

ESCUELA DE INGENIERÍA DE  
TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

**Aplicación web multimedia con sistema de anotación  
de contenido de las intervenciones usando la  
tecnología Blockchain**

**Titulación:** Grado en Ingeniería en Tecnologías de la Telecomunicación

**Mención:** Telemática

**Autor:** Sr. Rubén Delgado González

**Tutores:** Sr. Álvaro Suárez Sarmiento  
Sra. Elsa María Macías López

**Fecha:** Junio 2021



ESCUELA DE INGENIERÍA DE  
TELECOMUNICACIÓN Y ELECTRÓNICA



**Aplicación web multimedia con sistema de anotación  
del contenido de las intervenciones usando la  
tecnología Blockchain**

**HOJA DE EVALUACIÓN**

Calificación: \_\_\_\_\_

**Presidente**

Fdo.: \_\_\_\_\_

**Vocal**

**Secretario/a**

Fdo.: \_\_\_\_\_

Fdo.: \_\_\_\_\_

Las Palmas de Gran Canaria, a 25 de junio de 2021



## Resumen

Actualmente, es muy común el uso de aplicaciones de videoconferencia y de comunicación multimedia ya que permiten a organizaciones de cualquier ámbito realizar reuniones de forma telemática. Esto permite que los usuarios no tengan que desplazarse para tener una reunión física presencial, lo cual ha sido especialmente necesario durante la situación actual de pandemia. Además, en muchas de las reuniones telemáticas existe la necesidad de registrar, de forma unívoca e inalterable, las intervenciones realizadas por los participantes de la reunión.

Para dar solución a este problema, se pretende el desarrollo de una aplicación web de comunicación multimedia, que pueda usarse en diferentes dispositivos y navegadores, que permita realizar reuniones telemáticas interactivas en tiempo real. Para guardar las intervenciones que realicen los usuarios, se plantea un sistema que permita un registro consensuado (en principio) de las intervenciones de los usuarios, así como una plataforma que permita asegurar la autenticidad de las intervenciones almacenadas. Para asegurar la integridad de las intervenciones almacenadas, se plantea la implementación de una *blockchain* prototipo que guarde información bajo demanda de los usuarios que permita verificar la autenticidad de aquellas intervenciones que hayan sido registradas en el sistema.

De las pruebas realizadas, y el análisis de la usabilidad de la aplicación y el rendimiento de la comunicación multimedia, se infiere que el sistema desarrollado permite el desarrollo de reuniones telemáticas y ofrece un respaldo unívoco de las intervenciones realizadas durante dichas reuniones.

## **Abstract**

Currently, the use of videoconferencing applications is very common, as they allow organizations of any kind to hold meetings remotely. This means that users do not need to move to have a physical face-to-face meeting, which has been especially necessary during the current pandemic situation in which telepresence prevails due to social distancing. In addition, in many telematic meetings there is a need to record, in an univocal way, the interventions made by the users of the meeting.

To solve this problem, we intend to develop a videoconferencing web application, which can be used in different devices and browsers, that allows interactive remote meetings to be held in real time. To store the interventions made by users, a system is proposed that allows a consensual record of user's interventions, as well as a platform to ensure the authenticity of the stored interventions. To ensure the integrity of the stored interventions, we propose the implementation of a prototype blockchain that stores information on demand to verify the authenticity of those interventions that have been registered in the system.

From the tests that have been carried out, and the analysis of the usability of the application and the performance of the multimedia communication, it is inferred that the developed system allows the development of telematic meetings and offers an univocal storage of the interventions made during these meetings.

# Índice

Resumen .....	2
Abstract .....	3
Índice .....	4
<b>1. Introducción .....</b>	<b>8</b>
1.1. Antecedentes.....	9
1.2. Objetivos.....	10
1.3. Ámbito y evolución del TFG.....	11
1.4. Estructura de la memoria .....	14
<b>2. Análisis del problema .....</b>	<b>16</b>
2.1. Introducción.....	17
2.2. Reuniones telemáticas mediante videoconferencia .....	19
2.3. Actas de reuniones y registro de intervenciones .....	21
2.4. Verificación de la autenticidad del registro de intervenciones.....	24
<b>3. Análisis de requisitos y funcional .....</b>	<b>26</b>
3.1. Introducción.....	27
3.2. Análisis de requisitos .....	28
3.3. Casos de uso .....	30
3.3.1 Casos de uso de la aplicación web de comunicación multimedia .....	30
3.3.2. Casos de uso de <i>blockchain</i> .....	35
3.4. Análisis funcional .....	38
3.5. Análisis estructural .....	41
<b>4. Análisis de la implementación del sistema .....</b>	<b>46</b>
4.1. Introducción.....	47
4.2. Análisis, diseño e implementación del sistema de autenticación de usuarios .....	51
4.3. Análisis, diseño e implementación de gestión de usuarios.....	54
4.4. Análisis, diseño e implementación de reuniones multimedia .....	57
4.4.1. Estructuras de una reunión .....	58
4.4.2. Creación de una reunión multimedia .....	59
4.4.3. Notificación de usuarios.....	62

4.4.4. Acceso a una reunión multimedia .....	64
4.4.5. Señalización de comunicación multimedia .....	67
4.5. Análisis, diseño e implementación del sistema de registro de intervenciones .....	74
4.5.1. Estructura de una intervención.....	75
4.5.2. Sistema de grabación de intervenciones .....	78
4.5.3. Sistema de consenso de intervenciones .....	83
4.5.4. Consulta de intervenciones realizadas.....	89
4.6. Análisis de la blockchain implementada .....	94
4.6.1. Estructuras de datos.....	97
4.6.2. Estructura de la red blockchain.....	103
4.6.3. Gestión de la <i>blockchain</i> .....	107
4.6.4. Algoritmo de consenso.....	111
4.7. Análisis, diseño e implementación de la verificación de intervenciones.....	114
4.7.1. Inserción de información de intervenciones en la blockchain .....	115
4.7.2. Verificación de intervenciones.....	121
<b>5. Análisis de interfaz de usuario y resultados .....</b>	<b>126</b>
5.1. Interfaz de usuario de la aplicación web multimedia .....	127
5.1.1 Ejemplos de uso .....	137
5.1.2. Interfaz de la aplicación web en dispositivos móviles .....	146
5.2. Interfaz de usuario de administración de la blockchain: Nodo raíz.....	147
5.3. Evaluación de resultados experimentales.....	152
5.3.1. Rendimiento de la comunicación multimedia .....	152
5.3.2. Resultados de la blockchain implementada .....	157
5.3.3. Verificación de intervenciones realizadas.....	162
<b>6. Conclusiones .....</b>	<b>165</b>
6.1. Conclusiones.....	166
6.2. Posibles ampliaciones.....	167
<b>Bibliografía.....</b>	<b>169</b>
<b>Pliego de condiciones .....</b>	<b>181</b>
Pl.1. Condiciones hardware .....	182
Pl.2. Condiciones software .....	182
Pl.3. Condiciones de uso por parte del usuario.....	183



Pl.4. Condiciones de licencia .....	183
Pl.5. Derechos de autor .....	184
Pl.6. Restricciones.....	184
Pl.7. Garantía .....	184
Pl.8. Limitación de responsabilidad.....	184
<b>Presupuesto .....</b>	<b>186</b>
Pr.1. Componentes del presupuesto.....	187
Pr.2. Recursos materiales .....	187
Pr.3. Trabajo tarifado por tiempo empleado.....	189
Pr.4. Redacción del trabajo .....	189
Pr.5. Material fungible.....	190
Pr.6. Derechos de visado del COITT.....	190
Pr.7. Gastos de tramitación y envío .....	191
Pr.8. Aplicación de impuestos y coste total .....	191
<b>Anexos .....</b>	<b>194</b>
<b>Anexo A. Herramientas utilizadas .....</b>	<b>196</b>
A1. JavaScript.....	197
A2. Node.js.....	197
A3. React.....	198
A4. Redis .....	200
A5. Heroku .....	200
A6. Firebase .....	201
A7. Postman.....	203
A8. Bibliotecas utilizadas .....	204
<b>Anexo B. WebRTC.....</b>	<b>210</b>
B1. Introducción a WebRTC.....	211
B2. Captura de contenido multimedia en el navegador .....	211
B3. Establecimiento de conexión .....	213
B3.1. Servidores TURN y STUN .....	215
B3.2. RTCPeerConnection .....	217
B4. Proceso de señalización .....	217
<b>Anexo C. Blockchain .....</b>	<b>220</b>

C1. Definición y conceptos básicos.....	221
C2. Mecanismos de consenso .....	222
C2.1. Proof of Work.....	223
C2.2. Proof of Stake.....	224
C2.3. Proof of Authority .....	224
C3. Tipos de blockchain .....	225
C4. Tokenización.....	227
<b>Anexo D. Proceso de señalización de la comunicación multimedia .....</b>	<b>228</b>
D.1 Señalización.....	229
<b>Anexo E. Aspectos no funcionales de la arquitectura de software .....</b>	<b>233</b>
E1. Análisis de la responsividad de la interfaz de usuario.....	234
E2. Despliegue en Heroku.....	236
<b>Glosario .....</b>	<b>240</b>

# 1. Introducción

---

El objetivo de este capítulo es exponer el ámbito de este *Trabajo de Fin de Grado (TFG)*, presentando una idea general del sistema desarrollado y los objetivos de su implementación. Además, se presentan algunas nociones básicas de las tecnologías usadas y los elementos que han motivado la realización del TFG, así como la estructura de la memoria.

## 1.1. Antecedentes

Actualmente, es muy común el uso de aplicaciones de videoconferencia, telefonía y comunicación multimedia en general. Estas aplicaciones proporcionan ventajas a sus usuarios los cuales no tienen necesidad de desplazarse para tener una reunión física fuera de sus puestos de trabajo o de su propio hogar. Por este motivo, el desarrollo de aplicaciones web multimedia es muy importante y, durante la situación actual de pandemia en la que prima la tele-presencialidad, su uso ha aumentado de manera exponencial [1].

El Mercado de las aplicaciones de videoconferencia ha aumentado notablemente en los últimos años. Aplicaciones de comunicación multimedia como *Zoom* [2], *Google Meet* [3] o *Microsoft Teams* [4], entre otros, han tenido un éxito enorme en los últimos años, especialmente durante el periodo de pandemia. Por ejemplo, *Microsoft Teams* ha sido una de las aplicaciones que más ha crecido durante la pandemia, siendo utilizada por más de 500.000 organizaciones y contando con 145 millones de usuarios conectados al día [5]. El aumento del uso de esta plataforma es uno entre muchos de los ejemplos existentes que muestran el rápido crecimiento de las aplicaciones de esta índole.

Además, el dominio de aplicación de las videoconferencias es muy grande. Esto es, hoy en día las videoconferencias se utilizan para: enseñanza (impartición de clases online), medicina (telemedicina y consultas remotas con los pacientes), reuniones de empresas y organizaciones, como comunidades de vecinos, juntas de propietarios... Justamente, en algunos de estos dominios de aplicación, surge la necesidad imperiosa de registrar la intervención de alguno de los participantes en la videoconferencia. Para ello, estas aplicaciones deberían permitir guardar aquellas intervenciones relevantes realizadas durante las reuniones, asegurando un respaldo eficiente y seguro para consultar dichas intervenciones. Además, sería necesario poder garantizar la autenticidad de las intervenciones consultadas, asegurando que el contenido de las mismas es el mismo que se ha realizado en la reunión y no ha sido modificado.

Para ello, la tecnología *Blockchain* ofrece diferentes mecanismos que pueden resultar especialmente útiles para garantizar un respaldo inmutable de las intervenciones realizadas. Esta tecnología se puede considerar, de forma resumida, que es un registro de información descentralizado unívoco e inalterable [6]. Cada bloque de la cadena almacena

un conjunto de información en la red, los cuales deben ser comprobados y aceptados por los nodos mineros de la red mediante un algoritmo de consenso previamente especificado [7]. De esta forma, se asegura que toda la información almacenada en la *blockchain* ha sido completamente validada mediante un método neutral en el que no pueden intervenir agentes externos, y donde dicha información (que compone la *blockchain* en sí misma) se almacena de forma íntegra e inmutable, encontrándose disponible en todo momento.

Dicha descentralización e inmutabilidad de los datos son características de gran utilidad en muchos campos en los que se requiere un registro inalterable de información, en el que dichos datos no pueden ser modificados o eliminados fácilmente o incluso se dificulta tanto que sería prácticamente imposible hacerlo [8]. Es por ello por lo que, gracias a dichas propiedades, se haya extendido su aplicación en sistemas de propiedad intelectual, procesos electorales, verificación de noticias falsas...

Estas propiedades de la tecnología *Blockchain* la hacen especialmente útil para respaldar y verificar la autenticidad del almacenado de las intervenciones de los participantes en la videoconferencia.

## **1.2. Objetivos**

En nuestro caso, estamos interesados en el desarrollo de aplicaciones web para videoconferencia multimedia que permita guardar, mediante un sistema de anotaciones de las intervenciones realizadas, contenidos que indiquen los usuarios (y consensuen el resto), integrando la tecnología *Blockchain* para garantizar la integridad de dichas intervenciones. Por lo tanto, en este TFG se pretende: estudiarla en profundidad y analizar su uso en el ámbito de las aplicaciones multimedia web con control de intervenciones del usuario, para desarrollar una aplicación web multimedia y una *blockchain* prototipo cuya funcionalidad se ajuste a dicho campo de trabajo. Hasta dónde llega nuestro conocimiento, no existe ninguna aplicación similar que disponga de las mismas funciones que la que deseamos implantar.

El objetivo principal de este TFG es el diseño e implementación de una aplicación web de comunicación multimedia que posea un sistema que permita guardar y almacenar

intervenciones, junto a una *blockchain* que se ajuste a dicho sistema y permita almacenar de forma unívoca el contenido de las intervenciones almacenadas.

Este objetivo general se lleva a cabo mediante la consecución de los diferentes objetivos operativos detallados a continuación:

- **O1.** Diseño e implantación de una aplicación web multimedia prototipo de videoconferencia multi-parte interactiva y en tiempo real. Esta aplicación también debe tener capacidades responsivas. Esto es, se debe ejecutar en un terminal de cualquier tipo: teléfono móvil, tableta, portátil, computador de sobremesa... Debe tener una interfaz de usuario simple y minimalista que facilite su uso.
- **O2.** Diseño e implantación de una *blockchain* específica para el problema concreto de verificación de la persistencia coherente de contenidos de intervenciones de los usuarios en la aplicación de videoconferencia del objetivo anterior. En esta *blockchain* se guardaría información a demanda que permita verificar la correctitud de aquellas intervenciones que un videoconferencista desee registrar, y en principio, el resto de los videoconferencistas estén de acuerdo.
- **O3.** Realización de pruebas para verificar el funcionamiento correcto de la aplicación web y la *blockchain*. Esto es, estamos interesados en observar tanto, el rendimiento de la aplicación web, como el de la red de comunicación, así como el de la usabilidad de la misma. Por otro lado, es importante verificar que la aplicación realiza un registro consensuado (en principio), de las intervenciones de los usuarios, y que el sistema de respaldo o verificación de la veracidad de esas intervenciones funciona correctamente a través de la *blockchain*.

### **1.3. Ámbito y evolución del TFG**

El ámbito de este TFG es el diseño de aplicaciones de videoconferencia web usando nuevas tecnologías web responsivas para adaptar el funcionamiento a distintas pantallas, teniendo en cuenta tecnologías para registrar eventos de forma inalterable. En este TFG no se analizan tecnologías empresariales concretas de videoconferencias sobre redes específicas (como las redes telefónicas) y si estamos interesados en Internet. Además, no se propone

una aplicación profesional sino un prototipo que nos permita ensayar soluciones novedosas a los problemas concretos de registro de eventos bajo demanda del usuario. Tampoco procede la comparación de nuestro prototipo con otras herramientas profesionales de videoconferencia web, porque no estamos interesados en su rendimiento, sino en ensayar nuevas ideas de registro de eventos de la propia videoconferencia. Nuestro prototipo no incluye tratamiento de archivos, ni pizarras compartidas porque está dedicada exclusivamente al registro de eventos de audio de la videoconferencia.

Oficialmente, un TFG tiene asignada una cantidad de 300 horas para su completa realización dentro del plan de estudio del grado de Ingeniería en Tecnologías de Telecomunicación. Como se especificó en el anteproyecto, se proponía lograr unos objetivos concretos, que en la Tabla 1.1 mencionamos (exceptuando las tareas relacionadas con redacción de la memoria TFG, el cual no está recogido en la Tabla 1.1). En la evolución del TFG se realizaron tareas de investigación y documentación de las tecnologías que se usarían en el desarrollo del mismo. Se desarrollaron prototipos que se han ido realizando mediante iteraciones e incrementos a partir de un esbozo inicial y finalmente se logró conseguir una aplicación web que cumplió con todos los objetivos.

<b>Tarea</b>	<b>Fecha de inicio y finalización</b>	<b>Objetivos</b>	<b>Horas</b>
<b>Documentación sobre tecnología <i>blockchain</i></b>	8/2/2021 al 5/2/2021	O2	20
<b>Desarrollo de nodo de <i>blockchain</i><sup>1</sup></b>	15/2/2021 al 17/2/2021	O2	10
<b>Desarrollo de nodo raíz de <i>blockchain</i></b>	17/2/2021 al 23/2/2021	O2	15
<b>Desarrollo de interfaz de panel de administración del nodo raíz<sup>2</sup></b>	23/2/2021 a 25/2/2021	O2	10

Despliegue en la nube de la <i>blockchain</i> <sup>3</sup>	25/2/2021 - 26/2/2021	O2	5
Documentación sobre <i>WebRTC</i>	1/3/2021 al 5/3/2021	O1	20
Implementación de prototipo de plataforma de comunicación multimedia multi-parte usando <i>WebRTC</i>	8/3/2021 al 16/3/2021	O1	25
Desarrollo y despliegue de aplicación web <sup>3</sup> ( <i>frontend y backend</i> )	16/3/2021 al 25/3/2021	O1	30
Implementación de sistema de autenticación de usuarios <sup>4</sup>	25/3/2021 al 25/3/2021	O1	1
Implementación de la plataforma de comunicación multimedia multi-parte en <i>React</i>	26/3/2021 al 30/3/2021	O1	10
Desarrollo del sistema de grabación y almacenamiento de intervenciones	30/3/2021 al 9/4/2021	O1	20
Diseño y desarrollo del mecanismo de consenso de validación de intervenciones realizadas	9/4/2021 al 16/4/2021	O1	20
Integración de la <i>blockchain</i> con el sistema de almacenamiento de intervenciones	16/4/2021 al 23/4/2021	O1 y O2	20
Desarrollo del sistema de verificación de intervenciones	23/4/2021 al 30/4/2021	O1 y O2	20
Desarrollo de sistema de gestión de usuarios <sup>4</sup>	30/4/2021 al 3/5/2021	O1	4



<b>Adaptación de interfaz responsiva para dispositivos móviles</b>	3/5/2021 al 4/5/2021	O1	5
<b>Realización de pruebas de rendimiento de la comunicación multimedia<sup>5</sup></b>	4/5/2021 al 6/5/2021	O3	10
<b>Realización de pruebas del sistema de verificación de intervenciones<sup>5</sup></b>	7/5/2021 al 12/5/2021	O3	15

*Tabla 1.1. Tareas realizadas para el desarrollo del TFG*

<sup>1</sup> El desarrollo de la interfaz de administración de la *blockchain* surgió durante el desarrollo de la *blockchain* prototipo, ya que requería de un sistema que permitiera gestionar dicha *blockchain* por parte de un usuario administrador mediante un panel de administración.

<sup>2</sup> Las fechas previstas para el desarrollo de la *blockchain* prototipo se adelantaron respecto a las previstas en el anteproyecto debido a que fue más conveniente su desarrollo tras realizar las tareas de documentación sobre la tecnología *Blockchain*. Con esto, se retrasaron las fechas para la realización la tarea de documentación de WebRTC tras terminar la implementación de la *blockchain* prototipo.

<sup>3</sup> Tanto la *blockchain* desarrollada como la aplicación web fueron desplegadas en la nube (mediante *Heroku*) para probar su funcionamiento en un entorno real, lo cual no fue previsto en el anteproyecto.

<sup>4</sup> El sistema de autenticación y gestión de usuarios no fue previsto en el anteproyecto, ya que surgió la necesidad de su implementación durante el desarrollo del sistema de videoconferencia multi-parte para la creación y control de acceso de las reuniones multimedia.

<sup>5</sup> El tiempo estimado para el desarrollo de las pruebas y análisis del rendimiento del sistema fue mayor del esperado. No obstante, debido al tiempo ahorrado en la realización de tareas de otras partes del sistema, se pudo invertir más tiempo en realizar las tareas relacionadas con el testeo y análisis del sistema desarrollado.

## **1.4. Estructura de la memoria**

En el capítulo 1 se comentan las líneas generales del problema planteado y la solución propuesta, describiendo la motivación del TFG y los objetivos de este.

En el capítulo 2 se trata con profundidad el problema que se plantea en TFG y que se pretende resolver en este proyecto.

En el capítulo 3 se realiza un análisis del sistema implementado, describiendo en detalle las funcionalidades de dicho sistema para resolver los problemas indicados en el capítulo 2.

En el capítulo 4 se explica en detalle el diseño, implementación y desarrollo del sistema desarrollado, explicando cómo se ha realizado la funcionalidad del sistema en este TFG. Para ello, se ha tenido en cuenta todo el análisis de la arquitectura propuesta en el capítulo 3.

En el capítulo 5 se analizan el uso de la aplicación web implementada, mostrando las diferentes partes de la interfaz de usuario. A su vez, se evalúa el rendimiento y resultados obtenidos del sistema desarrollado.

En el capítulo 6 se exponen las conclusiones a las que se ha llegado tras el desarrollo del proyecto, comentando posibles ampliaciones y mejoras que se podrían añadir al proyecto en el futuro.

En los últimos capítulos muestran el pliego de condiciones y un análisis del coste económico del desarrollo e implementación del proyecto.

En relación con los anexos, en el anexo A se describen las tecnologías y herramientas más relevantes utilizadas en el proyecto. En el anexo B se explica el funcionamiento de *Web Real-Time Communications (WebRTC)*. En el anexo C se expone las principales características y nociones básicas de la tecnología *Blockchain*. En el anexo D se explica el proceso de señalización diseñado para el establecimiento de la comunicación multimedia mediante WebRTC. Por último, en el anexo E se comentan algunos aspectos no funcionales de la arquitectura de software desarrollada.

## 2. Análisis del problema

---

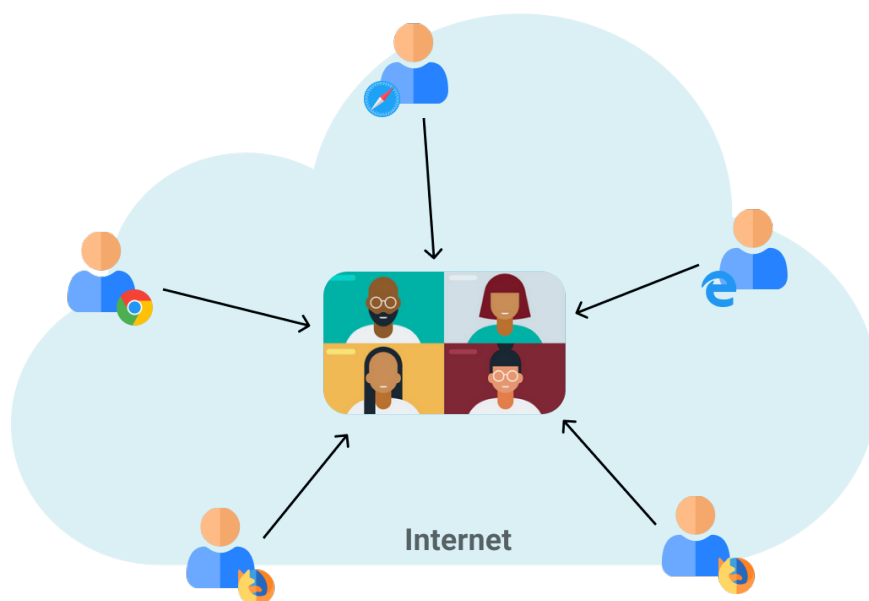
En este capítulo se presenta el problema que se va a resolver describiendo la situación de las reuniones telemáticas y la necesidad de guardar constancia de acciones y acuerdos que realicen en las mismas.

## 2.1. Introducción

En el objetivo principal de este TFG se contempla el almacenamiento de las intervenciones de un participante en una videoconferencia en la cual se están tratando temas de interés para todos (por ejemplo, una reunión de vecinos) y que deben ser guardadas y no alteradas para que quede constancia eficaz de su intervención. En la Figura 2.1 se muestra una idea gráfica de la videoconferencia que establecerían diferentes vecinos para la realización telemática de una reunión en su comunidad de vecinos.

En la Figura 2.2 se muestra que debe existir un software servidor que se encargue de gestionar la solicitud de registro de la intervención (para cualquier vecino), y ello significa que al menos debe existir una plataforma de almacenamiento secundaria persistente que sea capaz de almacenar dichas intervenciones. Esto permitiría a los usuarios grabar las intervenciones que crean relevantes y puedan ser almacenadas para su posterior consulta.

Por otro lado, para poder verificar la autenticidad de las intervenciones almacenadas en el almacenamiento secundario, debe existir un sistema que sea capaz de comprobar la integridad de dichas intervenciones cuando sean consultadas. Este proceso de verificación se muestra en la Figura 2.3. Esto permitiría a los usuarios de conocer que las intervenciones almacenadas en el sistema están respaldadas de forma segura.



*Figura 2.1. Reuniones multimedia entre usuarios*

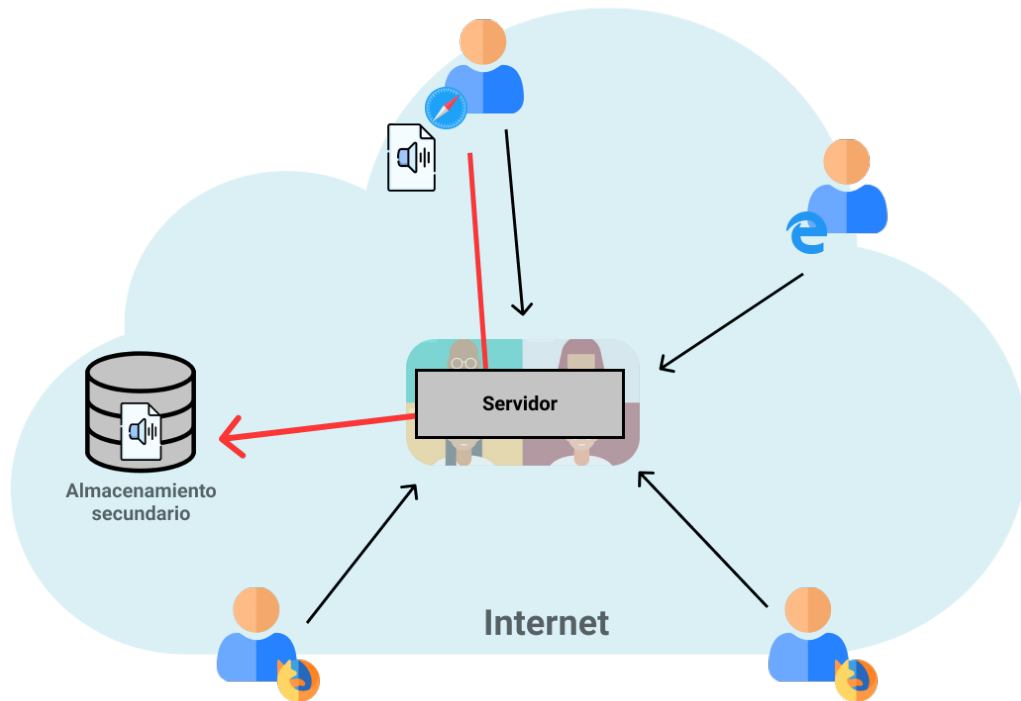


Figura 2.2. Almacenamiento de intervención realizada

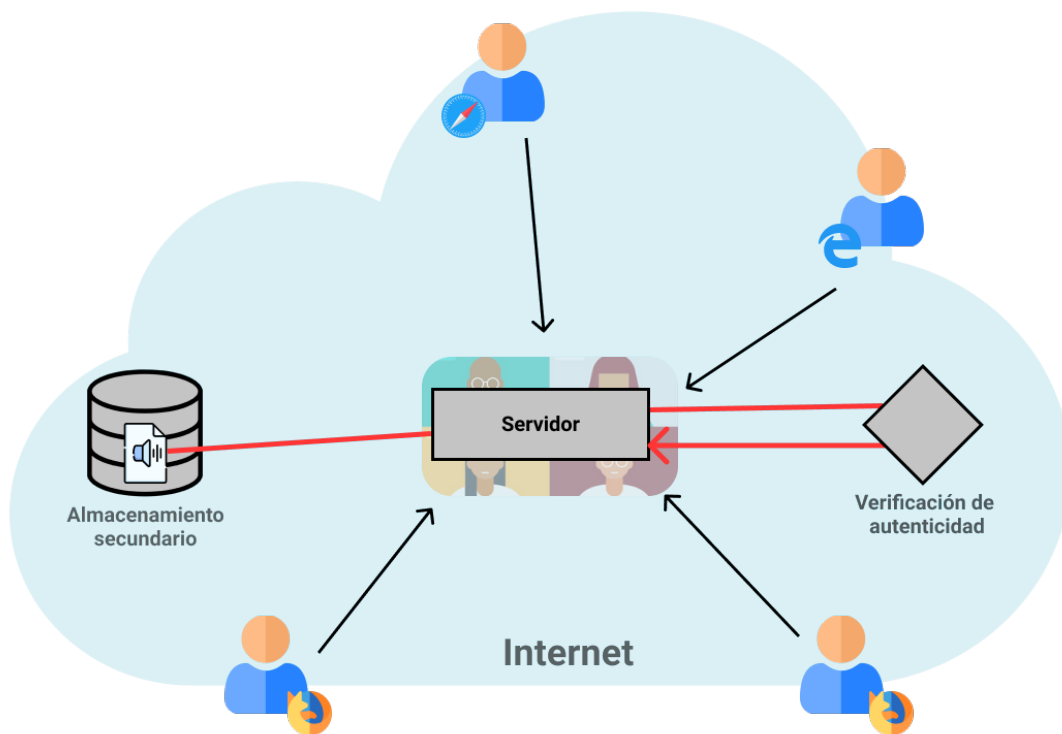


Figura 2.3. Verificación de una intervención.

## **2.2. Reuniones telemáticas mediante videoconferencia**

Las aplicaciones multimedia y de videoconferencia permiten a organizaciones de cualquier ámbito realizar reuniones de forma telemática. Esto permite que los usuarios no tengan que desplazarse para tener una reunión física presencial, ya que pueden realizar dichas reuniones a través de plataformas que permitan realizar reuniones multimedia telemáticas.

Con la situación actual de pandemia, en la que prima la tele-presencialidad debido al distanciamiento social, el uso de estas aplicaciones se ha convertido en una necesidad para muchas empresas. De igual manera, las aplicaciones multimedia son una herramienta esencial en puestos de trabajo remotos, los cuales han aumentado considerablemente estos últimos años [9]. Entre 2019 y 2022, se espera un aumento del 77% del teletrabajo [10]. Cada vez vivimos en un Mundo más conectado, y el teletrabajo ha llegado a muchas empresas y organizaciones para quedarse, incluso pasada la situación de pandemia, por lo que el uso de las aplicaciones de comunicación multimedia se estima que seguiría en aumento. De hecho, el Mercado de aplicaciones de videoconferencia, el cual estaba en una trayectoria en alza antes de la pandemia, se estima seguiría creciendo entre 2018 y 2023 [11].

Además, las aplicaciones de comunicación multimedia descritas, cuando son utilizadas para la realización de reuniones telemáticas, necesitan tener un sistema que permita convocar y organizar dichas reuniones. Esto permite a las organizaciones y empresas tener una plataforma que, además de permitir que sus participantes puedan comunicarse de forma multimedia, pueda ser capaz de registrar cuando se celebrarían las reuniones y notificar a los participantes sobre ello, ya sea a través de correo electrónico u otros medios que permitan informar al usuario.

Es por ello por lo que, en la gran mayoría de plataformas de videoconferencia, es esencial un sistema que permita convocar las reuniones telemáticas previamente a su celebración. Como se muestra en la Figura 2.4, mediante dicho sistema de notificación los usuarios podrían recibir, no solo cuándo se ha convocado la reunión, sino que recibirían la información necesaria para acceder a la misma (como un enlace de acceso).

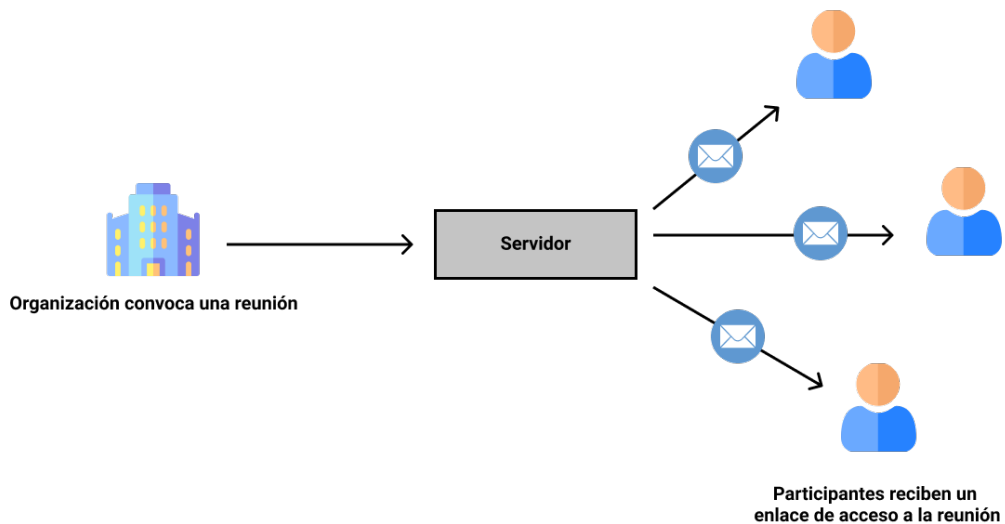


Figura 2.4. Notificación de creación de reuniones a los participantes

No obstante, para que el sistema descrito sea capaz de notificar a los usuarios que participarían en la reunión, este debe tener registrados a dichos usuarios (los cuales pertenecerían a la organización o empresa que convoca la reunión). De esta manera, además de poder comunicar a los usuarios sobre la realización de las reuniones, se permitiría garantizar la seguridad de acceso a las reuniones convocadas.

Si la plataforma de comunicación multimedia no implementara un sistema de registro y autenticación de usuarios, un usuario externo podría acceder a una reunión si obtuviera el enlace de acceso a la misma, aunque no hubiera sido invitado a la reunión como se muestra en la Figura 2.5. Por este motivo, un sistema de registro y autenticación de usuarios es absolutamente necesario para controlar el acceso a las reuniones telemática convocadas.

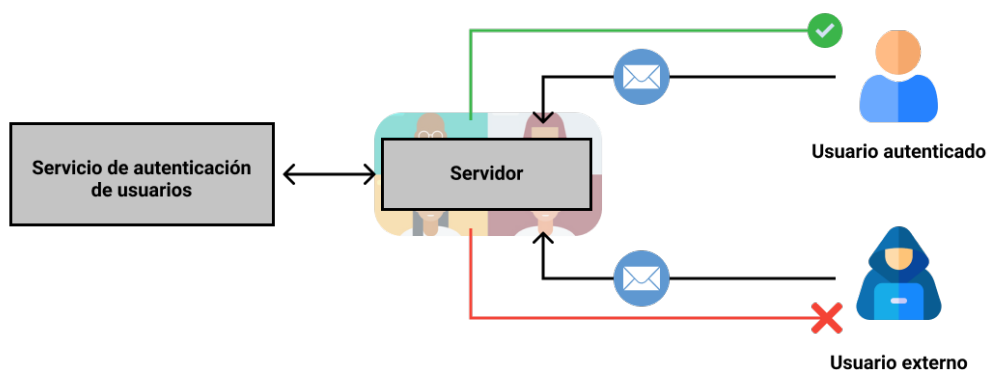


Figura 2.5. Autenticación para acceso a reuniones multimedia

## **2.3. Actas de reuniones y registro de intervenciones**

Uno de los problemas de las reuniones telemáticas es la dificultad de dejar constancia de lo que se ha mencionado en la reunión, así como de las intervenciones relevantes que se hayan realizado en la misma. De tal manera que la prueba de que se hayan realizado o dicho algo en la reunión no se pueda ver alterada por los intervinientes en la reunión o incluso terceras partes. Es por ello, que, en reuniones de carácter formal, muchas veces es necesaria una manera de guardar o registrar intervenciones y acuerdos realizados por las partes que se encuentran en la reunión.

Esta necesidad de registrar acuerdos realizados en reuniones en un acta se encuentra en muchas organizaciones. Por ejemplo, en reuniones empresariales, según el Título V de la Ley de Sociedades del Capital, todos los acuerdos realizados en las mismas deben constar en acta, la cual debe ser aprobada por la propia junta o su presidente en un plazo de 15 días [12]. A su vez, en comunidades de vecinos, según el Artículo 19 de la Ley de Propiedad Horizontal, los acuerdos adoptados, así como la fecha de realización de estos, deben quedar expresados en el acta de la reunión [13].

Como se puede inferir, cualquier organización que realice reuniones de carácter formal necesita que los acuerdos realizados en la reunión queden registrados de alguna manera para dejar constancia de su realización. Para ello, aunque en los ejemplos descritos se utilizan actas de reuniones para registrar lo acontecido en la reunión, la evidencia real de que se ha realizado una intervención o acuerdo en una reunión telemática es el propio contenido multimedia del participante en la reunión. Por este motivo es por el que en este tipo de reuniones tele-presenciales, es necesario un mecanismo que permita grabar y almacenar el contenido multimedia de la reunión.

Para ello, muchas plataformas de videoconferencia y comunicación multimedia ofrecen servicios de grabación para que se guarde el contenido multimedia de la realización de la grabación. Sin embargo, guardar una grabación completa de la reunión realizada puede llegar a ser costoso e ineficiente. En Estados Unidos, se celebran 11 millones de reuniones telemáticas al día, en las que se estima que tienen una duración media es de hasta 60 minutos [14]. Si se requiriera guardar el contenido multimedia de cada una de las reuniones realizadas, se generarían aproximadamente más de 600 millones de minutos de



contenido multimedia, el cual debe ser almacenado por si fuera necesario consultarlo posteriormente.

Es por ello por lo que grabar el contenido completo de una reunión es una solución poco eficiente para almacenar las intervenciones realizadas en una reunión. Además, es muy común que solo sea necesario guardar una parte o apartado concreto, por lo que no es necesario grabar el contenido completo de la reunión, sino solo aquellas intervenciones relevantes durante la misma. Grabar únicamente intervenciones concretas durante la reunión permite que su consulta posterior sea mucho más fácil y rápida para el usuario.

Asimismo, si se implementara un sistema de grabación de intervenciones de la reunión, solo sería necesario almacenar el audio de la intervención, ya que es suficiente para corroborar quien lo ha realizado y se ahorraría mucho más espacio que almacenando el vídeo de la intervención realizada. En la Figura 2.6 se muestra este mecanismo, en el que los usuarios grabarían el audio de las intervenciones que realicen durante la reunión para que sean almacenadas de forma persistente.

Por otro lado, el sistema de registro de intervenciones debería incorporar un mecanismo que evite que cualquier usuario almacene el sistema cualquier intervención sin importar si su contenido es relevante para el ámbito de la reunión. Es por ello por lo que, cuando se realice una intervención, esta debería ser consensuada por los participantes de reunión y posteriormente, ser almacenada en la plataforma.

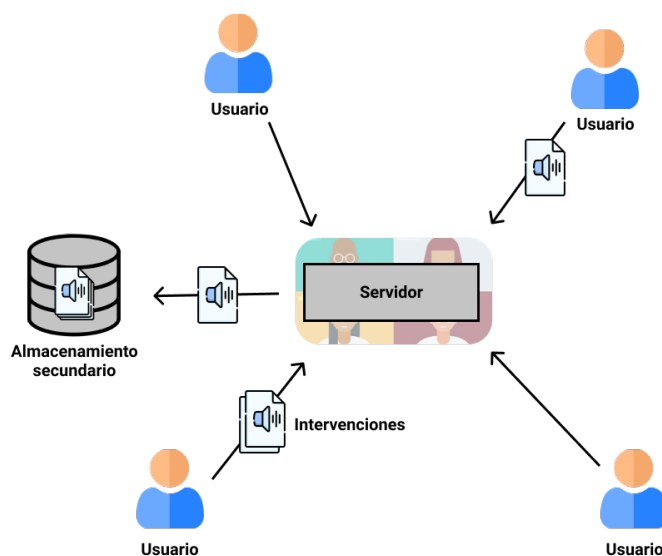


Figura 2.6. Sistema de registro de intervenciones

Esto no solo evita el posible problema de que se guarde cualquier tipo de intervención realizada, sino que se asegura la conformidad de todas las partes participantes en la reunión con la intervención que se ha realizado. En la Figura 2.7 se muestra a los usuarios que deben validar una intervención para que sea consensuada y almacenada correctamente en la plataforma. Además, en este proceso de validación de la intervención se debe llegar a un consenso entre los usuarios para que sea guardada en la plataforma (ya sea por mayoría o por un usuario secretario encargado de validar las intervenciones).

Por último, cabe a destacar que dicho mecanismo guardaría el contenido multimedia de las intervenciones en un servidor o en una plataforma de almacenamiento de archivos. Esto implica que el contenido de dichas intervenciones podría ser modificado sin consentimiento, ya sea por un tercero que obtenga acceso a los datos almacenados de manera ilícita, o incluso por el administrador que gestione el servidor o el sistema de almacenamiento. Por tanto, no se podría garantizar que la intervención guardada sea la misma que la intervención realizada durante la reunión ya que esta podría haber sido modificada *a posteriori*, por lo que no hay forma de saber si la intervención guardada ha sido alterada de alguna manera.

Por este motivo, garantizar la integridad y autenticidad de las intervenciones guardadas es un aspecto crítico, especialmente en reuniones en las que debe quedar registrada la realización de acuerdos en un acta tal y como se ha descrito anteriormente.

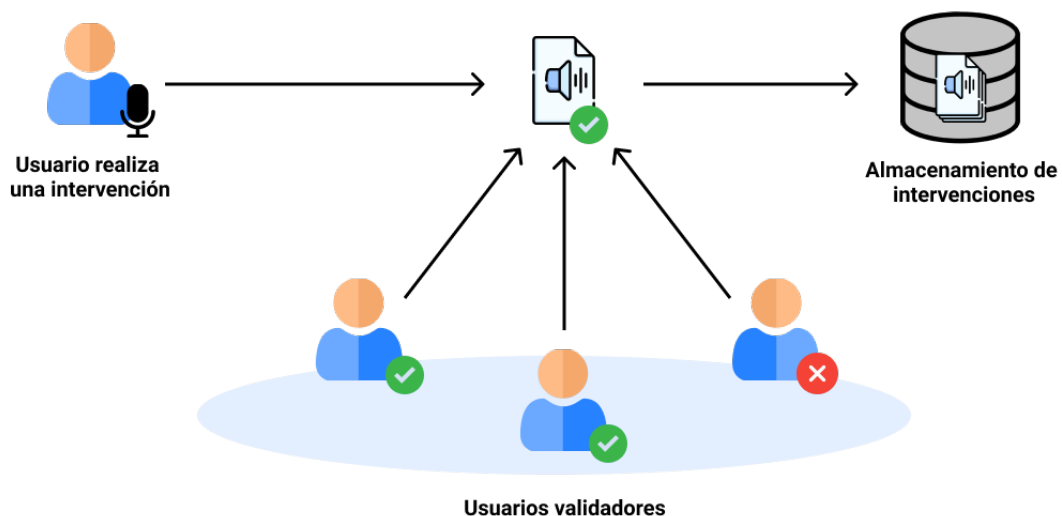


Figura 2.7. Validación de una intervención

## 2.4. Verificación de la autenticidad del registro de intervenciones

Como se ha descrito en el apartado anterior, las intervenciones realizadas y validadas quedarían guardadas en un almacenamiento secundario. Esto permitiría su posterior consulta por los participantes de la reunión, para ver el contenido multimedia de las intervenciones acabada la reunión, si lo desearan. Sin embargo, el principal inconveniente de seguridad de este sistema es garantizar la autenticidad de las intervenciones almacenadas. Esto es, que cuando se consulte una intervención, se deba poder garantizar que el contenido de esta sea el mismo que el que se grabó durante la reunión.

Por un lado, el propio contenido multimedia (en este caso, el audio de la intervención) podría ser prueba suficiente de que la intervención es auténtica, ya que se podría escuchar si la voz grabada en la intervención es la misma que la del autor. Sin embargo, con los avances en técnicas de inteligencia artificial, existen herramientas capaces de generar audio a partir de una voz humana de referencia [15], [16]. Además, el propio autor de la intervención podría grabar otra intervención y sustituirla por la original si consiguiera acceso al almacenamiento secundario de las intervenciones.

En la Figura 2.8 se muestra como un tercero que consiguiera acceso al sistema de almacenamiento podría alterar las intervenciones almacenadas en la misma, lo que ocasionaría que, si un usuario consultara la intervención, estaría viendo el contenido multimedia modificado en lugar del original. Este aspecto es vital, ya que puede que un agente externo o la propia entidad que gestiona el almacenamiento secundario podría alterar el contenido de las intervenciones almacenadas de forma ilegítima.

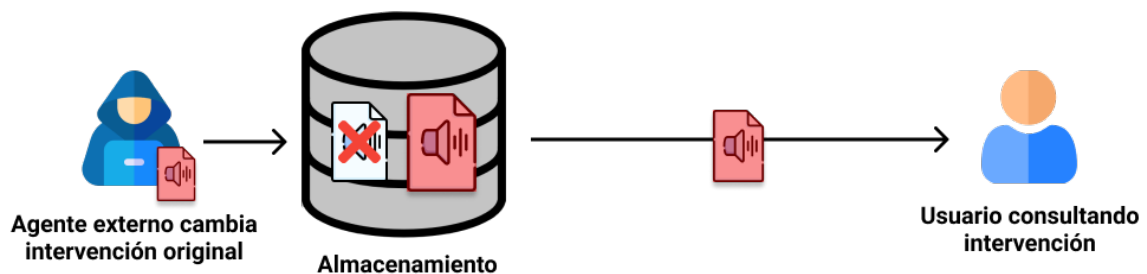
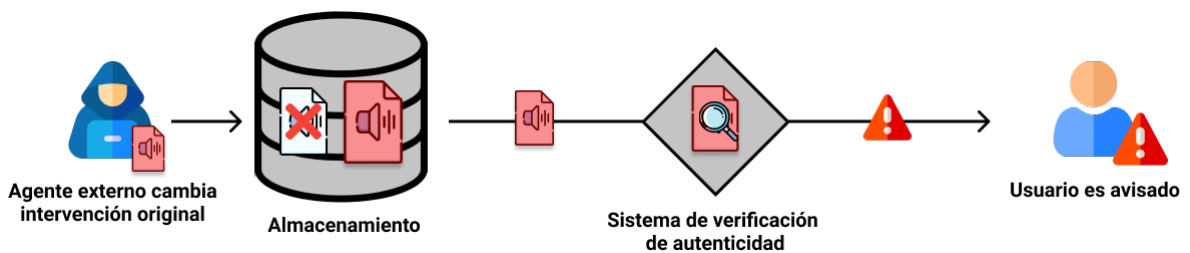


Figura 2.8. Intervención modificada por agente externo

Por este motivo, la plataforma de almacenamiento de intervenciones debe constar con un sistema que permita verificar que las intervenciones validadas, cuando sean consultadas, contengan el mismo contenido multimedia que cuando se realizaron durante la reunión multimedia. De esta manera, si un agente externo que consiguiera acceso al almacenamiento multimedia de forma ilegítima modificara el contenido multimedia de una intervención, cuando un usuario consulte dicha intervención podría ver que esta ha sido modificada utilizando dicho subsistema de verificación de intervenciones.

En la Figura 2.9 se muestra cómo, integrando un subsistema de verificación de integridad de las intervenciones, se podría comprobar la integridad de las intervenciones consultadas por los usuarios.



*Figura 2.9. Verificación de una intervención modificada*

# 3. Análisis de requisitos y funcional

---

Una vez comentado el problema que se pretende resolver, en este capítulo se describe la solución propuesta, analizando los requisitos, casos de uso del sistema y un análisis funcional. Por último, se realiza un análisis estructural previo a la implementación y desarrollo del sistema, en el que se muestra el funcionamiento del sistema.

### 3.1. Introducción

El principal riesgo que se corre al almacenar el contenido multimedia de las intervenciones realizadas por los participantes de una reunión telemática es la posible modificación de dicho contenido multimedia. Debido a esto, no se podría garantizar la autenticidad del audio de las intervenciones realizadas cuando se consulten. Esto se debe a que el sistema guarda las intervenciones en una plataforma de almacenamiento, es posible que el contenido de las mismas pueda ser modificado o alterado de forma ilícita, ya sea por un agente externo, el propio autor de la intervención, o la entidad que administra la plataforma de almacenamiento.

Algunos criterios de diseño generales para resolver estos problemas son los siguientes:

- *Registro únicamente del audio:* dado que con la señal de audio de un participante en la videoconferencia se le podría autenticar mediante parámetros biométricos, nosotros hemos decidido almacenar únicamente el audio de los usuarios. No almacenando el video porque sería redundante y además podríamos estar registrando información que no es relevante. El registro de ese audio es sencillo porque en principio sólo debemos almacenar la información que registra su micrófono dejando el altavoz del sistema de videoconferencia en estado de apagado. Esto evita que grabemos a algún otro participante de la videoconferencia.
- *Verificación de autenticidad de registro de intervenciones:* para asegurar que las intervenciones almacenadas en el sistema no han sido modificadas, se debe integrar un subsistema que permita almacenar cierta información de la intervención que se pueda utilizar para comprobar la integridad del contenido multimedia de la intervención. Este subsistema debe proporcionar un registro inmutable de dicha información de forma que, al consultar una intervención, al utilizar este subsistema se podría ver si dicha intervención es la original o si ha sido alterada.
- *Autenticación de usuarios:* solo aquellos usuarios que estén autenticados en el sistema y hayan sido invitados a una reunión pueden acceder a las reuniones multimedia a las que hayan sido convocados. Además, solo estos pueden realizar y

grabar intervenciones en dicha reunión, evitando que agentes externos accedan a la reunión y realicen intervenciones sin consentimiento.

- *Administrador*: para otorgar acceso a usuarios a la plataforma y convocar reuniones, debe existir un usuario con el rol de administrador. Este se encargaría de dichas gestiones, mejorando la seguridad del sistema ya que solo un grupo de usuarios previamente designados serían administradores.
- *Algoritmo para poner de acuerdo a los usuarios y permitir registrar intervenciones*: entre los usuarios que participen en una reunión, se debe poder consensuar qué intervenciones son válidas y cuáles no. Para ello, mediante un algoritmo para llegar a un acuerdo entre los usuarios, estos podrían decidir qué intervenciones deben quedar registradas en el sistema para su posterior consulta. Además, si los usuarios no llegaran a un consenso en un tiempo determinado, se debería validar la intervención por desistimiento positivo de forma automática.
- *Enviar notificaciones de reuniones*: cuando se convoca una reunión, los usuarios que hayan sido invitados a la misma deben ser notificados de la realización de dicha reunión. Además, esta notificación debe proveer a los usuarios de la información necesaria, como un enlace de acceso, para que los usuarios puedan acceder a dicha reunión convocada.

### **3.2. Análisis de requisitos**

En el sistema desarrollado, los usuarios deben poder acceder a reuniones, las cuales se definen como salas de comunicación multimedia, en las que los usuarios, a través de un enlace o invitación, pueden unirse e interactuar con otros usuarios. Dichas reuniones deben ser creadas previamente, donde se configuraría que usuarios pueden unirse a la reunión. Además, también se definirían diferentes parámetros relativos a la celebración de la reunión, como la fecha de realización o su nombre.

Una vez creada la reunión, los usuarios que hayan sido invitados serían notificados y pueden unirse a ella y comunicarse de forma multimedia con otros usuarios que se encuentren en ese momento en la reunión. Dentro de dicha sala multimedia se podrían

realizar intervenciones que los usuarios deseen que queden guardadas. Para ello, mediante la funcionalidad de grabación que proporcionaría la propia aplicación web, el usuario puede grabar sus intervenciones para que queden registradas y guardadas en el sistema.

Sin embargo, para evitar que cualquier intervención se almacene sin el previo acuerdo de otros usuarios, estas se deben validar. Para ello, al crear una reunión se define un mecanismo de validación el cual determina que usuarios son los encargados de validar las intervenciones realizadas en la reunión. Dichos usuarios deben aceptar o rechazar dichas intervenciones para decidir si son válidas y se pueden guardar en el sistema. Hay dos tipos de mecanismos de validación:

- *Validación por usuario validador:* al crear la reunión, se designa un único usuario, el cual sería el encargado de decidir qué intervenciones deben ser validadas y añadidas al sistema. Dicho usuario se podría definir como el secretario de la reunión, ya que es el encargado de decidir qué intervenciones realizadas en la reunión son válidas y deben ser guardadas en el sistema para que quede constancia de su realización.
- *Validación por votación:* en este caso, todos los usuarios participantes en la reunión pueden validar intervenciones. Sin embargo, una intervención se añadiría al sistema si es validada por un número determinado de usuarios, el cual depende del porcentaje de validaciones necesarias y el número de participantes de la reunión (tanto el porcentaje de validaciones necesarias y el número de participantes de la reunión son parámetros que se configurarían al crear la reunión).

Por ejemplo, si el porcentaje de validaciones necesarias es del 50%, solo es necesario que la intervención se valide por la mitad de los participantes de la reunión. Por otro lado, si el porcentaje requerido es de 100%, es necesaria que la intervención sea validada por todos los usuarios participantes de dicha reunión.

Una vez que se designan los usuarios encargados de validar la intervención, dichos usuarios tienen un periodo de 15 días para validarlas, ya sea aceptando que se guarden en el sistema o no. Tras este plazo de tiempo, las intervenciones pendientes de validación se aceptarían de forma automática por desistimiento positivo. Por motivos de simplicidad y dado que se trata de generar un prototipo y no una herramienta completa disponible para ser



comercializada, no se comprueba si el usuario validador estaba conectado en el momento en el que hizo la intervención. Por otro lado, este es un aspecto que encierra cierta complejidad, porque no se puede garantizar que, aun estando conectado, el usuario validador esté activo y en modo escucha activa. Verificar este requisito necesitaría el diseño de un prototipo mucho más complejo cuyo objetivo no es el de este TFG.

Una vez que se ha validado la intervención por los usuarios validadores, esta se almacena en la plataforma y se puede consultar por los usuarios que forman parte de la reunión. A su vez, se debería implementar un mecanismo que permita garantizar la integridad de las intervenciones almacenadas para asegurar que no han sido alteradas o modificadas una vez han sido validadas y almacenadas en el sistema. Para el desarrollo de este mecanismo, se utiliza una red *blockchain* que permita guardar de forma inmutable un *token* de las intervenciones, el cual permitiría comprobar la autenticidad de las intervenciones validadas que hayan sido almacenadas en el sistema.

### **3.3. Casos de uso**

En este apartado se describen los casos de uso del sistema desarrollado, diferenciando entre los casos de uso correspondientes a la utilización de la aplicación web multimedia y la *blockchain* desarrollada.

#### **3.3.1 Casos de uso de la aplicación web de comunicación multimedia**

La aplicación web tiene varias funcionalidades que permiten a los usuarios usar el sistema de registro de intervenciones, así como hacer uso de las salas de comunicación multimedia.

Los actores correspondientes que participan y usan la aplicación web son los siguientes:

- *Administrador*: es el encargado de crear las reuniones y definir las características del mecanismo de consenso que se usaría en dichas reuniones (designar los usuarios que encargarían de validar las intervenciones, mecanismo de votación para validación de intervenciones, participantes de una reunión multimedia...).
- *Usuario*: es el que interactúa en la comunicación multimedia. Su objetivo es usar la aplicación para comunicarse con otros participantes y consultar las intervenciones realizadas y validadas de una reunión multimedia.

- *Usuario validador*: dependiendo del mecanismo de validación configurado para la reunión multimedia, hay usuarios que se encargarían de validar las intervenciones realizadas para que sean añadidas a la cadena de bloques. El *Administrador* es el encargado de, al crear la reunión, designar los usuarios validadores y el mecanismo de validación utilizado.

Seguidamente, se describen los casos de uso, explicando detalladamente las acciones que llevan a cabo cada uno de los actores descritos en la aplicación web. En primer lugar, los casos de uso del *Administrador* son los que se describen a continuación, los cuales se muestran en la Figura 3.1.

### **Crear una reunión multimedia**

Un *Administrador* debe ser capaz de crear una reunión definiendo diferentes parámetros de su configuración: nombre de la reunión, participantes de la reunión multimedia, mecanismo de validación de intervenciones (designando los *Usuarios Validadores* de la reunión), así como la fecha de inicio y finalización de la reunión. Al crear la reunión, se genera un enlace que se envía por correo electrónico a los usuarios participantes de la reunión para que puedan acceder a la sala de comunicación multimedia.

### **Administrar usuarios administradores**

Un *Administrador* debe ser capaz de gestionar los usuarios administradores de la aplicación web. Para ello, este debe poder consultar que usuarios tienen el rol de *Administrador*, pudiendo ser capaz de quitar el rol de *Administrador* a otro usuario y dotar del rol de *Administrador* a un usuario determinado.

### **Añadir usuarios**

Un *Administrador* debe poder dotar de acceso a la aplicación web a un nuevo usuario. Para ello, un *Administrador* debe ser capaz de generar las credenciales necesarias para que nuevos usuarios puedan acceder a la aplicación web. Este sistema, en el que el registro de nuevos usuarios queda gestionado por los administradores, permite controlar que clientes tienen acceso a la aplicación web. Este sistema permite saber que usuarios se registren, en lugar de utilizar un sistema de registro abierto en el que cualquier usuario podría registrarse libremente.

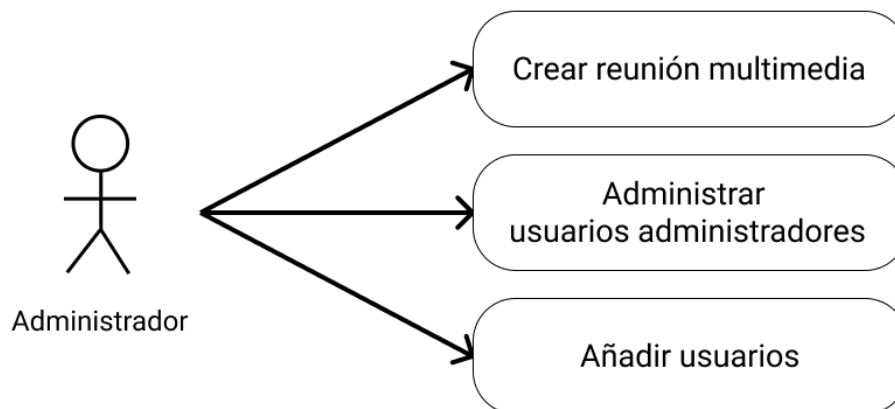


Figura 3.1. Diagrama de casos de uso del Administrador

Seguidamente, se describen los casos de uso del *Usuario Validador*, el cual es el encargado de realizar la validación de las intervenciones realizadas. El diagrama de dichos casos de uso de muestran en la Figura 3.2.

### Consultar intervenciones pendientes de validación

El usuario que ha sido designado como validador de las intervenciones realizadas debe ser capaz de consultar aquellas intervenciones que, tras realizarse en una reunión multimedia, quedan pendientes de ser validadas para su inserción en la cadena de bloques. Al consultarlas, el *Usuario validador* puede consultar la información de dicha intervención (autor de la intervención y fecha de realización de la intervención) y revisar el contenido multimedia que se ha grabado, para verificar el contenido de la misma antes de validarla.

### Validar intervenciones

Por último, dada una intervención pendiente de validación, el *Usuario Validador* debe ser capaz de validar las intervenciones que se han realizado en las reuniones multimedia donde haya sido designado como validador de las intervenciones. Dicho *Usuario Validador* puede aceptar la intervención o rechazar la intervención realizada.

Dependiendo del mecanismo de validación de la reunión, si la intervención ha sido aceptada por los *Usuarios Validadores* necesarios para que la intervención sea validada, se añadiría a la cadena de bloques y pasaría a estar validada. En caso de haber sido rechazada por una cantidad determinada de *Usuarios Validadores* según el mecanismo de validación, la intervención no se añadiría a la cadena de bloques. Cabe a destacar que, pasado un

periodo de tiempo determinado, si una intervención no ha sido aceptada o rechazada por el *Usuario Validador*, esta intervención sería aceptada por desistimiento positivo.

Por último, los casos de uso de los *Usuarios* de la aplicación web se describen a continuación y se muestran en la Figura 3.3.

### **Acceder a una reunión multimedia**

Aquellos *Usuarios* que hayan sido invitados como participantes en una reunión pueden acceder a la sala de comunicación multimedia de dicha reunión. Para ello, pueden acceder a través del enlace de la reunión que reciben en su correo electrónico cuando se crea la reunión, o bien a través de la propia aplicación web.

