Multistreaming de video y audio en una red ad hoc de dispositivos móviles corrientes

Multistreaming of video and audio in an ad-hoc network of common mobile devices

Daniel Alfaro Miranda (GII)
Luis Pozas Palomo (GII)

UNIVERSIDAD COMPLUTENSE
MADRID

TRABAJO DE FIN DE GRADO
Grado en Ingeniería Informática
Facultad de Informática

Dirigido por:
Simon Pickin

Curso 2020/21

Agradecimientos

Queremos agradecer toda la comprensión y el apoyo brindado por nuestros familiares y amigos, sin los cuales no habríamos llegado a estar donde estamos.

También gracias a la Fundación Shuttleworth [1], por la concesión de un "Flash Grant"que ha permitido la compra de los teléfonos móviles para hacer este trabajo, así como a Andrew Lamb, *fellow* de la Fundación Shuttleworth que recomendó la concesión de dicha financiación.

Gracias a nuestro tutor Simon Pickin, por su paciencia y colaboración en este trabajo y su gran desempeño como profesor durante el grado.

Por último, queremos destacar la aportación de los creadores de la biblioteca libstreaming [2], el cual nos ahorro mucho tiempo y sirvió de gran ayuda para este y muchos otros proyectos.

Índice general

Agradecimientos II

Palabras Clave VII

Keywords VIII

Resumen IX

Abstract X

1. Introducción 1
- 1.1. Motivación 1
- 1.2. Objetivos 2
- 1.3. Plan de trabajo 4

2. Introduction 5
- 2.1. Motivation 5
- 2.2. Goals 6
- 2.3. Work plan 8

3. Antecedentes 9
- 3.1. Comunicación DTD 9
  - 3.1.1. Bluetooth 10
  - 3.1.2. Wifi 10
  - 3.1.3. Wifi Ad-hoc mode 11
  - 3.1.4. Wifi Direct 12
  - 3.1.5. Miracast 13
  - 3.1.6. Wifi Aware 13
  - 3.1.7. LTE Direct 14
  - 3.1.8. NFC 15
- 3.2. Protocolos de interés 15
  - 3.2.1. SCTP 15
  - 3.2.2. RTP - RTCP 16
  - 3.2.3. RTSP 16
  - 3.2.4. SDP 17
  - 3.2.5. HLS (HTTP Live Streaming) 17
  - 3.2.6. PPSPP (Peer-to-Peer Streaming Peer Protocol) 18
  - 3.2.7. RIST 18
- 3.3. Software de interés (énfasis en software libre) 18
  - 3.3.1. WebRTC 18
  - 3.3.2. VLC 19

3.3.3. FFmpeg 19
3.3.4. Gstreamer 19
3.3.5. Libstreaming 20

4. Elección de tecnologías 21
4.1. Android 21
4.2. WiFi Aware 21
4.3. Java 21
4.4. Android Studio 21
4.5. RTP sobre UDP 22
4.6. RTSP y SDP 22
4.7. Libstreaming 23
4.8. LibVLC 23
4.9. Balsamiq Wireframes 23
4.10. GitHub 23
4.11. LaTeX y Overleaf 24

5. Especificación 25
5.1. Requisitos 26
5.2. Prototipo 27

6. Diseño 34
6.1. Interfaz gráfica 34
6.2. Gestión de las conexiones RTSP, RTP y RTCP 35
6.3. Funcionamiento de Wifi Aware 38
6.3.1. Creación de conexiones con Wifi Aware 39
6.3.2. Gestión de Wifi Aware 39
6.4. Diseño orientado al Multihopping 40
6.4.1. Identificación de los streams en la red distribuida 40
6.4.2. Gestión centralizada de los streams en el dispositivo 41
6.5. Aplicación del botón de pánico 41

7. Implementación 44
7.1. RTSP, RTP y libstreaming 44
7.1.1. Análisis de libstreaming y orientación de la implementación . 44
7.1.1.1. Estado de libstreaming 44
7.1.1.2. Cambios en libstreaming por los alumnos de 2018/19 45
7.1.1.3. Conclusiones 46
7.1.2. Adaptación de libstreaming a IPv6 47
7.1.3. Agrupar el acceso a los flujos multimedia 48
7.1.4. Transmisión múltiple del flujo multimedia 49
7.1.4.1. Implementación de modalidades RTSP 50
7.1.4.2. Adaptación del servidor para la transmisión múltiple 51
7.1.4.3. Adaptación del cliente para la transmisión múltiple . 52
7.2. Wifi Aware 52

8. Conclusiones y trabajo futuro **55**
8.1. Recapitulación y evaluación 55
8.2. Trabajo futuro 57
8.3. Observaciones finales 59

9. Conclusions and future work **60**
9.1. Summary and evaluation 60
9.2. Future work 62
9.3. Concluding remarks 63

10. Aportación de los participantes **64**
10.1. Daniel Alfaro Miranda 64
10.1.1. Investigación 64
10.1.2. Desarrollo 64
10.1.3. Memoria 65
10.2. Luis Pozas Palomo 65
10.2.1. Investigación 65
10.2.2. Desarrollo 66
10.2.3. Memoria 66

A. Detalles del diseño **68**
A.1. Investigación y análisis inicial 68
A.1.1. Proyecto 2018/19 68
A.1.2. Proyecto 2020/21 69
A.1.3. Conclusiones 70
A.2. Elección de modalidad RTSP 71
A.3. Creación de conexiones con Wifi Aware 72

B. Detalles de implementación **74**
B.1. Estado de libstreaming y los cambios en el curso 2018/19 74
B.1.1. Estado de libstreaming 74
B.1.2. Cambios en libstreaming por los alumnos de 2018/19 74
B.2. Envío del flujo multimedia a mas de un destino 76
B.3. Adaptación de servidor y cliente RTSP a Wifi Aware 78
B.4. Gestión de Wifi Aware 79

Índice de figuras

1.1. Plan de trabajo. 4

5.1. Prototipo de la pantalla de selección del modo de uso. 29
5.2. Prototipo de la pantalla de inicio. 30
5.3. Prototipo de la pantalla de navegación. 31
5.4. Prototipo de la pantalla de la cámara. 32
5.5. Prototipo de la pantalla de galería. 33

6.1. Diseño comunicación RTSP, RTP y RTCP 37
6.2. Proceso de conexión de dispositivos con Wifi Aware. [7] 39
6.3. Diagrama multihopping. 40
6.4. Diagrama para la gestión de los streams. 42
6.5. Botón de pánico. 43

10.1. Diagrama de Gantt. 67

Palabras Clave

- Wifi Aware
- D2D
- Android
- RTSP
- Libstreaming
- Redes Ad hoc móviles (MANETs)
- streaming de vídeo
- difusión múltiple de vídeo

Keywords

- Wifi Aware
- D2D
- Android
- RTSP
- Libstreaming
- Mobile Ad hoc Networks (MANETs)
- video streaming
- multicast streaming

Resumen

El propósito principal de este proyecto ha sido especificar y desarrollar una aplicación de *streaming* de vídeo y audio a través de una red ad hoc formada entre dispositivos móviles. La idea es que los móviles cercanos se conecten directamente unos con otros, es decir, sin que ningún dispositivo se conecte a un punto de acceso Wifi ni a una estación base de una red de telefonía móvil, formando una red descentralizada y sin limite de dispositivos. La finalidad de formar esta red es transmitir vídeo y audio en casos en los que no se pueda conectar a la red convencional, ya sea por causas de emergencia o de restricción de acceso.

Este proyecto constituye la continuación de dos anteriores, "Device to Device streaming en dispositivos móviles" [3] y "Crowdstreaming: transmisión de vídeo y audio por una red ad hoc de teléfonos móviles" [4]. Los proyectos anteriores fueron aproximaciones a la implementación final de este proyecto y nos sirvieron de útil referencia. El primero se baso en Wifi Direct, pero encontraron ciertas limitaciones en la tecnología que no les permitió completar los objetivos principales de su proyecto. El segundo optó por la nueva tecnología de comunicación D2D, Wifi Aware, la cual esta en desarrollo y tiene muy poca documentación y ejemplos. Por ciertos problemas, que no les permitió investigar hasta pasado varios meses, tuvieron que crear una aplicación con funcionalidad reducida.

Nosotros para este proyecto, por medio de Wifi Aware, hemos creado una aplicación de *streaming* que cumple nuestro propósito principal en cuanto a la creación de la red descentralizada y al envío múltiple y a través de múltiples dispositivos (*multihopping*) del flujo multimedia, además de ciertos objetivos relacionados con los casos de uso de la aplicación que se verán mas adelante.

El código fuente de nuestro proyecto esta alojado en el siguiente repositorio publico de GitHub: https://github.com/luispozas/TFG

Abstract

The main purpose of this project has been to specify and develop a video and audio streaming application over an ad hoc network formed between mobile devices. The idea is that nearby cell phones connect directly to each other, i.e. without any device connecting to a Wifi access point or a base station of a cell phone network, forming a decentralized network with no limit of devices. The purpose of forming this network is to transmit video and audio in cases where it is not possible to connect to the conventional network, either for emergency or access restriction reasons.

This project is a continuation of two previous projects, "Device to Device streaming on mobile devices" [3] and "Crowdstreaming: video and audio transmission in a MANET (mobile ad hoc network)" [4]. The previous projects were approximations to the final implementation of this project and served as useful references. The first one was based on Wifi Direct, but they found certain limitations in the technology that did not allow them to complete the main goals of their project. The second opted for the new D2D communication technology, Wifi Aware, which is under development and has very little documentation and examples. Due to certain problems, which did not allow them to investigate for several months, they had to create an application with reduced functionality.

For this project, through Wifi Aware, we have created a streaming application that fulfills our main purpose in terms of decentralized networking and multi-device multihopping of the media stream, as well as certain objectives related to the use cases of the application that will be discussed later.

The source code of our project is hosted in the following GitHub public repository: https://github.com/luispozas/TFG

1. Introducción

1.1. Motivación

En MANETs, los usuarios cercanos se comunican directamente configurando sus interfaces en modo ad-hoc, con el propósito de transferencia de datos, ya sea directamente o a través de un nodo de tránsito para redirigir el tráfico a otros nodos de la red. En muchas situaciones como catástrofes o emergencias, donde no se dispone de las infraestructuras de red que nos proporcionan los operadores de red, esta clase de red es una solución ideal porque nos ofrece entornos descentralizados.

Tal como se observa en la Sección 1.1 de [3], "se podría formar una red ad-hoc para pasar vídeos y/o mensajes de ayuda de un dispositivo a otro hasta encontrar uno que disponga de acceso a Internet, y de esta forma los datos enviados desde una zona sin conexión se publicarían en Internet y llegarían a su destino.

Otro de los posibles casos de uso para una aplicación como la que proponemos podría ser el dar la posibilidad de transmitir vídeos a dispositivos lejanos en países en los que se cometen crímenes contra los derechos humanos, de tal forma que el vídeo se distribuya de un dispositivo a otro saltándose el control y censura que países como estos establecen en las redes, dando la posibilidad de ocultar el origen de la transmisión para que no se pueda identificar al denunciante. Por este motivo, la aplicación no deberá dejar trazas de los vídeos, ni en los móviles emisores ni en los que lo redistribuyen. Si la red es capaz de cruzar la zona de censura los vídeos podrían ser publicados en Internet con ayuda de organizaciones como Witness [5] que se dedican a promover la creación y custodia de vídeos de este tipo.".

En concreto, para la custodia de vídeos existe el sistema SecureDrop [6], diseñado para que periodistas suban sus documentos o vídeos a un servidor en la red Tor de forma anónima.

Las redes móviles ad hoc se están convirtiendo en una parte importante de la próxima generación de redes, debido a su flexibilidad, capacidad de autoconfiguración, ausencia de infraestructura, facilidad de mantenimiento, capacidad de autogestión y reducido coste. En este contexto, en las dos últimas décadas ha habido una gran cantidad de investigación en la comunidad académica con respecto a las redes móviles. Una de las áreas de investigación en este ámbito son la transmisión de datos sobre MANETs, esto es debido al aumento en la capacidad computacional de los *smartphones* y su gran popularidad en la sociedad actual.

Es importante mencionar el desafío de proporcionar una cierta calidad en el medio inalámbrico compartido y en la topología dinámica de las MANETs en relación con soportar servicios de streaming de vídeo en redes móviles ad-hoc. Los principa-

les problemas son el enrutamiento dinámico (debido a que los nodos son móviles), QoS (calidad de servicio), necesidad de no usar demasiada energía y finalmente la seguridad para disminuir la vulnerabilidad de ataques.

En el curso 2019/2020 se propuso un TFG [4] con el mismo objetivo que el nuestro usando Wifi Aware para establecer la comunicación, dicho proyecto no usaba un protocolo de control lo que provocaba una latencia y una perdida de calidad en el *streaming*, además de solo poder realizar una comunicación *multicast* entre pares de dispositivos por lo que no se contemplaron problemas de enrutamiento.

En el curso 2018/2019 se propuso un TFG [3] con el mismo objetivo pero usando Wifi Direct, esta tecnología dio problemas en poder implementar las funcionalidades descritas pero lograron realizar una comunicación *multicast* con protocolo de control entre pares de dispositivos. Con nuestro proyecto vamos a resolver estos problemas estableciendo comunicaciones mediante Wifi Aware, adaptando a estas el protocolo de comunicación del TFG de hace dos años y realizar *multihopping* y *multicast* para crear una red más amplia, además de solventar los problemas de los bucles que van ligados a ella.

1.2. Objetivos

El objetivo del proyecto en el que se enmarca tanto este TFG como los TFGs de los dos últimos años ya mencionados se describe en la Sección 1.2 de [3] como sigue: “diseñar e implementar una aplicación para dispositivos móviles capaz de conectar los dispositivos entre ellos sin utilizar la infraestructura de red de los operadores de telecomunicaciones. Una vez creado el enlace entre dispositivos, la aplicación debe poder transmitir vídeo en directo desde un móvil a otro, que debe a su vez ser capaz de recibir ese vídeo y –reproducirlo,” (es decir, visualizarlo en pantalla) “si el usuario lo quiere– siendo capaz al mismo tiempo que recibe el vídeo de retransmitirlo a otros dispositivos a los que esté conectado, actuando de esta manera como un nodo transitorio entre dispositivos no conectados directamente entre ellos.”

Los escenarios de uso de la aplicación que se pretende construir en el proyecto en el que se enmarca tanto este TFG como los TFGs de los dos últimos años se explican en la Sección 1.2 de [4] como sigue: “Dada los diferentes modos de uso que se pueden dar para la aplicación, hemos decidido distinguir dos escenarios relevantes según el entorno donde se vaya a usar. Estos son:

- ***Humanitarian*:** Este escenario se centra en casos donde no hay conexión a internet debido a acciones ajenas al ser humano como puede ser catástrofes naturales o apagones de la red eléctrica de forma accidental. Permitiendo que la información se transmita desde los focos de dicho escenario hasta puntos donde haya cobertura, donde se puede pedir ayudar o compartir los vídeos con los medios de comunicación.
- ***Witness*:** Este escenario estaría enfocado a situaciones en las que un régimen opresivo limita o censura las comunicaciones. Ejemplos de estos escenarios

podrían ser ejecuciones o actos contra los derechos humanos, los cuales serían útiles grabarlos para su denuncia internacional."

Otro escenario de uso de la aplicación, parecido al que se ha llamado *Humanitarian*, que se nos ha ocurrido es para emergencias en zonas aisladas que nunca tienen cobertura de telefonía móvil (es decir, la falta de red no se debe a un catástrofe natural). Un ejemplo podría ser un accidente de escalada en una zona aislada de la sierra de Gredos.

Los objetivos iniciales de este trabajo son los siguientes:

- Comprobar la idoneidad para nuestra aplicación, de entre las opciones existentes, del sistema Wifi Aware [7] del Wifi Alliance para la comunicación D2D entre teléfonos móviles y para la creación de una red ad hoc compuesta de teléfonos móviles.
- Comprobar la idoneidad para nuestra aplicación, de entre las opciones existentes, de la biblioteca de software libre libstreaming [2] para formar la base de la implementación del streaming de la cámara de vídeo y del micrófono de un teléfono móvil, lo que implicaría descartar la posibilidad de basar nuestra aplicación en la implementación construida en el TFG del curso 2020/21.
- Comprobar la conveniencia para nuestra aplicación de incorporar y adaptar las extensiones de libstreaming implementados en el TFG del curso 2018/19 y, de este modo, partir de una base más adecuada para nuestros fines que la biblioteca original.

Suponiendo la idoneidad de las tecnologías Wifi Aware y libstreaming para formar la base de nuestra aplicación, y la conveniencia de incorporar y adaptar las extensiones de libstreaming implementados en el TFG del curso 2018/19, los objetivos principales de este trabajo son los siguientes:

- Adaptar la biblioteca libstreaming a Wifi Aware (que incluye su adaptación a IPv6 ya que Wifi Aware requiere esta versión del protocolo IP).
- Diseñar e implementar el *multihopping* de *streams* en una red ad hoc de teléfonos móviles creados con Wifi Aware.
- Diseñar e implementar el envío de *streams* a múltiples destinos de una red ad hoc de teléfonos móviles creados con Wifi Aware.
- Diseñar e implementar tanto en el dispositivo origen de un *stream* que transita por una red de teléfonos móviles creados con Wifi Aware como en cualquier dispositivo de la red que lo recibe (posiblemente simultáneamente con la recepción de otros *streams*) la posibilidad de visualizarlo en pantalla.
- Diseñar e implementar tanto en el dispositivo origen de un *stream* que transita por una red de teléfonos móviles creados con Wifi Aware como en cualquier

dispositivo de la red que lo recibe (posiblemente simultáneamente con la recepción de otros *streams*) la posibilidad de guardarlo en fichero para su posterior visualización.

- Diseñar e implementar alguna o algunas de las extensiones que podrían ser útiles en uno de los escenarios de uso mencionados anteriormente.

1.3. Plan de trabajo

Para poder llevar a cabo el proyecto y conseguir los objetivos propuestos, hemos realizado la planificación que se muestra en la Figura 1.1.

TAREA		INICIO	FIN
Investigación		15/09/2020	10/12/2020
	Tecnologías de comunicación	15/09/2020	19/09/2020
	Wifi Aware	19/09/2020	28/09/2020
	Proyecto 2019/2020	28/09/2020	20/10/2020
	Proyecto 2018/2019	20/10/2020	29/10/2020
	Libstreaming	29/10/2020	10/12/2020
Redacción de memoria		10/12/2020	05/06/2021
Especificación de requisitos		10/12/2020	24/12/2020
Prototipo		24/12/2020	13/01/2021
Diseño y desarrollo		13/01/2021	15/05/2021
	Adaptación Servidor y Cliente RTSP	13/01/2021	7/02/2021
	Multihopping	7/02/2021	20/02/2021
	Transmisión múltiple	20/02/2021	25/03/2021
	Guardar *streams*	25/03/2021	10/04/2021
	Aplicación del botón de pánico	10/04/2021	20/04/2021
	Interfaz gráfica	20/02/2021	15/05/2021

Figura 1.1: Plan de trabajo.

2. Introduction

2.1. Motivation

In MANETs, nearby users communicate directly by configuring their interfaces in ad-hoc mode, for the purpose of data transfer, either directly or through a transit node to redirect traffic to other nodes in the network. In many situations such as disasters or emergencies, where network infrastructures provided by network operators are not available, this kind of network is an ideal solution because it offers decentralized environments.

As observed in Section 1.1 of [3], "an ad hoc network could be formed to pass videos and/or help messages from one device to another, until a device is found that has access to the internet, and in this way the data sent from an area without connection will be published on the internet and reach their destination.

Another possible use case for an application like the one we propose could be to give the possibility of transmitting videos to distant devices in countries where human rights crimes are committed, in such a way that the video would pass from one device to another bypassing the control and censorship that countries like these establish in conventional networks, in addition to giving the possibility of hiding the origin of the transmission, in such a way that the complainant can not be identified. For this reason, in addition, the application should not leave traces of the videos, neither in the issuing mobiles nor in those that redistribute them. If the network is able to cross the censorship zone, the videos could be published on the internet with the help of organizations such as Witness [5] that are dedicated to promoting the creation and custody of videos of this type.".

In particular, for video custody there is the SecureDrop [6] system, designed for journalists to upload their documents or videos to a server on the Tor network anonymously.

Mobile ad hoc networks are becoming an important part of the next generation networks, due to their flexibility, self-configurability, lack of infrastructure, ease of maintenance, self-management capability and low cost. In this context, the last two decades have seen a great deal of research in the academic community regarding mobile networks. One of the areas of research in this field is data transmission over MANETs, this is due to the increase in computational capacity of smartphones and their great popularity in today's society.

It is important to mention the challenge of providing a certain quality in the shared wireless medium and in the dynamic topology of MANETs in relation to the support of video streaming services in mobile ad-hoc networks. The main issues are

dynamic routing (due to mobile nodes), QoS (quality of service), the need for low energy use and finally security to decrease vulnerability to attacks.

In the course 2019/2020 a bachelor's degree final project [4] with the same goal as ours was proposed using Wifi Aware to establish the communication, this project did not use a control protocol which caused latency and a loss of quality in the streaming, in addition to only being able to perform multicast communication between pairs of devices so routing problems were not contemplated.

In the course 2018/2019 a bachelor's degree final project [3] was proposed with the same goal but using Wifi Direct, this technology gave problems in the implementation of the described functionalities but managed to perform a multicast communication with control protocol between pairs of devices. With our project we will solve these problems by establishing communications using Wifi Aware, adapting to these communications the implementation of the libstreaming library, together with the modifications introduced in the bachelor's degree final project of 2018/19, and perform multihopping and multicast to create a wider network, in addition to solving the problems of the loops that can arise in this context.

2.2. Goals

The goal of the project in which context this bachelor's degree final project and those of the previous two years mentioned in Section 2.1 have been carried out is described in Section 1.2 of [3] as follows: "design and implement an application for mobile devices capable of connecting devices to each other without using the network infrastructure of telecommunication operators. Once the link between devices is created, the application must be able to stream live video from one mobile to another, which must in turn be able to receive that video and –play it back," (i.e, display it on screen) "if the user wants to– while being able to retransmit the video to other devices to which it is connected, thus acting as a transitory node between devices not directly connected to each other."

The usage scenarios of the application being developed through bachelor's degree final projects, of which the present project is the most recent, are described in Section 1.2 of [4] as follows: "Given the different possible modes of use for the application, we have decided to distinguish two relevant scenarios depending on the environment where it is to be used. These are:

- **Humanitarian:** This scenario focuses on cases where there is no internet connection due to non-human actions such as natural disasters or accidental power outages. Allowing the information to be transmitted from the sources of such a scenario to points where there is coverage, where you can ask for help or share the videos with the media.
- **Witness:** This scenario would be focused on situations in which an oppressive regime limits or censors communications. Examples of such scenarios could be

executions or acts against human rights, which would be useful to record for international denunciation."

Another usage scenario of the application, similar to the one called Humanitarian, that we have come up with is for emergencies in isolated areas that never have cell phone coverage (i.e. the lack of network is not due to a natural disaster). An example could be a climbing accident in an isolated area of the Gredos mountain range.

The initial goals of this work are the following:

- Check the suitability for our application, among the existing options, of the Wifi Alliance's Wifi Aware [7] system for D2D communication between cell phones and for the creation of an ad hoc network composed of cell phones.
- Check the suitability for our application, among the existing options, of the open-source software library libstreaming [2] to form the basis of the streaming implementation of the video camera and microphone of a cell phone, which would imply ruling out the possibility of basing our application on the implementation built in the bachelor's degree final project of the 2020/21 course.
- Check the suitability for our application of incorporating and adapting the libstreaming extensions implemented in the bachelor's degree final project of the 2018/19 academic year and, thus, starting from a base more suitable for our purposes than the original library.

Assuming the suitability of Wifi Aware and libstreaming technologies to form the basis of our application, and the convenience of incorporating and adapting the libstreaming extensions implemented in the 2018/19 bachelor's degree final project, the main goals of this work are the following:

- Adapt the libstreaming library to Wifi Aware (which includes its adaptation to IPv6 since Wifi Aware requires this version of the IP protocol).
- Design and implement multihopping of streams in an ad hoc network of cell phones created with Wifi Aware.
- Design and implement multicast streaming over an ad hoc network of cell phones created with Wifi Aware.
- Design and implement both in the source device of a stream that transits through a network of cell phones created with Wifi Aware and in any device of the network that receives it (possibly simultaneously with the reception of other streams) the possibility of displaying it on the screen.
- Design and implement both in the source device of a stream that transits through a network of cell phones created with Wifi Aware and in any device of the network that receives it (possibly simultaneously with the reception of other streams) the possibility of saving it in a file for future viewing.
- Design and implement one or some extensions that could be useful in one of the above mentioned usage scenarios.

