Aplicación de videollamada P2P

Dylan Rivera Valín
Plan de Estudios del Estudiante

Félix Freitag

06/01/2019

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	Aplicación móvil de videollamada mediante P2P.
Nombre del autor:	Dylan Rivera Valín
Nombre del consultor:	Félix Freitag
Fecha de entrega (mm/aaaa):	01/2018
Área del Trabajo Final:	Sistemas distribuidos
Titulación:	*Máster en Ingeniería Informática*

Resumen del Trabajo (máximo 250 palabras):

Desarrollo desde cero de una aplicación de videollamadas utilizando WebRTC. Adaptación de WebRTC para funcionamiento en dispositivos móviles. Comunicación previa con Socket.IO, envío de notificaciones a terminales y posible implantación en entorno real.

Abstract (in English, 250 words or less):

Development from the beggining of a video call application with WebRTC. Adaptation of WebRTC for working on mobile devices. Signaling between peers with Socket.IO, sending notifications to devices and posible implementation in real environment.

Palabras clave (entre 4 y 8):

Ionic, WebRTC, videollamada, P2P, push, app

Índice

1. Introducción 1
1.1 Contexto y justificación del Trabajo 1
1.2 Objetivos del Trabajo 1
1.3 Enfoque y método seguido 1
1.4 Planificación del Trabajo 2
1.5 Breve sumario de productos obtenidos 3
1.6 Breve descripción de los otros capítulos de la memoria 3
2. Tecnologías 4
2.1 WebRTC 4
2.1.1 Emparejamiento de peers 5
2.1.2 Servidores STUN 5
2.1.3 ¿Por qué WebRTC? 6
2.2 NodeJS 6
2.2.1 Socket.io 6
2.3 Ionic 7
2.3.1 Desarrollo basado en componentes 8
2.3.2 TypeScript 8
2.3.4 Look & feel adaptado al dispositivo 8
2.3.5 Desarrollo y compilado de aplicaciones 8
3. Entendiendo WebRTC 9
3.1 RTCPeerConnection 9
3.2 RTCOffer 9
3.3 RTCSessionDescription 9
3.4 setLocalDescription() y setRemoteDescription() 10
3.5 Workflow de ejemplo 10
4. Aplicación de signaling 12
4.1 El ID de sala 12
4.2 Preshared 12
4.3 userId y professionalId 12
4.4 Conexión con elementos externos 12
4.5 Eventos 13
4.5.1 Evento ‘create’ 13
4.5.2 Evento ‘join’ 13
4.5.3 Evento ‘message’ 14
4.5.4 Otros eventos 14
5. Aplicación de videollamada 15
5.1 Ionic y sus componentes 15
5.1.1 Ionic Platform 15
5.1.2 Notificaciones Push 16
5.2 Registro de llamadas 17
5.3 Llamada 18
5.3.1 Llamada – Médico 18
5.3.2 Llamada – Paciente 18
6. Viabilidad en implantación real 21
7. Problemas encontrados 22

8. Conclusiones 23
9. Glosario 25
10. Bibliografía 26
11. Anexos 27

Lista de figuras

Ilustración 1. Pantalla de registro de llamadas 17
Ilustración 2. Pantalla de notificación de llamada entrante 19

1. Introducción

1.1 Contexto y justificación del Trabajo

En la actualidad, en la empresa donde estoy trabajando (consultoría tecnológica en el ámbito de la salud privada), tenemos un producto donde un usuario puede consultar sus informes, pedir cita previa, hablar con un médico... y pensamos que sería interesante añadir la opción de videollamada entre paciente/medico.

Al ser una empresa pequeña, interesaba realizar un piloto para buscar una solución con poco coste (por eso P2P) y fácil de mantener (por eso aplicación móvil híbrida). Con lo cual, gracias a este proyecto se podrá validar la viabilidad de una implantación real de esta aplicación.

1.2 Objetivos del Trabajo

Crear una aplicación híbrida (web/móvil) de videollamada

1.3 Enfoque y método seguido

Se eligió utilizar WebRTC (una API estándar de comunicación web P2P) y adaptarlo a una aplicación híbrida.
Todo el desarrollo de la aplicación se ha realizado desde cero. Se ha partido desde la API de WebRTC en Javascript y se ha implementado en Ionic sin apenas información, ya que no hay o no se han encontrado librerías ya implementadas open-source.

1.4 Planificación del Trabajo

Tareas planificadas:

Tarea	Inicio	Fin
PEC 2	2/10/18	5/11/18
Configuracion entorno desarrollo Confi...	2/10/18	2/10/18
Desarrollo aplicación de signaling	3/10/18	20/10/18
Instalacion y configuracion servidor TURN	21/10/18	24/10/18
Desarrollo aplicacion Web basica	25/10/18	28/10/18
Testeo y arreglar errores	29/10/18	2/11/18
Elaboracion de informe para PEC 2	3/11/18	4/11/18
PEC 3	5/11/18	15/12/18
Creación aplicativo de videollamada	5/11/18	13/11/18
Análisis para obtener mejor calidad/esta...	14/11/18	17/11/18
Aplicar resultados al algoritmo de videol...	18/11/18	23/11/18
Fase 1 - Aplicación iOS/Android/Web	24/11/18	5/12/18
Testing y arreglar errores - version ALPHA	6/12/18	11/12/18
Elaboracion de informe para PEC 3	12/12/18	14/12/18
Anotaciones y memoria	2/10/18	14/12/18
PEC 4	15/12/18	7/01/19
Informe final	15/12/18	6/01/19
Funcionalidades extra a la APP	15/12/18	6/01/19
Sistema de notificaciones	17/12/18	20/12/18
UI Moderna y usable	15/12/18	6/01/19
Login con captcha	27/12/18	28/12/18
Securización	29/12/18	31/12/18

Ver anexos para el diagrama de Gantt completo.

