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

8 décembre 2023

Table des matières

LISTINGS

Table des figures

1 Introduction

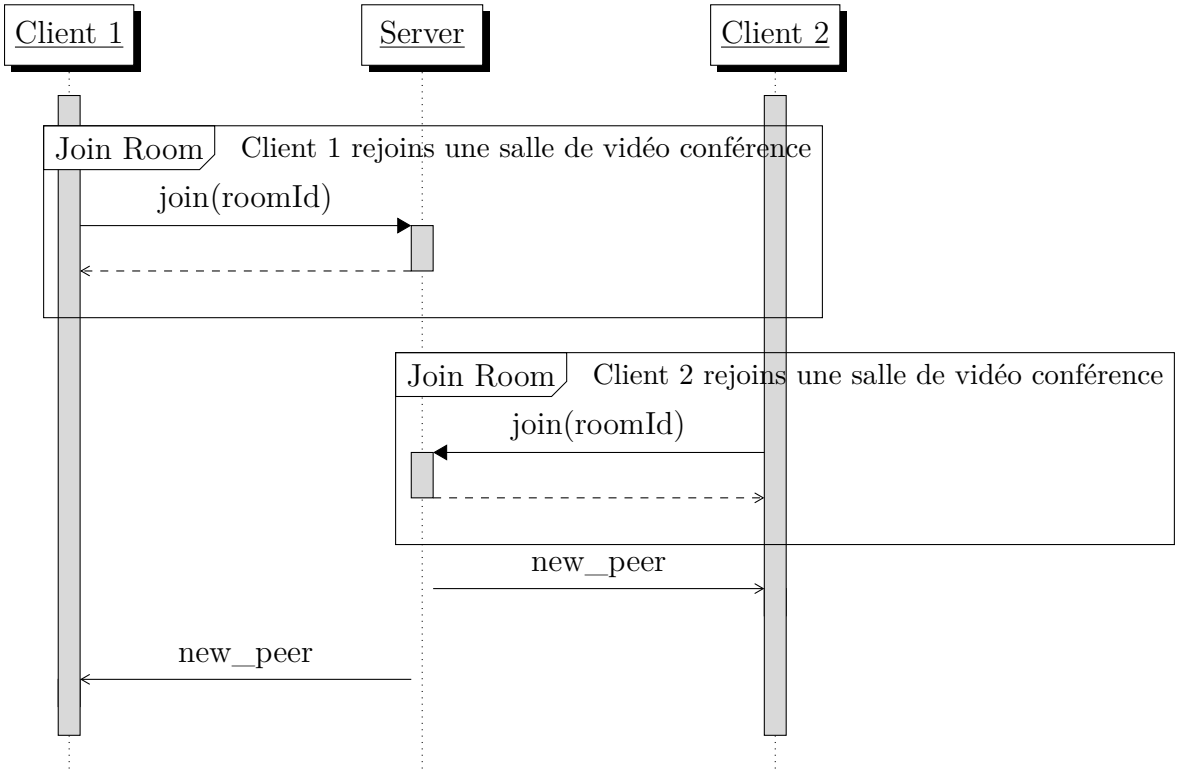
Dans un monde de plus en plus connecté, la communication en temps réel via Internet est devenue une nécessité cruciale, tant pour les interactions personnelles que professionnelles. Notre projet pour la SAE5.ROM.03, s'inscrit dans cette dynamique en offrant une solution de communication basée sur la 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.

