INE3: Communication Multimédia P2P en temps réel sur le web

TP3 : La plateforme WEBRTC

Description

L'objectif de ce TP est de développer une application P2P en se basant sur les technologies WEBRTC et HTML5 selon les principes présentés dans le cadre du cours. L'application doit permettre à deux utilisateurs, devant leurs navigateurs, de télécharger une page web et d'ouvrir une session P2P pour échanger des flux media (audio et vidéo) et de données (généralement de texte). Les flux media (audio et vidéo) sont capturés à partir des périphériques multimédia de chaque utilisateur. Le flux de données, quant à lui, est saisi à travers le clavier moyennant un forum de discussion.

Les utilisateurs de l'application peuvent avoir des machines avec des caractéristiques différentes (audio, video, résolution, bande passante …) et peuvent être derrière des NATs et des Pare-feux. Pour gérer cette hétérogénéité d'environnements des utilisateurs, un canal de signalisation s'avère nécessaire pour, d'une part, négocier des paramètres communs de session et pour, d'autre part, découvrir chaque pair et franchir les NATs et les Pare-feux. L'implémentation de ce service de signalisation repose sur un ensemble de technologies et de serveurs, dont on peut citer entre autres XHR, Websocket, NodeJS, ICE, STUN et TURN.

Environnement de travail

Pour développer cette application, vous devez vous disposer de:

- Un navigateur Chrome/Firefox/Opera/Safari/IE/… dans une de ses récentes versions.
- Un environnement XHR/Websocket/NodeJS/… installé.
- Une installation des modules NodeJS pour les besoins de l'application.
- Une camera et un microphone intégrés/attachés à votre machine.
- Une connexion Internet avec un très bon débit.
- …

Implémentation de l'application

Pour implémenter l'application, nous allons procéder étape par étape en distinguant quatre aspects, à savoir : la capture de flux media à partir des périphériques multimédia de l'utilisateur, l'établissement d'un canal de signalisation entre les deux utilisateurs, l'établissement d'une session RTCPeerConnection pour échanger des flux audio et video des deux utilisateurs, l'établissement d'une session RTCDataChannel pour échanger des données arbitraires entre les deux utilisateurs, la configuration des serveurs ICE, STUN et TURN pour la traversée des NATs et des pare-feux en cas de leurs existences. Chacun de ces aspects sera traité

séparément dans ce qui suit pour enfin aboutir à l'intégration de tout l'ensemble dans une application riche, portable et accessible à tout internaute!

Capture des flux audio et vidéo du pair local :

Pour capturer les flux audio et vidéo d'un utilisateur, on doit utiliser l'API MediaStream du WEBRTC. La visualisation des flux obtenus est faite en utilisant HTML5 et JavaScript. Le code source de l'application se présente comme suit :

Le fichier HTML 5 :

```
<!DOCTYPE html>
<html>
<head>
      <title>getUserMedia very simple demo</title>
</head>
<body>
      <div id="mainDiv">
      <h1>Exemple de <code> getUserMedia()</code> </h1>
      <p> Il s'agit d'appeler <code>getUserMedia()</code> et d'afficher le
      resultat en utilisant la balise HTML5 <video> </p>
      <p> Examiner les codes sources HTML et JavaScript ...</p>
      <video autoplay></video>
      <script src="js/getUserMedia.js"></script>
      </div>
</body>
</html>
```

Le fichier JavaScript :

```
navigator.getUserMedia = navigator.getUserMedia
                        || navigator.webkitGetUserMedia
                        || navigator.mozGetUserMedia;

var constraints = {audio: false, video: true};
var video = document.querySelector("video");
function successCallback(stream) {
      window.stream = stream;
      if (window.URL) {
         video.src = window.URL.createObjectURL(stream);
      } else {
         video.src = stream;
      }
      video.play();
      }
      function errorCallback(error){
            console.log("navigator.getUserMedia error: ", error);
```

```
}

navigator.getUserMedia(constraints, successCallback, errorCallback);
```

1) Saisir le code de l'application, le structurer et l'exécuter à partir de votre navigateur.
2) Modifier le code source de l'application en ajoutant des contraintes de résolution sur les medias capturés.

