INSA Lyon – Département Télécommunications

Année universitaire : 2025 – 2026

Appareil de diffusion vidéo non intrusif

WebRTC

Encadrant :
Stéphane Frenot - Damien Reimert

Réalisé par :

- Orhon Gabriel
- Chkoundali Yasmine
- Mohammi Islam
- Abidi Jean
- Adjami Axel

Sommaire

Abstract....3
1. Introduction....4
2. Présentation du scénario d'usage....4
2.1 Diagramme de séquences....5
2.2 Diagramme de composants....6
3. Fonctionnement de WebRTC....7
3.1 Fonctionnement général et liste des composants....7
3.2 Rôle des protocoles....9
3.3 API utilisées....11
4. Code détaillé....12
4.1 Liste des fonctions principales....12
4.2 Code de chaque fonction....12
5. Conclusion....16

Abstract

Ce document explique le fonctionnement de WebRTC et son utilisation dans le cadre de notre projet de diffusion d'écran en temps réel. Le projet utilise WebRTC pour diffuser l'écran d'un PC utilisateur vers un Raspberry Pi connecté à un vidéoprojecteur via une connexion pair-à-pair (P2P). Nous détaillons les composants essentiels de WebRTC. Nous expliquons également le rôle de STUN et TURN dans la découverte des adresses réseau et la gestion de la connectivité à travers des NAT, et précisons que, bien que TURN et STUN ne soient pas nécessaires dans notre cas, nous utilisons STUN pour découvrir les chemins réseau potentiels, mais étant donné que les deux appareils sont sur le même réseau local eduroam, nous optons finalement pour l'utilisation de l'adresse locale, qui est le chemin le plus efficace et direct. Enfin, nous détaillons les fonctions principales de notre implémentation.

1. Introduction

L'objectif de ce projet est de réaliser un système de diffusion vidéo non intrusif permettant d'afficher l'écran d'un utilisateur sur un vidéoprojecteur à l'aide d'un Raspberry Pi. Pour répondre à ce besoin, nous avons retenu WebRTC, car cette technologie est nativement supportée par les navigateurs. Comme nous disposons d'un cas d'usage concret de référence, ce document ne cherche pas à présenter WebRTC de manière exhaustive. Nous nous concentrons sur les éléments directement utiles à notre application.

