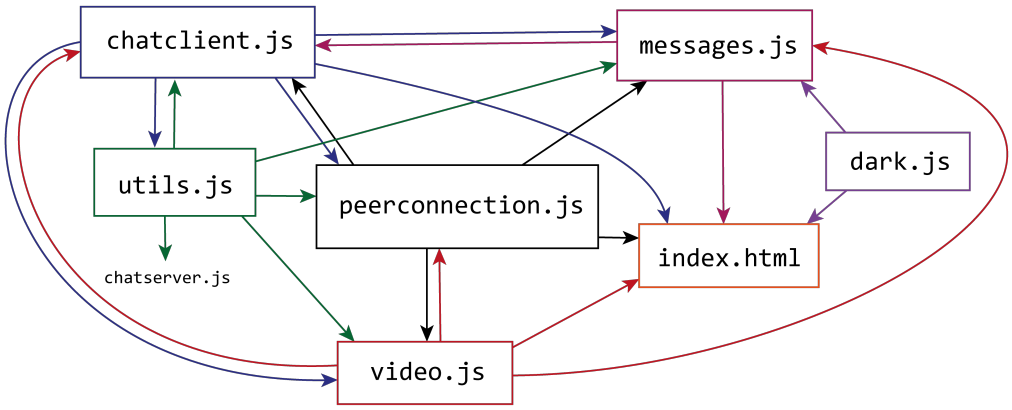

FIGURE 6 – *Architecture simplifiée local-screen-sharing*

Si vous vouliez rajouter une fonction `foo()` dans *messages.js* qui serait appelée dans *chatclient.js*, il vous suffit de suivre l'exemple en figure 7 ci-contre.

Dans un premier temps la fonction doit être déclarée avec le mot-clé `export`, puis dans *imports.js*, il suffit de l'importer et de rajouter la ligne `window.foo = foo;` pour qu'elle soit accessible aux autres fichiers mais aussi à la console du navigateur. Nous avons donc réalisé ce processus pour toutes les fonctions définies dans un fichier mais appelées par d'autres.

```
// messages.js
export function foo() { ... }

// imports.js
import { foo } from "./messages.js";

export function import_everything() {
    ...
    window.foo = foo;
    ...
}

// chatclient
foo();
```

FIGURE 7 – *Ajouter une fonction*

Au moment de la séparation des fichiers nous avons aussi dû rajouter des assesseurs (*getter*) et des mutateurs (*setter*) pour accéder et/ou modifier des variables globales définies dans d'autres fichier. Par exemple, `getVideo()` permet à *peerconnection.js* d'obtenir la variable de notre stream local définie dans *video.js*.

3 Le WebRTC

Notre projet repose entièrement sur le **WebRTC**. Nous avons cherché et expérimenté ce qu'il était possible de faire avec cette interface de programmation (API).

3.1 Qu'est-ce que le WebRTC ?

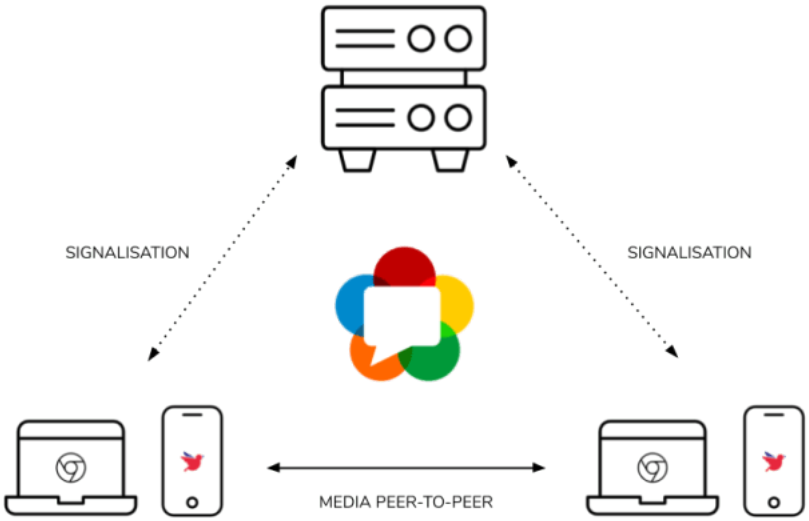

FIGURE 8 – *Schéma simplifié de fonctionnement du WebRTC (wazo.io)*

Selon le site **webrtc.org**, l'objectif du projet **WebRTC** est de "permettre le développement d'applications RTC (*Communication en temps réel*) riches et de haute qualité pour les navigateurs, les plateformes mobiles et les appareils IoT (*Internet des objets*), et leur permettre de communiquer via un ensemble commun de protocoles". La figure 8 présente un schéma simple de ce à quoi sert le WebRTC.

Ce système facilite les applications de navigateur à navigateur pour les appels vocaux, le chat vidéo et le partage de fichiers. Usuellement, il utilise un serveur de signalisation (*Signaling Server*) qui, conjointement à un serveur *STUN*, est nécessaire pour fournir la page initiale et synchroniser les connexions entre deux points de terminaison WebRTC.

— ***Processus de Signaling*** **:** L'établissement d'une connexion WebRTC entre deux machines nécessite l'utilisation d'un serveur de signalisation pour déterminer comment les connecter sur le réseau. Le serveur de signalisation joue le rôle d'intermédiaire pour permettre à deux machines (*Peers*) de se trouver et d'établir une connexion tout en minimisant autant que possible l'exposition d'informations potentiellement privées.
Les *Peers* doivent échanger des informations sur la connexion réseau. Ces informations

sont connues sous le nom de *ICE candidate* et détaillent les méthodes disponibles à travers lesquelles le *Peer* est capable de communiquer.

— ***Rôle du serveur STUN*** : Le serveur *STUN* permet aux clients de connaître leurs adresses publiques, le type de NAT derrière lequel ils se trouvent et le port côté Internet associé par le NAT à un port local particulier.

WebRTC en réseau local

Dans le cas de notre application destinée à un réseau local, **l'utilisation du serveur *STUN* n'est pas nécessaire**. En effet, les endpoints du WebRTC peuvent générer des *ICE candidates* en utilisant uniquement leurs adresses connues dans le réseau local. Les informations sont en effet échangées pendant le processus du *Signaling*. La *Peer Connection* peut donc être établie sans qu'il soit nécessaire de contacter un serveur *STUN* externe au réseau local.

3.2 Messagerie

L'envoi de messages entre plusieurs clients, dont un diagramme de séquence est disponible en figure 9, se fait de la façon suivante :

— Afin d'envoyer un message, le client va envoyer un objet *JSON* de type `"msg"` au serveur via la *WebSocket*.

— Le serveur, ayant un *handler* pour l'événement `"msg"`, va recevoir l'objet *JSON* de l'autre côté de la *WebSocket* et va le traiter en fonction de son type. Dans notre cas l'objet est de type message et il va donc envoyer à tous les clients connectés ce message encore une fois sous forme d'objet *JSON* en précisant le type message.

— Par le même mécanisme de *handler*, les clients vont recevoir cet objet *JSON*, voir qu'il est de type message et ajouter le message à l'*HTML* de leur page.

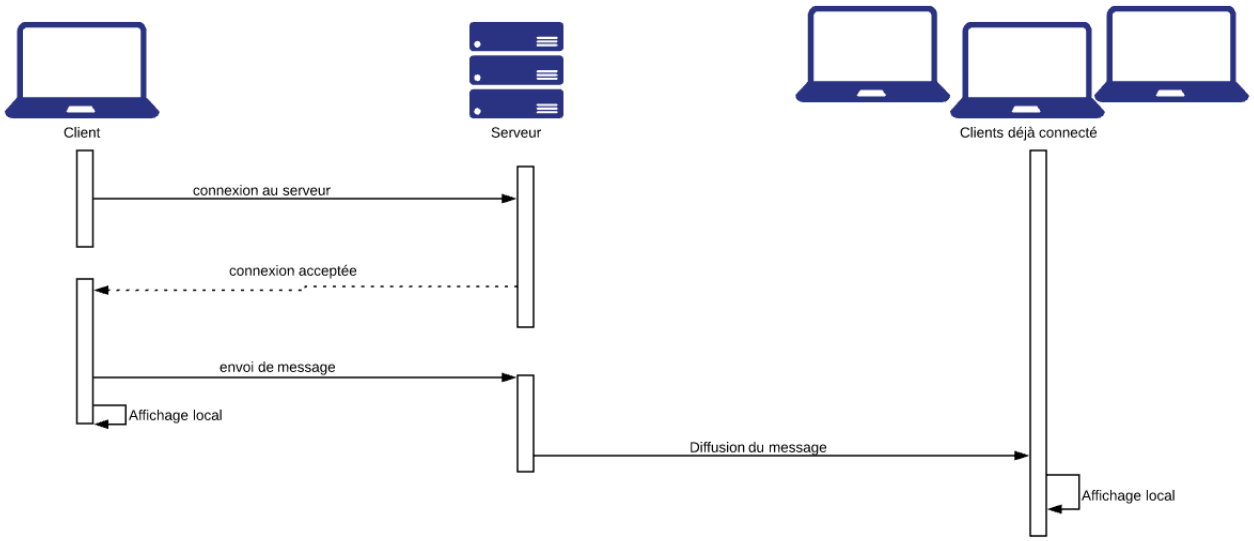

FIGURE 9 – *Flux Clients/Serveur dans le cas de la* ***messagerie***

3.3 Partage d'écran

WebRTC permet de mettre en place un partage d'écran entre deux clients via une *Peer Connection*. Pour cela :

1. Il faut dans un premier temps que les clients autorisent l'accès à leur écran. La fonction *getDisplayMedia* demande l'autorisation au client et récupère le stream de son écran.
2. Ensuite, un des deux clients va initialiser l'appel. Le fonctionnement de la demande d'appel est similaire à la messagerie sauf que le type de l'objet *JSON* n'est pas *"msg"* mais *"video-offer"* (offre dans le diagramme 10).

Après que le second client ait accepté, il faut mettre en place la *Peer Connection*.

