MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

TRABAJO FIN DE MASTER

DESARROLLO DE APLICACION P2T CON WEBRTC

Autor: Pedro Murcia Morilla

Director: Rogelio Martínez Perea

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Desarrollo de aplicación P2T con WebRTC

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2021/22 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Pedro Murcia Morilla Fecha: ……/ ……/ ……

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Rogelio Martínez Perea Fecha: ……/ ……/ ……

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D._____

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: _____, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente)*.

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a …….. de ……... de …….

ACEPTA

Fdo……

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

TRABAJO FIN DE MASTER

DESARROLLO DE APLICACION P2T CON WEBRTC

Autor: Pedro Murcia Morilla

Director: Rogelio Martínez Perea

Madrid

Agradecimientos

A mi familia y amigos

DESARROLLO DE APLICACIÓN P2T CON WEBRTC

Autor: Murcia Morilla, Pedro.
Director: Martínez Perea, Rogelio.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

En este proyecto, se realiza el desarrollo de una aplicación P2T, similar al *walkie-talkie*, que emplee WebRTC como motor de comunicación multimedia. Se usará React para la parte visual y Janus en un contenedor Docker para poder desplegarlo fácilmente. El resultado es una aplicación que puede enviar audio, vídeo y datos cuando es tu turno.

Palabras clave: WebRTC, React, P2T, walkie-talkie, Janus

1. Introducción

Las tecnologías de internet no hacen más que evolucionar hasta niveles en los que la diferencia entre una aplicación nativa y una de navegador se hace bastante imperceptible. Casos como WebAssembly son los más notorios, que ofrecen una interfaz JavaScript para ejecutar código nativo en el navegador. Ya se usan en varias aplicaciones como SketchUp o AutoCAD Web App [1].

Una de las áreas donde se ha visto más evolución es en el área de las comunicaciones multimedia, donde han surgido plataformas de *streaming* o *Video on Demand* (VoD) a partir de innovaciones. Algunas de estas plataformas, como Netflix, YouTube o Amazon Prime Video, emplean tecnologías de Internet para transmitir los vídeos al cliente. De todas estas tecnologías, WebRTC es un estándar que permite la comunicación entre dos navegadores [2], ya que hace mucho más sencilla la implementación de una comunicación multimedia.

De todos los servicios de comunicación, los servicios P2T son los más empleados por los servicios de emergencias, policía y bomberos. P2T tiene un comportamiento similar al *walkie-talkie*, y tiene una estructura muy sencilla. Es por esto por lo que se trata de un servicio bastante resistente a fallos de comunicación, y hace que la comunicación sea más clara a través de turnos de palabra. Las ventajas incluyen además que el ancho de banda empleado es mínimo, por lo que se pueden emplear varios canales para la comunicación.

2. Definición del proyecto

En este proyecto, se desarrollará una aplicación que utilice los servicios de WebRTC para establecer una comunicación por turnos entre varios participantes. Como objetivo futuro, se planteará para añadir compatibilidad a la comunicación P2T con los servicios de Internet. Se empleará React para la parte visual, y Janus como *gateway* WebRTC para las comunicaciones que se realicen en la aplicación. Para la parte de la lista, se empleará una base de datos no relacional en formato JSON llamada Firebase, que tiene varias herramientas que permiten la facilidad de establecer valores en función del momento en el que se crean. De esta manera, la lista se puede crear de una forma sencilla y fácil de convertir en otro tipo de información.

3. Descripción del modelo/sistema/herramienta

La aplicación empleará tres servicios para establecer la comunicación WebRTC. En primer lugar, la autenticación del usuario en Firebase permitirá obtener la información sobre las salas a las que se ha unido, identificadores de sesión y otros. Esta información es accesible a través del segundo servicio de base de datos en tiempo real de Firebase, que se implementará para cargar la información pertinente para establecer la sesión. Por último, se generará un componente que implemente llamadas al servidor Janus para gestionar la sesión WebRTC y las diferentes salas.

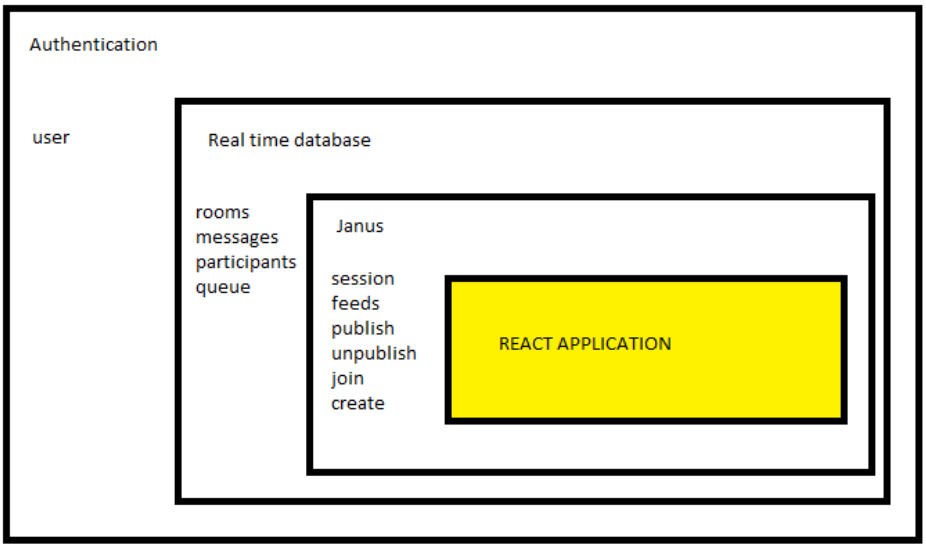


Ilustración 1. Arquitectura general de la aplicación

4. Resultados

- Una aplicación que manda correctamente la información de los *streams* locales al resto de participantes
- Una lista de espera que añade a los participantes cuando piden la palabra y se eliminan cuando acaba
- Una interfaz de usuario moderna con React y MaterialUI.

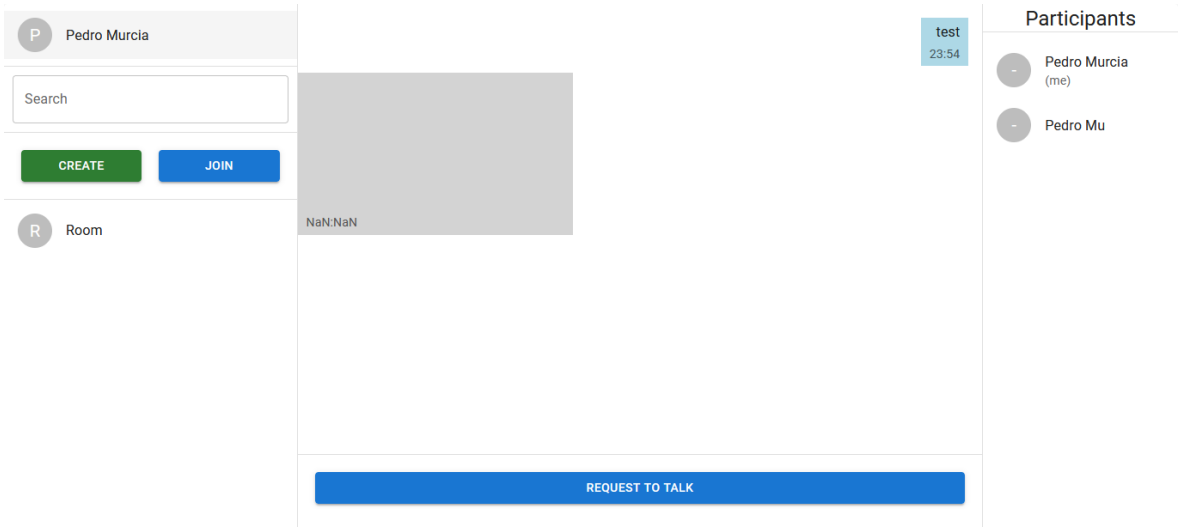


Ilustración 2. Muestra de la aplicación

5. Conclusiones

La aplicación que se ha creado se conecta correctamente con el *Gateway* Janus para generar comunicaciones multimedia entre varios participantes. El establecimiento de la comunicación se realiza por turnos, por lo que la comunicación es ordenada y clara. Se pueden mandar tanto mensajes como audio y vídeo para facilitar la dinamicidad en el uso de este servicio. Esta aplicación, además, se puede transformar en código JavaScript que se puede servir directamente desde un servidor HTTP. El interfaz de usuario es moderno, y las tecnologías y metodologías empleadas siguen el estándar de la industria, que permite un mantenimiento más sencillo y vías para añadir nuevas funcionalidades en el futuro.

6. Referencias

[1] «Made with WebAssembly,» [En línea]. Available: https://madewithwebassembly.com/. [Último acceso: 28 Enero 2022].

[2] Google Developers, «WebRTC,» [En línea]. Available: https://webrtc.org/. [Último acceso: 1 Enero 2022].

P2T APPLICATION DEVELOPMENT WITH WEBRTC

Author: Murcia Morilla, Pedro.
Supervisor: Martínez Perea, Rogelio.
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

In this project, the development of a P2T application, like the walkie-talkie, that uses WebRTC as a multimedia communication engine, is carried out. React will be used for the visual part and Janus in a Docker container to be able to deploy it easily. The result is an app that can send audio, video, and data when it's your turn.

Keywords: WebRTC, React, P2T, walkie-talkie, Janus

1. Introduction

Internet technologies have evolved to a level where the difference between a native application and a browser application becomes quite imperceptible. Cases like WebAssembly are the most notorious, offering a JavaScript interface to execute native code in the browser. They are already used in various applications such as SketchUp or AutoCAD Web App [1].

One of the areas where more evolution has been seen is in multimedia communications, where streaming platforms or Video on Demand (VoD) have emerged from these innovations. Some of these platforms, such as Netflix, YouTube, or Amazon Prime Video, use Internet technologies to transmit the videos to the client. Of all these technologies, WebRTC is a standard that allows communication between two browsers [2] and makes the implementation of multimedia communication much easier.

Of all communication services, P2T services are the most used by emergency services, police, and firefighters. P2T behaves similarly to walkie-talkie and has a very simple structure. Therefore it is a fairly resistant service to communication failures and makes communication clearer through turns. The advantages also include that the bandwidth used is minimal, so several channels can be used for communication.

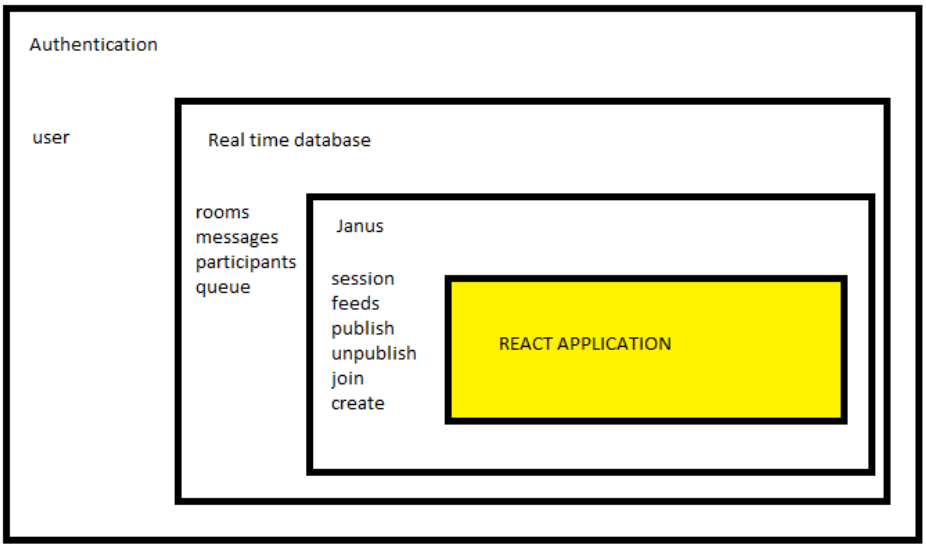
2. Project definition

In this project, an application will be developed that uses the services of WebRTC to establish a communication between several participants. As a future objective, it will be proposed to add compatibility to P2T communication with Internet services. React will be used for the UI, and Janus will be used as the WebRTC gateway for the communications that take place in the application. For the list part, a non-relational database in JSON format called Firebase will be used, which has several tools that allow the ease of setting values based on when they are created. In this way, the list can be created in a simple way and easy to convert into other types of information.

3. Model/system/tool description

The application will use three services to establish WebRTC communication. First, the user authentication in Firebase will allow to obtain the information about the rooms

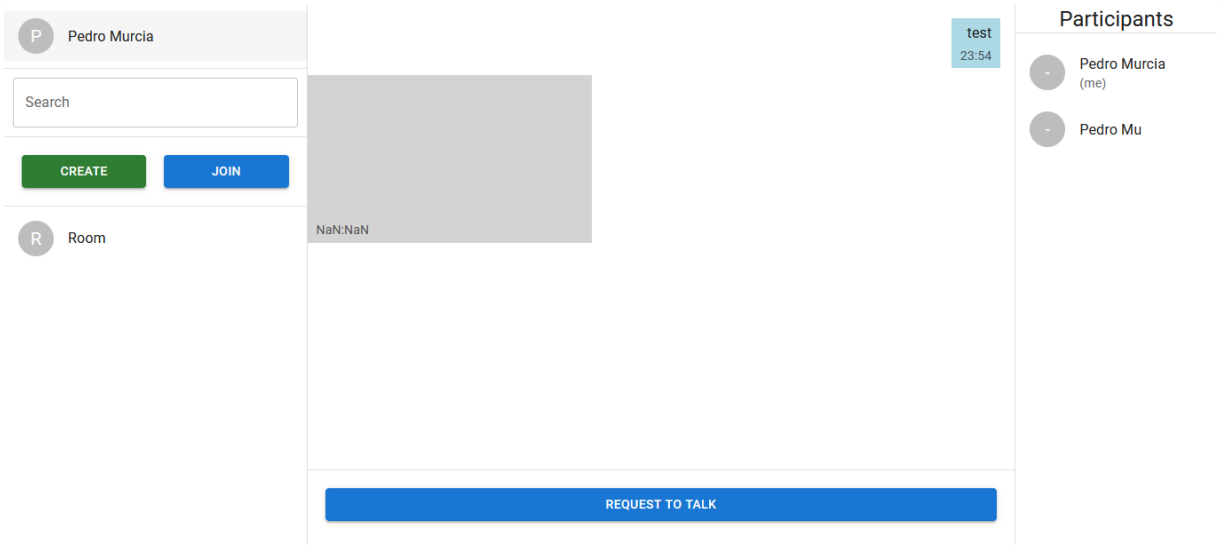
joined, session identifiers and others. This information is accessible through the second Firebase real-time database service, which will be implemented to load the relevant information to establish the session. Finally, a component will be generated that implements calls to the Janus server to manage the WebRTC session and the different rooms.