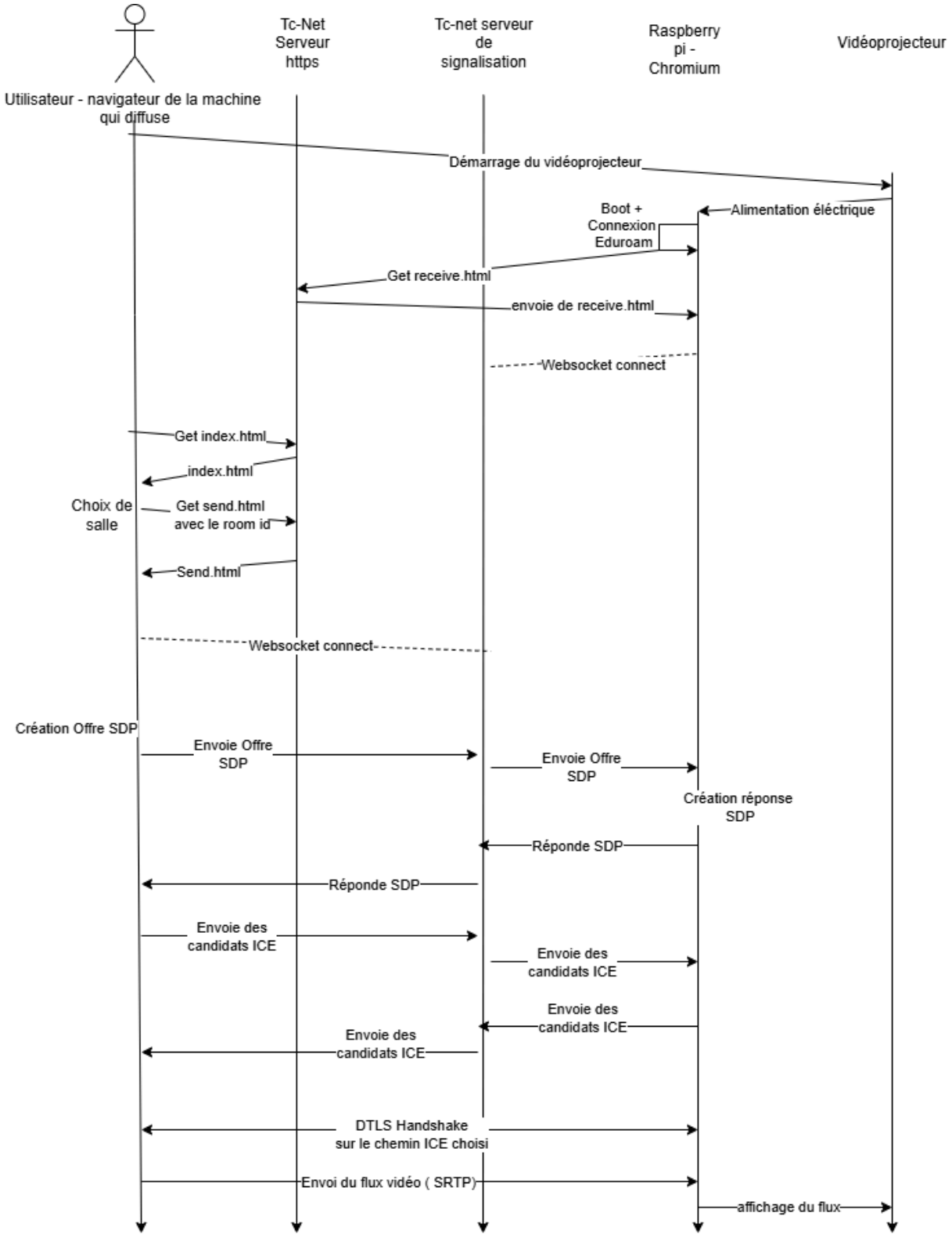
2. Présentation du scénario d'usage

Cette partie décrit le cas d'usage principal de notre projet, qui consiste à permettre la diffusion de l'écran d'un utilisateur vers un vidéoprojecteur dans une salle, sans installation logicielle, uniquement via un navigateur web.

Le fonctionnement de notre scénario repose sur :

- Un navigateur web côté utilisateur
- Un navigateur Chromium côté Raspberry Pi
- Un serveur de signalisation et un serveur https hébergés sur TC-Net.

2.1 Diagramme de séquences



2.2 Diagramme de composants

Le diagramme de composants permet de décrire l'architecture globale du système de communication vidéo basé sur WebRTC. Il met en évidence les différents composants matériels et logiciels impliqués, ainsi que leurs interactions. Cette représentation facilite la compréhension du rôle de chaque élément dans le fonctionnement global du système.

Dans le cadre de ce scénario, l'architecture repose sur une communication pair-à-pair entre navigateurs, assistée par un serveur de signalisation utilisé uniquement lors de l'établissement de la connexion.

2.2.1 Machines

Le système WebRTC repose sur les machines suivantes :
- Machine Utilisateur : Il s'agit du poste client de l'utilisateur initiateur de l'appel vidéo. Cette machine dispose d'un navigateur web compatible WebRTC .
-Raspberry pi : Cette machine utilise chromium comme navigateur web intégrant les API WebRTC et des périphériques multimédias.
-Serveur de signalisation : Machine serveur hébergée sur tc-net, chargée de faciliter l'échange des messages de signalisation entre les deux utilisateurs. Ce serveur ne transporte pas les flux audio et vidéo, mais uniquement les informations nécessaires à l'établissement de la connexion (SDP, ICE candidates).

