



**POLITÉCNICA**

"Ingeniamos el futuro"

CAMPUS  
DE EXCELENCIA  
INTERNACIONAL



# **Graduado en Máster de Ingeniería Informática**

Universidad Politécnica de Madrid

Escuela Técnica Superior de  
Ingenieros Informáticos

## **TRABAJO FIN DE MÁSTER**

(Desarrollo de aplicaciones Android en entornos de  
proximidad)

Autor: Jorge Amorós Anaya      MADRID, JUNIO 2019

Tutor: Luis Mengual Galán

# Índice General

<b>Resumen del trabajo</b>	<b>8</b>
Español	8
Abstract in English	9
<b>Capítulo 1 : Introducción y motivación</b>	<b>10</b>
1.1 Motivación	10
Ilustración 1: Google Nearby.	11
1.2 Trabajos previos	12
<b>Capítulo 2: Documentos de proyecto</b>	<b>13</b>
2.1 Objetivo	13
2.2 Descripción	14
2.3 Alcance	14
2.4 Entregables	14
2.5 Riesgos y plan de contingencia	15
2.6 Tecnologías a utilizar	15
2.7 Estimación de tiempo	16
Tabla 1: Estimación de tiempo	17
2.8 Calendario de trabajo	17
Tabla 2: Horarios de trabajo	18
<b>Capítulo 3: Gestión y análisis del proyecto</b>	<b>19</b>
3.1 Objetivo del trabajo	20
3.2 Especificación de requisitos inicial	20
3.2.1 Requisitos funcionales	21
Tabla 3: Requisitos funcionales del sistema	23
3.2.2 Requisitos NO funcionales	23
Tabla 4: Requisitos no funcionales del sistema	25
3.3 Re-Especificación de requisitos	25
Tabla 5: Tabla de re-especificación de requisitos	26
3.3 Re-Estimación de los tiempos	26
Tabla 6: Re-estimación de los tiempos	27
Ilustración 2: Comparación entre estimaciones	27
3.4 Casos de uso	28
3.4.1 Acciones que no requieren conexión	28

Ilustración 3: Primer caso de uso. Acciones sin conexión	28
3.4.2 Conexión entre dos usuarios	29
Ilustración 4: Caso de uso 2. Conexión entre usuarios	29
3.4.3 Petición de chat	30
3.4.4 Envío de archivos	31
Ilustración 6: Caso 4. Envío de archivo múltiple	31
<b>Capítulo 4: Google Nearby</b>	<b>32</b>
Ilustración 7: Wifi vs Bluetooth	34
4.1 Permisos	34
Ilustración 8: Permisos para la API Nearby	35
4.1 Estrategia	35
4.1.1 P2P Cluster	35
4.1.1 P2P Star	36
4.1.1 P2P Point to point	36
4.3 Anunciar y descubrir	37
4.3.1 Anunciar	37
Ilustración 9: Función startAdvertising()	37
4.3.2 Descubrir	38
Ilustración 10: Función startDiscovery()	39
4.3 Manejo de las conexiones	39
4.3 Intercambio de datos	39
<b>Capítulo 5 : Diseño de proyecto</b>	<b>41</b>
5.1 Introducción	41
5.2 Arquitectura del sistema. Modelo Vista Controlador	41
Ilustración 11: Modelo Vista y Controlador	42
Ilustración 12: MVC vs MVP	43
Ilustración 13: Desacople vista y controlador	44
5.3 Base de datos	45
Ilustración 14: Base de datos	45
5.4 Estructura del proyecto	46
Ilustración 15: Estructura del proyecto	46
<b>Capítulo 6. Desarrollo del proyecto</b>	<b>48</b>
6.1 Introducción	48
6.2 Librerías utilizadas	49
6.3 Manifiesto.xml	50
6.3 Implementación de la API Nearby en el Framework	52
6.3.1 Persistencia	53

## Trabajo de fin de máster

Ilustración 16: Clase aplicación	54
6.3.2 Implementación	55
6.3.2.1 Clase "Application"	55
Ilustración 17: UML clase Aplicación	55
6.3.2.2 Clase de servicios	56
Ilustración 18: MyNearbyConnectionService	57
6.3.3.3 Conexión persistente a través de diferentes actividades.	58
Ilustración 19: Interfaces de funcionalidad	59
6.3.3 MyNearbyConnectionService en detalle	60
Tabla 7: Atributos de clase de servicio	61
Tabla 8: Métodos de la clase servicios	62
6.3.4 El manejador de peticiones	63
Tabla 9: Métodos del manejador de peticiones	64
6.4 La actividad Base	64
Ilustración 20: ToolBar en android	65
6.5 Peticiones	65
Tabla 10: Atributos comunes de una petición	66
Tabla 11: Métodos comunes de una petición	67
6.5.1 Petición de amistad	68
Ilustración 21: Flujo de datos en una petición de amistad	68
6.5.1.1 Tipos de peticiones	68
Tabla 12: Pasos o tipos de payloads de petición de amistad	69
6.5.2 Petición de chat	69
Ilustración 22: Flujo de datos en una petición de chat	70
6.5.2.1 Tipos de peticiones	70
Tabla 13: Pasos o tipos de payloads para peticiones de chat	71
6.5.3 Petición de envío de archivo	71
Ilustración 23: Flujo de datos en una petición de archivo	72
6.5.3.1 Tipos de peticiones	72
Tabla 14: Pasos o tipos de payload para peticiones de envío de archivos	73
6.5.4 Petición de envío de archivo múltiple	73
Ilustración 24: Flujo de datos en petición de archivo multiple	74
6.5.4.1 Tipos de peticiones	74
Tabla 15: Pasos o tipos de payloads en envío de archivo múltiple	75
6.6 Vistas, controladores e interfaz	75
6.6.1 Custom Dialog	76
Ilustración 25: Ejemplo de dialog positivo	76
6.6.2 Módulo Home	77

## Trabajo de fin de máster

Ilustración 26: Diagrama de clases del módulo Home	78
6.6.2.1 Interfaz	78
Ilustración 27: Vista y funcionalidad de la pantalla home	79
Ilustración 28: Opción files	81
6.6.2.2 Dialogs y notificaciones	82
Ilustración 29: Ejemplos de diálogos en Home	82
Ilustración 30: Ejemplo de notificación en Home	83
6.6.2.3 Controlador	84
6.6.3 Módulo lista de amigos	88
Ilustración 31: Diagrama de clases de la lista de amigos	89
6.6.3.1 Interfaz	90
Ilustración 32: Vista y funcionalidad de la pantalla de amigos	90
6.6.3.2 Dialogs	91
6.6.3.3 Controlador	91
6.6.4 Módulo Chat	92
Ilustración 33: Diagrama de clases del módulo chat	93
6.6.4.1 Interfaz	94
Ilustración 34: Vistas y funcionalidad de la pantalla de chat	94
6.6.4.3 Dialogs	94
6.6.4.4 Controlador	94
6.6.5 Módulo Opciones	96
Ilustración 35: Diagrama de clases del módulo de opciones	97
6.6.5.1 Interfaz	98
Ilustración 36: Vistas y funcionalidades de la pantalla de opciones	98
6.6.5.3 Dialogs	99
6.6.5.4 Controlador	99
6.7 Servicios de seguridad	100
6.7.1 Integridad de los datos	101
Ilustración 37: Integridad de los datos	101
<b>7.Conclusiones</b>	<b>103</b>
<b>Bibliografía</b>	<b>104</b>
<b>Anexo</b>	<b>105</b>
Clase de servicios	106
Clase Manejadora de peticiones	118
Clase Base	118

## Índice de figuras

1. *Ilustración 1. Google Nearby.*
2. *Ilustración 2. Comparación entre estimaciones*
3. *Ilustración 3. Caso de uso 1. Acciones sin conexión*
4. *Ilustración 4. Caso de uso 2. Conexión entre usuarios*
5. *Ilustración 5. Caso de uso 3. Petición de chat*
6. *Ilustración 6. Caso de uso 4. Envío de archivo múltiple*
7. *Ilustración 7. Wifi vs Bluetooth*
8. *Ilustración 8. Permisos para API Nearby*
9. *Ilustración 9. Función startAdvertising()*
10. *Ilustración 10. Función startDiscovery()*
11. *Ilustración 11. Modelo, Vista y Controlador*
12. *Ilustración 12. MVC vs MVP*
13. *Ilustración 13. Desacople Vista y Controlador*
14. *Ilustración 14. Tablas en Base de datos*
15. *Ilustración 15. Estructura del proyecto*
16. *Ilustración 16. Contexto de la clase aplicación*
17. *Ilustración 17. UML clase Aplicación*
18. *Ilustración 18. MyNearbyConnectionService*
19. *Ilustración 19. Interfaces de funcionalidad*
20. *Ilustración 20. ToolBar en android*
21. *Ilustración 21. Flujo de datos en una petición de amistad*
22. *Ilustración 22. Flujo de datos en una petición de chat*
23. *Ilustración 23. Flujo de datos en una petición de archivo*
24. *Ilustración 24. Flujo de datos en una petición de archivo múltiple*
25. *Ilustración 25. Ejemplo de Dialog positivo*
26. *Ilustración 26 : Diagrama de clases del módulo Home*
27. *Ilustración 27. Vista y funcionalidad de la pantalla home.*
28. *Ilustración 28. Opción Files*
29. *Ilustración 29. Ejemplos de diálogos en Home*
30. *Ilustración 30. Ejemplos de notificación en Home*
31. *Ilustración 31. Diagrama de clases de la lista de amigos*

- 32. *Ilustración 32. Vista y funcionalidad de la pantalla de amigos*
- 33. *Ilustración 33. Diagrama de clases del módulo chat*
- 34. *Ilustración 34. Vistas y funcionalidad de la pantalla de chat*
- 35. *Ilustración 35. Diagrama de clases del módulo de chat.*
- 36. *Ilustración 36. Vistas y funcionalidades de la pantalla de opciones*
- 37. *Ilustración 37. Integridad de los datos*

## **Índice de tablas**

- 1. *Tabla 1. Estimación de tiempo*
- 2. *Tabla 2. Horarios de trabajo*
- 3. *Tabla 3. Requisitos funcionales del sistema*
- 4. *Tabla 4. Requisitos no funcionales del sistema*
- 5. *Tabla 5. Re-especificación de requisitos*
- 6. *Tabla 6. Re-estimación de los tiempos*
- 7. *Tabla 7. Atributos de clase de servicio*
- 8. *Tabla 8. Métodos de la clase servicios*
- 9. *Tabla 9 . Métodos del manejador de peticiones*
- 10. *Tabla 10. Atributos comunes de una petición*
- 11. *Tabla 11. Métodos comunes de una petición*
- 12. *Tabla 12. Pasos o tipos de payloads de petición de amistad*
- 13. *Tabla 13. Pasos o tipos de payloads de petición de chat*
- 14. *Tabla 14. Pasos o tipos de payloads de petición de envío de archivos*
- 15. *Tabla 15. Pasos o tipos de payloads de petición de envío de archivo múltiple*

# Resumen del trabajo

## Español

En este Trabajo de Fin de Máster hemos querido centrarnos en un tema muy popular en la actualidad como es el desarrollo de una aplicación para entornos móviles Android. El objetivo de la aplicación será ofrecer servicios en entornos de proximidad, tales como el envío de archivos punto a punto, o de un punto a varios receptores, chats cifrados punto a punto o el envío de mensajes por medio de notificaciones.

Por entorno de proximidad entendemos todo aquello que se encuentra próximo y con lo que podemos entablar una conexión a través de tecnologías como Bluetooth o Wifi con un rango aproximado de 100 metros. En esta aplicación no utilizaremos los datos móviles(GPRS, 3G, 4G) bajo ningún concepto, y es que una de las principales ventajas es el ahorro de datos cuando enviamos grandes archivos, así como la posible comunicación entre personas en entornos donde las redes de telecomunicaciones están saturadas.

Para añadirle valor al trabajo, se han cifrado los mensajes enviados entre usuarios mediante el uso de certificados de clave pública y privada, cifrados en Hash y claves AES.

Para terminar , en este trabajo se parte con una base cercana nula sobre el desarrollo en el framework de Android. Se ha tenido que hacer un esfuerzo adicional en el desarrollo para alcanzar un nivel mínimo con el cual poder crear esta aplicación. Pero incluso así se necesita una cierta base para comprender ciertas partes de esta memoria, ya que las partes más básicas de Android relacionadas con las actividades, las vistas, los diálogos y su creación dinámica no van a ser explicadas al entender que el punto principal del trabajo es entender las tecnologías de proximidad y en concreto la API de Google que las proporciona denominada Nearby.



## Abstract

In this Final Master Project we want to focus in a currently very popular topic. Nowadays, the development of applications for Android mobile environments. The aim of the application will be to offer services in proximity environments, such as sending point-to-point files, or from one point to several receivers, point-to-point encrypted chats or sending messages through notifications.

By proximity environment we understand everything that is close and with which we can establish a connection through technologies such as Bluetooth or Wifi (within 100 metres). In this application we will not use mobile data (GPRS, 3G or 4G) under any circumstances, as is one of the main advantages, saving data while sending large files as well as the possible communication between people in environments where telecommunications networks are saturated.

To add value to this Master project, messages sent between users have been encrypted using public and private key certificates, encrypted in Hash and AES keys.

Finally, it has to be highlighted the fact that at the beginning of this Final Master Project my experience and previous knowledge about development using the Android framework was nonexistent. Therefore, an additional effort has been done to reach an starting level on Android development. However, to fully comprehend this document, it is required to have some context on Android development, since the most basic parts of Android such as activities, views, dialogs and their dynamic creation are not going to be explained. Assuming that the main point of the work is to learn about the proximity technologies and in particular the Google API Nearby and the functionality it provides.

# Capítulo 1 : Introducción y motivación

A continuación, vamos a introducir la idea principal del proyecto, el entorno android y los objetivos perseguidos con la realización de este trabajo. La idea al terminar de leer este capítulo es tener una concepción general sobre los temas a tratar a lo largo de este documento.

## 1.1 Motivación

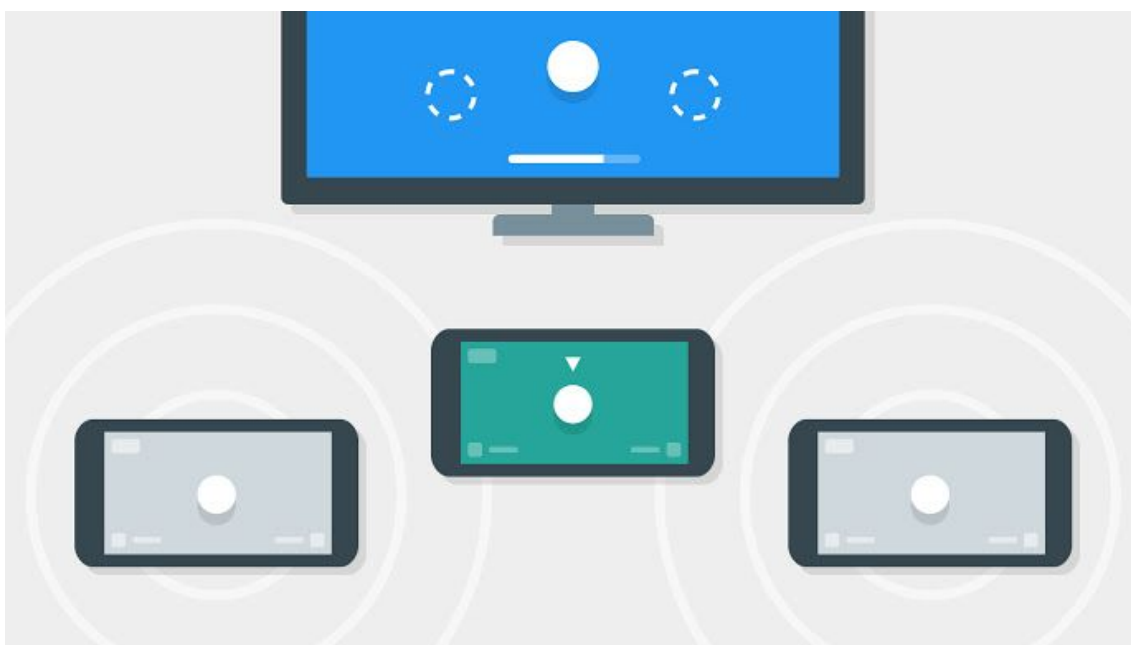
No voy a presentar Android a estas alturas en 2019. El sistema operativo móvil que domina en el mercado móvil y creado por Google allá por 2007 <sup>1</sup>. No puedo ocultar mi pasión por la tecnología, y especialmente por los móviles. Aprender a desarrollar aplicaciones en android siempre ha sido una de mis tareas pendientes aprovechando que se programa en Java y que es un lenguaje que ya conozco. Por desgracia, por una cosa o por la otra nunca he tenido el tiempo suficiente para poder ponerme con ello. Cuando terminé el grado ya rondó por mi cabeza realizar un TFG relacionado con Android pero me propusieron una trabajo muy interesante relacionado con la realidad aumentada y el dispositivo Microsoft Hololens y no pude rechazarlo. Con este TFM he tenido la oportunidad de aprender, por fin, parte de los entresijos de este sistema

---

<sup>1</sup> "Android, el sistema operativo." [https://www.android.com/intl/es\\_es/](https://www.android.com/intl/es_es/).

operativo. Y digo entresijos por solo he rascado la superficie de las posibilidades que nos ofrece este framework móvil. Dominarlo debe ser extremadamente complejo, pero estoy bastante satisfecho con los resultados logrados a lo largo de estos 4 meses partiendo desde una base prácticamente nula.