Le serveur est utilisé comme **serveur de signalisation** ce qui signifie que les deux clients échangent leur *ICE candidate* (des informations sur la connexion réseau) via celui-ci, et ce, dans le but d'établir de façon interactive (offre/réponse) une connexion indépendante sans passer par le serveur. Une fois la *Peer Connection* établie, chaque client l'utilise pour transmettre son flux et recevoir celui de l'autre client. Leur code HTML est alors mis à jour pour intégrer la vidéo de l'écran partagé. Il est aussi possible pour un client de changer les *fps* du flux vidéo qu'il transmet en appliquant une contrainte sur son flux.

Lorsqu'un des deux clients met fin à l'appel, la *Peer Connection* est fermée et les écrans ne sont plus partagés. Il leur est alors possible d'initier un nouvel appel. Les échanges de ce processus d'établissement de la connexion *Peer-to-Peer* sont illustrés par la figure 10.

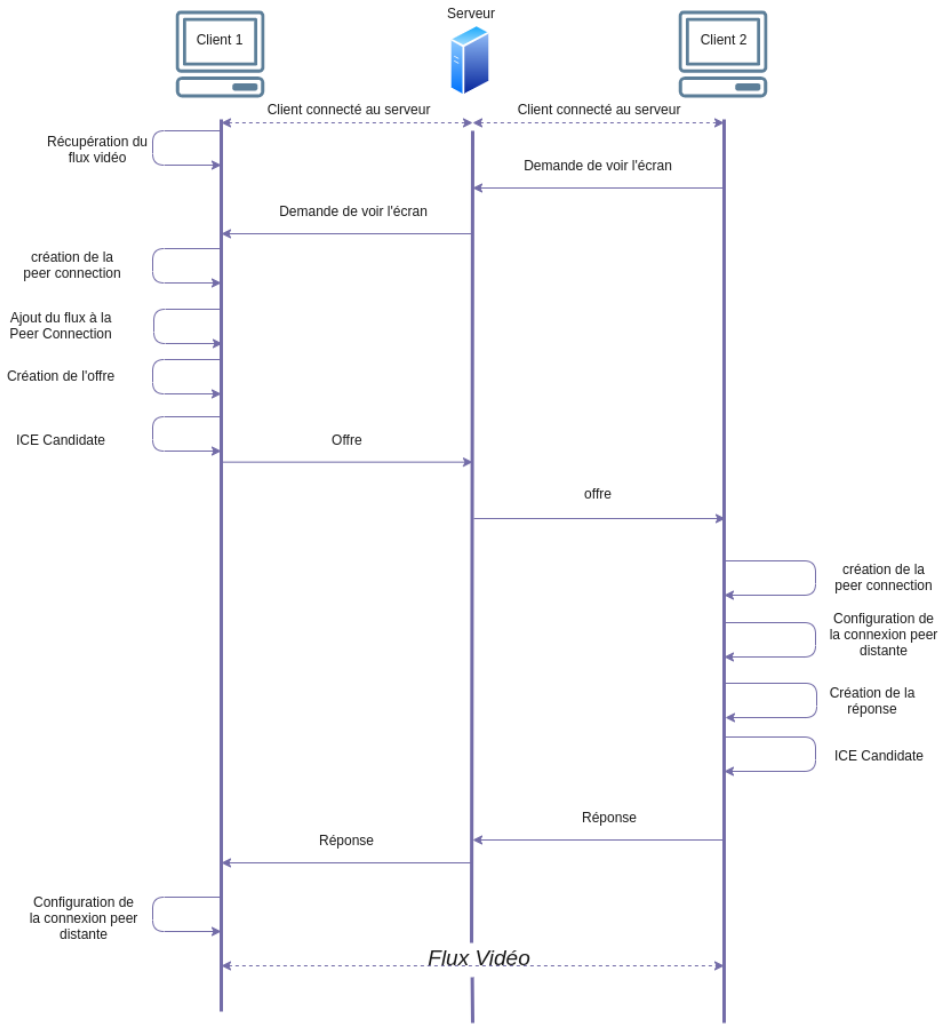

FIGURE 10 – *Flux simplifiés Clients/Serveur dans le cas du* ***partage d'écran***

3.4 Partage d'audio

Le partage d'audio est assez **semblable au partage d'écran**. Dans ce cas, la fonction *getUserMedia* est utilisée pour demander l'autorisation au client et récupérer le stream audio. **L'implémentation est légèrement différente selon le navigateur utilisé.** Le stream audio est ensuite ajouté au stream vidéo puis le fonctionnement est le même que précédemment.

4 Fonctionnement en diagrammes de séquence

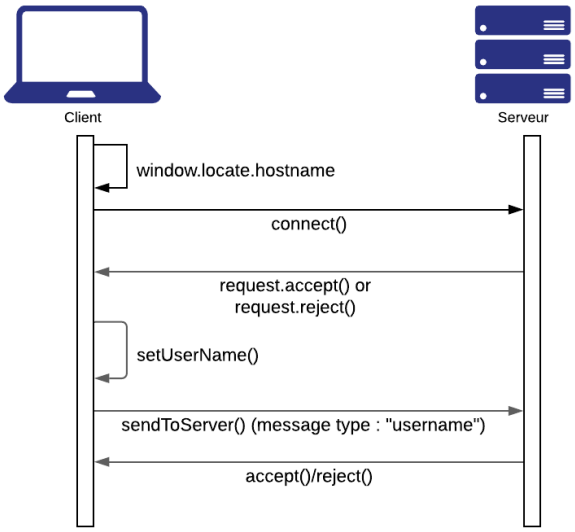

FIGURE 11 – *Connexion au serveur -Diagramme de séquence -*

La figure 11 décrit une demande de connexion d'un client au serveur. Le client envoie une demande de connexion au serveur (c-à-d `connect()`), si le serveur l'accepte, le client devra choisir un nom pour s'identifier des autres utilisateurs puis il l'envoie au serveur pour que ce dernier vérifie si ce nom est déjà pris ou non. S'il est déjà utilisé le serveur s'occupe du changement du nom et ensuite le diffuse à tous les utilisateurs connectés (c-à-d `reject()`), sinon il le diffuse directement sans le changer (c-à-d `accept()`).

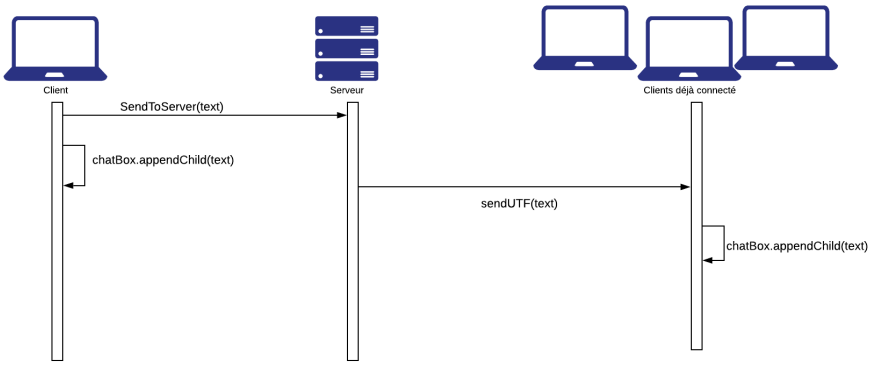

FIGURE 12 – *Envoi d'un message -Diagramme de séquence-*

La figure 12 détaille l'envoi d'un message par un client. Pour qu'un client transmette un message aux autres utilisateurs, il envoie une `socket` contenant le `text` du message au serveur, et ce dernier se charge de la transmission du message aux autres utilisateurs (c-à-d `sendUTF(text)`).

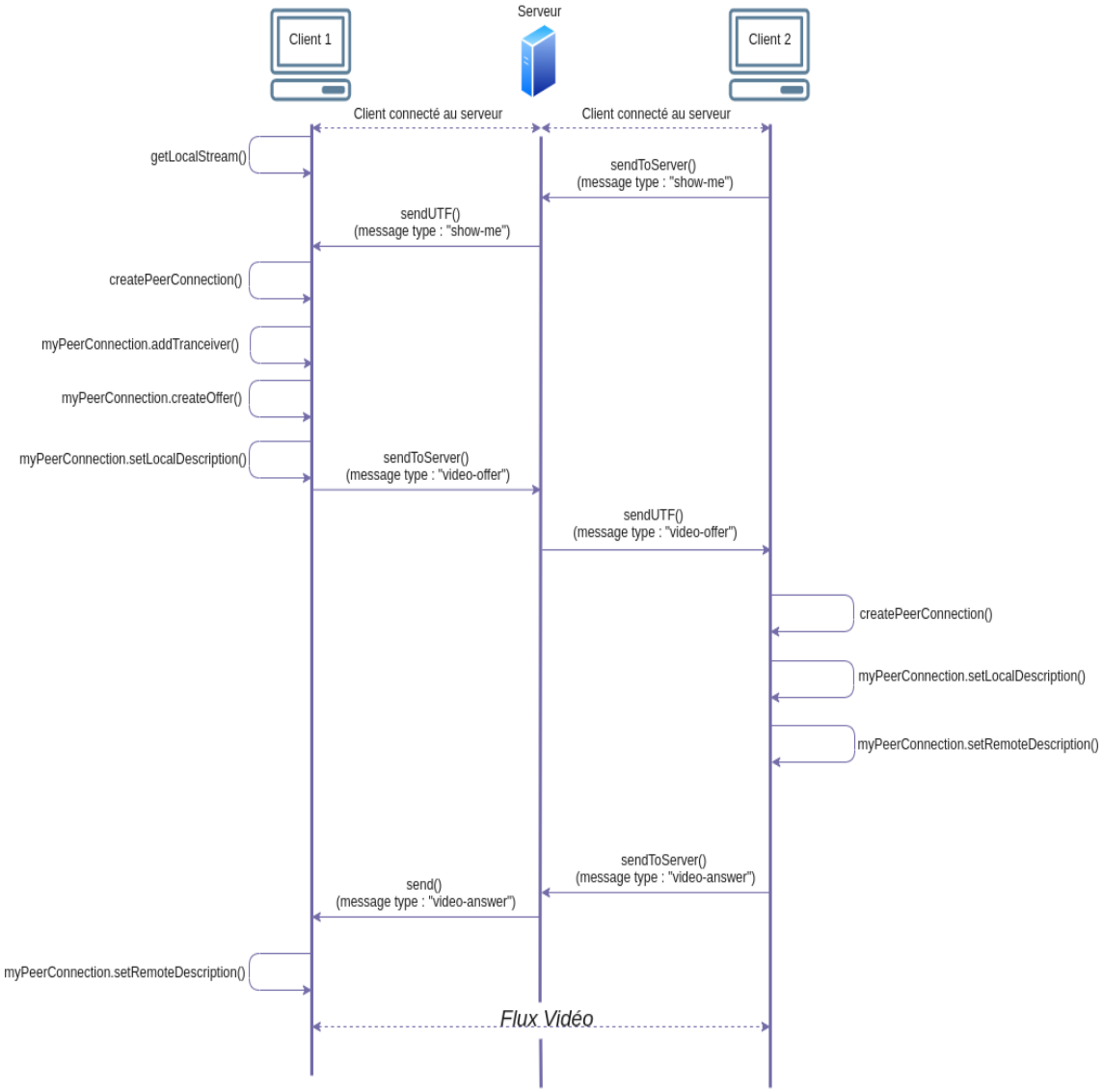

FIGURE 13 – *Établissement de la connexion Peer-to-Peer pour le partage d'écran -Diagramme de séquence simplifié-*

