

P3: INTERACTIVIDAD AVANZADA

How to cook Chili

21753 - Gestión y Distribución de la Información Empresarial

Curso 2020/21

Integrantes:

Sergio Garcia Puertas - [REDACTED]
[REDACTED] - [REDACTED]

Índice

Índice	2
Funcionalidades de interfaz avanzada	3
Servicios de interfaz avanzada	6

Funcionalidades de interfaz avanzada

Al entrar en la página, nos pide que entremos un nombre de sala. Si la sala no existe la creará y seremos nosotros el host. Por el contrario, si la sala ya existe, nos uniremos a ella como invitados. Antes de entrar a la sala debemos introducir el nombre de usuario que aparecerá en la página.

alumnes-ltim.uib.es dice

Enter room name:

Aceptar Cancelar

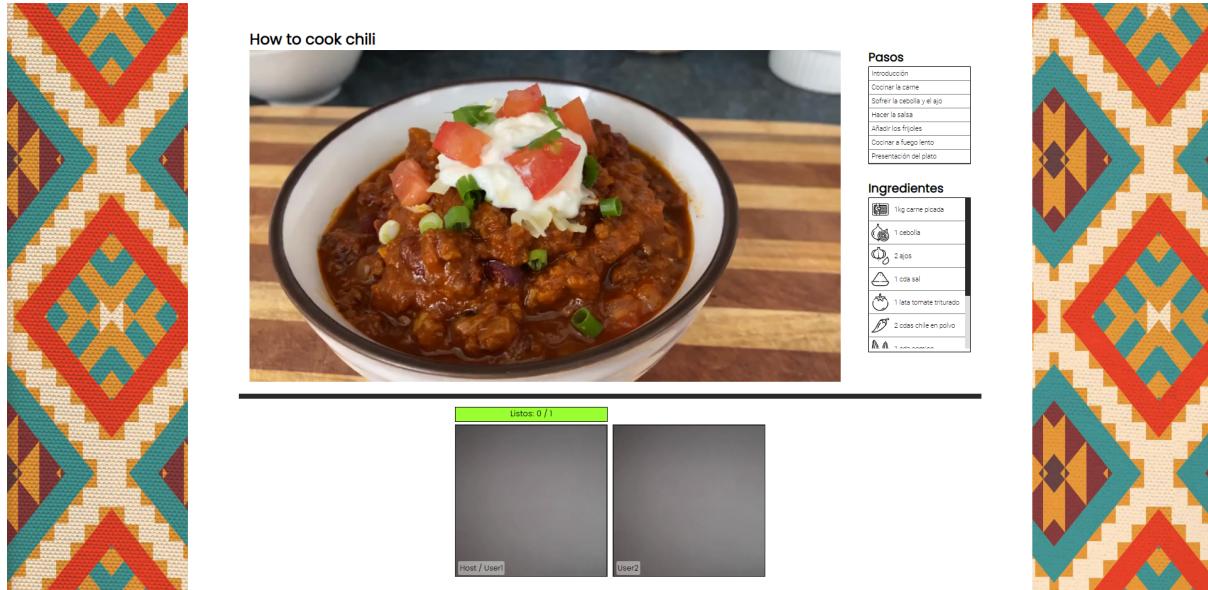
alumnes-ltim.uib.es dice

Enter username:

Aceptar Cancelar

En función de si somos el host o un invitado, tendremos vistas diferentes. En la imágenes a continuación se muestra la vista del host:

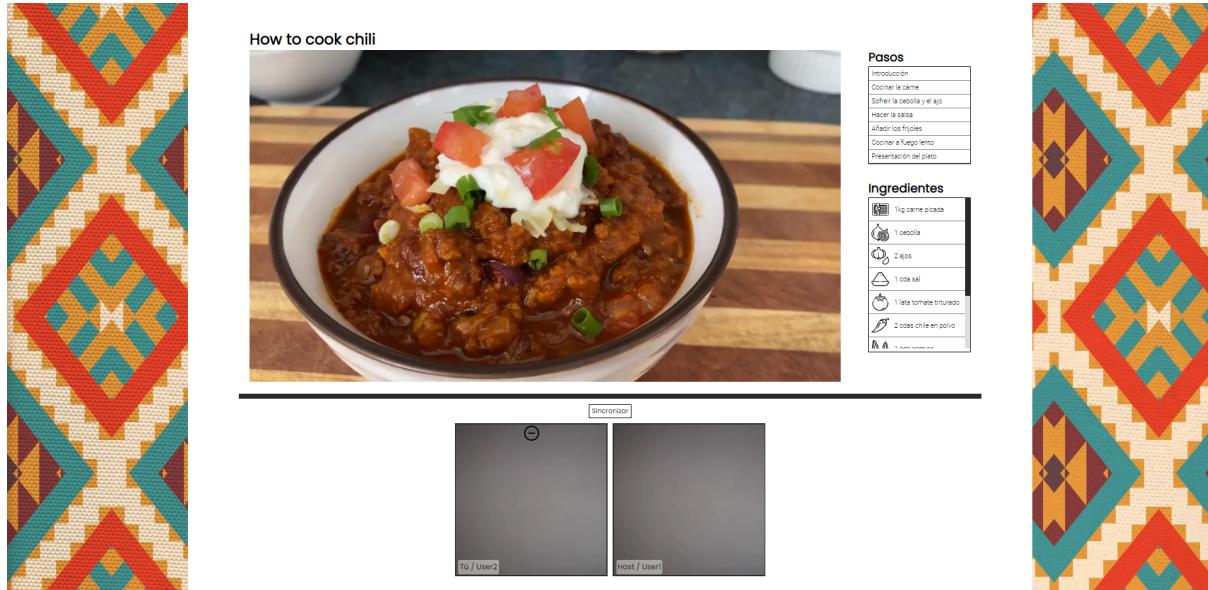
La parte superior de la página es igual a la de la práctica 2 e incluye todas sus funcionalidades menos una. Hemos eliminado el botón que había encima del reproductor para alternar entre DASH y HLS. Ahora se reproduce directamente con DASH.



En la siguiente imagen se puede ver los nuevos elementos que aparecen en la vista del host. Estos son el video de su cámara y las de todos los invitados así como la cantidad de invitados que están "listos". Esta funcionalidad se explica en la parte de los invitados.

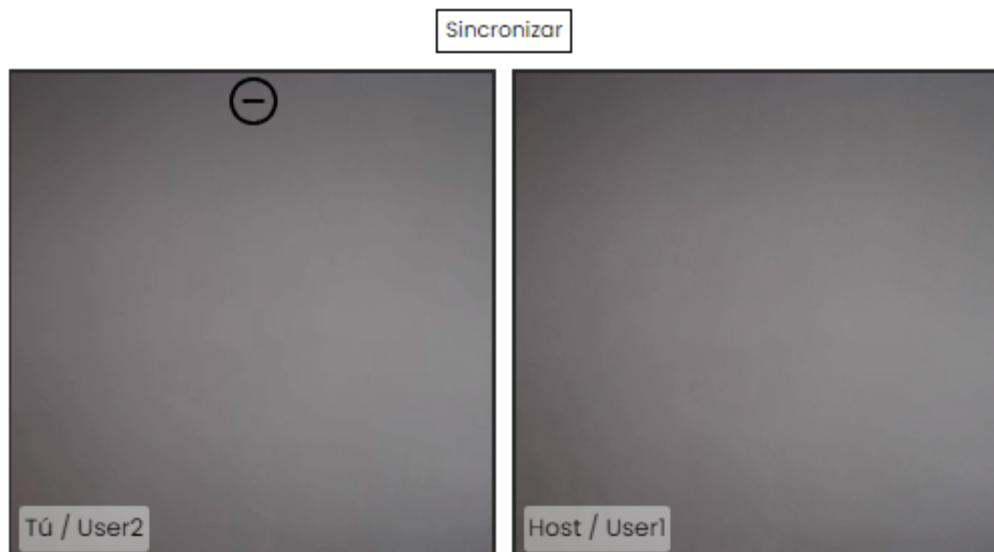


La siguiente imagen es la vista de la página cuando se accede como invitado. La parte superior, al igual que para el host, es igual a la de la práctica 2.

