Rapport SAE5.ROM.03

PLUVIOSE Louis - KARAPETYAN Mikhail

Application de communication WebRTC

Sujet proposé par : Monsieur Philipe Hensel

Université de Haute-Alsace
Institut Universitaire de Technologie de Colmar
Département Réseaux et Télécommunications

10 décembre 2023

Table des matières

LISTINGS

Table des figures

1 Introduction

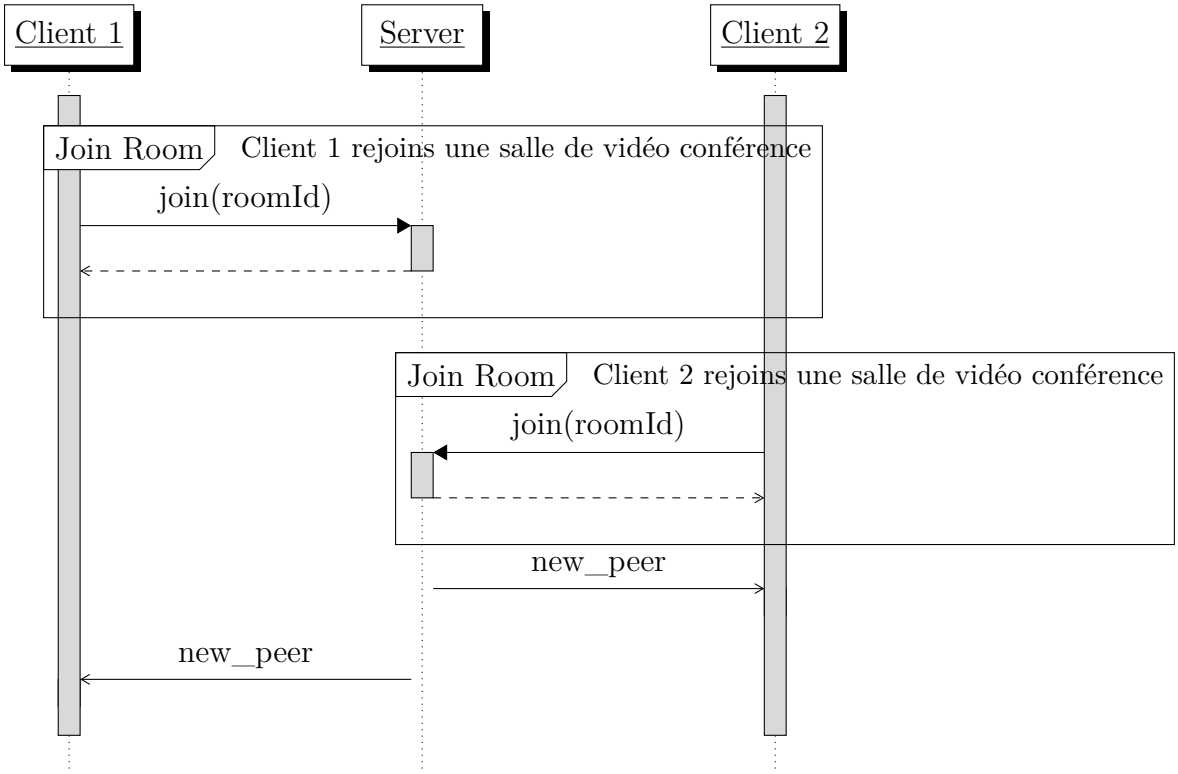
Dans un monde de plus en plus connecté, la communication en temps réel via Internet est devenue une nécessité cruciale, tant pour les interactions personnelles que professionnelles. Notre projet pour la SAE5.ROM.03, s'inscrit dans cette dynamique en offrant une solution de communication basée sur la technologie WebRTC (Web Real-Time Communication). Nous avons voulu via ce projet nous lancer un défi : celui de permettre des échanges audio et vidéo en temps réel directement depuis le navigateur web, sans nécessité de télécharger des logiciels tiers ou de créer des comptes d'utilisateur.

La technologie WebRTC, un standard ouvert et gratuit, permet de réaliser des appels vidéo et audio de haute qualité avec une faible latence, garantissant ainsi une communication fluide et efficace. Notre application SAE5.ROM.03 est conçue pour être intuitive et facilement accessible, offrant une interface utilisateur élégante et des fonctionnalités adaptées à divers contextes, que ce soit pour des réunions, des sessions de travail collaboratif, ou des conversations personnelles.

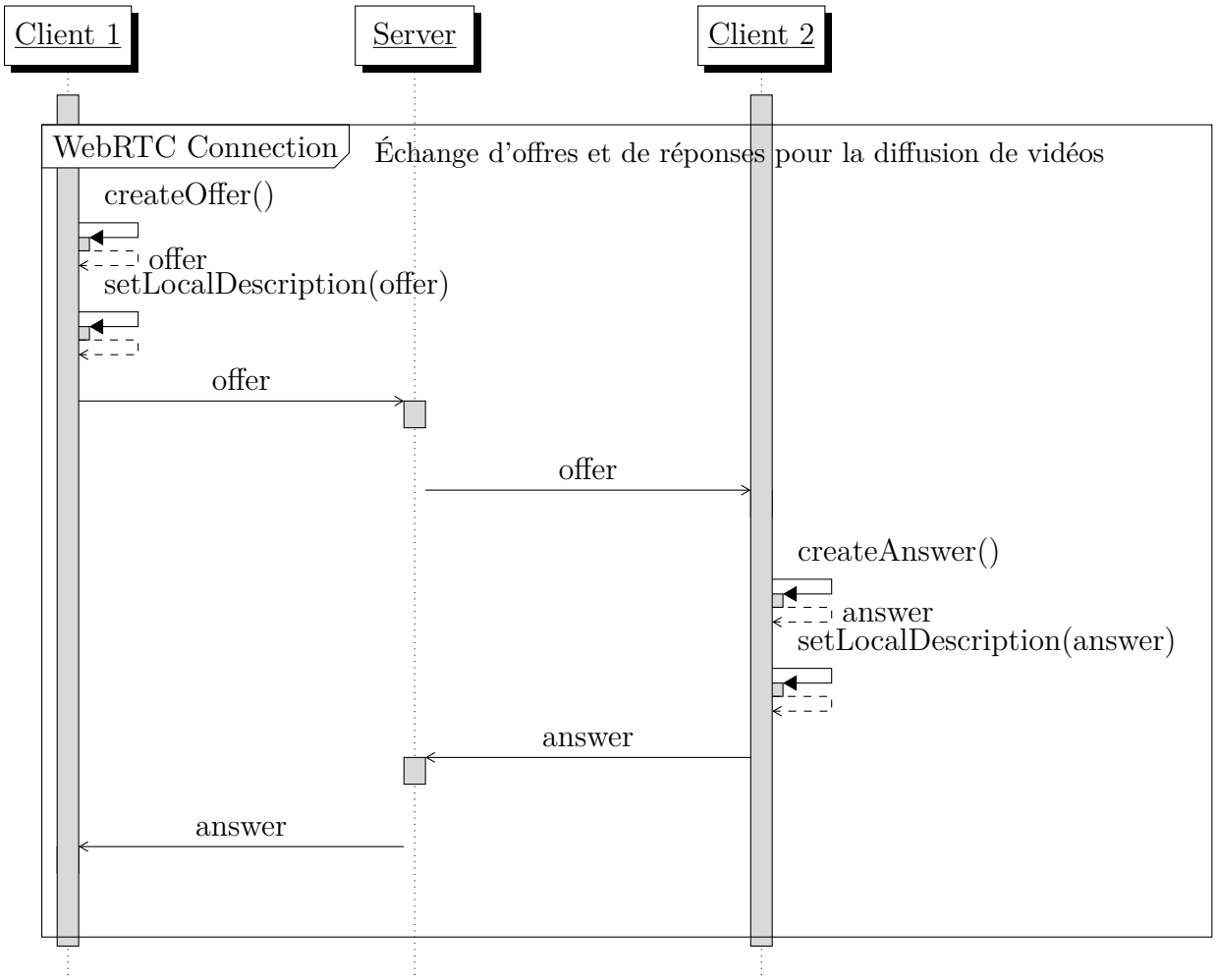
L'accent est mis sur la facilité d'utilisation. Les utilisateurs peuvent créer et rejoindre des salles de conférence virtuelles en quelques clics, tout en bénéficiant d'une connexion fiable. De plus, l'application prend en charge plusieurs participants.