Ilustration 1. General architecture of the application

4. Results

- An application that correctly sends the information of the local streams to the rest of the participants
- A waiting list that adds participants when they ask to speak and removes them when it's over
- A modern user interface with React and MaterialUI



Ilustration 2. Application sample

5. Conclusions

The application that has been created successfully connects with the Janus Gateway to generate multimedia communications between multiple participants. The establishment of communication is done in turns, so communication is orderly and clear. Both audio and video messages can be sent to facilitate dynamic use of this service. This application can also be transformed into JavaScript code that can be served directly from an HTTP server. The user interface is modern, and the technologies and methodologies used follow the industry standard, which allows for easier maintenance and ways to add new functionality in the future.

6. References

[1] «Made with WebAssembly,» [Online]. Available: https://madewithwebassembly.com/. [Last access: 28 January 2022].

[2] Google Developers, «WebRTC,» [Online]. Available: https://webrtc.org/. [Last access: 1 January 2022].

Índice de la memoria

Índice de figuras

Capítulo 1. INTRODUCCIÓN

En la actualidad, las tecnologías de Internet son cada vez más complejas y ofrecen una funcionalidad que añade más opciones a la comunicación en Internet desde los navegadores. De un tiempo a esta parte, los navegadores más populares incluyen funcionalidad más exigente, como por ejemplo *WebAssembly*, que es una tecnología que permite compilar programas en la web para diferentes lenguajes de programación a bajo nivel, como C o C++ [1]. Sin embargo, gran parte de la innovación ha sido provocada por los servicios de *streaming* o *Video On Demand* (VoD) en Internet, como Netflix, Amazon Prime Video, YouTube o Disney+, por citar unos ejemplos. Desde que HTML5 se convirtió en la tecnología hegemónica de información en Internet, el estándar ha ido incluyendo nuevas características que facilitan el soporte multimedia para vídeos, sin necesidad de conocimiento del formato origen, para generar una capa JavaScript independiente de la capa multimedia [2]. Además, ya que W3C, organización que diseña el estándar HTML5 entre otros, genera estándares que son libres de acceso, la mayoría de los navegadores lo incluyen por defecto y la compatibilidad de uno de estos servicios es mucho más alta.

Toda esta nueva funcionalidad permite generar aplicaciones con un rendimiento más cercano a un programa nativo, por lo que muchos servicios se están pasando a aplicaciones en Internet. El ejemplo más claro es *WebAssembly*, que permite que en Internet se puedan encontrar emuladores de diferentes consolas antiguas que funcionan con JavaScript. Aunque se emplea también para el ocio, otras soluciones comerciales implementan *WebAssembly*, como SketchUp, Unity o AutoCAD Web App [3]. Sin embargo, se ha convertido en una herramienta fundamental de la industria emplear *frameworks* de JavaScript para el desarrollo de aplicaciones tanto nativas como de navegador. Por ejemplo, React Native es un *framework* que permite generar aplicaciones móviles de distintas plataformas, y se trata de una de las tecnologías más populares entre los programadores [4]. En resumen, la distancia entre Internet y el hardware nativo se está reduciendo, por lo que mucha de la funcionalidad se está convirtiendo en código JavaScript que se puede ejecutar en el navegador.

Una de las tecnologías más interesantes desarrolladas para la comunicación entre varios dispositivos es WebRTC, que es un protocolo y una definición de instrucciones mediante los cuales se "puede agregar capacidades de comunicación en tiempo real a [una] aplicación que funciona sobre un estándar abierto" [5]. En otras palabras, el navegador facilita la creación de una sesión multimedia en tiempo real mediante API estandarizadas por W3C y los diferentes protocolos que se emplean en la comunicación. Existen varias soluciones que aplican esta tecnología, como Houseparty, Discord, Google Meet o Microsoft Teams, que son aplicaciones que ofrecen servicios de comunicación en tiempo real desde un dispositivo a otro. Estos servicios son similares a varios de los servicios ya existentes en el mundo de las telecomunicaciones, y son más interesantes por su disponibilidad casi inmediata.

Uno de estos servicios es el servicio de *walkie-talkie*, conocido como *Push To Talk* o P2T. Se trata de un sistema sencillo de implementar, con bastantes aplicaciones en la actualidad, como emergencias o comunicaciones rápidas y cortas entre varios puntos de una estructura industrial. El estándar que se emplea actualmente es PoC, de las siglas *P2T over Cellular*, que implementa el flujo de una comunicación *half-duplex* (o un único canal de envío o recepción de información) en las redes celulares de los proveedores de servicios de telecomunicaciones. La mayoría de los servicios de P2T se aplican sobre una red celular, ya sea pública o privada como TETRA de los servicios de emergencias, pero en menor medida se aplican en Internet.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

En esta sección, se procederá a explicar las tecnologías que se van a usar en este proyecto. Por un lado, tenemos el instrumento que nos permitirá realizar las comunicaciones multimedia a través de Internet, que tiene el nombre de WebRTC. Por otro lado, se explicará la naturaleza de la aplicación que se desarrollará, que tendrá una forma de *Push To Talk*, o P2T.

2.1 WEBRTC

Web Real Time Communications, o WebRTC, es una de las tecnologías que permite generar comunicaciones multimedia entre navegadores. Según la especificación de W3C, "[it] defines a set of ECMAScript APIs in WebIDL to allow media and generic application data to be sent to and received from another browser or device implementing the appropriate set of real-time protocols" [6]. En otras palabras, permite realizar las operaciones de comunicación multimedia a través de funciones predefinidas en ECMAScript. Las opciones que ofrece WebRTC a través de sus API son varias, que incluyen la conectividad entre participantes utilizando ICE para atravesar los problemas que producen los NAT en las comunicaciones, además de la capacidad de enviar datos generales y de vídeo propio del dispositivo [6]. Por lo tanto, es un estándar para poder realizar todo el proceso de comunicación multimedia en el navegador.

Por un lado, el estándar conjunto de WebRTC define el protocolo que se debe utilizar para implementar esta tecnología. Esta parte del estándar está desarrollada por IETF en el RFC 8825, y sirve como complemento a la API desarrollada por W3C. Este estándar define los diferentes niveles de la comunicación entre participantes, y los requisitos para cada uno de los niveles. En primer lugar, la capa de *Data Transport*, que se refiere a "the sending and receiving of data over the network interfaces" [7], propone utilizar los protocolos TCP y UDP compatibles con IPv4 e IPv6 [8]. Después, la capa de *Data Framing*, o transporte del

contenido multimedia, requiere el uso de los protocolos RTP y SRTP, siendo este último la versión más segura para las comunicaciones. A continuación, el formato de los datos transmitidos requiere que se utilicen los códecs Opus, PCMA y PCMU para audio [9], y los códecs obligatorios de vídeo son VP8 y H.264 Constrained Baseline [10]. Para la gestión de la conexión, se debe utilizar la misma semántica que en SIP para los SDP, que además permita la comunicación con dispositivos *legacy* de SIP [7]. Por último, se enlaza el estándar con el documento complementario de W3C para la presentación y el control de los usuarios sobre la comunicación WebRTC, que está definido en el RFC 8829 [11].

Por otro lado, IETF y W3C definen el comportamiento para la gestión de la presentación y control de la comunicación. El funcionamiento base de una comunicación WebRTC entre dos navegadores es simétrico, y se puede realizar a partir de los diferentes elementos que implementa el estándar. La comunicación se puede dividir en tres capas diferentes, donde una capa de comunicación es para la señalización, otra capa es la de los servidores STUN/TURN y la última capa es la que transmite la información. La capa de señalización sirve para gestionar la comunicación, y tiene un comportamiento similar a la misma capa en el protocolo SIP, y puede ser un canal de WebSockets o HTTP, entre otros [6]. En el Anexo A se puede ver un ejemplo de comunicación entre los diferentes navegadores, que se explicará con más detalle a continuación.

Para empezar una comunicación multimedia, uno de los participantes debe iniciar el proceso a través de la creación de un `RTCPeerConnection`, que es una instancia que permite a una aplicación establecer una comunicación *peer-to-peer* con otro `RTCPeerConnection` u otro *endpoint* que implemente los protocolos [6]. La aplicación crea esta conexión para que el navegador pueda registrarlo, y en la creación manda información sobre el protocolo ICE. Este nuevo `RTCPeerConnection` tiene un estado de señalización, un estado de conexión, un estado de recolección ICE y un estado de conexión ICE [6], por lo que este nuevo objeto es el centro de todas las comunicaciones entre los diferentes participantes. Una vez tenemos este objeto, el siguiente paso es preparar el SDP, o *Session Description Protocol*, que es la información sobre la sesión multimedia. Para ello, se manda el SDP con los *tracks* que se van a enviar por el canal de señalización, similar a cómo se haría en SIP.

En este punto, cabe recalcar una de las características que implementa WebRTC. Junto al desarrollo de WebRTC, se está trabajando para definir el estándar de Media Capture and Streams. Este estándar sirve para pedir datos multimedia locales, tanto audio como vídeo, al navegador a través de APIs JavaScript [12]. El fundamento de este estándar es solicitar la información fácilmente, tal que se especifique en un objeto JavaScript la información que se solicita y se llame a la función `navigator.mediaDevices.getUserMedia`. El resultado de esta petición es un *stream* con toda la información solicitada divididas en *tracks*. Por lo tanto, a través de esta función, se pueden obtener los *tracks* que se pueden añadir al SDP de la conexión WebRTC. Este stream del navegador no es el único que se puede meter, pero es el caso más común debido a las necesidades de las diferentes aplicaciones de comunicación en tiempo real, como videollamadas o chats en general.

Una vez se ha añadido correctamente los *tracks* de información, ya se puede crear el SDP para mandarlo al participante remoto. Se llama a la función `createOffer`, y se ajusta el descriptor de sesión local a este SDP. Es en el momento en que se llama a la función `setLocalDescription` cuando el navegador inicia el proceso ICE con la comunicación con el servidor TURN/STUN. Después, se manda por el canal de señalización el SDP al resto de participantes.

En el caso de los receptores de los SDP, o en otras palabras los invitados a la llamada reciben esta información a través del canal de señalización. A través de un evento o cualquier operación similar, en el momento en el que se recibe la oferta SDP se crea el objeto `RTCPeerConnection` en el navegador y se ajusta el SDP entrante como el descriptor remoto de sesión a través de la función `setRemoteDescription`. Una vez realizada esta configuración, el navegador receptor recibe la información del SDP, que incluye los *tracks* empleados en la comunicación. De estos *tracks* se puede obtener la información de los *streams* de datos que se pueden utilizar en la aplicación final. Al final de la operación, se realiza un proceso similar a la creación de SDP de oferta, pero en este caso se denomina como uno de respuesta, y se crea con `createAnswer`. Una vez lo ha recibido el iniciador de la comunicación, y el SDP de oferta se ajusta como SDP remoto, la negociación de

candidatos ICE empieza, y cuando este proceso ha acabado, se transmiten los datos a través de Internet.

El ejemplo mostrado antes funciona correctamente si se trata de una comunicación de uno a uno, pero se puede convertir en un problema cuando se trata de varios participantes. Como el estándar define una comunicación de uno a uno, una comunicación entre varios implicaría multiplicar los recursos para establecer una comunicación mutua con todos los participantes. No sólo eso, sino que se define el estándar para una comunicación directa entre dos navegadores, por lo que algunos dispositivos SIP no funcionarían con WebRTC. Sin embargo, el grupo de trabajo del IETF define un nuevo tipo de entidad llamado *WebRTC Gateway*, que se definen como "a WebRTC-compatible endpoint that mediates media traffic to non-WebRTC entities" [7]. Para los dispositivos WebRTC, este *endpoint* también se comporta como un cliente WebRTC, por lo que tanto dispositivos WebRTC como *legacy* se pueden comunicar con él. Este *gateway* se puede programar para realizar las funciones de un servidor multimedia mientras se comunica uno a uno con los clientes WebRTC conectados a él.

2.2 PUSH TO TALK

En el contexto de las comunicaciones multimedia, los servicios *Push To Talk* (P2T) son aquellos servicios que tienen un funcionamiento similar a un *walkie-talkie*. En otras palabras, son aquellas comunicaciones en las que sólo uno de los participantes puede emitir en un momento determinado, mientras que el resto de los participantes reciben el mensaje. La OMA ha desarrollado un estándar para las aplicaciones PoC, o *Push To Talk over Cellular*, para los proveedores de servicios [13, p. 11]. Este servicio está inherentemente orientado a los datos, por lo que también se ofrecen servicios de mensajería, como creación de listas, presencia y disponibilidad, entre otros [13, p. 11].

Para explicar cómo funciona un servicio P2T, el documento de la OMA define diferentes casos de uso, y el caso "Shopping like crazy" [13, pp. 13-16] muestra un caso genérico de

flujo de aplicación. Para que funcione el sistema, es necesario que haya tanto participantes como un proveedor del servicio, que se encargue de realizar las gestiones multimedia.

En primer lugar, uno de los participantes debe mandar una petición al proveedor para crear un grupo PoC. Este participante se comportará como el administrador del grupo y se le otorgarán dichos derechos cuando el proveedor crea la sala. Cuando se ha creado la sala, se manda una invitación a cada uno de los participantes invitados. Sin embargo, el proceso cambia cuando se quiere unir un nuevo participante a la sala que no estaba invitado desde el comienzo. Dado que al comienzo el rol de PoC Host lo tenía el proveedor, y este rol se transfiere al participante administrador, los nuevos participantes deben realizar la petición a este administrador. A continuación, se puede ver un flujo de cómo funcionaría esta primera sección, tomado del documento de la OMA [13, p. 15].

