ÉCOLE SUPÉRIEURE POLYTECHNIQUE
*Dakar — Sénégal*

# SmartTechConnect

*Plateforme Collaborative Pair-à-Pair pour Étudiants*

**Node.js • Socket.IO • PeerJS / WebRTC • MySQL**

| | |
|---|---|
| **Étudiant** | Al Hassane BA |
| **Module** | Applications Client/Serveur et Web (ACW) |
| **Enseignant** | Dr. KEBA GUEYE |
| **Établissement** | DGI — École Supérieure Polytechnique, Dakar |
| **Filière** | DUT2TR |
| **Année** | 2025 – 2026 |

RAPPORT TECHNIQUE — PROJET DE FIN DE MODULE

# Table des matières

Table des matières ... 2
1. Introduction ... 3
2. Architecture Technique ... 3
  2.1 Vue d'ensemble en couches ... 3
  2.2 Schéma d'architecture ... 3
  2.3 Structure des fichiers du projet ... 4
3. Choix Techniques et Justifications ... 5
  3.1 Node.js + Express.js ... 5
  3.2 Socket.IO pour la communication temps réel ... 5
  3.3 PeerJS + WebRTC pour les appels vidéo ... 6
  3.4 bcrypt pour la sécurité des mots de passe ... 6
  3.5 MySQL / MariaDB ... 6
4. Phases de Réalisation ... 6
  Phase 1 — Initialisation de l'environnement ... 6
  Phase 2 — Serveur HTTP et authentification ... 6
  Phase 3 — Chat et tableau blanc en temps réel ... 8
  Phase 4 — Appels vidéo/audio P2P ... 9
  Phase 5 — Tests et déploiement ... 12
5. Difficultés Rencontrées ... 13
  5.1 Partage de session entre Express et Socket.IO ... 13
  5.2 Distinction des connexions chat / vidéo ... 14
  5.3 Synchronisation asynchrone du Peer ID ... 14
  5.4 Libération des ressources MediaStream ... 14
6. Base de Données ... 15
  6.1 Schéma relationnel ... 15
  6.2 Opérations CRUD ... 16
  6.3 Pool de connexions ... 17
7. Description des Fonctionnalités ... 17
  7.1 Authentification et sécurité des sessions ... 17
  7.2 Chat temps réel et historique ... 18
  7.3 Tableau blanc collaboratif ... 19
  7.4 Appels vidéo/audio P2P ... 19
8. Tests et Validation ... 20
  8.1 Résultats des tests fonctionnels ... 20
  8.2 Conditions de test ... 21
9. Conclusion ... 24
  Perspectives d'amélioration ... 25

# 1. Introduction

Le présent rapport décrit la conception et le développement de **SmarttechConnect**, une plateforme web éducative collaborative réalisée dans le cadre du module **Applications Client/Serveur et Web (ACW)** à l'École Supérieure Polytechnique de Dakar.

Ce projet de fin de module couvre l'ensemble des compétences acquises au cours des cinq chapitres du programme : modèles réseau, sockets, développement web dynamique, communication temps réel avec **Socket.IO**, et communication pair-à-pair avec **PeerJS/WebRTC**.

La plateforme offre les fonctionnalités suivantes :

- **Authentification sécurisée** — inscription et connexion avec hachage bcrypt, gestion de sessions
- **Chat instantané par salons** — messagerie en temps réel avec historique persistant en base de données
- **Tableau blanc collaboratif** — dessin partagé synchronisé entre plusieurs utilisateurs
- **Appels vidéo/audio P2P** — communication pair-à-pair via PeerJS/WebRTC sans serveur relais

> **Objectif du projet**
> Concevoir et développer une application fullstack Node.js intégrant de manière cohérente les protocoles HTTP, WebSocket et WebRTC, en respectant les bonnes pratiques de sécurité et d'architecture.

# 2. Architecture Technique

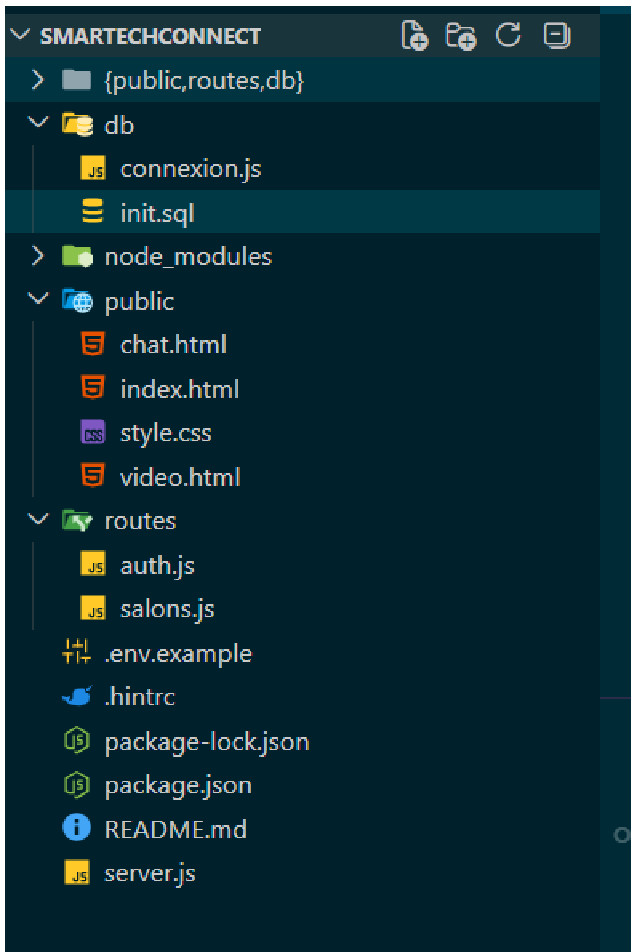
## 2.1 Vue d'ensemble en couches

L'application suit une architecture multicouche à responsabilités séparées :

| Couche | Technologie | Rôle |
|---|---|---|
| Client (Front-end) | HTML5 / CSS3 / JavaScript | Interface utilisateur, interactions, rendu des pages |
| Serveur HTTP | Node.js + Express.js | Routage, API REST, fichiers statiques, sessions |
| Temps réel | Socket.IO | Chat, tableau blanc, signalement P2P |
| Pair-à-pair (P2P) | PeerJS / WebRTC | Appels vidéo et audio directs entre navigateurs |
| Base de données | MySQL / MariaDB | Persistance : utilisateurs, salons, messages |

## 2.2 Schéma d'architecture

## 2.3 Structure des fichiers du projet




# 3. Choix Techniques et Justifications

## 3.1 Node.js + Express.js

Node.js repose sur un modèle **événementiel non-bloquant** (event loop), particulièrement adapté aux applications avec de nombreuses connexions simultanées. Contrairement à un modèle multi-threads, chaque connexion ne crée pas un fil d'exécution dédié, ce qui réduit considérablement la consommation mémoire.

**Express.js** simplifie le routage HTTP, la gestion des middlewares et la définition des API REST. Sa légèreté permet l'intégration native de **express-session** pour les sessions utilisateur et de fichiers statiques pour le front-end.

## 3.2 Socket.IO pour la communication temps réel

Socket.IO a été retenu face à une implémentation WebSocket native pour plusieurs raisons techniques :

- **Reconnexion automatique** — gestion transparente des coupures réseau
- **Système de rooms natif** — idéal pour isoler les salons de discussion
- **Fallback long-polling** — compatibilité garantie même sans WebSocket
- **Partage de session** — le middleware Express-session est réutilisé pour authentifier les connexions WebSocket

### 3.3 PeerJS + WebRTC pour les appels vidéo

PeerJS encapsule l'API WebRTC native et simplifie l'échange de signaux **SDP/ICE** nécessaires à l'établissement d'une connexion pair-à-pair. Le serveur PeerJS est intégré directement à Express via **ExpressPeerServer**, éliminant toute dépendance au serveur public 0.peerjs.com.

> **Avantage architectural**
> Une fois la connexion P2P établie, le flux vidéo circule directement entre les navigateurs sans transiter par le serveur. Cela réduit la charge serveur à zéro pour les données média et garantit une latence minimale.

### 3.4 bcrypt pour la sécurité des mots de passe

Les mots de passe ne sont **jamais stockés en clair**. bcrypt applique un **sel aléatoire** unique par utilisateur et un **facteur de coût** paramétrable, rendant les attaques par force brute et par tables arc-en-ciel computationnellement impraticables.

### 3.5 MySQL / MariaDB

Une base de données **relationnelle** garantit l'intégrité référentielle entre utilisateurs, salons et messages. Le pool de connexions **mysql2** assure la résilience et les performances de la couche d'accès aux données sous charge.