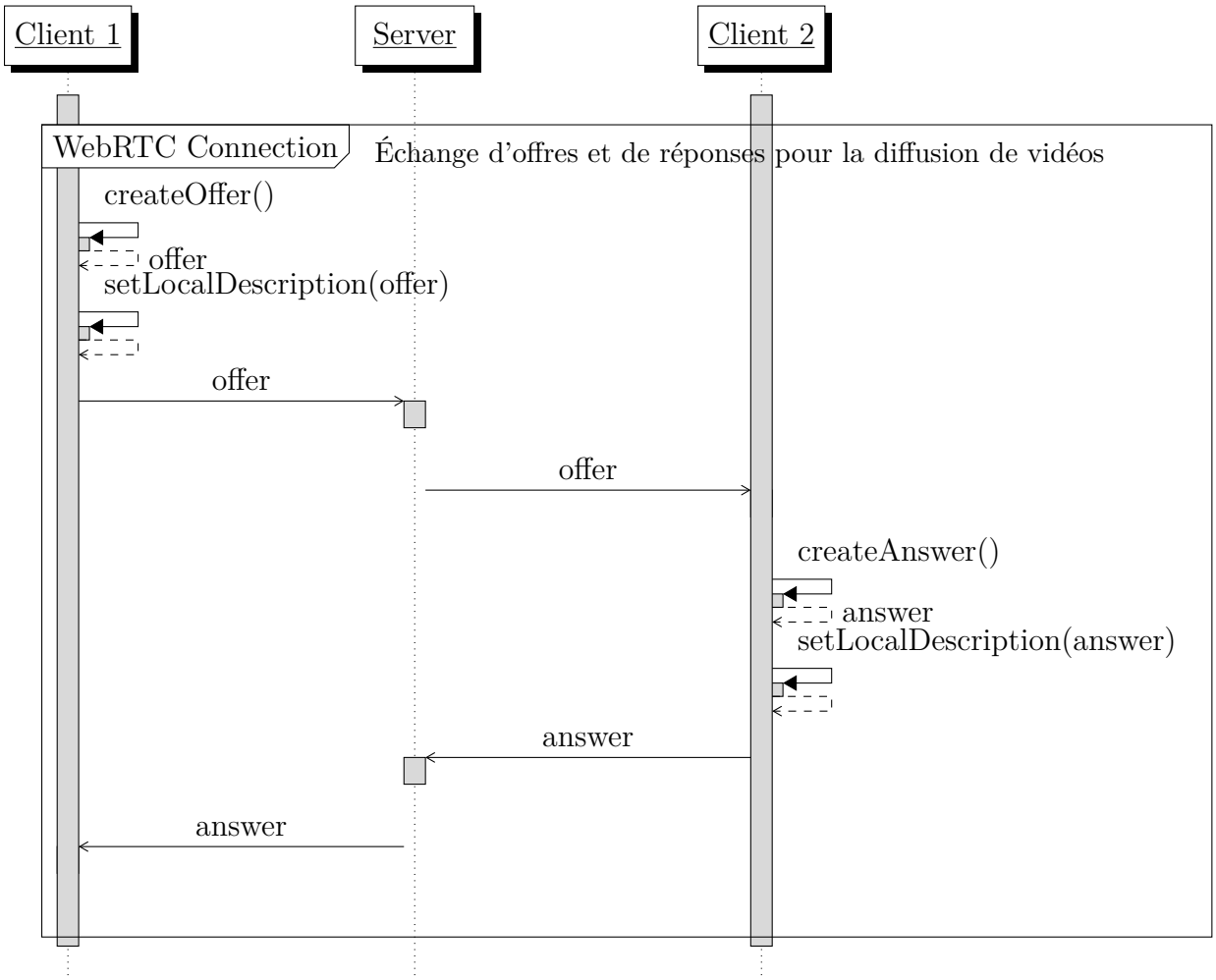
L'accent est mis sur la facilité d'utilisation. Les utilisateurs peuvent créer et rejoindre des salles de conférence virtuelles en quelques clics, tout en bénéficiant d'une connexion fiable. De plus, l'application prend en charge plusieurs participants.