2.3. Work plan

In order to carry out the project and achieve the proposed goals, we defined the project plan shown in Figure 1.1.

3. Antecedentes

3.1. Comunicación DTD

La comunicación Device-to-Device (DTD) es un tipo de comunicación directa entre dos dispositivos móviles, que intercambian información sin tener que atravesar redes centralizadas o estaciones de telefonía móvil (Base Stations). El término ha evolucionado de tal forma que se asocia a los intentos por las operadoras de telefonía de descongestionar y mejorar sus redes. En las redes móviles tradicionales todas las comunicaciones deben atravesar una estación de telefonía (BS), incluso si los dispositivos implicados en la comunicación se encuentran próximos.

La tecnología DTD tiene un gran potencial para mejorar la eficiencia en el uso de la energía y el espectro de frecuencias, aprovechándose de esta proximidad entre dispositivos. Las comunicaciones se pueden establecer en el espectro de frecuencias utilizado actualmente por las redes móviles o fuera de este, lo que trae distintos aspectos a tener en cuenta a la hora de su implementación.

En general la comunicación DTD promete ganancias en la reutilización del espectro electromagnético, conectividad a una mayor distancia de las estaciones de telefonía, independencia con respecto a las redes centralizadas y la descongestión de la red gracias a los nuevos caminos para alcanzar a los dispositivos, entre otras. Los operadores están interesados en usar esta comunicación para la última parte de la conexión en situaciones de congestión local, por ejemplo, un evento deportivo. Mediante el D2D se podría gestionar el tráfico de una zona congestionada a base de estaciones que no están en esta zona. Precisamente, si hay mucho uso de la red de teléfono móvil en una zona concreta, hay muchos nodos en esta zona que podrían usarse para la comunicación D2D.

Actualmente se está investigando cómo afrontar los problemas [8] relacionados con la comunicación DTD, siendo las operadoras de telefonía móvil las más interesadas. Entre ellos tenemos:

- La búsqueda continua de dispositivos y servicios mediante Wifi o Bluetooth LE. Utiliza medios de sincronización para hacer un consumo eficiente de la batería.
- La administración del nuevo escenario de interferencias causadas por las conexiones DTD. La introducción de estas conexiones no debería degradar el uso tradicional centralizado de la red y es necesario tener en cuenta que estas redes tradicionales pueden afectar a la calidad del servicio requerido por las DTD.
- La reducción del consumo energético en las tareas de descubrimiento de servicios y otros dispositivos.

- La integración de las redes DTD con las centralizadas y su aprovechamiento para extender el alcance de las estaciones de telefonía y añadir posibles caminos para descongestionar la infraestructura actual. Utilizando dispositivos a modo de puente se podrían conectar otros fuera de rango con la infraestructura principal, algo bastante necesario en la tecnología 5G, que tiene problemas con cobertura.

- Distinción entre dispositivos públicos y de conexión privada con restricciones, así como seguridad y privacidad en las comunicaciones.

Si se lograra solucionar estos problemas, esta tecnología sería un gran aporte para su uso en áreas como IoT, comunicaciones de emergencia sin infraestructura o 5G incluso. A continuación, se muestra una visión actual de tecnologías basadas en la comunicación DTD.

3.1.1. Bluetooth

Es un estándar de tecnología inalámbrica, mantenido por la organización Bluetooth SIG, para el intercambio de datos entre dispositivos a corta distancia y la creación de redes de área personal (WPAN). Utiliza ondas de radio de alta frecuencia (UHF) en las bandas ISM (no comerciales) de 2,4GHz. Especifica un protocolo basado en paquetes, con una arquitectura maestro/esclavo que permite conectar como máximo 7 esclavos a un maestro, en lo que se llama una piconet.

La última versión Bluetooth 5.x permite un rango de hasta 500 m [9] y una velocidad máxima de 2 Mbit/s [10], a un rango mucho menor. Introduce características enfocadas al área del IoT [11], centradas en la reducción de consumo energético Low Energy (LE) y funcionalidad orientada a servicios sin conexión, como servicios de navegación entre otros. Además introduce “Bluetooth LE Audio” [12] (2020) que intenta mejorar la calidad de sonido y la eficiencia energética entre los dispositivos que se conectan para funciones de audio, además de permitir su transferencia a varios dispositivos a la vez.

Esta tecnología se revela inadecuada para la transmisión de video y audio por su insuficiente ancho de banda y su radio de alcance limitado.

3.1.2. Wifi

Es una familia de protocolos de red inalámbrica para la capa de enlace y la capa física del modelo OSI, pensados principalmente para conectar dispositivos formando una WLAN. Están basados en la familia de estándares IEEE 802.11. Las conexiones Wifi utilizan protocolos de seguridad para evitar que se conecten terceros a la red inalámbrica y poder encriptar los datos. La más segura y destacada en la actualidad es WPA2, aunque siguen en uso otros estándares menos seguros como WEP o WPA.

En 2009 surgió el estándar 802.11n, que permite la comunicación a través de frecuencias de 2.4 y de 5GHz. Esta última, al funcionar en una frecuencia superior,

tiene un menor alcance, pero también permite al dispositivo operar a una mayor velocidad, la cual oscila entre 72 y 600 Mbit/s [13]. El siguiente avance tecnológico ocurrió en 2014 con 802.11ac, nombrado Wifi 5 por la Wifi Alliance, que opera a 5Ghz ofreciendo una tasa de bits de hasta 6933 Mbit/s [14]. En 2019 la Wifi Alliance publicó el borrador de un nuevo estándar, 802.11ax, llamado Wifi 6, que promete operar en las bandas de 2.4, 5 y 6 GHz ofreciendo tasas de bits entre 600 y 9680 Mbit/s, además de incorporar nuevos protocolos de seguridad más robustos como WPA3 [15]. Está pendiente de ser aprobado por la IEEE actualmente (2020), aunque ya existen dispositivos que lo implementan.

Para la realización de este proyecto queremos crear una red ad-hoc en la que los dispositivos se comuniquen directamente entre sí, sin necesidad de utilizar un punto de acceso. Para ello el uso de los protocolos Wifi nos parece lo más adecuado por sus capacidades físicas y de difusión.

El término 'Wifi' describe unos protocolos que permiten la comunicación hertziana entre dos dispositivos, una especie de Ethernet por el aire. Pero el uso principal de Wifi es con un punto de acceso, o bien doméstico, o bien público. En el caso del punto de acceso doméstico, suelen estar combinado tres funciones en un único aparato:

1. Punto de acceso: por un lado, se conecta con dispositivos individuales por Wifi y por otro lado, se conecta con otra(s) red(es) (en este caso, Internet) por cable o fibra óptica.

2. *Switch* (de una red de área local, o LAN, con topología estrella): cuando recibe un mensaje de un dispositivo conectado directamente al aparato cuyo destinatario es otro dispositivo también conectado directamente al aparato, lo envía al destinatario correcto. Si todos los nodos de la red local se comunican con el *switch* por Wifi, la LAN es una LAN inalámbrica (o "wireless LAN", o WLAN).

3. *Router*: encamina los mensajes enviados hacía, y recibidos desde, otras redes (LANs, WANs,...) de Internet.

En el caso de un punto de acceso público, no suele estar incluida la funcionalidad de *switch*.

3.1.3. Wifi Ad-hoc mode

Es una estructura de red inalámbrica basada en la tecnología Wifi, donde los dispositivos pueden comunicarse directamente entre sí. Pertenece a una característica adicional que se especifica en la familia de estándares 802.11, conocido como conjunto de servicios básicos independientes (IBSS).

El modo Ad-hoc elimina la necesidad de utilizar un punto de acceso, permitiendo la comunicación directa entre dispositivos. Este modo es utilizado para situaciones donde es necesario una comunicación rápida y eficiente, se suele usar en grupos

pequeños donde el propósito principal de la conexión es compartir archivos. Este tipo de redes descentralizadas son típicamente más robustas, debido a los múltiples caminos que ofrecen para transportar información. La probabilidad de que se caiga la red por un fallo en un punto se reduce significativamente. Cuenta además con mayor flexibilidad y escalabilidad que redes centralizadas, al no tener una topología fija.

Estas estructuras aún se encuentran en desarrollo y no tienen un uso muy extendido debido a los retos que plantean, como una mayor complejidad en la comunicación, posibles reducciones en el rendimiento a causa de la topología dinámica, un excesivo uso de energía entre otras. Por estas razones y por la programación a más bajo nivel que daría problemas con actualizaciones del sistema operativo y el descubrimiento ineficiente, no es útil para el desarrollo de este proyecto.

Estándares como Wifi Direct y Wifi Aware buscan superar estas limitaciones haciendo uso de este modo para crear conexiones inalámbricas Peer to Peer entre dispositivos móviles, formando MANETs, Mobile Ad hoc networks.

3.1.4. Wifi Direct

Es un estándar diseñado para la comunicación Peer to Peer, en el que los usuarios pueden intercambiar archivos de forma inalámbrica sin necesidad de un enrutador central para organizar el tráfico y transmitir los paquetes. A diferencia de Bluetooth, el intercambio de datos se realiza a gran velocidad, pudiendo ser hasta diez veces más rápido.

Esta tecnología sirve para la impresión inalámbrica, retransmisión de tu pantalla en otro dispositivo, compartición de pantalla o incluso para explotar al máximo las funcionalidades del NFC para el proceso de descubrimiento y detección de otros dispositivos.

La tecnología de Wifi Direct ofrece a los dispositivos compatibles una forma de descubrir y conectarse de forma segura mediante WPS y WPA2. Cuando varios dispositivos se conectan forman un grupo, el cual tiene un Propietario de Grupo que actúa como *switch* (y posiblemente también como punto de acceso y *router*) de una WLAN con topología estrella formado por todos los dispositivos del grupo. Solamente el Propietario de Grupo necesita implementar el estándar para que sea posible la conexión, también puede hacer la función de puente con redes con infraestructura y así dotar de acceso a internet a los dispositivos del grupo.

En 2011, Google implementó Wifi Direct en Android 4.0 de forma nativa por lo que cualquier dispositivo Android en la actualidad dispone de esta tecnología. Wifi Direct parece una buena solución para trabajar con MANETs, por su tasa de bits y su incorporación en la mayoría de dispositivos móviles, pero consideramos que no lo es para nuestra aplicación de *streaming*.

Las razones son que Wifi Direct tiene ciertas limitaciones en cuanto al Propietario de grupo que comprometen la calidad de la transmisión. Como se puede ver de forma detallada en la investigación que realizaron los alumnos del proyecto de 2018/19, en el capitulo 5.1.3 de [3], los cuales utilizaron Wifi Direct para conectar los dispositivos, las limitaciones son:

- Todas las comunicaciones deben pasar por el Propietario de grupo, lo cual restringe la distancia y el número de conexiones, además de sumar complejidad a la transmisión (página 9 de [16]).
- La imposibilidad de ser miembro de más de un grupo de Wifi Direct es una parte opcional de la especificación del Wifi Alliance que Google elegió no implementar en Android. Para pasar un stream entre un grupo y otro, el nodo que quiere pasar el stream a otro grupo, tendría que ir guardándolo en memoria y, en un momento dado, para la transmisión, salir del grupo y empezar a retransmitir en el otro grupo, añadiendo una latencia en el proceso de salir de un grupo y juntarse a otro (página 9 de [16] y [17]).

3.1.5. Miracast

Es un protocolo D2D para la transmisión de video/audio estandarizado por la Wifi Alliance. Grandes empresas como Google le han dado la espalda en favor de protocolos propietarios que pasan por un punto de acceso, switch o router en vez de D2D.

Este protocolo no nos sirve para este proyecto por dos razones, la primera es porque está construido encima de Wifi Direct y, por tanto, hereda sus problemas, y la segunda es porque está orientado a "mirroring", es decir, enviando a otra pantalla, no a la transmisión general.

3.1.6. Wifi Aware

Es una tecnología que hace uso de Wifi para la creación de conexiones Peer to Peer entre dispositivos móviles. En base a las preferencias en tiempo real de los usuarios y la distancia a los dispositivos, permite buscar y formar conexiones sin necesidad de puntos de acceso públicos ni GPS.

Su especificación surgió en 2015, pero no fue tenida en cuenta hasta sus últimas versiones alrededor de 2019. Está implementado en Android a partir de la versión 8 con alguna adición de funcionalidad en Android 9 y cada vez más móviles pueden hacer uso de ella.

Sus principales características son:

- La búsqueda continua de dispositivos y servicios mediante Wifi o Bluetooth LE. Utiliza medios de sincronización para hacer un consumo eficiente de la batería.

- La medida precisa de la distancia con respecto a otros dispositivos con Wifi Aware, mediante el protocolo Wifi Fine Time Measurements (FTM). Permite tomar acciones o avisar al usuario cuando se cumplen ciertas condiciones basadas en la medida precisa de la distancia con otros dispositivos (Geofencing).
- Capacidad de utilizar *multicast* con los dispositivos de la red.
- Permite que distintas aplicaciones creen redes independientes, haciendo uso del hardware de forma simultánea.
- Intercambio de datos a través de conexiones IP.

Los dispositivos forman grupos (*clusters*) en los que se sincronizan los mensajes para buscar servicios y nuevos dispositivos de forma continua. El intercambio de mensajes se planifica en intervalos y canales específicos para reducir el consumo energético. El flujo de datos no pasa por un Propietario de Grupo como ocurría con Wifi Direct, además permite el intercambio de estos datos y servicios de manera multidireccional y simultáneo con la buena tasa de bits que tiene Wifi.

Esta es la tecnología que hemos utilizado para implementar nuestra aplicación de *streaming* en MANETs. La capacidad de buscar y establecer conexiones de forma eficiente, el uso condicional de las distancias, la capacidad de usar *multicast* e IP para los *streams* y la resolución de las restricciones de Wifi Aware en cuanto a las transmisiones entre grupos, nos brindan una buena base para ofertar *streams* de calidad.

3.1.7. LTE Direct

Es una tecnología Device to Device para el descubrimiento de dispositivos por proximidad, que busca mejorar las implementaciones actuales, basadas en GPS o Bluetooth LE, utilizando el estándar para telecomunicaciones inalámbricas de alta velocidad LTE, empleada mayoritariamente en la actualidad en la arquitectura 4G.

Promete mejorar los rangos de alcance de Wifi Direct, Wifi Aware y Bluetooth, hasta los 500 m, así como su escalabilidad a miles de conexiones. Tiene como objetivos la eficiencia energética y la privacidad en cuanto al descubrimiento de otros dispositivos. Utiliza medios de sincronización para buscar dispositivos a intervalos pequeños de tiempo y así ahorrar energía, de forma similar a como lo hace Wifi Aware. Plantea además la posibilidad de desplegar servicios privados, que solamente un espectro limitado de dispositivos pueda acceder a ellos.

Busca coexistir con otras tecnologías de descubrimiento de dispositivos como Bluetooth LE, aplicando la mejor en función de la distancia. A pesar de que la especificación de esta tecnología se introdujo en 2015 en el estándar de 3GPP Versión 12, en la actualidad no hay más que unas pocas implementaciones [18] y pruebas [19] que confirman el potencial y los casos de uso de la tecnología.

Qualcomm anuncia modelos de negocio para LTE Direct dirigidos a la monetización por parte de las empresas de telefonía móvil [20], como una suscripción mensual por su uso o cargos por el acceso a las APIs de LTE Direct, lo cual plantea restricciones de cara a los desarrolladores y a la difusión de aplicaciones que utilicen LTE Direct.

3.1.8. NFC

Es una tecnología de comunicación inalámbrica que permite la transferencia de datos en corto alcance sin la necesidad de emparejamiento previo, funciona en la banda de los 13.56 MHz y su tasa de transferencia puede alcanzar los 424 kbit/s, por lo que su enfoque en lugar de ser para la transmisión de grandes cantidades de datos, va más dirigido hacia la comunicación instantánea concretamente de identificación y validación de equipos y personas, es usada en la tecnología de pago sin contacto (tarjeta o móvil).

La tecnología NFC puede funcionar en dos modos, el primero es el modo activo en el que ambos dispositivos generan un campo electromagnético e intercambian datos, el segundo es el modo pasivo en el que solo hay un dispositivo activo y el otro aprovecha el campo que genera para intercambiar la información.

Debido a la baja velocidad de transmisión y sobre todo al alcance máximo de 20 centímetros para poder realizar la comunicación, observamos cómo esta tecnología es incompatible con el objetivo de este proyecto.

3.2. Protocolos de interés

3.2.1. SCTP

El SCTP (stream control transmission protocol) [21] es un protocolo de transporte fiable de la familia de protocolos de Internet que permite la transmisión de mensajes de telecomunicación a través de redes IP. Reúne diversas características de dos protocolos también encargados de la transferencia de datos: el TCP (orientado a la conexión) y el UDP (sin conexión). Además, contiene, entre otras cosas, mecanismos para controlar la congestión y mejorar la tolerancia de fallos en el envío de paquetes.

Este protocolo puede multiplexar varios “flujos de datos” en una asociación SCTP. Cada mensaje está asociado con un número de flujo y los mensajes que pertenecen al mismo flujo se entregan en orden. Sin embargo, mientras que un flujo puede estar bloqueado esperando el siguiente mensaje en secuencia, la entrega de otros flujos puede continuar, evitando el bloqueo de cabeza de línea. Además dicho protocolo tiene como uno de los principales objetivos la seguridad, por ejemplo haciendo uso del handshake de cuatro vías que es imprescindible para la transferencia de datos mediante SCTP, sirve para evitar el conocido problema de TCP llamado el ataque de inundación de SYN.

Gracias a su gran flexibilidad, el SCTP también se usa para otros fines como por ejemplo, para controlar y administrar grupos de servidores.

3.2.2. RTP - RTCP

Son dos protocolos que proporcionan servicios de entrega de datos, como el audio y vídeo en tiempo real.

RTP (Real-Time Transport Protocol) es un protocolo de transporte utilizado sobretodo para la transmisión multimedia en tiempo real. Suele ejecutarse sobre UDP para hacer uso de sus servicios de multiplexación y no proporciona ningún mecanismo para garantizar la recepción de los paquetes o proporcionar otras garantías de calidad de servicio, pero es usado por RTP en lugar de TCP debido a que reduce el tiempo de envío de los paquetes a través de la red. En aplicaciones de *streaming* de vídeo y audio como la de este proyecto es más importante la transmisión rápida que la pérdida de ciertos paquetes durante la comunicación.

RTCP (RTP Control Protocol) es un protocolo de control utilizado por RTP para monitorizar la calidad de servicio y transmitir información entre los extremos. Estos paquetes de control son enviados a todos los participantes de la sesión aproximadamente cada cinco segundos y pueden utilizarse para estabilizar la calidad de la transmisión.

3.2.3. RTSP

RTSP (Real-Time Streaming Protocol) [22] es un protocolo no orientado a conexión perteneciente a la capa de aplicación, usado para la configuración y el control de la entrega de datos en tiempo real, como por ejemplo para solicitar un *stream* multimedia. Suele usarse junto a protocolos de transmisión de flujo como RTP.

RTSP define dos modalidades de comunicación que mostramos a continuación:

1. **Modalidad de publicación:** Consiste en que el nodo que implementa el cliente, activamente, solicita al servidor RTSP que acepte su *stream* y lo distribuya a otros clientes, para ello inicialmente el cliente RTSP envía un mensaje ANNOUNCE donde se describen los canales multimedia que ofrece, a continuación se envía un mensaje SETUP por cada canal multimedia, es decir uno para el vídeo y otro para el audio, en el que se establecen los puertos que se van a utilizar para la comunicación y finalmente se envía un mensaje RECORD indicando al servidor que comienza el *streaming*.

2. **Modalidad de reproducción:** Consiste en que el nodo que implementa el cliente solicita un *stream* al servidor RTSP, el cual puede ser o bien producido por la cámara y micrófono del propio servidor mencionado o por otro dispositivo mediante la modalidad anterior. Para ello inicialmente se envía un mensaje DESCRIBE en el que en función de la URI enviada, el servidor responde con la información del *stream* asociado, a continuación el cliente RTSP envía un

mensaje SETUP por cada canal multimedia del que esté interesado y específica cómo va a ser la transmisión, finalmente se envía un mensaje PLAY para notificar al servidor RTSP que puede empezar a enviar el flujo solicitado.

En el contexto de nuestra aplicación, la modalidad que mejor se adapta a nuestros escenarios de uso es la de publicación, de esta forma un nodo que implementa la parte cliente de RTSP que quiere enviar un *stream* a otro nodo que implementa la parte servidor de RTSP lo puede transmitir con esta modalidad al otro nodo. Es notable mencionar que esta modalidad ya no existe en RTSP 2.

Este protocolo está destinado a controlar múltiples sesiones y proporcionar un medio para elegir canales de entrega de datos, como canales UDP basados en RTP. RTSP tiene tres partes fundamentales que son: establecimiento de la sesión (SETUP), control de entrega de datos (PLAY y PAUSE) y un modelo de extensibilidad (OPTIONS). La funcionalidad de reproducción básica de RTSP permite la entrega de una variedad de contenido solicitado al cliente, al ritmo previsto por el creador del contenido. Sin embargo RTSP también puede manipular la entrega al cliente de dos formas:

1. Según la proporción de tiempo de datos entregados por unidad de tiempo de reproducción.

2. Según la relación del tiempo de reproducción entregado por unidad de tiempo.

3.2.4. SDP

SDP (Session Description Protocol) [23] se utiliza como un protocolo auxiliar para transmitir detalles de los datos, direcciones de transporte y otra información de sesión a los participantes de la comunicación. Proporciona una representación estándar para dicha información independientemente del transporte de esta, sin embargo, no está destinado para la negociación del contenido o de las codificaciones de los datos debido a que se consideran fuera del alcance de este protocolo.

Este protocolo es usado por ejemplo en RTSP cuando un cliente y un servidor negocian un conjunto apropiado de parámetros para la entrega de flujos de vídeo y audio, utilizando parcialmente la sintaxis SDP para describir esos parámetros. Una descripción de sesión SDP, además de los detalles de dirección y puerto, incluye el tipo de medio (vídeo, audio, etc.), protocolo de transporte (RTP / UDP / IP, H.320, etc.) y el formato de los medios (H.261, MPEG-1, etc.).

3.2.5. HLS (HTTP Live Streaming)

HTTP Live Streaming o HLS es un protocolo de comunicación utilizado para la transmisión de audio y vídeo sobre HTTP que permite una velocidad de bits adaptable. Además, la adaptación puede cambiar dinámicamente durante una transmisión. Este protocolo funciona dividiendo el flujo general en una secuencia de pequeñas descargas de archivos basadas en HTTP, cada una de las cuales descarga un fragmento corto de un flujo de transporte.

A diferencia de los protocolos basados en UDP como RTP, este puede atravesar cualquier servidor de seguridad o servidor proxy que permita el tráfico HTTP estándar debido a que usa el puerto 80. La codificación del vídeo se realiza por segmentos, normalmente cada 10 segundos, además se adapta la calidad de imagen reproducida según los recursos de banda ancha de cada usuario.

Para lograr realizar la transmisión mediante Adaptive Bit-Rate (ABR), inicialmente es necesario codificar el vídeo en varias velocidades de bits según el perfil de transmisión del destino, a continuación se segmentan estos archivos codificados para que funcione bien con los protocolos de transmisión basados en HTTP que descargan este contenido de manera progresiva. Seguidamente el reproductor de vídeo puede solicitar los recursos que más se adapten al dispositivo y a la conexión del usuario basándose en dos tipos de algoritmos que son de rendimiento o de búfer.

3.2.6. PPSPP (Peer-to-Peer Streaming Peer Protocol)

El uso habitual del término 'P2P streaming' describe el envío de *streams* siguiendo el procedimiento de dividirlos en trozos y enviarlos por distintos caminos en una red *overlay* [24], al estilo de BitTorrent. Se está intentando estandarizar protocolos para este tipo de streaming, en particular el Peer-to-Peer Streaming Peer Protocol (PPSPP) [25].

El caso de uso habitual es con pocos dispositivos que siempre actúan como emisores o muchos dispositivos que siempre actúan cómo receptores. Se observa un parecido entre el interés principal en usos de P2P Streaming, que es aliviar la carga en las redes de un gran número de emisores de vídeo, y el interés principal en usos de DTD, que es aliviar la carga de las redes de los operadores, es decir, "traffic offloading".

3.2.7. RIST

RIST (Reliable Internet Stream Transport) es un protocolo de transporte diseñado para la transmisión confiable de vídeo a través de redes propensas a la perdida de datos, con baja latencia y alta calidad. Hace uso de RTP sobre UDP para paliar las limitaciones de TCP. La confiabilidad se consigue haciendo uso de retransmisiones basadas en NACK. Esta actualmente bajo desarrollo por el Video Services Forum.