## 4. Phases de Réalisation

### Phase 1 — Initialisation de l'environnement

Création du projet Node.js, installation des dépendances, configuration de la base de données MySQL.

| Dépendance | Version | Rôle |
|---|---|---|
| express | ^4.18.2 | Serveur HTTP, routage, middlewares |
| socket.io | ^4.6.1 | WebSocket temps réel (chat, dessin, signalement) |
| mysql2 | ^3.6.5 | Pool de connexions MySQL |
| bcrypt | ^5.1.1 | Hachage sécurisé des mots de passe |
| express-session | ^1.17.3 | Gestion des sessions utilisateur |
| peer | ^1.0.2 | Serveur PeerJS intégré (WebRTC) |
| nodemon | ^3.0.2 | Rechargement automatique en développement |

### Phase 2 — Serveur HTTP et authentification

Mise en place du serveur Express avec les middlewares essentiels. Création des routes **/auth/register** et **/auth/login**. Implémentation du middleware **requireAuth** pour protéger les routes sensibles.

```javascript
/**
 * server.js — Point d'entrée principal de SmarttechConnect
 * Combine Express (HTTP), Socket.IO (temps réel) et PeerJS (P2P vidéo)
 * Module ACW | Dr. K. GUEYE | ESP Dakar 2024-2025
 */

const express         = require('express');
const http            = require('http');
const { Server }      = require('socket.io');
const session         = require('express-session');
const path            = require('path');
const db              = require('./db/connexion');

// ── Import des routes ──────────────────────────────────
const authRoutes   = require('./routes/auth');
const salonsRoutes = require('./routes/salons');

// ── Initialisation Express & HTTP ──────────────────────
const app    = express();
const server = http.createServer(app);  // Socket.IO a besoin du serveur HTTP brut
const io     = new Server(server, {
  cors: { origin: '*' }   // En production : restreindre l'origine
});

const PORT = process.env.PORT || 3000;
// Lorsque l'on teste depuis une VM ou un autre poste, on veut que
```

```javascript
// ── Middlewares Express ─────────────────────────────────
app.use(express.json());                            // Parser JSON pour les API
app.use(express.urlencoded({ extended: true }));  // Parser formulaires HTML
app.use(sessionMiddleware);                         // Sessions utilisateur
app.use(express.static(path.join(__dirname, 'public'))); // Fichiers statiques

// Partager la session Express avec Socket.IO
io.use((socket, next) => {
  sessionMiddleware(socket.request, {}, next);
});

// ── Routes HTTP ─────────────────────────────────────────
app.use('/auth',       authRoutes);
app.use('/api/salons', salonsRoutes);

// Middleware de protection des routes authentifiées (pages HTML)
function requireAuth(req, res, next) {
  if (!req.session.utilisateur) {
    return res.redirect('/');   // Rediriger vers la page de login
  }
  next();
}

// Pages HTML principales
app.get('/', (req, res) => {
  // Si déjà connecté, rediriger vers le chat
  if (req.session.utilisateur) return res.redirect('/chat');
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.get('/chat',  requireAuth, (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'chat.html'));
});
```

## Phase 3 — Chat et tableau blanc en temps réel

Intégration Socket.IO côté serveur et client. Gestion des rooms par salon, persistance des messages en base de données.

| Événement Socket.IO | Direction | Description |
|---|---|---|
| rejoindre-salon | Client → Serveur | Rejoindre une room Socket.IO (salon) |
| nouveau-message | Client → Serveur | Envoyer un message (persisté en BDD + broadcast) |
| message-reçu | Serveur → Room | Diffuser un message à tous les membres du salon |
| trace-blanc | Client → Room | Synchroniser un tracé de dessin en temps réel |
| effacer-blanc | Client → Room | Effacer le tableau blanc partagé |
| liste-utilisateurs | Serveur → Tous | Mettre à jour la liste des utilisateurs connectés |

```
// ── Événement : rejoindre un salon ──────────────────────
socket.on('rejoindre-salon', async ({ salonId, salonNom }) => {
  // Quitter l'ancien salon si nécessaire
  const userData = utilisateursConnectes.get(socket.id);
  if (userData.salonId) {
    socket.leave(`salon_${userData.salonId}`);
    socket.to(`salon_${userData.salonId}`).emit('utilisateur-parti', {
      nom: user.nom, salonId: userData.salonId
    });
    // mettre à jour la liste du salon quitté
    broadcastMembresSalon(userData.salonId);
  }
```

```
// ── Événement : nouveau message de chat ──────────────────
socket.on('nouveau-message', async ({ salonId, contenu }) => {
  if (!contenu || !contenu.trim() || !salonId) return;

  const message = {
    contenu:       contenu.trim(),
    auteur_nom:    user.nom,
    auteur_avatar: user.avatar,
    envoye_le:     new Date().toISOString()
  };

  try {
    // Sauvegarder en base de données
    await db.query(
      'INSERT INTO messages (contenu, utilisateur_id, salon_id) VALUES (?, ?, ?)',
      [message.contenu, user.id, salonId]
    );

    // Diffuser à TOUS les membres du salon (émetteur inclus)
    io.to(`salon_${salonId}`).emit('message-recu', { ...message, salonId });

  } catch (err) {
    console.error('Erreur sauvegarde message :', err);
    socket.emit('erreur', { message: 'Impossible d\'envoyer le message.' });
  }
});
```

```
// liste des membres (en ligne+hors ligne) pour le salon actuel
socket.on('liste-salon-membres', membres => {
  salonMembres = membres;
  afficherMembresSalon();
});

socket.on('message-recu', (msg) => {
  if (msg.salonId === salonActuel?.id) {
    afficherMessage(msg);
    scrollBas();
  }
});
```

```
// ── Événement : tracé du tableau blanc ──────────────────
// Synchronise les traits dessinés entre tous les membres d'un salon
socket.on('trace-blanc', ({ salonId, trace }) => {
  // Rediffuser aux AUTRES membres du salon (pas à l'émetteur)
  socket.to(`salon_${salonId}`).emit('trace-blanc', trace);
});

// ── Événement : effacer le tableau blanc ────────────────
socket.on('effacer-blanc', ({ salonId }) => {
  socket.to(`salon_${salonId}`).emit('effacer-blanc');
});
```

```
// Liste globale de tous les utilisateurs connectés (mise à jour à chaque connexion/déconnexion)
// Utilisée seulement pour maintenir `salonMembres` à jour si quelqu'un rejoint/quitte le salon.
socket.on('liste-utilisateurs', (users) => {
  // si nous avons une liste de salon en cours, mettre à jour statu en ligne
  if (salonActuel) {
    salonMembres = salonMembres.map(u => {
      const match = users.find(x => x.id === u.id && x.salonId === salonActuel.id);
      if (match) {
        return { ...u, online: true, socketId: match.socketId };
      }
      // si l'utilisateur était en ligne et n'est plus dans salon, le marquer hors ligne
      return { ...u, online: false, socketId: null };
    });
    afficherMembresSalon();
  }
});
```

## Phase 4 — Appels vidéo/audio P2P

Intégration du serveur PeerJS et implémentation du flux de signalement via Socket.IO.

| Étape | Description |
|---|---|
| 1. Génération du Peer ID | Chaque client ouvre /video → PeerJS génère un ID unique |
| 2. Enregistrement | Le Peer ID est transmis au serveur via Socket.IO (enregistrer-peer-id) |
| 3. Initiation de l'appel | Alice émet appel-sortant avec son Peer ID vers Bob |
| 4. Notification entrante | Bob reçoit appel-entrant et peut accepter ou refuser |
| 5. Négociation WebRTC | Si accepté : Bob appelle peer.call(peerIdAlice) → échange SDP/ICE |

