Chapitre 1

Configuration des Interfaces WebRTC

1.1 Introduction

WebRTC offre des opportunités sans précédent aux développeurs qui souhaitent intégrer des communications en temps réel dans leurs applications.
Les API WebRTC getUserMedia, RTCPeerConnection et RTCDataChannel jouent chacune leur propre rôle dans la capture, la transmission et la diffusion de données en temps réel (depuis la webcam et le microphone d'un ordinateur) vers un autre navigateur, sans qu'un utilisateur ait à télécharger des plug-ins ou des modules complémentaires[**?**].

1.2 Signalisation

La spécification WebRTC inclut des API pour communiquer avec un serveur ICE , mais le composant de signalisation n'en fait pas partie(**??**). La signalisation est nécessaire pour que deux pairs partagent la façon dont ils doivent se connecter. Habituellement, cela est résolu via une API Web standard basée sur HTTP (c'est-à-dire un service REST ou un autre mécanisme RPC) où les applications Web peuvent relayer les informations nécessaires avant que la connexion entre homologues ne soit lancée[**?**].
L'extrait de code provenant de [**?**] suivant, montre comment un service de signalisation fictif peut être utilisé pour envoyer et recevoir des messages de manière asynchrone.

```
// Configurer un canal de communication asynchrone qui
// sera utilisé lors de la configuration de la connexion peer
const signalingChannel = new SignalingChannel(remoteClientId);
signalingChannel.addEventListener('message', message => {
  // Nouveau message du client distant reçu
});

// Envoyer un message asynchrone au client distant
signalingChannel.send('Hello!');
```

1.3 RTCPeerConnection

L'API RTCPeerConnection est chargée de connecter deux (02) navigateurs ensemble afin qu'ils puissent partager des médias en temps réel[**?**]. Elle est au cœur de la connexion peer-to-peer entre chacun des navigateurs.

1.3.1 RTCConfiguration

Chaque connexion peer-to-peer est gérée par un objet RTCPeerConnection. Le constructeur de cette classe prend un seul objet RTCConfiguration comme paramètre. Cet objet définit la configuration de la connexion peer-to-peer et doit contenir des informations sur les serveurs ICE à utiliser [1] [**?**].
L'établissement de connectivité interactive (ICE) est une technique utilisée dans les réseaux informatiques pour trouver des moyens pour que deux ordinateurs communiquent aussi directement que possible dans les réseaux peer-to-peer. Il est le plus souvent utilisé pour les médias interactifs tels que la voix sur IP (VoIP), les communications peer-to-peer, la vidéo et la messagerie instantanée. Dans de telles applications, mieux vaut éviter de communiquer via un serveur central (ce qui ralentirait la communication et serait coûteux), mais la communication directe entre les applications clientes sur Internet est très délicate en raison des traducteurs d'adresses réseau (NAT), des pare-feu et d'autres barrières du réseau[**?**].

1. Le dictionnaire complet des membres de RTCConfiguration peut être accédée à l'addresse https ://developer.mozilla.org/en-US/docs/Web/API/RTCConfiguration

1.3.2 Initialisation

Une fois la configuration effectuée il suffit d'instancier l'interface de l'API RTCPeerConnection afin d'initialiser une connexion et ce sans oublier de fournir l'argument configuration dans la signature du constructeur. Le bout de code suivant, tiré de [**?**] montre comment effectuer celà.

```
const configuration: RTCConfiguration = {
  iceServers: [{'urls': 'stun:stun.l.google.com:19302'}],
};
const peerConnection = new RTCPeerConnection(configuration);
```

1.3.3 RTCSessionDescription

RTCPeerConnection coordonne l'échange de métadonnées cruciales entre deux navigateurs. Ces données définissent l'adresse IP et le numéro de ports publiquement identifiables d'un navigateur afin que les médias en temps réel puissent être échangés.
Pour que deux points de terminaison WebRTC commencent à se parler, trois types d'informations doivent être relayés :

1. Les informations de contrôle de session déterminent quand initialiser, fermer et modifier les sessions de communication.
2. Les données réseau relayent l'adresse IP et le numéro de port de chaque point de terminaison afin que les appelants puissent trouver les personnes appelées.
3. Les données média concernent les codecs et les types de médias que les appelants ont en commun.