2 Diagrammes de séquence

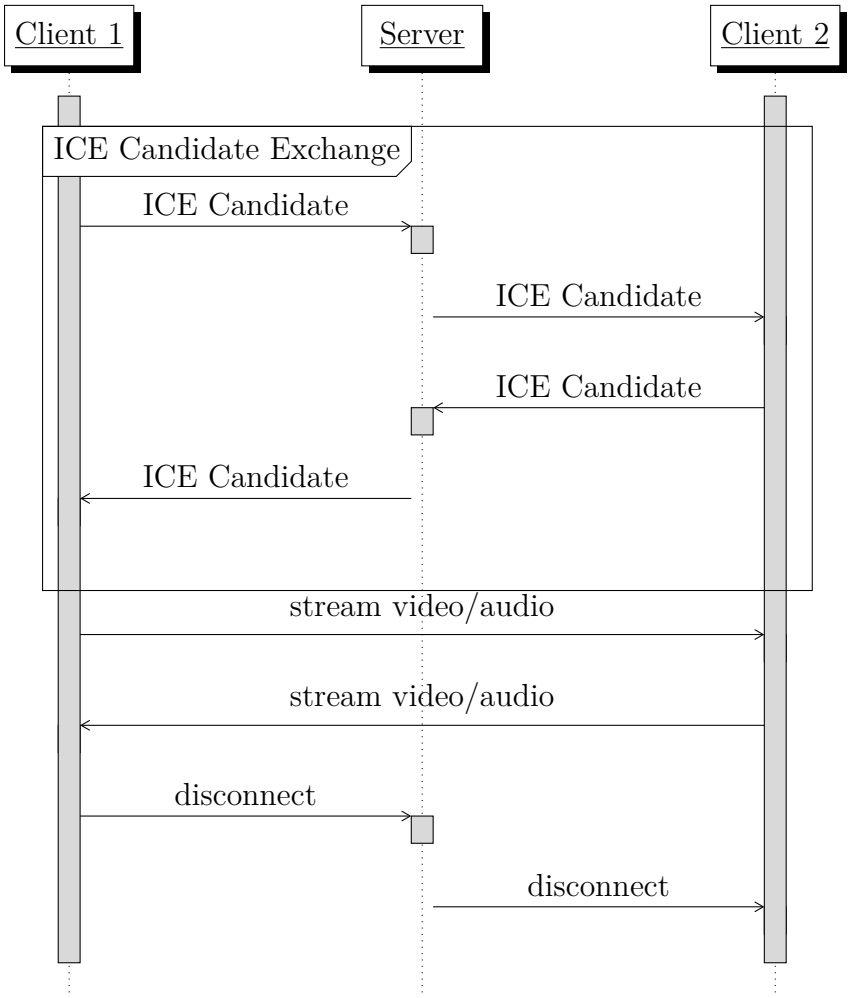
2.1 Connexion à la salle de réunion



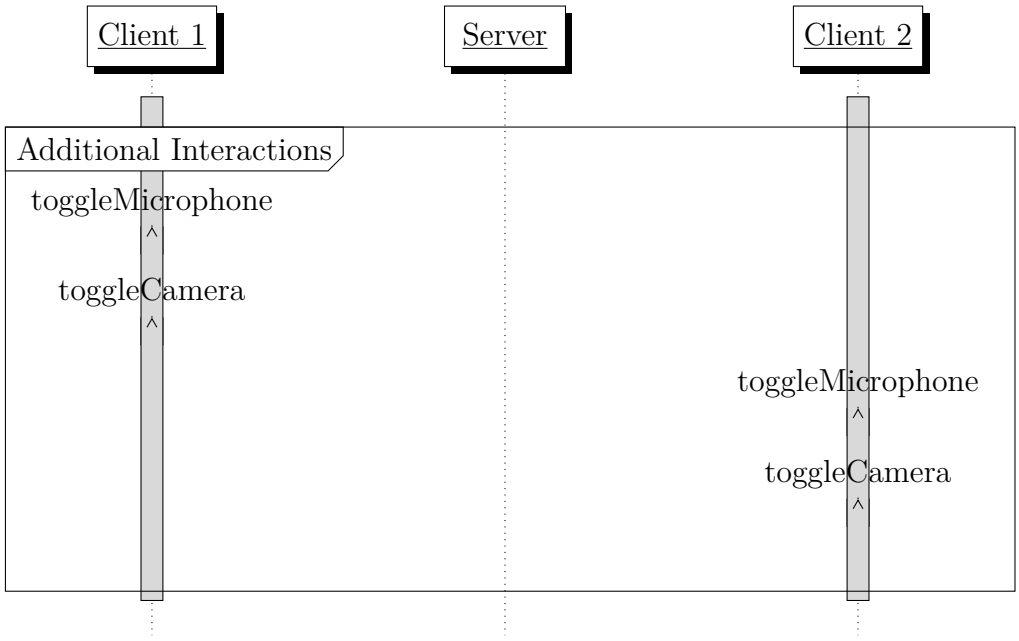
2.2 Connection WebRTC



2.3 Échanges de candidats ICE



2.4 fonctionnalités supplémentaires



3 Utilisation de l'application

4 Fonctionnement de l'application de réunion

Le code relatif à l'application est disponnible sur GitHub à l'adresse suivante : et en annexe de ce document.

4.1 DOM Elements

- `const roomSelectionContainer = document.getElementById('room-selection-container` Cette ligne crée une constante roomSelectionContainer et lui attribue l'élément DOM (Document Object Model) dont l'identifiant (ID) est 'room-selection-container'. L'élément récupéré est un conteneur dans le code HTML où les utilisateurs peuvent sélectionner ou entrer des informations pour rejoindre une salle de chat vidéo.
- `const roomInput = document.getElementById('room-input');` : cette ligne crée une constante roomInput qui référence l'élément DOM avec l'ID 'room-input'. Cet élément est un champ de saisie où les utilisateurs peuvent entrer le nom ou le numéro de la salle qu'ils souhaitent rejoindre.
- `const connectButton = document.getElementById('connect-button');` : Cette ligne crée une constante connectButton qui référence un élément DOM, probablement un bouton, avec l'ID 'connect-button'. Ce bouton est utilisé pour initier la connexion à la salle de chat vidéo après que l'utilisateur a saisi les informations nécessaires.
- `const videoChatContainer = document.getElementById('video-chat-container');` : videoChatContainer est défini comme référençant l'élément DOM avec l'ID 'video-chat-container'. Cet élément est le conteneur principal où la vidéo du chat et d'autres éléments d'interface liés au chat vidéo seront affichés une fois la connexion établie.
- `const localVideoComponent = document.getElementById('local-video');` : cette ligne définit localVideoComponent comme une référence à l'élément DOM avec l'ID 'local-video'. Cet élément est utilisé pour afficher la vidéo locale de l'utilisateur, c'est-à-dire, la vidéo capturée par la caméra de l'appareil de l'utilisateur.

En résumé, ce code initialise des constantes pour référencer divers éléments d'une interface de chat vidéo, probablement définis dans le HTML. Ces références sont ensuite utilisées dans d'autres parties du code JavaScript pour gérer la logique de connexion à une salle de chat vidéo et pour contrôler l'affichage et la mise à jour de l'interface utilisateur en fonction des actions de l'utilisateur.

4.2 Variables

— `const socket = io();` : Cette ligne crée une variable socket en utilisant io(), qui est une fonction de la bibliothèque Socket.IO. Socket.IO est une bibliothèque JavaScript pour les applications web en temps réel qui permet une communication bidirectionnelle en temps réel entre les navigateurs web et les serveurs. socket serait utilisé pour gérer les communications en temps réel, comme envoyer et recevoir des signaux pour établir des connexions vidéo.

— `const mediaConstraints = { audio: true, video: { width: 1280, height: 720 } };` : Cette ligne définit les contraintes pour les médias qui seront utilisés dans la communication. Ici, audio : true signifie que l'audio sera activé, et video : width : 1280, height : 720 définit la résolution de la vidéo à 1280x720 pixels.

— `let localStream;` : Déclaration d'une variable localStream. Cette variable sera utilisée plus tard pour stocker le flux de médias local (audio et vidéo) de l'utilisateur.

— `let roomId;` : Déclaration d'une variable roomId qui sera utilisée pour stocker l'identifiant de la salle de chat vidéo à laquelle l'utilisateur se connecte.

— `let peerConnections = {};` : Cette ligne initialise un objet JavaScript peerConnections comme un dictionnaire vide. Ce dictionnaire sera utilisé pour stocker toutes les connexions peer-to-peer (P2P) établies durant la session de chat vidéo. Chaque connexion P2P représente une connexion directe à un autre utilisateur dans le chat vidéo.