| Étape | Description |
|---|---|
| 6. Flux vidéo P2P | Le flux vidéo s'affiche dans <video> — direct entre navigateurs |
| 7. Coupure propre | stream.getTracks().forEach(t => t.stop()) libère la caméra |

```
// ── 0. Analyse de l'URL (mode automatique depuis chat)
const urlParams = new URLSearchParams(window.location.search);
const peerParam = urlParams.get('peer'); // id du pair (pour le receveur ouvert par chat)

// ── 1. Socket connecté → récupérer mon ID ──────────────────
socket.on('connect', () => {
  monSocketId = socket.id;
  document.getElementById('mon-id').textContent = socket.id;
  document.getElementById('btn-etre-appelant').disabled = false;
  document.getElementById('btn-etre-receveur').disabled = false;
  setStatut('🟢 Connecté', 'online');
  log(`✅ Connecté au serveur — ID : ${socket.id}`, 'succes');

  // Si la page a été ouverte avec ?peer=<id> on passe automatiquement en récepteur
  if (peerParam) {
    log(`📩 Ouvert avec peer=${peerParam}, mode RECEVEUR auto`, 'info');
    document.getElementById('btn-etre-receveur').click();
  }
});

socket.on('connect_error', (err) => {
  log(`❌ Erreur Socket.IO : ${err.message}`, 'erreur');
});
```

```
  // Stocker le peerId associé à cet utilisateur pour le signalement
  socket.on('enregistrer-peer-id', ({ peerId }) => {
    // Trouver la socket chat de cet utilisateur et lui transmettre le peerId
    for (const [sid, data] of utilisateursConnectes.entries()) {
      if (data.id === user.id) {
        utilisateursConnectes.set(sid, { ...data, peerId });
        io.emit('liste-utilisateurs', Array.from(utilisateursConnectes.values()));
        break;
      }
    }
  });
```

```
// ── Événements PeerJS / Signalement vidéo ──────────────
// L'appelant envoie une demande d'appel à un pair
socket.on('appel-sortant', ({ destinataireSocketId, appelantNom, peerIdAppelant }) => {
  io.to(destinataireSocketId).emit('appel-entrant', {
    appelantSocketId: socket.id,
    appelantNom,
    peerIdAppelant
  });
  console.log(`📞 ${user.nom} appelle ${destinataireSocketId}`);
});

// Le destinataire accepte l'appel
socket.on('appel-accepte', ({ appelantSocketId, peerIdDestinataire }) => {
  io.to(appelantSocketId).emit('appel-accepte', { peerIdDestinataire });
});

// Refus ou fin d'appel
socket.on('appel-refuse', ({ appelantSocketId }) => {
  io.to(appelantSocketId).emit('appel-refuse');
});

socket.on('fin-appel', ({ interlocuteurSocketId }) => {
  if (interlocuteurSocketId) {
    io.to(interlocuteurSocketId).emit('fin-appel');
  }
});
```

```
// — 7. Recevoir les signaux WebRTC ——————————————————
socket.on('webrtc-signal', async ({ from, type, data }) => {
  log(`📩 Signal reçu [${type}] de ${from.substring(0,12)}`, 'info');

  if (type === 'offer') {
    // Je reçois une offre → créer la réponse
    log('📨 Offre reçue ! Création de la réponse…', 'info');
    setStatut('📞 Appel entrant…', 'calling');

    if (!streamLocal) {
      log('⚠️ Flux local non prêt, attente…', 'warn');
      await demanderMedia();
    }

    creerPC(from);
    await pc.setRemoteDescription(new RTCSessionDescription(data));
    const answer = await pc.createAnswer();
    await pc.setLocalDescription(answer);

    socket.emit('webrtc-signal', { dest: from, type: 'answer', data: answer });
    log('📤 Réponse SDP envoyée', 'succes');

  } else if (type === 'answer') {
    // Je reçois la réponse à mon offre
    log('✅ Réponse SDP reçue !', 'succes');
    await pc.setRemoteDescription(new RTCSessionDescription(data));

  } else if (type === 'ice') {
    // Candidat ICE reçu
    try {
      await pc.addIceCandidate(new RTCIceCandidate(data));
    } catch(e) {
```

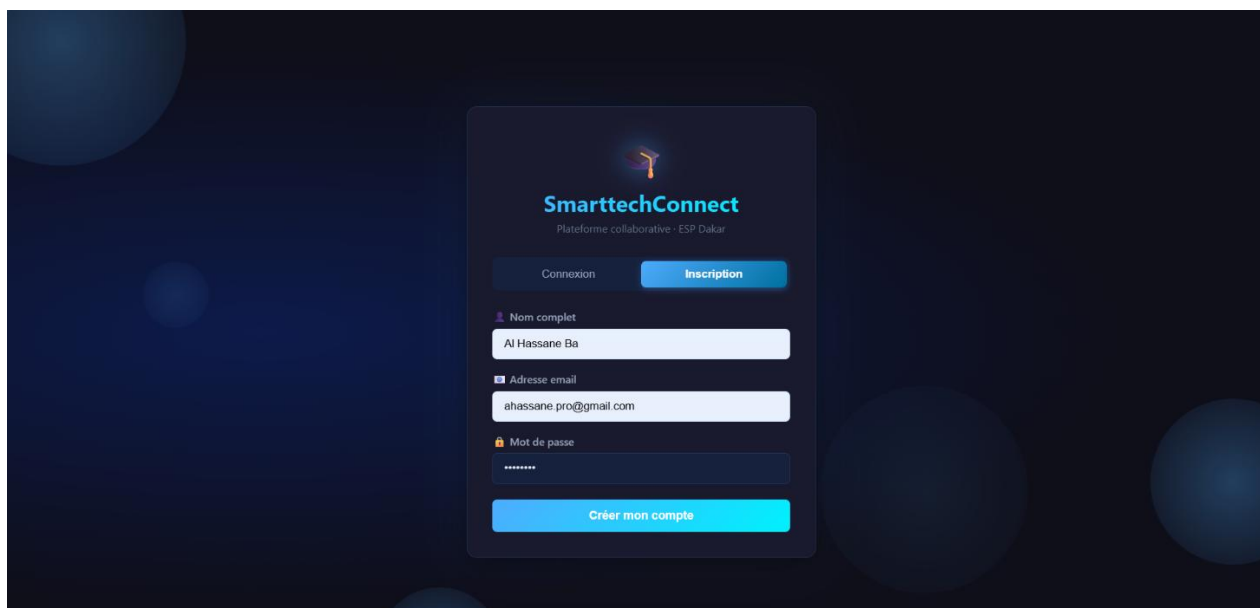
```
// — 4. Créer RTCPeerConnection ——————————————————
function creerPC(destId) {
  if (pc) { pc.close(); pc = null; }
  destSocketId = destId;

  pc = new RTCPeerConnection(iceConfig);

  // Ajouter les pistes locales
  streamLocal.getTracks().forEach(track => pc.addTrack(track, streamLocal));

  // Recevoir le flux distant
  pc.ontrack = (e) => {
    log('👥 Flux distant reçu !', 'succes');
    const vd = document.getElementById('video-distant');
    vd.srcObject = e.streams[0];
    document.getElementById('placeholder').style.display = 'none';
    setStatut('🟢 Appel connecté', 'active');
    document.getElementById('btn-raccrocher').disabled = false;
  };
```

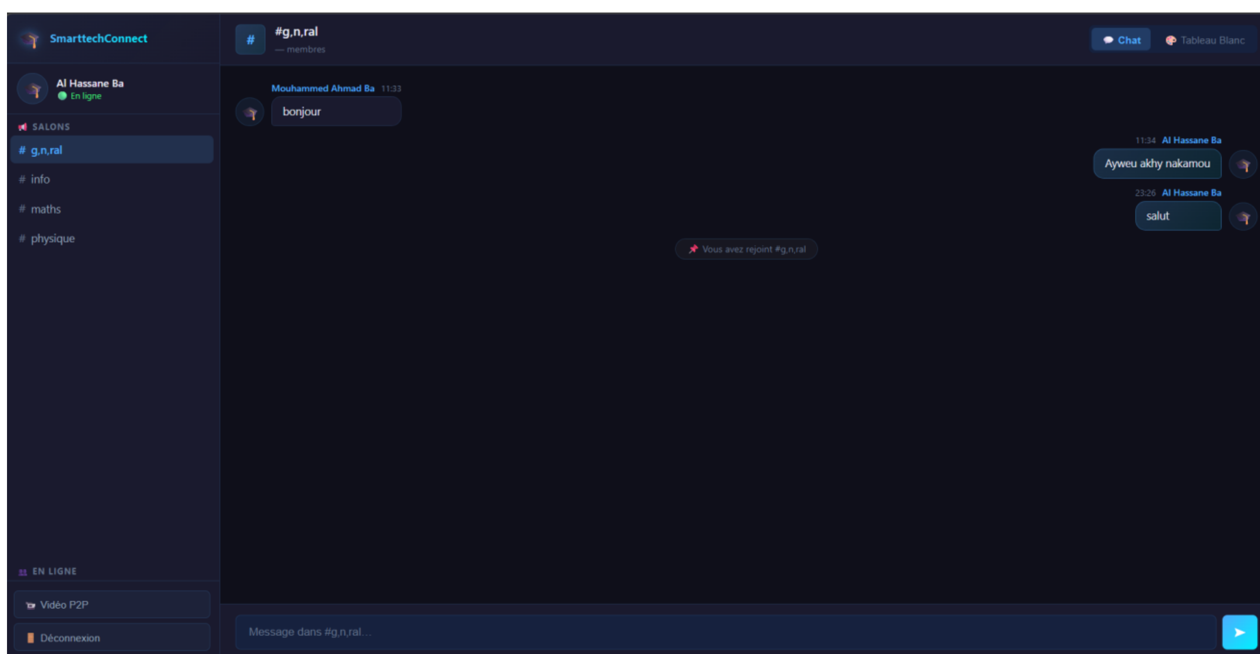
```
// — 7. Recevoir les signaux WebRTC ——————————————————————
socket.on('webrtc-signal', async ({ from, type, data }) => {
  log(`📩 Signal reçu [${type}] de ${from.substring(0,12)}`, 'info');

  if (type === 'offer') {
    // Je reçois une offre → créer la réponse
    log('📨 Offre reçue ! Création de la réponse…', 'info');
    setStatut('📞 Appel entrant…', 'calling');

    if (!streamLocal) {
      log('⚠️ Flux local non prêt, attente…', 'warn');
      await demanderMedia();
    }

    creerPC(from);
    await pc.setRemoteDescription(new RTCSessionDescription(data));
    const answer = await pc.createAnswer();
    await pc.setLocalDescription(answer);
```

```
// — 8. Terminer l'appel ——————————————————————————————
function terminerAppel() {
  if (pc) { pc.close(); pc = null; }
  const vd = document.getElementById('video-distant');
  vd.srcObject = null;
  document.getElementById('placeholder').style.display = 'flex';
  document.getElementById('label-distant').textContent = 'Interlocuteur';
  document.getElementById('btn-raccrocher').disabled   = true;
  setStatut('🟢 Connecté', 'online');
  log('📵 Appel terminé.', 'info');
  // Notifier l'interlocuteur
  if (destSocketId) {
    socket.emit('webrtc-signal', { dest: destSocketId, type: 'hangup', data: null });
  }
}

socket.on('webrtc-signal', ({ type }) => {
  if (type === 'hangup') { log('📵 Interlocuteur a raccroché.', 'warn'); terminerAppel(); }
});

document.getElementById('btn-raccrocher').addEventListener('click', terminerAppel);
```