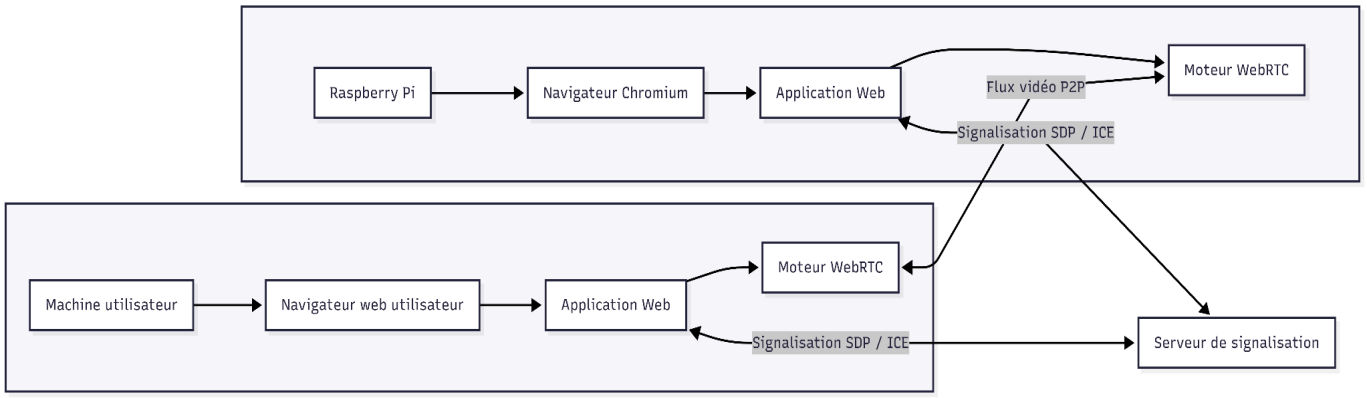
2.2.2 Logiciels

Chaque machine héberge un ou plusieurs composants logiciels jouant un rôle précis dans le système :
- Navigateurs Web : Le navigateur côté utilisateur et chromium sur le raspberry pi . Ils intègrent le moteur WebRTC qui fournit les API nécessaires à la capture des flux multimédias, à l'établissement des connexions pair-à-pair et au transport sécurisé des données.
-Application Web de communication vidéo : Il s'agit de l'interface utilisateur développée. Elle s'appuie sur les API WebRTC pour contrôler la capture des médias, la création des connexions et la gestion des échanges avec le serveur de signalisation.
-Moteur WebRTC : Composant interne au navigateur, responsable du traitement bas niveau :

encodage et décodage des flux audio et vidéo, gestion des protocoles (ICE, DTLS, SRTP), et transmission des données en temps réel.
-Serveur de signalisation : Implementé à l'aide de Node.js et WebSocket, il permet l'échange des messages de signalisation entre les navigateurs. Il joue un rôle clé dans la phase d'établissement de la communication, mais n'intervient plus une fois la connexion pair-à-pair établie.

Diagramme de composants – WebRTC

Le diagramme ci-dessous illustre l'architecture logicielle et matérielle du système, ainsi que les relations entre les différents composants.



Les utilisateurs interagissent avec l'application web via leur navigateur. L'application web s'appuie sur le moteur WebRTC intégré au navigateur pour accéder aux périphériques multimédias et établir une connexion directe avec le correspondant.

Le serveur de signalisation intervient uniquement pour permettre l'échange des informations nécessaires à la négociation de la connexion. Une fois cette phase terminée, le flux vidéo est échangé directement entre les deux navigateurs, sans transiter par le serveur, garantissant ainsi une communication à faible latence.

3. Fonctionnement de WebRTC

3.1 Fonctionnement général et liste des composants

3.1.1 Fonctionnement général

Le fonctionnement d'une communication WebRTC peut être décrit selon les étapes suivantes :

1. **Capture et préparation des médias**
   L'application récupère un flux multimédia sous forme de pistes (MediaStreamTrack).

2. **Création d'une connexion P2P**
   Une connexion WebRTC est initialisée en créant un objet RTCPeerConnection. Cet objet gère l'ensemble du processus d'établissement et de maintien de la connexion entre les deux pairs.