— `const iceServers = { ... };` : Cette partie définit un objet iceServers qui contient la configuration des serveurs ICE (Interactive Connectivity Establishment). ICE est utilisé dans les applications WebRTC pour faciliter la connexion entre les pairs à travers différents types de réseaux et de pare-feu/NAT.

À l'intérieur de cet objet, deux types de serveurs sont configurés :

Un serveur STUN (stun :stun.l.google.com :19302), qui aide à découvrir l'adresse IP publique de l'utilisateur.

Un serveur TURN (turn :relay1.expressturn.com :3478 avec un nom d'utilisateur et un mot de passe), qui est utilisé pour relayer le trafic si une connexion P2P directe n'est pas possible.

En résumé, Ce code initialise des variables clés pour la gestion de la communication en temps réel et du streaming vidéo. Il crée une connexion Socket.IO, définit les contraintes des médias (audio et vidéo), prépare des variables pour stocker le flux local et l'ID de la salle, et configure un dictionnaire pour gérer les connexions peer-to-peer. Il définit également des serveurs ICE pour aider à la connectivité réseau dans WebRTC.

4.3 Button Listeners

— `connectButton.addEventListener('click', () => { joinRoom(roomInput.value); });` : Cette ligne attache un écouteur d'événement au bouton connectButton. Lorsque ce bouton est cliqué, la fonction anonyme déclenche l'appel de la fonction joinRoom avec la valeur actuelle de l'élément roomInput comme argument. joinRoom est une fonction qui gère la logique pour rejoindre une salle de chat vidéo spécifique basée sur l'identifiant de la salle fourni.

— `Déclarations des boutons hangUpButton, toggleMicButton et toggleCameraButton` : Ces lignes récupèrent des références à trois boutons de l'interface utilisateur : un pour raccrocher l'appel, un pour activer/désactiver le microphone, et un pour activer/désactiver la caméra.

— `Ajout de gestionnaires d'événements pour les boutons` :

`hangUpButton.addEventListener('click', hangUpCall);` : ajoute un gestionnaire d'événement pour gérer l'action de raccrocher l'appel.

`toggleMicButton.addEventListener('click', toggleMicrophone);` : ajoute un gestionnaire pour activer ou désactiver le microphone.

`toggleCameraButton.addEventListener('click', toggleCamera);` : ajoute un gestionnaire pour activer ou désactiver la caméra.

— `let isRoomCreator = false;` : Cette ligne déclare une variable isRoomCreator et l'initialise à false. Cette variable est utilisée pour suivre si l'utilisateur actuel est celui qui a créé la salle de chat vidéo. Cela peut affecter la logique de l'application, comme qui a le droit de contrôler certaines fonctionnalités de la salle ou d'inviter d'autres utilisateurs.

En résumé, Dans cet extrait, des écouteurs d'événements sont ajoutés à divers boutons de l'interface utilisateur. Ces boutons permettent de rejoindre une salle de chat vidéo, de raccrocher l'appel, d'activer/désactiver le microphone, et d'activer/désactiver la caméra. Une variable isRoomCreator est également définie pour suivre si l'utilisateur est le créateur de la salle.

4.4 Socket Event Callbacks

— `socket.on('room_created', async () => { ... });` : Quand le serveur envoie un message room_created, cela signifie que l'utilisateur a créé une nouvelle salle. Le flux vidéo local est configuré et isRoomCreator est défini sur true.
— `socket.on('room_joined', async () => { ... });` : Ce gestionnaire réagit au message room_joined, indiquant que l'utilisateur a rejoint une salle existante. Le flux local est configuré, isRoomCreator est défini sur false, et un message start_call est émis pour commencer l'appel.
— `socket.on('full_room', () => { ... });` : Réagit au message full_room, indiquant que la salle que l'utilisateur tente de rejoindre est pleine. Affiche une alerte pour informer l'utilisateur.
— `socket.on('start_call', async () => { ... });` : Lors de la réception d'un message start_call, si l'utilisateur est le créateur de la salle, il initie les connexions Peer-to-Peer (P2P) avec les autres participants.
— `socket.on('webrtc_offer', async (data) => { ... });` : Gère les offres WebRTC reçues d'autres pairs. Si le flux local n'est pas encore configuré, il le fait, puis traite l'offre WebRTC.
— `socket.on('webrtc_answer', (data) => { ... });` : Gère les réponses WebRTC reçues en réponse à une offre envoyée précédemment.
— `socket.on('webrtc_ice_candidate', (data) => { ... });` : Gère les messages de candidats ICE, qui sont des informations nécessaires pour établir la connectivité réseau dans WebRTC.
— `socket.on('new_peer', async (peerId) => { ... });` : Réagit à un message indiquant qu'un nouveau pair a rejoint la salle, et initie la configuration d'une connexion P2P avec ce nouveau pair.

En résumé, cet extrait définit plusieurs gestionnaires d'événements pour gérer les communications via Socket.IO dans notre application de chat vidéo WebRTC. Ces événements incluent la création et la jonction à des salles, la gestion de salles pleines, le démarrage d'appels, et la gestion des offres, réponses, et candidats ICE de WebRTC, ainsi que la gestion de l'arrivée de nouveaux pairs.

4.5 Fonctions Principales

1. `joinRoom(room)`
   — But : Afficher l'interface de la conférence vidéo.
   — Fonctionnement : Masque le conteneur de sélection de salle et affiche le conteneur de chat vidéo. Masque également le texte d'attente.
2. `showVideoConference()`
   — But : Afficher l'interface de la conférence vidéo.
   — Fonctionnement : Masque le conteneur de sélection de salle et affiche le conteneur de chat vidéo. Masque également le texte d'attente.
3. `setLocalStream(mediaConstraints)`
   — But : Configurer le flux de médias local.
   — Fonctionnement : Obtient le flux multimédia local selon les contraintes fournies et l'affecte à l'élément vidéo local.
4. `handleNewPeer(peerId)`
   — But : Gérer l'arrivée d'un nouveau pair.
   — Fonctionnement : Crée une connexion peer-to-peer avec le nouveau pair, ajoute le flux local à cette connexion, crée une offre WebRTC et l'envoie au pair via le socket.
5. `setupPeerConnection(peerId, isInitiator)`
   — But : Configurer une connexion peer-to-peer.
   — Fonctionnement : Crée une nouvelle connexion WebRTC avec les serveurs ICE configurés, ajoute des gestionnaires pour les événements track et icecandidate, et si l'utilisateur est l'initiateur, crée et envoie une offre WebRTC.
6. `addRemoteStream(event, peerId)`
   — But : Ajouter un flux distant à l'interface utilisateur.
   — Fonctionnement : Crée un nouvel élément vidéo pour le flux distant ou utilise un élément existant, puis associe le flux distant à cet élément vidéo.
7. `handleIceEvent(event, peerId)`
   — But : Traiter une offre WebRTC reçue.
   — Fonctionnement : Configure la description distante, gère les candidats ICE mis en cache, crée une réponse et l'envoie au pair.
8. `handleAnswer(data)`
   — But : Traiter une réponse WebRTC reçue.
   — Fonctionnement : Met à jour la description distante de la connexion peer-to-peer.
9. `handleIceCandidate(data)`
   — But : Gérer les candidats ICE reçus.
   — Fonctionnement : Ajoute les candidats ICE à la connexion peer-to-peer correspondante.
10. `hangUpCall()`
    — But : Terminer l'appel en cours.
    — Fonctionnement : Ferme toutes les connexions peer-to-peer, arrête le flux local et nettoie l'interface utilisateur.
11. `toggleMicrophone()`
    — But : Activer ou désactiver le microphone.
    — Fonctionnement : Bascule l'état d'activation de la piste audio du flux local.
12. `toggleCamera()`
    — But : Activer ou désactiver la caméra.
    — Fonctionnement : Bascule l'état d'activation de la piste vidéo du flux local.

5 Fonctionnement de l'application de messagerie

5.1 Initialisation et Connexion Socket.IO

— `Écouteur d'événement 'DOMContentLoaded` : Le code s'exécute une fois que le contenu de la page est complètement chargé.
— `Création d'une socket : const socket = io();` : initialise une connexion Socket.IO avec le serveur.
— `Génération d'un identifiant utilisateur : const userId = Date.now().toString();` crée un identifiant unique pour chaque utilisateur en utilisant le timestamp actuel.
— `Connexion à Socket.IO` : L'écouteur socket.on('connect', ...) est déclenché lorsque le client est connecté au serveur Socket.IO.