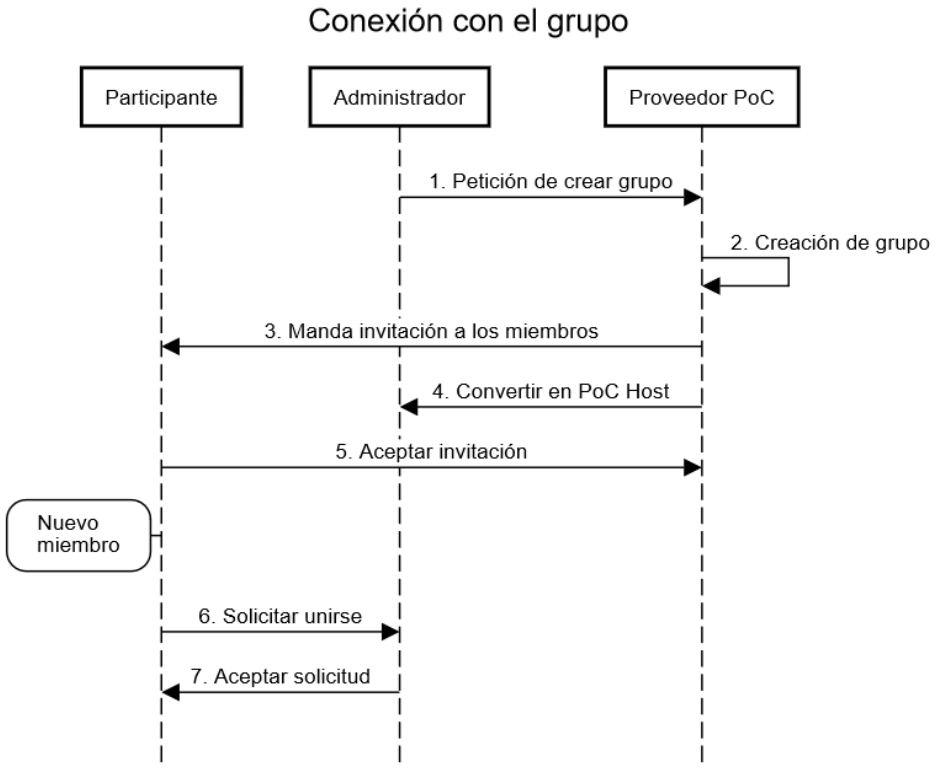


Figura 1. Diagrama de conexión a una sala PoC (traducido de [13, p. 15])

Una vez los participantes se encuentran en el grupo, cada uno de los miembros se comunica con el proveedor para realizar los diferentes procesos. Cuando alguien quiere hablar, se

manda una petición al proveedor para que se añada al usuario a la cola de espera. Si no hay nadie hablando, el turno se da al usuario que esté primero en esta lista. En caso de que no sea el primero en la lista, se ha de esperar a que el usuario anterior termine de hablar, que permite al usuario siguiente cancelar la petición. Cuando el usuario que está al mando de la comunicación está hablando, esa información se transmite a los participantes, que estarán escuchando. Si sucede cualquier evento externo a esta comunicación, el proveedor mandará eventos al resto de participantes para notificarles de su estado. Sin embargo, en casos como el usuario recibiendo llamadas externas, el propio dispositivo se encargará de cambiar el modo de escucha. El gráfico a continuación muestra un flujo de aplicación, y se puede ver en el documento de la OMA [13, p. 16].

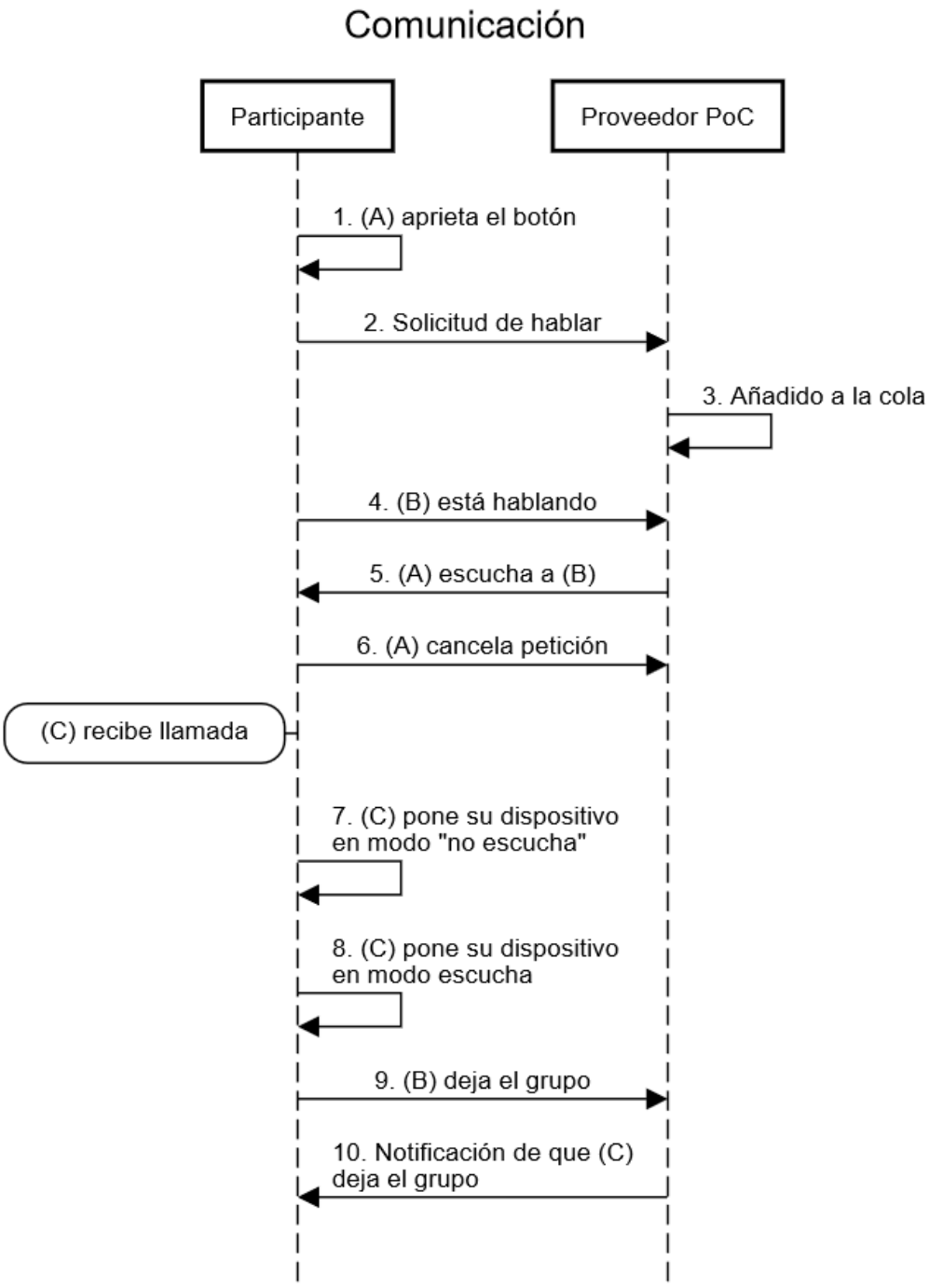


Figura 2. Diagrama de comunicación en PoC (traducido de [13, p. 16])

Por lo tanto, P2T se trata de un sistema sencillo de comunicación, en la que sólo se puede transmitir la información de uno en uno (conocido como *half-duplex*), que tiene un comportamiento similar al de un *walkie-talkie*. Los participantes solicitan el turno de palabra

pidiéndole al proveedor que se les añada en la lista de espera, y el servidor provee del turno en función de esa lista.

Capítulo 3. ESTADO DE LA CUESTIÓN

En este capítulo, se estudiarán las diferentes alternativas que implementan comunicaciones multimedia, WebRTC o P2T en su servicio, para analizar las posibilidades de un servicio de estas características en el mundo actual.

En el mercado ya hay varias aplicaciones que implementan WebRTC para la comunicación en tiempo real. Por un lado, ya que Discord está programado para soportar varias plataformas, comentan que "the only way our team can support all these platforms is to take advantage of […] WebRTC" [14]. Por concretar, "Discord's audio and video features are implemented using WebRTC" [14]. Por otro lado, Houseparty emplea testRTC para su sistema WebRTC [15], mientras que Snapchat compró un servicio PaaS de WebRTC en 2014 [16]. Por último, Google es el principal sustentador de WebRTC, y muchos de los servicios de comunicación de Google emplean WebRTC, como Google Meet o Google Hangouts.

No obstante, ya existen en el mundo actual servicios P2T y PoC estables y de uso muy común. Los servicios de emergencias, la policía y los bomberos emplean P2T para la comunicación entre los diferentes operativos mediante la red TETRA, que "permite utilizar varios canales a demanda, tanto de transmisión de coz como de datos" [17]. Además, las aplicaciones P2T son más eficientes para el tipo de comunicación que se requiere, ya que por su sencillez de diseño ocupan menos ancho de banda, es más rápido el establecimiento de la comunicación y se puede utilizar para comunicaciones entre pocas personas. Dicho de otra manera, cuando se necesita una comunicación veloz sin interferencias entre los participantes para una comunicación clara, los servicios P2T son los más adecuados. Otros lugares donde se puede emplear esta tecnología son en recintos feriales, oficinas o eventos en general, donde la coordinación de los miembros de la seguridad es muy importante.

Para una solución moderna P2T, podemos encontrar aplicaciones que lo implementan a nivel industrial como Zello. Zello es un servicio que utiliza un protocolo propio de comunicación

a través Internet con WebSockets, y otro tipo de canales, aunque WebRTC no está implementado en su servicio. Según la página oficial, Zello es intuitvo de usar, veloz en las comunicaciones, además de seguro y compatible con varias plataformas y otros accesorios [18].

Por último, existe una aplicación P2T que emplea WebRTC en su servicio. Wazo Platform es un proyecto open-source que permite la creación de una aplicación P2T con su API [19]. Al contrario que Zello, se trata de una aplicación de mucho menor tamaño y de un uso más general, pero implementa la funcionalidad que permite realizar una aplicación P2T a través de WebRTC. Según la página oficial, "based on core Open Source components, Wazo Platform aims to provide all the building blocks to create a full-featured, carrier grade, Programmable Telecom Infrastructure" [20].

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 JUSTIFICACIÓN

El objetivo de este trabajo es desarrollar una aplicación P2T que funcione con los servicios que ofrecen los navegadores de Internet, como WebRTC y JavaScript. Al emplear WebRTC y JavaScript, el conjunto de la aplicación y la funcionalidad se pueden encapsular en el navegador, por lo que la aplicación se puede servir con un servidor sencillo que reduce los gastos operativos al mantenimiento de los servidores y los servicios de WebRTC.

Por un lado, permite adaptar las soluciones P2T a las nuevas tecnologías de Internet, además de permitir que las innovaciones que van surgiendo en el desarrollo web se puedan implementar en la aplicación para complementar, mejorar o actualizar el servicio final de una forma más sencilla.

Por otro lado, gracias a que WebRTC tiene una estructura similar a SIP, WebRTC permite la conexión con dispositivos más antiguos o *legacy* que implementan este protocolo. Esto permite la posibilidad de interconectividad entre estos dispositivos *legacy* con las nuevas tendencias de desarrollo web. En otras palabras, esta aplicación ofrece la posibilidad de trabajar directamente desde Internet o redes más antiguas, en una forma similar a los teléfonos fijos y los teléfonos VoIP.

4.2 OBJETIVOS

- Ofrecer un servicio moderno de *walkie talkie*

En primer lugar, la aplicación deberá seguir los estándares actuales de la industria de la programación web. Aunque las tecnologías de Internet no dejan de aparecer, existen comunidades de programadores que mantienen y trabajan para que ciertas herramientas funcionen correctamente. Sin embargo, se debe basar en una experiencia de usuario sencilla

y eficaz, que implique una comunicación casi instantánea con la aplicación. No sólo eso, sino que la aplicación debe ser muy interactiva para que la sensación del usuario sea buena en cuanto a los tiempos de espera y de carga. En resumen, fácil de usar y rápida como una aplicación nativa.

- Mostrar la capacidad de WebRTC para adaptar tecnologías clásicas

La "Ley de Atwood" dicta que "any application that *can* be written in JavaScript, *will* eventually be written in JavaScript" [19]. Jeff Atwood, creador del foro de ayuda StackOverflow, dio nombre a esta ley que parece cumplir con su premisa. La API que ofrece el estándar de WebRTC es un conjunto de funciones JavaScript para establecer una comunicación multimedia desde un dispositivo o navegador a otro. En otras palabras, la "Ley de Atwood" se está cumpliendo en las comunicaciones multimedia, ya que se pueden escribir en JavaScript y muy probablemente acabarán estando escritas en JavaScript. Debido a que las comunicaciones se pueden realizar a través de JavaScript, se puede replicar la estructura de comunicaciones P2T por una versión software en JavaScript.

- Ofrecer diferentes modos de comunicación

Por último, WebRTC permite la comunicación en tiempo real de datos, vídeo y audio gracias a una especificación de la API. Sin embargo, existen diferentes tipos de comunicaciones en función de los usuarios que lo usen. En esta aplicación el objetivo es ofrecer a los usuarios varias alternativas de mandar información, desde datos hasta audio y vídeo completo. De esta manera, los usuarios tienen más flexibilidad en la manera de usar estos servicios.

4.3 METODOLOGÍA

En el desarrollo de esta aplicación se trabajará con un *framework* JavaScript que permita la interactividad de la aplicación con las comunicaciones en tiempo real que se están realizando. Dado que la tecnología más popular para el desarrollo web de interfaz de usuario es React, ese será el *framework* empleado. Mientras se realiza el desarrollo en React, se

alojará en un contenedor Docker el *gateway* WebRTC. El objetivo de generar un contenedor donde desplegar el *gateway* es para que sea más homogéneo y no genere errores.

Con el interfaz de usuario y el *Gateway* WebRTC, el siguiente paso es incluir en la aplicación los servicios de autenticación de usuarios y base de datos. Estos servicios servirán para gestionar la información de WebRTC, de los usuarios y de la aplicación. Después de añadir estos servicios, se deberá añadir la conexión con el *gateway* WebRTC para realizar la comunicación multimedia

El último paso es programar los flujos de aplicación, como crear de salas, unirse a salas o mandar mensajes o audio y vídeo. Sobre todo, es imperativo programar la lista de espera para la comunicación, que dependa únicamente de los participantes de la sala y el momento en el que se solicita la palabra. En otras palabras, la lista debe contener el usuario y el momento en el que se añadió a la lista, y el usuario que tendrá la palabra será el primero de la lista por orden de inclusión a la lista. Por último, se implementará la capacidad de mandar diferentes tipos de *streams* a los participantes en función de los intereses del emisor.

Capítulo 5. SISTEMA/MODELO DESARROLLADO

En este capítulo se procederá a explicar la aplicación desarrollada, que constará de diferentes partes. En primer lugar, se explicarán los servicios utilizados para el desarrollo de la aplicación, seguido del diseño de la interfaz de usuario. A continuación, se mostrará el modelo de datos empleado en la aplicación, y por último se explicarán diferentes flujos de la aplicación.

5.1 SERVICIOS

En esta sección, se procederá a explicar los servicios que se utilizarán para esta aplicación. Debido a que esta aplicación necesita ciertas características para funcionar, es necesario explicar las herramientas externas seleccionadas y la motivación detrás de su elección. De esta manera, se busca que la selección pueda adecuarse lo máximo posible a la funcionalidad deseada de la aplicación.