1.5 Breve sumario de productos obtenidos

Aplicación funcional en web y móvil (Android).
Aplicación de signaling para comunicar los peers

1.6 Breve descripción de los otros capítulos de la memoria

Capítulo 2 – Tecnologías: Explicación de las tecnologías utilizadas y por qué.
Capítulo 3 – Entendiendo WebRTC: Explicación de WebRTC a nivel interno, sus métodos y características.
Capítulo 4 – Aplicación de signaling: Descripción de la aplicación de signaling a nivel interno
Capítulo 5 – Aplicación de videollamada: Descripción de la aplicación híbrida a nivel interno
Capítulo 6 – Viabilidad en implantación: Resumen de conclusiones obtenidas tras el desarrollo realizado
Capítulo 7 – Problemas encontrados: Lista de problemas encontrados durante el desarrollo y explicación breve
Capítulo 8 – Conclusiones
Capítulo 9 – Glosario
Capítulo 10 – Bibliografía
Capítulo 11 – Anexos

2. Tecnologías

Este apartado tiene por objetivo presentar las tecnologías utilizadas en el proyecto y poder entender lo que ofrecen para después ver su implementación.

2.1 WebRTC

WebRTC es una API que está siendo elaborada por la World Wide Web Consortium (W3C) para permitir a las aplicaciones del navegador realizar llamadas de voz, chat de vídeo y uso compartido de archivos mediante P2P sin plugins. *[1]*

Los principales componentes de WebRTC son:

- getUserMedia(), que permite a un navegador web acceder a la cámara y el micrófono
- RTCPeerConnection, que establece las llamadas de audio / vídeo
- RTCDataChannels, que permiten a los navegadores a compartir datos a través de peer-to-peer

getUserMedia() es un método que solicita permiso del usuario para acceder a la cámara. Si el usuario garantiza el acceso, obtendremos un objeto MediaStream. Este objeto representa un flujo de contenido de medios. Un flujo contiene varias pistas, como audio o video.

Actualmente está disponible en:

- Microsoft Edge
- Firefox desde la versión 34 (septiembre de 2015)
- Chrome desde la versión 53 (agosto de 2016)
- Safari desde la versión 11 (septiembre de 2017)
- Opera desde la versión 40 (septiembre de 2016)
- Safari de iOS desde la versión 11 (septiembre de 2017)
- Navegador Android desde la versión 67 (mayo de 2017)
- Chrome para Android

La ausencia destacable de esta lista es Internet Explorer, navegador que no es compatible con la función getUserMedia() y por tanto no permitirá el uso de esta aplicación.

La configuración de una llamada entre peers de WebRTC implica tres tareas:

- Crear un RTCPeerConnection para cada extremo de la llamada y, en cada extremo, añadir el stream local de getUserMedia()
- Obtener y compartir información de red: Los puntos potenciales de conexión son conocidos como candidatos ICE.
- Obtener y compartir descripciones locales y remotas: Metadatos sobre medios locales en formato SDP.

Siguiendo estas premisas, tenemos la creación de RTCPeerConnections, la inclusión del stream de datos en el frontend y la información de la red en el backend gracias a la aplicación de signaling.

2.1.1 Emparejamiento de peers

ICE es un framework que permite a WebRTC superar las complejidades de las redes del mundo real. El trabajo de ICE consiste en encontrar el mejor camino para conectar a sus compañeros.

ICE prueba todas las posibilidades en paralelo y encuentra la opción más eficiente que funcione. Primero intenta hacer una conexión directa utilizando las direcciones de los hosts y si falla (que fallará si los dispositivos están detrás de NATs), ICE obtiene una dirección externa utilizando un servidor STUN. Si esto también falla entonces el tráfico se enruta a través de un servidor TURN.

En otras palabras:

- Un servidor STUN se usa para obtener la dirección de red externa
- Un servidor TURN se utiliza para retransmitir el tráfico si la conexión directa (peer to peer) falla.

Cualquier servidor TURN soporta STUN. Un servidor TURN es un servidor STUN con funcionalidad de retransmisión añadida.

2.1.2 Servidores STUN

Los servidores STUN son necesarios para esta aplicación, están abiertos a internet y tienen una tarea sencilla, comprobar la IP:puerto de una petición y enviar la dirección pública como respuesta.

En otras palabras, la aplicación utilizará el servidor STUN para descubrir su IP:puerto desde una perspectiva pública.

Una vez se tiene este dato, se envía esta información utilizando signaling y se establece la conexión.

Los servidores STUN no tienen mucho que hacer ni recordar, así que servidores modestos pueden soportar un gran número de peticiones.

La mayoría de llamadas RTC hacen una conexión satisfactoria utilizando STUN, un 86% si nos fijamos en webrtcstats.com.

2.1.3 ¿Por qué WebRTC?

Al realizar una aplicación con una parte compleja como es la videollamada P2P es importante apoyarse en estándares testeados para así encontrar soporte, actualizaciones y documentación relativas a la implementación.

Por tanto, la adopción de WebRTC es una buena medida, dado que nos garantiza compatibilidad con la mayoría de navegadores y soporte para la realización del proyecto.

Crear una solución desde cero para transmitir datos utilizando P2P hubiera sido muy costoso, amén de los errores que podríamos encontrar para implementarlo en ciertos navegadores. Al ser WebRTC un estándar, tenemos que adaptarnos a él, pero nos garantiza compatibilidad.

2.2 NodeJS

Node.js es una librería y entorno de ejecución de E/S dirigida por eventos y, por lo tanto, asíncrona. Se ejecuta sobre el intérprete de JavaScript creado por Google llamado V8 que utiliza también Chrome.

Proporciona un entorno de ejecución del lado del servidor que compila y ejecuta javascript a velocidades increíbles. El aumento de velocidad es debido a que V8 compila Javascript en código de máquina nativo, en lugar de interpretarlo o ejecutarlo como bytecode.