Cabe a destacar que, si un *Usuario* intenta acceder al enlace de una reunión a la que no ha sido invitado, no podría acceder a la sala de comunicación multimedia. Lógicamente, solo aquellos usuarios invitados a la reunión pueden acceder, evitando que, si el enlace de la reunión ha sido comprometido, no puedan acceder terceros a dicha sala de comunicación multimedia, aunque posean el enlace de acceso a la reunión.

### **Realizar una intervención**

Dentro de la reunión multimedia, los *Usuarios* deben ser capaces de grabar el contenido multimedia de aquellas intervenciones que realicen para guardar constancia de estas. Dichas intervenciones no podrían tener una duración mayor a un tiempo determinado y el *Usuario*, tras haber realizado la intervención, le daría un nombre que describa el contenido o propósito de la intervención. Posteriormente, el *Usuario* debe poder revisar el contenido multimedia antes de subir dicho contenido a la plataforma. Si el *Usuario* que ha realizado la intervención no está conforme o no quiere que la intervención realizada sea subida a la plataforma, puede cancelar la acción y realizar otra posteriormente.

### **Ver mis intervenciones**

Desde la aplicación web, dadas las reuniones multimedia a las que haya sido invitado el *Usuario*, este puede ver las intervenciones que ha realizado sin importar si están pendientes de ser validadas, o si han sido validadas o rechazadas. Cabe a destacar que, si una intervención ha sido rechazada por los *Usuarios Validadores*, esta solo sería visible por

el autor de la intervención, de forma que este podría saber que la intervención ha sido rechazada y, por tanto, no ha sido añadida a la cadena de bloques.

### Ver mis intervenciones validadas

Por otro lado, el *Usuario* también puede ver las intervenciones que han sido validadas en las reuniones en las que ha sido designado como participante. Esto permite que cada *Usuario* pueda consultar las intervenciones que han sido aceptadas por los *Usuarios Validadores* de la reunión.

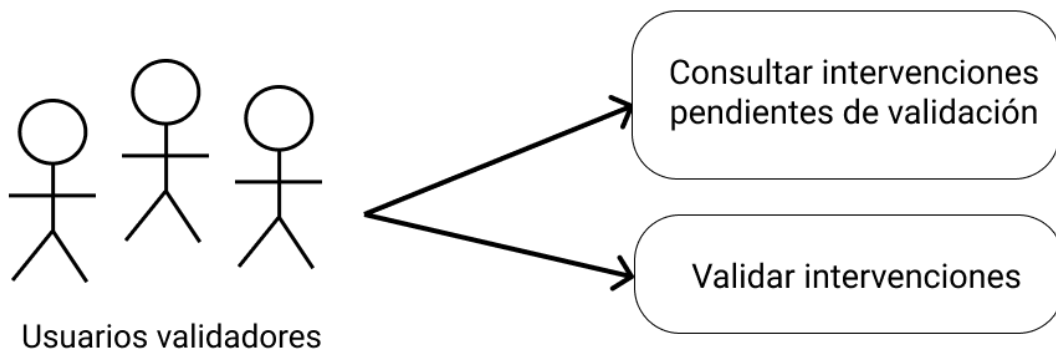


Figura 3.2. Casos de uso de los usuarios validadores

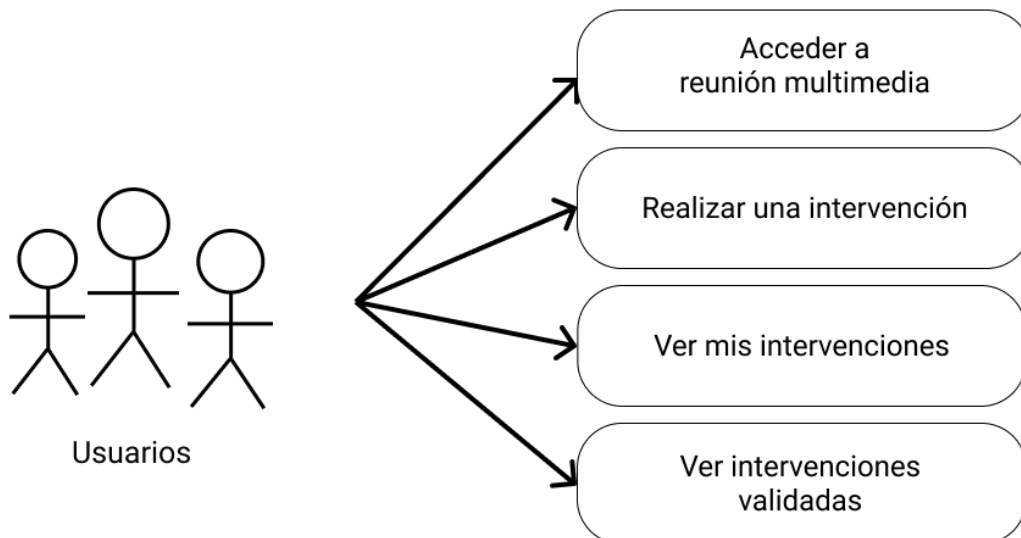


Figura 3.3. Diagrama de casos de uso de los Usuarios

### 3.3.2. Casos de uso de *blockchain*

La *blockchain* se encargaría de almacenar de forma unívoca un *token* que represente la intervención validada en el sistema. Este *token* permitiría comprobar la integridad de una intervención para saber si esta ha sido modificada. Dicha *blockchain* estaría formada por dos tipos de agentes:

- *Nodo raíz*: es la unidad central de la *blockchain* que administran qué nodos pueden participar en la *blockchain* y puede consultar los datos almacenados en la *blockchain*. Solo hay uno en la *blockchain*. Cabe a destacar que este ofrecería una interfaz para su gestión que es solo visible para un *Administrador*.
- *Nodo*: es el resto de los nodos que participarían en la *blockchain* que se encargan de realizar el proceso de consenso de la cadena de bloques y añadir datos.

Por un lado, los casos de uso del *Nodo raíz*, los cuales se muestran en la Figura 3.4, se describen a continuación.

#### **Gestionar nodos validadores**

El *Nodo raíz* es la autoridad central de la *blockchain*, por lo que se debe encargarse de decidir qué nodos pueden participar en la misma y realizar el proceso de minado. Para ello, el *Nodo raíz* debe poder gestionar una lista de nodos, añadiendo nuevos nodos o eliminando nodos de dicha lista. De esta forma, se controla qué nodos pueden formar parte de la *blockchain* y dar acceso a nuevos nodos que se quieran incorporar al sistema.

#### **Crear una transacción**

Cuando se valida una intervención, se debe añadir una transacción correspondiente al *token* de dicha intervención. Para ello, el *Nodo raíz*, al recibir el *token* de una nueva intervención que se debe añadir a la cadena de bloques, debe ser capaz de añadir la transacción recibida a un conjunto de transacciones pendientes de ser añadidas a la cadena de bloques, comprobando previamente que su contenido ha sido firmado digitalmente de forma correcta. El conjunto de transacciones al que es añadida la transacción realizada es el *transaction pool* [17], y contiene todas aquellas transacciones que aún no han sido almacenadas en la cadena de bloques.

## Comunicar transacciones al resto de los nodos

A su vez, debe comunicar al resto de nodos que conforman la red *blockchain* esta nueva transacción y elegir, entre la lista de nodos validadores, el nodo que debe realizar el proceso de minado para añadir la transacción a la cadena bloques.

## Buscar una transacción

Para consultar el contenido almacenado en la cadena de bloques, se debe poder buscar una transacción determinada (correspondiente a una intervención) dentro de los bloques que componen la cadena de bloques. Esto permitiría que, dada una intervención almacenada en el sistema, se pueda verificar su integridad buscando su transacción correspondiente en la *blockchain*. Esta transacción contendría el *token* de la intervención que permitiría verificar que el contenido multimedia de dicha intervención no ha sido alterado.

## Consultar el contenido de la cadena de bloques

Por último, desde el *Nodo Raíz* se debe poder consultar el contenido actual almacenado en la *blockchain*. Esto permitiría al *Administrador* que gestiona la *blockchain* ver la cadena de bloques almacenada para saber el contenido actual que hay guardado en la misma fácilmente.

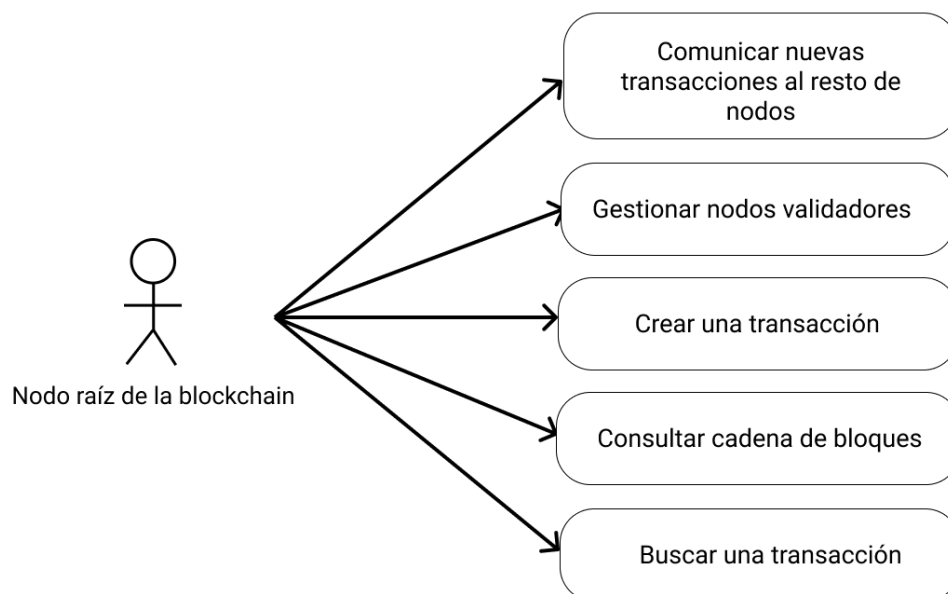


Figura 3.4. Casos de uso del nodo raíz

Por otro lado, los casos de uso de los *Nodos*, los cuales se muestran en la Figura 3.5, se describen a continuación.

### Minar un bloque

Para añadir una transacción a la cadena de bloques, el nodo participante en la *blockchain* debe ser capaz de minar un bloque cuando este reciba una nueva transacción del *Nodo raíz* y haya sido elegido para realizar el proceso de minado. Dicho proceso de minado solo ocurre si el *Nodo raíz*, que envía las transacciones creadas a todos los nodos, elige a un *Nodo* para realizar el proceso de minado

### Realizar consenso de la cadena de bloques

Cuando se mina un bloque, la cadena de bloques con el nuevo bloque añadido debe ser compartida en la red *blockchain* para que los nodos que conforman la red tengan la misma cadena de bloques. Para ello, el nodo de la *blockchain* debe ser capaz de, tras recibir una nueva cadena de bloques, comprobar la validez de la *blockchain* aplicando un mecanismo de consenso. Dicho mecanismo de consenso permite asegurar que no se añadan bloques falsos a la cadena de bloques e impide que posibles nodos maliciosos modifiquen la cadena de bloques sin seguir el mecanismo de consenso.

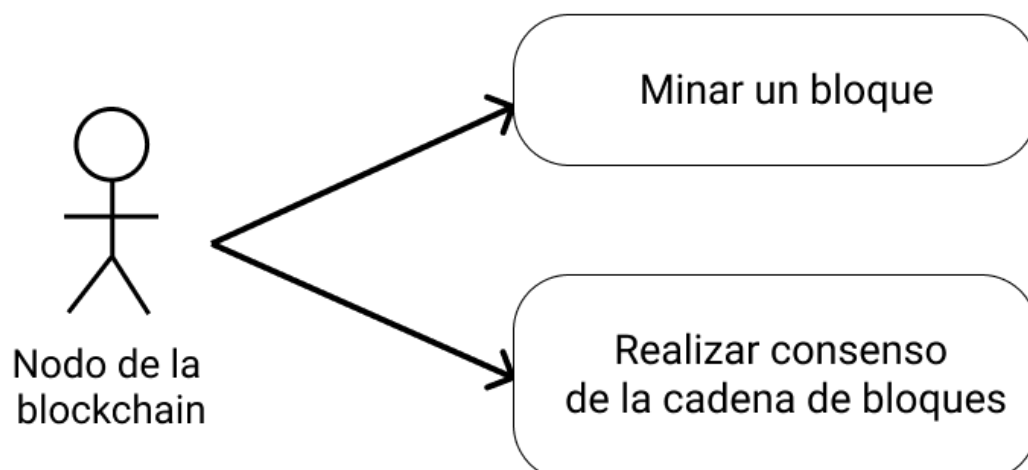


Figura 3.5. Casos de uso de un *Nodo de la blockchain*



### 3.4. Análisis funcional

A continuación, se describe el análisis funcional del sistema, donde se muestran los *mockups* de la aplicación web realizados con la herramienta *Figma* [18], para mostrar las diferentes funcionalidades de la aplicación.

Para acceder a la aplicación web, se presentaría un formulario de inicio de sesión en el que el usuario debe introducir sus credenciales. Para ello, la aplicación debe proporcionar un sistema de autenticación de usuarios, de forma que, si las credenciales del usuario son correctas, se accedería a la aplicación web donde se pueden ver las reuniones a las que ha sido invitado el usuario y que aún no han expirado. En la Figura 3.6 se muestra el *mockup* de las próximas reuniones a las que puede acceder el usuario.

Desde aquí el usuario puede acceder a una de las reuniones que se muestran. De igual manera, también debe poder desde un enlace o invitación, el cual se envía por correo electrónico cuando se crea la reunión para notificar al usuario. En la Figura 3.7 se muestra el *mockup* de la sala multimedia de la reunión, desde la cual el usuario debe poder comunicarse de forma multimedia con otros usuarios.

A su vez, también debe poder ser capaz de registrar las intervenciones realizadas. Para ello, mediante la funcionalidad de grabación que proporcionaría la propia aplicación web, el usuario puede grabar sus intervenciones para que queden guardadas en el sistema. Antes de guardarlas, el usuario podría reproducir el contenido multimedia de la intervención para comprobar que se ha grabado correctamente y le daría un nombre para que sea más fácil identificar cada intervención cuando los usuarios las consulten más tarde.

Reunión	Tipo de consenso	Fecha	Acceder
Reunión sobre propuesta de TFG	Por usuario validador on 24.05.2019	May 26, 2019 7:30 PM	ACCEDER
Lectura de TFG	Por votación Porcentaje: 50%	May 26, 2019 7:30 PM	ACCEDER

Figura 3.6. Mockup de próximas reuniones

Los usuarios con el rol de administrador deben crear y convocar dichas reuniones multimedia. Para ello, tal y como se muestra en la Figura 3.8, también deben configurar diferentes parámetros cuando se crea la reunión (nombre de la reunión, usuarios invitados a la reunión, mecanismo de validación utilizado...). Además, los usuarios con el rol de administrador también deben poder gestionar los usuarios de la aplicación (como otorgar acceso a nuevos usuarios y dar el rol de administrador a otros usuarios).

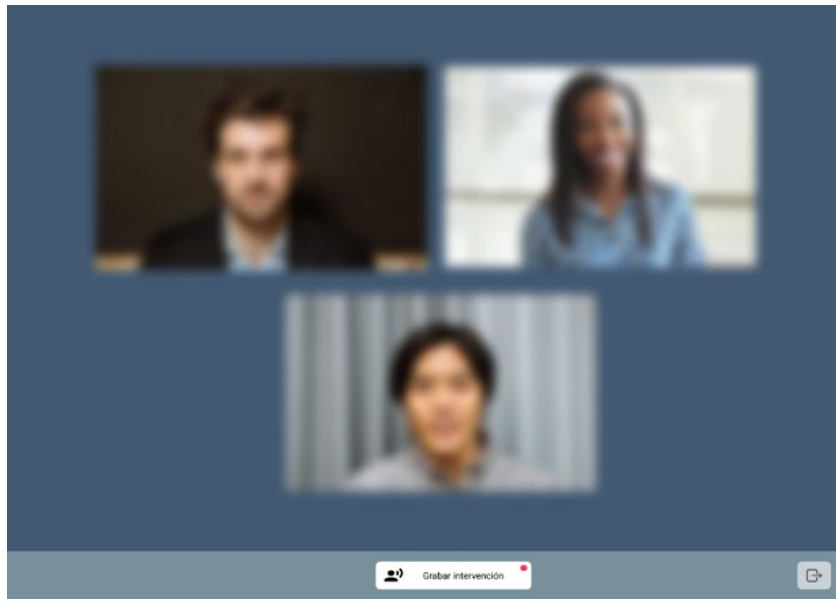


Figura 3.7. Mockup de la reunión multimedia

**Crear reunión** ADMINISTRADOR ↗

Nombre de la reunión

Fecha

Tipo de consenso  
 Consenso por votación  
 Consenso por usuarios validadores

Porcentaje de consenso para validación

Añadir usuarios a la reunión  
 ▼

- Usuario prueba ✖
- Usuario prueba 2 ✖
- Usuario prueba 3 ✖
- Usuario prueba 4 ✖

Figura 3.8. Mockup para la creación de una reunión

Cuando se realicen intervenciones en las reuniones multimedia, estas quedarían pendientes de ser validadas por usuarios validadores designados según el mecanismo de validación utilizado en la reunión. Estas intervenciones pendientes de validación se mostrarían, como se muestra en la Figura 3.9, en una pantalla donde los usuarios pueden aceptar o rechazar las intervenciones que aún no han sido validadas. Además, los usuarios deben poder reproducir el contenido multimedia de la intervención antes de validarla para comprobar el contenido de la misma.

En este punto, las intervenciones que hayan sido aceptadas por los usuarios validadores designados en la reunión serían añadidas al sistema de verificación de autenticidad de intervenciones de forma que cuando se consulten posteriormente, se pueda hacer uso de dicho sistema para comprobar la integridad de las intervenciones consultadas.

Para consultar las intervenciones que se han realizado, como se muestra en la Figura 3.10, el usuario puede consultar las intervenciones que haya realizado en la reunión, así como las intervenciones realizadas por otros usuarios de la reunión que ya hayan sido validadas. Por otro lado, además de poder reproducir el contenido multimedia de las intervenciones que se están consultando, desde aquí también se debe poder comprobar la autenticidad de las intervenciones que hayan sido validadas, para comprobar que no han podido ser modificadas o alteradas. Para ello, se utilizaría la *blockchain* implementada que permitiría verificar que el contenido multimedia no haya sido modificado.



Figura 3.9. Mockups de las intervenciones pendientes de validación




Listado de intervenciones realizadas en la reunión				Filter
Detalles de la intervención	Autor de la intervención	Fecha	Acción	
 Acuerdo sobre reunión TFG	Matt Damon <small>on 24.05.2019</small>	May 26, 2019 <small>8:00 AM</small>	<a href="#">VER</a> <a href="#">VERIFICAR</a>	
 Acuerdo de sesión de evaluación	Robert Downey <small>on 24.05.2019</small>	May 26, 2019 <small>7:30 PM</small>	<a href="#">VER</a> <a href="#">VERIFICAR</a> 	

Figura 3.10. Mockup para consultar las intervenciones realizadas

En cuanto a la gestión del sistema de verificación de intervenciones, en el que usaría una *blockchain* para dicho cometido, esta debería ser gestionada por un usuario administrador que se encargaría de gestionar qué nodos pueden participar en la red blockchain a través del nodo raíz. Asimismo, desde dicho nodo raíz, el usuario administrador podría consultar la cadena de bloques actual, para ver los datos que hay almacenados en la misma.

### 3.5. Análisis estructural

En primer lugar, para el desarrollo de la aplicación web de comunicación multimedia, se ha seguido un patrón de diseño *Modelo Vista Controlador (MVC)* [19]. En este patrón, el usuario interactúa con un cliente web (Vista), que se comunica con diferentes controladores dependiendo de la petición realizada. Dichos controladores se encargan de gestionar y atender las peticiones recibidas en el servidor. Por último, para poder acceder a los datos y la lógica de negocio, se utilizan el modelo, el cual interactúa con la base de datos del sistema.

Para la comunicación entre el cliente web y el servidor, se implementa una *Application Programming Interface (API)* a la cual el cliente realiza peticiones dependiendo de los recursos y operaciones solicitados. En la Figura 3.11 se muestra la arquitectura de la API y

los bloques principales del MVC a implantar en el servidor web y el flujo de transacciones de acciones que se desencadena cuando se recibe una petición en el servidor web. En primer lugar, cuando la petición llega al servidor, el servidor se encarga de ver el tipo de la petición (*POST, GET...*) y el método de la API asociado a dicha ruta. Si la ruta no existe, se devuelve un código de error. De igual manera, el servidor web comprueba si la petición ha sido realizada por un usuario autenticado en la aplicación web. En caso de no ser así, se devuelve un código de error.

En caso de que exista la ruta y la petición haya sido realizada por un usuario autenticado, dicho método es gestionado por un controlador, el cual es el encargado de tratar la petición y realizar consultas al modelo si fuera necesario. El modelo, posteriormente, es el encargado de interactuar con la base de datos y devolver los datos solicitados, si se hubiera solicitado alguno en la petición.

Finalmente, se prepara la respuesta con los datos solicitados si los hubiera, y se devuelve una respuesta con un código de éxito si se ha realizado la operación correctamente. En caso de no poder realizarse, se devolvería una respuesta con un código de error si no ha podido realizarse. Esto podría ocurrir si los datos no existieran, por ejemplo.

Mediante la API descrita en la Figura 3.11, que permite al cliente web solicitar datos o realizar operaciones al servidor web, se pueden realizar las diferentes operaciones necesarias en el sistema. Por ejemplo, para la creación de reuniones, el cliente web realizaría una petición a la API del servidor para realizar esta operación. El servidor posteriormente crearía un registro en la base de datos para dicha reunión y devolvería al usuario el resultado de la operación. Este tipo de operaciones, en el que el cliente web solicita la realización de operaciones al servidor, se pueden extrapolar a las necesarias en la aplicación: acceso a reuniones, creación de intervenciones, consulta y validación de intervenciones...

Por otro lado, dependiendo del estado de autenticación del usuario que accede a la aplicación y su rol, este puede acceder a diferentes vistas de la aplicación (Figura 3.12). Si el usuario estuviera autenticado, este podría acceder a la aplicación web y navegar a través de las diferentes vistas que ofrece. Además, si el usuario tiene el rol de administrador,

puede acceder a otras vistas para la gestión de los usuarios, aunque también puede seguir utilizando el resto de las funcionalidades de la aplicación. No obstante, si el usuario que accede a la aplicación web no estuviera autenticado, accedería a una vista para iniciar sesión (Vista de *Login*).

En cuanto al sistema de verificación de las intervenciones realizadas, se realiza desde un sistema independiente el cual se va a encargar de gestionar el sistema que garantizar la integridad de las intervenciones almacenadas. El servidor web se comunica con este servidor mediante una API, que funciona como la arquitectura descrita en la Figura 3.11. Cuando se necesita verificar o añadir el contenido de una intervención al sistema de verificación, el servidor web realizaría una petición a través de la API del nodo raíz encargado del sistema de verificación de las intervenciones, por lo que la infraestructura del sistema resultante sería la que se muestra en la Figura 3.13.

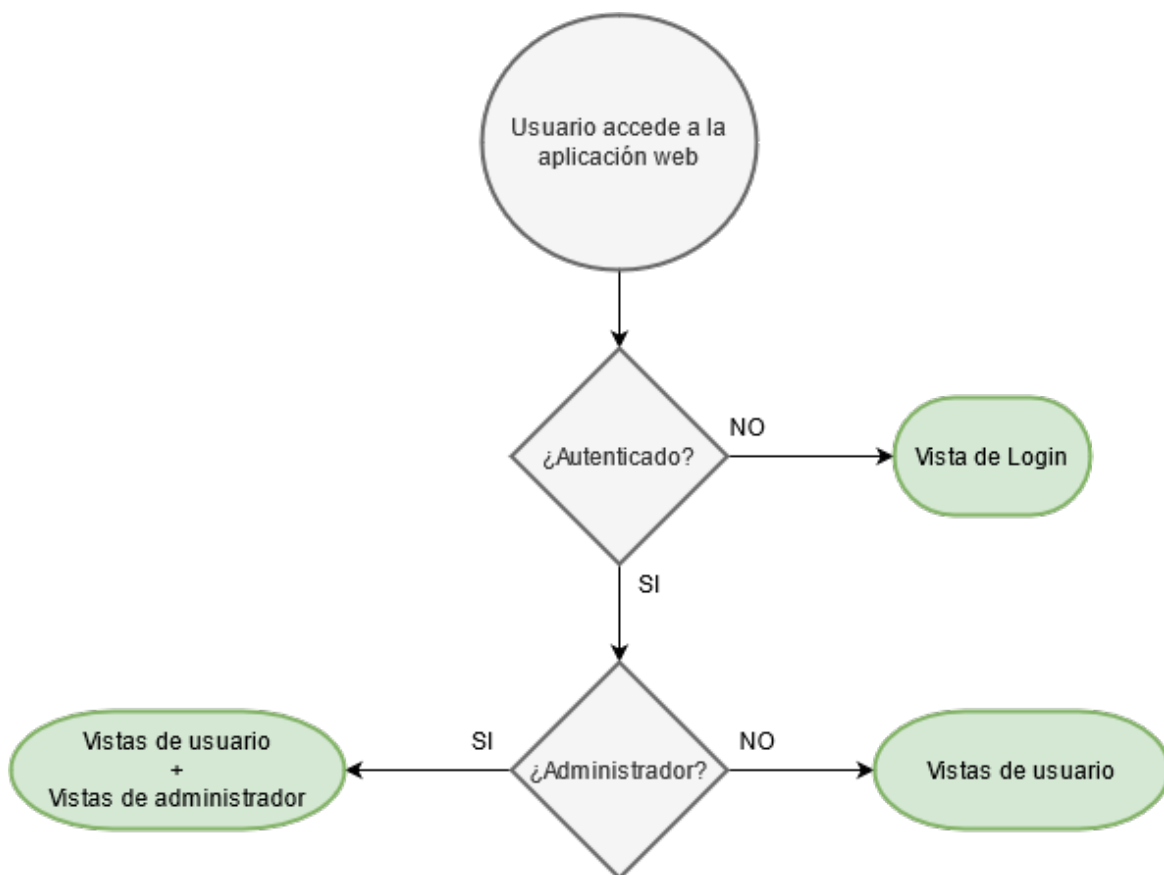


Figura 3.12. Vistas según autenticación de los usuarios

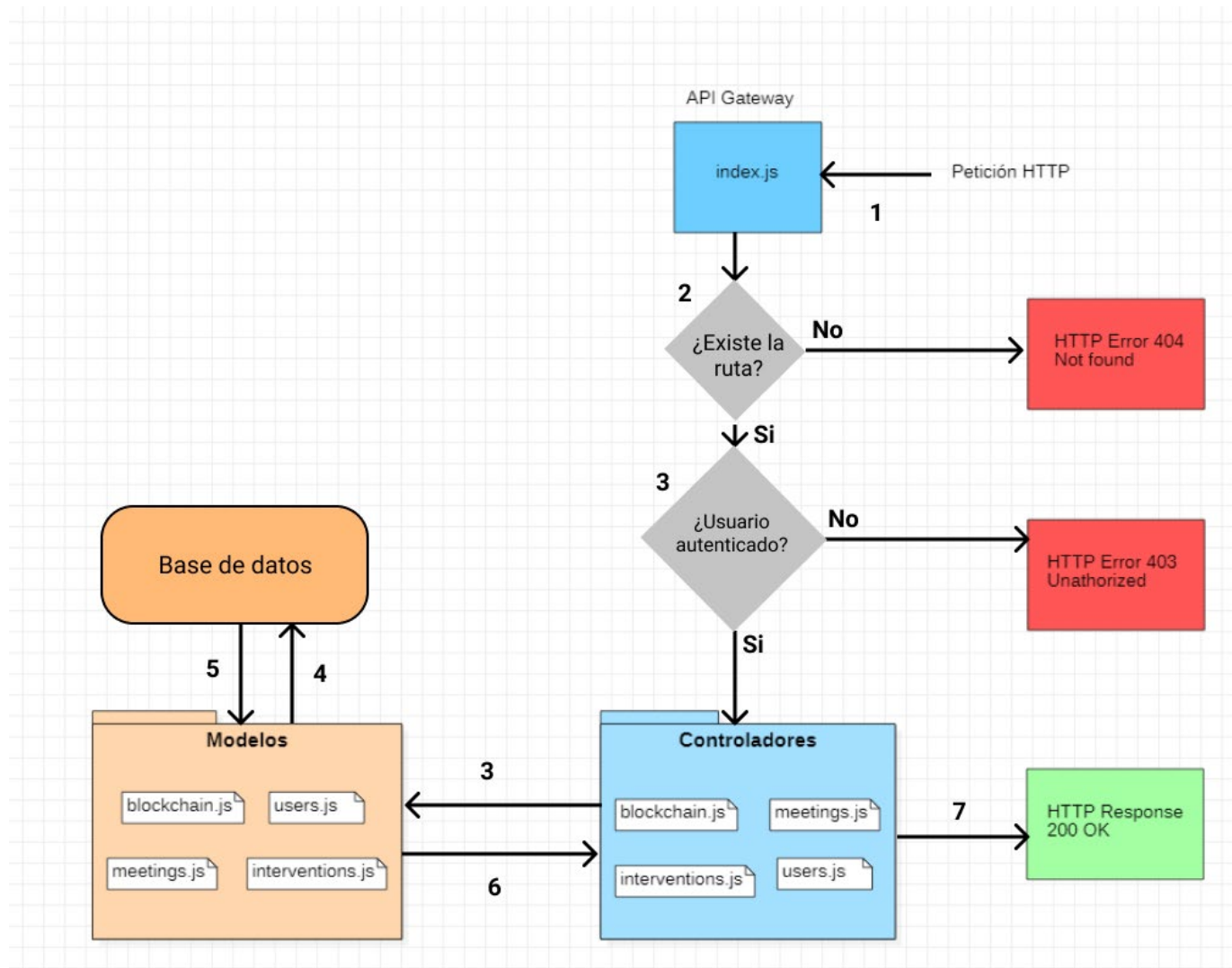


Figura 3.11. Arquitectura de la API de comunicación entre el cliente web y el servidor

En cuanto al sistema encargado de la verificación de las intervenciones, se utiliza una red *blockchain*. En esta, existiría un nodo raíz que utilizaría una arquitectura MVC al igual que la aplicación web, ya que esta tiene la responsabilidad de manipular datos (gestionar qué nodos se encargan de la validación), aplicar ciertas reglas de negocio y presentar un panel de administración para gestionar la red *blockchain* y consultar los datos que se encuentran almacenados en la misma. Por último, para comunicar el nodo raíz y el resto de los nodos que componen la red *blockchain*, se utiliza el patrón de mensajería *Publisher/Subscriber*, el cual permite a los nodos de la red *blockchain* enviar y recibir mensajes a través de canales de comunicación a todos los nodos.

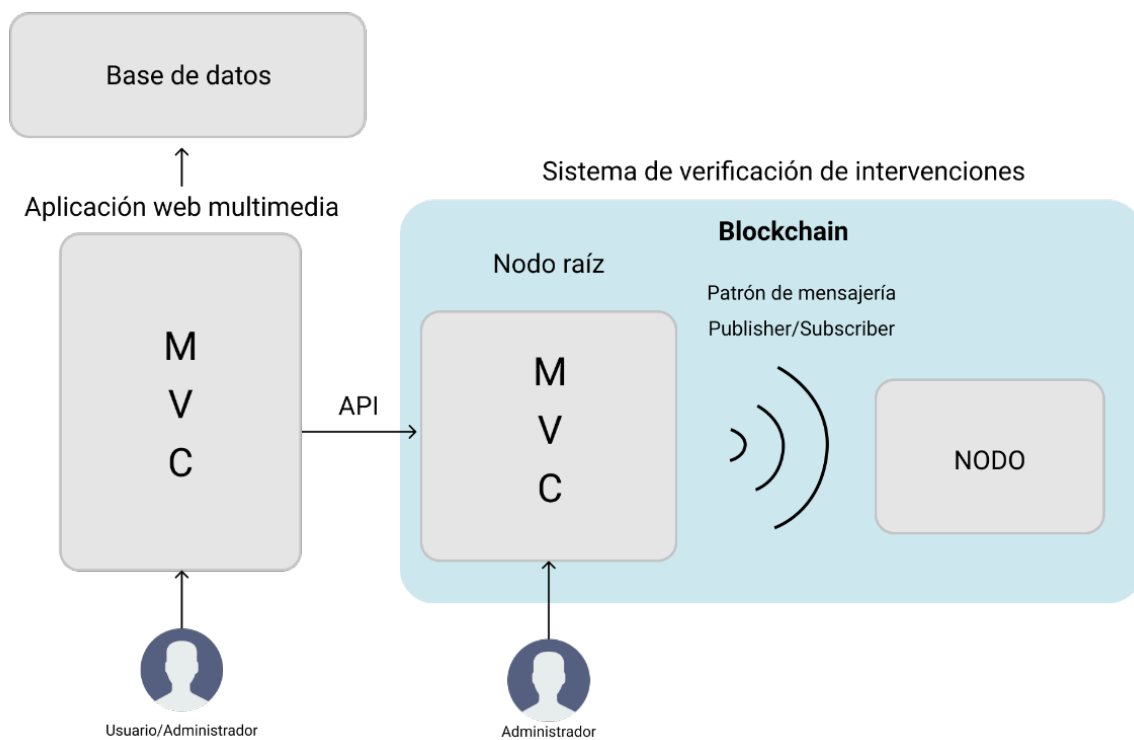


Figura 3.13. Arquitectura del sistema



# 4. Análisis de la implementación del sistema

---

El objetivo de este capítulo consiste en describir cómo se han desarrollado las diferentes partes del sistema y funcionalidades descritas en el Capítulo 3. Para ello, se explica de forma detallada el funcionamiento del sistema, mostrando partes del código relevantes para su explicación.

## 4.1. Introducción

La implementación propuesta para la realización de reuniones telemáticas donde es necesario dejar constancia de los acuerdos o hechos ocurridos, es la implementación una aplicación web de comunicación multimedia con un sistema que permita registrar el contenido multimedia de las intervenciones realizadas en las reuniones telemáticas que se hagan en dicha aplicación. Cada uno de los módulos que componen el sistema que se describen a continuación se encuentran subidos en *Github* [20]–[22].

En primer lugar, la Figura 4.1 muestra la arquitectura desplegada de la aplicación web multimedia, utilizando los *frameworks* y tecnologías descritas en el Anexo A. En primer lugar, para la interacción del usuario con la aplicación, se utiliza la interfaz de usuario desarrollada con *React*, el cual permite crear interfaces de usuario de manera rápida y eficiente mediante componentes reutilizables.

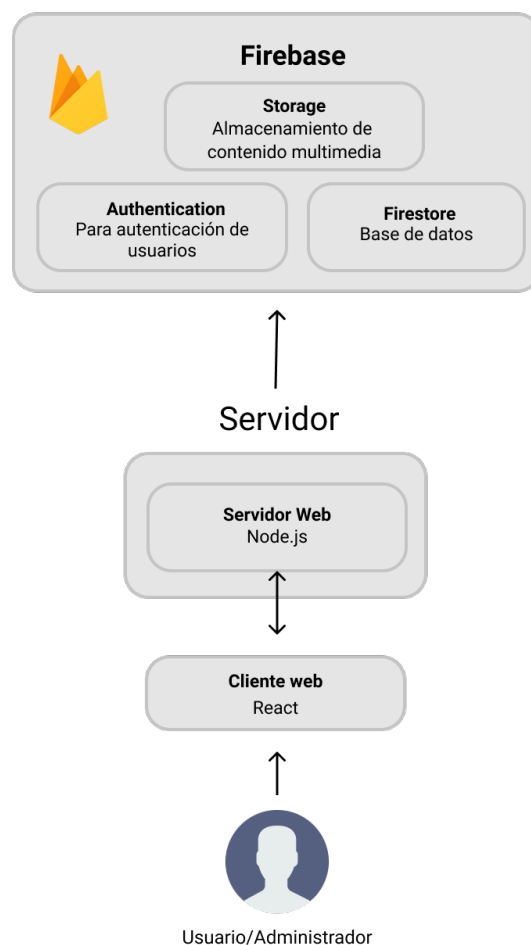


Figura 4.1. Arquitectura y tecnologías utilizadas en la implantación del sistema

Para la realización del servidor web, se hace uso del entorno de *Node.js*. Usar este entorno permite desarrollar y prototipar rápidamente el servidor web y hacer uso de gran cantidad de bibliotecas creadas por otros desarrolladores mediante el *Node.js Package Manager (NPM)* [4]. Tanto para la base de datos, como los servicios de autenticación y almacenamiento de archivos, se utiliza *Firebase* (concretamente, *Cloud Firestore*, *Cloud Storage* y *Authentication*). Este permite acelerar el desarrollo de la aplicación ya que se ocupa de proveer de la infraestructura necesaria para el almacenamiento de datos de forma persistente, así como de la gestión de usuarios.

En cuanto a al establecimiento de la comunicación multimedia en tiempo real interactiva, he decidido utilizar *Web Real-Time Communicacitons (WebRTC)*. WebRTC es un *framework* desarrollado por Google que permite el intercambio de información en tiempo real a través de internet, el cual es especialmente utilizado para el intercambio de flujos multimedia en tiempo real entre clientes web. En la solución propuesta, WebRTC aporta diferentes API y funcionalidades, que se describen en el Anexo B, que permiten implementar el sistema de comunicación multimedia entre usuarios para la aplicación web.

En la Figura 4.2 se muestra la comunicación multimedia utilizando WebRTC, en el que el servidor web funcionaría como intermediario para el establecimiento de la conexión, y posteriormente, se establecería la comunicación multimedia de igual a igual entre clientes web utilizando los mecanismos que proporciona WebRTC.

Por otro lado, el sistema de registro de las intervenciones realizadas en las reuniones multimedia necesita un mecanismo que permita asegurar la integridad del contenido multimedia de las intervenciones almacenadas en el sistema. Para solucionar este problema, se utiliza la tecnología *blockchain*, la cual se explica de forma detallada en el Anexo C. La descentralización e inmutabilidad que ofrece dicha tecnología son características que permiten mantener un registro inalterable de información, donde dichos datos no pueden ser alterados.

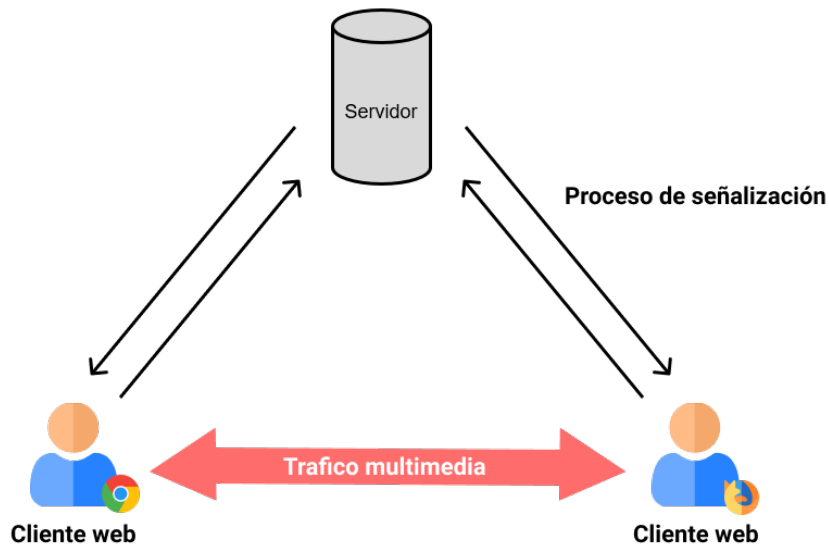


Figura 4.2. Comunicación multimedia utilizando WebRTC

En nuestro caso, la tecnología *Blockchain* se podría integrar en el sistema de registro de intervenciones comentado en la aplicación web multimedia. De esta forma, se puede almacenar el contenido de las intervenciones en la cadena de bloques, proporcionando el mecanismo necesario para garantizar que dichas intervenciones no han sido modificadas gracias a la propiedad de inmutabilidad que proporciona esta tecnología para los datos almacenados.

En la Figura 4.3 se muestra la implementación de la arquitectura de la *blockchain* desarrollada, la cual está compuesta por un nodo raíz (que gestiona la *blockchain*) y el conjunto de nodos que conforman la red *blockchain*. De igual manera que la aplicación web comentada en este apartado, la *blockchain* utiliza Node.js para desarrollar su lógica y, en el nodo raíz, se utiliza *React* para la interfaz de administración de la *blockchain*. Por último, para la comunicación entre los nodos que se encuentran en la *blockchain* se ha utilizado el patrón *Publisher/Subscriber* mediante un servidor *Remote Dictionary Server (Redis)*, el cual permite a los nodos suscribirse a canales de comunicación para recibir y enviar mensajes.

En la Figura 4.4 se muestra la implementación de la arquitectura de global del sistema, donde el servidor web interactúa con la *blockchain* a través del nodo raíz, el cual ofrece una API con la que el servidor web realiza las peticiones para añadir datos a la cadena de bloques o consultar los datos almacenados. En la Figura 4.5 se muestra un refinamiento de las diferentes partes que componen el sistema, las se explican a continuación.

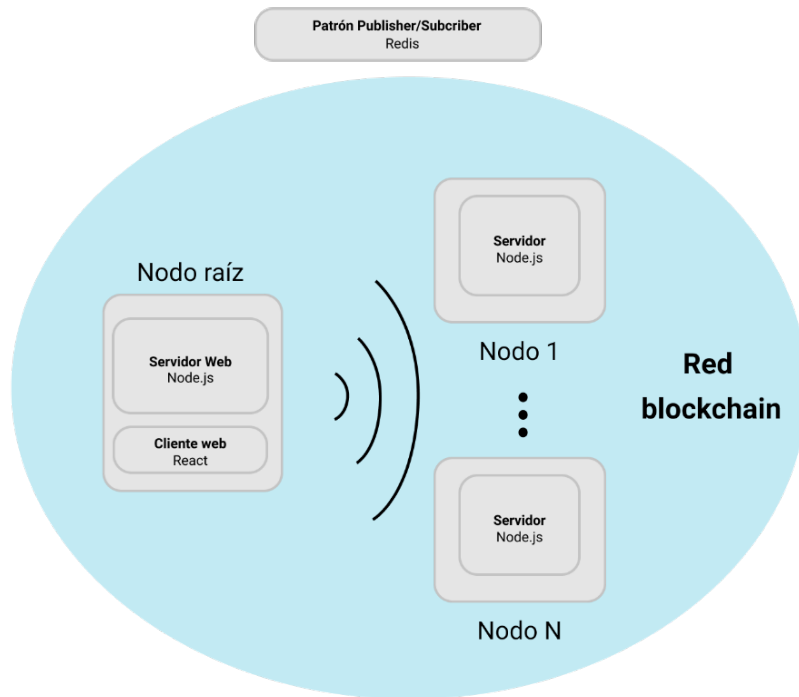


Figura 4.3. Implementación de la arquitectura de la blockchain

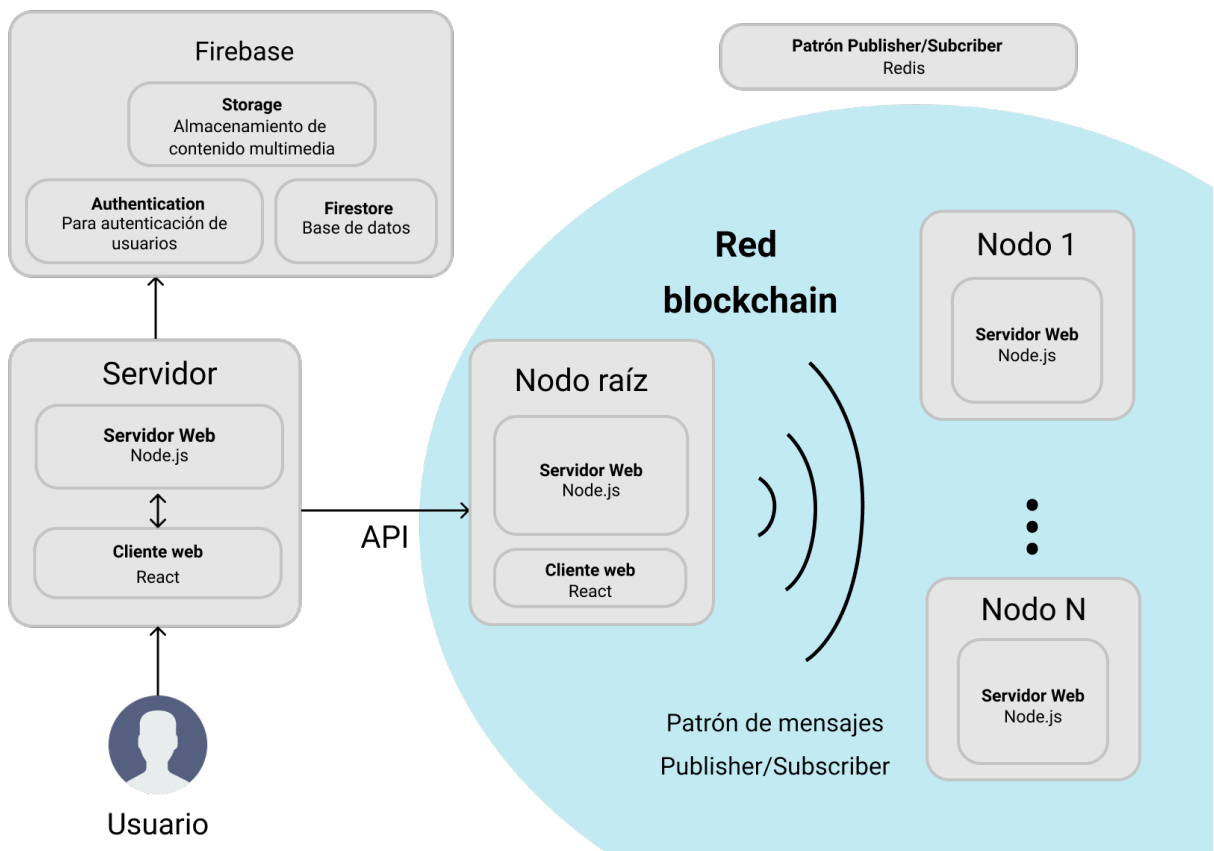


Figura 4.4. Implementación de la arquitectura de software del sistema

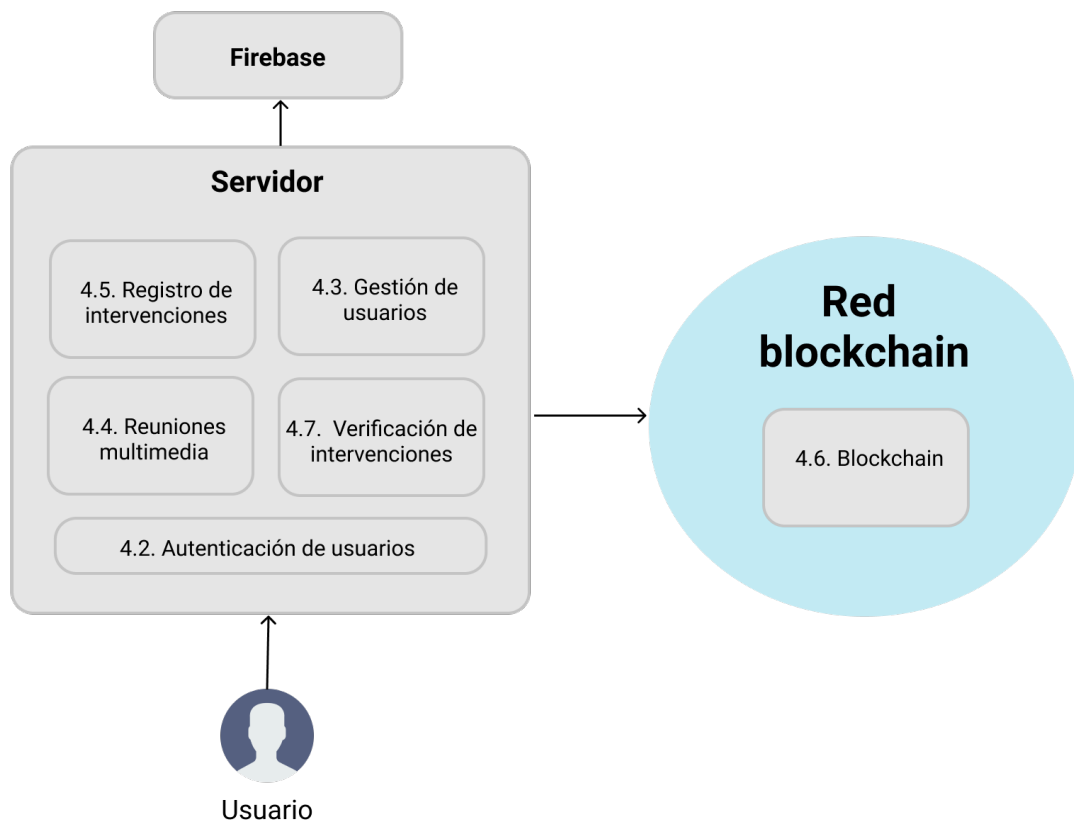


Figura 4.5. Partes que componen el sistema implementado

## 4.2. Análisis, diseño e implementación del sistema de autenticación de usuarios

Para la autenticación de usuarios, se utiliza el servicio de *Firebase Authentication*, el cual ofrece servicios de *backend* para la gestión de las sesiones de usuario y el almacenamiento de estos. Este ofrece diferentes formas de realizar la autenticación de los usuarios, utilizando otros proveedores de inicio de sesión como *Google, Facebook, Twitter...* Sin embargo, en este proyecto se ha utilizado la autenticación basada en correo electrónico y contraseña tradicional.

Desde el *frontend* de la aplicación, en la vista *Login*, se presenta un formulario de inicio de sesión en el que el usuario debe introducir su correo electrónico y contraseña. Al completar y enviar el formulario, se ejecuta la función *handleSubmit* que contiene la función

`signInWithEmailAndPassword` del *Software Development Kit (SDK)* de *Firebase Authentication* (Figura 4.6), en el que se inicia la sesión del usuario si las credenciales son correctas.

Posteriormente, al haber cambiado el estado de la sesión del usuario, se invoca la función `onAuthStateChanged` en el componente raíz del cliente web (Figura 4.7), `App.js`, el cual contiene todas las vistas y rutas de la aplicación. Desde dicha función, se comprueba si se ha iniciado sesión con las credenciales correctas. En caso de ser así, del objeto `userRecord`, el cual contiene diferente información acerca del usuario creada por *Firebase*, se extrae el correo electrónico del usuario, y el rol de este (si es un usuario con el rol de administrador). A su vez, se establece la variable `isLoggedIn` que se utiliza para renderizar las vistas de la aplicación dependiendo de si el usuario se ha autenticado (Figura 4.8).

```
const handleSubmit = (event) => {
  fire.auth().signInWithEmailAndPassword(email, password)
    .then(() => {
      console.log('Logged in correctly')
      message.success(`Se ha iniciado sesión correctamente con el usuario ${email}`)
    })
    .catch((error) => {
      console.error('Incorrect username and password', error)
      message.error('El usuario o la contraseña no son válidos');
    })
}
```

Figura 4.6. Función `handleSubmit` de la vista `Login`

```
fire.auth().onAuthStateChanged(async (userRecord) => {
  if(userRecord) {
    console.log(userRecord)
    const token = await userRecord.getIdTokenResult()
    setIsAdmin(token.claims.admin || false)
    setName(userRecord.displayName || userRecord.email)
    setIsLoggedIn(true)
  } else {
    setIsLoggedIn(false)
  }
})
```

Figura 4.7. Función `onAuthStateChanged` en el archivo `App.js` del cliente web

```

return (
  <Router>
    <UserContext.Provider value={{name, isAdmin}}>
      <Switch>
        {!isLoggedIn
          ? <Route component={NoAuthRoutes}/>
          : <Route component={AuthRoutes}/>
        }
      </Switch>
    </UserContext.Provider>
  </Router>
);

```

Figura 4.8. Componente raíz del cliente web

Por parte del servidor con el que se comunica la aplicación web, en todas las peticiones recibidas a la API se comprueba si el usuario que la realiza está autenticado. Además, cuando operación solicitada solo la pueden hacer usuarios con el rol de administrador, cuando se recibe la petición se comprueba si dicho usuario tiene dicho rol para hacer la operación. En la Figura 4.9 se muestra un ejemplo del manejo de las peticiones en el servidor, en el que se comprueba que el usuario está autenticado y si tiene el rol de administrador para realizar la operación. En caso de no estarlo, se devuelve un mensaje de error al usuario.

```

/**
 * Ruta: /users/add-admin
 * Descripción: Otorga el rol de administrador a un usuario
 */
usersRouter.post('/add-admin', async (req, res) => {
  const { email } = req.body
  const auth = req.currentUser;

  //Comprobación si está autenticado y si es administrador
  if(auth) {
    if(auth.admin) {
      try {
        await userService.addAdmin(email)
        return res.send('admin added')
      } catch(error) {
        console.error(error)
        return res.status(404).send('User not found')
      }
    }
  }

  return res.status(403).send('Not authorized')
})

```

Figura 4.9. Ejemplo de comprobación de autenticación al recibir petición en el servidor



### 4.3. Análisis, diseño e implementación de gestión de usuarios

En la aplicación web, para dotar de acceso a la aplicación a usuario nuevos y gestionar la lista de usuarios administradores, se han desarrollado diferentes vistas en la aplicación web para realizar dichas gestiones. Cabe a destacar que estas acciones solo pueden ser realizadas por un usuario con el rol de administrador.

En primer lugar, para la creación de nuevos usuarios, se utiliza la vista *AddUser*. Desde aquí, se muestra un formulario para introducir el nombre del usuario y el correo electrónico del mismo en el cliente web. Los datos introducidos se envían al servidor a la ruta */users/add-user*, el cual es manejado por el controlador *users.js* que se muestra en la Figura 4.10. Este controlador invoca a la función *addUser* del modelo, y devuelve una respuesta fallida o con éxito dependiendo si se ha podido añadir al usuario.

La función *addUser* (Figura 4.11) crea un nuevo usuario utilizando la función *createUser* del SDK de *Firebase Authentication*. Cabe destacar que la contraseña se establece en el servidor de forma aleatoria utilizando la biblioteca *randomstring* [23] y es enviada al correo electrónico del usuario utilizando la biblioteca *nodemailer*.

```
/**
 * Ruta: /users/add-user
 * Descripción: Crea un nuevo usuario y envía las credenciales a su email
 */
usersRouter.post('/add-user', async (req, res) => {
  const { email, name } = req.body
  const auth = req.currentUser;

  if(auth) {
    if(auth.admin) {
      try {
        await userService.addUser(email, name)
        return res.send('user added')
      } catch(error) {
        console.error(error)
        return res.status(400).send('Error adding user')
      }
    }
  }

  return res.status(403).send('Not authorized')
})
```

Figura 4.10 Manejador de la ruta */users/add-user* en el controlador *users.js*

```

const addUser = async (email, name) => {
  const password = randomstring.generate(8)

  await firebaseAdmin
    .admin
    .auth().createUser({
      email: email,
      password: password,
      displayName: name
    })

  await sendEmailCredentials(name, email, password)
}

```

Figura 4.11 Función `addUser` del modelo `users.js` en el servidor

Por otro lado, para la gestión de los usuarios administradores, se utilizan los *Custom Claims* [24] que ofrece *Firebase Authentication* para crear atributos personalizados a los usuarios para el establecimiento de roles o grupos de usuario. Para ello, la aplicación web muestra la vista *Admins*, que contiene un campo de entrada de texto para añadir nuevos usuarios y una tabla con la lista de usuarios administradores, donde se puede eliminar el rol de administrador en los usuarios elegidos. En la Figura 4.12 se muestran los contenidos descritos de dicha vista.

Para otorgar el rol de administrador a un usuario, el cliente web envía una petición a la ruta `/users/add-admin` con el correo electrónico de dicho usuario. Desde el modelo, utilizando la función `addAdmin`, se consulta si existe un usuario con dicho email y posteriormente se actualizan sus *Custom Claims* para otorgarle el rol de administrador, como se muestra en la Figura 4.13.

De igual manera, para eliminar un usuario, el cliente web envía una petición a `/users/delete-admin` indicando el usuario al que se le quieren quitar los privilegios de administrador. Desde el modelo, utilizando se modifican los *Custom Claims* para realizar dicha acción como se muestra en la Figura 4.14.

```

return (
  <>
    <HeaderText>Gestionar administradores</HeaderText>
    <Space size={10} direction="vertical">
      <p>Otorgar privilegios de administrador a un usuario puede demorarse</p>
      <AddAdmin
        value={adminEmail}
        onChangeEmail={e => setAdminEmail(e.target.value)}
        onSubmit={onAddAdmin}/>
    </Space>
    <AdminsTable
      onDelete={onDeleteAdmin}
      loading={isLoading}
      error={error}
      admins={admins}/>
  </>
)

```

Figura 4.12. Contenido de la vista Admins en el cliente web

```

/**
 * Otorga privilegios de administración a un usuario
 * @param {String} email - Correo electrónico del usuario
 */
const addAdmin = async (email) => {
  console.log('email')

  //Se busca el usuario correspondiente al correo
  await firebaseAdmin
    .admin
    .auth()
    .getUserByEmail(email)
    .then((userRecord) => {
      //Se actualizan sus custom claims
      firebaseAdmin
        .admin
        .auth()
        .setCustomUserClaims(userRecord.uid, {admin: true})
    })
}

```

Figura 4.13. Función addAdmin del modelo users.js en el servidor

```

/**
 * Quita los privilegios de administración a un usuario
 * @param {String} uid - Identificador del usuario
 */
const deleteAdmin = async (uid) => {
  await firebaseAdmin
    .admin
    .auth()
    .setCustomUserClaims(uid, {admin: false})
}

```

Figura 4.14. Función deleteAdmin del modelo users.js en el servidor

## 4.4. Análisis, diseño e implementación de reuniones multimedia

En este apartado se describe de forma detallada el sistema de videoconferencia mediante reuniones multimedia. Para ello, se trata la estructura de la configuración de una reunión y el cómo se crea para invitar a los usuarios y establecer el mecanismo de validación de las intervenciones realizadas en la reunión. Por último, se explican las operaciones que se realizan para acceder a una reunión y el proceso de señalización para establecer la comunicación multimedia utilizando WebRTC.

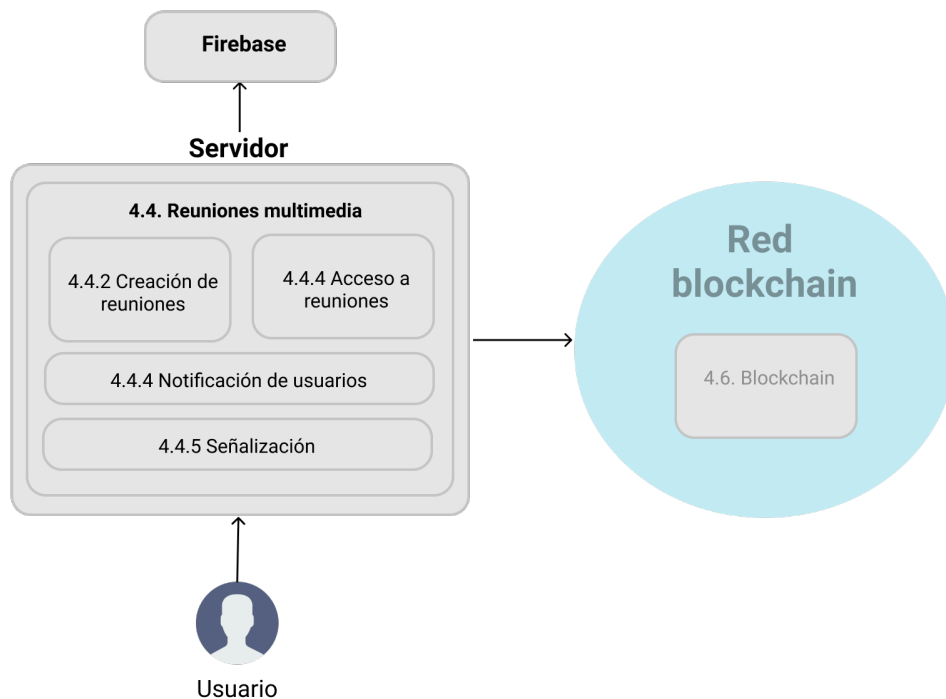


Figura 4.15. Diagrama del sistema de creación de reuniones multimedia

En la Figura 4.15 se muestran las diferentes partes que conforman el sistema de las reuniones multimedia. A continuación, se describe la estructura de datos utilizada para definir los parámetros de una reunión y las diferentes partes que habilitan la creación, acceso y realización de reuniones multimedia en el sistema.

#### 4.4.1. Estructuras de una reunión

Para almacenar las características de configuración de una reunión, se utiliza la base de datos no relacional *Firestore*. Cada reunión representa un documento en dicha base de datos dentro de la colección *meetings*. En cada documento donde se define en cada una de ellas los siguientes parámetros, los cuales se muestran en la Figura 4.16:

- *meetingID*: identificador *Universally Unique Identifier (UUID)* de la reunión.
- *name*: nombre de la reunión para que los usuarios puedan identificar la reunión fácilmente.
- *participants*: lista de usuarios invitados a la reunión, donde se almacena el correo electrónico de cada usuario invitado.
- *accessToken*: *token* generado de forma aleatoria para acceder a una reunión multimedia
- *endDate*: fecha de finalización de la reunión.
- *startDate*: fecha de inicio de la reunión.
- *percentage*: es el porcentaje de usuarios validadores que deben aceptar la intervención realizada para que sea validada en el sistema.
- *validator*: es el usuario validador que se ha designado para validar las intervenciones realizadas en la reunión.