3.3. Software de interés (énfasis en software libre)

Todo el software presentado en esta sección, salvo WebRTC y libstreaming, está orientado principalmente a la recepción de streams y su procesamiento en el receptor.

3.3.1. WebRTC

WebRTC (Web Real-Time Communication) [26] es una tecnología de estándar abierto que permite establecer comunicaciones en tiempo real, como conferencias

de video, sobre los principales navegadores web, como Safari, Chrome, Firefox, etc. Soporta vídeo, voz y contenido multimedia, que permite a los usuarios incorporar videoconferencias a cualquier servicio web y crear potentes soluciones de colaboración de vídeo basadas en la tecnología WebRTC.

Para que este protocolo transfiera datos en tiempo real, estos deben ser cifrados con anterioridad mediante DTLS (Datagram Transport Layer Security), que está diseñado para proteger la privacidad de las comunicaciones e integrado en la mayoría de los navegadores que soportan WebRTC . Además los datos de vídeo y audio también se encriptan usando otro método llamado SRTP (Secure Real-Time Protocol). De esta manera se garantiza una comunicación segura en tiempo real.

En la actualidad esta tecnología permite tratar los flujos de vídeo y audio en un navegador moderno sin tener la necesidad de usar una aplicación especifica, además está ganando terreno muy rápidamente y se propone revolucionar los estándares de comunicación.

Finalmente, es reseñable mencionar que WebRTC sin navegador constituye otra alternativa para nuestro proyecto. Google distribuye la librería libwebrtc, aunque desde el principio de 2020 no lo hace de forma precompilada para Android [27]. Hay varias empresas ofreciendo SDKs para usar WebRTC en Android sin navegador, muchas de estas ofertas están basadas en libwebrtc [28].

3.3.2. VLC

Es un reproductor multimedia de código abierto [29] con versiones para muchos sistemas operativos, capaz de reproducir distintos formatos de vídeo sin la necesidad de instalación de códecs externos.

Para emitir un stream RTP incluye una librería llamada LiveMedia de un grupo denominado Live555, pero esta librería es inmadura y no existe implementación en Android. Además, las páginas de Live555 están fuera de servicio desde hace años [30].

3.3.3. FFmpeg

FFmpeg es una colección de software libre desarrollada en GNU/Linux que puede ser compilada casi por cualquier sistema operativo que permite grabar, convertir y hacer *streaming* de vídeo y audio. La parte que concierne a la emisión de streams está bastante poco desarrollada [31] y con poca documentación [32].

3.3.4. Gstreamer

GStreamer es un *framework* multimedia escrito en el lenguaje de programación C que permite reproducir flujos de vídeo y audio entre otras. Aunque existe en Android, la parte que concierne a la emisión de streams está muy poco documentada, en particular, el uso de SDP.

3.3.5. Libstreaming

Libstreaming [2] es una librería que nos permite realizar fácilmente la transmisión de vídeo y audio de un dispositivo con Android usando el protocolo RTP sobre UDP. Además, incluye codificadores compatibles como H.264 (para el vídeo) y AAC (para el audio).

Es notable mencionar que ningún otro software libre de este tipo ha tenido el uso extendido que ha tenido libstreaming. Utiliza una licencia Apache 2.0 lo cual nos permite la libertad de usar este software para cualquier propósito, distribuirlo, modificarlo y distribuir versiones modificadas de este.

4. Elección de tecnologías

4.1. Android

Como el desarrollo de este proyecto es una aplicación para dispositivos móviles, la cual requiere un gran número de nodos para conseguir un tránsito adecuado del stream y de esta manera que pueda cumplir con su propósito, hemos optado por usar Android ya que según los datos de mediados de 2020 se percibe una tasa de utilización del 86.1 % [33], además de la amplia documentación que podemos encontrar sobre ello.

Por consiguiente, el sistema operativo que utilizaremos es Android 9+ que da soporte con su API a la tecnología WiFi Aware y es el que utilizan los dispositivos con los que vamos a realizar las pruebas, lo que facilita el desarrollo de este proyecto.

4.2. WiFi Aware

Hemos elegido WiFi Aware por varias razones, la primera es que añade a la conexión Wifi estándar la posibilidad de comunicarse y buscar otros dispositivos aportándonos más facilidad y un bajo consumo de energía. Otra de las razones es que queremos mejorar la aplicación que realizaron nuestros compañeros el año pasado usando esta tecnología e intentar solucionar los problemas que plantearon.

4.3. Java

Java es un lenguaje de programación multiplataforma y orientado a objetos. Al igual que Kotlin, es uno de los lenguajes más usados para el desarrollo de aplicaciones Android ya que funciona en prácticamente cualquier dispositivo, además nos proporciona mayor robustez ya que nos ofrece manejo automático de la memoria.

Una de las principales razones por la cual nos hemos decantado a usar Java es por la enorme cantidad de funcionalidad base que disponemos para ser utilizada, además de que la mayor parte de información que disponemos sobre Wifi Aware está disponible en este lenguaje.

4.4. Android Studio

Debido a la utilización de Android como sistema operativo para el desarrollo del proyecto, hemos elegido Android Studio, que es el entorno de desarrollo oficial de aplicaciones Android. Además de poder probar la aplicación de forma virtual y en nuestros dispositivos físicos que disponemos para el proyecto, nos proporciona también la posibilidad de creación de interfaces gráficas de forma visual. Todo esto nos proporciona una gran facilidad a la hora de desarrollar la aplicación de este proyecto.

4.5. RTP sobre UDP

RTP es el protocolo de transporte de flujos multimedia en tiempo real que utilizamos. Hemos elegido este protocolo debido a que es de los más utilizados en sistemas de retransmisión de vídeo y audio. Además este protocolo es usado por la librería "libstreaming" que ha sido la elegida para la realización del proyecto.

TCP es un estándar para el uso de RTP pero en este caso no hemos decidido utilizarlo porque debido a los mecanismos de control de congestión que este realiza, nos ralentizará la transmisión del *stream*, por este motivo hemos elegido UDP ya que en servicios de *streaming* la pérdida de algún paquete del envío es insignificante y nos demorara menos la entrega de datos.

4.6. RTSP y SDP

Para el envío de información entre cliente y servidor, hemos decidido usar RTSP con RTP sobre UDP para intentar conseguir una transmisión de vídeo y audio lo más eficiente posible controlando estos flujos de manera sincronizada. Las peticiones RTSP que se realizan son:

- **OPTIONS:** Suele enviarse al inicio para establecer los parámetros necesarios como el número que se asigna, la versión del servidor o los métodos soportados por este.
- **DESCRIBE:** Se encarga de inicializar la sesión utilizando la descripción del flujo necesario para la reproducción.
- **SETUP:** Especifica los parámetros de transporte como los puertos y el protocolo a utilizar.
- **PLAY:** Indica cuándo se debe iniciar la transmisión de datos utilizando los parámetros especificados en el SETUP.
- **PAUSE:** Detiene de forma temporal el flujo de datos.
- **TEARDOWN:** Detiene por completo el flujo de datos y libera los recursos.

- **GET_PARAMETER** y **SET_PARAMETER:** Se utiliza para recuperar o establecer algún parámetro del flujo de video o audio que se está transmitiendo.
- **REDIRECT:** Se utiliza para informar al cliente que el servidor ha cambiado de dirección.

Cabe mencionar que RTSP utiliza SDP para realizar la negociación de los parámetros para la entrega de los flujos de vídeo y audio, por ese motivo utilizaremos este protocolo para desempeñar la función mencionada anteriormente.

4.7. Libstreaming

Hemos decidido utilizar esta librería ya que nos permite con pocas líneas de código realizar la transmisión, además implementa parte del protocolo RTSP y partimos del código que realizaron los alumnos de 2018/2019, los cuales hicieron algunas extensiones muy útiles que podremos aprovechar para nuestra aplicación. Las otras alternativas mencionadas en la sección anterior no realizan bien la emisión de *streams*.

4.8. LibVLC

Para gestionar el contenido multimedia, vamos a utilizar libVLC, que nos proporciona una gran calidad y flexibilidad a la hora de reproducir y guardar los *streams*. Esta biblioteca de código abierto tiene una licencia LGPL2.1.

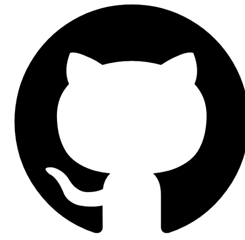
Hemos descartado FFmpeg ya que está a más bajo nivel (VLC utiliza FFmpeg) y respecto a GStreamer, es debido a que nos ha resultado más fácil la integración de VLC en el proyecto.

4.9. Balsamiq Wireframes

Balsamiq Wireframes es una herramienta que nos permite crear el diseño de las interfaces de usuario de forma rápida y sencilla para nuestra aplicación, además nos ofrece un complemento para Google Drive para poder usarla de forma colaborativa.

4.10. GitHub

GitHub es una de las principales plataformas para la creación de proyectos de software libre que se caracteriza por sus funciones colaborativas que permiten a los usuarios aportar ideas o modificaciones sobre ellos. La web utiliza el sistema de control de versiones Git que facilita a los desarrolladores administrar su proyecto, llevando un registro de cambios en los archivos que permite ver las diferencias entre ellos e incluso fusionar las distintas versiones.



Para el desarrollo de este proyecto hemos utilizado un repositorio alojado en GitHub para poder trabajar de forma colaborativa reduciendo de esta manera el número de conflictos que puedan ocasionarse.

4.11. LaTeX y Overleaf

LaTeX es un sistema de composición de textos, orientado especialmente a la creación de libros, artículos y documentos científicos proporcionando una alta calidad profesional. Una de las principales ventajas respecto a otros editores más convencionales es que nos permite separar claramente el contenido y el formato del documento, además de proporcionarnos gran ayuda para la gestión del índice y la bibliografía.

Overleaf es un editor colaborativo basado en la nube que se utiliza para escribir y editar documentos en LaTeX, compila el código de manera automática mostrando los resultados simultáneamente y no necesita la instalación de paquetes. Por todas estas facilidades que nos proporcionan estas dos tecnologías, hemos decidido utilizarlas para la realización de esta memoria.

5. Especificación

La idea principal de la aplicación es conseguir realizar una transmisión de vídeo y audio entre dos o más dispositivos móviles, creando una red ad-hoc mediante Wifi Aware sin pasar por una infraestructura fija que nos permita compartir el *stream* hasta un dispositivo que tenga conexión a internet o que pueda custodiar el vídeo con seguridad.

Para poder realizar una transmisión tenemos que definir tres tipos de modos que son los que usarán los dispositivos conectados para poder interaccionar con el resto:

- **Modo "emisor":** Se encarga de transmitir el vídeo y audio producido por la cámara y micrófono del propio dispositivo a través de la red creada mediante Wifi Aware.
- **Modo "receptor":** Se encarga de recibir el *stream* de la MANET.
- **Modo "dual":** Se encarga de realizar la función de un *router* básico, es decir, recibe un *stream* y simultáneamente lo emite hacia un tercer dispositivo.

Para poder cumplir con los objetivos del proyecto, y dado que vamos a usar un protocolo distinto al que idearon los miembros del proyecto del TFG del año pasado, debemos contemplar los siguientes hitos en los que se dividirá nuestra aplicación.

- Transmisión de vídeo y audio a través de Wifi Aware (modo emisor).
- Recepción de vídeo y audio (modo receptor).
- Transmisión de vídeo y audio recibido de otro dispositivo hacia un tercer dispositivo (modo dual).
- Transmisión de vídeo y audio recibido de otro dispositivo hacia múltiples dispositivos (*router* simple).
- Recepción y transmisión de múltiples flujos de vídeo a múltiples dispositivos (*router* completo).
- Permitir la visualización de los *streams*.
- Guardar los *streams* en la memoria del dispositivo, para una posterior visualización.
- Emisión del vídeo y audio a un punto de acceso Wifi o a una estación base de la red de telefonía móvil.

5.1. Requisitos

En cuanto a los requisitos de la aplicación debemos considerar ambos escenarios para los que puede funcionar nuestra aplicación:

- **“Humanitarian”**: Para casos de desastre y de emergencia remota en los que no es posible conectarse a la infraestructura de red por razones imprevistas.
- **“Witness”**: Enfocado en denunciar violaciones o actos opresivos de los derechos humanos en lugares donde no es aconsejable acceder porque podría acarrear un riesgo para la seguridad de la persona o un tercero impide el acceso a la infraestructura de red. En este escenario la privacidad del usuario es esencial para su seguridad. No se pueden transmitir metadatos que permitan identificar al creador del *stream*.

Para definir los requisitos vamos a utilizar una metodología ágil basada en historias de usuario. Para centrarnos en todas las funcionalidades comunes que puede tener nuestra aplicación en ambos escenarios, vamos a dividirlos en los siguientes apartados:

- **Inicio de la aplicación:**
  Como usuario, quiero poder comprobar la disponibilidad de Wifi Aware en mi dispositivo móvil para poder usar la aplicación.
  Como usuario quiero conceder los permisos que sean necesarios para poder utilizar la aplicación.
- **Transmisión del *stream*:**
  Como usuario, quiero abrir el micrófono y la cámara del dispositivo para iniciar una retransmisión de vídeo y audio
  Como usuario, quiero iniciar el *stream* para difundirlo por la red ad-hoc de Wifi Aware.
- **Nodo de transito:**
  Como usuario, quiero que los *streams* recibidos se reenvíen a otros nodos participantes
- **Visualizar un *stream* que se ha recibido:**
  Como usuario, quiero ver un listado de los flujos de vídeo y audio disponibles que he recibido de otros usuarios.
  Como usuario, quiero abrir un *stream* para proceder a visualizarlo.
- **Guardar el *stream* capturado en el dispositivo:**
  Como usuario, quiero guardar los *streams* que he realizado, si lo considero oportuno, para tener una copia de ellos en el dispositivo móvil.
- **Guardar el *stream* recibido de otro dispositivo:**
  Como usuario, quiero guardar los *streams* que he recibido, si lo considero oportuno, para tener una copia de ellos en el dispositivo móvil.

- ***Streams* guardados:**
  Como usuario, quiero ver un listado de los *streams* que he guardado.
  Como usuario, quiero seleccionar un *stream* de los que se han guardado para poder visualizarlo.
  Como usuario quiero eliminar definitivamente un *stream* de los que se han guardado.

- **Subir el *stream* a internet:**
  Como usuario con conexión a internet, quiero subir los *streams* que estoy recibiendo a internet.
  Como usuario con conexión a internet, quiero subir los *streams* que he guardado en mi dispositivo móvil a internet.

A continuación mostramos los requisitos funcionales que caracterizan a cada escenario:

1. **"Humanitarian":**
   Como usuario, quiero pixelar las caras para proteger la identidad de las personas que estoy grabando.
   Como usuario, quiero seleccionar un destino para enviar el *stream* a un dispositivo concreto.

2. **"Witness":**
   Como usuario, quiero poder ocultar mi identidad en todo momento para evitar futuras repercusiones.

   - **Botón de pánico:**
     Como usuario, quiero eliminar los datos almacenados en la aplicación mediante la acción de un botón externo.
     Como usuario, quiero salir y eliminar de recientes la aplicación mediante la acción de un botón externo.
     Como usuario, quiero eliminar definitivamente la aplicación mediante la acción de un botón externo.

5.2. Prototipo

En cuanto al diseño de la aplicación, tras la realización de los bocetos en papel hemos creado un diseño de mayor calidad de la vista de la interfaz gráfica que mostramos en esta sección. Decidimos seguir una serie de principios que describimos a continuación para crear una experiencia de usuario satisfactoria.

En la vista inicial de la aplicación se muestran los dos escenarios de uso para la aplicación, una vez seleccionado el considerado aparece la vista principal donde se muestran los flujos de vídeo y audio disponibles, donde hemos aplicado el principio de igualdad [34] para destacar una de las funcionalidades principales de la aplicación, mostrando los *streams* que comparten tanto colores como tamaño y forma, de esta manera destacamos este contenido por encima del resto dándoles un color más vivo

y un tamaño distinto. Además hemos aplicado el principio de cierre cuando existen demasiados *streams*, estos elementos no se muestran completos y el usuario puede llegar a imaginar como son, completando la información visual que falta. Finalmente usando el principio de proximidad podemos ver tres bloques que son: la información del dispositivo, los flujos de vídeo y audio comentados anteriormente y el botón de iniciar un *stream*, donde el usuario los percibirá por separado.

En cuanto al menú de la aplicación hemos decidido aplicar el principio de dirección común debido a que los elementos construyen un patrón y flujo idéntico, estos se perciben como un conjunto y nos permite darle una jerarquía y una cohesión al contenido.

Finalmente hemos creado las vistas de grabación donde según el modo de uso seleccionado, se solicitará el nombre del *stream* o no antes de iniciar la grabación, además de las vistas de galería y ajustes que nos permiten visualizar los flujos de vídeo y audio que hemos ido guardando en el dispositivo y guardar las preferencias del usuario respectivamente.

Figura 5.1: Prototipo de la pantalla de selección del modo de uso.

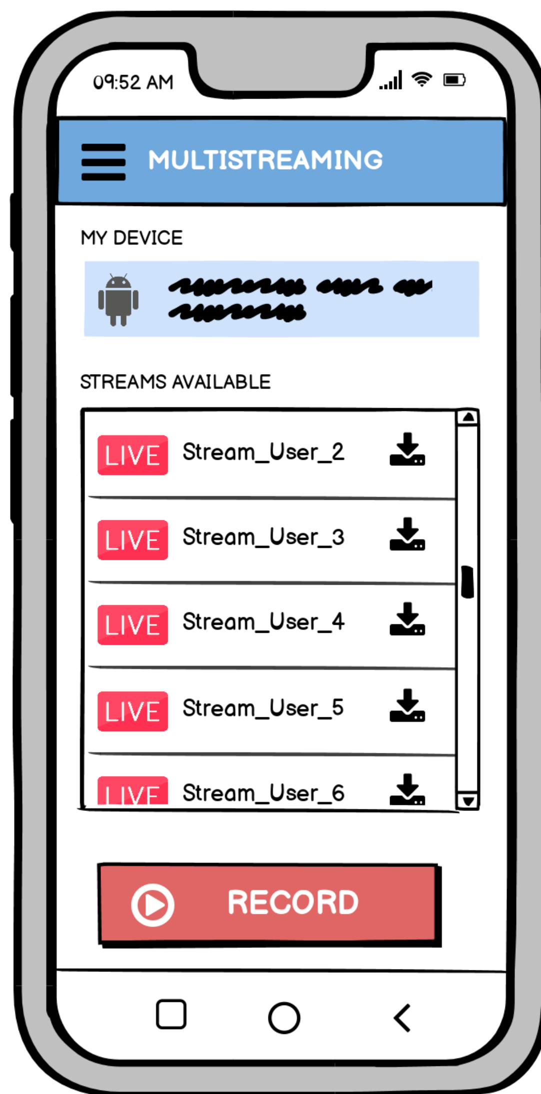


Figura 5.2: Prototipo de la pantalla de inicio.

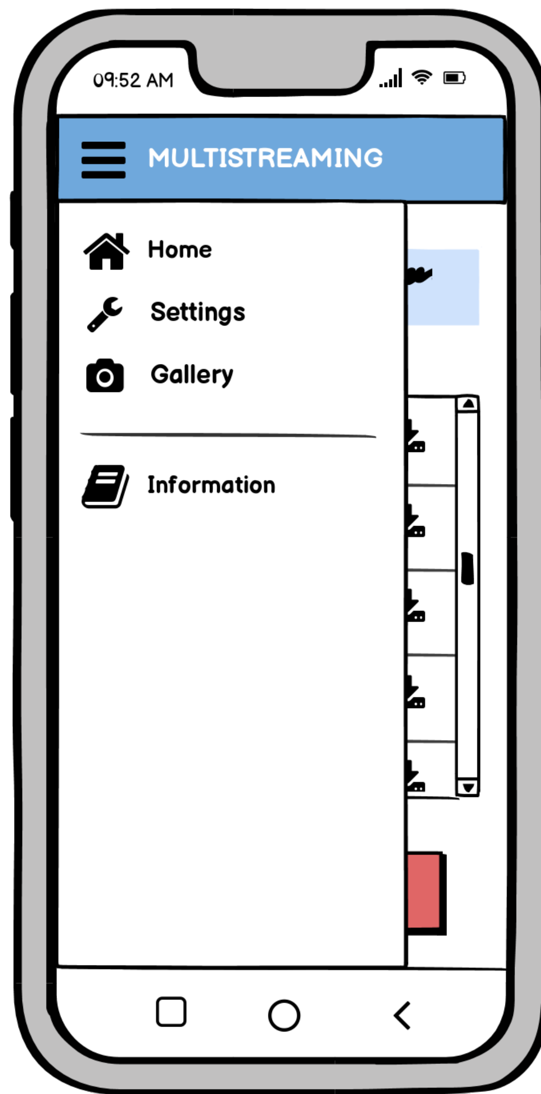


Figura 5.3: Prototipo de la pantalla de navegación.

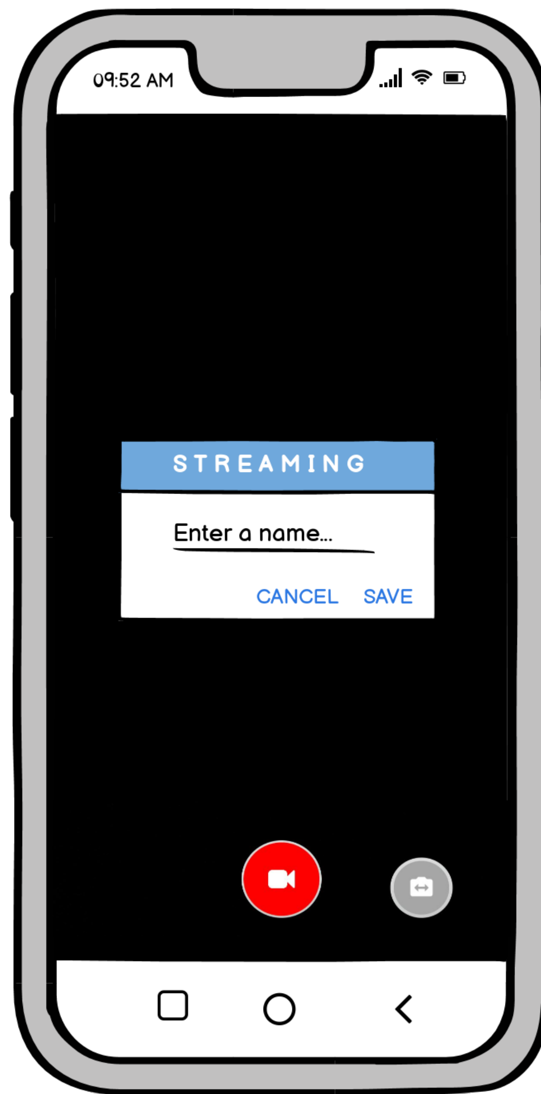


Figura 5.4: Prototipo de la pantalla de la cámara.

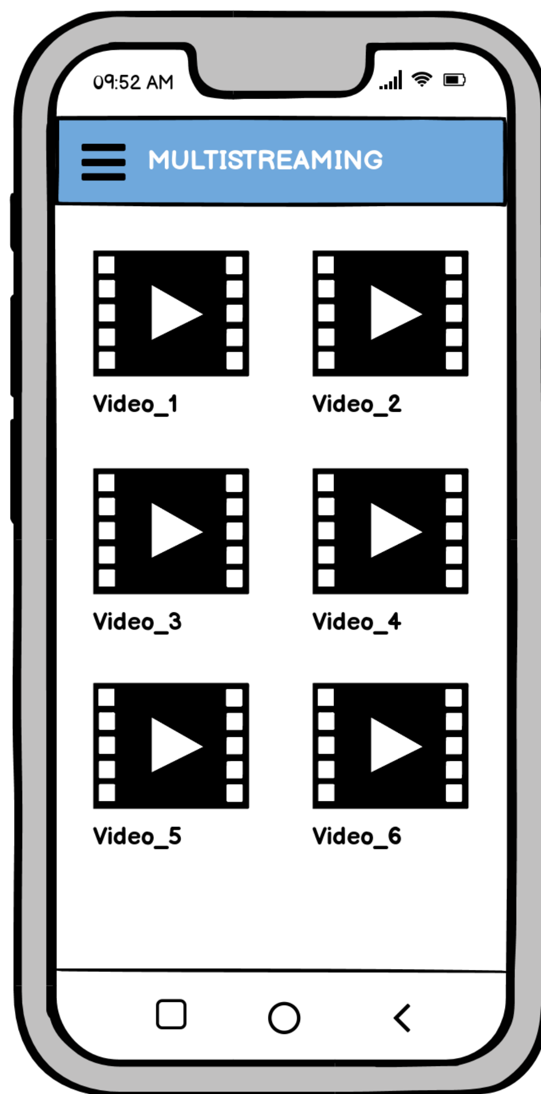


Figura 5.5: Prototipo de la pantalla de galería.