En este proyecto el objetivo será conseguir el envío de archivos y mensajes cifrados entre dos dispositivos sin la necesidad de la red de internet. Esto se va a conseguir mediante la implementación de la API de Google Nearby Connections. Esta API implementa por encima los servicios de bluetooth, wifi e infrarrojos de un dispositivo móvil de manera que es capaz de activar uno u otro para recibir datos de manera transparente al usuario. De esta manera se junta mi interés en desarrollar en Android con el reciente interés que tengo por la ciberseguridad, cifrando la información mediante el uso de certificados de clave pública y privada.



*Ilustración 1: Google Nearby.*

En conclusión, en el mundo en el que vivimos, y más concretamente en el mundo de la informática, Android ocupa un lugar muy importante al formar los smartphones una parte vital de nuestro día a día. Aprender a realizar aplicaciones puede ser una habilidad muy interesante de cara a un futuro. Aprender unos mínimos en cuanto a la securización de los datos transmitidos también es muy interesante, pues últimamente el tema de la ciberseguridad está muy de moda debido a los diversos ataques informáticos contra las grandes empresas del sector.

## 1.2 Trabajos previos

Como hemos comentado se parte prácticamente desde cero. Lo único conocido sería el lenguaje de programación java así como el lenguaje SQL para bases de datos. Aunque dominar el lenguaje de Java es una gran ventaja, el framework de Android sigue siendo muy grande, complejo y difícil de dominar. En apenas cuatro meses de trabajo apenas podremos rascar parte de sus posibilidades.

# Capítulo 2: Documentos de proyecto

En este capítulo vamos a tratar aspectos como el alcance, entregables o riesgos que podamos encontrarnos. La lista es la siguiente :

- Objetivo y justificación
- Descripción
- Alcance
- Entregables
- Riesgos y planes de contingencia
- Descomposición de tareas
- Tecnologías
- Estimación del trabajo
- Calendario del trabajo
- Ciclo de vida

## 2.1 Objetivo

El objetivo del proyecto es la realización de una aplicación que pueda servir para comunicarte con otras personas si necesitas de una red móvil y proporcionar

privacidad a esa información mediante el cifrado de datos. Tanto para hablar por escrito como para enviar archivos. Esto puede ser útil en ambientes donde la cobertura móvil es inexistente como en las grandes acumulaciones de personas que se dan en conciertos y festivales. También es interesante si se quieren enviar archivos muy grandes y ahorrar tarifa de datos.

## 2.2 Descripción

Se partirá de la comprensión de la API de Google Nearby Connections entendiendo sus limitaciones (rango y banda ancha para transmisión de archivos) para crear una aplicación consistente y tolerante a lo fallos de conexión que puedan darse con este tipo de tecnologías inalámbricas. Adicionalmente se añadirá cifrado a estas transmisiones entre dispositivos. También se intentará crear una aplicación visualmente amigable mediante el uso de Material Design. Esto solo sería posible en caso de disponer del tiempo necesario para su implantación.

## 2.3 Alcance

Nos daremos por satisfechos en el momento en que seamos capaz de conectarnos a un dispositivo, enviar un archivo cifrado correctamente y descifrarlo en el dispositivo final de manera transparente al usuario. Aspectos como la múltiple conexión con diversos dispositivos y la gestión de notificaciones va a depender de la estabilidad de la conexión que pueda proporcionar el API y del tiempo disponible.

## 2.4 Entregables

Los entregables del proyecto serán los siguiente:

- Aplicación en formato .apk de android.
- Memoria final explicando la arquitectura y pasos en el desarrollo del trabajo.

## 2.5 Riesgos y plan de contingencia

- Errores en la estimación de las diferentes fechas del proyecto.

Debido a la poca experiencia desarrollando en Android, lo normal serían desviaciones en las fechas estimadas de finalización de tareas.

→ Plan a seguir:

◆ Recalculado de fecha sobre la marcha.

- Errores en el entorno de desarrollo (Android Studio)

Similar a la anterior en cuanto a la falta de rodaje en esta materia y debido a que se desarrolla en diferentes plataformas es posible que en algún momento del proyecto nos enfrentemos a problemas técnicos con estas herramientas y se puedan producir pérdidas de datos.

→ Plan a seguir:

◆ Utilización de gitlab para almacenado del proyecto en la nube.

- Errores en la comprensión del API Nearby

Es posible que las capacidades del API hayan sido malinterpretadas o que el rendimiento de esta no sea el esperado para lo que se intenta desarrollar.

→ Plan a seguir:

◆ Reformulación de los requisitos de la aplicación.

## 2.6 Tecnologías a utilizar

A continuación se resumen las principales tecnologías utilizadas a lo largo del proyecto:

- Android Studio

Entorno de desarrollo de la aplicación.

## Trabajo de fin de máster

- SQLite android

Framework de base de datos adaptado en Android. Muy ligero.

- Api de Google Nearby

APi que permite intercambio de datos entre dispositivos mediante tecnologías inalámbricas.

- Sistema

Se ha desarrollado tanto en windows, como en unix como en un sistema MacOS. Esto ha sido posible gracias al almacenamiento del proyecto en un repositorio Git. En nuestro caso se ha utilizado el dominio de Gitlab.com.

- Librerías externas

Para facilitar ciertas tareas se implementarán librerías de terceros.

## 2.7 Estimación de tiempo

A continuación el desglose estimado del tiempo necesario para la consecución de las diversas tareas a realizar :

Tarea	Trabajo	Duración	Comienzo	Fin
Trabajo de fin de máster	486h	87 días	14 Febrero	14 Junio
Documentación	50h	87 días	14 Febrero	14 Junio
Comprensión y análisis del entorno de seguridad y las librerías JCE de java y del entorno de desarrollo de Android Studio.	75h	21 días	14 Febrero	14 Marzo



## Trabajo de fin de máster

Comprensión de los principios básicos de Seguridad: Servicios, mecanismos y protocolos	50	11 días	15 Marzo	29 Marzo
Comprensión de las APIS de desarrollo para las comunicaciones de proximidad Nearby Connections API, wifi-direct, bluetooth	87h	14 días	29 Marzo	17 abril
Desarrollo de la infraestructura de comunicaciones y seguridad para los intercambios entre el Cliente móvil y el Sistema de Registro y entre los sistemas móviles entre sí.	100h	23 días	15 Abril	15 Mayo
Desarrollo de la Interfaz del cliente Android	75h	14 días	16 Mayo	4 Junio
Evaluación y pruebas	49h	9 Días	5 Junio	16 Junio

*Tabla 1: Estimación de tiempo*

## 2.8 Calendario de trabajo

El alumno trabaja actualmente 8 horas en horario de 8:00 a 16:00. Posteriormente realiza otras actividades y llega a casa a las 19:00. El plan de trabajo dedicado al TFM pasa por dedicarle 4h diarias, entre las 19:00 y las 23:00 en horario de Lunes a Viernes. Los fines de semana podrá trabajar 9 horas repartidas la mitad por la mañana y el resto por la tarde. Pudiendo realizar horas extraordinarias si fuese necesario. Teniendo 87 días que vienen a ser unas 13 semanas. Tenemos unos 65 días laborables donde haremos 260 horas y 26 días donde acumularemos otras 234 horas dando un total de 494 que restando imprevistos cuadra con lo estipulado.

TFM
TRABAJO
OTRAS ACTIVIDADES

## Trabajo de fin de máster

Horarios	Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo
8:00 - 9:00							
9:00 - 10:00							
10:00 - 11:00							
11:00 - 12:00							
12:00 - 13:00							
14:00 - 15:00							
15:00 - 16:00							
16:00 - 17:00							
17:00 - 18:00							
19:00 - 20:00							
20:00 - 21:00							
21:00 - 22:00							
22:00 - 23:00							

*Tabla 2: Horarios de trabajo*

# Capítulo 3: Gestión y análisis del proyecto

La gestión de proyectos es un conjunto de metodologías para planificar y dirigir los procesos de un proyecto. Un proyecto comprende un cúmulo específico de operaciones diseñadas para lograr un objetivo con un alcance, recursos, inicio y final establecidos. Los objetivos de la gestión de proyectos son:

- Gestionar el inicio y la evolución de un proyecto;
- Controlar y responder ante problemas que surjan durante un proyecto;
- Facilitar la finalización y aprobación del proyecto.

Al ser un proyecto corto, de apenas unos pocos meses, en este capítulo vamos a desarrollar los objetivos principales de este proyecto. Al tratarse de una aplicación desarrollada desde cero, incluiremos una especificación de requisitos que contendrá una descripción completa del sistema que se va a desarrollar. Adicionalmente se hará una trazabilidad de estos requisitos a lo largo del tiempo, es decir, como hemos vuelto a planificar estos en función del tiempo, fallos de comprensión de la API a la hora de implementar funcionalidades, dificultades encontradas y cualquier otro elemento que afecte a dichos requisitos.

### 3.1 Objetivo del trabajo

Anteriormente hemos definido el objetivo del trabajo y ahora vamos a ampliar un poco este aspecto. Como hemos comentado el objetivo principal es el desarrollo de una aplicación en android por diversos motivos. El primero y quizás más importante era la curiosidad que nos surgía sobre todo el mundo relacionado con Android. La tecnología móvil creo que es fascinante cuanto menos. El siguiente era abordar un problema que surge habitualmente cuando te encuentras en áreas con grandes aglomeraciones y es la pérdida de la cobertura móvil para poder comunicarte con otras personas. Tanto para hablar por escrito como para enviar archivos. Es importante recalcar que este problema va a desaparecer con las nuevas tecnologías 5G aunque en su momento no se notó este cambio. Adicionalmente, puede ser interesante si quieres enviar un archivo grande y deseas ahorrar datos.

También se quiere que los archivos que se envían vayan cifrados para asegurar cierta seguridad en el usuario. En la propia API de Nearby se especifica que los datos van en claro, así que es una manera de aportar valor al trabajo.

A modo general y resumiendo lo que hemos comentado se desea conseguir :

- Amplios conocimientos desarrollando aplicaciones en entornos Android.
  - Manejo de librerías externas
  - Creación de interfaces de usuario
  - Manejo de hilos de independientes
  - Aprendizaje sobre la arquitectura de aplicación android
- Manejo completo de la API Nearby de google.
- Conocimiento sobre certificados y seguridad en Android.
- Amplio manejo en el entorno de desarrollo Android Studio.
- Aumento de conocimiento en el sistema de control de versiones comúnmente conocido como git.

### 3.2 Especificación de requisitos inicial

Como hemos comentado se trata de un desarrollo completamente nuevo, luego no contemplaremos apartados como el planteamiento del problema o estudio de versiones anteriores.

La especificación de requisitos <sup>2</sup> (ERS) es una descripción completa del comportamiento de un sistema que se va a desarrollar. Podemos dividir los requisitos en dos grupos:

- Requisitos funcionales: también llamados casos de uso. Describen las interacciones entre el usuario y el software. Operaciones y actividades que el sistema debe poder desempeñar.
- Requisitos no funcionales: Imponen restricciones sobre el diseño.

### 3.2.1 Requisitos funcionales

Como hemos comentado, en los requisitos funcionales <sup>3</sup> se describen las interacciones entre el usuario y el sistema. Vamos a nombrarlos en escala descendente en función de la importancia que poseen.

Requisitos funcionales	
RF01	El sistema NO usará ningún tipo de red móvil a excepción de la propia descarga de la misma.
RF02	El sistema permitirá mostrarte a otros usuarios o ocultarte.
RF03	El sistema permitirá descubrir a otros usuarios y dejar de hacerlo.
RF04	El sistema permite resetear todos los estados. Es decir, eliminar y desconectar todo lo previamente descubierto y conectado..
RF05	El sistema permite cambiar tu nombre de usuario.
RF06	El sistema mostrará visualmente los usuarios que aparezcan en su rango de acción

<sup>2</sup> "Especificación de requisitos de software - Wikipedia, la enciclopedia ...." Se consultó el junio 3, 2019.

[https://es.wikipedia.org/wiki/Especificaci%C3%B3n\\_de\\_requisitos\\_de\\_software](https://es.wikipedia.org/wiki/Especificaci%C3%B3n_de_requisitos_de_software).

<sup>3</sup> "3. Técnicas para Identificar Requisitos Funcionales y No Funcionales ...." Se consultó el junio 3, 2019.

<https://sites.google.com/site/metodologiareq/capitulo-ii/tecnicas-para-identificar-requisitos-funcionales-y-no-funcionales>.

## Trabajo de fin de máster

RF07	El sistema diferenciará a los usuarios, aquellos que son amigos y aquellos que no lo son.
RF08	En el sistema solo se puede interactuar con un usuario si es amigo.
RF09	El sistema ofrecerá una vista donde se podrán ver los usuarios amigos.
RF10	En la vista de amigos, estos podrá ser eliminados o bloqueados temporalmente.
RF11	El sistema permite enviar una petición de amistad.
RF12	El sistema permite conectarte a un usuario amigo.
RF13	El sistema permite, una vez conectado, iniciar un chat seguro.
RF14	El sistema permite, una vez conectado, enviar un archivo cifrado de cualquier tamaño.
RF15	El sistema permite, una vez conectado, desconectarse del usuario.
RF16	El sistema permite, desde la pantalla deslizable, consultar la carpeta donde se almacenan los archivos recibidos.
RF17	El sistema permite, desde la pantalla deslizable, acceder al menú de opciones del sistema.
RF18	El sistema permite, desde la pantalla deslizable, compartir un archivo con todos los usuarios amigos en el rango aceptable.
RF19	El sistema permite, desde la pantalla deslizable, enviar una notificación a cualquier usuario dentro de rango.
RF20	El sistema permite, en la pantalla deslizable, seleccionar una imagen de perfil.
RF21	El sistema permite, en la pantalla deslizable, acceder a un menú de ayuda al usuario.
RF22	El sistema permite, en la pantalla deslizable, acceder a un menú de ayuda al usuario con las principales características.

RF23	El sistema permite conexiones simultáneas entre diferentes dispositivos.
RF24	Si la aplicación se encuentra en segundo plano, recibirá notificaciones entrantes en función del servicio.
RF25	Un usuario no podrá comunicarse con un usuario que esté bloqueado.
RF26	Todas las conexiones entre dispositivos van cifradas.
RF27	El sistema te permite cancelar el envío de un archivo
RF28	El sistema te permite esconder el progreso de envío de un archivo.

*Tabla 3: Requisitos funcionales del sistema*

Los no funcionales son aquellos requerimientos que no se refieren directamente a las funciones específicas que entrega el sistema, sino a las propiedades emergentes de éste como la fiabilidad, la respuesta en el tiempo y la capacidad de almacenamiento.

De forma alternativa, definen las restricciones del sistema como la capacidad de los dispositivos de entrada/salida y la representación de datos que se utiliza en la interfaz del sistema.

En los requisitos no funcionales<sup>4</sup> vamos a comentar todas las restricciones que afectan a nuestro sistema.

### 3.2.2 Requisitos NO funcionales

Requisitos NO funcionales	
RNF01	El sistema mostrará en la pantalla inicial en todo momento el estado del dispositivo encontrado. En rojo si no está conectado y verde si lo está. De igual manera para identificar si es amigo o no.
RNF02	Se admiten un número ilimitado de dispositivos conectados. El límite viene marcado por el móvil utilizado y la capacidad de la api.

<sup>4</sup> "Técnicas para Identificar Requisitos Funcionales y No Funcionales ...." Se consultó el junio 3, 2019.

<https://sites.google.com/site/metodologiareq/capitulo-ii/tecnicas-para-identificar-requisitos-funcionales-y-no-funcionales>.