3. **Ajout des pistes et déclenchement de la négociation**
   Lorsque des pistes multimédias sont ajoutées à RTCPeerConnection, cela déclenche la phase de négociation. Cette négociation se fait via l'échange de descriptions de session (offer/answer) contenant les paramètres nécessaires à la communication.

4. **Signalisation (échange de messages via le serveur)**
   Les informations de négociation doivent être transmises au pair distant via un serveur de signalisation. Les principaux messages échangés sont :

   - une offre
   - une réponse
   - des informations de connectivité (ICE candidates)

5. **Connexion réseau et échange des flux**
   Une fois la connectivité établie, les flux circulent directement entre les pairs.

3.1.2 Composants WebRTC

Le fonctionnement précédent mobilise plusieurs composants :

- **Peers (Les pairs)**

  Deux navigateurs/applications qui échangent un flux en temps réel.

- **Serveur de signalisation**

  Infrastructure permettant d'échanger les messages offer/answer et ICE candidates entre les pairs (via WebSocket dans notre cas).

- **Serveurs STUN et TURN (optionnels selon le réseau)**

  Session Traversal Utilities for NAT (STUN) et **Traversal Using Relays around NAT (TURN)** sont des protocoles utilisés dans WebRTC pour faciliter la découverte des pairs et la communication à travers des **Network Address Translation(NAT)** et des pare-feu. **STUN** aide à découvrir l'adresse IP publique d'un appareil, tandis que **TURN** fonctionne comme un relais de données lorsque la connexion directe échoue, servant de solution de secours pour traverser des configurations NAT plus complexes.

  Dans le cadre de notre projet, nous n'utilisons pas de serveur TURN car les deux appareils sont sur eduroam. Dans ce contexte, la communication directe entre les appareils peut se faire via leurs adresses IP locales, rendant ainsi l'utilisation de **TURN** inutile. De plus, puisque les deux pairs sont dans le même réseau, nous n'avons même pas besoin de **STUN** pour découvrir une adresse publique, car les chemins locaux suffisent à établir une connexion P2P efficace.

3.2 Rôle des protocoles

(A) Connectivité

- **Interactive Connectivity Establishment protocol (ICE):** ICE est le mécanisme qui permet d'établir la connectivité réseau entre deux pairs, pour cela chaque pair commence par collecter les adresses et ports possibles par lesquels il peut être joint, appelés *ICE candidates*. Ces candidats appartiennent à trois catégories :
  - **Host candidates** : qui correspondent aux adresses locales de la machine (pour notre scénario on est concerné que par ce type de candidats vu qu'on est sur Eduroam).
  - **Server reflexive candidates (srflx)** : qui représentent l'adresse publique découverte grâce à un serveur **STUN** .
  - **Relay candidates** : qui sont des adresses fournies par un serveur **TURN** servant de relais lorsque la connexion directe est impossible.

ICE privilégie d'abord une connexion directe locale (*host*), puis une connexion directe via l'adresse publique obtenue avec STUN (*srflx*), et n'utilise TURN (*relay*) qu'en dernier recours.

(B) Négociation

SDP permet aux pairs de se mettre d'accord sur les paramètres de communication:

- type de flux (vidéo)
- codecs et paramètres d'encodage
- informations réseau nécessaires au transport

Dans notre projet, SDP est encapsulé dans les messages WebSocket de type **`offer`** et **`answer`**

(C) Signalisation : WebSocket sécurisé (WSS)

Le protocole WebSocket est utilisé comme canal de signalisation pour transporter les messages nécessaires à la négociation WebRTC :

- **`ready` :** indiquer qu'un pair est prêt
- **`offer` / `answer` :** négociation SDP
- **`candidate` :** échange ICE
- **`bye` :** fermeture de session

(D) Sécurité

Datagram Transport Layer Security (DTLS) intervient lors de l'établissement de la session. DTLS permet aux deux pairs de réaliser un *handshake* cryptographique, de s'authentifier mutuellement et de négocier des clés de chiffrement. Dans WebRTC, ces clés ne servent pas seulement à sécuriser des messages de contrôle : elles sont utilisées pour dériver les clés qui chiffreront ensuite les flux multimédias. DTLS assure ainsi la base de confiance de la communication .