Échange des données entre les pairs en utilisant l'API RTCDataChannel :

L'API RTCDataChannel permet d'échanger des données arbitraires (texte, binaire …) entre deux pairs et ce en se basant sur objet RTCPeerConnection. Dans ce qui suit, on implémente une application pour échanger des données en utilisant RTCDataChannel et RTCPeerConnection :

Le fichier HTML 5 :

```
<!DOCTYPE html>
<html>
<head>
      <title>DataChannel example</title>
</head>
<body>
      <textarea rows="5" cols="50" id="dataChannelSend" disabled
      placeholder="
      1: Press Start; 2: Enter text; 3: Press Send."></textarea>
      <textarea rows="5" cols="50" id="dataChannelReceive"
      disabled></textarea>
      <div id="buttons">
      <button id="startButton">Start</button>
      <button id="sendButton">Send</button>
      <button id="closeButton">Stop</button>
      </div>
      <script src="js/dataChannel.js"></script>
</body>
</html>
```

Le client JavaScript :

```
var sendChannel, receiveChannel;

var startButton = document.getElementById("startButton");
var sendButton = document.getElementById("sendButton");
var closeButton = document.getElementById("closeButton");

startButton.disabled = false;
sendButton.disabled = true;
closeButton.disabled = true;

startButton.onclick = createConnection;
```

```
sendButton.onclick = sendData;
closeButton.onclick = closeDataChannels;

function log(text) {
      console.log("At time: " + (performance.now() / 1000).toFixed(3) +
      " --> " + text);
}
function createConnection() {
// Chrome
if (navigator.webkitGetUserMedia) {
      RTCPeerConnection = webkitRTCPeerConnection;
// Firefox
} else if(navigator.mozGetUserMedia){
      RTCPeerConnection = mozRTCPeerConnection;
      RTCSessionDescription = mozRTCSessionDescription;
      RTCIceCandidate = mozRTCIceCandidate;
}
log("RTCPeerConnection object: " + RTCPeerConnection);

var servers = null;
var pc_constraints = {
'optional': [
{'DtlsSrtpKeyAgreement': true}
]};
localPeerConnection = new RTCPeerConnection(servers,pc_constraints);
log("Created local peer connection object, with Data Channel");
try {
      sendChannel = localPeerConnection.createDataChannel(
      "sendDataChannel",{reliable: true});
      log('Created reliable send data channel');
} catch (e) {
alert('Failed to create data channel!');
log('createDataChannel() failed with following message: '
+ e.message);
}

localPeerConnection.onicecandidate = gotLocalCandidate;
sendChannel.onopen = handleSendChannelStateChange;
sendChannel.onclose = handleSendChannelStateChange;

window.remotePeerConnection = new RTCPeerConnection(servers,
      pc_constraints);
log('Created remote peer connection object, with DataChannel');
remotePeerConnection.onicecandidate = gotRemoteIceCandidate;
remotePeerConnection.ondatachannel = gotReceiveChannel;
localPeerConnection.createOffer(gotLocalDescription,onSignalingError);
startButton.disabled = true;
closeButton.disabled = false;
}
```

```
function onSignalingError(error) {
      console.log('Failed to create signaling message : ' + error.name);
}

function sendData() {
      var data = document.getElementById("dataChannelSend").value;
      sendChannel.send(data);
      log('Sent data: ' + data);
}

function closeDataChannels() {
      log('Closing data channels');
      sendChannel.close();
      log('Closed data channel with label: ' + sendChannel.label);
      receiveChannel.close();
      log('Closed data channel with label: ' + receiveChannel.label);
      localPeerConnection.close();
      remotePeerConnection.close();
      localPeerConnection = null;
      remotePeerConnection = null;
      log('Closed peer connections');
      startButton.disabled = false;
      sendButton.disabled = true;
      closeButton.disabled = true;
      dataChannelSend.value = "";
      dataChannelReceive.value = "";
      dataChannelSend.disabled = true;
      dataChannelSend.placeholder = "1: Press Start; 2: Enter text; \
      3: Press Send.";
}
function gotLocalDescription(desc) {
      localPeerConnection.setLocalDescription(desc);
      log('localPeerConnection\'s SDP: \n' + desc.sdp);
      remotePeerConnection.setRemoteDescription(desc);
      remotePeerConnection.createAnswer(gotRemoteDescription,onSignalingEr
      ror);
}

function gotRemoteDescription(desc) {
      remotePeerConnection.setLocalDescription(desc);
      log('Answer from remotePeerConnection\'s SDP: \n' + desc.sdp);
      localPeerConnection.setRemoteDescription(desc);
}

function gotLocalCandidate(event) {
      log('local ice callback');
      if (event.candidate) {
            remotePeerConnection.addIceCandidate(event.candidate);
            log('Local ICE candidate: \n' + event.candidate.candidate);
      }
```

```
}

function gotRemoteIceCandidate(event) {
      log('remote ice callback');
      if (event.candidate) {
            localPeerConnection.addIceCandidate(event.candidate);
            log('Remote ICE candidate: \n ' + event.candidate.candidate);
      }
}

function gotReceiveChannel(event) {
      log('Receive Channel Callback: event --> ' + event);
      receiveChannel = event.channel;
      receiveChannel.onopen = handleReceiveChannelStateChange;
      receiveChannel.onmessage = handleMessage;
      receiveChannel.onclose = handleReceiveChannelStateChange;
}

function handleMessage(event) {
      log('Received message: ' + event.data);
      document.getElementById("dataChannelReceive").value = event.data;
      document.getElementById("dataChannelSend").value = '';
}

function handleSendChannelStateChange() {
      var readyState = sendChannel.readyState;
      log('Send channel state is: ' + readyState);
      if (readyState == "open") {
            dataChannelSend.disabled = false;
            dataChannelSend.focus();
            dataChannelSend.placeholder = "";
            sendButton.disabled = false;
            closeButton.disabled = false;
      } else {
            dataChannelSend.disabled = true;
            sendButton.disabled = true;
            closeButton.disabled = true;
      }
}

function handleReceiveChannelStateChange() {
      var readyState = receiveChannel.readyState;
      log('Receive channel state is: ' + readyState);
}
```

