Institut National des Postes et Télécommunications

Module : Communication Multimédia P2P en temps réel

Rapport de TP3 : La Plateforme WebRTC

Implémentation d'un système de Visioconférence et Chat P2P

Réalisé par :
Aymane ASKRI

Encadré par :
Asmae ELHAMZAOUI

Année Universitaire : 2025-2026

Table des matières

Introduction **2**

1 Architecture et Concepts Théoriques **3**
1.1 Le besoin de Signalisation 3
1.2 Traversée des NAT (STUN/TURN) 4

2 Implémentation de la Solution **5**
2.1 Serveur de Signalisation (Socket.io) 5
2.2 Capture Média et Initialisation 5
2.3 Établissement du P2P et DataChannel 6

3 Extensions et Analyse **7**
3.1 Extension 2 : Alternatives à Socket.io 7
3.2 Extension 3 : Implémentation Générique 7
3.3 Extension 4 : Mode Vidéoconférence (Multipartites) 7

Conclusion **9**

Introduction

L'évolution du web vers des applications riches et interactives a rendu nécessaire la communication en temps réel sans plugins tiers (comme Flash par le passé). **WebRTC** (Web Real-Time Communication) est la réponse standardisée à ce besoin, permettant l'échange de flux audio, vidéo et de données arbitraires directement entre navigateurs (Peer-to-Peer).

L'objectif de ce TP est de concevoir une application complète intégrant :

— La capture des flux multimédias (Caméra/Micro).

— Un canal de signalisation pour la découverte des pairs (via NodeJS).

— L'établissement d'une connexion P2P pour le streaming vidéo.

— L'utilisation de `RTCDataChannel` pour un chat textuel sécurisé et rapide.

Ce rapport détaille l'architecture technique, l'implémentation des mécanismes de signalisation SDP/ICE, et explore les extensions possibles vers des systèmes de vidéoconférence multipartites.

Chapitre 1

Architecture et Concepts Théoriques

1.1 Le besoin de Signalisation

Bien que WebRTC soit une technologie P2P, deux navigateurs ne peuvent pas se connecter directement "par magie". Ils ne connaissent ni l'adresse IP, ni les ports, ni les capacités multimédias (codecs) de l'autre. Nous avons donc mis en place un **Serveur de Signalisation** utilisant **Node.js** et **Socket.io**. Son rôle est uniquement de relayer les messages administratifs (SDP Offer, Answer, ICE Candidates) avant que la connexion directe ne soit établie.