Node.js fue formulado para generar un sistema escalable y que tuviese la consistencia suficiente como para poder generar un elevado número de conexiones de forma simultánea con el servidor.

Generalmente cuando se crea un gran número de conexiones el rendimiento y la velocidad de las aplicaciones y páginas web se ven perjudicados. Esto se debe a que la gran mayoría de tecnologías que trabajan desde el lado del servidor accionan las peticiones de forma aislada y mediante hilos independientes. Por eso, cuando la cantidad de solicitudes que se hacen van en aumento, los recursos y el consumo de los mismos también se incrementan.

A este tipo de limitaciones que se generan en el propio servidor también es necesario sumar todas aquellas que posee el cliente (desde la velocidad de su conexión a internet o la memoria RAM de su dispositivo, por ejemplo).

La cantidad de solicitudes, así como los procesos entrantes y salientes, se convierten en uno de los factores limitantes, pero Node.js ha sido concebido para optimizar este hándicap.

2.2.1 Socket.io

Socket.io es una librería en JavaScript para Node.js que permite una comunicación bidireccional en tiempo real entre cliente y servidor. Para ello se basa principalmente en Websocket pero también puede usar otras alternativas

como sockets de Adobe Flash, JSONP polling o long polling en AJAX, seleccionando la mejor alternativa para el cliente justo en tiempo de ejecución.

El funcionamiento de Socket.io es sencillo, el cliente se conecta al servidor al iniciar la aplicación, por tanto, hay un socket abierto mientras la aplicación está encendida que permite la comunicación en tiempo real entre cliente y servidor.

Socket.io se organiza en salas (rooms) y permite crearlas o unirse con facilidad.

Un cliente se conecta a Socket.io y decide unirse a la sala ‘Sala 1’, donde ya hay otros clientes. Una vez está en esa sala, Socket.io permite enviar mensajes que llegarán sólo a los clientes que estén dentro de esa sala, lo que hace que la comunicación entre ellos sea más fácil.

Otra de las características claves de socket.io es que funciona mediante eventos, es decir, tanto cliente como servidor emiten eventos que pueden ser recogidos en cliente o servidor y ejecutar el código asociado.

¿Porqué Socket.io? La facilidad que ofrece al conectarse y compartir información entre cliente y servidor, sumado a la concepción de “salas” es idóneo para la creación de una aplicación de estas características. Como tratamos cada llamada en una sala independiente, nos permite tener agrupados a los dos peers en una sala de Socket.io para así poder lanzar eventos relativos a esa llamada, emitir mensajes a los clientes de una sala…

2.3 Ionic

Ionic es un framework para el desarrollo de aplicaciones híbridas pensado para móviles y tablets, aunque ahora se utiliza también para realizar prácticamente cualquier aplicación web.

Desde el punto de vista más práctico, Ionic es una herramienta con la que se obtienen rápidos resultados sin invertir muchos recursos, ya que podemos reutilizar el código para las diferentes versiones que queremos desarrollar.

La aplicación híbrida es la que permite desarrollar apps para móviles en base a las tecnologías web: HTML + CSS + Javascript. Estas se ejecutan en lo que se denomina un *web view*. Es decir, utiliza el navegador del dispositivo para poder mostrarla.

Estas aplicaciones resultan sumamente interesantes ya que:

1. El mismo código fuente permite que podamos visualizarlas en multitud de sistemas operativos.
2. El coste y tiempo del desarrollo es menor porque no hacen falta equipos que desarrollen para otras plataformas.
3. Se adapta mejor para los desarrolladores que vienen de la web.

En contra, respecto a las aplicaciones que son nativas de cada sistema, también encontramos tres inconvenientes:

1. Menor rendimiento.
2. Menos potentes frente a las nativas.
3. Podemos utilizar menos recursos del dispositivo (hardware) y a los que tenemos acceso, dependemos de plugins.

2.3.1 Desarrollo basado en componentes

Algo que debemos destacar es que las aplicaciones están compuestas por un árbol de componentes, por lo que serán más fácilmente escalables y sostenibles. Podremos resolver pequeños problemas, porque los componentes están estructurados modularmente. Existen componentes para multitud de cosas; implementar un botón, realizar un sistema de navegación por tabs, selectores de fechas, etc.

2.3.2 TypeScript

El uso del lenguaje TypeScript está pensado para mejorar el trabajo de los desarrolladores haciéndolos más productivos. TypeScript es, en realidad, Javascript agregando algunas cosas más que van a ayudarnos durante la etapa de desarrollo y en el mantenimiento futuro de las aplicaciones (como un tipado fuerte). TypeScript requiere una transpilación del código, pero de eso se encarga Ionic por debajo, no representado ningún problema para el desarrollador.

2.3.4 Look & feel adaptado al dispositivo

Cuando compilamos una aplicación para cada sistema, los componentes de Ionic se adaptan estéticamente a las reglas de cada uno. Por lo que la experiencia de usuario no variará a lo que estemos acostumbrados y los desarrolladores no tendrán que añadir ningún recurso adicional para que esto sea así.

2.3.5 Desarrollo y compilado de aplicaciones

Cuando terminemos la app deberemos compilarla para disponer de los ejecutables específicos de cada dispositivo. Esto lo deberemos de realizar cada vez que queramos lanzar una nueva versión y ofrecérsela al usuario para su descarga.

¿Porqué Ionic? El tener un solo desarrollo para varias plataformas es algo que se valora muchísimo a la hora de crear proyectos, sobre todo si se dispone de pocos recursos como es el caso. Tener que mantener una sola aplicación frente a tres es un factor decisivo, amén de la facilidad de programación ya que he trabajado con Angular antes.