Estos dos últimos valores, *validator* y *percentage*, se establecen dependiendo del mecanismo de validación configurado en la reunión. Si el mecanismo de validación es de *consenso por votación*, el porcentaje determinaría el número de usuarios de la reunión que deben aceptar

la intervención para que sea validada. Si se utilizara dicho mecanismo de validación, el valor del campo *validator* sería *none*.

No obstante, si el mecanismo de validación fuera de *consenso por usuario validador*, el campo *validator* contendría el correo electrónico del usuario validador encargado de aceptar o rechazar las intervenciones realizadas. En este caso, el valor del campo *percentage* sería 0 ya que no se está utilizando.

#### 4.4.2. Creación de una reunión multimedia

Para convocar nuevas reuniones, un usuario con el rol de administrador debe crear una nueva reunión multimedia desde la vista *CreationMeeting* en el cliente web. En ella, se presenta un formulario con los siguientes campos:

- *Nombre de la reunión*: campo de texto para introducir el nombre designado a la reunión
- *Añadir participantes*: campo de selección y entrada múltiple donde se introducen los correos electrónicos de los usuarios invitados a la reunión
- *Tipo de consenso*: seleccionable para elegir el mecanismo de consenso de la reunión, el cual puede ser de *Consenso por votación* o *Consenso por usuario validador*. Dependiendo del valor escogido, se mostrarían los siguientes campos en el formulario.
  - *Porcentaje de la votación*: si se escoge *Consenso por votación*, se muestra una *slider* para designar el porcentaje de votación utilizado.
  - *Usuario validador*: si se escoge *Consenso por usuario validador*, se muestra un campo de selección única para elegir el usuario validador de la reunión.
- *Fecha de la reunión*: un selector de fechas para escoger la fecha de inicio y finalización de la reunión.

Cuando se rellenan estos campos, tras enviar el formulario, se ejecuta la función *handleSubmit*, que recoge los datos introducidos en el formulario y crea un objeto con la información de la reunión.

Como se muestra en la Figura 4.17, los valores de los campos *validator* y *percentage* dependen del mecanismo de consenso utilizado. Con estos datos, se realiza una petición a la

ruta `/meetings/create-meeting` en el servidor (Figura 4.18), que utiliza la función `createMeeting` del modelo para añadir dichos datos a *Firestore*.

Para ello, como se muestra en la Figura 4.19, se genera un ID tipo UUID para la reunión y un *token* de acceso. Posteriormente, con los datos del formulario recibidos desde el cliente web, junto con el ID y el *token* de acceso a la reunión generados, se crea un nuevo registro en *Firestore* para almacenar la información de la reunión.

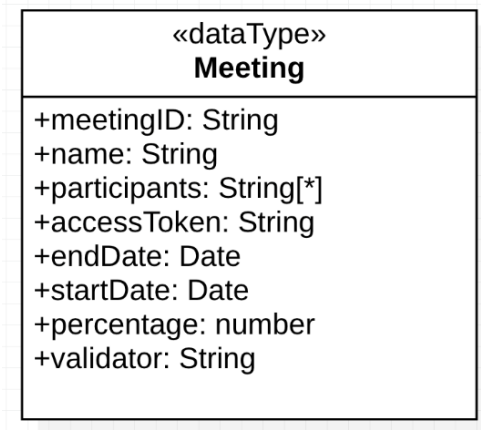


Figura 4.16. Estructura de datos de una reunión

```
const handleSubmit = (values) => {
  const meetingInfo = {
    name: values.name,
    participants: values.participants,
    startDate: values.date[0],
    endDate: values.date[1],
    validator: values.validator || 'none',
    percentage: values.consensus === 'voting'
      ? percentage
      : 0
  }

  createMeeting({...meetingInfo,})
    .then(() => {
      message.success('Reunión creada correctamente')
      form.resetFields();
    }).catch(err => {
      message.error('Ha ocurrido un error - ', err)
    })
}
```

Figura 4.17. Función `handleSubmit` en la vista `CreationMeeting` en el cliente

```

/**
 * Ruta: /meetings
 * Descripción: Crea una reunión
 */
meetingsRouter.post('/create-meeting', async (req, res) => {
  const auth = req.currentUser;

  if(auth) {
    if(auth.admin) {
      await meetingService.createMeeting(req.body)
      return res.send('meeting created')
    }
  }
  return res.status(403).send('Not authorized')
})

```

Figura 4.18. Manejador de la ruta /meetings/create-meeting en el servidor

```

/**
 * Crea en Firestore una reunión
 *
 * @param {meeting} meetingInfo - Información de la reunión a crear
 */
const createMeeting = async (meetingInfo) => {
  const {name, percentage, participants, validator, startDate, endDate} = meetingInfo
  const meetingId = uuidv4();
  const accessToken = randomstring.generate(10);

  const meetingRef = meetingsRef.doc(meetingId);

  await meetingRef.set({
    name: name,
    percentage: percentage,
    participants: participants,
    meetingId: meetingId,
    accessToken: accessToken,
    validator: validator,
    startDate: firebaseAdmin.admin.firestore.Timestamp.fromDate(new Date(startDate)),
    endDate: firebaseAdmin.admin.firestore.Timestamp.fromDate(new Date(endDate))
  })

  sendEmailMeeting(name, participants, meetingId)
}

```

Figura 4.19. Función createMeeting en el modelo meetings.js en el servidor



### 4.4.3. Notificación de usuarios

Cuando se crea una reunión, se envía un correo electrónico a los usuarios invitados a la reunión para notificar la reunión convocada y proporcionar el enlace de acceso a la misma. Para ello, se utiliza la biblioteca *nodemailer* que permite enviar correos electrónicos desde *Node.js*. Sin embargo, el proceso de correo electrónicos puede requerir cierto tiempo por lo que es recomendable que esta tarea se realice en segundo plano o en un servicio para ello.

En nuestro caso, se ha optado por la utilización de la biblioteca *bull* que, utilizando un servidor *Redis*, permite crear colas de tareas para realizar diferentes operaciones en segundo plano. Estas tareas serían servidas por un proceso secundario que ejecuta el código del archivo *worker.js* que procesa y sirve dichas tareas.

En la Figura 4.20 se muestra el funcionamiento de este sistema, en el que el servidor añade dichas tareas a una cola alojada en el servidor *Redis* y estas son luego servidas por el proceso que ejecuta *worker.js*.

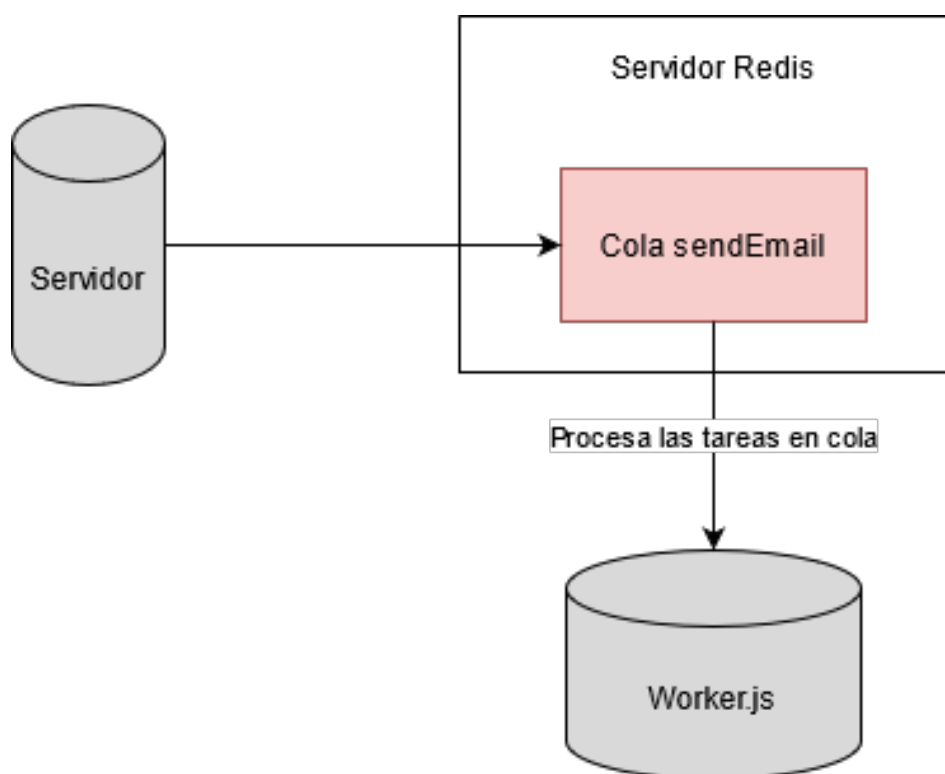


Figura 4.20. Envío de correo electrónicos en segundo plano

Para ello, cuando se crea una reunión en el servidor, se llama a la función *sendEmailMeeting*, la cual añade una tarea a la cola designada para el envío en segundo plano de correos electrónicos (Figura 4.21). A esta nueva tarea se le adjunta el contenido del correo electrónico y la lista de correos electrónicos de los usuarios que han sido convocados a la reunión.

Esta tarea es posteriormente procesada por el código del archivo *worker.js*, que se muestra en la Figura 4.22, en el cual se reciben las tareas en la cola *sendEmail*, el cual envía un correo electrónico a cada usuario de la lista recibida. En cuanto a la lógica del envío del correo electrónico, se ha creado un archivo *emailService.js* que, utilizando la biblioteca *nodemailer*, realiza el envío del correo electrónico utilizando la función *sendEmail* (Figura 4.23).

```
/**
 * Envía un email con el enlace de acceso a la reunión a la lista de usuarios indicada
 * @param {String} meeting - Nombre de la reunión
 * @param {String[]} participants - Lista de correo electrónicos de los participantes
 * @param {String} meetingID - ID de la reunión
 */
const sendEmailMeeting = async (meeting, participants, meetingID) => {
  await emailQueue.add({
    subject: `Ha sido invitado a la reunión: ${meeting}`,
    text: `https://bc-videochat.herokuapp.com/meetings/${meetingID}`,
    users: participants
  })
}
```

Figura 4.21. Función *sendEmailMeeting* en el modelo *meetings.js* en el servidor

```
/**
 * Procesa el envío de emails
 */
sendEmail.process(maxJobsPerWorker, async (job, done) => {
  const { users, subject, text } = job.data
  console.log('sending email to', users, subject, text)

  //Envía un email a cada usuario
  users.forEach(user => {
    emailService.sendEmail({
      to: user,
      subject,
      text
    })
  })

  done(null);
})
```

Figura 4.22. Código encargado de la cola de envío de correos electrónicos en *worker.js*

```

/**
 * Realiza el envío de un correo
 * @param mailDetails - Datos para realizar el envío del email
 */
const sendEmail = ({to, subject, text}) => {
  const mailDetails = {
    from: constants.EMAIL,
    to,
    subject,
    text
  }

  mailTransporter.sendMail(mailDetails, function(err, data) {
    if(err) {
      console.log('Error Occurs', err);
    } else {
      console.log('Email sent successfully');
    }
  });
}

```

Figura 4.23. Función `sendEmail` del archivo `emailService.js`

#### 4.4.4. Acceso a una reunión multimedia

Para acceder a una reunión multimedia, en la vista `NextMeetings` se muestra una tabla con la lista de reuniones cuya fecha de finalización no ha expirado. En la Figura 4.24 se muestran los componentes que componen la vista `NextMeetings`, donde el componente `NextMeetingsTable` contiene dicha tabla de reuniones pendientes.

Esta tabla contiene una serie de columnas donde se muestra el nombre de la reunión, fecha... En la Figura 4.25 se muestran las columnas de dicha tabla. Para acceder a la reunión, en una de las columnas se muestra un componente `Link` que permite redirigir al usuario a otra vista de la aplicación, según el enlace que se indique a este componente. En este caso, la columna `Entrar` contiene el componente `Link` que redirige al usuario a una nueva pestaña al enlace de la reunión multimedia. Desde dicho enlace de la reunión, el cual se puede acceder desde la vista `NextMeetings` o mediante el enlace de la reunión multimedia enviado por correo electrónico a los participantes de la reunión, se accede a la vista `Videochat`, la cual contiene la interfaz de usuario y la lógica necesaria para el establecimiento de la comunicación multimedia.

Desde esta vista en el cliente web, se comprueba en primer lugar si el usuario ha sido invitado a la reunión. Para ello, se utiliza la función `accessMeeting` en el cliente que envía una petición a la ruta `/meetings/access-meeting` con el ID de la reunión a la que el cliente quiere acceder (Figura 4.26).

```

return (
  <>
    <HeaderText>Próximas reuniones</HeaderText>
    <NextMeetingsTable
      error={error}
      loading={isLoading}
      meetings={meetings}/>
  </>
)

```

Figura 4.24. Componentes de la vista NextMeetings

```

const columns = [
  { title: 'Reunion', dataIndex: 'name', key: 'name'},
  { title: 'Fecha de inicio', dataIndex: 'start', key: 'start', responsive: ['lg'] },
  { title: 'Fecha de finalización', dataIndex: 'end', key: 'end', responsive: ['md'] },
  {
    title: 'Consenso',
    dataIndex: 'percentage',
    key: 'percentage',
    render: (percentage) => (
      <span>
        {percentage === 0 ? "Usuario validador" : `Votación - ${percentage}%`}
      </span>
    ),
    responsive: ['md']
  },
  {
    title: 'Entrar',
    dataIndex: 'action',
    key: 'action',
    render: (id) => (
      <Link
        target="_blank"
        rel="noopener noreferrer"
        to={`/meetings/${id}`}>Unirse a reunión</Link>
    )
  },
];

```

Figura 4.25. Columnas de la tabla de próximas reuniones

```

export const accessMeeting = async (meetingID) => {
  const header = await createToken();
  const getMeetingsURL = `${BACKEND_URL}/meetings/access-meeting`

  const payload = {
    meetingID
  }

  const res = await axios.post(getMeetingsURL, payload, header);
  return res.data;
}

```

Figura 4.26. Función accessMeeting del archivo meetingService.js en el cliente web

En la Figura 4.27 se muestra el proceso que realiza el servidor cuando un usuario quiere unirse a una reunión, en el que comprueba si la reunión existe y si el usuario forma parte de los usuarios invitados. Para ello, el servidor consulta en *Firestore* el contenido de dicha reunión solicitada.

En caso de existir, se comprueba si el usuario que ha realizado la petición para acceder a la reunión se encuentra entre su lista de participantes (campo *participants*). En caso de que haya sido invitado a la reunión, el servidor responde al cliente web con el *token* de acceso a la reunión para que el usuario pueda establecer la comunicación multimedia cuando se realice el proceso de señalización. En la Figura 4.28 se muestra el manejador de la petición en el servidor en el cual se realizan las comprobaciones descritas para entregar el *token* de acceso a la reunión al cliente.

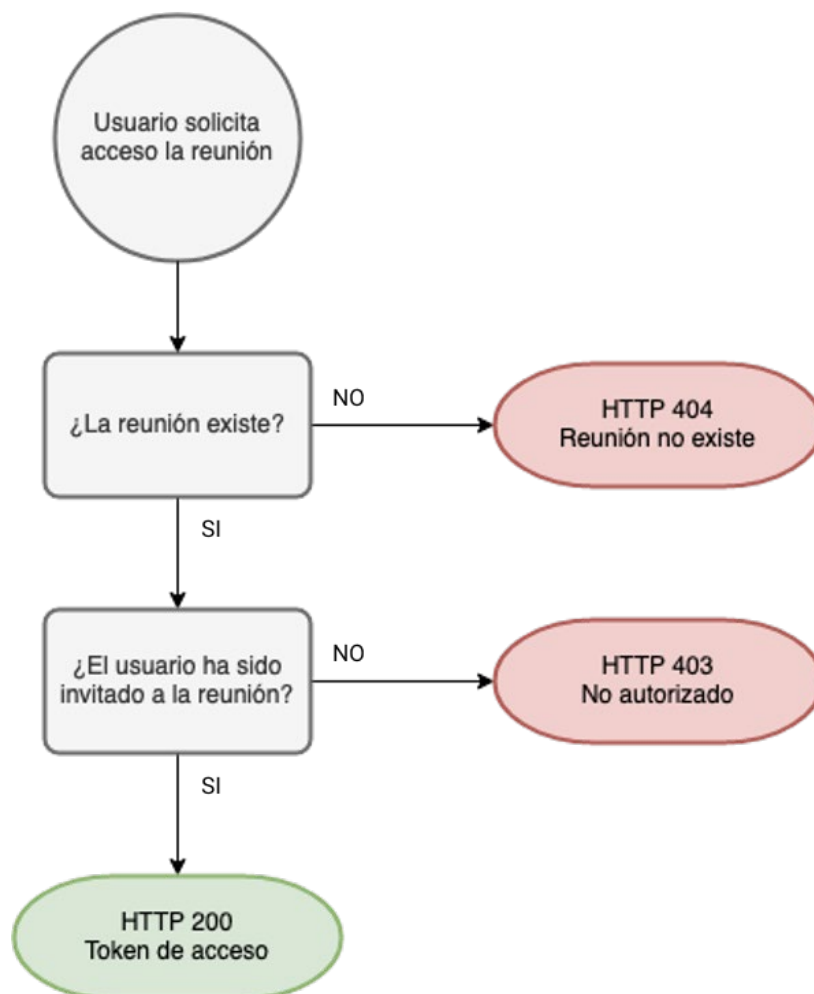


Figura 4.27. Proceso de comprobación de acceso a reunión en el servidor

```

/**
 * Ruta: /meetings/access-meeting
 * Descripción: Comprueba si el usuario tiene acceso a una reunión
 */
meetingsRouter.post('/access-meeting', async (req, res) => {
  const { meetingID } = req.body
  const auth = req.currentUser;

  if(auth) {
    console.log(req.body)
    const meeting = await meetingService.getMeeting(meetingID)

    if(meeting) {
      if(meeting.participants.includes(auth.email)) {
        return res.send(meeting.accessToken)
      } else {
        return res.status(403).send('Not authorized')
      }
    }
    return res.status(404).send('Meeting not found')
  }
  return res.status(403).send('Not authorized')
})

```

Figura 4.28. Manejador de la ruta `/meetings/access-meetings` en el servidor

#### 4.4.5. Señalización de comunicación multimedia

Una vez que el cliente web recibe el *token* de acceso a la reunión, este comienza el proceso de señalización para establecer la comunicación multimedia utilizando WebRTC. El cliente realiza un intercambio de mensajes que contienen la información necesaria para que los usuarios involucrados en la comunicación puedan establecer conexión y, posteriormente, intercambiar el tráfico multimedia.

Para comprender mejor este apartado, en el Anexo D se explica de forma detallada el proceso de señalización desarrollado, en el que se explican los diferentes mensajes intercambiados entre los pares que quieren establecer la comunicación multimedia, así como un diagrama con el flujo de mensajes intercambiado para realizar el proceso de señalización.

Para comenzar el proceso de señalización, se ejecuta el método *startConnection* en el cliente web, el cual configura la conexión utilizando la biblioteca *socket.io* para realizar el intercambio de mensajes de señalización con el servidor y comienza el proceso de señalización. A continuación, se describen los diferentes mensajes de señalización intercambiados para establecer la comunicación multimedia:

- *join*: se produce cuando un cliente quiere unirse a una reunión multimedia, enviando el ID de la reunión a la que se une y el *token* de acceso a la misma.

Cuando el servidor recibe este mensaje, primero comprueba que el *token* de acceso a la reunión coincide con el *token* almacenado en el registro de la reunión almacenado en *Firestore* mediante la función *isAuthorized* del modelo *meetings.js*. Esta función se muestra en la Figura 4.29; el código devuelve un valor booleano dependiendo de si el *token* es válido (*true*) o no (*false*) para la reunión solicitada. Tras comprobar que el *token* es válido, el servidor comprueba el número de usuarios que se encuentran en dicha sala. Si la sala está vacía, el servidor responde al cliente con un mensaje *room\_created*. Si ya se ha unido un usuario a la sala, el servidor responde al cliente con un mensaje *room\_joined*. De igual manera, se añade el *socket* que ha enviado el mensaje a un grupo, denominado *room* [25] en la API de *socket.io*, para poder transmitir los mensajes fácilmente a cada uno de los *sockets* de los clientes que se hayan conectado a la reunión multimedia.

- *room\_created*: si un cliente recibe este mensaje significa que no había ningún otro cliente conectado a la reunión multimedia. Por tanto, lo único que debe hacer es obtener el contenido multimedia propio utilizando la API *getUserMedia*, el cual se utiliza en la función *setLocalStream* para recoger el flujo multimedia del usuario (Figura 4.30), y esperar hasta que otro cliente acceda a la sala multimedia.
- *room\_joined*: si un cliente recibe este mensaje significa que debe comenzar la comunicación multimedia con los pares que ya se encuentran en la reunión. Para ello, envía el mensaje *start\_call* (Figura 4.31), el cual es reenviado por el servidor a todos los clientes que se encuentran en la sala multimedia (Figura 4.32).

```
/**
 * Se comprueba si el token de acceso a la reunión es válido
 * @param {String} meetingID - ID de la reunión
 * @param {String} meetingToken - Token de acceso a la reunión
 */
const isAuthorized = async (meetingID, meetingToken) => {
  const doc = await meetingsRef.doc(meetingID).get();

  return doc.data().accessToken === meetingToken
}
```

Figura 4.29. Función *isAuthorized* del modelo *meetings.js* del servidor

```

/**
 * Recoge el stream local multimedia usando API getUserMedia
 */
const setLocalStream = async () => {
  console.log('Local stream set')

  let stream
  try {
    stream = await navigator.mediaDevices.getUserMedia(mediaConstraints)
  } catch (error) {
    console.error('Could not get user media', error)
    setIsMediaDenied(true)
    return false;
  }

  localStreamRef.current = stream
  userVideoRef.current.srcObject = stream;
  return true;
}

```

Figura 4.30. Función `setLocalStream` de la vista `Videochat`.

```

/**
 * Mensaje room_joined al unirse a una sala con pares conectados. Comienza la llamada enviando
 * start_call
 */
socketRef.current.on('room_joined', async (event) => {
  console.log(`Current peer ID: ${socketRef.current.id}`)
  console.log(`Socket event callback: room_joined by peer ${socketRef.current.id}, joined room ${meetingID}`)

  await setLocalStream()

  console.log(`Emit start_call from peer ${socketRef.current.id}`)
  socketRef.current.emit('start_call', {
    roomId: meetingID,
    senderId: socketRef.current.id
  })
})
})

```

Figura 4.31. Manejador del mensaje `room_joined` en el cliente

```

// These events are emitted to all the sockets connected to t
socket.on('start_call', (event) => {
  socket.broadcast.to(event.roomId).emit('start_call', {
    senderId: event.senderId
  })
})

```

Figura 4.32. Manejador del mensaje `start_call` en el servidor



- *start\_call*: si un cliente lo recibe es que hay otro cliente que quiere establecer una conexión multimedia. Al recibirlo, se crea un objeto *RTCPeerConnection* para gestionar la conexión, al cual se le pasan por parámetros una lista de los servidores STUN y TURN que se usan en la comunicación (Figura 4.33).

Posteriormente, se ejecuta la función *setupPeer* (Figura 4.34) la cual configura una serie de funciones para atender a diferentes eventos que se producen durante la comunicación multimedia y el proceso de señalización:

- *Ontrack*: se produce cuando se recibe el flujo multimedia del par que con el que se ha establecido la comunicación. Es manejado por la función *setRemoteStream* que actualiza la lista de flujos multimedia recibidos para mostrarlos en pantalla al usuario.
  - *Oniceconnectionstatechange*: se produce cuando el cambia el estado del agente *Interactive Connectivity Establishment (ICE)* del cliente, el cual es manejado por la función *checkPeerDisconnect* para saber si el par se ha desconectado.
  - *Onicecandidate*: se produce cuando el agente ICE genera un candidato ICE para que sea enviado al par con el que se quiere establecer la comunicación, para realizar la negociación de candidatos ICE.
- *webrtc\_offer*: contiene la información de señalización de la conexión del otro cliente. Si un cliente recibe este mensaje, crea y configura con la función *setupPeer* un objeto *RTCPeerConnection* para gestionar la conexión con el cliente que ha enviado el mensaje y se añade la información de señalización enviada por el otro par con la función *setRemoteDescription*.

En este caso, el mensaje es solo enviado por el servidor al equipo emparejado (*peer*) que envió el mensaje *start\_call* en lugar de ser enviado a todos los usuarios de la sala multimedia. El manejador de los mensajes *webrtc\_offer*, que realiza las operaciones descritas, se muestra en la Figura 4.35.

```

/**
 * Mensaje start_call recibido y crea el objeto RTCPeerConnection para enviar la oferta al otro par
 */
socketRef.current.on('start_call', async (event) => {
  const remotePeerId = event.senderId;
  console.log(`Socket event callback: start_call. RECEIVED from ${remotePeerId}`)

  const peer = new RTCPeerConnection(iceServers)
  setupPeer(peer, remotePeerId)

  addLocalTracks(peer)

  sendOffer(remotePeerId, await createOffer(peer))
})

```

Figura 4.33. Manejador del mensaje start\_call en el cliente.

```

const setupPeer = (peer, remotePeerId) => {
  peer.ontrack = (event) => setRemoteStream(event, remotePeerId)
  peer.oniceconnectionstatechange = (event) => checkPeerDisconnect(event, remotePeerId);
  peer.onicecandidate = (event) => sendIceCandidate(event, remotePeerId)

  peersRef.current.push({
    peerID: remotePeerId,
    peer
  })
}

```

Figura 4.34. Función setupPeer de la vista Videochat en el cliente

```

/**
 * Mensaje webrtc_offer recibido con la oferta y envía la respuesta al otro par
 */
socketRef.current.on('webrtc_offer', async (event) => {
  console.log(`Socket event callback: webrtc_offer. RECEIVED from ${event.senderId}`)
  const remotePeerId = event.senderId;

  const peer = new RTCPeerConnection(iceServers)
  setupPeer(peer, remotePeerId)

  peer.setRemoteDescription(new RTCSessionDescription(event.sdp))
  console.log(`Remote description set on peer ${socketRef.current.id} after offer received`)

  addLocalTracks(peer)

  sendAnswer(remotePeerId, await createAnswer(peer))
})

```

Figura 4.35. Manejador del mensaje webrtc\_offer en el cliente

- *webrtc\_answer*: es similar al mensaje *webrtc\_offer*. La diferencia radica en que este mensaje corresponde con la respuesta del par que recibe la oferta, para responder al par con el que se desea conectar. Cuando se recibe este mensaje, se recoge la

información de señalización enviada en este por el otro cliente y se añade al objeto *RTCPeerConnection* correspondiente al par que ha enviado el mensaje *webrtc\_answer*, como se muestra en la Figura 4.36.

- *webrtc\_ice\_candidate*: Estos mensajes son enviados por los clientes cuando el agente ICE del navegador encuentra un candidato ICE. Este mensaje, con la información del candidato ICE, es necesario para que el protocolo ICE pueda establecer la ruta de conexión de igual a igual entre los clientes involucrados. El manejador de los mensajes *webrtc\_ice\_candidate* se muestra en la Figura 4.37, en el que añade el candidato ICE recibido con la función *addIceCandidate* del objeto *RTCPeerConnection* correspondiente al cliente que envió dicho candidato ICE.

```
/**
 * Mensaje webrtc_answer recibido y termina el proceso offer/answer.
 */
socketRef.current.on('webrtc_answer', async (event) => {
  const senderPeerId = event.senderId
  console.log(`Socket event callback: webrtc_answer. RECEIVED from ${senderPeerId}`)

  const peerInfo = peersRef.current.find(peer => peer.peerID === senderPeerId)

  peerInfo.peer.setRemoteDescription(new RTCSessionDescription(event.sdp))
  console.log(`Remote description set on peer ${socketRef.current.id} after answer received`)
})
```

Figura 4.36. Manejador del mensaje *webrtc\_answer*

```
/**
 * Mensaje webrtc_ice_candidate. Candidato ICE recibido de otro par
 */
socketRef.current.on('webrtc_ice_candidate', (event) => {
  const senderPeerId = event.senderId;
  console.log(`Socket event callback: webrtc_ice_candidate. RECEIVED from ${senderPeerId}`)

  // ICE candidate configuration.
  var candidate = new RTCIceCandidate({
    sdpMLineIndex: event.label,
    candidate: event.candidate,
  })

  const peerInfo = peersRef.current.find(peer => peer.peerID === senderPeerId)

  peerInfo.peer.addIceCandidate(candidate)
})
```

Figura 4.37. Manejador del mensaje *webrtc\_ice\_candidate* en el cliente.

Una vez que se han intercambiado todos los mensajes descritos en el proceso de señalización, comienza la comunicación multimedia entre los usuarios. Para ello, la vista *Videochat* (Figura 4.38) contiene los elementos *Hyper-Text Markup Language (HTML) video* para mostrar el contenido multimedia del cliente, y un *array* con los objetos *RTCPeerConnection* de cada equipo emparejado con el que se ha establecido la conexión, que se transforma en un *array* de componentes de *Video* (utilizando la función de alto nivel *map* de Javascript [26]) para cada flujo multimedia de los pares de la comunicación multimedia. Los componentes *Video*, que se muestran en la Figura 4.39, contienen elementos HTML *video* para mostrar el contenido multimedia de cada uno de los pares conectados.

Cuando un par se desconecta de conexión establecida utilizando WebRTC, se ejecuta la función *checkPeerDisconnect* (Figura 4.40) que comprueba si el cliente se ha desconectado. En caso de que su estado de conexión indica que, si se ha desconectado o ha fallado su conexión, se elimina el objeto *RTCPeerConnection* correspondiente a dicha conexión del *array* que contiene los objetos *RTCPeerConnection* correspondientes a cada conexión establecida, de forma que se deja de ver en la pantalla el contenido multimedia del usuario que se ha desconectado.

```
<div className="main_videos_videochat">
  <div className="video_grid">
    <video
      muted
      ref={userVideoRef}
      autoPlay
      playsInline
      className="user_videochat"/>
    {peerStreams.map(peer =>
      <Video
        key={peer.id}
        stream={peer.stream}
        peer={peer.id}
      />
    )}
  </div>
</div>
```

Figura 4.38. Componentes para mostrar el contenido multimedia en la vista *Videochat*

```

import React, { useRef, useEffect } from 'react';
import '../App.css'

const Video = (props) => {
  const {peer, stream} = props
  const ref = useRef();

  useEffect(() => {
    ref.current.srcObject = stream;
  }, []);

  return (
    <video
      key={peer}
      playsInline
      autoPlay
      ref={ref}
      className="user_videochat"
    />
  );
}

```

Figura 4.39. Componente Video del cliente

```

/**
 * Comprueba si el par se ha desconectado cuando recibe el evento onicestatechange del objeto RTCPeerConnection
 */
const checkPeerDisconnect = (event, remotePeerId) => {
  const peerInfo = peersRef.current.find(peer => peer.peerID === remotePeerId)
  const state = peerInfo.peer.iceConnectionState;
  console.log(`connection with peer ${remotePeerId}: ${state}`);

  if (state === "failed" || state === "closed" || state === "disconnected") {
    console.log(`Peer ${remotePeerId} has disconnected`);
    console.log('los streams', peerStreams)
    setPeerStreams(peerStreams.filter(stream => stream.id !== remotePeerId))
  }
}

```

Figura 4.40. Función checkPeerDisconnect en la vista Videochat

## 4.5. Análisis, diseño e implementación del sistema de registro de intervenciones

Partiendo de la Figura 4.3, en la Figura 4.41 se muestran las diferentes partes que conforman el sistema de registro de intervenciones para la grabación, validación y consulta de las mismas. A continuación, se describen las diferentes partes descritas y la estructura de datos utilizada para almacenar la intervención y su contenido multimedia en *Firebase*.

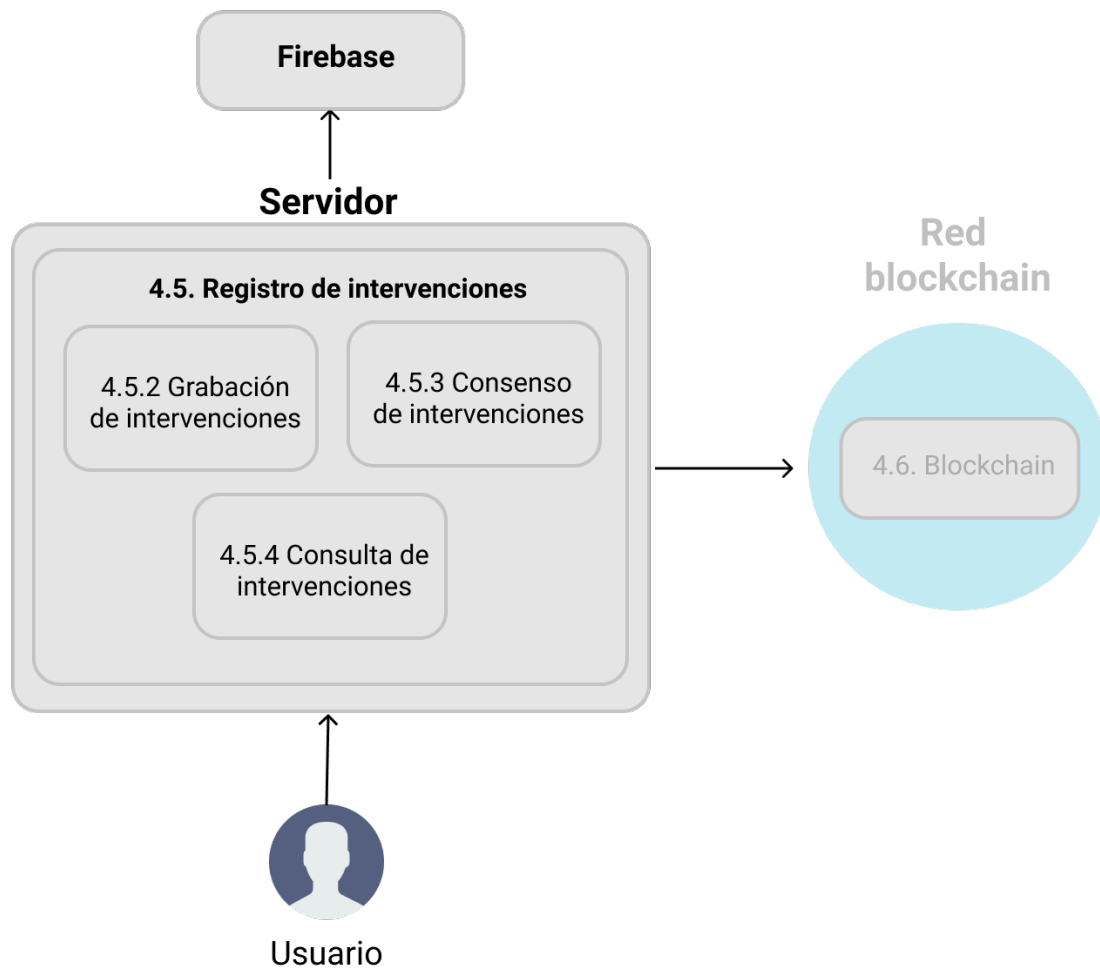


Figura 4.41. Diagrama del sistema de registro de intervenciones

#### 4.5.1. Estructura de una intervención

En una reunión multimedia, se puede guardar las intervenciones que se realicen en ella. Para ello, además de almacenar el contenido multimedia de la intervención, es necesario que se registren diferentes características de la intervención, como el autor de esta, cuando se realizó... Además, también es necesario almacenar el estado actual de la validación de la intervención dependiendo de sus mecanismos de consenso asociados, para saber si la intervención ha sido aceptada o rechazada por los usuarios validadores designados en la reunión. Para ello, se almacena en *Firestore* un documento que almacena las características descritas de una intervención y que contiene los siguientes campos, los cuales se muestran en la Figura 4.42:

- *ID*: identificador de la intervención.
- *name*: nombre de la reunión.
- *author*: correo electrónico del usuario que realizó la intervención.
- *date*: fecha en la que se realizó la intervención.
- *url*: enlace para ver el contenido multimedia de la intervención.
- *meetingID*: ID de la reunión en la cual se realizó la intervención. Se utiliza para poder relacionar las intervenciones y las reuniones almacenadas a la hora de realizar consultas a *Firestore*.
- *state*: estado de validación de la intervención. Dependiendo de si ya ha sido aceptada, rechazada o aún está pendiente de validación, toma los valores de *ACCEPTED*, *DENIED* y *PENDING*, respectivamente.
- *validationsNeeded*: número de validaciones necesarias que necesita la intervención para que sea aceptada, que depende del mecanismo de validación de la reunión.
- *accepted*: *array* de los usuarios que han aceptado la intervención.
- *denied*: *array* de los usuarios que han rechazado la intervención.
- *pending*: *array* de los usuarios que aún no han validado la intervención.

Cabe a destacar que el contenido de los *arrays* *accepted*, *denied* y *pending* iría cambiando cuando se fuera validando la intervención por los usuarios. Por ejemplo, un Usuario validador comienza estando en el *array pending*, ya que no hay validado aún la intervención. Si la acepta, se eliminaría dicho usuario del *array pending* y se añadiría al *array accepted*. De esta forma se puede saber en todo momento el estado de las validaciones de la intervención.

Dado que en una reunión pueden realizarse múltiples intervenciones, se puede establecer una relación de composición entre las estructuras de datos de las reuniones y las intervenciones. *Firestore*, al ser una base de datos no relacional basadas en documentos, permite colecciones dentro de documentos. Con esto, un registro de la base de datos de una reunión contiene una colección de registros de intervenciones, como se muestra en la Figura 4.43.

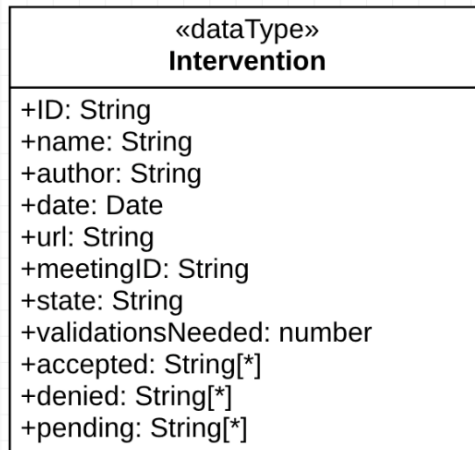


Figura 4.42. Diagrama UML de la intervención

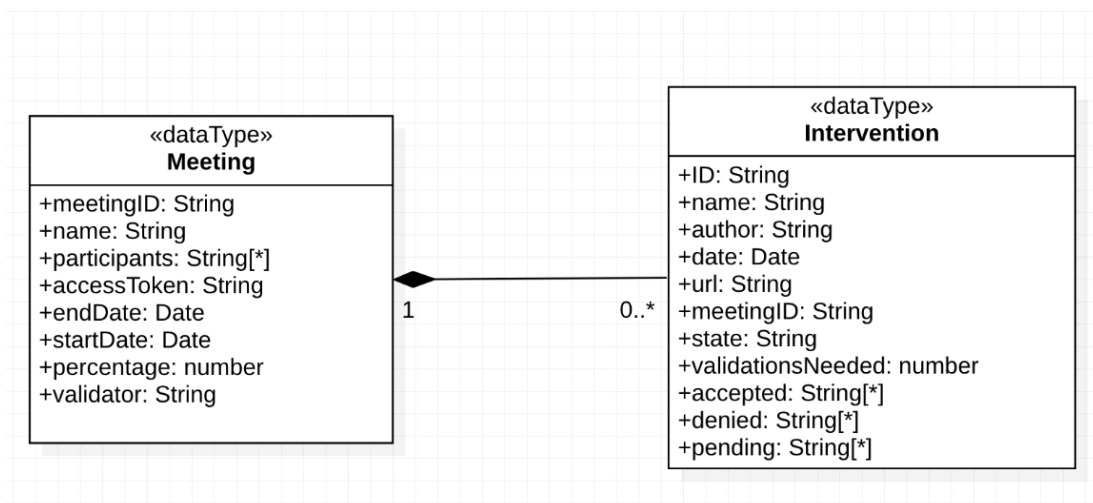


Figura 4.43. Diagrama UML de las estructuras de datos de Firestore

*Firestore* permite: realizar el diagrama en el que una reunión puede contener una colección de las intervenciones realizadas, realizar consultas muy eficientes ya que a través de una única consulta conociendo el ID de una reunión y consultar todas las intervenciones contenidas en su colección de intervenciones.

En la Figura 4.44 se muestra un ejemplo de esta relación entre intervenciones y reuniones en *Firestore*, desde la consola de administración proporcionada por *Firebase* para ver los datos almacenados. En la esquina superior izquierda de la Figura 4.44, se muestra la colección de reuniones, que contiene los registros de las reuniones almacenadas. A su vez, cada uno de los registros de las reuniones contiene una colección de intervenciones, que puede contener un conjunto de los registros de las intervenciones realizadas en dicha reunión.



e2fdf466-251e-430d-b581-b4e5bdab7b0d		
<p>+ Iniciar colección</p> <p>interventions</p> <p>+ Agregar campo</p> <pre>meetingId: "e2fdf466-251e-430d-b581-b4e5bdab7b0d" name: "Nueva reunión" participants   0 "cuenta.rubendelgado@gmail.com" percentage: 0 startDate: 6 de mayo de 2021, 09:40:59 UTC+1 validator: "cuenta.rubendelgado@gmail.co"</pre>	<p>interventions</p> <p>8b73c48a-0061-41fb-99f2-97cb073c7ee</p> <p>+ Agregar documento</p>	<p>8b73c48a-0061-41fb-99f2-97cb073c7eed</p> <p>+ Iniciar colección</p> <p>+ Agregar campo</p> <pre>accepted   0 "cuenta.rubendelgado@gmail.com" author: "cuenta.rubendelgado@gmail.com" date: 2 de junio de 2021, 16:03:15 UTC+1 denied meetingID: "e2fdf466-251e-430d-b581-b4e5bdab7b0d" name: "Intervención importante" pending state: "ACCEPTED" url: "https://firebasestorage.googleapis.com/v0/b/bc-videochat-40b6b.appspot.com/o/125fb0c1-2aa1-4353-b666-f2e52207d37e?alt=media&amp;token=50c5dea0-3545-4c89-b81e-8ff11880bf72" validationsNeeded: 1</pre>

Figura 4.44. Ejemplo de datos almacenados en Firestore.

## 4.5.2. Sistema de grabación de intervenciones

Para grabar las intervenciones realizadas desde las reuniones multimedia, la vista *Videochat*, desde la cual se realiza la comunicación multimedia, tiene un componente *InterventionRecorder* que se encarga de implementar el sistema de grabación y subida a la plataforma de las intervenciones que se realicen. Este componente (Figura 4.45) muestra un botón para detectar cuando el usuario quiere grabar una intervención y una cuenta atrás para que las intervenciones realizadas no superen un tiempo límite definido por la constante *INTERVENTION\_TIME\_LIMIT*, el cual está configurado para ser de 2 m como máximo.

```
{
  isRecording
    ? (
      <div className="countdown">
        <Badge status="processing" size="default"/>
        <Countdown
          date={Date.now() + INTERVENTION_TIME_LIMIT}
          onComplete={handleIntervention}/>
      </div>
    )
    : null
}
<Button
  type="primary"
  onClick={handleIntervention}>
  {isRecording ? "Terminar grabación" : "Grabar intervención"}
</Button>
```

Figura 4.45. Cuenta atrás y botón de grabación del componente *InterventionRecorder*

Cuando el usuario pulsa el botón para grabar una intervención, se inicia la grabación utilizando la biblioteca *useMediaRecorder*. Esta ofrece un *hook* que facilita la grabación del contenido multimedia del usuario y utiliza la API *MediaRecorder* para capturar el contenido multimedia del usuario.

Dicho *hook*, que se muestra en la Figura 4.46, ofrece diferentes variables para realizar y monitorizar la grabación, entre los que cabe destacar la variable *mediaBlob*, que contendría el contenido multimedia tras la grabación, y las funciones *stopRecording* y *startRecording* para iniciar y parar la grabación. Por otro lado, se especifica el códec del audio generado y que solo se necesita grabar el audio del cliente.

Cuando se termina la grabación del contenido multimedia del usuario, se muestra un modal con un formulario para dar nombre a la intervención y un reproductor de audio para escuchar el contenido multimedia capturado utilizando la biblioteca *ReactAudioPlayer*, como se muestra en la Figura 4.47. Además, dicho modal muestra dos botones para cancelar el guardado de la intervención si el usuario no quiere guardar la intervención y otro para guardar la intervención.

Si el usuario pulsa el botón para guardar la intervención, se ejecuta la función *handleSubmitIntervention*, que llama a la función *uploadIntervention* que se encarga de subir directamente al servicio *Firebase Storage* el contenido multimedia de la intervención. Para ello, dicha función recibe un *Binary Large Object (Blob)* del contenido multimedia capturado y un ID de la intervención.

```
let {
  status,
  mediaBlob,
  stopRecording,
  startRecording
} = useMediaRecorder({
  blobOptions: { type: 'audio/webm;codecs=opus' },
  mediaStreamConstraints: { audio: true }
});
```

Figura 4.46. Hook para grabación de contenido multimedia utilizando *useMediaRecorder*.

```

<Modal
  title="Guardar intervención realizada"
  visible={interventionRecorded}
  onCancel={handleCancelIntervention}
  confirmLoading={uploading}
  footer={[
    <>
      <Button key="back" onClick={handleCancelIntervention}>Cancelar</Button>
      <Button loading={uploading} type="primary" form="interventionForm" htmlType='submit'>
        Guardar intervención
      </Button>
    </>
  ]]}>

  <Form
    form={form}
    name="interventionForm"
    onFinish={handleSubmitIntervention}>
    <Form.Item
      label="Nombre de la reunión"
      name="name"
      rules={[{ required: true }]}>
      <Input allowClear onChange={({target}) => setInterventionName(target.value)}>
    </Form.Item>
    {
      (status === 'stopped')
      ? <ReactAudioPlayer
        src={URL.createObjectURL(mediaBlob)}
        controls/>
      : null
    }
  </Form>
</Modal>

```

Figura 4.47. Contenido del modal mostrado tras grabar una intervención

Este realiza el proceso de subida del archivo a *Firebase Storage* como se muestra en la Figura 4.48, y al terminar el proceso de subida, se ejecuta la función *createInterventionInfo*. Esta última función se utiliza para enviar al servidor la información de la intervención creada para que cree un registro en *Firestore* para dicha intervención.

La ruta del servidor que se encarga de servir a la petición para crear un registro para la intervención realizada es */interventions/new-intervention*, a la cual se envía el ID de la intervención y de la reunión en la que se ha realizado la intervención, el nombre de la intervención introducido por el usuario y el *Uniform Resource Locator (URL)* generado por *Firebase Storage* tras subir el contenido multimedia a su plataforma.

```

/**
 * Sube el contenido multimedia de una intervención a Firebase
 *
 * @param {Blob} audio - Audio de la intervención
 * @param {String} name - Nombre de la intervención
 * @param {String} meetingID - ID de la reunión
 */
export const uploadIntervention = async (audio, name, meetingID) => {
  const interventionID = uuidv4();
  const storageRef = fire.storage().ref(interventionID)
  return new Promise((resolve, reject) => {
    storageRef.put(audio)
      .then((snapshot) => {
        snapshot.ref.getDownloadURL().then(downloadURL => {
          console.log(name, downloadURL, meetingID, interventionID)
          createInterventionInfo(name, downloadURL, meetingID, interventionID)
        })
        resolve(snapshot);
      })
  })
}

```

Figura 4.48. Función `uploadIntervention` del archivo `interventionService` en el cliente

En el servidor, al recibir la petición se crea una nueva entrada en la base de datos con la información de la intervención realizada y se crea una tarea programada para validar la intervención por desistimiento positivo. Esto es, cuando pasa un tiempo determinado (en nuestro caso, 15 días) sin que la intervención haya sido aceptada o denegada, esta es validada por desistimiento positivo.

Para poder realizar esto, se programa una tarea para que se ejecute 15 días después de la realización de la intervención mediante la función `scheduleJobToValidate`. Esta función, que se muestra en la Figura 4.49, crea una tarea programada según la fecha indicada, y pasado dicho tiempo se ejecuta el código del `callback` en la función `scheduleJob`, en la que se actualiza el campo `state` de la intervención a `ACCEPTED`.

Para añadir la intervención a `Firestore`, la función `createIntervention`, del modelo `interventions`, se encarga de crear un registro para la intervención en `Firestore`. En primer lugar, dado el ID de la reunión en la que se ha grabado la intervención, se consultan los campos `validator` y `percentage` para, según características del mecanismo de validación utilizado en la reunión, se pueda designar los usuarios que se encargarían de validar la intervención y el número de validaciones necesarias para que se acepte la misma.

Estos dos parámetros se calcularían dependiendo de si el mecanismo de consenso de la reunión utiliza *Consenso por usuario validador* (en el registro de la reunión, el campo *validator* contendría el correo electrónico del usuario y el campo *percentage* sería 0) o *Consenso por votación* (en este caso, el campo *validator* sería *none* y campo *percentage* contendría el porcentaje de validaciones necesarias):

- *Consenso por usuario validador*: el usuario validador de la reunión sería el designado en el campo *validator* y el número de validaciones necesarias sería 1, ya que solo es necesaria la validación de dicho usuario.
- *Consenso por votación*: los usuarios validadores serían todos los usuarios participantes de la reunión y el número de validaciones dependería del porcentaje de validación configurado, como se muestra en la Figura 4.50.

Sabiendo estos valores, se añade a *Firestore* un registro con la información de la intervención (Figura 4.51). El campo *pending* contendría un *array* con los usuarios validadores determinados anteriormente y el campo *validationsNeeded* el número de validaciones necesarias calculado. El estado de la intervención sería *PENDING* ya que aún no ha sido validada, y los campos *accepted* y *denied* estaría vacíos ya que aún no se han realizado validaciones para aceptar o rechazar la intervención. Además, la intervención se añade a la colección *interventions* de la reunión especificada.

```
/**
 * Crea una tarea en segundo plano que se ejecuta un tiempo determinado después
 * de una intervención, para validar en caso de desistimiento positivo
 *
 * @param {String} - ID de la intervención
 */
const scheduleJobToValidate = (jobID) => {
  console.log('creando tarea secundaria', jobID)

  //Se crea la fecha en la que se debe ejecutar la tarea
  const endDate = new Date()
  endDate.setDate(endDate.getDate() + constants.DAYS_BEFORE_VALIDATION_WITHDRAWAL)

  //Se crea el trabajo programado y la operación que debe realizar
  schedule.scheduleJob(jobID, endDate, () => {
    console.log('Intervención a validar por desistimiento positivo', jobID);
    interventionService.updateInterventionState(jobID, 'ACCEPTED')
  });
}
```

Figura 4.49. Función *scheduleJobToValidate* del controlador *intervention.js* del servidor

```

const meetingInfo = await getMeetingInfo(meetingID);

const pendingValidators = meetingInfo.validator === 'none'
  ? meetingInfo.participants
  : [meetingInfo.validator]

const validationsNeeded = (meetingInfo.validator === 'none')
  ? Math.ceil(meetingInfo.participants.length * (meetingInfo.percentage / 100))
  : 1;

```

Figura 4.50. Usuarios validadores y número de validaciones necesarias para la intervención

```

await db.collection('meetings')
  .doc(meetingID)
  .collection('interventions')
  .doc(interventionID).set({
    ID: interventionID,
    author: author,
    meetingID: meetingID,
    name: name,
    date: new Date(),
    url: url,
    pending: pendingValidators,
    accepted: [],
    denied: [],
    validationsNeeded: validationsNeeded,
    state: 'PENDING'
  })

```

Figura 4.51. Código para añadir registro de la intervención a Firestore

### 4.5.3. Sistema de consenso de intervenciones

Una vez que una intervención ha sido subida a *Firebase Storage* y se ha creado un nuevo registro para la información de la intervención en *Firestore*, para que la intervención sea aceptada, esta debe ser validada por los usuarios validadores designados. Estos, pueden aceptar o rechazar la intervención y, dependiendo del estado de esta y el número de validaciones necesarias, su estado pasaría a ser aceptado o rechazado.

Para ello, en la vista *PendingInterventions* (Figura 4.51), se muestra una tabla con la lista de intervenciones pendientes de validación que permite seleccionar múltiples intervenciones, así como dos botones para aceptar o rechazar aquellas intervenciones que hayan sido seleccionadas.

```

return (
  <>
    <HeaderText>Intervenciones pendientes de validación</HeaderText>

    <div className="validation_buttons_wrapper">
      <Space>
        <Button
          onClick={() => handleClickButtons(true)}
          type="primary"
          disabled={!isSelected}>Aceptar intervención</Button>
        <Button
          onClick={() => handleClickButtons(false)}
          type="danger"
          disabled={!isSelected}>Rechazar intervención</Button>
      </Space>
    </div>

    <PendingInterventionsTable
      error={error}
      loading={isLoading}
      interventions={interventions}
      onSelect={rowSelection}/>
  </>
)

```

Figura 4.51. Componentes de la vista de intervenciones pendientes

Dicha lista de intervenciones se obtiene del servidor realizando una petición a la ruta `/interventions/pending`. En el servidor, en el modelo `interventions`, se utiliza la función `getPendingInterventions`, que consulta a `Firestore` las intervenciones pendientes de ser validadas por el usuario en cuestión. Para ello, como se muestra en la Figura 4.52, se realiza una consulta en la que se filtra por las intervenciones en las que el campo `pending` contiene el correo electrónico del usuario que realiza la petición y cuyo estado sea `PENDING`.

De nuevo, desde la vista `PendingInterventions` en la cual se muestran las intervenciones pendientes de ser validadas por el usuario, cuando el usuario acepta o rechaza las intervenciones seleccionadas se ejecuta la función `handleClickButtons` que llama a la función `validateInterventions` del archivo `interventionService` en el cliente para enviar al servidor la información necesaria para aceptar o rechazar las intervenciones seleccionadas. Esta función realiza una petición al servidor a la ruta `/interventions/validate`, en la que se envía un campo `state` (que contiene un valor dependiendo de si se ha aceptado o rechazado las intervenciones, `ACCEPTED` o `DENIED` respectivamente) y un array con el ID de la intervención y de la reunión en la que se ha realizado la intervención (Figura 4.53).

```

/**
 * Consulta a Firestore las intervenciones pendientes de validación
 *
 * @param {String} user - Email del usuario
 */
const getPendingInterventions = async (user) => {
  let interventions = [];

  const querySnapshot = await db.collectionGroup('interventions')
    .where('pending', 'array-contains', user)
    .where('state','==', 'PENDING')
    .get()

  querySnapshot.forEach((doc) => {
    interventions.push(doc.data())
  });

  return interventions;
}

```

Figura 4.52. Función `getPendingInterventions` en el modelo `interventions` en el servidor

```

/**
 * Realiza la validación de las intervenciones seleccionada
 *
 * @param {boolean} accept - Estado de la validación (true: aceptado - false: denegado)
 * @param {Interventions} intervenciones - Lista de intervenciones a validar
 */
export const validateIntervention = async (accept, interventions) => {
  const header = await createToken();
  const validateURL = `${interventionsURL}/validate`

  const state = accept ? "ACCEPTED" : "DENIED";
  const interventionsMap = interventions.map(intervention => ({
    ID: intervention.ID,
    meetingID: intervention.meetingID
  })))

  const payload = {
    interventions: interventionsMap,
    state: state
  }

  try {
    const res = await axios.post(validateURL, payload, header);
    console.log(res.data)
    return res.data;
  } catch(e) {
    console.error(e)
  }
}

```

Figura 4.53. Función `validateInterventions` del archivo `interventionService` en el cliente



En el servidor, tras recibir la petición, invoca a la función *validateIntervention* a la que se le pasan los valores enviados por el cliente y el usuario que ha realizado la petición. En esta función, la cual se muestra en la Figura 4.54, se actualiza el estado de validación de las intervenciones que se le hayan pasado a este método. Para ello, se elimina el usuario que ha realizado la validación del campo *pending* y se añade al *array denied* o *accepted* según si el usuario a aceptado o denegado la intervención.

Para evitar posibles conflictos si se llamara a esta función al mismo tiempo si dos usuarios validan una intervención a la vez, se utiliza la funcionalidad *batch commits*, que permite realizar varias operaciones de escritura en una petición a *Firestore* [27]. Para ello, se añaden las operaciones que se vayan a realizar a la variable *batch* y luego con el método *commit* se ejecutan en una sola petición.

```
/**
 * Realiza el proceso de validación para la lista de intervenciones a validar
 * @param {[interventions]} interventions - Intervenciones a validar
 * @param {String} user - Email del usuario
 * @param {String} state - Estado a validar (ACCEPTED ó DENIED)
 */
const validateIntervention = async (interventions, user, state) => {
  const batch = db.batch();

  interventions.forEach(async (intervention) => {
    console.log('Validando intervention ', intervention.ID, ' de la reunion ', intervention.meetingID, user)
    const interventionRef = db.doc(`meetings/${intervention.meetingID}/interventions/${intervention.ID}`)

    if(state === 'ACCEPTED') {
      batch.update(interventionRef,
        {
          'pending': firebaseAdmin.admin.firestore.FieldValue.arrayRemove(user),
          'accepted': firebaseAdmin.admin.firestore.FieldValue.arrayUnion(user)
        }
      )
    } else {
      batch.update(interventionRef,
        {
          'pending': firebaseAdmin.admin.firestore.FieldValue.arrayRemove(user),
          'denied': firebaseAdmin.admin.firestore.FieldValue.arrayUnion(user)
        }
      )
    }
  })
  await batch.commit();
}
```

Figura 4.54. Función *validateIntervention* del modelo *interventions.js*

Posteriormente, tras realizar la validación del usuario, se comprueba si hay validaciones suficientes para cambiar el estado de validación de una intervención. Para ello, se llama a la función *checkValidation* (Figura 4.57) a la que se le pasa la lista de intervenciones que han sido validadas. En esta función se comprueban las siguientes condiciones que se describen a continuación:

- Si la intervención es rechazada, esta no se añade al sistema de verificación de intervenciones y la podría consultar el autor de la intervención. Para que una intervención se rechace, se debe cumplir la condición que se muestra en la Figura 4.55.

$$N^{\circ} \text{ validaciones rechazadas} > N^{\circ} \text{ de validadores} - N^{\circ} \text{ validaciones necesarias}$$

*Figura 4.55. Condición para que una intervención sea rechazada*

- Si se acepta la intervención se añade al sistema de verificación de intervenciones y se cancela la tarea programada para validar por desistimiento positivo ya que, al haberse aceptado, ya no sería necesaria. Para que una intervención sea aceptada, se debe cumplir la condición que se muestra en la Figura 4.56.

$$N^{\circ} \text{ validaciones aceptadas} > N^{\circ} \text{ validaciones necesarias}$$

*Figura 4.56. Condición para que una intervención sea aceptada*

En caso de que no se cumpla ninguna de las condiciones descritas, la intervención seguiría pendiente de ser validada por los usuarios validadores restantes. Para comprobar estas condiciones, en la Figura 4.58 se muestra el código encargado que contiene estas condiciones. En primer lugar, se consulta en *Firestore*, con la función *getIntervention*, la información de la intervención y se calcula el número de usuario validadores.