## Phase 5 — Tests et déploiement

Tests réalisés avec deux navigateurs simultanés (Chrome + Edge) sur réseau local. Validation de l'ensemble des fonctionnalités.




# 5. Difficultés Rencontrées

## 5.1 Partage de session entre Express et Socket.IO

Socket.IO gère ses propres handshakes HTTP et n'applique pas automatiquement les middlewares Express. La solution consiste à passer le même objet middleware aux deux :

```
const sessionMiddleware = session({ secret: '...', ... });

// Application à Express
app.use(sessionMiddleware);

// Application à Socket.IO
```

```
io.use((socket, next) => sessionMiddleware(socket.request, {}, next));

// Accès dans les handlers Socket.IO
const user = socket.request.session?.utilisateur;
```

## 5.2 Distinction des connexions chat / vidéo

La page **/video** et la page **/chat** utilisent toutes deux Socket.IO mais avec des besoins différents. Une connexion vidéo ne doit pas être enregistrée comme un utilisateur chat. Solution : un paramètre de requête différencie les types de connexion.

```
// Côté client — video.html
const socket = io({ query: { type: 'video' } });

// Côté serveur — routage selon le type
const type = socket.handshake.query?.type || 'chat';
if (type === 'video') {
  // Logique signalement P2P uniquement
  return;
}
// Logique chat normale...
```

## 5.3 Synchronisation asynchrone du Peer ID

Le Peer ID PeerJS est généré de façon **asynchrone** côté client, après la connexion Socket.IO. Un événement dédié **enregistrer-peer-id** est nécessaire pour mettre à jour le registre des utilisateurs et propager l'information aux autres clients.

## 5.4 Libération des ressources MediaStream

Une caméra reste active si les pistes MediaStream ne sont pas explicitement arrêtées en fin d'appel.

```
// Libération correcte des ressources en fin d'appel
function terminerAppel() {
  if (localStream) {
    localStream.getTracks().forEach(track => track.stop());
    localStream = null;
  }
  if (connexionActive) {
    connexionActive.close();
    connexionActive = null;
  }
}
```

> **Bonne pratique WebRTC**

> Toujours appeler track.stop() sur chaque piste avant de nullifier la référence au MediaStream. Sans cela, le voyant de la caméra reste allumé même après la fin de l'appel.

# 6. Base de Données

## 6.1 Schéma relationnel

| Table | Colonnes principales | Contraintes |
|---|---|---|
| utilisateurs | id (PK), nom, email, mot_de_passe, avatar, date_creation | email UNIQUE, mot_de_passe haché bcrypt |
| salons | id (PK), nom, description, cree_le | nom NOT NULL |
| messages | id (PK), contenu, utilisateur_id (FK), salon_id (FK), envoye_le | Clés étrangères vers utilisateurs et salons |

```sql
-- ==================================
-- Table utilisateurs
-- ==================================

CREATE TABLE IF NOT EXISTS utilisateurs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL,
    email VARCHAR(150) UNIQUE NOT NULL,
    mot_de_passe VARCHAR(255) NOT NULL,
    avatar VARCHAR(255),
    date_creation TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- ==================================
-- Table salons (channels de chat)
-- ==================================

CREATE TABLE IF NOT EXISTS salons (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL
);
```

```sql
-- ==================================
-- Table messages
-- ==================================

CREATE TABLE IF NOT EXISTS messages (
    id INT AUTO_INCREMENT PRIMARY KEY,
    utilisateur_id INT,
    salon_id INT,
    message TEXT NOT NULL,
    date_envoi TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (utilisateur_id) REFERENCES utilisateurs(id)
        ON DELETE CASCADE,

    FOREIGN KEY (salon_id) REFERENCES salons(id)
        ON DELETE CASCADE
);
```

```sql
-- ====================================
-- Données initiales
-- ====================================

INSERT INTO salons (nom) VALUES
('General'),
('Developpement'),
('Support'),
('IA');

INSERT INTO utilisateurs (nom,email,mot_de_passe,avatar) VALUES
('Admin','admin@test.com','Test1234!','avatar1.png'),
('Alice','alice@test.com','Test1234!','avatar2.png'),
('Bob','bob@test.com','Test1234!','avatar3.png');
```

## 6.2 Opérations CRUD

- **CREATE** — Inscription d'un utilisateur (hash bcrypt + INSERT), envoi d'un message
- **READ** — Connexion (SELECT par email + bcrypt.compare), chargement de l'historique des messages d'un salon
- **UPDATE** — Mise à jour du profil utilisateur
- **DELETE** — Suppression de compte et de messages associés

```
mysql> use smartechconnect
Database changed
mysql> show tables
    -> ;
+--------------------------+
| Tables_in_smartechconnect |
+--------------------------+
| messages                 |
| salons                   |
| utilisateurs             |
+--------------------------+
3 rows in set (0.01 sec)

mysql>
```

```
mysql> select * from messages;
+----+--------------------+----------------+----------+---------------------+
| id | contenu            | utilisateur_id | salon_id | envoye_le           |
+----+--------------------+----------------+----------+---------------------+
|  1 | bonjour            |              2 |        1 | 2026-02-23 11:33:22 |
|  2 | Ayweu akhy nakamou |              1 |        1 | 2026-02-23 11:34:09 |
|  3 | salut              |              1 |        1 | 2026-02-23 23:26:46 |
+----+--------------------+----------------+----------+---------------------+
3 rows in set (0.00 sec)

mysql>
```

```
mysql> select * from utilisateurs;
+----+--------------------+-----------------------+--------------------------------------------------------------+--------+---------------------+
| id | nom                | email                 | mot_de_passe                                                 | avatar | cree_le             |
+----+--------------------+-----------------------+--------------------------------------------------------------+--------+---------------------+
|  1 | Al Hassane Ba      | ahassane.pro@gmail.com | $2b$10$5v8ZD3cP1oZSi0l0KKpFQOuVcggHJgl2wPFVcfNqbvNIGpkwNKxhu |        | 2026-02-23 11:00:46 |
|  2 | Mouhammed Ahmad Ba | ba.ahmad@esp.sn       | $2b$10$nC4SNEAF7eaeulpUXRjdheRXhe2.JVoKndklxH3xdeCu2kERU987. |        | 2026-02-23 11:08:20 |
|  3 | Etudiant           | etud1@test.com        | $2b$10$68rmSTC07Pcee4nJU.OkLuH7Ogo2QWvaNh41Xq4fPqDMNa.4PMC1e |        | 2026-02-27 04:30:58 |
+----+--------------------+-----------------------+--------------------------------------------------------------+--------+---------------------+
3 rows in set (0.00 sec)

mysql>
```

```
mysql> select * from salons;
+----+----------+--------------------------------------+---------------------+
| id | nom      | description                          | cree_le             |
+----+----------+--------------------------------------+---------------------+
|  1 | général  | Salon principal ouvert à tous        | 2026-02-23 10:59:51 |
|  2 | maths    | Entraide et exercices de mathématiques | 2026-02-23 10:59:51 |
|  3 | info     | Questions et projets informatiques   | 2026-02-23 10:59:51 |
|  4 | physique | Physique et sciences appliquées      | 2026-02-23 10:59:51 |
+----+----------+--------------------------------------+---------------------+
4 rows in set (0.00 sec)

mysql>
```