3. Entendiendo WebRTC

WebRTC provee los métodos para enviar el video entre dos clientes y, una vez establecida la conexión, se utiliza P2P para la transmisión. Para lograr establecer la conexión, se necesita realizar un proceso de signaling, es decir, los clientes necesitan intercambiar información:

- Mensajes de control de sesión para abrir o cerrar comunicación.
- Mensajes de error
- Metadatos de “media” (codecs, configuración de codecs, ancho de banda y tipos)
- Datos “clave” para establecer conexiones seguras.
- Datos de red, como la IP y puerto del host para que sea “visible”

Este intercambio no es posible mediante P2P, ya que los peers no se conocen de antemano, por lo que es necesario un servidor de signaling (o emparejamiento de peers, algo así como el tracker de BitTorrent). *[2]*

3.1 RTCPeerConnection

La interfaz RTCPeerConnection representa una conexión WebRTC entre un computador local y un par remoto (otro computador). Esta interfaz provee métodos para conectar un equipo remoto (remote peer), mantener y monitorear esa conexión y cerrarla una vez que no se necesite más. *[3]*

3.2 RTCOffer

El método createOffer () de la interfaz RTCPeerConnection inicia la creación de una oferta SDP con el fin de iniciar una nueva conexión WebRTC a un peer remoto. La oferta del SDP incluye información sobre cualquier MediaStreamTrack que ya esté adjunta a la sesión de WebRTC, el códec, las opciones compatibles con el navegador y cualquier candidato que el agente de ICE ya haya reunido con el fin de enviarlo por el canal de señalización a un posible peer y así solicitar una conexión o para actualizar la configuración de una conexión existente.

El valor de retorno es una Promesa que, cuando se ha creado la oferta, se resuelve con un objeto RTCSessionDescription que contiene la oferta recién creada. *[3]*

3.3 RTCSessionDescription

La interfaz RTCSessionDescription describe un extremo de una conexión, o una conexión potencial y cómo está configurada. Cada RTCSessionDescription consta de un tipo de descripción que indica qué parte del proceso de negociación de oferta / respuesta describe y del descriptor de SDP de la sesión. *[3]*

El proceso de negociación de una conexión entre dos peers implica el intercambio de objetos RTCSessionDescription de ida y vuelta, cada descripción sugiere una combinación de opciones de configuración de conexión que admite el remitente de la descripción. Una vez que los dos peers acuerdan una configuración para la conexión, la negociación se completa. *[3]*

3.4 setLocalDescription() y setRemoteDescription()

Los métodos setLocalDescription() y setRemoteDescription() cambian la descripción asociada con un extremo de la conexión. Reciben como parámetro un objeto RTCSessionDescription y, en el caso de setLocalDescription, pone como extremo local el objeto sesión pasado (la configuración local) y, en el caso de setRemoteDescription marca como extremo remoto al otro peer (la configuración remota) para así establecer la conexión.

Es decir, con el método setLocalDescription ponemos en el RTCPeerConnection la información local y en setRemoteDescription la información del otro peer para establecer los puntos de conexiónen RTCPeerConnection.

3.5 Workflow de ejemplo

Para iniciar la llamada RTCPeerConnection tiene dos tareas:

- Determinar las condiciones de los medios locales, como las capacidades de resolución y códec. Estos son los metadatos utilizados para el mecanismo de oferta y respuesta.
- Obtener posibles direcciones de red para el host de la aplicación, conocidas como candidatos.

Una vez que se han determinado estos datos locales, se deben intercambiar a través de un mecanismo de signaling con el peer remoto.

Ejemplo de Peer 1 (Ana) intentando llamar a Peer 2 (David):

1. Ana crea un objeto RTCPeerConnection.
2. Ana crea una “oferta” (un SDP sesión) con el método createOffer()
3. Ana llama a setLocalDescription() con esta “oferta”.
4. Ana envía la oferta a David utilizando el mecanismo de signaling.
5. David llama a setRemoteDescription() con la oferta de Ana, así que David ya sabe la configuración de Ana.
6. David llama a createAnswer().
7. David setea su respuesta como una descripción local llamando a setLocalDescription()
8. David utiliza el método de signaling para enviar la respuesta a Ana.
9. Ana setea la descripción igual que David, con setRemoteDescription().
10. Ana y David también necesitan intercambiar información de la red. La expresión “encontrar candidatos” se refiere al proceso de encontrar interfaces y puertos de red utilizando el framework ICE.
11. Ana crea un RTCPeerConnectionObject con un handler onIceCandidate.
12. El handler se llama cuando el candidato de red está disponible.

13. En el handler, Ana envía su información de candidato a David utilizando el canal de signaling.
14. Cuando David obtiene el mensaje de candidato de Ana, ella llama addIceCandidate(), para añadir el candidato a la descripción de peer remota.

4. Aplicación de signaling

En este capítulo se explica el desarrollo y funcionamiento de la aplicación de signaling, necesaria para establecer el contacto previo a la conexión de los dos peers.

Como introducción, hay que tener en cuenta que esta aplicación está pensada para que las llamadas se realicen siempre desde un usuario médico a un paciente, es decir, el paciente no podrá realizar llamadas al médico.

4.1 El ID de sala

El id de sala es un objeto JSON codificado en Base64 que contiene los siguientes campos:

- preshared: Una clave que debe coincidir con la del servidor para autenticar las llamadas.
- userId: Id del usuario al que se va a llamar
- professionalId: Id del profesional que realiza la llamada

Esto es muy importante de entender, ya que sobre esta idea parte el desarrollo de la aplicación de signaling.

Gracias a que en el id de sala llevamos codificados ciertos parámetros, nos será posible más adelante saber quién está llamando a quién y cómo (aparte de validaciones de seguridad).

4.2 Preshared

La preshared es una clave que está hardcoded en el servidor y nos sirve como filtro previo a las peticiones. La preshared que se envía desde los clientes debe coincidir con la que tenemos en el servidor, sino no se podrán realizar acciones.