En primer lugar, se empleará el servicio de Firebase para dos funciones diferentes. Firebase es un servicio de Google Cloud que proporciona un acceso más sencillo a las herramientas de Google Cloud y su gestión. Se presenta como una ayuda a la programación, e incluye las herramientas más comunes para una página web, como el servicio de *Hosting* de contenido estático o funciones de machine Learning. Sin embargo, para esta aplicación sólo se emplearán el servicio de Autenticación de usuarios y el de Base de Datos en tiempo Real. Por un lado, el servicio de autenticación facilita la gestión de usuarios en la aplicación, y servirá para poder conectar las sesiones de WebRTC con su contraparte en la cuenta de Google u otros. Por ejemplo, en caso de guardar datos sobre un cierto usuario, se puede comprobar rápidamente la información de usuario en el resto de la aplicación.

El segundo servicio de Firebase que se va a emplear es el de la base de datos en tiempo real. Esta es una base de datos NoSQL en formato JSON, que permite guardar la información de una forma muy sencilla para el lenguaje de programación que se estará empleando en el

proyecto. Sin embargo, la principal facilidad de esta base de datos es que, con el SDK de Firebase, los cambios detectados en la base de datos remotas se pueden observar desde la aplicación, por lo que hace más sencilla la comunicación de los cambios desde la base de datos a la aplicación cliente. Es en esta base de datos donde, además de guardar la información de los usuarios, se guardará también la información de la lista de participación, las salas y los participantes, entre muchos otros.

Una vez vista la parte de infraestructura básica para una aplicación, lo siguiente es introducir el servicio de WebRTC para realizar las comunicaciones entre los diferentes participantes. En primer lugar, se utilizará el servidor Janus como nexo WebRTC. Janus es, según el informe de Amirante et al. [20], "a general purpose, open source WebRTC gateway" específico para sistemas Linux. Este servidor tiene una estructura modular, por lo que la funcionalidad deseada se ha de agregar a la conexión con el servidor, junto con las funciones que se deben realizar cuando suceden ciertos eventos. Para realizar esto, Janus ofrece una API para poder implementar estos procesos de una forma más sencilla, en forma de código JavaScript que se puede importar al framework React. Este servidor ofrece facilidades a la implementación de WebRTC para la comunicación en tiempo real, por lo que reduce la cantidad de errores posibles por parte de la programación propia de la aplicación.

Para implementar estos servicios, se empleará una característica específica del *framework* React. En React, existe la funcionalidad de *Context*, que permite la comunicación de datos a niveles inferiores de la aplicación en una relación Proveedor/Consumidor, o una relación de un elemento emisor de información y un elemento receptor de esos datos. Esto es, se generarán tres componentes proveedores de información que se situarán en la parte más alta de la jerarquía de la aplicación, anidados en el orden en el que se ha explicado al comienzo de esta sección. Cuando los tres servicios se hayan cargado correctamente, la aplicación podrá funcionar debidamente, por lo que se reducen los errores posibles en tiempo de ejecución de la aplicación en sí por los fallos de los servicios implementados.

5.2 DISEÑO DE INTERFAZ DE USUARIO

Para el desarrollo del interfaz de usuario, se empleará el *framework* React. Según la página oficial de la herramienta, React es "una biblioteca de JavaScript para construir interfaces de usuario" [21] desarrollada originalmente por Facebook y más tarde liberada para la comunidad *open-source*. Se trata de uno de los *frameworks* más populares en la actualidad, y uno de los frameworks mejor valorados en JavaScript, como muestran la encuesta de Stack Overflow [4]. Una de las ventajas de ser tan popular es que la comunidad de desarrolladores que trabajan en mejorar React es muy numerosa y activa, por lo que no faltan recursos para obtener la funcionalidad necesaria.

En principio, React es un *framework* sencillo de base, cuyo principal cometido es que el programador diseñe las vistas para la aplicación, mientras que "React se encargará de actualizar y renderizar de manera eficiente los componentes correctos cuando los datos cambien" [21]. Para ello, la programación en React se basa en "componentes encapsulados que manejen su propio estado" [21]. En otras palabras, la programación en React se basa en pequeños módulos del programa que son capaces de guardar la información acerca del estado en el que se encuentran, y cuando sucede un cambio en el estado o en los datos que se mandan desde un elemento superior (propiedades) automáticamente se regenera el contenido para ajustarse a estos nuevos datos. La principal ventaja de este modelo de programación es que los cambios de datos en tiempo real pueden producir un cambio muy rápido en la aplicación, por lo que la comunicación y la actualización de los datos en cada usuario es mucho más eficaz.

El estilo temático de la aplicación se basa en los fundamentos del diseño *Material*. Según la página oficial, "Material Design is an adaptable system of guidelines, components, and tools that support the best practices of user interface design." [22]. En otras palabras, se trata de un estándar desarrollado por Google para sus servicios, aplicaciones y, principalmente, dispositivos Android. Debido a los últimos, es el estándar más reconocible para desarrollar interfaces de usuarios en Internet salvo para los dispositivos de las marcas de Apple. Por lo tanto, existen diferentes aplicaciones de estos estándares en Internet. Sin embargo, debido a

la gran comunidad detrás del desarrollo de React, este *framework* dispone de una librería que introduce componentes y elementos de Material llamada *Material UI* (o MUI). Gracias a esta librería, introducir nuevos componentes que siguen el estándar Material es más sencillo, sin necesidad de sacrificar en personalización del estilo de estos elementos. Es decir, esta librería proporciona estilos por defecto, sin necesidad de eliminar la capacidad del desarrollador de modificar la visualización de estos elementos.

El diseño visual de esta aplicación se puede dividir en tres secciones con funcionalidad diferente, y el código que lo muestra se puede ver en el Anexo B. El primer cuarto del ancho de la página está cubierto por el menú de la aplicación, en una configuración vertical. Este menú consta de diferentes secciones para opciones diferentes. Primero, podemos encontrar el perfil del usuario que se ha conectado con su cuenta de Google. Después tenemos un buscador para los chats que el usuario tiene activos, junto con dos botones de acción: uno sirve para crear una nueva sala, mientras que el otro sirve para unirse a una sala existente. Por último, tenemos la lista de grupos a los que pertenece el usuario que permite acceder a los mensajes de la sala específica. Se emplean `Grid` y `List`, el primero por las capacidades de escalado que ofrece y el segundo por la capacidad de ordenar los diferentes `ListItem` dependiendo de un *key*.

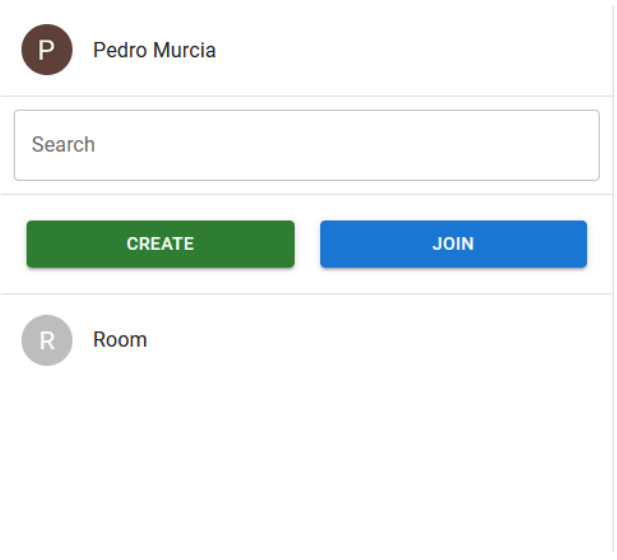


Figura 3. Muestra de la barra derecha en la aplicación

En la zona media de la página, tenemos la sección donde se muestran todos los mensajes de la sala seleccionada en la parte anterior. A medida que se reciben nuevos mensajes de la sala, éstos se muestran en esta sección, y el último mensaje aparece en la parte más baja. Debajo de la sección de mensajes, tenemos una zona que tiene triple utilidad. En primer lugar, si el usuario no está en la lista de participación, habrá un botón para solicitar el turno en la lista. Mientras que el usuario esté en la lista, pero no tenga el turno para hablar, la zona en cuestión mostrará un botón que mostrará el estado de espera al turno, que al volver al pulsarse eliminará al usuario de la lista de participación. Por último, cuando el usuario ha recibido el permiso para hablar, se mostrará los inputs pertinentes para realizar la comunicación. A continuación

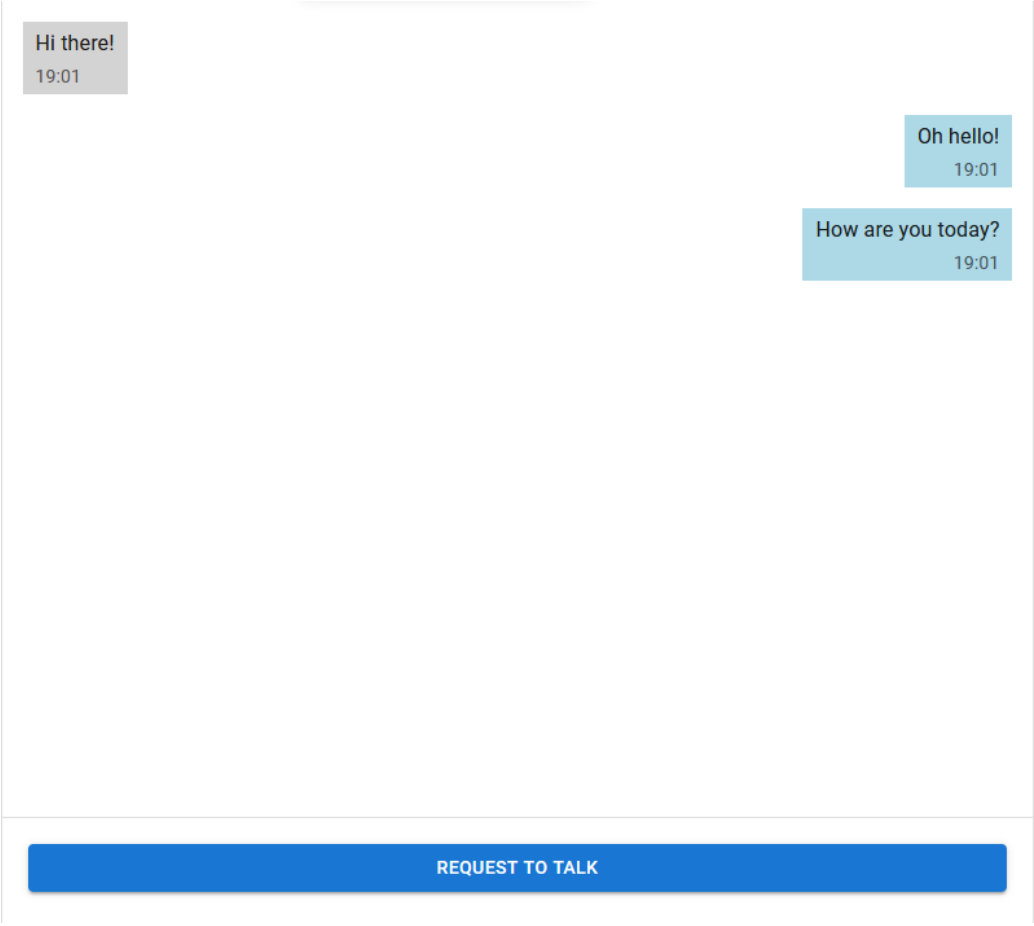


Figura 4. Muestra de la zona de mensajes

Figura 5. Muestra de la barra de envío

A la izquierda de este último elemento, nos encontramos la lista de participación. En este componente aparecerá la lista de participantes de la sala, que además de mostrar los usuarios participantes, tiene también la función de mostrar la lista de participación. Para ello, al lado del nombre de cada participante, se pone un pequeño avatar para mostrar la posición de cada uno de los usuarios en la lista. Esta lista se actualiza automáticamente cuando se detecta un cambio en la lista, y aquellos participantes que no hayan pedido turno para hablar aparecerán al final de la lista con un guion en vez de la posición. Para que sea más fácil distinguir entre los que han pedido turno y los que no, los que están en la lista tendrán un color de fondo más llamativo en la posición de la lista.

Figura 6. Muestra de la lista de participantes en una sala

5.3 MODELO DE DATOS

En esta sección, se explicará la información que se está empleando y cómo está estructurada para producir una aplicación funcional como la que tenemos. En la aplicación desarrollada, tenemos datos en tiempo real como datos clásicos de estado. Por un lado, los datos en tiempo real son aquellos que vienen de la comunicación WebRTC entre participantes de una sala, como los mensajes de la sala o los cambios en el estado de la sala y sus participantes. Por

otro lado, tenemos la información de las salas a las que pertenece el usuario, e información más general sobre la aplicación. Mientras que los primeros se están actualizando a menudo, los últimos no requieren una demanda intensiva de recursos. Por lo tanto, los datos de aplicación generales se guardarán tanto en la base de datos remota como en el navegador, a través de la utilidad `localStorage`.

En primer lugar, tenemos el componente de mensaje, que es consumidor de los datos de los proveedores de WebRTC y autenticación. Los datos que se necesitan en este componente son el usuario conectado con Firebase y los participantes de la sala. Además, se le pasan tres datos más referidos al autor del mensaje, la fecha de publicación del mensaje y el contenido del mensaje. Con esta información, se puede mostrar el mensaje como un mensaje remoto o propio, en un grupo con más participantes o menos. Dentro de este componente, los valores que nos ayuda a formatear el mensaje de forma correcta son `ownMessage` y `remoteGroupMessage`. Este componente es final, ya que solo muestra datos.

Estos componentes de mensaje se generan cuando el componente ChatsLayout recibe una actualización en los mensajes. En este componente específico, se consumen los datos del servicio de WebRTC, autenticación y base de datos. De WebRTC, se obtiene toda la información de WebRTC, como la sesión actual, la sala actual y los participantes, además de la funcionalidad de la sesión WebRTC, como la función para mandar información. Sin embargo, este componente tiene variables de estado específicos para el funcionamiento, para que el funcionamiento de la aplicación sea lo más correcta posible. En este componente, la variable de estado más importante es la de `canTalk`, que realiza cambios en el interfaz de usuario en función del estado del participante en la lista de espera para hablar.

Con los componentes de interfaz explicados, los más importantes para explicar el modelo de datos se encuentran en los proveedores explicados en el apartado 3.1. Como son proveedores de datos, mucha de esta información se encuentra como estado en cada uno de los componentes. En primer lugar, están los servicios de Firebase. Sin embargo, debido a cómo está definido el SDK, se ha optado por crear un archivo común que realice la inicialización del SDK de Firebase y que los dos componentes proveedores de datos accedan a esa misma

información. Este fichero común realiza la inicialización a partir de variables de entorno proporcionadas por el proyecto Firebase, y se exporta la aplicación Firebase junto con otras variables y funciones de Firebase para que se puedan utilizar en el resto de los componentes.