2 Diagrammes de séquence

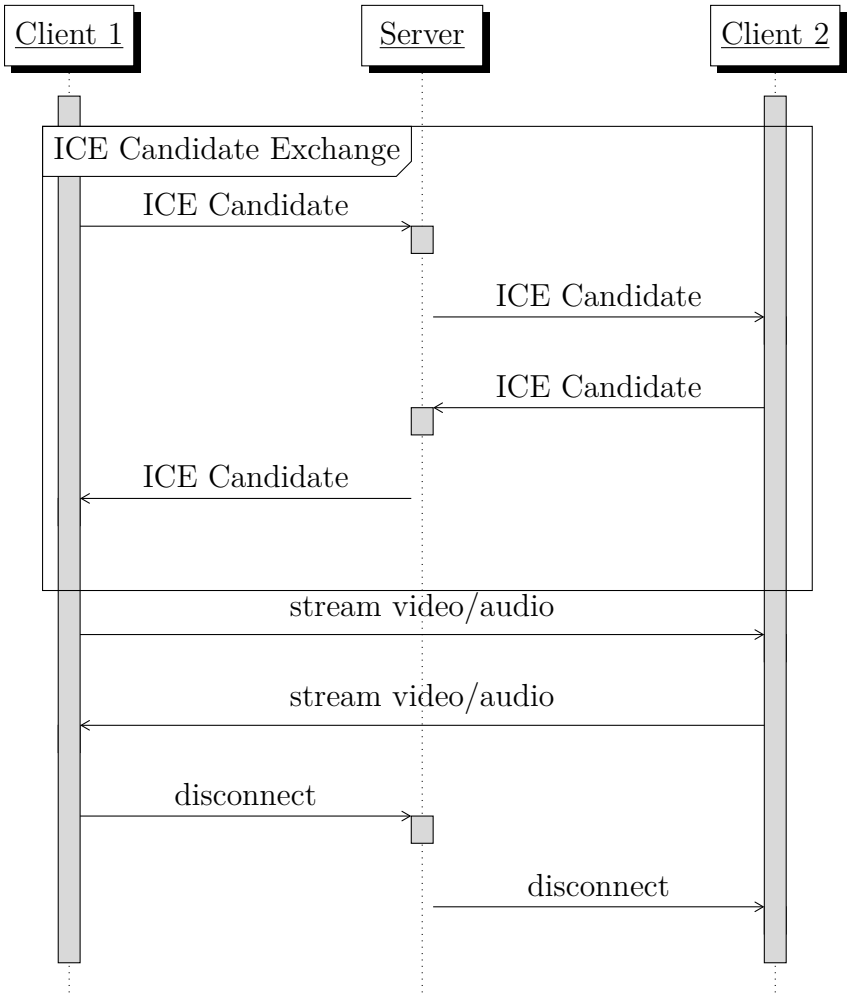
2.1 Connexion à la salle de réunion



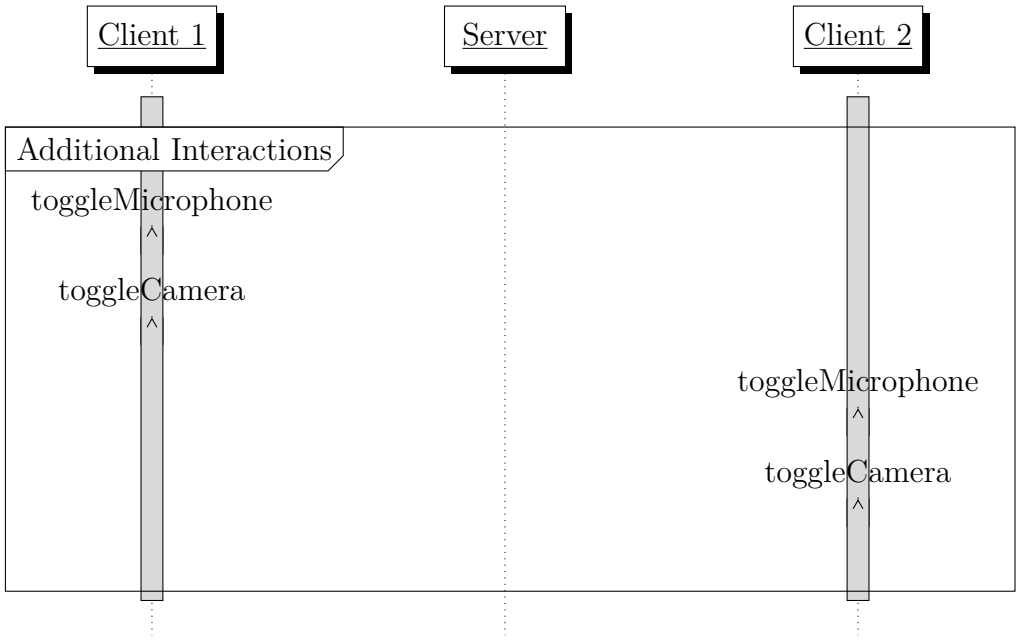
2.2 Connection WebRTC



2.3 Échanges de candidats ICE



2.4 fonctionnalités supplémentaires



3 Utilisation de l'application

4 Fonctionnement de l'application

4.1 DOM Elements

Cette section récupère et stocke des références à divers éléments du DOM qui seront manipulés ou utilisés tout au long du script.

```
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

4.2 Variables

```
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

— `socket` : Crée une connexion socket.io pour la communication en temps réel avec le serveur.
— `mediaConstraints` : Spécifie les contraintes des médias (audio et vidéo) pour WebRTC.
— `localStream` : Représente le flux média local (audio et vidéo) de l'utilisateur.
— `roomId` : Stocke l'ID de la salle de chat actuelle.
— `peerConnections` : Un dictionnaire pour stocker les connexions peer WebRTC.
— `iceServers` : Contient les serveurs STUN et TURN utilisés pour la traversée de NAT et le relais.

4.3 Button Listeners

```
  connectButton.addEventListener('click', () => {
  joinRoom(roomInput.value);
});

const hangUpButton = document.getElementById('hangup-button');
const toggleMicButton = document.getElementById('toggle-mic-button');
