

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

APLICACIÓN WEB DE SENSADO COLABORATIVO PARA OBTENER ACCESO GRATUITO A REDES WIFI

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Autor: José David Sánchez López-Trejo

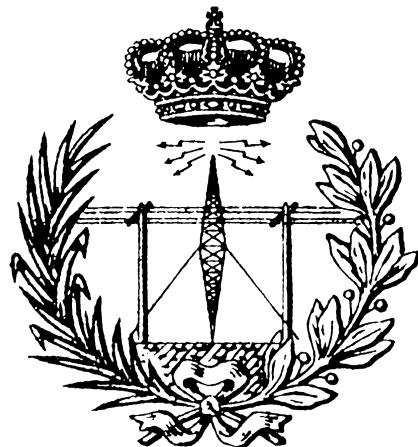
Tutores: Álvaro Suárez Sarmiento
Elsa María Macías López

Fecha: Noviembre 2017



UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

APLICACIÓN WEB DE SENSADO COLABORATIVO PARA OBTENER ACCESO GRATUITO A REDES WIFI

HOJA DE FIRMAS

Alumno/a

Fdo.: José David Sánchez López-Trejo

Tutor/a

Tutor/a

Fdo.: Álvaro Suárez Sarmiento

Fdo.: Elsa María Macías López

Fecha: Noviembre 2017



UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

APLICACIÓN WEB DE SENSADO COLABORATIVO PARA OBTENER ACCESO GRATUITO A REDES WIFI

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: Nombre del Presidente

Vocal

Fdo.: Nombre del vocal

Secretario/a

Fdo.: Nombre del secretario

Fecha: Noviembre 2017

Esta página se ha dejado en blanco intencionadamente.

Índice

1. Introducción	1
1.1. Aplicaciones y servicios telemáticos y sensores	2
1.2. Computadores empotrados y tecnologías Web para servicios telemáticos	3
1.3. Objetivos	4
1.4. Estructura de la memoria	5
2. Sensado colaborativo para servicios telemáticos	7
2.1. Introducción	8
2.2. Introducción al sensado colaborativo mediante teléfonos móviles	9
2.3. Sensado colaborativo para obtener acceso gratuito a Internet	10
2.4. Idea básica del sistema propuesto	11
3. Tecnologías utilizadas	15
3.1. Introducción	16
3.2. <i>Software</i> necesario para el funcionamiento básico de la Raspberry Pi 3	17
3.2.1. Raspbian y WiFi	18
3.2.2. Hostapd	18
3.3. El Servidor RADIUS	21
3.3.1. El Servidor RADIUS	21
3.3.2. Contabilidad	23
3.3.3. Seguridad	24
3.3.4. FreeRADIUS	24
3.4. CovaChilli	25
3.4.1. Funcionamiento	26
3.4.2. Configuración	27
3.4.3. Interfaz <i>JavaScript Object Notation</i>	28
3.5. El entorno node.js para programación de back-end	35
3.5.1. El gestor de paquetes npm	36
3.5.2. El <i>framework Express</i>	36
3.5.3. El módulo <i>formidable</i>	38
3.5.4. El <i>middleware Body-Parser</i>	38
3.6. Programación del front-end	39
3.6.1. El HTML y el CCS	39
3.6.2. El javascript: bibliotecas, API y complementos	40
4. Análisis previo y funcional	45
4.1. Objetivo y análisis de requisitos	46
4.2. Análisis previo	47

4.3. Análisis funcional	49
4.3.1. Módulo Usuarios	50
4.3.2. Módulo Control de Acceso	54
4.3.3. Módulo Control de Red	54
5. Análisis orgánico de la implantación del <i>software</i>	59
5.1. Módulos de Control de Red y de Acceso	60
5.1.1. Ajustes e instalaciones preliminares	60
5.1.2. Configuración del Servidor RADIUS	62
5.1.3. Instalación de CoovaChilli	63
5.1.4. Instalación y configuración de Hostapd	66
5.1.5. La interfaz Web de daloRADIUS	67
5.2. Módulo de Usuarios	73
5.2.1. Configuración previa y estructura del módulo	74
5.2.2. Submódulo de Control Usuarios	77
5.2.3. Submódulo Servidor.	82
5.2.4. Submódulos de Aplicación Web	91
6. Análisis de la evaluación empírica	109
6.1. Resultados con una tableta Android	110
6.2. Consumo de RAM del sistema en la Raspberry Pi	111
6.3. Consumo de RAM de los navegadores Web	115
6.4. Tráfico HTTP y TCP	117
6.5. Análisis de los archivos de audio generados	118
7. Conclusiones y posibles ampliaciones	121
7.1. Conclusiones	122
7.2. Posibles ampliaciones futuras	122
7.2.1. Replicabilidad	123
7.2.2. Privacidad	123
7.2.3. Procesado del audio obtenido	123
7.2.4. Desacoplar procesos	124
7.2.5. Unificar <i>software</i>	124
A. Instalación del <i>hardware</i> necesario	125
B. Descarga automática del código de la Nube	129
C. Reanudación automática del Servidor en caso de fallo	131
D. Ciclo de Vida del TFG	135

E. Problemas Acontecidos durante el Desarrollo de la Solución	139
F. Tabla de Desarrollo Temporal	141
Referencias	143
Referencias Web	143
Glosario	147
8. Presupuesto y pliego de condiciones	151
8.1. Hola	152

1. Introducción

En este capítulo presentamos la idea general del sistema desarrollado y su entorno, los objetivos perseguidos, algunos conceptos básicos relacionados con las tecnologías usadas y la estructura de la memoria.

1.1. Aplicaciones y servicios telemáticos y sensores

A finales del Siglo XX quedó claro que Internet había cambiado para siempre la vida de los seres humanos. Los nuevos servicios telemáticos que era capaz de proporcionar no hicieron más que despegar, cambiando definitivamente la forma de vida de las personas. A principios del Siglo XXI se ha visto una nueva revolución en los servicios telemáticos basados en Internet: el uso de sensores para aumentar la potencia de estos servicios. Surge así un nuevo concepto denominado Internet of things (IoT) [1]. Este amplio concepto, lo que en el fondo aprecia es la posibilidad de usar Internet como red de comunicación universal para comunicar datos provenientes de sensores a todo lo largo y ancho del Mundo; con la posibilidad de usar valores sensados para aumentar la expresividad de los servicios telemáticos; el caso más simple: poder referenciar fotos en base al lugar en el que se toman y poder comunicarlas de tal manera que el receptor pueda saber gráficamente donde se hicieron las fotos de manera instantánea.

En los últimos años ha surgido una derivada de la IoT que consiste en usar los dispositivos móviles y ordenadores cuentan con una amplia gama de sensores que les permiten monitorizar gran cantidad de datos sobre su entorno. Esto facilita la obtención de sensado y ofrece una alta variedad de funciones relacionadas con los sensores. Los dispositivos de esta clase presentes habitualmente son acelerómetros, giroscopios, sensores de temperatura, sonido, luz... El conjunto de técnicas relacionadas con esta modalidad de sensado suele denominarse *sensado móvil*.

La presencia de estos sensores en los diferentes dispositivos de uso cotidiano junto con las tecnologías de ubicación actuales permite la captura y recopilación de grandes cantidades de datos localizados con las consiguientes posibilidades de uso. Los proyectos de ciudades inteligentes o *smartcities* pueden hacer uso de esta facilidad para realizar diversas tareas de análisis particularizados por distritos, zonas comerciales u otros criterios y en tiempo real. Mediante aplicaciones dedicadas y el cuidado adecuado de la privacidad, las empresas pueden solicitar a los usuarios de sus servicios el acceso puntual a ciertos sensores del teléfono. Los datos que recopilan les pueden ser de utilidad a cambio de servicios añadidos. Ejemplos de estos servicios añadidos sería el acceso gratuito a su red *Wireless Fidelity* (WiFi), y ejemplos de empresas que podrían proporcionar este servicio son hoteles o franquicias de restauración.

Para llevar esto a cabo, los puntos de acceso actuales pueden contar con mecanismos que requieran de ciertas acciones por parte de los dispositivos que se conectan a ellos antes de proporcionarles acceso a internet. En el caso de este *Trabajo Fin de Grado* (TFG), partimos de la situación en la que se implementaría un servidor que proporcione acceso a Internet a un dispositivo móvil a cambio de los datos de su sensor. Extendiendo

este supuesto a un caso general, una empresa que tenga dispositivos conectados a sus redes puede ofertar servicios contextuales a los mismos a cambio de que hagan sentido gratuito.

1.2. Computadores empotrados y tecnologías Web para servicios telemáticos

No sólo se ha avanzado en la comercialización de dispositivos móviles, sino que en los últimos años ha habido una verdadera explosión de nuevos computadores de muy reducidas dimensiones que caben íntegramente en una sola placa electrónica. Ejemplos de estos computadores son los denominados *Raspberry Pi* [2]. En este tipo de computadores se pueden alojar clientes y servidores de servicios telemáticos para Internet.

La tecnología Web ha alcanzado un alto grado de desarrollo, por lo que ahora tiene el potencial de ofrecer y explotar servicios que hasta hace poco tiempo requerían de una aplicación nativa, particularmente en los dispositivos móviles que ya están ampliamente extendidos y cuyo tráfico web generado representa más la mitad del total mundial.

Los tradicionales elementos de la programación Web en el Cliente como pueden ser *HyperText Markup Language* (HTML) [3], *Cascading Style Sheets* (CSS) [4] y *JavaScript* [5], [6] (JS) se han visto reforzados con la salida de nuevos *frameworks* e interfaces de programación de aplicaciones (*Application Programming Interface*, API) que implementan una amplia gama de funciones cada vez mayor y para la que anteriormente se requería mucho más esfuerzo de computación. El *Web RealTime Communications* (WebRTC) [7] es uno de estos *frameworks* en el ámbito de la comunicación multimedia en tiempo real. Del mismo modo, las tecnologías en el servidor como *Pre Hypertext Processor* (PHP) [8] han visto la aparición de alternativas que utilizan lenguajes propios del lado del cliente como Node.js [9], que gozan de gran adopción y de una comunidad de desarrolladores muy activa.

Dada la amplia variedad de dispositivos móviles existentes en la actualidad (clientes móviles), de tabletas, computadores de sobremesa... es necesario el diseño de aplicaciones Web que sean capaces de producir resultados adaptables a los distintos tipos de pantalla. Estas aplicaciones Web se denominan responsivas (*responsive*) [10]. Este hecho es importante tenerlo en cuenta a la hora de diseñar servicios telemáticos basados en sensores capaces de enviar datos a distintos tipos de terminales programando únicamente un único código para la aplicación Web.

Por todo esto, hoy en día es posible el uso de todas estas tecnologías para obtener datos de los sensores de un dispositivo que acceda a una aplicación web, implementando medios

alternativos de acceso a un sistema. Estos sistemas toman cada día mayor importancia como nuevos servicios de telecomunicación y es el ámbito en el que desarrollamos este TFG.

1.3. Objetivos

El objetivo general de este TFG es dar un uso real a todo lo expuesto anteriormente, uso de sensores móviles y tecnologías web actuales, desarrollando un sistema completo de sensado móvil aprovechando el micrófono de los dispositivos móviles. Sobre una Raspberry Pi 3 se instala un punto de acceso WiFi a Internet y un servidor Web que exporta una aplicación Web responsiva. Los terminales móviles podrían acceder a esa aplicación Web para abrir una sesión de acceso a Internet. Cosa que lograrían si habilitan el acceso a su micrófono para que se registre el ruido o sonido ambiente durante una cierta cantidad de tiempo.

Usando una de las API de WebRTC, estos datos se recopilan junto a la ubicación del dispositivo y una marca de tiempo y se envían a un servidor Web implementado con Node.js en una Raspberry Pi 3 que actuaría como punto de acceso WiFi a Internet. Este punto de acceso contaría con un grupo de elementos software que, trabajando conjuntamente con la aplicación web, proporcionarán acceso a internet al dispositivo a cambio de recibir los datos del sensor cada cierto tiempo, eventualmente disponiendo de dichos ficheros para realizar algún otro procesamiento como mapeo de niveles de audio o pruebas de localización acústica.

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

- Desarrollar un prototipo de la aplicación Web responsiva y realizar pruebas en dispositivos reales (computador empotrado y dispositivos móviles).
- Desarrollar un prototipo del servidor Web que proporcione acceso a Internet al recibir los datos de la aplicación Web responsiva cada cierto tiempo y realizar pruebas en dispositivos reales.
- Realizar pruebas de campo de captura, transmisión y representación de los datos en entornos reales observando el rendimiento del sistema completo.

1.4. Estructura de la memoria

En este primer capítulo hemos expuesto las ideas básicas del entorno y el objetivo del TFG.

En el capítulo 2 presentamos las ideas básicas del sensado colaborativo y su aplicación para el acceso gratuito a Internet a través de una red WiFi con infraestructura de una sola celda.

En el capítulo 3 presentamos una descripción somera de todas las tecnologías utilizadas para el desarrollo del proyecto.

En el capítulo 4 mostramos el análisis de requisitos y el funcional (visión de alto nivel de los distintos módulos funcionales principales de que consta el sistema) del sistema completo para proporcionar una visión general de los distintos componentes del sistema.

En el capítulo 5 analizamos orgánicamente el diseño de los componentes software que hemos utilizado para implantar en la práctica el sistema. El análisis de la instalación del software básico de la Rapsberry Pi 3 se muestra en el Apéndice A. Leído ese Apéndice se puede entender mejor qué software explicado en este capítulo, que se ha instalado y configurado.

En el capítulo 6 se muestra un análisis de los resultados experimentales para demostrar la potencia del sistema desarrollado.

Finalmente en el capítulo 7 se muestran las conclusiones y algunas posibles ampliaciones del sistema, una vez hemos aprendido que es posible implantarlo y observado distintos problemas que han ido surgiendo sobre la marcha. Justamente para analizar esos aspectos y otros detalles se han provisto otros apéndices.

En el capítulo del Pliego de condiciones y presupuesto se exponen primero, la posible problemática en el uso del sistema y un análisis del coste económico de su implantación.

2. Sensado colaborativo para servicios telemáticos

El sensado colaborativo es una técnica que permite obtener datos sensados de una ubicación física utilizando la colaboración de un número indeterminado de dispositivos provistos de sensores. Utilizando esta técnica de sensado se puede favorecer el acceso a servicios telemáticos sin intercambio económico. En este capítulo presentamos las ideas básicas del sensado colaborativo y su aplicación para el acceso gratuito a Internet a través de una red WiFi con infraestructura de una sola celda.

2.1. Introducción

Hoy en día existe una gran cantidad de dispositivos de sensado de propiedades físicas haciendo posible la implantación del paradigma IoT. En este paradigma se aprovecha la cantidad masiva de sensores desplegados a lo largo y ancho de todo el mundo para obtener mediciones de parámetros físicos de interés. Además, estos datos se pueden transportar a lo largo y ancho de todo el mundo a través de Internet. Nosotros no estamos interesados en sensores insertados en un dispositivo electrónico miniaturizado empotrado y funcionando de forma autónoma. Estamos interesados en aquellos sensores que se implantan como parte de otros dispositivos, como pueden ser los relojes sofisticados (*smartwatches*), los teléfonos móviles sofisticados (*smartphones*), las tabletas portables o los computadores portátiles de media-alta gama.

Los dispositivos anteriores son capaces de incluir sensores (en un sentido amplio del significado de sensado) de: temperatura, acelerómetro (que permite averiguar datos sobre el movimiento de personas), nivel de señal electromagnética para comunicación inalámbrica (un ejemplo es el *Received Signal Strength Indicator* (RSSI)), las imágenes (mediante una o varias cámaras fotográficas), de ruido o sonido ambiente (mediante un micrófono) o imágenes y sonido o ruido en movimiento (vídeo)...

Un detalle muy importante es que una gran parte de los ciudadanos actualmente disponen de alguno de los dispositivos anteriores. Además, siempre los llevan consigo a todos sus quehaceres diarios y cada vez se utilizan para hacer muchos de estos quehaceres cotidianos. Esto significa que cada persona que deambule por una ciudad diariamente es susceptible de recoger una gran cantidad de datos sensados del entorno que le rodea, sin hacer ningún esfuerzo adicional. Un ejemplo usual es el de un ciudadano que camina diariamente desde su casa hasta la sede de su trabajo llevando consigo su teléfono móvil sofisticado. Bastaría que diera (una sola vez) los permisos adecuados, para que su teléfono móvil se encargara de contar el número de pasos que ha dado, distribuirlos por metros recorridos y tiempo que ha tardado en darlos. Si además se ha medido el nivel de oxígeno en la sangre, a través del sensor adecuado del teléfono móvil, al comienzo y al final del recorrido, y tiene el software adecuado, se le puede avisar del nivel de stress que ha sufrido en su esfuerzo e incluso recomendándole que camine de otra forma diferente. Este no es más que un ejemplo sencillo de cómo utilizar *individualmente* los sensores de un dispositivo móvil. Nosotros estamos interesados en el uso de estos sensores para realizar tareas colaborativas.

2.2. Introducción al sensado colaborativo mediante teléfonos móviles

La técnica de sensado móvil (*mobile sensing*) [11] es aquella, que en sentido amplio permite usar el teléfono móvil para sensar datos del entorno compartiéndolos con otros entes (personas o máquinas), para lograr llevar a cabo una tarea en común.

Ejemplos de sistemas en los que se puede emplear esta técnica es un sistema de aparcamiento colaborativo. Supongamos que una persona acude a un gran aparcamiento de un gran centro comercial, con la cámara fotográfica del teléfono móvil es capaz de captar imágenes de aparcamientos que están vacíos y su identificación. Acto seguido, una aplicación identifica al aparcamiento vacío y automáticamente envía su identificación a un servidor en la Nube, junto con una etiqueta de tiempo que indica el momento del día en que fue recogida dicha información. Un procesado en la Nube de todas las identificaciones que se reciben puede exportar (para accederlos mediante una aplicación móvil), un listado de aparcamientos libres en un momento dado. Entonces, un usuario que desee aparcar en dicho aparcamiento, antes de acudir hasta él, o cuando está en el camino para acudir a él, podría consultar dicha información y hacer una estimación de cuándo podría llegar al aparcamiento. Si además recibe información del Centro comercial indicando cuántos coches hay cerca de esos aparcamientos libres, cuántos están entrando al aparcamiento y cuantos están saliendo, entonces, el conductor se podría hacer una idea de si le vale la pena intentar acudir a un aparcamiento concreto o bien dirigirse a otro. En este ejemplo, el principal objetivo es compartir información distribuidamente y en tiempo real para lograr un objetivo común a todos los usuarios de ese sistema: aparcar en el menor plazo de tiempo posible. Podemos decir que este es un ejemplo de intereses simétricos porque todos los usuarios tienen el mismo objetivo: encontrar aparcamiento rápidamente.

Un ejemplo en el que no todos los usuarios están interesados en exactamente lo mismo podría ser el siguiente: un conjunto de usuarios utilizan sus teléfonos móviles como estaciones meteorológicas (a partir de sus distintos sensores pueden dar datos sobre temperatura, humedad, presión...), en aquellos lugares en los que se encuentren. Envían los datos a un *Organismo Público* (que opera en un determinado lugar del mundo), encargado de proveer datos meteorológicos al público en general. Cualquier persona del mundo podría consultar los datos de este Organismo, por ejemplo para averiguar datos sobre el estado metereológico de una playa concreta. Sin embargo, aquellos usuarios que no han proporcionado datos meteorológicos y que quieran consumirlos, deben proveer datos sobre el estado de las carreteras o de otro tipo (de interés para el Organismo público porque también puede proveer esos datos). Esto es, unos usuarios

están interesados en proveer datos que ellos mismos podrían consumir o bien datos que otros usuarios pudieran proveer.

Un tercer y último ejemplo puede ser aquel en el que un conjunto de usuarios podría proveer datos a cambio de que se les proporcione otros. Por ejemplo, supongamos que un usuario pone en marcha una aplicación móvil capaz de detectar la canción que está sonando en un momento determinado en un parque determinado de una ciudad (para ello utiliza el micrófono de su teléfono móvil como sensor básico). Esos datos los transmite a una discográfica interesada en averiguar qué canciones se escuchan en los parques de las ciudades junto con datos relativos a la hora del día... La cuestión es ¿por qué estaría un usuario interesado en proveer datos de ese tipo? una posible respuesta sería que a cambio esperaría recibir una canción gratuita por cada 100 canciones que averiguara. En este caso tenemos una variante del sensado móvil colaborativo en el que hay usuarios que sensan datos a cambio de un beneficio concreto.

Este TFG trata de diseñar un sistema de sensado móvil colaborativo, basado en el micrófono del teléfono móvil, en el que los usuarios que sensan se benefician de un servicio telemático concreto.

2.3. Sensado colaborativo para obtener acceso gratuito a Internet

Supongamos la existencia de una empresa que opere en infraestructuras físicas a las que se supone un elevado tránsito o alojamiento de personas, como aeropuertos u hoteles. Si estas empresas desearan monitorizar ciertos aspectos de sus recintos como niveles de ruido, temperatura, humedad... habrían de adquirir, configurar y mantener el equipamiento dedicado a estos propósitos, con los correspondientes costes asociados a cada uno de estos aspectos que, dependiendo de la calidad deseada, pueden ser ciertamente elevados.

Por otra parte, estas empresas suelen tener el acceso WiFi a Internet como parte de su carta de servicios a los clientes. Los despliegues de red utilizados para implementar estos accesos habitualmente cuentan con servicios de portal cautivo, que requieren de cierta operativa por parte del usuario que se conecta a la Red a cambio de obtener finalmente dicho acceso. Esta operativa suele incluir la introducción de datos personales para crear una cuenta de usuario con la que hacer *login* en futuros accesos, la selección de publicidad que recibir en el correo electrónico o su visualización... en definitiva, acciones que luego son utilizadas por la empresa para contacto comercial, potenciales clientes y otros procesos de marketing (Figura 2.1).

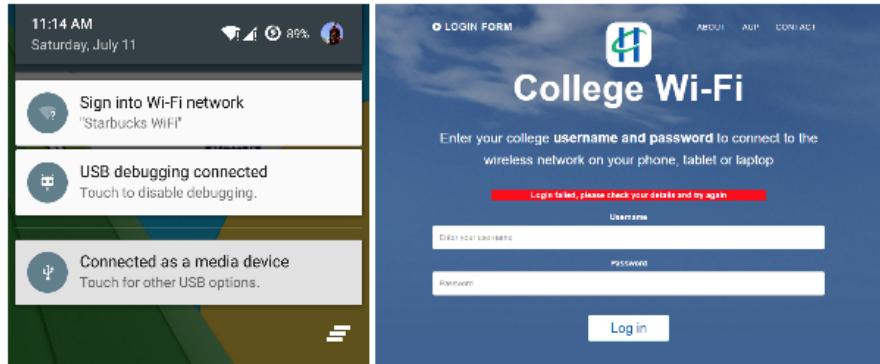


Figura 2.1: Ejemplo de página Web que permite el acceso a Internet a través de WiFi

La solución propuesta en este TFG trataría de aunar la necesidad de monitorización de infraestructuras de la empresa y el acceso a Internet de sus usuarios, implementando un servicio de portal cautivo dotado de una aplicación web con la que se adquieran datos de los sensores del dispositivo en lugar de la clásica introducción manual de datos o credenciales a cambio de la obtención del acceso a la red.

Este sensado colaborativo podría utilizarse para diversas estadísticas (evolución temporal, mapeos de niveles...) con los que la empresa podía obtener una realimentación con unos niveles mínimos de fiabilidad. Los datos recabados no tendrían la misma calidad que el sensado realizado por equipamiento específico de alto rendimiento, pero el coste sería menor y su mantenimiento y configuración más sencillos de realizar. Además, esta solución software tendría una alta escalabilidad y versatilidad de configuraciones, por ejemplo pudiendo permitir el acceso a Internet a cambio de obtención de datos en intervalos regulares en los que el portal cautivo debe mantenerse abierto en alguna ventana o pestaña del navegador, cortándose el acceso si se cierra, o dando un tiempo de acceso fijo (*lease-time*) a cambio de una lectura del micrófono durante un intervalo de tiempo limitado.

2.4. Idea básica del sistema propuesto

Como se ha mencionado en el apartado anterior, las soluciones de portal cautivo empleadas habitualmente requieren que el usuario del servicio introduzca manualmente ciertos datos personales o credenciales o que seleccione y visualice publicidad en procesos a menudo tediosos, repetitivos o confusos. El estilo de vida actual, en su búsqueda de la inmediatez, favorece y recompensa las soluciones en las que el usuario pierde el menor tiempo posible en el proceso de acceso al servicio.

Por ello, un servicio de portal cautivo que solo requiera del sensado del dispositivo solo precisaría de la obtención de los permisos pertinentes de acceso al hardware, ya

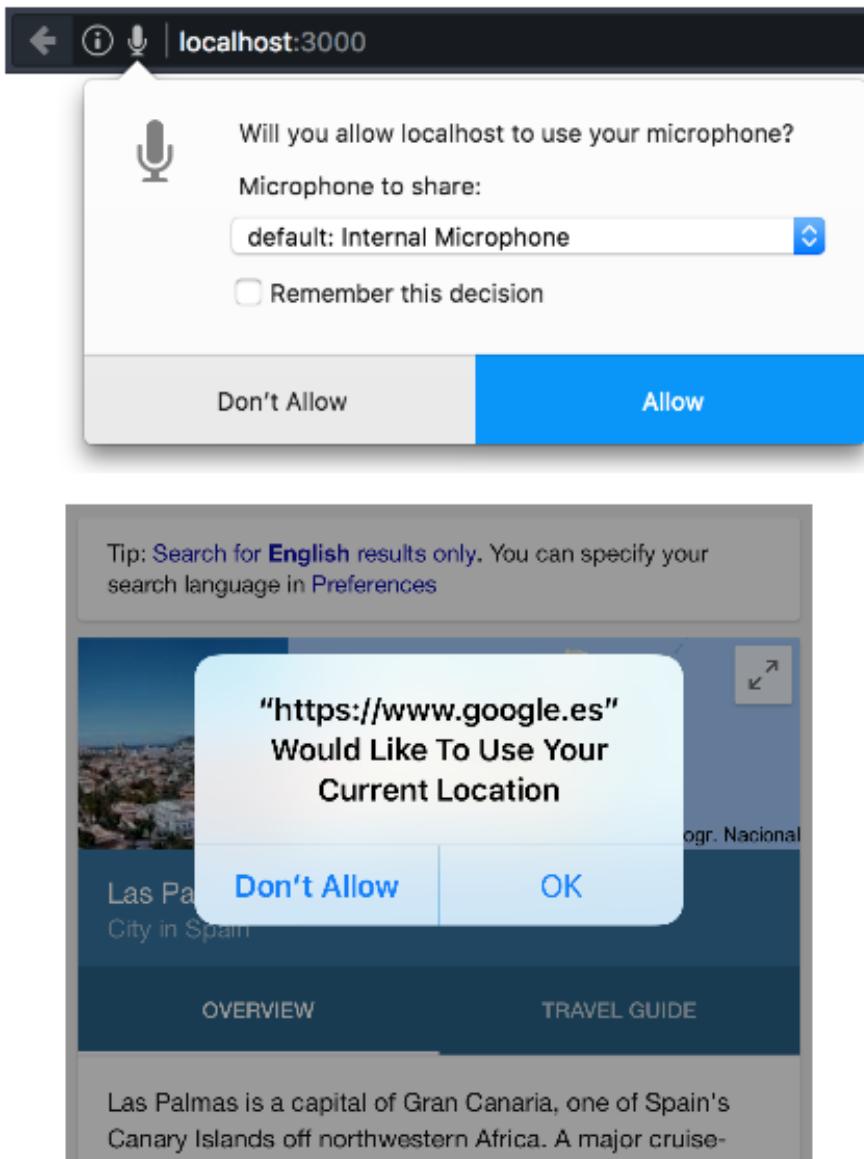


Figura 2.2: Ejemplo de ventanas que solicitan permisos para usar sensores del teléfono móvil

implementados en los navegadores actuales en forma de popups sencillos con la opción de aceptar o declinar tal acceso. De esta forma e incluso aunque el servicio de portal cautivo utilizado requiera de credenciales, su obtención y el subsiguiente acceso a Internet se hace de una forma mucho más rápida para el usuario, solo teniendo que aceptar y dar permiso a la aplicación web para llevar a cabo el procedimiento automáticamente (la primera vez que la utiliza en un local determinado). En la Figura 2.2 se muestran dos ejemplos típicos de ventanas en la que se solicita permiso al usuario para que el sistema operativo (o navegador) pueda utilizar datos de los sensores del teléfono móvil).

Una implantación que acometa este sistema puede desarrollarse de diversas formas. En el caso de la que concierne a este TFG, se ha optado por utilizar una Raspberry Pi 3

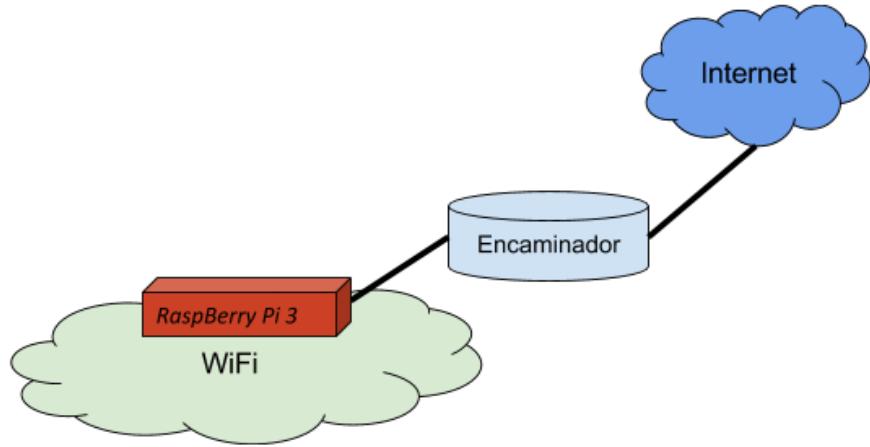


Figura 2.3: Esquema general de la infraestructura para proporcionar acceso a Internet mediante una aplicación de sensado móvil colaborativo

como elemento principal del diseño (en el capítulo 4 se presenta con mayor detalle). En ella se instala todo el *software* necesario para proporcionar el servicio. Se utiliza el modelo 3 de la Raspberry Pi porque es el primero que cuenta con módulo WiFi integrado en el dispositivo, a diferencia de modelos anteriores en los que había que instalar un módulo USB aparte. En la figura 2.3 se muestra el esquema hardware general planteado, en el que se observa que con la interfaz WiFi de la Raspberry Pi 3 permitimos que los usuarios se conecten al encaminador que proporciona acceso a Internet de forma controlada.

La idea básica de la funcionalidad del sistema de sensado móvil colaborativo implantado es que aquel usuario que desee acceder a Internet, estando en la instalación de un centro que lo permita a través de su WiFi, simplemente permitiría (una sola vez) el uso del micrófono de su teléfono móvil para sensar ruido o sonido que le rodea. A cambio obtiene unos minutos de acceso gratuito a Internet. De esta forma simplificamos el proceso típico de acceder a un portal cautivo con complejos procesos de autorización (cada vez que se desea acceder a Internet). Esto supone un beneficio adicional para el usuario. Además, el propietario del local puede obtener mapas de ruido o sonido de su local a partir del sensado distribuido que hacen los distintos usuarios. Estamos pues ante un escenario en el que no todos los miembros que colaboran tienen exactamente el mismo objetivo; pero todos obtienen un beneficio en la colaboración: el usuario accede a Internet de forma gratuita, el propietario obtiene mapas de ruido con un hardware muy barato.

Hasta donde alcanza nuestro conocimiento, no sabemos de ninguna solución similar a ésta, lo que ahonda en la novedad de este TFG.

3. Tecnologías utilizadas

Para el correcto funcionamiento de nuestro sistema de sensado móvil colaborativo asimétrico con beneficios dispares es necesario el uso del hardware básico presentado en el capítulo 2 y de un conjunto de aplicaciones software. En este capítulo presentamos una descripción somera de todas las tecnologías utilizadas para el desarrollo del proyecto.

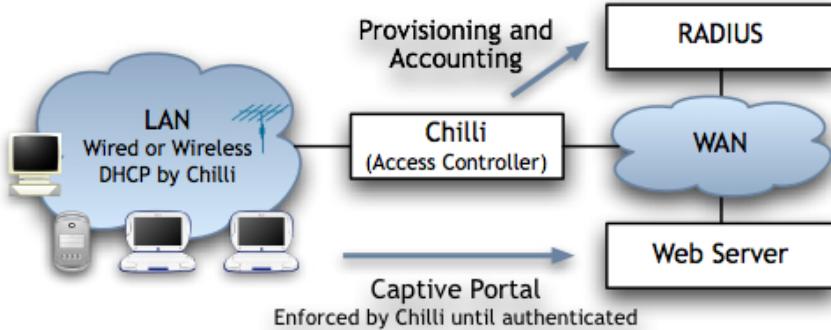


Figura 3.1: Esquema general del Software utilizado para la implantación del portal cautivo.

Fuente: <http://coova.github.io/CoovaChilli/>

3.1. Introducción

Como habíamos comentado en el capítulo 2, el hardware que necesita el propietario del local es básicamente una Raspberry Pi 3. Para que el sistema de sensado funcione debemos instalar en la Raspberry Pi 3 software que nos permita definir la funcionalidad del portal cautivo que nos permitiría implantar la aplicación de sensado móvil colaborativo de acceso a Internet. En la Figura 3.1 se muestra un esquema general del *software* necesario. En concreto instalamos el siguiente:

- *Host Access Point Daemon (Hostapd)* [12]: este daemon de linux se encarga de configurar el módulo WiFi en modo AP. De esta manera, la Raspberry Pi 3 puede actuar como un punto de acceso configurable, conectándose sus clientes a través de su interfaz WiFi y obteniendo acceso a la red a través de la interfaz Ethernet de acuerdo al esquema que se muestra en la Figura 2.3.
- *CoovaChilli* [13]: el software controlador de acceso que proporcionaría direcciones *Internet Protocol* (IP) a las conexiones entrantes con su *Dynamic Host Control Protocol* (DCHP), redirigiría dichas conexiones al portal cautivo para su autenticación en el sistema y las gestionaría junto a un sistema de Autenticación, Autorización y Contabilización (AAA) que ha de ser instalado aparte, habitualmente un servidor *Remote Authentication Dial-In User Service* (RADIUS) [14].
- *FreeRADIUS* [15]: el servicio de AAA gracias al cual CoovaChilli controla los usuarios del sistema. Para su utilización es necesaria su previa integración con MySQL [8], que ha de estar instalado en el sistema.
- *daloRADIUS* [16]: plataforma Web destinada a controlar el servidor RADIUS de forma gráfica. En este TFG, su contenido es proporcionado por el servidor Web

NGINX.