(E) Transport médias

- **Real-time Transport Protocol (RTP) :** assure le transport des flux multimédias dans une communication WebRTC. Il découpe les données audio/vidéo encodées en

paquets RTP et les encapsule pour les envoyer sur le réseau (généralement au-dessus d'UDP afin de minimiser la latence). Chaque paquet contient des informations essentielles, notamment un numéro de séquence et un timestamp, qui permettent au récepteur de réordonner les paquets, de détecter les pertes, de mesurer les irrégularités d'arrivée (*jitter*) et de reconstruire un flux continu malgré le caractère non fiable d'UDP (où des paquets peuvent être perdus ou arriver en désordre).

- **Secure Real-time Transport Protocol (SRTP)** : est la version sécurisée de RTP et assure le transport des flux multimédias chiffrés dans WebRTC. Les paquets multimédias sont toujours envoyés sous forme de paquets RTP, mais SRTP ajoute une couche de sécurité : il applique un chiffrement pour garantir la confidentialité du contenu, ainsi qu'une authentification / intégrité pour empêcher qu'un attaquant puisse modifier les paquets durant le transport.

- **RTCP (RTP Control Protocol)** : complète RTP en fournissant un canal de contrôle et de supervision, RTCP permet aux pairs d'échanger des statistiques (taux de pertes, jitter, RTT, qualité de réception). Ces informations servent au moteur WebRTC pour adapter dynamiquement la transmission (par exemple ajustement du débit).

3.3 API utilisées

- **`RTCPeerConnection :`** `l'API principale de WebRTC. Elle permet de créer et maintenir une connexion p2p entre les deux pairs. Elle gère la négociation entre les deux pairs (offer/answer), la recherche d'un chemin réseau fonctionnel grâce à ICE, et la transmission sécurisée des flux.`

- **`RTCIceCandidate :`**`est une interface qui représente un candidat ICE. Un objet RTCIceCandidate contient notamment la chaîne candidate (description du candidat), ainsi que des informations permettant de l'associer à une section média de la négociation. Ces candidats sont produits au fur et à mesure de la collecte ICE et sont fournis à l'application via l'événement icecandidate; ils peuvent ensuite être transmis au pair distant par le mécanisme de signalisation afin d'être ajoutés à la connexion avec addIceCandidate(), permettant ainsi à WebRTC de tester différents chemins réseau et de`

sélectionner un chemin fonctionnel entre les deux pairs.

4. Code détaillé

4.1 Liste des fonctions principales

- createPeerConnection(roomID)
- handleOffer(offer)
- handleAnswer(answer)
- handleCandidate(candidate)
- makeCall(roomID)
- hangup()

4.2 Code de chaque fonction

```
function createPeerConnection() {
    peerConnection = new RTCPeerConnection({iceServers: [{ urls: 'stun:stun.tc-net.bzy.li:3478' },
     {urls: 'turn:turn.tc-net.bzy.li:3478', username : 'tc-net', credential :'tc-net'}]});
    peerConnection.oniceconnectionstatechange = () => {
      const state = peerConnection.iceConnectionState;
      console.log('ICE state:', state);
      if (state === "failed" || state === "disconnected" || state === "closed") {
        location.reload();
      }
    }
    peerConnection.onicecandidate = e => {
        const message = {
            type : 'candidate',
            candidate : null,
            roomId : roomID,
        };
        if (e.candidate) {
            message.candidate = e.candidate.candidate;
            message.sdpMid = e.candidate.sdpMid;
            message.sdpMLineIndex = e.candidate.sdpMLineIndex;
        }
        console.log(message);
        signaling.send(JSON.stringify(message));
    };
    peerConnection.ontrack = e => { // Quand on reçoit le flux avec WebRTC
      title.style.display= 'none';
      remoteVideo.style.display = 'inline';
      remoteVideo.srcObject = e.streams[0]; // On met le flux vidéo dans notre tag html vidéo
    }
};
```