1) Saisir le code de l'application, le structurer et l'exécuter à partir de votre navigateur.
2) Modifier le code source de l'application pour remplacer l'API RTCDataChannel par Websocket et comparer ensuite les deux implémentations.

Établissement d'un canal de signalisation entre les deux pairs :

La signalisation est un processus qui permet de découvrir les pairs et de négocier les paramètres communs de session à établir entre eux (voir les notes du cours). Dans ce qui suit, on implémente un canal de signalisation en utilisant NodeJS/Socket.io :

Le fichier HTML 5 :

```
<!DOCTYPE html>
<html>
<head>
<title> WebRTC client</title>
</head>
<body>
<script src='/socket.io/socket.io.js'></script>
<div id="scratchPad"></div>
<script type="text/javascript" src="js/simpleNodeClient.js"></script>
</body>
</html>
```

Le client JavaScript :

```
div = document.getElementById('scratchPad');
var socket = io.connect('http://localhost:8181');
channel = prompt("Enter signaling channel name:");
if (channel !== "") {
   console.log('Trying to create or join channel: ', channel);
   socket.emit('create or join', channel);
}
socket.on('created', function (channel){
console.log('channel ' + channel + ' has been created!');
console.log('This peer is the initiator...');
div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
(performance.now() / 1000).toFixed(3) + ' --> Channel '
+ channel + ' has been created! </p>');
div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
(performance.now() / 1000).toFixed(3) +
' --> This peer is the initiator...</p>');
});

socket.on('full', function (channel){
console.log('channel ' + channel + ' is too crowded! \
Cannot allow you to enter, sorry :-(');
div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
(performance.now() / 1000).toFixed(3) + ' --> \
channel ' + channel + ' is too crowded! \
Cannot allow you to enter, sorry :-( </p>');
});

socket.on('remotePeerJoining', function (channel){
```

```
console.log('Request to join ' + channel);
console.log('You are the initiator!');
div.insertAdjacentHTML( 'beforeEnd', '<p style="color:red">Time: ' +
(performance.now() / 1000).toFixed(3) +
' --> Message from server: request to join channel ' +
channel + '</p>');
});

socket.on('joined', function (msg){
console.log('Message from server: ' + msg);
div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
(performance.now() / 1000).toFixed(3) +
' --> Message from server: </p>');
div.insertAdjacentHTML( 'beforeEnd', '<p style="color:blue">' +
msg + '</p>');
div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
(performance.now() / 1000).toFixed(3) +
' --> Message from server: </p>');
div.insertAdjacentHTML( 'beforeEnd', '<p style="color:blue">' +
msg + '</p>');
});

socket.on('broadcast: joined', function (msg){
div.insertAdjacentHTML( 'beforeEnd', '<p style="color:red">Time: ' +
(performance.now() / 1000).toFixed(3) +
' --> Broadcast message from server: </p>');
div.insertAdjacentHTML( 'beforeEnd', '<p style="color:red">' +
msg + '</p>');
console.log('Broadcast message from server: ' + msg);

var myMessage = prompt('Insert message to be sent to your peer:', "");
socket.emit('message', {
channel: channel,
message: myMessage});
});

socket.on('log', function (array){
console.log.apply(console, array);
});

socket.on('message', function (message){
console.log('Got message from other peer: ' + message);
div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
(performance.now() / 1000).toFixed(3) +
' --> Got message from other peer: </p>');
div.insertAdjacentHTML( 'beforeEnd', '<p style="color:blue">' +
message + '</p>');

var myResponse = prompt('Send response to other peer:', "");
socket.emit('response', {
```

```
    channel: channel,
    message: myResponse});
    });

    socket.on('response', function (response){
    console.log('Got response from other peer: ' + response);
    div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
    (performance.now() / 1000).toFixed(3) + ' --> Got response from other
    peer: </p>');
    div.insertAdjacentHTML( 'beforeEnd', '<p style="color:blue">' +
    response + '</p>');

    var chatMessage = prompt('Keep on chatting.Write "Bye" to quit
    conversation', "");

    if(chatMessage == "Bye"){
    div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' (performance.now() /
    1000).toFixed(3) + ' --> Sending "Bye" to server...</p>');
    console.log('Sending "Bye" to server');
    socket.emit('Bye', channel);
    div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
    (performance.now() / 1000).toFixed(3) +' --> Going to
    disconnect...</p>');
    console.log('Going to disconnect...');

    socket.disconnect();
    }else{
    socket.emit('response', {
    channel: channel,
    message: chatMessage});
    }
    });

    socket.on('Bye', function (){
    console.log('Got "Bye" from other peer! Going to disconnect...');
    div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
    (performance.now() / 1000).toFixed(3) +
    ' --> Got "Bye" from other peer!</p>');
    div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +
    (performance.now() / 1000).toFixed(3) +
    ' --> Sending "Ack" to server</p>');

    console.log('Sending "Ack" to server');
    socket.emit('Ack');

    div.insertAdjacentHTML( 'beforeEnd', '<p>Time: ' +(performance.now() /
    1000).toFixed(3) + ' --> Going to disconnect...</p>');
    console.log('Going to disconnect...');
    socket.disconnect();
});
```