La figure 13 illustre l'établissement d'une connexion *peer-to-peer*, après que le client 2, qui est connecté au serveur, initie un appel avec une autre personne connectée en cliquant sur son pseudo figurant dans la barre des participants. Cela engendre, du côté du client cliqué (1), la création d'une *PeerConnection* (`createPeerConnection()`), l'ajout de son flux à cette dernière, puis l'envoi d'une offre video contenant son **SDP** local (Session Description Protocol : contenant des informations locales telles que les protocoles supportés pour l'audio ou la vidéo) via le serveur de signalisation. Ayant reçu cette offre le client 2 crée sa *Peer Connection*, la configure de la même manière puis envoie une réponse contenant son **SDP** local à son tour au client 1 pour l'ajouter à sa *Peer Connection*. Ensuite d'autres échanges des *ICE Candidate* (non représentés par le digramme) se font pour établir la connexion *Peer-to-Peer* indépendante du serveur. Et à partir de là, le flux passe entre les deux clients indépendamment du serveur.

5 Tests

Chaque semaine, **nous nous retrouvions en présentiel pour tester l'avancement de notre projet**. Nos tests consistaient à reproduire en temps réel les cas d'usage de notre système de partage d'écran. Il s'agit de se retrouver dans une salle de TD avec plusieurs machines connectées à un même réseau filaire et utiliser le système de partage comme il serait utilisé par un professeur et des étudiants.

Chaque scénario de test permet de tester une fonctionnalité comme l'audio, la vidéo ou la messagerie. L'intérêt de se réunir pour cela est que les tests soient faits sur différents supports ce qui permet de détecter des bugs qui ne concernent que certains navigateurs par exemple.

- Un scénario de test pour vérifier la fonctionnalité audio :

  1. Utilisateur 1 lance le serveur et s'y connecte avec le navigateur Chrome par exemple.
  2. Utilisateur 2 se connecte au serveur avec différents navigateurs et lance un appel avec Utilisateur 1.
  3. Vérification de l'audio des deux utilisateurs et des boutons qui permettent de les rendre muets pour chaque navigateur.

  Résultats du test :
  Ce test permet de détecter un bug de compatibilité de notre système avec différents navigateurs. En effet, un utilisateur connecté avec le navigateur Firefox reçoit l'audio mais n'émet aucun son.

- Un scénario de test pour vérifier la fonctionnalité chat :
  Nous pouvons remarquer que plusieurs tests peuvent être effectués en simultané sur une même session. En effet, le scénario précédent regroupant Utilisateur 1 et Utilisateur 2 peut par la même occasion permettre le test de la fonctionnalité chat.

  1. Utilisateur 1 envoie le message "Hello World"
  2. Utilisateur 2 vérifie qu'il reçoit le message.
  3. Utilisateur 2 envoie des messages plus spécifiques : il s'agit des messages commençant par "!md" comme "!md (saut de ligne) :smile :" et des commandes commençant par "./" comme "./mute".
  4. Utilisateur 1 vérifie qu'il reçoit un message au format Markdown ou bien que la commande a bien été exécutée (exemple : l'interlocuteur a bien désactivé le son).

Résultats du test :

Ce test a été très utile pour arriver à une fonctionnalité de chat complète de manière efficace. Tout au long du développement de cette fonctionnalité, les messages critiques ont permis de détecter les bugs très rapidement ce qui accèlère le processus.

- Un scénario de test pour vérifier le changement de fenêtre :

  1. Utilisateur 1 commence à partager son écran n°1 et Utilisateur 2 l'observe.
  2. Utilisateur 2 vérifie que l'écran qu'il reçoit est le bon.
  3. Sans quitter l'appel, Utilisateur 1 change de fenêtre. Il peut le faire soit par le bouton **Changer de fenêtre** soit par la commande `./switch`.
  4. Utilisateur 2 vérifie qu'une nouvelle fenêtre est partagée.

  Résultats du test :

  On vérifie par ce test que d'une part le bouton **Changer de fenêtre** et la commande `./switch` fonctionnent. D'autre part on se rend compte de si le changement de fenêtre s'opère correctement.

- Un scénario de test pour le partage de flux audio :

  L'audio est une fonctionnalité ajoutée que tardivement. En effet, **le partage de fenêtre n'est pas lié au partage audio en WebRTC**. Nous avons donc privilégié l'implémentation du partage d'écran avant celle de l'audio.

  Une fois rajouté, le partage de flux audio a été testé selon ce scénario :

  1. Utilisateur 1 lance un partage d'écran. Un accès à son microphone est demandé.
  2. Utilisateur 2 observe Utilisateur 1 et vérifie qu'il peut l'entendre.
  3. Utilisateur 1 se mute. Utilisateur 2 ne doit plus pouvoir entendre Utilisateur 1.
  4. Utilisateur 1 réactive son micro, Utilisateur 2 doit de nouveau l'entendre.
  5. Utilisateur 1 et 2 changent tour à tour de navigateur et répètent le scénario.

  Résultats du test :

  Ce test nous a permis d'observer que l'audio ne fonctionnait pas sur Firefox. En effet, lorsqu'un utilisateur de Firefox est observé, aucun son n'est émis. En revanche, les utilisateurs de Chrome peuvent émettre du son. Un utilisateur de Chrome peut être écouté par un utilisateur de Firefox.

- Un scénario de test pour vérifier que l'on peut lancer plusieurs appels en parallèle.

Grâce au retour en présentiel, nous avons pu tester ensemble de lancer plusieurs appels en parallèle. Nous nous sommes rendus compte que dans un premier temps notre serveur n'en était pas capable. Nous l'avons donc implémenté, puis lancé ce scénario de test pour vérifier son fonctionnement.

1. Utilisateur 1 lance un appel avec Utilisateur 2.
2. On vérifie que leur appel fonctionne en lançant les test précédents.
3. Utilisateur 3 et 4 lancent un autre appel en parallèle.
4. On vérifie que l'appel 1-2 fonctionne toujours et on lance aussi les tests précédents sur l'appel 3-4.

Résultats du test :
Ce test est celui que l'on lance à chaque changement dans le code, car il permet de faire face à tous nos cas d'utilisation. C'est un test très complet qui met à l'épreuve toutes nos fonctions client. Il est donc exécuté sans arrêt lors de nos séances en présentiel.

6 Livrable rendu

6.1 Un manuel d'utilisation

Sur la page GitHub du projet, vous retrouverez un Wiki décrivant avec détail comment prendre en main notre application. Il y a notamment les pages suivantes :

— **Installation** pour récupérer le code sur sa machine et installer les dépendances nécessaires.
— **Lancer le serveur** pour une première utilisation du serveur.
— **Se connecter à une session** pour qu'un client rejoigne le serveur précédemment lancé.
— **Partager son écran & son micro** pour partager sa première fenêtre.
— **Rejoindre un appel** pour que plusieurs clients connectés partageant leur écran s'appellent.
— **Envoyer des messages** pour connaître les fonctionnalités de notre chat

Ce Wiki saura répondre à une très grande partie de vos interrogations si vous voulez utiliser notre application. Vous devriez être capable à partir d'une configuration nulle aller jusqu'à la fin du tutoriel pour faire un tour de toutes les fonctionnalités implémentées.

Si vous n'avez pas accès à la page GitHub (car elle est privée), vous retrouverez toute notre documentation dans l'archive, dans le dossier `docs`.

6.2 Fonctionnalités

Dans ce projet on a réussi à implémenter une version de rendu fonctionnelle qui peut être utilisée pour échanger des flux vidéo, audio et textuels entre utilisateurs dans un réseau local. Les fonctionnalités que cette version offre sont :

— **connexion d'un utilisateur au niveau du serveur**, en insérant un identifiant. Afin de distinguer les différents utilisateurs, il est impossible d'avoir plusieurs utilisateurs connectés avec le même identifiant, ce qui nous a poussé à ajouter un entier à la fin d'un identifiant déjà existant.

— **Partager un flux vidéo** par utilisation du bouton en figure 14 :

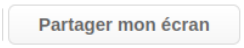

FIGURE 14 – *Bouton pour partager l'écran*

En effet, lorsqu'on appuie sur ce bouton, nous observons comme en figure 15 l'apparition d'une demande de choix de partage du flux audio et vidéo. Concernant le flux vidéo, nous nous intéressons uniquement au partage d'écran.

— **Échanger le flux audio** (sur Google Chrome) : Pendant la mise en connexion entre utilisateurs ils choisissent l'option de partager leur flux audio.

FIGURE 15 – *Autorisation de partage de flux audio*

— **Récupérer le partage d'un utilisateur** : Afin de pouvoir initier une connexion entre deux utilisateurs. Nous avons décidé de générer une liste de participants à laquelle chaque utilisateur a accès et ainsi cliquer sur le nom de cet utilisateur va permettre de récupérer son flux vidéo et audio. Selon différents cas, des messages d'erreur s'affichent, par exemple dans le cas où un utilisateur est déjà en appel ou lorsque l'utilisateur n'a pas d'écran partagé.