Cette fonction initialise la connexion WebRTC côté récepteur. Elle crée un objet RTCPeerConnection avec une configuration ICE (serveurs STUN et TURN) afin de permettre l'établissement de la connexionh. Elle surveille ensuite l'état de la connexion ICE et recharge la page si la connexion échoue ou se ferme (failed, disconnected, closed). Lors de la collecte des candidats ICE, chaque candidat découvert est envoyé via le serveur de signalisation WebSocket pour être transmis au pair distant. Enfin, lorsque le flux vidéo est reçu, l'événement ontrack est déclenché : la fonction masque le titre de la page et affecte le flux reçu à la balise vidéo, ce qui permet d'afficher la diffusion en temps réel.

```
async function handleOffer(offer) {
  if (peerConnection) {
    console.error('existing peerconnection');
    return;
  }
  await createPeerConnection();
  await peerConnection.setRemoteDescription(offer);

  const answer = await peerConnection.createAnswer();
  signaling.send(JSON.stringify({type: 'answer', sdp: answer.sdp, roomID : roomID}));
  await peerConnection.setLocalDescription(answer);
}
```

Cette fonction gère la réception d'une offre WebRTC envoyée par le pair émetteur. Elle vérifie d'abord qu'aucune connexion WebRTC n'est déjà active (sinon elle ignore l'offre). Ensuite,elle applique l'offre reçue avec setRemoteDescription, ce qui initialise la négociation SDP côté récepteur. Une réponse est alors générée avec createAnswer, puis envoyée au pair distant via le serveur de signalisation WebSocket (message de type answer contenant le SDP). Enfin, setLocalDescription est appelé pour enregistrer localement la réponse, ce qui finalise la négociation et permet de poursuivre l'établissement de la connexion.

```
async function handleAnswer(answer) {
  if (!peerConnection) {
    console.error('no peerconnection');
    return;
  }
  await peerConnection.setRemoteDescription(answer);
}
```

Cette fonction traite la réponse envoyée par l'autre pair. Elle vérifie d'abord que la connexion RTCPeerConnection existe bien . Ensuite, elle appelle setRemoteDescription(answer) afin d'enregistrer la description distante (SDP) reçue.

```
async function handleCandidate(candidate) {
  if (!peerConnection) {
    console.error('no peerconnection');
    return;
  }
  if (candidate.candidate) {
    console.log('adding new candidate');
    const iceCandidate = new RTCIceCandidate({
        candidate: candidate.candidate,
        sdpMid: candidate.sdpMid,
        sdpMLineIndex: candidate.sdpMLineIndex
    });
    await peerConnection.addIceCandidate(iceCandidate);
  }
}
```

Cette fonction gère la réception d'un candidat ICE envoyé par l'autre pair via le serveur de signalisation. Elle commence par vérifier que la connexion RTCPeerConnection existe ; sinon il n'est pas possible d'ajouter de candidat. Si le message contient bien un candidat (champ candidate non nul), elle crée un objet RTCIceCandidate à partir des informations reçues (candidate, sdpMid, sdpMLineIndex), puis l'ajoute à la connexion avec addIceCandidate.

```
//--------------  WEBSOCKET CONNECTION ---------------
const signaling = new WebSocket('wss://ws.tc-net.bzy.li'); // Serveur de signalement
signaling.onmessage = e => {
    if (!localStream) {
        console.log('Not ready yet'); // On a pas encore pu obtenir le flux vidéo de l'utilisateur qui veut diffuser
        return;
    }
    const message = JSON.parse(e.data);  // Le type de message qui a été reçu
    switch (message.type) {
    case 'answer':                       // Le vidéoprojecteur/raspberry pi a répondu à l'offre de diffusion
      handleAnswer(message);
      break;
    case 'candidate':                    // Candidat ICE reçu
      handleCandidate(message);
      break;
    case 'ready':                        // Le raspbery pi/device est prêt à recevoir un flux
      if (peerConnection) {
        console.log('already in call, ignoring'); // on était déjà dans un appel donc on peut ignorer
        return;
      }
      console.log('Remote device ready');
      break;
    default:
      console.log('unhandled', e);      // Message inconnu
      break;
  }
};
//--------------  END OF WEBSOCKET CONNECTION ---------------
```