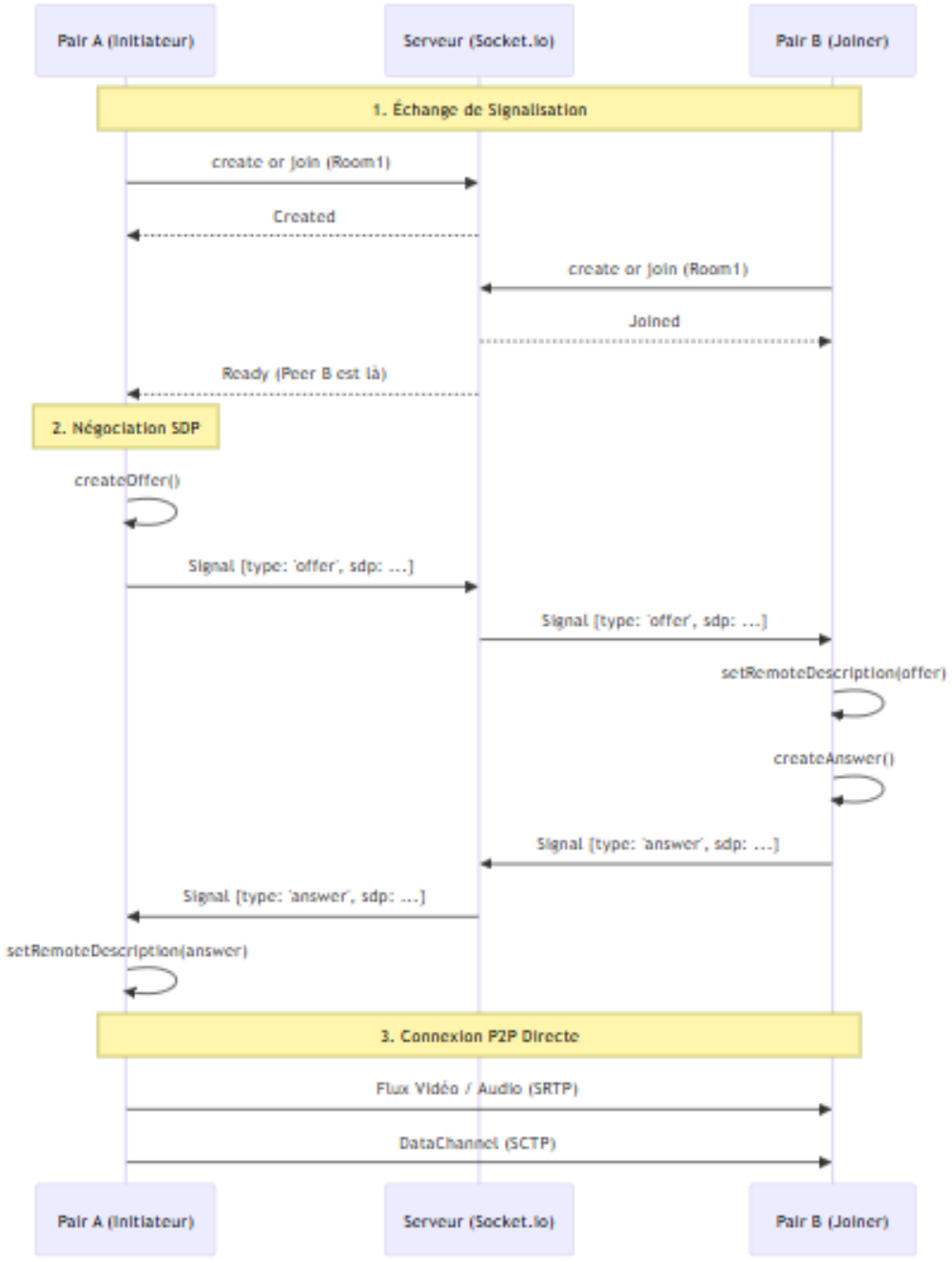

FIGURE 1.1 – Diagramme de séquence : Processus de Signalisation et Handshake

1.2 Traversée des NAT (STUN/TURN)

La majorité des utilisateurs se trouvent derrière des routeurs NAT (Network Address Translation). Pour qu'un pair puisse être joint, il doit connaître son adresse IP publique. Nous avons configuré l'objet `RTCPeerConnection` avec un serveur **STUN** (celui de Google : `stun:stun.l.google.com:19302`). Le serveur STUN renvoie au pair son adresse IP publique et son port, permettant la création de candidats ICE (*Interactive Connectivity Establishment*).

Chapitre 2

Implémentation de la Solution

2.1 Serveur de Signalisation (Socket.io)

Le serveur gère des "salles" (Rooms). Lorsqu'un client se connecte, il crée une salle ou rejoint une salle existante.

```
socket.on('create or join', (room) => {
    const myRoom = io.sockets.adapter.rooms.get(room) || { size: 0 };
    const numClients = myRoom.size;

    if (numClients === 0) {
        socket.join(room);
        socket.emit('created', room);
    } else if (numClients === 1) {
        io.sockets.in(room).emit('join', room);
        socket.join(room);
        // Signal au createur qu'il peut lancer l'offre
        io.sockets.in(room).emit('ready');
    } else {
        socket.emit('full', room); // Limite a 2 pour le P2P simple
    }
});
```

Listing 2.1 – Gestion des salles côté Serveur

2.2 Capture Média et Initialisation

L'API `navigator.mediaDevices.getUserMedia` est utilisée pour obtenir les flux. Nous utilisons les promesses (Async/Await) pour une syntaxe moderne et propre.

```
async function startCapture() {
    try {
        const stream = await navigator.mediaDevices.getUserMedia({
            audio: true,
            video: true
        });
        localVideo.srcObject = stream;
        localStream = stream;
        // Connexion au serveur de signalisation
        socket.emit('create or join', room);
    } catch (e) {
        console.error('Erreur acces camera', e);
```

```
    }
}
```