- *Node.js*: tecnología JavaScript (para el servidor), complementada con paquetes npm, con la que implantamos el back-end de nuestra aplicación Web de portal cautivo, que interactuaría con CoovaChilli por medio de su interfaz JSON.
- *Front end Web*: página Web a la que acceden los clientes a través de su teléfono móvil implantada en Node.js a partir de: HTML, CSS y JavaScript. Esta página Web es el principal elemento diferenciador respecto a otros portales cautivos habituales, dado que su propósito sería el de recopilar del usuario, a través de su navegador, los datos del micrófono del dispositivo conectado al portal cautivo a través de su navegador. Esto es posible gracias a la *Application Programming Interface (API) MediaStream Recording* [17], tecnología estrechamente relacionada con *WebRTC* [7].

A continuación se realiza un análisis más pormenorizado de las tecnologías utilizadas. Se considera que el lector tiene un conocimiento al menos elemental de las tecnologías Web tradicionales, por lo que estas solo se introducen de forma superficial exceptuando aquellos aspectos concretos que tienen relevancia en este TFG. Los elementos *software* más específicos reciben un análisis pormenorizado, entrando en mayores detalles.

3.2. *Software* necesario para el funcionamiento básico de la Raspberry Pi 3

La Raspberry Pi es un tipo de sistema empotrado de pequeño tamaño. Es un computador de un sola placa madre, de reducidas dimensiones. Inicialmente fue planteado como un dispositivo de bajo coste (35 \$) y consumo energético (1.5 W) en su último modelo. Está orientado a propósitos educativos en colegios y países en vías de desarrollo, aunque su uso se extendió rápidamente a otros ámbitos como la robótica. Sus modelos son desarrollados por la Raspberry Pi Foundation en el Reino Unido y actualmente se le considera el ordenador de esa región más vendido.

Desde su introducción en Febrero de 2012 ha habido varias generaciones de este dispositivo, siendo la más reciente la Raspberry Pi 3 Model B que es la usada en este TFG. Este modelo consta de un *Sistema en Chip* (SoC) de Broadcom, que incluye un procesador quad-core compatible con -ARMv8 de 64 bits que opera a una frecuencia de 1.2 GHz y un procesador gráfico. Cuenta también con 1GB de *Random Access Memory* (RAM) LPDDR2 a 900 MHz. Para almacenar el sistema operativo se utiliza una tarjeta MicroSDHC, y aunque puede instalarse una gran variedad de sistemas operativos diferentes el recomendado es Linux basado en Debian (renombrado como *Raspbian*).

En cuanto a entrada-salida la Raspberry Pi 3 cuenta con cuatro puertos *Universal Serial Bus* (USB), interfaz Ethernet 10/100 Mbps y salidas *High-Definition Multimedia Interface* (HDMI) y minijack de audio. También incorpora 40 pines entrada-salida de propósito general (*GPIO, General Purpose Input-Output*) para operaciones a bajo nivel. Otro aspecto de importancia crucial para este TFG es la conectividad inalámbrica, dado que la Raspberry Pi 3 incorpora chipsets WiFi (versión IEEE 802.11n) para 2.4 GHz a 150 Mbps y Bluetooth 4.1 a 24 Mbps. Además de esto, en la actualidad se fabrica una amplia gama de accesorios destinados a ampliar la funcionalidad de la Raspberry Pi como cámaras de vídeo, interfaces de control de LEDs y sensores y otras placas de expansión.

3.2.1. Raspbian y WiFi

El Raspbian es el sistema operativo recomendado por la Raspberry Pi Foundation desde 2015. Está basado en Debian Jessie y fue desarrollado por Mike Thompson y Peter Green poniendo especial énfasis en los procesadores ARM de bajo rendimiento de la Raspberry Pi. Incluyen Python, Scratch, Java, Sonic Pi, Mathematica entre otros programas relevantes.

Utiliza *Pi Improved Xwindows Environment Lightweight* (PIXEL) como entorno gráfico de escritorio. Consiste en un entorno de escritorio *Lightweight X11 Desktop Environment* (LXDE) modificado y el gestor de ventanas Openbox. La principal característica de la Raspberry Pi 3 para este TFG es su capacidad de manejo de la tecnología WiFi. Esta tecnología especifica el nivel físico y el subnivel de control de acceso al medio (MAC, *Medium Access Control*). La versión con la que se ha trabajado es la IEEE 802.11n, que soporta el uso de varias antenas simultáneamente (*MIMO, Multiple Input Multiple Output*), aprovechando la propagación multirayecto para aumentar la tasa de transferencia, e incorpora mejoras en seguridad, agregación de trama, técnicas de *beamforming* para optimizar la emisión de señal junto a otros aspectos. Aunque IEEE 802.11n también permite trabajar en la frecuencia de 5 GHz, el dispositivo que hemos utilizado como punto de acceso WiFi trabajaba en 2.4 GHz.

3.2.2. Hostapd

Es una utilidad software cuya funcionalidad es transformar las interfaces de red en puntos de acceso y servidores de autenticación. Implementa la gestión de puntos de acceso IEEE 802.11, autenticación IEEE 802.11X, *WiFi Protected Access* (WPA), WPA2, *Extensible Authentication Protocol* (EAP), cliente RADIUS, servidor EAP y

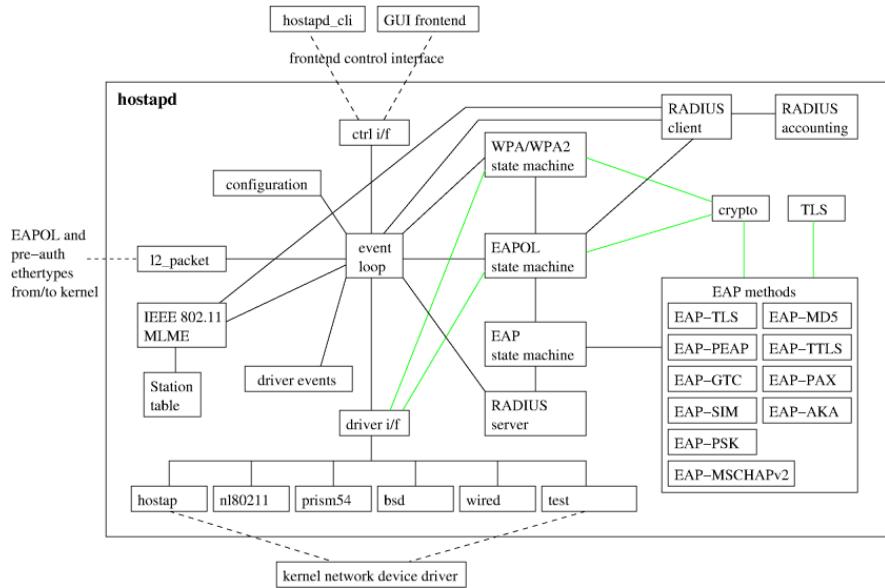


Figura 3.2: Esquema general del funcionamiento de Hostapd.

Fuente: http://w1.fi/wpa_supplicant-devel/

servidor de autenticación RADIUS. Soporta interfaces con drivers presentes en sistemas Linux y FreeBSD.

Como *daemon* ha sido diseñado para operar en segundo plano como componente *backend*, soportando aplicaciones front-end separadas. Ha sido programado para funcionar de forma modular por medio de código C organizado en archivos separados. En la Figura 3.2 se muestra un esquema de la organización del funcionamiento de Hostapd.

El hostapd se configura por medio de un archivo de texto que lista todos los parámetros de configuración, alojado en el archivo `/etc/hostapd/hostapd.conf`, que cuenta con los siguientes atributos relevantes y comúnmente utilizados, entre otros:

- Atributos de Interfaz Inalámbrica:
 - *interface*: interfaz inalámbrica va a utilizarse.
 - *bridge*: interfaz puente si la interfaz inalámbrica es parte de ella.
 - *driver* (controlador utilizado): suele ser n18011.
- Atributos de Entorno Inalámbrico:
 - *ssid*: nombre o *Service Set IDentifier* (SSID) de la red que aparecería en la lista de redes disponibles al hacer la habitual búsqueda de red desde un dispositivo inalámbrico.
 - *hw_mode*: fija el modo de operación de la interfaz y los canales permitidos. Los

valores permitidos dependen del hardware pero siempre son un subconjunto de a , b o g . Aunque se configure una red IEEE 802.11n no es aquí donde debe indicarse, ya que IEEE 802.11n opera sobre la funcionalidad de IEEE 802.11a o IEEE 802.11g.

- *channel*: el canal en el que hostapd operaría como punto de acceso. Debe ser un canal soportado por el modo de hardware establecido en el atributo *hw_mode*. Los canales de IEEE 802.11 son de 20 MHz de anchura (4 canales) en el espectro, solapándose entre ellos, por lo que deben escogerse de forma que esto no suceda. Por ejemplo, si la mayoría de puntos de acceso de la red utilizan el canal 6 sería óptimo utilizar el canal 1 o el canal 11 para evitar interferencias por solapamiento.
- Atributos de IEEE 802.11n:
 - *ieee80211n*: atributo booleano que al asignársele el valor 1 activa las funcionalidades propias de IEEE 802.11n.
 - *ht_capab*: lista de las capacidades IEEE 802.11n soportadas por el dispositivo utilizado. Actúan como flags, activándose al ser escritos entre corchetes. Ejemplos de ello serían habilitar el uso de canales de 40 MHz (*/HT40+*) o activar el modo DSSS/CCK en dichos canales (*/DSSS_CCK-40*).
- Autenticación y cifrado:
 - *macaddr_acl*: controla el filtrado de direcciones MAC.
 - *auth_algs*: campo de bits en el que el primero es para autenticación abierta, el segundo es por autenticación de Clave Compartida (WEP), pudiendo activar ambos introduciendo un 3.
 - *ignore_broadcast_ssid*: activa o desactiva la difusión del SSID.
 - *wpa*: Un campo de bits como *auth_algs*. El primero activa WPA1, el segundo activa WPA2 y el 3 activa los dos.
 - *wpa_psk* o *wpa_passphrase*: contraseña para la autenticación WPA.
 - *wpa_key_mgmt*: controla con qué algoritmos de gestión de claves podría autenticarse un cliente.
 - *wpa_pairwise*: controla el encriptado de datos de WPA.
 - *rsn_pairwise*: controla el encriptado de datos de WPA2.

En el bloque de código 3.1 se muestra un ejemplo de archivo de configuración.

```
interface=wlan0
driver=nl80211
ssid=RaspAP
hw_mode=g
channel=8
wpa=2
wpa_psk=928519398acf811e96f5dcac68a11d6aa876140599be3dd49612e760a2aaac0e
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP
rsn_pairwise=CCMP
beacon_int=100
auth_algs=3
wmm_enabled=1
```

Código 3.1: Ejemplo de archivo de configuración de Hostapd

3.3. El Servidor RADIUS

CoovaChilli hace uso de servicios de AAA separados para poder funcionar. Habitualmente este servicio es un servidor RADIUS instalado de forma separada. Por ello, se exponen algunos detalles de su funcionamiento, junto a detalles la solución concreta de servidor RADIUS utilizada en este TFG, FreeRADIUS, y otros aspectos relevantes.

RADIUS es un protocolo cliente/servidor que proporciona gestión de AAA de forma centralizada para aquellos clientes de un servicio de Red. Fue desarrollado en 1991 por parte de Livingston Enterprises, Inc. como un protocolo de autenticación y contabilidad para servidores de acceso, convirtiéndose en un estándar de la *Internet Engineering Task Force* (IETF). Se implanta a nivel de aplicación y puede utilizar tanto *Transmission Control Protocol* (TCP) como *User Datagram Protocol* (UDP). RADIUS suele ser el back-end para autenticaciones de IEEE 802.11x, como un proceso en segundo plano ejecutándose en un servidor UNIX o Windows. Actualmente se utiliza de forma prácticamente masiva por proveedores de servicios de Internet para gestionar el acceso al mismo o a redes internas, inalámbricas y a servicios de correo electrónico.

3.3.1. El Servidor RADIUS

La autenticación y autorización de RADIUS es la descrita en la *Request For Comments* (RFC) 2865 mientras que la contabilidad es la descrita en la RFC 2866.

Para la autenticación y autorización, el cliente envía una petición al *Servidor de Acceso a*

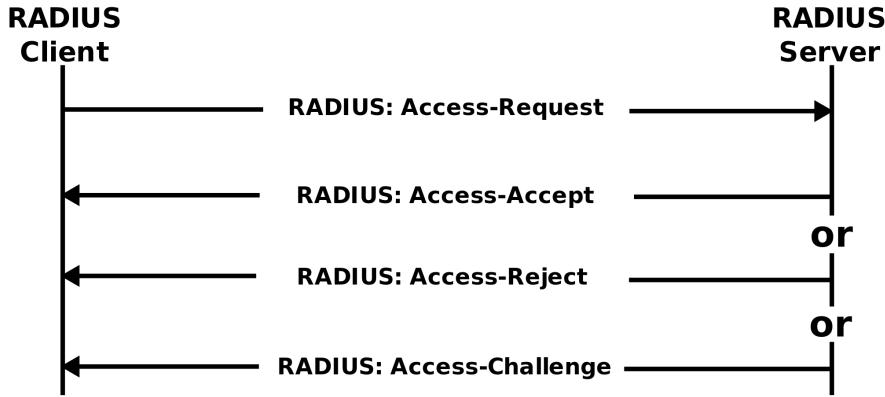


Figura 3.3: Ejemplo de mensajes de aceptación de RADIUS.

Fuente: <https://en.wikipedia.org/wiki/RADIUS>

la Red (NAS) para obtener acceso a un recurso de red particular utilizando credenciales de acceso. Estas credenciales llegan a este dispositivo mediante los protocolos de nivel de enlace. En respuesta, el NAS envía un mensaje RADIUS de Petición de Acceso al servidor RADIUS, pidiendo autorización para conceder el acceso mediante el protocolo RADIUS. Esta petición incluye credenciales de acceso, habitualmente un nombre de usuario y contraseña o un certificado de seguridad proporcionados por el usuario. El servidor RADIUS comprueba que la información es correcta utilizando patrones de autenticación como *Password Authentication Protocol* (PAP), *Challenge-Handshake Authentication Protocol* (CHAP) o EAP. En este punto se verifica la información de identificación con las almacenadas en un archivo del servidor o una fuente externa, como una base de datos SQL. Tras esto, el servidor RADIUS puede responder de tres formas (Figura 3.3):

- *Access Accept*: se concede acceso al usuario.
- *Access Challenge*: solicita información adicional, como una contraseña secundaria o un PIN.
- *Access Reject*: se deniega el acceso a todos los recursos de red solicitados por el usuario.

Estas tres respuestas RADIUS pueden incluir atributos de mensaje que pueden dar una razón para el rechazo, la petición de la información adicional (*Access Challenge*) o un mensaje de bienvenida.

Los atributos de autorización se entregan al NAS especificando los términos de acceso. Por ejemplo, una respuesta *Access Accept* podría incluir los siguientes atributos:

- IP específicas o subconjunto de IP posibles a ser asignadas al usuario.

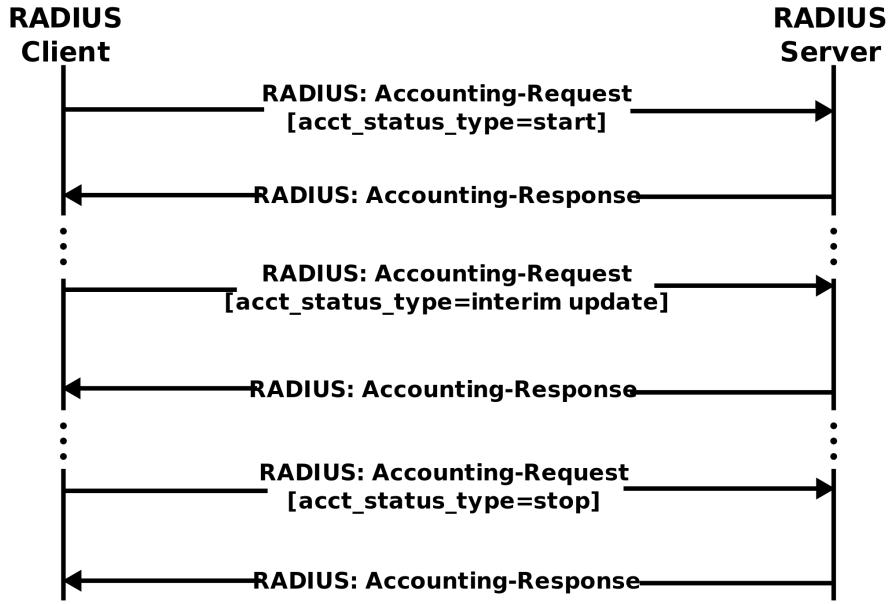


Figura 3.4: Ejemplo de mensajes de contabilidad en RADIUS.

Fuente: <https://en.wikipedia.org/wiki/RADIUS>

- Un tiempo máximo de conexión.
- Lista de prioridades u otras restricciones de acceso para el usuario.
- Parámetros de Calidad de Servicio (QoS).

3.3.2. Contabilidad

Cuando se proporciona acceso al usuario el NAS envía un mensaje de *Accounting Start* (un paquete de petición de contabilidad RADIUS que contiene un atributo `acct_status_type` con el valor `start`) para señalizar el comienzo del acceso a la red. Este registro normalmente contiene el identificador de usuario, dirección de red y un identificador de sesión único.

De forma periódica el NAS puede enviar al servidor RADIUS unos registros *Interim Update* (un paquete similar al *Accounting Start* pero con el valor de `acct_status_type` establecido como `interim-update`) para actualizar el estado de la sesión. Estos registros suelen contener la duración actual de la sesión y otra información sobre el uso de datos (Figura 3.4).

Cuando el acceso a la red del usuario se cierra el NAS envía un registro de Accounting Stop (similar a lo anterior, `acct_status_type` fijado en `stop`) al servidor RADIUS, proporcionando información de la duración final de uso en términos de tiempo, paquetes

transferidos, datos transferidos, razón de la desconexión y otra información relacionada con el acceso a la red.

El propósito de estos datos es principalmente para facturar al cliente de forma adecuada, aunque también se usa para propósitos estadísticos o de monitorización de red. Todos estos mensajes suelen contar con su propio sistema de mensajes de reconocimiento (*acknowledgement, ack*), reintentando los registros de contabilidad a intervalos determinados hasta que dicho *ack* es recibido.

3.3.3. Seguridad

El protocolo RADIUS transmite contraseñas ocultas utilizando un secreto compartido y el algoritmo de hash MD5. Para aumentar aún más la protección del tráfico RADIUS pueden utilizarse medidas adicionales como túneles IPsec. Solo las credenciales de usuario son protegidas por RADIUS, otra información que pasa a través de este podría ser susceptible a efectos de seguridad. Algunas medidas para solucionar estos problemas pueden encontrarse en el protocolo RadSec, con el que se transportan los datagramas RADIUS mediante TCP y *Transport Layer Security* (TLS).

3.3.4. FreeRADIUS

Aunque habitualmente se utilice este nombre para referirse tan solo al servidor, FreeRADIUS es realmente una suite gratuita de RADIUS distribuida bajo la GPL, versión 2, en descarga y uso gratuitos. Su desarrollo comenzó en Agosto de 1999 por Alan DeKok y Miquel van Smoorenburg utilizando un diseño modular para animar la participación de la comunidad. Incluye dicho servidor RADIUS, una biblioteca de clientes RADIUS con licencia basada en *Berkeley Software Distribution* (BSD), una biblioteca *Pluggable Authentication Module* (PAM), un módulo Apache y otras utilidades relacionadas con RADIUS. Es rápido, modular, escalable y con una gran cantidad de opciones. En la actualidad se encuentra en su versión 3, que incluye soporte para RADIUS sobre TLS incluyendo RadSec.

Es el servidor RADIUS de código abierto más popular y el servidor RADIUS más desplegado en el mundo con una base de usuarios estimada en más de 100 millones según una encuesta de 2006 citada en la Web del proyecto. Soporta todos los protocolos de autenticación habituales y el servidor cuenta con una herramienta Web para la administración de usuarios basada en PHP llamada *Dialup Admin*. Es la base de muchos productos y servicios RADIUS comerciales, como sistemas embebidos o WiMAX. Proporciona servicios de *Authentication, Authorization and Accounting* (AAA) a numerosas

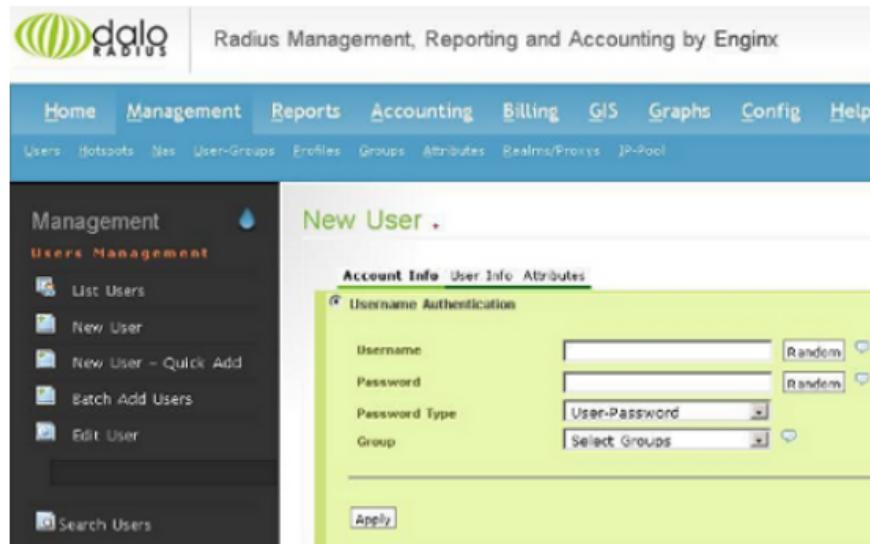


Figura 3.5: Pantalla de adición de nuevo usuario.

empresas de envergadura, compañías de telecomunicaciones y proveedores de servicios de internet de Tier 1 (como podrían ser AT&T, Orange o Telefónica). También se usa en la comunidad académica, incluyendo *eduroam*, que lo implementa junto a RadSec para incrementar la seguridad.

Los módulos incluidos en el núcleo del servidor soportan bases de datos MySQL, PostgreSQL y Oracle entre otros. También soportan todos los tipos de autenticación *Extensible Authentication Protocol* (EAP) populares, como *Protected EAP* (PEAP) y *EAP Tunneled Transport Layer Security* (EAP-TTLS). Tiene incluidos más de 100 diccionarios de fabricantes, asegurando su compatibilidad con una amplia gama de dispositivos *Network Access Server* (NAS).

Desde su versión 2 tiene soporte de *hosting* virtual, *IPv6* y *Virtual Local Area Network Management Policy Server* (VMPS).

Existen varias herramientas para gestionar FreeRADIUS aparte de la ya mencionada Dialup Admin. En este TFG se utiliza la solución daloRADIUS, una aplicación basada en Web orientada a la gestión de hotspots y despliegues de proveedores de servicios de Internet. Cuenta con una interfaz sencilla, informes gráficos, contabilidad, procesos de facturación y se integra con Google Maps para geolocalización. El aspecto general de esta interfaz de usuario se muestra en la Figura 3.5.

3.4. CoovaChilli

CoovaChilli es un *software* controlador de acceso WiFi a Internet de código abierto basado en el antiguo proyecto ChilliSpot, ya abandonado. Lanzado bajo *GNU General*

Public License (GPL), en la actualidad es mantenido por los aportes de personal original de ChilliSpot. CoovaChilli carece de interfaz gráfica, por lo que ha de controlarse utilizando la orden *chilli* en el terminal. Proporciona un entorno de portal cautivo y utiliza RADIUS o *Hyper Text Transfer Protocol* (HTTP) para proporcionar acceso y contabilidad. CoovaChilli es parte integral de CoovaAP, un firmware basado en OpenWRT especializado en *hotspots*. Soporta dos métodos de acceso diferentes para una red local inalámbrica: *Universal Access Method* (UAM) y WPA.

3.4.1. Funcionamiento

Utiliza mediante tres interfaces principales:

- De enlace descendente (*downlink*) que acepta conexiones de los clientes, mediante DHCP y ARP. Los clientes pueden encontrarse en dos estados: no autenticados y autenticadas. En este último caso, sus peticiones Web son redirigidas a un portal cautivo, el servidor Web de autenticación que usualmente les solicita usuario y contraseña. El servidor Web envía las credenciales del usuario al proceso *chilli* mediante redirecciones del navegador.
- De RADIUS para autenticar a los clientes. Para el método de acceso UAM se utilizan desafíos y contraseñas CHAP de acuerdo a la RFC 2865. Para la autenticación por WPA se utiliza el atributo de mensaje EAP para RADIUS de acuerdo a la RFC 2869. Los atributos de mensaje descritos en la RFC 2548 se usan para transferir claves de cifrado desde el servidor RADIUS hacia CoovaChilli. Si la autenticación tiene éxito el estado del cliente pasa a ser de autenticado.
- De enlace ascendente (*uplink*) para dirigir el tráfico hacia otras redes.

La interfaz de enlace ascendente se implementa utilizando drivers *network tunnel* (TUN) que opera con el encaminamiento de datagramas IP y network TAP que opera a nivel de puente (*bridge*) con datagramas de nivel de enlace (normalmente Ethernet). TUN/TAP funcionan como dispositivos de comunicación virtuales del núcleo del sistema operativo encargados de recoger datos del nivel de aplicación y enviarlos a la Red y viceversa. Cuando el proceso *chilli* comienza se habilita una interfaz TUN y se llama a un *script* de configuración externo opcional. Esto es, las interfaces TUN/TAP son interfaces *software*, solo existen en el núcleo del sistema y se utilizan para realizar funciones de red a nivel de espacio de usuario. Una vez implantadas actúan como cualquier otra interfaz del sistema. Se les pueden asignar direcciones IP, su tráfico puede ser analizado, se puede determinar qué caminos le apuntan, añadirles directivas de cortafuegos...

3.4.2. Configuración

El archivo principal de CoovaChilli es `/usr/local/etc/chilli.conf` el cual incluye otros tres: `main.conf`, `hs.conf` y `local.conf`. Los dos primeros son creados por los scripts del shell existentes en el archivo `functions` basándose en las configuraciones de otros archivos mencionados a continuación y obteniendo algunas configuraciones del servidor RADIUS y otras URL. El archivo `local.conf` está reservado para configuraciones de lugares específicos. Las configuraciones por defecto que son establecidas en `chilli.conf` se encuentran en `/usr/local/etc/chilli/defaults`. Los valores existentes en este último archivo pueden sobreescibirse durante la inicialización de `chilli` si existe otro archivo en `/usr/local/etc/chilli/config`, que no es más que una copia del archivo `defaults` sobrescribiendo los valores de los atributos que se desea cambiar. Cada vez que se enciende el dispositivo en el que está instalado CoovaChilli, el script presente en `/usr/local/etc/init.d/chilli` se ejecuta tomando las configuraciones de los archivos `defaults` y `config` mencionados anteriormente, ejecutando también el script `/usr/local/etc/chilli/functions` para asistir en esta configuración y la de otros archivos relevantes para el funcionamiento del programa, creando de este modo los respectivos archivos `main.conf`, `hs.conf` y `local.conf`.

Adicionalmente, CoovaChilli implementa un servidor Web mínimo destinado a servir contenido en el directorio `/etc/local/etc/chilli/www/`, lugar en el que puede ubicarse un servicio de portal cautivo sencillo. Por defecto, en este directorio se implementa un portal cautivo con pantalla de espera que redirige a un formulario ensamblado con plantillas HTML y archivos de Web scripts *Common Gateway Interface* (CGI) para contenido dinámico. Estos archivos (con la extensión `.chi`) son procesados con el software Haserl [18], por lo que si se usa la configuración por defecto de CoovaChilli este programa ha de estar instalado.

Aunque CoovaChilli cuenta con numerosas opciones, en este TFG se usa y modifica tan solo un subconjunto de ellas, asignándoles valores en los archivos `defaults` y `config`:

- `HS_WANIF`: especifica la interfaz de red que cuenta con acceso a internet, habitualmente la interfaz Ethernet del dispositivo, `eth0`.
- `HS_LANIF`: interfaz a la que se conectan los clientes, habitualmente la interfaz WiFi del dispositivo, `wlan0`.
- `HS_NETWORK`: dirección IP de la red a la que se conectan los clientes, por ejemplo 192.160.10.0. Habitualmente esta dirección es la determinada al configurar una IP estática para `wlan0` en el archivo `/etc/network/interfaces` del sistema operativo.

- **HS_UAMLISTEN**: la IP del dispositivo de red al que se conectan los clientes. Por supuesto, debe pertenecer al rango IP de la red especificada en el atributo anterior, siguiendo el ejemplo dicha IP sería 192.168.10.1.
- **HS_UAMALLOW**: redes y dominios en los que los usuarios no autenticados sí tienen permitida la navegación. Habitualmente solo se permite la red local desde la que también se serviría el portal cautivo (el mismo valor que el atributo **HS_NETWORK**), aunque para otros servicios de portal cautivo más avanzados podría permitirse el acceso a otros dominios, por ejemplo a la URL de la correspondiente API de Facebook para la autenticación por esta red social o al dominio de PayPal para realizar pagos de tarifas.
- **HS_UAMSECRET**: clave secreta con la que se cifran las credenciales que se envíen desde el servidor Web del portal cautivo hacia CoovaChilli.
- **HS_UAMFORMAT**: ubicación y puerto del servidor Web que proporcionaría el portal cautivo.
- **HS_UAMHOMEPAGE**: ubicación y puerto a la que se redirigirían las peticiones Web de los clientes al conectarse a la red. Habitualmente, esto es una página de bienvenida que luego redirige hacia el contenido servido en la ubicación especificada en **HS_UAMFORMAT**.
- **HS_SSID**: SSID de la red.

3.4.3. Interfaz *JavaScript Object Notation*

JavaScript Object Notation (JSON) es perfil de transmisión de datos desde un servidor Web a un navegador. Tiene menos datos de cabecera que el *Xtensible Markup Language* (XML) habitualmente utilizado en *Asynchronous JavaScript And XML*(AJAX) y otros servicios y por ello se ha considerado una alternativa a este y ha alcanzado gran popularidad. CoovaChilli usa JSON para realizar el control de usuarios de forma que pueda utilizarse un portal cautivo de cualquier tipo e incluso en servidores distintos al incorporado en el programa a los que pueda acceder el cliente, previa habilitación del acceso al dominio correspondiente en el atributo **HS_UAMALLOW** del archivo de configuración. Un usuario puede autenticarse a través de ella, obtener el estado de su conexión o desconectarse del portal cautivo.

Para comunicarse con la interfaz JSON de CoovaChilli se utiliza una biblioteca JavaScript, instalada por defecto en `/usr/local/etc/chilli/www/ChilliLibrary.js`. Esta biblioteca crea el objeto global *chilliController*, que ha de inicializarse con los

valores necesarios según nuestra configuración. El portal cautivo que se implemente pasa a utilizar los métodos expuestos por el objeto *chilliController* para comunicarse con CoovaChilli, enviándole peticiones HTTP GET a este y recibiendo respuestas en el formato JSON.

El objeto *chilliController* puede enviar los siguientes órdenes a CoovaChilli:

- *logon*: intenta un login utilizando CHAP. La respuesta contendrá los datos de sesión y los datos iniciales de contabilidad. Es llamado mediante el método *logon(username, password)*.
- *logoff*: termina la sesión actual. Es llamado mediante el método *logoff()*.
- *status*: proporciona los datos de contabilidad más recientes. Es llamado mediante el método *refresh()*.

Cuando un cliente es autorizado, el objeto *chilliController* consulta periódicamente la información de contabilidad, enviando órdenes de *status* para actualizar los datos (orden *autorefresh*). CoovaChilli facilita una plantilla de código en su página Web que puede alojarse en el HTML de un portal cautivo que utilice esta interfaz (Bloque de código 3.2).

```

<script src="http://my.host.com/ChilliLibrary.js"></script>
<script>
// The included script creates a global chilliController object
// If you use non standard configuration, define your configuration
chilliController.host = "10.0.0.1"; // Default is 192.168.182.1
chilliController.port = 4003 ; // Default is 3990
chilliController.interval = 60 ; // Default is 30 seconds

// then define event handler functions
chilliController.onError = handleErrors;
chilliController.onUpdate = updateUI ;

// when the reply is ready, this handler function is called
function updateUI( cmd ) {
    alert ( 'You called the method' + cmd +
    '\n Your current state is =' + chilliController.clientState ) ;
}

// If an error occurs, this handler will be called instead
function handleErrors ( code ) {
    alert ( 'The last contact with the Controller failed. Error code =' + code );
}
// finally, get current state
chilliController.refresh();
</script>

```

Código 3.2: Plantilla de código de CoovaChilli para el manejo de la interfaz JSON

El *chilliController* es un objeto global creado por el script incluido al principio. Por defecto, está configurado para contactar con CoovaChilli en 192.168.182.1:3990 cada 30 segundos. Si deseamos valores diferentes, pueden modificarse antes de llamar a cualquier método tal y como se ve en el bloque de código. Tras configurar el objeto *chilliController* se debe llamar al método *refresh()* para determinar el estado actual del cliente.

CoovaChilli ha implementado los siguientes atributos para el objeto ChilliController. Al ser una interfaz que continúa expandiéndose y desarrollándose existen otros atributos aún no implementados pero presentes en la documentación que se han omitido en este TFG.

- *host*: la dirección IP del controlador de acceso de CoovaChilli.
- *port*: el puerto HTTP para las peticiones JSON.
- *interval*: intervalo medido en segundos. Mientras el usuario está autenticado, los

datos de contabilidad y de sesión se actualizan mediante *polling* a CoovaChilli a este intervalo.