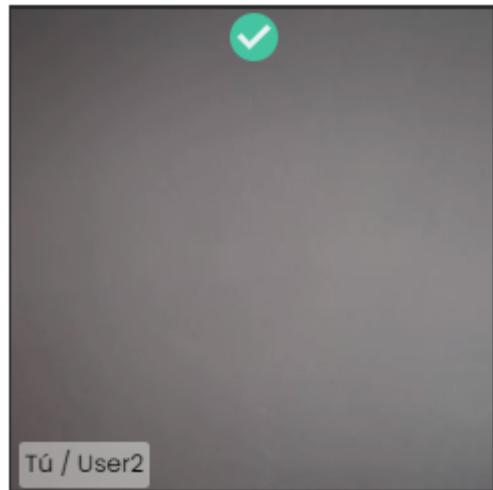


La parte inferior vemos que es distinta a la del host. Los invitados también ven el video de su cámara, así como la del host y la del resto de invitados. Encima de las cámaras tienen un botón “sincronizar”, que al clicar en él se sincroniza su video con el del host. Esta función está pensada para que durante una clase de cocina, si un usuario vuelve atrás para mirar algún paso, luego pueda volver al ritmo del host, que es quien da la clase.

Además, el invitado tiene un botón en su cámara que al clicar en él cambia de imagen y le indica al host que está listo. Esto hace que el contador del host (el recuadro verde) varíe en función de los invitados que estén listos o dejen de estarlo.



Así cambia el botón cuando un usuario está listo:



Servicios de interfaz avanzada

Hemos usado el framework **Express.js** para facilitar el uso de Node.js. En concreto lo utilizamos para crear la aplicación, definir qué archivos ha de enviar a los clientes cuando se conectan al servidor y para especificar el puerto al que está escuchando la aplicación.

```
const express = require('express');
const app = express();
const http = require('http');
const server = http.createServer(app);
app.use(express.static('/root/public'));
server.listen(8080);
```

Para las funcionalidades de compartición de video, audio y otros datos usamos **Socket.IO** y **Web RTC**. En la parte del servidor, estas tecnologías se usan en el archivo index.js. En la parte cliente, se usan en el archivo main.js.

En concreto, utilizamos Socket.IO para dos cosas: para crear salas y para que los usuarios envíen información al resto de usuarios de la misma sala.

Cuando un usuario se conecta a la página web e introduce el nombre de la sala y el nombre de usuario, se conecta al servicio de socket.io del servidor.

Este es el trozo de código de index.js (en el servidor) que gestiona las salas.

```
io.on('connection', function(socket) {
  var room_id;
  var listo = false;

  socket.on('create or join', function(room) {
    const clients = io.sockets.adapter.rooms.get(room);
    const clientesSala = clients ? clients.size : 0;
    if (clientesSala == 0) {
      socket.join(room);
```

```

    room_id = room;
    socket.emit('mensaje','Has creado la sala: ' + room);
    socket.emit('created', room, socket.id);
    socket.emit('printID', socket.id);
} else if (clientesSala > 0 && clientesSala <= 8) {
    socket.join(room);
    room_id = room;
    socket.emit('mensaje','Te has unido a la sala: ' + room);
    socket.emit('joined', room, socket.id);
    socket.to(room).emit('nuevoUsuario', "Se ha unido un usuario nuevo");
    socket.emit('printID', socket.id);
} else { // max 8 clientes

    socket.emit('mensaje', 'La sala a la que intentas unirte está completa');

}

});

...
});


```

En la parte cliente tenemos:

```

if (room !== '') {
    socket.emit('create or join', room);
}

socket.on('nuevoUsuario', function(mensaje) {
    console.log(mensaje);
});

socket.on('printID', function(id) {
    console.log('Hola cliente: ' + id);
});

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

socket.on('created', function(room) {
    created();
});

socket.on('joined', function (room){
    joined();
});

socket.on("connect", () => {
    socket.emit('conectado',room);
});

```

Entonces, cuando un cliente se conecta, realiza un `socket.emit('create or join', room);`. Esto lo recibe el servidor y ejecuta su `socket.on('create or join', function(room) {...});`. Esta función mira si la sala existe. Si no existe la crea, une al usuario y hace un `socket.emit('created', room, socket.id);`. Esto lo recibe el cliente y actualiza su variable "isRoomCreator" a true, además de poner su video local mediante la API "getUserMedia". Si por el contrario la sala ya existía, el servidor hace un `socket.emit('joined', room, socket.id);` y el cliente pone su video de manera local y mantiene su variable "isRoomCreator" a false.