Listing 2.2 – Capture vidéo locale

2.3 Établissement du P2P et DataChannel

C'est le cur de l'application (Extension 1). L'initiateur crée une offre SDP et un canal de données.

```
function createPeerConnection() {
    rtcPeerConnection = new RTCPeerConnection(rtcConfig);

    // Gestion des candidats reseau (ICE)
    rtcPeerConnection.onicecandidate = (event) => {
        if (event.candidate) {
            socket.emit('message', { type: 'candidate', candidate: event.candidate });
        }
    };

    // Reception du flux distant
    rtcPeerConnection.ontrack = (event) => {
        remoteVideo.srcObject = event.streams[0];
    };

    // Ajout du flux local
    localStream.getTracks().forEach(track => rtcPeerConnection.addTrack(track, localStream));
}

// Creation du canal de chat (DataChannel)
function createDataChannel() {
    dataChannel = rtcPeerConnection.createDataChannel("chat");
    dataChannel.onmessage = (event) => {
        appendMessage("Pair: " + event.data);
    };
}
```

Listing 2.3 – Création de l'offre et du DataChannel

Chapitre 3

Extensions et Analyse

3.1 Extension 2 : Alternatives à Socket.io

Dans ce TP, nous avons utilisé **Socket.io**. Bien que très pratique (gestion automatique des déconnexions, notion de "rooms"), ce n'est pas la seule solution.

- **WebSocket Natif (ws) :** Plus léger, standardisé par le W3C, mais nécessite de réimplémenter la logique de reconnexion et de broadcast manuellement.
- **XHR / SSE :** On pourrait utiliser des requêtes HTTP (Polling) ou Server-Sent Events, mais la latence serait trop élevée pour une mise en relation rapide nécessaire au WebRTC.

3.2 Extension 3 : Implémentation Générique

Pour rendre l'application adaptable à n'importe quel environnement réseau (notamment les réseaux d'entreprise bloquants), il est nécessaire d'externaliser la configuration ICE. Une implémentation générique chargerait un fichier `config.json` au démarrage contenant la liste des serveurs STUN et TURN. Cela permettrait à l'utilisateur de spécifier ses propres identifiants TURN sans modifier le code source JavaScript.

3.3 Extension 4 : Mode Vidéoconférence (Multipartites)

Notre implémentation actuelle est une topologie **Mesh** (Maillage complet) : chaque pair se connecte à tous les autres.

- **Problème :** Si N utilisateurs sont connectés, chaque utilisateur doit gérer $N - 1$ flux montants et descendants. Pour 5 utilisateurs, cela sature rapidement le CPU et la bande passante du client.
- **Solution (SFU/MCU) :** Pour passer à l'échelle, il faut utiliser une architecture centralisée.
  - **SFU (Selective Forwarding Unit) :** Le client envoie un seul flux au serveur, qui le duplique vers les autres participants (Architecture utilisée par Jitsi, Google Meet).
  - **MCU (Multipoint Control Unit) :** Le serveur mixe tous les flux en une seule vidéo composite (Plus lourd pour le serveur, mais très léger pour le client).

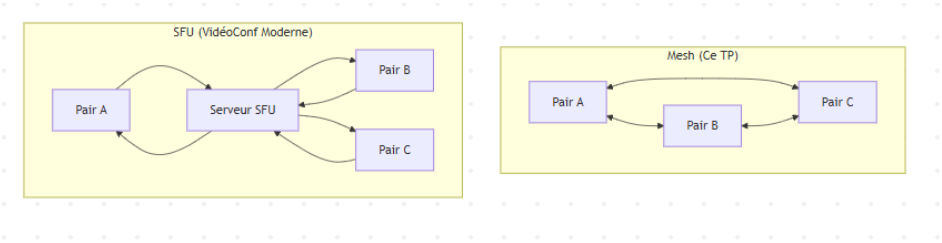

FIGURE 3.1 – Comparaison Architecture Mesh (TP) vs SFU (Production)

Conclusion

Ce TP a permis de démystifier la complexité apparente du WebRTC. Nous avons réussi à implémenter une communication vidéo et textuelle totalement P2P, sécurisée (DTLS/SRTP par défaut) et performante.

Les points clés retenus sont :

1. La distinction claire entre la **signalisation** (qui passe par le serveur) et le **média** (qui passe en direct).
2. L'importance cruciale du protocole **ICE** et des serveurs STUN pour traverser les routeurs domestiques.
3. La puissance de l'API **RTCDataChannel** qui permet de créer des applications temps réel (jeux, partage de fichiers) au-delà de la simple visioconférence.