5.2 Éléments de l'Interface Utilisateur

— `Sélection des éléments DOM` : Les éléments pour le bouton d'envoi, la zone d'affichage des messages, le champ de saisie du nom d'utilisateur et le champ de saisie du message sont récupérés.

5.3 Envoi de Messages

— `Écouteur d'événement pour le bouton d'envoi` : Lorsque le bouton d'envoi est cliqué, le nom d'utilisateur et le message sont récupérés.
— `Validation et envoi du message` : Si le message n'est pas vide, il est envoyé au serveur via socket.emit('chat_message', userId, username, message).
— `Réinitialisation du champ de saisie du message` : Le champ de saisie du message est vidé après l'envoi du message.

5.4 Réception et Affichage de Messages

— `Écouteur d'événement pour 'chat_message'` : Lorsqu'un message est reçu du serveur, il déclenche l'exécution du code suivant :
— `Création de conteneurs pour le message` : Un div pour contenir le message et un autre div pour le texte du message sont créés et stylisés.
— `Identification de l'expéditeur` : Un élément strong est utilisé pour afficher le nom de l'expéditeur en gras. Il affiche "Vous" si l'identifiant de l'expéditeur correspond à celui de l'utilisateur actuel, sinon le nom d'utilisateur de l'expéditeur.
— `Ajout du message` : Le texte du message est ajouté après le nom de l'expéditeur.
— `Stylisation basée sur l'expéditeur` : Les styles sont appliqués différemment pour distinguer visuellement les messages envoyés par l'utilisateur actuel de ceux des autres utilisateurs.
— `Ajout du message à l'interface utilisateur` : Le conteneur de message est ajouté à messagesDiv.
— `Défilement automatique` : L'affichage des messages défile automatiquement vers le bas pour montrer les nouveaux messages.

5.5 Résumé

Ce script crée un client de chat en temps réel. Lorsqu'un utilisateur envoie un message, il est transmis au serveur via Socket.IO, puis distribué à tous les clients connectés. Les messages entrants sont affichés dans l'interface utilisateur avec une distinction entre les messages de l'utilisateur actuel et ceux des autres utilisateurs. Ce script démontre une utilisation typique de Socket.IO pour une application de chat interactive.

6 Fonctionnement du serveur de l'application

6.1 Initialisation d'Express et Configuration SSR

— `Importation des modules nécessaires` : Express, le module HTTP, Socket.IO, et le gestionnaire SSR d'Astro.
— `Création d'une application Express` : const app = express() ;
— `Configuration du chemin de base` : const base = '/' ; définit le chemin de base pour l'application.
— `Servir les fichiers statiques` : Les fichiers générés par Astro sont servis en tant que fichiers statiques.
— `Gestion SSR pour toutes les requêtes GET` : app.get('*', ssrHandler) ; utilise le gestionnaire SSR d'Astro pour toutes les requêtes GET, permettant le rendu des pages côté serveur.

6.2 Configuration du Serveur HTTP et Socket.IO

— `Création d'un serveur HTTP` : Enveloppe l'application Express avec un serveur HTTP.
— `Initialisation de Socket.IO` : Crée une nouvelle instance de Socket.IO liée au serveur HTTP.

6.3 Gestion des Messages et Événements Socket.IO

— `Stockage des messages` : Un tableau messages est utilisé pour stocker les messages du chat.
— `Gestion des connexions Socket.IO` :

Lorsqu'un utilisateur se connecte, son ID de socket est enregistré, et les messages précédents sont envoyés à ce client.

Écoute les messages de chat entrants, les stocke, et les diffuse à tous les clients connectés.

Gère l'adhésion des utilisateurs aux salles de chat. Si une salle a moins de 4 utilisateurs, un nouvel utilisateur peut rejoindre. Sinon, un événement full_room est émis.

Gère le démarrage d'un appel vidéo en diffusant un événement start_call aux pairs dans la même salle.

Traite les offres WebRTC, les réponses et les candidats ICE en les transmettant aux pairs concernés.

6.4 Gestion de la Déconnexion

— `Déconnexion d'un utilisateur` : Lorsqu'un utilisateur se déconnecte, un message est enregistré, et d'autres actions peuvent être effectuées comme la gestion de l'état de la salle.

6.5 Démarrage du Serveur

— `Port du serveur` : Le serveur écoute sur un port spécifié ou sur le port 8080 par défaut.

— `Démarrage du serveur` : Le serveur commence à écouter les connexions entrantes.

6.6 Résumé

Ce script serveur gère une application de chat en temps réel avec des fonctionnalités de chat vidéo en utilisant WebRTC et Socket.IO. Il gère également le rendu côté serveur pour les pages front-end construites avec Astro, offrant une expérience utilisateur fluide et intégrée.

7 UI/UX de l'application

L'expérience utilisateur (UX) et la conception de l'interface utilisateur (UI) jouent un rôle essentiel dans le succès d'une application web. Cette partie du rapport se penche attentivement sur ces deux aspects cruciaux, examinant de près l'UI/UX de l'application en question. Notre objectif est de fournir une évaluation approfondie de la convivialité, de l'esthétique et de la fonctionnalité de l'interface, ainsi que de l'expérience globale qu'elle offre à ses utilisateurs.

Au cours de cette analyse, nous examinerons la conception visuelle, la facilité de navigation et la réactivité sur différentes plateformes. À travers cette démarche, nous chercherons à identifier les points forts de l'application ainsi que les domaines qui pourraient bénéficier d'améliorations. Les recommandations formulées dans ce rapport visent à optimiser l'interaction des utilisateurs avec l'application, favorisant ainsi une expérience utilisateur exceptionnelle.

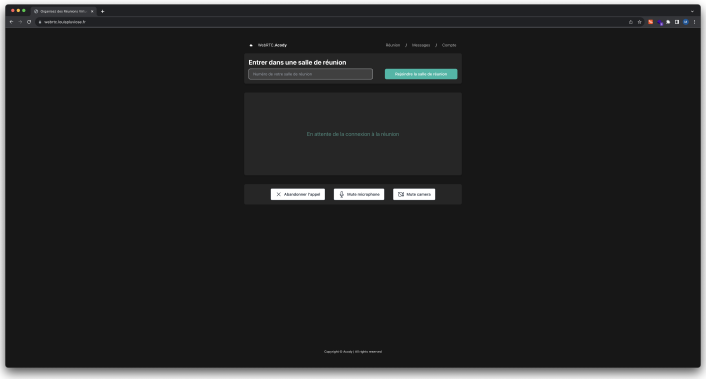


Fig. 0.1: Page Réunion de l'application

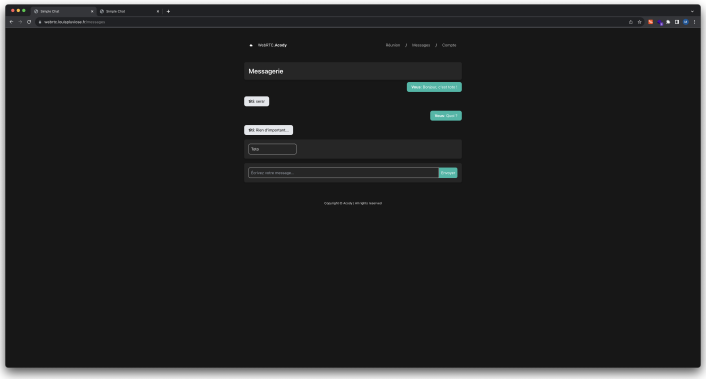


Fig. 0.2: Page Messagerie de l'application

8 Technologies utilisées

8.1 Astro.js - Framework frontend utilisé

Nous avons utilisé le framework Astro.js pour la réalisation de la partie frontend de notre application web. C'est un framework récent sur le marché qui adopte une approche "server-first", privilégiant le rendu côté serveur par rapport au rendu côté client dans le navigateur. Cela offre des performances très élevées, ce qui nous intéresse dans le cas de notre projet. De plus, sa compatibilité avec des frameworks tels que "TailwindCSS" et "React.js" nous a permis d'appliquer un design moderne et convivial pour l'utilisateur.

8.2 TailwindCSS et React.js

Tailwind CSS est un framework CSS utilitaire qui simplifie le développement et la stylisation des interfaces utilisateur. Contrairement aux frameworks CSS traditionnels basés sur des composants prédéfinis, Tailwind CSS fournit des classes utilitaires directement applicables dans le code HTML. Ces classes permettent de définir rapidement et de manière cohérente des styles tels que la couleur, la taille, la marge, le rembourrage, etc. L'approche de Tailwind CSS encourage une flexibilité accrue tout en offrant une base solide pour la conception.