## Trabajo de fin de máster

RNF03	El sistema viene activado con la estrategia CLUSTER por defecto y no se puede cambiar.
RNF04	La imagen de perfil no podrá tener un tamaño superior a 200 KB.
RNF04	Si un dispositivo se desconecta por que se encuentra fuera de rango o por algún error interno de la aplicación, debe verse reflejado y desaparecer de la lista de dispositivos. Si además estaban en medio de una conexión debe notificarlo.
RNF05	Cualquier error entre dispositivos, tanto como timeouts o errores interno por mala programación deben visualizarse en pantalla con un código en rojo y su descripción amigable con el usuario.
RNF06	Bajo ningún concepto un error debe suponer el cierre de la aplicación. Deben estar controlados.
RNF07	El cambio del nombre de perfil supone un reinicio interno de la aplicación para mostrarse con dicho nombre cambiado.
RNF08	Las operaciones o peticiones de servicio entre usuarios, como una petición de amistad o de transmisión de archivos llevarán un timeout establecido por defecto. Si se cumple sin terminar la petición se produce un error en ambos dispositivos.
RNF09	En la pantalla de amistad el borrado de un usuario se refleja en la lista
RNF10	En la pantalla de amistad el bloqueo de un usuario se refleja en la lista, mostrandose en la contigua de bloqueos.
RNF11	Internamente el sistema mantiene el estado y las conexiones de los dispositivos en el tiempo, y adicionalmente también si este pasa a un segundo plano.
RNF12	Una petición de conexión SIEMPRE requiere la confirmación del usuario a quien se le envía.
RNF13	Una petición de amistad SIEMPRE requiere la confirmación del usuario a quien se le envía.
RNF14	Una petición de chat SIEMPRE requiere confirmación del usuario a quien se le envía.



RNF15	Cualquier envío de archivos no requiere de confirmación en el receptor del mismo.
RNF16	Cualquier lista que crezca demasiado en tamaño habilitará un scroll lateral.
RNF17	En el envío de un fichero se mostrará una barra de estado con el % enviado. Podrá cancelarse en cualquier momento dicha transferencia.
RNF18	Si se intenta conectar con un dispositivo actualmente ocupado la conexión es rechazada. Ocupado significa que esté enviando un fichero , o que se encuentre en una sala de chat con otro usuario.
RNF19	El sistema mostrará una interfaz amigable basada en las nuevas tecnologías de Material Design de Android

*Tabla 4: Requisitos no funcionales del sistema*

### 3.3 Re-Especificación de requisitos

Como suele ser habitual en estos casos hay algunos requisitos que se han visto modificados por diversas causas, en algunos casos debido a la falta de tiempo y en otros casos debido a limitaciones en la API de Nearby.

Los requisitos eliminados/modificados son los siguientes :

Requisito	Estado	Descripción	Fecha
RF26	Modificado	Por cómo está implementada la API es complicado cifrar archivos, especialmente archivos grandes. Sin embargo no hay problema en cifrar los mensajes de texto por lo tanto el chat irá cifrado.	01/04/2019
RNF02	Modificado	Ahora el número de dispositivos conectados va a depender del tipo de	15/04/2019

## Trabajo de fin de máster

		estrategia que tengas puesta. Dejamos modificar dicha estrategia porque los envíos en modos cluster de archivos son muy lentos al utilizar únicamente tecnología bluetooth.	
RNF03	Eliminado	Relacionado con RNF02, ahora se deja modificar la estrategia.	20/05/2019
RNF18	Eliminada	Prácticamente implica cambiar todo el diseño de la aplicación y hemos ido cortos de tiempo	03/06/2019

*Tabla 5: Tabla de re-especificación de requisitos*

### 3.3 Re-Estimación de los tiempos

Los tiempos estimados en realizar cada una de las tareas también han sido modificados. Al final se ha tardado más tiempo en comprender cómo funciona Android en general. Sin embargo hemos tardado menos en tener el uso de la api de Nearby. Todo este tiempo perdido se ha restado en la creación de la interfaz y es por ello que se ha eliminado dicho requisito.

Tarea	Trabajo	Duración	Comienzo	Fin
Trabajo de fin de master	486h	87 días	14 Febrero	14 Junio
Documentación	50h	87 días	14 Febrero	14 Junio
Comprensión y análisis del entorno de seguridad y las librerías JCE de java y del entorno de desarrollo de Android Studio.	105h	28 días	14 Febrero	21 Marzo
Comprensión de los principios básicos de Seguridad: Servicios, mecanismos y protocolos	50	11 días	21 Marzo	2 Abril

## Trabajo de fin de máster

Comprensión de las APIS de desarrollo para las comunicaciones de proximidad Nearby Connections API, wifi-direct, bluetooth	57h	14 días	2 Abril	10 abril
Desarrollo de la infraestructura de comunicaciones y seguridad para los intercambios entre el Cliente móvil y el Sistema de Registro y entre los sistemas móviles entre sí.	150h	23 días	10 Abril	1 Junio
Desarrollo de la Interfaz del cliente Android	25h	14 días	1 Junio	15 Junio
Evaluación y pruebas	49h	9 Días	1 Junio	16 Junio

Tabla 6: Re-estimación de los tiempos

Pudiéndose ver la comparación en el siguiente gráfico.

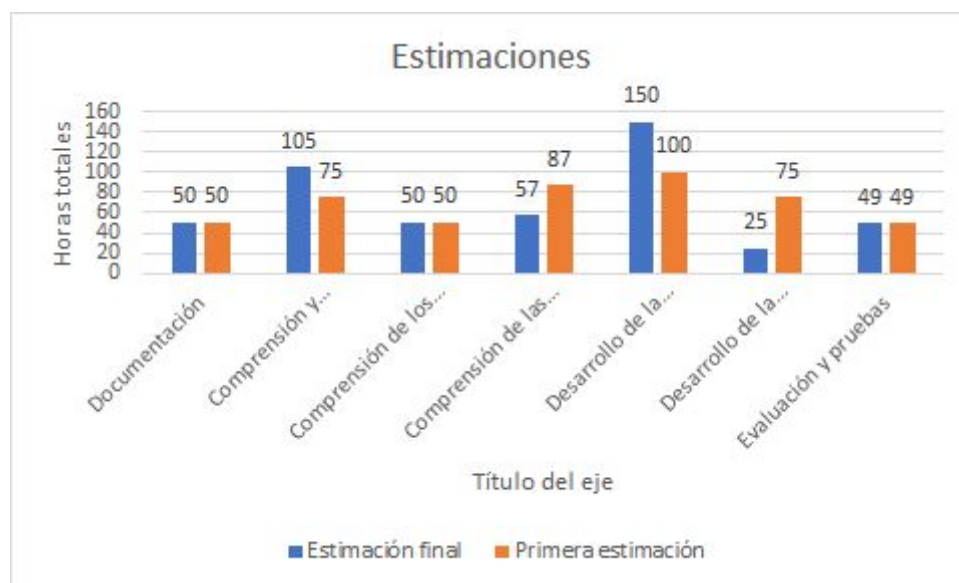


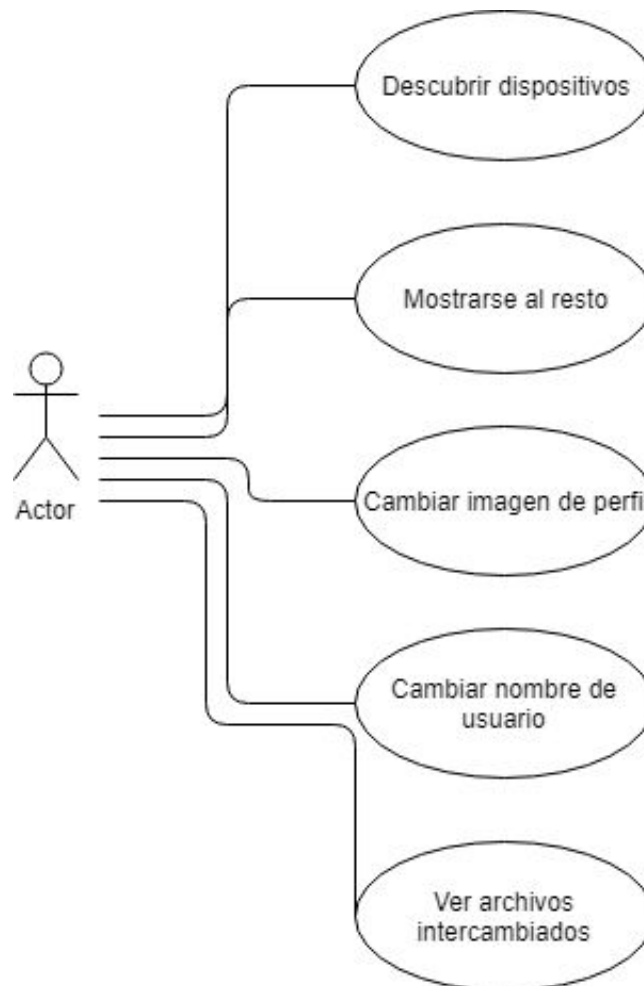
Ilustración 2: Comparación entre estimaciones

Como comentamos anteriormente, hemos fallado en la estimación de tiempos al no tener experiencia en el desarrollo de este tipo de proyectos.

### 3.4 Casos de uso

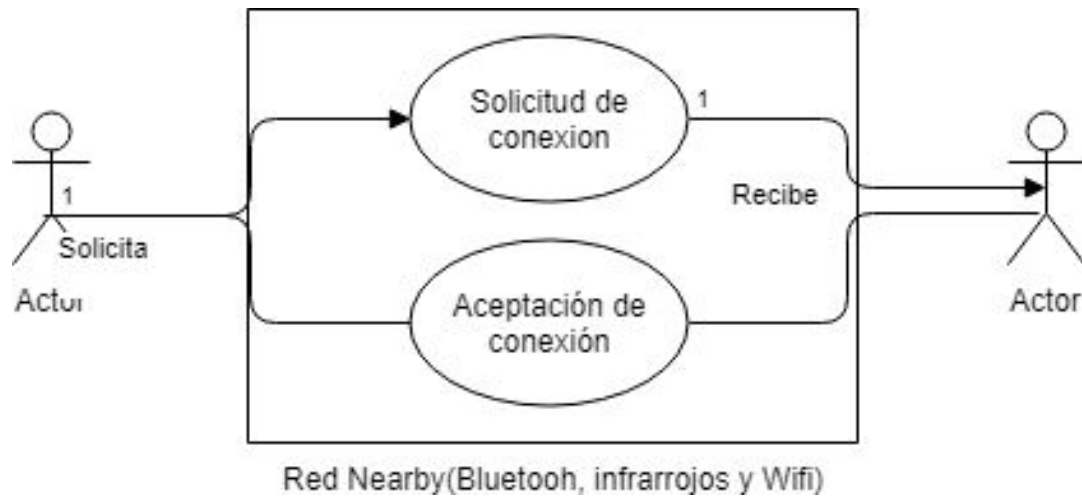
Para los casos de uso solo existe un rol y es el del usuario que llamaremos actor. Pueden estar involucrados en un caso de uso uno, dos o más usuarios. Los trataremos de menor a mayor complejidad, es decir de menor número de usuarios involucrados a uno mayor.

#### 3.4.1 Acciones que no requieren conexión



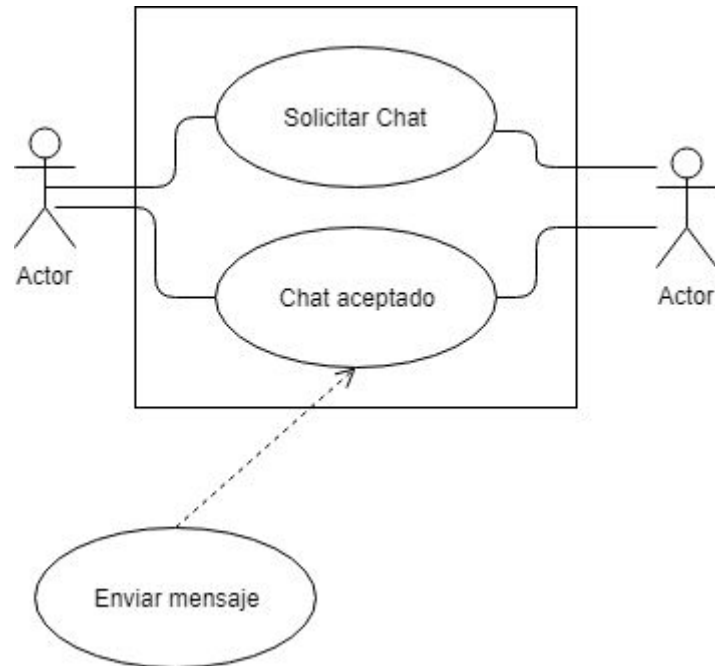
*Ilustración 3: Primer caso de uso. Acciones sin conexión*

### 3.4.2 Conexión entre dos usuarios



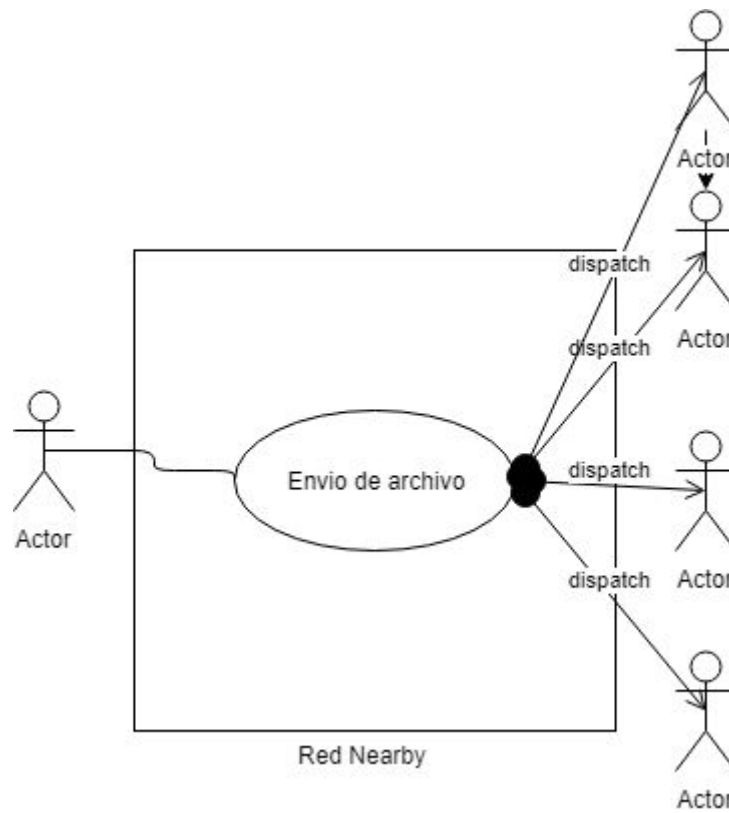
*Ilustración 4: Caso de uso 2. Conexión entre usuarios*

### 3.4.3 Petición de chat



*Ilustración 5: Caso de uso 3. Petición de chat*

### 3.4.4 Envío de archivos



*Ilustración 6: Caso 4. Envío de archivo múltiple*

# Capítulo 4: Google Nearby

En este capítulo vamos a hacer una introducción a la tecnología de Google Nearby ya que es imprescindible para abordar el proyecto. Hasta ahora solo hemos hablado de que vamos a permitir establecer chats de conversación entre usuarios y enviar archivos, pero veremos en un poco más de detalle la tecnología con la que se lleva a cabo este proceso por debajo.

A lo largo del trabajo vamos a trabajar con la API de Google denominada Nearby <sup>5</sup>. Esta tiene dos vertientes, por un lado Nearby Connections y por otro lado Nearby Messages.

La primera permite a las aplicaciones descubrir, conectarse e intercambiar datos con dispositivos cercanos en tiempo real, independientemente de la conectividad de la red. Utiliza Bluetooth, Wi-Fi y otras tecnologías, y está diseñado para ocultar la complejidad de estas tecnologías para que su aplicación pueda enfocarse en descubrir e interactuar con dispositivos cercanos fácilmente .

---

<sup>5</sup> "Nearby | Google Developers." <https://developers.google.com/nearby/>. Se consultó el 4 jun.. 2019.



La segunda, messages, permite que las aplicaciones publiquen pequeños mensajes a los que otros dispositivos pueden suscribirse y recibir cuando están cerca. Los mensajes se transmiten a través de la nube. Los dispositivos cercanos se detectan mediante Bluetooth y audio casi ultrasónico.

La finalidad de nuestra aplicación como hemos comentado anteriormente será la de enviar mensajes y archivos cifrados en un entorno donde la red de internet no existe. Por lo tanto trabajaremos durante el trabajo con la primera API, connections.

Esta api nos permite por ejemplo

- Pizarra colaborativa: anote las ideas con los participantes cercanos en una pizarra virtual compartida.
- Juegos multijugador locales: configura un juego multijugador e invita a otros usuarios cercanos a unirse.
- Juegos multipantalla: use un teléfono o tableta como un controlador de juegos para jugar juegos que se muestran en un dispositivo Android cercano de pantalla grande, como Android TV.
- Transferencias de archivos sin conexión: comparta fotos, vídeos o cualquier otro tipo de datos de forma rápida y sin necesidad de una conexión de red.

Nosotros como es lógico nos centraremos en esa última opción.