6. Diseño

Una vez definida la especificación y los requisitos de la aplicación nos pusimos a diseñar su estructura y funcionamiento. Cabe decir que, al ser un proyecto de innovación, había que investigar las tecnologías disponibles, así como la tecnología elegida de la que disponíamos poca documentación y sin ejemplos para seguir, por ese motivo el diseño ha ido evolucionando a lo largo de todo nuestro trabajo y en este punto presentamos la versión final del mismo.

6.1. Interfaz gráfica

La *Activity* es la componente gráfica principal de una aplicación Android, la cual esta formada por la parte lógica que define su comportamiento y la parte gráfica que es un XML con todos los elementos que estamos viendo de una pantalla. Otro elemento importante son los *Fragments* que son una porción de la interfaz de usuario que puede añadirse o eliminarse de la interfaz de manera independiente al resto de elementos de la actividad, y puede reutilizarse en otras actividades. En relación con la interfaz gráfica, hemos creado cinco actividades que nos permiten mostrar toda la funcionalidad especificada en detalle en la Sección 5:

1. **ModeActivity:** Es la encargada de proporcionar a la aplicación los permisos necesarios para el funcionamiento correcto de esta, además de fijar el modo de uso según los dos escenarios especificados en la Sección 5.1. Al seleccionar el modo "Humanitarian" se podrán compartir datos como el autor o titulo del *stream*, datos que, en el modo "Witness", o no son transmitidos, o como en el caso del titulo, se generan de forma aleatoria.

2. **MainActivity:** Es la actividad principal debido a que proporciona toda la lógica de intercambio de vistas del menú desplegable que esta incluye. El menú se divide en cuatro fragmentos que son los siguientes:

   - **MainFragment:** Se encarga de dar la posibilidad de iniciar el *streaming* de los datos capturados por la cámara trasera y el micrófono del dispositivo, además de mostrar la información del usuario y los *streams* disponibles para visualizarlos o guardarlos. Este fragmento se encarga de crear las sesiones y realizar la publicación y suscripción de servicios utilizando la clase `WifiAwareViewModel` explicada en detalle en la Sección 6.3.2. Al escoger el modo "Humanitarian", se podrá introducir en la parte superior el nombre del usuario para enviarlo como autor junto con su *stream*.
   - **GalleryFragment:** Se encarga de mostrar y dar la posibilidad de visualizar los *streams* guardados en el dispositivo. Este fragmento también permite eliminar los vídeos que se desee.

- **SettingFragment:** Se encarga de mostrar al usuario las preferencias de la aplicación permitiendo su edición. En ellas destacamos las del botón de pánico, que se explica en más detalle en la Sección 6.5, el cual nos permite salir y quitar la aplicación de recientes, eliminar todos los datos de esta o desinstalar toda la aplicación.
- **InfoFragment:** Se encarga de mostrar al usuario la información de uso de la aplicación.

3. **StreamActivity:** Es la actividad que se encarga de mostrar una previsualización de la cámara, así como un botón para iniciar y finalizar la captura de los datos de la cámara y micrófono.

4. **ViewStreamActivity:** Esta actividad se encarga de visualizar un *stream* usando libVLC. Es imprescindible para mostrar tanto un *stream* que se está recibiendo como los que están guardados en el dispositivo.

5. **ExitActivity:** Esta actividad se encarga de salir de la aplicación y es utilizada únicamente por el botón de pánico.

6.2. Gestión de las conexiones RTSP, RTP y RTCP

Para gestionar las conexiones en el servidor, utilizamos la API de Java NIO que utiliza `ServerSocketChannel` y `SocketChannel`. Estas clases nos permiten tener conexiones no bloqueantes para procesar los datos. Además, utilizamos la clase Java NIO, `Selector`, para gestionar las peticiones de los clientes en un solo *thread*, al contrario de otras metodologías que proporcionan un *thread* por cliente para gestionar su comunicación.

Debido al funcionamiento de Wifi Aware en relación con la creación de las conexiones, descrito en la Sección 6.3.1, se necesita utilizar un *socket* o canal que procese eventos de tipo “accept” por cada pareja de dispositivos, es decir, un `ServerSocketChannel` por cada dispositivo cliente.

A continuación vamos a explicar el servidor RTSP que implementa libstreaming y que fue modificado por los alumnos de 2018/19 creando las siguientes clases.

- La clase `RTSPServerSelector` se encarga de crear los `ServerSocketChannel` y de registrarlos en el `Selector` para recibir eventos de tipo “accept”. De esta forma cuando un cliente se quiere conectar con el servidor, se creará un `SocketChannel` y se registrará en el `Selector` para recibir eventos de tipo “read”.
- La clase `RTSPServerWorker` encargada de tratar los datos del protocolo RTSP enviados por un cliente al servidor. En esta clase se utiliza un *thread* paralelo para no añadir retardos en las conexiones e ir modificando el estado del servidor según las peticiones que se van realizando en la comunicación RTSP. Cabe destacar que hemos tenido que modificar el modo reproducir sobre un

stream con origen en el propio dispositivo, lo cuál se extiende en detalle en la Sección 7.1.

En cuanto al cliente RTSP que nos proporciona libstreaming, solo puede utilizar la modalidad publicar, ver la Sección 3.2.3, sobre un *stream* con origen en el mismo dispositivo, por lo que hemos tenido que ampliarlo para que sea capaz de publicar un *stream* procedente de otro dispositivo. En cuanto a la modalidad reproducir, que no ha sido elegida para la comunicación RTSP entre dispositivos de este proyecto, decisión que se explica en el apéndice A.2, tampoco está implementada por libstreaming ni por los alumnos de 2018/19 pero sera necesaria para la visualización y/o guardado de los *streams*. En la Sección 7.1 se explicará en detalle que modalidades están implementadas en el código del que partimos así como la adición/modificación de estas.

Los `ServerSocketChannel` y los `SocketChannel` se crean de una forma especial para usarlos sobre Wifi Aware. Al llamar al método *addNewConnection* del servidor RTSP, con la `DiscoverySession` y el `PeerHandle` que proporciona la API de Wifi Aware para identificar al dispositivo del otro extremo, se creará un `ServerSocketChannel` para atenderle. Esta forma de conexión está descrita en la Sección 6.3.1. El diagrama UML de las clases descritas anteriormente se puede observar en la Figura 6.1.

Utilizamos distintas clases como la `ReceiveSession` y `RebroadcastSession` que nos proporcionaron los alumnos de 2018/19 y hemos tenido que adaptarlas a IPv6, además de modificar el objeto `Session` que nos proporciona libstreaming para que pueda utilizarse de forma múltiple con varios destinos. Estas clases se utilizan para representar y controlar los envíos de un flujo multimedia basado en RTP y RTCP. Dependiendo del origen y el destino utilizamos una u otra:

- `Session`: Destinada al envío del flujo de vídeo y audio generado por el dispositivo.
- `ReceiveSession`: Destinada a la recepción de un flujo de vídeo y audio generado por otro dispositivo.
- `RebroadcastSession`: Destinada al reenvió de un flujo de vídeo y audio generado por otro dispositivo.

La `ReceiveSession` ofrece el servicio de usar los servidores UDP mostrados en el diagrama de la Figura 6.1 para el reenvío de los flujos recibidos. Cuando se crea y configura, recibe en sus *sockets* el flujo multimedia y, mediante los servidores UDP que utiliza, proporciona métodos para añadir *sockets* y reenviarles el flujo. La `RebroadcastSession` hace uso de este servicio.

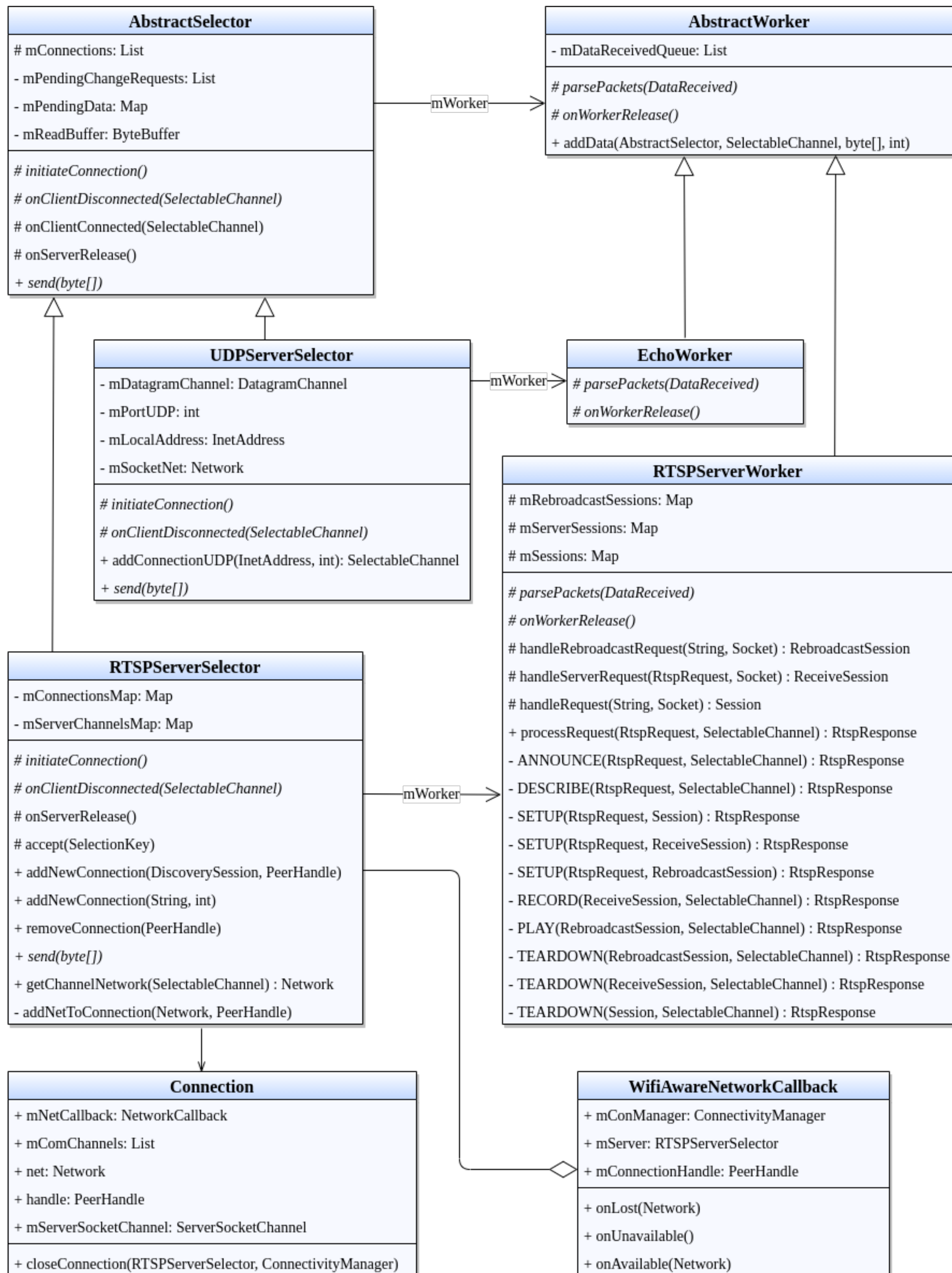


Figura 6.1: Diseño comunicación RTSP, RTP y RTCP

6.3. Funcionamiento de Wifi Aware

Wifi Aware es una tecnología que permite la conectividad device-to-device entre dispositivos móviles sin la necesidad de conexión a una infraestructura de red a través de un punto de acceso wifi o una estación base de telefonía móvil. La tecnología se encarga de descubrir automáticamente dispositivos y servicios de forma continua en segundo plano. Los dispositivos pueden estimar de forma precisa la distancia con respecto a otros y realizar acciones automáticas en base a esa estimación, como un intercambio de datos basado en el protocolo IP.

Wifi Aware distingue a los dispositivos que publican un servicio, los *publishers*, y los que se suscriben a uno, los *subscribers*. Un nodo hace "publish" para anunciar un servicio y "subscribe" para encontrar un servicio. Una vez establecida la conexión se pueden intercambiar mensajes usando la API de Wifi Aware o establecer una conexión basada en IPv6. Además, un dispositivo puede ser *publisher* y *subscriber* a la vez, lo que permite asociaciones entre dispositivos más complejas.

Para conectar dispositivos entre sí, el servicio en segundo plano de Wifi Aware pasa por 3 etapas que describimos a continuación y podemos ver en la Figura 6.2:

1. **Descubrimiento de dispositivos vecinos:** Un dispositivo con Wifi Aware descubre a otros e intercambian información entre ellos.
2. **Sincronización:** El dispositivo crea o se une a un grupo de otros dispositivos para sincronizar el envío de mensajes entre ellos. La comunicación se fija a periodos de tiempo y canales específicos para reducir el consumo energético.
3. **Descubrimiento de servicios:** Una aplicación en el dispositivo crea una sesión de *subscriber* para encontrar servicios y/o una sesión de *publisher* para publicar su presencia y servicios. Se puede realizar de dos maneras que vamos a explicar a continuación:

    - "Active subscribe - solicited publish": El *subscriber* envía mensajes de búsqueda de servicio, mientras el *publisher* escucha los mensajes y contesta a los que buscan un servicio, que *matchea* con el servicio que él ofrece. Esto se puede entender mejor mediante la siguiente analogía:

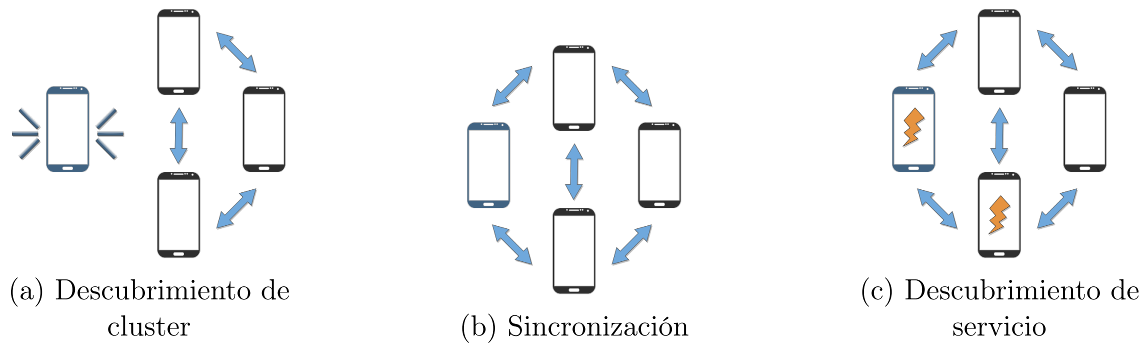
        Subscriber: "¿Hay un médico en la sala? "

        Publisher: "Sí, aquí estoy"

    - "Passive subscribe - unsolicited publish": El *publisher* envía mensajes de anuncio de servicio, mientras el *subscriber* escucha los mensajes y contesta a los que anuncian un servicio que *matchea* con el servicio que él busca. Esto se puede entender mejor mediante la siguiente analogía:

        Publisher: "El chatarrero, el chatarrero..."

        Subscriber: "Para, para, tengo algo para tí"



(a) Descubrimiento de cluster (b) Sincronización (c) Descubrimiento de servicio

Figura 6.2: Proceso de conexión de dispositivos con Wifi Aware. [7]

6.3.1. Creación de conexiones con Wifi Aware

Para crear una conexión cliente-servidor entre dispositivos móviles a través de Wifi Aware hemos creado una sesión *publisher* de Wifi Aware en el dispositivo donde ejecutará un servidor RTSP (es decir, este dispositivo ofrece el servicio: servidor-RTSP-en-modalidad-publicar listo para recibir *streams*) y una sesión *subscriber* Wifi Aware en el dispositivo donde ejecutará un cliente RTSP (es decir, este dispositivo quiere usar el servicio servidor-RTSP-en-modalidad-publicar que ofrece el *publisher* Wifi Aware para poder enviar *streams* a este servidor RTSP), de esta forma cuando se realice el proceso de descubrimiento el cliente puede enviar el stream al servidor con la modalidad publicar, la cual se ha explicado en detalle en la Sección 3.2.3. El proceso que hay que seguir en detalle para la creación de la conexión se explica en apéndice A.3.

6.3.2. Gestión de Wifi Aware

Para controlar toda la conectividad en relación con Wifi Aware, hemos creado una clase llamada `WifiAwareViewModel`. Esta clase nos ofrece principalmente una interfaz para publicar y suscribirnos a servicios de Wifi Aware. También, como Wifi Aware puede cambiar su disponibilidad en función de la activación de la ubicación y el Wifi, incluimos un *callback* para notificar a la GUI u otras clases de estos cambios.

La clase `WifiAwareViewModel` se encarga además de gestionar las conexiones entre parejas de dispositivos de manera secuencial, ya que se encontró un problema al no llevar un control sobre esto. Esta gestión secuencial de las conexiones se extenderá en la Sección 7.2. Los mensajes que se intercambian los *publishers* y *subscribers* a la hora de publicar y suscribirse a servicios se describe en detalle en el apéndice B.4.

Para poder realizar el envió múltiple, o multicast, es necesario crear un servidor RTSP en el *publisher* y crear clientes RTSP en los *subscribers* según se conecten a otros *publishers*. Una vez el *subscriber* descubre un *publisher*, iniciamos todo el proceso de conexión especifico de Wifi Aware, descrito en detalle en el apéndice A.3, cuya lógica está implementada dentro del servidor y cliente RTSP.

6.4. Diseño orientado al Multihopping

En esta sección se explica el diseño que hemos implementado para conseguir que un dispositivo sea capaz de recibir un *stream* y reenviarlo a todos los dispositivos de los que está suscrito, de esta manera conseguimos que el *stream* circule por la red ad-hoc.

Para conseguir el multihopping, consideramos necesario que cada dispositivo tenga una sesión de *publisher* y *subscriber* de Wifi Aware como se observa en la Figura 6.3.

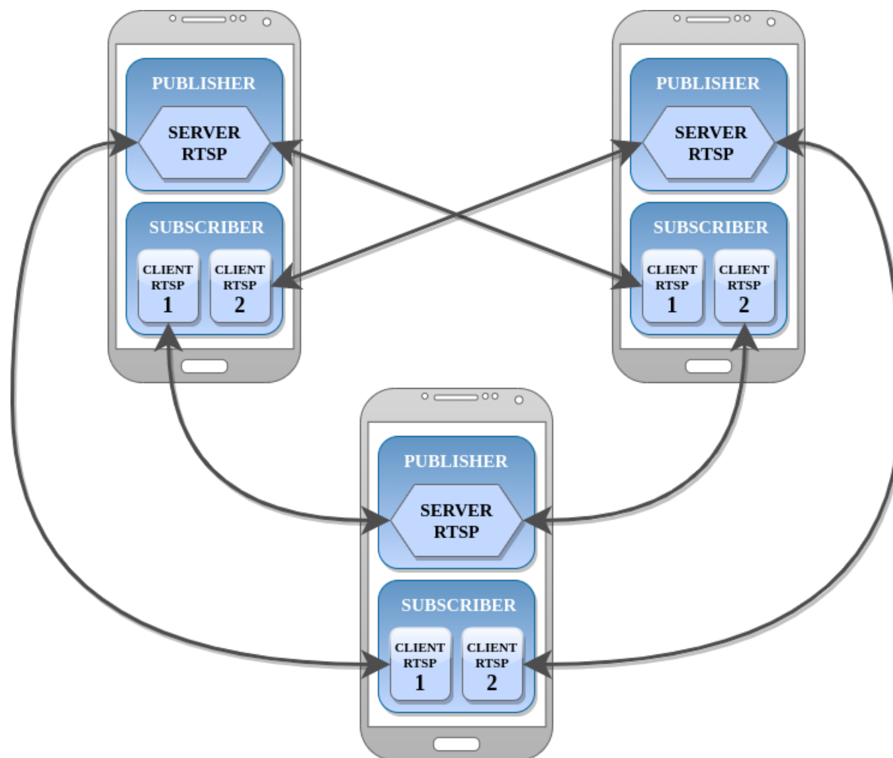


Figura 6.3: Diagrama multihopping.

Siguiendo este diseño, el dispositivo funciona como servidor RTSP y cliente RTSP a la vez para poder transmitir el *stream* en la modalidad publicar (Sección 6.3.1). La sesión de *publisher* tiene asociado un servidor RTSP. Cuando una sesión de *subscriber* de otro dispositivo descubre la de *publisher*, intentará crear una conexión de Wifi Aware. Si esta conexión se establece correctamente, se crea un cliente RTSP para comunicarse con el servidor del *publisher*.

6.4.1. Identificación de los streams en la red distribuida

En las redes distribuidas como las MANETs, la identificación de recursos compartidos no se puede resolver de la misma forma que en una red centralizada. En este tipo de redes no tenemos un punto centralizado donde publicar nuestros recursos y

donde generar un identificador único teniendo en cuenta los que ya existan en la red. En nuestro caso para identificar cada *stream* utilizamos un UUID, *Universal Unique Identifier*, que es un número de 128 bits generado de tal forma que la probabilidad de que se creen dos idénticos es tan baja que es despreciable.

Existen actualmente varias versiones de UUID. En la versión 1 y 2 se utiliza la dirección MAC del dispositivo y la fecha y hora, en la versión 3 y 5 se generan a partir de aplicar una función *hash* a un nombre, MD5 y SHA1 respectivamente, y en la versión 4 se genera de forma aleatoria. La versión que utilizamos es la 4, ya que la 1 y la 2 podrían comprometer la confidencialidad del modo "Witness" (Sección 5.1), al poder extraer la dirección MAC, la fecha y la hora para identificar al creador del *stream*. Este identificador lo utilizamos en las URI del protocolo RTSP para que el cliente o el servidor identifiquen el *stream* en sus mensajes.

6.4.2. Gestión centralizada de los streams en el dispositivo

Para conseguir implementar el reenvío de los *streams* siguiendo nuestra infraestructura de red, es conveniente agrupar la gestión de los *streams* recibidos por el servidor RTSP de un dispositivo, más el *stream* generado por la cámara y micrófono del propio dispositivo, en su caso, en una sola clase, con el fin de facilitar el acceso a ellos por parte de los distintos clientes RTSP del dispositivo. Para ello creamos una clase llamada `StreamingRecord`, que se va a usar conforme al patrón *Singleton* y contiene una representación de los *streams* con todo lo necesario para reenviarlos a otro dispositivo o visualizarlos. En ella se almacenan los identificadores únicos de cada uno de ellos junto con, en el caso de los *streams* recibidos una `ReceiveSession` y en el caso del *stream* creado en el dispositivo un objeto `SessionBuilder` para crear objetos `Session`.

En la clase `StreamingRecordObserver` están definidos *callbacks* para notificar de los cambios de disponibilidad en los *streams*. Al implementar esta interfaz, los clientes RTSP, serán notificados de la existencia de nuevos *streams* que va recibiendo el servidor y podrán reenviarlos. Podemos observar el diagrama UML en la Figura 6.4.

6.5. Aplicación del botón de pánico

En cuanto a la aplicación del botón de pánico, realizamos un *fork* de "Ripple", un proyecto de un grupo de desarrolladores (Guardian Project) que crean aplicaciones móviles seguras y de código fuente abierto. Ripple es un "botón de pánico" que envía mensajes de activación a cualquier aplicación que sea un "respondedor de pánico" las cuales pueden reaccionar a estos eventos.

El botón de pánico es de gran utilidad en el escenario de uso *witness*, donde en caso de emergencia como la captura del dispositivo por parte del régimen opresivo, se permite al usuario deshacerse de cualquier rastro que pueda incriminarlo, respaldando así su seguridad.