Ces informations seront encapsulées dans un objet RTCSessionDescription.
L'interface RTCSessionDescription décrit une extrémité d'une connexion (ou connexion potentielle) et comment elle est configurée. Chaque objet RTCSessionDescription consiste en un type de description indiquant quelle partie du processus de négociation il décrit (offre ou réponse) et du descripteur SDP de la session[**?**].
Le processus de négociation d'une connexion entre deux pairs implique l'échange d'objets RTCSessionDescription dans les deux sens, par le biais du canal de signalisation(**??**), chaque description suggérant une combinaison d'option de configuration de connexion que l'expéditeur de la description prend en charge. Une fois que les deux pairs se sont mis d'accord sur une configuration pour la connexion, la négociation est terminée[**?**].

1.3.3.1 Initialisation, transmission et négotiation

Voici comment une description de session sera initiée en tant qu'une offre puis envoyée de la part du client local vers un client distant, sans oublier comment ce même client négociera une potentielle réponse de son correspondant[**?**].

```
// Écouter la récéption d'un message sur le canal de signalisation
signalingChannel.addEventListener('message', async message => {
  if (message.answer) { // Si le message comporte une réponse
    // assigner la réponse en tant que description de session distante
    const remoteDesc = new RTCSessionDescription(message.answer);
    await peerConnection.setRemoteDescription(remoteDesc);
  }
});

// Créer une description de session associée à l'offre
const offer = await peerConnection.createOffer();
```

```
await peerConnection.setLocalDescription(offer);
// Puis envoyer cette offre au client distant
signalingChannel.send({'offer': offer});
```

Et ci-après, comment du côté d'un client distant auquel sera demandée une communication, la négociation d'une offre qui lui aboutit sera faite[**?**].

```
// Écouter la récéption d'un message sur le canal de signalisation
signalingChannel.addEventListener('message', async message => {
  if (message.offer) { // Si le message contient une offre
        // assigner l'offre en tant que description de session distante
    const remoteDesc = new RTCSessionDescription(message.offer);
    peerConnection.setRemoteDescription(remoteDesc);

    // Puis créer une description de session associée à la réponse
    const answer = await peerConnection.createAnswer();
    await peerConnection.setLocalDescription(answer);

        // Efin renvoyer cette réponse au client ayant envoyé l'offre
    signalingChannel.send({'answer': answer});
  }
});
```

1.3.4 Candidats ICE

Avant que deux pairs puissent communiquer en utilisant WebRTC, ils doivent échanger des informations de connectivité. Étant donné que les conditions du réseau peuvent varier en fonction d'un grand nombre de facteurs, un service externe est généralement utilisé pour découvrir les candidats possibles pour se connecter à un pair. Ce service s'appelle ICE et utilise un serveur STUN ou TURN. STUN signifie «Session Traversal Utitlities for NAT», et est généralement utilisé indirectement dans la plupart des applications WebRTC[**?**].
TURN (Traversal Using Relay NAT) est la solution la plus avancée qui intègre les protocoles STUN et la plupart des services commerciaux basés sur WebRTC utilsent un serveur TURN pour établir des liens entre pairs. L'API WebRTC prend en charge à la fois STUN et TURN directement et il est regroupé sous le terme complet «Internet Connectivity Establishment»[**?**]. Lors de la création d'une connexion WebRTC, nous fournissons généralement un ou plusieurs serveurs ICE dans la configuration de RTCPeerConnection(1.3.1).