Por un lado, el servicio de autenticación de Firebase tiene pocas variables de estado, pero la más importante es la de `user`, que es la que contiene la información del usuario que ha entrado a la aplicación a través del servicio de Firebase. Otras variables de estado se refieren al proveedor de Google para generar el login, y una variable que activa el resto de la aplicación cuando el usuario se ha conectado correctamente. Este componente provee únicamente el valor de `user`.

```
<AuthContext.Provider value={{ user, signInWithGoogle }}>
  {children}
</AuthContext.Provider>
```

Por otro lado, el servicio de la base de datos es algo más complejo, pero la base es la misma que con la autenticación. En este componente, las variables de estado se refieren a la información de la base de datos, como el objeto de la base de datos en sí, como referencias a los elementos de salas y usuarios de la aplicación (`roomsRef` y `usersRef`) que sirven para registrar funciones de escucha de eventos en la base de datos. Al igual que en el componente anterior, se guarda una variable para activar el resto de la aplicación cuando la base de datos está conectada con la aplicación. Este componente además de leer el usuario también guarda la información de todas las salas (`roomsInfo`) para poder leerla después en el componente principal de la aplicación o el proveedor de WebRTC. Este componente exporta la base de datos, `roomsInfo`, además de funciones para trabajar en la base de datos u obtener el estado a partir de la información de la base de datos, como la funcionalidad de estar en la cola o ser el turno.

```
<DatabaseContext.Provider
  value={{
    database,
    readData,
    writeData,
    removeFromQueue,
    addToQueue,
```

```
    isOnQueue,
    isMyTurn,
    roomsInfo,
    appendMessage,
    addUserRoom,
    addToRoomParticipants,
    usersInfo,
    addRoomToList,
    removeRoomFromList,
    saveUserVideoRoomId,
  }}
>
  {children}
</DatabaseContext.Provider>
```

En último lugar, el proveedor de WebRTC hace uso de los anteriores proveedores para realizar su función. Este componente guarda información sobre la sesión WebRTC del usuario, como los participantes, los mensajes, la sala conectada, el plugin utilizado y la sesión actual. La anterior información es la que se provee al resto de la aplicación para poder realizar su función, aunque el componente guarda más datos para poder realizar más funciones.

```
<JanusContext.Provider
  value={{
    currentSession,
    sendData,
    currentRoom,
    transactions,
    participants,
    messages,
    joinVideoRoom,
    createVideoRoom,
    publishOwnFeed,
    unpublishOwnFeed,
    remoteStream,
    localTracks,
    localStream,
    toggleAudio,
    toggleVideo,
    audioOn,
    videoOn,
  }}
>
  {children}
</JanusContext.Provider>
```

Debido a que la base de datos utilizada tiene un formato de NoSQL, esto es, no estructurado en tablas como MySQL o PostgreSQL, el modelo de datos se tiene que adaptar al formato base de la base de datos de Firebase. Como en esta base de datos se almacena en JSON, trabajar con la base de datos desde el programa es más sencillo. En la raíz de la base de datos tenemos dos grupos de información. Por un lado, tenemos *users*, que guardará la información de los usuarios en la aplicación, tales como las salas a las que se ha unido o información sobre el ID en la sesión. En otras palabras, este grupo contiene la información de usuario tanto para el propio usuario en su inicialización como para otros usuarios.

Por otro lado, tenemos el grupo para la gestión de las salas, llamada *rooms*. Este grupo contiene la información de las salas de la aplicación, y contiene cada uno de los códigos de las salas para el acceso del objeto pertinente. Cada sala contiene la siguiente información como valor del código de sala: mensajes, participantes y lista de espera. Primero, los mensajes que se mandan por el proveedor de WebRTC se almacenan en la base de datos para poder ser descargados en cualquier momento. Por ejemplo, cuando un usuario se una, se descarga la lista con todos los mensajes ordenados cronológicamente y los muestra en la aplicación. Después, tenemos tanto los participantes del grupo como los participantes activos. En este punto, los usuarios pueden ver la lista de los miembros del grupo para acceder a la información pública del usuario, o solicitar un grupo privado con este usuario. Por último, como se comentará en la siguiente sección, la lista de espera contendrá como claves de objeto unos valores dependientes del momento de creación, y el valor corresponderá con el UID del usuario. De esta manera, los datos se ordenan cronológicamente y el identificador es único. A continuación, una muestra de lo que se guarda en la base de datos.

```
{
  "rooms" : {
    "<ROOM ID>" : {
      "description" : "Room",
      "messages" : {
        "<TIMESTAMP>" : {
          "date" : "<TIMESTAMP>",
          "from" : "<USER UID>",
          "room" : "<ROOM ID>",
          "text" : "test",
```

```
          "textroom" : "message",
          "transaction" : "<TRANSACTION ID>",
          "whisper" : false
        }
        // ...
      },
      "participants" : {
        "<USER UID>" : {
          "display" : "<DISPLAY NAME>",
          "videoRoomId" : "<VIDEO ROOM ID>",
          "videoRoomPrivateId" : "<VIDEO ROOM PRIVATE ID>"
        }
        // ...
      }
    }
  },
  "users" : {
    "<USER UID>" : {
      "rooms" : {
        "<TIMESTAMP>" : "<ROOM ID>"
      },
      "videoRoomId" : "<VIDEO ROOM ID>"
    }
    // ...
  }
}
```

5.4 *FLUJOS DE APLICACIÓN*

Una vez definidos los elementos que forman la aplicación, el último paso es explicar la conexión entre los diferentes componentes para hacer un servicio funcional. Para ello, se explicará el funcionamiento de cada componente desde el elemento raíz, para comprobar el flujo de la información y el comportamiento de la aplicación.

Cuando la aplicación se inicia, el primer elemento que se activa es el proveedor de autenticación de Firebase. Este proveedor genera un observador de eventos específico para este servicio tal que, si el usuario se ha registrado correctamente, el atributo `showChild` se pone en verdadero, para que el React realice la actualización del interfaz de usuario, generando el resto de aplicación. Si no se ha registrado, aparece en la pantalla un botón para mostrar una ventana emergente donde poder realizar el *login* a través de Google, donde después de un registro correcto se muestran el resto de los componentes. A continuación, se puede ver el fragmento de código que realiza esta operación.

```
useEffect(() => {
  console.log("starting provider");
  let provider = googleMainProvider;
  console.log({ provider });
  provider.setCustomParameters({
    prompt: "select_account",
  });
  console.log({ provider });
  setGoogleProvider(provider);

  onAuthStateChanged(auth, (user) => {
    if (user) {
      setUser(user);
      setShowChild(true);
    }
  });
}, []);
```

Con el usuario registrado, el siguiente paso es realizar la conexión con la base de datos en tiempo real. En el momento en el que la base de datos se conecta, se generan los observadores de eventos para las referencias de salas y usuarios, para que los cambios en la base de datos se reflejen en el estado del componente proveedor. Firebase proporciona estas funciones, por lo que en estos observadores se debe realizar la manipulación directa del estado del componente. Además, se generan dos funciones específicas para la manipulación de la cola de espera de la sala. A la hora de registrarse en la cola de una sala, se realiza un *push*, o una inserción en la lista, con la información del UID del usuario como valor, mientras que la clave del objeto se genera en función de la fecha en la que se mandó la petición. Dado que se trabaja desde un elemento central, no se produce así ningún problema de concurrencia u otros problemas de simultaneidad de comunicación de datos. En esta función no se realiza una modificación del estado, ya que los observadores que se generaron anteriormente realizan esta función justo a continuación, y facilita la sincronización de todos los usuarios con la base de datos. Para salirse de la lista de espera, primero se busca en la información de la lista la clave temporal correspondiente con el usuario que se quiere salir. Después, con esa clave, se procede a eliminarla de la base de datos para esa sala en particular. Cuando se ha añadido al usuario en la lista, se le proporcionan 20 segundos de inactividad, por lo que, si pasan diez segundos sin actividad por parte del portador del turno, automáticamente se elimina a este usuario de la lista de espera.

```
const addToQueue = (room) => {
  let databaseRef = ref(database, `rooms/${room}/queue`);
  let databaseListRef = push(databaseRef);
  set(databaseListRef, user.uid);
};

const removeFromQueue = (room) => {
  let roomQueue = roomsInfo[room].queue;
  let timestampId = null;
  for (let t in roomQueue) {
    if (roomQueue[t] == user.uid) {
      timestampId = t;
      break;
    }
  }

  if (!timestampId) return;

  let queueRef = ref(database, `rooms/${room}/queue/${timestampId}`);
  remove(queueRef);
};
```

Con el estado del proveedor de la base de datos, se programan dos funciones que servirán para conocer el estado del usuario en la lista de espera. La primera de estas es `isOnQueue`, que comprueba si el usuario que quiere entrar en la lista de espera está ya en la lista, para no duplicar las entradas. La segunda es `isMyTurn`, que comprueba si el usuario dado es el primero en la lista de espera. De esta manera, se puede realizar una función u otra para poder realizar diferentes operaciones.

```
const isOnQueue = (room) => {
  let roomQueue = roomsInfo[room]?.queue;
  if (!roomQueue) return false;
  let usersInQueue = Object.values(roomQueue);
  return usersInQueue.includes(user.uid);
};

const isMyTurn = (room) => {
  let roomQueue = roomsInfo[room]?.queue;
  if (!roomQueue) return false;
  let firstKey = Object.keys(roomQueue)[0];
  return roomQueue[firstKey] == user.uid;
};
```

Si se ha realizado la conexión con la base de datos correctamente, el último paso antes de llegar a la aplicación es realizar la conexión con Janus. El código de la API proporcionado por Janus está pensado para poder ser ejecutada en un archivo JavaScript puro, por lo que antes de continuar hay que dividir el código en trozos de funcionalidad más pequeños. Y, debido a la estructura de la API de ejecutar funciones cuando un proceso haya acabado, se dividen éstos en diferentes promesas, que se resuelven cuando el objeto final se ha obtenido en cada sección. De esta división obtenemos la distinción entre `init`, `createSession` y `attachPlugin`, donde se especifican las operaciones que se deben realizar para los diferentes eventos de la comunicación WebRTC. De estos procesos, que se ejecutan cuando se carga el proveedor, se obtiene un *pluginHandle* que se puede utilizar para realizar las diferentes peticiones al servidor, como mandar un mensaje o unirse a una sala. Cada vez que se recibe un mensaje, o un nuevo participante se ha unido a la sala, se emplea este *handle* para modificar el estado del componente cuando se recibe la respuesta del servidor. Todos estos datos se pasan después al proveedor para poder ser utilizados a lo largo de toda la aplicación.

```
const init = () => {
  return new Promise((resolve, _) => {
    let initParams = {
      debug: "all",
      dependencies: defaultDependencies,
      callback: () => {
        resolve();
      },
    };

    Janus.init(initParams);
  });
};

const createSession = () => {
  return new Promise((resolve, reject) => {
    let session;
    let params = {
      server: ["ws://localhost:8188/", "http://localhost:8088/janus"],
      success: () => {
        console.log("Success!", session.getSessionId());
        addNewSession(session);
        resolve(session);
      }, // Connected correctly
      error: (error) => {
        console.error("Error creating session", error);
        reject(error);
```

```
      },
      destroyed: () => console.log("Destroyed session!"),
    };

    session = new Janus(params);
  });
};

useEffect(() => {
  if (user) {
    // User logged in
    setMyUsername(escapeXmlTags(user.displayName));

    transactionsTemp = transactions;

    // We start Janus service
    const prepareJanus = async () => {
      await init();
      let session = await createSession();
      if (session) {
        let pluginHandle = await attachVideoRoomPlugin(session);
        if (pluginHandle) addNewPluginHandle(pluginHandle);
        else console.error(pluginHandle);
      } else {
        console.error(session);
      }
    };
    prepareJanus();
  }
}, [user]);
```

Con estos tres proveedores ajustados, el último paso es mostrar los datos en la aplicación, donde se accede al nombre del usuario para mostrar en el perfil. Los mensajes que se muestran con el componente de mensaje visto anteriormente, y la información de la base de datos se actualiza automáticamente en la aplicación. Esto es, cuando sucede un cambio en la base de datos, el resto de la aplicación es capaz de renderizarlo. Para poder realizarlo, React ofrece una funcionalidad llamada React Hooks, que proporciona una manera de ejecutar funciones cuando una cierta variable ha cambiado de valor. Cuando esto lo unimos al *provider*, sucede que cada vez que hay un cambio en la base de datos, se actualiza el estado del proveedor de la base de datos, y en cada uno de los componentes se puede realizar una función cuando esta variable ha cambiado y se está leyendo de ésta. Por lo tanto, toda la aplicación se actualiza al mismo tiempo, haciendo que la interactividad sea mayor. Un

ejemplo para el proveedor WebRTC sería el siguiente, que permite cargar la información de una sala específica cuando se realiza el cambio de sala.

```
useEffect(() => {
  if (currentRoom) {
    console.log("Connecting to room", currentRoom);
    tempRoom = currentRoom;
    let messagesRaw = roomsInfo[currentRoom]?.messages;
    let messagesTemp = messagesRaw ? Object.values(messagesRaw) : [];
    console.log({ roomsInfo, messagesTemp });
    messagesTemp = messagesTemp.map((message) => parseMessage(message));
    console.log({ roomsInfo, messagesTemp });
    setMessages(messagesTemp);

    let roomParticipants = roomsInfo[currentRoom]?.participants;
    console.log({ roomParticipants });
    let mappedParticipants = {};

    if (roomParticipants) {
      for (let p in roomParticipants) {
        mappedParticipants = {
          ...mappedParticipants,
          [p]: roomParticipants[p]["display"],
        };
      }
      setParticipants(mappedParticipants);
    }
  }
}, [currentRoom]);
```

Para realizar las operaciones que permite Janus, se programan en el proveedor varias funciones de apoyo, como `publishOwnFeed`, `unpublishOwnFeed`, `joinVideoRoom`, `createVideoRoom`, `toggleMute` y `toggleVideo`, que realizan las operaciones de publicación y despublicación de los streams locales, la creación y unión a salas, además del control del stream enviado a través del muteado de los diferentes *tracks*. Debido a lo largo que es este componente, no se publicará en este documento. Para acceder a la última versión de este proyecto, habrá disponible un repositorio de Github público[1] donde se puede acceder a la información de este componente, con el nombre de `JanusProvider`.