El objetivo de esta API es abstraer en cierto modo el funcionamiento de las tecnologías WIFI, Bluetooth, BLE<sup>6</sup> dejando solo las fortalezas y minimizando los puntos débiles de cada unas. Esto es interesante puesto que de cara a implementar la API es muy sencillo de hacer pero contará con algunas desventajas. Principalmente, que es imposible seleccionar qué tecnología usar en cada momento. Si estoy transfiriendo un archivo quiero la máxima velocidad, es decir, quiero utiliza el WIFI. Sin embargo la API no siempre reacciona correctamente y utiliza el bluetooth dando lugar a una transmisión muchas más lenta.

---

<sup>6</sup> "Bluetooth de baja energía - Wikipedia, la enciclopedia libre." [https://es.wikipedia.org/wiki/Bluetooth\\_de\\_baja\\_energ%C3%ADa](https://es.wikipedia.org/wiki/Bluetooth_de_baja_energ%C3%ADa). Se consultó el 4 jun.. 2019.

## WiFi vs. Bluetooth

	Bluetooth	Wifi
Specifications authority	Bluetooth SIG	IEEE, WECA
Year of development	1994	1991
Bandwidth	Low ( 800 Kbps )	High (11 Mbps )
Hardware requirement	Bluetooth adaptor on all the devices connecting with each other	Wireless adaptors on all the devices of the network, a wireless router and/or wireless access points
Cost	Low	High
Power Consumption	Low	High
Frequency	2.4 GHz	2.4 GHz
Security	It is less secure	It is more secure
Range	10 meters	100 meters
Primary Devices	Mobile phones, mouse, keyboards, office and industrial automation devices	Notebook computers, desktop computers, servers
Ease of Use	Fairly simple to use. Can be used to connect upto seven devices at a time. It is easy to switch between devices or find and connect to any device.	It is more complex and requires configuration of hardware and software.

*Ilustración 7: Wifi vs Bluetooth*

Como se puede observar en la tabla , y aunque es una tabla con cierta antigüedad, las velocidades de transmisión son considerablemente inferiores.

### 4.1 Permisos

La API requiere de ciertos permisos <sup>7</sup> para poder ejecutarse. Ellos son los siguientes

---

<sup>7</sup> "Permissions overview | Android Developers." <https://developer.android.com/guide/topics/permissions/overview>. Se consultó el 4 jun.. 2019.

```
<!-- Required for Nearby Connections -->
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<!-- Optional: only required for FILE payloads -->
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

*Ilustración 8: Permisos para la API Nearby*

Como ACCESS\_FINE\_LOCATION y READ\_EXTERNAL\_STORAGE se consideran permisos de sistema peligrosos, además de añadirlos al manifiesto de android, solicitaremos estos permisos en tiempo de ejecución, siendo el usuario quien acepta explícitamente estos permisos.

## 4.1 Estrategia

Una parte muy importante del API es la estrategia <sup>8</sup>. Cuando se invoca el método para comenzar a descubrirnos al resto de mundo, irá con una estrategia asociada. En función de esta se pueden dar diversos panoramas.

Es importante indicar que dos usuarios sólo serán visibles el uno con el otro, y por tanto solo podrán conectarse entre ellos, si la estrategia que utilizan es la misma.

### 4.1.1 P2P Cluster

Es una estrategia de igual a igual que admite una topología de conexión M-to-N o en forma de clúster. En otras palabras, esto permite la conexión de grupos amorfos de dispositivos dentro del alcance de la radio (~ 100 m), donde cada dispositivo puede iniciar conexiones salientes a otros dispositivos M y aceptar conexiones entrantes de otros dispositivos.

---

<sup>8</sup> "Strategies | Nearby Connections API | Google Developers." 11 abr.. 2019, <https://developers.google.com/nearby/connections/strategies>. Se consultó el 4 jun.. 2019.

Será la que utilicemos por defecto en la aplicación, pero es sin duda la más lenta. Incluso así, permite establecer un chat y enviar imágenes no muy grandes sin muchos problemas.

#### 4.1.1 P2P Star

P2P\_STAR es una estrategia de igual a igual que admite una topología de conexión de 1 a N o en forma de estrella. En otras palabras, esto permite conectar dispositivos dentro del rango de radio (~ 100 m) en forma de estrella, donde cada dispositivo puede, en cualquier momento, desempeñar el papel de un concentrador (donde puede aceptar conexiones entrantes de N otros dispositivos), o un radio (donde puede iniciar una conexión saliente a un solo concentrador), pero no ambos.

Esta estrategia es más interesante para archivos grandes, y pedirá ser activada en cuanto desees recibir un archivo superior a 20 megabytes.

#### 4.1.1 P2P Point to point

P2P\_POINT\_TO\_POINT es una estrategia de igual a igual que admite una topología de conexión 1-a-1. En otras palabras, esto permite conectar dispositivos dentro del rango de radio (~ 100 m) con el mayor rendimiento posible, pero no permite más de una conexión a la vez.

Esta estrategia se presta mejor a situaciones en las que la transferencia de datos es más importante que la flexibilidad de mantener múltiples conexiones.

También se solicita si el archivo es mayor de 20 megas. La decisión sería del usuario. La diferencia es que en esta estrategia ningún usuario podría enviarte archivos o peticiones de chat. Directamente no podrían conectarse contigo.

## 4.3 Anunciar y descubrir

Como hemos comentado podemos anunciarnos a otros o estar invisibles. Y a la vez podemos descubrir que usuarios tenemos cerca o simplemente esperar conexiones entrantes mientras no hacemos nada.

El API se encarga de avisarte cuando un dispositivo nuevo aparece en la red, cuando se desaparece de la red o cuando has empezado a anunciarse. Esto lo hace con las clases `connectionLifecycleCallback` y `endpointDiscoveryCallback` respectivamente. La primera implementa llamadas cuando comienza la conexión con un dispositivo y la segunda llamadas relacionadas con el descubrimiento y pérdidas de dispositivos en la red.

Un punto muy interesante que se comenta en la documentación oficial es que descubrir a otras personas genera operaciones muy pesadas de radiofrecuencia que puede hacer provocar inestabilidad en las conexiones que ya tenías establecidas. Así pues nosotros, como recomiendan, pararemos el descubrimiento cada vez que intentemos conectar con alguien. Supondremos que ya hemos descubierto todo lo que queríamos.

### 4.3.1 Anunciar

El método que usamos para anunciarnos, sacado de la documentación oficial, es el siguiente:

```
private void startAdvertising() {
    AdvertisingOptions advertisingOptions =
        new AdvertisingOptions.Builder().setStrategy(STRATEGY).build();
    Nearby.getConnectionsClient(context)
        .startAdvertising(
            getUserNickname(), SERVICE_ID, connectionLifecycleCallback, advertisingOptions)
        .addOnSuccessListener(
            (Void unused) -> {
                // We're advertising!
            })
        .addOnFailureListener(
            (Exception e) -> {
                // We were unable to start advertising.
            });
}
```

*Ilustración 9: Función startAdvertising()*

Como podemos observar, recibe cuatro parámetros :

- Un **nombre** de usuario. En nuestra aplicación este nombre está formado por,
  - Nombre de usuario real
  - Id interno de la aplicación
  - Nombre del dispositivo
- Un **ID** del servicio. En nuestro caso será el nombre del paquete de la aplicación. Este es único y sirve para aceptar solo paquetes con ese id y que no se junten con otras aplicaciones cercanas que envíen paquetes similares.
- Una función **callback**. La función a la que llama cuando se inicia una conexión o se desconecta un dispositivo.
- Un conjunto de **opciones**. Aquí nosotros podremos establecer la estrategia con la que nos anunciamos.

Con respecto al nombre de usuario, podrá ser seleccionado por el usuario en el menú de opciones. Será *Anonymous* por defecto.

El id interno del dispositivo es el número que genera android cada vez que el móvil se restaura de fabrica <sup>9</sup>. Se recupera con una llamada a Settings.Secure.ANDROID\_ID .Así nos aseguramos que es medianamente único. En el enlace asociado se pueden encontrar otras manera de utilizar numeros unicos pero esta nos ha parecido el planteamiento más correcto.

El nombre del dispositivo se mostrará en pantalla, puesto que dos usuarios pueden tener el mismo nombre de usuario y no es algo que podamos controlar, al menos es posible que su dispositivo móvil sea diferentes.

### 4.3.2 Descubrir

El método para descubrir, sacado de la documentación oficial, es el siguiente.

---

<sup>9</sup> "How to retrieve an Unique ID to identify Android devices ? - Medium." 9 feb.. 2017, <https://medium.com/@ssaurel/how-to-retrieve-an-unique-id-to-identify-android-devices-6f99fd5369eb>. Se consultó el 7 jun.. 2019.