Posteriormente, si se denegara la intervención, no se realiza ninguna acción y su estado se actualizaría al de denegado. Si la intervención se aceptara, se cancela el trabajo programado

para realizar la validación por desistimiento positivo, se actualiza su estado al de aceptado y comienza el proceso para añadir dicha intervención al sistema de verificación de intervenciones. Por último, si las condiciones anteriores no se cumplieran, la intervención seguiría estando pendiente de ser validada.

```
const interventionInfo = await interventionService.getIntervention(ID)

const totalValidators = interventionInfo.denied.length
                        + interventionInfo.accepted.length
                        + interventionInfo.pending.length

//Condición si la intervención ha sido denegada
const isInterventionDenied = (
  interventionInfo.denied.length > (totalValidators - interventionInfo.validationsNeeded)
)

//Condición si la intervención ha sido aceptada
const isInterventionAccepted = (
  interventionInfo.accepted.length >= interventionInfo.validationsNeeded
)
```

Figura 4.57. Código para de las condiciones para validar una intervención

```
if(isInterventionDenied) {
  await interventionService.updateInterventionState(intervention.ID, 'DENIED')
} else if (isInterventionAccepted) {
  //Cancela el trabajo programado una vez que la intervención ha sido validada
  try {
    let validateJob = schedule.scheduledJobs[ID];
    validateJob.cancel();
  } catch (error) {
    console.log(error)
  }

  //Actualiza su estado y se genera su token par añadir a la blockchain
  await interventionService.updateInterventionState(intervention.ID, 'ACCEPTED')
  await createBlockchainToken(interventionInfo.url, meetingID, intervention.ID)
} else {
  console.log(`Intervention ${intervention.ID} is still PENDING for validation`)
}
```

Figura 4.58. Operaciones realizadas según el estado de la intervención

#### 4.5.4. Consulta de intervenciones realizadas

Para consultar las intervenciones realizadas, la vista *Meetings* (Figura 4.59) muestra una tabla de todas reuniones a las que ha sido invitado el usuario. Para consultar dicha lista de reuniones, se realiza una petición a la ruta */meetings* de la API del servidor. En el código asociado a esa ruta se invoca a la función *getAllMeetings* del modelo *meetings.js* (Figura 4.60). Esta función consulta en *Firestore* las reuniones en las que el usuario que ha realizado la petición haya sido invitado.

```
return (
  <>
    <HeaderText>Reuniones realizadas</HeaderText>
    <p>
      Seleccione una reunión para consultar las intervenciones
      realizadas en dicha intervención
    </p>
    <MeetingsTable
      loading={isLoading}
      error={error}
      meetings={meetings}/>
  </>
)
```

Figura 4.59. Componentes de la vista *Meetings*

```
/**
 * Consulta a Firestore la lista de reuniones a las que ha sido invitado un usuario
 * @param {String} user - Email del usuario
 * @return {meetings} Lista de reuniones
 */
const getAllMeetings = async (user) => {
  let meetings = [];

  const snapshot = await meetingsRef.where('participants', 'array-contains', user)
    .orderBy('endDate', 'desc')
    .get();

  snapshot.forEach((doc) => {
    meetings.push(doc.data());
  });

  return meetings;
}
```

Figura 4.60. Función *getAllMeetings* en el modelo *meetings.js* en el servidor

La tabla de la vista *Meetings* contiene una serie de columnas en las cuales se muestra diferente información de la reunión. Para ver las intervenciones de una de ellas, la columna *Ver Intervenciones* (Figura 4.61) contiene el componente *Link* que redirige al usuario a la vista *Interventions* que contiene una lista de las intervenciones realizadas en la reunión seleccionada. Desde esta vista se muestra información de la reunión en la que se han realizado las intervenciones y dos tablas (Figura 4.62), en las que una contiene las intervenciones que ha realizado el usuario en la reunión y otra contiene todas las intervenciones aceptadas de la reunión.

```

{
  title: 'Ver intervenciones',
  dataIndex: 'action',
  key: 'action',
  render: (text, record) => (
    <Link to={{
      pathname: `~/dashboard/interventions/${text}`,
      interventionProps: {
        ...record
      }
    }}>Intervenciones</Link>
  )
}

```

Figura 4.61. Columnas Ver intervenciones de la tabla de la vista *Meetings*

```

<>
<Space direction="vertical" size={20}>
  <MeetingInfo
    error={meetingError}
    loading={meetingLoading}
    meeting={meeting}/>
  {!meetingError
    ? <Tabs
      defaultActiveKey="1"
      onChange={({activeKey}) => handleTabChange(activeKey)}>
      <TabPane tab="Mis intervenciones" key="user-interventions">
        <InterventionsTable
          interventions={userInterventions}
          loading={userLoading}
          error={userError}/>
      </TabPane>
      <TabPane tab="Intervenciones validadas" key="validated-interventions">
        <InterventionsTable
          interventions={validatedInterventions}
          loading={validatedLoading}
          error={validatedError}/>
      </TabPane>
    </Tabs>
    : null}
  </Space>
</>

```

Figura 4.62. Componentes de la vista *Interventions*

Para consultar la información de la reunión en concreto, se realiza una petición a la ruta */meetings/id* de la API del servidor, enviando el ID de la reunión que se está consultando. El servidor consulta a *Firestore* la información de dicha reunión con la función *getMeeting*, y como se muestra en la Figura 4.63, comprueba si el usuario que realiza la petición es uno de los participantes de la reunión, devolviendo un mensaje de error si no lo fuera, como se muestra en la Figura 4.64 en el manejador de la petición en el servidor.

Para consultar las intervenciones realizadas por el usuario, se realiza una petición a la ruta */interventions* de la API del servidor, que devuelve una lista de las intervenciones en las que el usuario haya sido el autor de la intervención. Para ello, como se muestra en la Figura 4.65, la función *getInterventions* en el modelo *interventions.js*, consulta a *Firestore* aquellas intervenciones en las que el campo *author* coincide con el usuario que realiza la petición.

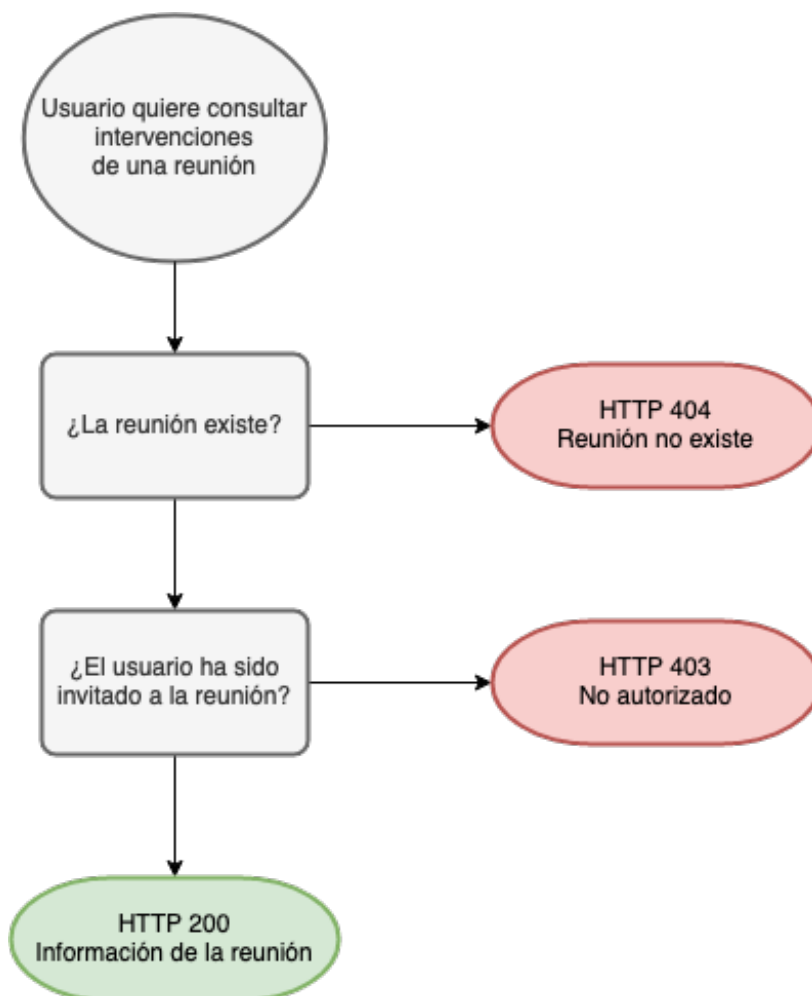


Figura 4.63. Diagrama para consultar información de una reunión

```

/**
 * Ruta: /meetings/id
 * Descripción: Devuelve la información de una reunión dado su ID
 */
meetingsRouter.post('/id', async (req, res) => {
  const { meetingID } = req.body
  const auth = req.currentUser;

  if(auth) {
    const meeting = await meetingService.getMeeting(meetingID)

    if(meeting) {
      if(meeting.participants.includes(auth.email)) {
        return res.send(meeting)
      } else {
        return res.status(403).send('Not authorized')
      }
    }
    return res.status(404).send('Meeting not found')
  }
  return res.status(403).send('Not authorized')
})

```

Figura 4.64 Manejador de la ruta /meetings/id en el controlador meetings.js en el servidor

```

/**
 * Consulta a Firestore las intervenciones realizadas por
 * un usuario en una reunión
 * @param {String} meetingID ID de la reunión
 * @return {intervention[]} Lista de intervenciones
 */
const getInterventions = async (meetingID, user) => {
  let interventions = [];

  const querySnapshot = await db.collection('meetings')
    .doc(meetingID)
    .collection('interventions')
    .where('author', '==', user)
    .get()

  querySnapshot.forEach((doc) => {
    interventions.push(doc.data())
  });

  return interventions;
}

```

Figura 4.65. Función getInterventions del modelo interventions.js

Por otro lado, para consultar todas las intervenciones que hayan sido validadas en la reunión, se realiza una petición a la ruta `/interventions/validated`, que devuelve una lista de las intervenciones que hayan sido aceptadas hasta el momento. Para ello, utilizando la función `getValidatedInterventions` (Figura 4.66) en el mismo modelo, se consultan en *Firestore* aquellas intervenciones en las que el campo `state`, el cual representa el estado de validación de la intervención (que puede ser `PENDING`, `ACCEPTED` o `DENIED`) tiene el valor `ACCEPTED`.

En ambas tablas, se muestra información de la intervención, como el autor, nombre de la intervención... En la Figura 4.67 se muestran las columnas de dicha tabla, en las que se puede acceder al contenido multimedia de la intervención haciendo *click* en el hipervínculo del URL del contenido multimedia de la columna *Ver contenido multimedia*.

Por otro lado, para ver las validaciones realizadas en cada intervención por los usuarios designados como validadores de la reunión, cada entrada de la tabla de intervenciones contiene una tabla anidada con la lista de validaciones de cada intervención. Para ello, en el campo `expandable` de la tabla de intervenciones, se pasa el componente `ValidationsTable`, el cual es el encargado de renderizar la lista de validaciones realizadas (Figura 4.68).

```
/**
 * Consulta a Firestore las intervenciones validadas en una reunión
 * @param {String} meetingID ID de la reunión
 * @return {interventions} Lista de intervenciones validadas de una reunión
 */
const getValidatedInterventions = async (meetingID) => {
  let interventions = [];

  const querySnapshot = await db.collection('meetings')
    .doc(meetingID)
    .collection('interventions')
    .where('state', '==', 'ACCEPTED')
    .get()

  querySnapshot.forEach((doc) => {
    interventions.push(doc.data())
  });

  return interventions;
}
```

Figura 4.66. Función `getValidatedInterventions` del modelo `meetings.js`



```

const columns = [
  { title: 'Nombre', dataIndex: 'name', key: 'name'},
  { title: 'ID de la intervención', dataIndex: 'key', key: 'key', responsive: ['lg'] },
  { title: 'Fecha', dataIndex: 'dateFormatted', key: 'dateFormatted', responsive: ['md'] },
  { title: 'Autor', dataIndex: 'author', key: 'author', responsive: ['md'] },
  {
    title: 'Estado',
    dataIndex: 'state',
    key: 'state',
    render: (state) => renderState(state),
    responsive: ['md']
  },
  {
    title: 'Ver contenido multimedia',
    key: 'url',
    dataIndex: 'url',
    render: (text) => <a href={text} target="_blank">Abrir enlace</a>,
  },
  {
    title: 'Verificar',
    key: 'verify',
    render: (record) => (renderVerify(record)),
  }
];

```

Figura 4.67. Columnas de la tabla de intervenciones

```

return (
  <Table
    pagination={true}
    columns={columns}
    dataSource={interventionsForTable}
    expandable={{
      expandedRowRender: record => <ValidationsTable intervention={record}/>,
    }}/>
)

```

Figura 4.68. Tabla de intervenciones en el componente InterventionsTable

## 4.6. Análisis de la blockchain implementada

Para el sistema de verificación de autenticidad de las intervenciones se ha desarrollado un prototipo de *blockchain*. Dicha *blockchain* debe ser capaz de almacenar de forma unívoca información de la intervención que permite, consultando los datos almacenados en ella, comprobar la integridad de la intervención.

Para ello, almacenar directamente en la *blockchain* el contenido multimedia de las intervenciones es, en términos computacionales, prohibitivamente caro. Esto se debe a que la *blockchain* es una base de datos distribuida, por lo que cada nodo que compone la red *blockchain* tiene una copia de la cadena de bloques. Si la cadena de bloques almacena gran cantidad de información, esta ocuparía una excesiva cantidad de almacenamiento en cada nodo y el proceso de consenso, en el que se comparte la cadena de bloques y se recorre la cadena de bloques para comprobar que los datos almacenados son válidos, podría ser extremadamente lento e ineficiente. Por este motivo, en lugar de almacenar el contenido multimedia de las intervenciones, se podría almacenar un *token* que represente dicha intervención.

Dicho *token* se crearía al procesar el contenido multimedia de la intervención a través de un algoritmo *hash* (en nuestro, *SHA-256* [28]), el cual generaría una cadena de caracteres única para el contenido multimedia de dicha intervención, tal y como se muestra en la Figura 4.69. Dicho *hash* de la intervención sería único para el contenido multimedia de la intervención, por lo que cambiar el contenido multimedia de la misma generaría un *hash* totalmente diferente.

Por tanto, la información de las intervenciones que se almacenaría de forma inalterable en la cadena de bloques sería el *hash* o *token* del contenido multimedia de las intervenciones. Además, los datos que se almacenarían en la *blockchain* pertenecerían exclusivamente a la entidad que use la aplicación web de comunicación multimedia. Esto implica que la *blockchain* que se integre con el sistema de registro de intervenciones debe ser privada o permissionada, ya que solo los usuarios autenticados dentro de la aplicación web de la entidad pueden añadir datos a la cadena de bloques y consultar los datos almacenados en la misma.

A su vez, dado que la *blockchain* implementada es privada, se usaría un mecanismo de consenso acorde a este tipo de *blockchain*. El mecanismo *PoA* es ideal para este tipo de aplicaciones cuando la *blockchain* es privada. Con dicho mecanismo de consenso se consigue una alta eficiencia de la red *blockchain* y se reduce enormemente el coste energético para añadir datos a la cadena de bloques. Para ello, desde un nodo central se escoge que nodo de la red *blockchain* debe realizar el proceso de minado, en lugar de que todos los nodos compitan para minar un bloque como ocurre en *PoW*, en el que el coste energético sería mayor.

En la Figura 4.70 se presentan las diferentes partes que conforman la *blockchain* implementada, partiendo por las estructuras de datos utilizada en el sistema y como se organiza la red *blockchain* desarrollada. Por último, se describe como, al ser una *blockchain* privada, esta es gestionada por el nodo raíz y el algoritmo de consenso utilizado para que todos los nodos tengan la última cadena de bloques válida.



Figura 4.69. Generación del token de la intervención

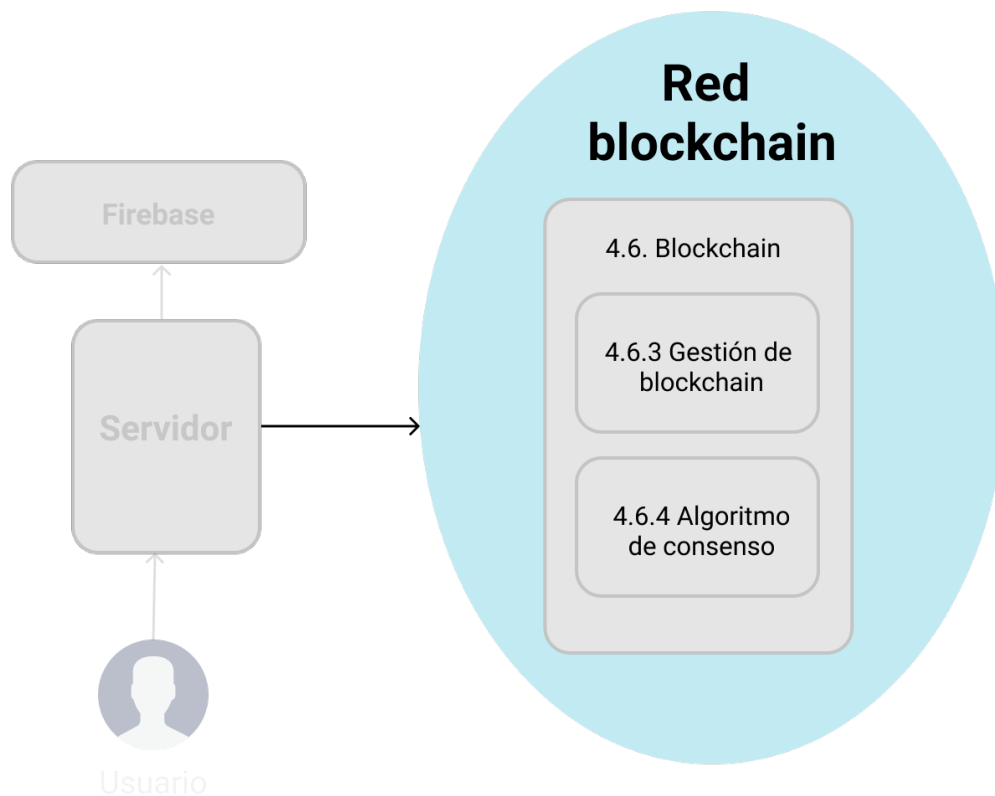


Figura 4.70. Partes que conforman la blockchain

### 4.6.1. Estructuras de datos

La cadena de bloques está formada por tres entidades: las transacciones que almacenan la información necesaria de las intervenciones, los bloques que componen la cadena de bloques y almacenan cada transacción, y, por último, la propia cadena de bloques.

En primer lugar, las transacciones contienen la información sobre la intervención: el ID de la aplicación web que ha realizado la intervención, la marca de tiempo de la creación de la transacción y el *hash* del contenido multimedia. La parte más importante de la transacción es dicho *hash*, ya que es el que se utiliza para garantizar la integridad de las intervenciones para comprobar si han sido modificadas. Dicho *hash* de la intervención se obtiene al aplicar la función SHA256 al flujo de bits del contenido multimedia mediante la biblioteca *crypto*.

Para poder identificar cada intervención, se usa un UUID para cada transacción con la biblioteca *uuid*. En cuanto a la firma de la transacción, se utiliza la implementación del algoritmo de criptografía de curvas elípticas de la biblioteca *elliptic*.

En la Figura 4.71 se muestra el diagrama *Unified Modelling Language (UML)* de una transacción, la cual está formada por dos atributos:

- *Id*: identificador de la transacción
- *Input*: conjunto de datos que almacena la transacción. Estos están definidos en una estructura de datos denominada *Input*, la cual contiene los siguientes datos:
  - *timestamp*: marca de tiempo de cuando se crea la transacción
  - *interventionToken*: *token* del contenido multimedia de la intervención.
  - *signature*: firma criptográfica utilizando el algoritmo criptográfico ECC del *token* de la intervención.
  - *videoAppID*: clave pública del par de claves de la aplicación web de comunicación multimedia que ha realizado la firma.

Además, un objeto *Transaction* debe poder validar las transacciones recibidas comprobando la firma del *token* (método *validateTransaction*) y crear un objeto *Input* que contendría los datos explicados anteriormente que almacena la transacción (método *createInput*).

La clase *TransactionPool* (en la Figura 4.72 se muestra su estructura: el atributo *transactionMap* contiene un array de objetos de la clase *Transaction*) que representa el conjunto de transacciones pendientes de ser añadidas a la cadena de bloques, debe ser tal que permita:

- Añadir transacciones a la lista de transacciones pendientes (método *setTransaction*) o vaciar la lista de transacciones pendientes (método *clear*).
- Dependiendo de las transacciones que se encuentren en la cadena de bloques, eliminar aquellas transacciones que ya se encuentren en la cadena de bloques (método *clearBlockchainTransactions*) de la lista de transacciones pendientes (Figura 4.73).
- Crear nuevas transacciones (método *createTransaction*).

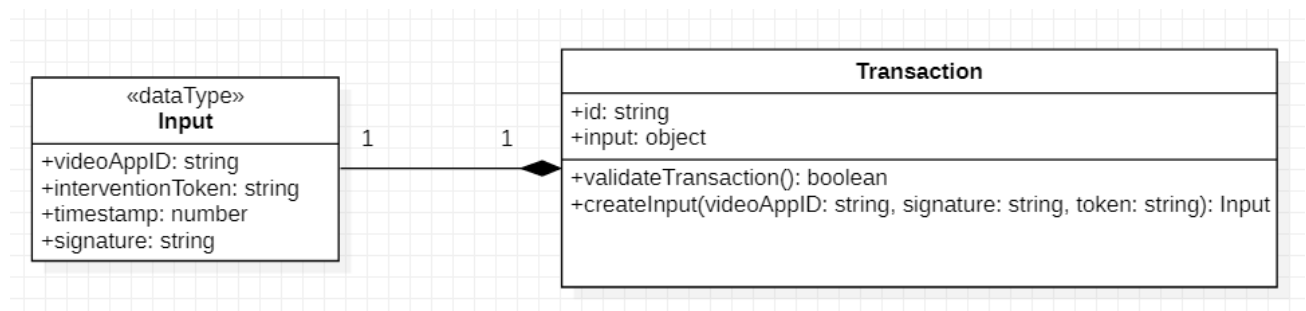


Figura 4.71. Diagrama UML de la transacción

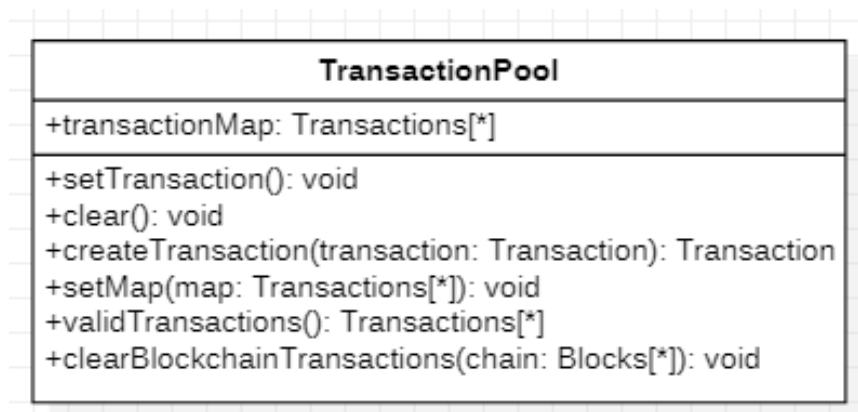


Figura 4.72. Diagrama UML de la clase *TransactionPool*

```

clearBlockchainTransactions({chain}) {
  for(let i=1; i<chain.length; i++) {
    const block = chain[i];

    for(let transaction of block.data) {
      if(this.transactionMap[transaction.id]) {
        delete this.transactionMap[transaction.id]
      }
    }
  }
}
}

```

Figura 4.73. Método *clearBlockchainTransactions* de la clase *TransactionPool*

La clase *Block* describe la lógica e información de los bloques (Figura 4.74). Los atributos que conforman un bloque son:

- *timestamp*: representa la marca de tiempo de su creación.
- *lastHash*: almacena el *hash* del bloque anterior, para crear la cadena de bloques.
- *data*: contiene la información que se guarda en la *blockchain*. Esta variable almacenaría un *array* de objetos *Transaction*, que representarían las transacciones que se han añadido a la cadena de bloques cuando se ha minado el bloque.
- *nonce*: es un campo del bloque usado para el proceso de minado.
- *difficulty*: representa el grado de dificultad del proceso de minado del bloque, el cual se explica a continuación.
- *hash*: es el *hash* generado del bloque actual.

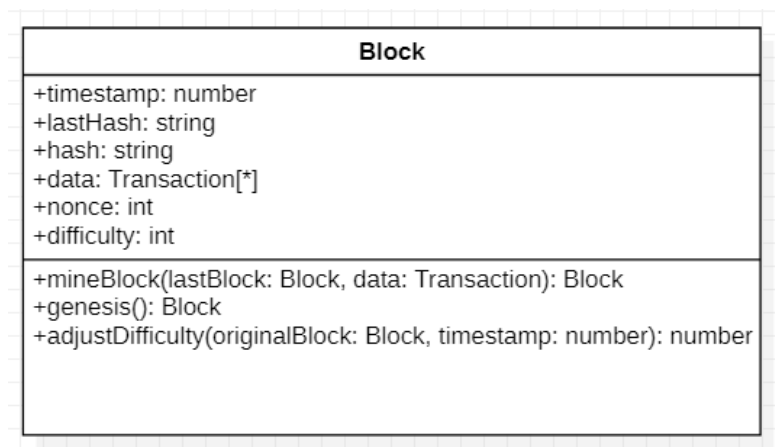


Figura 4.74. Diagrama UML de la clase *Block*

Además, un objeto *Block* debe ser capaz de realizar las siguientes operaciones:

- Generar el primer bloque que conforma la cadena de bloques (método *genesis*), el cual se crea a partir de los datos por defecto que se muestran en la Figura 4.75.
- Realizar el proceso de minado de bloques (método *mineBlock*), partiendo de un valor *nonce* inicial igual a 0, se generan *hashes* (variando el valor *nonce*) procesando toda la información que contiene el bloque (marca de tiempo, *hash* del bloque anterior, datos almacenados, *nonce* y dificultad) hasta que el *hash* generado contenga un número de ceros igual al valor de la dificultad establecida del bloque (Figura 4.76).

```
const MINING_DIFFICULTY = 10;

const GENESIS_DATA = {
  timestamp: 1622079334,
  lastHash: '----',
  hash: 'first-hash',
  difficulty: MINING_DIFFICULTY,
  nonce: 0,
  data: [],
}
```

Figura 4.75. Contenido por defecto del bloque génesis

```
/**
 * Realiza el proceso de minado de bloques
 * @param {Block} - Bloque anterior
 * @param {Transaction[*]} - Transacciones que se van a añadir
 */
static mineBlock({ lastBlock, data }) {
  const lastHash = lastBlock.hash;
  let hash, timestamp;
  let { difficulty } = lastBlock;
  let nonce = 0;

  //Proceso de minado
  do {
    nonce++;
    timestamp = Date.now();
    difficulty = MINING_DIFFICULTY
    hash = cryptoHash(timestamp, lastHash, data, nonce, difficulty)
  } while(hexToBinary(hash).substring(0, difficulty) !== '0'.repeat(difficulty));

  //Bloque creado
  return new this({ timestamp, lastHash, data, difficulty, nonce, hash })
}
```

Figura 4.76. Función *mineBlock* de la clase *Block*

La clase *Blockchain* (en la Figura 4.77 se muestra su estructura: el atributo *chain* contiene un array de objetos de la clase *Block*) que representa a un bloque de la *blockchain*, debe ser tal que permita:

- Después de minado un bloque, añadirlo (método *addBlock*) a la blockchain.
- Para poder realizar el mecanismo de consenso, utilizando el método *isValidChain* (revisa que los bloques son válidos los *hashes* generados, dificultad de minado del bloque...) y el método *isValidTransactions* (comprueba que las transacciones almacenadas son válidas comprobando su firma).
- Reemplazar la cadena de bloques actual (método *replaceChain*) por otra que se haya recibido si cumple ciertas condiciones (longitud de la cadena, validez de los bloques y transacciones almacenados...).
- Consultar las intervenciones almacenadas en la cadena de bloques (método *getTransaction*), recorriendo cada uno de los bloques que componen a la misma.

El diagrama UML de cada una de las principales estructuras de datos que se han descrito que componen la *blockchain* se muestra en la Figura 4.78, donde observa que un objeto de la clase *Block* contiene una o más transacciones (objetos de la clase *Transaction*), y la cadena de bloques (clase *Blockchain*) contiene uno o más bloques (objetos de la clase *Block*). A su vez, la clase *TransactionPool* posee una lista de las transacciones pendientes de ser añadidas a la cadena de bloques mediante el proceso de minado.

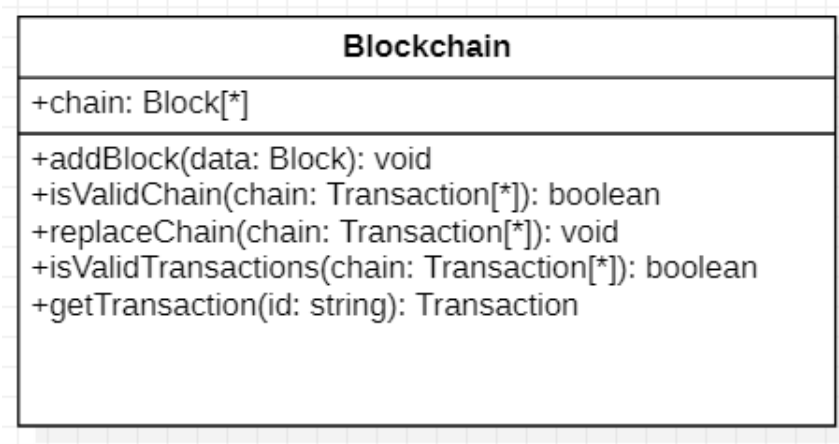


Figura 4.77. Diagrama UML de las clases *Block* y *Blockchain*



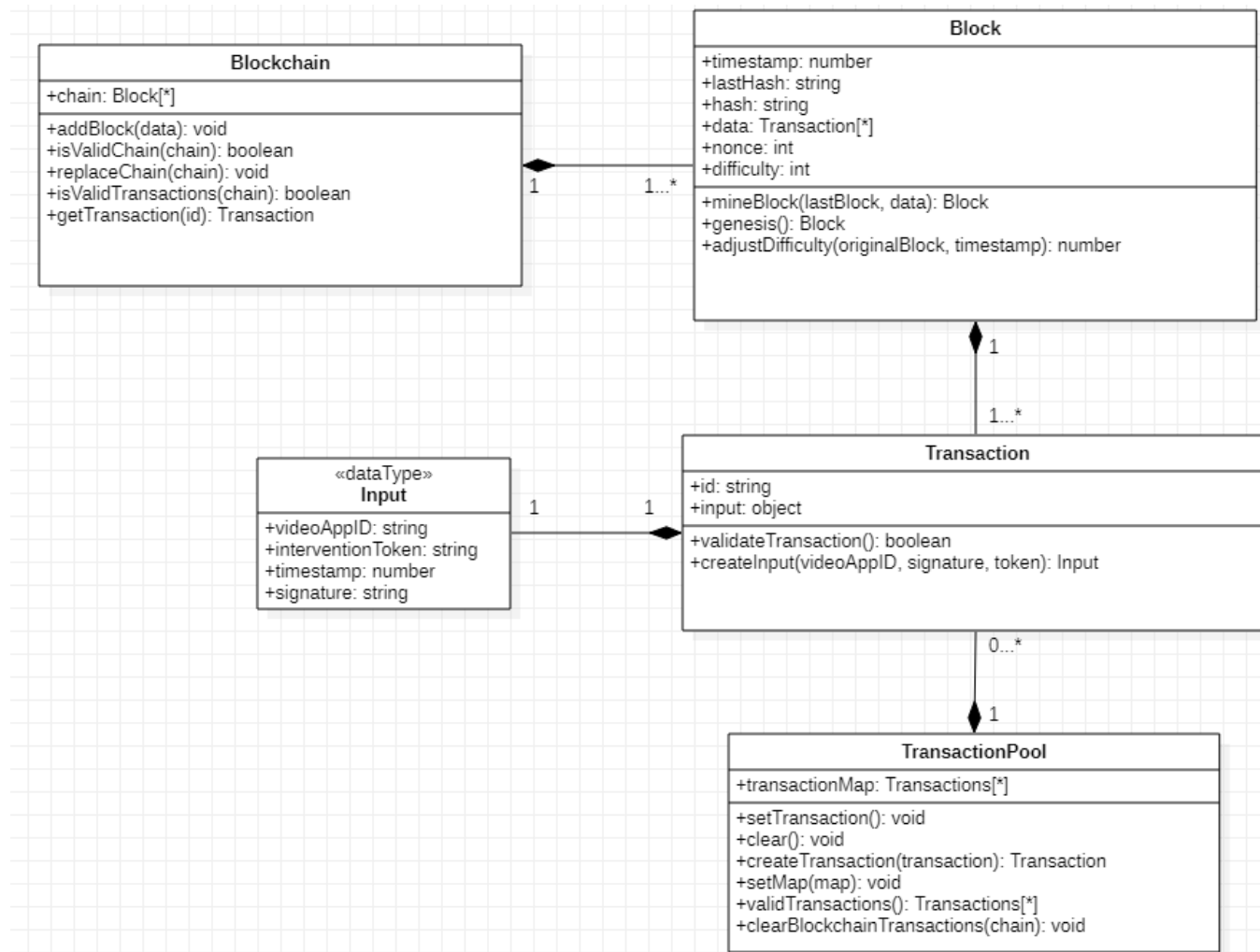


Figura 4.78. Diagrama UML de las clases principales de la blockchain implementada

#### 4.6.2. Estructura de la red blockchain

La *blockchain* implementada debe ser privada ya que la información de las intervenciones almacenadas en la misma forma parte de la entidad u organización que utilice la aplicación web. En este tipo de *blockchain*, las transacciones y bloques son validados y añadidos a la cadena de bloques por unos nodos designados previamente, los cuales se denominan validadores. Dichos nodos validadores se encargan de validar y realizar el proceso de minado de bloques de manera automática.

Sin embargo, en las *blockchain* privadas es necesaria la existencia de una unidad central o nodo raíz, el cual se encarga de gestionar las acciones realizadas en la *blockchain*. A su vez, el nodo raíz ofrece una API para la comunicación entre la aplicación web multimedia y la *blockchain*, de forma que dicha aplicación web puede consultar los datos almacenados en la misma.

Es por ello, que en la red *blockchain* implementada está compuesta por un nodo raíz, el cual es la unidad central de la *blockchain*, y una serie de nodos designados por el nodo raíz que participan en la red *blockchain*. En la Figura 4.79, se muestra un esquema de la infraestructura de la *blockchain* descrita.

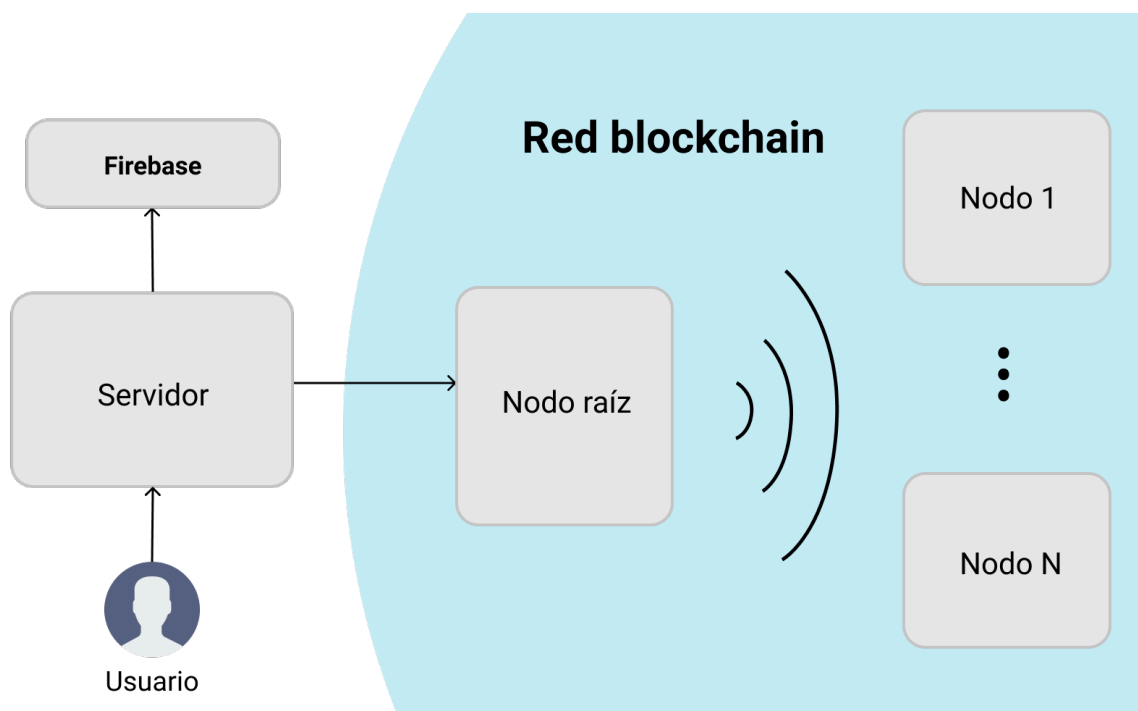


Figura 4.79. Estructura de la red blockchain

## Comunicación entre el nodo raíz y la aplicación web

Para poder añadir la información de las intervenciones realizadas en la aplicación web multimedia, el servidor de la aplicación web debe poder comunicarse con el nodo raíz de la *blockchain*. Para ello, dicho nodo raíz ofrece una API para añadir transacciones y consultar los datos almacenados en la cadena de bloques. Las rutas que ofrece son:

- */api/transact*: se encarga de añadir la transacción recibida al *pool* de transacciones, el cual contiene las transacciones pendientes de ser minada, y difundir la transacción recibida con el resto de los nodos que componen la *blockchain*.

El código implementado que maneja la petición recibida en esta ruta se muestra en la Figura 4.80, donde se muestra cómo se crea la transacción, se añade al *pool* de transacciones y, por último, se difunde al resto de nodos.

```
blockchainController.post('/transact', (req, res) => {
  console.log('[POST REQUEST] /api/transact')
  const {videoAppID, interventionID, interventionToken, signature} = req.body

  let transaction;

  //Se crea la transacción
  try {
    transaction = Singleton.getTransactionPool().createTransaction({videoAppID,
      interventionID,
      interventionToken,
      signature
    })
  } catch (error) {
    console.log('Error', transaction)
    return res.status(400).send(error.message)
  }

  //Añade la transacción al pool de transacciones
  Singleton.getTransactionPool().setTransaction(transaction)

  //Se difunde la transacción recibida
  Singleton.getPubSub().broadcastTransaction(transaction)

  res.send('Transaction received and broadcasted in the blockchain')
})
```

Figura 4.80. Manejador de la ruta */blockchain/transact*

- */api/get-transaction*: devuelve la transacción solicitada de la cadena de bloques, si existe. En la Figura 4.81 se muestra el código correspondiente a esta ruta, que hace uso del método *getTransaction* de la clase *Blockchain*, el cual se muestra en la Figura 4.82 y se encarga de buscar si existe una transacción en la cadena de bloques con el ID solicitado.

## Comunicación entre nodos de la red blockchain

Para la comunicación entre el nodo raíz y el resto de los nodos que componen la red *blockchain*, se utiliza el patrón de mensajes *Publisher/Subscriber* utilizando un servidor *Redis*. Toda la lógica de comunicación utilizando dicho patrón de mensajes se encuentra en la clase *PubSub*, cuyo UML se muestra en la Figura 4.83.

```

blockchainController.post('/get-transaction', (req, res) => {
  const { interventionID } = req.body

  const transaction = Singleton.getBlockchain().getTransaction({id: interventionID})

  if(transaction) {
    console.log('Transaction found', transaction)
    res.json(transaction)
  } else {
    console.log('Transaction not found')
    res.status(204).send('Transaction not found')
  }
});

```

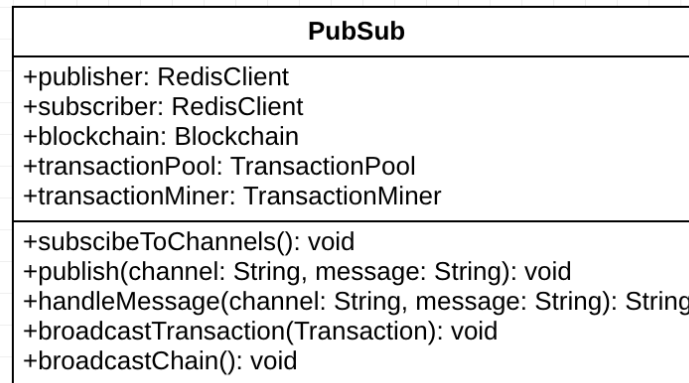
Figura 4.81. Manejador de la ruta */blockchain/get-transaction*

```

getTransaction({id}) {
  for(let block of this.chain){
    for(let transaction of block.data) {
      if(transaction.id === id) {
        return transaction;
      }
    }
  }
  return null;
}

```

Figura 4.82. Método *getTransaction* de la clase *Blockchain*



*Figura 4.83. Diagrama UML de la clase PubSub*

Esta clase tiene los atributos *publisher* y *subscriber*, los cuales se utilizan para instanciar los clientes *Redis* para el envío y la recepción de mensajes a través del patrón *Publisher/Subscriber* proporcionado por el servidor *Redis*. A su vez, tiene una referencia a la cadena de bloques (*blockchain*) y al pool de transacciones (*transactionPool*). En cuanto a los métodos de la clase *PubSub*, *subscribeToChannels*, *publish* y *handleMessage* se utilizan para la suscripción de los canales de comunicación, envío y recepción de datos a través del patrón *Publisher/Subscriber*. En el patrón *Publisher/Subscriber* implementado, hay 2 canales de comunicación:

- *BLOCKCHAIN*: para el envío y recepción de las cadenas de bloques. Este se utiliza para compartir la cadena de bloques tras realizar el proceso de minado, de forma que todos los nodos recibirían la nueva cadena de bloques
- *TRANSACTION*: para el envío y recepción de nuevas transacciones. Este se utiliza para compartir una nueva transacción (que es creada y enviada por el nodo raíz al resto de nodos) con los nodos que componen la red para realizar el proceso de minado y añadir la transacción a la cadena de bloques.

Los métodos más importantes de esta clase son *broadcastTransaction* y *broadcastBlockchain*, ya que son los utilizados para enviar nuevas transacciones y la cadena de bloques a través de los canales especificados cuando es necesario.

### 4.6.3. Gestión de la *blockchain*

En las *blockchain* privadas es necesaria la existencia de una unidad central o nodo raíz, el cual se encarga de gestionar las acciones realizadas en la *blockchain*. Dicho nodo se encarga de designar los nodos validadores, gestionar qué nodo se debe encargar de realizar el proceso de minado y ofrecer una API para consultar los datos almacenados en la *blockchain*.

Para la gestión de los nodos validadores, así como consultar el contenido de la cadena de bloques y el pool de transacciones, el nodo raíz ofrece una interfaz web donde se pueden realizar dichas acciones. Sin embargo, estas solo pueden ser realizadas por un usuario administrador. Para ello, se ha utilizado el servicio de *Firebase*, *Firebase Authentication*, para gestionar la autenticación del usuario administrador.

#### Administrar nodos validadores

El usuario administrador, desde el nodo raíz, puede gestionar qué nodos se encargan de realizar el proceso de minado. Para ello, cada nodo que compone la red *blockchain* tiene un ID, *NODE\_ID*, el cual se designa desde el archivo *.env* (Figura 4.84). Este archivo contiene las variables de entorno de la aplicación, las cuales son variables externas de la aplicación que se pueden acceder en la ejecución de la aplicación.

Dicha variable descrita permita identificar cada nodo que participe en la *blockchain*. Esto permite que el nodo raíz, sabiendo el ID del nodo, pueda añadir dicho nodo a la lista de usuarios validadores. Esta lista de usuarios administradores se almacena de forma persistente en *Firestore* en una colección denominada *nodes*, la cual guarda los IDs de los nodos validadores (Figura 4.85).

Cuando el usuario administrador necesita añadir un nodo a dicha lista de usuarios validadores, se dirige a la vista *AddNode*. Esta vista, la cual se muestra su implementación en la Figura 4.86, posee un componente *Input* que recibe el identificador del nodo que se va a añadir a la lista de nodos validadores. Posteriormente, cuando el usuario pulsa el botón de Añadir, se llama al método *addNode*, que recibe el identificador escrito en el componente *Input* y lo pasa al método *addNodeDB*.

# Node ID

NODE\_ID='17b9c926-b637-4017-848d-4080d2b0e895'

Figura 4.84. Variable de entorno NODE\_ID del archivo .env de un nodo de la blockchain

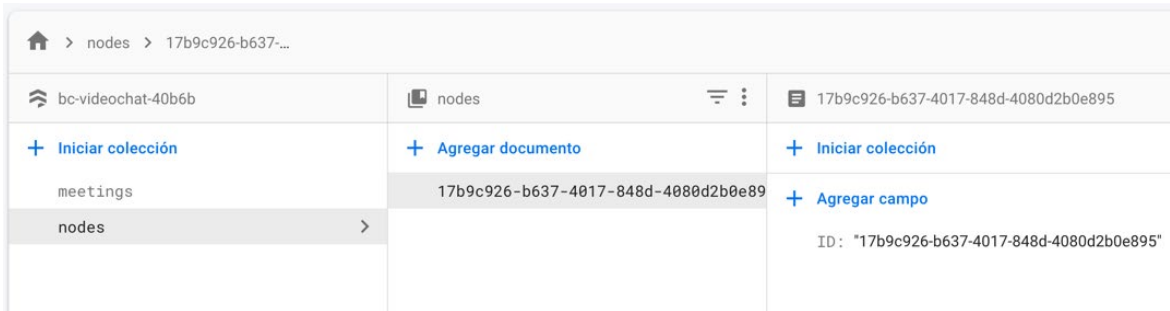


Figura 4.85. Colección de nodos validadores en la interfaz web de Firestore

```
const AddNode = (props) => {
  const [nodeID, setNodeID] = useState('')

  const addNode = async () => {
    console.log('add node', nodeID)
    try {
      await addNodeDB(nodeID)
      message.success(`Node ${nodeID} has been added successfully`)
      props.history.push('/')
    } catch (e) {
      console.error(e)
      message.error(`Node ${nodeID} could not be added`)
    }
  }

  return (
    <div className="main">
      <Title level={2}>Añadir un nodo</Title>
      <Divider />

      <Space direction="horizontal" align="center" size={20}>
        <Input
          size="large"
          placeholder="ID del nodo"
          onChange={(e) => setNodeID(e.target.value)}
        />
        <Button type="primary" onClick={addNode}>Añadir</Button>
      </Space>
    </div>
  )
}
```

Figura 4.86. Código del componente AddNode

La función *addNodeDB* envía una petición a la ruta */admin/add-node* de la API del servidor el ID del nodo que se va a añadir. El manejador de dicha ruta de la API se muestra en la Figura 4.87, donde dependiendo de si se ha podido realizar o no la operación, se devuelve una respuesta al cliente web.

Posteriormente, utilizando la función *addNode* que se muestra en la Figura 4.88, utilizando el SDK de *Firebase* para *Firestore*, se añade una entrada a la colección *nodes* con el ID del nodo añadido.

Para eliminar un nodo de la lista de nodos validadores, el proceso es muy similar. Desde el cliente web, se realiza una petición a la ruta */admin/delete-node* de la API ofrecida por el servidor, enviando junto a la petición el ID del nodo que se va a eliminar. Posteriormente, en el servidor se recibe dicha petición y se elimina dicho nodo de la colección *nodes* de *Firestore*. En la Figura 4.89 se muestra la función *deleteNode*, la cual se invoca desde el servidor realizar la acción descrita.

```
adminController.post('/add-node', async (req, res) => {
  const auth = req.currentUser;

  if(auth) {
    try {
      await adminService.addNode(req.body.ID)
      return res.status(200).send()
    } catch(error) {
      console.log(error)
      return res.status(404).send('Error creating node from database')
    }
  }
  return res.status(403).send('Not authorized')
})
```

Figura 4.87. Manejador de la ruta de la API */admin/add-node*

```
const addNode = async (id) => {
  await db.collection('nodes')
    .doc(id).set({
      ID: id
    })
}
```

Figura 4.88. Función *addNode* del servidor



```

const deleteNode = async (id) => {
  await db.collection('nodes')
    .doc(id)
    .delete()
}

```

Figura 4.89 Función deleteNode del servidor

Esta lista de nodos validadores permite al nodo raíz otorgar permiso a nodos a participar en la *blockchain*. Cuando un nuevo nodo quiere participar en la *blockchain*, este debe comunicar al usuario administrador el ID de su nodo de forma que el administrador, mediante la vista *AddNode*, añadiría dicho nodo a la lista de nodos validadores.

Posteriormente, cuando el nodo se inicia para unirse a la red *blockchain*, realiza una petición a la ruta */admin/sync-node* de la API del nodo raíz, enviando su ID, la cual se muestra en la Figura 4.90. En este código, el servidor realiza una consulta a la colección *nodes*, la cual contiene los ID de los nodos validadores, y comprueba si el ID del nodo que ha realizado la petición está en la lista de nodos validadores.

En caso de que esté, el servidor le responde con la cadena de bloques, el *pool* de transacciones actual, y el enlace para conectarse al servidor *Redis* y poder comunicarse y recibir mensajes con otros nodos. En caso de que el nodo que ha realizado la solicitud no se encuentra entre los nodos validadores, este devuelve un mensaje de error ya que dicho nodo no está autorizado para participar en la *blockchain*. El proceso descrito en el que un nodo accede a la *blockchain* se muestra en la Figura 4.91.

```

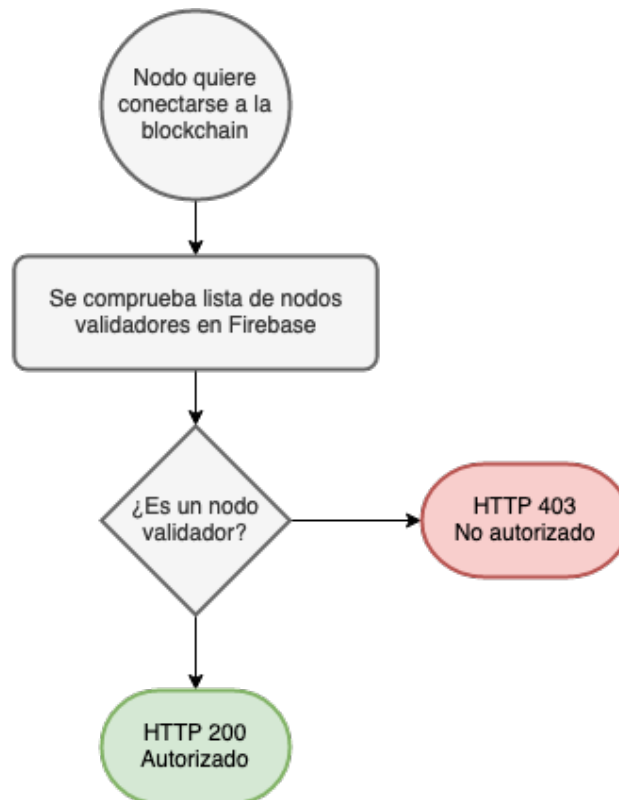
adminController.post('/sync-node', async (req, res) => {
  const { id } = req.body
  const transactionPool = Singleton.getTransactionPool().transactionMap
  const blockchain = Singleton.getBlockchain().chain
  const redisUrl = process.env.REDIS_URL || 'redis://127.0.0.1:6379'

  console.log(redisUrl)

  if(await adminService.isValidator(id)) {
    res.send({ transactionPool, blockchain, redisUrl })
  } else {
    res.status(403).send('Not authorized')
  }
})

```

Figura 4.90. Manejador de la petición */admin/sync-node* del nodo raíz



Se envía la cadena de bloques actual, el pool de transacciones y el enlace para conectarse al servidor Redis

Figura 4.91. Diagrama de conexión inicia de un nodo a la red blockchain con el nodo raíz

#### 4.6.4. Algoritmo de consenso

Cuando se mina un nuevo bloque se añade a la cadena de bloques del nodo que ha realizado el minado. Posteriormente, para que los cambios de la cadena de bloques lleguen a todos los nodos de la red *blockchain*, esta cadena de bloques a la que se le ha añadido el nuevo bloque minado se debe compartir en la red *blockchain*. Para ello, el nodo que ha realizado el minado comparte la nueva cadena de bloques a través del canal *BLOCKCHAIN* mediante la función *broadcastChain* que se muestra en la Figura 4.92.

```

broadcastChain() {
  this.publish({
    channel: CHANNELS.BLOCKCHAIN,
    message: JSON.stringify(this.blockchain.chain)
  })
}

```

Figura 4.92. Función *broadcatChain* del archivo *PubSub.js*

El resto de los nodos, cuando reciben este mensaje a través de canal *BLOCKCHAIN* (Figura 4.93), ejecutan la función *replaceChain* en la que se comprueba que la cadena de bloques recibida cumple ciertas condiciones para que sea válida. En caso de cumplirlas, se reemplaza la cadena de bloques actual por la que se ha recibido y se vacía el pool de transacciones de las transacciones que han sido añadidas a la cadena de bloques.

Dicha función *replaceChain*, que se muestra en la Figura 4.94, realiza las siguientes comprobaciones antes de reemplazar la cadena de bloques actual por la que se ha recibido:

- *La cadena de bloques recibida es menor que la actual*: si la longitud de la cadena de bloques recibida es menor que la actual, significa que se han minado menos bloques en dicha cadena de bloques que se ha recibido. Esto implica que se ha realizado menos trabajo computacional que la cadena de bloques actual, por lo que dicha cadena de bloques no sería válida.
- *Comprobación de los bloques de la cadena de bloques*: mediante la función *isValidChain* (Figura 4.95) se comprueba que los bloques están correctamente relacionados utilizando los *hashes* de cada uno de los bloques. Además, se comprueba que el *hash* del contenido del bloque fue calculado correctamente y que la dificultad no ha sido alterada.
- *Comprobación de las transacciones de cada bloque*: mediante la función *validateTransactionData* (Figura 4.96), se comprueba que cada una de las transacciones ha sido firmada correctamente y que no existen transacciones duplicadas.

```
case CHANNELS.BLOCKCHAIN:  
    this.blockchain.replaceChain(parsedMessage, true, () => {  
        this.transactionPool.clearBlockchainTransactions({  
            chain: parsedMessage  
        })  
    })  
    break;
```

Figura 4.93. Manejador de los mensajes recibidos en el canal *BLOCKCHAIN*

```

replaceChain(chain, validateTransactions, onSuccess) {
  //Se comprueba la longitud de la cadena recibida
  if(chain.length <= this.chain.length) {
    console.error('The incoming chain must be longer')
    return;
  }

  //Se comprueba el contenido de cada bloque
  if(!Blockchain.isValidChain(chain)) {
    console.error('The incoming chain must be valid')
    return;
  }

  //Se comprueba el contenido de cada transacción
  if(validateTransactions && !this.validTransactionsData({chain})) {
    console.error('The incoming chain has invalid transaction data')
    return;
  }

  if(onSuccess) onSuccess();

  console.log('Replacing chain with', chain)
  this.chain = chain
}

```

Figura 4.94. Función `replaceChain` en clase `Blockchain`

```

static isValidChain(chain) {
  if(JSON.stringify(chain[0]) !== JSON.stringify(Block.genesis())) {
    return false;
  }

  for(let i=1; i<chain.length; i++) {
    const {timestamp, lastHash, hash, nonce, difficulty, data} = chain[i];
    const actualLastHash = chain[i-1].hash;
    const lastDifficulty = chain[i-1].difficulty;

    //El hash del bloque anterior coincide con el hash almacenado del bloque anterior (cadena)
    if(lastHash !== actualLastHash){
      return false;
    }

    const validatedHash = cryptoHash(timestamp, lastHash, nonce, difficulty, data );

    //El hash del contenido del bloque fue calculado correctamente
    if(hash !== validatedHash) {
      return false;
    }

    if(lastDifficulty === difficulty) {
      return false;
    }
  }
  return true;
}

```

Figura 4.95. Función `isValidChain` de la clase `Blockchain`

```

validTransactionsData({chain}) {
  for(let i=1; i<chain.length; i++) {
    const block = chain[i]
    const transactionSet = new Set();

    for(let transaction of block.data) {
      if(!Transaction.validateTransaction(transaction)) {
        console.error('Invalid transaction')
        return false;
      }

      if(transactionSet.has(transaction)) {
        console.error('Transaction is duplicated (appears more than once in the block)')
        return false;
      }

      transactionSet.add(transaction)
    }
  }
  return true
}

```

Figura 4.96. Función *isValidTransactionData* de la clase *Blockchain*

Si se superan todas las condiciones descritas, esto implica que la cadena de bloques recibida es válida y se reemplaza la cadena de bloques actual por la que se ha recibido. De esta manera, todos los nodos de la cadena de bloques tienen la última cadena de bloques cada vez que se realice el minado de un bloque.

## 4.7. Análisis, diseño e implementación de la verificación de intervenciones

En este apartado se describe la integración de la *blockchain* en el sistema para lograr el sistema de verificación de autenticidad de las intervenciones validadas por los usuarios. En la Figura 4.97 se muestran las diferentes partes que conforman el sistema de verificación de intervenciones, el cual tiene una gran relación con la *blockchain* desarrollada, ya que esta almacena los *tokens* de las intervenciones que permiten comprobar la autenticidad de las intervenciones.

A continuación, se describen las diferentes partes que conforman el sistema de verificación de intervenciones que utiliza la *blockchain* desarrollada para comprobar la autenticidad de las intervenciones almacenadas.

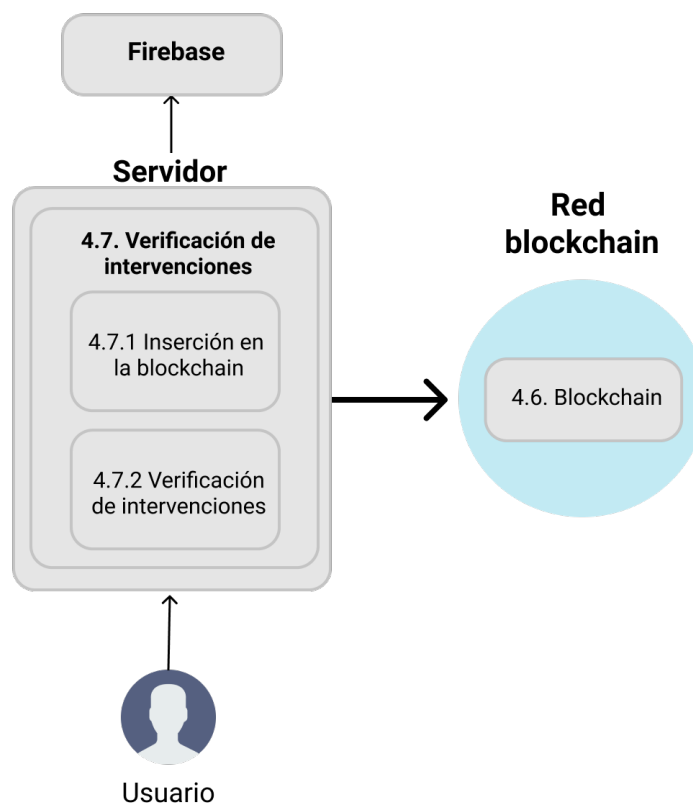


Figura 4.97. Diagrama del sistema de verificación de intervenciones

#### 4.7.1. Inserción de información de intervenciones en la blockchain

Cuando una intervención ha sido aceptada tras ser validada por los usuarios validadores designados en la videoconferencia, se debe generar el *token* de la intervención y, posteriormente este debe ser añadido a la cadena de bloques de la *blockchain*. Para generar el *token* de la intervención, se ejecuta la función *createBlockchainToken* en el servidor de la aplicación web de comunicación multimedia.

Dicha función, que se muestra en la Figura 4.98, se añade una tarea para generar el *token* de la intervención, la cual tiene que ser ejecutada en segundo plano. Esto se debe a que descargar el contenido multimedia de la intervención y calcular el *hash* de este puede demorarse. Por ello, de forma similar al envío de correos electrónicos mediante tareas en segundo plano, el proceso de cálculo de *tokens* de intervenciones se realiza mediante este sistema de realización de tareas mediante la biblioteca *bull* y el servidor *Redis*, como se muestra en la Figura 4.98.

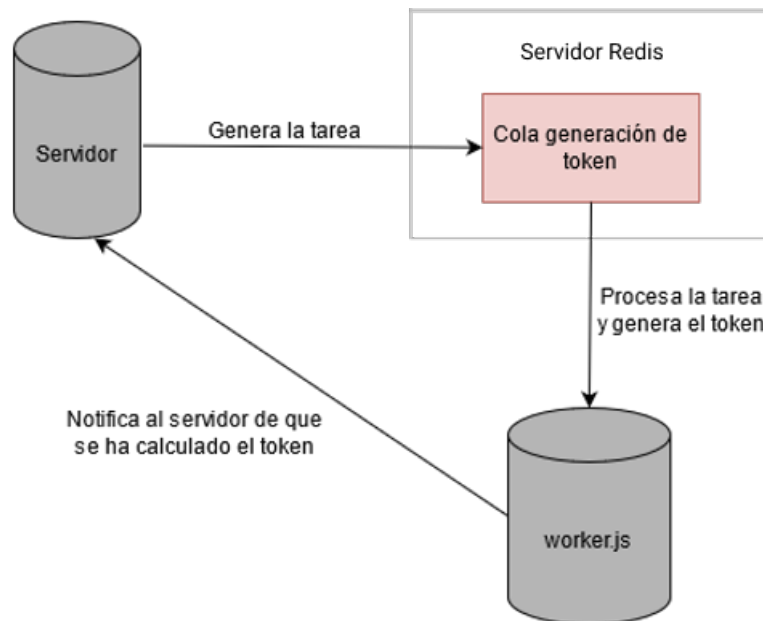


Figura 4.98. Cálculo de hash en segundo plano

En el archivo *worker.js* se encuentra el código encargado de procesar las tareas de generación del *token* de las intervenciones (Figura 4.99). Al terminar de calcular el *hash*, este devuelve un objeto *result* con el *token* de la intervención y el ID de la reunión y de la intervención mediante la función *done*. Para el cálculo del *hash* del contenido multimedia de la intervención, se invoca a la función *getAudioHash*, la cual descarga el audio del URL que se adjunta en la tarea y calcula el *hash* del contenido multimedia como se muestra en la Figura 4.100.

Cuando se completa la tarea descrita, en el controlador *interventions.js* se encuentra el código encargado de recibir el evento de que se ha completado la tarea y ejecuta la función *addToBlockchain* (Figura 4.101) que envía a la ruta */api/transact* de la API del nodo raíz el *token* de la intervención para que este sea añadido a la cadena de bloques, así como la firma de dicho *token* y la clave pública del par de claves con el que se ha realizado la firma, para certificar que fue añadido por la aplicación de comunicación multimedia donde se ha realizado la intervención.