[1] Link del proyecto: https://github.com/pmurcia/P2T

A continuación, se muestra el código de creación de la conexión con el plugin *videoroom* con una explicación de las secciones importantes. El caso de newRemoteFeed es similar a este, por lo que se puede usar como ejemplo.

```
const attachVideoRoomPlugin = (session) => {
  // let transactionsNow = transactions;
  return new Promise((resolve, reject) => {
    let videoRoom;
    // let transactionsNow;

    let params = {
      plugin: "janus.plugin.videoroom",
      success: function (pluginHandle) {
        // Plugin attached! 'pluginHandle' is our handle
        console.log("pluginHandle", pluginHandle);

        videoRoom = pluginHandle;
        resolve(pluginHandle);

      },
      …

      },
```

Cuando se recibe un mensaje del plugin, se activa el evento *onmessage*. Primero, debido a que ciertas peticiones son asíncronas, es necesario comprobar si este nuevo mensaje tiene un número de transacción asociado. También hay un caso especial de unión a una sala que no lleva asociado un número de transacción, pero sí se puede identificar. Si hay una transacción, es porque lleva asociado una función que se debe realizar. Esto es, se manda el mensaje y se ejecuta una función cuando se recibe la respuesta. Una vez ejecutada la transacción, se deja de ejecutar el código de esta parte.

Si aparece un nuevo *publisher*, o sea, un nuevo participante capaz de publicar *streams*, se crea un nuevo "Remote Feed" que mostrará este nuevo *stream* en el elemento de vídeo de la aplicación. Si un *publisher* deja de publicar, se elimina este remoto.

```
      onmessage: function (msg, jsep) {
        // We got a message/event (msg) from the plugin
        // If jsep is not null, this involves a WebRTC negotiation
        if (msg["error"]) {
          console.error("Onmessage", msg["error"]);
        }
        Janus.debug(" ::: Got a message (publisher) :::", msg);
        let transaction = msg["transaction"];
```

```
        console.log("Execute transaction", { transactionsTemp, transaction });

        // Special case of join
        let msgPvtId = msg["id"];
        if (
          msg["videoroom"] == "joined" &&
          Object.keys(transactionsTemp).length > 0 &&
          msgPvtId == tempmyid
        ) {
          console.log("Joining room transaction");
          let id = Object.keys(transactionsTemp)[0];
          transactionsTemp[id](msg);
          delete transactionsTemp[transaction];
          setTransactions(transactionsTemp);
          return;
        }

        if (transaction && transactionsTemp[transaction]) {
          transactionsTemp[transaction](msg);
          delete transactionsTemp[transaction];
          setTransactions(transactionsTemp);

          return;
        }
        let event = msg["videoroom"];
        Janus.debug("Event: " + event);
        if (event) {
          if (event === "joined") {
            // Publisher/manager created, negotiate WebRTC and attach to existing
feeds, if any
            setMyPvtId(msg["private_id"]);
            // Any new feed to attach to?
            if (msg["publishers"]) {
              let list = msg["publishers"];
              Janus.debug("Got a list of available publishers/feeds:", list);
              for (let f in list) {
                let id = list[f]["id"];
                let display = list[f]["display"];
                let streams = list[f]["streams"];
                for (let i in streams) {
                  let stream = streams[i];
                  stream["id"] = id;
                  stream["display"] = display;
                }
                // feedStreams[id] = streams;
                setFeedStreams((prev) => {
                  return { ...prev, [id]: streams };
                });
                Janus.debug("  >> [" + id + "] " + display + ":", streams);
                newRemoteFeed(id, display, streams);
              }
            }
          } else if (event === "destroyed") {
```

```
        // The room has been destroyed
        Janus.warn("The room has been destroyed!");
      } else if (event === "event") {
        // Any info on our streams or a new feed to attach to?
        if (msg["streams"]) {
          let streams = msg["streams"];
          for (let i in streams) {
            let stream = streams[i];
            stream["id"] = myid;
            stream["display"] = myusername;
          }
          setFeedStreams((prev) => {
            return {
              ...prev,
              [myid]: streams,
            };
          });
        } else if (msg["publishers"]) {
          let list = msg["publishers"];
          Janus.debug("Got a list of available publishers/feeds:", list);
          let mainPublisher = list[0];
          publisherId = mainPublisher.id;
          for (let f in list) {
            let id = list[f]["id"];
            let display = list[f]["display"];
            let streams = list[f]["streams"];
            for (let i in streams) {
              let stream = streams[i];
              stream["id"] = id;
              stream["display"] = display;
            }
            // feedStreams[id] = streams;
            setFeedStreams((prev) => {
              return {
                ...prev,
                [id]: streams,
              };
            });
            Janus.debug("  >> [" + id + "] " + display + ":", streams);
            newRemoteFeed(id, display, streams);
          }
        } else if (msg["leaving"]) {
          // One of the publishers has gone away?
          let leaving = msg["leaving"];
          Janus.log("Publisher left: " + leaving);
          let remoteFeed = null;
          for (let i = 1; i < 6; i++) {
            if (feeds[i] && feeds[i].rfid == leaving) {
              remoteFeed = feeds[i];
              break;
            }
          }
          if (remoteFeed) {
```

```
            Janus.debug(
              "Feed " +
                remoteFeed.rfid +
                " (" +
                remoteFeed.rfdisplay +
                ") has left the room, detaching"
            );
            setFeeds((prev) => {
              let newFeeds = prev;
              newFeeds[remoteFeed.rfindex] = null;
              return [...newFeeds];
            });
            remoteFeed.detach();
          }
          // delete feedStreams[leaving];
          setFeedStreams((prev) => {
            let newFeedStreams = prev;
            delete newFeedStreams[leaving];
            return {
              ...newFeedStreams,
            };
          });
        } else if (msg["unpublished"]) {
          // One of the publishers has unpublished?
          let unpublished = msg["unpublished"];
          Janus.log("Publisher left: " + unpublished);
          if (unpublished === "ok") {
            // That's us
            videoRoom.hangup();
            return;
          }
          let remoteFeed = null;
          for (let i = 1; i < 6; i++) {
            if (feeds[i] && feeds[i].rfid == unpublished) {
              remoteFeed = feeds[i];
              break;
            }
          }
          if (remoteFeed) {
            Janus.debug(
              "Feed " +
                remoteFeed.rfid +
                " (" +
                remoteFeed.rfdisplay +
                ") has left the room, detaching"
            );
            setFeeds((prev) => {
              let newFeeds = prev;
              newFeeds[remoteFeed.rfindex] = null;
              return [...newFeeds];
            });
            remoteFeed.detach();
          }
```

```
              setRemoteStream(null);
              setFeedStreams((prev) => {
                let newFeedStreams = prev;
                delete newFeedStreams[unpublished];
                return {
                  ...newFeedStreams,
                };
              });
            } else if (msg["error"]) {
              if (msg["error_code"] === 426) {

              } else {
                console.error(msg["error"]);
              }
            }
          }
        }
        if (jsep) {
          console.log({ jsep });
          videoRoom.handleRemoteJsep({ jsep: jsep });
        }
      },
```

Si el usuario tiene la palabra, se le proporcionará la capacidad de mandar sus propios *streams* a la sala. El evento *onlocalstream* se activa cuando existe este stream, por lo que la función que se realiza es de mostrar el *stream* en el elemento de vídeo de la aplicación. Sin embargo, antes de añadir el *stream*, se elimina el *track* de audio para que no haya eco.

```
      onlocalstream: function (localStream) {
        // console.log("onlocalstream", stream);
        Janus.debug(
          "Local stream " + (localStream.active ? "active" : "inactive") + ":",
          localStream
        );

        if (!localStream) {
          // We've been here already
          console.error("No stream found");
          return;
        }

        // We need to remove the audio tracks from the stream, so echo doesn't
appear
        let stream = new MediaStream();
        let videoTracks = localStream.getVideoTracks();
        for (let track of videoTracks) {
          stream.addTrack(track.clone());
        }
        setLocalStream(stream.clone());
      },
```

Si se recibe un mensaje por el *DataChannel*, se activa el evento *ondata*. Lo que se realiza aquí es similar a lo que se realiza en *onmessage*, con la principal diferencia de que se pueden recibir mensajes de texto a través de este canal. En caso de que se reciban, se guarda en una variable de estado local junto al resto de mensajes. Cabe destacar que, aunque esto se realice en tiempo real, cuando se carga la aplicación se recoge la información de los mensajes de la base de datos de Firebase.

```
      ondata: function (data) {
        Janus.debug("We got data from the DataChannel!", data);

        let json = JSON.parse(data);
        let transaction = json["transaction"];
        console.log("Execute transaction", { transactionsTemp });
        if (transaction && transactionsTemp[transaction]) {
          // Someone was waiting for this
          transactionsTemp[transaction](json);
          delete transactionsTemp[transaction];
          setTransactions(transactionsTemp);

          return;
        }
        let what = json["textroom"];
        if (what === "message") {
          // Incoming message: public or private?
          let msg = escapeXmlTags(json["text"]);
          let from = json["from"];
          let dateString = json["date"];
          let whisper = json["whisper"];
          if (whisper) {
            // Private message
            console.log("Private message", msg);
          } else {
            // Public message
            console.log("Public message", msg);
          }

          // Save the message to a database
          let newMessage = {
            text: msg,
            author: from,
            timestamp: dateString,
          };
          setMessages((prevMessages) => {
            return [...prevMessages, newMessage];
          });
        } else if (what === "announcement") {
          // Room announcement
          let msg = escapeXmlTags(json["text"]);
```

```
          let dateString = getDateString(json["date"]);
        } else if (what === "join") {
          // Somebody joined
          let username = json["username"];
          let display = json["display"];
          let newParticipantObject = {};
          let newParticipantValue = escapeXmlTags(display ? display : username);
          console.log({ newParticipantObject });
          setParticipants((prevParticipants) => {
            return {
              ...prevParticipants,
              [username]: newParticipantValue,
            };
          });

        } else if (what === "leave") {
          // Somebody left
          let username = json["username"];
          let when = new Date();
          setParticipants((prevParticipants) => {
            delete prevParticipants[username];
            return {
              ...prevParticipants,
            };
          });
        } else if (what === "kicked") {
          // Somebody was kicked
          let username = json["username"];
          let when = new Date();
          setParticipants((prevParticipants) => {
            delete prevParticipants[username];
            return {
              ...prevParticipants,
            };
          });
          if (username === user.uid) {
            console.log("You have been kicked from the room");
          }
        } else if (what === "destroyed") {
          if (json["room"] !== currentRoom) return;
          // Room was destroyed, goodbye!
          Janus.warn("The room has been destroyed!");
        }
      },
    };

    console.log("Attached to current session", session);
    session.attach(params);
  });
};
```

A continuación, se mostrará el resto del código que permite la gestión de la sesión WebRTC.

```
const sendData = (data) => {
  let pluginHandle = currentPluginHandle;
  if (data === "") {
    // bootbox.alert('Insert a message to send on the DataChannel');
    return;
  }
  let json = {
    textroom: "message",
    transaction: randomString(12),
    room: currentRoom,
    text: data,
  };

  console.log({ json, data });

  // Note: messages are always acknowledged by default. This means that you'll
  // always receive a confirmation back that the message has been received by the
  // server and forwarded to the recipients. If you do not want this to happen,
  // just add an ack:false property to the message above, and server won't send
  // you a response (meaning you just have to hope it succeeded).
  pluginHandle.data({
    text: JSON.stringify(json),
    error: (reason) => {
      console.log(reason);
    },
    success: () => {
      let messageModel = {
        ...json,
      };
      let what = json["textroom"];

      // Update message with info
      if (what === "message") {
        // Incoming message: public or private?
        let msg = escapeXmlTags(json["text"]);
        let from = user.uid;
        let dateString = Date.now();
        let whisper = false;
        // Public message
        console.log("Public message", msg);

        // Prepare message to save to database
        messageModel = {
          ...messageModel,
          date: dateString,
          from: from,
          whisper: whisper,
        };

        // Save to current messages
        // let messagesUpdated = messages;
        let newMessage = {
          text: msg,
```

```
          author: from,
          timestamp: dateString,
        };
        setMessages((prevMessages) => {
          return [...prevMessages, newMessage];
        });
      } else if (what === "announcement") {
        // Room announcement
        let msg = escapeXmlTags(json["text"]);
        let dateString = getDateString(json["date"]);
      }

      // Save message to database
      appendMessage(messageModel);
    },
  });
};
```

La función anterior manda un mensaje a través del canal de datos. En caso de que se mande correctamente, se añade el mensaje enviado a los mensajes propios. Aquí se puede apreciar que el contenido del mensaje coincide con el de la base de datos, dado que la última línea de código ejecuta una función para guardar el mensaje en la base de datos.