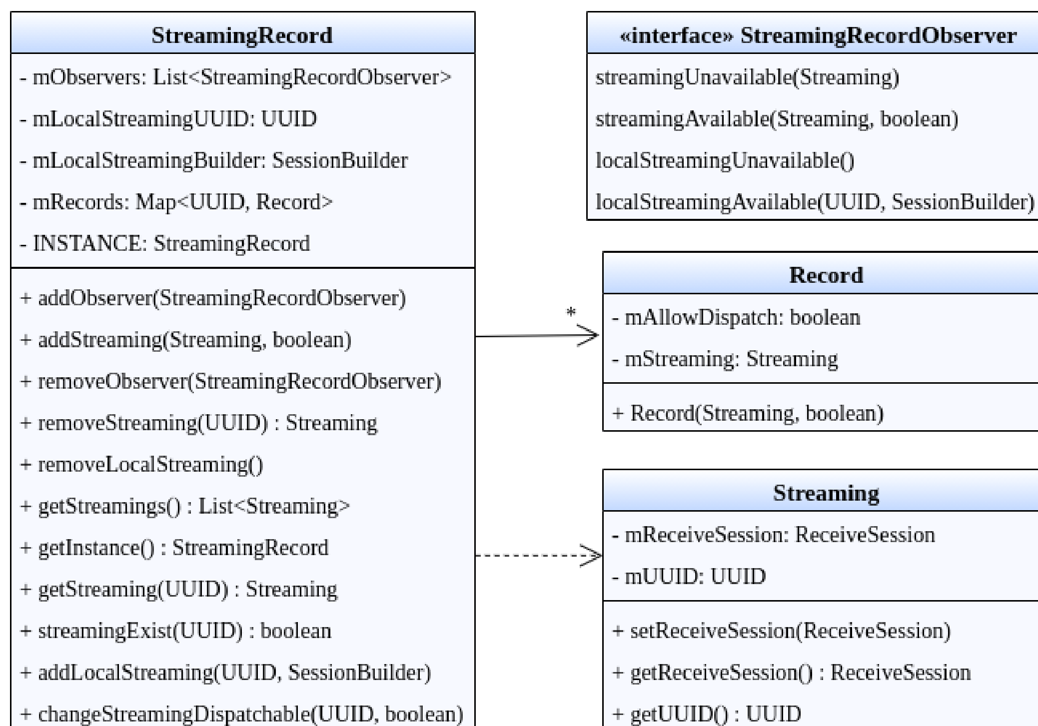


Figura 6.4: Diagrama para la gestión de los streams.

Nosotros hemos adaptado la aplicación para este proyecto, añadiendo una interfaz más simple e intuitiva como se puede observar en la Figura 6.5 sin necesidad de configuración previa, ya que esta se realiza en la aplicación respondedora de eventos, que en nuestro caso es "Multistreaming". Los diferentes eventos a los que responde nuestra aplicación cuando se envía el mensaje de activación desde el botón de pánico son los siguientes:

- Salir y eliminar la aplicación de recientes.
- Eliminar todos los datos almacenados.
- Desinstalar toda la aplicación.

La configuración de estas acciones se realiza a través del menú de ajustes de "Multistreaming" mediante unos botones de activación.

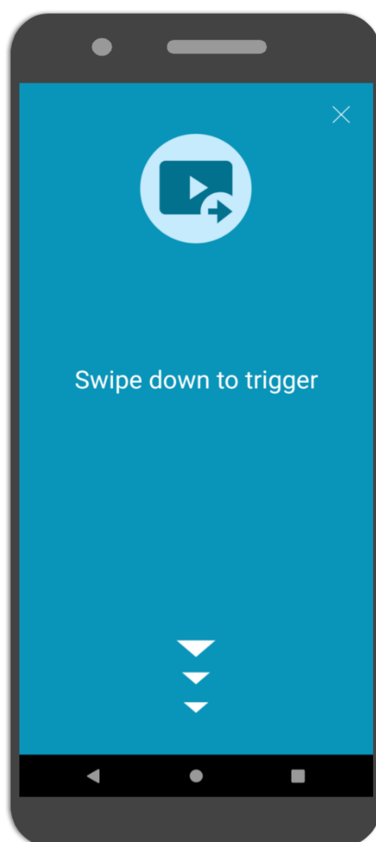


Figura 6.5: Botón de pánico.

7. Implementación

En esta sección vamos a presentar la implementación de nuestro proyecto, dividida en capas según los protocolos usados. En esta etapa hemos investigado y afrontado numerosas dificultades y decisiones, las cuales están explicadas en detalle en el apéndice B, para simplificar y facilitar la comprensión de este apartado.

7.1. RTSP, RTP y libstreaming

Antes de empezar con la implementación, tuvimos que investigar tanto el código de los alumnos de años anteriores, como libstreaming y su posible adaptación a Wifi Aware. En el apéndice A.1, desarrollamos toda nuestra investigación y en el apéndice A.1.3 las razones por las que tomamos algunas de las siguientes decisiones. La decisión más importante que tomamos fue la de **adaptar libstreaming a Wifi Aware.**

Como vimos en la Sección 3.2.3, **RTSP tiene dos modalidades de transmisión. Nosotros nos decantamos por utilizar únicamente la modalidad publicar para transmitir los streams entre los dispositivos.** Las razones y sus implicaciones están expuestas en el apéndice A.2.

Como esta definido en la Sección 5.1 de especificación, nuestro objetivo es que cada dispositivo pueda:

- Crear un *stream* y transmitirlo a múltiples dispositivos, con la capacidad de visualizarlo y/o guardarlo mientras se crea.
- Recibir *streams* de múltiples dispositivos y reenviarlos a múltiples dispositivos, con la capacidad de visualizarlo y/o guardarlo mientras se recibe/reenvía.

Para la implementación de esta capa, **partimos del código de libstreaming con los cambios que hicieron los alumnos del proyecto de 2018/19**.

7.1.1. Análisis de libstreaming y orientación de la implementación

7.1.1.1. Estado de libstreaming

En primer lugar, libstreaming proporciona la implementación de un cliente RTSP en modalidad publicar y un servidor RTSP en modalidad reproducir, en ambos casos aplicando la modalidad solamente a *streams* con origen en el mismo dispositivo que hace uso de la biblioteca, es decir, en el cliente RTSP solo se puede publicar un *stream*

y al servidor RTSP solo se le puede solicitar un *stream* que se este capturando en el mismo dispositivo.

La biblioteca implementa el cliente RTSP con la intención de ser usado para transmitir el *stream* generado a partir de la cámara y el micrófono del dispositivo del cliente a un servidor RTSP que implemente la modalidad publicar, como Wowza Media Server. Asimismo, implementa el servidor RTSP con la intención de ser usado para transmitir el *stream* generado a partir de la cámara y el micrófono del dispositivo del servidor a un cliente RTSP que implemente la modalidad reproducir, como libVLC, y que solicite su envío al servidor.

Según se cita en la pagina de Github de libstreaming [2]:

- «With the RTSP client: if you want to stream to a Wowza Media Server, it's the way to go. The example 3 illustrates that use case.»
- «With the RTSP server: in that case the phone will act as a RTSP server and wait for a RTSP client to request a stream. This use case is illustated in the example 1.»

7.1.1.2. Cambios en libstreaming por los alumnos de 2018/19

Los alumnos de 2018/19, orientaron parte del proyecto a adaptar libstreaming para que su cliente y servidor RTSP pudieran trabajar juntos. Para ello, añadieron en el servidor RTSP la implementación de la modalidad publicar, para poder recibir *streams* de otros dispositivos, y modificaron la modalidad reproducir para que pudiera reenviar los *streams* recibidos de otros dispositivos con la modalidad publicar. Esta implementación de la modalidad reproducir, nos permite visualizar y/o guardar los *streams* recibidos en el dispositivo mediante el uso del cliente RTSP de libVLC, como se vera en detalle más adelante.

La implementación de libstreaming de la modalidad reproducir en el servidor RTSP, para enviar un *stream* con origen en el mismo dispositivo, decidieron no adaptarla a los cambios que hicieron en el servidor, sobretodo en cuanto al uso de la URI para referirse a un *stream*, y quedo en desuso. Nosotros necesitamos esta funcionalidad para poder guardar el *stream* del usuario mientras se captura en el mismo dispositivo.

Además de estos cambios, añadieron funcionalidad para poder visualizar un *stream* haciendo uso de libVLC. Una vez conseguían la URI, basada en IPv4, que identificaba el *stream* RTSP que querían visualizar dentro del servidor RTSP, creaban una instancia de la biblioteca pasándole la URI, y el cliente RTSP de libVLC, que implementa la modalidad reproducir, le solicitaba el *stream* al servidor RTSP de libstreaming y lo presentaba en la pantalla.

Cabe destacar, que en la implementación de libstreaming el código para configurar y controlar la cámara esta mezclado con el código para transmitir *streams*, lo

que según se investigo por los alumnos de 2018/19, imposibilita el envió múltiple (*multicast*) del *stream*. Ellos empezaron la implementación de una solución pero no lograron resolver este problema. Nosotros partimos de un código con el que podemos enviar desde el cliente RTSP un solo *stream* a un único servidor RTSP y podemos recibir solamente un *stream* por cliente en el servidor RTSP.

Otro punto a tener en cuenta, es que los alumnos de 2018/19 buscaron implementar el *multihopping* con su objetivo de conseguir que el servidor y cliente RTSP de la biblioteca pudieran trabajar juntos. Entendiendo por salto, el paso de la información de un nodo de la red a otro [35], en nuestro caso el paso de un stream de un dispositivo a otro, los alumnos de 2018/19, llegaron a implementar el primer salto, del cliente RTSP al servidor RTSP de otro dispositivo mediante la modalidad publicar y un segundo salto del servidor RTSP a un cliente RTSP mediante la modalidad reproducir para visualizarlo. Quedó pendiente por implementar la capacidad de que un dispositivo pudiera recibir y enviar *streams* a la vez, es decir realizar un segundo salto con la modalidad publicar a otro dispositivo, y que el cliente RTSP fuera capaz de reenviar un *stream* con origen en otro dispositivo, en ambos casos utilizando únicamente la modalidad RTSP publicar ya que es la modalidad que hemos elegido para realizar los saltos entre dispositivos.

7.1.1.3. Conclusiones

Recapitulando las secciones anteriores:

1. Libstreaming implementa un servidor RTSP con la modalidad reproducir y un cliente RTSP con la modalidad publicar, en ambos casos solo para *streams* con origen en el mismo dispositivo.

2. Los alumnos de 2018/19 implementaron la modalidad publicar en el servidor RTSP, además añadieron la modalidad reproducir en el servidor para *streams* con origen en otro dispositivo y dejaron en desuso y por adaptar la modalidad reproducir en el servidor para *streams* con origen en el mismo dispositivo.

3. En cuanto a *multihopping*, los alumnos de 2018/19 solo lograron el primer salto. Para completarlo es necesario implementar en el cliente RTSP la modalidad publicar para *streams* con origen en otro dispositivo e idear una forma de que un dispositivo pueda recibir y enviar *streams* a la vez.

4. En este código no es posible el envió múltiple, el cliente RTSP solo puede enviar un *stream* a un único servidor RTSP, y el servidor RTSP solo puede recibir un único *stream* por cliente.

5. Los alumnos de 2018/19 implementaron la capacidad de visualizar un *stream* recibido en el dispositivo mediante el uso del cliente RTSP con la modalidad reproducir de libVLC.

6. El código para configurar y controlar la cámara esta mezclado con el código que se usa para transmitir un *stream*, lo que impide enviar el *stream* a más de

un destino por conflictos en la cámara y micrófono. Los alumnos de 2018/19 avanzaron en este problema pero aún quedó pendiente por resolver.

Para más detalle sobre la funcionalidad implementada en libstreaming y las adaptaciones de los alumnos de 2018/19 referirse al apéndice B.1.

Teniendo en cuenta la investigación del apéndice A.1, **consideramos necesario implementar la siguiente funcionalidad para cumplir con la especificación:**

1. Adaptar libstreaming a IPv6. Wifi Aware no trabaja sobre IPv4.
2. Adaptar libstreaming para poder enviar el flujo de la cámara y micrófono a más de un destino. Este problema se consideraba solucionado por los alumnos de 2018/19 pero, como se verá más adelante, no es así.
3. Implementar un sistema para que los *streams* que reciba el dispositivo y el generado por su propia cámara/micrófono se puedan reenviar, visualizar y/o guardar.
4. Recuperar en el servidor RTSP la modalidad reproducir para *streams* con **origen en el propio dispositivo**.
5. Implementar en el cliente RTSP la modalidad publicar para reenviar *streams* con origen en **otro** dispositivo.
6. Adaptar el servidor para que pueda negociar y recibir más de un *stream* por cliente.
7. Adaptar el cliente para que envíe más de un *stream* al servidor y reaccione a los cambios de disponibilidad de los *streams* recibidos.

En los siguientes puntos iremos desarrollando la implementación de cada uno.

7.1.2. Adaptación de libstreaming a IPv6

Para poder utilizar libstreaming sobre Wifi Aware, necesitábamos adaptar la biblioteca para poder utilizar IPv6, ya que **Wifi Aware solamente funciona sobre esta versión de IP.** Los alumnos de 2020/21 intentaron adaptar la biblioteca pero no lo lograron (apéndice A.1.2).

El problema que tuvieron los alumnos de 2020/21 era un error de asociación de los *sockets* al uso de la red de Wifi Aware. La correcta asociación de los *sockets* viene descrita en el apéndice A.3. Nosotros adaptamos libstreaming para seguir esta metodología en la creación de los *sockets* RTSP, RTP y RTCP entre los dispositivos. Esto implicó además cambiar el funcionamiento del servidor, en el que inicialmente se escuchaba en una dirección y puerto para las nuevas peticiones de conexión. Debido al funcionamiento de Wifi Aware, que aísla los *sockets* y la resolución de direcciones IPv6 para cada pareja de dispositivos, tuvimos que hacer que, una vez

el dispositivo conectara con otro a nivel de Wifi Aware, se creara un nuevo *socket* en el servidor para escuchar peticiones de conexión de ese dispositivo únicamente, en vez de la política habitual de utilizar un único *socket* para escuchar las peticiones de conexión de todos los clientes. Para más detalle sobre estos cambios referirse al apéndice B.3.

En cuanto al protocolo RTSP, utiliza un protocolo adicional internamente para describir el *stream* que se va a enviar. Hay varias opciones para esto, pero en libstreaming solo están implementadas las descripciones con SDP, y estas solo con IPv4. **Tuvimos que examinar el estándar SDP [36] y extender el protocolo a IPv6.** Entrando en detalle, en libstreaming hay un objeto llamado `Session` que sirve para enviar el flujo de la cámara y micrófono locales a nivel de RTP y RTCP. Este objeto es el que genera la descripción SDP del flujo multimedia y por tanto el que adaptamos a IPv6.

Como pudimos ver con VideoLan media player, **libVLC tampoco acepta visualización y/o guardado del *stream* por RTSP sobre IPv6.** Intentamos buscar una biblioteca que lo aceptara, pero, como comentamos en el apéndice A.1, IPv6 no está muy extendido de momento en RTSP. Como solución, **modificamos el servidor para que escuchara peticiones de reproducción de *stream* en la dirección interna IPv4 de *loopback* 127.0.0.1.** Tras esto cuando el usuario quiera visualizar y/o guardar el *stream*, creamos una instancia de libVLC y la configuramos para que, utilizando la modalidad reproducir de RTSP, se conecte al servidor local y le solicite el *stream* que el usuario requiera.

7.1.3. Agrupar el acceso a los flujos multimedia

Tras decidir que en el dispositivo íbamos a integrar un servidor RTSP y múltiples clientes RTSP, según la infraestructura comentada en el punto 6.4, **tuvimos que pensar en una forma de que el servidor RTSP pudiera compartir los *stream* recibidos con otras partes de la aplicación**, como la GUI, para mostrarlos, o los clientes RTSP, para reenviarlos. También había que solucionar un problema que surgió con la distribución de un *stream* generado localmente a múltiples clientes RTSP (recordamos que creamos un cliente RTSP por cada dispositivo al que queremos enviar el *stream*), ver Sección 7.1.4.

Para resolver esto, **creamos una clase para agrupar los accesos a *streams* dentro de la aplicación.** Esta clase la llamamos `StreamingRecord` y sigue el patrón *singleton*. **Creamos un objeto `Streaming` pensado para contener lo necesario para generar un envío, del *stream* que representa, a otro dispositivo.** Además, está pensada para incluir información adicional para gestionar criterios de *routing*, como el número de saltos por los que pasa el *stream* para llegar al dispositivo o la distancia total al origen.

Por falta de tiempo y al considerar más prioritarios otros avances, la información adicional para *routing* la dejamos para una futura mejora. La distancia a un

dispositivo se puede obtener utilizando la API de Wifi Aware y tanto la distancia como el numero de saltos por los que ha pasado el *stream* se podrían transmitir en la descripción SDP del *stream*, que solicite el servidor RTSP del siguiente salto, para que decida mejor si, la recepción del *stream* le interesa más a través de un nodo u otro. También, por la misma razón, tuvimos que separar del objeto `Streaming` la capacidad de representar un *stream* generado en el dispositivo. Nuestra intención era que representase tanto a los flujos recibidos como generados localmente, pero por falta de tiempo no pudimos integrar estos últimos, por lo que solo guarda como información la `ReceiveSession` comentada en el apéndice B.2.

La clase `StreamingRecord`, por tanto, conforma un almacén local que agrupa los *streams* recibidos, objetos `Streaming`, además de poder contener información para enviar el flujo local. Esta información es un `SessionBuilder` para crear los objetos `Session`, comentados en el apéndice B.2, así como el UUID del *stream* local, comentado en la Sección 6.4.1.

La clase implementa un sistema de *callbacks* mediante la interfaz `StreamingRecordObserver`. Un objeto que implemente esta interfaz como el `RtspClient` o la `StreamingActivity`, encargada de actualizar los *streams* mostrados al usuario, puede recibir *callbacks* de la clase `StreamingRecord` cuando se añade o elimina un *stream* recibido o generado.

7.1.4. Transmisión múltiple del flujo multimedia

Una vez conectados los dispositivos entre sí, cada uno debería ser capaz de enviar el flujo de su cámara y micrófono y el flujo que reciba de otros a los dispositivos alcanzables.

El reenvió de un flujo recibido a múltiples destinos, fue implementado parcialmente por los alumnos del año 2018/19. Ellos crearon dos clases, `ReceiveSession` y `RebroadcastSession`. La primera sirve para encapsular el flujo multimedia recibido y la segunda para encapsular el flujo multimedia reenviado. Estas clases trabajan con los flujos a nivel de RTP y RTCP. Sin embargo, la parte de la implementación del reenvío del flujos que concierne a RTSP es incompleta; falta implementar el envío de flujos (con un cliente RTSP en modalidad publicar) que no tienen su origen en el mismo dispositivo. Además, **falta extender el servidor y cliente RTSP para que puedan enviar más de un flujo y a más de un destino.** Estos cambios se verán en detalle al final de esta sección.

Estos objetos generan una descripción en SDP para describir el flujo que encapsulan. **Tuvimos que adaptar esta descripción a IPv6,** como con la clase `Session` de libstreaming, comentado en la Sección 7.1.2, que es utilizado para enviar el flujo local del dispositivo. Adicionalmente, **tuvimos que adaptar estas clases a Wifi Aware**, ya que la creación de sockets es especial en este tipo de red. **Los detalles de esta adaptación están explicados en detalle en el apéndice B.2.**

En cuanto al envío del flujo de la cámara y el micrófono locales, los alumnos del año 2018/19, hicieron adaptaciones en libstreaming para enviarlo a más de un destino, pero estas adaptaciones no pudieron llegar a probarlas. Al probarlas nosotros, constatamos que su solución no resolvía el problema por incompleta. Libstreaming utiliza el objeto `Session` para leer los datos de la cámara y micrófono y convertirlos en flujos RTP y RTCP para enviarlos. **El problema que tiene usar esta biblioteca para enviar el flujo a múltiples destinos es que más de un objeto no puede leer a la vez del hardware.**

Los alumnos del curso 2018/19 crearon dos clases, `VideoPacketizerDispatcher` y `AudioPacketizerDispatcher`, cuyo objetivo es hacer de intermediario entre los codificadores de vídeo y audio y las clases encargadas de empaquetar y enviar estos flujos, ver Sección B.1 para más detalles.

Gracias a estas clases varios objetos `Session` pueden leer simultáneamente de la cámara y el micrófono. Esto teóricamente funciona, pero otro problema más importante, que no se había considerado, es que **la configuración y el control de la cámara estaba integrado en el propio objeto utilizado para enviar el flujo.** Al hacer pruebas y crear **varios de estos objetos, intentaban configurar todos la cámara a la vez, provocando que no funcionase ninguno.** Sumado a esto, el código de configuración y control de la cámara de libstreaming estaba deprecado y sospechábamos que el problema del desenfoque de la cámara, ver Sección A.1.1 para más detalles, estaba relacionado con esto.

Como solución, **concluimos que era mejor estudiar la nueva API de la cámara, Camera2 [37], e implementar una clase que se usará con el patrón *singleton*.** Esta clase la llamamos `CameraController`. En esta clase podemos configurar una de las cámaras del dispositivo para que nos de su salida, tanto a la pantalla para visualizar lo que se quiere grabar como al `VideoPacketizerDispatcher` para repartir los datos de la cámara entre los objetos `Session` que empaquetan y envían el flujo a los distintos destinos.

Gracias a esto, además, **pudimos resolver el problema del desenfoque** e investigando **logramos mejorar la calidad del vídeo de 720p a 1080p**.

Los detalles concretos sobre la implementación de la clase `CameraController` y otros problemas resueltos en esta sección vienen desarrollados en la Sección B.2.

Aun con estos cambios, para conseguir el envío múltiple tanto de los flujos locales como de los provenientes de otros dispositivos, era necesario completar las modalidades RTSP y modificar el servidor y el cliente para poder transmitir a más de un destino.

7.1.4.1. Implementación de modalidades RTSP

En la biblioteca libstreaming, la implementación de las modalidades RTSP en el cliente y el servidor no están completas, ver la Sección B.1 para más detalles.

El servidor puede utilizar la modalidad reproducir sobre *streams* con origen en otro dispositivo, funcionalidad implementada por los alumnos de 2018/19. Esta modalidad es la que se utiliza para visualizar y/o guardar un *stream* en el dispositivo con libVLC. Por lo tanto, con esta parte de la modalidad, **solamente se pueden visualizar/guardar los *streams* que vienen de otro dispositivo. Para poder guardar el *stream* en el dispositivo en el que se graba, es suficiente con adaptar la implementación de la modalidad reproducir en el servidor** para guardarlo con libVLC. Esta parte de la modalidad reproducir estaba implementada y era usada en libstreaming, pero con los cambios que realizaron los alumnos de 2018/19, al no necesitarla, la dejaron en desuso y no la actualizaron a los cambios que realizaron en el servidor RTSP. Las adaptaciones necesarias están principalmente relacionadas con la estructura de las URI y las acciones en el servidor RTSP para encontrar el *stream* solicitado.

Gracias a la implementación del envió múltiple del flujo multimedia local, explicada al inicio de esta sección y a la de la Sección 7.1.3 y basándonos en el protocolo utilizado en la otra parte de la modalidad reproducir del servidor RTSP, conseguimos implementarlo sin problemas.

En el cliente RTSP, solo estaba implementada la modalidad publicar para enviar los *streams* creados localmente. Para poder reenviar los *streams* recibidos en el dispositivo teníamos que completar la modalidad publicar.

Al igual que con el servidor RTSP, gracias a la implementación del reenvío múltiple del flujo multimedia proveniente de otros dispositivos, explicada al inicio de la sección y a la de la Sección 7.1.3 y basándonos en el protocolo utilizado en el la otra parte de la modalidad publicar del cliente RTSP, para enviar los flujos creados localmente, completamos esta parte.

Tras haber completado estas modalidades, el protocolo RTSP estaba completo para realizar la transmisión múltiple. El dispositivo que graba utiliza la modalidad publicar para enviarlo a otro y este a su vez puede reenviarlo con la modalidad publicar y/o visualizarlo/guardarlo con la modalidad reproducir, como se menciono anteriormente, haciendo uso del cliente RTSP de libVLC a través de la dirección de loopback IPv4.

7.1.4.2. Adaptación del servidor para la transmisión múltiple

El servidor RTSP estaba prácticamente listo para la transmisión múltiple. **Añadimos el uso del `StreamingRecord`**, explicado en la Sección 7.1.3. Al procesar el método RTSP RECORD, por el que el cliente RTSP indica al servidor RTSP que va a comenzar a transmitir, comprobamos que el *stream* no haya sido registrado en el `StreamingRecord` por otro cliente y lo añadimos como *stream* disponible para el reenvío. En el caso de que ya esté en el `StreamingRecord`, respondemos al cliente con un código RTSP de rechazo del *stream*. Siguiendo este método, evitamos los problemas con los bucles directamente, al solo tener en cuenta como origen inme-

diato del *stream* al primer nodo que inició la transmisión y descartar todos aquellos que lleguen a continuación por otros caminos.

Dejamos para el trabajo futuro optimizar la recepción del *stream* por el mejor nodo. Se podrían tener en cuenta métricas mas óptimas.