En el nodo raíz, cuando se reciben estos datos, como se muestra en la Figura 4.102, se genera un objeto transacción con el ID de la intervención, el *token* generado, y la firma y clave pública utilizada para la misma mediante la función *createTransaction*, que comprueba que el *token* fue firmado correctamente con la clave pública recibida.

```

/**
 * Procesa cálculo de hash del audio para añadir a la blockchain
 */
workQueue.process(maxJobsPerWorker, async (job, done) => {
  console.log('[work] New intervention to hash received: ', job.data)
  const {url, meetingID, interventionID} = job.data

  let interventionToken;
  try {
    interventionToken = await getAudioHash(url)
  } catch(error) {
    throw new Error('Error al realizar proceso de hash', error);
  }

  console.log(interventionToken)

  const result = {
    interventionToken: interventionToken,
    interventionID: interventionID,
    meetingID: meetingID
  }

  console.log('New intervention hashed', result)
  done(null, result);
});

```

Figura 4.99. Código encargado de procesar las tareas de generación de tokens

```

/**
 * Descarga el audio del url y genera un hash (token del audio) a partir del mismo
 * @param {String} url - URL del contenido multimedia
 */
const getAudioHash = async (url) => {
  //Descarga el audio
  const config = {
    responseType: 'stream'
  };

  let response;
  try {
    response = await axios.get(url, config)
  } catch(error) {
    throw new Error('Audio no encontrado')
  }

  //Calcula el hash del audio
  var hash = crypto.createHash('sha256');
  hash.setEncoding('hex');
  return new Promise((resolve) => response.data.pipe(hash).on('finish', () => {
    hash.end()
    resolve(hash.read())
  }
  ))
}

```

Figura 4.100. Función getAudioHash del archivo worker.js



```

/**
 * Añade el token de una intervención a la blockchain
 * @param interventionID - ID de la intervención
 * @param intervenitonToken - Hash generado del audio de la intervención
 */
const addToBlockchain = async (interventionID, interventionToken) => {
  const signature = videoKeyPair.sign(interventionToken)
  const videoAppID = videoKeyPair.getPublicKey()

  payload = {
    videoAppID,
    interventionID,
    interventionToken,
    signature
  }

  try {
    await axios.post(`${BLOCKCHAIN_ROOT_URL}/api/transact`, payload)
  } catch (e) {
    console.error(e);
  }
}

```

Figura 4.101. Función *addToBlockchain* en el modelo *blockchain.js*

Si la firma es válida, la transacción se añade al *pool* de transacciones del nodo raíz y se comparte con el resto de los nodos de la red *blockchain* para realizar el proceso de minado mediante la función *broadcastTransaction*. Dicha función, que se encuentra en el archivo *pubsub.js*, elige de forma aleatoria un nodo entre la lista de nodos validadores almacenados en *Firestore*. Posteriormente, envía un mensaje que contiene la transacción a añadir a la cadena de bloques y el ID del nodo encargado de realizar el proceso de minado. Este mensaje se envía a través del canal *TRANSACTION*, como se muestra en la Figura 4.103.

Desde los nodos de la *blockchain*, el mensaje es recibido a través del canal *TRANSACTION* como se muestra en la Figura 4.104, en el que se añade la transacción recibida al *pool* de transacciones. A continuación, comprueban si el ID del nodo validador elegido para realizar el proceso de minado es el mismo que el suyo. En caso de que esto ocurra, comienza el proceso de minado mediante la función *mineTransactions* y cuando este proceso termine, se comparte la cadena de bloques con el nuevo bloque al resto de nodos de la red *blockchain* mediante la función *broadcastChain* para realizar el proceso de consenso.

```

blockchainController.post('/transact', (req, res) => {
  console.log('[POST REQUEST] /api/transact')
  const {videoAppID, interventionID, interventionToken, signature} = req.body

  let transaction;

  //Se crea la transacción
  try {
    transaction = Singleton.getTransactionPool().createTransaction({videoAppID,
      interventionID,
      interventionToken,
      signature
    })
  } catch (error) {
    console.log('Error', transaction)
    return res.status(400).send(error.message)
  }

  //Añade la transacción al pool de transacciones
  Singleton.getTransactionPool().setTransaction(transaction)

  //Se difunde la transacción recibida
  Singleton.getPubSub().broadcastTransaction(transaction)

  res.send('Transaction received and broadcasted in the blockchain')
})

```

Figura 4.102. Manejador de la petición /api/transact del nodo raíz de la blockchain

```

async broadcastTransaction(transaction) {
  //Se elige el nodo que realiza el minado
  const nodes = await adminService.getNodes()
  const peerToMine = getRandomInt(0, nodes.length-1)
  const peerToMineID = nodes[peerToMine].ID

  const message = {
    peerToMineID,
    transaction
  }

  //Envía el mensaje en el canal TRANSACTION
  this.publish({
    channel: CHANNELS.TRANSACTION,
    message: JSON.stringify(message)
  })
}

```

Figura 4.103. Función broadcastTransaction del archivo pubsub.js

```

case CHANNELS.TRANSACTION:
    this.transactionPool.setTransaction(parsedMessage.transaction)

    if(process.env.NODE_ID === parsedMessage.peerToMineID) {
        this.transactionMiner.mineTransactions();
        this.broadcastChain();
    }

    break;

```

Figura 4.104. Manejador de mensajes recibidos en el canal TRANSACTION en los nodos

Para realizar el proceso de minado, se recogen todas aquellas transacciones del *pool* de transacciones que sean válidas (el *token* de la intervención almacenada en la transacción haya sido firmado correctamente) y realiza el minado de dichas transacciones mediante la función *addBlock*. Dicha función genera un bloque minado mediante la función *mineBlock* de la clase *Block* (Figura 4.105), que realiza el proceso de minado generando *hashes* cambiando el valor del *nonce* hasta encontrar un *hash* que contenga el número de ceros consecutivos indicados en campo *difficulty*.

Este proceso de minado se muestra en la Figura 4.76, donde se procesa el *hash* del contenido del bloque hasta encontrar dicho *hash*.

```

mineTransactions() {
    //consulta las transacciones pendientes validas
    const validTransactions = this.transactionPool.validTransactions();

    if(validTransactions.length === 0) {
        console.log('There is no transactions to be mined')
        return;
    }

    //mina un bloque para las transacciones elegidas
    this.blockchain.addBlock({ data: validTransactions })

    //vacía el pool de transacciones tras el proceso de minado
    this.transactionPool.clear();
}

```

Figura 4.105. Función *mineTransactions* de la clase *TransactionMiner*

## 4.7.2. Verificación de intervenciones

Desde la vista *Interventions* los usuarios pueden verificar el contenido de las intervenciones que hayan sido validadas desde haciendo *click* en la columna Verificar. Esta columna es *renderizada* mediante la función *verifyInterventions*, como se muestra en la Figura 4.106, que muestra un componente *Link* que lleva a al usuario a la vista *InterventionVerification* si la intervención ha sido validada.

Desde la vista *InterventionVerification* se realiza el proceso de verificación de integridad de la intervención que se haya seleccionado. Los pasos seguidos para el proceso de verificación se muestran en la Figura 4.107, los cuales consisten en:

1. Consultar en la cadena de bloques si la transacción que contiene la intervención que se está consultando se encuentra en la cadena de bloques. En caso de no existir, implicaría que la transacción aún no ha sido añadida a la cadena de bloques. Si existiera, se usaría el *token* de la intervención almacenado en la transacción para realizar el proceso de verificación.
2. Comprobar que la firma de la transacción es correcta.
3. Calcular el *hash* del contenido multimedia de la intervención.
4. Comparar el *token* almacenado en la *blockchain* con el *hash* calculado del contenido multimedia. Si no coincidieran, significaría que el contenido multimedia ha sido alterado ya que no coincide con el *hash* almacenado en la cadena de bloques cuando se validó la intervención.

```
const renderVerify = (intervention) => {
  if(intervention.state === 'ACCEPTED') {
    return (
      <Link to={{
        pathname: `/dashboard/verify/${intervention.ID}`,
        interventionProps: {
          ...intervention
        }
      }}>Verificar</Link>
    )
  } else {
    return <Text disabled>Verificar</Text>
  }
}
```

Figura 4.106 Función *renderVerify* en el componente *InterventionTables*

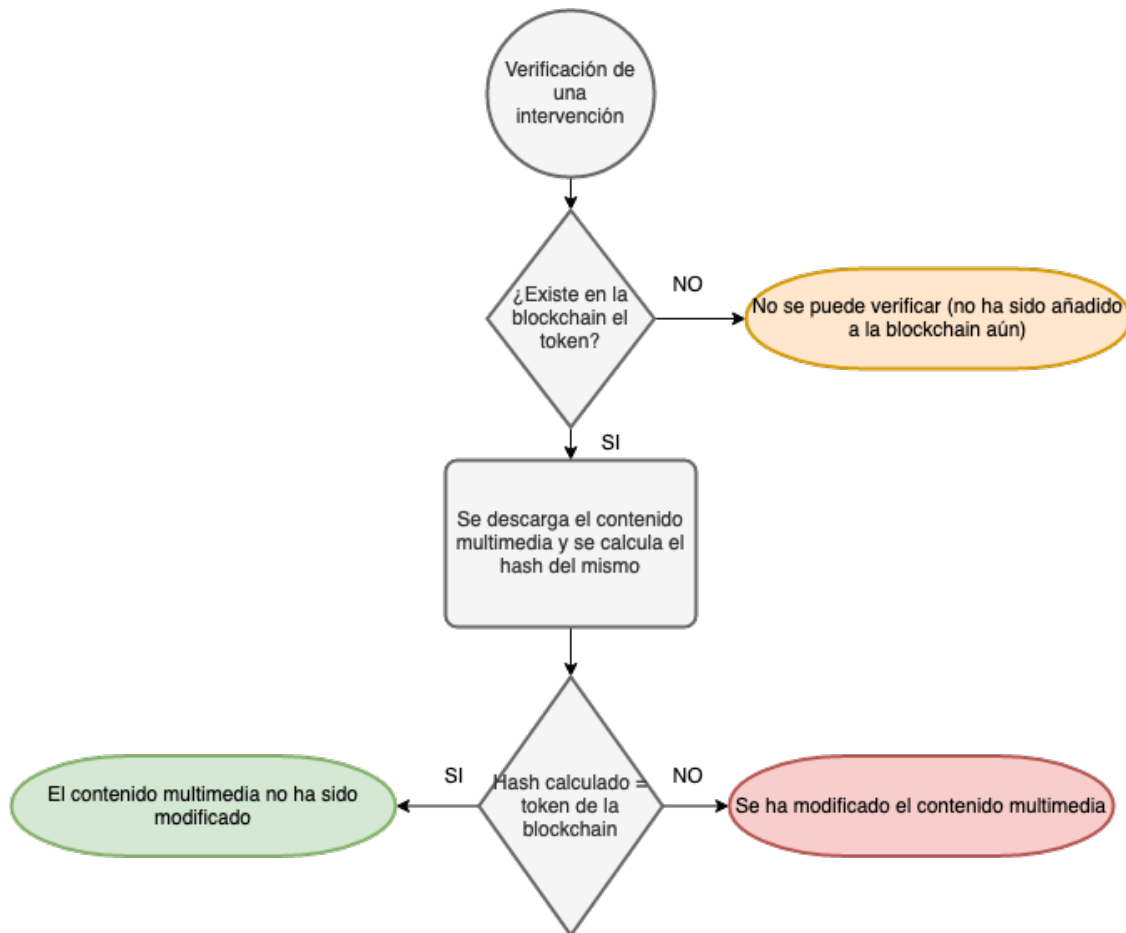


Figura 4.107. Proceso de verificación de autenticidad de una intervención

Para consultar en la cadena de bloques la transacción que contiene el *token* de la intervención que quiere consultar, el servidor web realiza una petición a la ruta `/api/get-transaction` de la API del nodo raíz de la *blockchain*. En la Figura 4.108 se muestra el manejador de dicha ruta en el servidor.

```

blockchainController.post('/get-transaction', (req, res) => {
  const { interventionID } = req.body

  const transaction = Singleton.getBlockchain().getTransaction({id: interventionID})

  if(transaction) {
    console.log('Transaction found', transaction)
    res.json(transaction)
  } else {
    console.log('Transaction not found')
    res.status(204).send('Transaction not found')
  }
});

```

Figura 4.108. Manejador de la ruta `/api/get-transaction` en el nodo raíz

Desde aquí, el nodo raíz invoca a la función *getTransaction* (Figura 4.109), la cual recorre cada una de las transacciones almacenadas en la cadena de bloques hasta encontrar aquella cuyo ID coincide con el ID de la intervención que se está consultando.

Una vez se encuentra la transacción se envía como respuesta al servidor web. Posteriormente, la aplicación web realiza una petición a la ruta */blockchain/verify* para que el servidor web realice el proceso de cálculo del *hash* del contenido multimedia de la intervención. Dado que el proceso de cálculo del *hash* puede demorarse ya que requiere cierta capacidad de procesamiento por parte del servidor y es necesario descargar el contenido multimedia de *Firebase*, esta tarea se realiza en segundo plano, de igual manera a como se realiza el cálculo del *hash* cuando se añade el *token* de una intervención a la *blockchain*.

Además, se necesita enviar al cliente web el resultado del proceso de cálculo de *hash* para que este pueda comparar el *token* de la intervención almacenado en la *blockchain* con el *hash* calculado en el servidor. Para ello, se realiza una conexión a través de *socket.io* que permite al servidor enviar al cliente web el resultado del cálculo del *hash* cuando este termine. En la Figura 4.110 se muestra el diagrama del proceso de cálculo de *hash* del contenido multimedia de la intervención, enviando posteriormente el resultado al cliente a través de una conexión mediante *socket.io*.

```
getTransaction({id}) {  
  for(let block of this.chain){  
    for(let transaction of block.data) {  
      if(transaction.id === id) {  
        return transaction;  
      }  
    }  
  }  
  return null;  
}
```

Figura 4.109. Función *getTransaction* de la clase *TransactionPool*

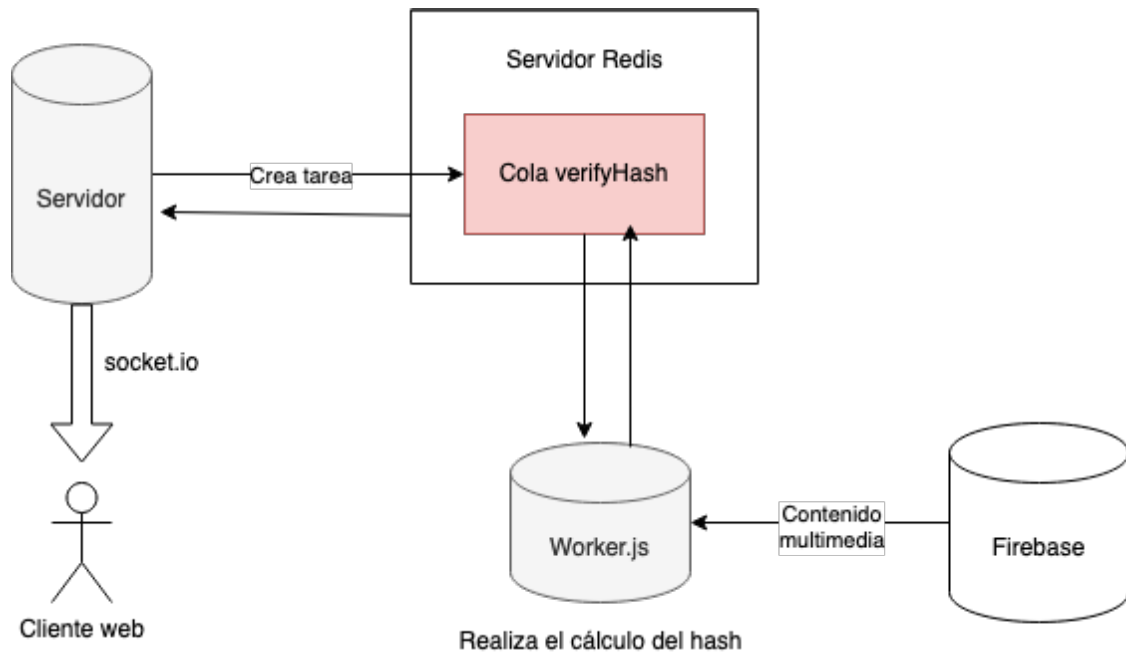


Figura 4.110. Proceso de cálculo de hash del contenido multimedia

Una vez que recibe el *hash* calculado, el cliente web compara el *token* solicitado de la intervención que estaba almacenado en la *blockchain* con el *hash* calculado recientemente por el servidor. En caso de que no coincidieran significaría que el contenido multimedia de la intervención almacenado ha sido alterado o modificado, ya que no coincide con el *hash* que se añadió a la *blockchain* cuando se validó la intervención. Si coinciden, se puede asegurar que el contenido multimedia no ha sido modificado.





# 5. Análisis de interfaz de usuario y resultados

---

En este capítulo se analiza la interfaz de usuario para la aplicación web de comunicación multimedia y la administración del nodo raíz de la *blockchain*, describiendo las funcionalidades de cada vista que compone la interfaz desarrollada. A su vez, también se realiza una evaluación de los resultados experimentales obtenidos en cuanto al rendimiento de la *blockchain* implementada y la comunicación multimedia en la aplicación web.

## 5.1. Interfaz de usuario de la aplicación web multimedia

A continuación, se muestran las distintas pantallas de la aplicación web multimedia, comentando brevemente qué se muestra en cada una de ellas y el uso por parte del usuario. La aplicación web está publicada en *Heroku* [29].

### Login

La primera vista que vería el usuario al acceder a la aplicación web por primera vez se muestra en la Figura 5.1. En esta, se muestra una vista de inicio explicando brevemente en que consiste la aplicación y un formulario de inicio sesión para que el usuario introduzca sus credenciales y acceda a la aplicación.

Tras iniciar sesión, el usuario puede acceder a las diferentes vistas que componen la aplicación multimedia. En primer lugar, a la izquierda se despliega un menú para acceder a las diferentes funcionalidades de la aplicación web. Dependiendo de si el usuario es administrador o un usuario normal, se mostrarían las siguientes opciones.

Si el usuario no es administrador, en la Figura 5.2 se muestra el menú para dichos usuarios, en el cual se muestra la navegación a las siguientes vistas:

- *Reuniones*: tabla de reuniones sin finalizar a las que ha sido invitado el usuario.
- *Validar*: tabla de las intervenciones pendientes de ser validadas por el usuario.
- *Consultar*: tabla de las reuniones en las que ha participado el usuario.
- *Ayuda*: pantalla de ayuda donde se explica el uso de la aplicación.

### Aplicación web de comunicación multimedia

\* Usuario:


\* Contraseña:  

Figura 5.1. Interfaz de usuario de la vista Login

Si el usuario es administrador, además de las opciones descritas, se ofrecen diferentes opciones para gestionar el funcionamiento del sistema, como administrar los usuarios y crear reuniones (Figura 5.3). Las opciones que ofrece son:

- *Crear una reunión*: formulario para crear una nueva reunión multimedia.
- *Administradores*: tabla con los usuarios administradores.
- *Añadir usuario*: formulario para dar acceso a la aplicación a un nuevo usuario.

En la parte superior de la interfaz, se muestra el componente *HeaderBar*, el cual muestra el nombre de la entidad que gestiona la aplicación web y el nombre del usuario que ha iniciado sesión. Por último, también se ofrece un botón para cerrar sesión a la derecha, tal y como se muestra en la Figura 5.4.

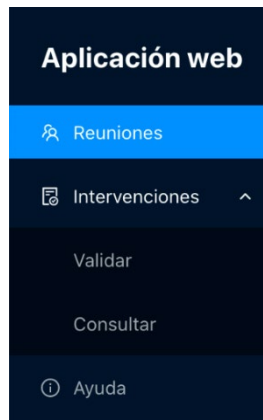


Figura 5.2. Menú para los usuarios no administradores

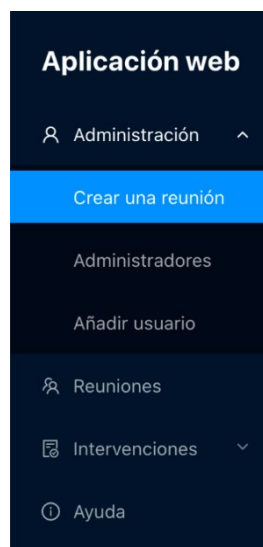


Figura 5.3. Menú para usuarios administradores

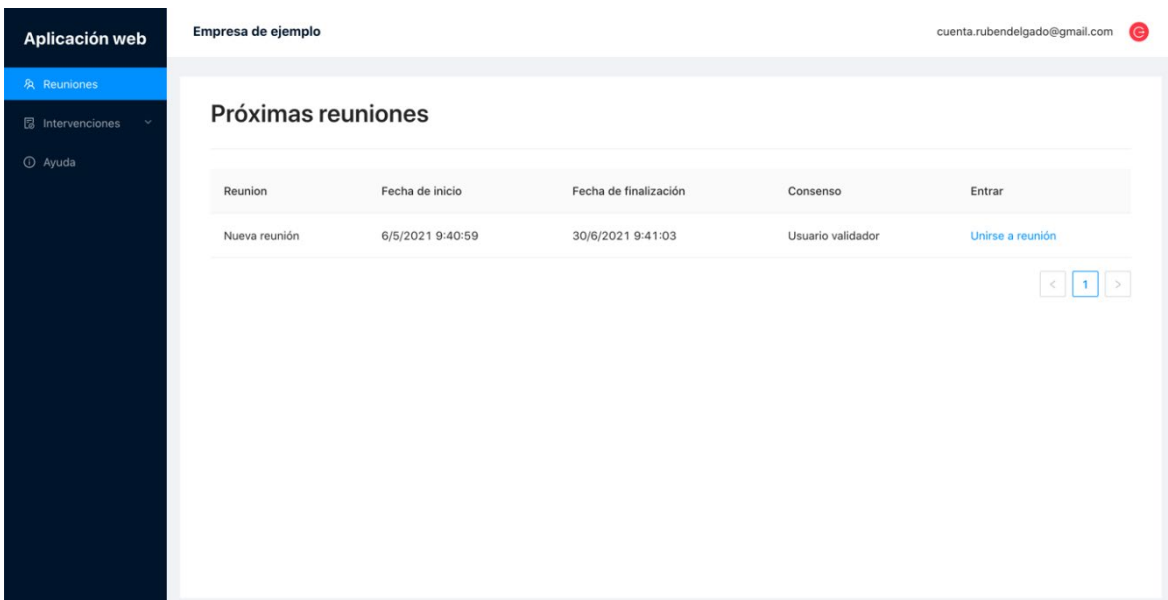
Figura 5.4. Cabecera de la aplicación web

## NextMeetings

En esta interfaz de usuario, correspondiente a la vista *NextMeetings* (Figura 5.5), se muestra una tabla de las reuniones a las que el usuario ha sido invitado y aún no se haya cumplido la fecha de finalización de la reunión. En cada entrada de la tabla, que corresponde a una reunión, se muestra el nombre de la reunión, la fecha de inicio y finalización de esta, el mecanismo de validación (el cual puede ser por *Usuario Validador* o por *Votación*) y un hipervínculo para acceder a la reunión multimedia.

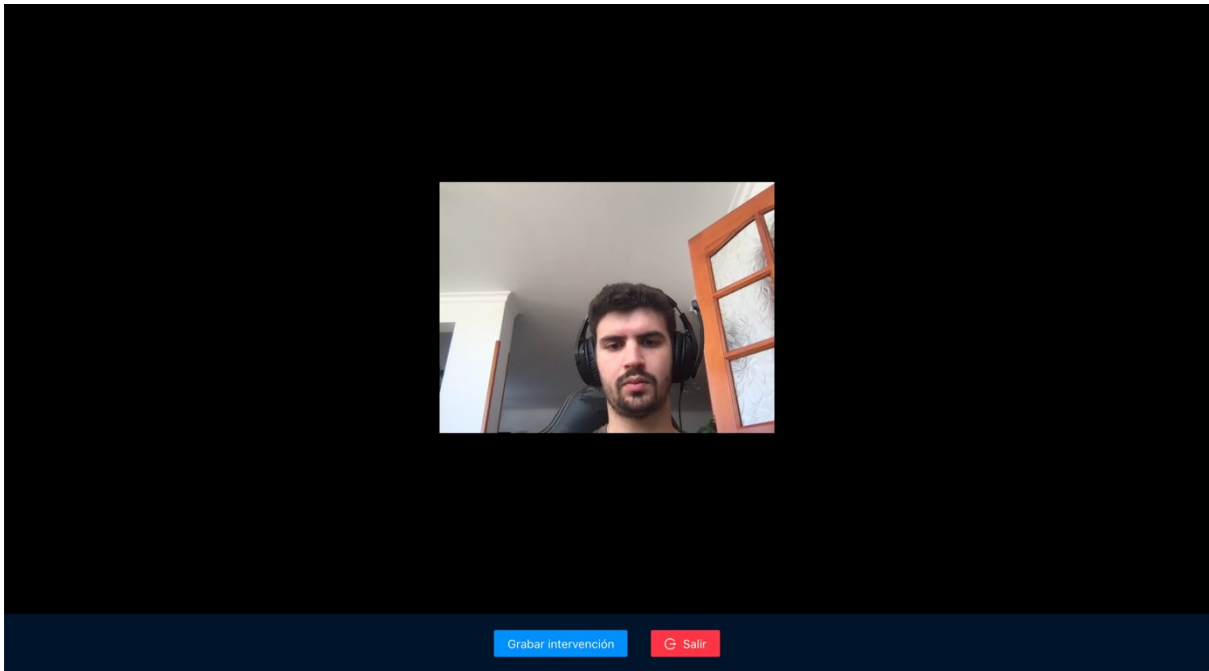
## Videochat

En esta vista se muestra la interfaz de usuario para la comunicación multimedia entre los clientes (Figura 5.6). En ella, se muestra un cuadro de vídeo con el contenido multimedia para cada uno de los participantes que se encuentren en la reunión. Por último, en la parte inferior se muestra un botón para grabar el contenido de la intervención y otro para salir de la reunión multimedia.



Reunion	Fecha de inicio	Fecha de finalización	Consenso	Entrar
Nueva reunión	6/5/2021 9:40:59	30/6/2021 9:41:03	Usuario validador	<a href="#">Unirse a reunión</a>

Figura 5.5. Interfaz de usuario de próximas reuniones



*Figura 5.6. Interfaz de usuario de la sala de la reunión multimedia*

Si el usuario no hubiera sido invitado a la reunión o esta no existiera, se mostraría un mensaje de error (Figura 5.7) advirtiéndolo al usuario de que no puede acceder ya que no está autorizado a acceder a la reunión.

### **PendingInterventions**

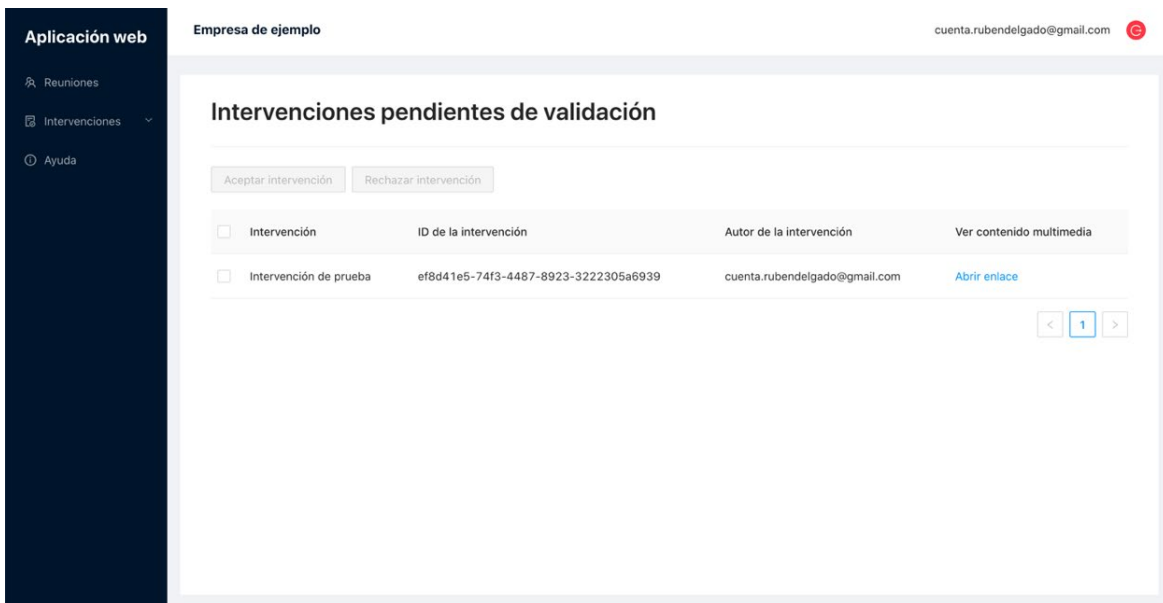
En esta interfaz de usuario, correspondiente a la vista *PendingInterventions*, se muestra una tabla con las intervenciones que están pendientes de ser validadas por parte del usuario (Figura 5.8). En cada una de las entradas de la tabla, en la que cada una corresponde a una intervención, se muestra el nombre e identificador de la intervención, el usuario que ha realizado la intervención y un hipervínculo para reproducir el contenido multimedia de la intervención.



## **Error al acceder a la reunión**

No está autorizado a entrar en esta reunión

*Figura 5.7. Mensaje de error al acceder a una reunión*



*Figura 5.8. Interfaz de usuario de las intervenciones pendientes de validación*

Para aceptar o rechazar las intervenciones, se ofrece dos botones para dichas acciones (los cuales se encuentran deshabilitados si no se ha seleccionado ninguna intervención), así como un sistema de selección múltiple para elegir aquellas intervenciones que el usuario validaría.

## Meetings

En esta interfaz de usuario, correspondiente a la vista *Meetings*, se muestra una tabla con reuniones en las que el usuario ha sido invitado (Figura 5.9), de forma que este puede elegir aquella reunión de cual quiera consultar sus intervenciones. Cada entrada de la tabla, que corresponde a una reunión, muestra el nombre e identificador de la reunión, la fecha de finalización de la reunión y un enlace para consultar las intervenciones realizadas.

## Interventions

En esta interfaz de usuario, correspondiente a la vista *Interventions*, se muestra una tabla con las intervenciones realizadas en una reunión concreta, como se observa en la Figura 5.10. A su vez, se muestra información acerca del mecanismo de validación utilizado en la reunión, ya sea por usuario validador o por votación, y el identificador de la reunión.

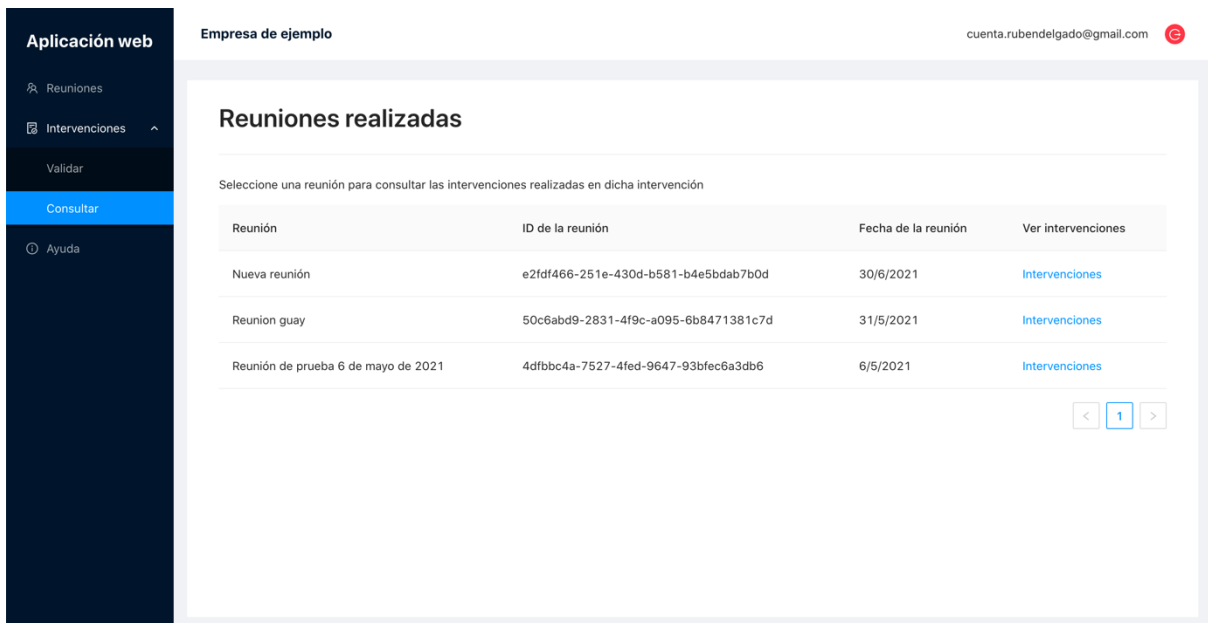


Figura 5.9. Interfaz de reuniones realizadas

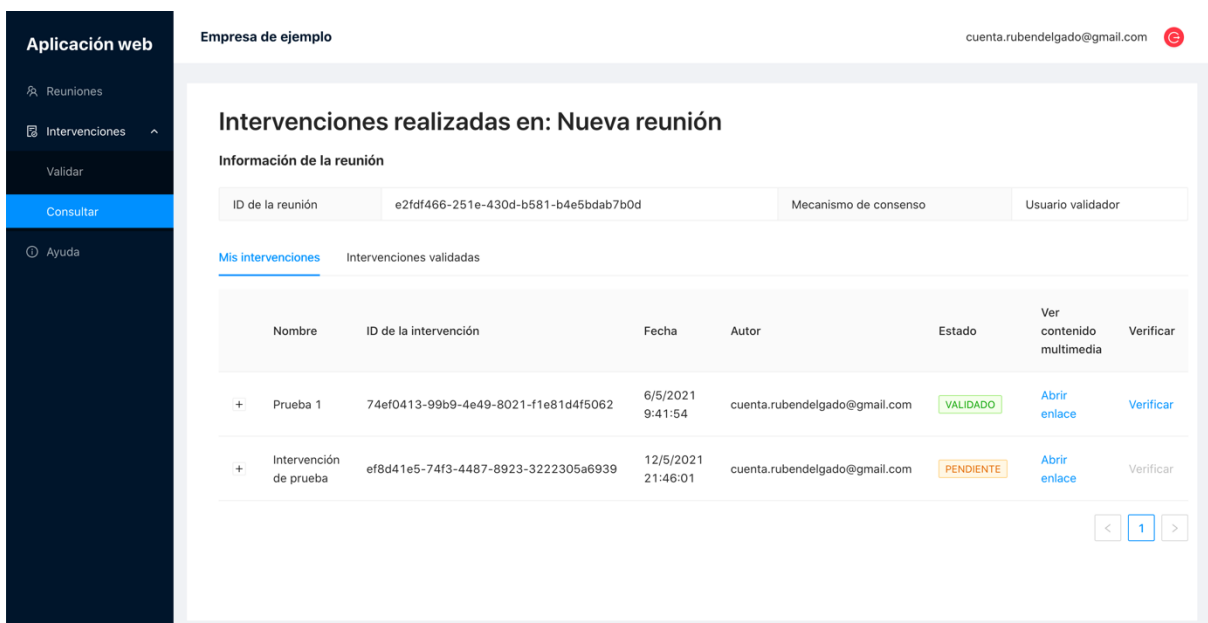


Figura 5.10. Interfaz de usuario de las intervenciones realizadas

Dentro de esta pantalla, se pueden consultar las intervenciones realizadas por el usuario (donde se mostrarían todas las intervenciones que ha realizado en la reunión, sin importar si aún está pendiente de validación o si ha sido rechazada o validada) y todas las intervenciones realizadas en la reunión que hayan sido validadas.

Cada entrada de la tabla correspondiente a una intervención muestra el nombre y el identificador de la intervención, la fecha en la que se ha realizado la intervención, el usuario

que la ha realizado y, por último, su estado de validación. El estado de validación puede ser:

- **PENDIENTE:** aún no se ha aceptado o rechazado, ya que quedan usuarios validadores que no han validado la intervención todavía.
- **ACEPTADO:** la intervención ha sido aceptada por los usuarios validadores, por lo que se ha añadido a la *blockchain* y se puede verificar la integridad de su contenido.
- **RECHAZADO:** la intervención ha sido rechazada por los usuarios validadores.

En caso de que un usuario que pertenezca a la reunión, conociendo el ID de la reunión quisiera acceder a dicha vista a través del URL, se mostraría un mensaje de error describiendo que la reunión consultada no existe (Figura 5.11).

## InterventionVerification

En esta interfaz de usuario, correspondiente a la vista *InterventionVerification*, se muestra el proceso de verificación de la integridad de la intervención con la *blockchain*, así como un enlace de descarga del audio de la intervención (Figura 5.12). En esta, se muestran los diferentes pasos que se realizan para comprobar la autenticidad de la intervención que se está consultando.

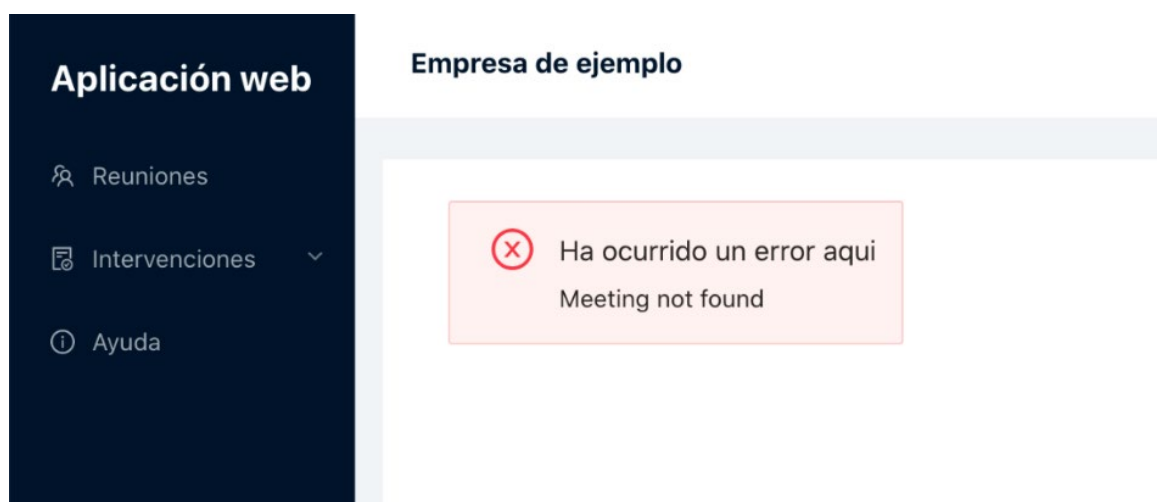


Figura 5.11. Mensaje de error al acceder a las intervenciones de una reunión en la que el usuario no es participante.



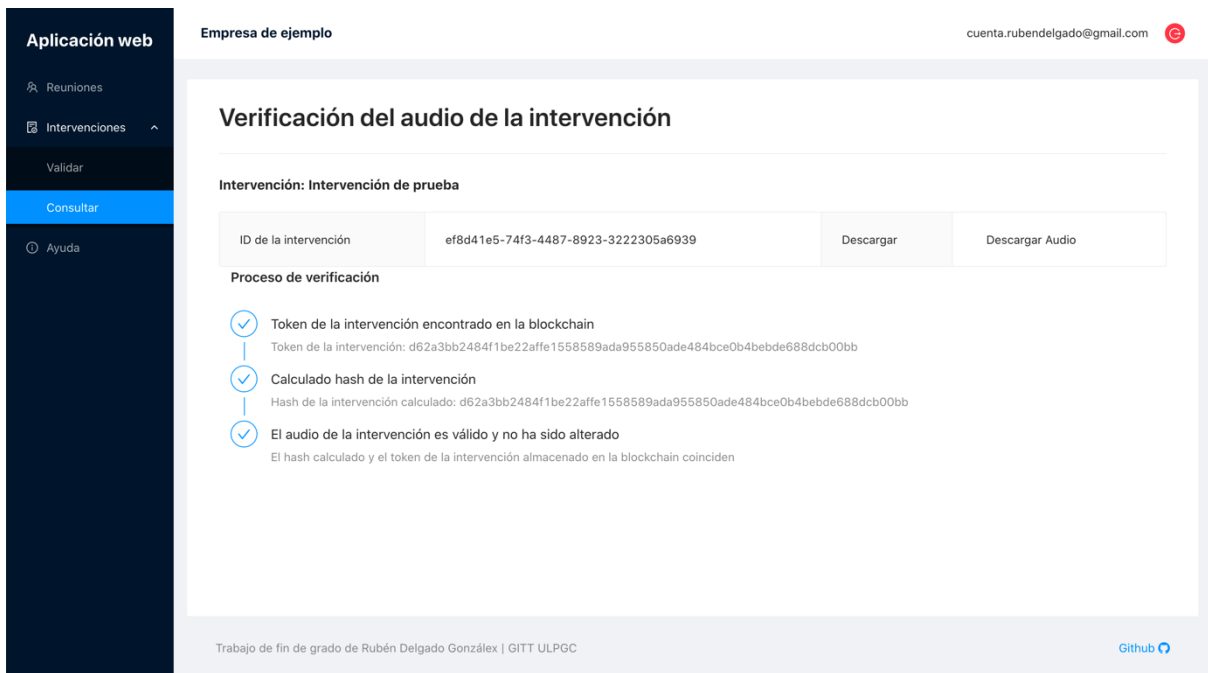


Figura 5.12. Interfaz de usuario de verificación de intervención

En caso de que un usuario consiguiera el enlace para acceder al proceso de verificación de una intervención en la que no ha sido invitado en la reunión en donde se ha realizado dicha intervención, se mostraría un mensaje de error, como se muestra en la Figura 5.13.

## Help

En esta vista se describe brevemente el funcionamiento de la aplicación (Figura 5.14). Además, para complementar la explicación del uso de la aplicación y las diferentes opciones que ofrece, se muestra un vídeo [30] en el que se observa visualmente cómo funciona la aplicación, así como la interacción con la aplicación.

## Verificación del audio de la intervención



Figura 5.13. Mensaje de error en proceso de verificación de intervención

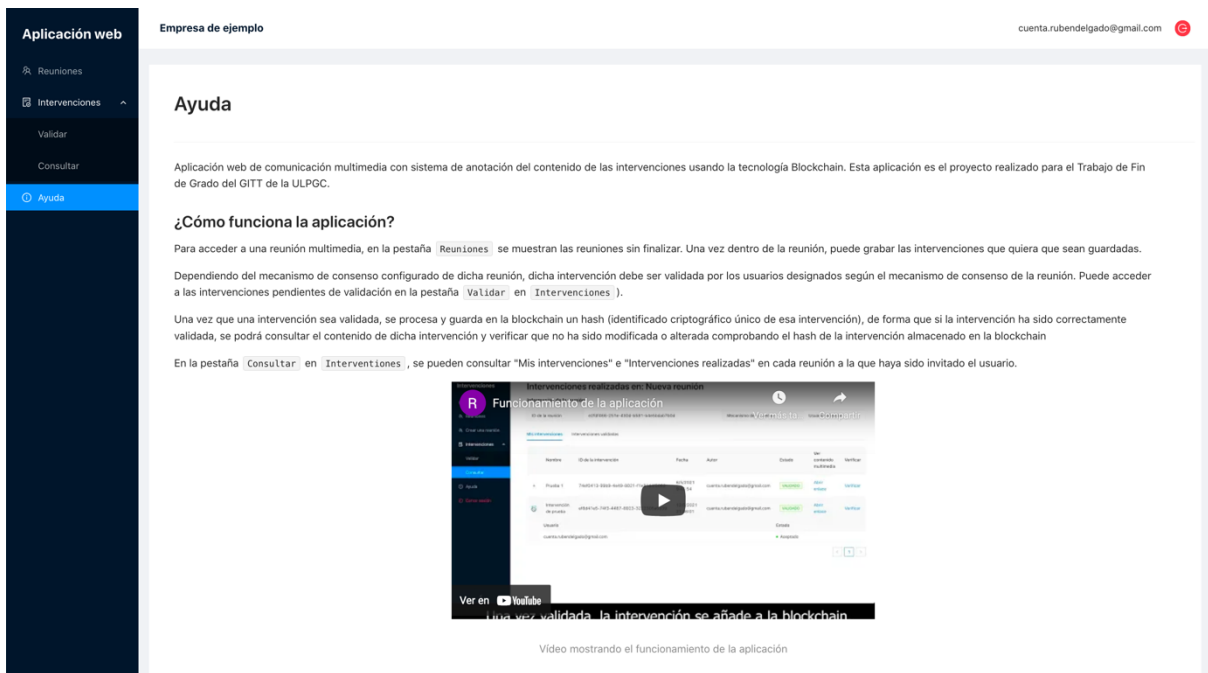


Figura 5.14. Interfaz de usuario de ayuda

## CreationMeeting

En esta interfaz de usuario, la cual corresponde con la vista *CreationMeeting*, se muestra un formulario para la creación de una reunión (Figura 5.15). En esta vista, a la cual solo puede acceder un usuario administrador, se realiza la configuración de los parámetros de una reunión.

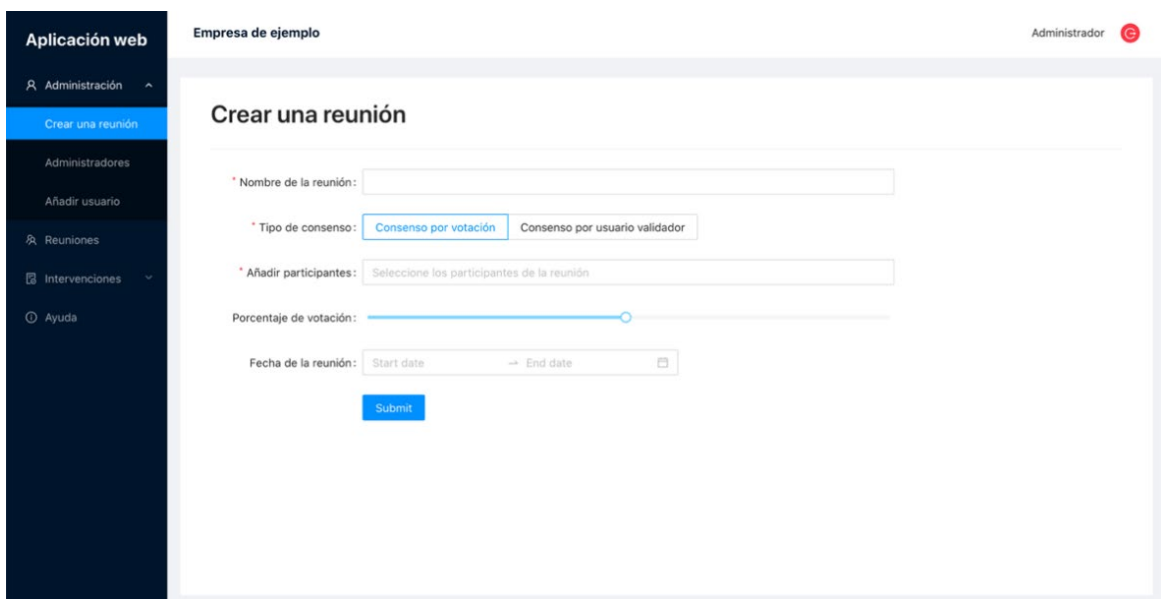


Figura 5.15. Interfaz de usuario para la creación de una reunión

## Admins

En esta interfaz de usuario, a la cual le corresponde la vista *Admins*, se muestra una tabla con los diferentes usuarios administradores. En dicha tabla, cada entrada corresponde con un usuario con el rol de administración, donde se muestra el correo electrónico de cada uno de ellos.

Además, en esta vista, a la cual solo puede acceder un usuario administrador, se puede quitar el rol de administrador de un usuario determinado mediante el botón *Eliminar*, tal y como se muestra en la Figura 5.16. Asimismo, también se pueden dotar a un usuario del rol de administrador introduciendo en el campo de texto superior a la tabla el correo electrónico del usuario.

## AddUser

Por último, en esta interfaz de usuario, la cual corresponde con la vista *AddUser* (Figura 5.17), se muestra un formulario para dotar a un usuario de acceso a la aplicación web. Para ello, dicho formulario contiene dos campos de texto donde el usuario administrador, el cual es el único que tiene acceso a esta vista, debe introducir el nombre del usuario y su correo electrónico.

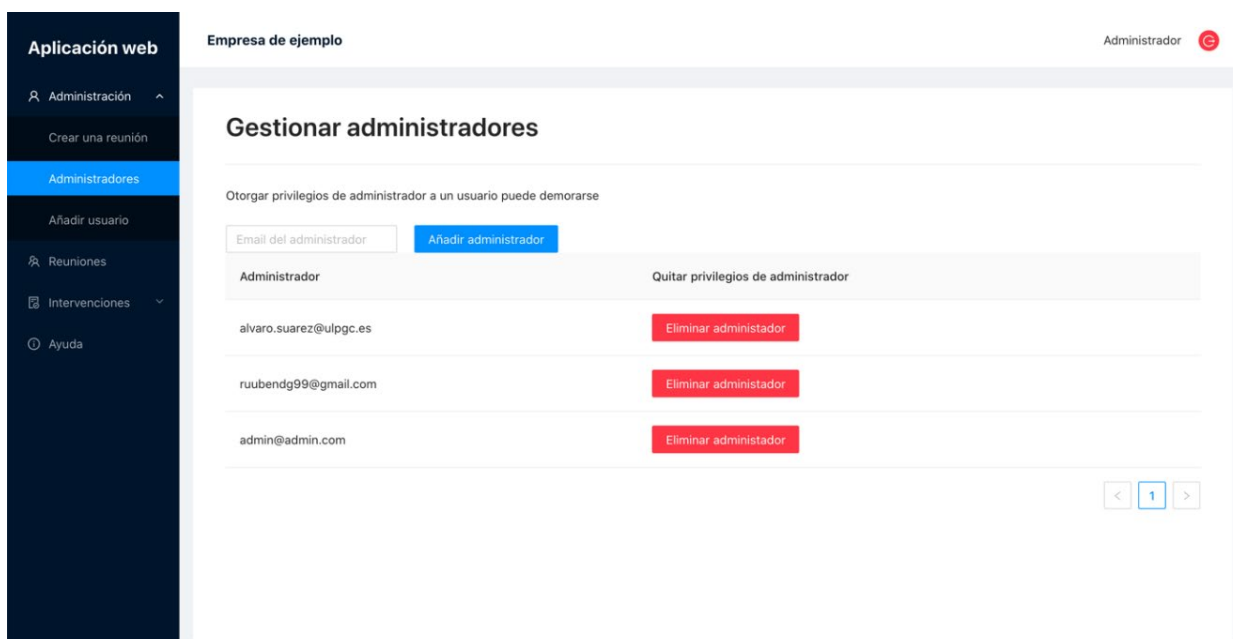


Figura 5.16. Interfaz de usuario de gestión de usuarios administradores

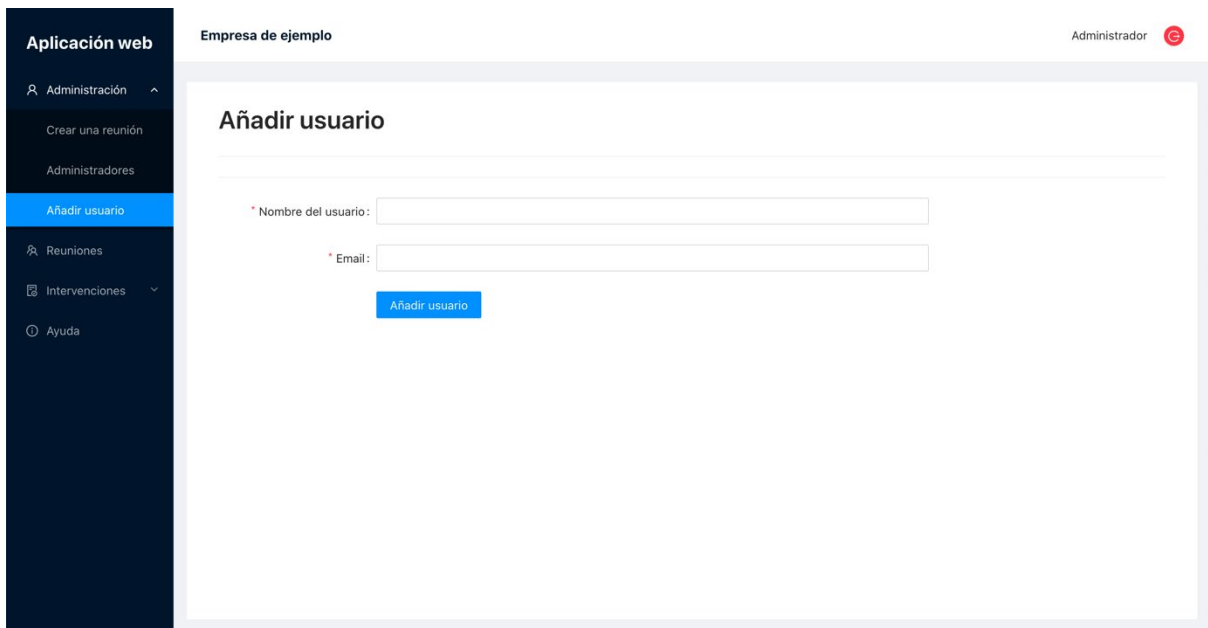


Figura 5.17. Interfaz de usuario para añadir nuevos usuarios

Haciendo *click* en el botón *Añadir Usuario*, como se muestra en la Figura 5.17, crearía un nuevo usuario con las credenciales introducidas y se enviaría un correo electrónico a dicho usuario con la contraseña de acceso a la aplicación web.

### 5.1.1 Ejemplos de uso

En este apartado se explican los pasos que debe seguir el usuario para poder realizar las principales acciones que ofrece la aplicación web multimedia: *Crear de una reunión*, *Acceder a una reunión multimedia*, *Realizar una intervención* y *Consultar y validar intervenciones*.

#### Crear una reunión multimedia

Para crear una reunión multimedia, el usuario administrador debe dirigirse a la vista *CreationMeeting* a través de la opción *Crear una reunión* en el Menú de *Administración*. En esta vista (Figura 5.18) se muestra un formulario con los siguientes campos que debería rellenar el usuario administrador:

- *Nombre de la reunión*: nombre para describir brevemente cual es el asunto de la reunión.

- *Añadir participantes*: campo de entrada múltiple donde se introduce el correo electrónico de los usuarios que participarían y serían invitados a la reunión.
- *Tipo de consenso*: el mecanismo de validación utilizado para designar quienes deben ser los usuarios validadores de las intervenciones realizadas en la reunión. Dependiendo de la opción escogida, se deben rellenar unos campos determinados del formulario.
  - *Consenso por votación*: se mostraría una slider, como se muestra en la Figura 5.19, donde el usuario administrador debe elegir el porcentaje de validaciones necesarias para que una intervención sea aceptada. Dependiendo del número de participantes de la reunión y el porcentaje especificado, se muestra una línea de texto con el número de validaciones necesarias.
  - *Consenso por usuario validador*: se mostraría un campo de selección única (Figura 5.20), donde el usuario administrador debe introducir el correo electrónico del usuario que designaría como único usuario encargado de realizar el proceso de validación de las intervenciones de la reunión.
- *Fecha de la reunión*: por último, se muestran dos campos de selección de fechas, en las que el usuario administrador debe seleccionar el momento de inicio y final de la reunión.

## Crear una reunión

\* Nombre de la reunión:

\* Tipo de consenso:  Consenso por votación  Consenso por usuario validador

\* Añadir participantes:

Porcentaje de votación:

Fecha de la reunión:  →

Figura 5.18. Formulario de creación de una reunión

\* Tipo de consenso:  Consenso por votación  Consenso por usuario validador

\* Añadir participantes: rubendg99@gmail.com × prueba@prueba.com × alvaro.suarez@ulpgc.es ×  
 cuenta.rubendelgado@gmail.com × prueba2@prueba.com ×

Porcentaje de votación:    
 Con un porcentaje del 75, es necesaria la validación de 4 usuarios

*Figura 5.19. Campos del formulario con consenso por votación*

\* Tipo de consenso:  Consenso por votación  Consenso por usuario validador

\* Añadir participantes: prueba@prueba.com × cuenta.rubendelgado@gmail.com ×

\* Usuario validador: cuenta.rubendelgado@gmail.com

*Figura 5.20. Campos del formulario con consenso por usuario validador*

## Acceder a una reunión multimedia

Un usuario puede acceder de dos formas a una reunión a la que haya sido invitado. La primera de ellas es a través de un enlace directo a la reunión. Cada vez que un usuario administrador crea una reunión, el sistema envía un correo electrónico a los usuarios que han sido invitados a la reunión en el cual se adjunta el enlace de acceso a la reunión (Figura 5.21).

Accediendo a dicho enlace de acceso, el usuario entraría directamente en la reunión. En caso de no haber iniciado sesión, se mostraría el formulario de inicio de sesión y posteriormente accedería a la reunión. En caso de que un agente externo obtuviera el enlace de acceso, al entrar en la reunión se mostraría un mensaje de error y no podría entrar en la reunión multimedia.

La otra manera de acceder a una reunión es a través de la vista *NextMeetings*, a la cual se accede desde la opción *Reuniones* en el menú. Esta vista muestra la lista de reuniones, y un enlace de acceso en cada una de ellas para acceder, tal y como se muestra en la Figura 5.22.

Ha sido invitado a la reunión: Nueva reunión


 video.intervenciones.tfg@gmail.com  
para mí ▾  
<https://bc-videochat.herokuapp.com/meetings/e2fdf466-251e-430d-b581-b4e5bdab7b0d>

Figura 5.21. Ejemplo de correo electrónico de invitación a una reunión

## Próximas reuniones

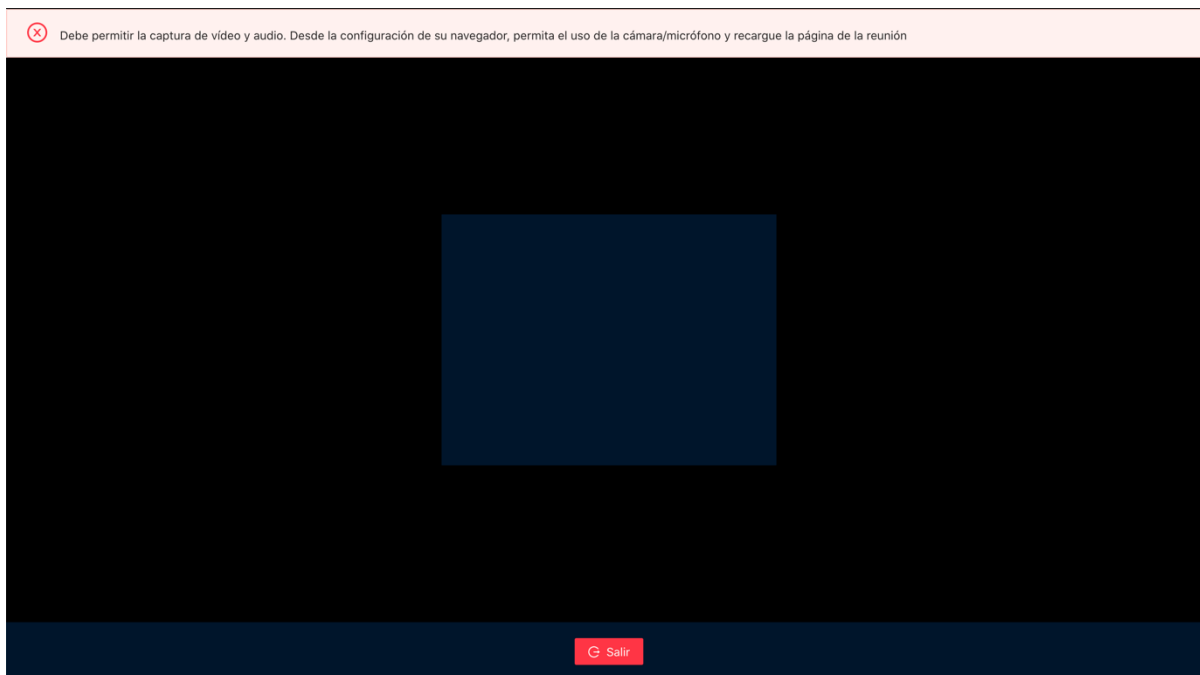
Reunion	Fecha de inicio	Fecha de finalización	Consenso	Entrar
Nueva reunión	6/5/2021 9:40:59	30/6/2021 9:41:03	Usuario validador	<a href="#">Unirse a reunión</a>

Figura 5.22. Tabla con las próximas reuniones

Una vez se accede a la reunión, el navegador solicita el acceso a los dispositivos multimedia del usuario (cámara y micrófono) para realizar la comunicación multimedia con otros usuarios (Figura 5.23). La interfaz de la solicitud varía dependiendo del navegador web utilizado. En caso de denegar el acceso a los dispositivos solicitados, se muestra un mensaje de error (Figura 5.24) instando al usuario a que debe permitir el acceso a dichos dispositivos para poder acceder a la sala multimedia. Para ello, debe volver a cargar la página y aceptar la solicitud.



Figura 5.23. Solicitud de acceso a la cámara y el micrófono en Firefox.



*Figura 5.24. Interfaz de la sala multimedia si no se permite el acceso a la cámara y el micrófono.*

Finalmente, una vez se permite la utilización de la cámara y el micrófono, el usuario ya puede acceder a la reunión y comunicarse con los usuarios que se encuentren en la misma.