React.js est une bibliothèque JavaScript développée par Facebook pour la construction d'interfaces utilisateur interactives. React utilise une approche basée sur les composants, permettant de créer des morceaux d'interface réutilisables et modulaires. Cette approche facilite la gestion de l'état de l'application et la mise à jour dynamique de l'interface en réponse aux changements.

Dans le cadre de notre projet, nous avons utilisé Tailwind CSS pour simplifier la stylisation en exploitant ses classes utilitaires directement dans le code HTML. Cela a accéléré le processus de conception en nous permettant de définir rapidement et de manière cohérente les styles des éléments.

Parallèlement, nous avons intégré React.js pour structurer l'interface utilisateur de manière modulaire. Les composants React ont été utilisés pour diviser l'application en parties réutilisables, facilitant ainsi la maintenance et la gestion de l'état global de l'application.

9 Contexte de l'Application

L'application vise à faciliter les réunions virtuelles en temps réel. Elle offre une plateforme pour la communication vidéo instantanée, améliorant ainsi la collaboration à distance. Les fonctionnalités clés comprennent la possibilité d'entrer dans une salle de réunion en utilisant un numéro spécifique, la gestion du son et de la caméra, ainsi que la capacité à abandonner les appels en un clic. En plus des fonctionnalités de réunion vidéo en temps réel, l'application intègre également une fonctionnalité de messagerie pour une communication asynchrone entre les utilisateurs. Cette fonctionnalité permet aux utilisateurs d'échanger des messages textuels, complétant ainsi l'expérience de communication collaborative.

9.1 Public Cible

L'application cible un large éventail d'utilisateurs professionnels cherchant à optimiser leurs réunions virtuelles. Elle s'adresse particulièrement aux équipes travaillant à distance, aux entreprises cherchant à améliorer la communication interne et externe, ainsi qu'aux individus ayant besoin d'une solution efficace pour les réunions en ligne.

9.2 Objectifs de l'Application (Réunions)

1. **Faciliter les Réunions Virtuelles :** Offrir une plateforme conviviale pour l'organisation de réunions virtuelles.
2. **Communication Vidéo en Temps Réel :** Permettre des appels vidéo en temps réel pour une interaction plus dynamique.
3. **Simplicité d'Utilisation :** Assurer une expérience utilisateur intuitive pour maximiser l'adoption de l'application.
4. **Optimiser la Collaboration à Distance :** Fournir des fonctionnalités simples et efficaces pour améliorer la collaboration à distance.

9.3 Objectifs de l'Application (Messagerie)

1. **Communication Asynchrone :** Fournir une plateforme de messagerie pour permettre des échanges asynchrones entre les utilisateurs.
2. **Facilité d'Utilisation :** Assurer une interface conviviale pour la saisie et l'envoi de messages.
3. **Identité de l'Utilisateur :** Permettre aux utilisateurs de spécifier leur nom d'utilisateur pour une identification personnalisée.

9.4 Principales Fonctionnalités (Réunions)

1. **Entrée dans une Salle de Réunion :** Les utilisateurs peuvent rejoindre une salle de réunion en saisissant un numéro spécifique.
2. **Gestion du Son et de la Caméra :** Contrôle de la fonction audio et vidéo pour une expérience personnalisée.
3. **Abandonner les Appels en un Clic :** Facilité pour quitter rapidement une réunion.
4. **Affichage Vidéo en Temps Réel :** Possibilité de visualiser les flux vidéo en temps réel des participants.

9.5 Principales Fonctionnalités (Messagerie)

1. **Saisie de Nom d'Utilisateur :** Les utilisateurs peuvent entrer leur nom d'utilisateur pour une identification personnalisée.
2. **Saisie de Message :** Interface pour la saisie et l'envoi de messages texte.
3. **Affichage des Messages :** Les messages échangés sont affichés dans une interface dédiée.
4. **Styles Différenciés :** Les messages de l'utilisateur actuel sont stylisés différemment pour une distinction visuelle.

9.6 Implémentation Technique (Réunions)

1. **Utilisation de Socket.io :** Les connexions peer-to-peer WebRTC sont établies via les événements de socket, notamment les événements `room_created`, `room_joined`, et `start_call`.
2. **RTCPeerConnection :** Pour créer et gérer les connexions peer-to-peer.
3. **Interface Utilisateur :** La partie vidéo est divisée en un conteneur local et plusieurs conteneurs distants.

9.7 Implémentation Technique (Messagerie)

1. **Utilisation de Socket.io :** La messagerie utilise Socket.io pour la gestion des communications en temps réel.
2. **Identifiant Unique de l'Utilisateur :** Chaque utilisateur est associé à un identifiant unique généré au moment de la connexion.
3. **Événements Socket.io :** L'application utilise des événements Socket.io tels que 'connect' et 'chat_message' pour gérer la communication.

10 Analyse UI/UX

10.1 Navigation

La navigation au sein de l'application s'effectue via une barre de navigation positionnée en haut de la page. Cette barre comprend deux boutons : "Réunions" et "Messages". L'utilisateur final est ainsi invité à faire son choix à travers cette barre de navigation, lui offrant une manière claire et accessible d'accéder aux fonctionnalités désirées.

Fig. 0.3: Barre de navigation sur un écran large

10.1.1 Réalisation Technique

Le code source est présent en annexe (voir Annexe X). Voici une explication technique à travers différents points sur la structure de la "NavBar" (barre de navigation) de l'application.

1. **Structure HTML et classes CSS :** - La structure de la barre de navigation est définie dans une balise `header`. - La classe `lg:flex` rend la barre de navigation flexible sur les écrans de taille large. - La balise `Astronav` encapsule l'ensemble de la barre de navigation.
2. **Logo et titre :** - Un logo représenté par un fichier SVG est inclus, utilisant la classe `icon`. - Le titre de la page, "WebRTC.Acody," est placé à côté du logo et est stylisé avec des classes CSS.
3. **Bouton de menu pour les écrans de taille réduite :** - Pour les écrans de taille réduite (inférieure à lg), un bouton de menu est affiché grâce à la classe `icon-menus`.

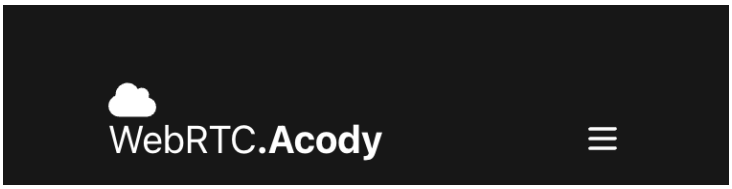


Fig. 0.4: Bouton de menu pour les écrans de taille réduite

4. **MenuItems et liens de navigation :** - La balise `MenuItems` enveloppe la liste de navigation et la classe `hidden` la rend initialement invisible sur les écrans larges. - La liste de navigation est structurée avec des liens vers les sections du site telles que "Réunion," "Messages".

5. **Styles CSS :** - Des styles CSS personnalisés sont définis pour les séparateurs, les liens de la liste, le survol des liens, l'opacité, et les filtres d'image. - Des classes telles que `separator-mob` sont utilisées pour ajuster le style en fonction de la taille de l'écran. - Les filtres d'image, tels que `hue-rotate`, sont appliqués pour des effets visuels lors du survol.
6. **Media Queries :** - Des règles de media queries sont utilisées pour adapter le style en fonction de la largeur de l'écran. - Les séparateurs sont masqués sur les écrans de taille réduite, et des styles spécifiques sont appliqués aux éléments pour une meilleure lisibilité.