— **Activer/Désactiver le partage d'écran** à l'aide du bouton de la figure 16 :

FIGURE 16 – *Bouton pour cacher l'écran*

Lorsqu'un utilisateur décide de cacher son écran, nous constatons l'affichage d'un écran noir pour l'émetteur et le destinataire.

— **Changer l'écran partagé** : Cette fonctionnalité permet à l'utilisateur de changer la fenêtre partagée sans interrompre sa connexion avec un autre utilisateur. Notons que le destinataire recevra un flux vidéo contenant la nouvelle fenêtre.

— **Activer/Désactiver le son** : Ceci est réalisé à travers les boutons `mute / unmute` qui permettent d'activer/désactiver le son sachant qu'elle n'est accessible que lorsqu'un partage est réalisé.

— **Augmenter/diminuer les fps** lors du partage d'écran, dans le but d'économiser la consommation des ressources. Cette fonctionnalité est réalisée par un clic sur le bouton **Mode Développeur** en figure 17 puis il est possible de changer la valeur du **framerate** comme on peut le voir en figure 18.

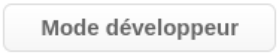

FIGURE 17 – *Bouton Mode Développeur*

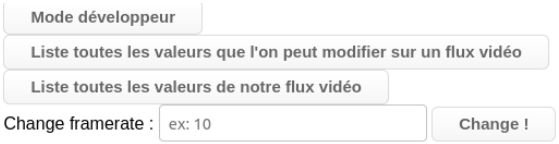

FIGURE 18 – *Liste des fonctionnalités développeur*

Nous avons aussi la possibilité de changer les fps d'un utilisateur avec qui nous sommes en connexion. Cette fonctionnalité est plus intéressante parce qu'elle permet de changer la valeur du flux vidéo reçu et ainsi économiser les ressources la consommation des ressources.

— **Envoyer des messages dans la conversation publique** par écrit dans la zone de texte de la figure 19 et envoyer ce texte aux autres participants par un bouton **Envoyer** produira un message tel que vous pouvez observer en figure 20.

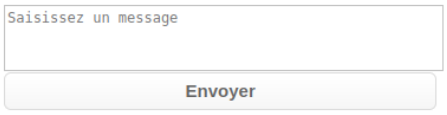

FIGURE 19 – *Zone de texte*

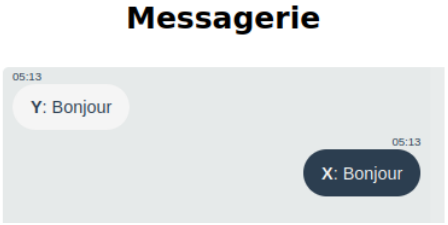

FIGURE 20 – *Messagerie*

Un tel message est récupéré par tous les utilisateurs connectés. D'autres fonctionnalités sont disponibles au niveau de la messagerie telles que :

— **Formatage des messages sous forme Markdown**, afin de rendre la conversation plus lisible et simple à parcourir. Ceci pourrait être aussi utile pour l'envoi de code entre utilisateurs comme on peut le voir en figures 21 et 22. Notons qu'une telle fonctionnalité nécessite comme entête **!md** suivi d'un saut de ligne et ensuite le texte à formater. Ci-dessous un exemple récupéré de la messagerie de notre outil.

````
!md
Salut ! Voici la fonction que
j'ai utilisé pour afficher
***Hello World*** :

```python
def hello():
    print("Hello world!")
```
Dis-moi si tu parviens à
l'utiliser :smile:
````

FIGURE 21 – *Message brut*

FIGURE 22 – *Message formaté envoyé*

— Utilisation des **Emojis** dans la messagerie par utilisation des codes des Emojis préfixés par **!md** comme on peut le voir en figures 23 et 24.

FIGURE 23 – *Message brut avec un code Emoji* `:tada:`

FIGURE 24 – *Message envoyé avec l'Emoji correspondant*

— Possibilité d'exécution de commandes dans la messagerie comme :

- **./share :** Pour partager un écran.
- **./stop :** Pour arrêter le partage d'écran.
- **./switch :** Pour changer la fenêtre partagée.
- **./hide :** Pour cacher le partage.

- **./dark :** Pour changer entre les modes d'affichage clair et sombre.
- **./fps 10 :** Pour changer les fps de la fenêtre partagée
- **./fps user 10 :** Pour changer les fps de l'écran partagé par l'utilisateur *user*.

Dans la figure 25, nous observons **l'interface générée** lors du lancement du serveur et la connexion des clients. Une barre de messagerie est générée à gauche de la fenêtre, une autre barre contenant les clients connectés est positionnée à droite de la fenêtre et le contenu du partage qui est fourni au milieu. Nous constatons dans cet exemple que l'utilisateur 3 est en appel avec l'utilisateur 4, ce dernier ne recevant aucun partage nous mène à conclure que c'est l'utilisateur 3 qui a initié cet appel en récupérant les flux audio et vidéo de l'utilisateur 4. Au niveau de la figure 26 , nous observons le comportement après la récupération du flux partagé de l'utilisateur 3 par l'utilisateur 4.

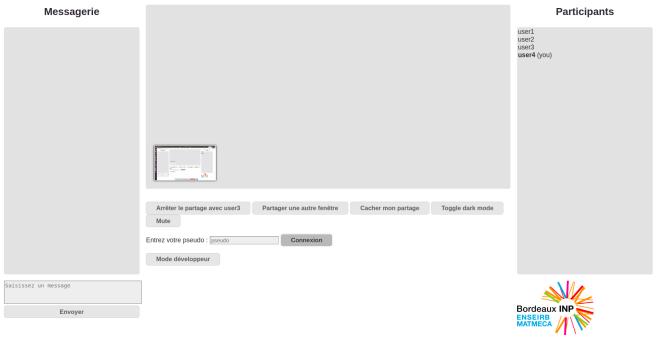

FIGURE 25 – *Interface générée par la connexion au serveur*

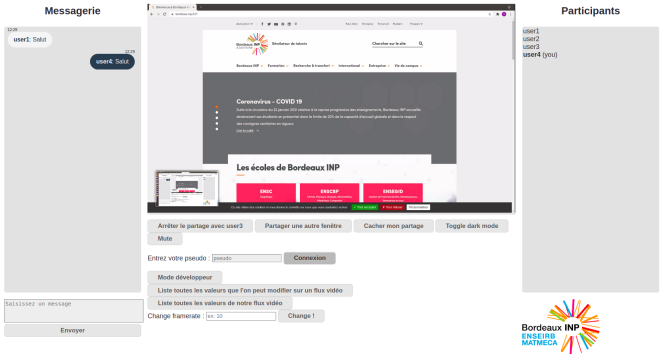

FIGURE 26 – *Interface générée par la connexion au serveur*

L'affichage généré par la console disponible en figure 27 donne des informations sur la communication réalisée entre serveur et entre les différents clients. En effet, nous avons essayé d'afficher les étapes de la communication au niveau de la console à chaque communication. Cette méthode de fonctionnement nous a été très utile puisqu'elle permettait de comprendre les différents problèmes détectés lors des tests.

```
[14:34:41] *** Call recipient has accepted our call                                    (index):3
[14:34:41] *** ICE connection state changed to checking                                (index):3
[14:34:41] {"type":"new-ice-candidate","target":"user1","candidate":{"candidate":"candidate:2913169731 1 udp    (index):3
2122260223 10.168.119.212 54751 typ host generation 0 ufrag aCpk network-id 1 network-cost 10","sdpMid":"0","sdpMLineIndex":0}}
[14:34:41] *** Adding received ICE candidate: {"candidate":"candidate:2913169731 1 udp 2122260223 10.168.119.212    (index):3
54751 typ host generation 0 ufrag aCpk network-id 1 network-cost 10","sdpMid":"0","sdpMLineIndex":0}
[14:34:41] *** WebRTC signaling state changed to: stable                               (index):3
[14:34:41] *** ICE connection state changed to connected                               (index):3
[14:34:41] *** ICE gathering state changed to: complete                                (index):3
[14:34:46] {"name":"user2","target":"user1","type":"stop-sharing"}                     (index):3
[14:34:46] *** Received hang up notification from other peer                           (index):3
[14:34:46] {"type":"onCallList","users":["user1"]}                                    (index):3
[14:34:46] {"name":"user2","target":"user1","type":"end-call"}                        (index):3
** Request to end the call has been received **                                       (index):3
[14:34:46] *** Received hang up notification from other peer                           (index):3
[14:34:46] Closing the call                                                            (index):3
[14:34:46] --> Closing the peer connection                                             (index):3
```

FIGURE 27 – *Exemple de messages générés par la console*

6.3 Voies d'amélioration

Vu la nature du sujet, les fonctionnalités possibles ne sont pas limitées. Ainsi plusieurs fonctionnalités peuvent être améliorées ou ajoutées au niveau de l'outil actuel.

— **Afficher la qualité de la connexion**. Nous voulions avoir pour tous les utilisateurs un moyen de connaître la qualité de connexion ou le débit internet d'un autre utilisateur. Quand un utilisateur a des problèmes de connexion, on peut changer les fps des partages pour lui permettre de profiter de l'échange. En cherchant sur le sujet, nous avons vu que ce genre d'information n'est pas disponible sur tous les navigateurs. En effet, les méthodes `availableIncomingBitrate` et `availableOutgoingBitrate` décrites dans la doc de MDN Web Docs ne sont pas encore disponibles. Il n'y a que `availableOutgoingBitrate` qui est disponible sur Chrome.

— **Le serveur en Ruby**. On a davantage expérimenté sur le WebRTC donc on a surtout travaillé sur des codes en javascript. De plus on a perdu du temps sur des codes Ruby trop compliqués (avec `WebRTC-ruby`). Finalement l'exemple le plus prometteur en Ruby (celui avec **webrick** et **sinatra**), nous ne l'avons découvert que tardivement, nous avons fait alors le choix de ne pas implémenter le serveur en Ruby mais de le laisser en javascript. Ce choix nous assure que le serveur fonctionne puisque nous le maîtrisons davantage.