```
private void startDiscovery() {
    DiscoveryOptions discoveryOptions =
        new DiscoveryOptions.Builder().setStrategy(STRATEGY).build();
    Nearby.getConnectionsClient(context)
        .startDiscovery(SERVICE_ID, endpointDiscoveryCallback, discoveryOptions)
        .addOnSuccessListener(
            (Void unused) -> {
                // We're discovering!
            })
        .addOnFailureListener(
            (Exception e) -> {
                // We're unable to start discovering.
            });
}
```

*Ilustración 10: Función startDiscovery()*

Es muy similar a la anterior, exceptuando que no necesita el nombre del usuario para nada.

### 4.3 Manejo de las conexiones

Cuando se quiere conectar con otro dispositivo, se envía una petición. Solo se produce una conexión cuando es aceptado en ambos dispositivos. Lo interesante de esto que es que si la arquitectura lo permite se podrían producir conexiones entre usuarios de forma invisible al cliente sin necesidad de tener que autenticarse cada vez que se requiera. Esto posibilita el poder enviar archivos o notificaciones previa conexión invisible mediante alguna condición pre pactada anteriormente.

En nuestro caso, habilitaremos en el panel de opciones la posibilidad de, si tienes registrado a una persona como amiga y esta intenta una conexión contigo, tu no tengas que aceptar la conexión sino que automáticamente es aceptada.

### 4.3 Intercambio de datos

Una vez que se establecen conexiones entre dispositivos, puede intercambiar datos enviando y recibiendo objetos de carga útil. Una carga útil puede representar una matriz de bytes simple, como un mensaje de texto corto; un archivo, como una foto o un video; o una transmisión, como la captura de audio desde el micrófono del dispositivo. Esta carga útil recibe el nombre en la API de **payload**.

Por cada payload se genera un PayloadUpdate a la hora de enviarse. Este objeto simplemente indica si el payload está en proceso de enviarse, o está ya terminado. Lo

recibe tanto el emisor (podrías mostrar un progreso de archivo saliente) como el receptor (podrías mostrar un progreso de archivo entrante).

Dentro de un payload podemos identificar 3 partes bien diferenciadas :

- Un **identificador interno**. Cada payload se identifica por un número que generamos y que será de gran importancia como explicaremos más adelante.
- El **tipo** de Payload. Tenemos dos tipos de payloads que enviaremos:
  - **Bytes**. El tipo más simple. Útiles para enviar metadatos o mensajes. Su tamaño máximo es 32768 bytes.
  - **File**. Se crean desde un archivo local. No tienen límite de tamaño e internamente se trocean en payloads de tipo byte ahorrando el trabajo de implementación.
  - **Stream**. Para crear archivos en el momento como grabaciones de un micrófono. No las usaremos.
- El **contenido** del Payload. En forma de array de bytes y de un tamaño máximo de 32 KB aproximadamente.

Una vez definido lo que es un Payload tenemos que entender cómo funciona el intercambio de datos.

1. El emisor envía un Payload.
2. En el emisor se genera un PayloadUpdate. Si es de tipo bytes sólo se genera uno con un resultado de éxito. Si es de tipo File se pueden generar los que sean necesarios hasta completar la transferencia.
3. En el receptor se recibe el payload.
4. Adicionalmente se recibe también, como en el paso 2, uno o varios payloads update.
5. Hasta que no llega un PayloadUpdate con el estado de **terminado**, no podemos tratar el payload inicial.



# Capítulo 5 : Diseño de proyecto

## 5.1 Introducción

En este capítulo vamos a hacer una introducción a la arquitectura del sistema que hemos escogido. Veremos el patrón Modelo Vista Controlador <sup>10</sup> orientado a Android y porqué ha sido el escogido a la hora del desarrollo de la aplicación. Aunque debido a la funcionalidad de la aplicación apenas guardamos datos, también veremos por encima como tenemos implementada la base de datos.

## 5.2 Arquitectura del sistema. Modelo Vista Controlador

Mantener un código limpio y reutilizable es la clave para que un proyecto salga adelante. Ahorra tiempo y muchos quebraderos de cabeza. Es por eso que la arquitectura MVC/MVP está muy extendida en los proyectos software en los tiempos actuales. Una arquitectura de este estilo suele cumplir los siguiente requisitos :

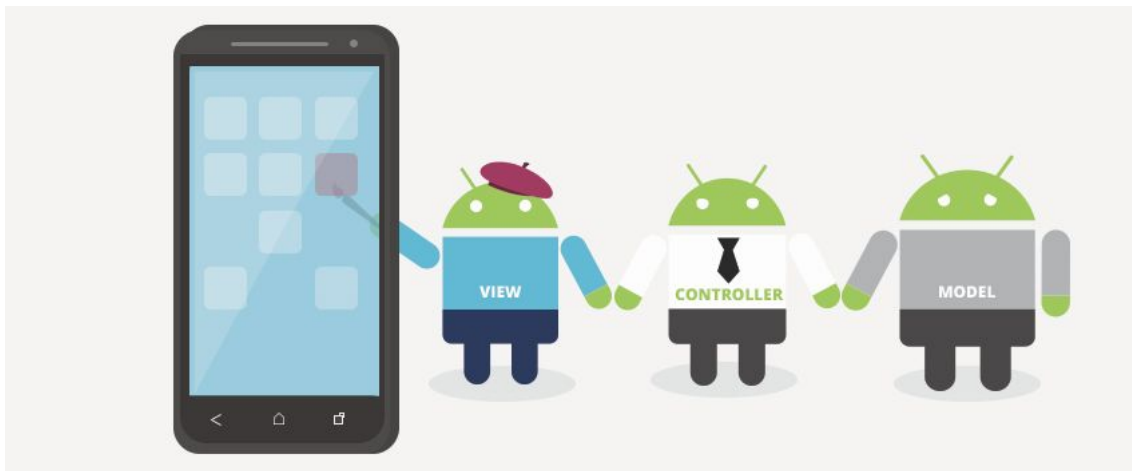
---

<sup>10</sup> "Modelo-vista-controlador - Wikipedia, la enciclopedia libre." <https://es.wikipedia.org/wiki/Modelo%F2%80%93vista%F2%80%93controlador>. Se consultó el 6 jun.. 2019.

- Código legible y mantenible.
- Código modular que proporciona alto grado de desacoplamiento
- Código testable

Una arquitectura MVC/MVP consigue estos principios mediante la división en tres capas<sup>11</sup>.

- El componente que almacena el estado del sistema (si este estado es persistente o no). Este componente se conoce como modelo.
- El componente que maneja la entrada-salida de / para el usuario. Este componente se conoce como vista.
- El componente que encapsula la funcionalidad lógica del sistema. Este componente se conoce como Controlador / Presentador.

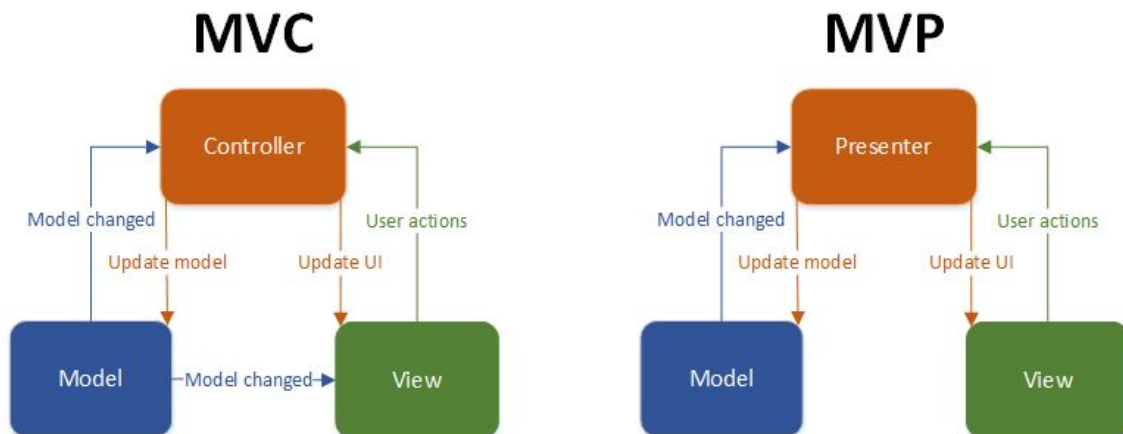


*Ilustración 11: Modelo Vista y Controlador*

Los componentes de una buena implementación de MVC / MVP se deben desacoplar tanto como sea posible: debería ser posible cambiar de una entidad de entrada / salida (Vista) a otra, o cambiar la ubicación o el tipo de mecanismo de almacenamiento persistente (Modelo) sin afectando a otros componentes.

---

<sup>11</sup> "MVP and MVC in Android - part 1 » - TechYourChance." 22 abr.. 2018, <https://www.techyourchance.com/mvp-mvc-android-1/>. Se consultó el 6 jun.. 2019.



*Ilustración 12: MVC vs MVP*

Nuestro acercamiento a esa arquitectura será la vista en la imagen MVP, donde los 3 componentes están completamente desacoplados. Pero como está extendido el concepto de MVC para todos, y controlador es mucho más representativo que presentador usaremos este nombre MVC.

Por defecto en Android, la clase que controla vista y lógica de negocio es la actividad. No se da una manera en el framework de poder separar ambos de una manera sencilla por lo tanto debemos de tomar la decisión de si la actividad albergará lógica o vista.

Una actividad en Android siempre extiende de la clase Contexto. Esta clase está presente siempre en cualquier librería de terceros y se sobreentiende entonces que aquí es donde debería ir la lógica de negocio. Nosotros seguiremos el patrón que se sugiere en el post “La actividad revisada”<sup>12</sup> donde la vista es instancia en una clase y no sabe nada ni tiene ninguna lógica más allá de la ofrecida por la vista. El controlador tiene acceso a esta vista pero no sabe nada de los componentes de ella y la actualiza mediante los disparadores de la vista. Finalmente el controlador actualizará el modelo si es necesario.

Por lo tanto, el patrón que seguiremos será el siguiente:

<sup>12</sup> "Android Architecture: Part 10, The Activity Revisited - Android Developer." 9 jul.. 2012, <http://www.therealjoshua.com/2012/07/android-architecture-part-10-the-activity-revisited/>. Se consultó el 6 jun.. 2019.

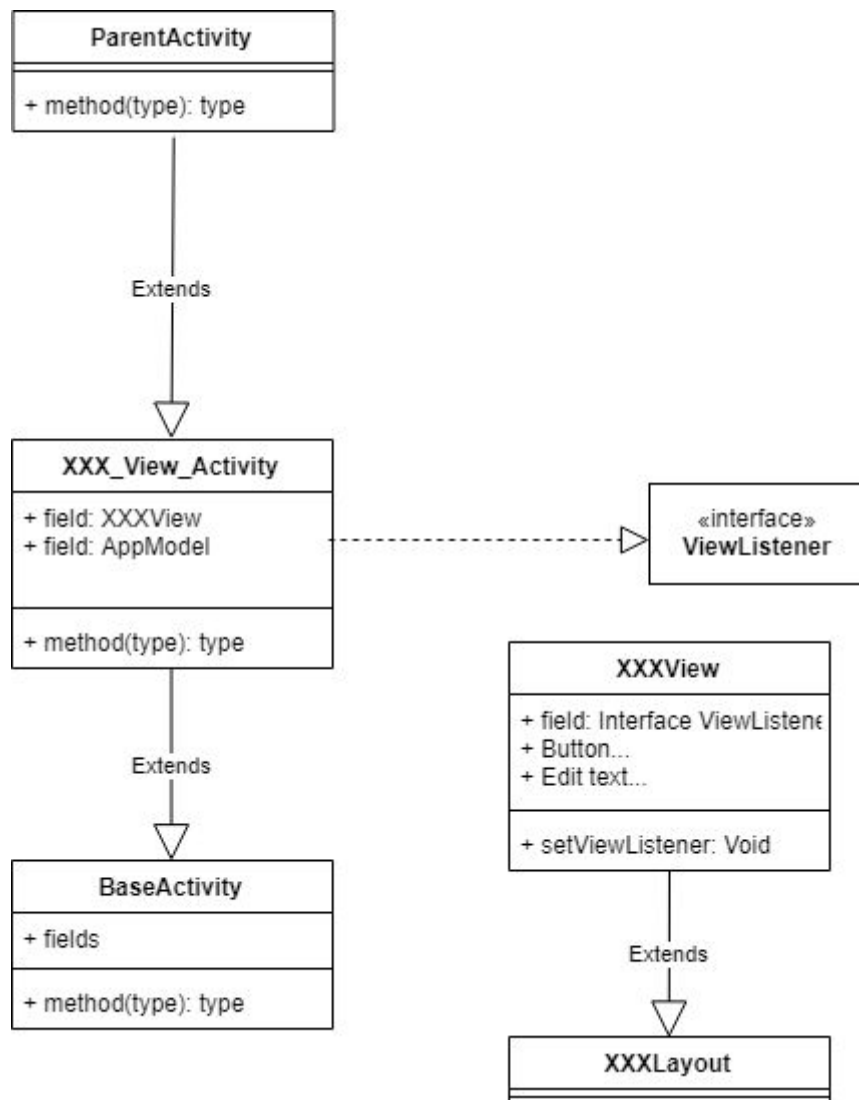


Ilustración 13: Desacople vista y controlador

XXX sería el módulo de la aplicación donde se encuentre. Por ejemplo el chat, o la pantalla principal.

Los elementos que intervienen son los siguientes:

- **XXXView**. Es la vista desacoplada. Por ejemplo si queremos personalizar una vista y desacoplarla, por ejemplo un ConstraintLayout, nuestra clase sería XXXConstraintLayoutView. Además esta vista tendría una interfaz pública ViewListener con los métodos representativos de esa vista. Por ejemplo, si hay un botón 1 en dicha vista, la interfaz tendrá un método llamado

onClickBoton1(). La clase que llame a esta vista deberá implementar la interfaz y setearla en la vista con el método `setViewListener()` que se ve en el diagrama.

- **XXX\_View\_Activity.** Sería el controlador. Internamente tendrá un campo que será la vista instanciada mediante el método `findViewById()` de la clase activity. Llamadas a esta clase actualizan la vista, pero el controlador no tiene acceso a los objetos de la vista, y la vista no sabe nada del controlador. Se puede observar que entiende una clase Base del sistema y que además la lógica de negocio se implementa en una clase padre por encima.

Como hemos visto, esta es una manera sencilla de desacoplar vista y controlador. El controlador implementa la lógica de negocio que requiere la vista. Cuando un usuario interactúa con dicha vista, activa un disparador recogido en el controlador y este, una vez procese la información, actualizará la vista con los nuevos datos.

## 5.3 Base de datos

Como venimos comentando, la aplicación guarda muy pocos datos en su memoria interna. Únicamente almacenaremos los datos de los amigos una vez que sean registrados, junto con su imagen de perfil. También almacenaremos los datos del historial del chat de los usuarios. La tabla es la siguiente :

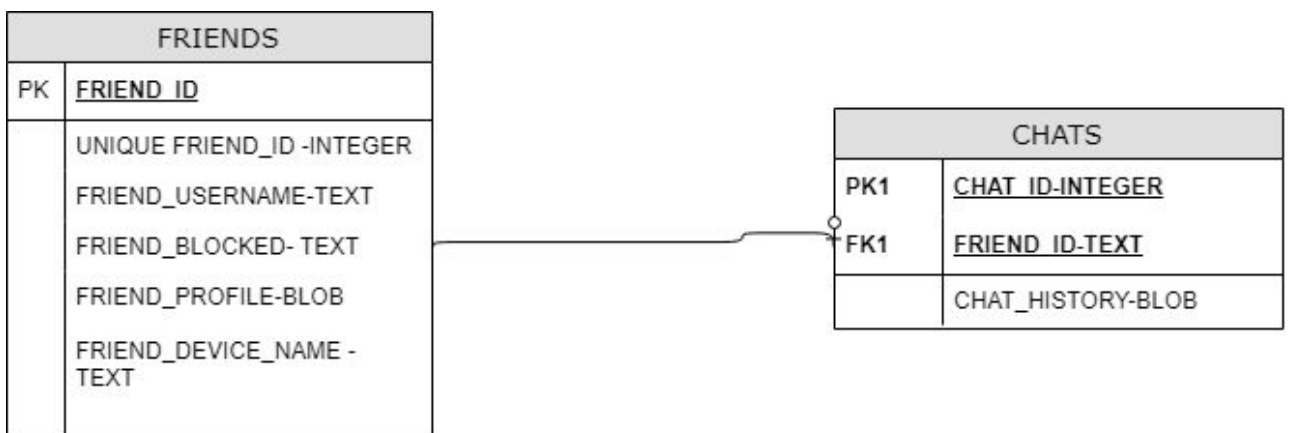
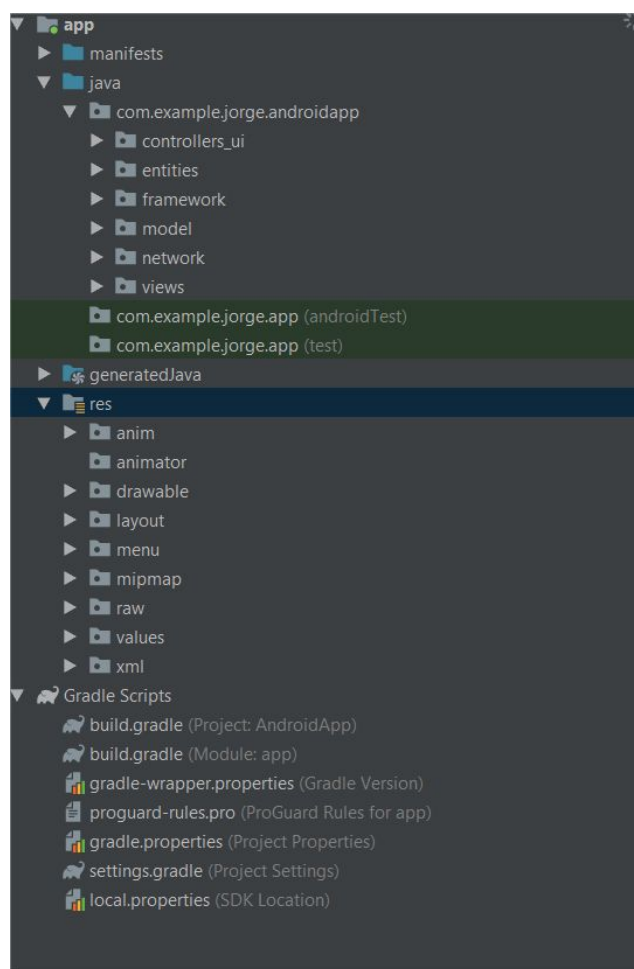


Ilustración 14: Base de datos

## 5.4 Estructura del proyecto

Se ha establecido como API mínima de android para ejecutar el proyecto la 24, Android 7.0. Fue lanzada en 2016 y consideramos que móviles más antiguos de esta fecha es difícil de encontrar, y si existen al menos estarán actualizados por el proveedor.

La estructura del proyecto es la siguiente :



*Ilustración 15: Estructura del proyecto*

## Trabajo de fin de máster

Como podemos observar tenemos los controladores , la vista y el modelo separado en paquetes diferentes . Además tenemos un paquete denominado Framework donde irán parte del core de la aplicación, es decir , la implementación de la API de Nearby, la seguridad, el sistema de logs, los providers... y un paquete llamado Network, que es donde irán el tratamiento de las peticiones como veremos más adelante.

# Capítulo 6. Desarrollo del proyecto

## 6.1 Introducción

Android es un sistema programado en Java, lo que implica su ejecución en múltiples entornos de desarrollo y permite, además, la proliferación de multitud de librerías de terceros, así como la fácil migración de las ya existentes al entorno Android. Comentaremos qué librerías se han utilizado, su funcionalidad y porque han sido escogidas.

En este capítulo hablaremos del desarrollo del proyecto, el diseño e implementación de las clases, y los problemas que han ido surgiendo.



## 6.2 Librerías utilizadas

El conjunto de librerías utilizadas es el siguiente:

- **appcompat-v7: 28.0.0**

La librería de soporte es una biblioteca estática para usar las API que no están disponibles para las versiones anteriores de la plataforma o las API de utilidad que no forman parte de las API del marco. Compatible en dispositivos que ejecutan API 14 o posterior.

- **Constraint-layout: 1.1.3**

Librería utilizada para poder utilizar los constraint-layout. Un tipo de layout mucho más completo que los anteriores.

- **Google Play Services - Nearby: 16.0.0**

Librería necesaria para poder implementar la transmisión de archivos por wifi o bluetooth.

- **Apache-common-langs 3.3.5**

Apache Commons Lang, un paquete de clases de utilidad de Java para las clases que están en la jerarquía de java.lang, o se considera que es tan estándar como para justificar la existencia en java.lang.

- **hdodenhof:circleimageview 3.0.0**

Una sencilla implementación de ImageView para imágenes de perfil.

- **Commons-Collection**

El paquete Apache Commons Collections contiene tipos que amplían y aumentan el Java Collections Framework.

- **Commons-IO**

El paquete Apache Commons Collections contiene tipos que amplían y aumentan el Java IOFramework.

- **NumberProgressBar**

Un barra de progreso diferente al estándar de android.

- **Lifecycle**

Librería para controlar el ciclo de vida de la aplicación. La usaremos para saber cuando la aplicación está en segundo plano.

- **Timber 2.5.0**

Un logger con una API pequeña y extensible que proporciona utilidad sobre la clase de log normal de Android.

- **ButterKnife**

Elimina las llamadas a findViewById usando @BindView en los campos.

## 6.3 Manifiesto.xml

A continuación podemos observar el manifiesto <sup>13</sup> de la aplicación.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.example.jorge.androidapp">

    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission
android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission
android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission
android:name="android.permission.CHANGE_WIFI_STATE" />
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>

</manifest>
```

---

<sup>13</sup> "Manifiesto de la app | Android Developers." <https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419>. Se consultó el 12 jun.. 2019.

```
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

<application
    android:name=".framework.nearby.NearbyApplication"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:ignore="GoogleAppIndexingWarning">
    <activity

android:name=".controllers_ui.activities.friends.FriendListActivity"
    android:label="@string/title_activity_friend_list"
    android:theme="@style/AppTheme.NoActionBar" />
    <activity

android:name=".controllers_ui.activities.settings.ActivitySettings"
    android:label="@string/title_activity_settings" />
    <activity
android:name=".controllers_ui.activities.SplashActivity" />
    <activity

android:name=".controllers_ui.activities.chat.ChatActivity"
    android:label="@string/title_activity_message_list"
    android:theme="@style/AppTheme.NoActionBar" />

    <uses-library
        android:name="org.apache.http.legacy"
        android:required="false" />

    <activity

android:name=".controllers_ui.activities.home.HomeActivity"
    android:label="@string/app_name"
    android:theme="@style/AppTheme.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category
android:name="android.intent.category.LAUNCHER" />
```

```
        </intent-filter>
    </activity>

    <provider
        android:name=".framework.providers.NearbyDownloadsFileProvider"
        android:authorities="com.example.jorge.androidapp.fileprovider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/provider_paths"/>
    </provider>

    <service
        android:name=".framework.nearby.NearbyConnectionsService" />
</application>

</manifest>
```

Como se puede observar en el fichero de manifiesto, tenemos 4 vistas en nuestra aplicación. La principal, la lista de amigos, la sala de chat y las opciones. Adicionalmente se pueden ver los permisos que pide la aplicación , así como los proveedores definidos y la clase Aplicación personalizada.

## 6.3 Implementación de la API Nearby en el Framework

El primer reto al que nos enfrentamos es cómo vamos a integrar la API de Nearby en nuestra aplicación. Necesitamos un objeto que persista siempre que la aplicación se encuentre en primer, o segundo plano y que se encargue de recibir notificaciones entrantes de otros dispositivos. Además habrá que guardar en dicho objeto el estado de nuestras conexiones, es decir, con quien estamos conectados, a qué dispositivos podemos ver , cuales perdemos y actualizar en consecuencia el estado del sistema. Vamos a desarrollar dicha idea a continuación en los siguientes apartados.

### 6.3.1 Persistencia

La idea principal entorno a la aplicación es la de crear un solo objeto que proporcione por un lado todos los servicios propios de la API de Nearby como pueden ser descubrir otros puntos de acceso, convertirse en un punto de acceso visible al resto de usuarios, conectar a un punto de acceso, notificaciones relacionadas con estos servicios ... y que además actúe ya no solo como un objeto que proporcione dichos servicios, sino como un contenedor de los estados de los dispositivos con los cuales interactúa. Esto es almacenar internamente qué dispositivos están conectados, cuáles ha descubierto, cuál se ha desconectado, cuál está en cola esperando a conectarse ...

Por supuesto, para que esto pueda darse es necesario que el objeto sea persistente en la memoria del dispositivo o al menos lo más persistente posible. En android y dada su arquitectura hay diversas aproximaciones que tomar.

Una de las maneras más comunes de mantener la persistencia en android es pasar el objeto entre actividades mediante un Serializable o Parcelable<sup>14 15</sup> lo que implicaría tener que enviar siempre el objeto entre actividades. Además el rendimiento se vería afectado al serializar y deserializar si es un objeto grande.