También usamos Socket.IO para transmitir información que no es ni video ni audio entre los usuarios como puede ser el tiempo de video del Host, el nombre de usuario, si un usuario está listo o no, etc.

El trozo de código que gestiona esto desde el servidor es el siguiente:

```
socket.on('cliente_listo', (valor, roomId) => {
  socket.broadcast.to(roomId).emit('cambio_num_clientes', valor)
  if (valor == 1) {
    listo = true;
  } else {
    listo = false;
  }
});
socket.on('conectado', (roomId) => {
  socket.broadcast.to(roomId).emit('nuevo_usuario')
});
socket.on('disconnect', () => {
  socket.broadcast.to(room_id).emit('cliente_desconectado', socket.id, listo)
});
socket.on('sincronizar', (idcliente) => {
  socket.broadcast.to(room_id).emit('get_tiempo', idcliente)
});
socket.on('envia_tiempo', (idcliente, tiempo) => {
  socket.to(idcliente).emit('set_tiempo', tiempo)
});
```

El trozo de código que gestiona esto desde el cliente es el siguiente:

```
function usu_listo() { // para alternar entre estado "listo" y "no listo"
  if (listo) {
    socket.emit('cliente_listo', -1, room);
    listo = false;
    document.getElementById("check").src = "iconos/uncheck.svg"
  } else {
    socket.emit('cliente_listo', 1, room);
    listo = true;
    document.getElementById("check").src = "iconos/check.svg"
  }
}

socket.on("connect", () => {
  socket.emit('conectado', room);
```

```

});