— **L'audio sur Firefox**. On sait qu'il est possible de partager un flux audio sur Firefox, puisque des applications similaires y parviennent. Cependant nous n'avons pas réussi à l'implémenter. Nous pensons qu'avec plus de temps nous aurions pu trouver comment le faire.

— **Lancer des appels à plusieurs participants**. Nous sommes en effet toujours sur des pairs d'appels. Nous avions une petite idée de comment faire des appels en groupe. Au moment ou deux personnes entrent en appel, nous aurions pu créer une liste des *peerconnections* qui s'agrandit à chaque fois qu'un utilisateur veut se joindre à l'appel. À chaque fois qu'un utilisateur rejoint ou quitte l'appel, on envoie une mise à jour des participants au membres de l'appel.

— Possibilité **d'identifier** des utilisateurs au niveau de la messagerie.

— L'**envoi de fichiers** en pièce jointe dans le chat.

— L'ajout de tout les emojis disponibles en prévisualisation au lieu de devoir taper les codes Markdown correspondants.

7 Retour d'expérience

Ce projet nous a confronté à l'outil **WebRTC** dont nous devions comprendre le fonctionnement, et avec le lequel nous devions expérimenter, pour **découvrir ce qu'il est possible ou**

non de faire en réseau local.

Nous nous sommes rendus compte qu'il est **assez difficile de travailler avec le WebRTC**, que ce soit en réseau local ou non. En effet, **le navigateur utilisé et le support réseau utilisé jouent un grand rôle dans le fonctionnement de cet outil**.

La plupart des problèmes rencontrés étaient dûs au fait que pour tester une fonctionnalité, il faut la **tester sur plusieurs utilisateurs**, sur un réseau filaire et sur un réseau Wi-Fi pour enfin attester que la fonctionnalité a bien été implémentée si nous ne rencontrons pas de problèmes sur nos tests.

Ce genre de **test est assez difficile à réaliser** puisque pour tester le support réseau, on cherche à avoir plusieurs machines ou bien connectées au même réseau Wi-Fi ou toutes reliées au même switch en filaire. Nous ne pouvions donc réaliser ces tests que lors des nos réunions en présentiel, à l'école. Bien sûr, lors de ces séances, nous découvrions plusieurs bugs qui n'apparaissaient pas lorsque l'on réalisait nos tests en *localhost*, sur une seule machine.

Nous étions donc grandement freinés puisque les fonctionnalités ne pouvaient être testées qu'**une fois par semaine**. L'absence de créneau horaire pour ce projet dans notre emploi du temps nous obligeait à organiser des réunions sur notre temps libre et nous n'avions qu'un créneau libre en commun sur la semaine.

7.1 Freins à la mise en oeuvre

7.1.1 Les différences de Navigateur

Lorsqu'on travaille avec le WebRTC, nous devons sans cesse regarder **les compatibilités entre les fonctionnalités que l'on veut utiliser et les navigateurs**. Pour cela les pages sur l'API WebRTC sur le site developer.mozilla.org nous ont été d'une très grande utilité, puisque pour chaque méthode, nous pouvions obtenir un tableau des compatibilités avec les différents navigateurs et leur version.

FIGURE 28 – *Compatibilité des navigateurs pour la méthode* `getUserMedia`

La figure 28 présente les compatibilités entre la méthode `getUserMedia` et les différents navigateurs. On peut apprendre de ce tableau que cette méthode ne sera pas fonctionnelle sur Internet Explorer (il n'est que très peu utilisé donc cela limite le nombre de cas de non-fonctionnement), et qu'il faut au moins :

— la version 53 sur Chrome,

— la version 68 sur Firefox,

— la version 11 sur Safari,

— etc.

Lorsque nous nous renseignions sur les méthodes que nous voulions utiliser, nous regardions si elles étaient disponibles au moins pour les navigateurs les plus utilisés ou s'il existait des méthodes équivalentes pour implémenter plusieurs versions selon le navigateur utilisé.

7.1.2 Les différences des supports réseau

En plus des différences observées entre les navigateurs internet, nous avons observé des petites différences suivant si nous étions en réseau local filaire ou sans-fil.

Par exemple, occasionnellement nous avons remarqué qu'en réseau filaire, **un appel** lancé entre deux utilisateurs **s'arrêtait** sans l'action de l'un des deux utilisateurs. Nous n'avions ni message de fin d'appel dans les logs client ou serveur, ni de message d'erreur. Ce problème a été observé quelques fois sans que nous ayons pu identifier son origine.

Nous n'avons jamais observé ce problème en réseau sans-fil. Nous suspectons un **défaut du matériel** utilisé mais sans pouvoir en être sûrs.

7.1.3 La fragilité de certains outils

Lors de notre expérimentation de **Peerjs**, qui est une bibliothèque qui enveloppe l'implémentation WebRTC pour fournir une API facile à utiliser, nous utilisions le service PeerServer Cloud pour servir les sessions. Cependant, **cet outil a fait preuve d'une certaine fragilité**. En fait, ce serveur tombe de temps en temps en panne ou bien ne répond plus aux requêtes envoyées, et la réalisation des tests n'est plus possible. Cela était un problème assez important car nous nous sommes retrouvé à l'école pour tester notre implémentation mais les serveurs distants ne nous répondant pas, nous ne pouvions rien tester sans un serveur fonctionnel.

7.1.4 La difficulté d'appliquer une méthode agile

Depuis le début du projet, nous nous sommes retrouvés face à la nécessité de **mener plusieurs expérimentations**, pour découvrir le fonctionnement du WebRTC dans un premier temps, et pour établir une version de base sur laquelle nous pouvons construire notre application dans un second temps. Il était donc difficile de tracer un plan agile et définir des **sprints** avec des

tâches bien précises que l'on peut évaluer au cours des semaines. Cependant, avec les retours des réunions que nous organisions dans l'école toutes les semaines, **nous avons essayé d'adopter une méthode presque agile** qui se manifeste dans la réalisation des tests pour évaluer ce qui a été fait pendant la semaine précédente et la définition des objectifs pour les semaines qui suivent. Nous nous sommes contentés de l'outil *issues* de Github pour organiser ces objectifs sous forme de tâches, assigner des personnes, visualiser l'avancement de la réalisation et partager les difficultés rencontrées avec le reste de l'équipe. Cet outil, malgré son arrivée un peu tardive dans notre méthode d'organisation nous a permis de transformer les tâches qui ont été définies en des **User Stories** et visualiser leur avancement tout comme dans un réel **backlog**.

7.1.5 La difficulté de prise en main de Ruby

L'un des besoins du projet était d'implémenter un serveur en Ruby, qui permettra de facilement intégrer notre application dans la forge de l'école. Cependant, il nous a été difficile d'apprendre à utiliser ce langage pour réaliser des programmes avancés comme un serveur HTTPS. Nous avons réussi à trouver certains morceaux de code qui faisaient presque la même chose mais qui manquaient de documentation. Nous n'avons donc pas pu les adapter à notre situation. Nous sommes parvenus à comprendre qu'il était possible de réaliser des serveur HTTP avec le framework **sinatra** mais n'avons pas pu adapter ce serveur à notre projet, qui lui, nécessite un serveur HTTPS. Nous avons, dans une étape avancée de la réalisation de notre application, compris que la prise en main de Ruby et savoir trouver la bonne documentation nécessitent du temps. Nous avons donc préféré investir le temps qui nous restait dans l'amélioration des fonctionnalités que notre application implémente.

7.2 Enseignements à tirer de cette expérience

Tout projet étant étroitement lié au contexte de sa réalisation, le but ultime d'un retour d'expérience est d'aboutir à des enseignements qui permettront d'améliorer la gestion des projets futurs.

7.2.1 Le travail en équipe sur un projet complexe

Ce projet se distingue des projets que nous avons réalisés précédemment au fur et à mesure de notre enseignement par son **effectif** et sa **complexité**. En effet, relativement à notre expérience avec les projets précédents, le nombre 7 (nombre d'élèves impliqués dans ce projet) peut être considéré comme grand.
Les enseignements que nous avons tirés du travail sur un projet complexe en grand effectif sont les suivants :

— Dès le début du projet, il est important de faire régner **un esprit d'équipe**. Nous avons vite remarqué qu'à la différence d'un projet en effectif réduit, l'esprit d'équipe ne s'installe pas automatiquement. Il est nécessaire de prendre des mesures organisationnelles pour garantir l'implication de chacun dans le projet.

— Pour garantir **une implication constante de l'équipe** sur un projet aussi complexe, il est nécessaire de découper la conception en une suite de petits challenges. Chaque petit challenge est alors découpé en tâches réalisées par des sous groupes de petite taille. Il s'agit d'un moyen simple et efficace pour utiliser l'esprit d'équipe automatique cité auparavant qui s'installe dans une équipe de petite taille.

7.2.2 Le travail sur un projet longue durée

Ce projet se distingue également par sa durée. Les enseignements que nous avons tirés du travail sur un projet longue durée sont les suivants :

— Pour que chaque membre puisse suivre l'avancement du projet pendant toute sa durée, il est nécessaire de bien renseigner chaque modification effectuée. Ceci a été fait en choisissant **des titres de commits judicieux**, en utilisant un **README** qui trace l'avancement du projet et **le système d'issues** sur GitHub qui s'est avéré très utile pour la correction de bugs.

— Il est naturel de constater parfois des baisses de régime. Dans ce cas, il a été important de rappeler à tout un chacun l'intérêt de notre projet d'un point de vue technique et éducationnel. Ceci a été fait par le responsable pédagogique du projet ou par des membres de l'équipe.

8 Conclusion

Ce projet très centré sur le WebRTC nous a demandé un certain temps à consacrer à lire de la documentation pour se renseigner sur son fonctionnement pour ensuite s'en servir. Nous avons remarqué également que cette API n'a pas un fonctionnement identique sur tous les navigateurs et donc qu'il fallait sans cesse tester notre implémentation sur différents cas d'utilisation. Nous avons expérimenté ce qu'il était possible ou impossible de faire en réseau local sur les différents navigateurs afin de proposer une application qui fonctionne sur les navigateurs les plus utilisés.