La aproximación que vamos a seguir es un poco diferente pero es más sencillo de implementar y eficiente. Por supuesto también tienes sus inconvenientes los cuales discutiremos a continuación.

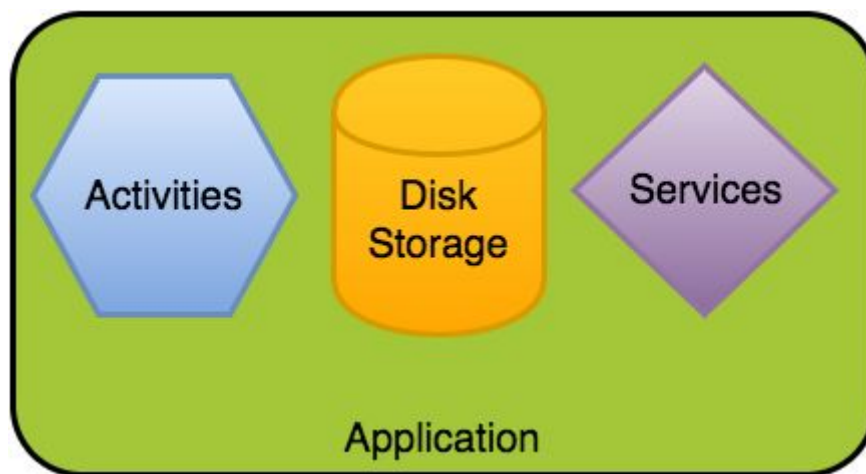
Antes de nada necesitamos conocer la clase aplicación<sup>16</sup> de Android. Esta es la clase base dentro de una aplicación de Android que contiene todos los demás componentes, como actividades y servicios. Además esta y cualquier subclase de ella ,se instancia antes que nada en la aplicación. Lo podemos observar gráficamente en la siguiente imagen :

---

<sup>14</sup> "Parcelable | Android Developers." <https://developer.android.com/reference/android/os/Parcelable>. Se consultó el 7 jun.. 2019.

<sup>15</sup> "how to persist an object in android? - Stack Overflow." 31 mar.. 2014, <https://stackoverflow.com/questions/22755655/how-to-persist-an-object-in-android>. Se consultó el 7 jun.. 2019.

<sup>16</sup> "Understanding the Android Application Class · codepath ... - GitHub." [https://github.com/codepath/android\\_guides/wiki/Understanding-the-Android-Application-Class](https://github.com/codepath/android_guides/wiki/Understanding-the-Android-Application-Class). Se consultó el 7 jun.. 2019.



*Ilustración 16: Clase aplicación*

En muchas aplicaciones, no es necesario trabajar directamente con una clase de aplicación. Sin embargo, hay algunos usos aceptables de una clase de aplicación personalizada:

- Tareas especializadas que deben ejecutarse antes de la creación de su primera actividad
- Inicialización global que debe compartirse entre todos los componentes (informes de fallos, persistencia)
- Métodos estáticos para un fácil acceso a los datos estáticos inmutables como un objeto cliente de red compartido.

En nuestro caso necesitamos que sea persistente a lo largo de toda la aplicación. Como desventajas podríamos decir que nunca se debe almacenar datos de instancia mutables dentro del objeto Aplicación porque no se garantiza que el objeto de la aplicación permanezca en la memoria para siempre, este puede ser eliminado por el SO si hay problemas de memoria o debido a algún Null Pointer. Contrariamente a la creencia popular, la aplicación no se reiniciará desde cero. Android creará un nuevo objeto de aplicación e iniciará la actividad donde estaba el usuario antes para dar la ilusión de que la aplicación nunca se eliminó en primer lugar. Para nosotros esto no supone un problema puesto que lo trataremos volviendo a escanear los dispositivos cercanos y mostrándolos en pantalla de nuevo.

Por lo tanto nosotros vamos a implementar esta aproximación, extendiendo la clase de Aplicación del sistema de android y personalizarla para que siempre tenga nuestro objeto con los servicios de la api de Nearby.

## 6.3.2 Implementación

Primeramente vamos a ver como creamos nuestra clase de aplicación que soportará, como hemos comentado, todo el sistema. Después veremos la implementación del objeto proveedor de servicio que estará contenido en esta clase de aplicación. Finalmente explicaremos cómo se comunica este objeto con las diversas actividades que vamos a crear posteriormente y el enfoque que le hemos dado.

### 6.3.2.1 Clase “Application”

Podemos ver el diagrama a continuación :

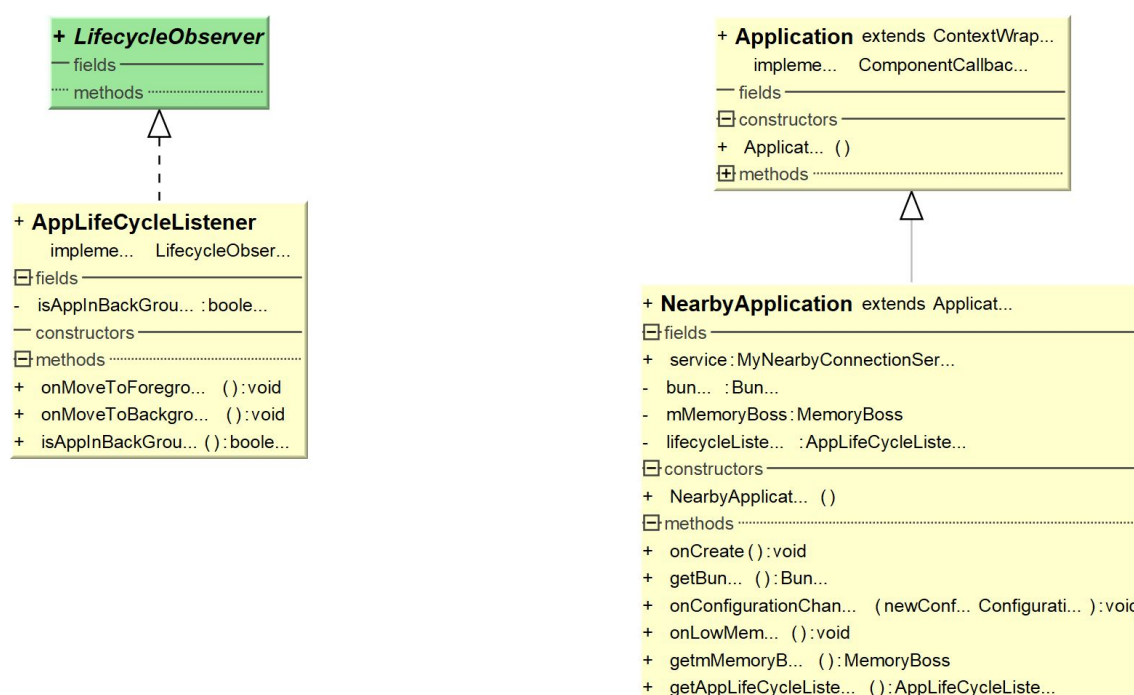


Ilustración 17: UML clase Aplicación

Como se observa en la ilustración 2 se crea una clase que extiende de la clase **Application** del sistema y que contiene únicamente el objeto proveedor de servicios llamado **MyNearbyConnectionService**.

En el método `onCreate()` instanciamos la clase de servicio :

```
this.service = MyNearbyConnectionService.getInstance(this);
```

Adicionalmente se muestra una clase que actúa como listener de los eventos del sistema. Concretamente de *onMoveToForeground()* y *onMoveToBackGround()*. Esto no servirá para saber en los controladores/actividades si estamos en primer o segundo plano.

#### 6.3.2.2 Clase de servicios

La clase proveedora de servicios se encarga implementar la API de Nearby de una manera más manejable y adaptada a las necesidades de la aplicación.

Esta clase almacena los dispositivos descubiertos, conectados o pendientes de conectar en todo momento y se encarga de actualizar sus estados dinámicamente según entren o salgan del sistema.

El diagrama de dicha clase se puede observar en la siguiente ilustración.



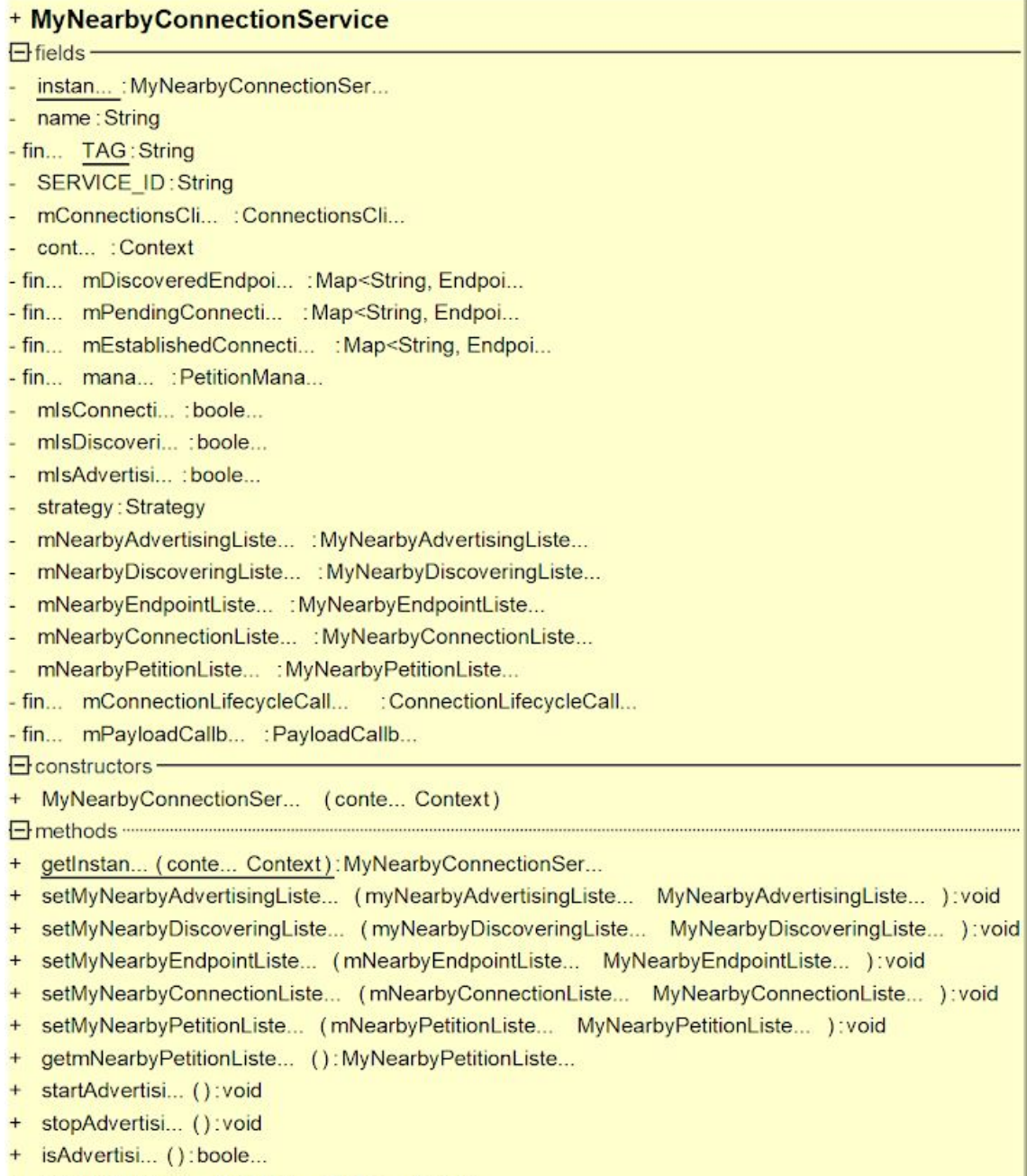


Ilustración 18: `MyNearbyConnectionService`

Es una clase muy grande y faltan bastantes métodos. Se puede encontrar la clase completa en el [anexo](#).

Como se puede observar:

- Campos. Guardan el estado de la aplicación. Dispositivos vistos, conectados, peticiones en curso y la actividad en la que nos encontramos y donde disparará los diferentes eventos. Esto se explica un poco más adelante.
- Métodos. Todo lo relacionado con las conexiones: buscar un dispositivo, conectarse a él, desconectarse, establecer una estrategia, iniciar una petición ...

#### 6.3.3.3 Conexión persistente a través de diferentes actividades.

Una vez explicado que el objeto que provee servicios se va a guardar dentro del contexto de la aplicaciones vamos a ver cómo se relaciona esto con las actividades del sistema. Como llamar a dicho servicio desde cualquier actividad y como las actividades reciben llamadas de dicho objeto <sup>5</sup>.

Para comenzar, lo que realizamos es crear una serie de interfaces cada cual orientada a una funcionalidad.

<b>+ MyNearbyClientListen...</b> — fields — <div> methods ..... </div> + connect... (endpoi... Endpo... , type:String):void
<b>+ MyNearbyConnectionListener</b> — fields — <div> methods ..... </div> + onConnectionInitia... (endpoi... Endpo... , connectionIn... Connectionl... ):void + onConnectionFai... (endpoi... Endpo... ):void
<b>+ MyNearbyDiscoveringListen...</b> — fields — <div> methods ..... </div> + onDiscoveryStart... ():void + onDiscoveryFail... ():void + onDiscoveryStopp... ():void
<b>+ MyNearbyAdvertisingListen...</b> — fields — <div> methods ..... </div> + onAdvertisingStar... ():void + onAdvertisingFai... ():void + onAdvertisingStop... ():void
<b>+ MyNearbyPetitionListener</b> — fields — <div> methods ..... </div> + onUpdatePetit... (petiti... AbstractPeticionNea... , upda... PayloadTransferUpd... ):void + onPetition... (petiti... AbstractPeticionNea... ):void + onPetitionFail... (petiti... AbstractPeticionNea... ):void
<b>+ MyNearbyEndpointListener</b> — fields — <div> methods ..... </div> + onEndpointDiscove... (endpoi... Endpo... ):void + onEndpointConnec... (endpoi... Endpo... ):void + onEndpointDisconnec... (endpoi... Endpo... ):void + onEndpointL... (endpoi... Endpo... ):void + disconn... (endpoi... Endpo... ):void

Ilustración 19: Interfaces de funcionalidad

Estas interfaces se aplicarán a la actividad en función de lo que esta haga con respecto a la API de Nearby. Por ejemplo, si es una actividad que descubre dispositivos

necesitará tener implementada la interfaz `MyNearbyEndpointListener` para poder tratar aquellos dispositivos que descubra, se conecten, desconecten o se pierdan.

Posteriormente en la clase de servicio crearemos referencias a dichas interfaces.

```
private MyNearbyAdvertisingListener mNearbyAdvertisingListener;  
  
private MyNearbyDiscoveringListener mNearbyDiscoveringListener;  
  
private MyNearbyDataListener mNearbyDataListener;  
  
private MyNearbyEndpointListener mNearbyEndpointListener;  
  
private MyNearbyConnectionListener mNearbyConnectionListener;
```

Cada una con sus getter y setters. De esta manera cada actividad en su método `onCreate()` o en su método `onStart()` debe invocarse el método set correspondiente en función de las interfaces que implementa. De esta manera cuando una llamada en la api API necesaria, ejecutará su lógica y además ejecutará el método callback en la actividad que tenga puesta, si existe.

Por ejemplo, si llega que se descubre un dispositivo, la clase de servicio actualiza su array interno, y además, llamara a su interfaz (si no es nula) `mNearbyDiscoveryListener`. Si está en un chat, no hará nada, pero si está en el home actualizará visualmente que apareció un nuevo dispositivo.

### 6.3.3 MyNearbyConnectionService en detalle

En este apartado comentaremos la manera de funcionar y los métodos más importantes y que se deben de conocer de esta clase, considerada probablemente como la más importante en el proyecto.

Antes hemos visto una imagen que resumía los aspectos más importantes de la clase, veamos a continuación y con más detalle los atributos de esta.