Unirse a una sala provoca una respuesta asíncrona, por lo que en el *attach* del plugin se tuvo que realizar esa excepción para este caso. Aquí se puede ver cómo cada transacción tiene una función asociada, por lo que se puede ejecutar cuando se recibe el mensaje. Si el usuario se une a una sala específica, se ajusta la variable de estado de sala actual a la nueva sala, y automáticamente se carga la información asociada a esa sala. No sólo eso, sino que si el usuario no está en la lista de participantes, se le añade con su identificador de *publisher*.

```
const joinVideoRoom = (roomId) => {
  let pluginHandle = currentPluginHandle;
  if (!myusername) {
    return;
  }
  let transaction = randomString(12);
  let pvtId = myid;
  let join = {
    request: "join",
    transaction: transaction,
    room: roomId,
    display: myusername,
    ptype: "publisher",
    id: pvtId,
  };
  console.log({ join });
```

```
console.log({ pvtId });
tempmyid = pvtId;

const transactionFunc = (response) => {
  console.log({ response });

  if (response["videoroom"] === "event") {
    // Something went wrong
    if (response["error_code"] === 417) {
      console.error("No room with that code");
    } else if (
      response["error_code"] == 420 ||
      response["error_code"] == 421
    ) {
      console.log("Already logged in");
      setCurrentRoom(roomId);
    } else {
      console.error(response["error"]);
    }
    return;
  }

  // We're in
  setCurrentRoom(roomId);

  // If I'm not in the room, add it to my room
  let userRooms = usersInfo[user.uid]?.rooms;
  console.log({ usersInfo, user, userRooms });

  if (userRooms) {
    userRooms = Object.values(userRooms);
    console.log({ userRooms });
    if (!userRooms.includes(roomId)) addUserRoom(roomId);
  } else {
    addUserRoom(roomId);
  }

  // Add myself to overall participants
  addToRoomParticipants(roomId, {
    videoRoomId: response["id"],
    videoRoomPrivateId: response["private_id"],
  });

  setMyPvtId(response["private_id"]);
  // TODO needed???
  // Any participants already in?
  console.log("Participants:", response.attendees);
  if (response?.participants && response?.participants?.length > 0) {
    let joinedParticipants = {};

    for (let i in response.participants) {
      let p = response.participants[i];
```

```
        let newParticipantObject = {};
        let newParticipantValue = escapeXmlTags(
          p.display ? p.display : p.username
        );
        // newParticipantObject[p.username] = escapeXmlTags(
        //   p.display ? p.display : p.username
        // );
        joinedParticipants = {
          ...joinedParticipants,
          [p.username]: newParticipantValue,
        };

        if (p.username !== user.uid) {
          // Send private message as joined participant
          sendPrivateMsg(p.username);
        }
      }

      // Add new participants to state
      setParticipants((prevParticipants) => {
        return {
          ...prevParticipants,
          ...joinedParticipants,
        };
      });
    }
  };

  transactionsTemp = {
    ...transactions,
    [transaction]: transactionFunc,
  };

  setTransactions((prevTransactions) => {
    return {
      ...transactions,
      [transaction]: transactionFunc,
    };
  });

  pluginHandle.send({
    message: join,
    transaction: transaction,
    error: function (reason) {
      console.error(reason);
    },
    success: (res) => {
      console.log("JOINED CORRECTLY", res);
    },
  });
};
```

El proceso de creación de una sala de videoconferencia es más sencillo, ya que se trata de una solicitud síncrona. Esto es, se puede definir una función en la propia llamada al plugin para ejecutar cuando se recibe la confirmación. Se crea con la intención de guardarlo en el archivo de configuración de Janus con el *flag* de permanente. Si la sala creada es permanente, entonces se ha creado correctamente la sala, se añade el número de la sala al usuario y se ajusta el nombre de la sala, que por defecto tendrá el nombre de "Room" en la base de datos.

```
const createVideoRoom = (description = "Room") => {
  let pluginHandle = currentPluginHandle;

  console.log({ pluginHandle, currentPluginHandle });

  let create = {
    request: "create",
    permanent: true,
    is_private: true,
  };

  descriptionTemp = description;

  const transactionFunc = (response) => {
    console.log({ response });
    if (response["videoroom"] === "error") {
      // Something went wrong
      if (response["error_code"] === 417) {
        console.error("No room with that code");
      } else {
        console.error(response["error"]);
      }
      return;
    }

    if (!response["permanent"]) {
      console.error("Could not save to config file: Permissions problems");
      return;
    }

    // Room created correctly
    // setCurrentRoom(response["room"]);
    addUserRoom(response["room"]);

    console.log({ response, descriptionTemp });

    // Add Room to list of rooms
    addRoomToList({
      roomId: response["room"],
      description: descriptionTemp,
    });
```

```
    descriptionTemp = "";
  };

  pluginHandle.send({
    message: create,
    success: transactionFunc,
  });
};
```

Las funciones a continuación sirven cuando el usuario tiene la palabra. Para publicar el stream local, se solicita un *offer* en el que se especifica que se van a enviar audio y vídeo sin opción a recibir. También se manda el *flag* de *data* que indica que se va a abrir un canal de datos para posibilitar mandar mensajes de un participante a otro. Esta oferta se gestiona después con la función handleRemoteJsep, que se puede ver en el proceso de conexión al plugin. Cuando se crea correctamente la oferta, se manda al plugin la información del SDP junto a la solicitud de publicación.

```
const publishOwnFeed = (useAudio, useVideo) => {
  let pluginHandle = currentPluginHandle;

  console.log({ useAudio, useVideo, pluginHandle, currentPluginHandle });

  // Publish our stream
  pluginHandle.createOffer({
    media: {
      audioRecv: false,
      videoRecv: false,
      audioSend: useAudio,
      videoSend: useVideo,
      data: true,
    }, // Publishers are sendonly
    success: function (jsep) {
      Janus.debug("Got publisher SDP!", jsep);
      let publish = {
        request: "configure",
        audio: useAudio,
        video: useVideo,
        data: true,
      };
      pluginHandle.send({
        message: publish,
        jsep: jsep,
      });

      if (!audioOn) pluginHandle.muteAudio();
      if (!videoOn) pluginHandle.muteVideo();
    },
```

```
    error: function (error) {
      Janus.error("WebRTC error:", error);
    },
  });
};
```

Cuando el usuario ha dejado de hablar, se solicita la despublicación del *stream*, que tiene una estructura muy sencilla, como se puede ver a continuación. Esto termina el envío de *streams*, y se hacen ajustes al elemento de vídeo.

```
const unpublishOwnFeed = () => {
  let pluginHandle = currentPluginHandle;

  let unpublish = {
    request: "unpublish",
    transaction: randomString(12),
  };
  pluginHandle.send({ message: unpublish });
  setLocalStream({});
};
```

Por último, las funciones de *toggle* son casi idénticas. Dado que Janus ofrece la posibilidad de mutear tanto audio como vídeo, las siguientes funciones emplean esa funcionalidad para que el proceso sea más sencillo desde el interfaz de usuario.

```
const toggleAudio = () => {
    console.log("toggleAudio", currentPluginHandle.isAudioMuted());
    let muted = currentPluginHandle.isAudioMuted();
    Janus.log((muted ? "Unmuting" : "Muting") + " local stream...");
    if (muted) currentPluginHandle.unmuteAudio();
    else currentPluginHandle.muteAudio();
    setAudioOn(currentPluginHandle.isAudioMuted());
  };

  const toggleVideo = () => {
    console.log("toggleVideo", currentPluginHandle.isVideoMuted());
    let muted = currentPluginHandle.isVideoMuted();
    Janus.log((muted ? "Unmuting" : "Muting") + " local stream...");
    if (muted) currentPluginHandle.unmuteVideo();
    else currentPluginHandle.muteVideo();
    setVideoOn(currentPluginHandle.isVideoMuted());
  };
```

Capítulo 6. CONCLUSIONES Y TRABAJOS FUTUROS

La aplicación resultante de este trabajo implementa correctamente la funcionalidad de P2T de lista de espera mediante WebRTC para la comunicación multimedia. Esta aplicación se carga correctamente, y carga directamente los datos requeridos sin necesidad de recargar la página, que es una ventaja cuando se considera que cuando se inicia la aplicación se debe crear una sesión Janus nueva. La comunicación WebRTC se realiza correctamente y es rápida, aunque sería interesante añadir optimización a la negociación de la comunicación. Por último, es factible servir esta aplicación a través de un servidor HTTP, y ofrece diferentes tipos de comunicación entre usuarios.

Como futuro objetivo, sería interesante implementar una interfaz de usuario más intuitiva y dinámica, con más animaciones o diferente comportamiento para dejar más claro al usuario qué tipo de operaciones se están realizando desde la aplicación. Además, sería interesante también añadir la posibilidad de grabar las comunicaciones que se mandan para un futuro consumo del contenido multimedia. Esto es, que al entrar de nuevo en la aplicación se puedan ver las comunicaciones anteriores de la sala.

Por último, como objetivos a más largo plazo, habría que añadir la capacidad de incorporar dispositivos SIP *legacy* a las comunicaciones. De esta manera, se podría integrar la aplicación en sí con la infraestructura ya existente en muchas aplicaciones. Dado que se está dependiendo directamente de Janus, otro de los objetivos sería sustituir Janus por otras librerías como MediaSOUP o ffmepg, para acabar con un servidor de propio diseño adaptado a P2T.

Capítulo 7. BIBLIOGRAFÍA

[1] W3C, «W3C WebAssembly Working Group,» [En línea]. Available: https://www.w3.org/wasm/. [Último acceso: 1 Enero 2022].

[2] «Media Source Extensions™,» W3C, 30 Septiembre 2021. [En línea]. Available: https://www.w3.org/TR/media-source-2/. [Último acceso: 1 Enero 2022].

[3] «Made with WebAssembly,» [En línea]. Available: https://madewithwebassembly.com/. [Último acceso: 28 Enero 2022].

[4] Stack Overflow, «Stack Overflow Developer Survey 2021,» 2021. [En línea]. Available: https://bit.ly/3qMjuMo.

[5] Google Developers, «WebRTC,» [En línea]. Available: https://webrtc.org/. [Último acceso: 1 Enero 2022].

[6] W3C, «WebRTC 1.0: Real-Time Communication Between Browsers,» W3C, 26 Enero 2021. [En línea]. Available: https://www.w3.org/TR/webrtc/. [Último acceso: 1 Enero 2022].

[7] «RFC 8825 - Overview: Real-Time Protocols for Browser-Based Applications,» IETF, Enero 2021. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc8825. [Último acceso: 20 Enero 2022].

[8] «RFC 8835 - Transports for WebRTC,» IETF, Enero 2021. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc8835. [Último acceso: 12 Enero 2022].

[9] «RFC 7874 - WebRTC Audio Codec and Processing Requirements,» IETF, Mayo 2016. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc7874. [Último acceso: 12 Enero 2022].

[10] «RFC 7742 - WebRTC Video Processing and Codec Requirements,» IETF, Mayo 2016. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc7742. [Último acceso: 12 Enero 2022].

[11] «RFC8829 - JavaScript Session Establishment Protocol (JSEP),» IETF, Enero 2021. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc8829. [Último acceso: 12 Enero 2022].

[12] «Media Capture and Streams,» W3C, 20 Enero 2022. [En línea]. Available: https://www.w3.org/TR/mediacapture-streams. [Último acceso: 20 Enero 2022].

[13] Open Mobile Alliance, «Push to Talk over Cellular Requirements,» 9 Junio 2006. [En línea]. Available: http://www.openmobilealliance.org/release/PoC/V1_0-20060609-A/OMA-RD-PoC-V1_0-20060609-A.pdf. [Último acceso: 15 Enero 2022].

[14] J. Vass, «How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC,» Discord Blog, 10 Septiembre 2018. [En línea]. Available: https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc. [Último acceso: 12 Enero 2022].

[15] testRTC, «How Houseparty uses testRTC as an integral part of its WebRTC testing,» TestRTC, [En línea]. Available: https://testrtc.com/houseparty-testrtc-testimonial/. [Último acceso: 28 Enero 2022].

[16] L. Barr, «Built with WebRTC: Snapchat,» Blacc Spot Media, 29 Febrero 2016. [En línea]. Available: https://www.linkedin.com/pulse/built-webrtc-snapchat-lantre-barr. [Último acceso: 28 Enero 2022].

[17] A. García Pérez, «Comunicaciones TETRA: La red de las fuerzas de seguridad,» *Revista Digital de ACTA,* 2013.

[18] Zello, «Zello | The Most Reliable Push-To-Talk Walkie Talkie App,» [En línea]. Available: https://zello.com. [Último acceso: 28 Enero 2022].

[19] Wazo Platform, «Creating a Cross Environment push-to-talk Application Using Wazo,» 12 Octubre 2021. [En línea]. Available: https://wazo-platform.org/blog/creating-cross-platform-push-to-talk-app. [Último acceso: 3 Febrero 2022].

[20] «Wazo Platform Main Page,» [En línea]. Available: https://wazo-platform.org/. [Último acceso: 3 Febrero 2022].

[21] Wikipedia, «Jeff Atwood - Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Jeff_Atwood. [Último acceso: 30 Enero 2022].

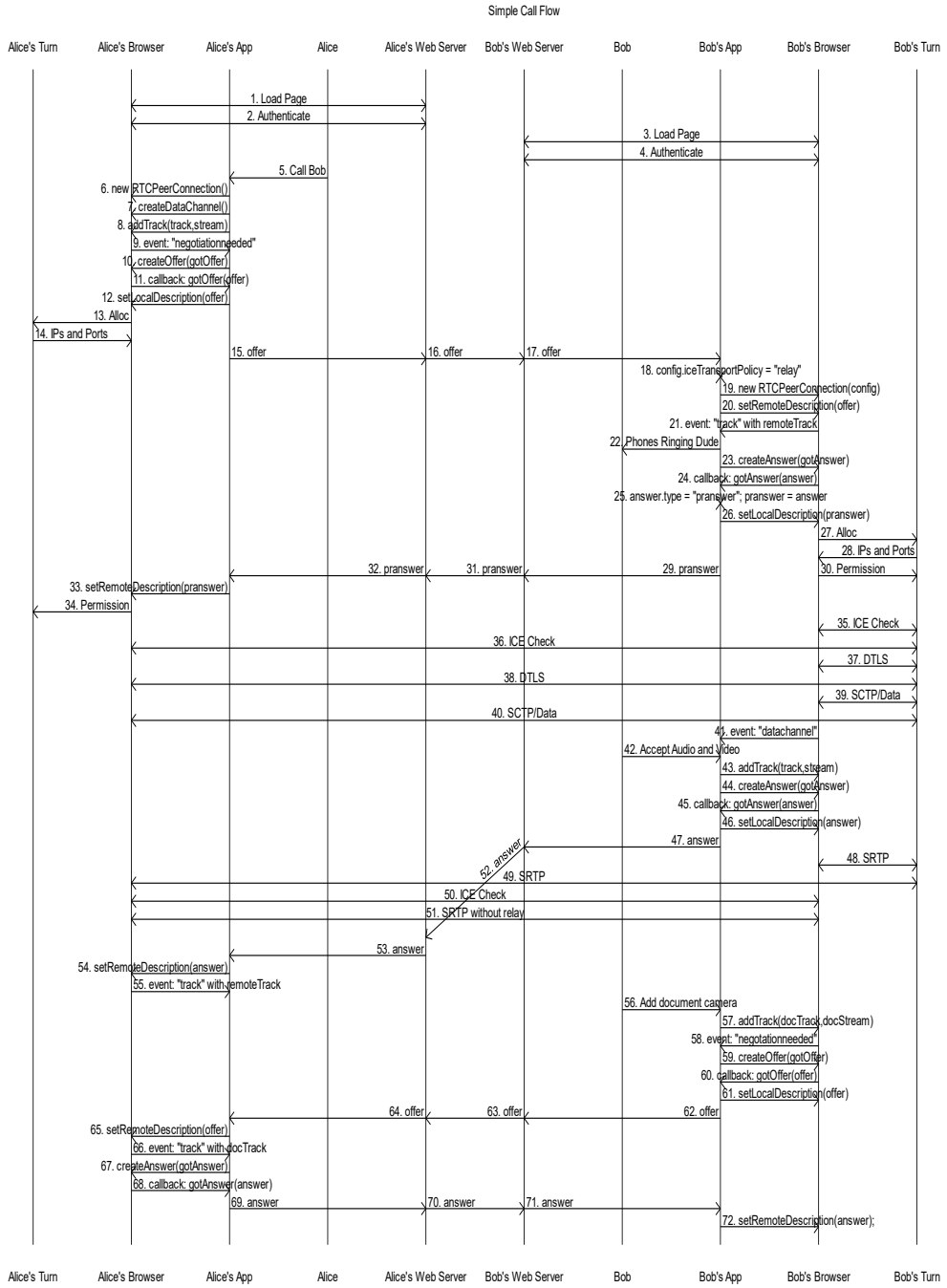
[22] A. Amirante, T. Castaldi, L. Miniero y S. P. Romano, «Janus: a general purpose WebRTC gateway,» *IPTComm '14*, nº 7, pp. 1-8, 2014.