### **Realizar una intervención**

Dentro de una reunión, los usuarios participantes pueden guardar las intervenciones que realicen para que queden almacenadas en el sistema. Para ello, deben hacer uso del botón *Grabar intervención* que se muestra en la parte inferior. Posteriormente, el usuario tiene 2 minutos para realizar la intervención, mostrando el tiempo restante de grabación como se muestra en la Figura 5.25. De igual manera, el usuario puede detener la grabación de la intervención antes utilizando el botón *Terminar intervención*. Si ha terminado la cuenta atrás de 2 minutos y el usuario no ha terminado la grabación, esta se detiene de forma automática pasado dicho periodo de tiempo.

Una vez terminada la grabación, se muestra un modal que contiene una entrada de texto para que el usuario introduzca un nombre que describa brevemente el contenido de la intervención realizada. Además, para escuchar el contenido multimedia de la misma, el modal tiene un reproductor de audio para que el usuario pueda comprobar que se ha grabado correctamente el contenido de la intervención (Figura 5.26).



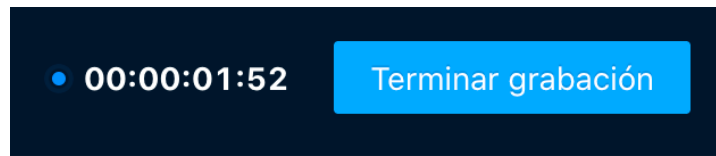


Figura 5.25. Controles de grabación de intervención

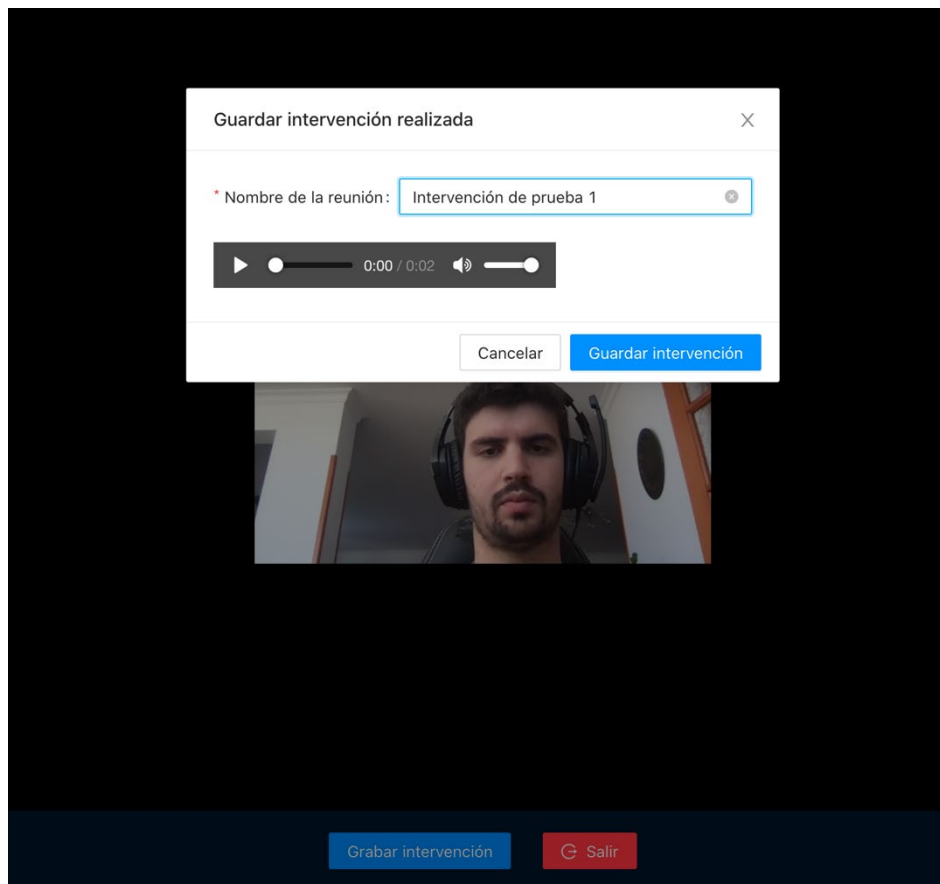


Figura 5.26. Modal para guardar la intervención

Pulsando el botón *Guardar intervención*, se registraría la intervención para que pueda ser validada por los usuarios validadores de la reunión, y añadida posteriormente a la *blockchain* si ha sido aceptada. En caso de pulsar el botón *Cancelar*, se desecharía la intervención realizada.

### Consultar y validar intervenciones

Una vez que se ha realizado una intervención en una reunión multimedia, esta debe ser validada por los usuarios designados para ello según la configuración de la reunión. Dichos usuarios pueden consultar las intervenciones pendientes de ser validadas desde la vista

*PendingInterventions* (Figura 5.27), a la cual se puede acceder desde el Menú de intervenciones en la opción *Validar*.

En primer lugar, el usuario selecciona aquellas intervenciones que quiera validar. Para previsualizar el contenido multimedia de dichas intervenciones antes de validarlas, el usuario puede hacer *click* en *Abrir enlace* de la intervención que quiera consultar. Esta opción abre en una nueva pestaña en la que se puede ver el contenido multimedia en aquellos navegadores compatibles con el formato *WebM* [31]. Posteriormente, el usuario validaría las intervenciones seleccionadas haciendo uso de los botones *Aceptar Intervención* o *Rechazar intervención* que se muestran en la Figura 5.27.

Además de consultar las intervenciones pendientes, todos los usuarios pueden consultar las intervenciones que hayan realizado o hayan sido validadas en las reuniones en las que han participado. Para ello, desde la vista *Meetings*, a la cual se puede acceder desde el Menú de intervenciones en la opción *consultar*, el usuario debe seleccionar aquella reunión de la cual quiere consultar las intervenciones realizadas (Figura 5.28).

Posteriormente, se muestra la vista *Interventions*, la cual muestra las intervenciones realizadas en la reunión seleccionada. En esta vista, el usuario puede consultar las intervenciones que haya realizado (en la pestaña *Mis Intervenciones*) donde se mostrarían todas las intervenciones que haya realizado, sin importar si aún no han sido validadas o si han sido aceptadas o rechazadas (Figura 5.29).

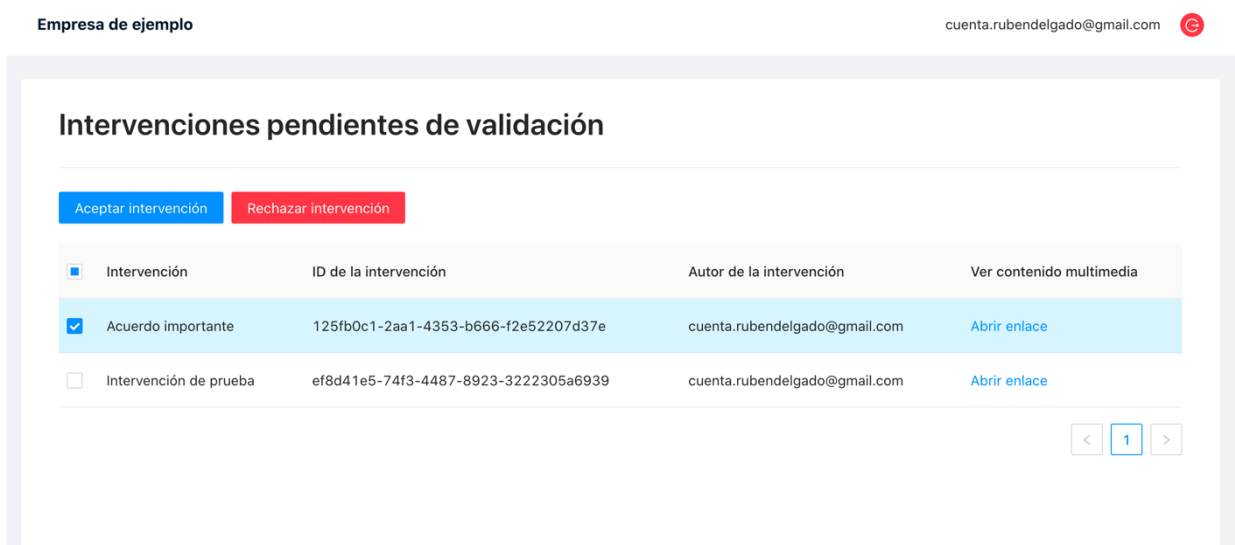


Figura 5.27. Proceso de validación de una intervención

## Reuniones realizadas

Seleccione una reunión para consultar las intervenciones realizadas en dicha intervención

Reunión	ID de la reunión	Fecha de la reunión	Ver intervenciones
Nueva reunión	e2fdf466-251e-430d-b581-b4e5bdab7b0d	30/6/2021	<a href="#">Intervenciones</a>
Reunion guay	50c6abd9-2831-4f9c-a095-6b8471381c7d	31/5/2021	<a href="#">Intervenciones</a>
Reunión de prueba 6 de mayo de 2021	4dfbbc4a-7527-4fed-9647-93bfec6a3db6	6/5/2021	<a href="#">Intervenciones</a>

< 1 >

Figura 5.28. Tabla con las reuniones a las que el usuario haya sido invitado

[Mis intervenciones](#) Intervenciones validadas

	Nombre	ID de la intervención	Fecha	Autor	Estado	Ver contenido multimedia	Verificar
+	Prueba 1	74ef0413-99b9-4e49-8021-f1e81d4f5062	6/5/2021 9:41:54	cuenta.rubendelgado@gmail.com	VALIDADO	<a href="#">Abrir enlace</a>	<a href="#">Verificar</a>
+	Intervención importante	8b73c48a-0061-41fb-99f2-97cb073c7eed	2/6/2021 16:03:15	cuenta.rubendelgado@gmail.com	VALIDADO	<a href="#">Abrir enlace</a>	<a href="#">Verificar</a>
+	Intervención de prueba	ef8d41e5-74f3-4487-8923-3222305a6939	12/5/2021 21:46:01	cuenta.rubendelgado@gmail.com	PENDIENTE	<a href="#">Abrir enlace</a>	<a href="#">Verificar</a>
+	Prueba 2	f21f888c-9c82-4fbb-9ad2-623edbe3fbeb	2/6/2021 18:13:12	cuenta.rubendelgado@gmail.com	DENEGADO	<a href="#">Abrir enlace</a>	<a href="#">Verificar</a>

Figura 5.29. Tabla con las intervenciones realizadas por el usuario

El usuario también puede consultar quién ha validado las intervenciones que se muestran en la tabla. Para ello, a la izquierda de cada entrada de la tabla, hay un desplegable que muestra una tabla anidada con los diferentes usuarios validadores y el estado de validación (Figura 5.30), el cual puede ser:

- *Aceptado*: el usuario validador ha aceptado la intervención.
- *Rechazado*: el usuario validador ha rechazado la intervención.
- *Pendiente*: el usuario validador aún no ha validado la intervención.
- *Aceptado por desistimiento positivo*: el periodo en el que el usuario validador puede validar la intervención ha expirado, por lo que, aplicando desistimiento positivo, la intervención ha sido aceptada automáticamente.

Nombre	ID de la intervención	Fecha	Autor	Estado	Ver contenido multimedia	Verificar
Prueba 1	74ef0413-99b9-4e49-8021-f1e81d4f5062	6/5/2021 9:41:54	cuenta.rubendelgado@gmail.com	VALIDADO	<a href="#">Abrir enlace</a>	<a href="#">Verificar</a>

Usuario	Estado
cuenta.rubendelgado@gmail.com	● Aceptado

*Figura 5.30. Tabla anidada con los usuarios validadores y estado de validación*

Por último, el usuario puede verificar la integridad de aquellas intervenciones que hayan sido aceptadas. Las intervenciones que han sido finalmente aceptadas por los validadores siguiendo el mecanismo de validación configurado en la reunión, son añadidas a la cadena de bloques de la *blockchain*. Por ello, los usuarios pueden comprobar la autenticidad de las intervenciones validadas en la opción *Verificar*.

Haciendo *click* en dicha opción, se muestra la vista *InterventionVerification*, la cual muestra el proceso de verificación de una intervención con la *blockchain*. Dicho proceso se muestra en la Figura 5.31 en 3 pasos:

1. Se realiza una consulta al nodo raíz de la *blockchain* para buscar la transacción correspondiente a la intervención que se quiere verificar en la cadena de bloques. Si no se encuentra la transacción (porque no se ha minado un bloque con dicha transacción aún), se muestra un mensaje de error.
2. Se calcula el *hash* del contenido multimedia de la intervención en el servidor
3. Se comprueba si el *hash* calculado en el servidor y el *hash (token)* almacenado en la cadena de bloques son iguales. Si es así, se puede garantizar la autenticidad del contenido de la intervención. En caso contrario, no se puede verificar que el contenido de la intervención sea el mismo, ya que este puede haber sido alterado.

Si alguno de los pasos falla, se mostraría un mensaje de error al usuario. Por último, el usuario puede descargar el contenido multimedia de la intervención en el botón *Solicitar descarga*. Este generaría un enlace para descargar el contenido multimedia, el cual se puede descargar haciendo *click* en el hipervínculo *Descargar contenido* (Figura 5.32).

#### Intervención: Prueba 1

ID de la intervención	74ef0413-99b9-4e49-8021-f1e81d4f5062	Descargar	Descargar Audio
-----------------------	--------------------------------------	-----------	-----------------

Proceso de verificación

- ✓ Token de la intervención encontrado en la blockchain  
Token de la intervención: a233d3e500db80bc92dbab017e68bbba3a057179afaaaf653bd541f42f998e4c
- ✓ Calculado hash de la intervención  
Hash de la intervención calculado: a233d3e500db80bc92dbab017e68bbba3a057179afaaaf653bd541f42f998e4c
- ✓ El audio de la intervención es válido y no ha sido alterado  
El hash calculado y el token de la intervención almacenado en la blockchain coinciden

*Figura 5.31. Proceso de verificación de la intervención*

Descargar	Descargar contenido
-----------	---------------------

*Figura 6.32. Descarga del contenido multimedia de la intervención*

### 5.1.2. Interfaz de la aplicación web en dispositivos móviles

Para que la aplicación se pudiera utilizar en cualquier tipo de terminal, se desarrolló una interfaz responsiva. Esto es, que la interfaz de usuario se adapta a la resolución de la pantalla cambiando el estilo de algunos elementos de la interfaz para que esta se puede ver correctamente en cualquier tipo de dispositivo, ya sean ordenadores de sobremesa o terminales móviles.

Para poder lograr esto, en primer lugar, el menú lateral se puede esconder mediante el botón que se encuentra en la esquina superior izquierda. Esto permite que, en dispositivos móviles, dicho panel de navegación lateral pueda ocultarse para que no ocupe espacio en la pantalla si no se está utilizando.

En la Figura 5.33 se muestra como dicho menú de navegación permanece oculto en pantallas de dispositivos móviles y, tras pulsar el botón para desplegarlo indicado en la figura, se muestra el menú de navegación.

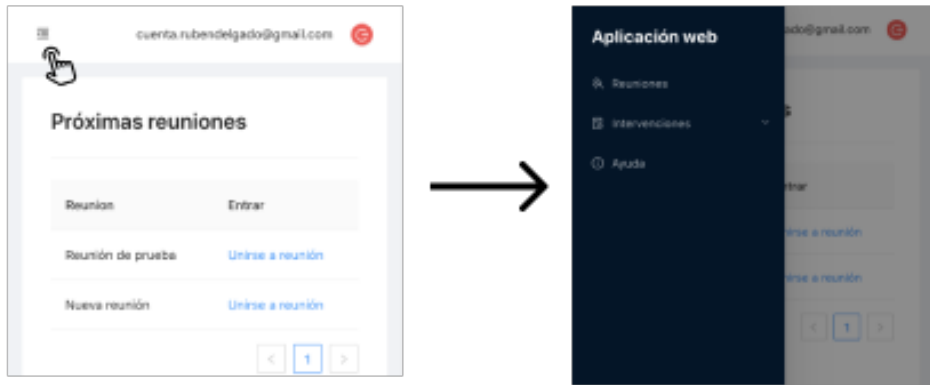


Figura 5.33. Despliegue de menú lateral

En la Figura 5.34 se muestran las principales vistas de la aplicación web en la resolución de un dispositivo móvil, en el que la interfaz se adapta a dicha resolución, ocultando elementos de las tablas menos importantes para que se pueda visualizar la información correctamente.

## 5.2. Interfaz de usuario de administración de la blockchain: Nodo raíz

Para la administración de la *blockchain*, el nodo raíz ofrece una interfaz web para que los usuarios administradores puedan consultar fácilmente el contenido de la blockchain, así como gestionar los nodos validadores. Para su utilización, el nodo raíz se encuentra publicado en *Heroku* [32]. A continuación, se muestran las distintas pantallas de que ofrece el nodo raíz para la administración de la *blockchain*.

### Login

La primera vista que verá el usuario al acceder a la aplicación web por primera vez se muestra en la Figura 5.35. En ella, se muestra un formulario de inicio sesión para que el usuario administrador introduzca sus credenciales y acceda a la aplicación. En caso de que inicie sesión un usuario sin el rol de administrador, se muestra un mensaje de error (Figura 5.36), ya que solo los usuarios con el rol de administrador pueden realizar gestiones sobre la administración de la *blockchain*.

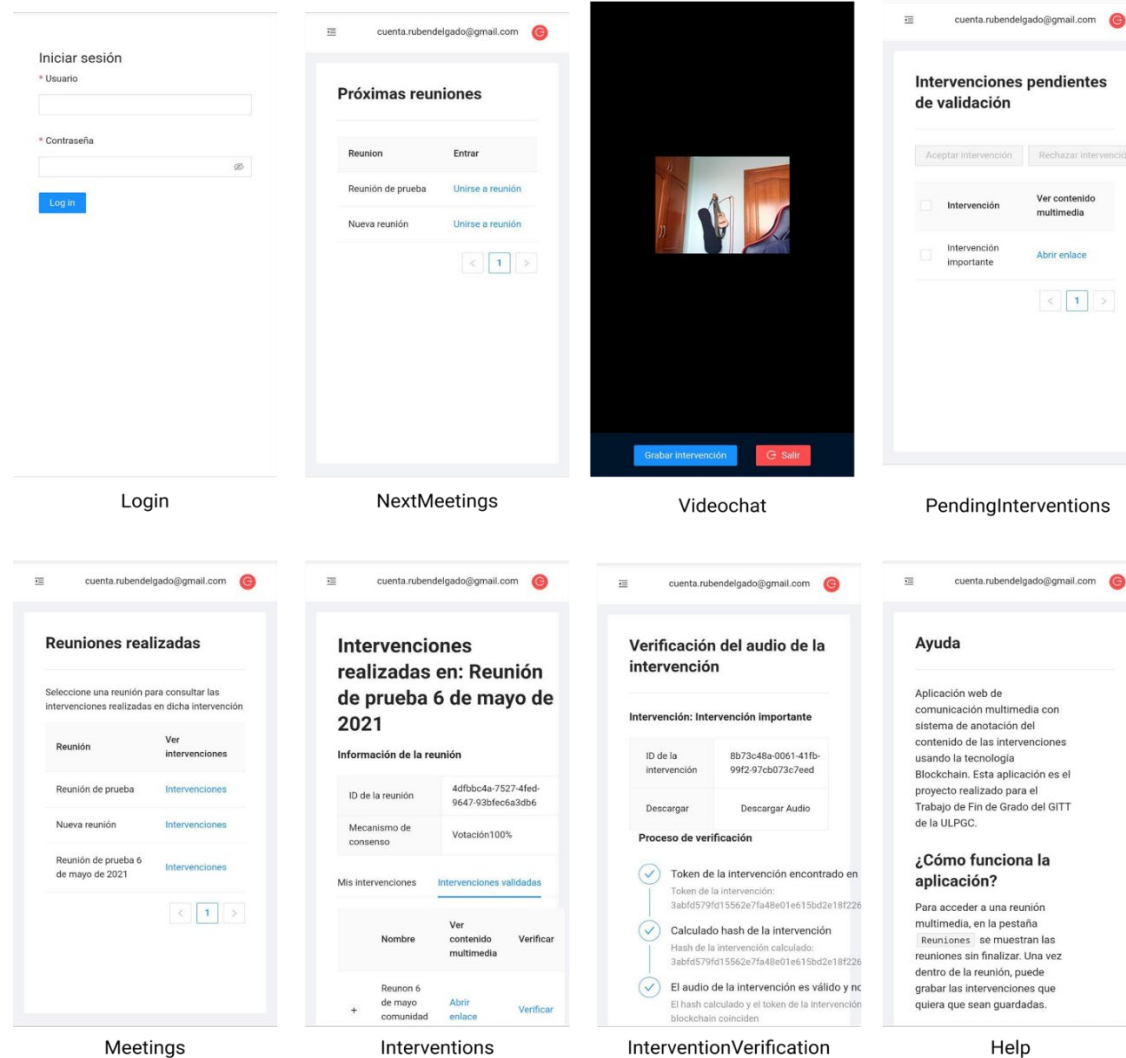


Figura 5.34. Principales vistas de la aplicación en un dispositivo móvil

## Panel de administración de la blockchain

\* Usuario:

\* Contraseña:

Iniciar sesión

Figura 5.35. Interfaz de usuario de inicio de sesión del panel de administración de la blockchain



Solo pueden acceder usuarios con el rol de administrador

Salir

Figura 5.36. Mensaje de error en inicio de sesión

Una vez que un usuario administrador inicia sesión correctamente, se muestra la interfaz de usuario de la aplicación web para administrar la *blockchain*. A la izquierda, se muestra un menú con las diferentes vistas que ofrece dicha aplicación, tal y como se muestra en la Figura 5.37.

Las opciones a las que se puede navegar desde el menú son:

- *Administrar nodos*: vista desde la que se pueden ver los nodos validadores.
- *Añadir nodo*: vista desde la cual se pueden añadir nuevos nodos validadores a través de un formulario.
- *Cadena de bloques*: tabla que muestra el estado actual de la cadena de bloques.
- *Transacciones*: tabla que muestra el estado actual del pool de transacciones.



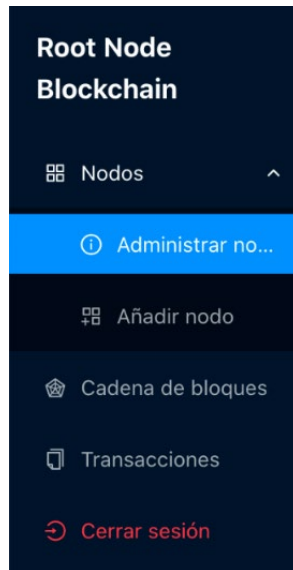


Figura 5.37. Menú del panel de administración de la blockchain

## Nodes

En esta interfaz de usuario, la cual corresponde con la vista *Nodes*, se muestra una tabla con los nodos validadores, donde se muestra el identificador de cada uno de los nodos validadores. Para eliminar un nodo entre los nodos validadores, como se muestra en la Figura 5.38, se ofrece un botón *Eliminar* para eliminar cualquiera de los nodos validadores.

## AddNode

En esta interfaz de usuario, la cual corresponde con la vista *AddNode*, se muestra un formulario con un único campo de texto para añadir un nodo a la lista de nodos validadores (Figura 5.39). En dicho campo de texto, el usuario administrador debe introducir el identificador del nodo que va a ser nodo validador.

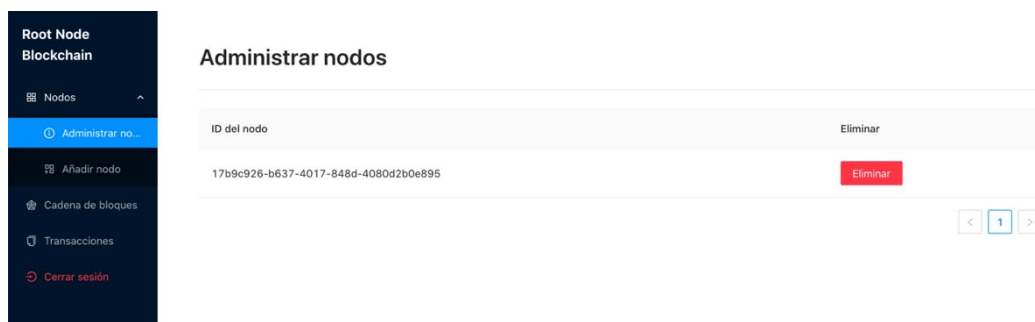


Figura 5.38. Interfaz de usuario para administrar nodos

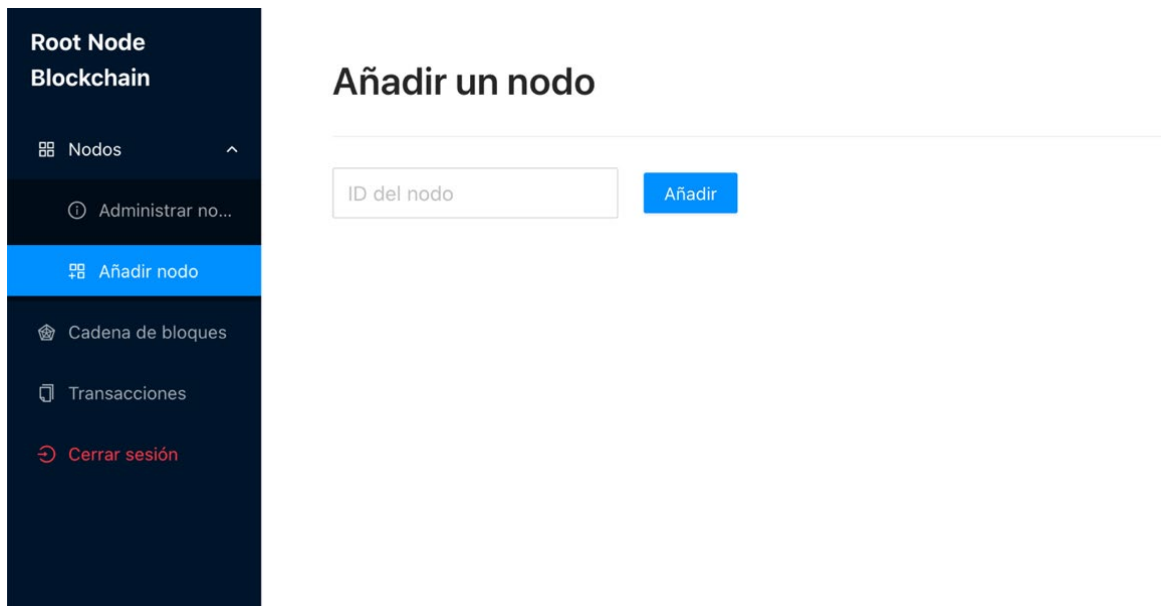


Figura 5.39. Interfaz de usuario para añadir un nodo validador

## Blockchain

En esta interfaz de usuario, la cual corresponde con la vista *Blockchain*, se muestra una tabla con el contenido actual de la cadena de bloques. Cada una de las entradas de dicha tabla corresponde con un bloque de la cadena de bloques, en el que se muestra el *hash* del bloque, el instante de tiempo en el cual se minó el bloque, la dificultad de minado, el valor *nonce* y el *hash* del bloque anterior, tal y como se muestra en la Figura 5.40.

A su vez, mediante una tabla anidada en cada una de las entradas para cada bloque, se muestran las transacciones que almacenan cada uno de ellos (Figura 5.41).

## TransactionPool

Por último, en esta última interfaz, la cual corresponde con la vista *TransactionPool*, se muestra una tabla con las transacciones pendientes de ser añadidas a la cadena de bloques (Figura 5.42). En ella, cada entrada de la tabla corresponde con una transacción, donde se muestra el ID de la transacción (que corresponde con el ID de la intervención), los primeros caracteres del *token* de la transacción y la fecha en la que se ha creado.

Bloque	Hash del bloque	Marca de tiempo	Dificultad de minado	Nonce	Hash del bloque anterior	
0	first-hash	19/1/1970 18:34:39	10	0	----	
+	1	003d9ce6ace91cd7d63d...	2/6/2021 20:47:52	10	1926	first-hash
+	2	003dd22a1e8f50301043...	2/6/2021 20:48:38	10	500	003d9ce6ace91cd7d63d...

Figura 5.40. Interfaz de usuario del contenido de la cadena de bloques

+	2	003dd22a1e8f50301043...	2/6/2021 20:48:38	10	500	003d9ce6ace91cd7d63d...
ID de la transacción	Token de la intervención					
8b73c48a-0061-41fb-99f2-97cb073c7eed	d4726077887bf21e51184b1d6b8d547d4de96cd1bd9f98c5c41561c760ab0c94					

Figura 5.41. Ejemplo de tabla anidada de transacciones

ID de la transacción	Token	Fecha de realización
8b73c48a-0061-41fb-99f2-97cb073c7eed	d4726077887bf21e5118...	2/6/2021 20:54:21

Figura 5.42. Interfaz de usuario del contenido del pool de transacciones

## 5.3. Evaluación de resultados experimentales

En este apartado se comenta la viabilidad de la implementación de este proyecto, describiendo el rendimiento general obtenido en la *blockchain* implementada y la aplicación web de comunicación multimedia.

### 5.3.1. Rendimiento de la comunicación multimedia

En cuanto al sistema de comunicación multimedia desarrollado, tras realizar diferentes pruebas entre varios usuarios en diferentes redes, se puede concluir que funciona correctamente ya que permite la comunicación multimedia entre los clientes que se

encuentran en las salas multimedia. Partiendo del resultado positivo conseguido, a continuación, se analizan diferentes aspectos de red como la latencia o la tasa de paquetes perdidos, ya que las comunicaciones en tiempo real son muy sensibles a este tipo de variables de red.

Para analizar estos parámetros, los navegadores Chrome ofrecen una herramienta para los desarrolladores que permite monitorizar el estado de una comunicación multimedia de *WebRTC*, observando aspectos como los mensajes de señalización empleados o los códecs utilizados en el tráfico multimedia. Para hacer uso de esta herramienta, se debe entrar en la URL *chrome://webrtc-internals* durante la sesión de una comunicación multimedia desde el cual se puede descargar toda la información recogida en un archivo de texto.

Para visualizar mejor los aspectos de red de la conexión, es posible descargar este archivo y usar herramientas que permiten analizar de forma más cómoda a través de gráficas dichos datos. En este caso, se utiliza la herramienta *testRTC* [33], que permite monitorizar y testear la calidad de la comunicación multimedia establecida mediante WebRTC con una interfaz más práctica que la ofrecida en *chrome://webrtc-internals*.

Usando las herramientas descritas, los resultados analizados se han obtenido en una prueba de comunicación entre dos clientes: uno desde la isla de Gran Canaria y otro desde la isla de Tenerife. Ambos se encontraban conectados en su propia red inalámbrica privada. Los códecs utilizados de los flujos multimedia han sido: *Opus* [34] para el audio y *VP8* [35] para el vídeo. En primer lugar, el *throughput* (tasa efectiva de transferencia de datos utilizada) se muestra en la Tabla 5.1, diferenciando el *throughput* del enlace ascendente y descendente. El ancho de banda necesario para la transferencia de vídeo es mucho mayor que para la transferencia de audio.

	Audio	Vídeo
Canal de subida	64 Kbps	4,182 Mbps
Canal de bajada	56 Kbps	4,23 Mbps

Tabla 5.1. Throughput del tráfico multimedia

Por otro lado, la tasa de pérdida de paquetes es un parámetro importante en la comunicación multimedia, ya que, si esta tasa es elevada, esto puede empeorar la *Quality of Experience (QoE)*. En el resultado obtenido (Tabla 5.2), la tasa de pérdida de paquetes se mantiene por debajo del 1%, lo cual es un margen razonable para la transmisión de flujos multimedia [36].

	Audio	Vídeo
Canal ascendente	0.135 %	0.12 %
Canal descendente	0.067 %	0.058 %

*Tabla 5.2. Tasa de pérdida de paquetes*

Sin embargo, existen instantes de tiempo donde la pérdida de paquetes es significativa. No obstante, esto puede deberse a situaciones adversas en la red, como episodios de congestión.

El *jitter* es otro parámetro importante en transmisiones de datos en tiempo real. Este se define como la fluctuación o variación del tiempo de retraso entre paquetes. Su valor debería ser menor a 30 ms [36], ya que un porcentaje mayor podría empeorar la QoE en aplicaciones de comunicación en tiempo real como la desarrollada. En los resultados obtenidos en la Tabla 5.3, el valor medio del *jitter* es menor a 30 ms en los diferentes canales.

	Audio	Vídeo
Canal ascendente	2.9 ms	16.7 ms
Canal descendente	2.6 ms	2.9 ms

*Tabla 5.3. Valor medio del jitter*

Por último, la latencia es una variable crítica para asegurar una QoE correcta y una comunicación multimedia fluida entre los clientes. Esta puede definirse como el tiempo que necesita un paquete de datos para llegar desde el emisor al destino. La latencia se puede

medir mediante el *Round Trip Time (RTT)*, el cual es el tiempo que tarda un paquete en ser enviado más el tiempo en recibir un paquete de confirmación (*ACK*) de que se ha recibido el paquete enviado.

Para una comunicación multimedia con un QoE aceptable, el RTT debe ser menor a 300 ms [36], lo cual se cumple con los resultados obtenidos (Tabla 5.4).

	Audio	Vídeo
Canal ascendente	13.7 ms	15.5 ms

*Tabla 5.4. Valor medio del RTT*

Los resultados obtenidos manifiestan que la comunicación multimedia conseguida utilizando WebRTC es correcta y ofrece una QoE aceptable para los clientes. Esto refleja que se ha cumplido el objetivo de proporcionar una plataforma de comunicación multimedia.

Sin embargo, existe un caso de uso de la realización de reuniones multimedia que no es óptimo para la arquitectura utilizada para la transmisión del tráfico multimedia. En el modelo de comunicación multimedia utilizado con WebRTC en este sistema, se utilizan conexiones *Peer-to-Peer (P2P)*, es decir, el tráfico multimedia se transmite de forma directa entre los clientes involucrados en la comunicación multimedia.

El principal problema de usar una arquitectura P2P en WebRTC ocurre cuando se realiza una comunicación multimedia entre gran cantidad de clientes, ya que cada participante debe enviar su contenido multimedia a todos los demás participantes. Si suponemos que hay  $N$  participantes en la llamada, el mismo flujo multimedia debe enviarse  $N-1$  veces por un enlace ascendente a los  $N-1$  participantes. Esto requiere una cantidad significativa de ancho de banda de ascendente de los participantes, así como un coste computacional importante para cada dispositivo cliente, ya que debe codificar el mismo flujo varias veces para ser enviado, lo cual puede ser especialmente prohibitivo en dispositivos móviles o terminales con poca capacidad computacional.

En la Figura 5.43 se muestran los enlaces creados en una comunicación multimedia entre 4 clientes utilizando una arquitectura P2P, en la que se acaba formando una red mallada debido a los enlaces que hay entre cada uno de los participantes. En esta se observa cómo el cliente A tiene 3 enlaces de subida, lo cual ocuparía, teniendo en cuenta los resultados obtenidos en la Tabla 5.1, aproximadamente 16 Mbps para enviar su flujo multimedia al resto de los participantes.

Por ello, si se necesitase realizar comunicaciones multimedia entre muchos clientes, se podría utilizar un *Selective Forwarding Unit (SFU)* [37], el cual es un servidor utilizado para la retransmisión de flujos multimedia. Utilizando una arquitectura con un SFU, permite a los clientes enviar un único flujo multimedia al servidor (reduciendo el ancho de banda de subida necesitado y el coste computacional de codificación del flujo multimedia del usuario), y este lo retransmitiría al resto de clientes conectados al SFU.

Utilizando una arquitectura con un SFU, permite a los clientes enviar un único flujo multimedia al servidor (reduciendo el ancho de banda de subida necesitado y el coste computacional de codificación del flujo multimedia del usuario), y este lo retransmitiría al resto de clientes conectados al SFU. En la Figura 5.44 se muestra cómo, utilizando un SFU, cada cliente solo necesitaría enviar su tráfico multimedia a dicho dispositivo y este reenviaría cada uno de los flujos multimedia a los participantes de la reunión.

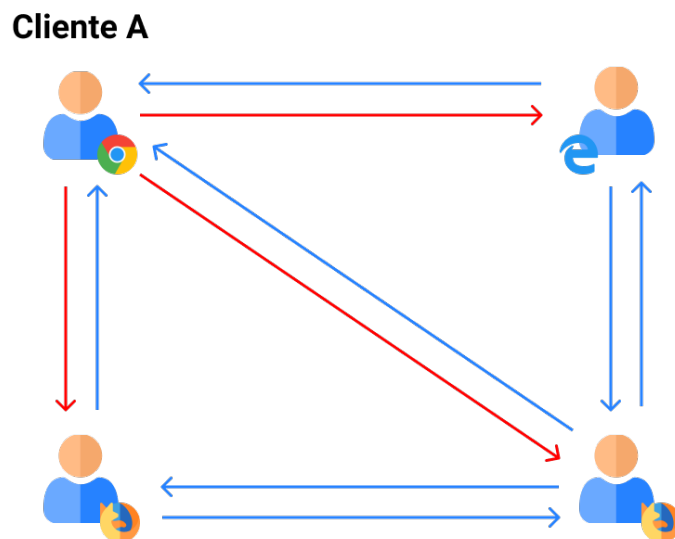


Figura 5.43. Comunicación multimedia utilizando arquitectura P2P

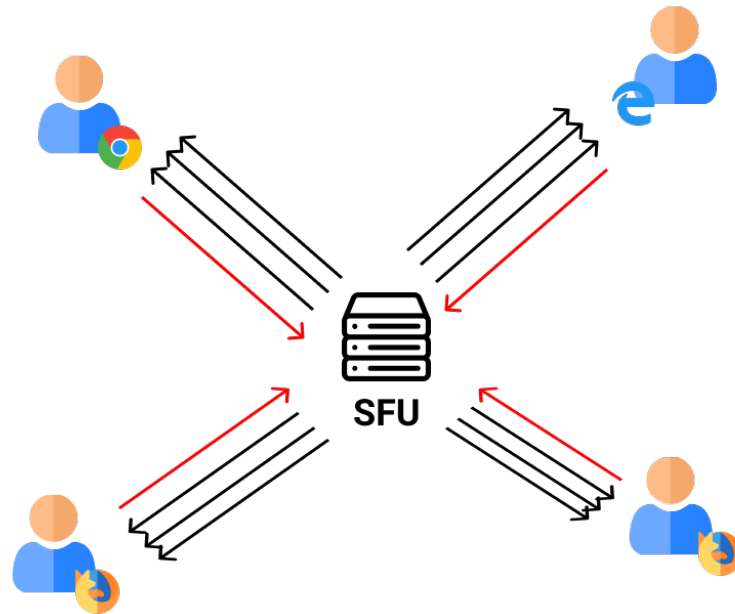


Figura 5.44. Comunicación multimedia utilizando SFU

Aunque con una arquitectura utilizando SFU se pierden las ventajas de establecer una conexión P2P entre los clientes, es una arquitectura más adecuada para el establecimiento de comunicación multimedia en reuniones telemáticas donde hay una cantidad elevada de participantes.

### 5.3.2. Resultados de la blockchain implementada

En cuanto a la *blockchain* implementada, se han analizado diferentes parámetros acerca de la generación de tokens y minado de bloques. También comprueba la seguridad de la red *blockchain* cuando un nodo intenta unirse a red *blockchain* realizando una petición al nodo raíz.

En primer lugar, en cuanto a la generación de *tokens*, se ha realizado una prueba en la que se ha calculado el tiempo medio de empleado para el cálculo del *hash* de una intervención de 2 minutos (tiempo máximo de grabación de una intervención). Generando el *hash* de 5 intervenciones, se ha obtenido un tiempo medio de 925 ms, lo cual es un valor razonable para que el proceso encargado de calcular el hash no llegue sobrecargarse.

En cuanto al minado en la *blockchain* implementada, el proceso de minado de bloques depende de la configuración de la dificultad de minado en los nodos, lo cual establece el



número de ceros consecutivos que debe haber a la izquierda del *hash* generado. A medida que se incrementa la dificultad, se incrementa exponencialmente el tiempo que se tarda en realizar el proceso de minado. Por este motivo, es importante encontrar una dificultad adecuada.

En la Tabla 5.5 se muestran los valores obtenidos tras realizar pruebas cambiando la dificultad de minado y utilizando un *script*, el cual se muestra en la Figura 5.45, que crea una cadena de bloques, y realiza el minado de 100 bloques de forma automática añadiendo los datos de una intervención de prueba, calculando el tiempo medio de minado entre cada bloque minado y mostrando el resultado en pantalla.

```
const Blockchain = require('./data/blockchain')

//Datos de ejemplo
const blockchain = new Blockchain();
const mockData = {
  id: 'b9899b3f-e294-404c-8e55-44f12b5b4a2b',
  videoAppID: '0x02f54ba86dc1ccb5bed0224d23f01ed87e4a443c47fc690d7797a13d41d234',
  signature: 'b04e95d0b09d1c3846dc2c1df871b44c780a47844534e36cfee5212f181660ae',
  interventionToken: 'b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7c'
}
blockchain.addBlock({ data: [mockData]})

let prevTimestamp, nextTimestamp, nextBlock, timeDiff, average;
const times = [];

for( let i=0; i<100; i++) {
  prevTimestamp = blockchain.chain[blockchain.chain.length-1].timestamp;

  //Realiza el minado de un nuevo bloque
  blockchain.addBlock({ data: [mockData]})

  //Calcula el tiempo de minado empleado
  nextBlock = blockchain.chain[blockchain.chain.length-1];
  nextTimestamp = nextBlock.timestamp;
  timeDiff = nextTimestamp - prevTimestamp;

  //Calcula el tiempo medio entre todos los tiempos de minado recogidos
  times.push(timeDiff);
  average = times.reduce((total, num) => (total + num))/times.length

  //Muestra el tiempo de minado de cada bloque
  console.log(`Time to mine block: ${timeDiff}ms. Average time: ${average}ms`)
}
```

Figura 5.45. Script para probar el mecanismo de minado de bloques

Dificultad de minado	Tiempo medio de 100 bloques minados
4	0,30 ms
8	1,97 ms
12	26,91 ms
16	356,54 ms
18	1,33 seg.
22	32,81 seg.

Tabla 5.5. Tiempo del algoritmo de minado según la dificultad

Los tiempos del proceso de minado obtenidos manifiestan la complejidad que puede alcanzar el algoritmo de minado empleado y el aumento del tiempo de minado a mayor de la dificultad empleada. Cuanto mayor sea la dificultad de minado, se puede asegurar con mayor seguridad la inmutabilidad de los datos almacenados, ya que requiere mayor capacidad computacional alterar la cadena de bloques por un agente externo. Sin embargo, cuanto mayor sea la dificultad, mayor gasto computacional (y, por ende, energético) empleado para realizar el minado de bloques y añadir datos a la *blockchain*.

Por otro lado, dado que la *blockchain* implementada es privada, se ha comprobado que solo aquellos nodos autorizados por el nodo raíz pueden acceder a la red *blockchain* y añadir bloques a la cadena de bloques. Cuando un nodo que no ha sido designado como nodo validador intenta acceder a la red *blockchain*, este recibe una respuesta de error con el código HTTP 403 (No autorizado), como se muestra en la Figura 5.46 al realizar una prueba con *Postman*.

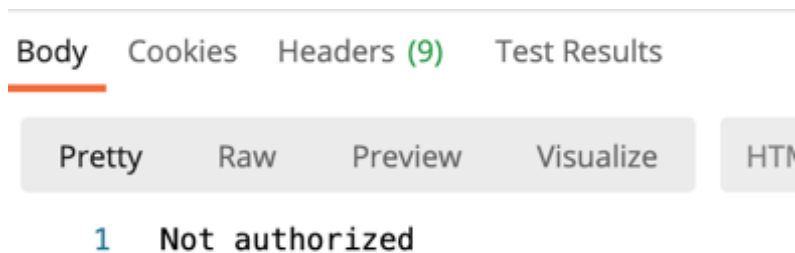


Figura 5.46. Respuesta JSON fallida del nodo raíz al acceder a la red *blockchain*

Si el nodo que intenta acceder a la red *blockchain* si está designado como nodo validador, al conectarse a la red *blockchain* a través del nodo raíz, este recibe una respuesta exitosa con el enlace para acceder al servidor *Redis* y así poder participar en la *blockchain*. Esto demuestra que la *blockchain* privada ha sido correctamente implementada, ya que solo aquellos nodos designados por el administrador de la *blockchain* pueden participar en ella.

En la Figura 5.47 se muestra la respuesta recibida cuando el nodo que solicita el acceso a la red *blockchain* es un nodo designado por el nodo raíz, en el que le envía el pool de transacciones actual, la cadena de bloques y enlace al servidor *Redis* para participar en la *blockchain* y comunicarse con otros nodos. Por otro lado, se ha comprobado que para añadir datos a la *blockchain* funciona correctamente. Cuando se valida una transacción, el servidor web realiza a la ruta del nodo raíz */api/transact* con el contenido de la intervención. Como se muestra en la Figura 5.48, la petición es recibida por el nodo raíz, el cual crea una transacción para la intervención recibida y la envía, a través del servidor *Redis*, al resto de nodos de la red para que la añadan a su pool de transacciones.

```

Body Cookies Headers (9) Test Results
Pretty Raw Preview Visualize JSON
1 {
2   "transactionPool": {},
3   "blockchain": [
4     {
5       "timestamp": 1622079334,
6       "lastHash": "----",
7       "hash": "first-hash",
8       "data": [],
9       "nonce": 0,
10      "difficulty": 10
11    }
12  ],
13  "redisUrl": "redis://:pd685fdeceba9347da25d950bd936672c41974e301f2cb1f2b737640f:
14 }
  
```

Figura 5.47. Respuesta JSON del nodo raíz para acceder a la red *blockchain*

```

2021-06-09T08:18:40.643867+00:00 app[web.1]: [POST REQUEST] /api/transact
2021-06-09T08:18:40.728889+00:00 app[web.1]: añadiendo transacción Transaction {
2021-06-09T08:18:40.728891+00:00 app[web.1]: id: '8b73c48a-0061-41fb-99f2-97cb073c7eed',
2021-06-09T08:18:40.728892+00:00 app[web.1]: input: {
2021-06-09T08:18:40.728893+00:00 app[web.1]: timestamp: 1623226720649,
2021-06-09T08:18:40.728896+00:00 app[web.1]: videoAppID: '040b95675e77723bd259320479d55d5e4
4e44cdb3554b3f50fc7ae187e5a506',
2021-06-09T08:18:40.728904+00:00 app[web.1]: signature: '304502210097a13bba5b45e41ca4efd39t
523fc06f34e3f1c6ef1eaff084e1fa60265f44729',
2021-06-09T08:18:40.728905+00:00 app[web.1]: interventionToken: 'd4726077887bf21e51184b1d6t
2021-06-09T08:18:40.728905+00:00 app[web.1]: }
2021-06-09T08:18:40.728906+00:00 app[web.1]: }
  
```

Figura 5.48. Intervención recibida en el nodo raíz para añadir a la cadena de bloques

Además de enviar la transacción al resto de nodos, el nodo raíz elige de forma aleatoria, entre la lista de nodos validadores, el nodo que se encargaría de realizar el minado de un bloque para añadir la transacción recibida en la *blockchain*. Para ello, junto a la transacción creada, el nodo raíz envía el ID del nodo encargado de realizar el minado. Cuando el nodo validador elegido recibe la transacción, realiza el proceso de minado para crear un nuevo bloque con la transacción recibida como se muestra en la Figura 5.49.

Una vez que se realiza el proceso de minado, la cadena de bloques, con el nuevo bloque minado, es compartida por el nodo que ha realizado el minado a través del servidor *Redis* para que la reciban el resto de los nodos que conforman la red *blockchain*.

Si la cadena de bloques es válida, se reemplaza la cadena de bloques actual por la nueva cadena de bloques recibida, la cual contiene el bloque recientemente añadido. En la Figura 5.50 se muestra como la nueva cadena de bloques es recibida en el nodo raíz, y tras comprobar su validez, se reemplaza por la cadena de bloques anterior.

```

2021-06-09T08:18:40.960556+00:00 app[web.1]: mining transaction
2021-06-09T08:18:40.961459+00:00 app[web.1]: validate {
2021-06-09T08:18:40.961460+00:00 app[web.1]: id: '8b73c48a-0061-41fb-99f2-97cb073c7eed',
2021-06-09T08:18:40.961461+00:00 app[web.1]: input: {
2021-06-09T08:18:40.961461+00:00 app[web.1]: timestamp: 1623226720649,
2021-06-09T08:18:40.961462+00:00 app[web.1]: videoAppID: '040b95675e77723bd259320479d55d5e4
4e44cdb3554b3f50fc7ae187e5a506',
2021-06-09T08:18:40.961462+00:00 app[web.1]: signature: '304502210097a13bba5b45e41ca4efd39f
523fc06f34e3f1c6ef1eaff084e1fa60265f44729',
2021-06-09T08:18:40.961463+00:00 app[web.1]: interventionToken: 'd4726077887bf21e51184b1d6f
2021-06-09T08:18:40.961463+00:00 app[web.1]: }
2021-06-09T08:18:40.961463+00:00 app[web.1]: }

```

Figura 5.49. Transacción recibida en el nodo validador designado para realizar el minado

```

2021-06-09T08:18:41.301904+00:00 app[web.1]: Replacing chain with [
2021-06-09T08:18:41.301906+00:00 app[web.1]: {
2021-06-09T08:18:41.301906+00:00 app[web.1]: timestamp: 1622079334,
2021-06-09T08:18:41.301906+00:00 app[web.1]: lastHash: '----',
2021-06-09T08:18:41.301907+00:00 app[web.1]: hash: 'first-hash',
2021-06-09T08:18:41.301908+00:00 app[web.1]: data: [],
2021-06-09T08:18:41.301908+00:00 app[web.1]: nonce: 0,
2021-06-09T08:18:41.301908+00:00 app[web.1]: difficulty: 10
2021-06-09T08:18:41.301909+00:00 app[web.1]: },
2021-06-09T08:18:41.301909+00:00 app[web.1]: {
2021-06-09T08:18:41.301909+00:00 app[web.1]: timestamp: 1623226721217,
2021-06-09T08:18:41.301909+00:00 app[web.1]: lastHash: 'first-hash',
2021-06-09T08:18:41.301914+00:00 app[web.1]: hash: '002b027e8e70cf1812a466eeb4864b7aa
2021-06-09T08:18:41.301914+00:00 app[web.1]: data: [ [Object] ],
2021-06-09T08:18:41.301915+00:00 app[web.1]: nonce: 1023,
2021-06-09T08:18:41.301915+00:00 app[web.1]: difficulty: 10
2021-06-09T08:18:41.301915+00:00 app[web.1]: }
2021-06-09T08:18:41.301916+00:00 app[web.1]: ]

```

Figura 5.50. Cadena de bloques recibida y validada en el nodo raíz

## Cadena de bloques

Bloque	Hash del bloque	Marca de tiempo	Dificultad de minado	Nonce	Hash del bloque anterior
0	first-hash	19/1/1970 18:34:39	10	0	----
1	002b027e8e70cf1812a4...	9/6/2021 9:18:41	10	1023	first-hash
ID de la transacción		Token de la intervención			
8b73c48a-0061-41fb-99f2-97cb073c7eed		d4726077887bf21e51184b1d6b8d547d4de96cd1bd9f98c5c41561c760ab0c94			

Figura 5.51. Lista de cadena de bloques en el panel de administración del nodo raíz

Desde el panel de administración del nodo raíz, como se muestra en la Figura 5.51, la tabla con la cadena de bloques se ha actualizado con el nuevo bloque añadido. Con esto, se manifiesta que el proceso de minado funciona de forma correcta y se puede añadir datos a la cadena de bloques.

### 5.3.3. Verificación de intervenciones realizadas

Por último, se ha comprobado que la *blockchain* implementada resuelve el problema planteado para verificar la integridad de las intervenciones realizadas. Integrar la *blockchain* implementada en sistema de registro de intervenciones ha permitido detectar si una intervención almacenada en el sistema ha sido alterada o modificada.

Cuando un usuario quiere verificar una intervención, se consulta en la *blockchain* la transacción correspondiente a dicha intervención para encontrar el *token* que contiene el *hash* del contenido multimedia de la intervención. Posteriormente, se calcula el *hash* del contenido multimedia de la intervención y si coincide con el *hash* almacenado en la cadena de bloques, se puede corroborar la autenticidad de la intervención almacenada en el sistema. En la Figura 5.52 se muestra un proceso de verificación con éxito si se cumplen las condiciones descritas.

Para que comprobar que el sistema es capaz de detectar si el contenido multimedia de una intervención ha sido alterado durante el proceso de verificación, se ha realizado una prueba en la que se ha cambiado el contenido multimedia de una intervención por otro en la plataforma de almacenamiento. Para ello, se ha eliminado en *Firebase*, a través de la interfaz de usuario que ofrecen desde su plataforma, el contenido multimedia almacenado

en *Firebase Storage*, y se ha subido uno nuevo con el mismo ID de la intervención y posteriormente, se ha cambiado el enlace del contenido multimedia de la intervención en *Firestore* por el enlace del nuevo contenido multimedia.

Realizando el proceso de verificación para la intervención alterada, el sistema desarrollado es capaz de detectar que la intervención ha sido modificada, tal y como se muestra en la Figura 5.53. Esto se debe a que el *token* de la intervención almacenado en la *blockchain* y el *hash* calculado no coinciden.

Por tanto, se puede concluir que el sistema de verificación de intervenciones, gracias a la integración de la *blockchain*, se puede utilizar para comprobar la integridad y autenticidad de las intervenciones almacenadas en el sistema. Esto resulta en que, si un tercero modificara o cambiara el contenido multimedia por otro, el sistema sería capaz de detectar que la intervención ha sido modificada.

#### Proceso de verificación

- ✓ Token de la intervención encontrado en la blockchain  
Token de la intervención: d4726077887bf21e51184b1d6b8d547d4de96cd1bd9f98c5c41561c760ab0c94
- ✓ Calculado hash de la intervención  
Hash de la intervención calculado: d4726077887bf21e51184b1d6b8d547d4de96cd1bd9f98c5c41561c760ab0c94
- ✓ El audio de la intervención es válido y no ha sido alterado  
El hash calculado y el token de la intervención almacenado en la blockchain coinciden

*Figura 5.52. Resultado de verificación con éxito de la intervención*

#### Proceso de verificación

- ✓ Token de la intervención encontrado en la blockchain  
Token de la intervención: d4726077887bf21e51184b1d6b8d547d4de96cd1bd9f98c5c41561c760ab0c94
- ✓ Calculado hash de la intervención  
Hash de la intervención calculado: 3abfd579fd15562e7fa48e01e615bd2e18f2269826921b6441613c629ef1b893
- ✗ No es posible confirmar la validez del audio de la intervención  
El hash calculado y el token almacenado en la blockchain no coinciden, por lo que es posible que el audio haya sido alterado

*Figura 5.53. Detección de intervención modificada durante el proceso de verificación*



# 6. Conclusiones

---

En este capítulo se describen las conclusiones conseguidas tras el desarrollo y despliegue del sistema. Finalmente, se presentan diferentes posibles ampliaciones que se podrían llevar a cabo en el TFG.



## 6.1. Conclusiones

En este TFG se ha implementado una plataforma web de comunicación multimedia, partiendo desde el diseño de la interfaz de usuario al desarrollo del código del servidor y la infraestructura para la autenticación de usuario y el almacenamiento de datos. A su vez, para el almacenamiento de las intervenciones realizadas en las reuniones realizadas en la plataforma, se ha implementado un sistema de registro de intervenciones que integra la tecnología *Blockchain* para verificar la integridad y autenticidad de las intervenciones registradas. Se ha podido concluir que la aplicación desarrollada funciona correctamente y tiene un buen rendimiento con las pruebas realizadas.

Por una parte, para el desarrollo de la comunicación multimedia, haber utilizado *WebRTC* me ha ayudado a entender su funcionamiento y cómo se debe utilizar para proporcionar un intercambio de tráfico multimedia entre clientes web. Esto me ha suscitado gran interés ya que es una tecnología muy versátil para soluciones donde se necesitan comunicaciones en tiempo real y de baja latencia a través de Internet. Asimismo, para el propio desarrollo de la aplicación web, el hecho de haber empleado diferentes tecnologías del ámbito del desarrollo web me ha brindado la oportunidad de adquirir experiencia con diferentes *frameworks* y herramientas.

Por otra parte, el desarrollo desde cero de la *blockchain* implementada en el sistema me ha permitido conocer los fundamentos de esta tecnología y los mecanismos que utiliza para poder mantener un registro inalterable de la información. A su vez, el trabajo de investigación previo sobre la *blockchain* me ha hecho prestar atención a esta tecnología que, aunque no es tratada de forma directa en la carrera, se puede aplicar en numerosos ámbitos donde se necesita una manera de asegurar la integridad de la información de forma descentralizada y segura. Con respecto a la *blockchain* implementada, se puede concluir que funciona correctamente, permitiendo almacenar información en la misma de manera unívoca y permite gestionar que nodos pueden participar en la misma.

En cuanto a la integración de las diferentes partes del sistema, este funciona correctamente de manera integrada, permitiendo la creación, notificación y realización de reuniones telemáticas, así como el registro de intervención y su consulta y verificación de

las mismas utilizando la *blockchain* implementada, garantizando la autenticidad de las intervenciones que hayan sido aceptadas por los usuarios validadores de las reuniones.

Por último, el haber tratado las diferentes tecnologías descritas, desempeñando un rol de desarrollador *fullstack* al tener que desarrollar tanto el *backend* como el *frontend* del sistema, me ha preparado para poder ser competitivo en empresas del sector. Asimismo, con el conocimiento adquirido me permite afrontar futuros proyectos en el ámbito de la Telemática.

Desde el punto de vista personal, no creí que fuera capaz de realizar un proyecto de cierta envergadura como el descrito. Sin embargo, con lo que he estudiado en la carrera he podido aplicar conocimientos y metodologías para afrontar la implementación del proyecto, solucionando el problema propuesto. Por otro lado, aunque haya habido tecnologías como *Blockchain* o *WebRTC* que no he visto en profundidad durante la carrera, durante el desarrollo del TFG he podido solventar las dudas que me han surgido investigando sobre dichas tecnologías, lo cual me ha resultado muy útil tanto a nivel profesional como académico.

## 6.2. Posibles ampliaciones

Tras valorar el trabajo realizado y el sistema implementado, se han establecido diferentes posibles ampliaciones, las cuales describimos a continuación:

- *Aumentar la fiabilidad del sistema de almacenamiento de intervenciones.* En el sistema de registro de intervenciones desarrollado, las intervenciones se almacenan en una única plataforma (en nuestro caso, mediante el servicio de almacenamiento de datos de *Firebase*). Esto puede ser un problema ya que, si la intervención es eliminada de dicha plataforma, no es posible recuperar el contenido multimedia de dicha intervención. Por este motivo, el contenido multimedia de las intervenciones validadas debería almacenarse en diversas plataformas en lugar de en un único servidor, lo cual mejoraría la fiabilidad del sistema.
- *Permitir videoconferencias con más participantes.* Cómo se ha descrito en el apartado 5.3.1, *Rendimiento de la comunicación multimedia*, la utilización de una

arquitectura utilizando un SFU permitiría mejorar la eficiencia de la comunicación multimedia en reuniones donde existen una cantidad moderada de usuarios. Por este motivo, se podría desplegar un SFU que permitiera, en aquellas reuniones donde se espera un número elevado de usuarios, utilizar el SFU desplegado para realizar la retransmisión del tráfico multimedia en la reunión.

- *Añadir persistencia de datos a los nodos de la red blockchain.* La cadena de bloques, que se aloja en los nodos desplegados en *Heroku*, no necesita de la utilización de una base de datos en cada uno para almacenar la información ya que los datos de la *blockchain* (bloques, transacciones...) no se eliminan si no se detiene el nodo. No obstante, si se el nodo se reiniciara, se eliminarían los datos que almacena (aunque cuando otro nodo retransmita la cadena de bloques tras realizar el proceso de minado, todos los nodos que se encuentren en la red *blockchain* recibirían esta última cadena de bloques). Sin embargo, sería conveniente un sistema de persistencia de datos en cada nodo para que no pierdan dicha información si se apagara.