Nombre del atributo	Tipo	Funcionalidad
TAG	String	A la hora de loguear, se utiliza como prefijo.
mDiscoveryEndpoints	Map<String,Endpoint>	Dispositivos descubiertos
mEstablishedEndpoints	Map<String,Endpoint>	Dispositivos conectados
mPendingConnections	Map<String,Endpoint>	Dispositivos pendientes
mIsDiscovering	boolean	Descubrimiento activado
mIsConnecting	boolean	Conectando en curso
mIsAdvertising	boolean	Anunciándose activado
currentStrategy	Strategy	Estrategia de conexión seleccionada
manager	PetitionManager	Se encarga de guardar las peticiones activas y redirigir los payloads a cada peticion correspondiente

*Tabla 7: Atributos de clase de servicio*

Los métodos más importantes serían los siguientes :

Nombre del método	Funcionalidad
mConnectionsLifeCycleCallBack	<p>Se pasa como parametro en startAdvertisment().</p> <ul style="list-style-type: none"> <li>- Trata el inicio de una conexión poniendo el dispositivo en pendientes. Actualiza array interno.</li> <li>- Trata la conexión con un dispositivo. Poniéndolo en establecidos. si hay un fallo lo trata. Actualiza array interno.</li> <li>- Trata la desconexión de un dispositivo. Actualiza array interno.</li> </ul>
mConnectionsEndpointCallBack	<p>Se le pasa a startDiscovering().</p> <ul style="list-style-type: none"> <li>- Trata el descubrimiento de un dispositivo comprobando que su ID es el correspondiente a la aplicación. Actualiza su array interno.</li> <li>- Trata cuando se pierde un dispositivo por rango o cualquier otra causa. Actualiza su array interno.</li> </ul>
acceptConnection	Se acepta una conexión entrante. Se le pasa una función para el tratamiento de datos.
mPayloadCallback	La función que se le pasa a acceptConnection(). Llama al manager pasandole el payload recibido y el dispositivo al que pertenece.
Otros	Hay multitud de métodos triviales, como desconectar(), sendPayloadD(), log() ...

Tabla 8: Métodos de la clase servicios

El resto de métodos se pueden consultar en el anexo.

### 6.3.4 El manejador de peticiones

El manejador de peticiones se encarga de redireccionar los payloads entrantes entre las peticiones que actualmente se encuentran en ejecución. Guarda en el objeto un HashMap donde la clave es el ID del dispositivo receptor y el valor un array de peticiones. Como hemos comentando anteriormente, las peticiones vienen con un ID, formado por un número aleatorio y acabado con por dos dígitos. Estos dos dígitos son los que identifican la petición, cogiendo esto y el dispositivo desde donde se recibe el payload la clase es capaz procesar en la petición correcta la información.

Los principales métodos de esta clase son los siguientes.

Nombre del método	Funcionalidad
startPetition()	Recibe el ID del dispositivo objetivo y una petición. Se encarga de crear la petición y lanzar el método onStart() de la petición.
deletePetition()	Recibe el Id del onEndpointDisconnectedobjetivo y el tipo de petición que tiene que borrar.
addPayload()	Recibe el ID del dispositivo objetivo y el payload a procesar. Recoge el tipo de payload que es y busca la petición correspondiente. Ejecutando el método processPaylaod() de la petición.
addUpdatePayload()	Actúa de igual manera que addPayload() pero recibiendo y procesando un addUpdatePayload.
addOutGoingPayload()	Actúa de igual manera que los dos anteriores. Recibe un payload, pero de tipo saliente y no entrante.

onEndpointLost()	Recibe el ID del dispositivo y el tipo de la petición y ejecuta el método onEndpointLost() en la petición.
onEndpointDisconnected()	Recibe el ID del dispositivo y el tipo de la petición y ejecuta el método onEndpointDisconnected() en la petición.

Tabla 9: Métodos del manejador de peticiones

El código completo puede verse en el [anexo](#).

## 6.4 La actividad Base

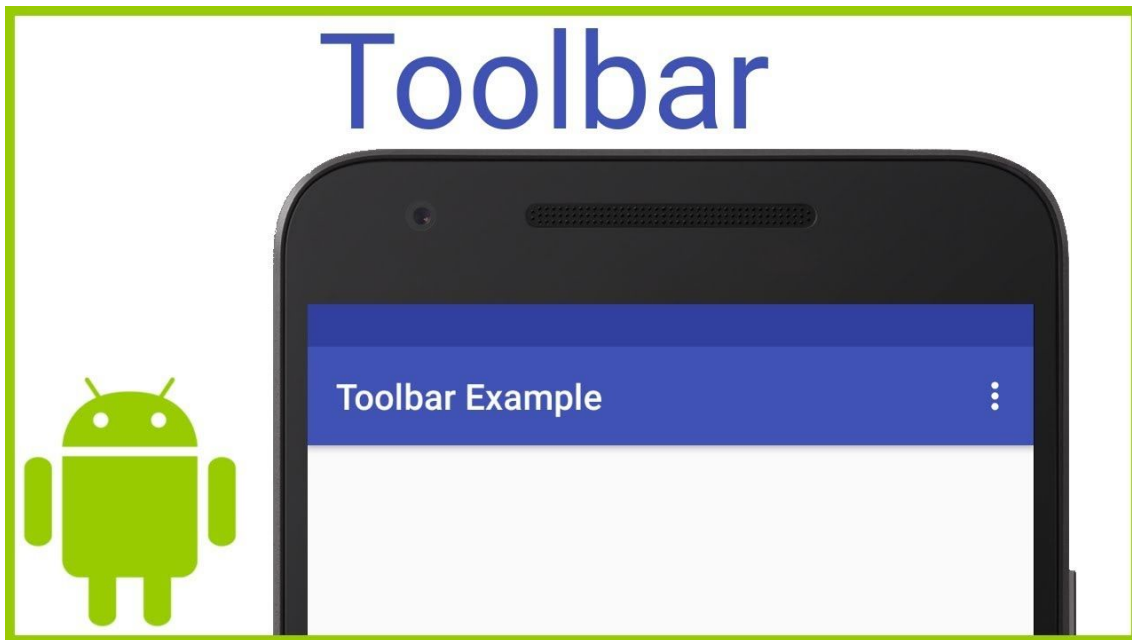
Siguiendo un poco con el desarrollo cronológico del proyecto, una vez creada la clase que iba a manejar la funcionalidad de las conexiones se comienza a trabajar en un primer prototipo de aplicación.

Anteriormente y como se puede ver en la ilustración 11, todos los controladores, o actividades, extienden una clase Base. Esta clase base contiene funcionalidad común a todos los controladores. Estas son las siguientes :

- Comprueba los permisos definidos en el manifiesto de la aplicación. Si no están aceptados los pide.
- Establece el contenedor de la vista.
- Recupera el servicio que da acceso a los métodos de Nearby. Esto lo hace con una llamada a *getApplication()*.
- Establece el Toolbar <sup>17</sup>.
- Bindea los métodos y los campos para que la librería ButterKnife funcione sobre esta actividad.
- Establece un método abstracto denominado *setServices()* donde cada actividad debe establecer en el servicio que llamadas quiere recibir en función de las interfaces que implementa.
- Inicializa el gestor de notificaciones.
- Inicializa el gestor de logs Timber.

<sup>17</sup> "Set up the app bar | Android Developers." <https://developer.android.com/training/appbar/setting-up>. Se consultó el 7 jun.. 2019.





*Ilustración 20: ToolBar en android*

El código completo de la clase Base se puede encontrar en el [anexo](#).

## 6.5 Peticiones

En este apartado hablaremos de una de las partes más importantes de la aplicación, las peticiones. Cuando invocamos cualquier servicio entre dos dispositivos se dispara una petición que se guarda en memoria hasta que se finaliza, da error, el usuario la cancela o bien salta un aviso de tiempo cumplido (timeout).

Todas las comunicaciones entre una petición y el controlador actual se realizarán a través de la interfaz MyNearbyPetitionListener. Se llamará al servicio y si existe se invocará con los parámetros necesarios.

Todo payload que sale de una petición lleva incluido en su identificador, un par de números que identifican el tipo de petición. De esta manera es sencillo en el otro dispositivo procesar el payload.

Una petición siempre hereda de la clase AbstractPetitionNearby, y por lo tanto todas comparten una serie de atributos y de funcionalidades parecidas. La lógica concreta de cada petición se implementa por encima. Los atributos más importantes que se comparten entre peticiones son los siguientes :

Nombre del atributo	Tipo	Funcionalidad
isStarted	Boolean	Indica si ha empezado la petición
isCompleted	Boolean	Indica si se ha completado la petición
isFailure	Boolean	Indica si la petición ha fallado o se ha cancelado.
isTimeout	Boolean	Indica si la petición ha superado el tiempo permitido.
incomingFilePayloads	SimpleArrayMap<Long, Payload>	Se guardan los payloads que recibimos
outgoingPayloads	SimpleArrayMap<Long, Payload>	Se guardan los payloads que enviamos
destinationEndpoint	Endpoint	Dispositivo con el que estamos tratando la petición
service	MyNearbyConnectionService	El servicio de Nearby para poder comunicarnos con los controladores
TAG	String	Tag utilizado para los logs

*Tabla 10: Atributos comunes de una petición*

Los métodos más importantes que tiene que sobrescribir una petición son los siguientes:

Nombre del método	Funcionalidad
processIncomingPetitionUpdate	Cómo debe tratar la petición específicamente el payloadUpdate entrante.
processOutgoingPetitionUpdate	Cómo debe tratar la petición específicamente el payloadUpdate de salida.
onPetitionTimeOut	Cómo debe comportarse la petición si se consume el tiempo que se le otorga al empezar.
startPetition	Lógica para el inicio de una petición
onEndpointLost/Disconnected	Tratamiento cuando se pierde o desconecta un dispositivo en medio de una petición.
isPayloadFromPetition	Recibe un número, y comprueba si este se corresponde con la petición donde se llama.

*Tabla 11: Métodos comunes de una petición*

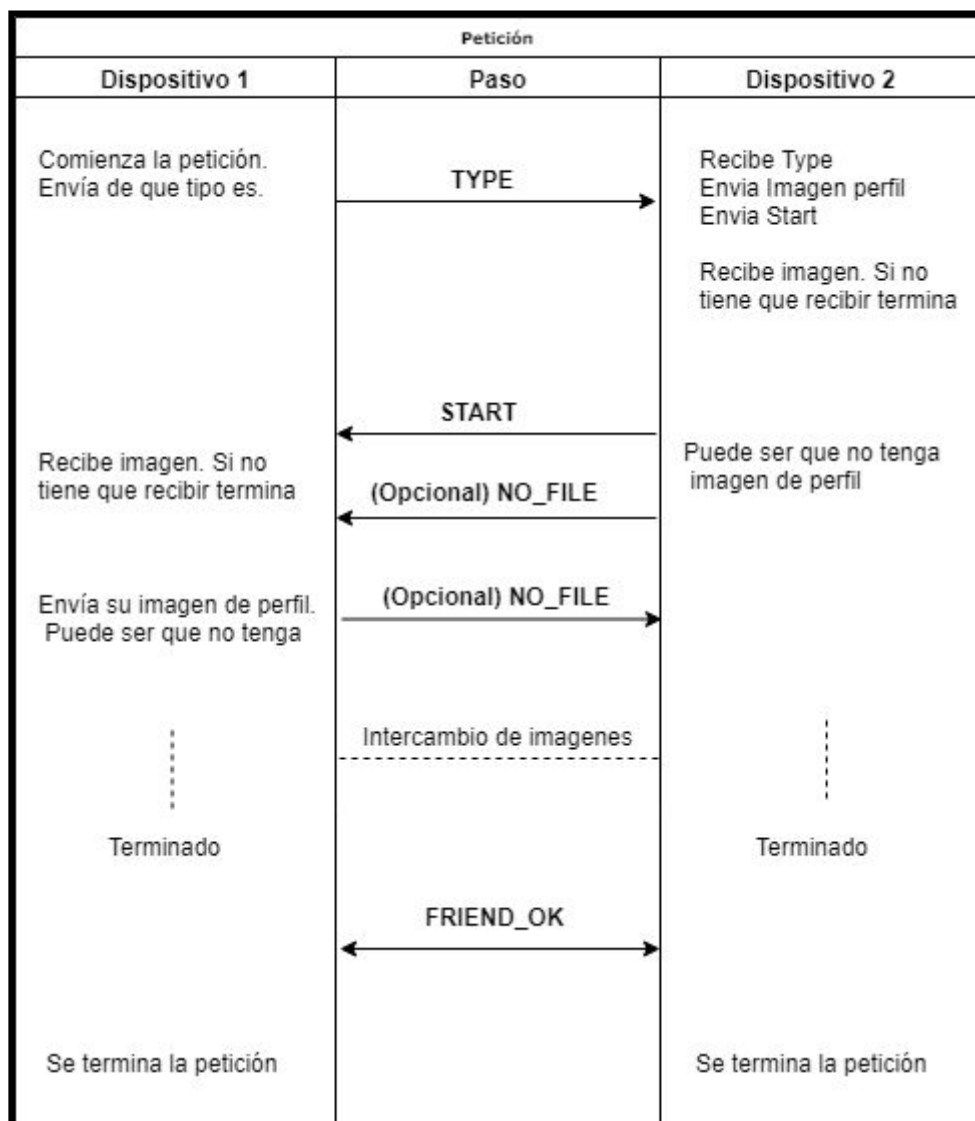
Una petición se ejecuta por pasos. Recibe instrucciones del dispositivo con el que está conectado y avanza de paso. Actualizando el controlador y la vista con los datos del paso. Por ejemplo en una transmisión de fichero, irá actualizando el porcentaje de archivo enviado o recibido.

Tenemos varios tipos de peticiones en el sistema :

- Peticiones de amistad
- Peticiones de chat
- Peticiones de envío de archivo
- Peticiones de archivo multiple

### 6.5.1 Petición de amistad

En una petición de amistad se intercambian los datos entre dispositivos así como la imagen de perfil si existe guardándose en el modelo o base de datos. Los pasos que se siguen se pueden observar en la siguiente imagen :



*Ilustración 21: Flujo de datos en una petición de amistad*

#### 6.5.1.1 Tipos de peticiones

Los payloads llevan un número de paso asociado que puede ir del 10 al 19. Las funcionalidades son las siguientes :

Tipo	Uso
REQUEST_FRIEND_TYPE	Lanzada por el emisor de la petición, indica comienzo de proceso.
REQUEST_FRIEND_START	Lanzada por el receptor cuando recibe un type, para indicar que comienza con el proceso.
REQUEST_FRIEND_IMAGE_PROFILE	Se envía a la actividad cuando recibe un archivo, parcial o completo.
REQUEST_FRIEND_WAITING	Esperando a que termine el dispositivo contrario.
REQUEST_FRIEND_NO_FILE	Mandada por el dispositivo cuando no hay imagen de perfil
REQUEST_FRIEND_END_OK	Dispositivo ha terminado, se lo comunica al contrario.
REQUEST_FRIEND_TIMEOUT	Tiempo terminado en dispositivo emisor.
REQUEST_FRIEND_ERROR	Error en dispositivo emisor.

*Tabla 12: Pasos o tipos de payloads de petición de amistad*

### 6.5.2 Petición de chat

En una petición de chat, el dispositivo emisor manda una petición y espera la respuesta del receptor. Este puede aceptar o rechazar. Si se acepta y no se ha cumplido el tiempo de espera, se lanza una pantalla de chat donde pueden intercambiar mensajes a través de la petición.

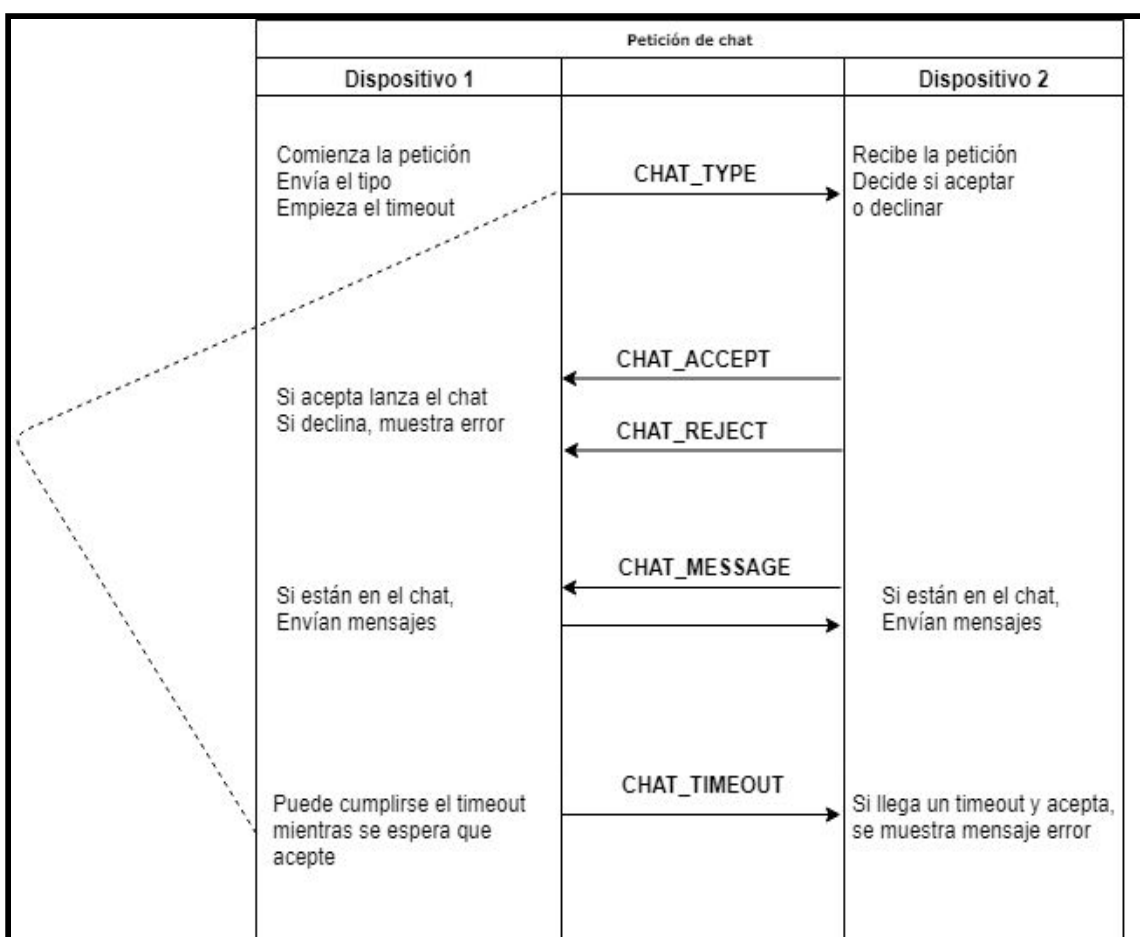


Ilustración 22: Flujo de datos en una petición de chat

#### 6.5.2.1 Tipos de peticiones

Los payloads llevan un número de paso asociado que puede ir del 20 al 29. Las funcionalidades son las siguientes :

Tipo	Uso
CHAT_TYPE	Lanzada por el emisor de la petición, indica comienzo de proceso.
CHAT_START	Lanzada por el receptor cuando recibe un type, para indicar que comienza con el proceso.
CHAT_ACCEPT	Se envía para confirmar que se inicia un chat al dispositivo receptor.