Provisionnement incrémentiel (ruisselement ICE)

Une fois qu'un objet RTCPeerConnection est créé, le framework sous-jacent utilise les serveurs ICE pour rassembler les candidats à L'établissement de la connectivité (candidats ICE). L'évènement 'icegatheringstatechange' de l'API RTCPeerConnection signale à quel état se trouve la collecte ICE (`'new'`, 'gathering' ou alors 'complete')[**?**].
Bien qu'il soit possible pour un pair d'attendre la fin de la collecte ICE, il est généralement beaucoup plus efficace d'utiliser une technique de «ruissellement» et de transmettre chaque candidat ICE au pair distant au fur et à mesure qu'il en est découvert. Cela résulte par une réduction considérable du temps d'installation de la connectivité des pairs et permet d'entamer une conversation vidéo avec moins de retard[**?**].
Pour rassembler des candidats ICE, il suffit d'ajouter un écouteur pour l'évènement 'icecandidate'. L'événement RTCPeerConnectionICEevent émis sur cet écouteur contiendra la propriété 'candidate' qui représente un nouveau candidat qui devrait être envoyé au pair distant(**??**).

```
// Écouter les candidats ICE locaux sur l'objet RTCPeerConnection local
peerConnection.addEventListener('icecandidate', event => {
  if (event.candidate) {
    signalingChannel.send({'new-ice-candidate': event.candidate});
  }
});
// Écouter les candidats ICE distants et les ajouter
// à la connexion RTCPeerConnection locale
signalingChannel.addEventListener('message', async message => {
  if (message.iceCandidate) {
    try {
      await peerConnection.addIceCandidate(message.iceCandidate);
    } catch (e) {
      console.error('Error adding received ice candidate', e);
    }
  }
});
```

Une fois que les candidats ICE sont reçus, nous devons nous attendre à ce que l'état de notre connexion pair passe éventuellement à un état connecté. Pour détecter cela, nous ajoutons un écouteur à notre RTCPeerConnection où nous écoutons les événements 'connectionstatechange'[**?**].

1.4 getUserMedia

L'API getUserMedia permet d'accéder aux flux multimédias (vidéo, audio ou les deux) à partir des périphériques locaux. À elle seule, cette API est capable d'acquérir de l'audio et de la vidéo, sans envoyer les données ni les stocker dans un fichier. Pour avoir un chat fonctionnel, nous devons envoyer ces données en utilisant l'API RTCPeerConnection.
À elle seule, cette API est uniquement capable d'acquérir de l'audio et de la vidéo, sans envoyer les données ni les stocker dans un fichier. Pour avoir un chat fonctionnel, nous devons envoyer ces données en utilisant l'API RTCPeerConnection[**?**].

1.4.1 Mediadevices

L'interface Navigator représente l'état et l'identité de l'agent utilisateur. Il permet aux scripts de l'interroger et de s'enregistrer pour effectuer certaines activités. Un objet Navigator peut être récupéré à l'aide de la propriété window.navigator[**?**].
L'interface Navigator dans le navigateur fournit des fonctions et des propriétés pour accéder à l'état et aux fonctionnalités du navigateur grâce auxquelles nous pouvons obtenir l'état du navigateur ou accéder à un large éventail de fonctionnalités. Par exemple, en utilisant navigator.online, nous pouvons obtenir l'état de connexion du navigateur avec Internet[**?**].
L'interface MediaDevices comprise dans Navigator, permet d'accéder aux périphériques d'entrée multimédia connectés tels que les caméras et les microphones, ainsi que le partage d'écran. En substance, il vous permet d'accéder à n'importe quelle source matérielle de données multimédias[**?**].
Les fonctions de l'objet mediaDevices fournissent des fonctionnalités telles que le partage d'écran, l'obtention de flux depuis la caméra et les microphones. La fonction getUserMedia est celle dont nous avons besoin pour récupérer les flux de la caméra et des microphones[**?**].