socket.on('nuevo_usuario', function() { // para sumar un usuario al total de
conectados
  if (isRoomCreator) {
    totalClientes += 1;
    document.getElementById("numPersonas").innerHTML = clientesListos + " / " +
totalClientes;
  }
}

socket.on('cambio_num_clientes', function(valor) { // para actualizar el num de
usuarios listos
  if (isRoomCreator) {
    clientesListos += valor;
    document.getElementById("numPersonas").innerHTML = clientesListos + " / " +
totalClientes;
  }
}

socket.on('cliente_desconectado', function(id, listo) { // para eliminar los datos de
un usuario que se desconecta
  if (isRoomCreator) {
    if (listo) { // si el usuario estaba listo, hay que eliminar su voto en el
contador de listos
      clientesListos -= 1;
    }
    totalClientes -= 1;
    document.getElementById("numPersonas").innerHTML = clientesListos + " / " +
totalClientes;
  }
  $("#camara"+id).remove();
  $("#contCamara"+id).remove();
}

function sincronizar() {
  socket.emit("sincronizar", socket.id);
}

socket.on("get_tiempo", function(idcliente) {
  // sólo el host de la sala debe contestar la petición
  if (isRoomCreator) {

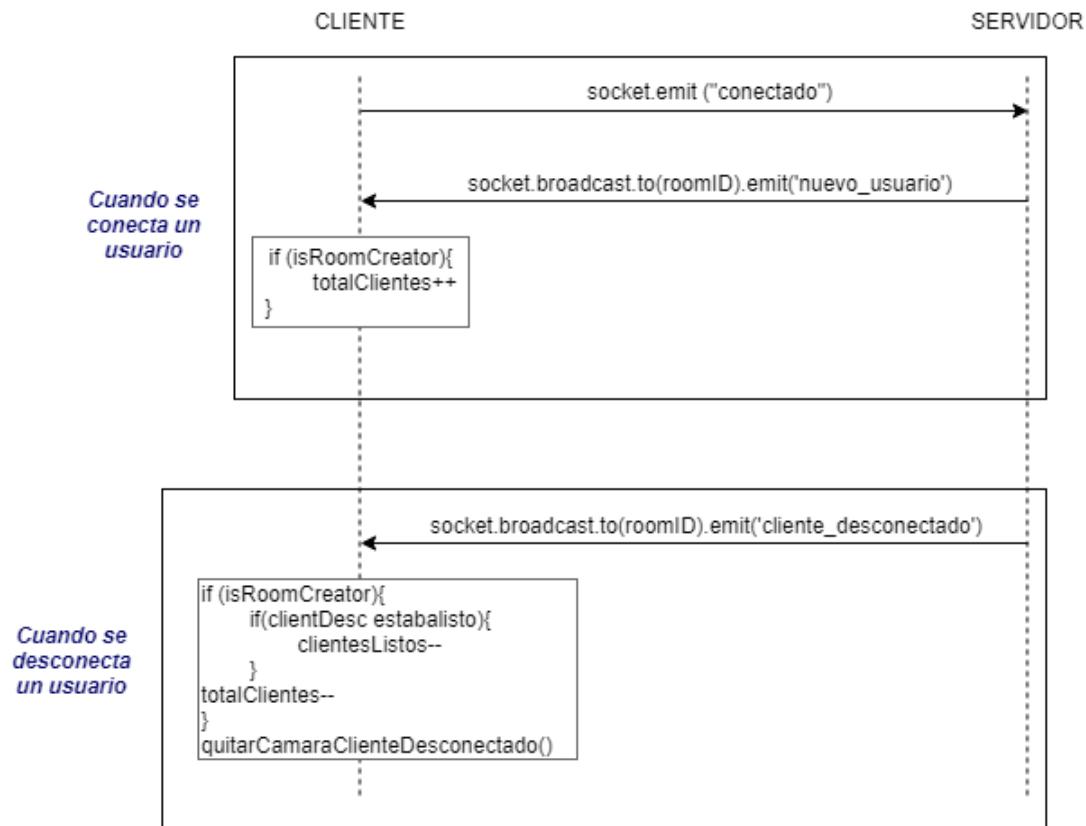
socket.emit("envia_tiempo",idcliente,document.getElementById("myVideo").currentTime)
  }
})

socket.on("set_tiempo", function(tiempo) {
  document.getElementById("myVideo").currentTime = tiempo;
})

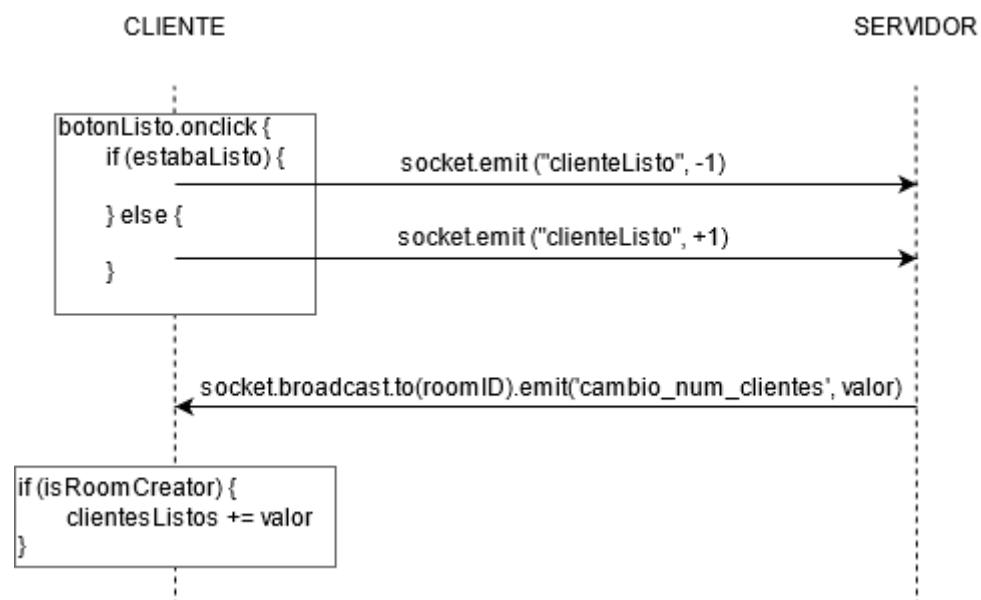
```

Estos trozos de código se utilizan principalmente para:

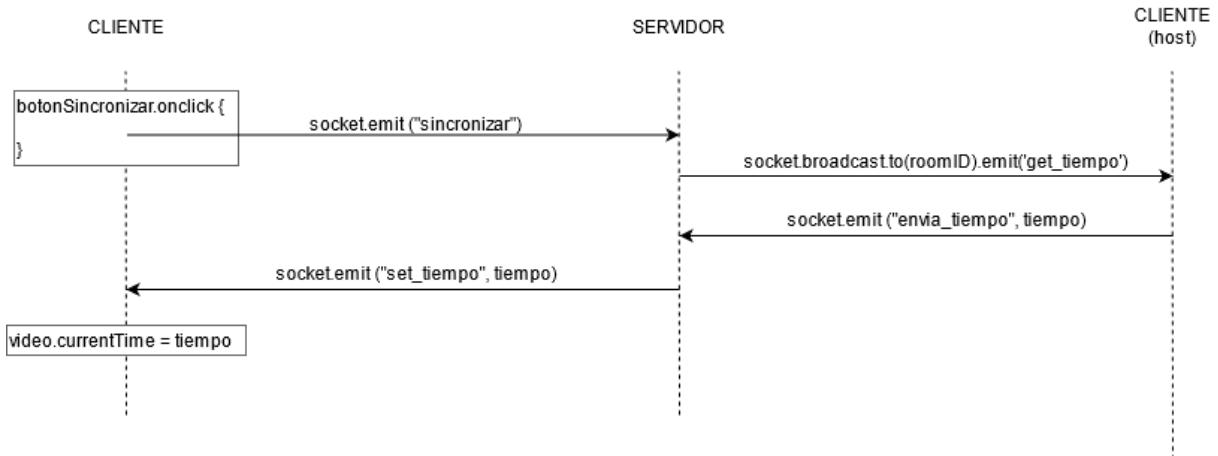
1. Gestionar los clientes conectados: cuando un cliente se conecta/desconecta actualizar los valores que le aparecen al host.



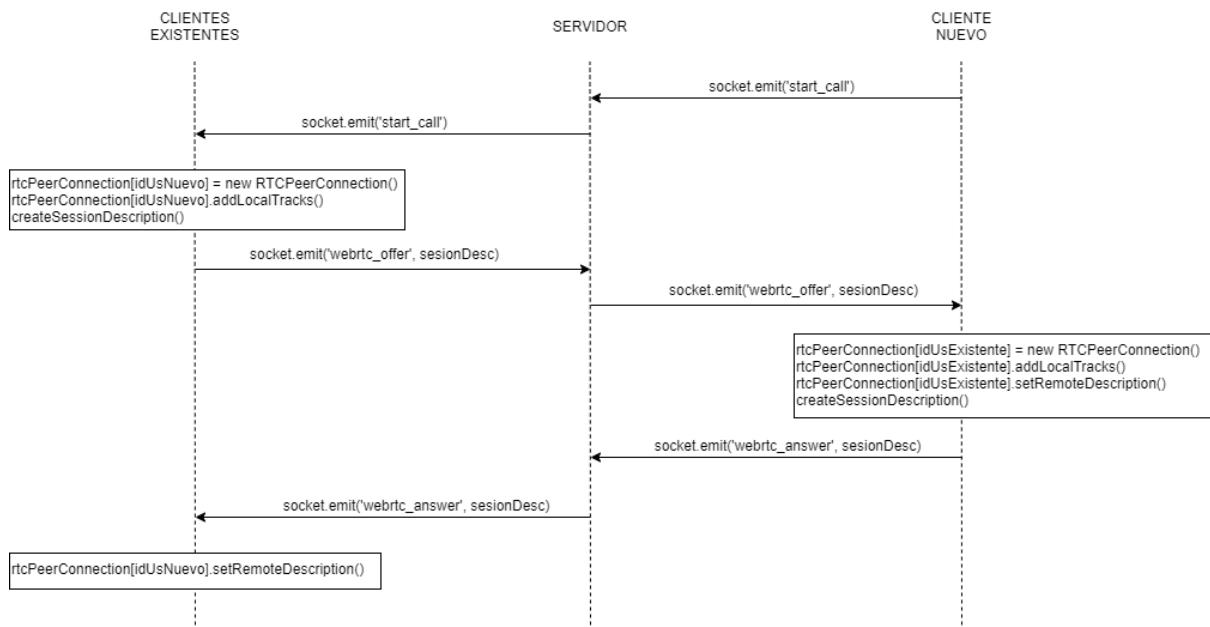
2. Gestionar los clientes listos: cuando un cliente hace click en el botón de "listo/no listo" se actualiza el valor para el host.