### 6.3 Pool de connexions

L'utilisation de **createPool()** (mysql2) permet de réutiliser les connexions existantes plutôt que d'en créer une nouvelle à chaque requête. Cela améliore significativement les performances sous charge et évite les épuisements de connexions.

```
// Pool de connexions : évite de recréer une connexion à chaque requête
const pool = mysql.createPool({
  host:     process.env.DB_HOST     || 'localhost',
  user:     process.env.DB_USER     || 'hassane',
  password: process.env.DB_PASS     || 'alkamane',
  database: process.env.DB_NAME     || 'smartechconnect',
  waitForConnections: true,
  connectionLimit:    10,    // max 10 connexions simultanées
  queueLimit:         0,
  charset:            'utf8mb4'
});

// Vérification de la connexion au démarrage
pool.getConnection()
  .then(conn => {
    console.log('✅ Base de données MySQL connectée');
    conn.release();
  })
  .catch(err => {
    console.error('❌ Erreur connexion MySQL :', err.message);
    console.error('   → Vérifiez que MySQL est démarré et que les identifiants sont corrects.');
  });

module.exports = pool;
```

## 7. Description des Fonctionnalités

### 7.1 Authentification et sécurité des sessions

L'inscription vérifie l'unicité de l'email, hache le mot de passe avec **bcrypt (facteur de coût 10)** et crée la session. La connexion charge le profil et vérifie le hash via **bcrypt.compare()**. Les sessions ont une durée de vie de 8 heures.

```
  // Comparer le mot de passe avec le hash bcrypt
  const valide = await bcrypt.compare(mot_de_passe, user.mot_de_passe);
  if (!valide) {
    return res.status(401).json({ erreur: 'Email ou mot de passe incorrect.' });
  }

  // Créer la session (sans le hash du mot de passe !)
  req.session.utilisateur = {
    id:     user.id,
    nom:    user.nom,
    email:  user.email,
    avatar: user.avatar
  };

  res.json({ succes: true, message: 'Connexion réussie !', nom: user.nom });

} catch (err) {
  console.error('Erreur connexion :', err);
  res.status(500).json({ erreur: 'Erreur serveur, réessayez.' });
}
```

## 7.2 Chat temps réel et historique

Les utilisateurs rejoignent des salons correspondant à des **rooms Socket.IO**. Chaque message est instantanément persisté en base de données et diffusé à tous les membres via **io.to(`salon_${salonId}`).emit()**. L'historique est chargé à l'arrivée dans le salon.

```
 try {
   // Sauvegarder en base de données
   await db.query(
     'INSERT INTO messages (contenu, utilisateur_id, salon_id) VALUES (?, ?, ?)',
     [message.contenu, user.id, salonId]
   );

   // Diffuser à TOUS les membres du salon (émetteur inclus)
   io.to(`salon_${salonId}`).emit('message-recu', { ...message, salonId });

 } catch (err) {
   console.error('Erreur sauvegarde message :', err);
   socket.emit('erreur', { message: 'Impossible d\'envoyer le message.' });
 }
});
```

```
// helper : envoi de la liste des membres (tous les utilisateurs) d'un salon
// avec leur statut en ligne/ligne et, si connecté, leur socketId.
async function broadcastMembresSalon(salonId) {
  try {
    const [rows] = await db.query('SELECT id, nom, avatar FROM utilisateurs ORDER BY nom');
    const membres = rows.map(u => {
      const entry = { id: u.id, nom: u.nom, avatar: u.avatar, online: false, socketId: null };
      for (const [sid, data] of utilisateursConnectes.entries()) {
        if (data.id === u.id && data.salonId === salonId) {
          entry.online = true;
          entry.socketId = sid;
          break;
        }
      }
      return entry;
    });
    io.to(`salon_${salonId}`).emit('liste-salon-membres', membres);
  } catch (err) {
    console.error('Erreur récupération membres salon :', err);
  }
}
```

## 7.3 Tableau blanc collaboratif

Le tableau blanc utilise l'**API Canvas HTML5**. Les tracés (coordonnées, couleur, épaisseur) sont sérialisés en JSON et transmis via Socket.IO à tous les membres du même salon. La synchronisation est quasi-instantanée sur réseau local.

```
<!-- ── VUE TABLEAU BLANC ─────────────────────────────── -->
<div id="view-whiteboard" class="view">
  <div class="whiteboard-toolbar">
    <!-- Couleurs -->
    <div class="tool-group">
      <span class="tool-label">Couleur</span>
      <div class="color-palette">
        <button class="color-btn active" data-color="#FFFFFF" style="background:#FFFFFF;"></button>
        <button class="color-btn" data-color="#FF6B6B" style="background:#FF6B6B;"></button>
        <button class="color-btn" data-color="#FFD93D" style="background:#FFD93D;"></button>
        <button class="color-btn" data-color="#6BCB77" style="background:#6BCB77;"></button>
        <button class="color-btn" data-color="#4DABF7" style="background:#4DABF7;"></button>
        <button class="color-btn" data-color="#CC5DE8" style="background:#CC5DE8;"></button>
        <button class="color-btn" data-color="#FFA94D" style="background:#FFA94D;"></button>
      </div>
    </div>
    <!-- Taille du pinceau -->
    <div class="tool-group">
      <span class="tool-label">Taille</span>
      <input type="range" id="brush-size" min="2" max="40" value="4" />
      <span id="brush-preview" class="brush-preview"></span>
    </div>
    <!-- Outil -->
    <div class="tool-group">
      <button id="tool-pen"   class="tool-btn active" title="Crayon">✏️</button>
      <button id="tool-erase" class="tool-btn" title="Gomme">🧹</button>
    </div>
```

```
395  function resizeCanvas() {
396    const rect = canvas.getBoundingClientRect();
397    const newW = rect.width  || canvas.parentElement.clientWidth  || 800;
398    const newH = rect.height || canvas.parentElement.clientHeight || 500;
399
400    // Sauvegarder uniquement si canvas a déjà une taille valide
401    let img = null;
402    if (canvas.width > 0 && canvas.height > 0) {
403      try { img = ctx.getImageData(0, 0, canvas.width, canvas.height); } catch(e) {}
404    }
405
406    canvas.width  = newW;
407    canvas.height = newH;
408    ctx.fillStyle = '#1a1a2e';
409    ctx.fillRect(0, 0, canvas.width, canvas.height);
410    if (img) ctx.putImageData(img, 0, 0);
411  }
412
413  // Initialiser le canvas (taille fixe — pas de resize sur canvas caché)
414  window.addEventListener('load', () => {
415    canvas.width  = 800;
416    canvas.height = 500;
417    ctx.fillStyle = '#1a1a2e';
418    ctx.fillRect(0, 0, canvas.width, canvas.height);
419  });
```

## 7.4 Appels vidéo/audio P2P

Une fois la connexion établie via PeerJS, le flux vidéo circule **directement entre les navigateurs** (WebRTC), sans transiter par le serveur. Le serveur intervient uniquement pour le signalement (échange des Peer IDs). Les flux local et distant s'affichent dans des éléments `<video>`.

```
12    <header class="video-header">
13      <div class="logo-sm">🦞</div>
14      <h2>Appel Vidéo P2P — SmarttechConnect</h2>
15      <div id="statut" class="statut-badge">🔴 Initialisation…</div>
16    </header>
17
18    <div class="videos-zone">
19      <div class="video-remote-wrapper">
20        <video id="video-distant" autoplay playsinline class="video-remote"></video>
21        <div id="placeholder" class="video-placeholder">
22          <div class="placeholder-icon">👤</div>
23          <p>En attente de l'interlocuteur…</p>
24        </div>
25        <div class="video-label" id="label-distant">Interlocuteur</div>
26      </div>
27      <div class="video-local-wrapper">
28        <video id="video-local" autoplay muted playsinline class="video-local"></video>
29        <div class="video-label local-label">Vous</div>
30      </div>
31    </div>
32
```

```
127    // Identifier si c'est une connexion depuis la page vidéo
128    // La page vidéo envoie { type: 'video' } comme auth lors de la connexion
129    const typeConnexion = socket.handshake.query?.type || 'chat';
130
131    if (typeConnexion === 'video') {
132      // Connexion vidéo : ne PAS l'ajouter à utilisateursConnectes
133      // Elle sert uniquement à relayer le Peer ID pour les appels
134      console.log(`📹 Connexion vidéo : ${user.nom} (socket: ${socket.id})`);
135
136      // Relais des signaux WebRTC aussi pour les sockets "video"
137      socket.on('webrtc-signal', ({ dest, type, data }) => {
138        if (!dest) return;
139        io.to(dest).emit('webrtc-signal', { from: socket.id, type, data });
140        console.log(`🔗 WebRTC signal [${type}] (video) : ${socket.id.substring(0,8)} → ${dest.substring(0,8)}`
141      });
142
143      // Stocker le peerId associé à cet utilisateur pour le signalement
144      socket.on('enregistrer-peer-id', ({ peerId }) => {
145        // Trouver la socket chat de cet utilisateur et lui transmettre le peerId
146        for (const [sid, data] of utilisateursConnectes.entries()) {
147          if (data.id === user.id) {
148            utilisateursConnectes.set(sid, { ...data, peerId });
149            io.emit('liste-utilisateurs', Array.from(utilisateursConnectes.values()));
150            break;
151          }
152        }
153      });
```