- *language*: el idioma preferido para el mensaje de respuesta (código de idioma de dos letras ISO, por ejemplo ‘en’).
- *clientState*: estado del terminal. *UNKNOWN* significa que aún no se ha recibido información desde CoovaChilli. Otros valores posibles son *NOT_AUTHORIZED*, *AUTHORIZED* y *AUTH_PENDING*.
- *command*: el último comando enviado a CoovaChilli (que podría estar pendiente): *logon*, *logoff*, *refresh*, *autorefresh*.
- *sessionId*: identificador único de sesión generado por CoovaChilli (*Acct-Session-Id*). Se utiliza para asegurar que las propiedades del miembro de la sesión y el miembro de la contabilidad pertenezcan a la misma sesión.
- *message*: mensaje a ser mostrado al usuario. Puede ser el Mensaje Respuesta RADIUS o un mensaje generado por CoovaChilli.
- *redir*: este atributo expone datos leídos en la URL de la redirección inicial que se realiza hacia el portal cautivo indicado en el atributo de configuración de *HS_UAMHOMEPAGE*. Solo existe si la página especificada en dicho atributo incluye un script manejador de *chilliController*.
- *originalURL*: URL pedida originalmente por el cliente antes de su redirección.
- *redirectionURL*: URL a la que se va a redirigir a continuación.
- *macAddress*: atributo RADIUS *Calling-Station-Id*. La dirección MAC del dispositivo cliente.
- *ipAddress*: atributo RADIUS *Framed-IP-Address*.
- *location*: este atributo contiene información básica sobre la ubicación.
- *name*: atributo RADIUS *WISPr-Location-Name*, el nombre de la ubicación también almacenada en el atributo locationname o radiuslocationname de chilli.conf.
- *session*: expone los atributos RADIUS recibidos en el paquete RADIUS *access-attempt*. Estos atributos son fijos durante una sesión.
- *startTime*: el momento de inicio de sesión de CoovaChilli. Es un objeto Date de ECMAScript. No es un atributo RADIUS.
- *sessionTimeout*: temporizador para la finalización de sesión.

- *idleTimeout*: temporizador para el tiempo de inactividad.
- *accounting*: valores de contabilidad de Volumen/Tiempo. Van cambiando durante la sesión.
- *sessionTime*: atributo RADIUS *Acct-Session-Time*. La duración de la sesión.
- *idleTime*: tiempo de inactividad calculado por CoovaChilli (el tráfico desde o hacia CoovaChilli es ignorado). No es un atributo RADIUS, pero el objeto controlador lo utiliza para planificar la siguiente actualización de datos tras una desconexión por IdleTimeout.
- *inputOctets*: atributo RADIUS *Acct-Input-Octets*.
- *outputOctets*: atributo RADIUS *Acct-Output-Octets*.
- *inputGigawords*: atributo RADIUS *Acct-Input-Gigawords*.
- *outputGigawords*: atributo RADIUS *Acct-Output-Gigawords*.

CoovaChilli ha implementado los siguientes manejadores de eventos:

- *onUpdate*: función llamada cuando el objeto *chilliController* se actualiza. Las actualizaciones ocurren cuando nuevos datos se reciben desde el controlador. Esto puede ocurrir después de que un método se llame explícitamente (*logon*, *logoff*, *refresh*) o automáticamente a intervalos determinados por el atributo *interval*, cuando se autoriza al cliente (*autorefresh*). La función recibe el nombre de la orden que causó la actualización como argumento.
- *onError*: función llamada cuando no puede obtenerse una respuesta JSON correcta del controlador (se ha caído el enlace inalámbrico, la sintaxis JSON es incorrecta...). La función recibe un código de error como argumento.

Como se ha mencionado anteriormente, los órdenes se envían a CoovaChilli mediante peticiones HTTP. El parámetro GET *lang* se usa en *logon* para pasar el idioma preferido. El campo *version* se utiliza para controlar las diferentes versiones del protocolo JSON de CoovaChilli. Si un parámetro para hacer *callbacks* existe, CoovaChilli pone el texto de salida JSON entre paréntesis y lo adjunta con la función de *callback*. La salida sería como una llamada a una función con el objeto JSON pasado como parámetro.

Para el logon, siguiendo el ejemplo del bloque de código que configuró el objeto *chilliController* en apartados anteriores, cuando se llama al método *logon()* del Controlador el objeto genera un desafío CHAP aleatorio (string hexadecimal) y realiza la petición `http://192.168.182.1:3990/json/logon?username=XXXX&`

chapchallenge=YYYY&chappassword=0123456789abcdef&lang=EN. CoovaChilli responde con un objeto en formato JSON del bloque de código 3.3:

```
{  
    "version" : "1.0",  
    "clientState" : 1 ,  
    "sessionId" : "4662e92b0000000e" ,  
    "message" : "You're now connected" ,  
    "location" : {  
        "name": "Coova labs"  
    },  
  
    "redir" : {  
        "macAddress" : "00-30-1B-B5-03-6B",  
        "originalURL" : "http://my.yahoo.com/",  
        "redirectionURL" : "http://www.coova.org/welcome.php",  
        "ipAddress" : "192.168.182.47"  
    },  
  
    "session" : {  
        "startTime" : 137550720,  
        "terminateTime" : 13756072,  
        "sessionTimeout" : 3600,  
        "idleTimeout" : 240,  
        "maxInputOctets" : 100000000,  
        "maxOutputOctets" : 100000000,  
        "maxTotalOctets" : 100000000,  
        "bandwidthMaxDown" : 1000000,  
        "bandwidthMaxUp" : 1000000  
    },  
  
    "accounting": {  
        "sessionTime" : 2,  
        "idleTime" : 0,  
        "inputOctets" : 0,  
        "outputOctets" : 0,  
        "inputGigawords" : 0,  
        "outputGigawords" : 0  
    }  
}
```

Código 3.3: Respuesta de la interfaz JSON a un logon()

Si por el contrario la autorización no tiene éxito la respuesta JSON será la que se muestra en el bloque de código 3.4.

```
{
  "version": "1.0",
  "clientState": 0,
  "message": "This username does not exist",
  "location" : { "name": "My HotSpot" }
}
```

Código 3.4: Respuesta de la interfaz JSON a un logon() fallido

Para el logoff, cuando se llama al método de desconexión se realiza la petición a `http://192.168.182.1:3990/json/logoff?lang=en&callback=myfunc`. La respuesta JSON pasa a tener esta forma: `myfunc ("version": "1.0", "clientState": 0)`.

Para el *Refresh*, cuando se llama al método de actualización se realiza la petición a `http://192.168.182.1:3990/json/status?lang=en`. La respuesta JSON en este caso se muestra en el bloque de código 3.5.

```
{
  "version" : "1.0",
  "clientState" : 1 ,
  "sessionId" : "4662e92b0000000e" ,
  "accounting": {
    "sessionTime" : 1230,
    "idleTime" : 240,
    "inputOctets" : 2912981 ,
    "outputOctets" : 51498511,
    "inputGigawords" : 0,
    "outputGigawords" : 0
  }
}
```

Código 3.5: Respuesta de la interfaz JSON a un refresh()

No es necesario repetir las propiedades fijas de sesión. El atributo `sessionId` se usa para asociar los nuevos valores de contabilidad recibidos con los valores de sesión recibidos con la orden `logon` inicial.

Si el cliente no está autorizado se envía esta respuesta a las peticiones la respuesta sería la del bloque de código 3.6.

```
{  
  "version" : "1.0",  
  "clientState" : 0  
}
```

Código 3.6: Respuesta de la interfaz JSON a una petición no autorizada

3.5. El entorno node.js para programación de back-end

El JavaScript junto a HTML y CSS son los tres pilares básicos de la programación *front-end* (cliente). Los servidores (*back-end*), suelen estar programados en otros lenguajes como *Hypertext Preprocessor* (PHP), Java [19] o Python [20]... Poco a poco el Javascript también se ha venido usando para programar los *back-ends* debido a la aparición de Node.js. Esto se debe a la iniciativa de intentar unificar todo el desarrollo Web en un único lenguaje de programación introduciendo el paradigma *JavaScript en todas partes*.

El Node.js tiene una arquitectura dirigida por eventos capaz de manejar la entrada-salida asíncrona, en un esfuerzo por aumentar la escalabilidad y la tasa de transmisión en aplicaciones Web con gran cantidad de operaciones de este tipo o aplicaciones Web en tiempo real. Su funcionalidad se basa en bibliotecas JavaScript conectadas entre ellas y al sistema operativo por medio de *bindings* en C++. Utiliza una combinación del motor JavaScript V8 de Google [21], un bucle de eventos y una API de entrada-salida de bajo nivel. El motor V8 fue desarrollado para Google Chrome, escrito en el lenguaje C++. Se encarga de compilar el código JavaScript a código máquina nativo en lugar de interpretarlo en tiempo real. El bucle de eventos se implementa en un único hilo de ejecución que atiende llamadas de entrada-salida no bloqueantes, permitiendo un número elevado de conexiones concurrentes sin incurrir en los costes de recursos del cambio de contexto entre hilos. Mediante el patrón de diseño software *observador*, cada función que realice una operación de entrada-salida debe usar un *callback*, una función pasada como parámetro que se ejecuta al terminar dicha operación.

Con Node.js es muy sencillo implantar un servidor Web básico que responda a cualquier petición utilizando tan solo una de sus funcionalidades incluidas, el paquete HTTP, tal como se muestra en el código del bloque de código 3.7.

```

// Load HTTP module
var http = require("http");

// Create HTTP server and listen on port 8000 for requests
http.createServer(function(request, response) {

    // Set the response HTTP header with HTTP status and Content type
    response.writeHead(200, {'Content-Type': 'text/plain'});

    // Send the response body "Hello World"
    response.end('Hello World\n');
}).listen(8000);

// Print URL for accessing server
console.log('Server running at http://127.0.0.1:8000/');

```

Código 3.7: Implementando un servidor web con Node.js

Este sencillo código, al ejecutarse en Node.js, implementa una respuesta de texto plano *Hello World* al recibir cualquier petición HTTP en la dirección IP de *loopback* del sistema, 127.0.0.1 en IPv4, y el puerto 8000.

3.5.1. El gestor de paquetes npm

Como parte del entorno de Node.js se incluye *npm*, un gestor de paquetes JavaScript utilizado para instalar bibliotecas adicionales almacenadas en el registro npm, un repositorio online de paquetes públicos. Permite tanto instalar como distribuir módulos y gestionar las dependencias de un proyecto determinado desde la línea de órdenes, dependencias que luego pueden utilizarse en el proyecto mediante el método *require()*. Puede gestionar paquetes JavaScript instalados globalmente. Mediante un archivo *package.json* ubicado habitualmente en la raíz de un proyecto Node.js, *npm* puede instalar todas las dependencias necesarias con una sola orden, atendiendo al rango de versiones posibles para evitar incompatibilidades.

3.5.2. El *framework Express*

Express, también referido como *Express.js*, es un *framework* para Node.js. Instalable mediante *npm*, está diseñado para desarrollar tanto aplicaciones Web como API y se ha convertido en el *framework* de servidor estándar de facto para Node.js. Intenta ser

mínimo a la par que flexible, pudiendo implantar un servidor sencillo en unas pocas líneas de código e ir implementando las funcionalidades requeridas según sea necesario.

Express representa parte del *back-end* de lo que se ha dado a conocer como el *MEAN stack*, un conjunto de subsistemas de software basados en JavaScript y constituido por *MongoDB* [22], un sistema de bases de datos no-SQL, *Angular*, un *framework* para implantar patrones de diseño software de *Modelo-Vista-Controlador* (MVC) en aplicaciones Web de página única, *Express* y el propio Node.js.

Para implantar la funcionalidad del código del ejemplo anterior, esta vez haciendo uso del *framework* Express, puede escribirse el código del bloque de código 3.8 en un archivo JavaScript llamado por ejemplo *app.js* y luego ejecutar este archivo en la línea de órdenes mediante la orden *node app.js*.

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
    res.send('Hello World!');
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000!');
});
```

Código 3.8: Servidor web implementado con Express

Las dos primeras líneas realizan la importación del módulo Express y crean la aplicación en la variable *app*, que contendría todos los métodos para recibir las peticiones HTTP, configurar lógicas adicionales y otros controles de comportamiento de la aplicación Web. La orden *app.get()* representa la definición de un camino; especifica una función *callback* como segundo parámetro que se ejecutaría cuando haya una petición HTTP GET con la ruta determinada en el primer parámetro (relativa a la raíz). La función *callback* tiene objetos de petición y respuesta como argumentos y llama al método *send()* en la respuesta para devolver la *string* *Hello World*. El bloque final inicia el servidor en el puerto 3000 e imprime un log en la consola. Si se visita *localhost:3000* en el navegador local después de ejecutar este archivo con Node.js se vería la respuesta.

3.5.3. El módulo *formidable*

El módulo *formidable* de análisis sintáctico (*parsing*) de datos de tipo formulario, especializado en subida de archivos a servidores, incluyendo subida múltiple. Con el uso de este paquete *npm* es posible gestionar los datos enviados al servidor desde el portal cautivo. Para poder hacer uso del módulo primero ha de incluirse al archivo que ejecutamos con *node* mediante la orden *require('formidable')*, como también sucedía con *express* y con el resto de módulos. Su API funciona de la siguiente manera:

- Se crea un objeto formulario entrante y se asocia a una variable con la siguiente asignación `var form = new formidable.IncomingForm();`.
- Tras esto, pueden especificarse detalles como el directorio en el se suben los archivos con el siguiente código: `form.uploadDir = "/ruta/hacia/el/archivo";`.
- Para realizar el parsing de una petición que incluya datos de tipo formulario se usa la siguiente función, donde *request* es la petición entrante: `form.parse(request);`.

Los eventos se manejan con la función *on*, que incluye una *string* con el nombre del evento como primer parámetro y una función *callback* como segundo parámetro, a ejecutar cuando dicho evento ocurre. Ejemplos de eventos son '*progress*', '*file*', '*error*' y '*end*'. Un ejemplo de manejo de eventos que gestionaría qué ocurre cuando se ha recibido una petición y todos los archivos ya han terminado de llegar al disco se muestra en el bloque de código 3.9.

```
form.on('end', function() {
  // Lugar ideal para enviar una respuesta.
});
```

Código 3.9: Función de callback para cuando termina una transferencia

3.5.4. El *middleware Body-Parser*

Realiza análisis sintáctico los cuerpos de las peticiones entrantes, a cuyos contenidos puede accederse después mediante el atributo *req.body*. Tras incluirse en el proyecto, por ejemplo bajo la variable *bodyParser*, puede ser llamado desde el objeto *express()* para utilizar sus funciones mediante la orden *use()*. En el bloque de código 3.10. se presenta un ejemplo genérico para habilitar el parsing de JSON y URL-encoded.

```

var express = require('express')
var bodyParser = require('body-parser')

var app = express()

// parse de application/json
app.use(bodyParser.json())

// parse de application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }))

app.use(function (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.write('you posted:\n')
  res.end(JSON.stringify(req.body, null, 2))
})

```

Código 3.10: Habilitando opciones de parsing

3.6. Programación del front-end

El *front-end* implica la producción de contenido, apariencia y comportamiento de una página o aplicación Web de forma que el usuario pueda verla e interactuar con ella. En la actualidad las tecnologías usadas para el desarrollo *front-end* se hallan en proceso de cambio y actualización constante, complicado aún más por la aparición en escena de los diferentes dispositivos actuales capaces de acceder a estas aplicaciones Web, con diferentes tamaños de pantalla y distinto grado de soporte de las tecnologías disponibles.

Las herramientas básicas del desarrollo front-end la forman un conjunto de tres tecnologías. HTML, CSS y JavaScript.

3.6.1. El HTML y el CCS

La última versión de HTML es conocida como HTML5 y fue publicada en octubre de 2014. Esta versión contiene nuevas formas de manejar elementos, sobre todo en el ámbito de la multimedia (como el audio y el vídeo), y nuevos descriptores y posibilidades de metadatos con los que se busca conseguir una mayor interoperabilidad entre sistemas y una mayor semántica de la Web. Dado que introduce varias API y marcado para aplicaciones Web complejas y también ha sido diseñado teniendo en mente dispositivos de bajo consumo, su uso también está planteado para aplicaciones móviles multiplataforma.

El CSS define el aspecto o la presentación de la página Web. Un documento CSS consiste en listar unas reglas de visualización —color, tipografía, fondo, bordes— para cada elemento de marcado usado después en HTML. De esta forma, CSS trata de separar el contenido y la presentación del mismo, de forma que si se quiere cambiar el diseño gráfico de un sitio Web no haya que modificar todos y cada uno de los documentos HTML existentes, sino un número mucho más reducido de documentos CSS que estén referidos en estos.

Al mismo tiempo, la variedad de módulos existentes actualmente trata de dar un mayor dinamismo y responsividad a las páginas Web, incluyendo animaciones, diseño en múltiples columnas que se adaptan a los tamaños de pantalla...

3.6.2. El javascript: bibliotecas, API y complementos

JavaScript es un lenguaje de programación interpretado de alto nivel que define el comportamiento de una aplicación o página Web (interactividad y ejecución de programas online). Todos los navegadores modernos soportan una implementación parcial o completa de la especificación ECMAScript [23].

El *Document Object Module* (DOM) es la especificación de la jerarquía de los elementos usados por HTML, que puede entenderse como un diagrama de árbol en el que cada objeto representa una parte del documento. JavaScript manipula el DOM pudiendo acceder a cada elemento y realizando cambios en él que luego se verían reflejados en la visualización del sitio Web, a efectos creando HTML dinámico en el que pueden alterarse los elementos y atributos, cambiar el estilo CSS, reaccionar a eventos de la página o crear eventos nuevos. En la Figura 3.6 se muestra un ejemplo gráfico del DOM.

Aunque en este TFG se trabaja con el estándar de JavaScript llamado ECMAScript 2016, en la actualidad la última versión es ECMAScript 2017, lanzada en Junio de este mismo año. A continuación se detallan algunos elementos de importancia que se utilizaron en la elaboración de este TFG. Su lugar de uso se detalla en el siguiente capítulo del presente documento.

Web Real-Time Communications

WebRTC es una tecnología que permite a las aplicaciones Web capturar y transmitir datos de audio y vídeo, así como intercambiar datos entre navegadores sin un intermediario. Los estándares que componen WebRTC hacen posible compartir datos y realizar videoconferencias *peer-to-peer*, sin que sea necesario que los usuarios instalen

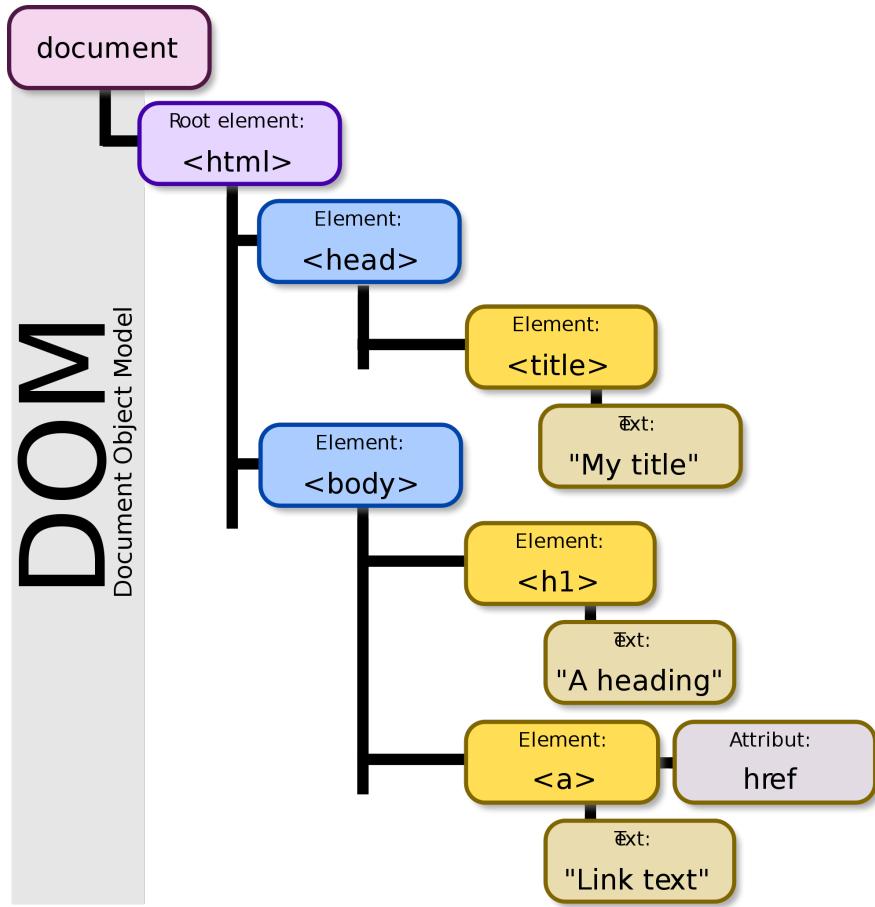


Figura 3.6: Esquema general del DOM.

Fuente: https://en.wikipedia.org/wiki/Document_Object_Model

complementos u otros elementos *software* de terceros. Junto a las API *MediaStream Recording* y *Media Capture and Streams* proporcionan a la Web una gama de capacidades multimedia que incluyen captura y grabación de multimedia en disco, conferencias de audio y vídeo, intercambio de archivos e incluso gestión de identidad.

Es un estándar actualmente en desarrollo por la *World Wide Web Consortium* (W3C) y el IETF. Para transferir los datos multimedia se utiliza el protocolo *Realtime Transport Protocol* (RTP).

El *getUserMedia()* es el método principal mediante el que se solicitan al usuario los permisos necesarios para usar un dispositivo de entrada multimedia (como la cámara o el micrófono). Produce una interfaz *MediaStream* consistente en las pistas de los tipos de multimedia solicitados. Generalmente se usa accediendo a la interfaz *MediaDevices* con la siguiente orden

```
navigator.mediaDevices.getUserMedia(constraints).then(...)
```

siendo *constraints* el parámetro que indica el tipo de multimedia que se solicita y *then*

la función que contendría los callbacks a ejecutar en caso de éxito o error según la *Promise API*.

La API *MediaStream Recording* está muy relacionada captura los datos generados por el objeto *MediaStream* para su análisis, procesado o guardado en disco. Se basa en una única interfaz, *MediaRecorder*, que se encarga de obtener los datos del *MediaStream* mediante una serie de eventos. El proceso de grabación es muy sencillo:

- Se crea el *MediaStream* fuente de los datos multimedia.
- Se crea el objeto *MediaRecorder*, especificando la transmisión de origen y otras opciones pertinentes.
- Configurar el gestor *MediaRecorder.ondataavailable* para el evento *dataavailable*, que sería llamado cada vez que haya datos disponibles.
- Controlar el proceso de grabación mediante los métodos disponibles, por ejemplo con *MediaRecorder.start()* para comenzar y *MediaRecorder.stop()* para detener la grabación.
- El evento *dataavailable* tiene un atributo *data* que contiene la información multimedia adquirida, pudiendo procesarse como se requiera.

La transmisión de datos en Web puede verse enormemente beneficiada por JSON soportado de forma nativa desde ECMAScript 5, aunque debido a su popularidad son ya muchos los lenguajes que han incluido generadores y analizadores de su sintaxis.

AJAX

El AJAX utiliza peticiones HTTP para que las aplicaciones Web puedan enviar datos al servidor y recibir datos del mismo en segundo plano, sin interferir en el comportamiento y la visualización de la Web. De este modo, junto a la manipulación del DOM realizada por JavaScript, se permite cambiar el contenido dinámicamente con información procedente del servidor sin tener que actualizar la totalidad de la página. En la actualidad se utiliza JSON en lugar de XML por su integración nativa con JavaScript.

jQuery

A menudo, trabajar con determinadas funciones de JavaScript (manipulación del DOM, manejar eventos, usar AJAX) puede generar documentos difíciles de leer por su extensión

y lo confuso de su sintaxis. La jQuery es una biblioteca JavaScript gratuita y de código abierto que trata de simplificar todos estos procesos del lado del cliente. Permite a los desarrolladores crear complementos con los que añadir una capa más de abstracción a las interacciones de bajo nivel siguiendo un enfoque modular. Actualmente se halla en su versión 3. Para utilizarlo en un sitio Web ha de ser incluido como se incluiría un archivo JavaScript cualquiera en un archivo HTML. La popularidad de jQuery radica en cómo simplifica la lectura y el análisis del código. Por ejemplo, si desde un archivo JavaScript deseamos realizar una petición AJAX de tipo GET al archivo del servidor *send-ajax.php* se tendría que escribir el bloque de código 3.11 (cliente).

```
// Inicializar la petición HTTP.
var xhr = new XMLHttpRequest();
xhr.open('get', 'send-ajax-data.php');

// Monitorizar los cambios de estado de la petición.
xhr.onreadystatechange = function () {
    var DONE = 4; // readyState = DONE significa que la petición se ha hecho.
    var OK = 200; // status 200 es un retorno con éxito.
    if (xhr.readyState === DONE) {
        if (xhr.status === OK) {
            console.log(xhr.responseText); // Aquí estaría la respuesta del
                servidor
        } else {
            console.log('Error: ' + xhr.status); // Ha ocurrido un error durante la
                petición.
        }
    }
};

// Enviar la petición a send-ajax-data.php
xhr.send(null);
```

Código 3.11: Petición AJAX normal

El mismo ejemplo utilizando jQuery, donde el carácter \$ representa el objeto jQuery a través del cual se accede a todos sus usos, se muestra en el bloque de código 3.12.

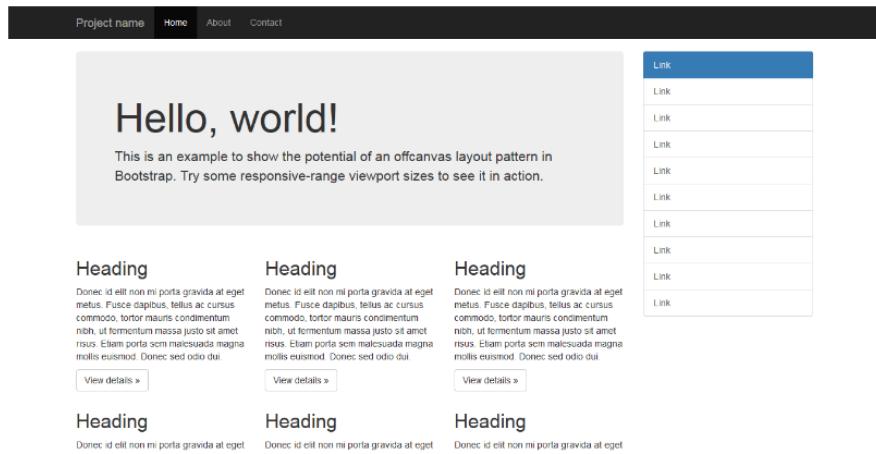


Figura 3.7: Ejemplo de diseño con Bootstrap.

Fuente: <http://getbootstrap.com/docs/4.0/examples/offcanvas/>