1.4.2 Capture des médias et contraintes

La méthode getUsermedia() à besoin qu'on lui passe un objet MediaStreamConstraints en tant qu'argument pour qu'elle fonctionne. Cet argument, active le type de communication voulu (vidéo, audio ou les deux) et contient les contraintes ainsi que les paramètres du flux de média qui sera généré par la suite. La suppression de bruit, l'annulation de l'écho, le ratio de la vidéo, le volume du son, la luminosité de l'image, etc. sont quelques exemples de ces paramètres en question[2].

```
// Caméra avec une résolution aussi proche que possible de 640x480
{
  'video': {
    'width': 640,
    'height': 480
    }
}
```

Le dictionnaire MediaStreamConstraints est utilisé lors de l'appel à getUsermedia pour spécifier les types de pistes à inclure dans le MediaStream renvoyé et, éventuellement, pour établir des contraintes pour les paramètres de ces pistes[**?**].
L'implémentation minimale requise par l'interface MediaStreamConstraints est la suivante : { video: true, audio: true }. Cette dernière, indique que nous voulons que le flux généré comporte une voix et une vidéo en utilisant les configurations de contraintes et paramètres par défaut.
Capturer les médias à partir des périphériques médias, dans le navigateur, en utilisant ces configurations de contraintes et de paramètres consiste à écrire les lignes de code suivantes.

```
const constraints: MediaStreamConstraint = { audio: true, video: true };
navigator.mediaDevices.getUserMedia(constraints);
```

1.4.3 Échange des flux multimédia

Une fois qu'un objet RTCPeerConnection est connecté à un peer distant, il est possible de diffuser de l'audio et de la vidéo entre eux. C'est le point où nous connectons le flux que nous recevons de getUserMedia() à RTCPeerConnection.

Ajout des pistes locales

Un flux multimédia se compose d'au moins une piste multimédia, et celles-ci sont ajoutées individuellement à la RTCPeerConnection lorsque nous voulons transmettre le média à l'homologue distant[**?**].

```
const localStream = await navigator.mediaDevices.getUserMedia({video: true, audio: true});
const peerConnection = new RTCPeerConnection(iceConfig);
localStream.getTracks().forEach(track => {
  peerConnection.addTrack(track, localStream);
});
```

Les pistes peuvent être ajoutées à un RTCPeerConnection avant qu'il ne soit connecté à un pair distant, il est donc logique d'effectuer cette configuration le plus tôt possible au lieu d'attendre que la connexion soit terminée[**?**].

2. Une liste complète des contraintes et paramètres de flux de média peut être consultée à l'addresse url : https ://developer.mozilla.org/en-US/docs/Web/API/MediaTrackConstraints

Ajout des pistes distantes

Pour recevoir les pistes distantes qui ont été ajoutées par l'autre pair, nous enregistrons un écouteur sur l'objet RTCPeerConnection local écoutant l'événement 'track'. Puisque la lecture est effectuée sur un objet MediaStream, nous créons d'abord une instance vide que nous remplissons ensuite avec les pistes du pair distant au fur et à mesure que nous les recevons[**?**].

```
const remoteStream = new MediaStream();
const remoteVideo = document.querySelector('#remoteVideo');
remoteVideo.srcObject = remoteStream;
peerConnection.addEventListener('track', async (event) => {
  remoteStream.addTrack(event.track, remoteStream);
});
```

1.5 RTCDatachannel

1.5.1 Introduction

La norme WebRTC couvre également une API pour l'envoi de données arbitraires sur une RTCPeerConnection. Les canaux de données prennent en charge les connexions à volume élevé et à faible latence; un canal de données est un canal non multimédia qui prend uniquement en charge le transfert de données. Il contourne les serveurs et fournit aux développeurs Web des canaux configurables pour transférer des données[**?**].
DataChannel permet l'échange bidirectionnel de données d'application arbitraires entre homologues — pensez à WebSocket, mais d'égal à égal, et avec des propriétés de livraison personnalisables du transport sous-jacent. Une fois la RTCPeerConnection établit, les pairs connectés peuvent ouvrir un ou plusieurs canaux pour échanger du texte ou des données binaires[**?** , p. 348].
Cela se fait en appelant createDataChannel() sur un objet RTCPeerConnection , qui renvoie un objet RTCDataChannel.

```
const peerConnection = new RTCPeerConnection(configuration);
const dataChannel = peerConnection.createDataChannel();
```