7.1.4.3. Adaptación del cliente para la transmisión múltiple

Al contrario que con el servidor, tuvimos que rehacer prácticamente todo el cliente RTSP para hacer posible la transmisión múltiple. **En libstreaming, el cliente RTSP solamente negociaba con el servidor RTSP el envío del flujo de la cámara local (modalidad publicar). Nosotros necesitábamos un cliente que pudiera gestionar el envío de más de un flujo multimedia** y que una vez conectase con el servidor se mantuviera a la espera de que introdujéramos nuevos *streams* en el `StreamingRecord` para enviarlos.

Cambiamos todo el proceso del cliente, aprovechando solo el código del protocolo RTSP usado para enviar el flujo local de la cámara y micrófono. Una vez iniciado el cliente, establece conexión con el servidor y se mantiene a la espera de la llegada de *streams* al servidor RTSP del mismo dispositivo para poder mandarlos, enviando mensajes de tipo OPTIONS de RTSP cada cierto tiempo. Si no se envían estos mensajes la conexión de Wifi Aware se pierde eventualmente.

Implementamos la interfaz `StreamingRecordObserver`, para recibir *callbacks* de los cambios de disponibilidad de los *streams* del dispositivo. Cuando recibe el *callback* de *stream* disponible, negocia con el servidor su envío, y cuando deja de estar disponible, le envía un mensaje de tipo TEARDOWN para cesar su transmisión.

Entrando en detalle sobre la gestión de los envíos en el cliente:

- Para los *streams* generados localmente creamos un objeto `Session` con el `SessionBuilder` recibido en el *callback* declarado en el StreamingRecordObserver y aplicamos los métodos del protocolo RTSP que tenía la implementación original de libstreaming.
- Para el reenvío de los *streams* recibidos en el dispositivo, creamos un objeto `RebroadcastSession` a partir de la `ReceiveSession` que devuelve el *callback*, como comentamos en la Sección B.2. Tuvimos que adaptar los métodos de libstreaming que implementan la parte cliente del protocolo RTSP (en modalidad publicar), para que, además de enviar el flujo local configurado con el objeto `Session`, pudiera enviar el flujo configurado con la `RebroadcastSession`.

7.2. Wifi Aware

Para gestionar la API de Wifi Aware y sus conexiones implementamos la clase `WifiAwareViewModel`, que funciona a modo de controlador. Con esta

clase podemos iniciar una sesión de Wifi Aware, la cual permite crear a su vez sesiones de *publisher* y/o *subscriber*.

Al comenzar la aplicación se inicia una sesión de *publisher* y otra de *subscriber*, siguiendo la infraestructura de red definida en la Sección 6.4. Toda la lógica que sigue la sesión de *publisher* y la de *subscriber* esta definida dentro del controlador. Esta lógica sigue el procedimiento explicado en el apéndice A.3, para crear conexiones cliente-servidor entre cada pareja de *subscriber-publisher*.

En concreto, **el *publisher* inicia el servidor RTSP del dispositivo y cuando un *subscriber* establece conexión con el, notifica al servidor para que cree un *socket* y se comunique con el otro dispositivo.**

El *subscriber* actúa de forma contraria, por cada *publisher* que encuentre y al que se conecte, se crea un cliente RTSP y se inicia al cliente de forma que cree un *socket* con el otro dispositivo.

La creación de estos *sockets* y su tratamiento es especial para Wifi Aware y tuvimos que modificar tanto el servidor como el cliente para utilizarlos. Estas modificaciones están desarrolladas en detalle en el apéndice B.3.

Cabe destacar que, **la creación de una conexión de Wifi Aware entre una pareja de dispositivos se complica cuando hay varios dispositivos intentando conectarse a la vez.** En la documentación solamente se especifica como conectar dos dispositivos [38], pero no menciona como gestionar varios intentos simultáneos de conexión. Si no se lleva un control sobre esta concurrencia, descubrimos que, en ocasiones, se producen errores que impiden a los dispositivos conectarse.

Nosotros tratamos este problema secuencializando las conexiones entre parejas de dispositivos. Aun con esto, aunque mucho menos probables, se producen fallos de conexión. Según nuestra infraestructura de red y teniendo implementado *multihopping* y el envío múltiple, el que falle la conexión entre un *publisher* y un *subscriber* no afecta al correcto funcionamiento de la aplicación.

En Android 12 se van a introducir mejoras para Wifi Aware [39], que puede que mitiguen este problema o podría investigarse otra forma de tratar esta concurrencia en un futuro. Para más detalle sobre como esta implementada esta gestión de conexiones, referirse al apéndice B.4.

El relación con **el problema de las sesiones que indican los alumnos del proyecto de 2019/20**, el cual explicamos en el apéndice A.1.2, **comprobamos que estaban equivocados en cuanto a su teoría de que el estado del Wifi y la ubicación tenía que estar siempre activo,** ya que si se desactivaba alguno se cerraba la sesión de Wifi Aware y más tarde no podía crearse una nueva.

Para solucionarlo, en el controlador recogemos un *callback* de la API

de Wifi Aware que indica si Wifi Aware deja de estar disponible. En ese caso cerramos la sesión actual, junto con la sesion de *publisher* y su servidor y la de *subscriber* y sus clientes, y notificamos al usuario. Cuando vuelva a estar disponible, por reactivar el Wifi o ubicación, podemos crear otra sesión para continuar con el uso normal de la aplicación, al contrario de lo que indicaban los alumnos de 2019/20.

8. Conclusiones y trabajo futuro

La creación y gestión de redes sin infraestructura entre dispositivos móviles es un área que está actualmente en investigación y que promete muchas ventajas, tanto para la gente que no tenga posibilidad de conectarse a la habitual infraestructura de red, como para los operadores de telefonía móvil e internet, para descongestionar sus redes o llegar a lugares con escasa cobertura, por ejemplo.

Con este proyecto hemos aplicado un enfoque orientado a la transmisión de vídeo y audio sobre este tipo de redes. Con este enfoque buscamos implementar una aplicación que fuera útil a la gente en situaciones de riesgo en las que no es posible acceder a la red convencional. Hemos distinguido dos escenarios de uso, un escenario orientado a transmitir vídeo y audio en accidentes o situaciones de emergencia en las que no se tiene acceso a la red, y otro orientado a denunciar crímenes y actos contrarios a los derechos humanos en lugares donde acceder a la red podría ser peligroso.

8.1. Recapitulación y evaluación

Para este proyecto partimos de la experiencia de los dos proyectos de los cursos 2018/19 y 2019/20. En el proyecto de 2018/19 implementaron su aplicación de *streaming* sobre la tecnología D2D, Wifi Direct. Como cuentan en su memoria, en el capitulo 5.1.3 [3], descubrieron ciertas limitaciones en la tecnología que les impedía transmitir entre los dispositivos, ya que un dispositivo “maestro” centralizaba las comunicaciones (es decir, una topología en estrella). Otros dispositivos cercanos podrían tener un dispositivo maestro diferente, lo que dividía a los dispositivos en redes ad hoc aisladas. A pesar de estas dificultades consiguieron integrar en su proyecto la biblioteca libstreaming para la transmisión de flujos multimedia. Consiguieron mejorar la implementación de la biblioteca, como se cuenta en la Sección 7.1, lo cual nos brindo una buena base para nuestro proyecto.

En cuanto al proyecto del curso de 2019/20, usaron la tecnología Wifi Aware como medio de comunicación D2D. Esta tecnología prometía superar las limitaciones de Wifi Direct así como mejorar la calidad de la red ad hoc. Es una tecnología muy temprana con muy poca documentación y apenas ejemplos de funcionamiento, sumado a que por razones ajenas a su voluntad no pudieron conseguir los móviles a tiempo, no pudieron cumplir con la mayoría de sus objetivos. Debido al poco tiempo no pudieron adaptar libstreaming a Wifi Aware, se encontraron ciertos problemas, ver Sección A.1.2 para más detalles, que, sumados a la poca documentación, les llevó a intentar otras alternativas. Finalmente decidieron transmitir los flujos utilizando el protocolo SCTP sin protocolo de control.

Nosotros para este proyecto, basándonos en las investigaciones de los años anteriores, decidimos intentar adaptar libstreaming a Wifi Aware. Wifi Aware es una tecnología en desarrollo que promete mucho más para las comunicaciones D2D que Wifi Direct. A parte de superar las limitaciones de Wifi Direct, está recibiendo atención por parte de Google y ya se esperan mejoras en la tecnología para la próxima versión de Android [39]. Libstreaming por su parte, es una biblioteca para *streaming* bastante accesible en cuanto a modificación se refiere, lo que nos permite centrarnos más en la investigación de Wifi Aware.

Como objetivos que propusimos y que hemos conseguido implementar tenemos los siguientes:

- **Adaptación libstreaming a IPv6 y a Wifi Aware:** Conseguimos que la biblioteca pudiera crear conexiones basadas en Wifi Aware, tras una costosa investigación tanto del código de libstreaming como del funcionamiento de Wifi Aware. Además, añadimos a la biblioteca la capacidad de transmitir streams sobre IPv4 o IPv6, ya que Wifi Aware necesitaba transmitir sobre IPv6. En la Sección 7.1.2 se explican estos cambios en detalle.

- ***Multihopping*:** En base al código de los alumnos de 2018/19, que consiguieron una transmisión entre una pareja de dispositivos, ampliamos la implementación de la biblioteca libstreaming para conseguir más de un salto en la transmisión, como se desarrolla en las Secciones 7.1.3 y 7.1.4.1.

- **Envío del *stream* a múltiples dispositivos:** Nosotros solventamos los problemas de los bucles como se explica en la Sección 7.1.4.2, y mas adelante en esta sección proponemos extender esta solución para poder mejorar el *routing* según otras métricas.

- **Guardar los *streams* en cualquier dispositivo participante de la comunicación para una posterior visualización:** Modificando en el servidor RTSP la modalidad reproducir y haciendo uso localmente del cliente RTSP en modalidad reproducir de libVLC, conseguimos la capacidad de visualizar/guardar los *streams* tanto en el dispositivo origen como el dispositivo que es un nodo de tránsito o receptor final.

- **Emisión del *stream* a un punto de acceso Wifi o estación base de la red de telefonía móvil:** Nosotros lo hemos conseguido gracias al *multihopping*. Cualquier dispositivo participante de la comunicación con acceso a internet es capaz de emitir el *stream*.

En cuanto al resto de las propuestas que realizaron los alumnos del curso 2019/20, propusieron dotar a la red de una mayor seguridad, para ello hemos dividido la aplicación en dos modos de operación según los escenarios de uso, en el Humanitario se da la opción de poner un identificador al *stream*, así como el nombre u otros datos del emisor, en el escenario *Witness* esta funcionalidad se restringe conservando en todo momento el anonimato del emisor, además hemos creado una aplicación auxiliar

de botón de pánico (Sección 6.5) para eliminar los datos de ella y dotar de más seguridad a la aplicación.

En lo que respecta a las adversidades que nos han surgido en la realización de este proyecto, uno de los principales problemas ha sido la poca documentación que teníamos para el desarrollo de Wifi Aware, aunque disponíamos de algunos ejemplos de los alumnos del curso 2019/2020, no era suficiente para poder realizar un desarrollo exhaustivo.

Otro problema fue la gestión de múltiples conexiones simultáneas con Wifi Aware, ya que en la documentación solamente se especifica como conectar dos dispositivos [38], pero no menciona como gestionar varios intentos simultáneos de conexión. Si no se lleva un control sobre esta concurrencia, descubrimos que, en ocasiones, se producen errores que impiden a los dispositivos conectarse. Esta gestión era fundamental para conseguir el envío múltiple de *streams* por la red y para solucionarlo realizamos la secuencialización de dichas conexiones como mencionamos en la Sección B.4.

8.2. Trabajo futuro

En relación con el futuro de la aplicación, hemos conseguido una base robusta en cuanto a la comunicación sobre la infraestructura descentralizada y a la transmisión de *streams* con la biblioteca libstreaming, lo que permite continuar el desarrollo futuro de este proyecto más centrado en avances relacionados con los escenarios de uso Humanitario y *Witness*, como la seguridad de la MANET, o la pixelación de las caras, por ejemplo.

Como propuestas para el trabajo futuro, en cuanto a las funcionalidades básicas tenemos las siguientes:

- **Optimizar la recepción del *stream* por el mejor nodo:** Se podrían tener en cuenta métricas como la distancia o el número de saltos, la cual influye bastante en el retraso del *stream*. En este caso, como solución alternativa a la nuestra para problema de los bucles, se podrían tener en cuenta las metodologías aplicadas por otros protocolos de *routing* en redes ad hoc como OLSR [40] y en DTNs (Delay Tolerant Networks) [41].
- **Adaptación de libstreaming a RTSP 2.0:** Se tendría que adaptar la biblioteca para hacer uso de la modalidad reproducir en la transmisión de los *streams* entre dispositivos, ya que la modalidad publicar en esta versión desaparece. Para ello seria necesario implementar en el cliente RTSP, que actualmente utiliza la modalidad publicar para enviar sus *streams* al servidor, la modalidad reproducir para solicitar y recibir *streams* de los servidores RTSP. En este caso solo haría falta implementar en el cliente RTSP las directivas del

protocolo RTSP en modalidad reproducir, ya que el servidor RTSP ya implementa la modalidad reproducir y el envío del flujo por RTP y RTCP ya está implementado.

Con este modelo de comunicación surge el problema de hacer llegar a los clientes RTSP el listado de identificadores de los *streams* disponibles para solicitar al servidor RTSP, ya que en RTSP no hay ningún procedimiento para obtener esta información y el cliente RTSP tiene que conocer previamente el identificador de un *stream* para solicitarlo. Como solución se podría hacer una distinción de casos en la directiva DESCRIBE del servidor RTSP por la que, en función de la URI solicitada, se responda al cliente RTSP la descripción de un *stream* en concreto o una lista de los identificadores de los *streams* que dispone el servidor RTSP, para una posterior solicitud. Otra opción sería transmitir la lista de identificadores de *streams*, previamente al envío de la directiva DESCRIBE de RTSP, a través de la API de mensajes de Wifi Aware.

Como propuestas para el trabajo futuro, en cuanto a las funcionalidades específicas de cada escenario de uso, tenemos las siguientes:

1. **Escenario Humanitario:**
   - Pixelación de caras, aprovechando otros códigos de software libre como el de ObscuraCam [42].
   - Posibilidad de añadir un destino final sugerido al *stream*.
   - Consideraciones de los metadatos para hacerlos más robustos y seguros, además de asegurar que incluyen las coordenadas estilo GPS.
2. **Escenario *Witness*:**
   a) **Seguridad:**
      - La red podría ser susceptible a ataques como, la inyección de *streams* que constituyen una actividad criminal, como la pornografía infantil, o con el fin de montar un ataque DDoS y la atribución, modificación o transmisión de *streams* suplantando la identidad del emisor. Como defensas contra estos ataques se podría hacer uso de una clave de grupo previamente acordada, para encriptar los *streams* que circulan por la red y de esta manera solo los usuarios con la clave puedan transmitirlos y visualizarlos, siguiendo de alguna forma el protocolo GDOI [43]. Para esto se podría aprovechar el código de InformaCam [44].
      - Para asegurar la autenticidad de los *streams* se podría utilizar H.264/SVC como formato de codificación de vídeo. Se podría utilizar para ello la biblioteca svcAuth [45] para Java, que permite la verificación de todos los *streams* transmitidos.

   b) **Cadena de custodia:** Realizar una conexión con una implementación de SecureDrop (diseñado y desarrollado por Aaron Swartz [46] junto con Kevin Poulsen) vía Tor para enviar *streams* almacenados.

c) **Privacidad:**

- Encriptado de *streams* guardados en ficheros haciendo uso de claves creadas por el propio usuario.
- Eliminar de los metadatos transmitidos cualquier información que pueda delatar al autor del vídeo.

8.3. Observaciones finales

Finalmente, con el desarrollo de este proyecto hemos hecho una contribución importante a la implementación de esta aplicación y, por tanto, al objetivo del proyecto en que se enmarca este TFG de construir y distribuir una aplicación de software libre que podrá tener una gran utilidad social. Además, estamos barajando hacer un fork del proyecto libstreaming llamado multistreaming que contendrá toda la nueva funcionalidad con la cual hemos extendido esta bien conocida librería.

9. Conclusions and future work

The creation and management of infrastructure-free networks between mobile devices is an area that is currently under investigation and holds great promise, both for people who do not have the possibility of connecting to the usual network infrastructure, and for cell phone and internet operators, to ease congestion in their networks or reach places with poor coverage, for example.

With this project we have applied an approach oriented to the transmission of video and audio over this type of networks. With this approach we sought to implement an application that would be useful to people in risk situations where it is not possible to access the conventional network. We have distinguished two usage scenarios, one scenario oriented to transmit video and audio in accidents or emergency situations where there is no network access, and another oriented to report crimes and acts contrary to human rights in places where network access could be dangerous.

9.1. Summary and evaluation

For this project we start from the experience of the two projects of the academic years 2018/19 and 2019/20. The students of the 2018/19 project implemented their streaming application on the D2D technology, Wifi Direct. As they recount in Section 5.1.3 of their final report [3], they discovered certain limitations in the technology that prevented them from streaming between devices, due to all the communications having to pass through a "master" device (i.e. a star topology). Other nearby devices might have a different master device, which divided the devices into isolated ad hoc networks. Despite these difficulties they managed to integrate the libstreaming library for multimedia streaming into their project. They managed to improve the implementation of the library, as reported in Section 7.1, which gave us a good basis for our project.

As for the 2019/20 course project, they used Wifi Aware technology as a means of D2D communication. This technology promised to overcome the limitations of Wifi Direct as well as improve the quality of the ad hoc network. It is a very early technology with very little documentation and hardly any working examples, added to the fact that for reasons beyond their control they could not get the cell phones in time, so that they could not meet most of their goals. Due to the lack of time they could not adapt libstreaming to Wifi Aware, they encountered certain problems, see Section A.1.2 for more details, which, added to the lack of documentation, led them to try other alternatives. They finally decided to transmit the streams using the SCTP protocol without control protocol.

For this project, based on previous years research, we decided to try to adapt libstreaming to Wifi Aware. Wifi Aware is a developing technology that promises much more for D2D communications than Wifi Direct. Apart from overcoming the limitations of Wifi Direct, it is getting attention from Google and improvements in the technology are already expected for the next version of Android [39]. Libstreaming, on the other hand, is a streaming library that is quite accessible in terms of modification, allowing us to focus more on studying Wifi Aware.

Our proposed goals, which we have been able to implement, are the following:

- **Adaptation of libstreaming to IPv6 and Wifi Aware:** We made the library capable of creating connections based on Wifi Aware, after a costly investigation of both the libstreaming code and the operation of Wifi Aware. In addition, we added to the library the ability to stream over IPv4 or IPv6, since Wifi Aware needed to stream over IPv6. These changes are explained in detail in the Section 7.1.2.
- **Multihopping:** Based on the 2018/19 students code, which achieved a stream between a pair of devices, we extend the implementation of the libstreaming library to achieve more than one hop in the stream, as developed in Sections 7.1.3 and 7.1.4.1.
- **Multicast:** We solve the looping problems as explained in Section 7.1.4.2, and later in this section we propose to extend this solution in order to improve routing according to other metrics.
- **Save the streams on any device participating in the communication for future viewing:** By modifying the play mode on the RTSP server and using the libVLC play mode RTSP client locally, we obtain the ability to view/save the streams on both the source device and the device that is a transit node or final receiver.
- **Broadcasting the stream to a Wifi access point or base station of the cell phone network:** We have achieved this thanks to multihopping. Any device participating in the communication with internet access is able to broadcast the stream.

As for the rest of the proposals made by the students of the 2019/20 course, they proposed to provide the network with greater security, for this we have divided the application into two modes of operation according to the usage scenarios, in the Humanitarian mode, the option is given to provide an identifier for the stream, as well as the name or other data of the sender, in the Witness scenario this functionality is restricted preserving at all times the anonymity of the sender, in addition we have created an auxiliary panic button application (Section 6.5) to remove data from it and provide more security to the application.

Regarding the adversities that have arisen in the realization of this project, one of the main problems has been the little documentation we had for the development

of Wifi Aware. Although we had some examples from the students of the 2019/2020 course, it was not enough to be able to carry out a comprehensive development.

Another problem was the management of multiple simultaneous connections with Wifi Aware, as the documentation only specifies how to connect two devices [38], but does not mention how to manage multiple simultaneous connection attempts. If this concurrency is not kept under control, we find that sometimes errors occur that prevent devices from connecting. This management was essential to achieve stream multicast over the network and to solve this we sequenced these connections as mentioned in Section B.4.

9.2. Future work

Regarding the future of the application, we have achieved a robust foundation in terms of communication over the decentralized infrastructure and streaming with the libstreaming library, which allows us to continue future development of this project more focused on advances related to Humanitarian and Witness usage scenarios, such as MANET security, or face pixelation, for example.

As proposals for future work, in terms of basic functionalities we have the following:

- **Optimize the reception of the stream by the best node:** Metrics such as distance or number of hops, which have a strong influence on the stream delay, could be taken into account. In this case, as an alternative solution to ours for the loop problem, methodologies applied by other routing protocols in ad hoc networks such as OLSR [40] and DNTs (Delay Tolerant Networks) [41] could be taken into account.
- **Adaptation of libstreaming to RTSP 2.0:** The library would have to be adapted to make use of the play mode in the transmission of streams between devices, since the record mode disappears in this version. To do so, in the RTSP client, which currently uses the record mode to send its streams to the server, it would be necessary to implement, the play mode to request and receive streams from the RTSP servers. In this case, it would only be necessary to implement in the RTSP client the RTSP protocol directives in play mode, since the RTSP server already implements the play mode and the sending of the stream by RTP and RTCP is already implemented.

  With this communication model, the problem of providing to RTSP clients the list of available stream identifiers to request from the RTSP server arises, since in RTSP there is no procedure to obtain this information and the RTSP client must previously know the identifier of a stream in order to request it. As a solution, a case distinction could be made in the DESCRIBE directive of the RTSP server so that, depending on the requested URI, the RTSP client receives the description of a specific stream or a list of the available stream

identifiers to request from the RTSP server, for a later request. Another option would be to transmit the list of stream identifiers, prior to sending the RTSP DESCRIBE directive, through the Wifi Aware message API.

As proposals for future work, in terms of specific functionalities for each usage scenario, we have the following:

1. **Humanitarian Scenario:**
   - Pixelation of faces, taking advantage of other free software codes such as ObscuraCam [42].
   - The possibility of adding a suggested final destination to the stream.
   - Consideration of metadata to make it more robust and secure, as well as ensuring that it includes GPS-style coordinates.
2. **Witness Escenario:**
   a) **Security:**
      - The network could be susceptible to attacks such as the injection of streams that constitute criminal activity, such as child pornography, or for the purpose of mounting a DDoS attack and the attribution, modification or transmission of streams by impersonating the sender. As defenses against these attacks, a previously agreed group key could be used to encrypt the streams circulating on the network so that only users with the key can transmit and view them, following the GDOI [43] protocol in some way. The InformaCam [44] code could be used for this purpose.
      - To ensure the authenticity of the streams, H.264/SVC could be used as the video encoding format. The svcAuth [45] library for Java, which allows verification of all transmitted streams, could be used for this purpose.

   b) **Chain of custody:** Make a connection to an implementation of SecureDrop (designed and developed by Aaron Swartz [46] together with Kevin Poulsen) via Tor to send stored streams.

   c) **Privacy:**
      - Encryption of streams stored in files using user-created keys.
      - Remove from the transmitted metadata any information that could reveal the identity of the author of the video.

9.3. Concluding remarks

Finally, with the development of this project we have made an important contribution to the implementation of this application and, therefore, to the goal of the project in which context this bachelor's degree final project has been carried out to build and distribute a free/open source software application that can have a great social utility. In addition, we are considering making a fork of the libstreaming project called multistreaming containing all the new functionality with which we have extended this well-known library.

10. Aportación de los participantes

Para la realización de este proyecto, hemos dividido el trabajo en una primera fase de investigación, en la que conjuntamente, hemos estudiado los proyectos de otros años para entender de qué punto partíamos, así como estudiamos nuevas tecnologías D2D como Wifi Aware y la viabilidad del uso de la biblioteca libstreaming, la cual nos proporcionaba un protocolo control para la transmisión de *streams*. En las siguientes fases nos centramos más en el desarrollo y la redacción de la memoria, siguiendo una metodología ágil en la que nos hemos reunido ambos miembros del equipo mediante Google Meet para repartirnos el trabajo, además de realizar reuniones semanales con el director del TFG para mantener la dirección y enfoque correcto del proyecto.