```
$.get('send-ajax-data.php')
  .done(function(data) {
    console.log(data);
  })
  .fail(function(data) {
    console.log('Error: ' + data);
});

```

Código 3.12: Petición AJAX con jQuery

Bootstrap

Bootstrap es un *framework* que simplifica la carga de trabajo, adaptada para el diseño de aplicaciones móviles. Contiene plantillas basadas en HTML y CSS para tipografías, formas, botones, navegación y otros componentes, así como algunos componentes opcionales de JavaScript. Utiliza un sistema de rejilla o *grid* para ayudar en el diseño de la web, proporcionando así un sencillo sistema de filas, cada una de ellas subdividida hasta en 12 columnas, con el que puede establecerse la proporción visual de los elementos de la pantalla. En la Figura 3.7 se muestra un ejemplo de uso del sistema de rejilla en Bootstrap, con un elemento sirviendo de cabecera y varios contenedores alineados bajo el mismo en grupos de tres.

4. Análisis previo y funcional

Una vez revisada la idea general que se busca en este TFG y las tecnologías de las que hacemos uso para lograrlo, en este capítulo mostramos el análisis previo (requisitos) y el funcional (visión de alto nivel de los distintos módulos funcionales principales de que consta el sistema).

4.1. Objetivo y análisis de requisitos

El objetivo de nuestro sistema de sensado móvil colaborativo, asimétrico con beneficio para todos los participantes, es implantar un sistema capaz de ofrecer un servicio de conexión WiFi a Internet a cambio de datos procedentes del sensor de audio (micrófono) del dispositivo móvil que utilice el cliente (aunque no limitado a este tipo de dispositivos). Para ello utilizamos una combinación de hardware y aplicaciones software ya existente, programando desde cero los módulos restantes.

Los requisitos del sistema son los siguientes:

- Soporte para múltiples navegadores y sistemas operativos, incluyendo móviles.
- Mínima interacción con el usuario: sólo se requiere de la provisión de permisos la primera vez que se utiliza el sistema en un Punto de acceso determinado. Si se cambia de punto de acceso es necesario volver a dar permisos.
- Que proporcione más datos aparte de los del sensor de audio.
- Proveer distintos modos de conexión a Internet.

El primer requisito podría alcanzarse de forma sencilla. Dado que a priori vamos a emplear tecnología Web basándonos en estándares que todos los navegadores modernos deben estar preparados para procesar no se requeriría instalar ninguna aplicación o complemento de navegador adicional en los clientes del servicio. Además, se tendría en cuenta el diseño Web responsive, de forma que la visualización del contenido se adapte al tamaño y características del dispositivo. Sin embargo, existen ciertas particularidades.

Los estándares utilizados para implantar la funcionalidad de captura de sonido, como WebRTC y MediaStream Recording, están aún abiertos y tienen poco tiempo, por lo que solo están soportados en versiones muy recientes de los navegadores o incluso, hasta hace poco tiempo, únicamente en sus versiones experimentales. Por ejemplo, Safari, el navegador Web de Apple por defecto para sus ordenadores con MacOS y los dispositivos iOS, ha comenzado a trabajar con los estándares WebRTC y relacionados hace tan solo un año, anunciando su compatibilidad para la versión 11 del navegador que salió en Septiembre de 2017. Por el momento, para Safari solo puede encontrarse una compatibilidad preliminar en su versión para desarrolladores, Safari Technology Preview [24].

En cuanto al segundo requisito, se plantea que las únicas interacciones que tenga que hacer el usuario con el servicio intermedio de portal cautivo antes de obtener acceso a Internet sea la concesión de permisos de ubicación y captura de audio y otro botón para comenzar el proceso de conexión, representando una ventaja sobre otros sistemas

de portal cautivo donde hay múltiples redirecciones, procesos de creación de cuentas de usuario y de inicio de sesión.

Para alcanzar el tercer requisito se trata de aumentar la usabilidad del servicio más allá de capturar un fragmento de sonido, obteniendo también su ubicación y una marca de tiempo para poder hacer análisis estadísticos, mapeos de niveles de señal 3D en función del tiempo...

El cuarto requisito ha surgido a partir de las pruebas realizadas y el desarrollo iterativo e incremental del sistema. Esto es, inicialmente se especificó únicamente un modo de conexión a Internet debiendo el cliente Web mantener abierta la pestaña del navegador Web en la que se proporcionó el permiso para acceder al micrófono. La aplicación javascript de esa pestaña capturaba un fragmento de audio de 5 segundos cada 3 minutos. La ventaja es que la ofrece al usuario una conexión más estable. La desventaja es que retiene el control del micrófono en modo exclusivo para esa aplicación Javascript. Haciendo desarrollos incrementales se detectó este comportamiento a priori no explícito, por lo que se observó que este comportamiento no sería adecuado para aquellos usuarios que quisieran hacer uso del micrófono a la vez que está conectado a Internet (por ejemplo, mensajería de audio vía aplicaciones de mensajería instantánea). Por ello, en segundo lugar se especificó como requisito previo que pudiera existir un nuevo modo de conexión a Internet. Esto es, ofrecer un tiempo de conexión fijo de 30 minutos a cambio de un único archivo de audio inicial, tras lo cual podría cerrarse la pestaña del navegador de la Aplicación Web y hacer uso libre del micrófono del dispositivo durante este tiempo.

4.2. Análisis previo

El sistema está compuesto de varios elementos que interactúan entre sí. A continuación se ofrece un resumen del funcionamiento de cada elemento, los módulos que lo componen y las tareas asociadas a los mismos:

- *Hardware necesario:* se supone que en el local a monitorizar ya existe un acceso a Internet al cual se puede conectar un Punto de acceso WiFi. Por ello, únicamente es necesario una Raspberry Pi 3 Model B con el sistema operativo Raspbian, un módulo software controlador del módulo WiFi 802.11 (Hostapd) para habilitar su funcionamiento como Punto de Acceso, el software de control de acceso y el servicio de portal cautivo. La configuración de este software para que funcionara adecuadamente llevó mucho tiempo debido a la falta de información o bien información obsoleta sobre estas aplicaciones. Por ello hubo de realizarse muchas

pruebas de ensayo-error hasta que definitivamente pudimos dar con la configuración correcta.

- *Aplicaciones software configurables*: para proveer el acceso a Internet es necesario utilizar las siguientes aplicaciones que ya existen y que se deben configurar adecuadamente. Destacar que esta tarea de configuración no es sencilla, debido a la falta de información veraz, la obsolescencia de la información existente y la dificultad de instalarlos en la Raspberry Pi 3. Por este motivo hubo de realizar múltiples pruebas de ensayo y error hasta dar con la configuración adecuada, cuyas ideas generales proporcionamos a continuación:

- *Control de acceso*:

- *RADIUS*: necesario para la autenticación de los usuarios. Se utilizó el daloRADIUS, para la gestión interna del acceso de usuarios, recibiendo las credenciales proporcionadas por el servicio de portal cautivo y gestionando su conexión.
- *CovaChilli y su entorno*: necesario para redirigir correctamente los paquetes entrantes (autorizados previamente por RADIUS) desde la interfaz inalámbrica a la interfaz cableada y viceversa. Aquí hubo que configurar las ubicaciones de nuestro portal cautivo propio, los ajustes de autenticación de usuarios y habilitar el uso de SSL.
- *Node.js*: necesario para implementar el servidor donde alojar nuestro portal cautivo, recibir y procesar los ficheros de audio generados e instalar servicios de monitorización del mismo.

- *Aplicaciones software desarrolladas desde cero*:

- *Portal cautivo*: enlaza a los usuarios con el control de acceso mediante un sistema de control de usuarios intermedio, realizando también la tarea de captura y almacenamiento de archivos de audio procedentes de los micrófonos de los dispositivos, que es lo que da sentido al sistema global. Su funcionalidad se detalla a continuación. Su funcionalidad básica es:
 - *Servidor Web*: implantado con Node.js, se encargaría de lo siguiente:
 - ◊ Servir la aplicación Web a los usuarios que se conecten a la red.
 - ◊ Recibir y almacenar archivos de audio de los usuarios conectados.
 - ◊ Llevar un control de los usuarios a los que se ha dado acceso a Internet.

- ◊ Informar a los clientes del estado del servidor.
- *Cliente Web*: implantado con , HTML, CSS y JavaScript, y se encarga de lo siguiente:
 - ◊ Ser el punto de entrada de los usuarios al sistema, ofreciendo dos modalidades de conexión diferentes.
 - ◊ Comprobar el estado del servidor.
 - ◊ Adquirir datos de ubicación y marca de tiempo de los usuarios.
 - ◊ Capturar una señal de audio de los usuarios en intervalos constantes.
 - ◊ Enviar la señal de audio al servidor.
 - ◊ Pedir credenciales de usuario al servidor.
 - ◊ Conectar al usuario a Internet con las credenciales recibidas.

4.3. Análisis funcional

La estructura global del sistema podría entenderse como un conjunto de módulos que proporcionen un nivel de abstracción adicional, útil para comprender el sistema. Después, pueden dividirse estos módulos en otros submódulos que representen las funciones desempeñadas por cada uno de ellos individualmente en el sistema global. Muchos de estos submódulos representan instalaciones de *software* completas en sí mismas.

Considerando esta aproximación al sistema podemos representar el mismo como una combinación de tres módulos o capas, todos ellos implantados sobre el *hardware* anteriormente mencionado y interrelacionados entre sí. En la Figura 4.1 se muestra el conjunto de módulos principales considerados.

Un primer módulo de *Usuarios* consiste tanto en la única interfaz que los usuarios del sistema (*Vista*) como el servidor que proporciona dicha interfaz junto a un primer control de usuarios (*Controlador* y una parte del *Modelo* relacionado con las acciones que lleva a cabo el servidor). El segundo módulo es el de *Control de Acceso*, que incluye todo el software dedicado a esta gestión, formado por CoovaChilli y su entorno (*Observador*). Por último se halla el módulo de Control de Red, que incluye todo el *software* que permite a la Raspberry Pi funcionar como Punto de Acceso inalámbrico y la configuración de las diferentes redes que entran en juego en el sistema.

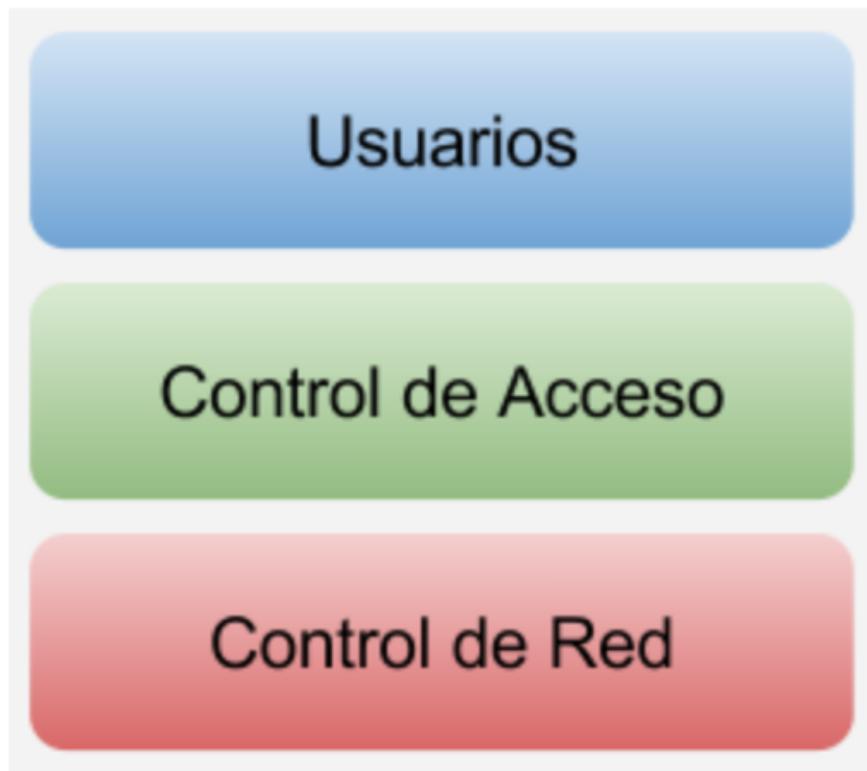


Figura 4.1: Diagrama de módulos funcionales.

Huelga decir que, si bien los clientes del servicio solo interactuarían con el módulo de Usuarios, los otros dos módulos del sistema son accesibles a un administrador, que se encargaría de su correcta configuración inicial y posterior mantenimiento.

4.3.1. Módulo Usuarios

Es el módulo de más alto nivel y el único punto de contacto visible de los usuarios con el servicio. Está compuesto de cinco submódulos; un grupo de tres submódulos que interactúan entre sí para proporcionar el servicio, que son la aplicación Web, el servidor y el control de usuarios; un módulo que hace de interfaz con el módulo inferior de Control de Acceso, biblioteca CoovaChilli; y un último módulo que configura y controla el uso de esta interfaz. En la Figura 4.2 se muestra un esquema básico de las relaciones entre los diferentes submódulos que conforman la capa de Usuarios.

La *Aplicación Web* contiene los datos del portal cautivo al que se redirigen los clientes cuando traten de conectarse a la red inalámbrica. Sus funciones, ya adelantadas en el análisis previo pero ampliadas aquí, son:

- Adquirir los permisos de ubicación y captura de audio del usuario.
- Comprobar el estado del servidor para evitar hacer el proceso si este está lleno

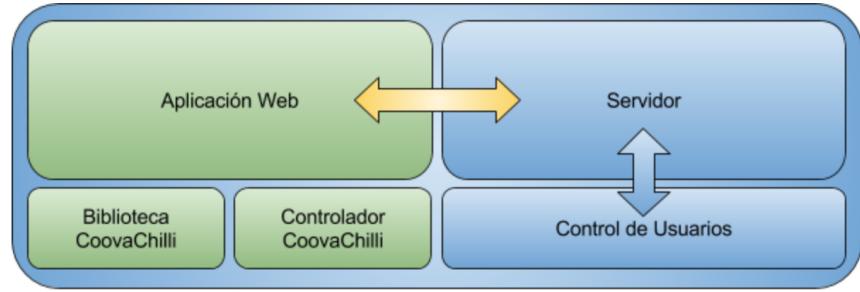


Figura 4.2: Diagrama de submódulos del módulo Usuarios.

(haciendo imposible la conexión al servicio).

- Capturar una muestra corta de dicho audio una única vez o a intervalos regulares, dependiendo del modo de conexión escogido, y enviarla al servidor.
- Recibir las credenciales de usuario enviadas por el servidor, utilizando como nombre de archivo las coordenadas y la marca de tiempo de la última consulta de geolocalización.
- Gestionar la conexión de los usuarios con las credenciales recibidas, todo ello por medio de la configuración y las funciones de la biblioteca CoovaChilli y su controlador.
- Además, está programada para cerrar la conexión si su pestaña del navegador se cierra o se recarga de alguna forma.

En la Figura 4.3 se muestra un diagrama de flujo con la funcionalidad básica del sistema en una primera conexión. En este diagrama se omite la fase de elección de los diferentes modos de conexión.

Todas las consultas que ocurren entre la aplicación Web y el servidor utilizan peticiones GET y POST gestionadas mediante AJAX.

El *Servidor* se encarga de enviar la Aplicación Web en forma de HTML estático a los usuarios que realicen peticiones al mismo, y de comunicarse con los clientes y el Control de Usuarios para aportar la funcionalidad deseada. Sus funciones ampliadas son:

- Servir la Aplicación Web.
- Comprobar el estado del servidor comunicándose con el *Control de Usuarios*, informando a la *Aplicación Web* si el servicio está disponible o por el contrario está lleno, con lo que no podría prestarse el servicio.
- Si recibe un archivo de audio de un cliente que aún no tiene credenciales asignadas, pide al *Control de Usuarios* unas credenciales que se encuentren disponibles en su

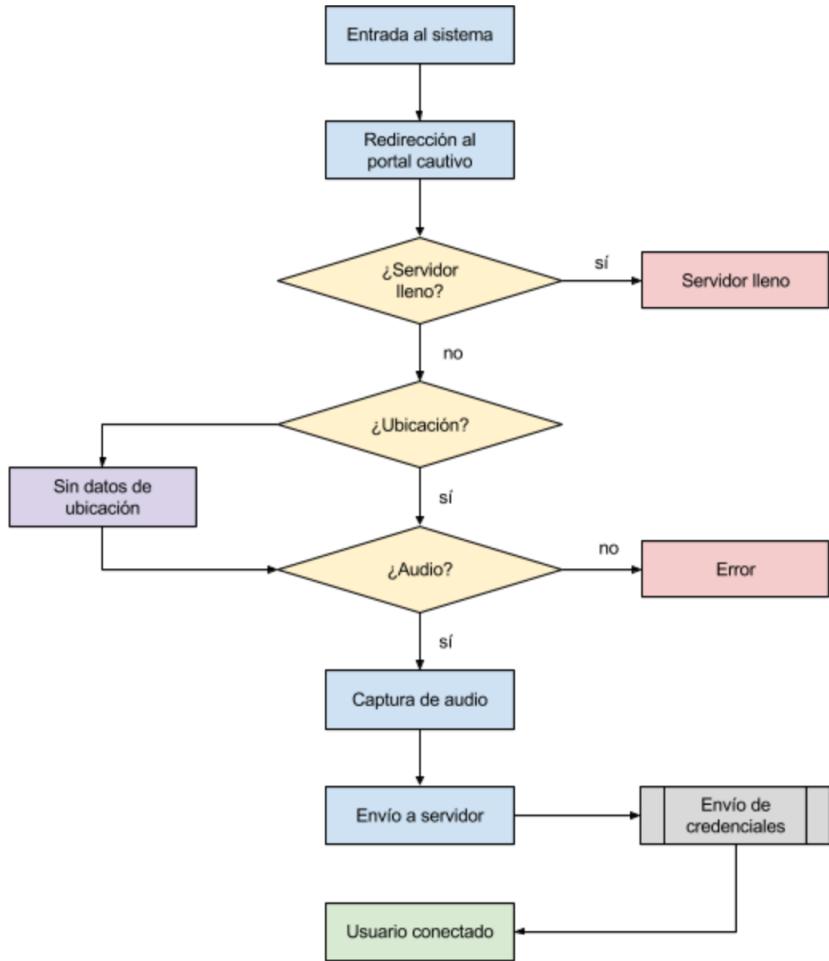


Figura 4.3: Diagrama de la funcionalidad de la conexión a Internet a través del portal cautivo modificado.

base de datos y las almacena a la espera de que la *Aplicación Web* las pida.

- Enviar las credenciales almacenadas a la *Aplicación Web*, informando al *Control de Usuarios* para que marque dichas credenciales como utilizadas.
- Recibir notificaciones de desconexión de usuarios por parte de la *Aplicación Web* y pasar dicha información al *Control de Usuarios* para marcar un usuario como libre en la base de datos.

Para llevar a cabo todas estas tareas se utiliza Node.js, los módulos de npm *Express* para la funcionalidad de servidor, *Formidable* para la gestión de archivos de audio entrantes, *Body-Parser* para leer determinadas variables procedentes de la Aplicación Web y los módulos del núcleo de Node.js, Fs para acceder al sistema de archivos y escribir en él y Path para trabajar con las rutas de archivos y directorios.

El *Control de Usuarios* se encarga de gestionar los usuarios activos e inactivos por

medio de una base de datos sencilla, implementada con dos archivos JSON, uno para cada modo de conexión, y dos archivos JavaScript que controlan a cada archivo JSON por separado actuando como módulos Node.js del *Servidor*. Si la cantidad de usuarios posible se hiciera lo suficientemente grande, este módulo y su archivo JSON podrían ser reemplazados por una aplicación de base de datos más potente (MySQL, MongoDB...). Los archivos base de datos JSON constan de un *array* de usuarios, cada uno de ellos con los siguientes atributos:

- *id*: un número entero igual o superior a 0 que actúa como identificador interno de usuario.
- *username*: el nombre del usuario.
- *password*: la contraseña del usuario.
- *isActive*: una variable booleana, que tiene la asignación *true* si el usuario está ocupado y *false* en caso contrario.

Las funciones de los archivos JavaScript que controlan al archivo JSON anterior son:

- Buscar un usuario inactivo en el archivo de base de datos, devolviendo el identificador y las credenciales de dicho usuario si lo hubiera.
- Asignar usuarios como activos o inactivos escribiendo en el atributo *isActive* correspondiente.
- Hacer externas las credenciales del usuario seleccionado para que el *Servidor* pueda accederlas y hacer sus operaciones.

La biblioteca *CoovaChilli* es un archivo JavaScript denominado **ChilliController.js**, proporcionado por la instalación de CoovaChilli en el directorio **/etc/chilli/www/**. Se encarga de enlazar la Aplicación Web con la interfaz JSON de CoovaChilli creando el objeto global *chilliController*, que puede utilizar los métodos descritos en el capítulo anterior para gestionar la conexión y desconexión de usuarios al servicio.

Por último, el *Controlador CoovaChilli* es un archivo JavaScript de configuración, que modifica los atributos por defecto del objeto *chilliController* creado por la biblioteca CoovaChilli y enlaza los métodos de dicho objeto con acciones concretas de la Aplicación Web.

4.3.2. Módulo Control de Acceso

Este módulo está basado en instalaciones y configuraciones de *software* concernientes la gestión interna del control de acceso al servicio. Puede dividirse en tres submódulos: *CoovaChilli*, *Servidor RADIUS* e *Interfaz Web de control de RADIUS*. Hemos preferido denominar a los dos primeros submódulos con los nombres originales del software utilizado por claridad. La funcionalidad que aprovechamos de estas aplicaciones es la que tienen, en particular, para este TFG. La tarea aquí básicamente es encontrar, tarea no sencilla, la instalación y configuración adecuada de estos submódulos. El tercer submódulo es una implantación propia que llevó mucha dificultad debido a la complejidad de la conexión entre estos submódulos. En la Figura 4.4 se muestra un diagrama de estos submódulos.

CoovaChilli es el submódulo que se encarga de la mayor parte del trabajo. Redirige las conexiones entrantes no autenticadas al portal cautivo implantado en el módulo de *Usuarios* y mantiene un contacto constante con el *Servidor RADIUS*. Como parte de este submódulo se destaca también la *Interfaz JSON*, que viene incluida en la instalación de CoovaChilli y hace posible comunicar dicho *software* con nuestra *Aplicación Web*.

El *Servidor RADIUS* es el gestor de AAA requerido por CoovaChilli para funcionar, aunque es posible implantar otro tipo de servicios para llevar a cabo esta gestión, en este TFG se ha utilizado la aplicación *FreeRADIUS*. En este submódulo también se incluye todo aquel *software* que sea obligatorio para que el servidor funcione, como las instalaciones de bases de datos y su respectiva configuración.

Por supuesto, para que el sistema pueda funcionar, la base de usuarios del *servidor RADIUS* y la existente en los archivos JSON del submódulo *Controlador de Usuarios*, pertenecientes al módulo de *Usuarios*, ha de ser la misma.

Opcionalmente, puede instalarse un administrador gráfico del Servidor RADIUS para hacer más intuitiva su gestión. En este caso se incluye el submódulo *Interfaz Web de control RADIUS*, representada por una instalación del administrador Web daloRADIUS funcionando sobre el servidor Web *NGINX*.

4.3.3. Módulo Control de Red

El módulo de *Control de Red* es el de más bajo nivel de nuestro sistema, encargado de configurar y gestionar el comportamiento de los componentes *hardware* de la Raspberry Pi. Puede dividirse en tres submódulos: la *Configuración de las Interfaces de Red*, *Hostapd* y el propio dispositivo sobre el que opera todo el servicio: la *Raspberry Pi* con

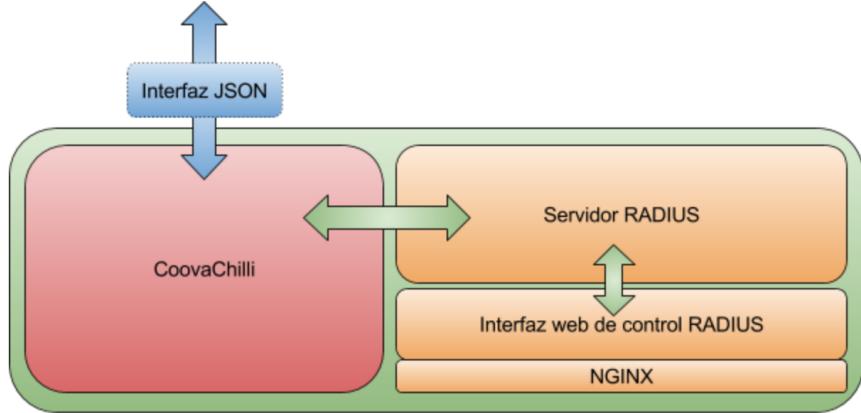


Figura 4.4: Diagrama de submódulos del módulo Control de Acceso.

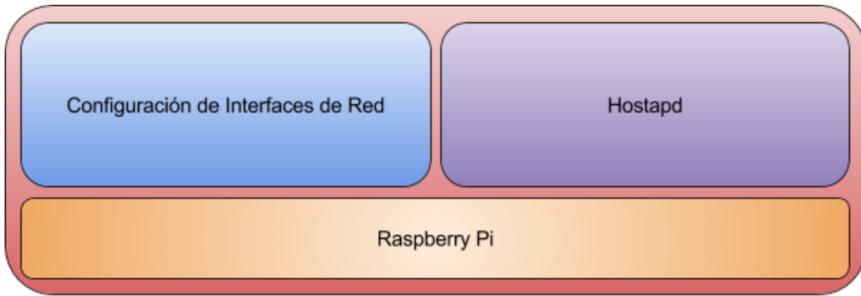


Figura 4.5: Diagrama de submódulos del módulo de Control de Red.

el sistema operativo Raspbian. Hemos preferido denominar a los módulos segundo y tercero como la aplicación original y el hardware por simplicidad para que se entienda que en esos módulos la tarea principal fue la de instalarlos y configurarlos adecuadamente (recurriendo a la técnica de ensayo y error en la mayoría de las veces). En la Figura 4.5 se muestra el esquema de submódulos de este módulo.

El submódulo *Configuración de Interfaces de Red* consiste en utilizar las herramientas instaladas en Raspbian (adquiriendo las que no estén disponibles por defecto) para implantar y configurar las redes necesarias para el funcionamiento del sistema. Las dos redes principales que intervienen, la cableada Ethernet *eth0* y la inalámbrica *wlan0*, varían su configuración según el entorno en el que se encuentren. La *wlan0* debe configurarse con IP estática y se le debe asignar la red correspondiente, dado que va a ser nuestro Punto de Acceso WiFi.

La configuración de *eth0* depende de la red que proporciona el acceso a Internet al sistema. Por ejemplo, la red cableada de la Universidad (donde hicimos pruebas) solo es accesible cuando la interfaz tiene una IP determinada en el punto de conexión, no pudiendo conectarse si no se asigna la IP adecuada. En una instalación de Internet casera puede configurarse la interfaz sin IP y dejar que el DHCP del router de la instalación asigne una dirección adecuada para establecer su red local.

Esta configuración incluye el paso necesario de habilitar el *flag* del *kernel* Linux del sistema operativo Raspbian para que permita el proceso de reenvío de paquetes IP, o IP forwarding, desde la interfaz cableada a la inalámbrica y viceversa. Sin este paso la instalación de CoovaChilli, que basa su funcionalidad de paso de paquetes entre interfaces mediante el uso de órdenes *iptables* de Linux [25], no podría funcionar.

Como también se adelantó en el anterior capítulo, *Hostapd* es un servicio que se ejecuta en segundo plano, configurando y permitiendo el uso de la interfaz inalámbrica wlan0 como punto de acceso WiFi (IEEE 802.11n), haciendo posible que los clientes se conecten a nuestro servicio.

El submódulo *Raspberry Pi*, con el sistema operativo Raspbian, proporciona las interfaces de red necesarias, los gestores de paquetes y compiladores para instalar todo el ecosistema *software* y las opciones de configuración de red que hacen posible la implementación total del servicio.

Para un mayor entendimiento global de todo el sistema instalado, configurado y desarrollado, en la Figura 4.6 se presenta un diagrama que engloba los tres módulos, incluyendo sus submódulos, y las interrelaciones existentes entre los mismos.

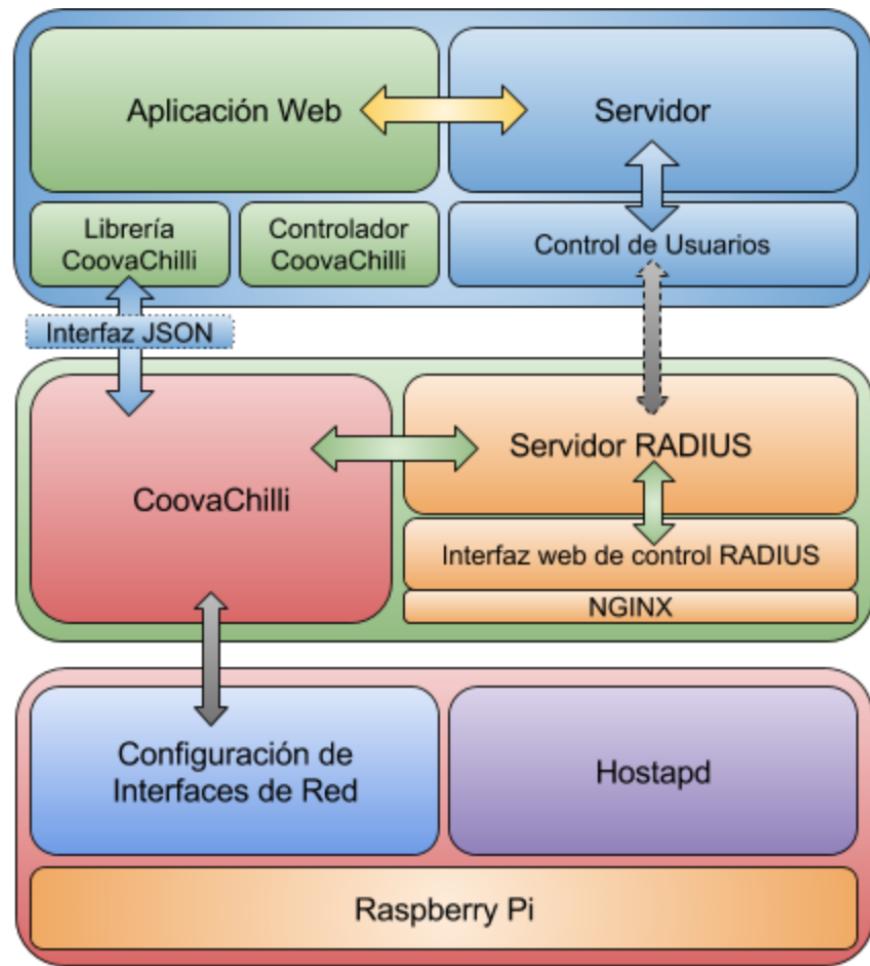


Figura 4.6: Esquema completo de los módulos funcionales del sistema.

5. Análisis orgánico de la implantación del *software*

Una vez revisado el análisis previo y funcional del sistema de sensado móvil colaborativo mediante una visión *top-down*, en este capítulo presentamos el análisis orgánico de la implantación mediante una visión *bottom-up*. Esto es, primero se configura el nivel inferior y el *Control de Red*, tras lo cual se va ascendiendo en el modelo hasta llegar a la implantación del portal cautivo, en el nivel de *Usuarios* y el nivel más alto del sistema. La descripción de la instalación del hardware se presenta en el apéndice A.

5.1. Módulos de Control de Red y de Acceso

Una vez instalado Raspbian y arrancada con éxito la Raspberry Pi 3 (explicado en el Apéndice A), es necesario seguir configurando el ecosistema e instalando el *software* necesario, idealmente mediante el gestor de paquetes y manejador de dependencias *apt-get* y trabajando en todo momento desde el terminal *bash*. Esto prepara el *software* de los módulos *Control de Red* y *Control de Acceso*. Se detalla el procedimiento llevado a cabo en el mismo orden en el que se implementó en la práctica para que sea reproducible de forma sencilla la instalación de un nuevo sistema.

5.1.1. Ajustes e instalaciones preliminares

Una de las primeras configuraciones consiste en modificar el archivo */etc/network/interfaces* para introducir los detalles de nuestra red inalámbrica. Este paso prepara el submódulo *Configuración de Interfaces Red* perteneciente al módulo de *Control de Red*. En este TFG se ha utilizado la siguiente configuración de red para la interfaz inalámbrica *wlan0*, que recordamos es la interfaz inalámbrica a la que se conectan eventualmente los usuarios del servicio.

```
auto wlan0
allow-hotplug wlan0
iface wlan0 inet static
    address 192.168.10.1
    netmask 255.255.255.0
    network 192.168.10.1
    post-up echo 1 > /proc/sys/net/ipv4/ip_forward
```

Código 5.1: Configuración de la interfaz de red *wlan0*

La última línea, *post-up echo 1 > /proc/sys/net/ipv4/ip_forward*, es la activación del *flag* del *kernel* que habilita el paso de datagramas IP tratados en la descripción del módulo de Control de Red. Si se prefiere utilizar esta orden de otra forma y así asegurarse de que el *flag* queda activado también puede editarse el archivo */etc/sysctl.conf*, quitando el carácter de comentario (#) en la línea *net.ipv4.ip_forward=1* y reiniciando el servicio de red mediante la siguiente orden: */etc/init.d/networking restart*.

Una vez configurada la interfaz inalámbrica pueden instalarse los paquetes necesarios para preparar los módulos. Se incluyen algunos paquetes previos necesarios para la instalación posterior de *software*, como herramientas para transferencia de datos (*libcurl*),

utilidades de red manejadas por CoovaChilli (*iptables*), compiladores (*gcc*) y gestores de dependencias para los mismos (*make*). Recordamos que esta instalación debe ser realizada por el superusuario, por lo que se debe entrar en *root* ejecutando primero la orden `sudo su` o por el contrario utilizar la orden previa `sudo` para cada orden a introducir. Además, pueden instalarse todos los paquetes necesarios con una sola orden escribiendo todos los nombres seguidos después de la orden `apt-get install` y forzando una respuesta afirmativa a todas las preguntas que nos realice el terminal al instalar los paquetes usando los modificadores `-y --force-yes`.

Los más relevantes para nuestro sistema son los paquetes *freeradius* y *mysql-server*, pero aquí se incluyen todos los instalados en la configuración de nuestro servicio:

```
sudo apt-get install -y --force-yes debconf-utils
```

Con este primer paquete puede establecerse una contraseña para el usuario *root* de la base de datos MySQL con las siguientes órdenes, a ejecutar después de que acabe el anterior, y sustituyendo la palabra CONTRASEÑA de las órdenes por la contraseña escogida. De esta forma, *mysql-server* no pide una contraseña de *root* en el momento de su instalación. Si no que se establece una contraseña diferente, la utilizada por defecto en la instalación de *mysql-server* para Raspbian es *raspbian..*

```
echo 'mysql-server mysql-server/root_password password CONTRASEÑA' |  
debconf-set-selections  
  
echo 'mysql-server mysql-server/root_password_again password  
CONTRASEÑA' | debconf-set-selections
```

Tras esto, ya pueden instalarse el resto de paquetes necesarios que habíamos comentado al principio.

```
sudo apt-get install -y --force-yes debhelper libssl-dev  
libcurl4-gnutls-dev mysql-server freeradius freeradius-mysql gcc make  
libnl libnl-dev pkg-config iptables
```

La herramienta *apt-get* instala también todos los paquetes de dependencias adicionales necesarias para el funcionamiento de los paquetes declarados.

Nótese que en este punto ya se ha empezado a instalar *software* correspondiente al módulo de *Control de Acceso*, como *FreeRADIUS* y el servidor de bases de datos *MySQL* con el que este va a funcionar. En el siguiente apartado se procede a su configuración.

5.1.2. Configuración del Servidor RADIUS

Durante el paso anterior se instaló el servidor de base de datos MySQL y el servidor RADIUS. A continuación se procede a crear la base de datos a usar por el programa FreeRADIUS utilizando para ello unos *scripts* proporcionados en su propia instalación.

Primero se crea la base de datos con la orden `mysql`, utilizando el usuario root y la contraseña especificada anteriormente.

```
echo 'create database radius' | mysql -u root -p CONTRASEÑA
```

Esto pone en marcha la interfaz de línea de órdenes de MySQL, con el usuario y contraseña detallados en las opciones, y le introduce el comando escrito entre comillas.

Tras esto, utilizamos una serie de *scripts* que la instalación de FreeRADIUS ubica en su árbol de directorios y los introducimos como órdenes de MySQL para la base de datos *radius* creada, de forma parecida a la utilizada en la última orden.

```
mysql -u root -p CONTRASEÑA radius < /etc/freeradius/sql/mysql/schema.sql
mysql -u root -p CONTRASEÑA radius < /etc/freeradius/sql/mysql/admin.sql
mysql -u root -p CONTRASEÑA radius < /etc/freeradius/sql/mysql/nas.sql
```

Código 5.2: Inserción de órdenes SQL desde ficheros de FreeRADIUS

Como opción antes de ejecutar estas órdenes, y para incrementar la seguridad, se puede modificar la contraseña por defecto de FreeRADIUS para operar en la base de datos (*radpass*) en el archivo `/etc/freeradius/sql/mysql/admin.sql`.

El siguiente paso es habilitar el soporte de SQL en el archivo de configuración de FreeRADIUS y como opción activar los contadores diarios. Para ello se debe editar el archivo ubicado en `/etc/freeradius/radiusd.conf` y eliminando el carácter de comentario (#) de las líneas `$INCLUDE sql.conf`, `$INCLUDE sql/mysql/counter.conf` y opcionalmente de `daily`.

Tras editar estos archivos, solo queda detener temporalmente la ejecución de FreeRADIUS, activar la autenticación SQL, comprobar que la configuración es correcta y volver a iniciar el servicio: `service freeradius stop`.

Para activar la autenticación SQL hay que editar el archivo `/etc/freeradius/sites-available/default` y quitar los comentarios de todas las líneas que contengan `sql`. Una vez hecho esto comprobamos la configuración y reanudamos el servicio.

```
freeradius -C  
service freeradius start
```

En este punto FreeRADIUS aún no ha terminado del todo su configuración, pero para ello es necesario que acaben las instalaciones de los demás componentes.

5.1.3. Instalación de CoovaChilli

Llegados a este punto podemos instalar y configurar el núcleo de funcionamiento de nuestro sistema, que es el servicio CoovaChilli. Como se comentó anteriormente, este *software* es un proyecto de código abierto basado en una antigua solución propietaria. Lamentablemente, su código no se encuentra disponible en forma de paquete precompilado que se pueda instalar mediante un orden sencillo con el gestor de paquetes *apt-get*, sino que se ha de descargar el código fuente y compilarlo manualmente, convirtiéndolo en un paquete .deb instalable de forma manual en sistemas basados en Debian. En este TFG se ha utilizado el código fuente de la aplicación alojado en GitHub, que permite dicha compilación, descargándolo mediante la herramienta de control de versiones *git* que ha de estar instalada en nuestro sistema operativo.

Como sucedió con el paso de FreeRADIUS, deben realizarse unas instalaciones previas de paquetes y dependencias auxiliares antes de comenzar la propia instalación de CoovaChilli y los demás elementos de *software*, entre los que se encuentra *git*.

```
apt-get install -y --force-yes git libjson-c-dev haserl gengetopt  
devscripts libtool bash-completion autoconf automake
```

Tras esto, hay que descargar el código fuente de CoovaChilli en un directorio y compilarlo en un paquete instalable en Debian, que puede lograrse con una utilidad del sistema operativo en unas pocas líneas. Este último proceso de compilación puede llevar tiempo y si no está todo previamente instalado y configurado puede fallar.

```
cd /usr/src  
git clone https://github.com/coova/coova-chilli.git
```

Tras acabar el proceso de descarga, comenzamos el proceso de compilación situándonos en la raíz del código fuente recién descargado con la línea anterior.

```
cd /usr/src/coova-chilli  
dpkg-buildpackage -us -uc
```

En este punto comienza la compilación del código fuente. Al finalizar, tendríamos un archivo de nombre similar a `coova-chilli_1.3.0_armhf.deb` en el directorio `/usr/src/` y podríamos proceder a instalarlo situándonos en dicho directorio y utilizando el gestor de paquetes `dpkg`.

```
cd /usr/src  
dpkg -i coova-chilli_1.3.0_armhf.deb
```

Configuración Inicial de CoovaChilli

Llegados a este punto el programa CoovaChilli está instalado y podemos proceder a su configuración. Esta configuración en su mayoría se basa, como se adelantó en la descripción del *software* del anterior capítulo, en modificar el archivo `/etc/chilli/defaults` o hacer una copia de éste bajo el nombre `config` teniendo allí las modificaciones con respecto a `defaults`. Sin embargo, hay dos pasos previos a la configuración de este archivo.

En primer lugar debemos modificar el *script* que CoovaChilli ejecuta al ponerse en marcha la interfaz TUN (ubicado en `/etc/chilli/up.sh`) introduciendo una regla de la herramienta *iptables* para permitir el reenvío de los paquetes IP desde la interfaz de red cableada `eth0`, que se almacena en una variable interna de la configuración de CoovaChilli llamada `HS_WANIF`.

```
iptables -I POSTROUTING -t nat -o $HS_WANIF -j MASQUERADE
```

También debemos activar el *flag* que permite activar CoovaChilli en el archivo `/etc/default/chilli/` cambiando el valor de la variable `START_CHILLI` de 0 a 1.

Tras esto, se debe editar el archivo de configuración de CoovaChilli introduciendo los valores adecuados a nuestro sistema en los atributos relevantes en este momento en el archivo `/etc/chilli/defaults` o bien en su copia renombrada como `/etc/chilli/config`, que al iniciar el sistema reemplaza los valores dados en `defaults` donde difieran. Dado que en este momento aún no se encuentra instalada la herramienta `hostapd` (ver más adelante) aún no tenemos habilitado un SSID para nuestro sistema, por lo que en un paso posterior tras la instalación de `hostapd` debemos volver a este archivo de configuración e introducir el nombre del SSID que hayamos establecido. En el caso de este TFG, el SSID utilizado es *RasPiDav*.

```
HS_WANIF=eth0
HS_LANIF=wlan0
HS_NETWORK=192.168.10.0
HS_UAMLISTEN=192.168.10.1
HS_UAMALLOW=192.168.10.0
HS_SSID=RasPiDav
HS_COAPORT=3799
HS_UAMSECRET=
```

Código 5.3: Configuración de CoovaChilli

Debido al funcionamiento de nuestro portal cautivo, que opera con la interfaz JSON de CoovaChilli, es necesario dejar en blanco el campo `HS_UAMSECRET`, de lo contrario la aplicación Web no podría iniciar sesión en el servicio con las credenciales obtenidas.

Instalación de Haserl

Dado que algunas de las funciones de CoovaChilli necesitan del programa Haserl (como su pequeño portal cautivo por defecto implementado con *Web Scripts*) hace falta instalarlo y añadir su ruta de instalación a la configuración de CoovaChilli. Para ello hacemos un procedimiento similar al usado en la instalación de CoovaChilli, salvo que en este caso compilamos el programa con la utilidad *make*.