Le serveur JavaScript :

```
var static = require('node-static');
var http = require('http');

var file = new(static.Server)();
var app = http.createServer(function (req, res) {
file.serve(req, res);
}).listen(8181);

var io = require('socket.io').listen(app);

io.sockets.on('connection', function (socket){

      socket.on('message', function (message) {
            log('S --> Got message: ', message);
            socket.broadcast.to(message.channel).emit('message', \
            message.message);
      });

      socket.on('create or join', function (channel) {
            var numClients = io.sockets.clients(channel).length;
            console.log('numclients = ' + numClients);

            if (numClients == 0){
                  socket.join(channel);
                  socket.emit('created', channel);
            } else if (numClients == 1) {
                  io.sockets.in(channel).emit('remotePeerJoining',
                  channel);
                  socket.join(channel);
                  socket.broadcast.to(channel).emit('broadcast: joined', 'S
                  --> broadcast(): client ' + socket.id + ' joined channel
                  ' + channel);
                  } else { console.log("Channel full!");
                           socket.emit('full', channel);
                  }
            });

      socket.on('response', function (response) {
            log('S --> Got response: ', response);
            socket.broadcast.to(response.channel).emit('response',
            response.message);
      });

socket.on('Bye', function(channel){
      socket.broadcast.to(channel).emit('Bye');
      socket.disconnect();
});
```

```
    socket.on('Ack', function () {
          console.log('Got an Ack!');
          socket.disconnect();
    });

    function log(){
          var array = [">>> "];
          for (var i = 0; i < arguments.length; i++) {
                array.push(arguments[i]);
          }
          socket.emit('log', array);
    }
});
```

1) Saisir le code de l'application, le structurer et l'exécuter à partir de votre navigateur.
2) Modifier le code source de l'application en essayant les autres technologies pour fournir le service de signalisation aux deux pairs communicants.

Extensions à faire et à remettre

Pour maitriser le WEBRTC et les technologies associées, je vous demande de faire les extensions suivantes :

1) Compléter le code de l'application en intégrant les différentes parties discutées dans ce TP.
2) Remplacer *Socket.io* par les autres technologies telles que XHR, Websocket, NodeJS natif, …
3) Passer à une implémentation générique de l'application en permettant à l'utilisateur de spécifier/configurer les technologies souhaitées au moment de l'usage.
4) Changer le mode P2P en mode vidéoconférence/multipartites.