Nous avons donc réussi à implémenter un serveur permettant aux clients s'y connectant de réaliser les actions suivantes, parfois en gérant les navigateurs au cas par cas :

- Partager leur écran ou la fenêtre de leur choix.
- Partager leur audio, sur Chrome.
- Utiliser un chat pour envoyer des messages aux autres utilisateurs ou envoyer des commandes au serveur pour effectuer des actions comme par exemple pour lancer un partage.
- Régler les fps de son partage ou les fps du partage d'un autre utilisateur, pour réduire les données échangées.

Pour conclure, ce PFA aura été pour nous une première expérience de gros projet en groupe. Il était à la fois beaucoup plus long que tous les projets que nous avions pu rencontrer mais c'est aussi un projet avec un grand nombre de participants. Nous avons donc dû s'organiser différemment et adopter des méthodes de travail pour que le groupe mène à bien ce projet.

9 Annexe

9.1 Architecture détaillée : Fonctions globales au projet

- Fonctions de **chatclient.js** :
  - **getChatclient** : messages.js, peerconnection.js, utils.js, video.js
  - **triggerUpdateWebcamStream** : video.js
  - **handleEnterKeyConnection** : index.html
  - **toggleDevelopperMode** : index.html
- Fonctions de **dark.js** :
  - **switch_mode** : messages.js, index.html
- Fonctions de **messages.js** :
  - **handleSendButton** : index.html
  - **handleEnterKeyMsg** : index.html
  - **isCommand** : chatclient.js
  - **handleCommand** : chatclient.js
  - **formatText** : chatclient.js
- Fonctions de **peerconnection.js** :
  - **getMyPeerConnection** : video.js
  - **updateMyPeerConnection** : video.js
  - **handleOnCallList** : chatclient.js
  - **endCall** : index.html, messages.js
  - **handleEndCall** : chatclient.js
  - **closeVideoCall** : video.js
  - **invite** : chatclient.js
  - **handleVideoOfferMsg** : chatclient.js
  - **handleVideoAnswerMsg** : chatclient.js
  - **handleNewICECandidateMsg** : chatclient.js
- Fonctions de **utils.js** :
  - **log** : chatserver.js, chatclient.js, messages.js, peerconnection.js, video.js
  - **log_error** : chatclient.js, video.js
  - **sendToServer** : chatclient.js, messages.js, peerconnection.js, video.js
  - **reportError** : peerconnection.js
- Fonctions de **video.js**
  - **getVideo** : chatclient.js, peerconnection.js
  - **updateWebcamStream** : peerconnection.js
  - **getLocalStream** : index.html, messages.js
  - **hideSharing** : index.html, messages.js
  - **mute** : index.html, messages.js
  - **handleGetUserMediaError** : peerconnection.js
  - **getConstraints** : index.html
  - **getSettings** : index.html
  - **changeFpsDeveloppper** : index.html
  - **handleChangeFps** : chatclient.js
  - **handleStopSharing** : chatclient.js

9.2 Document de spécification

PFA

Spécification des exigences logicielles

Informatique Deuxième Année

Partage d'écran en réseau "local"

Élèves :
Antoine Pringalle (apringalle@enseirb-matmeca.fr)
Benjamin Upton (bupton@enseirb-matmeca.fr)
Jalal Izekki (jizekki@enseirb-matmeca.fr)
Saad Margoum (smargoum@enseirb-matmeca.fr)
Souhail Nadir (snadir@enseirb-matmeca.fr)
Saad Zoubairi (szoubairi@enseirb-matmeca.fr)
Théo Lelasseux (tlelasseux@enseirb-matmeca.fr)
Zaid Zerrad (zzerrad@enseirb-matmeca.fr)

Enseignant :
David Renault
(renault@labri.fr)

18 / 01 / 2021

1 Introduction et motivation

L'idée de ce projet consiste à utiliser la transmission de flux vidéo pour mettre en place un système de partage d'écran sur un réseau local, comme par exemple sur le réseau de l'ENSEIRB-Matmeca. Un tel système permettrait de discuter facilement entre tandems d'une équipe, ou entre personnes d'un groupe de TD. L'une des applications possibles consisterait à ce qu'un enseignant puisse consulter facilement un ensemble d'écrans correspondant à la liste des personnes présentes dans une salle de TD.

Puisque la plupart des cours ont lieu désormais à distance à cause de la crise sanitaire, le projet proposé a évolué d'un outil qui ne marche qu'en réseau local, en un outil qui marche sur internet. De plus, il ne doit nécessiter que très peu de débit internet pour fonctionner, afin que les TDs puissent être suivis par tous, et ce même avec une mauvaise connexion internet.

1.1 Objectif et portée du document

L'objectif du document de spécification d'exigences logicielles est de décrire le comportement du système par une vision globale de ce dernier d'abord, et ensuite détailler les différentes exigences fonctionnelles. Ce document décrit aussi les exigences non fonctionnelles et les facteurs supplémentaires nécessaires à la description complète et compréhensible des besoins pour le projet.

1.2 Définitions, acronymes et abréviations

Dans la suite du document, le professeur désigne l'organisateur d'une réunion virtuelle. C'est celui qui programme et lance la réunion. L'élève ou l'étudiant sont les participants de la réunion, ils possèdent à priori moins de droits de modération sur la réunion dans un premier temps. Le terme utilisateur regroupe les deux entités précédentes, ce sont toutes les personnes susceptibles de se servir de cet outil.

1.3 Perspectives du projet

Ce projet s'inscrit dans un cadre pédagogique permettant de faciliter l'organisation de TDs, et qui permettra aussi de palier à plusieurs difficultés qui se posent notamment avec l'enseignement à distance. L'outil sujet du projet permettra à un groupe (d'étudiants et d'un professeur) de partager leur écran et leur son éventuellement pendant une séance de TD.

Cet outil diffère des outils déjà présents par un paramétrage plus affiné sur la qualité de sortie vidéo notamment. En effet, pour un partage de code, il n'est pas nécessaire de fournir un flux vidéo de 30 fps, 1 fps ou moins sont suffisants. Ce paramétrage a pour but de limiter le transfert de données coûteuses en terme de débit internet.

Nous ne savons pas encore s'il est possible de changer ces paramètres pendant la conférence ou s'il doivent être définis avant l'appel. Ce projet a donc pour but de tester ce qui est possible ou non de faire sur un outil de communication.

2 Description générale

2.1 Fonctions du produit

L'objectif du projet consisterait à mettre en place un serveur permettant de faire dialoguer et partager des écrans facilement à des groupes de personnes. Ainsi, on s'intéresse à donner la possibilité de passer des appels entre encadrants de TP et élèves, dans lesquels tout le monde peut partager son écran. Un des objectifs principaux est de réaliser ce partage à moindre coût réseau.

2.2 Caractéristiques des utilisateurs

Les utilisateurs seront des étudiants et des professeurs de l'ENSEIRB-Matmeca.

3 Contraintes de conception

3.1 Être programmé en Ruby

Nous devons faciliter l'adaptation de notre application avec la forge de l'école. Ainsi, le serveur doit être codé en Ruby.

3.2 Utiliser du WebRTC

4 Exigences spécifiques

4.1 Fonctionnalités

4.1.1 Partager son écran

- **Partager à une seule personne ou à un groupe.** Notre outil doit permettre à un étudiant de partager son écran à un groupe, qui serait une salle de classe virtuelle. Un étudiant doit pouvoir choisir entre partager son écran à une salle ou bien seulement à une personne lors d'un appel privé.
- **Pouvoir faire des sous-groupes.** Dans notre utilisation, il serait intéressant de pouvoir diviser une salle de classe en plusieurs petits groupes pour des travaux de groupe en TD. Notre salle doit donc pouvoir se subdiviser et chaque étudiant doit pouvoir joindre la salle qu'il veut.

4.1.2 Paramétrer l'appel

- **Paramétrer les FPS.** C'est l'objectif premier de ce travail, nous devons pouvoir régler la fréquence des images envoyées et reçues pour que l'appel soit le plus économe en données échangées si besoin (si un étudiant a une connexion faible, il doit pouvoir suivre une conférence et partager lui aussi son écran).
- **Paramétrer l'écran partagé.** Nous devons permettre à l'utilisateur d'activer ou déactiver son partage d'écran. Le partage d'écran est initialement désactivé pour l'utilisateur, c'est à lui de l'activer et choisir les fenêtres à partager.
- **Paramétrer les micros.** Nous devons permettre à l'utilisateur d'activer ou déactiver son micro.
- **Paramétrer l'interface** (nombre d'écrans, agencement, choisir quel écran on regarde...). L'utilisateur doit pouvoir sélectionner l'utilisateur dont il souhaite afficher l'écran. Il peut

aussi choisir d'afficher une grille de tous les écrans. Initialement, les écrans qui s'affichent en premier sont de ceux qui sont en train de parler.

— **Réduction de bruit** (noise cancellation). Un paramètre qui réduirait le bruit ambiant autour d'un étudiant pour lui permettre de travailler avec d'autres sans les gêner (ou sans qu'il ait à se mute à chaque fois qu'il ne parle pas).

— **Choisir ses périphériques d'entrée et de sortie pendant l'appel** (pas besoin de déco-reco). L'utilisateur doit pouvoir choisir son entrée audio (micro), son entrée vidéo (quel écran ou quelle fenêtre) et sa sortie audio (haut parleur, écouteurs ou bien casque) sans avoir à quitter ou actualiser la page.

4.1.3 Messagerie

— **Envoyer des messages au groupe de la réunion ou en privé.** Autrement que de parler vocalement, les utilisateurs doivent pouvoir échanger par message, soit sur une conversation de la salle, soit en privé à un autre utilisateur.

— **Pouvoir mentionner d'autres utilisateurs.** Il serait intéressant pour les étudiants de pouvoir mentionner (*@Équipe1* ou *@Paul Dupont*) d'autres étudiants pour leur dire de rejoindre leur sous-groupe vocal dans le cadre d'un TD ou bien pour faire venir le professeur afin qu'il réponde à une question.