```
cd /usr/src
wget
http://downloads.sourceforge.net/project/haserl/haserl-devel/haserl-0.9.35.tar.gz
tar zxvf haserl-0.9.35.tar.gz
cd /usr/src/haserl-0.9.35
./configure
make
make install
```

Código 5.4: Instalando Haserl

Tras esto actualizamos la ruta de *Haserl* para CoovaChilli en el archivo `/etc/chilli/wwwsh` buscando la línea `haserl=` y modificándola a `haserl=/usr/local/bin/haserl`. Tras esto, solo queda poner en marcha CoovaChilli con la siguiente orden.

```
service chilli start
```

5.1.4. Instalación y configuración de Hostapd

En este momento se prepara la herramienta *hostapd* para poner en marcha la interfaz WiFi como Punto de Acceso IEEE 802.11n, comenzando por su instalación y continuando con su configuración.

```
apt-get install -y --force-yes hostapd
```

Cuando termine la instalación se edita el archivo de configuración */etc/default/hostapd* añadiendo la ruta del archivo de configuración que contiene los atributos deseados para nuestro punto de acceso, que está ubicado en */etc/hostapd/hostapd.conf*. Para ello buscamos la línea que contenga **DAEMON_CONF=**, nos aseguraremos de que está sin comentar y le especificaremos la ruta entre comillas, de manera que quede de la siguiente forma: **DAEMON_CONF="/etc/hostapd/hostapd.conf"**.

Ahora configuramos el mismo archivo cuya ruta hemos indicado en el ajuste anterior, indicando todos los atributos para nuestro punto de acceso incluyendo el driver a utilizar (*nl80211*). Recordamos que la interfaz establecida ha de ser la misma que indicamos en el archivo de configuración **defaults** o **config** de CoovaChilli como **HS_LANIF**, en nuestro caso *wlan0*, que tiene la red estática ya configurada en el archivo *interfaces* al principio de la implantación. Además, el SSID también ha de ser el mismo que indicamos a CoovaChilli con la variable **HS_SSID**.

```
interface=wlan0
driver=nl80211
ssid=RasPiDav
hw_mode=g
channel=6
auth_algs=1
beacon_int=100
dtim_period=2
max_num_sta=255
rts_threshold=2347
fragm_threshold=2346
ieee80211n=1
wmm_enabled=1
ht_capab=[HT40] [SHORT-GI-20] [DSSS_CCK-40]
```

Código 5.5: Configuración de Hostapd

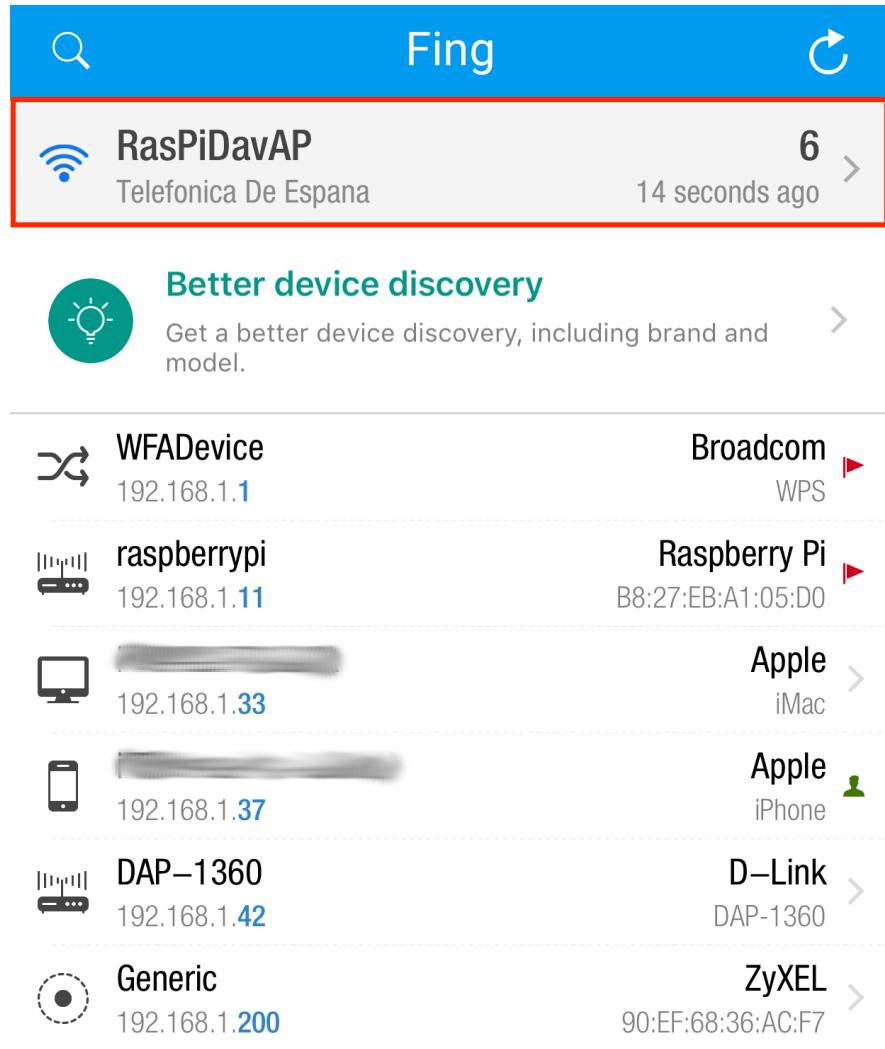


Figura 5.1: Ventana que muestra todas las redes WiFi disponibles y entre ellas la que hemos configurado (en este caso RasPiDavAP, recuadrada en rojo).

En este punto reiniciamos hostapd para que utilice los ajustes deseados. Al ejecutar este orden ya sería visible la red WiFi de la Raspberry Pi 3 (Figura 5.1) desde otros dispositivos con el SSID indicado:

```
service hostapd start
```

5.1.5. La interfaz Web de daloRADIUS

Para facilitar la administración del servicio, incluyendo la gestión del servidor RADIUS, puede resultar útil implementar servicios que ofrecen interfaces gráficas fáciles de comprender y manejar. Para hacer uso de uno de los servicios de este tipo más conocidos para FreeRADIUS, conocido como daloRADIUS, instalamos el servidor Web NGINX y alojamos allí los archivos necesarios para gestionar el servidor RADIUS desde una

aplicación tan sencilla y ubicua como el navegador Web.

En primer lugar, instalamos el servidor Web y las herramientas necesarias para utilizar daloRADIUS, que son sobre todo librerías de soporte PHP.

```
apt-get install -y --force-yes php5-mysql php-pear php5-gd php-db  
php5-fpm libgd2-xpm-dev libpcrecpp0 libxpm4 nginx php5-xcache
```

Cuando acabe esta instalación el programa *NGINX* queda instalado y podríamos alojar el contenido Web que deseamos servir en el directorio `/usr/share/nginx/html/`. En nuestro caso, descargamos los contenidos de daloRADIUS, los descomprimimos allí y renombramos el directorio generado para acceder a él al teclear la futura URL desde el navegador.

```
cd /usr/src  
wget https://sourceforge.net/projects/daloradius/files/latest/download  
tar zxvf download -C /usr/share/nginx/html/  
mv /usr/share/nginx/html/daloradius-0.9-9 /usr/share/nginx/html/daloradius
```

Código 5.6: Instalando daloRADIUS

Aunque no los usamos en este TFG, conviene remarcar que daloRADIUS también incluye portales cautivos de ejemplo para ser usados con *CoovaChilli* en el árbol de directorios que hemos descomprimido y renombrado, implementados usando PHP y JavaScript. Tras nuestros cambios, estos portales cautivos quedan ubicados en la ruta `berb+/usr/share/nginx/html/daloradius/contrib/chilli/+`.

La descarga de daloRADIUS contiene archivos específicos que lo habilitan para ser usado con *FreeRADIUS* y *MySQL*. Para utilizarlos debemos copiar dichos archivos existentes en el árbol de directorios de *daloRADIUS* y colocarlos en los directorios de *FreeRADIUS* indicados en las siguientes órdenes, reemplazando las originales. Por seguridad conviene detener el servicio *FreeRADIUS* antes de modificar archivos y hacer copia de seguridad de los archivos que quedan reemplazados, renombrándolos o cambiando las extensiones.

```

service freeradius stop
cp /usr/share/nginx/html/daloradius/contrib/configs/freeradius-2.1.8/cfg1/raddb/s_
ql/mysql/counter.conf
/etc/freeradius/sql/mysql/counter.conf
cp /usr/share/nginx/html/daloradius/contrib/configs/freeradius-2.1.8/cfg1/raddb/s_
ites-available/default
/etc/freeradius/sites-available/default
cp /usr/share/nginx/html/daloradius/contrib/configs/freeradius-2.1.8/cfg1/raddb/m_
odules/sql.conf
/etc/freeradius/sql.conf

```

Código 5.7: Configuración de daloRADIUS para trabajar con FreeRADIUS.

Dado que el último archivo que hemos ubicado (`/etc/freeradius/sql.conf`) procede de una instalación nueva de daloRADIUS contiene la contraseña utilizada por defecto en FreeRADIUS, por lo que debemos editarla para que tenga la contraseña que indicamos en el paso correspondiente del apartado 5.1.2. Recordamos que, si no se cambió, la contraseña utilizada por defecto es *radpass*.

Tras estos pasos podemos iniciar el servidor RADIUS de nuevo y utilizar el *script* específico de *daloRADIUS* incluido en la instalación para añadir la configuración de *daloRADIUS* en MySQL, añadiendo también los permisos de usuario correspondientes para poder manejar el servidor. Recordamos que la palabra CONTRASEÑA en mayúscula en este caso hace referencia a la contraseña utilizada para el usuario *root* del servidor MySQL.

```

service freeradius start
mysql -u root -p CONTRASEÑA radius <
/usr/share/nginx/html/daloradius/contrib/db/fr2-mysql-daloradius-and-freeradius.sql
echo "GRANT ALL ON radius.* to 'radius'@'localhost';" > /tmp/grant.sql
echo "GRANT ALL ON radius.* to 'radius'@'127.0.0.1';" >> /tmp/grant.sql
mysql -u root -p CONTRASEÑA < /tmp/grant.sql

```

Código 5.8: Creando bases de datos para daloRADIUS.

Además, actualizamos el archivo de daloRADIUS que contiene el usuario y contraseña de la base de datos para que tenga el usuario y contraseña del servidor RADIUS. Este archivo está en `/usr/share/nginx/html/daloradius/library/daloradius.conf.php` y ha de tener los siguientes valores en los campos correspondientes. Recordamos que

esta contraseña es la del servidor RADIUS, indicada por CONTRASEÑA-RADIUS, (por defecto *radpass* si no la cambiamos).

```
$configValues['CONFIG_DB_USER'] = 'radius';
$configValues['CONFIG_DB_PASS'] = 'CONTRASEÑA-RADIUS';
```

Configurando NGINX para servir daloRADIUS

Tras preparar *daloRADIUS* para su funcionamiento hemos de configurar el servidor NGINX para que nos sirva la página principal correcta al introducir la ruta adecuada en el navegador. Para esta configuración editamos el archivo */etc/nginx/sites-available/default* añadiendo la siguiente directriz de servidor. Para acceder a la interfaz Web en el futuro utilizamos el puerto 80.

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    root /usr/share/nginx/html;
    index index.html index.htm index.php;
    server_name _;
    location / {
        try_files $uri $uri/ =404;
    }
    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/var/run/php5-fpm.sock;
    }
}
```

Código 5.9: Directriz de servidor de NGINX para daloRADIUS.

Llegados a este punto solo queda comprobar que la configuración de NGINX es la correcta y reiniciar tanto el servidor Web como *hostapd*.

```
nginx -t
service nginx restart
service hostapd restart
```

Tras esto podemos acceder a *daloRADIUS* desde el navegador para realizar las gestiones deseadas utilizando la IP proporcionada a la interfaz cableada. Por ejemplo, si la IP de

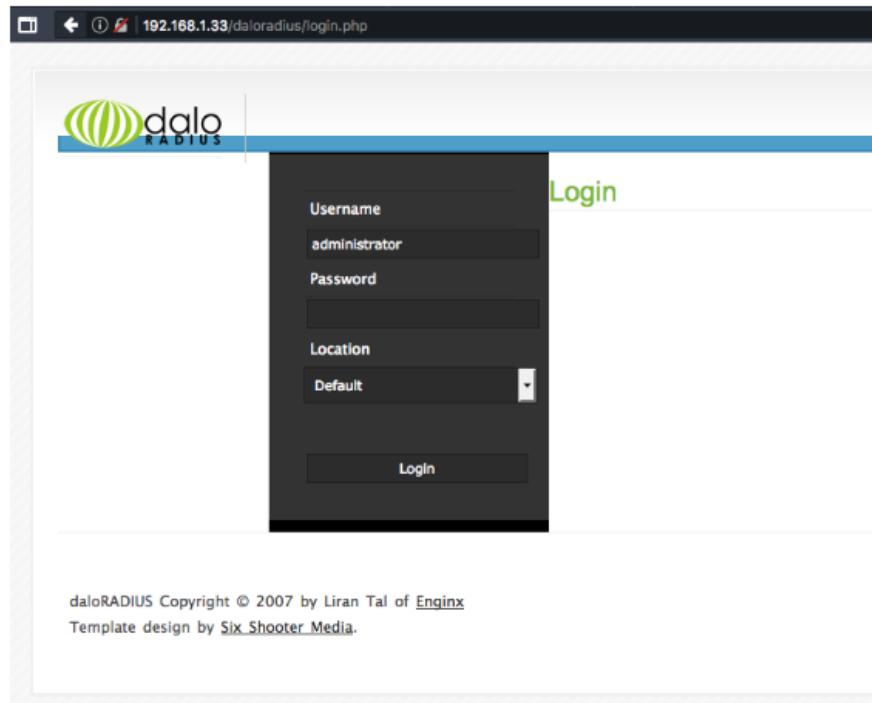


Figura 5.2: Vista de la página de entrada a daloRADIUS.

dicha interfaz es 192.168.1.30 se puede acceder a *daloRADIUS* accediendo a la URL <http://192.168.1.30/daloradius/>.

Utilizando daloRADIUS para añadir a los usuarios del servicio

Una vez terminada nuestra instalación solo queda añadir usuarios al servidor RADIUS para que puedan conectarse al servicio, usando para ello el portal cautivo cuya implantación veremos en los siguientes apartados. Gracias a la instalación de *daloRADIUS* podemos añadir estos usuarios de forma gráfica a través del navegador Web en lugar de utilizar órdenes SQL desde un terminal. Cuando accedamos a *daloRADIUS* accediendo a la URL descrita anteriormente se carga una vista similar a la de la Figura 5.2.

El usuario *administrator* está ya escrito en el campo *Username*. La contraseña para dicho usuario, a escribir en el campo inferior, es *radius*. Al hacer clic en el botón *Login* entramos en la pantalla principal de daloRADIUS (Figura 5.3).

Aquí hacemos clic en la pestaña *Management* del menú superior y tras ello en la opción *New User* del menú lateral izquierdo, lo que nos conduce al formulario de añadir nuevo usuario que nos permite escoger el tipo de autenticación a utilizar. En nuestro caso es la primera opción, *Username Authentication* (Figura 5.4).

Pulsando el botón *Apply* tras haber rellenado los campos se completa el proceso

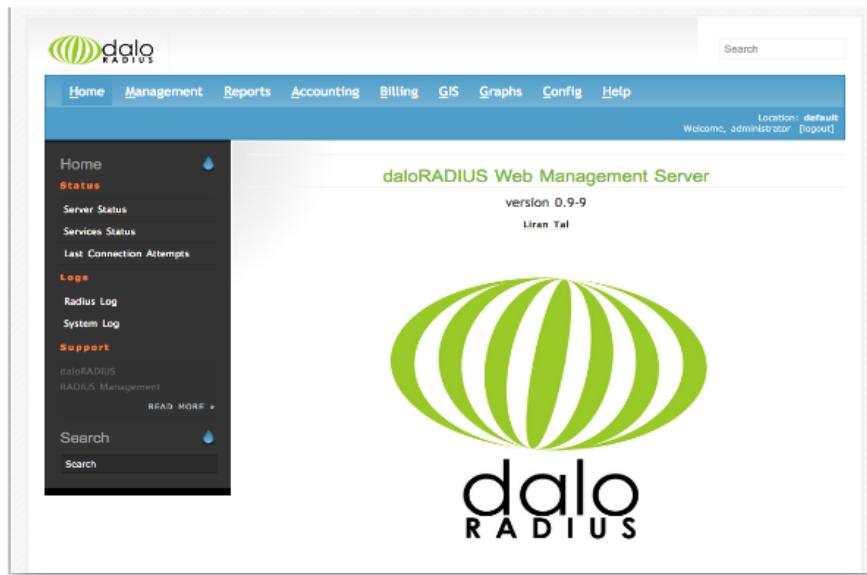


Figura 5.3: Vista principal de daloRADIUS.

Figura 5.4: Vista de la página de manejo de usuarios en daloRADIUS.

de creación de usuario. Haciendo clic en el elemento *List Users* del menú lateral izquierdo podemos ver una lista de los usuarios registrados en el servicio y varios menús desplegables con detalles sobre los mismos.

Además de registrar usuarios es posible asignarlos en grupos y modificar algunos atributos, por ejemplo el tiempo de conexión, máximo volumen de datos a transferir o incluso el intervalo de tiempo en el que es posible conectarse. Dichos atributos pertenecen a *FreeRADIUS*, aunque *daloRADIUS* permite usar atributos y variables de otros fabricantes, y pueden cambiarse en la pestaña *Attributes* al crear un nuevo

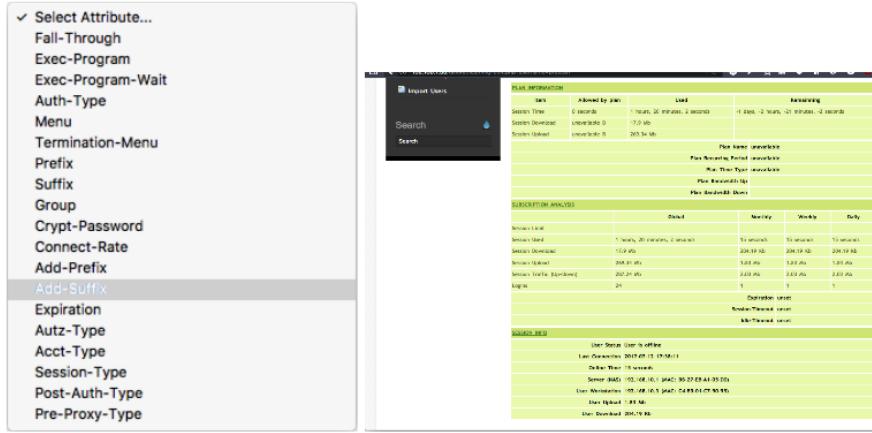


Figura 5.5: Vista de la página de Attributes de daloRADIUS.

usuario o seleccionar uno ya existente de la lista (Figura 5.5).

Para añadir usuarios en bloque, daloRADIUS ofrece un modo de añadir conjuntos completos de usuarios, facilitando en gran parte el proceso de población de la base de datos. Para ello, en el submenú *Management* puede seleccionarse la opción *Batch Users*, tras lo cual la opción de añadir usuarios en bloque, consultarlos o eliminarlos queda disponible en la barra lateral izquierda. Es posible gestionar el modo en que se generan los nombres de usuario, la longitud de los caracteres utilizados, el número de instancias o los grupos a los que pertenecerán. Este conjunto de usuarios, recién creado, puede exportarse a un archivo Excel y *Comma Separated Values* (CSV), que pueden ser pasados a un *parser* que pase los pares usuario-contraseña a otro formato, como se ha hecho en este TFG, o incluso imprimirse en forma de tarjetas para entregar a posibles usuarios (Figura 5.6).

Para implantar dos tipos de usuarios, el caso de este TFG, donde hay un conjunto de usuarios que solo tienen un tiempo de conexión de 30 m, puede crearse un segundo conjunto de usuarios que cuenten con dicha restricción haciendo uso del atributo *RADIUS Session-Timeout* definido en la RFC 2865. Tras seleccionar el atributo se le asigna el valor buscado medido en segundos (Figura 5.7).

5.2. Módulo de Usuarios

A diferencia de los dos módulos anteriores, cuya implantación se basa únicamente en la instalación y configuración de *software*, el módulo de usuarios cuenta con un mayor esfuerzo de programación, encontrándose el módulo en su totalidad contenido en único directorio del sistema constituido por archivos JavaScript, JSON, HTML y CSS. A continuación analizamos la implantación general y de cada uno de los submódulos que

Batch Users Management

Account Info		User Info	Billing Info	Attributes
Batch Id/Name	<input type="text"/>			
Batch Description	<input type="text"/>			
HotSpot	<input type="button" value="Select Hotspot"/>			
Username Prefix	<input type="text"/>			
<input checked="" type="radio"/> Create Random Users	Length of username string	<input type="button" value="8"/>		
<input type="radio"/> Create Incrementing Users	Starting Index	<input type="button" value="1"/>		
	Length of password string	<input type="button" value="8"/>		
	Number of Instances to create	<input type="button" value="1"/>		
	Group	<input type="button" value="Select Groups"/>		
	Group Priority	<input type="button" value="0"/>		
	Plan Name	<input type="button" value="Select Plan"/>		
<hr/>				
<input type="button" value="Apply"/>				

Figura 5.6: Vista de la página de entrada a daloRADIUS por lotes.

constituyen el módulo de usuarios, sobre todo aquellos que han sido programados desde cero, entrando en un mayor detalle sobre sus funciones individuales.

5.2.1. Configuración previa y estructura del módulo

Dado que el módulo de usuarios basa su funcionamiento en Node.js y npm debemos instalar estos dos componentes, pues habitualmente la versión de los mismos incluida por defecto en los sistemas operativos Linux es muy antigua. Los pasos necesarios para descargar e instalar ambas utilidades varían según el sistema operativo. En nuestro caso, Node.js no está incluido por defecto en los repositorios por lo que en principio no puede instalarse con un sencillo comando de *apt-get*. Podemos añadir un repositorio de su última versión para su fácil instalación y actualización con un único paso previo desde el terminal, tras el cual podremos instalarlo siguiendo el procedimiento habitual.

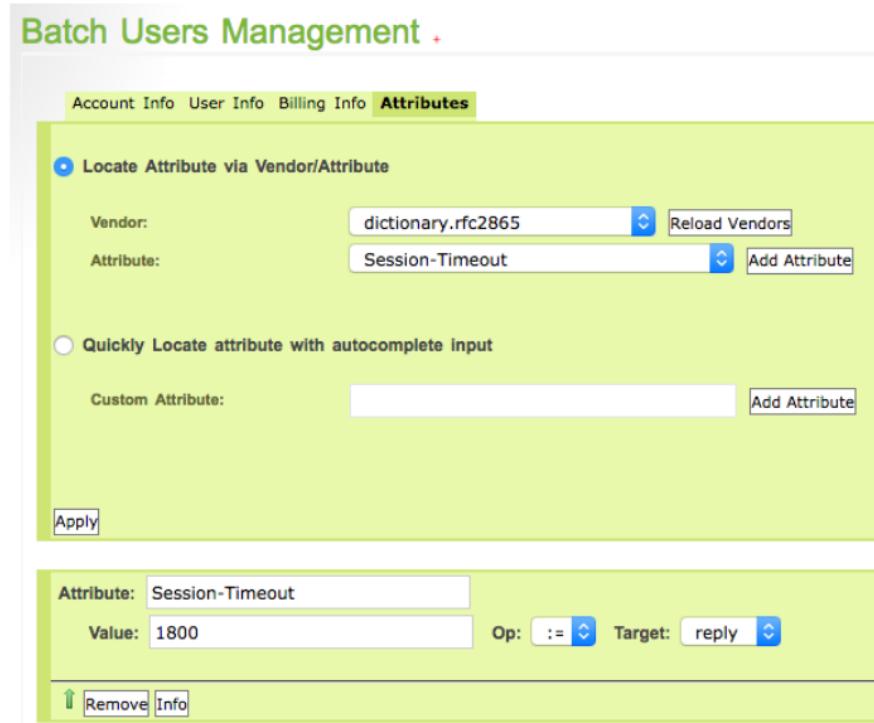


Figura 5.7: Vista de la página de RADUIS Session-timeout de daloRADIUS.

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Este procedimiento instala tanto Node.js como el gestor de paquetes npm, permitiendo su ejecución desde terminal con las órdenes *node* y *npm* respectivamente.

Se ha implementado el módulo de usuarios completo en un único directorio. En la Figura 5.8 se adjunta una imagen con la estructura del mismo sacada directamente del repositorio alojado en *GitHub*.

Cada archivo o directorio está directamente relacionado con uno de los submódulos, quedando distribuidos de la siguiente forma.

- *Control de Usuarios*: directorio *users* y todos sus archivos.
- *Servidor*: archivos *nodeserver.js*, *captiveportalserver.config.js*, *package.json* y *package-lock.json* y directorio *uploads*.
- *Aplicación Web*: archivos *index.html* y directorios *js*, *css*, *ssl* y todos sus archivos.

El funcionamiento de dicho módulo y las interrelaciones existentes entre sus submódulos se adelantaron en secciones anteriores de este capítulo. El diagrama de secuencia de la Figura 5.9 muestra la interacción de la aplicación Web (*Web App*), el servidor (*Server*) y el las acciones que puede hacer el usuario (*User Ctrl*).

<code>css</code>	Changed HTML imports for Bootstrap and jQuery
<code>js</code>	Modified WindowEventHandler for beforeunload
<code>ssl</code>	SSL second try
<code>uploads</code>	Added
<code>users</code>	Cleaning
<code>.gitignore</code>	Changed user interface and added alert handling
<code>CaptivePortal.iml</code>	Update CaptivePortal.iml
<code>README.md</code>	Edited Readme
<code>captiveportalserver.config.js</code>	Code formatting.
<code>favicon.ico</code>	Back to HTTP
<code>index.html</code>	Added a new connection mode for users who want access to their microp...
<code>nodeserver.js</code>	Cleaning
<code>package-lock.json</code>	Updated npm packages
<code>package.json</code>	Updated npm packages

Figura 5.8: Imagen de la estructura del directorio en el que se incluye el código programado para el módulo Usuario.

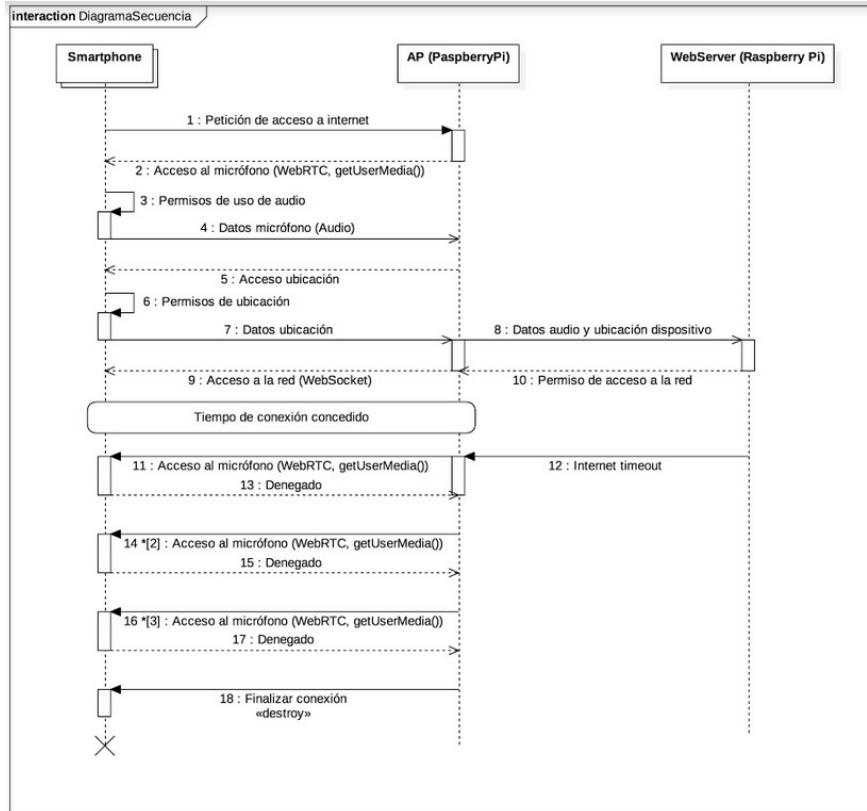


Figura 5.9: Diagrama de secuencia de los componentes del módulo de Usuario.

Donde las auto-interacciones 5, 6 y 7 son pedidas al usuario y las 13 y 16 son llevadas a cabo a través de la interfaz JSON de *CoovaChilli*. Sobre este último aspecto se profundiza más adelante.

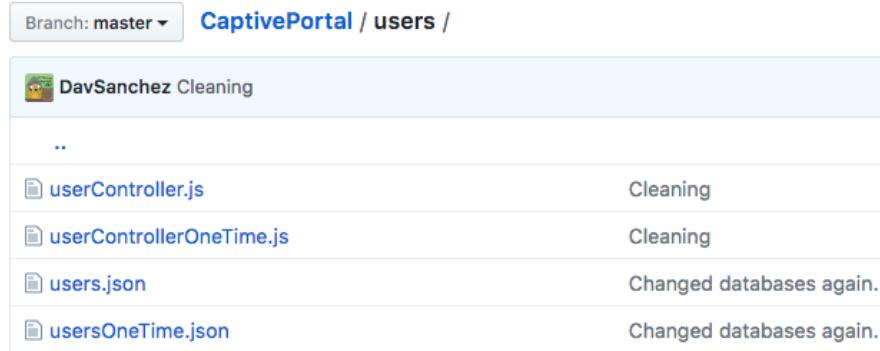


Figura 5.10: Vista del directorio con los archivos del submódulo Control de Usuarios.

5.2.2. Submódulo de Control Usuarios

Como ya adelantamos, en cuanto a estructura de directorios, el submódulo de control de usuarios está totalmente contenido en el directorio *users* de nuestro módulo de Usuario. Consta de cuatro archivos: dos bases de datos de usuarios JSON (*users.json* y *usersOneTime.json*) y los archivos JavaScript que controlan dichas bases de datos (*userController.js* y *userControllerOneTime.js*). Los archivos con el sufijo *-OneTime* en el nombre indican que son la base de datos y controlador correspondientes a la opción de conectarse durante 30 minutos a cambio de un único archivo de audio (Figura 5.10).

Si la base de usuarios se vuelve lo suficientemente grande y soporta una carga mucho mayor que la existente en las pruebas realizadas en este TFG este módulo podría reemplazarse por una herramienta de base de datos más potente a utilizar (SQL, MongoDB) o incluso dejar de ser un submódulo separado del propio servidor RADIUS del módulo de control de acceso, permitiendo al servidor trabajar directamente con la base de datos de FreeRADIUS.

Dado que cuatro los archivos constituyen dos pares muy similares se analizan en el mismo apartado.

Los archivos *users.json* y *usersOneTime.json*

Estos archivos alojan una base de datos muy sencilla en forma de un array de usuarios con cuatro atributos cada uno, estructurada de la manera mostrada en el bloque de código 5.10.

```
{
  "users": [
    {
      "id": 0,
      "username": "prueba1",
      "password": "prueba1",
      "isActive": false
    },
    {
      "id": 1,
      "username": "prueba2",
      "password": "prueba2",
      "isActive": false
    },
    {
      "id": 2,
      "username": "prueba3",
      "password": "prueba3",
      "isActive": false
    }
  ]
}
```

Código 5.10: Atributos de los usuarios en las bases de datos.

El atributo *id* se utiliza a efectos de programación, para llevar un control de los usuarios sin utilizar cadenas de caracteres; el *username* y *password* son obviamente las credenciales de cada usuario y el atributo booleano *isActive* actúa como un *flag* que marca los usuarios activos en ese momento en el sistema.

Los valores *username* y *password* deben corresponder con usuarios y contraseñas ya existentes en el servidor RADIUS, introducidas a través de daloRADIUS o ordens SQL en la base de datos tal y como se vio en apartados anteriores.

userController.js y *userControllerOneTime.js*

Estos archivos controlan la base de datos detallada en el apartado anterior, consultando y actualizando sus valores a petición del servidor. Sus funciones están definidas mediante los siguientes métodos y requerimientos.

```
var fs = require('fs');
var userObj = require('./users.json');
```

Código 5.11: Importación de módulos de UserController.js

El objeto *fs* es la importación del módulo *fs* de Node.js, que permite la lectura y escritura síncrona o asíncrona de archivos. Lo utilizamos para actualizar los campos de la base de datos según sea necesario. El objeto *userObj*, como puede observarse, es la importación del archivo de la propia base de datos JSON.

Aparte, el controlador de usuarios tiene tres funciones externas, a usar por el servidor, y cuatro funciones internas que usa el propio controlador. Comenzamos presentando las funciones internas y luego, de cara al análisis del servidor, pasamos a las externas.

```
function setUserActive(userId) {
    console.log("Estableciendo usuario " + userId + " como ocupado.");
    userObj.users[userId].isActive = true;
    writeUsersFile(userObj);
}

function setUserInactive(userId) {
    console.log("Estableciendo usuario " + userId + " como libre.");
    userObj.users[userId].isActive = false;
    writeUsersFile(userObj);
}
```

Código 5.12: Funciones internas de UserController.js.

Estas dos funciones se explican de forma sencilla con ayuda del *log* que emiten en consola al ser llamadas. Su función es modificar el objeto *userObj*, que recordemos contiene la base de datos, marcando el usuario pasado como parámetro como activo, y por tanto ocupado, o inactivo (libre). Tras modificar estos valores se escriben en el archivo original llamando a la función *writeUsersFile()*.

```

function writeUsersFile(userJSON){
    console.log("Guardando estado de usuarios en archivo JSON.");
    fs.writeFileSync("./users/users.json", JSON.stringify(userJSON, null, 2));
}

```

Código 5.13: Escritura del objeto *userObj* en fichero JSON.

Esta es la función que escribe en el archivo JSON el estado actual del objeto *userObj*, a efectos reemplazando el archivo antiguo por uno nuevo. El último parámetro pasado a la función *JSON.stringify()* sirve para dar formato de saltos de línea e indentación, con el objetivo de mejorar la lectura del archivo para humanos:

```

function prepareToConnect(userId) {
    console.log("Almacenando credenciales del usuario " + userId + " para el
    cliente.");
    return [userObj.users[userId].id, userObj.users[userId].username,
    userObj.users[userId].password, false, 0];
}

```

Código 5.14: Preparación de credenciales.

El objetivo de esta última función interna es retornar un vector de cinco elementos para ser usado en las funciones externas llamadas por el servidor. Los elementos son la *id*, *username* y *password* del usuario que tenga el *id* pasado por parámetros junto a dos *flags* que indican el tipo de usuario y el estado de su conexión. Este vector se almacena en la variable *creds* del servidor, que se detalla en el apartado 5.2.3.