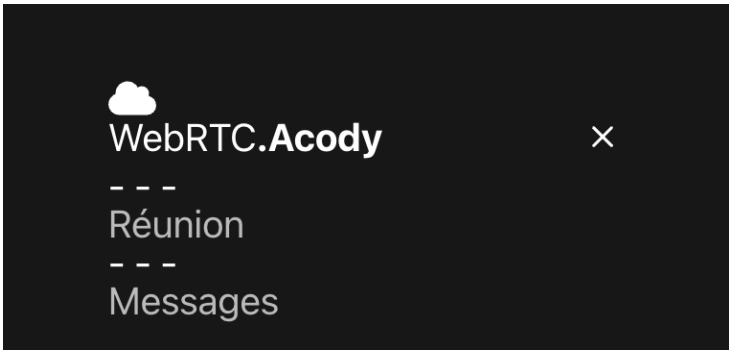


Fig. 0.5: Menu ouvert sur les écrans de taille réduite

10.1.2 Expérience utilisateur

1. **Clarté et Simplicité :** - La barre de navigation est simple et claire, avec seulement deux boutons principaux : "Réunions" et "Messages". Cela évite toute confusion et facilite la navigation.
2. **Logo et Titre :** - L'inclusion d'un logo et du titre "WebRTC.Acody" renforce l'identité visuelle de la page. Cela améliore la reconnaissance du projet et aide les utilisateurs à comprendre le contexte de la page.
3. **Adaptabilité pour les Petits Écrans :** - La barre de navigation est conçue pour s'adapter aux petits écrans. L'utilisation d'un bouton de menu (MenuIcon) pour les écrans de taille réduite montre une considération pour les utilisateurs sur des appareils mobiles.
4. **Effets Visuels au Survole :** - Les effets visuels au survol, tels que le changement d'opacité et la rotation de teinte, ajoutent une dimension interactive à la barre de navigation, améliorant l'expérience visuelle.

10.2 Page - Réunion

10.3 Page - Messagerie

11 Conclusion de l'analyse UI/UX

11.1 La Navigation

11.2 Analyse de l'UI (Interface Utilisateur)

11.3 Analyse de l'UX (Expérience Utilisateur)

11.4 Recommandations et Améliorations

12 Installation de l'application

12.1 Installation en dur

ATTENTION : au préalable, il faut installer NodeJS et NPM sur la machine !

Pour installer l'application en dur, il suffit de cloner le dépôt git suivant : .

Ensuite, il faut naviger dans le dossier `SAE5.ROM.03/webssr` et lancer la commande `npm install` pour installer les dépendances.
Enfin, il faut lancer la commande `npm run build` puis `node ./run-server.mjs` pour lancer l'application.

12.2 Installation en Docker

ATTENTION : au préalable, il faut installer Docker sur la machine !

Pour installer l'application en Docker, il suffit de cloner le dépôt git suivant : .

Ensuite, il faut naviger dans le dossier `SAE5.ROM.03/docker` et lancer la commande `docker build . -t nom-de-votre-application` pour construire l'image Docker.

Enfin, il faut lancer la commande `docker run -p 3000:3000 nom-de-votre-application` pour lancer l'application.

13 Bugs

13.1 Bugs connus

Nous avons pu détecter 2 bugs dans notre application.

13.1.1 Premier bug

Lors de certains appels, il arrive que la connexion se fasse mal et que un des interlocuteurs ne voit pas l'autre. Voici plusieurs façon de résolver ce problème :

- 1 : Il faut recliquer sur le bouton "Rejoindre la réunion"
- 2 : Si le problème persiste, Recharger la page
- 3 : Si le problème persiste toujours, il faut quitter la salle et en rejoindre une nouvelle

13.1.2 Deuxième bug

Au moment de raccrocher l'appel, il se peut la vidéo de l'autre interlocuteur reste affichée. Pour résoudre ce problème, il faut rafraichir la page.

13.2 Bugs inconnus

Si vous rencontrez d'autres bugs, merci de nous les signaler à l'adresse suivante : contact@louispluviose.fr.

14 Annexes

14.1 Code du backend

14.1.1 Client

14.1.1.1 DOM elements

```
1  const roomSelectionContainer =
       document.getElementById('room-selection-container');
2  const roomInput = document.getElementById('room-input');
3  const connectButton = document.getElementById('connect-button');
4  const videoChatContainer =
       document.getElementById('video-chat-container');
5  const localVideoComponent = document.getElementById('local-video');
```

1: DOM Elements

14.1.1.2 Variables globales

```
1  const socket = io();
2  const mediaConstraints = { audio: true, video: { width: 1280, height:
       720 } };
3  let localStream;
4  let roomId;
5  let peerConnections = {}; // Dictionary to hold all peer connections
6
7  const iceServers = {
8      iceServers: [
9          { urls: 'stun:stun.l.google.com:19302' }, // Serveur STUN
               existant
10         // Ajout de la configuration TURN
11         {
12             urls: 'turn:relay1.expressturn.com:3478', // URL du
                   serveur TURN
13             username: 'efJJ0L80U0GANH5V0A', // Nom d'utilisateur
14             credential: 'LO5Tdr8aoKohTHDL' // Mot de passe
15         }
16     ]
17 };
```

2: Variables

14.1.1.3 Button listeners

```
1  connectButton.addEventListener('click', () => {
2  joinRoom(roomInput.value);
3 });
4
5 const hangUpButton = document.getElementById('hangup-button');
6 const toggleMicButton = document.getElementById('toggle-mic-button');
7 const toggleCameraButton =
      document.getElementById('toggle-camera-button');
8
9 hangUpButton.addEventListener('click', hangUpCall);
10 toggleMicButton.addEventListener('click', toggleMicrophone);
11 toggleCameraButton.addEventListener('click', toggleCamera);
12
13 let isRoomCreator = false;
```

3: Button Listeners

14.1.1.4 *Socket event callbacks*

```
  socket.on('room_created', async () => {
  console.log('Socket event callback: room_created');
  await setLocalStream(mediaConstraints);
  isRoomCreator = true;
});

socket.on('room_joined', async () => {
  console.log('Socket event callback: room_joined');
  await setLocalStream(mediaConstraints);
  isRoomCreator = false;
  socket.emit('start_call', roomId);
});

socket.on('full_room', () => {
  console.log('Socket event callback: full_room');
  alert('The room is full, please try another one');
});

socket.on('start_call', async () => {
  console.log('Socket event callback: start_call');
  if (isRoomCreator) {
    createPeerConnections();
  }
});

socket.on('webrtc_offer', async (data) => {
  console.log('Socket event callback: webrtc_offer');

  if (!localStream) {
    console.log("Waiting to set local stream before handling offer");
    await setLocalStream(mediaConstraints);
  }

  if (!peerConnections[data.peerId]) {
    await setupPeerConnection(data.peerId, false); // false = not an
        initiator
  }
  await handleOffer(data);
});

socket.on('webrtc_answer', (data) => {
  console.log('Socket event callback: webrtc_answer');
  handleAnswer(data);
});

socket.on('webrtc_ice_candidate', (data) => {
  console.log('Socket event callback: webrtc_ice_candidate');
  handleIceCandidate(data);
});

socket.on('new_peer', async (peerId) => {
  console.log('Socket event callback: new_peer');
  await createPeerConnection(peerId, false);
});
```