Le pair distant peut recevoir des canaux de données en écoutant l'événement datachannel sur l'objet RTCPeerConnection . L'événement reçu est du type RTCDataChannelEvent et contient une propriété 'channel' qui représente le RTCDataChannel connecté entre les pairs[**?**].

```
const peerConnection = new RTCPeerConnection(configuration);
peerConnection.addEventListener('datachannel', event => {
  const dataChannel = event.channel;
});
```

1.5.2 Ouvrir et fermer des événements

Avant qu'un canal de données puisse être utilisé pour envoyer des données, le client doit attendre qu'il soit ouvert. Cela se fait en écoutant l'événement 'open'. De même, il y a un événement 'close' lorsque l'un des côtés ferme le canal[**?**].

```
// récupérer l'élément zone de texte
const messageBox = document.querySelector('#messageBox');
// Récupérer l'élément boutton
```

```
const sendButton = document.querySelector('#sendButton');

// Initier un connection puis un canal de données
const peerConnection = new RTCPeerConnection(configuration);
const dataChannel = peerConnection.createDataChannel();

// Activer la zone de texte et le bouton à l'ouverture
dataChannel.addEventListener('open', event => {
  messageBox.focus();
  messageBox.disabled = false;
  sendButton.disabled = false;
});

// Désactiver l'entrée lorsqu'il est fermé
dataChannel.addEventListener('close', event => {
  messageBox.disabled = false;
  sendButton.disabled = false;
});
```

1.5.3 messages

L'envoi d'un message sur un RTCDataChannel se fait en appelant la fonction send() avec les données que nous voulons envoyer. Le paramètre de data pour cette fonction peut être une chaîne de caractères, un Blob , un ArrayBuffer ou/et ArrayBufferView[**?**].

```
const messageBox = document.querySelector('#messageBox');
const sendButton = document.querySelector('#sendButton');

// Envoyer un simple message texte lorsque nous cliquons sur le bouton
sendButton.addEventListener('click', event => {
  const message = messageBox.textContent;
  dataChannel.send(message);
});
```

Le pair distant recevra les messages envoyés sur un RTCDataChannel en écoutant l'événement 'message'.

```
const incomingMessages = document.querySelector('#incomingMessages');

const peerConnection = new RTCPeerConnection(configuration);
const dataChannel = peerConnection.createDataChannel();

// Ajouter de nouveaux messages à la boîte des messages entrants
dataChannel.addEventListener('message', event => {
  const message = event.data;
  incomingMessages.textContent += message + '\n';
});
```

1.5.4 Configuration de l'ordre et de la fiabilité des messages

RTCDataChannel peut fonctionner en mode non fiable et non ordonné (analogue au User Datagram Protocol ou UDP), en mode fiable et ordonné (analogue au Transmission Control Protocol ou TCP) et en modes partiellement fiables[**?**] :

— Un mode fiable et ordonné garantit la transmission des messages ainsi que l'ordre dans lequel ils sont délivrés . Cela prend une surcharge supplémentaire, ce qui rend potentiellement ce mode plus lent.

— Un mode peu fiable et non ordonné ne garantit pas que chaque message arrive de l'autre côté ni dans quel ordre ils y arrivent . Cela supprime la surcharge, permettant à ce mode de fonctionner beaucoup plus rapidement.

— Le mode de fiabilité partielle garantit la transmission du message dans une condition spécifique, telle qu'un délai de retransmission ou un nombre maximal de retransmissions . L'ordre des messages est également configurable.