A continuación se detallan las funciones externas, que recordamos sirven de interfaz entre el *Control de Usuarios* y el *Servidor*, este las llama para obtener credenciales, pedir el estado de la base de datos y actualizar valores de la misma.

```

exports.userInactive = function(id) {
    setUserInactive(id);
};

exports.checkInactiveUser = function () {
    console.log('Buscando usuarios inactivos...');
    var counter = 0;
    for (var i = 0; i<userObj.users.length; i++){
        if (!userObj.users[i].isActive){
            counter++;
        }
    }
    return counter;
};

exports.getInactiveUser = function() {
    userObj = JSON.parse(fs.readFileSync('./users/users.json', 'utf8'));
    for (var i =0; i<userObj.users.length; i++){
        if (!userObj.users[i].isActive){
            setUserActive(i);
            return prepareToConnect(i);
        }
    }
    console.log("Parece que no hay usuarios libres...");
    return ["", "", "", "", ""];
};

```

Código 5.15: Funciones externas de userController.js

Nótese el cambio en la implantación de las funciones. Mientras que las funciones internas pueden programarse utilizando primero la palabra reservada *function*, el operador de asignación y tras este el nombre de la función y su comportamiento, Node.js requiere de una sintaxis diferente para externalizar funciones. Ha de utilizarse el formato `exports.nombreDeLaFuncion = function(...) {...}` para que otros archivos JavaScript de Node.js puedan utilizarlas, importando el archivo donde estas se encuentran de la misma forma que si fuese un módulo *npm*.

La primera función, *userInactive*, simplemente llama a la función interna *setUserInactive*, por lo que es el servidor quien decide cuándo un usuario se marca como inactivo por medio del controlador de usuarios. Las dos funciones siguientes tienen un comportamiento muy similar. Mientras que la primera de ellas solo se encarga de comprobar si la base de datos tiene al menos un usuario libre, recorriendola e incrementando coincidencias de usuarios libres por medio de un contador, la segunda de ellas tiene el cometido

de retornar el primer usuario libre que encuentre en su recorrido de la base de datos, retornando un vector de cinco elementos nulos si no hay ningún usuario activo en la base de datos.

5.2.3. Submódulo Servidor.

El servidor interactúa tanto con la aplicación Web, por medio de peticiones AJAX, como con el controlador de usuarios por medio de la importación de módulos y uso de las funciones externas, como ya se comentó en la sección dedicada al mismo. Es aquí donde la mayoría de paquetes *npm* —sobre todo Express— entran en funcionamiento, necesitando previamente realizar las importaciones necesarias. Obsérvese que se importan todos los módulos de los que se ha hablado previamente en el capítulo anterior, por lo que sus funciones o características no se repiten de nuevo en detalle:

```
var express = require('express');
var app = express();
var http = require('http');
var https = require('https');
var path = require('path');
var formidable = require('formidable');
var fs = require('fs');
var userController = require('./users/userController');
var userControllerOneTime = require('./users/userControllerOneTime');
var bodyParser = require('body-parser');
var creds = {
  id: -1,
  username: "prueba",
  password: "pruebaPass",
  oneTimePass: false,
  connected: 0
};
```

Código 5.16: Inicialización de variables del servidor node.

Aparte de las importaciones de módulos y paquetes *npm*, destacan las tres últimas declaraciones de variables. La variable *userController* que contiene al archivo JavaScript correspondiente al submódulo de controlador de usuarios, la variable *app* que no es más que el objeto Express que implementa todas las funcionalidades de servidor, y la variable *creds* que es el objeto donde se almacenan temporalmente las credenciales a enviar a la instancia de la aplicación Web, estando la *id* en el valor -1 para indicar cuando no hay ningún usuario almacenado.

Como ya vimos en el capítulo anterior al detallar las tecnologías a utilizar, el funcionamiento de servidor mediante Express se logra mediante unas pocas líneas de código. Sin embargo, previamente han de configurarse los diferentes *middlewares* a utilizar e implantarse los diferentes manejadores de rutas a las que se envían las peticiones HTTP. Empezando por orden, primero se establecen los *middlewares* de nivel de aplicación mediante las funciones `app.use(...)`.

```
app.use(express.static(path.join(__dirname, '')));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
  extended: true
}));
```

Código 5.17: Configurando las opciones del servidor Express.

Estos usos corresponden al servicio de archivos HTML estáticos y a los analizadores sintácticos de los datos recibidos, a cuyos contenidos podremos acceder si están codificados con los tipos declarados aquí, JSON y URL-encoded.

A continuación se procede a declarar los manejadores de las posibles rutas a las que llegan peticiones HTTP y el procedimiento a llevar a cabo cuando esto ocurra. Estas rutas se declaran mediante la sintaxis `app.method('/ruta', function(req, res) {...})`, donde *method* es el método HTTP con el que llega la petición (GET, POST, PUT...), *req* el objeto petición y *res* el objeto respuesta.

```
app.get('/', function(req, res){
  res.sendFile(path.join(__dirname, 'index.html'));
});
```

Código 5.18: Ruta raíz del portal cautivo.

La principal es, lógicamente, la petición HTTP correspondiente a la raíz del servicio Web. Esto corresponde a servir el archivo HTML `index.html` del que se habla en la sección del submódulo aplicación Web.

La Aplicación Web servida tiene que tener un medio para saber si la reserva de usuarios del servicio está completa, no pudiendo prestar el servicio a ningún usuario más. Para ello se implementa un manejador de ruta en `/serverstatus` que llama a las funciones correspondientes del controlador de usuarios, tal y como se ve en el siguiente código.

El cliente contacta con el servidor para conocer esta información por medio de una petición GET.

```
app.get('/serverstatus', function (req, res) {
  console.log("Petición de estado del servidor recibida");
  if (userController.checkInactiveUser()) {
    res.send("true");
    console.log("El servidor parece estar bien.");
  } else {
    res.send("false");
    console.log("El servidor parece estar lleno");
  }
});
```

Código 5.19: Ruta de comprobación de servidor.

Nótese que el condicional utilizado sería *true* si el contador utilizado en la función del controlador de usuarios *checkInactiveUser()* tiene un valor distinto de cero:

```
app.get('/creds', function (req, res) {
  console.log("Petición de credenciales recibida. Enviando...");
  var jsonCr = JSON.stringify(creds);
  console.log("Datos enviados: %j", jsonCr);
  res.send(jsonCr);
});
```

Código 5.20: Obtención de credenciales.

Esta ruta a */creds* se encarga de pasar a la *Aplicación Web* el valor actual de la variable *creds* implementada al principio del código fuente del servidor. El valor de dicho objeto cambiará, obteniendo las credenciales de un usuario recién marcado como activo del controlador de usuarios, cuando un usuario que no esté conectado al servicio envíe un archivo de audio por primera vez al servidor.

```

app.post('/userlogoff', function (req, res) {
    console.log("Datos recibidos UserLogoff: %j", req.body);
    if (req.body.oneTimePass === false) {
        console.log('Recibida desconexión de usuario. Desconectando al usuario ' +
        req.body.id);
        userController.userInactive(req.body.id);
        res.end('success');
    }
});

```

Código 5.21: Desconexión de usuario.

También hace falta una forma de saber que un usuario de la Aplicación Web se ha desconectado. Cuando esto ocurre la aplicación Web manda un aviso al servidor por medio de una petición POST a la ruta `/userlogoff`, con lo que el servidor hace que el controlador de usuarios marque el usuario en cuestión como desconectado en la base de datos.

```

function setCreds() {
    console.log('Estableciendo credenciales...');
    var data = userController.getInactiveUser();
    creds.id = data[0];
    creds.username = data[1];
    creds.password = data[2];
    creds.oneTimePass = data[3];
    console.log("Credenciales establecidas: %j", creds);
}

```

Código 5.22: Almacenando las credenciales.

Esta función tiene de cometido modificar los valores del objeto `creds`, que pasan a contener las credenciales del usuario enviado desde el controlador de usuarios. Existe además una función similar a esta llamada `setCredsOneTime()` que se encarga de la misma tarea pero utilizando el controlador correspondiente a los usuarios con una conexión única de 30 minutos.

A continuación implantamos los manejadores de ruta más relevantes del *Servidor*, los que gestionan la llegada de archivos de audio y los guardan en el directorio `uploads` del sistema de archivos:

```

app.post('/upload', function (req, res) {
    console.log("Petición para subir audio recibida");
    // create an incoming form object
    var form = new formidable.IncomingForm();
    // store all uploads in the /uploads directory
    form.uploadDir = path.join(__dirname, '/uploads');
    // every time a file has been uploaded successfully,
    // rename it to its original name
    form.on('file', function (field, file) {
        fs.rename(file.path, path.join(form.uploadDir, file.name), function () {
            });
    });
    // log any errors that occur
    form.on('error', function (err) {
        console.log('An error has occurred: \n' + err);
    });
    // once all the files have been uploaded, send a response to the client
    form.on('end', function () {
        console.log("Audio subido con éxito.");
        setCreds();
        res.end('success');
    });
    // parse the incoming request containing the form data
    form.parse(req);
});

```

Código 5.23: Procesado de ficheros de audio procedentes del cliente.

```

app.post('/loggedupload', function (req, res) {
    console.log("Petición para subir audio recibida de un usuario ya conectado");
    var form = new formidable.IncomingForm();
    form.uploadDir = path.join(__dirname, '/uploads');
    form.on('file', function (field, file) {
        fs.rename(file.path, path.join(form.uploadDir, file.name), function () {
        });
    });
    form.on('error', function (err) {
        console.log('An error has occurred: \n' + err);
    });
    form.on('end', function () {
        console.log("Audio subido con éxito.");
        res.end('success');
    });
    form.parse(req);
});

```

Código 5.24: Procesado de ficheros de audio procedentes del cliente ya conectado.

```

app.post('/onetimempassupload', function (req, res) {
    console.log("Petición para subir audio recibida de un usuario que pide 30
    minutos sin más.");
    var form = new formidable.IncomingForm();
    form.uploadDir = path.join(__dirname, '/uploads');
    form.on('file', function (field, file) {
        fs.rename(file.path, path.join(form.uploadDir, file.name), function () {
        });
    });
    form.on('error', function (err) {
        console.log('An error has occurred: \n' + err);
    });
    form.on('end', function () {
        console.log("Audio subido con éxito.");
        setCredsOneTime();
        res.end('success');
    });
    form.parse(req);
});

```

Código 5.25: Procesado de ficheros de audio procedentes de un cliente de 30 minutos.

Estos tres manejadores de ruta son prácticamente idénticos en cuanto a que gestionan

la llegada de archivos de audio al sistema, la única diferencia que existe entre ellos radica que la ruta `/loggedupload` se utiliza cuando el archivo de audio proviene de una instancia de la *Aplicación Web* cuyo usuario ya está conectado al servicio —esto es, tiene acceso a Internet—, por lo que no hace falta buscar unas credenciales inactivas nuevas para que le sean proporcionadas. Por ello, en el *callback* asociado al evento de finalización de la transacción de archivos, el primer manejador de ruta (`/upload`) llama a la función `setCreds()` para preparar las credenciales que son enviadas con las peticiones a las rutas vistas anteriormente. La ruta `/onetimepassupload` se encarga de los archivos de audio procedentes de los usuarios que han optado por la conexión de 30 minutos. Nótese también la diferencia entre la ruta de la petición HTTP de la primera ruta (`upload`) y la ruta del directorio de destino en el sistema de archivos (`uploads`), que puede llevar a conclusión debido a la similitud de los nombres.

```
app.post('/userconnected', function (req, res) {
    console.log("Datos recibidos: %j", req.body);
    if (req.body.connected != 1) { // Antes era (!req.body.state)
        console.log("El usuario no ha logrado conectarse.");
        if (req.body.oneTimePass == true) {
            console.log("Liberando usuario de 30 minutos " + req.body.id + "...");
            userControllerOneTime.userInactiveOneTime(req.body.id);

        } else {
            console.log("Liberando usuario normal... " + req.body.id + "...");
            userController.userInactive(req.body.id);
        }
        res.end('success');
    } else if (req.body.oneTimePass == true) {
        console.log("CUENTA ATRÁS DEL USUARIO " + req.body.id + " ACTIVADA.");
        setTimeout(function () {
            console.log('Marcando como libre en la base de datos al usuario ' +
                req.body.id + ' con tiempo ya agotado...');
            userControllerOneTime.userInactiveOnetime(req.body.id);
        }, 1920000); //Cuenta atrás de 30 minutos hasta que se libere al usuario.
        res.end('success');
    }
});
```

Código 5.26: Comprobando si el usuario logró conectarse.

Este último manejador de ruta se encarga de comprobar si el usuario se ha conectado correctamente en cada caso, liberando el usuario de la base de datos si la conexión no ha podido realizarse y estableciendo una cuenta atrás de 32 minutos para el caso de los

usuarios que hayan escogido la opción de 30 m, tras la cual se libera el usuario utilizado.

Tras implementar todos los manejadores, solo queda implementar la función que pone en marcha el servidor al llamar al archivo *nodeserver.js* mediante el comando *node*, que como ya habíamos visto se consigue en escasas líneas de código. La única diferencia radica en que en este TFG se utiliza la opción segura HTTPS, dado que los navegadores ya no aceptan que un servicio Web tome los datos de audio y ubicación del dispositivo si el contexto no es seguro. Los pasos para obtener un certificado y clave SSL necesarios para implementar HTTPS se detallan a continuación.

Para habilitar el servidor con HTTPS en nuestra aplicación Node.js solo debemos utilizar el siguiente código.

```
const options = {
  key: fs.readFileSync('./ssl/key.pem'),
  cert: fs.readFileSync('./ssl/cert.pem'),
  passphrase: 'pruebassl'
};

https.createServer(options, app).listen(3000, function () { // DEFAULT PORT 443
  console.log('HTTPS server listening on port 3000');
});
```

Código 5.27: Preparando el servidor web con HTTPS.

Donde la variable *options* contiene los archivos de certificado, clave y la *passphrase* necesarios para ofrecer el servicio.

Para implementar correctamente transacciones con HTTPS se han de realizar una serie de configuraciones para habilitar *Secure Sockets Layer* (SSL) en nuestros servicios, generando los certificados y claves pertinentes.

Existen varios servicios y proveedores de certificados autorizados con diferentes planes de precios. En este TFG se han utilizado certificados auto-firmados para ofrecer el servicio, lo que no es nada recomendable en una implantación real puesto que los navegadores Web advierten al usuario sobre la presencia de un certificado SSL de estas características, teniendo que añadir excepciones manualmente en dos direcciones diferentes (la de nuestro portal cautivo y la dirección IP de la interfaz JSON, 1.0.0.1) para ignorar dichas advertencias.

Para generar nuestra clave y certificado SSL puede utilizarse la herramienta OpenSSL, disponible en sistemas macOS y Linux. Con un solo orden pueden generarse los dos

archivos necesarios.

```
openssl req -x509 -sha256 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365
```

Tras una serie de preguntas, entre las que se encuentra la *passphrase* utilizada en el servidor, se generan los dos archivos necesarios: *key.pem* y *cert.pem*. Pueden ubicarse estos archivos en un directorio aparte y luego referenciarse en las configuraciones pertinentes. Para que la Aplicación Web Node.js haga uso de estos archivos, como ya se explicó anteriormente, hay que declarar sus rutas junto a la *passphrase* en el objeto *options* que luego se utiliza en la función *createServer()*. Se vuelve a adjuntar el código utilizado por comodidad.

```
const options = {
  key: fs.readFileSync('./ssl/key.pem'),
  cert: fs.readFileSync('./ssl/cert.pem'),
  passphrase: 'pruebassl'
};

https.createServer(options, app).listen(3000, function () { // DEFAULT PORT 443
  console.log('HTTPS server listening on port 3000');
});
```

Código 5.28: Preparando el servidor web con HTTPS.

También hay que habilitar el uso de SSL en el objeto *chilliController*, definiendo el atributo necesario en el archivo *chilliController.js*, que analizaremos más adelante, con la declaración: *chilliController.ssl = true;*.

Por último, el propio *software CoovaChilli* debe tener habilitado SSL en su configuración y referenciar el certificado y clave en la ruta donde se encuentren. Para ello se edita el archivo */etc/chilli/config* tal y como se vio en apartados anteriores, definiendo los siguientes atributos.

```
HS_REDIRSSL=on
HS_SSLKEYFILE=/etc/chilli/sslkey/cert.pem
HS_SSLCERTFILE=/etc/chilli/sslkey/cert.pem
```

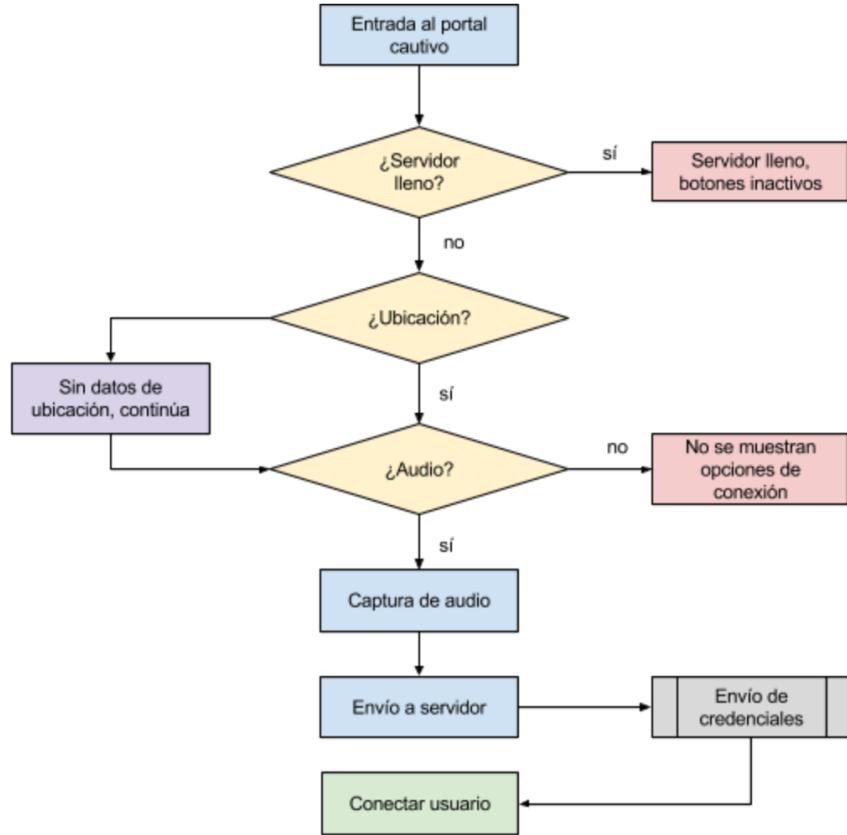


Figura 5.11: Organigrama que indica cómo cambian los botones.

5.2.4. Submódulos de Aplicación Web

Este submódulo es el que depende de un mayor número de archivos escritos en varios lenguajes y muy relacionados entre sí. Se procede a analizar uno por uno los más relevantes: *index.html*, *chilliController.js* y *script.js*.

El archivo *index.html*

El archivo HTML es el elemento principal del submódulo, enviado al navegador como respuesta a una petición HTTP GET al servidor Node.js, proporciona un entorno simple con un título, una breve descripción y tres botones. Esta descripción y la función de los botones mostrados cambian según las acciones llevadas a cabo por el usuario (Figura 5.11).

Usando Bootstrap se logra un diseño responsive. Esto es, la apariencia de la página Web mantiene unas proporciones estéticas adaptadas al tamaño del dispositivo utilizado, pudiendo utilizar cómodamente el servicio tanto desde un ordenador portátil como desde un teléfono móvil o una tablet. Se adjuntan capturas de pantalla de cómo se ve



Figura 5.12: Diseño responsivo de la página Web principal.

la interfaz Web en cada uno de los pasos desde un teléfono móvil (Figura 5.12).

El comportamiento de la aplicación Web aparte de su contenido visual vienen determinados por los *scripts* incluidos en el archivo HTML justo antes de cerrar la etiqueta *body*, que son analizados a continuación. Los archivos se encuentran en el directorio *js* de nuestro árbol de directorios.

```
<!DOCTYPE html>
<html lang="en">
<head>
// ...
</head>
<body>
// ...
<script src="js/jquery-3.2.1.min.js"></script>
<script src="js/bootstrap.min.js"></script>
<script src="js/ChilliLibrary.js"></script>
<script src="js/chilliController.js"></script>
<script src="js/script.js"></script>
</body>
</html>
```

Código 5.29: scripts importados en el documento HTML.

Los dos primeros *scripts* incluidos son los *frameworks* jQuery y Bootstrap, utilizados para facilitar la programación de peticiones AJAX y gestionar la interfaz de usuario, respectivamente, también se incluyó en la cabecera un archivo CSS de Bootstrap para funcionar junto al archivo JavaScript. Además, el *framework* Bootstrap por sí mismo requiere de jQuery para funcionar correctamente.

El archivo *ChilliLibrary.js*, como ya se adelantó en el capítulo 2, se encarga de crear

un objeto global denominado *chilliController* en nuestra *Aplicación Web*, mediante el cual podemos comunicarnos con CoovaChilli, gestionar el inicio y cierre de sesiones y consultar la información de contabilidad.

El archivo *chilliController.js* ajusta los parámetros del objeto *chilliController* y aporta los métodos a ser usados por la *Aplicación Web* actuando como capa intermedia entre el objeto *chilliController* y el resto de elementos de la misma, como puede verse en los métodos de *connect()* y *disconnect()* que son llamados por el archivo *script.js*.

```
chilliController.ssl = true;

function connect(username, password){
    console.log('Conectando...');

    if (username == "" || password == "") {
        console.log('Algo va mal... ¿Usuarios completos? User: '+username+'. Pass: '+password+'.');
        chilliController.logon(username, password);
    }
}

function disconnect(userCreds){
    console.log('Disconnecting...');

    liberateUser(userCreds);
    chilliController.logoff();
}
```

Código 5.30: Habilitación de SSL, conexión y desconexión.

La función *liberateUser()* se analiza en el apartado 5.2.4 dedicado al archivo *scripts.js*. Además, asocia funciones locales a eventos del objeto *chilliController* y se encarga de consultar el estado del mismo al ejecutarse por primera vez.

```

chilliController.onError = handleErrors;
chilliController.onUpdate = updateUI
// when the reply is ready, this handler function is called
function updateUI( cmd ) {
    console.log('You called the method ' + cmd +
        '\n Your current state is = ' + chilliController.clientState);
}
// If an error occurs, this handler will be called instead
function handleErrors ( code ) {
    console.log( 'The last contact with the Controller failed. Error code = ' +
        code );
}
// finally, get current state
chilliController.refresh();

```

Código 5.31: Funciones adicionales de chilliController.js

El archivo *script.js*

La funcionalidad de la Aplicación Web viene determinada en gran medida por este archivo JavaScript, que hace uso de los siguientes variables y métodos.

```

'use strict';

let id = val => document.getElementById(val),
    agreeBtn = id('agreeBtn'),
    recordBtn = id('recordBtn'),
    recordBtn30min = id('recordBtn30min'),
    alertsArea = id('alertsArea'),
    stream,
    recorder,
    chunks,
    media,
    serverStatus,
    locationTime;

var userCreds = {
    id: -1,
    username: "prueba",
    password: "pruebaPass",
    oneTimePass: false,
    connected: 0
};

```

Código 5.32: Funciones adicionales de chilliController.js

La primera variable utiliza la función flecha (*arrow function*) implementada en ECMAScript 6, y es equivalente a la sintaxis siguiente: `let id = function(val) { return document.getElementById(val) };`. Esta función es una forma abreviada de obtener los elementos del documento HTML, como puede verse en las cuatro siguientes variables: *agreeBtn*, *recordBtn*, *recordBtn30min* y *alertsArea* que corresponden a los elementos con ese identificador del documento HTML.

Las variables *stream*, *recorder*, *chunks* y *media* son utilizadas por las API WebRTC y *MediaStream Recording* para su funcionalidad, que veríamos en acción al analizar los métodos del archivo. Las variables *serverStatus*, *locationTime* y *userCreds* almacenan el estado del servidor en forma booleana, la latitud, longitud y marca de tiempo para usar en los nombres de los archivos de audio y las credenciales de usuario recibidas desde el servidor, respectivamente.

Al ser cargada la *Aplicación Web* se ejecutan los siguientes métodos por medio de *window.onload*.

```

window.onload = function () {
    prepareSite();
    checkServerStatus();
    setInterval(checkServerStatus(), 500000);
};

function prepareSite() {
    if (navigator.geolocation) {
        console.log('Intentando obtener ubicación...');
        try {
            navigator.geolocation.watchPosition(showPositionTime, positionError,
                geoOptions);
        }
        catch (err) {
            console.log("Error de ubicación: " + err);
            locationTime = 'LocError';
        }
    } else {
        console.log('Geolocation not supported');
        locationTime = 'Geolocation is not supported by this browser';
    }
}

function checkServerStatus() {
    console.log('Comprobando estado del servidor...');
    $.ajax({
        type: 'GET',
        url: '/serverstatus',
        dataType: 'text',
        success: function (data) {
            console.log('Respuesta recibida: ' + data);
            if (data === "true") {
                console.log('El servidor parece estar bien.');
                serverStatus = true;
            } else {
                console.log('Servidor parece lleno...');
                serverStatus = false;
                agreeBtn.disabled = true;
                agreeBtn.textContent = "Servidor lleno";
            }
        }
    });
}

```

Código 5.33: Preparando el portal cautivo según llega el usuario.

La función *prepareSite()* simplemente intenta activar la geolocalización del dispositivo a lo largo del tiempo mediante la función *watchPosition()*, lo que implica que nada más cargar la página aparece la petición de permisos de ubicación, que el usuario puede aceptar o negar sin que eso perturbe el resto de la experiencia de usuario. Obviamente, si el usuario niega los permisos el servicio no contaría con los datos de ubicación de los archivos de audio procedentes de esta instancia de la Aplicación Web, pero en principio se ha diseñado teniéndolo en cuenta.

La función *watchPosition()* cuenta con dos funciones *callback* (en orden, manejadores de éxito y error) como dos primeros parámetros y un tercer parámetro de opciones, que en este caso simplemente activa la alta precisión de los datos entregados a la aplicación Web. Aquí se ven las implantaciones de estos tres parámetros.

```
function showPositionTime(position) {
    console.log('Obteniendo ubicación y marca de tiempo...');

    locationTime = 'Lat' + position.coords.latitude +
        'Lon' + position.coords.longitude +
        'Time' + new Date(); // Esto añadiría también el Timestamp al nombre
}

var geoOptions = {
    enableHighAccuracy: true
};

function positionError(positionError) {
    console.log('Error ' + positionError.code + ' en la geolocalización: ' +
        positionError.message);
}
```

Código 5.34: Funciones y opciones para geolocalización.

Como puede verse, la función llamada al tener éxito en la solicitud de permisos de ubicación es la que almacena en la variable *locationTime* los datos de ubicación y marca de tiempo, tal y como comentábamos al principio de la sección dedicada a este archivo.

La función *checkServerStatus()* cuenta con la primera petición AJAX (de tipo GET a la ruta */serverstatus* que ya analizamos en la sección del servidor) realizada por el sistema. Aquí la *Aplicación Web* insta al servidor a que compruebe su estado, buscando si hay usuarios libres para que la instancia de la aplicación pueda unir al usuario al servicio. Se establece también un intervalo de tiempo pasado el cual se vuelve a consultar el estado del servidor.

También se implanta un manejador de eventos para desconectar al usuario en caso de que cierre la pestaña del portal cautivo, lo que logra el requisito de que el usuario tenga que tener una pestaña del navegador abierta con la instancia de la *Aplicación Web* en todo momento para no perder la conexión.

```
window.onbeforeunload = function (e) {
    if (userCreds.id >= 0) {
        disconnect(userCreds);
        userCreds.id = -1;
        //log('Disconnecting... ');
    }
    var dialogText = "Disconnecting...";
    e.returnValue = dialogText;
    return dialogText;
};
```

Código 5.35: Desconectando al usuario que abandona la página.

Este manejador llama a la función *disconnect()* del objeto *chilliController*, que a su vez llama a *liberateUser()*, ambas con el identificador de usuario actual como parámetro en caso de que estuviera conectado. La función *liberateUser()* no es más que otra petición AJAX al servidor con la ruta */userlogoff*, instándole a marcar el usuario como inactivo en la base de datos.

```
function liberateUser(creds) {
    console.log('Liberando usuario...');
    $.ajax({
        type: 'POST',
        url: '/userlogoff',
        data: creds,
        success: function (data) {
            console.log('success ' + data);
        }
    });
}
```

Código 5.36: Marcando usuario como desconectado en el servidor node.

Después de esto se programan los comportamientos de los botones. El primer botón que aparece en la visualización de la página es el que se debe pulsar al estar de acuerdo con las condiciones que se expresan en la primera descripción, teniendo este el identificador

agreeBtn. Este botón intenta configurar la captura de audio en el dispositivo según las API WebRTC y *MediaStream Recording*, configurando las opciones de la grabación (en nuestro caso declarando que se captura audio, utilizando el *MIME type* y la extensión de archivo correspondiente al formato *Ogg* [26], [27]) y determinando qué hacer con los archivos de audio generados según el tipo de usuario o si este tiene credenciales ya asignadas.

```

agreeBtn.onclick = e => {
    let mediaOptions = {
        audio: {                                     // Ajustes de sonido
            tag: 'audio',
            type: 'audio/ogg',
            ext: '.ogg',
            gUM: { audio: true }
        }
    };
}

media = mediaOptions.audio;
navigator.mediaDevices.getUserMedia(media.gUM).then(_stream => {
    stream = _stream;                           // Preparando la grabación 1
    id('gUMArea').style.display = 'none';
    id('preRecordArea').style.display = 'inherit';
    recorder = new MediaRecorder(stream);       // Preparando la grabación 2
    recorder.ondataavailable = e => {           // Preparando la grabación 3
        chunks.push(e.data);                    // Preparando la grabación 4
        if (recorder.state == 'inactive') {
            if (userCreds.id >= 0) {
                console.log("Guardando audio y enviando para usuario ya
                conectado");
                loggedUserSaveAndSend(); // guarda y envía
            } else if (userCreds.oneTimePass === false) {
                console.log("Guardando audio y enviando para usuario
                primerizo");
                saveAndSend(); // guarda y envía
            } else {
                console.log("Guardando audio y enviando para usuario de 30
                minutos");
                saveAndSendOneTimePass();
            }
        }
    };
    log('got media successfully');
}).catch(log);
};

```

Código 5.37: Preparación para grabar y enviar el fragmento de audio.

Lo que activa el evento *ondataavailable* es la finalización de una grabación de audio. Para controlar el inicio y final de estas grabaciones, así como su periodicidad, se utiliza una función que controla al segundo botón de la interfaz de usuario, que tiene el identificador *recordBtn*, así como las sencillas funciones de inicio y parada implementadas según la API *MediaStream Recording*. En caso de que el usuario seleccione la opción de conectarse

durante 30 minutos habría pulsado el botón con el identificador *recordBtn30min*, que tiene una estructura muy similar, solo variando en que no hay un intervalo regular donde la grabación vuelva a empezar.

```

recordBtn.onclick = e => {
    if (serverStatus === true) {
        console.log('El servidor parece estar bien...');
        userCreds.oneTimePass = false;
        id('preRecordArea').style.display = 'none';
        id('agreedArea').style.display = 'inherit';
        setTimeout(startRecording, 100);
        setInterval(startRecording, 180000);
    } else {
        console.log('Ha ocurrido un error en el servidor. ¿Podría estar
completo?');
    }
};

recordBtn30min.onclick = e => {
    if (serverStatus === true) {
        console.log('El servidor parece estar bien...');
        userCreds.oneTimePass = true;
        id('preRecordArea').style.display = 'none';
        id('agreedArea').style.display = 'inherit';
        setTimeout(startRecording, 100);
    } else {
        console.log('Ha ocurrido un error en el servidor. ¿Podría estar
completo?');
    }
};

function startRecording() {                                // Se ejecuta al pulsar el botón
Start
    chunks = [];                                         // Crea un array
    recorder.start();                                    // Empieza a grabar
    setTimeout(stopRecording, 5000);
}

function stopRecording() {
    recorder.stop();
}

```

Código 5.38: Grabando al ser autorizado por el usuario.

Como también puede verse en el último condicional de la función *agreeBtn.onclick*, la

anterior al último bloque de código, la función que se llamaría al final de cada grabación depende de si el usuario del servicio tiene unas credenciales asignadas o no en la variable *userCreds* de su instancia de la aplicación Web o si, por el contrario, solo optó por conectarse durante 30 minutos. Estas tres funciones son muy similares en cuanto a que preparan el archivo para su envío al servidor mediante AJAX y lo renombran según los datos de ubicación y tiempo obtenidos y las opciones de audio, pero las rutas son diferentes y solo se intenta obtener credenciales en el caso de que no existan ya unas asignadas.

```
function saveAndSend() {
    let blob = new Blob(chunks, { type: media.type });
    var fd = new FormData();
    fd.append('blob', blob, `${locationTime}${media.ext}`);
    console.log('[Normal] Enviando audio al servidor...');

    $.ajax({
        url: '/upload',
        type: 'POST',
        data: fd,
        processData: false,
        contentType: false,
        success: function (data) {
            console.log('[Normal] upload successful! ' + data);
            receiveResponse();
            setAlert("success");
        },
        error: function (data) {
            console.log('[Normal] upload error ' + data);
            setAlert("error");
        }
    });
}
```

Código 5.39: Guardando y enviando el fragmento de audio (usuario recién llegado).

```
function loggedUserSaveAndSend() {
    let blob = new Blob(chunks, { type: media.type });
    var fd = new FormData();
    fd.append('blob', blob, `${locationTime}${media.ext}`);
    console.log('[Logged] Enviando audio al servidor...');

    $.ajax({
        url: '/loggedupload',
        type: 'POST',
        data: fd,
        processData: false,
        contentType: false,
        success: function (data) {
            console.log('[Logged] upload successful! ' + data);
            setAlert("success");
        },
        error: function (data) {
            console.log('[Logged] upload error ' + data);
            setAlert("errorLogged");
        }
    });
}
```

Código 5.40: Guardando y enviando el fragmento de audio (usuario ya conectado).

```

function saveAndSendOneTimePass() {
    let blob = new Blob(chunks, { type: media.type });
    var fd = new FormData();
    fd.append('blob', blob, `${locationTime}${media.ext}`);
    console.log('[One-time] Enviando audio al servidor...');

    $.ajax({
        url: '/onetimempassupload',
        type: 'POST',
        data: fd,
        processData: false,
        contentType: false,
        success: function (data) {
            console.log('[One-time] upload successful! ' + data);
            receiveResponse();
            setAlert("successOneTime");
        },
        error: function (data) {
            console.log('[One-time] upload error ' + data);
            setAlert("error");
        }
    });
}

```

Código 5.41: Guardando y enviando el fragmento de audio (usuario de 30 minutos).