const toggleCameraButton =
   document.getElementById('toggle-camera-button');

hangUpButton.addEventListener('click', hangUpCall);
toggleMicButton.addEventListener('click', toggleMicrophone);
toggleCameraButton.addEventListener('click', toggleCamera);

let isRoomCreator = false;
```

3: Button Listeners

— `connectButton.addEventListener` : Écouteur d'événements pour le bouton de connexion pour rejoindre une salle.
— `hangUpButton.addEventListener` : Écouteur d'événements pour le bouton de raccrochage.
— `toggleMicButton.addEventListener` : Écouteur d'événements pour activer/désactiver le microphone.
— `toggleCameraButton.addEventListener` : Écouteur d'événements pour activer/-désactiver la caméra.

4.4 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

- `socket.on('room_created')` : Gère l'événement de création de salle.
- `socket.on('room_joined')` : Gère l'événement de rejoindre une salle.
- `socket.on('full_room')` : Gère l'événement lorsque la salle est pleine.
- `socket.on('start_call')` : Gère le début d'un appel WebRTC.

— `socket.on('webrtc_offer')` : Gère la réception d'une offre WebRTC.
— `socket.on('webrtc_answer')` : Gère la réception d'une réponse WebRTC.
— `socket.on('webrtc_ice_candidate')` : Gère la réception d'un candidat ICE.
— `socket.on('new_peer')` : Gère l'ajout d'un nouveau pair dans la salle.