Ce bloc gère la signalisation WebRTC via WebSocket. Il reçoit les messages answer et candidate, puis appelle respectivement handleAnswer et handleCandidate. Le message ready indique que le pair distant est prêt.

```
let globalRoomID;
sallesContainer.onclick = async (e) => {
  if (e.target.nodeName !== 'BUTTON') {return;}
  if (e.target.className !== 'rooms') {return;}
  let roomID = e.target.id;
  globalRoomID = roomID;
  localStream = await navigator.mediaDevices.getDisplayMedia({audio : false, video: {frameRate : {ideal : 30}, width : { max : 1920, ideal : 1280},
  localVideo.srcObject = localStream; // Affichage de la fenêtre capturé sur la page web
  redrawHomepageAfterStartCall();

  signaling.send(JSON.stringify({type : 'ready', roomID : roomID})); // On prévient le signaling server qu'on est prêt
  makeCall(roomID);
}

async function makeCall(roomID) {
    createPeerConnection(roomID);

    const offer = await peerConnection.createOffer();


    signaling.send(JSON.stringify({type: 'offer', sdp: offer.sdp, roomID : roomID})); // On envoi l'offre de diffusion vers le websocket
    await peerConnection.setLocalDescription(offer);                                  // On met a jour la LocalDescription (ce qu'on va recevoir, ..

    const sender = peerConnection.getSenders().find(s => s.track && s.track.kind ==='video');

    if (sender) {
      const params = sender.getParameters();
      if (!params.encoding) { params.encoding = [{}];}

      params.encoding[0].maxBitrate = 3000000;
      params.encoding[0].minBitrate = 3000000;

      await sender.setParameters(params);
    }
  }
```

La fonction makeCall démarre l'appel WebRTC côté émetteur. Elle crée d'abord la connexion RTCPeerConnection, puis génère une offre SDP avec createOffer. Cette offre est envoyée au pair distant via le serveur de signalisation (message offer), puis enregistrée localement avec setLocalDescription pour lancer la négociation. Enfin, elle récupère le sender vidéo (getSenders) et ajuste les paramètres d'encodage afin de fixer le bitrate du flux envoyé.

```
// Fonction pour récupérer le flux vidéo de l'utilisteur
startButton.onclick = async () => {
    localStream = await navigator.mediaDevices.getDisplayMedia({audio : false, video: true}); // https://developer.mozilla.org/en-US/docs/Web/API/Me
    localVideo.srcObject = localStream; // Affichage de la fenêtre capturé sur la page web

    startButton.disabled = true;
    hangupButton.disabled = false;

    signaling.send(JSON.stringify({type : 'ready', roomID : roomID})); // On prévient le signaling server qu'on est prêt
    makeCall();
};
```

Ce bloc est déclenché quand l'utilisateur clique sur Start. Il capture l'écran avec getDisplayMedia, affiche le flux local dans la balise vidéo, puis envoie un message ready au serveur de signalisation. Enfin, il lance l'appel WebRTC en appelant makeCall().

5. Conclusion

En s’appuyant sur notre scénario de diffusion d’écran, WebRTC s’est révélé adapté pour établir une communication temps réel sécurisée entre un émetteur et un récepteur. L’architecture mise en place sépare clairement la signalisation (réalisée via WebSocket) du transport multimédia (assuré en pair-à-pair une fois la connexion établie). Le projet met en évidence les étapes clés nécessaires à un fonctionnement opérationnel : négociation SDP, échange et validation de candidats ICE, et sécurisation des échanges via DTLS/SRTP. Les choix réalisés sont cohérents avec notre environnement réseau (eduroam), où la connectivité locale est suffisante .