# 8. Tests et Validation

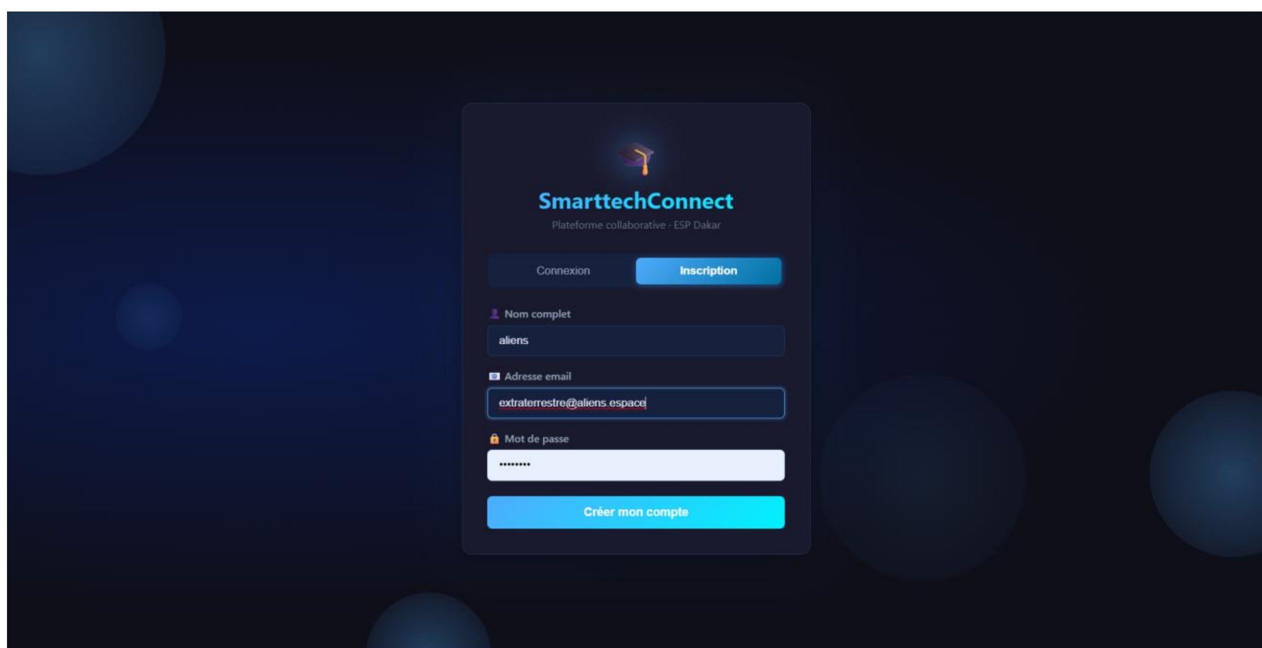
## 8.1 Résultats des tests fonctionnels

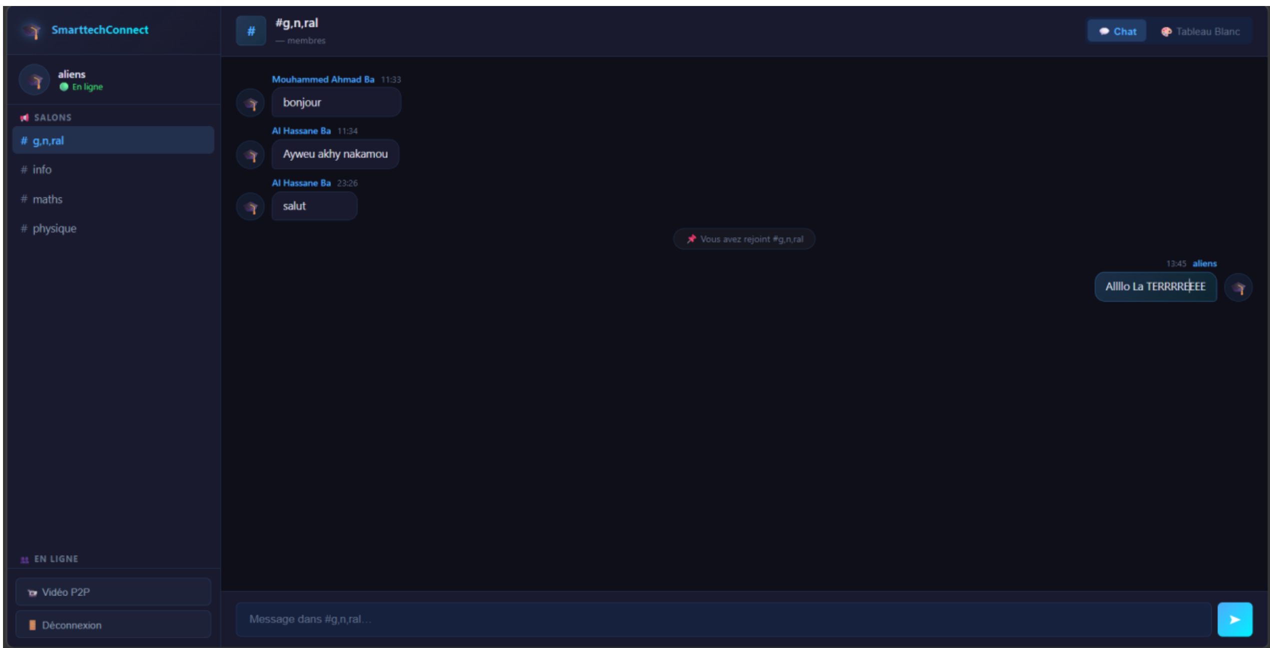
| Scénario de test | Résultat | Navigateurs testés |
|---|---|---|
| Inscription avec email déjà existant | ✓OK | Chrome |
| Connexion avec mot de passe incorrect | ✓OK | Chrome / Edge |
| Accès à /chat sans session active | ✓OK | Chrome |

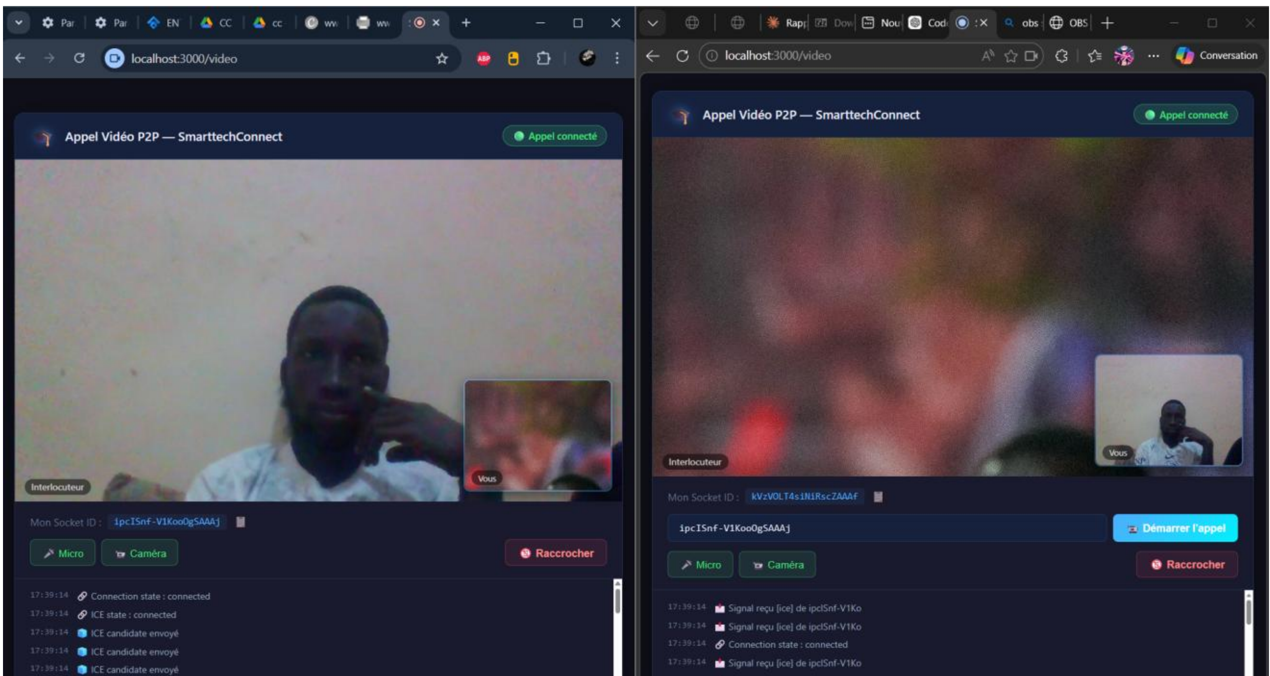
| Scénario de test | Résultat | Navigateurs testés |
|---|---|---|
| Chat entre deux clients simultanés | ✓OK | Chrome + Edge |
| Dessin collaboratif sur tableau blanc | ✓OK | Chrome + Edge |
| Appel vidéo P2P établi | ✓OK | Chrome + Edge |
| Coupure d'appel et libération de la caméra | ✓OK | Chrome + Edge |
| Déconnexion et mise à jour des connectés | ✓OK | Chrome |