[23] Facebook Open Source, «React - Una biblioteca de JavaScript para construir interfaces de usuario,» 2022. [En línea]. Available: https://es.reactjs.org/. [Último acceso: 20 Enero 2022].

[24] Google, «Home - Material Design,» 2022. [En línea]. Available: https://material.io/.

ANEXO A

Call Flow Browser to Browser [6], Section 10.5



ANEXO B

Componente visual principal (versión en publicación)

```
import React, { useContext, useEffect, useRef, useState } from "react";
import {
  Avatar,
  Grid,
  List,
  ListItem,
  ListItemIcon,
  ListItemText,
  Typography,
  Paper,
  Divider,
  TextField,
  IconButton,
  Button,
  ListItemAvatar,
  CircularProgress,
  Modal,
  Box,
} from "@mui/material";
import SendIcon from "@mui/icons-material/Send";
import MicIcon from "@mui/icons-material/Mic";
import MicOffIcon from "@mui/icons-material/MicOff";
import VideocamIcon from "@mui/icons-material/Videocam";
import VideocamOffIcon from "@mui/icons-material/VideocamOff";
import { makeStyles } from "@mui/styles";
import { JanusContext } from "../janus/JanusProvider";
import Message from "../Chats/Message";
import { AuthContext } from "../firebase/AuthProvider";
import { DatabaseContext } from "../firebase/DatabaseProvider";
import ProfileMenu from "./ProfileMenu";

const useStyles = makeStyles({
  table: {
    minWidth: 650,
  },
  chatSection: {
    width: "100%",
    height: "100vh",
  },
  headBG: {
    backgroundColor: "#e0e0e0",
  },
  borderRight500: {
    borderRight: "1px solid #e0e0e0",
```

```
  },
  borderLeft500: {
    borderLeft: "1px solid #e0e0e0",
    height: "100vh",
  },
  messageArea: {
    minHeight: "80vh",
    overflowY: "auto",
  },
});

export default function ChatsLayout() {
  const {
    currentSession,
    sendData,
    messages,
    currentRoom,
    participants,
    joinVideoRoom,
    publishOwnFeed,
    unpublishOwnFeed,
    localTracks,
    remoteStream,
    localStream,
    toggleAudio,
    toggleVideo,
    audioOn,
    videoOn,
  } = useContext(JanusContext);
  const { user } = useContext(AuthContext);
  const {
    addToQueue,
    isMyTurn,
    removeFromQueue,
    isOnQueue,
    roomsInfo,
    usersInfo,
  } = useContext(DatabaseContext);

  const [canTalk, setCanTalk] = useState(false);
  const [myRooms, setMyRooms] = useState();

  const [audioPlaying, setAudioPlaying] = useState(audioOn);
  const [videoPlaying, setVideoPlaying] = useState(videoOn);

  useEffect(() => {
    setAudioPlaying(audioOn);
    setVideoPlaying(videoOn);
  }, [audioOn, videoOn]);

  let messageInput = useRef(null);
  let messagesEnd = useRef(null);
  let talkTimerRef = useRef(null);
```

```
const videoElement = useRef(null);

const classes = useStyles();

useEffect(() => {
  console.log("New local stream", { localStream });
  console.log("New remote stream", { remoteStream });

  let isEmptyLocalStream =
    localStream &&
    Object.keys(localStream).length === 0 &&
    Object.getPrototypeOf(localStream) === Object.prototype;
  let isEmptyRemoteStream =
    remoteStream &&
    Object.keys(remoteStream).length === 0 &&
    Object.getPrototypeOf(remoteStream) === Object.prototype;

  console.log({ isEmptyLocalStream, isEmptyRemoteStream });

  if (videoElement?.current) {
    if (!isEmptyLocalStream && isEmptyRemoteStream) {
      videoElement.current.srcObject = localStream;
      console.log("Adding local source");
    } else if (isEmptyLocalStream && !isEmptyRemoteStream) {
      videoElement.current.srcObject = remoteStream;
      console.log("Adding remote source");
    } else {
      videoElement.current.srcObject = null;
      console.log("Removing video source");
    }
    // console.log(videoElement.current);
  }
}, [localStream, remoteStream]);

useEffect(() => {
  // Clear interval when the component unmounts
  return () => {
    clearTimeout(talkTimerRef.current);
    if (currentRoom) removeFromQueue(currentRoom);
  };
}, []);

useEffect(() => {
  console.log("UPDATED ROOMS INFO IN CHATS LAYOUT", isMyTurn(currentRoom));
  console.log("ROOMS INFO IN CHATS LAYOUT", { roomsInfo, currentRoom });
  setCanTalk(isMyTurn(currentRoom));
}, [roomsInfo]);

useEffect(() => {
  console.log(`CAN${canTalk ? "" : "NOT"} TALK`);
  if (canTalk) {
    console.log("CAN TALK PUBLISH OWN FEED");
    publishOwnFeed(true, true);
```

```
    talkTimerRef.current = setTimeout(() => {
      console.log("You took to much time to finish");
      unpublishOwnFeed();
      removeFromQueue(currentRoom);
    }, 20000);
  } else {
    clearTimeout(talkTimerRef.current);
  }
}, [canTalk]);

useEffect(() => {
  console.log("Chat messages", messages);
  if (currentRoom) scrollToBottom();
}, [messages]);

useEffect(() => {
  let rooms = usersInfo[user.uid]?.rooms;
  console.log({ rooms });
  if (rooms) {
    setMyRooms(Object.values(rooms));
  }
}, [usersInfo]);

const scrollToBottom = () => {
  if (messagesEnd) messagesEnd.scrollIntoView({ behavior: "smooth" });
};

// Send message
const sendMessage = (e) => {
  // publishOwnFeed(true, true);
  console.log("Sending message");
  let message = messageInput.current.value;
  if (message) {
    console.log(message);
    // Send message
    sendData(message);
    removeFromQueue(currentRoom);
    // Message sent
    messageInput.current.value = "";
  }

  unpublishOwnFeed();
};

const handleActivityClick = (e) => {
  console.log("Checking for queue");
  let onQueue = isOnQueue(currentRoom);
  if (!onQueue) {
    addToQueue(currentRoom);
  } else {
    console.log("Already on queue");
  }
};
```

```
// Send message when pressed enter
const checkEnter = (e) => {
  let theCode = e.keyCode ? e.keyCode : e.which ? e.which : e.charCode;
  if (theCode == 13) {
    sendMessage();
  } else {
    clearTimeout(talkTimerRef.current);
    talkTimerRef.current = setTimeout(() => {
      console.log("You took to much time to finish");
      unpublishOwnFeed();
      removeFromQueue(currentRoom);
    }, 20000);
  }
};

// Get queue for all participants in a room, whether on queue or not
const getQueueInfo = () => {
  let queue = roomsInfo[currentRoom]?.queue;
  let initial = [];

  console.log({ participants, queue });

  // Add information about queue order
  for (let p in participants) {
    let indexInQueue = -1;
    console.log({ p, queue });
    if (queue) indexInQueue = Object.values(queue)?.indexOf(p);

    console.log({ p, indexInQueue });

    if (indexInQueue < 0) indexInQueue = Object.keys(participants).length + 1;
    console.log({ p, indexInQueue });

    initial = [
      ...initial,
      {
        queuePos: indexInQueue + 1,
        queueUserDisplay: participants[p],
        queueUserId: p,
      },
    ];
  }

  let initialSorted = initial.sort((a, b) => {
    return a.queuePos - b.queuePos;
  });

  // Assing UI positions on list
  // let finalOrder = initialOrder.map(
  //   (item) => {
  //     this.acc++;
  //     return {
```

```
  //       ...item,
  //       listKey: this.acc,
  //     };
  //   },
  //   { acc: 1 }
  // );

  let final = [];
  initialSorted.forEach((item) => {
    let lastPos = Object.keys(participants).length + 1;
    if (item.queuePos >= lastPos) {
      item.queuePos = null;
    }

    final = [...final, item];
  });

  console.log({ final });

  return final;
};

return (
  <>
    <div>
      <Grid container component={Paper} className={classes.chatSection}>
        <Grid item xs={3} className={classes.borderRight500}>
          <ProfileMenu user={user} />
          <Divider />
          {/* Search function */}
          <Grid item xs={12} style={{ padding: "10px" }}>
            <TextField
              id="outlined-basic-email"
              label="Search"
              variant="outlined"
              fullWidth
            />
          </Grid>
          <Divider />
          {/* Join or create buttons */}
          <Grid container item xs={12} style={{ padding: "10px" }}>
            <Grid item xs={6} style={{ padding: "10px" }}>
              <CreateRoomButton />
            </Grid>

            <Grid item xs={6} style={{ padding: "10px" }} align="right">
              <JoinRoomButton />
            </Grid>
          </Grid>
          <Divider />

          {/* Rooms that user joined */}
          <List>
```

```
              {myRooms
                ? myRooms.map((roomId) => {
                    let roomDetail = roomsInfo[roomId];
                    return (
                      <ListItem
                        button
                        key={roomId}
                        onClick={() => {
                          console.log({ roomId });
                          joinVideoRoom(roomId);
                        }}
                      >
                        <ListItemIcon>
                          <Avatar
                            alt={roomDetail?.description || `Room ${roomId}`}
                            src="https://material-
ui.com/static/images/avatar/1.jpg"
                          />
                        </ListItemIcon>
                        <ListItemText
                          primary={roomDetail?.description || `Room ${roomId}`}
                        >
                          {roomDetail?.description || `Room ${roomId}`}
                        </ListItemText>
                      </ListItem>
                    );
                  })
                : null}
            </List>
          </Grid>

          <Grid item xs={7}>
            {/**** Message History *****/}
            {currentRoom && (
              <>
                <List className={classes.messageArea}>
                  {messages &&
                    messages.map(({ text, author, timestamp }, index) => (
                      <ListItem
                        key={index}
                        autoFocus={index === messages.length}
                      >
                        <Message author={author} timestamp={timestamp}>
                          {text}
                        </Message>
                      </ListItem>
                    ))}
                  <div
                    style={{ float: "left", clear: "both" }}
                    ref={(el) => {
                      messagesEnd = el;
                    }}
                  ></div>
```

```
            </List>

            <Message author={getQueueInfo()[0]}>
              <video autoPlay ref={videoElement} />
            </Message>

            <Divider />

            <Grid container style={{ padding: "20px" }}>
              {canTalk && (
                <>
                  <Grid item xs={9}>
                    <TextField
                      id="message-text"
                      placeholder="Enter message"
                      fullWidth
                      inputRef={messageInput}
                      onKeyPress={checkEnter}
                      disabled={!canTalk}
                      // autoFocus={canTalk}
                    />
                  </Grid>
                  <Grid item xs={3} align="right">
                    <IconButton
                      variant="contained"
                      color="primary"
                      aria-label="audio"
                      onClick={(e) => {
                        e.preventDefault();
                        toggleAudio();
                      }}
                    >
                      {audioPlaying ? (
                        <>
                          <MicIcon />
                        </>
                      ) : (
                        <>
                          <MicOffIcon />
                        </>
                      )}
                    </IconButton>
                    <IconButton
                      variant="contained"
                      color="primary"
                      aria-label="video"
                      onClick={(e) => {
                        e.preventDefault();
                        toggleVideo();
                      }}
                    >
                      {videoPlaying ? (
                        <>
```

```
                        <VideocamIcon />
                      </>
                    ) : (
                      <>
                        <VideocamOffIcon />
                      </>
                    )}
                  </IconButton>
                  <IconButton
                    variant="contained"
                    color="primary"
                    aria-label="message"
                    onClick={sendMessage}
                  >
                    <SendIcon />
                  </IconButton>
                </Grid>
              </>
            )}
            {!canTalk && (
              <>
                <Button
                  variant="contained"
                  color={isOnQueue(currentRoom) ? "warning" : "primary"}
                  aria-label="request-write"
                  onClick={handleActivityClick}
                  fullWidth
                >
                  {isOnQueue(currentRoom) ? (
                    <>
                      <CircularProgress
                        sx={{
                          color: "white",
                        }}
                      />
                    </>
                  ) : (
                    "Request to talk"
                  )}
                </Button>
              </>
            )}
          </Grid>
        </>
      )}
    </Grid>

    <Grid item xs={2} className={classes.borderLeft500}>
      <Typography variant="h5">Participants</Typography>

      <Divider />

      {/* Queue for speaking */}
```

```
            <List>
              {participants && (
                <>
                  {getQueueInfo().map(
                    ({ queuePos, queueUserDisplay, queueUserId }, key) => (
                      <ListItem key={key}>
                        <ListItemAvatar>
                          <Avatar
                            sx={{
                              bgcolor: queuePos ? "primary.main" : "default",
                            }}
                          >
                            {queuePos ? queuePos : "-"}
                          </Avatar>
                        </ListItemAvatar>
                        <ListItemText
                          primary={queueUserDisplay}
                          secondary={queueUserId == user.uid ? "(me)" : ""}
                        ></ListItemText>
                      </ListItem>
                    )
                  )}
                </>
              )}
            </List>
          </Grid>
        </Grid>
      </div>
    </>
  );
}

function ModalDialog({ open }) {
  return (
    <Modal
      open={open}
      aria-labelledby="modal-modal-title"
      aria-describedby="modal-modal-description"
    >
      <Box>
        <Typography id="modal-modal-title" variant="h6" component="h2">
          Text in a modal
        </Typography>
        <Typography id="modal-modal-description" sx={{ mt: 2 }}>
          Duis mollis, est non commodo luctus, nisi erat porttitor ligula.
        </Typography>
      </Box>
    </Modal>
  );
}

function CreateRoomButton() {
  const { createVideoRoom } = useContext(JanusContext);
```

```
  const handleClick = () => {
    createVideoRoom();
  };

  return (
    <Button
      variant="contained"
      color="success"
      aria-label="create-room"
      fullWidth
      onClick={handleClick}
    >
      Create
    </Button>
  );
}

function JoinRoomButton() {
  const { joinVideoRoom } = useContext(JanusContext);
  const handleClick = () => {
    let roomNumber = window.prompt("Please type in the room number");
    joinVideoRoom(parseInt(roomNumber));
  };

  return (
    <Button
      variant="contained"
      color="primary"
      aria-label="join-room"
      fullWidth
      onClick={handleClick}
    >
      Join
    </Button>
  );
}
```