4.5 Fonctions Principales

```
1     async function joinRoom(room) {
2     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      });
62    }
63
64    async function setupPeerConnection(peerId, isInitiator) {
65      const peerConnection = new RTCPeerConnection(iceServers);
66
67      if (localStream) {
68        localStream.getTracks().forEach(track =>
              peerConnection.addTrack(track, localStream));
69      } else {
70        console.error("Local stream is not defined in
              setupPeerConnection");
71        return;
72      }
73
74      peerConnection.ontrack = (event) =>
           addRemoteStream(event.streams[0], peerId);
75      peerConnection.onicecandidate = (event) => handleIceEvent(event,
           peerId);
76
77      peerConnections[peerId] = peerConnection;
78
79      if (isInitiator) {
80        const offer = await peerConnection.createOffer();
81        await peerConnection.setLocalDescription(offer);
82        socket.emit('webrtc_offer', { roomId, sdp: offer, peerId });
83      }
84    }
85
86    function addRemoteStream(event, peerId) {
87      let remoteVideoElement =
           document.getElementById(`remote-video-${peerId}`);
88      if (!remoteVideoElement) {
89        remoteVideoElement = createRemoteVideoElement(peerId);
90      }
91
92      let stream;
93      if (event.streams && event.streams.length > 0) {
94        // Use the stream from the event if it exists
95        stream = event.streams[0];
96      } else {
97        // Create a new stream and add the track to it
98        stream = new MediaStream();
99        if (event.track) {
100         stream.addTrack(event.track);
101       }
102     }
103
104     // Additional logging to debug
105     console.log(`Adding remote stream for peer ${peerId}`, stream);
106     if (stream.getTracks().length === 0) {
107       console.error('No tracks in the remote stream');
108     }
109
110     remoteVideoElement.srcObject = stream;
111     remoteVideoElement.muted = false; // Ensure remote video is not
           muted
112     remoteVideoElement.volume = 1; // Ensure volume is set to maximum
113   }
114
```

```
115   function createRemoteVideoElement(peerId) {
116     const remoteVideo = document.createElement('video');
117     remoteVideo.id = `remote-video-${peerId}`;
118     remoteVideo.autoplay = true;
119     remoteVideo.playsInline = true;
120     remoteVideo.classList.add('remote-video', ...tailwindClasses);
121
122     document.getElementById('remote-videos-container')
123           .appendChild(remoteVideo);
124     return remoteVideo;
125   }
126
127
128   async function handleIceEvent(event, peerId) {
129     if (event.candidate) {
130       socket.emit('webrtc_ice_candidate', {
131         roomId,
132         candidate: event.candidate,
133         peerId
134       });
135     }
136   }
137
138   async function createPeerConnection(peerId) {
139     const peerConnection = new RTCPeerConnection(iceServers);
140
141     \begin{verbatim}
142     localStream.getTracks().forEach(track =>
          peerConnection.addTrack(track, localStream));
143     \end{verbatim}
144       // localStream.getTracks().forEach(track =>
            peerConnection.addTrack(track, localStream));
145     \end{verbatim}
146     localStream.getTracks().forEach(track =>
          peerConnection.addTrack(track, localStream));
147
148     // Gestion de l'ajout des streams distants
149     peerConnection.ontrack = (event) => addRemoteStream(event, peerId);
150
151     // Gestion de l'ajout des streams distants
152     peerConnection.ontrack = (event) => {
153     let remoteVideoElement =
          document.getElementById(`remote-video-${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       // Process cached ICE candidates
208       if (peerConnection.cachedIceCandidates) {
209         peerConnection.cachedIceCandidates.forEach(cachedCandidate => {
210           peerConnection.addIceCandidate(new
                  RTCIceCandidate(cachedCandidate));
211         });
212         peerConnection.cachedIceCandidates = [];
213       }
214
215       const answer = await peerConnection.createAnswer();
216       await peerConnection.setLocalDescription(answer);
217
218       socket.emit('webrtc_answer', {
219         type: 'webrtc_answer',
220         sdp: answer,
221         roomId,
222         peerId: data.peerId,
223       });
224     } catch (error) {
225       console.error(`Erreur dans handleOffer pour le pair
              ${data.peerId}:`, error);
226     }
227   }
228
229   async function handleAnswer(data) {
230     const peerConnection = peerConnections[data.peerId];
```

```
231     console.log(`Peer connection state before setting remote
            description: ${peerConnection.signalingState}`);
232
233     if (peerConnection.signalingState === 'have-local-offer') {
234       try {
235         await peerConnection.setRemoteDescription(new
                RTCSessionDescription(data.sdp));
236         console.log(`Remote description set for peer ${data.peerId}`);
237       } catch (error) {