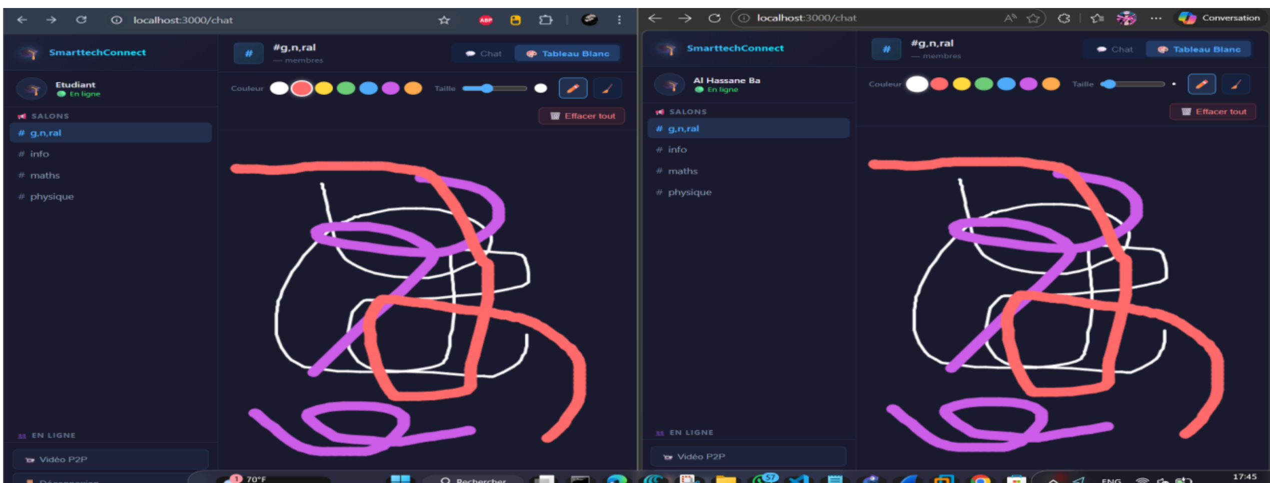
## 8.2 Conditions de test

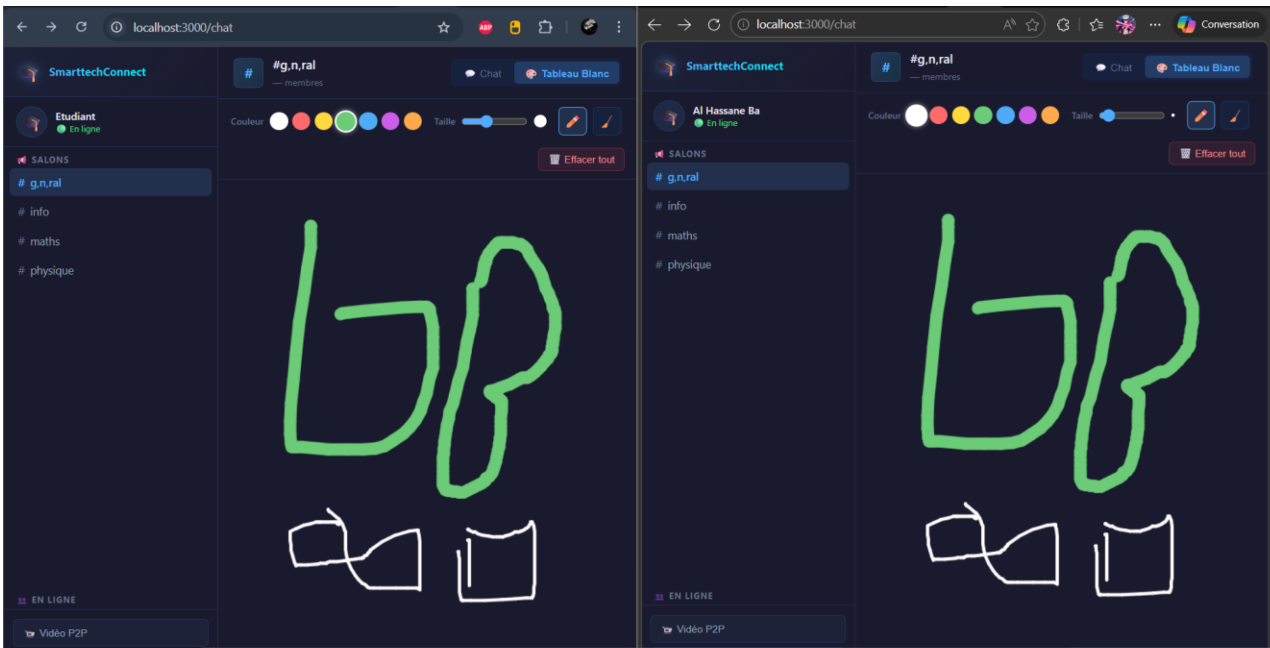
- Réseau local (LAN) — serveur sur localhost:3000
- Navigateurs : Google Chrome (dernière version) et Microsoft Edge
- Scénarios multi-clients : 2 à 3 utilisateurs simultanés
- Comptes de test : alice@esp.sn / moussa@esp.sn / fatou@esp.sn (mot de passe : Test1234!)

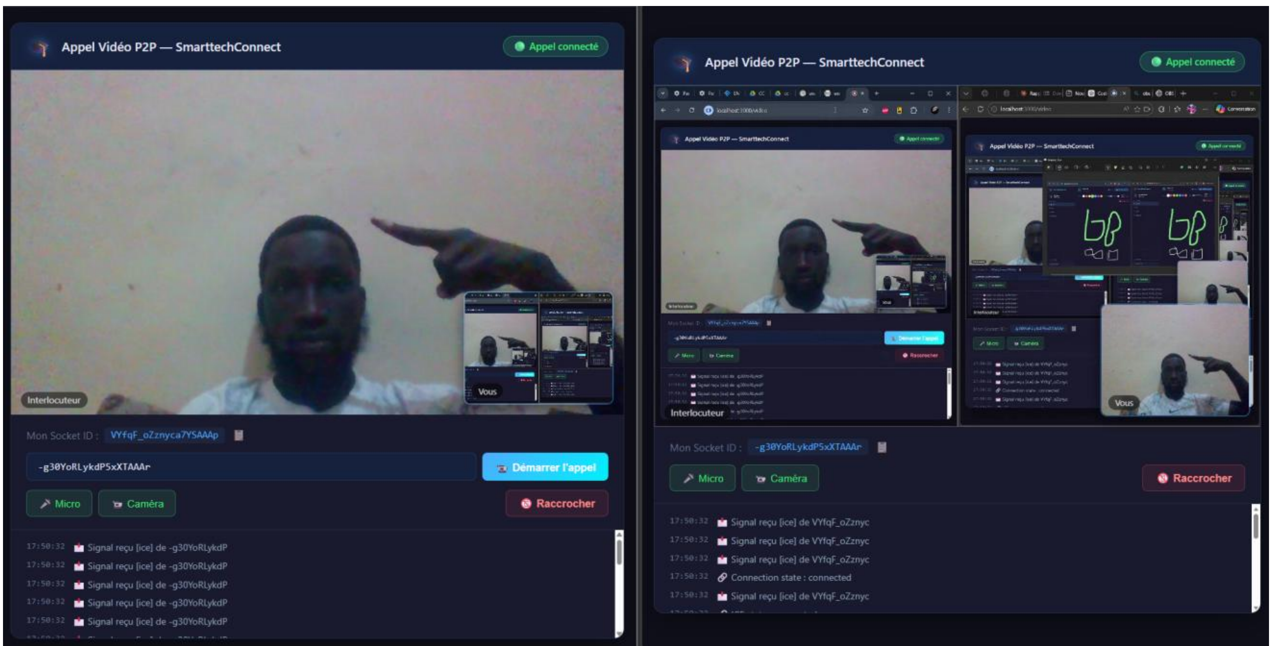



SmarttechConnect
#g,n,ral
Chat
Tableau Blanc
aliens
En ligne
SALONS
# g,n,ral
# info
# maths
# physique
Mouhammed Ahmad Ba 11:33
bonjour
Al Hassane Ba 11:34
Ayweu akhy nakamou
Al Hassane Ba 23:26
salut
Vous avez rejoint #g,n,ral
13:45 aliens
Alllo La TERRRREEEE
EN LIGNE
Vidéo P2P
Déconnexion
Message dans #g,n,ral...