Nótese que en la segunda función, *loggedUserSaveAndSend()*, aparte de variar la ruta, no se llama a la función *receiveResponse()* al finalizar la transacción con éxito. En el caso de la primera función, el servidor habría guardado en su variable *creds* las credenciales a utilizar, por lo que la *Aplicación Web* realiza otra petición AJAX para obtenerlas.

```
function receiveResponse() {
    console.log('Pidiendo credenciales...');
    $.ajax({
        type: 'GET',
        url: '/creds',
        dataType: 'json',
        success: function (data) {
            console.log('respuesta recibida: ' + data + '. Conectando...');

            console.log("Datos recibidos: %j", data);
            userCreds = data;
            console.log("Credenciales establecidas: %j", userCreds);
            getUserCredentials(userCreds);
        }
    });
}
```

Código 5.42: Recibiendo credenciales del servidor.

Al tener éxito en esta petición las credenciales quedan almacenadas en la variable local *userCreds*, por lo que solo queda conectarse con ellas por medio la función *connect()* del objeto *chilliController*, esperar un tiempo para que se realice la conexión y notificar al servidor del resultado del intento.

```

function getUserCredentials(data) {
    console.log('Conectando con username: ' + data.username + ' y password: ' +
    data.password);
    connect(data.username, data.password);

    setTimeout(2500, function () {
        if (chilliController.clientState === 1) {
            console.log("Estado de conexión a CoovaChilli: " +
            chilliController.clientState);
            userCreds.connected = chilliController.clientState;
            $.ajax({
                type: 'POST',
                url: '/userconnected',
                data: userCreds,
                success: function (data) {
                    console.log('success ' + data);
                }
            });
        } else if (chilliController.clientState === 2) {
            console.log("Estado de conexión a CoovaChilli: " +
            chilliController.clientState);
            setAlert("error");
        }
    });
}

```

Código 5.43: Intentando la conexión y comprobando el resultado.

Por último, la *Aplicación Web* tiene un campo de alertas que notifican al usuario de la recepción de archivos de audio o errores durante las mismas mediante la función *setAlert()*, llamada por otras funciones de este archivo ya expuestas anteriormente.

```

function setAlert(info) {
    var newDiv = document.createElement("div");
    switch (info) {
        case "success":
            newDiv.className = "alert alert-success";
            newDiv.role = "alert";
            newDiv.innerHTML = "<strong>¡Perfecto!</strong> Tu fragmento de audio
se ha subido con éxito.";
            break;
        case "error":
            newDiv.className = "alert alert-danger";
            newDiv.role = "alert";
            newDiv.innerHTML = "<strong>¡Vaya!</strong> Ha habido un error... Aún
no tienes internet. <strong>Trata de conectarte de nuevo.</strong>";
            break;
        case "errorLogged":
            newDiv.className = "alert alert-danger";
            newDiv.role = "alert";
            newDiv.innerHTML = "<strong>¡Vaya!</strong> Ha habido un error
enviando el fichero... Volveremos a intentarlo más tarde.";
            break;
        case "successOneTime":
            newDiv.className = "alert alert-success";
            newDiv.role = "alert";
            newDiv.innerHTML = "<strong>¡Perfecto!</strong> Tu fragmento de audio
se ha subido con éxito. <strong>Ya puedes cerrar esta pestaña en tu
navegador y disfrutar de tus 30 minutos de internet.</strong>";
            break;
    }
    alertsArea.appendChild(newDiv);
}

```

Código 5.44: Sistema de alertas para informar al usuario.

6. Análisis de la evaluación empírica

Presentados la instalación del *hardware* necesario y el análisis orgánico del software usado y programado desde cero, subrayar que por las características mínimas de la interfaz de usuario no es necesario dedicar más análisis que el realizado hasta el momento. En este capítulo presentamos los resultados experimentales de la evaluación del sistema con distintos tipos de terminales móviles.

The screenshot shows the Chrome DevTools Network tab with several network requests listed. The requests are as follows:

- Guardando audio y enviando para usuario primerizo [Normal] Enviendo audio al servidor... script.js:128
- Obteniendo ubicación y marca de tiempo... script.js:128
- [Normal] upload successful! success script.js:156
- Pidiendo credenciales... respuesta recibida: (object Object). Conectando... script.js:216
- Datos recibidos: {} script.js:223
- > (id: #, username: "Pruebad", password: "ValQ2Mh0", oneTimePass: false, connectable: true, establecidas: 1) script.js:223
- > (id: #, username: "Pruebad", password: "ValQ2Mh0", oneTimePass: false, connectable: true, establecidas: 1) script.js:223
- Conectando con username: Pruebad y password: ValQ2Mh0 script.js:223
- Conectando... chilliController.js:12
- Estado de conexión a CoavaChilli: 2 script.js:236
- success success script.js:243
- You called the method logon Your current state is = 1 chilliController.js:29
- Obteniendo ubicación y marca de tiempo... script.js:284

Below the Network tab, the Console tab shows the command `logoff()` being run.

Figura 6.1: Datos relevantes del proceso de conexión con éxito de una tableta Android.

6.1. Resultados con una tableta Android

A continuación se muestra cómo un dispositivo portátil (en este caso una *tablet* Samsung Galaxy GP-P5210, de 1 GB de RAM y con el sistema operativo Android 4.4.2) puede enviar su audio, recibir las credenciales en respuesta y conectarse a Internet. Las pruebas se hicieron con el navegador Chrome 61 y se monitorizaron los procesos con las herramientas de depuración remota de *Chrome DevTools* [28].

En la Figura 6.1 se muestran resaltados los datos relevantes en el proceso de conexión exitoso, en orden descendiente, correspondientes al guardado y envío de audio, recepción de credenciales y conexión con éxito, como revela el código de estado 1 al llamar a la función *logon()*.

Como se puede apreciar, cuando se envía el fichero de audio generado al servidor este responde con un objeto JavaScript que contiene las credenciales del tipo de usuario seleccionado. Tras recibir estos datos, el sistema realiza el *login* con dichas credenciales, conectando al usuario a internet (como puede verse en el código de estado de la conexión, que es 1).

Dado que el proceso de desconexión automática implica cerrar la pestaña, lo que cancelaría la sesión de depuración en curso, se realizó una desconexión manual llamando a la función *logoff()*. En la captura puede verse por orden descendiente el estado actual de la conexión a Internet tras llamar a la función *refresh()* y la llamada a la función de desconexión y el código de estado resultante a estar desconectado (Figura 6.2).

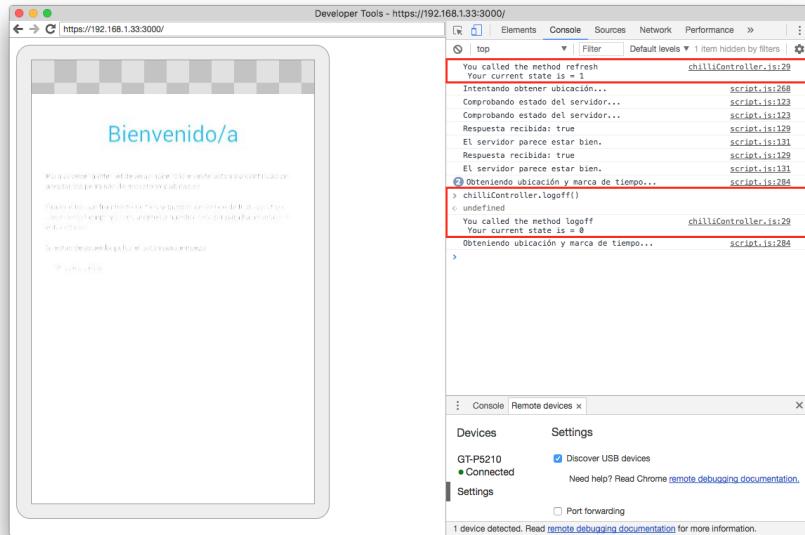


Figura 6.2: Proceso de desconexión.

6.2. Consumo de RAM del sistema en la Raspberry Pi

Tras realizar las pruebas de funcionamiento pertinentes se analizó el consumo total de memoria de los procesos implicados en la implantación del sistema. Utilizando la herramienta *htop*, disponible en sistemas Linux y que monitoriza la memoria física, memoria virtual, memoria de intercambio *swap* y consumo de CPU tanto en valor absoluto como en porcentajes entre otras muchas cosas, se monitorizaron las estadísticas concernientes a las instancias de Node.js, *hostapd*, *CoovaChilli* y la estructura en la que se apoya: *freeRADIUS* y *MySQL*.

En la Figura 6.3 se muestra la ventana genérica al ejecutar el comando *htop* desde el terminal.

En la Figura 6.4 se muestra el consumo de recursos de Node.js.

En esta captura puede apreciarse cómo el proceso que controla la instancia de la utilidad PM2 (que a su vez es la que controla la ejecución de nuestra instancia de *node*) es la que destaca en el uso de recursos, utilizando un porcentaje de memoria del 5.9 % y un 14.2 % de la CPU. Lo que supone un bajo consumo.

En la Figura 6.5 se muestra una cifra más inmediata del consumo total de nuestro proceso con la propia utilidad PM2, resaltada en azul en el campo superior izquierdo del monitor llamado con el comando *pm2 monit* (que se explica en el Anexo C). Que también se observa que es muy reducido.

En la Figura 6.6 se muestra el consumo de recursos de CoovaChilli. Se puede observar

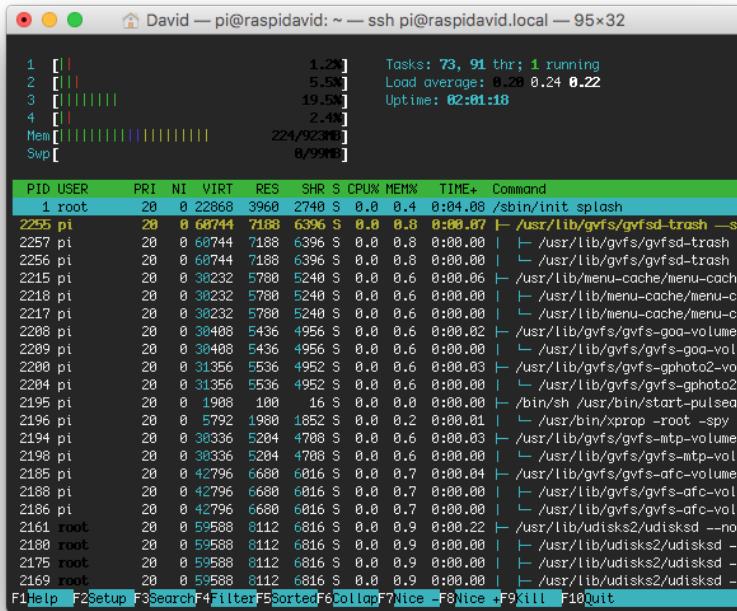


Figura 6.3: Ventana genérica para la orden htop.

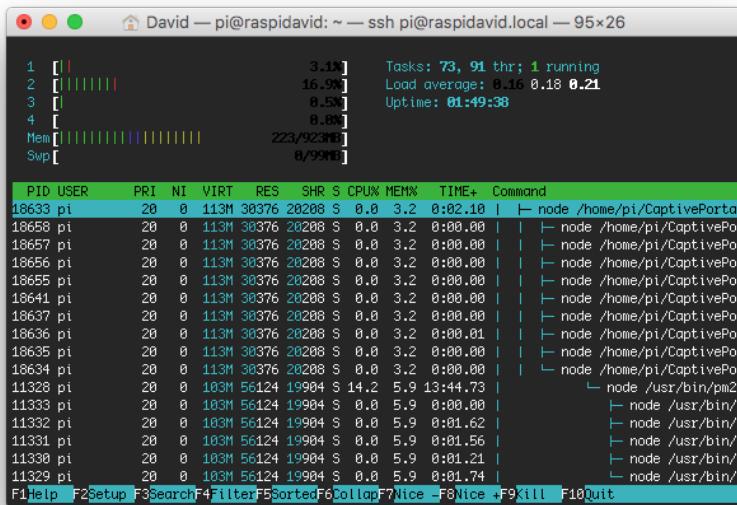


Figura 6.4: Consumo de recursos de Node.js.

que el proceso correspondiente a CoovaChilli es ligero, no ocupando apenas procesador y un porcentaje muy bajo de la memoria total de la Raspberry Pi.

En la Figura 6.7 se muestra el consumo de recursos de freeRADIUS. Se puede observar que consume poco procesador, aunque su consumo en memoria es varias veces mayor al

The screenshot shows the PM2 Dashboard interface. At the top, it displays "Process list" with one process named "nodeServer" and "Mem: 29 MB". Below this is a "Global Logs" section with a scrollable list of log entries related to the node server's activity, such as stopping the app, receiving signals, and handling user requests. To the right of the logs is a "Metadata" section containing details like Node.js version (0.7.0), watch & reload status, and commit information. At the bottom, there are navigation instructions ("left/right: switch boards | up/down/mouse: scroll | Ctrl-C: exit") and a link to "https://keymetrics.io/".

Figura 6.5: Ventana de consumo usando la orden pms monit.

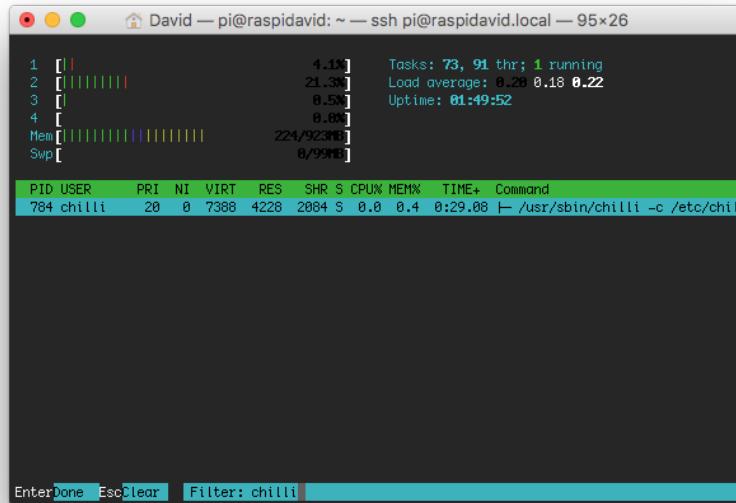


Figura 6.6: Consumo de procesador y memoria de CoovaChilli.

de CoovaChilli.

En la Figura 6.8 se muestra el consumo de recursos de MySQL. Tal y como sucede con freeRADIUS, el consumo de procesador es muy bajo aunque su consumo de memoria es de hasta diez veces mayor al de CoovaChilli. Lo cual es lógico porque CoovaChilli requiere únicamente el almacenamiento de muy pocos datos relativos a conversiones de direcciones IP, en cambio ya únicamente el motor de la MySQL requiere de mucha más

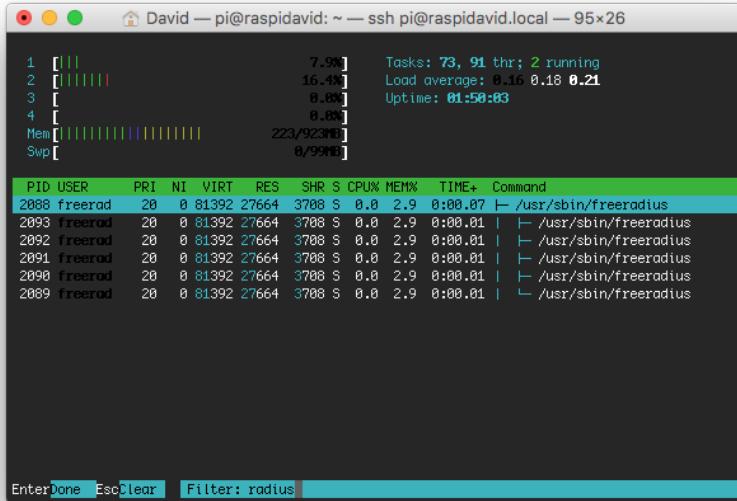


Figura 6.7: Consumo de procesador y memoria de freeRADIUS.

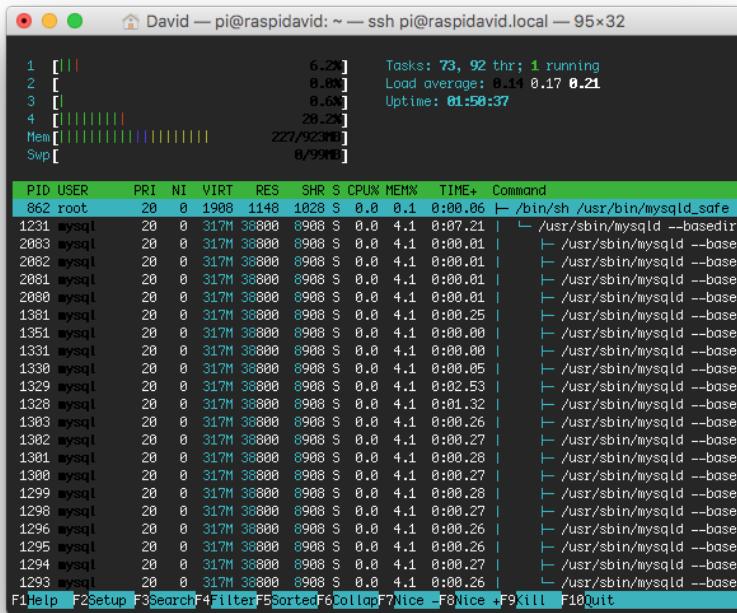


Figura 6.8: Consumo de procesador y memoria de MySQL.

memoria.

Por último, en la Figura 6.9 mostramos los recursos utilizados por *hostapd*, cuyas estadísticas de consumo son similares al proceso de CoovaChilli.

Con estos datos podemos considerar que el sistema implementado consume approxima-

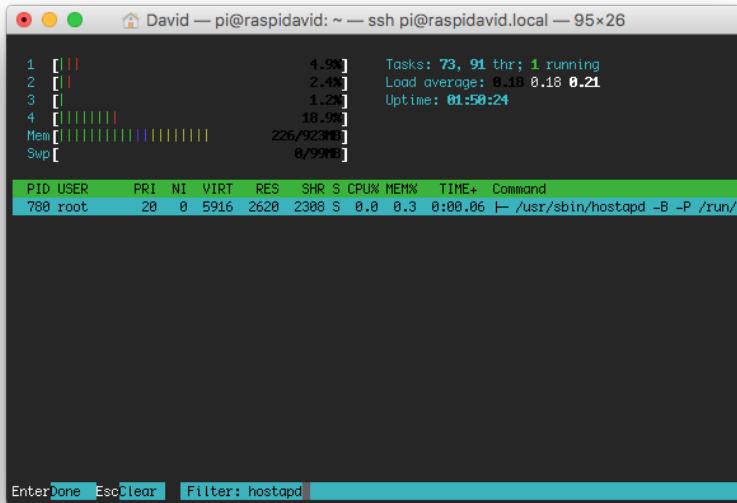


Figura 6.9: Consumo de procesador y memoria de Hostapd.

damente el 15 % de los recursos del procesador y memoria RAM de la Raspberry Pi 3. Sin embargo, hay que tener en cuenta que, aparte de un ligero sobredimensionamiento de los porcentajes, muchos procesos no se dedican exclusivamente al mantenimiento de nuestro sistema, pudiendo estar consumiendo las cantidades de recursos mostradas aquí debido a que también se dedican a otras funciones (como puede ser el caso de las múltiples instancias de los procesos de MySQL).

6.3. Consumo de RAM de los navegadores Web

Aparte del consumo de recursos del servidor, también se estudió el consumo de recursos del sistema con el que los clientes se conectan a nuestro servicio, que en este caso no es otra cosa que el navegador Web.

En la Figura 6.10 se muestra una captura de ventana de uno de los dispositivos móviles utilizados, una tablet Samsung Galaxy GP-P5210, de 1 GB de RAM, con Android 4.4.2 y el navegador Chrome 61.

En las pruebas de escritorio se ha analizado el consumo de los navegadores Mozilla Firefox 57 y Vivaldi 1.12 en un equipo con macOS 10.12.6, de 16 GB de RAM, y Microsoft Edge en un equipo con Windows 10 con 8 GB. En la Figura 6.11 se muestra el consumo de RAM en Mozilla Firefox para MacOS, resaltado en azul. El consumo de RAM de esta versión de Firefox puede parecer elevado sobre todo si lo comparamos con el consumo en la tablet, ya que incluso el consumo sin ningún contenido abierto

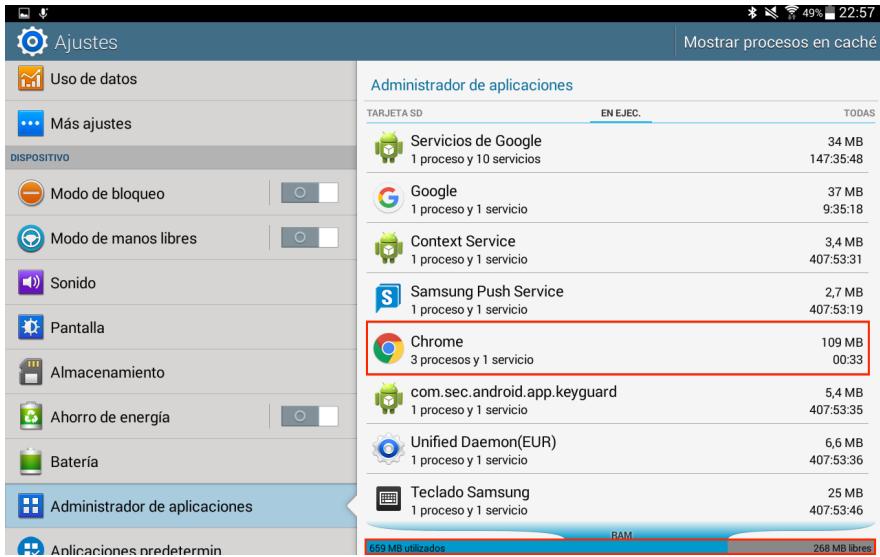


Figura 6.10: Ventana de consumo de RAM de los navegadores Web en una tablet Android.

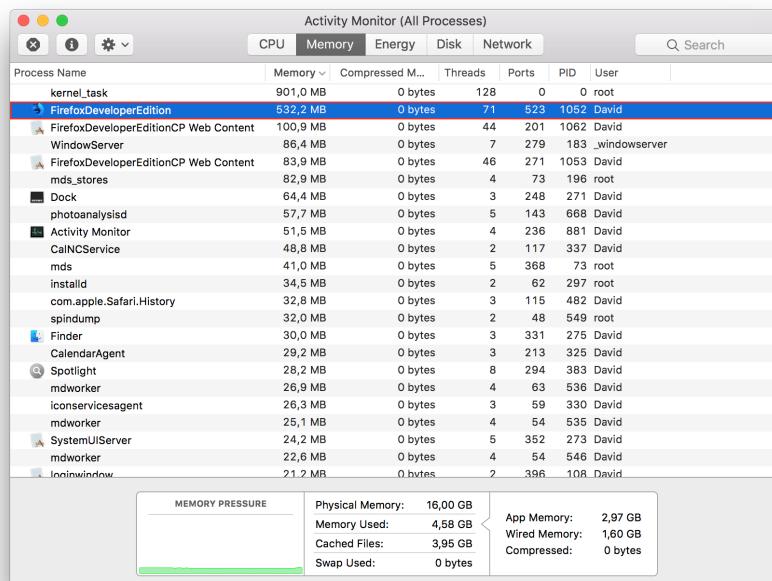


Figura 6.11: Consumo de RAM de Mozilla Firefox para un computador MacOS.

y en una red diferente es tan solo entre 25 y 75 MB menor. Esto se debe en parte a las extensiones específicas y herramientas de desarrollador que vienen instaladas por defecto en esta versión.

El consumo de recursos en Vivaldi sin extensiones ni complementos se mantiene en unas cifras inferiores sin tener en cuenta la presencia de los Vivaldi Helpers, aunque en total el consumo sigue siendo inferior a Firefox en escritorio y superior a Chrome en Android (Figura 6.12).

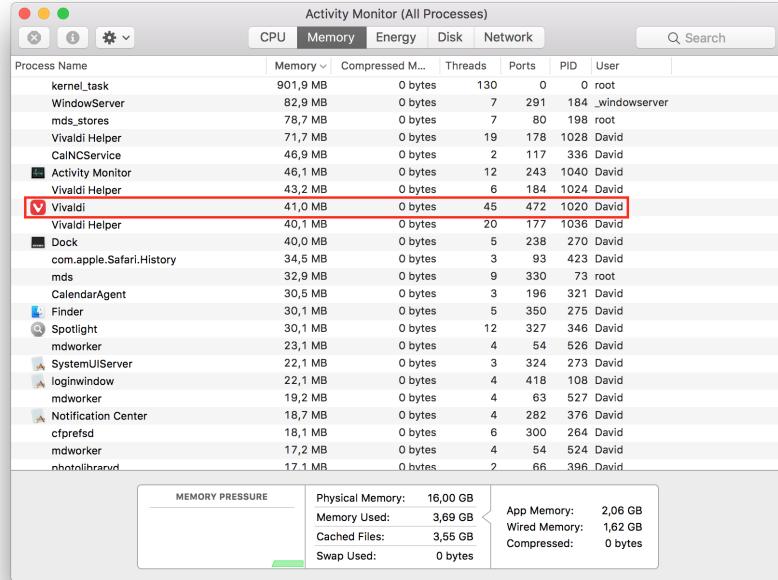


Figura 6.12: Consumo de RAM de Vivaldi para un computador MacOS.

El consumo de RAM en Microsoft Edge, resaltado en rojo, es mucho menor. Dado que la transferencia de archivos de audio por la red ocurre de forma puntual y a intervalos muy cortos de tiempo, los indicadores de consumo de red son nulos en el momento de tomar la captura debido a que se encontraban entre envíos (Figura 6.13).

6.4. Tráfico HTTP y TCP

En una de las pruebas del sistema con el navegador de escritorio Firefox y cuando aún no se había implementado SSL se realizó una captura del tráfico con Wireshark. En la Figura 6.14 se muestra el *throughput* medio de la transmisión de audio desde el navegador al servidor, consistente en 45 tramas TCP de una longitud total de 64206 bits.

Además, en las pruebas con sistemas de escritorio se realizó una prueba de velocidad del cliente conectado a través de nuestro sistema y se comparó con pruebas de velocidad cuando el usuario está conectado al mismo servicio contratado a la WiFi tradicional proporcionada por el equipo del operador. Las cifras para la conexión inalámbrica cuando se usa nuestro sistema en la comparativa descienden en 10 Mbps tanto en subida como en bajada, lo que es esperable dado que estamos añadiendo varias capas más de procesamiento a la transmisión (Figura 6.15).

Nombre	CPU	Memoria	Disco	Red
Aplicaciones (3)				
> Administrador de tareas	0,4%	19,3 MB	0,1 MB/s	0 Mbps
e Microsoft Edge	0%	22,0 MB	0 MB/s	0 Mbps
> Recortes	0%	2,8 MB	0 MB/s	0 Mbps
Procesos en segundo plano (4...)				
Aislamiento de gráficos de disp...	0,9%	8,7 MB	0 MB/s	0 Mbps
> Aplicación de subsistema de cola	0%	8,2 MB	0 MB/s	0 Mbps
Application Frame Host	0%	5,8 MB	0 MB/s	0 Mbps
Browser_Broker	0%	3,0 MB	0 MB/s	0 Mbps
COM Surrogate	0%	1,5 MB	0 MB/s	0 Mbps
COM Surrogate	0%	2,0 MB	0,2 MB/s	0 Mbps
Cortana	0,2%	91,0 MB	0,2 MB/s	0 Mbps
Device Association Framework ...	0%	4,7 MB	0 MB/s	0 Mbps
> Epson Scanner Service (64bit)	0%	1,0 MB	0 MB/s	0 Mbps

Figura 6.13: Consumo de RAM de Microsoft Edge para un computador con Windows.

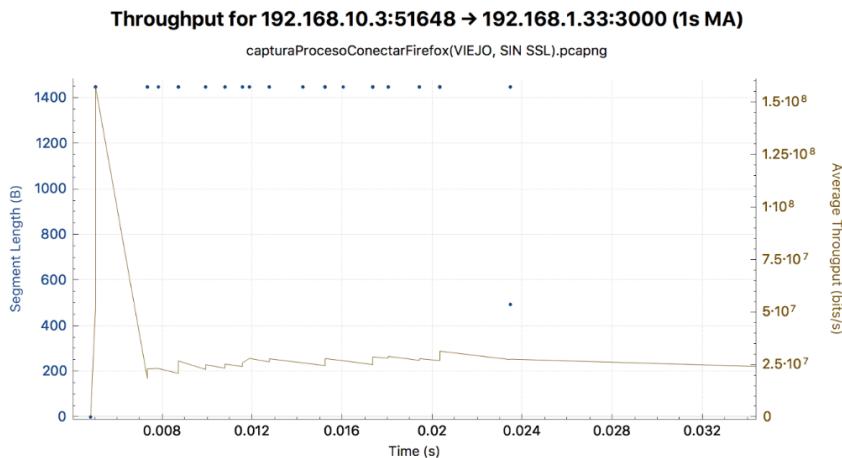


Figura 6.14: Throughput de tráfico TCP y HTTP para comunicación de un archivo de audio.

6.5. Análisis de los archivos de audio generados

Los archivos de audio generados y enviados al servidor son renombrados por la *Aplicación Web* siguiendo la estructura de la Figura 6.16: el prefijo *Lat* seguido de la latitud, el prefijo *Lon* seguido de la longitud y el prefijo *Time* seguido del *timestamp* completo.

Cada archivo tiene una duración de 4 segundos, está codificado en *Opus Audio* [29] y

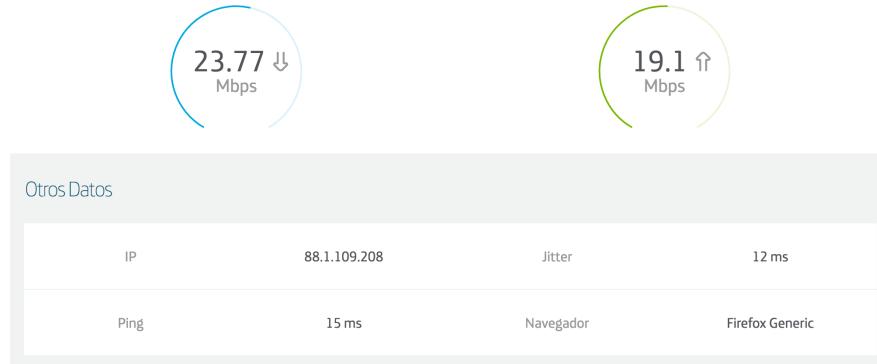


Figura 6.15: Prueba de velocidad de acceso a Internet a través de nuestro sistema.

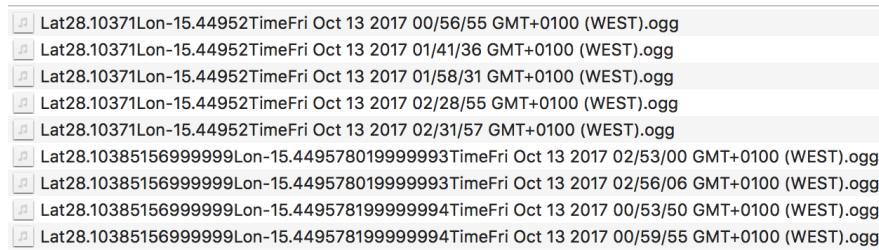


Figura 6.16: Formato del archivo de audio etiquetado con geolocalización y temporización.

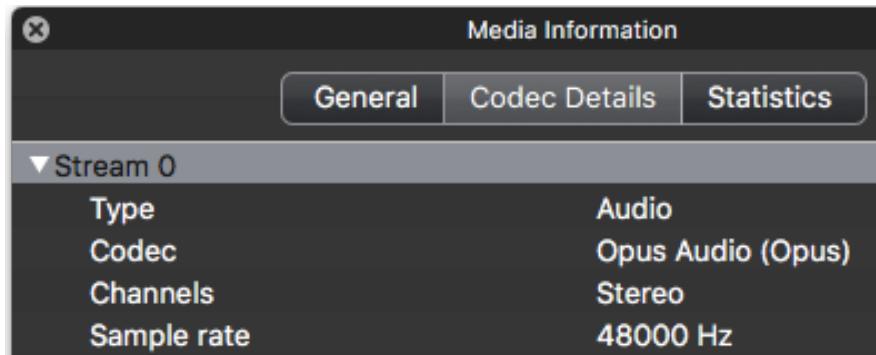


Figura 6.17: Codificación del audio.

almacenado en un contenedor de formato *Ogg*. La tasa de muestreo de la codificación es *fullband* (48 kHz, como puede verse en la imagen). El fichero generado ronda los 63 KB de tamaño, lo que confirma lo apreciado en la captura del tráfico TCP correspondiente a un fichero enviado, de tamaño total ligeramente mayor y dividido en varias tramas (Figura 6.17).

7. Conclusiones y posibles ampliaciones

Presentados los resultados experimentales de la evaluación del sistema con distintos tipos de terminales móviles. En este capítulos presentamos las conclusiones principales así como posibles ampliaciones.

7.1. Conclusiones

Se ha implementado un sistema de acceso a la red a cambio de audio recogido por el micrófono cuya interacción con los clientes se realiza de forma muy sencilla y sin hacer uso de aplicaciones o dispositivos externos al terminal, funcionando a través del navegador Web. El sistema al completo es transversal, ya que las diferentes módulos hardware y software utilizados se organizan en una arquitectura de niveles similar a la de la arquitectura de Internet. Además, para la Aplicación Web se han usado tecnologías en pleno desarrollo actual y con proyección de futuro como son Node.js y WebRTC.

Una vez probado en diversos dispositivos móviles de diferente tipo y gama (ordenadores, móviles y tablets) podemos determinar que el sistema desarrollado cumple con los objetivos planteados en la propuesta de TFG. Además, durante el desarrollo del mismo se ha adquirido gran familiaridad con las tecnologías implicadas, como las diferentes API y frameworks utilizados, y se ha comprobado cómo estas, particularmente las más recientes, evolucionan en el tiempo cambiando las condiciones actuales y futuras del sistema.