En la Figura 10.1, podemos observar una planificación estimada del TFG con un diagrama de Gantt y, a continuación, mostramos en detalle la aportación de cada participante desglosado en tareas.

10.1. Daniel Alfaro Miranda

10.1.1. Investigación

- Estudio de tecnologías de comunicación D2D: Wifi Aware, Wifi Direct, Wifi Ad hoc mode, Bluetooth...
- Estudio de protocolos de *streaming*: RTSP, SCTP, HLS, RTP y RTCP...
- Investigación sobre la librería libstreaming
- Estudio del proyecto 2018/2019
- Estudio del proyecto 2019/2020
- Estudio de herramientas y lenguajes de aplicaciones Android: Android Studio, XML y Apis de Android.

10.1.2. Desarrollo

- Prototipo inicial para el estudio de Wifi Aware y adaptación de libstreaming a IPv6.
- Especificación de los requisitos.
- Diseño de *mockups*.
- Diseño de clases con diagramas UML.

- Agrupación del acceso a los flujos multimedia y cambio en la infraestructura de red
- Implementación en el cliente RTSP de modalidad publicar para *streams* con origen en otro dispositivo y adaptación de la modalidad publicar para *streams* con origen en el mismo dispositivo.
- Implementación de clase para el control de la cámara.
- Adaptación del cliente y servidor RTSP para transmisión múltiple.
- Gestión de intentos de conexión concurrentes con Wifi Aware.
- Implementación de guardar los *streams* en los dispositivos participantes.
- Apoyo en la interfaz gráfica.

10.1.3. Memoria

- Resumen
- Introducción
- Estado del arte y antecedentes
- Elección de tecnologías
- Especificación
- Diseño
- Implementación
- Conclusiones y trabajo futuro
- Apéndices diseño
- Apéndices implementación.

10.2. Luis Pozas Palomo

10.2.1. Investigación

- Estudio de tecnologías de comunicación D2D: Wifi Aware, Wifi Direct, Wifi Ad hoc mode, Bluetooth...
- Estudio de protocolos de *streaming*: RTSP, SCTP, HLS, RTP y RTCP...
- Investigación sobre la librería libstreaming
- Estudio del proyecto 2018/2019
- Estudio del proyecto 2019/2020
- Estudio de herramientas y lenguajes de aplicaciones Android: Android Studio, XML y Apis de Android.

10.2.2. Desarrollo

- Prototipo inicial para el estudio de Wifi Aware y adaptación de libstreaming a IPv6.
- Especificación de los requisitos.
- Diseño de *mockups*.
- Diseño de clases con diagramas UML.
- Agrupación del acceso a los flujos multimedia y cambio en la infraestructura de red
- Adaptación en el servidor RTSP de la modalidad reproducir.
- Apoyo implementación de clase para el control de la cámara.
- Gestión de intentos de conexión concurrentes con Wifi Aware.
- Implementación de guardar los *streams* en los dispositivos participantes.
- Implementación del botón de pánico.
- Implementación de la interfaz gráfica.

10.2.3. Memoria

- Resumen
- Introducción
- Estado del arte y antecedentes
- Elección de tecnologías
- Especificación
- Diseño
- Implementación
- Conclusiones y trabajo futuro
- Apéndices diseño
- Apéndices implementación.

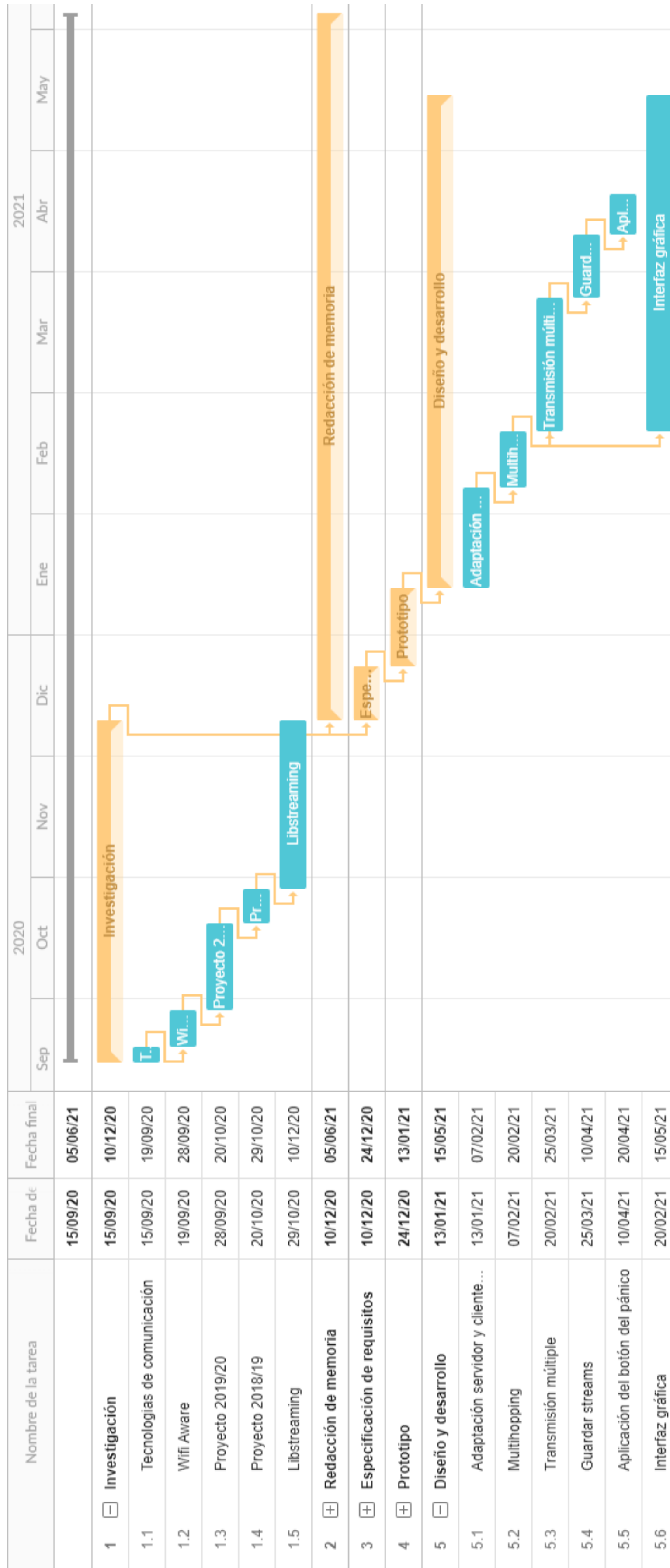


Figura 10.1: Diagrama de Gantt.

A. Detalles del diseño

A.1. Investigación y análisis inicial

Nuestro primer paso en el desarrollo del proyecto fue investigar los proyectos anteriores, leer sus memorias, entender sus decisiones, avances y problemas y analizar su código. En función de esto decidimos como orientar nuestro trabajo.

A.1.1. Proyecto 2018/19

Comenzamos con el proyecto **Device to Device Streaming en dispositivos móviles** [3] del curso 2018/19. Sus objetivos eran los mismos que los nuestros y decidieron desarrollar su proyecto apoyándose en la librería de código abierto **libstreaming** para la transmisión de cámara y vídeo por RTSP, RTP y RTCP y de **Wifi Direct** como medio de conexión D2D entre dispositivos.

En cuanto a la biblioteca libstreaming, tuvieron que adaptar las clases que utilizaba para transmitir del flujo de la cámara y el vídeo, ya que la biblioteca estaba pensada para transmitir el flujo a un único dispositivo y tenían conflictos con el hardware a la hora de transmitir varios flujos (Como se explica en el capitulo 6.5.1 de su memoria [3], esto no lo solucionaron completamente). También tuvieron que adaptar su servidor RTSP, ya que el cliente RTSP solo podía trabajar con la **modalidad publicar** y el servidor solo con la **modalidad reproducir**. El cliente RTSP de libstreaming estaba pensado para transmitir a un servidor RTSP como Wowza Media Server y el servidor RTSP de libstreaming estaba pensado para ser utilizado con un cliente RTSP con la modalidad reproducir como libVLC. **Implementaron la modalidad publicar en el servidor RTSP de libstreaming** para utilizarlo con el cliente RTSP de libstreaming, logrando implementar multihopping (capitulo 6.5.2 de [3]).

Para utilizar Wifi Direct implementaron un controlador y clases para automatizar la búsqueda y conexión de los dispositivos. Tras realizar pruebas encontraron una limitación para su aplicación a causa de que Wifi Direct trabaja asociando a los dispositivos por grupos aislados y necesitaban que el nodo de tránsito tuviera que cambiar continuamente de uno a otro para transmitir (capitulo 5.1.3 de [3]).

Un avance suyo también destacable es la **resolución de unos problemas de configuración de libVLC** que hacía que la imagen se viera girada y pequeña al reproducir. Probando la aplicación nos dimos cuenta de que **aun así necesitaba retoques si nos decidíamos a usarlo**, ya que la resolución no era la misma que la pantalla y aparecía un cuadrado negro en la parte superior y que al modificar la

configuración de libstreaming para utilizar la cámara frontal, al hacer *streaming* se veía la imagen al revés. Encontramos otro problema relacionado con la cámara y es que esta **no enfocaba bien**, lo que se notaba mucho ya que utilizaban una calidad baja de transmisión y no se podían leer textos.

A.1.2. Proyecto 2020/21

Nuestro siguiente paso fue estudiar el proyecto **Crowdstreaming: transmisión de vídeo y audio por una red ad hoc de teléfonos móviles** [4] del curso 2019/20. Hay que remarcar que por circunstancias fuera de su control, no tuvieron los dispositivos móviles hasta principios de marzo del 2020, lo que supuso un retraso en sus investigaciones. Para este proyecto se arriesgaron a utilizar **Wifi Aware en vez de Wifi Direct** para la conectividad, aún con la poca documentación y ejemplos. Consideraron el problema de los grupos de Wifi Direct, capitulo 5.1.3 de [3], un impedimento muy grande para conseguir su aplicación. Para la transmisión de vídeo y audio **decidieron intentar adaptar libstreaming a Wifi Aware** donde se encontraron con los **siguientes problemas:**

1. **Era necesario tener el Wifi y la ubicación del dispositivo activados antes de iniciar la aplicación** para crear una sesión de Wifi Aware. Esta sesión permitía crear las conexiones entre los dispositivos y los alumnos aseguraron que la sesión debía de crearse obligatoriamente al inicio de la aplicación y no se podía volver a crear una nueva más adelante.

2. **Wifi Aware utiliza direcciones IPv6 y libstreaming solo estaba implementado para utilizar IPv4.** Ellos intentaron adaptar libstreaming, haciendo que el cliente RTSP recibiera los *sockets* creados con la API de Wifi Aware, pero no les funciono por razones desconocidas.

Finalmente, **decidieron desistir de libstreaming** y realizar la transmisión por medio del protocolo SCTP, incluido como utilidad en la API de Wifi Aware, aun sabiendo que no utilizarían protocolo de control del *streaming*, esperando implementarlo más adelante.

Utilizaron libVLC para la reproducción del *streaming*. Ya que no utilizaban protocolo de control tuvieron que utilizar un buffer HTTP en un *socket* local, lo que introducía retrasos en la reproducción, sumado a la pérdida de calidad por no utilizar protocolo de control.

En el capitulo 6.6.1 de su memoria [4], se cita lo siguiente:

«No obstante, dado que no estamos usando ningún protocolo de control de sesiones de vídeo en nuestra implementación, implementamos un buffer a través de un proxy HTTP local, el cual nos permitía transformar el flujo del socket a un flujo legible para la librería libVLC. Aunque esto provoca un pequeño retraso en el stream, garantiza que no se produzcan cortes.»

A destacar, **implementaron el almacenamiento del streaming en el dispositivo** y una galería para verlos, consiguieron hacer pasar el *streaming* a través de un nodo intermedio (aunque nosotros no conseguimos verificarlo al probar su aplicación). El problema de los bucles en la transmisión no lo contemplaron al solo poder hacer una única transmisión desde un dispositivo.

A.1.3. Conclusiones

Por lo mencionado en el apartado 4 de elección de tecnologías, **consideramos que Wifi Aware es la más apropiada para nuestra aplicación**, a pesar de tener poca documentación. El hecho de tener este año los móviles para probar Wifi Aware desde el principio nos permitía dedicarle más tiempo a investigar.

En cuanto a la transmisión de vídeo y audio consideramos que la propuesta del año pasado de usar SCTP sin protocolo de control no tenía mucho futuro. A parte de empeorar la calidad por no usar protocolo de control, no es un estándar *de facto* en el mundo del *streaming*. Es mejor utilizar protocolos populares, ya que seguramente van a estar más optimizados y va a haber más código compatible con él, como libVLC, que puede reproducir un *streaming* distribuido por RTSP. Por estas razones empezamos a estudiar si podíamos adaptar otro software libre como libstreaming para manipulación de flujos multimedia.

Una opción que investigamos fue VLC media player, que reemplaza el antiguo VideoLan Server, y se puede usar como cliente y servidor para crear *streamings* en múltiples plataformas y con distintos códecs de audio y vídeo. El envío de flujos multimedia lo realizan por medio de la biblioteca Live555 en C++, que implementa el protocolo RTSP, RTP y RTCP. Utilizar VLC media player o directamente Live555 mejoraría bastante el tratamiento multimedia de la aplicación, al estar muy estudiado e implementado en C++, aunque requeriría mucho trabajo de investigación y adaptación.

No se puede usar directamente una biblioteca para transmitir el *streaming* debido a que no están adaptadas para utilizar la API de Wifi Aware. Esta API requiere crear los sockets de una manera especial, como se explica en la Sección A.3.

A pesar de toda la investigación tuvimos que rechazar la propuesta porque descubrimos que VLC no soporta IPv6 para los *streamings* por RTSP [47]. Está adaptado en su mayoría a IPv6, pero la parte de RTSP que viene de Live555, por el momento no. En general el *streaming* con IPv6 suele realizarse sobre HTTP, pero su uso sobre protocolos específicos de *streaming* como RTSP no está muy extendido por el momento, y mucho menos para Android. Por ello **intentar adaptar libstreaming a IPv6 cada vez nos parecía una mejor opción**. A destacar, WebRCT utiliza SRTP sobre IPv6 y podría ser una buena alternativa de desarrollo.

Elegimos intentar adaptar libstreaming por las siguientes razones:

- El código es más accesible que otras alternativas. Los *sockets* de Wifi Aware tienen que crearse de una forma especial y solo hay documentación para hacerlo en Java. Pensar en utilizar una biblioteca en C++ podría suponer mucho tiempo invertido en investigación.
- Tenemos más ejemplos de uso en otras aplicaciones.
- Los alumnos del proyecto de 2018/19 implementaron parcialmente la extensión de la biblioteca para incluir el multihopping y el envío múltiple, lo que nos podría ahorrar bastante trabajo.

A.2. Elección de modalidad RTSP

En la Sección 3.2.3, presentamos las modalidades que define RTSP. Con estas modalidades teníamos las siguientes opciones de implementación para multihopping:

1. **Utilizar solamente la modalidad RTSP publicar**. En esta modalidad, el cliente RTSP es el emisor del *stream* y se lo ofrece activamente a los servidores.
2. **Utilizar solamente la modalidad RTSP reproducir**. En esta modalidad, el servidor RTSP es el emisor del *stream* y espera a que los clientes se lo pidan.
3. **Alternar entre las dos modalidades** según algún criterio.

Nosotros decidimos utilizar la primera, solamente la modalidad publicar para transmitir los flujos multimedia entre dispositivos. Consideramos que tiene más sentido para la finalidad de nuestra aplicación al tener un carácter de propagación más activo, en el escenario *Witness* (Sección 5.1) interesa subir el *stream* en un nodo con internet lo antes posible y en el *Humanitarian* (Sección 5.1) avisar a emergencias rápidamente. **Utilizar la segunda o tercera implicaría tener que implementar la modalidad reproducir en el cliente RTSP de libstreaming, aparte de introducir complejidad innecesaria a la comunicación.** Con complejidad nos referimos a que, utilizando esta modalidad, el cliente tendría que conocer previamente los streams disponibles en el servidor para pedirle la transmisión de uno. RTSP no implementa ningún mensaje en el protocolo para transmitir esta información y tendría que hacerse de forma ajena a este.

Para implementar este modelo de transmisión ideamos la infraestructura de red explicada en la Sección 6.4.

Cabe destacar que, **en libstreaming, esta implementado RTSP 1.0**. En cierto momento de la implementación nos dimos cuenta de que **en RTSP 2.0, la modalidad publicar es eliminada**. En el RFC 7826 [48], se cita lo siguiente:

«The following protocol elements were removed in RTSP 2.0 compared to RTSP 1.0:»

- «the RECORD and ANNOUNCE methods and all related functionality (including 201 (Created) and 250 (Low On Storage Space) status codes);»

Esto supone que, **en un futuro se tendría que sustituir el uso de la modalidad publicar en nuestra aplicación** con todo lo que ello conlleva. En la Sección 8 de trabajo futuro, se comenta en detalle como se podría sustituir esta modalidad.

A.3. Creación de conexiones con Wifi Aware

Para crear una conexión cliente-servidor entre dispositivos a través de Wifi Aware se tienen que seguir los siguientes pasos:

1. En un dispositivo, el programa que quiere ofrecer un servicio, que llamaremos el *publisher* (en nuestro caso, el *publisher* implementará un servidor-RTSP-en-modalidad-publicar), invoca el método `publish` de la clase `WifiAwareSession`. En otro dispositivo, el programa que quiere usar el servicio, que llamaremos el *subscriber* (en nuestro caso, el *subscriber* implementa un servidor-RTSP-en-modalidad-publicar), invoca el método `subscribe` de la clase `WifiAwareSession`. En los dos dispositivos, el objeto `WifiAwareSession` se crea con un nombre de servicio, que sirve para que Wifi Aware case el servicio que ofrece el *publisher* con el que busca el *subscriber*.

2. Cuando el sistema de Wifi Aware descubre un *publisher* del servicio que un *subscriber* busca, notifica al *subscriber* en cuestión con un *callback* que tiene como argumento el objeto `PeerHandle` del *publisher*. Cuando el *subscriber* recibe el *callback* de descubrimiento, puede usar el objeto `PeerHandle` del *publisher* para enviarle un mensaje.

3. Cuando el *publisher* recibe un *callback* que tiene como argumento el objeto `PeerHandle` del *subscriber* y que sirve para notificarle que ha llegado un mensaje de un *subscriber* que quiere usar el servicio que ofrece, crea un `ServerSocket`, o `ServerSocketChannel` si utilizamos Java NIO, y guarda el número de su puerto.

4. El *publisher* puede usar el API de Android con el objeto `DiscoverySession`, obtenido al publicar su servicio, el objeto `PeerHandle` del *subscriber* y el puerto del `ServerSocket` para crear un objeto `NetworkRequest`. Este objeto sirve para solicitar un objeto `Network` que permite vincular (*bind*) *sockets* a una de las redes que tiene disponible el dispositivo, como puede ser Wifi, 4G o Wifi Aware. Esta vinculación sirve para que la resolución de direcciones IP y el envío de paquetes se realice a través de la red seleccionada. La API de Android proporciona una clase llamada `SocketFactory` para crear `Sockets` asociados a una red o, mediante la clase `ConectivityManager`, la asociación del thread entero a la red para que en la creación de cualquier tipo de *socket* y canal se vincule internamente al objeto `Network` [49]. Una vez solicitado el objeto `Network`, podemos recibir *callbacks* cuando la red esté o no disponible o cuando esta se pierda.

5. Cuando el *publisher* haya solicitado el objeto `Network` debe enviar un mensaje al *subscriber* para avisarle.

6. Cuando el *subscriber* reciba el mensaje, debe crear un `NetworkRequest` de la misma forma que en el *publisher*, pero sin especificar puerto.

7. Cuando el *subscriber* reciba el *callback* `onAvailable` de la `NetworkRequest`, el cual proporciona el objeto `Network` solicitado, puede obtener la dirección IPv6 y puerto del servidor y crear un *socket* con él. Tras recibir el *callback* `onAvailable`, se recibe siempre el *callback* `onCapabilitiesChanged`, el cual, incluye como parámetro un objeto de la clase `NetworkCapabilities` del que se pueden obtener la IPv6, y el puerto en caso de crearse la `NetworkRequest` en el punto 4 incluyendo el puerto, del otro extremo de la conexión.

8. Una vez las conexiones se cierren es importante eliminar el `NetworkRequest` del sistema para evitar pérdidas de memoria [50].

Estos pasos se basan en la documentación de Android sobre la creación de conexiones de Wifi Aware [38].

B. Detalles de implementación

B.1. Estado de libstreaming y los cambios en el curso 2018/19

En esta sección comentaremos en detalle la implementación de libstreaming y los cambios que introdujeron los alumnos del curso 2018/19 para su proyecto.

B.1.1. Estado de libstreaming

Libstreaming proporciona la implementación de un cliente RTSP en modalidad publicar y un servidor RTSP en modalidad reproducir, en ambos casos aplicando la modalidad solamente a *streams* con origen en el mismo dispositivo que hace uso de la biblioteca, es decir, en el cliente RTSP solo se puede publicar un *stream* y al servidor RTSP solo se le puede solicitar un *stream* que se este capturando en el mismo dispositivo.

El servidor RTSP esta definido en la clase `RtspServer` y el cliente RTSP en la clase `RtspClient`. La clase `RtspServer` extiende la clase `Service` de Android e internamente esta compuesto por un *thread* que atiende las peticiones de conexión y crea un nuevo *thread* para atender a cada cliente que se conecte. La implementación del servidor esta orientada a que un cliente, según la información que transmita en la URI del *stream* que quiere, fije los parámetros de la cámara y códecs de audio y vídeo para que el servidor inicie la captura de audio y vídeo y le transmita los datos. Esto supone un problema al atender a varios clientes, ya que no se maneja la posibilidad de que varios clientes soliciten distintos parámetros en la URI, lo que genera un conflicto en el uso de la cámara. Además, como se vera en detalle en el apéndice B.2, la clase `Session`, que lee los datos de la cámara y micrófono y los empaqueta y envía por RTP y RTCP, no esta implementada para poder utilizarse con mas de una instancia a la vez, lo que imposibilita enviar el mismo flujo a mas de un cliente.

La implementación del cliente RTSP permite publicar un solo *stream* a un único servidor RTSP. El envió del flujo multimedia se hace de la misma forma que en el servidor, mediante un objeto `Session`, a diferencia de que este objeto no esta configurado según la URI, si no según unos parámetros configurados al crear una instancia del cliente.

B.1.2. Cambios en libstreaming por los alumnos de 2018/19

Los alumnos de 2018/19, orientaron parte del proyecto a adaptar libstreaming para que su cliente y servidor RTSP pudieran trabajar juntos. Para ello, añadie-

ron en el servidor RTSP la implementación de la modalidad publicar, para poder recibir *streams* de otros dispositivos, y modificaron la modalidad reproducir para que pudiera reenviar los *streams* recibidos de otros dispositivos con la modalidad publicar. Esta implementación de la modalidad reproducir, nos permite visualizar y/o guardar los *streams* recibidos en el dispositivo mediante el uso del cliente RTSP de libVLC.

La implementación de libstreaming de la modalidad reproducir en el servidor RTSP, para enviar un *stream* con origen en el mismo dispositivo, decidieron no adaptarla a los cambios que hicieron en el servidor, sobretodo en cuanto al uso de la URI para referirse a un *stream*, y quedo en desuso. Nosotros necesitamos esta funcionalidad para poder guardar el *stream* del usuario mientras se captura en el mismo dispositivo.

Al implementar todos estos cambios reestructuraron el código del servidor RTSP. Dividieron la parte de tratamiento de las conexiones y clientes en la clase `RTSPServerSelector` y todo el protocolo RTSP y la creación de *streams* a la clase `RTSPServerWorker`. En la clase `RTSPServerSelector` se leen los datos de cada cliente, el `RTSPServerWorker` los procesa y la clase `RTSPServerSelector` los escribe. El tratamiento de los clientes en la clase `RTSPServerSelector` cambió, con respecto a la implementación de libstreaming, de un *thread* por cliente para leer y escribir sus datos, a un único *thread* que trate tanto la peticiones de nueva conexión como las lecturas y escrituras de todos los clientes, utilizando la API de Java NIO. Utilizaban en concreto la clase `Selector` de Java NIO y cabe destacar que libstreaming no utilizaba Java NIO.