4.3 userId y professionalId

Son los identificadores de las personas implicadas en la llamada. UserId corresponde al paciente y professionalId al médico.

4.4 Conexión con elementos externos

La aplicación de signaling se conecta también con la base de datos de la aplicación existente con los datos de pacientes para guardar datos como los tokens de dispositivo, obtener el histórico de llamadas, comprobar los datos de usuarios…

También se conecta con Firebase para la gestión de notificaciones y se han creado métodos para poder enviar notificaciones según el id de dispositivo.

4.5 Eventos

Como hemos comentado anteriormente, socket.io se basa en eventos, es decir, tenemos trozos de código que se ejecutan cuando se realizan ciertas acciones.

En el caso de socket.io, cuando un socket hace .emit() tiene que pasar un string que se corresponde con el nombre del evento y, si lo requiere, parámetros.

Esto vale tanto en cliente como en servidor, podemos emitir desde cliente y recoger en servidor y viceversa.

Ejemplo:

```
Cliente->   socket.emit('create');

Servidor-> socket.on('create', function(){
               Console.log('room created');
           });
```

Como podemos ver, la gestión de eventos es sencilla tanto en cliente como en servidor, basta con ejecutar el método emit() y pasar como parámetro un string que será el que luego tenemos que programar siguiendo la notación:

socket.on('NOMBRE DEL EVENTO');

Una vez comprendemos cómo funciona el sistema de eventos, pasamos a comentar los más críticos del lado del servidor.

4.5.1 Evento 'create'

El evento create se ejecuta cuando desde un navegador (médico) se crea una sala. Éste método recibe como parámetro el id de sala y, tras decodificar el base64, comprueba que la preshared coincida y, si es así, crea la sala.

Una vez la sala está creada, falta notificar al paciente que le están llamando, para lo cual gracias al parámetro userId (obtenido del id de sala) la aplicación de signaling se conecta a la base de datos de la aplicación y envía una notificación al terminal conforme le están llamando.

4.5.2 Evento 'join'

El evento join se ejecuta cuando desde el terminal se acepta la llamada. El método recibe como parámetro el id de sala y el usuario logueado en la aplicación.

Primero se comprueba que la preshared sea correcta y además se comprueba que el id del usuario incluido en el objeto base64 del id de sala coincida con el id de usuario de la aplicación. Gracias a esta medida, añadimos una capa más

de seguridad al comprobar que el usuario que intenta unirse a la llamada es realmente a quien va dirigida.

Si todo ha ido correctamente se emite 'ready' a todos los usuarios de la sala, evento que se recoge en el cliente para iniciar la llamada.

4.5.3 Evento 'message'

Este evento es sencillo, pero no por ello menos importante. Recibe como parámetro un string con el mensaje y lo envía a todos los usuarios de la sala.

Éste es el evento que se utiliza para enviar la información de sesión de un peer a otro. También se podría utilizar para enviar mensajes de texto si habilitáramos un chat.

4.5.4 Otros eventos

Aprovechando la conexión a Socket.io y la conexión de éste con la base de datos, se han desarrollado varios eventos más para el desarrollo de la aplicación.

Tenemos el evento **getCalls** que hace una consulta a base de datos y nos devuelve las llamadas del usuario en cuestión (necesario para la pantalla de registro de llamadas) y el evento **incomingCalls** que nos devuelve si hay alguna llamada en curso (necesario para las notificaciones, ya que así podemos disparar el evento de llamada entrante).

5. Aplicación de videollamada

La aplicación de videollamada en sí, se ha realizado con Ionic versión 4.0.0 beta rc.1

Se empezó desarrollando un piloto, una aplicación sencilla que al abrir simplemente realizara una videollamada a un cliente fijado previamente en el código.

Esta primera aproximación fue costosa, ya que acceder a una cámara de dispositivo mediante un código híbrido fue una odisea, pero al final se consiguió creando un código adaptable a la plataforma elegida.

En este capítulo se recorrerán las áreas de la aplicación de videollamada, así como su implementación.

5.1 Ionic y sus componentes

Lo primero a destacar es que la mayoría de la aplicación está diseñada con componentes de Ionic. Se puede apreciar cómo se siente realmente una aplicación móvil y es por los componentes de UI de Ionic Native.

Éstos han sido de fácil implementación y permiten cierta personalización como los tamaños o colores, que son pasados como parámetros en el código HTML.

También se han utilizado varios plugins nativos de Cordova, lo cual ha complicado el desarrollo al lanzar errores si se ejecutaban en Web. Es aquí donde entra a formar parte el componente Platform.

5.1.1 Ionic Platform

Ionic Platform es un componente de Ionic que nos da información sobre el dispositivo que está ejecutando el código. Es una herramienta muy fácil de utilizar pero muy útil en desarrollos híbridos.

Gracias a su método .is() que recibe como parámetro la plataforma que queramos, ha sido posible diferenciar el código de móvil del de navegador.

Dado que los plugins de Cordova instalados eran compatibles con iOS y Android, la diferenciación que hacemos en el código es platform.is('cordova') que nos devuelve 'true' si estamos en dispositivos corriendo Cordova (iOS o Android) y 'false' si estamos en el navegador.

5.1.2 Notificaciones Push

Se ha dotado a la aplicación de notificaciones push, para notificar cuando se recibe una llamada.

Para realizarlo, se ha utilizado la librería de ionic-native push y el plugin phonegap-plugin-push.

Para enviarlas se utiliza Firebase Cloud Messaging, ya que es necesario un servidor de notificaciones. Se ha de crear un proyecto en Firebase y obtener una clave de proyecto de Google (el archivo google-services.json), para integrarlo en el proyecto y validar que la aplicación corresponde con el proyecto de Firebase.

Una vez está todo configurado, el funcionamiento es sencillo. Es necesario crear un canal de notificaciones y suscribirse a ese canal (en Android). Una vez el terminal se registra conforme quiere recibir notificaciones, se nos asigna un deviceId, que es una clave única que identifica al terminal y que nos proporciona Firebase.