4: Socket Event Callbacks

14.1.1.5 *Fonctions*

```
  async function joinRoom(room) {
  if (room === '') {
```

```
3       alert('Please type a room ID');
4     } else {
5       roomId = room;
6       socket.emit('join', room);
7       showVideoConference();
8
9       try {
10        localStream = await
              navigator.mediaDevices.getUserMedia(mediaConstraints);
11        document.getElementById('local-video').srcObject = localStream;
12      } catch (error) {
13        console.error('Could not get user media', error);
14      }
15    }
16  }
17
18  function showVideoConference() {
19    roomSelectionContainer.style = 'display: none';
20    videoChatContainer.style = 'display: block';
21
22    // Masquer le texte d'attente
23    const waitingText = document.getElementById('waitingText');
24    if (waitingText) {
25      waitingText.style.display = 'none';
26    }
27  }
28
29
30  async function setLocalStream(mediaConstraints) {
31    if (!localStream) {
32      try {
33        const stream = await
              navigator.mediaDevices.getUserMedia(mediaConstraints);
34        console.log('Local stream obtained', stream);
35        localStream = stream;
36        const localVideoComponent =
              document.getElementById('local-video');
37        if (localVideoComponent) {
38          localVideoComponent.srcObject = stream;
39        } else {
40          console.error('The local video element was not found in the
                DOM.');
41        }
42      } catch (error) {
43        console.error('Could not get user media', error);
44      }
45    }
46  }
47
48  async function handleNewPeer(peerId) {
49    const peerConnection = await createPeerConnection(peerId);
50    localStream.getTracks().forEach(track => {
51      peerConnection.addTrack(track, localStream);
52    });
53
54    const offer = await peerConnection.createOffer();
55    await peerConnection.setLocalDescription(offer);
56    socket.emit('webrtc_offer', {
57      type: 'webrtc_offer',
58      sdp: offer,
59      roomId,
60      peerId,
61    });
```

```
}

async function setupPeerConnection(peerId, isInitiator) {
  const peerConnection = new RTCPeerConnection(iceServers);

  if (localStream) {
    localStream.getTracks().forEach(track =>
       peerConnection.addTrack(track, localStream));
  } else {
    console.error("Local stream is not defined in setupPeerConnection");
    return;
  }

  peerConnection.ontrack = (event) => addRemoteStream(event.streams[0],
     peerId);
  peerConnection.onicecandidate = (event) => handleIceEvent(event,
     peerId);

  peerConnections[peerId] = peerConnection;

  if (isInitiator) {
    const offer = await peerConnection.createOffer();
    await peerConnection.setLocalDescription(offer);
    socket.emit('webrtc_offer', { roomId, sdp: offer, peerId });
  }
}

function addRemoteStream(event, peerId) {
  let remoteVideoElement =
     document.getElementById(`remote-video-${peerId}`);
  if (!remoteVideoElement) {
    remoteVideoElement = createRemoteVideoElement(peerId);
  }

  let stream;
  if (event.streams && event.streams.length > 0) {
    // Use the stream from the event if it exists
    stream = event.streams[0];
  } else {
    // Create a new stream and add the track to it
    stream = new MediaStream();
    if (event.track) {
      stream.addTrack(event.track);
    }
  }

  // Additional logging to debug
  console.log(`Adding remote stream for peer ${peerId}`, stream);
  if (stream.getTracks().length === 0) {
    console.error('No tracks in the remote stream');
  }

  remoteVideoElement.srcObject = stream;
  remoteVideoElement.muted = false; // Ensure remote video is not muted
  remoteVideoElement.volume = 1; // Ensure volume is set to maximum
}

function createRemoteVideoElement(peerId) {
  const remoteVideo = document.createElement('video');
  remoteVideo.id = `remote-video-${peerId}`;
  remoteVideo.autoplay = true;
  remoteVideo.playsInline = true;
  remoteVideo.classList.add('remote-video', ...tailwindClasses);
```

```
121
122   document.getElementById('remote-videos-container')
123         .appendChild(remoteVideo);
124   return remoteVideo;
125 }
126
127
128 async function handleIceEvent(event, peerId) {
129   if (event.candidate) {
130     socket.emit('webrtc_ice_candidate', {
131       roomId,
132       candidate: event.candidate,
133       peerId
134     });
135   }
136 }
137
138 async function createPeerConnection(peerId) {
139   const peerConnection = new RTCPeerConnection(iceServers);
140
141   \begin{verbatim}
142   localStream.getTracks().forEach(track =>
        peerConnection.addTrack(track, localStream));
143   \end{verbatim}
144     // localStream.getTracks().forEach(track =>
          peerConnection.addTrack(track, localStream));
145   \end{verbatim}
146   localStream.getTracks().forEach(track =>
        peerConnection.addTrack(track, localStream));
147
148   // Gestion de l'ajout des streams distants
149   peerConnection.ontrack = (event) => addRemoteStream(event, peerId);
150
151   // Gestion de l'ajout des streams distants
152   peerConnection.ontrack = (event) => {
153   let remoteVideoElement =
        document.getElementById(`remote-video-${peerId}`);
154   if (!remoteVideoElement) {
155     remoteVideoElement = document.createElement('video');
156     remoteVideoElement.id = `remote-video-${peerId}`;
157     remoteVideoElement.autoplay = true;
158     remoteVideoElement.playsInline = true;
159     remoteVideoElement.classList.add('remote-video');
160     document.getElementById('remote-videos-container')
161         .appendChild(remoteVideoElement);
162   }
163   remoteVideoElement.srcObject = event.streams[0];
164 };
165
166
167   // Gestion des candidats ICE
168   peerConnection.onicecandidate = (event) => {
169     if (event.candidate) {
170       console.log(`Sending ICE candidate to peer ${peerId}`,
            event.candidate);
171       socket.emit('webrtc_ice_candidate', {
172         type: 'webrtc_ice_candidate',
173         candidate: event.candidate,
174         roomId,
175         peerId,
176       });
177     }
178   };
```

```
179
180   peerConnections[peerId] = peerConnection;
181
182   \usepackage{lastpage}
183   if (!isRoomCreator) {
184     const offer = await peerConnection.createOffer();
185     await peerConnection.setLocalDescription(offer);
186     socket.emit('webrtc_offer', {
187       type: 'webrtc_offer',
188       sdp: offer,
189       roomId,
190       peerId,
191     });
192   }
193 }
194
195
196 async function handleOffer(data) {
197   try {
198     if (!peerConnections[data.peerId]) {
199       await createPeerConnection(data.peerId);
200     }
201
202     const peerConnection = peerConnections[data.peerId];
203     console.log(\texttt{\'{E}tat} de la connexion Peer avant
          setRemoteDescription:
          $\backslash$texttt{${peerConnection.signalingState}}$`);
204
205     await peerConnection.setRemoteDescription(new
          RTCSessionDescription(data.sdp));
206
207     if (peerConnection.cachedIceCandidates) {
208       peerConnection.cachedIceCandidates.forEach(cachedCandidate => {
209         peerConnection.addIceCandidate(new
              RTCIceCandidate(cachedCandidate));
210       });
211       peerConnection.cachedIceCandidates = [];
212     }
213
214     const answer = await peerConnection.createAnswer();
215     await peerConnection.setLocalDescription(answer);
216
217     socket.emit('webrtc_answer', {
218       type: 'webrtc_answer',
219       sdp: answer,
220       roomId,
221       peerId: data.peerId,
222     });
223   } catch (error) {
224     console.error(`Erreur dans handleOffer pour le pair
          ${data.peerId}:`, error);
225   }
226 }
227
228 async function handleAnswer(data) {
229   const peerConnection = peerConnections[data.peerId];
230   console.log(`Peer connection state before setting remote description:
        ${peerConnection.signalingState}`);
231
232   if (peerConnection.signalingState === 'have-local-offer') {
233     try {
234       await peerConnection.setRemoteDescription(new
            RTCSessionDescription(data.sdp));
```

```
235       console.log(`Remote description set for peer ${data.peerId}`);
236     } catch (error) {
237       console.error(`Error in handleAnswer for peer ${data.peerId}:`,
              error);
238     }
239   } else {
240     console.log(`Peer connection not in the correct state to set remote
          description, current state: ${peerConnection.signalingState}`);
241   }
242 }
243
244
245 async function handleIceCandidate(data) {
246   const peerConnection = peerConnections[data.peerId];
247   if (peerConnection) {
248     if (!peerConnection.remoteDescription) {
249       console.log("Queueing ICE candidate as remote description is not
              yet set");
250       if (!peerConnection.cachedIceCandidates) {
251         peerConnection.cachedIceCandidates = [];
252       }
253       peerConnection.cachedIceCandidates.push(data.candidate);
254     } else {
255       console.log("Adding ICE candidate");
256       await peerConnection.addIceCandidate(new
              RTCIceCandidate(data.candidate));
257     }
258   }
259 }
260
261
262
263 function handleNewICECandidateMsg(data) {
264   const peerConnection = peerConnections[data.peerId];
265   peerConnection.addIceCandidate(new RTCIceCandidate(data.candidate));
266 }
267
268 function sendIceCandidate(candidate, peerId) {
269   socket.emit('webrtc_ice_candidate', {
270     roomId,
271     candidate,
272     peerId,
273   });
274 }
275
276 function addVideoStream(videoElement, stream, isLocal = false) {
277   \texttt{console.log('Adding videoElement.srcObject = stream;')}
278   videoElement.srcObject = stream;
279   videoElement.autoplay = true;
280   videoElement.playsInline = true;
281   videoElement.muted = isLocal;
282   if (isLocal) {
283     videoElement.id = 'local-video';
284     videoElement.style.backgroundColor = 'red';
285   } else {
286     videoElement.classList.add('remote-video');
287     videoElement.style.backgroundColor = 'green';
288   }
289   videoChatContainer.appendChild(videoElement);
290 }
291
292 function handleRemoteStreamAdded(stream, peerId) {
293   console.log(`handleRemoteStreamAdded called with peerId: ${peerId}`);
```

```
294   let videoElementId = `remote-video-${peerId}`;
295   let remoteVideoElement = document.getElementById(videoElementId);
296
297   if (!remoteVideoElement) {
298     console.log(`Creating new video element for peer ${peerId}`);
299     remoteVideoElement = document.createElement('video');
300     remoteVideoElement.id = videoElementId;
301     remoteVideoElement.autoplay = true;
302     remoteVideoElement.playsInline = true;
303     remoteVideoElement.classList.add('remote-video');
304     document.getElementById('remote-videos-container')
305           .appendChild(remoteVideoElement);
306   }
307   else {
308     console.log(`Replacing video element for peer ${peerId}`);
309   }
310
311   remoteVideoElement.srcObject = stream;
312 }
313
314
315
316 function hangUpCall() {
317   console.log("Hang Up Call");
318   for (let peerId in peerConnections) {
319     peerConnections[peerId].close();
320     delete peerConnections[peerId];
321   }
322
323   if (localStream) {
324     localStream.getTracks().forEach(track => track.stop());
325     localStream = null;
326   }
327
328   let remoteVideosContainer =
        document.getElementById('remote-videos-container');
329   while (remoteVideosContainer.firstChild) {
330     remoteVideosContainer.removeChild(remoteVideosContainer.firstChild);
331   }
332
333   const waitingText = document.getElementById('waitingText');
334   if (waitingText) {
335     waitingText.style.display = 'block';
336   }
337
338   videoChatContainer.style.display = 'none';
339
340   roomSelectionContainer.style.display = 'none';
341 }
342
343
344 function toggleMicrophone() {
345   const audioTrack = localStream.getAudioTracks()[0];
346   if (audioTrack) {
347     audioTrack.enabled = !audioTrack.enabled;
348     console.log("Microphone toggled. Now enabled:", audioTrack.enabled);
349
350     // Update the track on all peer connections
351     for (let peerId in peerConnections) {
352       const sender = peerConnections[peerId].getSenders().find(s =>
            s.track.kind === audioTrack.kind);
353       if (sender) {
354         sender.replaceTrack(audioTrack);
```

```
      }
    }
  }
}


function toggleCamera() {
  const videoTrack = localStream.getVideoTracks()[0];
  if (videoTrack) {
    videoTrack.enabled = !videoTrack.enabled;
    console.log("Camera toggled. Now enabled:", videoTrack.enabled);

    // Update the track on all peer connections
    for (let peerId in peerConnections) {
      const sender = peerConnections[peerId].getSenders().find(s =>
          s.track.kind === videoTrack.kind);
      if (sender) {
        sender.replaceTrack(videoTrack);
      }
    }
  }
}
```