238         console.error(`Error in handleAnswer for peer ${data.peerId}:`,
                error);
239       }
240     } else {
241       console.log(`Peer connection not in the correct state to set
              remote description, current state:
              ${peerConnection.signalingState}`);
242     }
243   }
244
245
246   async function handleIceCandidate(data) {
247     const peerConnection = peerConnections[data.peerId];
248     if (peerConnection) {
249       if (!peerConnection.remoteDescription) {
250         console.log("Queueing ICE candidate as remote description is
                not yet set");
251         if (!peerConnection.cachedIceCandidates) {
252           peerConnection.cachedIceCandidates = [];
253         }
254         peerConnection.cachedIceCandidates.push(data.candidate);
255       } else {
256         console.log("Adding ICE candidate");
257         await peerConnection.addIceCandidate(new
                RTCIceCandidate(data.candidate));
258       }
259     }
260   }
261
262
263
264   function handleNewICECandidateMsg(data) {
265     const peerConnection = peerConnections[data.peerId];
266     peerConnection.addIceCandidate(new RTCIceCandidate(data.candidate));
267   }
268
269   function sendIceCandidate(candidate, peerId) {
270     socket.emit('webrtc_ice_candidate', {
271       roomId,
272       candidate,
273       peerId,
274     });
275   }
276
277   function addVideoStream(videoElement, stream, isLocal = false) {
278     \texttt{console.log('Adding videoElement.srcObject = stream;')}
279     videoElement.srcObject = stream;
280     videoElement.autoplay = true;
281     videoElement.playsInline = true;
282     videoElement.muted = isLocal;
283     if (isLocal) {
284       videoElement.id = 'local-video';
285       videoElement.style.backgroundColor = 'red';
286     } else {
```

```
287       videoElement.classList.add('remote-video');
288       videoElement.style.backgroundColor = 'green';
289     }
290     videoChatContainer.appendChild(videoElement);
291   }
292
293   function handleRemoteStreamAdded(stream, peerId) {
294     console.log(`handleRemoteStreamAdded called with peerId:
          ${peerId}`);
295     let videoElementId = `remote-video-${peerId}`;
296     let remoteVideoElement = document.getElementById(videoElementId);
297
298     if (!remoteVideoElement) {
299       console.log(`Creating new video element for peer ${peerId}`);
300       remoteVideoElement = document.createElement('video');
301       remoteVideoElement.id = videoElementId;
302       remoteVideoElement.autoplay = true;
303       remoteVideoElement.playsInline = true;
304       remoteVideoElement.classList.add('remote-video');
305       document.getElementById('remote-videos-container')
306             .appendChild(remoteVideoElement);
307     }
308     else {
309       console.log(`Replacing video element for peer ${peerId}`);
310     }
311
312     remoteVideoElement.srcObject = stream;
313   }
314
315
316
317   function hangUpCall() {
318     console.log("Hang Up Call");
319     for (let peerId in peerConnections) {
320       peerConnections[peerId].close();
321       delete peerConnections[peerId];
322     }
323
324     if (localStream) {
325       localStream.getTracks().forEach(track => track.stop());
326       localStream = null;
327     }
328
329     let remoteVideosContainer =
          document.getElementById('remote-videos-container');
330     while (remoteVideosContainer.firstChild) {
331       remoteVideosContainer.removeChild(remoteVideosContainer.firstChild);
332     }
333
334     const waitingText = document.getElementById('waitingText');
335     if (waitingText) {
336       waitingText.style.display = 'block';
337     }
338
339     videoChatContainer.style.display = 'none';
340
341     roomSelectionContainer.style.display = 'none';
342   }
343
344
345   function toggleMicrophone() {
346     const audioTrack = localStream.getAudioTracks()[0];
347     if (audioTrack) {
```

```
348         audioTrack.enabled = !audioTrack.enabled;
349         console.log("Microphone toggled. Now enabled:",
                audioTrack.enabled);
350
351         // Update the track on all peer connections
352         for (let peerId in peerConnections) {
353           const sender = peerConnections[peerId].getSenders().find(s =>
                s.track.kind === audioTrack.kind);
354           if (sender) {
355             sender.replaceTrack(audioTrack);
356           }
357         }
358       }
359     }
360
361
362     function toggleCamera() {
363       const videoTrack = localStream.getVideoTracks()[0];
364       if (videoTrack) {
365         videoTrack.enabled = !videoTrack.enabled;
366         console.log("Camera toggled. Now enabled:", videoTrack.enabled);
367
368         // Update the track on all peer connections
369         for (let peerId in peerConnections) {
370           const sender = peerConnections[peerId].getSenders().find(s =>
                s.track.kind === videoTrack.kind);
371           if (sender) {
372             sender.replaceTrack(videoTrack);
373           }
374         }
375       }
376     }
```