Debemos tener esta clave siempre para así enviar las notificaciones. (Es posible que vaya cambiando durante las ejecuciones, es única por dispositivo y sesión digamos).

Así que lo primero que hay que hacer desde la aplicación es “registrarnos” en Firebase, obtener el DeviceID y guardarlo en base de datos asociándolo al id de usuario de la aplicación.

Una vez tenemos esto, enviar notificaciones desde Node.js es tan sencillo como enviar una petición al servidor de Firebase con los datos del proyecto de Firebase y el certificado generado, además de los datos propios de la notificación (título, mensaje, deviceId…)

5.2 Registro de llamadas

Para dotar a la aplicación de mayor funcionalidad y tener un producto más redondo, se ha creado una pantalla donde poder visualizar las llamadas.

Las llamadas aparecen en tres pestañas, gestionadas por el componente ion-tabs. Una vez tenemos la vista inicial, creamos rutas con el parámetro 'status'. Se recoge este parámetro y se envía un evento getCalls a socket.io con el status como parámetro, lo cual nos devuelve las llamadas de ese estado en concreto.

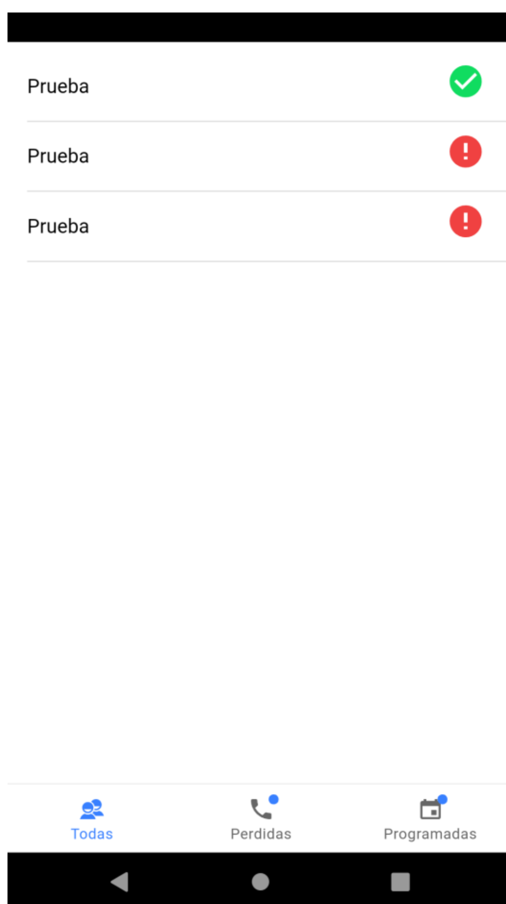

Ilustración 1. Pantalla de registro de llamadas

5.3 Llamada

Para realizar una videollamada, tal y como comentamos desde la aplicación de signaling es necesario que inicie la conexión un médico.

El médico iniciará la aplicación en modo web, por tanto, en este apartado vamos a diferenciar las dos vistas y sus comportamientos por separado.

5.3.1 Llamada – Médico

El médico abre directamente una URL con el formato siguiente:

IP:PUERTO/call/ID-DE-SALA

Este ID de sala ya vendrá generado de antes, ya que como hemos comentado, el ID de sala es un objeto JSON con la preshared, el id de usuario y el id del profesional.

Una vez estamos en la página, el sistema detecta que estamos en un navegador, por tanto, corresponde al médico y procedemos a lanzar el evento 'create' con parámetro el id de sala.

Si todo ha ido correcto, esto enviará una notificación al usuario. Si el usuario acepta y todo va correctamente, se disparará el evento 'ready' que iniciará la creación del RTCPeerConnection y el traspaso de mensajes mediante la aplicación de signaling.

Una vez la conexión se ha realizado, recibimos el stream del paciente y lo mostramos en pantalla.

5.3.2 Llamada – Paciente

La llamada para un paciente es un proceso más complejo y donde se han dedicado más esfuerzos para hacerlo un proceso similar a lo que sería una aplicación de videollamada normal y corriente.

Dando por hecho que el médico ha iniciado una llamada, la aplicación de signaling envía una notificación.

Una vez aquí, tenemos dos formas de actuar:

- Si el usuario está dentro de la aplicación
- Si el usuario no está dentro de la aplicación

Esta distinción es muy necesaria, ya que se ha trabajado mucho en hacer este proceso invisible para el usuario, pero desde el punto de vista del desarrollo ha habido varios problemas para implementar.

El principal problema que hay es que las notificaciones funcionan perfectamente en Ionic pero si la aplicación no está abierta, no se permite

personalizar la acción a realizar una vez pulsada la notificación (almenos no de forma estándar, tendríamos que entrar a tocar las builds de cada plataforma).

Si el usuario está con la aplicación abierta, el sistema detecta que hay una nueva notificación y si ésta contiene como dato adicional un roomId, el sistema redirige automáticamente hasta la llamada.

Si la aplicación no está abierta la notificación llega, pero no podemos recoger el roomId automáticamente y redirigir, por lo que se creó el evento 'incomingCalls'. Nada más abrir la aplicación, se dispara el evento incomingCalls que lanza una query a base de datos para ver si hay alguna llamada activa. Si es así, el sistema redirigirá hacia esa llamada.

Ésta 'trampa' es la única manera que hemos conseguido solucionar el problema de las notificaciones cuando la aplicación no está abierta. La notificación en sí nos recuerda que tenemos una llamada, pero los datos que contiene no sirven ya que no se pueden recoger al no estar la aplicación corriendo.

Con el tema de las notificaciones solucionado y funcionando, al entrar en la página de llamada el sistema detecta si estamos en móviles o navegador y, en este caso, nos muestra una pantalla similar a la de cualquier aplicación de videollamada con un sonido típico de teléfono sonando.