# Bibliografía

---

- [1] C. Trueman, "Pandemic leads to surge in video conferencing app downloads", *ComputerWorld*, Abr. 03, 2020. [En línea] Disponible en: <https://www.computerworld.com/article/3535800/pandemic-leads-to-surge-in-video-conferencing-app-downloads.html> (accedido por última vez el 6 de febrero de 2021).
- [2] Zoom, "Zoom", *Zoom* [En línea] Disponible en: <https://zoom.us/> (accedido por última vez el 11 de junio de 2021).
- [3] Google, "Google Meets", *Google Meets* [En línea] Disponible en: <https://meet.google.com/> (accedido por última vez el 11 de junio de 2021).
- [4] Microsoft, "Microsoft Teams", *Microsoft* [En línea] Disponible en: <https://www.microsoft.com/es-es/microsoft-teams/group-chat-software> (accedido por última vez el 11 de junio de 2021).
- [5] D. Curry, "Microsoft Teams Revenue and Usage Statistics (2021)", *BusinessOfApps*, 2021. [En línea] Disponible en: <https://www.businessofapps.com/data/microsoft-teams-statistics/> (accedido por última vez el 11 de junio de 2021).
- [6] Bit2Me, "¿Qué es la Cadena de Bloques (Blockchain)?", *Bit2Me Academy*. [En línea] Disponible en: <https://academy.bit2me.com/que-es-cadena-de-bloques-blockchain/> (accedido por última vez el 4 de mayo de 2021).
- [7] M. Crosby, N. Pradan, S. Verma, V. Kalyanaraman, "BlockChain Technology: Beyond Bitcoin", 2016.
- [8] F. Casino, E. Politou, E. Alepis, C. Patsakis, "Immutability and Decentralized Storage: An Analysis of Emerging Threats", *IEEE Access*, vol. 8, pp. 4737–4744, 2020, doi: 10.1109/ACCESS.2019.2962017.
- [9] K. Stone, "The State of Video Conferencing in 2020", *Getvoip*, 2020. [En línea] Disponible en: <https://getvoip.com/blog/2020/07/07/video-conferencing-stats/> (accedido por última vez el 7 de febrero de 2021).
- [10] O. Czmut, "50 video conferencing statistics for the year 2020 (remote work, webinars and more)", *LiveWebinar*, 2020. [En línea] Disponible en: <https://www.livewebinar.com/blog/webinar-marketing/50-video-conferencing-statistics-for-the-year-2020> (accedido por última vez el 24 de mayo de 2021).
- [11] Cisco, "18 video conferencing statistics for 2021", *Webex*, 2021. [En línea] Disponible en: <https://blog.webex.com/video-conferencing/18-video-conferencing-statistics->

- for-2021/ (accedido por última vez el 24 de mayo de 2021).
- [12] P. Sanz Bayón, "Comentarios sobre el Texto Refundido de la Ley de Sociedades de Capital (Real Decreto Legislativo 1/2010, de 2 de julio)", *Revista: Revista ICADE, Periodo: 4, Número: 81, Página inicial: 289, Página final: 293*, pp. 1–157, 2010.
- [13] Jefatura del Estado, "LEGISLACIÓN CONSOLIDADA Ley 49/1960, de 21 de julio, sobre propiedad horizontal.", pp. 1–19, 1960.
- [14] Highfive, "10 Video Conferencing Statistics", *Highfive*. [En línea] Disponible en: <https://highfive.com/blog/10-video-conferencing-statistics> (accedido por última vez el 5 de junio de 2021).
- [15] Y. Jia *et al.*, "Transfer learning from speaker verification to multispeaker text-to-speech synthesis", *Advances in Neural Information Processing Systems*, vol. 2018-Decem, no. NeurIPS, pp. 4480–4490, 2018.
- [16] Y. Wu. Ye Jia *et al.*, "Audio samples from 'Transfer Learning from Speaker Verification to Multispeaker Text-To-Speech Synthesis.'" [En línea] Disponible en: [https://google.github.io/tacotron/publications/speaker\\_adaptation/](https://google.github.io/tacotron/publications/speaker_adaptation/) (accedido por última vez el 5 de junio de 2021).
- [17] "Transaction Pool", *Computer Science Wiki*. [En línea] Disponible en: [https://computersciencewiki.org/index.php/Transaction\\_pool](https://computersciencewiki.org/index.php/Transaction_pool) (Accedido: May 19, 2021).
- [18] Figma, "Figma", *Figma*, [En línea] Disponible en: <https://www.figma.com/ui-design-tool/> (accedido por última vez el 8 de junio de 2021).
- [19] DesarrolloWeb, "¿Qué es MVC?", *DesarrolloWeb.com*. [En línea] Disponible en: <https://desarrolloweb.com/articulos/que-es-mvc.html> (accedido por última vez el 9 de mayo de 2021).
- [20] R. Delgado, "bc-videochat", *Github*, 2021. [En línea] Disponible en: <https://github.com/rrenub/bc-videochat> (accedido por última vez el 23 de mayo de 2021).
- [21] R. Delgado, "bc-videochat-peer", *Github*, 2021. [En línea] Disponible en: <https://github.com/rrenub/bc-videochat-peer> (accedido por última vez el 23 de junio de 2021).
- [22] R. Delgado, "bc-videochat-root-peer", *Github*, 2021. [En línea] Disponible en: <https://github.com/rrenub/bc-videochat-root-peer> (accedido por última vez el 23

de junio de 2021).

- [23] Klughammer, "randomstring", *Github*. [En línea] Disponible en: <https://github.com/klughammer/node-randomstring> (accedido por última vez el 15 de junio de 2021).
- [24] Google, "Custom access with Custom Claims and security", *Firebase*. [En línea] Disponible en: <https://firebase.google.com/docs/auth/admin/custom-claims> (accedido por última vez el 14 de junio de 2021).
- [25] Socket.io, "Rooms", *Socket.io*. [En línea] Disponible en: <https://socket.io/docs/v3/rooms/index.html> (accedido por última vez el 14 de junio de 2021).
- [26] "Array.prototype.map()", *MDN Web Docs*, 2021. [En línea] Disponible en: [https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/map) (accedido por última vez el 23 de junio de 2021).
- [27] Google, "Transactions and batch writes", *Firebase*. [En línea] Disponible en: <https://firebase.google.com/docs/firestore/manage-data/transactions> (accedido por última vez el 14 de junio de 2021).
- [28] J. Maldonado, "¿Qué es SHA-256? El algoritmo criptográfico usado por Bitcoin", *CoinTelegraph*, 2020. [En línea] Disponible en: <https://es.cointelegraph.com/explained/what-is-sha-256-the-cryptographic-algorithm-used-by-bitcoin> (accedido por última vez el 11 de mayo de 2021).
- [29] R. Delgado, "Aplicación web de comunicación multimedia", *Heroku*, 2021. [En línea] Disponible en: <https://bc-videochat.herokuapp.com/> (accedido por última vez el 14 de junio de 2021).
- [30] R. Delgado, "Funcionamiento de la aplicación", *Youtube*, 2021. [En línea] Disponible en: <https://youtu.be/2-uV4BSXs0I> (accedido por última vez el 23 de junio de 2021).
- [31] IONOS, "WebM: todo lo que debes saber sobre el formato de Google", *IONOS*. [En línea] Disponible en: <https://www.ionos.es/digitalguide/paginas-web/creacion-de-paginas-web/webm/> (accedido por última vez el 8 de junio de 2021).
- [32] R. Delgado, "Nodo raíz de la blockchain", *Heroku*, 2021. [En línea] Disponible en: <https://bc-videochat-root-peer.herokuapp.com/> (accedido por última vez el 8 de junio de 2021).
- [33] testRTC, "testRTC", *testRTC*, <https://testrtc.com/> (accedido por última vez el 9 de

- junio de 2021).
- [34] S. JM. Valin, Mozilla, K. Vos, T. Terriberry, "RFC 6716: Definition of Opus Audio Codec", 2012. [En línea] Disponible en: <https://datatracker.ietf.org/doc/html/rfc6716> (accedido por última vez el 9 de junio de 2021).
  - [35] G. J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, Y. Xu, "RFC 6386: VP8 Data Format", 2011. [En línea] Disponible en: <https://datatracker.ietf.org/doc/html/rfc6386> (accedido por última vez el 9 de junio de 2021).
  - [36] M. Grech, "Acceptable Jitter & Latency for VoIP: Everything Your Need to Know", *Getvoip*, 2018. [En línea] Disponible en: <https://getvoip.com/blog/2018/12/20/acceptable-jitter-latency/> (accedido por última vez el 9 de junio de 2021).
  - [37] "SFU (Selective Forwarding Unit)", *WebRTC Glossary*, 2017. [En línea] Disponible en: <https://webrtcglossary.com/sfu/> (accedido por última vez el 8 de junio de 2021).
  - [38] "JavaScript", *MDN Web Docs*, 2021. [En línea] Disponible en: <https://developer.mozilla.org/es/docs/Web/JavaScript> (accedido por última vez el 10 de mayo de 2021).
  - [39] Node.js, "Node.js", *Node.js*, <https://nodejs.org/es/> (accedido por última vez el 10 de mayo de 2021).
  - [40] A. Rai, "JavaScript - A prototype based language", 2019. [En línea] Disponible en: <https://theflyingmantis.medium.com/javascript-a-prototype-based-language-7e814cc7ae0b> (accedido por última vez el 10 de mayo de 2021).
  - [41] D. Austin, "What Are JavaScript Programming Paradigms?", *JavaScript in Plain English*, 2019. [En línea] Disponible en: <https://javascript.plainenglish.io/what-are-javascript-programming-paradigms-3ef0f576dfdb> (accedido por última vez el 10 de mayo de 2021).
  - [42] Google, "V8 JavaScript Engine", <https://v8.dev/> (Accedido: May 10, 2021).
  - [43] J. Luca, "Qué es NodeJS y para qué sirve", *OpenWebinars*, 2019. [En línea] Disponible en: <https://openwebinars.net/blog/que-es-nodejs/> (Accedido: May 10, 2021).
  - [44] Facebook, "React", *React*, [En línea] Disponible en: <https://es.reactjs.org/> (accedido por última vez el 10 de mayo de 2021).



- [45] A. Nawo, "10 most popular NPM packages", 2019. [En línea] Disponible en: <https://areknawo.com/10-most-popular-npm-packages/> (accedido por última vez el 10 de mayo de 2021).
- [46] Facebook, "React: Componentes y propiedades", *React docs*. [En línea] Disponible en: <https://es.reactjs.org/docs/components-and-props.html> (accedido por última vez el 10 de mayo de 2021).
- [47] Facebook, "Un vistazo a los Hooks", *React docs*. [En línea] Disponible en: <https://es.reactjs.org/docs/hooks-overview.html> (accedido por última vez el 11 de mayo de 2021).
- [48] Facebook, "Usando el Hook de estado", *React docs*. [En línea] Disponible en: <https://es.reactjs.org/docs/hooks-state.html> (accedido por última vez el 11 de mayo de 2021).
- [49] Redis, "Redis", *Redis*, [En línea] Disponible en: <https://redis.io/> (accedido por última vez el 11 de mayo de 2021).
- [50] Amazon, "¿Qué es Redis?", *AWS*, [En línea] Disponible en: <https://aws.amazon.com/es/elasticache/what-is-redis/> (accedido por última vez el 11 de mayo de 2021).
- [51] Google, "¿Qué es PubSub?", *Google Cloud*, [En línea] Disponible en: <https://cloud.google.com/pubsub/docs/overview> (accedido por última vez el 11 de mayo de 2021).
- [52] NodeRedis, "Node Redis: A high performance Node.js Redis client", *Github*. [En línea] Disponible en: <https://github.com/NodeRedis/node-redis> (accedido por última vez el 11 de mayo de 2021).
- [53] OptimalBits, "Bull: Premium Queue Package for handling distributed jobs and messages", *OptimalBits*, [En línea] Disponible en: <https://github.com/OptimalBits/bull> (accedido por última vez el 11 de mayo de 2021).
- [54] Heroku, "Heroku", *Heroku*, [En línea] Disponible en: <https://www.heroku.com/home> (accedido por última vez el 10 de mayo de 2021).
- [55] B. Violino, "What is PaaS? Platform-as-a-service explained", *InfoWorld*, 2019. [En línea] Disponible en: <https://www.infoworld.com/article/3223434/what-is-paaS-software-development-in-the-cloud.html> (accedido por última vez el 10 de mayo de 2021).

- 2021).
- [56] Heroku, "Heroku Dynos: Lightweight containers for running apps", *Heroku*. [En línea] Disponible en: <https://www.heroku.com/dynos> (accedido por última vez el 10 de mayo de 2021).
- [57] Heroku, "Heroku Add-ons", *Heroku*, [En línea], *Heroku*, Disponible en: <https://elements.heroku.com/addons> (accedido por última vez el 11 de mayo de 2021).
- [58] Heroku, "Heroku Redis: Reliable and powerful Redis as a service", *Heroku*, [En línea] Disponible en: <https://elements.heroku.com/addons/heroku-redis> (accedido por última vez el 11 de mayo de 2021).
- [59] Google, "Firebase", *Firebase*, [En línea] Disponible en: <https://firebase.google.com/?hl=es> (accedido por última vez el 11 de mayo de 2021).
- [60] O. Blancarte, "¿Qué es el Backend as a Service? (BaaS)", *Oscar Blancarte Software Architect*, 2018. [En línea] Disponible en: <https://www.oscarblancarteblog.com/2018/06/18/backend-as-service-baas/> (accedido por última vez el 11 de mayo de 2021).
- [61] S. López, "Firebase: qué es, para qué sirve, funcionalidades y ventajas", *Digital55*, 2020. [En línea] Disponible en: <https://www.digital55.com/desarrollo-tecnologia/que-es-firebase-funcionalidades-ventajas-conclusiones/> (accedido por última vez el 11 de mayo de 2021).
- [62] Google, "Cloud Firestore", *Firebase*, [En línea] Disponible en: <https://firebase.google.com/docs/firestore> (accedido por última vez el 11 de mayo de 2021).
- [63] Google, "Firebase Authentication", *Firebase*, [En línea] Disponible en: <https://firebase.google.com/docs/auth> (accedido por última vez el 11 de mayo de 2021).
- [64] Google, "Cloud Storage for Firebase", *Firebase*, [En línea] Disponible en: <https://firebase.google.com/docs/storage> (accedido por última vez el 11 de mayo de 2021).
- [65] Postman, "Postman", *Postman* <https://www.postman.com/> (accedido por última vez el 11 de junio de 2021).

- [66] V. Cuervo, "¿Qué es Postman?", *Arquitecto IT*, 2019. [En línea] Disponible en: <https://www.arquitectoit.com/postman/que-es-postman/> (accedido por última vez el 11 de mayo de 2021).
- [67] Express, "Express: Infraestructura web rápida, minimalista y flexible para Node.js", *Express*, <https://expressjs.com/es/> (accedido por última vez el 11 de mayo de 2021).
- [68] Axios, "Axios", *Github*. [En línea] Disponible en: <https://github.com/axios/axios> (accedido por última vez el 11 de mayo de 2021).
- [69] F. Indutny, "elliptic", *Github*. [En línea] Disponible en: <https://github.com/indutny/elliptic> (accedido por última vez el 11 de mayo de 2021).
- [70] Express, "cors", *Github*. [En línea] Disponible en: <https://github.com/expressjs/cors> (accedido por última vez el 11 de mayo de 2021).
- [71] "Control de acceso HTTP (CORS)", *MDN Web Docs*. [En línea] Disponible en: <https://developer.mozilla.org/es/docs/Web/HTTP/CORS> (accedido por última vez el 11 de mayo de 2021).
- [72] Node.js, "Crypto", *Node.js Documentation*. [En línea] Disponible en: <https://nodejs.org/api/crypto.html> (accedido por última vez el 11 de mayo de 2021).
- [73] Firebase, "Firebase Admin Node.js SDK", *Github*. [En línea] Disponible en: <https://github.com/firebase/firebase-admin-node> (accedido por última vez el 11 de mayo de 2021).
- [74] Firebase, "Firebase Javascript SDK", *Github*. [En línea] Disponible en: <https://github.com/firebase/firebase-js-sdk> (accedido por última vez el 11 de mayo de 2021).
- [75] Node-schedule, "Node Schedule", *Github*. [En línea] Disponible en: <https://github.com/node-schedule/node-schedule> (accedido por última vez el 11 de mayo de 2021).
- [76] A. Reinman, "Nodemailer", *Nodemailer*, [En línea] Disponible en: <https://nodemailer.com/about/> (accedido por última vez el 11 de mayo de 2021).
- [77] Socket.io, "Socket.IO", *socket.io* [En línea] Disponible en: <https://socket.io/> (accedido por última vez el 11 de mayo de 2021).
- [78] "WebSockets", *MDN Web Docs*, 2021. [En línea] Disponible en: [https://developer.mozilla.org/es/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/es/docs/Web/API/WebSockets_API) (accedido por última vez el 11 de mayo de 2021).

- [79] "AJAX", *MDN Web Docs*, 2021. [En línea] Disponible en: <https://developer.mozilla.org/es/docs/Web/Guide/AJAX> (accedido por última vez el 11 de mayo de 2021).
- [80] Uuidjs, "uuid", *Github*. [En línea] Disponible en: <https://github.com/uidjs/uuid> (accedido por última vez el 11 de mayo de 2021).
- [81] R. Salz, P. Leach, M. Mealling, "A Universally Unique IDentifier (UUID) URN Namespace", *RFC 4122*, 2005. [En línea] Disponible en: <https://www.rfc-editor.org/info/rfc4122> (accedido por última vez el 11 de mayo de 2021).
- [82] Ant design, "Ant design", *Ant Design*, [En línea] Disponible en: <https://ant.design/> (accedido por última vez el 11 de mayo de 2021).
- [83] ReactTraining, "React Router: Declarative routing for React", *Github*. [En línea] Disponible en: <https://github.com/ReactTraining/react-router> (accedido por última vez el 11 de mayo de 2021).
- [84] Wmik, "use-media-recorder", *Github*. [En línea] Disponible en: <https://github.com/wmik/use-media-recorder> (accedido por última vez el 11 de mayo de 2021).
- [85] "MediaRecorder", *MDN Web Docs*. [En línea] Disponible en: <https://developer.mozilla.org/en-US/docs/Web/API/MediaRecorder> (accedido por última vez el 11 de mayo de 2021).
- [86] J. McCandless, "ReactAudioPlayer", *NPM*. [En línea] Disponible en: <https://www.npmjs.com/package/react-audio-player> (accedido por última vez el 23 de junio de 2021).
- [87] Google, "WebRTC", *WebRTC*, [En línea] Disponible en: <https://webrtc.org/> (accedido por última vez el 10 de mayo de 2021).
- [88] S. Loreto and S. Pietro Romano, *O'Reilly*, 9781449371838 ISBN, *Real-Time Communication with WebRTC*. 2014.
- [89] J. R. A. Keranen, C. Holmberg, Ericsson, "RFC 5245: Interactive Connectivity Establishment (ICE)", *RFC 8845*, 2018. [En línea] Disponible en: <https://datatracker.ietf.org/doc/html/rfc8445> (accedido por última vez el 10 de mayo de 2021).
- [90] "getUserMedia", *MDN Web Docs*, 2021. [En línea] Disponible en: <https://developer.mozilla.org/es/docs/Web/API/MediaDevices/getUserMedia>

(accedido por última vez el 7 de junio de 2021).

- [91] M. Naranjo, "Videollamadas P2P y WebRTC: las más seguras de la red", *ADSLzone*, 2021. [En línea] Disponible en: <https://www.adslzone.net/como-se-hace/software/hacer-videollamadas-p2p/> (accedido por última vez el 7 de junio de 2021).
- [92] "Signaling", *WebRTC for the curious*, 2021. [En línea] Disponible en: <https://webrtcforthe curious.com/docs/02-signaling/> (accedido por última vez el 7 de junio de 2021).
- [93] GeekforGeeks, "Network Address Translation (NAT)", *GeeksforGeeks*, 2019. [En línea] Disponible en: <https://www.geeksforgeeks.org/network-address-translation-nat/> (accedido por última vez el 7 de junio de 2021).
- [94] "Why does WebRTC need a dedicated subsystem for connecting?" *WebRTC for the curious*, 2021. [En línea] Disponible en: <https://webrtcforthe curious.com/docs/03-connecting/> (accedido por última vez el 7 de junio de 2021).
- [95] D. W. J. Rosenberg, Cisco, R. Mahy, P. Matthews, "RFC 5389: Session Traversal Utilities for NAT (STUN)", *RFC 5389*, 2008. [En línea] Disponible en: <https://datatracker.ietf.org/doc/html/rfc5389> (accedido por última vez el 7 de junio de 2021)
- [96] J. R. R. Mahy, P. Matthews, Alcatel-Lucent, "RFC 5766: Traversal Using Relays around NAT (TURN)", *RFC 5766*, 2010. [En línea] Disponible en: <https://datatracker.ietf.org/doc/html/rfc5766> (accedido por última vez el 7 de junio de 2021)
- [97] BBVA, "¿Cuál es la diferencia entre una DLT y 'blockchain'?" *BBVA*. [En línea] Disponible en: <https://www.bbva.com/es/diferencia-dlt-blockchain/> (accedido por última vez el 24 de mayo de 2021).
- [98] A. L. Tsilidou and G. Foroglou, "Further applications of the blockchain", *ResearchGate*, 2015. [En línea] Disponible en: <https://www.researchgate.net/publication/276304843> (accedido por última vez el 7 de junio de 2021)
- [99] Binance, "¿Qué es un Algoritmo de Consenso?", *Binance Academy*. [En línea] Disponible en: <https://academy.binance.com/es/articles/what-is-a-blockchain-consensus-algorithm> (accedido por última vez el 24 de mayo de 2021).

- [100] Bit2Me, “¿Qué es Prueba de trabajo / Proof of Work (PoW)?”, *Bit2Me Academy*. [En línea] Disponible en: <https://academy.bit2me.com/que-es-proof-of-work-pow/> (accedido por última vez el 24 de mayo de 2021).
- [101] Bit2Me, “¿Qué es un Ataque del 51%?”, *Bit2Me Academy*. [En línea] Disponible en: <https://academy.bit2me.com/que-es-un-ataque-del-51/> (accedido por última vez el 24 de mayo de 2021).
- [102] A. A. G. Agung, R. G. Dillak, D. R. Suchendra, and H. Robbi, “Proof of work: Energy inefficiency and profitability”, *Journal of Theoretical and Applied Information Technology*, vol. 97, no. 5, pp. 1623–1633, 2019.
- [103] Bit2Me, “¿Qué es Prueba de participación / Proof of Stake (PoS)?”, *Bit2Me Academy*. [En línea] Disponible en: <https://academy.bit2me.com/que-es-proof-of-stake-pos/> (accedido por última vez el 24 de mayo de 2021).
- [104] Bit2Me, “¿Qué es PoA (Proof of Authority – Prueba de Autoridad)?” *Bit2Me Academy*. [En línea] Disponible en: <https://academy.bit2me.com/que-es-proof-of-authority-poa/> (accedido por última vez el 24 de mayo de 2021).
- [105] Bit2Me, “Cuántos tipos de blockchain hay”, *Bit2Me Academy*. [En línea] Disponible en: <https://academy.bit2me.com/cuantos-tipos-de-blockchain-hay/> (accedido por última vez el 24 de mayo de 2021).
- [106] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system”, *Decentralized Business Review*, 2008, p. 21260
- [107] “Bitcoin Energy Consumption Index”, *Digieconomist*. [En línea] Disponible en: <https://digieconomist.net/bitcoin-energy-consumption/> (accessed May 24, 2021).
- [108] G. Iredale, “The Rise of Private Blockchain Technologies”, *101 Blockchains*. [En línea] Disponible en: <https://101blockchains.com/private-blockchain/> (accedido por última vez el 24 de mayo de 2021).
- [109] E. Hill, “An introduction to Tokenization”, *Cardano Forum*, 2020. [En línea] Disponible en: <https://forum.cardano.org/t/an-introduction-to-tokenization/39609> (accedido por última vez el 9 de mayo de 2021).
- [110] “CSS Media Queries”, *MDN Web Docs*. [En línea] Disponible en: [https://developer.mozilla.org/es/docs/Web/CSS/Media\\_Queries/Using\\_media\\_queries](https://developer.mozilla.org/es/docs/Web/CSS/Media_Queries/Using_media_queries) (accedido por última vez el 14 de junio de 2021).
- [111] L. Torvalds, “Git”, *Git*, [En línea] Disponible en: <https://git-scm.com/> (accedido por

última vez el 14 de junio de 2021).

# Pliego de condiciones

---

En este capítulo se describen las cuestiones relacionadas con la información de licencias, los derechos de autoría y las responsabilidades. Así como las condiciones bajo las que se ha desarrollado este TFG y las que se establecen al usuario para poder usar el software desarrollado.



## Pl.1. Condiciones hardware

Durante el desarrollo de este TFG se han utilizado los dispositivos hardware recogidos en la Tabla Pl.1.

## Pl.2. Condiciones software

Las herramientas software utilizadas durante el desarrollo del TFG están separadas para el equipo principal (ordenador portátil Macbook Pro 2019) en la Tabla Pl.2 y el dispositivo móvil (Xiaomi Redmi Note 8 Pro) en la Tabla Pl.3.

Equipo	Modelo	Fabricante
Ordenador portátil	MacBook Pro 2019 (13"), CPU Intel Core i5 8ª Gen. 2,4 GHz, 8 GB RAM, 256 GB SSD	Apple
Dispositivo móvil	Xiaomi Redmi Note 8 Pro (6,5"), CPU MediaTek Helio G90T 2,05 GHz, 6 GB RAM, 128 GB	Xiaomi

Tabla Pl.1. Condiciones Hardware.

Aplicación	Versión
Sistema Operativo MacOS	Big Sur
Microsoft Office	Office 365
Visual Studio Code	2020.1
Postman	7.36.5

Google Chrome	91.0.4472.114
JavaScript	ES6
Node.js	14.13.1
React	17.0.2

Tabla Pl.2. Condiciones Software del equipo principal.

Aplicación	Versión
Sistema Operativo Android	Android Pie
Google Chrome	91.0.4472.101

Tabla Pl.3. Condiciones Software del dispositivo móvil

### **Pl.3. Condiciones de uso por parte del usuario**

El objetivo de este apartado es comentar las condiciones hardware y software necesarios para que el usuario pueda utilizar el servicio que brinda la aplicación.

Se recomienda encarecidamente, para utilizar la aplicación desarrollada, el uso de un navegador compatible con WebRTC, como Firefox o Google Chrome. Para el despliegue del sistema, se recomienda la utilización de Heroku ya que el sistema ha sido preparado para su despliegue en esta plataforma.

### **Pl.4. Condiciones de licencia**

El código desarrollado en este trabajo es propiedad de la Universidad de Las Palmas de Gran Canaria y todos los que pretendan hacer uso de él deben aceptar todas las cláusulas establecidas en esta licencia. El uso de las plataformas se podría hacer bajo autorización del autor, tutores del TFG, y la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

## **Pl.5. Derechos de autor**

Tanto el código como la información que se adjunta están protegidos por las leyes de propiedad intelectual que les sean aplicables, así como las disposiciones de los tratados internacionales. Por tanto, el código se considera un producto protegido por derechos de autor. Esto no es óbice para que una persona pueda copiar o utilizar el código bajo la autorización del autor, tutores del TFG, y la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

## **Pl.6. Restricciones**

No se permite el uso de ingeniería inversa. Sí se permite la transferencia del código a un tercero siempre que no se conserve ninguna copia, incluyendo actualizaciones o material escrito adicional.

## **Pl.7. Garantía**

El autor del TFG lo presenta *AS IS* (tal cual), sin garantía implícita de ningún tipo. No se responsabiliza de los daños que pudieran causar a equipos o personas por el uso del código o la documentación. El autor no asegura, garantiza, o realiza ninguna declaración sobre el uso y resultados derivados de la utilización del código y/o del resto de la información proporcionada.

El código no está exento de errores y no está diseñado para entornos de riesgo que requieran de un funcionamiento a prueba de fallos. El autor rechaza expresamente cualquier garantía explícita e implícita de adecuación del código para actividades de riesgo.

## **Pl.8. Limitación de responsabilidad**

En ningún caso el autor ni los tutores, ni la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de las Palmas de Gran Canaria serían responsables de los perjuicios directos, indirectos incidentales o consiguientes, gastos, lucro cesante, pérdida de ahorros, interrupción de negocios, pérdida de información comercial o de negocio, o cualquier otra pérdida que resulte del uso o de la incapacidad de usar el código o la documentación. El usuario conoce y acepta este riesgo, así como el resto de las cláusulas y restricciones. El autor no reconoce otra garantía que no haya sido indicada anteriormente.

## **Pl.9. Otras consideraciones**

En el supuesto de que cualquier disposición de esta licencia sea declarada total o parcialmente inválida, las cláusulas afectadas serán modificadas convenientemente de manera que sea ejecutable una vez modificada, permaneciendo el resto de este contrato en vigencia. Este contrato se rige por las leyes de España. El usuario acepta la jurisdicción exclusiva de los tribunales españoles con relación a las disputas derivadas de la presente licencia.

# Presupuesto

---

## Pr.1. Componentes del presupuesto

El presupuesto calculado se divide en las siguientes partes:

- Recursos materiales.
- Trabajo tarifado por tiempo empleado.
- Material Fungible.
- Redacción de la documentación.
- Derechos de visado del *Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT)*.
- Gastos de tramitación y envío.
- Aplicación de impuestos y coste total.

## Pr.2. Recursos materiales

Para la correcta ejecución y desarrollo de este TFG han sido necesaria una serie de recursos hardware y herramientas software, las cuales pueden llevar asociadas un coste en sus licencias. Entre estos recursos, destaca principalmente como la parte hardware el ordenador portátil en el que se ha documentado la memoria y desarrollado y ejecutado el código implementado en este TFG, y el dispositivo móvil para probar el funcionamiento y la interfaz de usuario de la aplicación web en terminales móviles. Por la parte del software, cabe destacar que para el paquete *Microsoft Office* para la redacción de la memoria se utilizó la licencia de estudiante, por lo que no han supuesto ningún coste. En cuanto al resto de herramientas de software, su uso era gratuito y para la utilización de *Firebase* y *Heroku* se utilizó el plan de servicios gratuito, por lo que tampoco supuso ningún coste adicional.

Con el fin de establecer un cálculo de la amortización, se presupone el sistema de amortización como lineal, de tal forma que se asume que el inmovilizado material se desprecia de forma constante a lo largo de su vida útil.

Así, para llevar a cabo el cálculo de la cuota de amortización anual, se calcula usando la Ecuación Pr.1.

$$\text{Cuota anual} = \frac{\text{Valor de adquisición} - \text{Valor residual}}{\text{Número de años de vida útil}}$$

*Ecuación Pr.1*

De esta manera, la amortización total aparece reflejada en la Tabla Pr.1, así como los diferentes valores y cuotas de los diferentes recursos empleados en este TFG.

Recurso	Valor de adquisición (€)	Valor residual (*) (€)	Vida útil (años)	Cuota anual (€)	Uso (meses)	Cuota aplicable (€)
MacBook Pro 2019 (13"), CPU Intel Core i5 8ª Gen. 2,4 GHz, 8 GB RAM, 256 GB SSD	1679,0	503,70	5	235,06	5	235,06
Xiaomi Redmi Note 8 Pro (6,5"), CPU MediaTek Helio G90T 2,05 GHz, 6 GB RAM, 128 GB	269,0	80,70	3	62,77	5	62,76
MATERIALES AMORTIZABLES	TOTAL					172,30

*Tabla Pr.1. Amortización total.*

(\*) El valor residual puede ser definido como el valor teórico supuesto que tendría el elemento después de su vida útil. En el caso de un terminal móvil y portátil se deprecian en una tasa del 30 % con respecto a su valor. Según *Artículo 34, LISR*.

De esta manera, la amortización total aparece reflejada en la Tabla Pr.1, así como los diferentes valores y cuotas de los diferentes recursos empleados en este TFG.

El coste de los materiales amortizables es de un total de ciento setenta y dos con treinta céntimos (172,30 €).

### Pr.3. Trabajo tarificado por tiempo empleado

Este concepto contabiliza los gastos que corresponden a la mano de obra, según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación.

Según la tabla retributiva de personal contratado en proyectos de investigación elaborada por la ULPGC en el año 2016, este salario, con una dedicación de 20 horas semanales, asciende a 896,31€ mensuales. Lo que se aproxima a una retribución de 11,20 €/h. Este TFG tal como comprende el Proyecto Docente ha conllevado 300 horas, por lo que se calcula el coste total por tiempo empleado en:

Por lo tanto, el trabajo tarificado por tiempo empleado asciende a la cantidad de tres mil trescientos sesenta euros (3.360,00€).

### Pr.4. Redacción del trabajo

Se ha utilizado la Ecuación Pr.2 para determinar el coste asociado a la redacción de la memoria del presente TFG.

$$R = 0,07 \cdot P \cdot C_n$$

*Ecuación Pr. 2*

Donde:

- $R$  son los honorarios por la redacción del trabajo.
- $P$  es el presupuesto.
- $C_n$  es el coeficiente de ponderación en función del presupuesto.

El valor del presupuesto  $P$  se calcula sumando los documentos del trabajo tarificado por tiempo empleado y de la amortización del inmovilizado material, tanto hardware como software. El resultado de los costes se muestra en la Tabla Pr.2.

Concepto	Coste (€)
Trabajo tarificado por tiempo empleado	3.360,00
Amortización del material	172,30



Total	3.532,90
-------	----------

*Tabla Pr.2. Coste asociado a la redacción de la memoria.*

Como el coeficiente de ponderación  $C_n$  para presupuestos menores de 30.050,00€ viene definido por el COITT con un valor de 1.00, el coste derivado de la redacción del TFG es de:

$$R = 0,07 \cdot 3.532,90 \cdot 1 = 247,26 \text{ €}$$

Ascendiendo de esta forma el coste de la redacción del trabajo a doscientos cuarenta y siete con veintiséis céntimos (247,26€).

### **Pr.5. Material fungible**

No se contempla ningún gasto por edición de documentos ni por gastos de material de oficina, por lo que el coste asociado al material fungible es de cero euros (0 €).

### **Pr.6. Derechos de visado del COITT**

El COITT establece que, para proyectos técnicos de carácter general, los derechos de visado para 2016 se calculan en base a la Ecuación Pr.3:

$$V = 0,006 \cdot P_1 \cdot C_1 + 0,003 \cdot P_2 \cdot C_2$$

*Ecuación Pr.3*

Donde:

- V es el coste de visado del trabajo.
- $P_1$  es el presupuesto del proyecto.
- $C_1$  es el coeficiente reductor en función del presupuesto.
- $P_2$  es el presupuesto de ejecución material correspondiente a la obra civil.
- $C_2$  es el coeficiente reductor en función a  $P_2$ .

El valor del presupuesto  $P_1$  se halla sumando los costes de las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del documento. Esta suma se muestra en la Tabla Pr.3. Al igual que en el caso

anterior, el coeficiente  $C_1$  para proyectos de presupuesto inferior a 30.050,00€ es de 1,00, asimismo el valor de  $P_2$  es de 0,00€ ya que no se realiza ninguna obra.

Concepto	Coste (€)
Trabajo tarifado por tiempo empleado	3.360,00
Amortización del material	172,30
Redacción del trabajo	238,47
Total	3.770,77

*Tabla Pr.3. Presupuesto  $P_1$ .*

De esta forma, aplicando a la Ecuación Pr.3 los datos de la Tabla P.4 y el coeficiente especificado se obtiene:

$$V = 0,006 \cdot 3.770,77 \cdot 1 = 22,62 \text{ €}$$

Los costes por derechos de visado del presupuesto ascienden a veintidós euros con sesenta y dos céntimos (22,62 €).

### **Pr.7. Gastos de tramitación y envío**

Los gastos de tramitación y documentos están estipulados en seis euros (6,00 €) por cada documento visado de forma telemática.

### **Pr.8. Aplicación de impuestos y coste total**

El presupuesto total del presente TFG está gravado por el *Impuesto General Indirecto Canario (IGIC)*, que está establecido en la actualidad en un siete por ciento (7 %). El coste total de este TFG se encuentra desglosado en la Tabla Pr.4.

Descripción	Subtotal (€)
Amortización de materiales	172,30

Trabajo tarifado por tiempo empleado	3.360,00
Redacción del trabajo	238,47
Costes de material fungible	0
Derechos de visado del COITT	21,87
Gastos de tramitación y envío	6,00
Suma	3.798,64
IGIC (7%)	265,90
TOTAL	4.064,54

*Tabla Pr.4. Coste total.*

El presupuesto total del TFG *Aplicación web multimedia con sistema de anotación del contenido de las intervenciones usando la tecnología Blockchain* asciende a: cuatro mil sesenta y cuatro con cincuenta y cuatro céntimos (4.064,54 €).

Fdo.: Rubén Delgado González

En Las Palmas de Gran Canaria a 25 de junio de 2021



# Anexos

---



# Anexo A. Herramientas utilizadas

---

En este capítulo se presentan las diferentes herramientas utilizadas en el desarrollo del TFG, entre las cuales se incluyen servicios en la Nube, bibliotecas y *frameworks* de código, entre otros. A continuación, se describen las principales tecnologías utilizadas para el desarrollo, despliegue y testeo del servidor y la aplicación web desarrollada, así como de la *blockchain* implementada.

## **A1. JavaScript**

*JavaScript* [38] es un lenguaje de programación ampliamente conocido en el desarrollo web, principalmente por su uso como lenguaje de programación para estilizar y programar el comportamiento e interfaz de las páginas web en el lado del cliente. Sin embargo, también se utiliza en otros entornos fuera del navegador web como es *Node.js* [39]. De hecho, en este proyecto es el lenguaje de programación utilizado para el desarrollo de la lógica del servidor web, así como para la interfaz del usuario.

Entre las características que ofrece JavaScript cabe a destacar que es un lenguaje de programación basado en prototipos [40] y multiparadigma [41], lo que permite que *JavaScript* pueda funcionar como un lenguaje de programación imperativo, lo cual permite realizar *Programación Orientada a Objetos (OOP)*, así como lenguaje de programación declarativo o funcional.

## **A2. Node.js**

*Node.js* es un entorno de ejecución de código abierto utilizado para la ejecución de código de *JavaScript* y creado por los creadores de dicho lenguaje de programación para permitir su ejecución en un entorno fuera del navegador como si se tratara de una aplicación independiente. Para realizar la ejecución de código de *JavaScript*, *Node.js* utiliza el motor de ejecución V8 [42] creado por Google, que convierte el código *JavaScript* en un código máquina que el computador ejecuta más rápido.

El modelo de *Node.js* es de entrada y salida sin bloqueo controlado por eventos [43], los cuales pueden ser desde solicitudes HTTP hasta lecturas de archivos locales. Este modelo, completamente controlado por eventos, permite el manejo simultáneo de peticiones, lo cual lo convierte en una plataforma idónea para la creación de aplicaciones de red rápidas y eficientes.



### A3. React

*React* [44] es una biblioteca de *JavaScript* para desarrollar interfaces de usuario en aplicaciones web. Fue creado por Facebook en 2013 como una biblioteca de código abierto y se ha convertido en una de las 10 bibliotecas más descargadas de *JavaScript* [45].

El aspecto más importante de *React* es la creación de componentes, los cuales son elementos HTML personalizables que se pueden reusar para crear interfaces de usuario de manera rápida y eficiente. Para realizar interfaces de usuario complejas, *React* permite crear componentes encapsulados que manejan su propio estado, así como pasar su estado a otros componentes mediante el uso de *props* [46].

En la Figura A.1 se muestra el componente *Meetings*. Dicho componente tiene encapsulados a su vez otros dos componentes, *HeaderText* y *MeetingsTable*, a los cuales le pasa diferente información (como las variables *meetings*, *isLoading* y *error*) a través de *props* (entradas arbitrarias de datos que se pueden pasar a componentes).

```
const Meetings = () => {
  const [
    meetings,
    isLoading,
    error,
    fetchMeetings
  ] = useFetch('/meetings')

  useEffect(() => {
    fetchMeetings();
  }, [])

  return (
    <div className="main">
      <HeaderText>Reuniones realizadas</HeaderText>
      <p>
        Seleccione una reunión para consultar las intervenciones
        realizadas en dicha intervención
      </p>
      <MeetingsTable
        loading={isLoading}
        error={error}
        meetings={meetings}/>
    </div>
  )
}
```

Figura A.1. Ejemplo de componente en React

Ofrece el uso de *Hooks* [47] (es una característica de React que permite extraer la lógica de estado de un componente para que pueda ser reutilizado en otros componentes) que pueden tener múltiples tipos por defecto (como *useState* [48], que permite añadir estado a un componente) o personalizados para que puedan ser utilizados en diferentes componentes.

En la Figura A.2 se muestra un *Hook* personalizado denominado *useDownload*. Este hace uso del *Hook* *useState* para declarar diferentes variables de estado (*data*, *loading* y *error*), y tiene la función *load*, que una vez invocada, permite descargar un archivo dada una URL que se le pasa por parámetros a dicho *Hook*.

```
const useDownload = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(false);

  const load = async () => {
    setLoading(true);

    try {
      const config = {
        responseType: 'blob'
      };

      const response = await axios.get(url, config);
      console.log(response.data)
      setData(response.data);
    } catch (e) {
      console.error('[useDownload] Error doing fetch', e.response)
      setError(e.response.data);
    } finally {
      setLoading(false);
    }
  }

  return [data, loading, error, load];
}
```

Figura A.2. Ejemplo de hook personalizado para descargar un archivo

## A4. Redis

*Remote Dictionary Server (Redis)* [49] es una base de datos de valores de clave almacenadas en memoria principal y de código abierto. Incorpora un conjunto de estructuras de datos en memoria que permiten diferentes funcionalidades como el almacenamiento en caché, administración de sesiones o gestor de cola de mensajes [50].

En nuestro caso, para la comunicación de la red *blockchain*, se ha utilizado *Redis* para implementar el patrón de diseño de software que maneja mensajes *Publisher/Subscriber* [51]: un sistema de mensajería asíncrona que permite la entrega de eventos y mensajes entre las máquinas que hacen uso de este servicio.

Por otro lado, en la aplicación web desarrollada, se ha utilizado *Redis* para la gestión de tareas en segundo plano mediante un sistema de colas que permite al servidor de la aplicación web delegar trabajos de cierta carga computacional que podrían bloquear al servidor (cálculo de *hashes*, envío de emails...) a un servidor secundario que se encarga de realizar estos trabajos.

Para la interacción con el servidor *Redis* desde el proyecto, se ha usado la biblioteca *node-redis* [52] que ofrece una interfaz fácil de usar para comunicarse con el servidor especificado. Asimismo, para la gestión de colas para las tareas en segundo plano se han utilizado las bibliotecas *bull* [53], que permiten crear colas para tareas asíncronas usando el servidor *Redis*.

## A5. Heroku

Heroku [54] es un *Platform-as-a-Service (PaaS)* [55] que permite desplegar aplicaciones en la Nube. Su principal característica es que permite desplegar fácilmente este tipo de aplicaciones, así como controlar y administrar los servidores que alojan estas aplicaciones a través de una interfaz de usuario (Figura A.3) de forma sencilla. Esto permite que el desarrollador no tenga que preocuparse de la infraestructura para alojar su aplicación en la nube, sino que puede centrarse directamente en el desarrollo de su aplicación.

Para el alojamiento de aplicaciones en la Nube, Heroku ofrece una colección de contenedores escalables y ligeros de Linux denominados *Dynos*, donde el desarrollador no

se tiene que preocupar de los detalles internos de dichos contenedores. Con el plan gratuito, se puede hacer uso de los *Free Dynos*, que permiten el despliegue de múltiples tipos de aplicaciones web con ciertas limitaciones de uso y almacenamiento, aunque existen otros tipos de *Dynos* de pago que ofrecen diferentes tipos de ventajas según las necesidades del proyecto [56].

En conjunción con los *Dynos*, Heroku ofrece múltiples servicios y herramientas que se pueden añadir a las instancias de los *Dynos* para proporcionar más funcionalidades a las aplicaciones desplegadas. Estos servicios se denominan *Heroku Add-ons* [57] y proporcionan desde bases de datos hasta sistemas de mensajería y colas para las aplicaciones. En nuestro caso, se ha utilizado *Heroku Redis* [58], el cual es un *add-on* que permite hacer uso de un servidor *Redis* desde el *Dyno* desplegado.

## A6. Firebase

*Firebase* [59] es un *Backend-as-a-Service (BaaS)* [60] desarrollado por Google para proporcionar diferentes servicios en la Nube para facilitar el desarrollo de aplicaciones web y móviles. Su función esencial es proporcionar diferentes herramientas que permitan acelerar el desarrollo de la aplicación, así como varias utilidades para analizar y monetizar aplicaciones [61]. En la figura A.4 se muestra la interfaz de usuario que ofrece para administrar los diferentes servicios que proporciona.

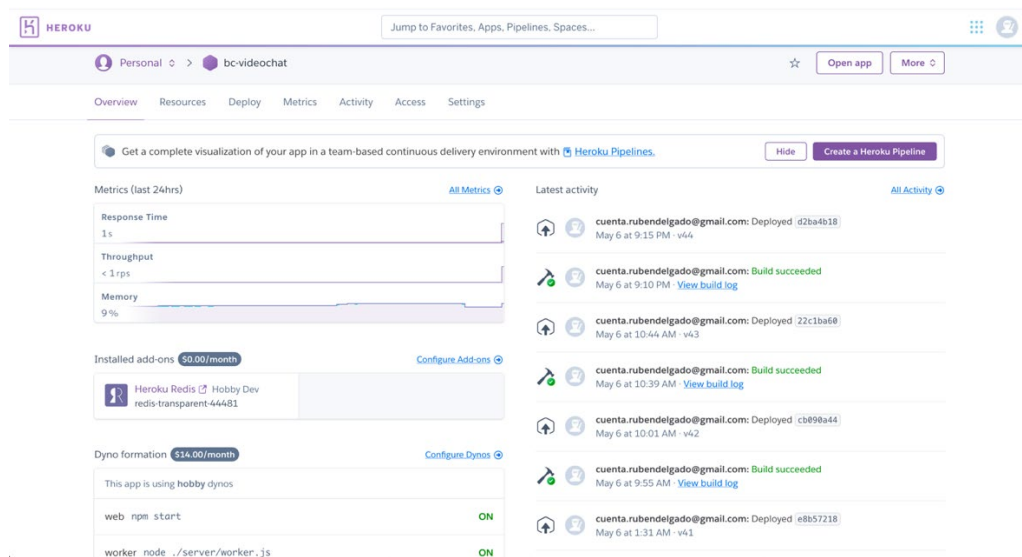


Figura A.3. Interfaz de usuario de Heroku

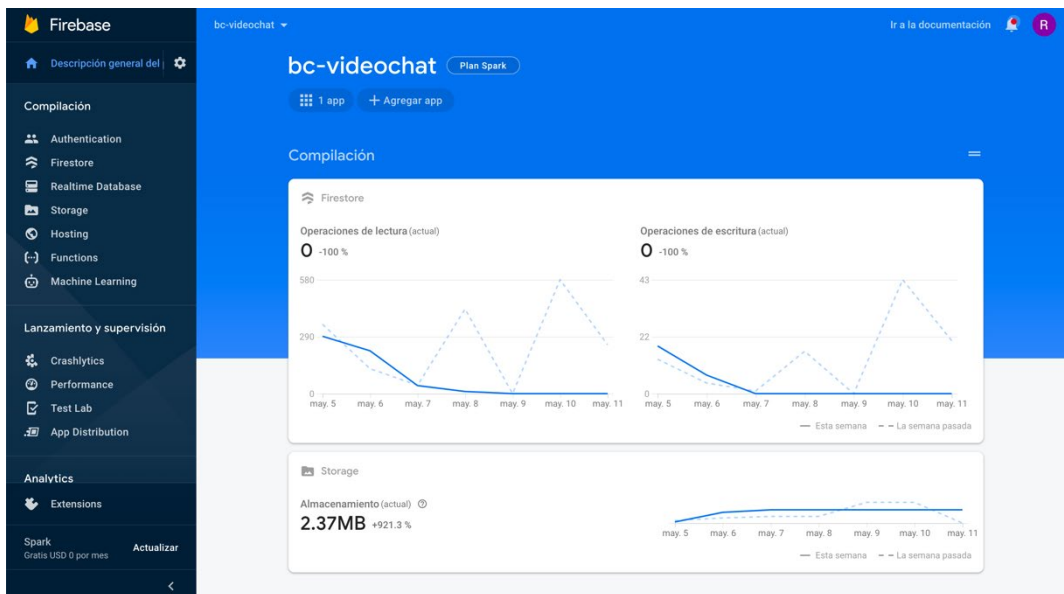


Figura A.4 Interfaz de usuario de Firebase

Ofrece una gran variedad de servicios en la Nube; nosotros hemos utilizado los servicios descritos a continuación:

- *Cloud Firestore*: es una base de datos no relacional, flexible y escalable, en la Nube para el almacenamiento y sincronización de datos [62]. Para asegurar la seguridad de los datos almacenados, permite establecer reglas personalizadas de lectura y escritura en la base de datos utilizada.

En la Figura A.5 se muestra un ejemplo del uso del SDK de *Firebase* para hacer uso de la API de *Cloud Firestore* en *Node.js*. En el código mostrado, se realiza una consulta a los datos de una reunión a través del servicio descrito.

```
/**
 * Consulta en Firestore la información de una reunión
 * @param {String} meetingID - ID de la reunión
 *
 * @return {meeting} La información de la intervención
 */
const getMeetingInfo = async (meetingID) => {
  const meetingDoc = await db.collection('meetings').doc(meetingID).get()
  return meetingDoc.data()
}
```

Figura A.5. Ejemplo de consulta a Firestore en Node.js

- *Firebase Authentication*: para la autenticación de usuarios proporciona servicios de identificación de usuarios mediante SDK para diferentes tipos de aplicaciones web y móviles [63]. Este servicio permite el registro tradicional de usuarios (mediante un email y contraseña) como un acceso alternativo mediante el inicio de sesión en plataformas externas
- *Cloud Storage*: es un sistema de almacenamiento de datos, el cual permite guardar y sincronizar archivos planos y archivos multimedia de cualquier tipo en la Nube [64].

En la Figura A.6 se muestra un ejemplo del uso de *Cloud Storage* en Node.js. En el código mostrado, se sube a *Cloud Storage* el audio de una intervención. Una vez subido, utilizando el enlace de descarga que proporciona *Cloud Storage*, se puede crear la información de la intervención.

## A7. Postman

Postman [65] es una herramienta de desarrollo de software que permite probar, consumir y depurar a las API. Para ello proporciona una interfaz de usuario intuitiva (Figura A.7) que permite personalizar las peticiones realizadas a nuestra API, modificando el cuerpo, encabezado y parámetros de la petición que sean necesarios.

Además de la realización de peticiones personalizadas a las API, permite gestionar y crear documentación basadas en la API probada en la aplicación, así como crear servidores de pruebas para las API antes de que éstas hayan sido desarrolladas [66].

```
storageRef.put(audio)
  .then((snapshot) => {
    snapshot.ref.getDownloadURL().then(downloadURL => {
      console.log(name, downloadURL, meetingID, interventionID)
      createInterventionInfo(name, downloadURL, meetingID, interventionID)
    })
  })
```

Figura A.6. Ejemplo de uso de *Cloud Storage* para subir un archivo desde Node.js

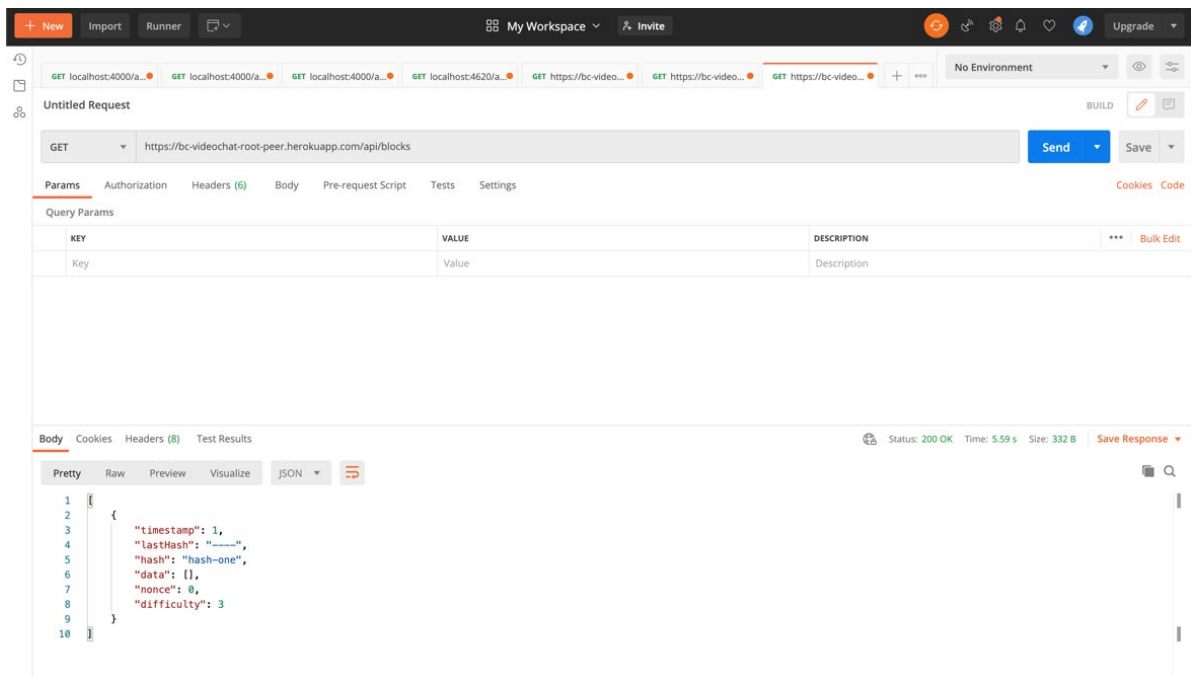


Figura A.7. Interfaz de usuario de Postman

## A8. Bibliotecas utilizadas

Para la elaboración de las diferentes partes del proyecto, hemos utilizado varias bibliotecas que ofrecen diversas funcionalidades y métodos que han permitido el desarrollo del proyecto. A continuación, se presentan las bibliotecas más importantes que se han utilizado en las diferentes partes del proyecto y para qué se han utilizado.

Para el desarrollo de la *blockchain* y el servidor de la aplicación web, hemos usado la biblioteca *Express* [67], el cual es un *framework* utilizado en Node.js para creación de servidores web. Este permite la implementación de manejadores de peticiones HTTP para los URL especificados, así como usar *middlewares* para el procesamiento de las peticiones.

Para la realización de peticiones HTTP se utiliza la biblioteca *axios* [68], tanto en los nodos de la *blockchain* desarrollados, así como en el servidor y la aplicación web. En la figura A.8 se muestra un ejemplo en *Node.js* en el que se realiza una petición HTTP a un nodo de la *blockchain* para añadir una transacción a la cadena de bloques.

La biblioteca *elliptic* [69] permite usar algoritmos criptográficos de curva elíptica. En nuestro caso, se utiliza para la creación de claves criptográficas y el proceso de verificación

de firmas (Figura A.9) en la cadena de bloques. Esta biblioteca ofrece diferentes tipos de curvas, aunque en este proyecto se utiliza la *secp256k1*. En la Figura A.9 se muestra la función *verifySignature* que, haciendo uso de la biblioteca *elliptic*, se utiliza para verificar una firma dada, sabiendo la clave pública correspondiente a clave privada que ha realizado la firma, los datos que se han firmado y la firma creada.

Para el servidor web de la aplicación, la biblioteca *cors* [70] es un middleware para Express que permite habilitar el Intercambio de *Cross Origin Resource Sharing (CORS)* [71], el cual consigue que se puedan solicitar recursos en una aplicación web desde un dominio diferente.

La biblioteca *crypto* [72] es un módulo integrado en *Node.js* que provee de múltiples funcionalidades y algoritmos criptográficos. En el proyecto, se utiliza en la aplicación web y la *blockchain* para el proceso de generación de *hash* (Figura A.10) con el algoritmo SHA-256. En la Figura A.10 se muestra la función *cryptoHash*, la cual utiliza biblioteca para la generación de un *hash* dados una serie de argumentos arbitrarios.[72] es un módulo integrado en *Node.js* que provee de múltiples funcionalidades y algoritmos criptográficos. En el proyecto, se utiliza en la aplicación web y la *blockchain* para el proceso de generación de *hash* (Figura A.10) con el algoritmo SHA-256. En la figura A.10 se muestra la función *cryptoHash*, la cual utiliza biblioteca para la generación de un *hash* dados una serie de argumentos arbitrarios.

```
payload = {
  videoAppID,
  interventionID,
  interventionToken,
  signature
}

try {
  await axios.post(`${BLOCKCHAIN_ROOT_URL}/api/transact`, payload)
} catch (e) {
  console.error(e);
}
```

Figura C.8. Ejemplo de código usando de la biblioteca axios



```

const verifySignature = ({publicKey, data, signature}) => {
  const keyFromPublic = ec.keyFromPublic(publicKey, 'hex')
  return keyFromPublic.verify(data, signature);
}

```

Figura A.9. Ejemplo de código usando la biblioteca *elliptic*

```

const cryptoHash = (...inputs) => {
  const hash = crypto.createHash('sha256')
  hash.update(inputs.map(input => JSON.stringify(input)).sort().join(' '))

  return hash.digest('hex')
}

```

Figura A.10. Ejemplo de uso de la biblioteca *crypto*

Las bibliotecas *firebase-admin-node* [73] y *firebase-js-sdk* [74] proporciona el SDK para usar los servicios de *Firebase* desde el servidor *Node.js* y la aplicación web, respectivamente. En nuestro caso, estas bibliotecas nos permiten usar los servicios de *Firebase Authentication*, *Cloud Firestore* y *Cloud Storage* con JavaScript de manera sencilla. [75] y *firebase-js-sdk* [74] proporciona el SDK para usar los servicios de *Firebase* desde el servidor *Node.js* y la aplicación web, respectivamente. En nuestro caso, estas bibliotecas nos permiten usar los servicios de *Firebase Authentication*, *Cloud Firestore* y *Cloud Storage* con JavaScript de manera sencilla.

Para la programación de ejecución de funciones en fechas establecidas, se utiliza la biblioteca *node-schedule* [75]. En el servidor web, esta biblioteca permite programar cuando una intervención realizada debe ser validada de forma automática transcurrido un periodo de tiempo determinado.

En la Figura A.11, se muestra la función *scheduleJobToValidate* crea una tarea programada para la validación automática de la intervención. Para ello, hace uso de la función *scheduleJob* que, dado el ID de la tarea y la fecha en la que se debe ejecutar, ejecuta el código necesario para actualizar el estado de la intervención.

```

/**
 * Crea una tarea en segundo plano que se ejecuta un tiempo determinado después
 * de una intervención, para validar en caso de desistimiento positivo
 *
 * @param {String} - ID de la intervención
 */
const scheduleJobToValidate = (jobID) => {
  console.log('creando tarea secundaria', jobID)

  //Se crea la fecha en la que se debe ejecutar la tarea
  const endDate = new Date()
  endDate.setDate(endDate.getDate() + constants.DAYS_BEFORE_VALIDATION_WITHDRAWAL)

  //Se crea el trabajo programado y la operación que debe realizar
  schedule.scheduleJob(jobID, endDate, () => {
    console.log('Intervención a validar por desistimiento positivo', jobID);
    interventionService.updateInterventionState(jobID, 'ACCEPTED')
  });
}

```

Figura A.11. Ejemplo de uso de la biblioteca *node-schedule*

La biblioteca *nodemailer* [76] permite realizar envíos de correo electrónicos desde *Node.js*, lo cual se utiliza en el proyecto para notificar a los usuarios de las reuniones a las que han sido invitados, así como para la notificación de intervenciones pendientes de validación por parte del usuario.

Para la comunicación bidireccional en tiempo real entre el cliente y el servidor web, se utiliza la biblioteca *socket-io* [77]. Para ello, proporciona al desarrollador una capa de abstracción que permite establecer fácilmente un canal de comunicación entre el cliente y el servidor (utilizando mecanismos como WebSockets [78], *Asynchronous JavaScript And XML (AJAX)* [79]...). En este proyecto, se utiliza para el proceso de señalización de WebRTC.

En la Figura A.12 se muestra un *listener*, que escucha el evento *disconnect*, el cual ocurre cuando un nodo cierra la conexión. Si esto ocurre, se ejecuta el código para eliminar el nodo de la sala de la reunión multimedia. Por otro lado, en la Figura A.13 se muestra el código necesario para emitir un evento *all users*, adjuntando al evento el contenido de la variable *usersInThisRoom* que contiene los identificadores de los usuarios conectados en la sala.

La biblioteca *uuid* [80] se utiliza para la creación de identificadores de datos mediante el *Request for Comments (RFC) 4122 Universally Unique Identifier (UUID)* [81].

```

socket.on('disconnect', () => {
  console.log(`EVENT: socket ${socket.id} has disconnected`)
  const roomID = socketToRoom[socket.id];
  let room = users[roomID];
  if (room) {
    room = room.filter(id => id !== socket.id);
    users[roomID] = room;
  }
});

```

Figura A.12. Ejemplo de listener usando la biblioteca socket.io

```

const usersInThisRoom = users[roomID].filter(id => id !== socket.id);

console.log('List of connected users:', usersInThisRoom)
socket.emit("all users", usersInThisRoom);

```

Figura A.13. Ejemplo de envío de mensajes usando la biblioteca socket.io

A continuación, se indican las bibliotecas más importantes que se han utilizado en el proyecto, para implementar el cliente web:

- *Ant-design* [82]: es una biblioteca de interfaz de usuario creada para *React*, la cual contiene componentes visuales (Figura A.14) de gran calidad para la creación de interfaces de usuario interactivas.
- *React-router* [83]: permite navegar entre las diferentes vistas de la aplicación web.
- *Use-media-recorder* [84]: permite grabar el contenido multimedia del cliente desde el navegador. Para ello, proporciona al desarrollador una capa de abstracción para utilizar la API *MediaRecorder* [85].
- *ReactAudioPlayer* [86]: permite la reproducción de audio en *React*.