Les performances des deux premiers modes sont à peu près les mêmes lorsqu'il n'y a pas de pertes de paquets. Cependant, en mode fiable et ordonné, un paquet perdu provoque le blocage d'autres paquets derrière lui, et le paquet perdu peut être périmé au moment où il est retransmis et arrive. Il est bien sûr possible d'utiliser plusieurs canaux de données au sein de la même application, chacun avec sa propre sémantique fiable ou non[**?**].

	Ordonné	Fiable	Politique de fiabilité partielle
Ordonné + fiable	oui	oui	n/a
Désordonné + fiable	non	partiel	n/a
Ordonné + partiellement fiable (retransmission)	oui	partiel	nombre de retransmissions
Non ordonné + partiellement fiable (retransmission)	non	partiel	nombre de retransmissions
Commandé + partiellement fiable (chronométré)	oui	partiel	délai d'attente (ms)
Non ordonné + partiellement fiable (chronométré)	non	partiel	délai d'attente (ms)

TABLEAU 1.1 – Fiabilité de DataChannel et configurations de livraison

Lors de la configuration d'un canal partiellement fiable, il est important de garder à l'esprit que les deux stratégies de retransmission sont mutuellement exclusives. L'application peut spécifier soit un délai d'attente, soit un nombre de retransmissions, mais pas les deux; cela générera une erreur. Sur ce, jetons un coup d'œil à l'API JavaScript pour configurer le canal[**?** , , p. 353] :

```
/* Options de configuration de l'ordre et de la fiabilité des messages
 * conf = {};
 * conf = { ordered: false };
 * conf = { ordered: true, maxRetransmits: customNum };
 * conf = { ordered: false, maxRetransmits: customNum };
 * conf = { ordered: true, maxRetransmitTime: customMs };
 * conf = { ordered: false, maxRetransmitTime: customMs };
*/

const conf = { ordered: false, maxRetransmits: 0 };

// ...

const peerConnection = new RTCPeerConnection(iceConfig);
const dataCahnnel = pc.createDataChannel('nomDuCanal', conf);
```

Chaque DataChannel peut être configuré avec des paramètres d'ordre et de fiabilité personnalisés, et les pairs peuvent ouvrir plusieurs canaux, qui seront tous multiplexés sur la même association SCTP. En conséquence, chaque canal est indépendant des autres, et les pairs peuvent utiliser différents canaux pour différents types de données, par exemple, une livraison fiable et dans l'ordre pour le chat peer-to-peer et une livraison partiellement fiable et dans le désordre pour les mises à jour d'applications transitoires ou de faible priorité[**?** , , p. 353].

1.5.5 Comparaison avec WebSocket

L'API DataChannel reflète intentionnellement celle de WebSocket : chaque canal établi déclenche les mêmes rappels 'onerror', 'onclose', 'onopen' et 'onmessage'[**?** , p. 348].
Cependant, RTCDataChannel adopte une approche différente[**?**] :

— Il fonctionne avec l'API RTCPeerConnection, qui permet la connectivité peer-to-peer. Cela peut entraîner une latence plus faible : pas de serveur intermédiaire et moins de « sauts ».

— Il utilise le protocole de transmission de contrôle de flux (SCTP), permettant une sémantique de livraison configurable - livraison hors-service et configuration de retransmission.

Étant donné que DataChannel est peer-to-peer et s'exécute sur un protocole de transport plus flexible, il offre également un certain nombre de fonctionnalités supplémentaires non disponibles pour WebSocket. l'ensemble de codes précédents met en évidence, certaines des différences les plus importantes [**?** , p. 348] :

— Contrairement au constructeur WebSocket, qui attend l'URL du serveur WebSocket, DataChannel est une méthode « factory » sur de l'objet RTCPeerConnection.

— Contrairement à WebSocket, chaque pair peut initier une nouvelle session DataChannel : le rappel 'onDataChannel' est déclenché lorsqu'une nouvelle session DataChannel est établie.

— Contrairement à WebSocket, qui s'exécute sur un transport TCP fiable et ordonné, chaque DataChannel peut être configuré avec une livraison personnalisée et une sémantique de fiabilité.