Esta pantalla, con un efecto visual potente de cara al usuario, ya que dota a la aplicación de mayor credibilidad, sirve para rechazar la llamada o aceptarla.

Ilustración 2. Pantalla de notificación de llamada entrante

Si se acepta, se crea el objeto RTCPeerConnection y se intercambian mensajes, es decir, comienza la videollamada.

Si se cancela, el sistema redirige a la pantalla de inicio (el registro de llamadas).

También, para hacer frente a los flujos propios de una aplicación móvil, se ha contemplado que el usuario pulse el botón de atrás mientras sale la pantalla previa a la llamada o durante ésta.

Si se realiza esa acción, se registra que hay una llamada en curso y se crea un header en la aplicación al más puro estilo iOS que nos recuerda que tenemos una llamada pendiente y, si pulsamos, nos redirige a la llamada de nuevo.

6. Viabilidad en implantación real

En el estado actual, la aplicación sirve como piloto para mostrar el potencial de WebRTC y una implantación real en red local.

Al inicio del proyecto, estaba planeado analizar la calidad de la videollamada según si era con wifi, mediante datos móviles… y cómo afectaba al enlace.

Esto no ha sido posible, ya que, aunque se montó la aplicación en un servidor de producción con ip externa, debido a cuestiones de seguridad no se permite utilizar WebRTC en sitios web no autorizados (es decir, sin certificado https validado por Google).

Se han realizado pruebas pero sin resultado, ya que al final había que terminar estando en red local porque no podíamos ni obtener el stream local (no funciona el método getUserMedia()) si no estábamos en localhost.

Debido a esto, el análisis de consumo de red no se ha realizado, por lo que se han dedicado más recursos a la aplicación en general para dotarla de más características que no estaban previstas al principio y así intentar paliar la falta de datos.

Una vez comentado esto, la viabilidad de implantación real del proyecto es bastante alta. Dado que la comunicación es P2P, no necesitamos servidores para almacenar el video, con lo cual la implementación actual funcionaría sin mayor coste que lanzar la aplicación de signaling (que consume muy poco).

Donde sí habría que dedicar más recursos es a la parte de la aplicación. Actualmente funciona, pero un desarrollo móvil cuenta con la complicación de que hay millones de dispositivos y miles de configuraciones, con lo cual habría que realizar un testeo bastante alto y proveer al sistema de una mejor recuperación de errores para depurar el resultado final.

7. Problemas encontrados

Durante la realización del trabajo se han encontrado varios problemas que han dificultado el desarrollo.

El primero fue la integración de WebRTC en Ionic. WebRTC es una API Web que ha de ser compatible con los navegadores para ejecutarse. La mayoría de navegadores de sobremesa la aceptan e incluso de móvil, pero sólo los nativos, y aquí está el problema.

En el caso de iOS, WebRTC es compatible con Safari, pero Ionic utiliza aplicaciones basadas en Webkit, por lo que la implementación básica no funcionará en iOS (sí funcionará si entramos en la web, pero no como aplicación nativa).

En el caso de Android, hemos tenido algún problema en algún terminal, ya que no muestra el video, pero sí lo envía.

La parte difícil fue adaptar la librería y los métodos de WebRTC que son JavaScript a Ionic (que utiliza Angular y Typescript). Se tuvieron que adaptar los métodos y callbacks para que funcionara.

Otro problema fue el mostrar el video, ya que en Angular hay opciones que no permiten realizarse de manera básica al estar fuertemente tipado (no así en JavaScript). Por lo que tuvieron que explorarse otras vías para poner como source del tag video el stream remoto.

Crear las notificaciones también fue un proceso laborioso, ya que no sabía cómo funcionaban y tener que depender de una entidad externa como Firebase y adaptar las peticiones allí fue más costoso de lo pensado al principio.

8. Conclusiones

El trabajo ha resultado ser un éxito. Hemos podido realizar un piloto de videollamada en WebRTC y además se ha dotado a la aplicación de ciertos elementos visuales que lo hacen un producto redondo y presentable. Se han cumplido la mayoría de los objetivos (aunque han costado más de lo que se pensaba inicialmente) pero también se han encontrado algunos aspectos donde habría que investigar más y que hacen que la implantación en un entorno real merezca unas cuantas horas más de desarrollo.

El core funciona y lo hace rápido, la aplicación va bien y fluida, pero hay ciertos aspectos que aún faltan controlar si se quiere utilizar como producto vendible.

Por ejemplo, faltaría reconectar los peers si la llamada se corta (actualmente sólo permite llamar y colgar). Faltaría adaptarlo a iOS y mirar por qué no funciona en algunos dispositivos y, sobretodo, faltaría una capa extra de control de errores que permitan al usuario entender qué está pasando y le permitan solucionarlo (se han subsanado muchos errores, pero seguro que faltan más por pulir).

De ahí a que el resultado sea calificado como éxito a medias. Es verdad que sirve para hacerse una idea del concepto, que el core es estable, pero faltaría toda la parte de la aplicación ya que hay tantos terminales y flujos que el trabajo de optimización llevado a cabo durante una semana se antoja poco para un producto final.

La planificación ha sido cumplida parcialmente. El análisis de los datos consumidos en red local vs red externa no se ha podido realizar por motivos técnicos, pero se han incluido funciones extra para paliar eso. Los demás hitos se han cumplido, pero ha habido que dedicarle más horas de las previstas. Es decir, se han cumplido los objetivos por tener más carga de trabajo, lo que en un entorno real nos hubiera llevado a:

- Realizar horas extra
- Aumentar el tiempo de entrega

Esto ha sido por sobreestimar el desarrollo de una aplicación móvil. Si hubiéramos realizado una aplicación Web los plazos hubieran sido más acertados, ya que no esperaba tantas complicaciones para obtener la cámara del dispositivo, enviarla y adaptar la api de WebRTC a Typescript.