5: Fonctions Principales

- `async function joinRoom(room)` : Rejoint une salle de chat en utilisant WebRTC.
- `function showVideoConference()` : Affiche l'interface de la vidéoconférence.
- `async function setLocalStream(mediaConstraints)` : Configure le flux média local.
- `async function handleNewPeer(peerId)` : Gère l'ajout d'un nouveau pair.
- `async function setupPeerConnection(peerId, isInitiator)` : Configure la connexion peer WebRTC.
- `function addRemoteStream(event, peerId)` : Ajoute un flux média distant à l'élément vidéo.
- `function createRemoteVideoElement(peerId)` : Crée un nouvel élément vidéo pour un pair distant.
- `async function handleIceEvent(event, peerId)` : Traite les événements de candidat ICE.
- `async function createPeerConnection(peerId)` : Crée une nouvelle connexion peer WebRTC.
- `async function handleOffer(data)` : Traite une offre WebRTC reçue.
- `async function handleAnswer(data)` : Traite une réponse WebRTC reçue.
- `async function handleIceCandidate(data)` : Traite un candidat ICE WebRTC reçu.
- `function handleNewICECandidateMsg(data)` : Ajoute un nouveau candidat ICE à la connexion peer.
- `function sendIceCandidate(candidate, peerId)` : Envoie un candidat ICE au serveur.

— `function addVideoStream(videoElement, stream, isLocal)` : Ajoute un flux vidéo à un élément vidéo.
— `function handleRemoteStreamAdded(stream, peerId)` : Gère l'ajout d'un flux média distant.
— `function hangUpCall()` : Gère la fin d'un appel.
— `function toggleMicrophone()` : Active ou désactive le microphone.
— `function toggleCamera()` : Active ou désactive la caméra.

5 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.

5.1 Contexte de l'Application

5.2 La Navigation

5.3 Analyse de l'UI (Interface Utilisateur)

5.4 Analyse de l'UX (Expérience Utilisateur)

5.5 Recommandations et Améliorations

6 Annexes

6.1 Sous-titre 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse auctor elit vitae mauris dignissim bibendum. Fusce facilisis, sapien sed varius finibus, quam lacus auctor lorem, ac dapibus sapien ante quis odio. Sed tincidunt pharetra dui, in ultricies enim tincidunt ac. Suspendisse quis tincidunt justo. Duis interdum vitae ipsum ac venenatis. Nullam bibendum ex ac nisl tristique, vel euismod ex faucibus.

6.2 Sous-titre 2

```
1  def hello(name):
2    print("Hello, " + name + "!")
3
4  hello("World")
```

6: Exemple de code Python