3. Gestionar la sincronización de un cliente con el host: cuando un cliente hace click en el botón de “sincronizar” se pone su video local en el mismo tiempo que el host.



Para la gestión del WebRTC se realizan las siguientes comunicaciones entre los clientes que ya estaban, el cliente nuevo y el servidor:



En resumen, el cliente nuevo avisa a todos los existentes de que se ha unido. Entonces, los existentes crean una `RTCPeerConnection` con él, crean una `sessionDescription` y se la envían. El cliente nuevo va recibiendo cada una de las ofertas de los clientes existentes y va creando una `RTCPeerConnection` para cada uno, una `sessionDescription` y les envía la correspondiente respuesta. Durante este proceso cada usuario va añadiendo dinámicamente las cámaras de los demás.

El trozo de código que gestiona esta parte en el servidor es el siguiente:

```

socket.on('start_call', async(roomId, idUsNuevo, nombreUsNuevo) => {
    socket.broadcast.to(roomId).emit('start_call', idUsNuevo, nombreUsNuevo)
});
  
```

```

socket.on('webrtc_offer', async(event, idUsNuevo, idUsExistente, nombreUsExistente)
=> {
  socket.to(idUsNuevo).emit('webrtc_offer', event.sdp, idUsExistente,
nombreUsExistente)
});
socket.on('webrtc_answer', async(event, idUsExistente, idUsNuevo) => {
  socket.to(idUsExistente).emit('webrtc_answer', event.sdp, idUsNuevo)
});
socket.on('webrtc_ice_candidate', async(event, idUsuario, roomId, otroUsuario) => {
  socket.to(idUsuario).emit('webrtc_ice_candidate', event, otroUsuario)
});

```

Y el código en la parte cliente es el siguiente:

```

socket.on('start_call', async (idUsNuevo, nombreUsNuevo) => { // nueva conexión RTC
  rtcPeerConnection[idUsNuevo] = new RTCPeerConnection(iceServers)
  addLocalTracks(rtcPeerConnection[idUsNuevo])
  rtcPeerConnection[idUsNuevo].ontrack =
function(event){setRemoteStream(event,idUsNuevo, nombreUsNuevo)}
  rtcPeerConnection[idUsNuevo].onicecandidate =
function(event){sendIceCandidate(event, idUsNuevo, socket.id)}

  // si es el host de la sala, pasa el nombre junto a su rol
  let nombrePas
  if (isRoomCreator) {
    nombrePas = "Host / " + nombre
  } else {
    nombrePas = nombre
  }

  // envia una oferta al nuevo usuario con su información
  await createOffer(rtcPeerConnection[idUsNuevo], idUsNuevo, socket.id, nombrePas)

})

socket.on('webrtc_offer', async (event, idUsuario, nombreUsExistente) => { // llega
la info de un usuario previamnte conectado a la sala
  rtcPeerConnection[idUsuario] = new RTCPeerConnection(iceServers)
  addLocalTracks(rtcPeerConnection[idUsuario])
  rtcPeerConnection[idUsuario].ontrack =
function(event){setRemoteStream(event,idUsuario, nombreUsExistente)} // añade el
contenido audiovisual
  rtcPeerConnection[idUsuario].onicecandidate =
function(event){sendIceCandidate(event,idUsuario, socket.id)}
  rtcPeerConnection[idUsuario].setRemoteDescription(new RTCSessionDescription(event))
  await createAnswer(rtcPeerConnection[idUsuario], idUsuario, socket.id) // envía su
información al usuario existente

})

socket.on('webrtc_answer', (event, otroUsuario) => {
  rtcPeerConnection[otroUsuario].setRemoteDescription(new
RTCSessionDescription(event))

```

```
)  
  
socket.on('webrtc_ice_candidate', (event, otroUsuario) => {  
    console.log('Socket event callback: webrtc_ice_candidate')  
  
    // ICE candidate configuration.  
    var candidate = new RTCIceCandidate({  
        sdpMLineIndex: event.label,  
        candidate: event.candidate,  
    })  
    rtcPeerConnection[otroUsuario].addIceCandidate(candidate)  
})
```