Pese a todo, considero que el trabajo entregado está por encima de mis expectativas (sobre todo gracias a la inclusión de las notificaciones y esa capa de User Interface aplicada estas últimas semanas) y que cumple la función como piloto o “demo” de lo que es posible realizar mediante WebRTC y Ionic.

De cara a futuras funcionalidades, se ha dejado la puerta abierta al envio de información entre peers mediante RTCPeerDataChannel, es decir, una vez que ya tenemos la conexión entre peers, enviar información sin pasar por el

servidor de signaling. También ha faltado explorar más la transmisión de video fuera de la red local, ya que los navegadores no aceptan transmisiones externas si no procede de un sitio web autorizado, es decir, con un certificado válido https, cosa de la que no disponemos a la hora de hacer el trabajo. Aun y así, la aplicación de signaling sí que se ha montado en un servidor externo, con resultados positivos.

9. Glosario

Definición de los términos y acrónimos más relevantes utilizados dentro de la Memoria.

P2P: Una red peer-to-peer, red de pares, red entre iguales o red entre pares (P2P, por sus siglas en inglés) es una red de dispositivos en la que todos o algunos aspectos funcionan sin clientes ni servidores fijos, sino una serie de nodos que se comportan como iguales entre sí. Es decir, actúan simultáneamente como clientes y servidores respecto a los demás nodos de la red. Las redes P2P permiten el intercambio directo de información, en cualquier formato, entre los ordenadores interconectados.

Peers: Nodo de la red P2P.

Aplicación híbrida: Las aplicaciones móviles híbridas son una combinación de tecnologías web como HTML, CSS y JavaScript, que no son ni aplicaciones móviles verdaderamente nativas, porque consisten en un WebView ejecutado dentro de un contenedor nativo, ni tampoco están basadas en Web, porque se empaquetan como aplicaciones para distribución y tienen acceso a las APIs nativas del dispositivo.

WebSockets: WebSocket es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor.

Callbacks: Una devolución de llamada o retrollamada (en inglés: callback) es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A".

Cordova: Apache Cordova es un marco de desarrollo móvil de código abierto. Permite utilizar las tecnologías estándar web como HTML5, CSS3 y JavaScript para desarrollo multiplataforma, evitando el lenguaje de desarrollo nativo de cada plataforma móvil.

Base64: Base 64 es un sistema de numeración posicional que usa 64 como base. Es la mayor potencia de dos que puede ser representada usando únicamente los caracteres imprimibles de ASCII. Esto ha propiciado su uso para codificación de correos electrónicos, PGP y otras aplicaciones.

10. Bibliografía

[1] https://es.wikipedia.org/wiki/WebRTC.
Fecha de consulta: 27/12/2018

[2] https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/.
Fecha de consulta: 01/10/2018 - Utilizado primero en PEC 1

[3]https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/
*Consulta recurrente debido a que posee información sobre los métodos de WebRTC

[4] https://nodejs.org/api/index.html
Fecha de consulta: Durante todo el desarrollo
*Consulta recurrente ya que contiene información sobre node y sus métodos

[5] https://codelabs.developers.google.com/codelabs/webrtc-web/#0
*Consulta sobre la que se basó la primera implementación de test y de donde se han extraido varios conceptos fundamentales

[6] https://ionicframework.com/
*Consulta recurrante durante todo el desarrollo. En el apartado de recursos se encuentran ejemplos de cómo implementar muchas funcionalidades.

11. Anexos

11.1 Diagrama de Gantt.

Para una correcta visualización se adjunta junto a la memoria, ya que es imposible colocarlo de una manera leíble en la memoria.

11.2 Entregables

Hay tres entregables:

Proyecto base
Aplicación de signaling
Versión Web
APK

11.2.1 Proyecto base

El proyecto base contiene el código de la aplicación híbrida. Si tenemos NodeJS instalado, hay que entrar en el directorio del proyecto y ejecutar 'npm install' para instalar las dependencias. Una vez esté todo instalado, si ejecutamos 'ionic serve' compilará la versión web para mostrarla en el navegador, si por el contrario escribimos 'ionic cordova run Android' compilará la versión de Android y la ejecutará si hay algún terminal/emulador conectado.

Si queremos que la aplicación apunte a nuestra aplicación de signaling hay que ir al archivo app.module.ts y poner en el campo SocketIoConfig la IP donde correrá la aplicación de signaling.

```
const config: SocketIoConfig = { url: 'http://IP:PUERTO', options: {} };
```

Por defecto apunta a un servidor de test ya que es donde está instalada la base de datos con los datos de prueba.

11.2.2 Aplicación de signaling

Es la aplicación de signaling que se utiliza como puente para comunicar los dos peers. Actualmente está corriendo en un servidor de test, ya que era necesario que se conectara a una base de datos para guardar los datos del token de registro. Si aún así se quiere ejecutar hay que tener node instalado y ejecutar el comando 'node app.js'.

11.2.3 Versión Web

Es la aplicación compilada para web, basta con ponerla en un servidor Apache para visualizarla (o ejecutar 'ionic serve' en el proyecto base).

11.2.4 APK

Es la aplicación compilada para Android, hay que instalarla en un dispositivo (primero asegurarse de que se acepta la instalación de aplicaciones de fuentes

desconocidas) y una vez instalada hay que darle los permisos manualmente para asegurar, ya que hay veces que no obtiene los permisos correctamente.

11.2.5 Sala de testing

Dado que actualmente en la última versión se valida si el usuario está en la base de datos y coincide con el del id de sala, se proporciona un CallID de testing para que puedan probar.

La url quedaría así:

ip:puerto/call/e3ByZXNoYXJlZDogMTIzNCwgdXNlcklkOiAxLCBwcm9mZXNza W9uYWxJZDogNjR9