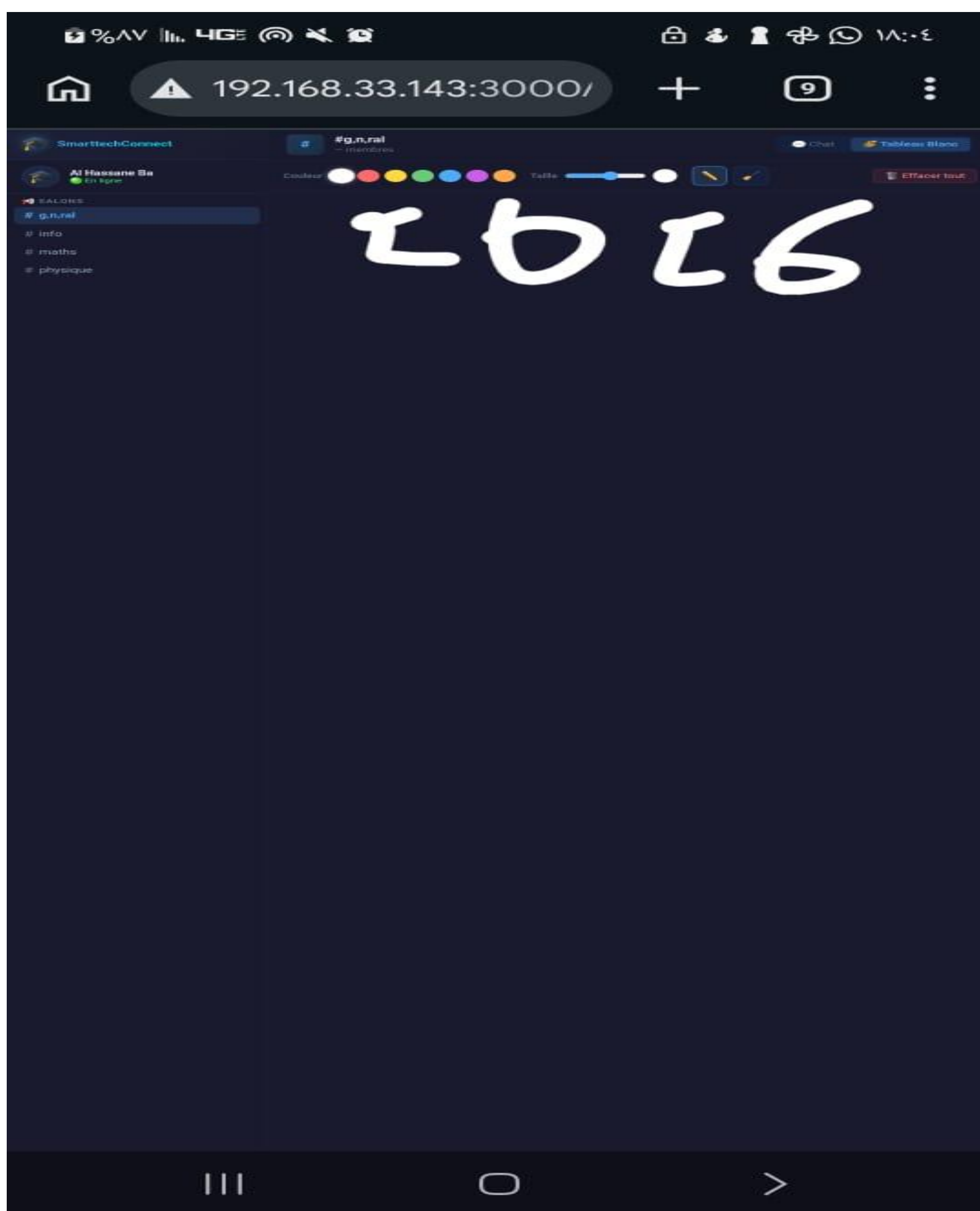

localhost:3000/video
Appel Vidéo P2P — SmarttechConnect
Appel connecté
Interlocuteur
Vous
Mon Socket ID : ipcISnf-V1KooOgSAAAj
Micro
Caméra
Raccrocher
17:39:14 Connection state : connected
17:39:14 ICE state : connected
17:39:14 ICE candidate envoyé
17:39:14 ICE candidate envoyé
17:39:14 ICE candidate envoyé
Mon Socket ID : kVzVOLT4sINiRscZAAAf
ipcISnf-V1KooOgSAAAj
Démarrer l'appel
17:39:14 Signal reçu [ice] de ipcISnf-V1Ko
17:39:14 Signal reçu [ice] de ipcISnf-V1Ko
17:39:14 Connection state : connected
17:39:14 Signal reçu [ice] de ipcISnf-V1Ko


localhost:3000/chat
SmarttechConnect
#g,n,ral
Chat
Tableau Blanc
Etudiant
En ligne
SALONS
# g,n,ral
# info
# maths
# physique
Couleur
Taille
Effacer tout
EN LIGNE
Vidéo P2P
Déconnexion
Al Hassane Ba


localhost:3000/chat
SmarttechConnect
Etudiant
En ligne
SALONS
g,n,ral
info
maths
physique
EN LIGNE
Vidéo P2P
#g,n,ral
— membres
Chat
Tableau Blanc
Couleur
Taille
Effacer tout
Al Hassane Ba
Conversation


Appel Vidéo P2P — SmarttechConnect
Appel connecté
Interlocuteur
Vous
Mon Socket ID : VYfqF_oZznyca7YSAAAp
-g30YoRLykdP5xXTAAAr
Démarrer l'appel
Micro
Caméra
Raccrocher
17:50:32 Signal reçu [ice] de -g30YoRLykdP
17:50:32 Signal reçu [ice] de -g30YoRLykdP
17:50:32 Signal reçu [ice] de -g30YoRLykdP
17:50:32 Signal reçu [ice] de -g30YoRLykdP
17:50:32 Signal reçu [ice] de -g30YoRLykdP
Mon Socket ID : -g30YoRLykdP5xXTAAAr
17:50:32 Signal reçu [ice] de VYfqF_oZznyc
17:50:32 Signal reçu [ice] de VYfqF_oZznyc
17:50:32 Signal reçu [ice] de VYfqF_oZznyc
17:50:32 Connection state : connected
17:50:32 Signal reçu [ice] de VYfqF_oZznyc

localhost:3000/chat




## 9. Conclusion

**SmarttechConnect** est une application web fullstack fonctionnelle qui intègre de manière cohérente Express.js, Socket.IO et PeerJS/WebRTC. Elle démontre concrètement les concepts théoriques vus en cours : architecture client-serveur, protocoles HTTP et WebSocket, modèle événementiel Node.js, et négociation WebRTC pair-à-pair.

Les principaux apprentissages de ce projet sont :

- **Séparation des responsabilités** — chaque couche (HTTP, WebSocket, P2P, BDD) a un rôle clairement défini
- **Réutilisation des composants** — la session Express est partagée avec Socket.IO, évitant toute duplication
- **Gestion des ressources** — libération correcte des connexions WebRTC et des pistes MediaStream
- **Sécurité by design** — hachage bcrypt, protection des routes, validation côté serveur

## Perspectives d'amélioration

- **HTTPS en production** — obligatoire pour getUserMedia hors localhost
- **Serveur TURN** — pour traverser les NAT stricts et les réseaux d'entreprise
- **Persistance des sessions en BDD** — via connect-mysql2 pour la résilience
- **Chiffrement E2E** — chiffrement bout-à-bout des messages du chat
- **Notifications push** — alertes d'appels entrants même hors de la page vidéo

**Sources et références**

Documentation officielle : Socket.IO (socket.io/docs), PeerJS (peerjs.com), Express.js (expressjs.com), MDN WebRTC API (developer.mozilla.org), bcrypt.js (npmjs.com/package/bcrypt), mysql2 (npmjs.com/package/mysql2).