— **Réagir aux messages.** Cette fonctionnalité permettrait de rendre un chat écrit plus vivant en autorisant les participants à réagir à un message posté par un emoji.

— **Répondre à un message.** Il est également très utile de pouvoir répondre à un message pour clarifier une conversation avec plusieurs interlocuteurs.

— **Pouvoir envoyer des messages en Markdown.** Avoir la possibilité d'envoyer des messages qui commençant par un une suite de symboles qui reste à définir pourrait permettre d'activer un affichage Markdown sur le chat vocal.

— **Possibilité d'envoyer des commandes dans la conversation pour effectuer certaines actions.** Certaines de ces commandes nécessiteront des droits de modérateur pour être exécutées. Parmi les commandes possibles :
  — Création d'un sondage rapide (Ex. `/poll A B C D`).
  — Envoi d'un message à un sous-groupe en particulier (Ex. `/msg @equipe1 "Revenez dans le salon principal"`).
  — Couper le micro de tous les étudiants (Ex. `/muteall`).

— **Envoi d'images.** Pouvoir copier/coller des images dans la messagerie, ou en importer depuis son ordinateur peut permettre aux étudiants de partager des captures d'écran. Cette option devrait être modérée pour limiter l'envoi de fichiers trop volumineux.

— **Envoi de fichiers.** Plutôt que d'envoyer le code avec un copier coller, on devrait pouvoir directement joindre des fichiers à la classe, au groupe ou au sous-groupe. Cette option devrait être modérée pour limiter l'envoi de fichiers trop volumineux.

4.1.4 Afficher des statistiques

— **Montrer les fps.** Il est intéressant de voir à quelle fréquence d'images le partage d'écran est fait au cours d'un appel. Cette option peut être affichée dans un des coins de l'écran et pourquoi pas désactivable dans un menu si elle ne nous intéresse pas.

— **Afficher les noms des utilisateurs.** Il est intéressant de voir la liste des utilisateurs

présents. Ainsi, afficher les identifiants des utilisateurs permettrait de faciliter la discussion.

— **Montrer le nombre de gens connectés.** Dans une salle de classe il est important de savoir rapidement pour l'utilisateur combien d'élèves sont connectés et possiblement déconnectés. L'affichage de ce paramètre peut être disponible pour tous cependant.

— **Montrer la qualité de connexion de chaque participant.** Cette statistique permet de voir quels sont les élèves qui ont une connexion internet instable, et de régler le paramètre des fps afin de permettre à tous de suivre le cours convenablement.

— **Montrer qui est l'encadrant / administrateur du TP.** Cette distinction peut se faire par un petit symbole à côté de son nom ou bien un cadre d'une couleur particulière autour de son écran. Cela faciliterait la recherche de cet encadrant pour sélectionner rapidement son écran quand il souhaite expliquer une notion utile au cours.

4.2 Product backlog

Un **Product backlog** est une liste des fonctionnalités priorisées qui sont à développer ou améliorer dans le cadre d'un projet informatique. Notre product backlog est représenté dans cette partie par une liste de user-stories organisées de la plus importante vers la moins importante.

— En tant qu'utilisateur, je veux partager mon écran afin de pouvoir expliquer certaines choses aux autres utilisateurs.

— En tant qu'utilisateur, je veux échanger par audio afin de faciliter la communication.

— En tant qu'utilisateur, je veux pouvoir envoyer des messages dans la conversation publique/privée afin de prendre des notes et communiquer en cas de présence de problèmes de micro.

— En tant qu'utilisateur, je veux pouvoir augmenter/diminuer mes fps quand je veux lors du partage d'écran, afin d'économiser la consommation des ressources.

— En tant qu'utilisateur, je veux pouvoir activer et désactiver le partage d'écran quand je veux, afin de mieux gérer la session. En tant que prof, je veux pouvoir activer et désactiver le micro de n'importe qui quand je veux, afin de mieux gérer les droits.

— En tant qu'utilisateur, je veux pouvoir voir les statistiques (fps, débit de connexion, informations sur la session) de chaque personne sur l'interface. Et ce, afin de détecter les problèmes de connexions.

— En tant qu'utilisateur, je veux pouvoir disposer d'une fonctionnalité de réduction de bruit, afin de filtrer les bruits indésirables.

— En tant qu'utilisateur, je veux pouvoir paramétrer l'interface (choisir quel écran regarder ou les afficher tous en grille). Et ce, afin de mieux suivre les explications.

— En tant qu'utilisateur, je veux pouvoir formater mes messages sous forme Markdown, afin de rendre la conversation plus lisible et simple à parcourir.

— En tant qu'utilisateur, je veux pouvoir réagir aux messages.

— En tant qu'utilisateur, je veux pouvoir envoyer des fichiers ou des images dans le chat, afin d'échanger des fichiers ou des images du code.

— En tant que professeur, je veux pouvoir créer des sous-groupes, afin d'organiser un travail de groupe.

4.3 Spécification des cas d'utilisation

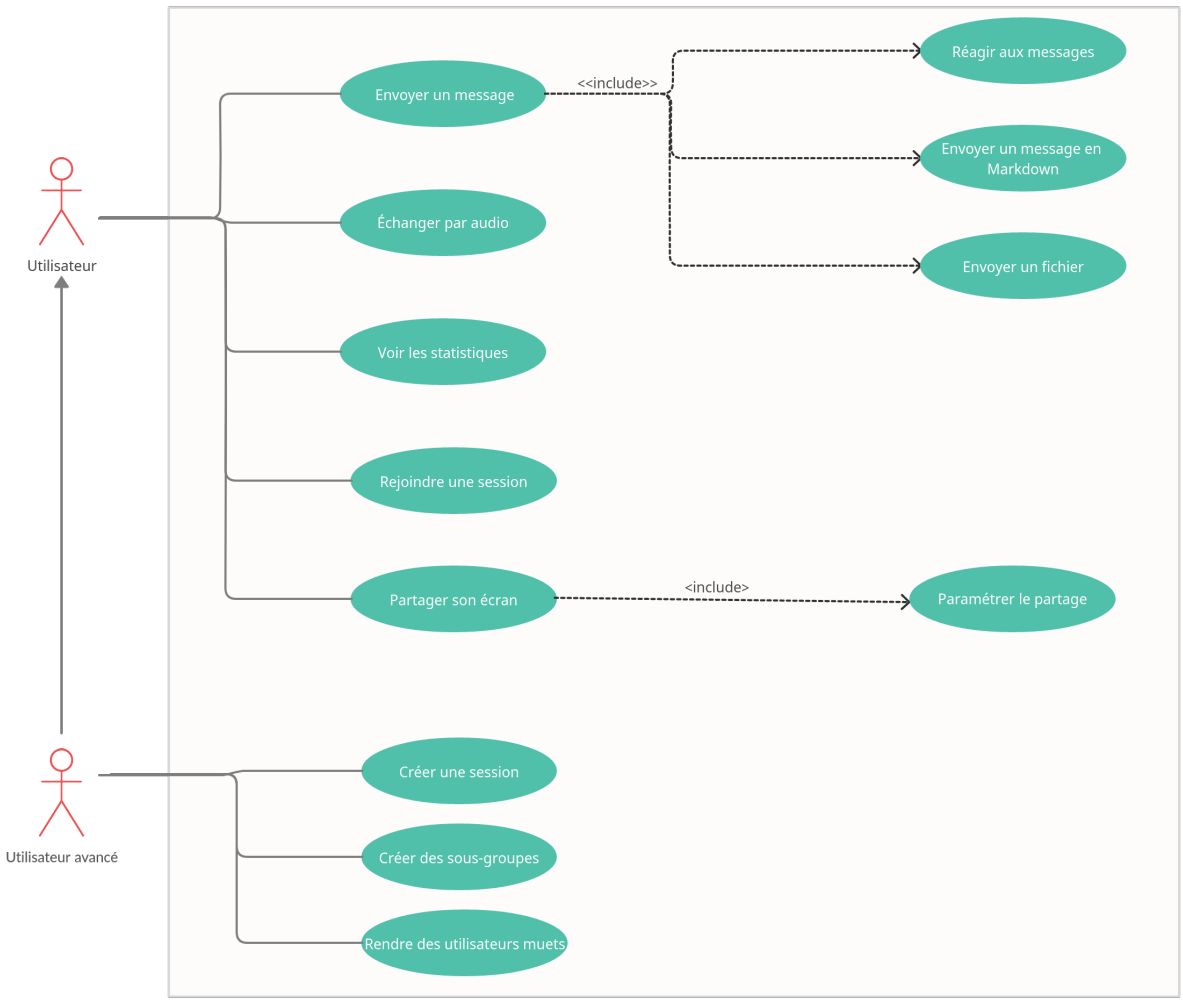

FIGURE 1 – Diagramme des cas d'utilisation

4.4 Exigences supplémentaires

4.4.1 Fonctionnel avec une connexion faible

L'application doit permettre à des étudiants ayant un débit de connexion très limité de suivre un cours. De plus, si l'application tourne sur les serveurs de l'école, il est intéressant qu'elle soit la moins gourmande possible. Ainsi, tout doit fonctionner avec la connexion la plus faible possible.

4.4.2 Capacité

Nous voulons assurer des conférences avec un nombre maximale de personnes dans la mesure du possible. Et ensuite voir quelles sont les réelles limites et où elles vont se trouver (du côté utilisateur, du côté serveur ou bien directement les applications WebRTC).

5 Architecture

Un client principal et les autres en étoile autour de lui, ils communiquent par messages. Il y a deux types de messages : ceux que les utilisateurs écrivent pour communiquer dans le chat et

ceux que les clients s'envoient pour que l'appel se passe bien et qui sont décrits en figure 2.