5: Fonctions Principales

14.1.2 Messagerie

```
document.addEventListener('DOMContentLoaded', () => {
  const socket = io();
  const userId = Date.now().toString();

  socket.on('connect', () => {
    console.log('Connecte au serveur Socket.io');
  });

  const sendButton = document.getElementById('sendButton');
  const messagesDiv = document.getElementById('messages');
  const usernameInput = document.getElementById('username');
  const messageInput = document.getElementById('message');

  if (sendButton && messagesDiv && usernameInput && messageInput) {
    sendButton.addEventListener('click', () => {
      const username = usernameInput.value || 'Anonyme';
      const message = messageInput.value;

      if (message.trim() !== '') {
        socket.emit('chat_message', { userId, username, message });
        messageInput.value = '';
      }
    });

    socket.on('chat_message', (data) => {
      const messageContainer = document.createElement('div');
      messageContainer.className = "flex my-2";

      const messageElement = document.createElement('div');
      messageElement.className = "px-4 py-2 rounded-lg shadow";

      const senderName = document.createElement('strong');
      senderName.textContent = data.userId === userId ? 'Vous' :
          data.username;

      messageElement.appendChild(senderName);
      messageElement.appendChild(document.createTextNode(`:
          ${data.message}`));

      if (data.userId === userId) {
        messageContainer.classList.add('justify-end');
        messageElement.classList.add('bg-teal-500', 'text-white');
      } else {
        messageContainer.classList.add('justify-start');
        messageElement.classList.add('bg-gray-200', 'text-black');
      }

      messageContainer.appendChild(messageElement);
      messagesDiv.appendChild(messageContainer);
      messagesDiv.scrollTop = messagesDiv.scrollHeight;
    });
  }
});
```

6: Code de la messagerie

14.1.3 Serveur

```
import express from 'express';
import { createServer } from 'http';
import { Server as SocketIOServer } from 'socket.io';
import { handler as ssrHandler } from './dist/server/entry.mjs';

const app = express();
const base = '/';

// Serve the static files from the Astro build
app.use(base, express.static('dist/client/'));
app.use('/scripts', express.static('dist/scripts'));

// Handle SSR for all get requests
app.get('*', ssrHandler);

// Create an HTTP server and configure Socket.io
const server = createServer(app);
const io = new SocketIOServer(server);

let messages = [];

// Socket.io event handling
io.on('connection', (socket) => {
  console.log('A user connected: ' + socket.id);

  messages.forEach((message) => {
    socket.emit('chat_message', message);
  });

  socket.on('chat_message', (data) => {
    messages.push(data);
    console.log('message: ' + data);
    io.emit('chat_message', data);
  });

  socket.on('disconnect', () => {
    console.log('User disconnected: ' + socket.id);
  });

  socket.on('join', (roomId) => {
    const selectedRoom = io.sockets.adapter.rooms.get(roomId);
    const numberOfClients = selectedRoom ? selectedRoom.size : 0;

    console.log(`Client ${socket.id} requesting to join room
       ${roomId}`);
    if (numberOfClients < 4) {
      console.log(`Joining room ${roomId} and emitting room_joined
         socket event`);
      socket.join(roomId);
      socket.to(roomId).emit('new_peer', socket.id); // Notify others
         in the room
    } else {
      console.log(`Can't join room ${roomId}, emitting full_room
         socket event`);
      socket.emit('full_room', roomId);
    }
  });

  socket.on('start_call', (roomId) => {
    console.log(`Broadcasting start_call event to peers in room
       ${roomId}`);
```

```
57        socket.broadcast.to(roomId).emit('start_call');
58      });
59
60      socket.on('webrtc_offer', (data) => {
61        console.log(`Received offer from ${socket.id} in room
              ${data.roomId}`);
62        socket.to(data.peerId).emit('webrtc_offer', {
63          peerId: socket.id,
64          sdp: data.sdp
65        });
66      });
67
68      socket.on('webrtc_answer', (data) => {
69        console.log(`Received answer from ${socket.id} in room
              ${data.roomId}`);
70        socket.to(data.peerId).emit('webrtc_answer', {
71          peerId: socket.id,
72          sdp: data.sdp
73        });
74      });
75
76      socket.on('webrtc_ice_candidate', (data) => {
77        console.log(`Received ICE candidate from ${socket.id} for room
              ${data.roomId}`);
78        socket.to(data.peerId).emit('webrtc_ice_candidate', {
79          peerId: socket.id,
80          label: data.label,
81          candidate: data.candidate
82        });
83      });
84
85      // Handle user disconnect
86      socket.on('disconnect', () => {
87        console.log('User disconnected: ' + socket.id);
88      });
89    });
90
91    // Start the server on the specified port or default to port 8080
92    const PORT = process.env.PORT || 8080;
93    server.listen(PORT, () => {
94      console.log(`Server is running on port ${PORT}`);
95    });
```

7: Code du serveur

15 Conclusion

Ce projet a représenté un défi stimulant et enrichissant pour notre groupe. L'opportunité d'explorer et de maîtriser la technologie WebRTC a marqué une étape importante dans notre parcours de développement. La création d'une application de communication en temps réel nous a permis non seulement d'approfondir nos compétences en Docker et NodeJS, mais aussi de comprendre les subtilités de la communication en ligne moderne.

Bien que nous ayons nourri l'ambition d'intégrer des fonctionnalités supplémentaires telles que le partage d'écran, un tableau blanc interactif, la gestion de contacts, la création de comptes utilisateurs, et même la liaison avec un serveur Asterisk, le temps imparti n'a pas suffi pour réaliser toutes ces idées. Cependant, cette limitation n'a en rien diminué notre fierté face aux accomplissements réalisés. Nous avons réussi à créer une application fonctionnelle et intuitive, dotée de fonctionnalités de base efficaces et faciles à utiliser.

Ce sentiment d'accomplissement est renforcé par notre engagement à poursuivre le développement de ce projet. Notre vision à long terme est de le rendre plus complet, plus performant et de le transformer en un produit open source. En continuant à travailler sur ce projet, nous espérons non seulement enrichir notre propre expérience, mais également contribuer à la communauté en offrant une solution de communication avancée et accessible à tous.

En somme, ce projet est plus qu'une simple réalisation technique ; il représente notre passion pour l'innovation et notre désir de repousser les limites de ce qui est possible dans le domaine de la communication numérique. Nous sommes impatients de voir où ce chemin nous mènera et sommes déterminés à faire de ce projet une référence dans le monde de la communication WebRTC.