Además, crearon dos clases, `ReceiveSession` y `RebroadcastSession`. La clase `ReceiveSession` es la encargada de recibir los datos de un flujo multimedia por RTP y RTCP y reenviarlos a los destinos que se le indiquen, como un servidor UDP que retransmite los datos que recibe. La clase `RebroadcastSession` es utilizada para reenviar un flujo multimedia recibido con una ReceiveSession. Al crear una instancia se le pasa una instancia de una `ReceiveSession`. Al fijar un destino para el reenvío del flujo, añade el *socket* destino de los datos al servidor de la `ReceiveSession` para que los reenvíe.

Además de estos cambios, añadieron funcionalidad para poder visualizar un *stream* haciendo uso de libVLC. Una vez conseguían la URI, basada en IPv4, que identificaba el *stream* RTSP que querían visualizar dentro del servidor RTSP, creaban una instancia de la biblioteca pasándole la URI, y el cliente RTSP de libVLC, que implementa la modalidad reproducir, le solicitaba el *stream* al servidor RTSP de libstreaming y lo presentaba en la pantalla.

En la implementación de libstreaming el código para configurar y controlar la cámara esta mezclado con el código para transmitir *streams*, lo que según se investigo por los alumnos de 2018/29, imposibilita el envió múltiple (*multicast*) del *stream*. Ellos empezaron la implementación de una solución pero no lograron resolver este

problema. Nosotros partimos de un código con el que podemos enviar desde el cliente RTSP un solo *stream* a un único servidor RTSP y podemos recibir solamente un *stream* por cliente en el servidor RTSP.

Los cambios que introdujeron fueron la creación de dos clases que siguen el patrón *Singleton*, `VideoPacketizerDispatcher` y `AudioPacketizerDispatcher`. Ya que las clases encargadas de codificar el audio y vídeo no pueden leer a la vez del hardware al generar conflictos de lectura, estas nuevas clases actúan de intermediario, almacenando los datos de vídeo y audio y copiándolo a cada clase codificadora. Estas clases codificadoras tienen que ser modificadas para suscribirse al envió de información de estas nuevas clases. Los alumnos de 2018/19 solo adaptaron la clase que codifica el formato de vídeo H264 y el formato de audio AAC, las demás quedaron en desuso.

Aun así, queda por resolver otro problema por el cual no es posible el envió múltiple, porque en libstreaming la configuración y control de la cámara están contenidos en el objeto `Session`, el cual se crea cada vez que se quiera enviar el flujo multimedia a un cliente. Al crear varios objetos `Session` hay un conflicto con la cámara y deja de funcionar la transmisión.

B.2. Envío del flujo multimedia a mas de un destino

En este punto vamos a desarrollar en profundidad como adaptamos libstreaming para poder enviar el flujo de la cámara y micrófono a mas de un destino. **Distinguimos dos casos, el reenvió de un flujo recibido y el envió de un flujo generado localmente.**

El reenvió de un flujo recibido a múltiples destinos, fue implementado parcialmente por los alumnos del curso 2018/19. Los datos se reciben por cuatro *sockets* UDP, RTP y RTCP de vídeo y RTP y RTCP de audio. Estos *sockets* de recepción están contenidos en un objeto de la clase `ReceiveSession`. A este objeto se le pueden añadir más *sockets* para reenviar los datos recibidos, es decir, implementa un servidor UDP que recibe y reenvía datos. La adición de estos *sockets* se hace por medio del objeto de la clase `RebroadcastSession`. Este objeto, el cual contiene la `ReceiveSession`, se configura al negociar con el servidor RTSP, receptor de los streams reemitidos, creando cuatro *sockets* para enviar los datos al servidor. Estos *sockets* se añaden al servidor UDP de la `ReceiveSession`, encargado de reenviar los datos.

Primero se adaptó la descripción SDP de los flujos de la `RebroadcastSession`. Esta, al igual que con el objeto `Session`, tenía que ser adaptada para describir un *streaming* por IPv6, como contamos en la Sección 7.1.2.

Además, los *sockets* de comunicación tenían que ser adaptados a Wifi Aware. Antes de crear los *sockets* RTP y RTCP, tanto los de recepción como los de envió, asociamos el *thread* que los va a crear, al objeto `Network` recibido por la API de Wifi Aware, el cual es diferente para cada pareja de dispositivos. En el punto

A.3, se explica en detalle como se crean este tipo de conexiones con Wifi Aware. Sin esta asociación no sería posible vincular los *sockets* a una dirección de Wifi Aware, ni recibir datos por esta red. Este objeto lo obtiene el servidor o el cliente al hacer la conexión de Wifi Aware. En el servidor, al gestionar más de un canal de comunicación, creamos una función para obtener el objeto `Network` asociado a un canal RTSP, así, según se atiende a un cliente podemos obtener el objeto `Network` de su conexión y asociar el *thread* encargado en crear los *sockets* de la `ReceiveSession`.

En cuanto al envío del flujo de la cámara y el micrófono locales a mas de un destino, los alumnos del año 2018/2019, hicieron adaptaciones en libstreaming para enviarlo a más de un destino, pero estas adaptaciones no pudieron llegar a probarlas y **no resolvieron el problema.** Como se comenta en el punto B.1, crearon dos clases, `VideoPacketizerDispatcher` y `AudioPacketizerDispatcher`, cuyo objetivo es hacer de intermediario entre los codificadores de vídeo y audio y las clases encargadas de empaquetar y enviar estos flujos.

Al negociar una transmisión RTSP se crea y configura un objeto `Session` de libstreaming. Este objeto se encarga de obtener el flujo de la cámara y micrófono y enviarlo al destino configurado. Como al utilizar varios objetos `Session`, iban a entrar en conflicto por leer simultáneamente del *hardware*, se introdujeron las clases `VideoPacketizerDispatcher` y `AudioPacketizerDispatcher` para que leyeran del *hardware* y copiaran lo leído a cada objeto `Session`. Esto teóricamente funciona, pero descubrimos los siguientes problemas:

- Estas clases estaban diseñadas según el patrón *Singleton*, y la pueden crear los objetos `Session` al transmitir. Tuvieron un error de implementación por el que no se implemento correctamente el patrón. **Tenían un problema de sincronización** en el código por el cual varios objetos `Session` intentaban crear a la vez un `VideoPacketizerDispatcher` o un `AudioPacketizerDispatcher` y en vez de crear uno se creaban varios, lo que provocaba una pérdida de recursos con los *thread* extra que no se podían detener y fallos por la competición en la lectura del *hardware*.

- Otro problema más importante, que no se había considerado, es que **la configuración y el control de la cámara estaba integrado en el objeto `Session`.** Al hacer pruebas y crear varios `Session`, intentaban configurar todos la cámara a la vez, provocando que no funcionase ninguno. Sumado a esto, el código de configuración y control de la cámara de libstreaming estaba deprecado y sospechábamos que el problema comentado anteriormente del desenfoque de la cámara, en el punto A.1.1, estaba relacionado con esto.

  Como solución, concluimos que **era mejor estudiar la nueva API de la cámara, Camara2 [37], e implementar una clase que siguiera el patrón de diseño *Singleton*.** Esta clase la llamamos `CameraController`. En esta clase podemos configurar una de las cámaras del dispositivo para que nos de su salida. La salida de la cámara puede distribuirla a varios objetos `Surface`. Un objeto `Surface` es un *buffer* especial para guardar imágenes "crudas" de la

cámara y puede tener uno, por ejemplo, el elemento de la interfaz encargado de mostrar lo que ve la cámara al usuario, lo que Android llama una *preview*. También, el objeto `MediaCodec`, encargado de codificar el vídeo en formato H264, proporciona una `Surface` de entrada que se le puede proporcionar a la API de la cámara.

En cuanto a la resolución, se puede obtener el flujo de la cámara a la mayor resolución soportada por el dispositivo siempre que solo haya un destino del flujo, es decir, una sola `Surface` asociada. Al usar más de una, se aplican restricciones a la resolución en función del tipo de *hardware* de la cámara y la clase que vaya a utilizar la `Surface` como origen de datos. Como nosotros usamos dos, la `Surface` de la *preview* y la `Surface` del `MediaCodec` incluimos en la clase una función para calcular la máxima resolución según el dispositivo, basándonos en la documentación de Android [51].

Tras implementar esta clase, eliminamos todo el código de la cámara del objeto `Session` e hicimos que la GUI se encargara de configurarla en el momento que el usuario comience a capturar vídeo y sonido. Para configurarla, se obtiene la `Surface` de la GUI para mostrar la *preview* y la del `MediaCodec` del `VideoPacketizerDispatcher`, se calculan y se establecen sus resoluciones, y se inicia la grabación.

Gracias a esto **pudimos resolver el problema del desenfoque e investigando como mejorar la calidad logramos que aumentara de 720p a 1080p.**

B.3. Adaptación de servidor y cliente RTSP a Wifi Aware

El servidor RTSP es un solo *thread* que utiliza la API de Java NIO. Consiste en un objeto `Selector` con el que se atienden las peticiones de *read*, *write* y *accept* sobre múltiples canales `SelectableChannel`. En la implementación de libstreaming consistía en un único `ServerSocketChannel` que se encargaba de las peticiones de *accept* y añadía al selector `SocketChannels` para *read* y *write*. Como la API de Wifi Aware aísla los *sockets* mediante los objetos `Network` (Sección A.3), había que cambiar la estructura a un `ServerSocketChannel` por dispositivo Wifi Aware, todo esto conservando la anterior para poder conectarse con servicios que no fueran de Wifi Aware, como libVLC. Creamos la clase `Connection` dentro del servidor para manejar mejor la información y gestionar el cierre de la conexión. Esta clase guarda el `PeerHandle` de la conexión (que identifica el otro extremo de la conexión de Wifi Aware), el `ServerSocketChannel` y los `SocketChannels` asociados, el objeto `Network` y `NetworkCapabilities` para crear o asociar nuevos *sockets* a la conexión y el objeto `NetworkRequest`, utilizado para la petición del objeto `Network`, para poder eliminar más tarde la solicitud del objeto `Network` del sistema, ya que podría producir perdidas de memoria [50].

En el servidor y cliente RTSP incluimos el código para establecer una conexión de Wifi Aware. Este código consiste en la creación de una solicitud de red de Wifi Aware al sistema y abrir un `ServerSocketChannel` en el servidor RTSP y un `Socket` en el cliente RTSP, siguiendo la referencia para crear conexiones de Wifi Aware explicada en la SecciónA.3. Para los `DatagramSockets` de RTP y RTCP, la API de Wifi Aware no daba ninguna referencia para su creación, por lo que, en el cliente, a la hora de crearlos para enviar un *stream*, asociamos el objeto `Network` al *thread* [49]. En el servidor lo hicimos de la misma forma, pero como los *sockets* no se creaban en el propio servidor RTSP, si no en las clases encargadas de transmitir el flujo, `Session` y `RebroadcastSession`, tuvimos que hacer que estas clases pudieran acceder a los objetos `Network` almacenados en el servidor RTSP.

B.4. Gestión de Wifi Aware

Al iniciar nuestra aplicación utilizamos la clase que creamos `WifiAwareViewModel`, para crear una sesión de Wifi Aware, la cual nos permite iniciar una sesión de *publisher* y *subscriber*. Esas sesiones se crean con las instancias de unas clases anónimas que implementan la interfaz `DiscoverySessionCallback`, la cual incluye como métodos los *callbacks* de la API de mensajes de Wifi Aware, *callbacks* como, por ejemplo:

- `onPublisherStarted`, llamado cuando el *publisher* publica su servicio.
- `onServiceDiscovered`, llamado cuando el *subscriber* descubre un *publisher*.
- `onMessageReceived`, llamado cuando el *subscriber* o *publisher* recibe un mensaje.

Hay una de estas clases implementadas dentro del `WifiAwareViewModel` para el *publisher* y otra para el *subscriber*, en las cuales se define toda su lógica. Para esta lógica seguimos el procedimiento explicado en el apéndice A.3, para crear conexiones cliente-servidor entre cada pareja de *subscriber-publisher*.

Como se comenta en la Sección 7.2, la creación de una conexión de Wifi Aware entre una pareja de dispositivos se complica cuando hay varios dispositivos intentando conectarse a la vez. En la documentación solamente se especifica como conectar dos dispositivos [38], pero no menciona como gestionar varios intentos simultáneos de conexión. Si no se lleva un control sobre esta concurrencia, descubrimos que, en ocasiones, se producen errores que impiden a los dispositivos conectarse.

Como solución, se nos ocurrió secuencializar las conexiones entre parejas de dispositivos. Al recibir el *callback* `onServiceDiscovered`, en el objeto que implementa la lógica del *subscriber*, si se esta creando una conexión con un *publisher*, almacenamos los datos de esta en una cola para poder procesar la conexión más tarde y si no, la comenzamos a procesar en el momento. El procesamiento de la conexión consiste en el envío de un mensaje al *publisher* y la espera de su respuesta. Una vez el *publisher* responde, se crea una nueva instancia de cliente RTSP, `RtspClient`, para

comunicarse con el servidor RTSP del *publisher* y se procesa la siguiente conexión pendiente de la cola. Antes de procesar la siguiente conexión pendiente, consideramos conveniente introducir un pequeño *delay* en el *thread* que procesa los *callbacks* de Wifi Aware, ya que descubrimos haciendo pruebas que mejora la reducción de errores en la creación de conexiones.

En el objeto que implementa la lógica del *publisher*, hacemos el mismo tratamiento con la cola y el *delay*. Una vez se recibe el *callback* `onMessageReceived` de un nuevo *subscriber*, si se esta procesando una conexión se añade a la cola y si no se procesa en el momento. Este procesamiento incluye la adición al servidor RTSP de los datos para crear un *socket* con el nuevo cliente, y el envió de un mensaje de confirmación al *subscriber*. Tras el envío se procesa la siguiente conexión esperando un *delay*.

Bibliografía

[1] Shuttleworth Foundation. URL: https://www.shuttleworthfoundation.org/.

[2] *libstreaming*. Ver. 2.0. 8 de feb. de 2019. URL: https://github.com/fyhertz/libstreaming (Último acceso: 12-06-2021).

[3] Iván Gulyk y Noel José Algora Igual. «Device to Device Streaming en dispositivos móviles». thesis. UCM, 2018-2019. URL: https://eprints.ucm.es/id/eprint/56536/1/1138503790-303288_IV%C3%81N_GULYK_Memoria_TFG_Device_to_Device_Streaming_en_Dispositivos_m%C3%B3viles_3940146_1463809280.pdf (Último acceso: 12-06-2021).

[4] Francisco Calero Velasco, Sergio Manzanaro Caraballo y David Salido Camacho. «Crowdstreaming: transmisión de vídeo y audio por una red ad hoc de teléfonos móviles». thesis. UCM, 2020-2021. URL: https://eprints.ucm.es/id/eprint/61913/1/SALIDO_CAMACHO_Crowdstreaming_transmision_de_video_y_audio_por_una_red_ad_hoc_de_telefonos_moviles_4398577_1858548838.pdf (Último acceso: 12-06-2021).

[5] Witness Org. URL: https://www.witness.org/.

[6] Freedom of the Press Foundation. *SecureDrop*. URL: https://securedrop.org/.

[7] «Wifi Aware Technology Overview.» En: *Wifi Alliance* (jul. de 2020).

[8] Lili Wei, Rose Qingyang Hu, Yi Qian y Geng Wu. «Enable device-to-device communications underlaying cellular networks: challenges and research aspects». En: *IEEE Communications Magazine* 52.6 (2014), págs. 90-96. DOI: 10.1109/MCOM.2014.6829950. (Último acceso: 12-06-2021).

[9] Martin Woolley. «Exploring Bluetooth 5 – Going the Distance». En: *Bluetooth Blog* (feb. de 2017). URL: https://www.bluetooth.com/blog/exploring-bluetooth-5-going-the-distance/ (Último acceso: 12-06-2021).

[10] Kai Ren. «Higher Speed How Fast Can It Be?» En: *Bluetooth Blog* (feb. de 2017). URL: https://www.bluetooth.com/blog/exploring-bluetooth-5-how-fast-can-it-be/ (Último acceso: 12-06-2021).

[11] Dave Hollander. «Bluetooth Device Networks Enable a Smarter IoT». En: *Bluetooth Blog* (ago. de 2017). URL: https://www.bluetooth.com/blog/revolutionizing-the-iot/ (Último acceso: 12-06-2021).

[12] Bluetooth SIG. *LE Audio*. URL: https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/le-audio/ (Último acceso: 12-06-2021).

[13] Ian Poole. «IEEE 802.11n WLAN Standard». En: *Electronics Notes*. URL: https://www.electronics-notes.com/articles/connectivity/wifi-ieee-802-11/802-11n.php (Último acceso: 12-06-2021).

[14] Alexandr Vitosinschi. *802.11ac wireless throughput testing and validation guide*. Mar. de 2018. URL: https://www.cisco.com/c/en/us/support/docs/wireless-mobility/wireless-lan-wlan/212892-802-11ac-wireless-throughput-testing-and.html (Último acceso: 12-06-2021).

[15] National Instruments Corp. *Introduction to 802.11ax High-Efficiency Wireless*. Ago. de 2020. URL: https://www.ni.com/es-es/innovations/white-papers/16/introduction-to-802-11ax-high-efficiency-wireless.html (Último acceso: 12-06-2021).

[16] Wi-Fi Alliance. *Wi-Fi CERTIFIED Wi-Fi Direct*. Oct. de 2010. URL: https://www.wi-fi.org/system/files/wp_Wi-Fi_Direct_20101025_Industry.pdf (Último acceso: 12-06-2021).

[17] Android Developers. *WifiP2pGroup*. URL: https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pGroup.html (Último acceso: 12-06-2021).

[18] Shekhar Jain, Yi Zhang y Luiz A. DaSilva. *LTE Standard-Compliant D2D Communication: Software-defined Radio Implementation and Evaluation*. Abr. de 2019. URL: https://arxiv.org/pdf/1904.12934v1.pdf (Último acceso: 12-06-2021).

[19] Qualcomm Technologies. *LTE Direct Trial*. Feb. de 2015. URL: https://www.qualcomm.com/media/documents/files/lte-direct-trial-whitepaper.pdf (Último acceso: 12-06-2021).

[20] Qualcomm Technologies. *LTE Direct Always-on Device-toDevice Proximal Discovery*. Ago. de 2014. URL: https://www.qualcomm.com/media/documents/files/lte-direct-always-on-device-to-device-proximal-discovery.pdf (Último acceso: 12-06-2021).

[21] «SCTP: el protocolo de transporte fiable y basado en mensajes». En: *Ionos* (jun. de 2019). URL: https://www.ionos.es/digitalguide/servidores/know-how/sctp/ (Último acceso: 12-06-2021).

[22] H.Schulzrinne, A.Rao y M.Stiemerling. *Real-Time Streaming Protocol Version 2.0*. RFC 7826. Internet Engineering Task Force (IETF), dic. de 2016. URL: https://tools.ietf.org/html/rfc7826 (Último acceso: 12-06-2021).

[23] M. Handley, V. Jacobson y C. Perkins. *SDP: Session Description Protocol*. RFC 4566. Jul. de 2006. URL: https://tools.ietf.org/html/rfc4566#page-3 (Último acceso: 12-06-2021).

[24] Wikipedia. *Overlay network*. URL: https://en.wikipedia.org/wiki/Overlay_network (Último acceso: 12-06-2021).

[25] A. Bakker y V. Grishchenko. *Peer-to-Peer Streaming Peer Protocol (PPSPP)*. RFC 7574. Internet Engineering Task Force (IETF), jul. de 2015. URL: https://www.ietf.org/rfc/rfc7574.txt.pdf (Último acceso: 12-06-2021).

[26] Ramón Jesús Millán Tejedor. «Qué es... WebRTC (Web real-time communications)». En: *ramonmillan.com* (2014). URL: https://www.ramonmillan.com/tutoriales/webrtcbeneficios.php (Último acceso: 12-06-2021).

[27] Oracle Corp. *Developing WebRTC-enabled Android Applications*. URL: https://docs.oracle.com/cd/E55119_01/doc.71/e55126/wd_androidapps.htm#WSEWD436 (Último acceso: 12-06-2021).

[28] Google LLC. *WebRTC Android development*. URL: https://webrtc.googlesource.com/src/+/refs/heads/master/docs/native-code/android/index.md (Último acceso: 12-06-2021).

[29] VideoLan Org. *VLC media player*. URL: https://www.videolan.org/vlc/index.es.html (Último acceso: 12-06-2021).

[30] VideoLan Wiki. *Live555*. Abr. de 2019. URL: https://wiki.videolan.org/Live555/ (Último acceso: 12-06-2021).

[31] Juan Navarro. «RTP (II): Streaming with FFmpeg». En: *Kurento Technologies* (abr. de 2019). URL: http://www.kurento.org/blog/rtp-ii-streaming-ffmpeg (Último acceso: 12-06-2021).

[32] FFmpeg. *Documentation*. URL: https://ffmpeg.org/documentation.html (Último acceso: 12-06-2021).

[33] Mónica Mena Roa. «Android e iOS dominan el mercado de los smartphones». En: *Statista* (jul. de 2020). URL: https://es.statista.com/grafico/18920/cuota-de-mercado-mundial-de-smartphones-por-sistema-operativo/ (Último acceso: 12-06-2021).

[34] Cameron Chapman. «Explorando los Principios Gestalt del Diseño». En: *Toptal Design Blog*. URL: https://www.toptal.com/designers/ui/exploring-the-gestalt-principles-of-design (Último acceso: 12-06-2021).

[35] Wikipedia. *Hop (networking)*. URL: https://en.wikipedia.org/wiki/Hop_(networking) (Último acceso: 12-06-2021).

[36] M. Handley, V. Jacobson y C. Perkins. *SDP: Session Description Protocol*. RFC 4566. Internet Engineering Task Force (IETF), jul. de 2006. URL: https://datatracker.ietf.org/doc/html/rfc4566 (Último acceso: 12-06-2021).

[37] Android Developers. *Camera2*. URL: https://developer.android.com/training/camera2 (Último acceso: 12-06-2021).

[38] Android Developers. *Descripción general del reconocimiento de Wi-Fi*. URL: https://developer.android.com/guide/topics/connectivity/wifi-aware#obtain_a_session (Último acceso: 12-06-2021).

[39] Android Developers. *Descripción general de las funciones y API*. URL: https://developer.android.com/about/versions/12/features#wifi-aware-enhancements (Último acceso: 12-06-2021).

[40] T. Clausen y P. Jacquet. *Optimized Link State Routing Protocol (OLSR)*. RFC 3626. Internet Engineering Task Force (IETF), oct. de 2003. URL: https://datatracker.ietf.org/doc/html/rfc3626 (Último acceso: 12-06-2021).

[41] Wikipedia. *Delay-tolerant networking*. URL: https://en.wikipedia.org/wiki/Delay-tolerant_networking (Último acceso: 12-06-2021).

[42] Guardian Project. *ObscuraCam: The Privacy Camera*. URL: https://guardianproject.info/apps/org.witness.sscphase1/ (Último acceso: 12-06-2021).

[43] B. Weis, S. Rowles y T. Hardjono. *The Group Domain of Interpretation*. RFC 6407. Internet Engineering Task Force (IETF), oct. de 2011. URL: https://datatracker.ietf.org/doc/html/rfc6407 (Último acceso: 12-06-2021).

[44] Guardian Project. *CameraV app and the InformaCam System*. URL: https://guardianproject.github.io/informacam-guide/en/InformacamGuide.html (Último acceso: 12-06-2021).

[45] Mohamed Hefeeda y Kianoosh Mokhtarian. *svcAuth*. URL: https://nsl.cs.sfu.ca/wiki/index.php/svcAuth (Último acceso: 12-06-2021).

[46] Wikipedia. *Aaron Swartz*. URL: https://es.wikipedia.org/wiki/Aaron_Swartz (Último acceso: 14-06-2021).

[47] VideoLan Wiki. *Documentation:Streaming HowTo/Streaming over IPv6*. Nov. de 2014. URL: https://wiki.videolan.org/Documentation:Streaming_HowTo/Streaming_over_IPv6/#Limitations (Último acceso: 12-06-2021).

[48] H. Schulzrinne, A. Rao, R. Lanphier, M. Westerlund y M. Stiemerling. *Real-Time Streaming Protocol Version 2.0*. RFC 7826. Internet Engineering Task Force (IETF), dic. de 2016. URL: https://datatracker.ietf.org/doc/html/rfc7826 (Último acceso: 12-06-2021).

[49] Android Developers. *Network*. URL: https://developer.android.com/reference/android/net/Network (Último acceso: 12-06-2021).

[50] Android Developers. *ConnectivityManager NetworkCallback*. URL: https://developer.android.com/reference/android/net/ConnectivityManager.NetworkCallback) (Último acceso: 12-06-2021).

[51] Android Developers. *CameraDevice*. URL: https://developer.android.com/reference/android/hardware/camera2/CameraDevice#regular-capture (Último acceso: 12-06-2021).