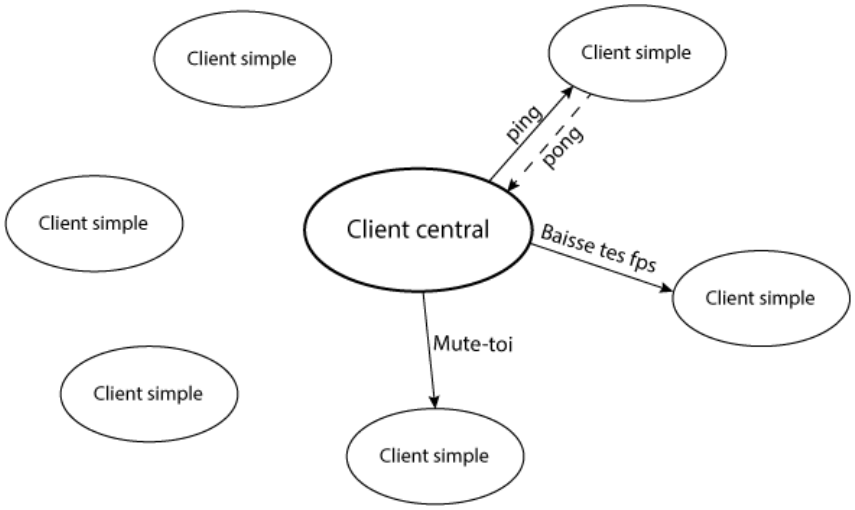

FIGURE 2 – Architecture des communications entre les clients

6 Plan de tests de validation

Scénario de test 1 : Vérifier la fonctionnalité audio

— Vérifier le bon fonctionnement de l'appel si un des utilisateurs ne possède pas de micro. L'utilisateur "Utilisateur sans micro" lance un appel et le message "micro non détecté" s'affiche sur son écran. L'appel se déroule normalement pour les autres utilisateurs possédant un micro.

— Vérifier que lors d'un appel chaque interlocuteur peut parler et entendre. "Utilisateur 1" et "Utilisateur 2" lancent un appel et communiquent en audio.

— Vérifier que le son n'est pas entendu si le micro de l'interlocuteur est coupé. "Utilisateur muet" lance un appel et coupe son micro. Les autres utilisateurs n'entendent pas "Utilisateur muet".

Scénario de test 2 : Vérifier la fonctionnalité chat

— Vérifier qu'un message envoyé par un utilisateur est bien reçu par les autres utilisateurs dans l'appel. "Utilisateur du chat" envoie le message "hello" sur le chat d'un appel et tous les autres utilisateurs reçoivent ce message.

— Vérifier que l'envoi des images, fichiers et textes avec caractères spéciaux se fait correctement. "Utilisateur du chat" envoie une image et le message "HLlÕ " sur le chat ainsi qu'un fichier texte "test.txt" vide. On vérifie que tous les éléments envoyés sont bien reçus.

Scénario de test 3 : Vérifier la fonctionnalité vidéo

— Vérifier que le partage d'écran concerne bien la fenêtre choisie par l'utilisateur. L'utilisateur "Screenshare" partage un terminal. Le terminal partagé et uniquement ce terminal est bien visible par tous les utilisateurs de l'appel.

— Vérifier que le paramètrage du fps est fonctionnel. L'utilisateur "Screenshare" choisit le paramètre 10 pour le fps de son partage d'écran. On vérifie que le fps est bien de 10 en sortie.

— Vérifier que la fenêtre partagée peut être visionnée par tous les autres utilisateurs dans l'appel ayant le droit. On ajoute les utilisateurs "Utilisateur avec droit" et "Utilisateur sans droit" à l'appel. On vérifie que seul "Utilisateur avec droit" peut visionner l'écran

partagé par l'utilisateur "screenshare".

N°	Action	Attendu
1	Lancer un appel	L'appel se lance
2	Partager son écran	L'interlocuteur voit notre écran
3	L'interlocuteur partage son écran	On voit son écran
4	Parler	L'interlocuteur nous entend
5	L'interlocuteur parle	On l'entend
6	On écrit un message dans le chat	L'interlocuteur le reçoit
7	L'interlocuteur écrit un message dans le chat	On le reçoit
8	Envoyer un fichier dans le chat	L'interlocuteur le reçoit
9	Envoyer une image dans le chat	L'interlocuteur la reçoit
10	Rejoindre un sous-groupe	sous-groupe rejoint
11	Modifier la valeur du fps	fps modifié
12	Choisir la fenêtre à partager	L'interlocuteur voit la fenêtre choisie

7 Planning prévisionnel : Diagramme de Gantt

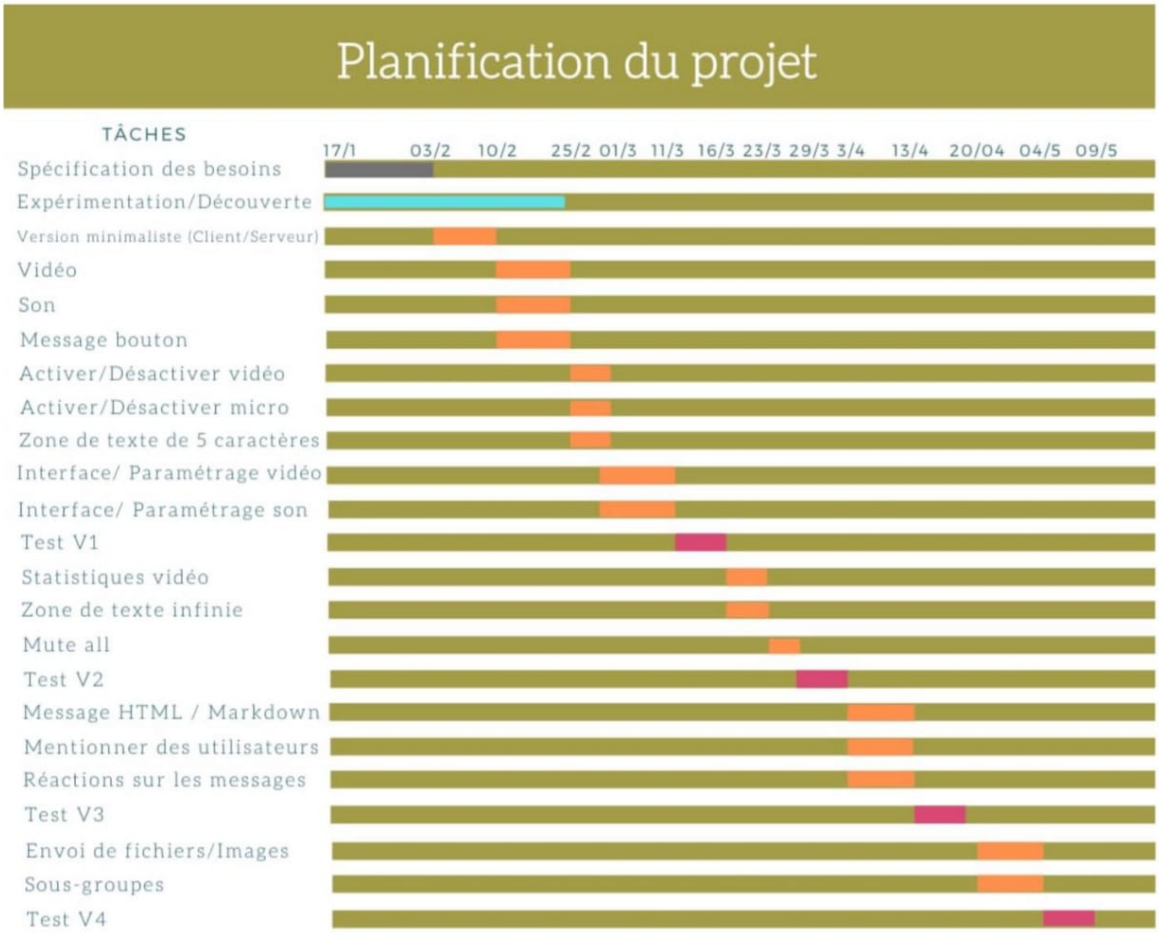

FIGURE 3 – Planification du projet

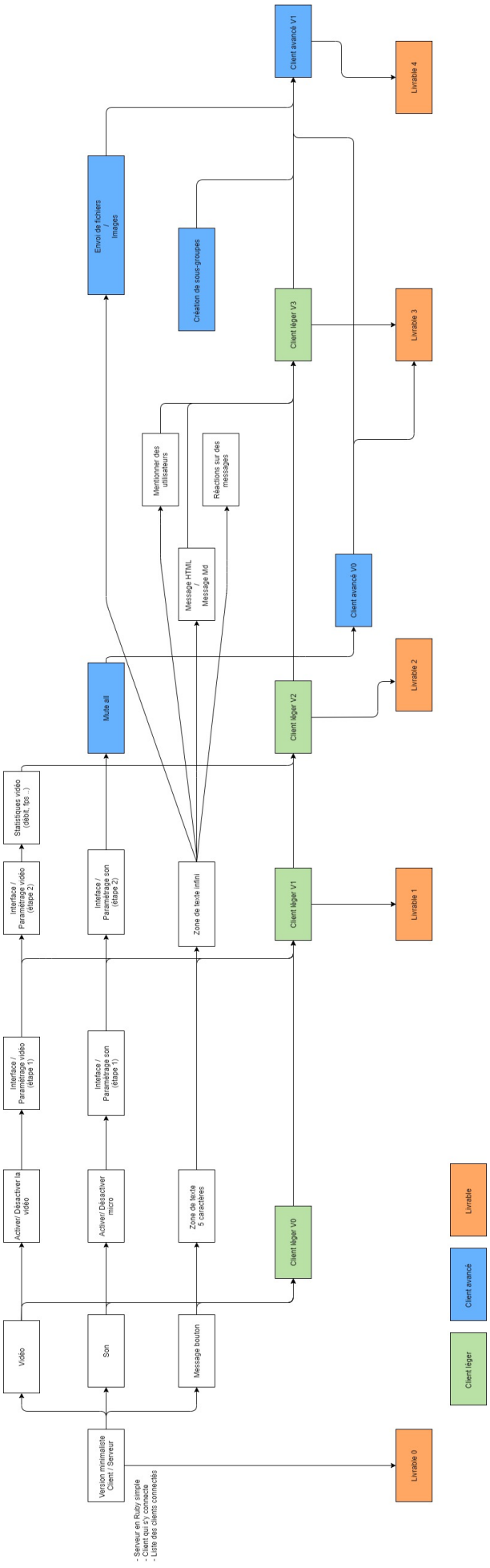

FIGURE 4 – Dépendances des tâches