Un ejemplo de ello son las versiones de Node.js, que durante el desarrollo del TFG y en el momento de escribir esta conclusión pasaron de la 7 a la 9, introduciendo diversos cambios. Sin embargo, un ejemplo de hecho más relevante es la decisión de Apple de implementar soporte WebRTC en su motor de renderizado web, *WebKit*, durante el desarrollo de este TFG, lo que ocasiona que desde la versión 11 de Safari, lanzado durante la escritura de este documento, algunas opciones de WebRTC puedan implementarse ahora en los dispositivos iOS. Lamentablemente, la API *MediaStream Recording* que utilizamos en este TFG sigue sin tener soporte, pero es de esperar que tal y como sucedió con otros aspectos de WebRTC se empiece a trabajar en ello próximamente.

Un inconveniente de trabajar con tecnologías de este tipo es que, a diferencia de lo mencionado en el párrafo anterior, no todos los cambios introducidos son positivos. En cualquier momento podría lanzarse una nueva versión de la API y una revisión de los navegadores que obligue a los desarrolladores a modificar su proyecto o que complique el mantener una sintaxis de código final consistente.

7.2. Posibles ampliaciones futuras

Existen varios aspectos a tener en cuenta a la hora de llevar a cabo este proyecto. A continuación se detallan algunos de ellos junto a posibles ampliaciones que pueden llevarse a cabo en el futuro.

7.2.1. Replicabilidad

En este Trabajo Fin de Grado se detalla paso a paso cómo conseguir el producto final, descargando y configurando todo el software y realizando el código desde cero. Sin embargo, el hecho de que el sistema esté implementado sobre una Raspberry Pi 3 permite replicar el sistema con gran facilidad. Al estar el sistema operativo en su totalidad almacenado en una tarjeta MicroSD, puede obtenerse un archivo imagen de esta y luego clonarse en todas las tarjetas de este tipo que se desee, obteniendo múltiples copias del sistema en las que solo haría falta modificar unos pocos archivos y configuraciones (como el archivo *config* de CoovaChilli o la contraseña del usuario *root* de Raspbian) para adecuarlo a las necesidades específicas de la implementación.

7.2.2. Privacidad

La privacidad se ha convertido en uno de los elementos más cruciales, analizados y debatidos en el campo de las tecnologías de la información y la comunicación. El sistema desarrollado en este TFG está diseñado para capturar una muestra de audio de los usuarios del servicio cada cierto tiempo, almacenando a priori un tipo de información potencialmente sensible en un servidor ajeno. En las implementaciones de este servicio el usuario debería ser plenamente consciente de lo que está ocurriendo con la información que se obtendría de él, por lo que es ético que la información ofrecida en el portal cautivo y las condiciones de uso sean lo más transparentes posibles para el usuario final.

Otra opción, que queda fuera del ámbito de este trabajo, es el cifrado de los datos transferidos o el uso de otras herramientas de anonimización que puedan emplearse a la hora de implementar el servicio, o que directamente el audio no sea transferido, sino que según las necesidades del sistema (como por ejemplo si solo se desea obtener el nivel de intensidad sonora) este sea procesado directamente en la *Aplicación Web* y se transmita tan solo un valor o resultado numérico que no proporcione información sensible sobre el usuario.

7.2.3. Procesado del audio obtenido

Aunque en un principio se deseaba entrar en este aspecto, las restricciones de tiempo hicieron imposible dedicar recursos a los posibles procesados de los archivos de audio generados por el sistema. A continuación se mencionan algunos de estos posibles usos.

- *Mapeo de nivel de audio:* uno de los usos más inmediatos que son posibles en este sistema es el de realizar mapas de ruido en el área de cobertura del punto de acceso

utilizando los ficheros generados y la información de ubicación que les acompaña. La información de estos mapas de nivel puede volverse más fiable si existen varios puntos de acceso que tomen medidas de dispositivos cliente cercanos. En estos casos, sería conveniente también obtener la información del dispositivo que ha grabado el audio para poder aplicar las correcciones y ponderaciones pertinentes a la muestra.

- *Localización acústica*: si contamos con varios puntos de acceso y grupos de usuarios cercanos entre sí en los que haya al menos un usuario conectado a cada punto (algo que podría lograrse con técnicas de balance de carga) podrían utilizarse técnicas de procesado de señal para intentar obtener la ubicación de estos grupos de forma acústica, atendiendo a la diferencia de fase e intensidad de las señales obtenidas. En estos casos podemos prescindir del servicio de geolocalización que nos proporciona el navegador web, utilizado en nuestro sistema, o continuar usándolo para realizar comparativas.

7.2.4. Desacoplar procesos

El *hardware* de acceso a la red y el servidor web pueden implementarse en sistemas separados, tales como un *gateway* dedicado al acceso a la red y un servidor del portal cautivo y la gestión de AAA. Mediante el uso del *software* mencionado anteriormente (*iptables*, *dnsmasq*...) puede obtenerse un mayor control sobre las conexiones y el ancho de banda utilizado, separándolo por ejemplo en grupos de usuarios, varias redes, filtrando por MAC, implementando redes privadas con VPN... todo ello sin tener que recurrir a CoovaChilli.

7.2.5. Unificar *software*

Por el contrario, se puede seguir con un dispositivo *hardware* único conectado a la red y centralizar toda la gestión en Node.js, haciendo que este sirva el portal cautivo y también dé de alta a los usuario en la red accediendo directamente a la base de datos e incluso ejecutando los órdenes o *scripts* de *iptables* directamente. Esto incrementaría la complejidad de la aplicación Node.js pero se obtendría a cambio una gestión centralizada del sistema al completo, ahorrando en elementos potencialmente superfluos como las bases de datos de usuarios intermedias que existen en la implementación actual.

Apéndices

Apéndice A Instalación del *hardware* necesario

Una vez revisado el análisis previo y funcional del sistema de sensado móvil colaborativo mediante una visión *top-down*, en este capítulo presentamos el análisis orgánico de la implantación mediante una visión *bottom-up*. Esto es, primero se configura el nivel inferior, la capa del *hardware* y el Control de Red, tras lo cual se va ascendiendo en el modelo hasta llegar a la implantación del portal cautivo, en la capa de Usuarios y el nivel más alto del sistema total.

En este Apéndice se asume que se dispone de conexión a Internet operativa a la que la Raspberry Pi 3 pueda conectarse (cableada o inalámbrica) y un ordenador de trabajo, no entrando en detalles sobre su configuración por entender que no es parte del TFG.

Para poner en marcha la Raspberry Pi 3 de forma sencilla es necesario disponer de la propia placa, una fuente de alimentación USB Micro (un adaptador de corriente como el de la mayoría de teléfonos móviles actuales), teclado y ratón, una tarjeta microSD de al menos 8 GB clase 4 y un monitor con entrada HDMI, para lo que puede servir una pantalla de televisión. También hace falta disponer de conexión a Internet, ya sea mediante red cableada o una red inalámbrica a la que la Raspberry Pi 3 pueda conectarse.

La tarjeta microSD debe de tener ya almacenado el sistema operativo Raspbian desde el principio o por el contrario hacer uso de la solución recomendada desde la propia página Web oficial, la herramienta *New Out Of Box Software* (*NOOBS*, que en inglés significa *novatos*). Esta herramienta es un gestor de instalaciones de sistemas operativos para la Raspberry Pi, disponible desde su página Web oficial. Puede comprarse la tarjeta SD con dicha herramienta preinstalada o puede instalarse manualmente haciendo uso de un ordenador, descargando el archivo comprimido de NOOBS y descomprimiéndolo en una tarjeta SD vacía y formateada en FAT32, de forma que los archivos estén en el directorio raíz de la tarjeta. Tras este proceso la tarjeta queda preparada para funcionar insertándose en la Raspberry Pi 3, enchufando la misma al teclado, ratón y monitor y por último a la corriente eléctrica.

Al ser enchufada a la corriente la Raspberry Pi se inicia de forma automática y carga la herramienta NOOBS, tras lo que aparece la pantalla de selección de sistemas operativos en el que debemos seleccionar Raspbian (Figura A.1) y pulsar el botón *Instalar* en la esquina superior izquierda del menú, tras lo cual comienza el proceso de instalación de Raspbian utilizando la totalidad de la tarjeta microSD para su sistema de archivos (si utilizamos la versión ligera de NOOBS primero se efectúa la descarga del mismo, ya que no vendría incluido en la herramienta). Al terminar, la herramienta avisa y se procede al reinicio del dispositivo.

Tras la instalación de Raspbian la herramienta NOOBS sigue estando disponible como partición de recuperación para la Raspberry Pi 3, accesible pulsando la tecla *shift* en el momento de encendido al aparecer por primera vez el logotipo de Raspberry Pi.

Aunque su implantación queda fuera del ámbito de este TFG, la Raspberry Pi cuenta con una serie de interfaces activables desde el entorno gráfico o la orden de terminal *raspi-config* que permiten su gestión remota, ya sea mediante *Secure Shell* (SSH) o servidores *Virtual Network Computing* (VNC) [30], tras lo que la Raspberry Pi podría



Figura A.1: Selección de Raspbian para ser instalado en la Raspberry Pi 3.

ser manejada en modo headless, sin estar conectada a teclado, ratón o salida de vídeo, tan solo a la corriente y a una red local por medio de alguna de sus interfaces de red.

No entramos en la configuración del hardware de la WiFi de la Raspberry Pi 3 porque ya viene instalada por defecto en Raspbian.

En este apartado no entramos en el detalle de la configuración de *hostapd* porque en el capítulo 5 presentamos toda la instalación del software necesario por orden de dependencias entre ellos, lo cual nos obligaría a explicar previamente aplicaciones que pertenecen a otros módulos.

Apéndice B Descarga automática del código de la Nube

Hoy en día la tendencia es a alojar los proyectos de libre distribución y código abierto en la Nube. Nosotros hemos alojado nuestro proyecto en la nube *GitHub*. En este apéndice proporcionamos la dirección Web de acceso a él y la instalación automática de todo el entorno.

Para implementar el servicio, en lugar de replicar el código manualmente, puede clonarse el repositorio online habilitado explícitamente para ello mediante la herramienta de control de versiones *git*, situándose en el directorio donde se quiera almacenar los archivos y ejecutando la siguiente orden en el terminal:

```
git clone https://github.com/DavSanchez/CaptivePortal.git
```

Tras esto, el servicio Web completo puede ponerse en marcha desde este mismo directorio mediante el orden node.

```
node nodeserver.js
```

Apéndice C Reanudación automática del Servidor en caso de fallo

Proporcionando la orden de arranque del Apéndice B (`node nodeserver.js`) es suficiente para que el sistema de sensorio móvil colaborativo arranque y proporcione los servicios desarrollados. sin embargo, se puede dar el caso de fallos del servidor en cuyo caso debemos tolerarlos. En este apéndice se explica como hacerlo.

La orden ejecutada justo al final del apartado anterior basta para poner en marcha el servicio, pero no protege ante la situación de que un fallo detenga su ejecución y requiere que un administrador lo ejecute siempre que el hardware se pone en marcha. Existen diversas utilidades para solucionar esta situación no deseada; la utilizada en este TFG es PM2.

PM2 es un gestor de procesos para Node.js que nos permite configurar el comportamiento del servicio Web implementado frente a cambios en los archivos locales (como una actualización) u otras situaciones. Puede instalarse a través del gestor de paquetes *npm*:

```
npm install pm2 -g
```

Tras instalar esta herramienta ejecutaríamos una serie de órdenes con ella que nos permiten reanudar la aplicación Node.js de forma automática nada más encender nuestra Raspberry Pi 3, reiniciando el servicio si ocurre algún error o incluso si actualizamos el código de forma local o remota (editándolo en otra parte, subiéndolo a GitHub y recuperándolo con la orden *git pull*):

```
pm2 startup
```

Esta orden detecta el sistema en el que nos encontramos y nos recomienda el orden a ejecutar para generar un *startup script*, que es ejecutado nada más encender el dispositivo. Una salida de ejemplo de este orden es la siguiente:

```
[PM2] You have to run this command as root. Execute the following command:  
sudo su -c "env PATH=$PATH:/home/unitech/.nvm/versions/node/v4.3/bin pm2  
startup <distribution> -u <user> --hp <home-path>"
```

Código C.1: PM2 sugiere un comando a introducir para poder ejecutarse al iniciar el sistema.

Tras este proceso, solo queda configurar los parámetros a usar por PM2, ejecutar las aplicaciones Node.js deseadas y guardar el entorno con la siguiente orden:

```
pm2 save
```

Este paso asegura que la aplicación Node.js vuelve a ejecutarse tras un reinicio del sistema, ya sea un reinicio programado o el ocurrido tras un corte accidental.

Para conseguir lo anterior es necesario configurar el entorno de programación. Para ello se puede utilizar un archivo de configuración como el empleado en este TFG, ubicado

en la raíz del repositorio de GitHub con el nombre `captiveportalserver.config.js`. El contenido de este archivo es el siguiente:

```
module.exports = {
  apps : [
    {
      name      : "nodeserver",
      script    : "./nodeserver.js",
      watch     : true,
      ignore_watch: [".git", ".node_modules", ".gitignore", ".uploads",
      "./users/users.json", "./users/usersOneTime.json"]
    }
  ]
}
```

Código C.2: Fichero de configuración del entorno node con PM2

Se observa que se trata esencialmente de un objeto JavaScript con los siguientes campos:

- *name*: el nombre asignado al proceso.
- *script*: la ubicación del archivo Node.js que ejecutaríamos normalmente con la orden *node*.
- *watch*: un flag que vigila cambios en nuestros archivos, reiniciando la aplicación Node.js si se detectara uno.
- *ignore_watch*: *Array* de archivos y directorios que no se vigilan con el *flag* anterior.

Si nuestro entorno requiriese de más aplicaciones Node.js ejecutadas simultáneamente podrían añadirse como nuevas instancias del objeto *apps*.

Una vez terminada la configuración de nuestro entorno, lo ejecutamos mediante PM2 (no mediante node) y guardamos el entorno tal y como se nos queda para que se ejecute de la siguiente forma cada vez que se inicie nuestro sistema.

```
pm2 start captiveportalserver.config.js
```

```
pm2 save
```

Por último, con la orden `pm2 monit` puede accederse a una vista de monitorización que nos permite ver una lista de las aplicaciones activas y su consumo en memoria, la salida en consola de cada una de ellas, el número de veces que se han reiniciado (Figura C.1).

```

Process list
0] nodeserver Mem: 31 MB
Global Logs
nodeserver > Petición de estado del servidor recibida
nodeserver > Buscando usuarios inactivos...
nodeserver > El servidor parece estar bien.
nodeserver > Petición de estado del servidor recibida
nodeserver > Buscando usuarios inactivos...
nodeserver > El servidor parece estar bien.
nodeserver > Petición para subir audio recibida
nodeserver > Audio subido con éxito.
nodeserver > Estableciendo credenciales...
nodeserver > Estableciendo usuario 0 como ocupado.
nodeserver > Guardando estado de usuarios en fichero JSON.
nodeserver > Almacenando credenciales del usuario 0 para el cliente.
nodeserver > Petición de credenciales recibida. Envio...
Custom metrics (http://bit.ly/
loop delay 0.73ms
Metadata
App Name nodeserver
Restarts 1
Uptime 15m
Script path /home/pi/CaptivePortalWebApp/nodeserver.js
Script args N/A
Interpreter node
Interpreter args N/A
soft/right: switch boards | up/down/mouse: scroll | Ctrl-C: exit To go further check out https://keymetrics.io.

```

Figura C.1: Vista de monitorización de aplicaciones activas con pm2 monit.

Apéndice D Ciclo de Vida del TFG

En este apéndice se presenta una visión rápida del ciclo de vida del TFG. El objetivo es mostrar cómo ha ido evolucionando la implantación del sistema de sensado.

El desarrollo de este TFG no consistió en considerar un único curso de acción y desarrollar la solución. Durante las etapas iniciales se tuvieron en cuenta e investigaron varias alternativas con el objetivo de llegar al mismo resultado final, elementos que se vieron reflejados en los mecanismos de seguimiento presentados. A continuación detallamos algunos de estas primeras aproximaciones y se explica el motivo de haberlas declinado en favor del camino finalmente escogido.

La primera solución propuesta para el TFG, apareciendo así en la primera versión de la propuesta o anteproyecto, consistía en programar una aplicación nativa para terminales Android que gestionara la captura del audio y la ubicación, habilitando el acceso a Internet una vez transferidos los archivos al servidor. Esta solución podría extenderse más adelante a sistemas iOS. Se pensó también implementar una aplicación universal mediante un *wrapper* como *Apache Cordova*, la tecnología anteriormente conocida como *PhoneGap*, con lo que con una sola programación se podrían obtener aplicaciones para diversos sistemas móviles.

Sin embargo, la propuesta de aplicación nativa resultaba altamente situacional, implicando que un hipotético usuario del servicio tuviera una aplicación instalada en su dispositivo de antemano si quería poder acceder a la red. Además, el desarrollo de aplicaciones nativas orientadas a terminales móviles excluía a los ordenadores portátiles, que sin ser *smartphones* ni *tablets* debían poder utilizar el servicio sin problema. Este enfoque requeriría entonces programar aplicaciones de escritorio separadas o implementar alguna otra solución específica para ellos, lo que alargaría el proceso de programación.

Finalmente, con el objetivo de facilitar lo más posible la implementación y el acceso al servicio, junto a la oportunidad de utilizar tecnologías de aparición reciente como las API de *WebRTC*, se optó por desarrollar una aplicación web, accesible desde cualquier dispositivo que cuente con un navegador compatible.

Para implementar el servidor se pensó desde el primer momento en utilizar la Raspberry Pi. Pese a que se disponía de unas cuantas en su modelo 2 y se intentó implementar en ellas, estas carecen de *chipset WiFi* propio, por lo que se debía adquirir por separado, conectarse a esta por USB y compilar el controlador del *hardware* utilizado. Además, no todos los dispositivos WiFi externos disponibles soportan trabajar en modo punto de acceso indispensable para nuestro proyecto, por lo que había que tener especial cuidado al adquirir uno de ellos. Con el objetivo de ahorrarnos toda esta problemática se decidió trabajar directamente con la Raspberry Pi 3, que ya incluye *chipset WiFi* 802.11n propio capaz de trabajar en el modo deseado, teniendo además el mismo precio que su modelo predecesor. Al igual que se tuvieron en cuenta varias alternativas para la implementación del lado del cliente, se experimentó con diversas posibilidades para el lado del servidor en la Raspberry Pi hasta optar por la escogida finalmente.

Durante el proceso de documentación, previa a la implementación del TFG se pensó en implantar el control de acceso a la red manualmente. Para ello se contemplaron diversas opciones para servidores DHCP (como *dnsmasq* o *isc-dhcp-server*) que proporcionarían direccionamiento IP a los dispositivos conectados. Junto a estos actuaría la utilidad de Linux *iptables*, que permite establecer reglas para el procesamiento de paquetes IP, pudiendo aplicarse permanentemente a través del paquete *iptables-persistent* o mediante *scripts* que se ejecutarían y aplicarían las reglas al iniciar el sistema operativo. Sin embargo, este último paquete presentó problemas al ser instalado en la Raspberry Pi, por lo que esta opción fue descartada al poco tiempo de descubrir la solución que proporcionaba CoovaChilli.

En cuanto a los servidores Web para alojar el portal cautivo, inicialmente se pensó en utilizar un enfoque tradicional con servidores como *Apache*, *Nginx* o soluciones ligeras como *lighttpd* junto a PHP o *web scripts CGI*. Finalmente fueron parcialmente descartados como implementación principal en favor de tecnologías web de gran popularidad actual como Node.js, usando el servidor web Nginx solamente para alojar la interfaz web daloRADIUS.

Aunque Raspbian es la oficial, la Raspberry Pi permite instalar otras distribuciones y sistemas operativos. Uno de ellos es *OpenWRT*, una solución de código abierto para sistemas empotrados orientados a encaminamiento de tráfico de red. Puede instalarse en pasarelas residenciales, ordenadores portátiles e incluso teléfonos móviles, por lo que su instalación es viable en la Raspberry Pi. OpenWRT tiene disponible una gran cantidad de protocolos como DNS y DHCP, NAT, mapeo de puertos, aplicaciones de seguridad... también es posible el uso de portales cautivos instalando *CoovaChilli*.

Finalmente se decidió no optar por esta solución debido a que la distribución basada en *Debian* predeterminada para la Raspberry Pi ofrece un soporte más amplio en cuanto a documentación y mayores posibilidades para implementar otras tareas relacionadas con el proyecto: gestión, análisis y procesado de los ficheros de audio, simulaciones de ubicación acústica...

Junto a CoovaChilli se evaluó otra solución de control de acceso a la red disponible para Linux: *PacketFence*. Este programa de código abierto es también muy completo, ofreciendo facilidades de monitorización, análisis de ficheros transmitidos, gestión centralizada tanto cableada como inalámbrica (802.1X) y utilidades de portal cautivo. Lamentablemente, cuando se estudió su uso *PacketFence* no garantizaba compatibilidad total para la Raspberry Pi, por lo que también fue descartado.

Apéndice E Problemas Acontecidos durante el Desarrollo de la Solución

En este apéndice se presentan algunas dificultades ocurridas durante el desarrollo del TFG que limitaron el diseño del sistema final y alargaron en el tiempo desarrollos que a priori no tenían dificultad.

Uno de los problemas a los que nos enfrentamos a la hora de implementar este TFG es el de los navegadores compatibles con WebRTC. La API *MediaStream Recording* utilizada aún no está soportada por todos los navegadores, siendo Safari, navegador de los dispositivos Apple, el ejemplo más relevante de esta falta de soporte. Este problema es mayor de lo que parece debido a que Apple obliga a que todos los navegadores disponibles en la App Store de iOS hagan uso del motor de renderizado web WebKit que también usa Safari, por lo que en un dispositivo iOS no podemos simplemente cambiar de navegador para que nuestro sistema funcione.

Aunque no presenta problema a la hora de ser probado en local, las API de geolocalización y de grabación multimedia requieren de un contexto seguro. Esto implica que el sistema no funciona sin hacer uso de HTTPS, por lo que hubo que implementar este sistema en nuestra aplicación web haciendo uso de la herramienta OpenSSL, la API correspondiente para HTTPS en Node.js y habilitar esta opción en CoovaChilli.

Dado que OpenSSL genera certificados auto-firmados, y por tanto no reconocidos por una autoridad de certificación, los navegadores no los aceptan de primera para proteger al usuario, por lo que a efectos de hacer las pruebas hubo que añadir las excepciones de la aplicación web y la interfaz JSON de *CoovaChilli* manualmente.

Apéndice F Tabla de Desarrollo Temporal

Se presenta una tabla de desarrollo temporal.

La cronología de desarrollo aproximado del proyecto se muestra en la Tabla F.1.

Tarea acometida	Tiempo invertido
Documentación inicial	10 horas
Especificación	10 horas
Opciones consideradas	10 horas
Ánálisis software de servidor	20 horas
Diseño final	20 horas
Implementación final (WebApp + Server)	160 horas
Pruebas en Raspberry Pi 2 + Interfaz WiFi	5 horas
Configuración de Seguridad HTTPS	5 horas
Pruebas en Raspberry Pi 3	10 horas
Correcciones y cambios	20 horas
Pruebas finales	20 horas
Elaboración de documentos finales	10 horas

Tabla F.1: Cronología aproximada de realización del TFG

Referencias

- [1] T. O'Reilly y C. Doctorow, *Opportunities and Challenges in the IoT: A Conversation with Tim O'Reilly and Cory Doctorow*. O'Reilly Media, Inc, 2015.
- [2] R. Barnes, *Official Raspberry Pi Projects Book*. Cbl Distribution Limited, 2015.
- [3] M. MacDonald, *HTML5; The Missing Manual*. O'Reilly Media, Inc, 2014.
- [4] A. Goldstein, L. Lazaris y E. Weyl, *HTML5 & CSS3 for the Real World*. SitePoint Pty. Ltd, 2015.
- [5] C. Aquino y T. Gandee, *Front-End Web Development: The Big Nerd Ranch Guide*. Pearson Technology Group, 2016.
- [6] A. Mardan, *Practical Node.js*, 1st. Heinz Weinheimer, 2014.
- [7] D. Ristic, *Learning WebRTC*, 1st. Packt Publishing Ltd., 2015.
- [8] R. Nixon, *Learning PHP, MySQL & JavaScript*. O'Reilly Media, Inc, 2015.
- [9] E. Brown, *Web Development with Node and Express*, 1st. O'Reilly Media, Inc, 2014.
- [11] E. M. Macías, Á. Suárez y J. Lloret, «Mobile Sensing Systems», *Sensors*, vol. 13, n.º 12, págs. 17 292-17 321, 2013.
- [14] J. Vollbretch, *The Beginnings and History of RADIUS*. Interlink Networks, 2006.
- [19] L. Pekowsky, *JavaServer Pages*. Addison-Wesley Professional, 2003.
- [20] D. Mertz, *Text Processing in Python*. Addison-Wesley Professional, 2003.
- [22] M. Satheesh, B. J. D'mello y J. Krol, *Web Development with MongoDB and NodeJS*. Packt Publishing Ltd., 2015.
- [31] M. Barlow, *Governing the IoT*. O'Reilly Media, Inc, 2016.
- [32] ——, *Smart Cities, Smarter Citizens*. O'Reilly Media, Inc, 2015.
- [33] M. Treseler, *Designing for the Internet of Things*. O'Reilly Media, Inc, 2016.
- [34] S. Chacon y B. Straub, *Pro Git*. Apress, 2014.
- [35] D. Stenberg, *Everything curl*, 2015.

Referencias Web

- [10] E. Marcotte. (2010). Responsive Web Design, dirección: <https://alistapart.com/article/responsive-web-design> (visitado 2017).
- [12] J. Malinen. (2013). hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator, dirección: <http://w1.fi/hostapd/> (visitado 05-2017).
- [13] CoovaChilli Community. (2017). CoovaChilli GitHub Repository, dirección: <https://github.com/coova/coova-chilli> (visitado 2017).

- [15] The FreeRADIUS Server Project y Contributors. (2017). FreeRADIUS Documentation, dirección: <http://freeradius.org/documentation/> (visitado 2017).
- [16] L. Tal. (2007). daloRADIUS, dirección: <http://www.daloradius.com/> (visitado 2017).
- [17] Sheppy, lazarljubenovic, SOCSIELEARNING, Ambar y dimaspirit. (2017). MediaStream Recording API, dirección: https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API (visitado 2017).
- [18] N. Angelacos. (2011). Haserl on Sourceforge, dirección: <http://haserl.sourceforge.net/> (visitado 03-2017).
- [21] Google Developers. (2017). Chrome V8, dirección: <https://developers.google.com/v8/> (visitado 2017).
- [23] kangax, webbedspace y zloirock. (2017). ECMAScript 2016+ compatibility table, dirección: <https://kangax.github.io/compat-table/es2016plus/> (visitado 2017).
- [24] Y. Fablet. (2017). Announcing WebRTC and Media Capture, dirección: <https://webkit.org/blog/7726/announcing-webrtc-and-media-capture/> (visitado 07-06-2017).
- [25] fisharebest. (2013). WifiDocs/CoovaChilli, dirección: <https://help.ubuntu.com/community/WifiDocs/CoovaChilli> (visitado 05-2017).
- [26] I. Goncalves, S. Pfeiffer, C. Montgomery y Xiph. (2008). RFC 5334: Ogg Media Types, dirección: <https://xiph.org/ogg/doc/rfc5334.txt> (visitado 2017).
- [27] S. Pfeiffer y CSIRO. (2003). RFC 3533: The Ogg Encapsulation Format Version 0, dirección: <https://xiph.org/ogg/doc/rfc3533.txt> (visitado 2017).
- [28] Google Developers. (2017). Chrome DevTools, dirección: <https://developers.google.com/web/tools/chrome-devtools/> (visitado 2017).
- [29] IETF codec working group. (2011). Opus Interactive Audio Codec, dirección: <http://opus-codec.org/> (visitado 09-2017).
- [30] Raspberry Pi Foundation. (2017). VNC (Virtual Network Computing), dirección: <https://www.raspberrypi.org/documentation/remote-access/vnc/> (visitado 03-2017).
- [36] Beavies. (2013). Creación de un portal cautivo desde cero, dirección: <http://beavies.blogspot.com.es/2013/03/creacion-de-un-portal-cautivo-desde-cero.html> (visitado 2017).
- [37] A. Reid. (2014). Modifying coova chilli to allow anonymous users for a private hotspot using automatic MAC addresses, dirección: <http://subgroup-ash.blogspot.com.es/2014/02/modifying-coova-chilli-to-allow.html> (visitado 2017).

- [38] J. Krause. (2013). Raspberry PI based FreeRadius Server with GUI, dirección: <http://www.binaryheartbeat.net/2013/12/raspberry-pi-based-freeradius-server.html> (visitado 2017).
- [39] adafruit. (2013). Setting up a Raspberry Pi as a WiFi access point, dirección: <https://learn.adafruit.com/setting-up-a-raspberry-pi-as-a-wifi-access-point/install-software> (visitado 03-2017).
- [40] Raspberry Pi Foundation. (2017). Setting up an NGINX web server on a Raspberry Pi, dirección: <https://www.raspberrypi.org/documentation/remote-access/web-server/nginx.md> (visitado 05-2017).
- [41] ——, (2017). Raspbian Documentation, dirección: <https://www.raspberrypi.org/documentation/raspbian/> (visitado 2017).
- [42] N. Rexford. (2017). [CoovaChilli] Understanding how Login works, dirección: <https://www.brightonchilli.org.uk/pipermail/coovachilli/2017-January/000181.html> (visitado 2017).
- [43] CoovaChilli Community. (2017). The CoovaChilli Archives (mailing list), dirección: <https://www.brightonchilli.org.uk/pipermail/coovachilli/> (visitado 2017).
- [44] ——, (2017). CoovaChilli - man pages, dirección: <http://coova.github.io/CoovaChilli/man-pages.html> (visitado 2017).
- [45] ——, (2017). CoovaChilli JSON Interface, dirección: <http://coova.github.io/CoovaChilli/JSON/> (visitado 2017).
- [46] C. Rigney, S. Willens, Livingston, A. Rubens, Merit, W. Simpson y Daydreamer. (2000). RFC 2865 (en FreeRADIUS), dirección: <http://freeradius.org/rfc/rfc2865.html> (visitado 04-2017).
- [47] D. Woitasen. (2012). Respuesta a “How to create a self-signed certificate with openssl?”, dirección: <https://stackoverflow.com/questions/10175812/how-to-create-a-self-signed-certificate-with-openssl> (visitado 06-2017).
- [48] waldner. (2010). TUN/TAP interface tutorial, dirección: <http://backreference.org/2010/03/26/tuntap-interface-tutorial/> (visitado 06-2017).
- [49] Keymetrics. (2017). PM2 Documentation, dirección: <http://pm2.keymetrics.io/docs/usage/cluster-mode/> (visitado 07-2017).
- [50] Node.js Foundation. (2017). Express API Reference, dirección: <http://expressjs.com/en/4x/api.html> (visitado 2017).
- [51] D. Wilson y kornel. (2017). npm packages documentation, dirección: <https://www.npmjs.com/> (visitado 2017).
- [52] Bootstrap. (2017). Bootstrap Documentation, dirección: <https://getbootstrap.com/docs/4.0/getting-started/introduction/> (visitado 2017).
- [53] The jQuery Foundation. (2017). jQuery API, dirección: <https://api.jquery.com/> (visitado 2017).

- [54] Node.js Foundation. (2017). Node.js v8 Documentation, dirección: <https://nodejs.org/dist/latest-v8.x/docs/api/> (visitado 2017).
- [55] ——, (2017). Anatomy of an HTTP Transaction, dirección: <https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/> (visitado 2017).
- [56] w3schools.com. (2017). w3schools.com, dirección: <https://www.w3schools.com/> (visitado 2017).
- [57] Mozilla Developer Network. (2017). Mozilla Developer Network web docs, dirección: <https://developer.mozilla.org/en-US/> (visitado 2017).
- [58] Statista, the Statistics Portal. (2017). Mobile internet traffic as percentage of total web traffic in August 2017, by region, dirección: <https://www.statista.com/statistics/306528/share-of-mobile-internet-traffic-in-global-regions/> (visitado 10-2017).

Glosario

AAA Authentication, Authorization and Accounting.

AJAX Asynchronous JavaScript And XML.

AP Access Point.

API Application Programming Interface.

ARM Advanced RISC Machine.

BSD Berkeley Software Distribution.

CCK Complementary Code Keying.

CGI Common Gateway Interface.

CHAP Challenge-Handshake Authentication Protocol.

CSS Cascading Style Sheets.

CSV Comma-Separated Values.

DCHP Dynamic Host Control Protocol.

DOM Document Object Model.

DSSS Direct-Sequence Spread Spectrum.

EAP Extensible Authentication Protocol.

EAP-TTLS EAP Tunneled Transport Layer.

GPIO General Purpose Input-Output.

GPL GNU General Public License.

HDMI High-Definition Multimedia Interface.

Hostapd Host Access Point Daemon.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

HTTPS HTTP Secure.

IEEE Institute of Electrical and Electronics Engineers.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

JS JavaScript.

JSON JavaScript Object Notation.

LED Light-Emitting Diode.

LPDDR2 Low Power Double Data Rate 2.

LXDE Lightweight X11 Desktop Environment.

MAC Medium Access Control.

MEAN MongoDB, Express, Angular and Node.js.

MicroSDHC Micro Secure Digital High-Capacity.

MIME Multipurpose Internet Mail Extensions.

MIMO Multiple-Input Multiple-Output.

NAS Network Access Server.

NAT Network Address Translation.

NOOBS New Out Of Box Software.

npm Node Package Manager.

PAM Pluggable Authentication Module.

PAP Password Authentication Protocol.

PEAP Protected EAP.

PHP Pre Hypertext Processor.

PIXEL Pi Improved Xwindows Environment Lightweight.

QoS Quality of Service.

RADIUS Remote Authentication Dial-In User.

RAM Random Access Memory.

RFC Request For Comments.

RSSI Received Signal Strength Indicator.

RTP RealTime Transport Protocol.

SoC System on Chip.

SQL Structured Query Language.

SSH Secure SHell.

SSID Service Set IDentifier.

SSL Secure Sockets Layer.

TAP Terminal Access Point.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TUN Network TUNnel.

UAM Universal Access Method.

UDP User Datagram Protocol.

URL Uniform Resource Locator.

USB Universal Serial Bus.

VNC Virtual Network Computing.

W3C World Wide Web Consortium.

WebRTC Web RealTime Communications.

WEP Wired Equivalent Privacy.

WiFi Wireless Fidelity.

WPA WiFi Protected Access.

WPA2 WiFi Protected Access 2.

XML eXtensible Markup Language.

8. Presupuesto y pliego de condiciones

En este apartado, se hace balance de los costes totales de este proyecto fin de carrera y las condiciones específicas actuales de su implementación. En el presupuesto se distinguen 3 partidas fundamentales de gasto: recursos materiales, costes de ingeniería y costes en bienes fungibles.

8.1. Hola