**VIDEO**  
Intervenciones

Inicio

Reuniones

Crear una reunión

Intervenciones

Validar

Consultar

Cerrar sesión

## Intervenciones realizadas en: Reunión de prueba 6 de mayo de 2021

**Información de la reunión**

ID de la reunión	4dfbbc4a-7527-4fed-9647-93bfec6a3db6	Mecanismo de consenso	Votación100%
------------------	--------------------------------------	-----------------------	--------------

Mis intervenciones [Intervenciones validadas](#)

Nombre	ID de la intervención	Fecha	Autor	Estado	Ver contenido multimedia	Verificar
Reunon 6 de mayo comunidad de Vecinos	1ca114ad-51cc-40fb-a988-ca8035d1dddb	6/5/2021 11:47:33	alvaro.suarez@ulpgc.es	VALIDADO	<a href="#">Abrir enlace</a>	<a href="#">Verificar</a>

Usuario	Estado
alvaro.suarez@ulpgc.es	● Aceptado
cuenta.rubendelgado@gmail.com	● Aceptado

< 1 >

*Figura A.14. Ejemplo de interfaz de usuario realizada usando componentes visuales de la biblioteca Ant-design*

# Anexo B. WebRTC

---

En este anexo se presentan los conceptos fundamentales para entender el funcionamiento de WebRTC para la comunicación multimedia.

## B1. Introducción a WebRTC

*Web Real-time Communications (WebRTC)* [87] es un *framework* de código abierto que permite el uso de funcionalidades de comunicación en tiempo real a través de Internet. Fue creado como un proyecto de código abierto por Google en 2011 para generar una *Application Programming Interface (API)* que permitiera a aplicaciones de navegador web intercambiar información multimedia entre pares con baja latencia y de manera eficiente [88]. Actualmente, los navegadores más populares como Firefox, Chrome o Safari soportan el uso de WebRTC sin la necesidad de una instalación previa de software adicional, lo cual ha permitido estandarizar el uso de esta tecnología.

Para ello, WebRTC ofrece bloques principales para las diferentes funcionalidades que permitan el intercambio y recogida de información multimedia desde el navegador:

- *getUserMedia*: proporciona las herramientas para recoger la información multimedia del usuario desde el navegador (micrófono, capturas de pantalla, cámara...). El acceso al micrófono y la cámara han de ser concedidos por el usuario y, cuando estén funcionando, deben estar indicados explícitamente.
- *RTCPeerConnection*: proporciona el soporte para establecer la conexión entre dispositivos emparejados (*pares*) los cuales son los clientes que realizarían la comunicación multimedia.
- *RTCDataChannel*: ofrece un canal bidireccional para el intercambio de información arbitraria entre pares, la cual no tiene por qué ser un flujo (*stream*) multimedia.

Previo a la comunicación multimedia, WebRTC usa el protocolo *Interactive Connectivity Establishment (ICE)* [89] para establecer la conectividad entre los pares que realicen la comunicación, determinando las rutas posibles para que dichos pares puedan realizar una comunicación directa.

## B2. Captura de contenido multimedia en el navegador

Para la captura del flujo multimedia del usuario, WebRTC ofrece la API *getUserMedia* [90]. Esta API proporciona la interfaz *MediaStream* que representa los flujos multimedia, los

cuales pueden ser de entrada (recibidos por otro par) o de salida (que se recogen de forma local, para ser transmitidos).

Dicha interfaz está compuesta por uno o más objetos *MediaStreamTrack*, que representa el flujo de datos y control multimedia específico de audio o vídeo. Cabe destacar que todos los objetos *MediaStreamTrack* dentro de un *MediaStream* están sincronizados para su correcta reproducción. A su vez, *MediaStreamTrack* está compuesto por uno o varios canales, los cuales es la unidad de *flujo* multimedia más pequeña considerada por la API. En la Figura B.1 se muestra un ejemplo común de los elementos de un *MediaStream*. Está formado por dos *MediaStreamTracks*, uno para vídeo y otro para audio, en el que el audio está compuesto por dos canales, dado que se trata de un audio estéreo.

Desde el punto de vista del desarrollador, para hacer uso de la API descrita se utiliza la función *getUserMedia* (Figura B.2), a la cual hay que indicarle por parámetros el contenido multimedia que se capturaría desde el navegador. Para ello, se utiliza un objeto *MediaStreamConstraints* (Figura B.3) donde se indica si se debe capturar el audio o el vídeo del usuario.

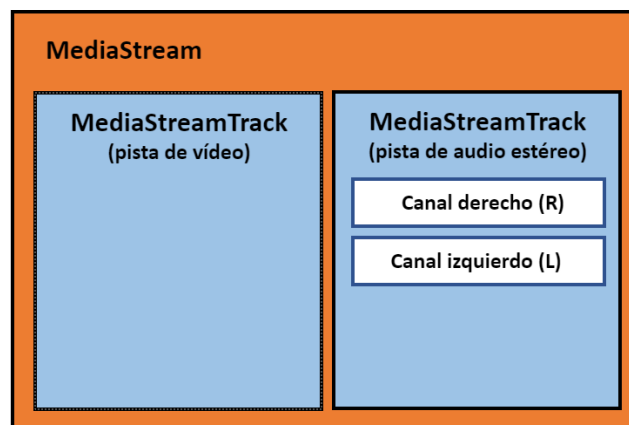


Figura B.1. Estructura del contenido multimedia en WebRTC

```
let stream
try {
  stream = await navigator.mediaDevices.getUserMedia(mediaConstraints)
} catch (error) {
  console.error('Could not get user media', error)
}
```

Figura B.2. Ejemplo de código utilizando *getUserMedia*

```
const mediaConstraints = {  
  audio: true,  
  video: true,  
}
```

Figura B.3. Ejemplo de objeto *MediaStreamConstraints*

Para el usuario, se muestra una petición desde el navegador donde se solicitan permisos para usar un dispositivo de entrada de vídeo y/o audio, dependiendo de los valores del objeto *MediaStreamConstraints* utilizado en *getUserMedia*. En la Figura B.4 se muestra un ejemplo de la solicitud de la captura del contenido multimedia en *Firefox*. Posteriormente, si el usuario acepta la petición, la función *getUserMedia* devuelve un objeto *MediaStream*, que contiene el flujo multimedia solicitado del usuario.

### B3. Establecimiento de conexión

WebRTC no utiliza un modelo cliente/servidor, sino que establece conexiones *Peer-to-Peer* (P2P). En una conexión P2P o de igual a igual, la comunicación se realiza de forma directa entre los dos usuarios. Las ventajas de este tipo de conexión, frente a la arquitectura cliente/servidor, es su capacidad de conectar de forma directa a los pares involucrados en la comunicación, lo cual reduce la latencia y coste del ancho de banda, ya que, al tratarse de una conexión directa, no son necesarios servidores intermedios que retransmitan los datos [91].

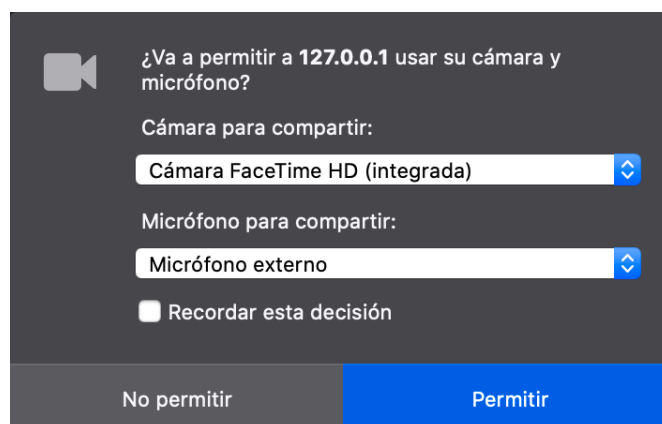


Figura B.4. Solicitud de permisos para capturar las entradas de vídeo y audio en Firefox



Sin embargo, para poder realizar la conexión de igual a igual, los pares involucrados deben intercambiar previamente información necesaria al otro cliente para poder establecer la conexión. El proceso de intercambio de dicha información se denomina señalización [92], en el que los clientes, antes de comunicarse de igual a igual, intercambian dicha información de señalización a través de un servidor, el cual es el responsable de conseguir que la información de señalización llegue a los pares involucrados, los cuales se podrían encontrar en redes diferentes (Figura B.5). Una vez que se consigue realizar correctamente, los pares realizan la comunicación P2P y transmiten el flujo multimedia de forma directa.

No obstante, establecer una conectividad de igual a igual puede ser complicado, ya que comúnmente los clientes se encuentran en redes diferentes, en las que se utilizan puntos de acceso y cortafuegos para acceder a fuera de la red. Dichos firewalls suelen incorporar el protocolo *Network Address Translation (NAT)* [93], el cual se utiliza para traducir direcciones y puertos privados (de los dispositivos que se encuentran dentro de la red) a direcciones públicas que se utilizarían para transmitir datos fuera de la red. Sin embargo, dependiendo de la configuración NAT establecida en la red, establecer una comunicación igual a igual no podría realizarse.

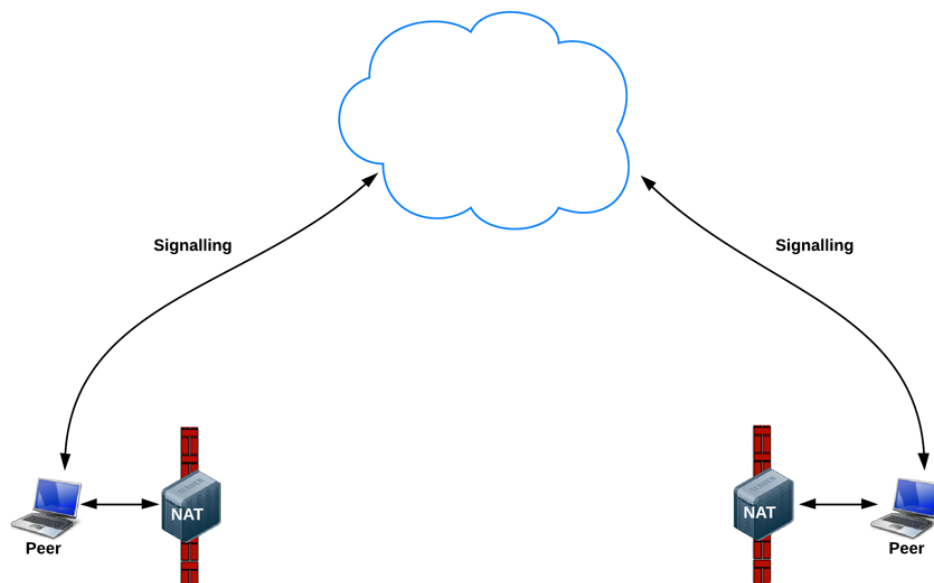


Figura B.5 Diagrama de proceso de señalización

Fuente: <https://temasys.io/ice-and-webrtc-what-is-this-sorcery-we-explain/>

Para superar este problema, WebRTC utiliza el protocolo ICE. En este, los clientes que establecerían la conexión utilizando dicho protocolo tienen un agente ICE, el cual es el encargado de gestionar el proceso establecimiento de la conexión. Cada agente ICE genera una serie de candidatos ICE, en el que cada candidato posee una posible dirección que se puede utilizar para conseguir establecer la comunicación entre los clientes [94]. Sin embargo, para poder superar los retos que plantean los diferentes tipos de NAT que podrían estar siendo utilizadas en las redes donde se encuentran los clientes, ICE utiliza diferentes tecnologías, entre los que cabe destacar el *Session Traversal Utilities for NAT (STUN)* [95] y el *Traversal Using Relays around NAT (TURN)* [96].

### **B3.1. Servidores TURN y STUN**

Para establecer la comunicación entre pares, es necesario que ambos conozcan la dirección *Internet Protocol (IP)* pública del par con el cual se van a comunicar. Sin embargo, es muy común que los pares se encuentren en redes locales donde el punto de acceso utiliza el protocolo NAT para la traducción de direcciones IP privadas a una dirección pública.

El protocolo STUN permite que un nodo pueda conocer su dirección IP pública utilizando un servidor STUN. Para ello, mediante una petición desde el par, el servidor STUN recoge la dirección IP y el puerto de dicha petición (que ha sido traducida usando NAT por el Gateway de la red donde se encuentra el par) y devuelve dicha dirección IP, de forma que el par, conociendo su dirección IP pública, puede enviarla al otro par para que sepa el par al que debe conectarse. Como se muestra en la Figura C.6, durante la realización del proceso de señalización, el cliente realizaría una consulta al servidor STUN para conocer su dirección IP pública para que pueda ser comunicada al otro cliente.

Sin embargo, si el par se encuentra en una red privada, dependiendo de la configuración NAT o del cortafuegos, no es posible hacer uso del protocolo STUN. Por ello, se usa el protocolo TURN. Los servidores TURN funcionan como *relays* del flujo multimedia, redireccionando la comunicación a través de ellos. Para ello, el servidor TURN permite a un host detrás de un NAT obtener una dirección IP y un puerto público de un servidor de retransmisión que reside en Internet.

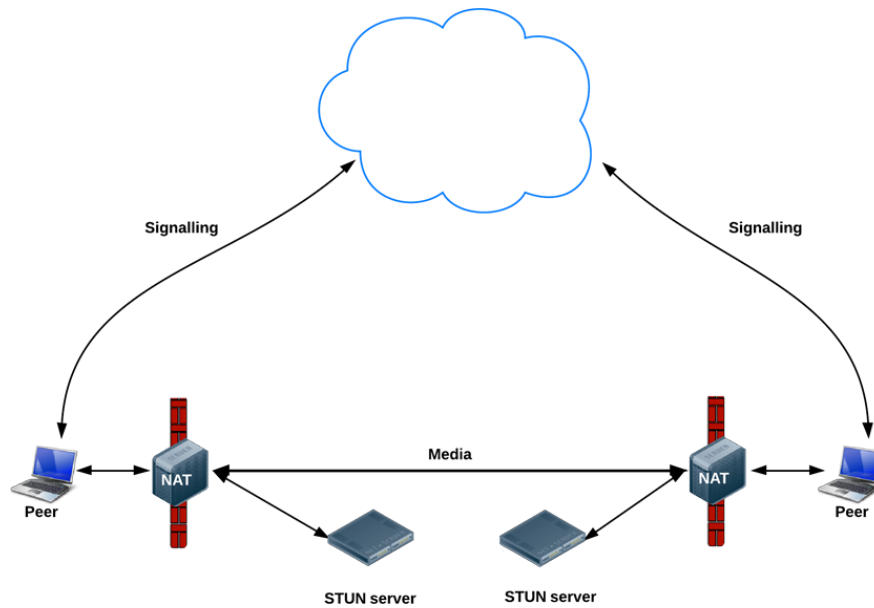


Figura B.6 Diagrama de conexión utilizando servidores STUN

Fuente: <https://temasys.io/ice-and-webrtc-what-is-this-sorcery-we-explain/>

El uso de los servidores TURN funcionan como una segunda opción si no es posible realizar la conexión entre los clientes utilizando los servidores STUN. Como se muestra en la Figura B.7, durante el proceso de señalización, en primer lugar, se intenta realizar la conexión utilizando los servidores STUN. En caso de no funcionar, se utilizarían servidores TURN para encaminar el tráfico multimedia entre los clientes.

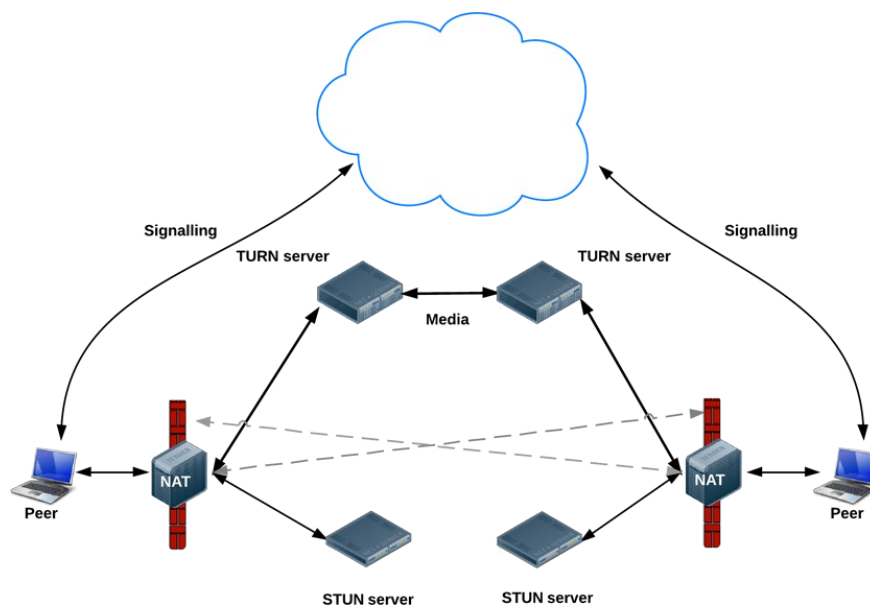


Figura B.7 Diagrama de conexión utilizando servidores TURN

Fuente: <https://temasys.io/ice-and-webrtc-what-is-this-sorcery-we-explain/>

Sin embargo, el uso de servidores TURN es mucho más costoso, ya que requieren de más ancho de banda y la latencia resultante es mayor ya que se debe pasar todo el tráfico de la comunicación a través de dichos servidores. Al contrario que los servidores TURN, para utilizar el protocolo STUN para solo es necesario realizar una petición al servidor STUN y posteriormente, una vez establecida la conexión, el tráfico multimedia se envía de forma directa a los clientes.

### **B3.2. RTCPeerConnection**

Para el establecimiento de conexión entre dos clientes, WebRTC ofrece la clase *RTCPeerConnection*. Esta se encarga del establecimiento y gestión de la comunicación para el intercambio de información multimedia. Cada objeto *RTCPeerConnection* representa una conexión entre dos clientes, por lo que, por cada canal bidireccional multimedia establecido, debe haber un objeto *RTCPeerConnection* que gestione dicha conexión.

*RTCPeerConnection* utiliza el protocolo ICE, junto a los servidores STUN y TURN para permitir que el tráfico multimedia pueda transmitirse entre los clientes involucrados en la comunicación, incluso si se encuentran en redes privadas con diferentes configuraciones NAT y cortafuegos.

## **B4. Proceso de señalización**

Tal y como se ha comentado en el apartado B.2 Establecimiento de conexión, para poder establecer una conexión directa entre dos clientes, se necesita que estos puedan intercambiar diferente información requerida para establecer la conexión entre los agentes ICE de cada cliente. Este intercambio de información se realiza durante el proceso de señalización, previo a la comunicación multimedia, donde se utiliza un servidor, denominado servidor de señalización, el cual retransmite los mensajes de señalización entre los clientes.

La topología común para realizar el proceso de señalización entre dos clientes, como se muestra en la Figura B.8, se realiza a través del servidor de señalización. Posteriormente, el tráfico multimedia se transmite entre los clientes involucrados de forma directa.

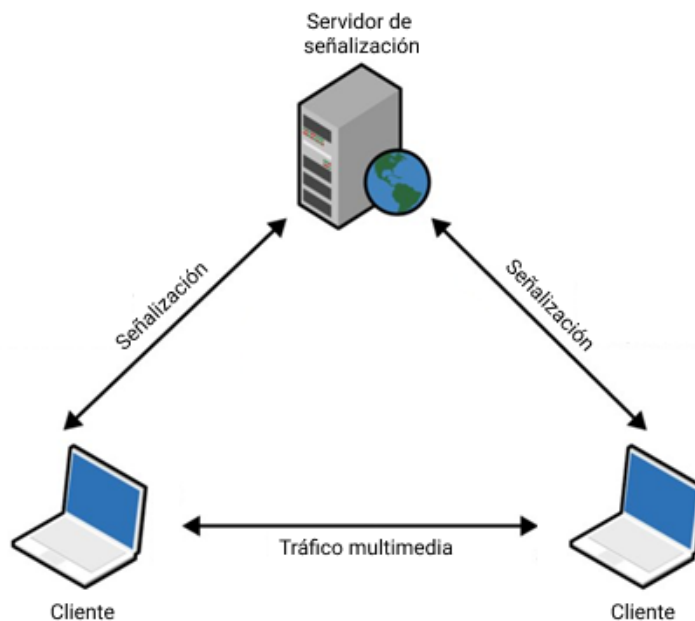


Figura B.8 Señalización en WebRTC

Adaptado de fuente: S. Loreto and S. Pietro Romano, *O'Reilly*, 9781449371838 ISBN, *Real-Time Communication with WebRTC*. 2014.

En el proceso de señalización, inicia un mecanismo de oferta y respuesta entre los pares involucrados. Con este fin, se utiliza el protocolo *Session Description Protocol (SDP)* para describir la información necesaria que deben intercambiar en dichos mensajes de oferta y respuesta para establecer la conexión. Los códecs utilizados y parámetros de conectividad (protocolos, dirección IP, candidatos ICE...) son enviados a través de dicho mecanismo, de forma que los clientes, una vez que reciben dichas ofertas o respuestas, pueden decidir qué códecs multimedia pueden enviar y recibir, y cómo y dónde enviarlos. Los métodos y protocolos usados para la transmisión de dicha información de señalización no están definidos por WebRTC, por lo que el método utilizado para transmitir dicha información queda a elección del desarrollador.

Los pasos que se realizarían durante el proceso de señalización previo a la comunicación multimedia se describen a continuación. Desde uno de los pares, se realizarían las siguientes operaciones:

1. Capturar el contenido multimedia del usuario utilizando la API *getUserMedia*.

2. Crear un objeto *RTCPeerConnection* que gestionaría la conexión y añadir el flujo multimedia capturado a dicho objeto con la función *addTrack*
3. Generar la información SDP de oferta y guardarla en el objeto *RTCPeerConnection* creado utilizando la función *setLocalDescription*.
4. Enviar dicha información SDP de oferta al par con el que se quiere establecer la comunicación
5. Recibir el mensaje de respuesta con la información SDP de dicho par y guardarla en el objeto *RTCPeerConnection* utilizando la función *setRemoteDescription*.
6. Por último, se procedería al intercambio de los candidatos ICE en el que los agentes ICE buscan la ruta óptima que se pueda establecer para realizar la comunicación multimedia.

En el par con el que se estaría comunicando se realizarían los siguientes pasos:

1. Capturar el contenido multimedia del usuario utilizando la API *getUserMedia*.
2. Una vez que se recibe el mensaje de respuesta con la información SDP del par con el que se establecería la comunicación, se crea un objeto *RTCPeerConnection* que gestionaría dicha conexión. A este objeto, se le añadiría el *stream* multimedia capturado con la función *addTrack* y la información SDP recibida utilizando la función *setRemoteDescription*.
3. Generar la información SDP de respuesta y guardarla en el objeto *RTCPeerConnection* creado utilizando la función *setLocalDescription*.
4. Por último, se realiza el intercambio de candidatos ICE.

# Anexo C. Blockchain

---

Para comprender cómo se podría integrar la tecnología *Blockchain* con el sistema de registro de intervenciones, en este apartado se pretende explicar los elementos básicos de su funcionamiento y realizar una aproximación a diferentes aspectos de esta tecnología que pueden aplicarse al problema planteado.

## C1. Definición y conceptos básicos

En la actualidad, un gran número de sistemas en Internet se encuentran centralizados o controlados por grandes compañías u organizaciones gubernamentales. Esta centralización de servicios hace que, incluso utilizando sistemas de redundancia, sean vulnerables a ataques que puedan hacer perder los datos, modificación de los datos almacenados... Por otra parte, la gestión de estos datos por parte de una única entidad plantea posibles problemas en que dicha entidad pueda manipular dichos datos sin que los usuarios sean conscientes de ello.

Las *Distributed Ledger Technologies (DLT)* ofrecen una solución para este problema, proveyendo de una base de datos descentralizada gestionada por múltiples participantes. En este aspecto, la tecnología *Blockchain* es un tipo de DLT [97].

En pocas palabras, la tecnología *Blockchain* es una base de datos descentralizada y distribuida [7], en el que los datos son gestionados por múltiples pares mediante un sistema de consenso, sin que haya una autoridad central que gestiona dicha base de datos. Dichos datos están almacenados en una secuencia de bloques, en el cual, mediante diferentes mecanismos criptográficos, cada bloque está relacionado con su anterior formando una cadena de bloques.

Las características más destacadas que diferencian la tecnología *Blockchain* de su precursor DLT son:

- *Operaciones disponibles:* en una base de datos tradicional, se permiten las operaciones *Create Read Update Delete (CRUD)*, que hacen referencia a las operaciones de escritura, lectura, actualización y borrado, respectivamente. Sin embargo, en una *blockchain* solo se permiten las operaciones de lectura y escritura.
- *Estructura:* en una *blockchain*, los datos se representan como una cadena de bloques (una lista enlazada de bloques).

Para formar esta lista de bloques enlazados, cada bloque contiene un *hash* del bloque anterior, formando una cadena de *hashes*. Dicho *hash* se obtiene al usar al procesar el contenido del bloque a través de una función resumen o función *hash*, obteniendo una cadena de caracteres (o *hash*) único para dicho bloque en función de los datos que



contiene. Dichos enlaces entre los bloques permiten mantener la inmutabilidad de la *blockchain* ya que, si se modifica un bloque, este modificaría su *hash*, lo cual alteraría el *hash* del siguiente bloque, y así sucesivamente. Por ello, los pares involucrados en la *blockchain* detectarían dicha discrepancia e ignorarían el cambio realizado [98].

El contenido de un bloque en *blockchain*, como se muestra en la Figura C.1, es el siguiente:

- Un *hash* del contenido del bloque.
- Una referencia *hash* del bloque anterior, para crear dicha cadena de bloques.
- Una marca de tiempo, en el que se registra el momento en que se ha creado el bloque.
- Un conjunto de registros, los cuales son los datos que almacena el bloque. Dichos datos se denominan transacciones, en las que cada una de ellas tiene una firma digital para certificar quien ha sido el autor o propietario de dicha transacción.
- Un *nonce*, el cual se usa para el proceso de minado y creación de nuevos bloques en la cadena.

## C2. Mecanismos de consenso

En sistemas centralizados, como sistemas bancarios, existe una autoridad central que controla el sistema de datos. Esta autoridad es la única responsable de mantener la integridad de sus datos.

Sin embargo, en un sistema distribuido como la *Blockchain* donde existen múltiples pares, que no tienen por qué confiar entre sí, es necesaria la implantación de una serie de reglas o protocolos que permitan que los pares se pongan de acuerdo para garantizar la seguridad e integridad de los datos mantenidos por los nodos que conforman la red *blockchain*. Estas reglas se denominan mecanismos de consenso [99], los cuales consisten en el uso de un acuerdo entre los nodos participantes del sistema distribuido, de forma que la red funcione correctamente frente a fallos y se garantice un consenso sobre los datos que están almacenando. A continuación, se explica los mecanismos más usados, aunque existen muchos otros mecanismos de consenso en la actualidad.

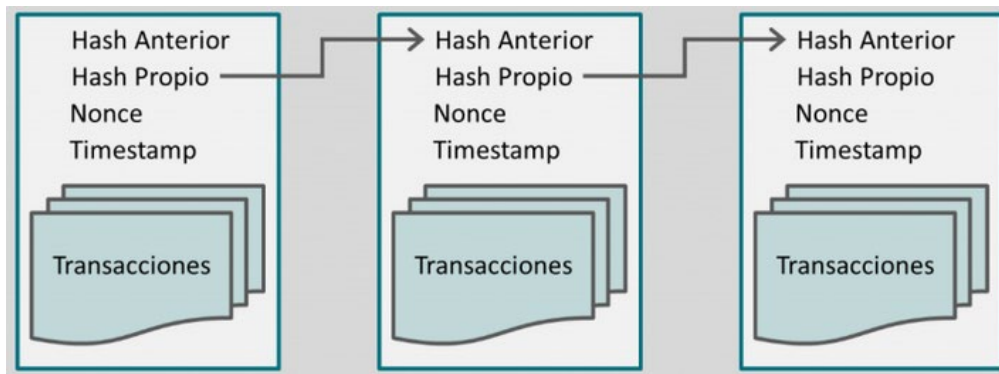


Figura C.1. Ejemplo de cadena de bloques

Fuente: <https://www.welivesecurity.com/la-es/2018/09/04/blockchain-que-es-como-funciona-y-como-se-esta-usando-en-el-mercado/>

### C2.1. Proof of Work

El *Proof of Work (PoW)* es el más conocido por su uso en el ámbito de las criptomonedas [100]. PoW consiste en la resolución de un problema matemático que requiere mucha potencia de computación para poder calcularse. Dicho problema es presentado a los pares de la red encargados de la creación de nuevo bloques, los cuales se suelen denominar mineros.

La complejidad de dicho problema depende de diversos factores de la red (cantidad de usuarios, carga de la red...) y el más común es el uso de una función *hash* aplicada a la cabecera del bloque, que genera una salida con un resultado determinado (por ejemplo, que el resultado de la función *hash* contenga un número concreto de ceros (0) al principio).

Para ello, en la cabecera se usa el campo *nonce*, que se debe ir variando hasta que encontrar aquel valor que al usar la función *hash* en la cabecera del bloque, se obtenga el resultado determinado. Una vez obtenido, el par que ha minado el bloque, lo comparte con toda la red para que se pueda validar dicho bloque y se añada a la cadena actual. El uso de PoW permite asegurar, debido a la cantidad de trabajo computacional realizado para encontrar el valor *nonce* adecuado, la integridad de los datos añadidos a la cadena de bloques. Esto se debe a que, si un tercero quisiera añadir o modificar un bloque, debe realizar un trabajo computacional en contra de toda la red, lo cual es prácticamente imposible (a no ser que se posea más del 50% de la capacidad computacional de red [101]).

El principal problema de este mecanismo de consenso es el consumo energético. A medida que más y más nodos participan en la red para realizar el proceso de minado mediante PoW, la dificultad computacional para añadir bloques es cada vez mayor. Esto hace que añadir datos a la cadena de bloques no sea solo ineficiente, sino que implica que el coste energético aumente cada vez más para añadir datos a la cadena de bloques [102].

## **C2.2. Proof of Stake**

En la *Proof of Stake (PoS)*, al contrario que en PoW, solo unos nodos determinados de la red, denominados validadores, son los encargados del proceso de validación y creación de bloques [103]. Los criterios usados para elegir dichos nodos son varios: desde el tiempo de participación, rol en la red... Esto se debe a que aquellos usuarios que gestionan los nodos validadores, debido a su grado de involucración en la red *blockchain*, quieren que esta funcione correctamente. Por este motivo, el resto de los nodos confían en que dichos nodos actuarían de forma correcta para mantener la *blockchain* en cuestión.

La principal ventaja de este mecanismo es que el proceso de minado en PoS no requiere una gran cantidad de potencia computacional (y, en consecuencia, de gasto energético para añadir datos a la cadena de bloques al contrario que la PoW) ya que los nodos validadores no competirían para la creación del siguiente bloque de la cadena, si no que el minado de los bloques de la cadena se asigna de forma aleatoria entre dichos pares validadores. Además, aumenta la seguridad de los datos ya que solo estos nodos son los responsables de la creación de bloques, eliminando terceros que puedan modificar o alterar la *blockchain*.

## **C2.3. Proof of Authority**

La *Proof of Authority (PoA)* se basa en que las transacciones deben ser validadas por unos usuarios administradores designados [104]. El resto de los usuarios deben confiar en estos administradores como la autoridad de la red al validar y añadir bloques a la *blockchain*. El funcionamiento de PoA consiste en elegir de forma aleatoria los validadores, que han sido previamente autorizados por una autoridad central, para la gestión de la *blockchain*. Estos validadores se encargarían de que la *blockchain* funcione correctamente y serían los responsables de realizar el proceso de minado para añadir datos a la *blockchain*.

Las ventajas que aporta PoA es que su eficiencia para añadir datos a la *blockchain*, ya que el mecanismo para añadir bloques se encuentra mucho más controlado. A su vez, reduce enormemente el coste energético de añadir datos a la *blockchain* frente a otros mecanismos como PoW.

La principal desventaja de este mecanismo de consenso es su naturaleza centralizada, ya que solo unas entidades autorizadas son aquellas que participan en la red. Esto compromete la descentralización propia de la tecnología *Blockchain*, ya que a pesar de que PoA no deja de ser distribuido porque hay varios nodos validadores en la red, existe una o varias autoridades centrales que administran la red *blockchain*.

### **C3. Tipos de blockchain**

Existen varios tipos diferentes de *blockchain* dependiendo del uso y las restricciones existentes para la validación y creación de bloques. Los más destacados son las redes *blockchain* públicas, y privadas o permissionadas [105].

Por un lado, en las redes *blockchain* públicas cualquier usuario puede desplegar un nodo público de la red para realizar validaciones de los datos y participar en los mecanismos de consenso, permitiendo que cualquier usuario pueda ser capaz de participar en el proceso de introducir nuevos bloques a la cadena. Asimismo, cualquier usuario puede realizar transacciones y consultar el contenido de los bloques al tratarse de una *blockchain* pública. La ventaja de este sistema es que es totalmente descentralizado ya que no existen una autoridad central que regule su funcionamiento.

Este tipo de redes *blockchain* es la más popular, siendo el ejemplo más claro su uso en el ámbito de las criptomonedas. Por ejemplo, la red *blockchain* de Bitcoin [106] es pública, ya que cualquier nodo puede descargar el código fuente y ejecutar un nodo participante en la red en su máquina local. Sin embargo, la principal desventaja de este sistema es que típicamente utilizan mecanismos de consenso como PoW para añadir bloques, los cuales son muy poco eficientes y suponen un gran coste energético por parte de los nodos para mantener la *blockchain*.

En la Figura C.2 se muestra una gráfica con el consumo estimado y mínimo en TWh por hora que se utiliza en el minado de bloques de bitcoin. En mayo de 2021, el consumo estimado de la red *blockchain* de bitcoin es 121.13 TWh, lo cual equivale al consumo energético de Países Bajos [107].

Para dar solución a empresas o aplicaciones donde se necesita un acceso restringido a la *blockchain*, las privadas o permissionadas han surgido como un tipo de *blockchain* que podría solucionar el problema del coste energético mencionado [108]. Las *blockchain* privadas cuentan con los mismos elementos que una *blockchain* pública. Sin embargo, las *blockchain* privadas constan de una unidad central que se encarga de controlar diferentes acciones dentro de la misma. Esta unidad central es la que permite dar acceso a usuarios, controlar permisos de la *blockchain*, gestionar nodos de la red... Esta opción es muy atractiva para empresas y entornos privados, donde el acceso a la red *blockchain* es restringido.

La principal ventaja de este sistema es que son más eficientes que las *blockchain* públicas, ya que la unidad central permite organizar qué nodos realizan el proceso de minado reduciendo el coste energético y disminuyendo el tiempo que tarda una transacción en ser añadida a la cadena de bloques. Sin embargo, este tipo de *blockchain* suelen ser gestionadas y controladas por una empresa o entidad, por lo que no tiene la naturaleza descentralizada de la *blockchain* pública.

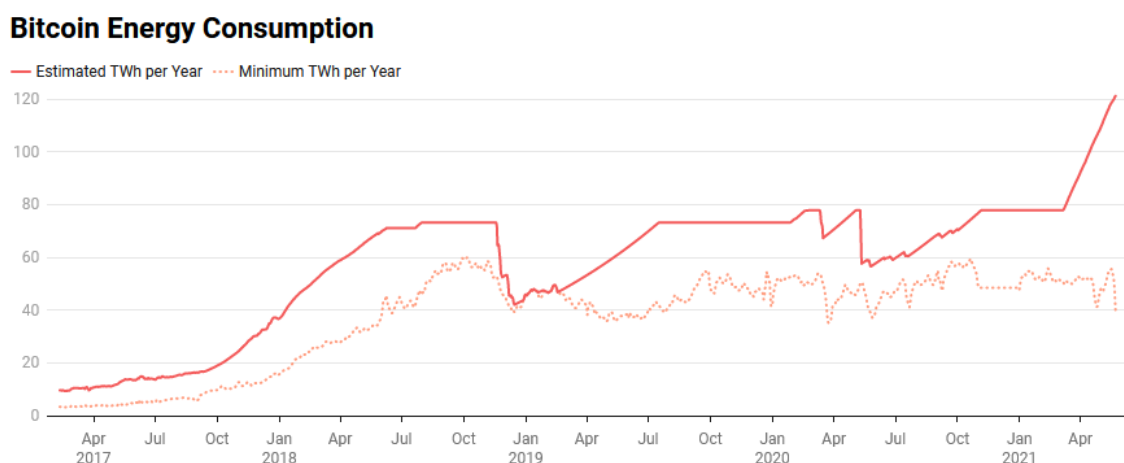


Figura C.2. Consumo energético de la red *blockchain* de Bitcoin.

Fuente: <https://digiconomist.net/bitcoin-energy-consumption/>

## C4. Tokenización

La tokenización es la transformación y representación de un activo dentro de una *blockchain* [109]. Para ello, se realiza un proceso de transformación que consiste en digitalizar características de dicho activo y almacenar dicha información en la *blockchain*. De forma general, cualesquiera elementos puede ser tokenizado, desde objetos del mundo real a datos digitales. Los datos que se almacenarían en la *blockchain* se denominan *tokens*, los cuales son objetos similares a monedas pero que carecen de valor de curso legal. Estos *tokens* permiten representar digitalmente cualquier aspecto de la realidad en una *blockchain*.

La principal ventaja del proceso de tokenización es que permite aumentar la seguridad y transparencia de los activos a los que representa [108], utilizando la *blockchain* para almacenar de forma inmutable dichos *tokens*. Los usos de la tokenización junto a la tecnología *blockchain* son infinitos, ya que prácticamente cualquier objeto o activo puede ser tokenizado.

Por ejemplo, en nuestro sistema de videoconferencia con registro verificado de las intervenciones de sus usuarios, los *tokens* son los *hashes* que se genera al aplicar la función *hash* SHA-256 al contenido multimedia de las intervenciones. Esto permite generar un token único para cada intervención, de forma que cada *token* almacenado en la cadena de bloques representaría unívocamente el contenido multimedia de la intervención que ha generado dicho *token*.

# **Anexo D. Proceso de señalización de la comunicación multimedia**

---

En este anexo se describe cómo se realiza el proceso de señalización necesario para establecer la comunicación multimedia en la aplicación web utilizando WebRTC.

## D.1 Señalización

En el proyecto desarrollado, se requiere la implementación de un sistema de comunicación multimedia entre diferentes pares. En WebRTC, previo a la comunicación multimedia, se necesitan intercambiar una serie de mensajes, mediante un proceso de señalización, que contienen la información necesaria para que los clientes involucrados en la comunicación puedan transmitir y recibir el tráfico multimedia. En el anexo C se explica detalladamente en que consiste WebRTC y cómo funciona el proceso de señalización.

En la Figura D.1 se muestra el proceso de señalización entre dos pares, A y B. Durante dicho proceso de señalización, se pueden distinguir tres fases diferentes:

- *Acceso a reunión multimedia*: señalada en amarillo en la Figura D.1, es la fase inicial en la que los pares se unen a las reuniones multimedia donde se comprueba el token de acceso y se captura el contenido multimedia del usuario.
- *Intercambio de mensajes oferta/respuesta*: señalado en rojo en la Figura D.1, se desarrolla el mecanismo de intercambio de información SDP entre los pares con el modelo oferta/respuesta de WebRTC.
- *Negociación de candidatos ICE*: por último, señalado en azul en la figura D.1, los agentes ICE de cada cliente intercambian sus candidatos ICE para establecer la ruta óptima para la posterior transmisión de tráfico multimedia entre los pares.

Una vez que se han completados las tres fases descritas, se puede comenzar la retransmisión del tráfico multimedia entre los pares que han realizado el proceso de señalización. Esta transmisión multimedia se realiza de igual a igual entre los pares.

Los pasos seguidos durante este proceso de señalización son los siguientes:

- En primer lugar, el par A envía un mensaje *join* al servidor con el identificador de la reunión multimedia a la que desea unirse y el token de acceso a la reunión.
- El servidor comprueba si dicha sala está vacía o si ya hay un par que se haya unido a dicha sala. En este caso, la sala está vacía al haberse unido A en primer lugar.



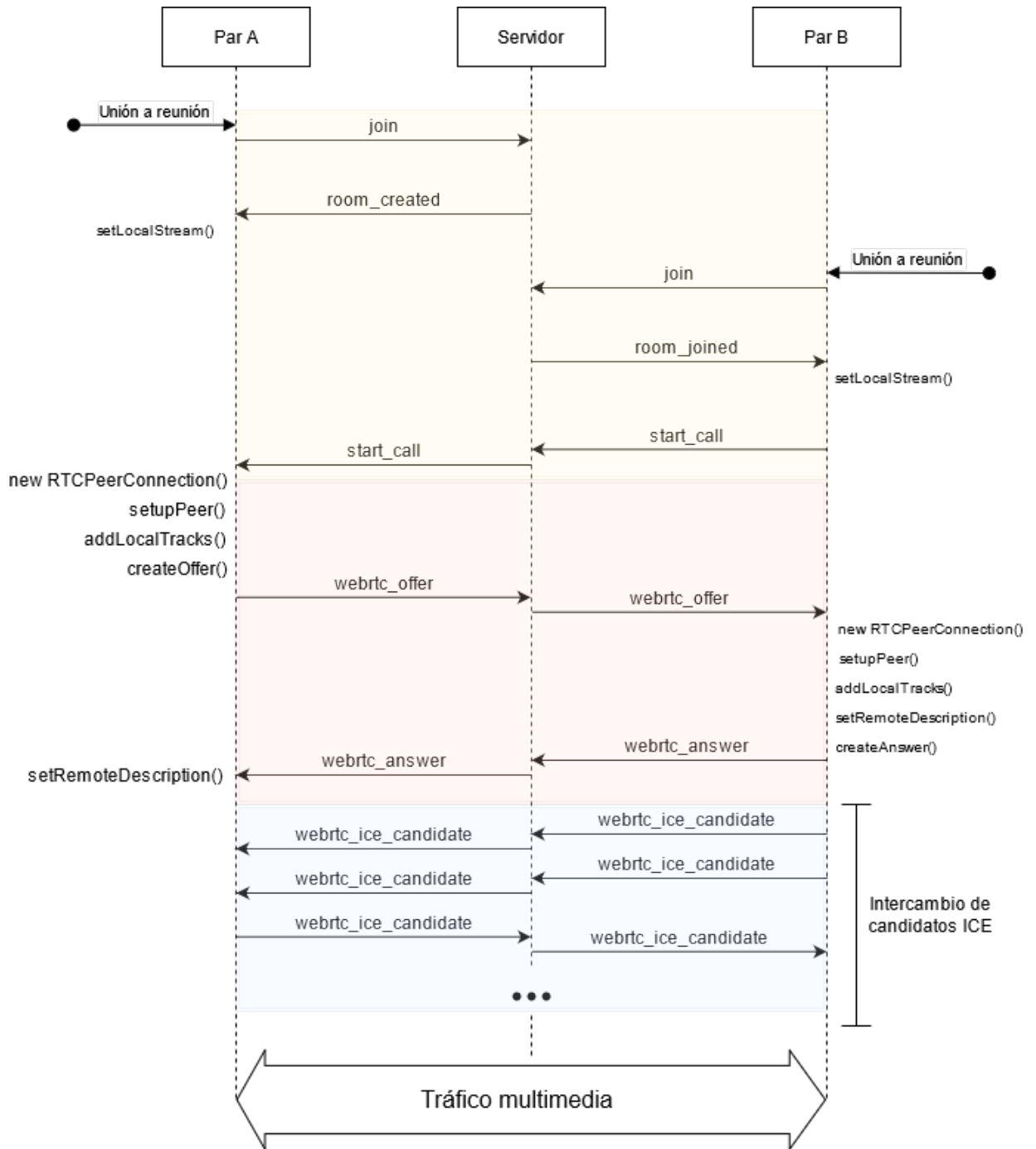


Figura D.1 Proceso de señalización entre dos pares

- El servidor responde con un mensaje *room\_created* al par A, ya que no hay ningún otro par dentro de la reunión multimedia a la que se ha conectado el par A. Dicho par comenzaría a capturar el contenido multimedia del usuario y esperaría a que se una otro par a la reunión.

- Posteriormente, el par B se une a la sala enviando un mensaje *join* al servidor, enviando junto a dicho mensaje el identificador de la reunión multimedia y el token de acceso a la misma.
- El servidor comprueba que la reunión no está vacía, y envía un mensaje *room\_joined* al par B ya que el par A ya se encontraba en la reunión multimedia.
- El par B, al recibir el mensaje *room\_joined*, procede a iniciar la llamada con el par o pares que hay en la sala. Para ello, el servidor reenvía un mensaje *start\_call* a los pares que se encuentran en la sala.
- El par A, tras recibir el mensaje *start\_call*, comienza el proceso de intercambio de información de señalización mediante el modelo de oferta y respuesta. A continuación, crearía y configuraría un objeto *RTCPeerConnection* para gestionar la conexión y se generaría el mensaje de respuesta con la información SDP para enviarlo al par B.
- El par B recibe dicha información SDP y crea y configura un objeto *RTCPeerConnection* para dicha conexión al igual que el par A. Posteriormente, genera y envía al par A el mensaje de respuesta con la información SDP.
- Por último, el agente ICE de cada navegador, procede a recoger los candidatos ICE para poder crear la conexión. Dichos candidatos son intercambiados mediante diferentes mensajes *web\_ice\_candidate*.

En este ejemplo, se muestra la conexión entre dos pares para la comunicación multimedia. Cuando se conectan más de dos pares, se realiza un proceso de señalización similar, en el que un tercer par se uniría a la reunión. En la Figura D.2 se muestran los mensajes intercambiados entre los pares que ya se encuentran en la reunión y un par C que quiere acceder. Los pasos seguidos son los siguientes:

- Dicho par C enviaría un mensaje *join* al servidor con el número de la sala a la que se desea unir.
- El servidor comprobaría que la sala no está vacía, por lo que enviaría al par C el mensaje *room\_joined*.

- El par C, al recibir el mensaje *room\_joined*, comienza el proceso de la llamada con los pares que se encuentran en la sala enviando un mensaje *start\_call* al servidor.
- Posteriormente, el servidor envía un mensaje *start\_call* a todos los pares que se encuentran en la sala creada.
- Los pares que ya se encontraban en la reunión (el par A y B), al recibir el mensaje *start\_call*, crean un objeto *RTCPeerConnection* que gestionaría esta nueva conexión y enviarían el mensaje de respuesta con la información SDP al par C.
- El par C recibiría un mensaje *webrtc\_offer* por cada uno de los pares que se encuentren en la reunión, y crearía un objeto *RTCPeerConnection* para cada conexión.

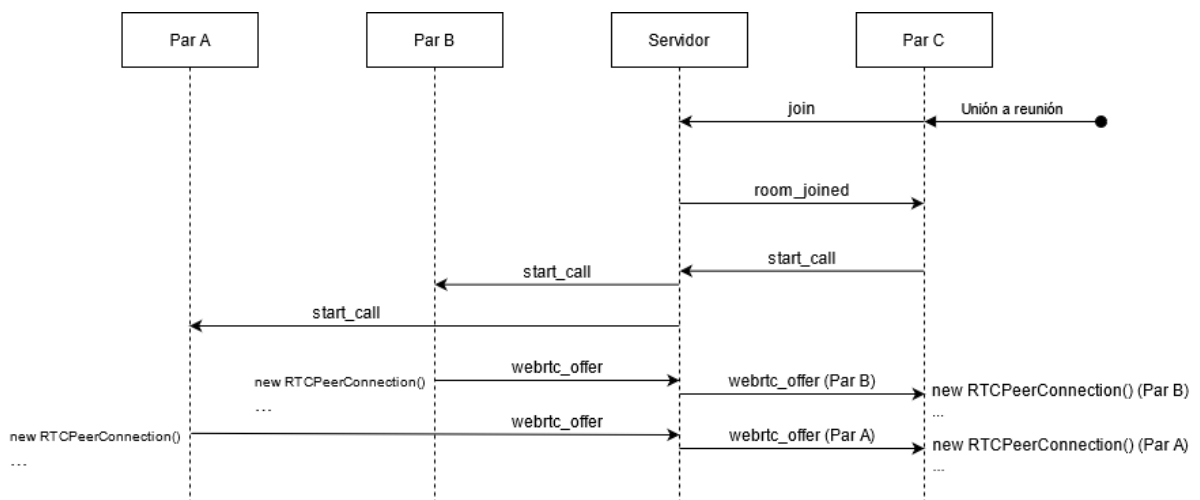


Figura D.2. Proceso de señalización al unirse un nuevo par (Par C) a la reunión

# **Anexo E. Aspectos no funcionales de la arquitectura de software**

---

En este anexo se describen dos aspectos funcionales de la arquitectura del sistema implementado. Por un lado, se describe cómo se ha adaptado la interfaz para conseguir que esta pueda ser utilizada en diferentes tipos de dispositivos, ya sean terminales móviles o portátiles y ordenadores de sobremesa. Por otro lado, se explica el procedimiento seguido para el despliegue del sistema desarrollado en la nube.

## E1. Análisis de la responsividad de la interfaz de usuario

En cuanto al desarrollo de la interfaz de usuario, se han tenido en cuenta los *mockups* descritos en el capítulo 3, *Análisis de requisitos y funcional*, utilizando *React* como se ha ido explicando en este capítulo para desarrollar la aplicación web. Un aspecto importante de la interfaz de usuario es que debería poder utilizarse en cualquier tipo de terminal (teléfonos móviles, tabletas, portátiles, ordenadores de sobremesa...) desde el navegador, para así poder acceder a reuniones multimedia desde diferentes dispositivos.

Para ello, la utilización de la biblioteca de componentes de *Ant Design* en *React* provee una gran variedad de elementos que proveen una interfaz de usuario sencilla e intuitiva. Para la utilización de dicha biblioteca en *React*, solo es necesario importar aquellos componentes que se usan en la vista que se está desarrollando, como se muestra en la Figura E.1, el cual importa diferentes tipos de componentes.

Posteriormente, para utilizarlos en el código, se usarían como si fueran componentes de *React*. Por ejemplo, en la Figura E.2 se muestra el uso del componente *Button* que provee *Ant Design*.

```
import {  
  Form,  
  Input,  
  Button,  
  Divider,  
  message  
} from 'antd';
```

Figura E.1. Ejemplo de importación de componentes de *Ant Design* en *React*

```
<Button type="primary" htmlType="submit">  
  |   Añadir usuario  
</Button>
```

Figura E.2. Ejemplo de componente *Button* de *Ant Design*

Para conseguir que la aplicación pudiera ser utilizada desde dispositivos móviles fue necesario desarrollar una interfaz responsiva. Esto es, que la interfaz de usuario de la aplicación web fuera capaz de adaptarse a cualquier tipo de dispositivo cuando donde se visualice. Con ello, la estructura de la página debe ser flexible, llegando a cambiar elementos si la resolución de la interfaz cambia a partir de un margen, de forma que se puede conseguir adaptar la interfaz a la resolución del dispositivo.

En cuanto a los componentes de *Ant Design*, muchos de sus componentes se adaptaban correctamente a la interfaz de usuario de forma responsiva. En el caso de las tablas de toda la aplicación web, como se muestra en la Figura E.3, se puede especificar mediante la propiedad *responsive* de las columnas de las tablas, aquellos puntos de ruptura en los que dicha columna se ocultaría dependiendo de la resolución de la pantalla. Por ejemplo, en las columnas de la tabla de reuniones, la columna ID de la reunión solo se podría visualizar en pantallas *lg* (ancho mínimo de 992 píxeles). Mientras tanto, en las columnas donde no se especifica dicho campo *responsive*, se mostrarían sin importar el tamaño de la interfaz.

En la Figura E.4 se muestran las columnas que se muestran al usuario cuando utiliza pantallas grandes, como las de un portátil u ordenador de sobremesa, mientras que en la Figura E.5 se muestran las columnas que se verían desde un dispositivo móvil, donde se ocultan aquellas columnas menos relevantes.

```
const columns = [
  { title: 'Reunión', dataIndex: 'name', key: 'name' },
  { title: 'ID de la reunión', dataIndex: 'meetingId', key: 'meetingId', responsive: ['lg'] },
  { title: 'Fecha de la reunión', dataIndex: 'date', key: 'date', responsive: ['md'] },
```

Figura E.3. Ejemplo de columnas responsivas

#### Reuniones realizadas

Seleccione una reunión para consultar las intervenciones realizadas en dicha intervención

Reunión	ID de la reunión	Fecha de la reunión	Ver intervenciones
Reunión de prueba	74264fd3-edb7-455f-b6d9-da32da470cbe	30/6/2021	<a href="#">Intervenciones</a>
Nueva reunión	e2fdf466-251e-430d-b581-b4e5bdab7b0d	30/6/2021	<a href="#">Intervenciones</a>
Reunión de prueba 6 de mayo de 2021	4dfbbc4a-7527-4fed-9647-93bfec6a3db6	6/5/2021	<a href="#">Intervenciones</a>

Figura E.4. Ejemplo de tabla en la pantalla de un portátil

## Reuniones realizadas

Seleccione una reunión para consultar las intervenciones realizadas en dicha intervención

Reunión	Ver intervenciones
Reunión de prueba	<a href="#">Intervenciones</a>
Nueva reunión	<a href="#">Intervenciones</a>
Reunión de prueba 6 de mayo de 2021	<a href="#">Intervenciones</a>

Figura E.5. Ejemplo de tabla en la pantalla de un teléfono móvil

Sin embargo, para lograr la interfaz responsiva en otros componentes y vistas, como la interfaz de usuario de la reunión multimedia o el menú lateral, utilizando *Cascading Style Sheets (CSS)*, se hace uso de los *CSS Media Queries* que permiten establecer condiciones que modifican el estilo de la aplicación web según la expresión que se especifique en la condición [110].

## E2. Despliegue en Heroku

Por último, para probar el sistema implementado fuera del entorno local de desarrollo, se ha desplegado en *Heroku* para que el sistema pueda ser accesible desde la web. Para ello, se ha desplegado la infraestructura que se muestra en la Figura E.6, en la que el proyecto (*bc-videochat*) de la aplicación web multimedia consta de 2 *dynos*: una *web dyno* para servir la aplicación web y un *worker dyno* para la realización de las tareas en segundo plano. A su vez, para poder utilizar la biblioteca *bull* para la planificación de tareas en segundo plano, se ha desplegado un servidor Redis que gestione las colas de dichas tareas.

Por otro lado, para la red *blockchain* desarrollada, se han creado tres proyectos: uno para el alojamiento del nodo raíz de la *blockchain* (*bc-videochat-root-peer*), y otros dos para el alojamiento de los nodos de la red *blockchain* (*bc-videochat-peer-1* y *bc-videochat-peer-2*). Para el sistema de comunicación entre los nodos, en el proyecto del nodo raíz se despliega un servidor Redis para habilitar el intercambio de mensajes en la red *blockchain*.

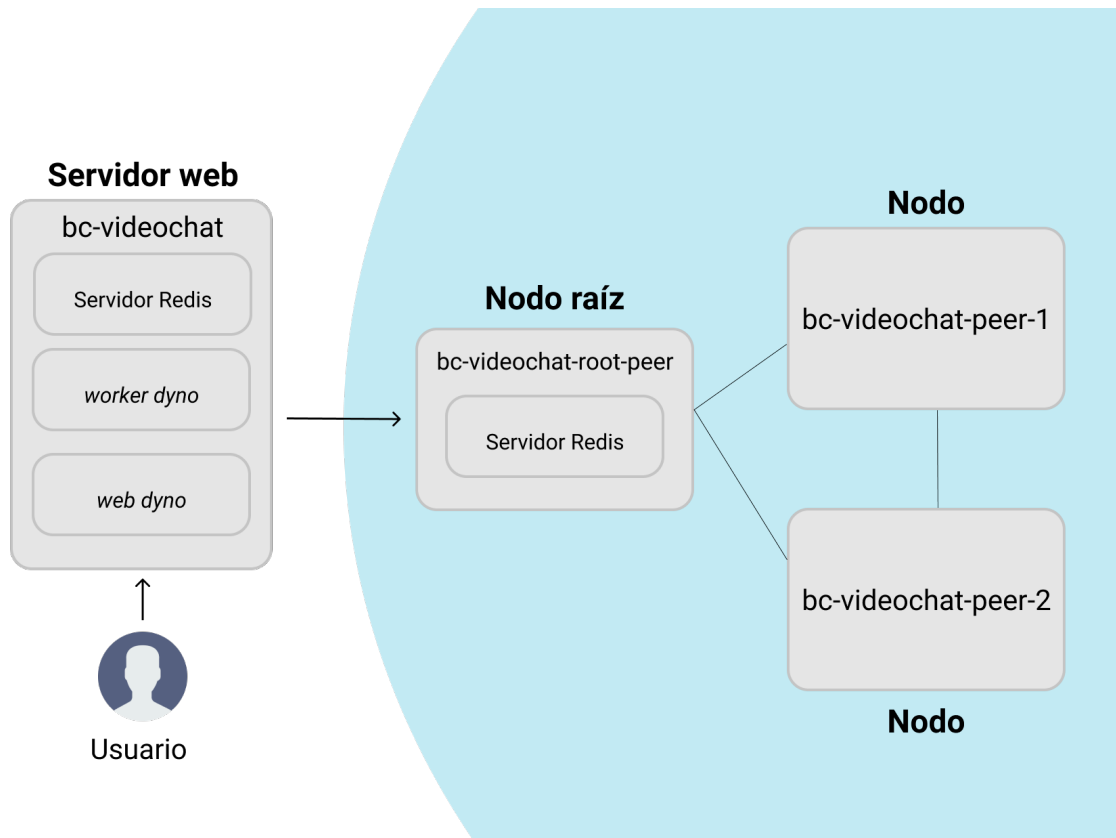


Figura E.6. Infraestructura del sistema desplegado en Heroku

Para la creación de cada proyecto, es necesario crear un repositorio con la herramienta *Git* [111]. En primer lugar, desde la carpeta del proyecto, se debe ejecutar la orden que se muestra en la Figura E.7 para crear un repositorio del proyecto.

Una vez creado el repositorio con el proyecto, hay que crear la aplicación en *Heroku* que aloja el proyecto. Para ello, usando las herramientas instaladas con *Heroku CLI*, desde la carpeta raíz del proyecto hay que ejecutar la orden que se muestra en la Figura E.8, la cual crea una rama remota en el repositorio del proyecto, que corresponde al código del proyecto que se ejecuta en la Nube.

Por último, hay que subir el código del repositorio a *Heroku*. Para ello, hay que registrar los cambios realizados en el repositorio y hacer *commit* de los cambios realizados. Hecho esto, se deben enviar los cambios realizados a la rama remota creada de *Heroku*. Las órdenes descritas se muestran en la Figura E.9, las cuales se deben ejecutar cada vez que se realicen cambios en el código desarrollado.



En cuanto a los servidores Redis desplegados, Heroku permite provisionar al proyecto especificado de un servidor Redis. En la Figura E.10 se muestra la orden utilizada para desplegar el servidor Redis en el proyecto de la aplicación web multimedia, *bc-videochat*.

Por último, para añadir un *worker dyno* para la realización de tareas en segundo plano en el proyecto *bc-videochat*, se debe crear un archivo *Procfile*, que contiene la configuración del proyecto desplegado en *Heroku*. En nuestro caso, en este se especifican los procesos que van a ejecutarse en el proyecto, donde se encuentran el *web dyno* y el *worker dyno*, que ejecutan las órdenes que se muestran en la Figura E.11 para que comience su ejecución.

Utilizando las órdenes y operaciones descritas en los diferentes proyectos, se consigue que estos puedan ser desplegados en la plataforma de *Heroku*.

```
$ git init
```

Figura E.7. Orden para crear el proyecto en Heroku

```
$ heroku create
```

Figura E.8. Orden para crear el proyecto en Heroku

```
$ git push heroku master
```

Figura E.9. Orden para subir cambios realizados a Heroku

```
$ heroku addons:create heroku-redis:hobby-dev -a bc-videochat
```

Figura E.10. Orden para desplegar un servidor Redis en un proyecto de Heroku

#### Procfile

- 1 web: npm start
- 2 worker: node ./server/worker.js

Figura E.11. Contenido del fichero Procfile



# Glosario

---

<b>Siglas</b>	<b>Descripción</b>
<i>TFG</i>	Trabajo de Fin de Grado
<i>WebRTC</i>	Web Real-Time Communications
<i>NPM</i>	Node.js Package Manager
<i>SDK</i>	Software Development Kit
<i>API</i>	Application Programming Interface
<i>SFU</i>	Selective Forwarding Unit
<i>P2P</i>	Peer-to-Peer
<i>NAT</i>	Network Address Translation
<i>STUN</i>	Session Traversal Utilities for NAT
<i>TURN</i>	Traversal Using Relays around NAT
<i>IP</i>	Internet Protocol
<i>DLT</i>	Distributed Ledger Technologies
<i>CRUD</i>	Create Read Update Delete
<i>PoW</i>	Proof of Work
<i>PoS</i>	Proof of Stake
<i>PoA</i>	Proof of Authority
<i>HTTP</i>	HyperText Transfer Protocol
<i>HTML</i>	HyperText Markup Language
<i>Redis</i>	Remote Dictionary Server
<i>PubSub</i>	Publisher/Subscriber
<i>PaaS</i>	Platform as a Service
<i>BaaS</i>	Backend as a Service

<i>URL</i>	Unified Resource Link
<i>ICE</i>	Interactive Connectivity Establishment
<i>SHA</i>	Secure Hash Algorithm
<i>CORS</i>	Cross Origin Resource Sharing
<i>AJAX</i>	Asynchronous Javascript And XML
<i>RFC</i>	Request for Comments
<i>UUID</i>	Universally Unique Identifier
<i>ID</i>	Identificador
<i>UML</i>	Unified Modelling Language
<i>Blob</i>	Binary Large Object