CHAT_REJECT	Se envía para rechazar conexión con el dispositivo receptor.
CHAT_MESSAGE	Se utiliza para intercambiar mensajes cifrados.
CHAT_END	Se utiliza para comunicar el fin del chat
CHAT_TIMEOUT	Tiempo terminado en dispositivo emisor.
CHAT_ERROR	Error en dispositivo emisor.

*Tabla 13: Pasos o tipos de payloads para peticiones de chat*

### 6.5.3 Petición de envío de archivo

En una petición de envío de archivo, el usuario que lo recibe solo se notifica al final del mismo. El usuario emisor podrá ver visualmente en pantalla el progreso de envío.

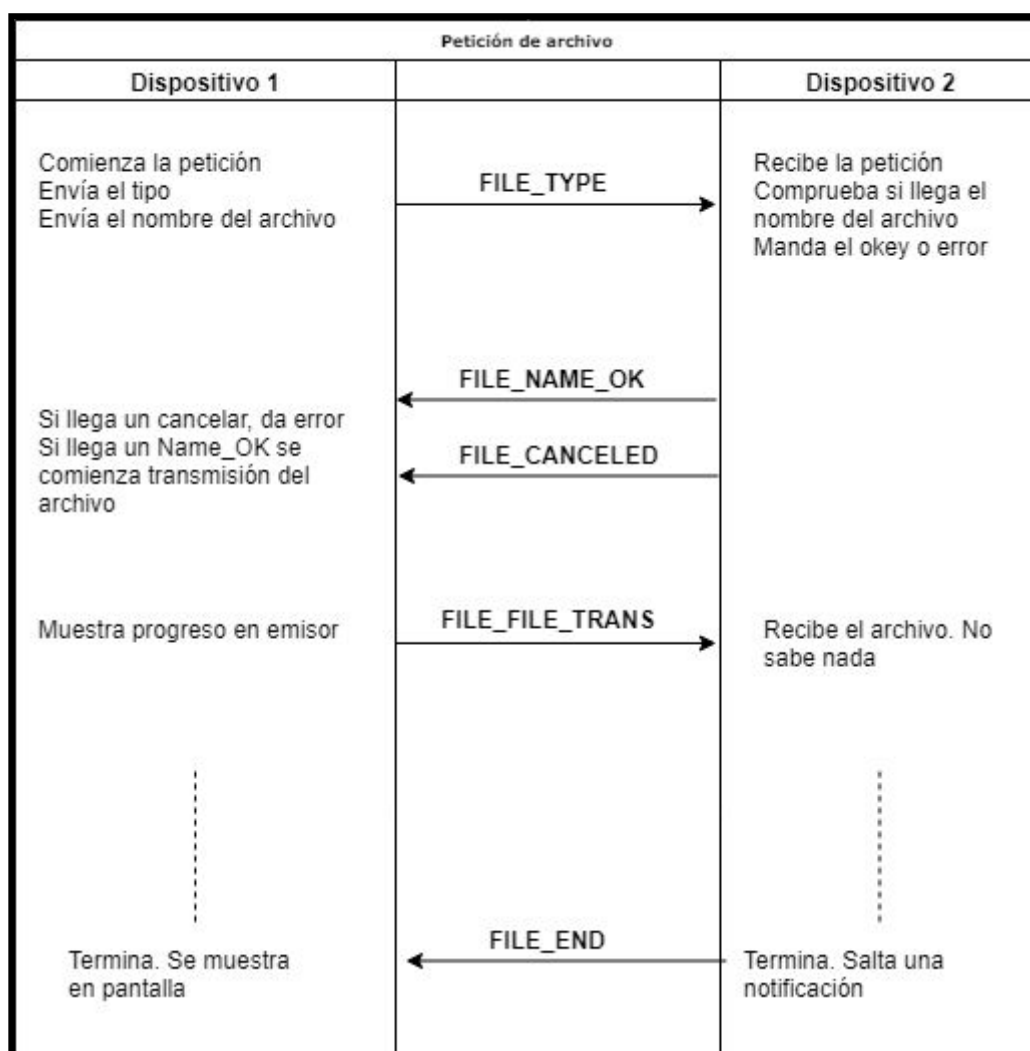


Ilustración 23: Flujo de datos en una petición de archivo

### 6.5.3.1 Tipos de peticiones

Los payloads llevan un número de paso asociado que puede ir del 30 al 39. Las funcionalidades son las siguientes :

Tipo	Uso
FILE_EXCHANGE_TYPE	Lanzada por el emisor de la petición, indica comienzo de proceso.
FILE_EXCHANGE_NAME_OK	Lanzada por el receptor cuando recibe un type, para indicar que ha recibido el type correctamente y que en el viene el



	nombre del fichero que va a recibir.
FILE_EXCHANGE_IN_PROGRESS	Lo recibe el receptor, en progreso un fichero entrante.
FILE_EXCHANGE_IN_PROGRESS_OUTSIDE	Lo recibe el emisor, fichero saliente.
FILE_EXCHANGE_END	Envío de fichero terminado.
FILE_EXCHANGE_TIMEOUT	Tiempo terminado en dispositivo emisor.
FILE_EXCHANGE_ERROR	Error en dispositivo emisor.

*Tabla 14: Pasos o tipos de payload para peticiones de envío de archivos*

#### 6.5.4 Petición de envío de archivo múltiple

En una petición de archivo múltiple, el dispositivo emisor intenta conectarse con todos los dispositivos amigos que se encuentren en el rango. Una vez conectado va enviando la imagen. Este proceso es , al igual que anterior, invisible de cara al emisor que solo es notificado una vez llega el archivo.

## Trabajo de fin de máster

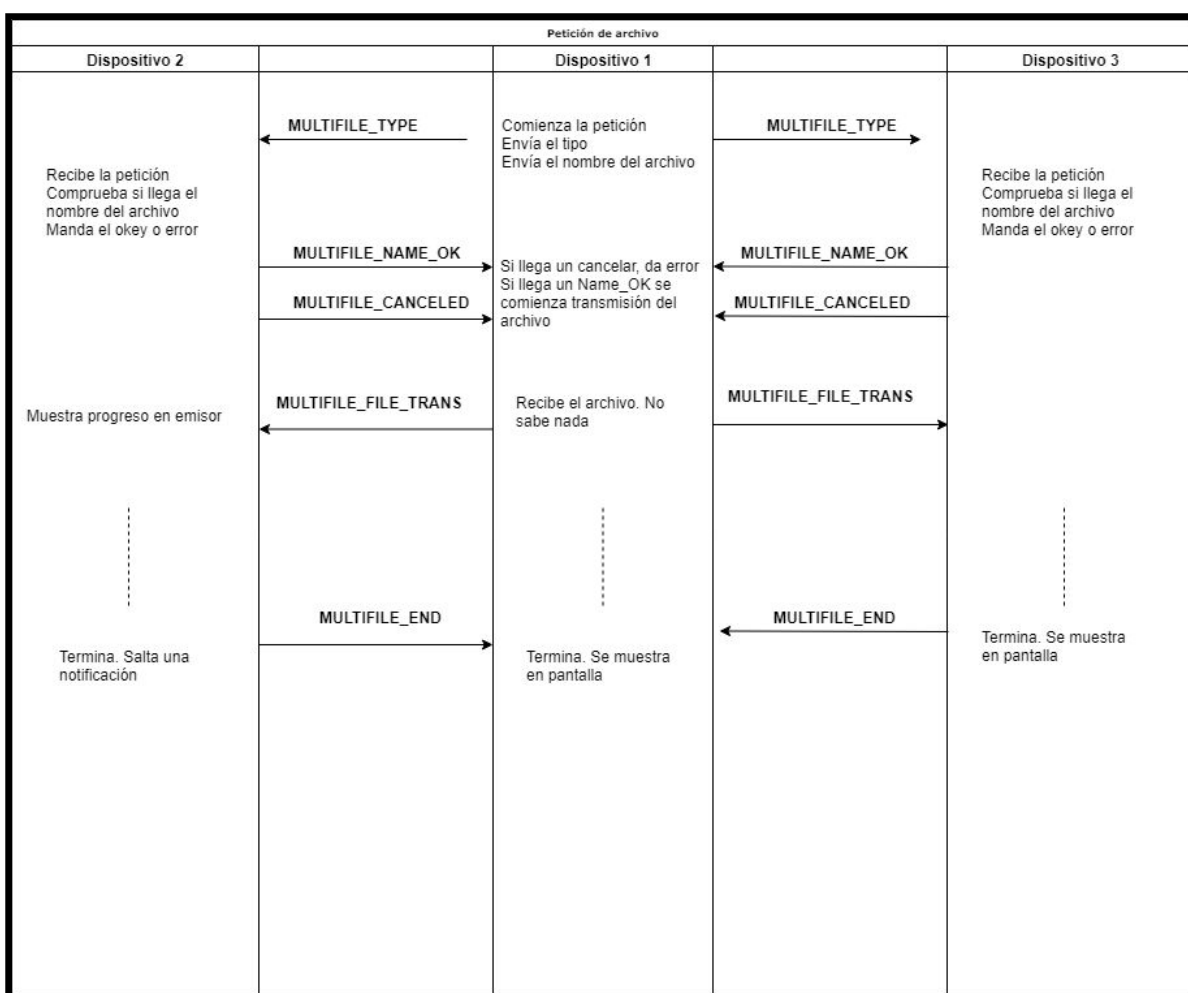


Ilustración 24: Flujo de datos en petición de archivo multiple

### 6.5.4.1 Tipos de peticiones

Los payloads llevan un número de paso asociado que puede ir del 40 al 49. Las funcionalidades son las siguientes :

Tipo	Uso
FILE_MULTIEXCHANGE_TYPE	Lanzada por el emisor de la petición, indica comienzo de proceso.
FILEMULTIEXCHANGE_ON_CONNECT	En emisor, conectando al receptor.
FILE_MULTIEXCHANGE_CONNECTED	En emisor, conectado al receptor.
FILE_MULTIEXCHANGE_PROCESSIN	Lo recibe el receptor, en progreso un

G	fichero entrante.
FILE_MULTIEXCHANGE_NAME_OK	Lanzada por el receptor cuando recibe un type, para indicar que ha recibido el type correctamente y que en el viene el nombre del fichero que va a recibir.
FILE_MULTIEXCHANGE_FILERECEIVED_END	En emisor, envío finalizado en receptor
FILE_MULTIEXCHANGE_FILE_RECEIVED_ERROR	En emisor, Error en dispositivo receptor.
FILE_MULTIEXCHANGE_ERROR	Error en la petición
FILE_MULTIEXCHANGE_NO_ENDPOINTS	No hay dispositivos cercanos

*Tabla 15: Pasos o tipos de payloads en envío de archivo múltiple*

## 6.6 Vistas, controladores e interfaz

Anteriormente y a lo largo de la memoria hemos repasado los siguientes conceptos :

- Qué es Nearby y cómo funciona.
- El modelo de arquitectura MVC que vamos a emplear.
- Cómo está implementada la API de Nearby en la clase contexto de la aplicación.
- Qué es la clase base que comparten todos los controladores.
- Qué es una petición, cómo se manejan y que tipos hay.
- Cómo es el flujo de una petición.

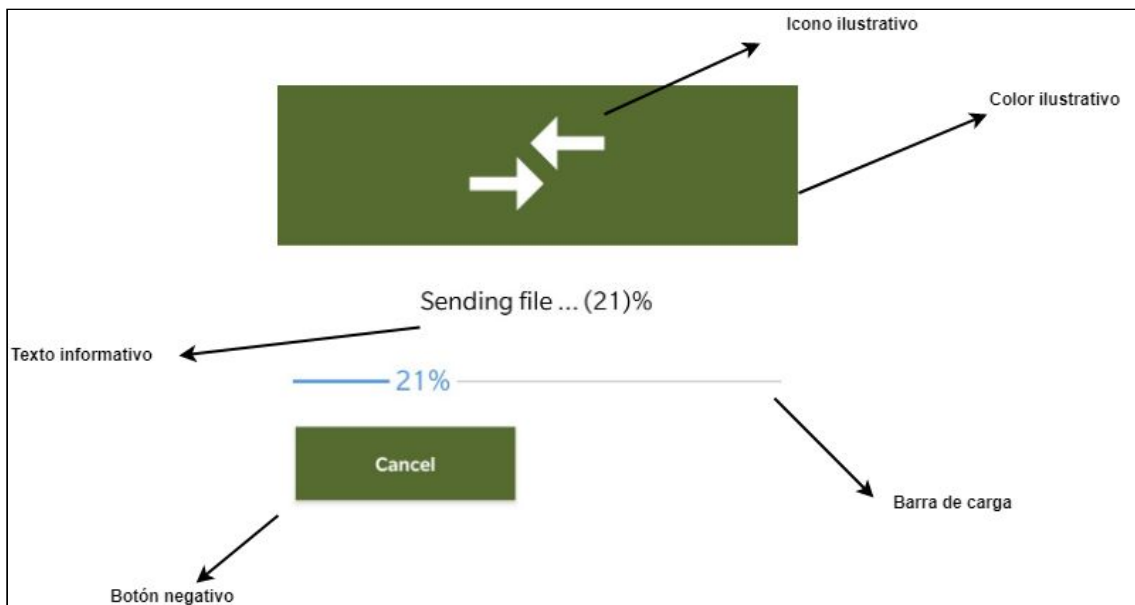
Una vez repaso estos aspectos podemos adentrarnos y ver la interfaz, y cómo se relaciona esta con el controlador. Como este lanza las peticiones y en función del estado de la petición actualiza la vista. Además de otras funcionalidad de la pantalla.

En el sistema disponemos de cuatro vistas o pantallas. Una sería la principal, desde se realizan la mayoría de las gestiones y denominada principal. Otra sería la lista de amigos. La siguiente es la sala del chat y finalmente la última sería la vista de opciones.

Antes de hablar específicamente sobre cada vista vamos a hablar sobre las ventanas emergentes o dialogs, que informan sobre el progreso, errores o avisos en el sistema.

### 6.6.1 Custom Dialog

En nuestro sistema hemos creado un diálogo o Dialog personalizado de tal manera que podemos cambiar el color, el texto, el logo o la barra de progreso en función de lo que queramos mostrar. De esta manera podemos mostrarle al usuario información de manera intuitiva.



*Ilustración 25: Ejemplo de dialog positivo*

Se pueden observar varios aspectos :

- Primeramente el icono, que representa la acción que se lleva a cabo en este momento. Por ejemplo en la imagen representa un intercambio de archivos.
- El color indica el estado de la petición. Verde si todo va bien, rojo si ha fallado algo y amarillo si se quiere indicar algo.
- El texto informativo , como su nombre indica, se encarga de darle feedback al usuario sobre el proceso.
- La barra de carga solo aparece cuando es necesario, puede ser finita con un porcentaje, o infinita indicando la espera de algo.

- Finalmente hay dos botones abajo, en este caso solo aparece el negativo. Podría aparecer a la derecha el positivo. Ambos son totalmente personalizables.

La clase base se encarga de instanciar el gestor de Dialogs que permite crear diálogos de manera muy intuitiva en el proyecto.

### 6.6.2 Módulo Home

Comenzaremos hablando del módulo más grande de la aplicación. La vista principal que se ve nada más arrancar la aplicación.

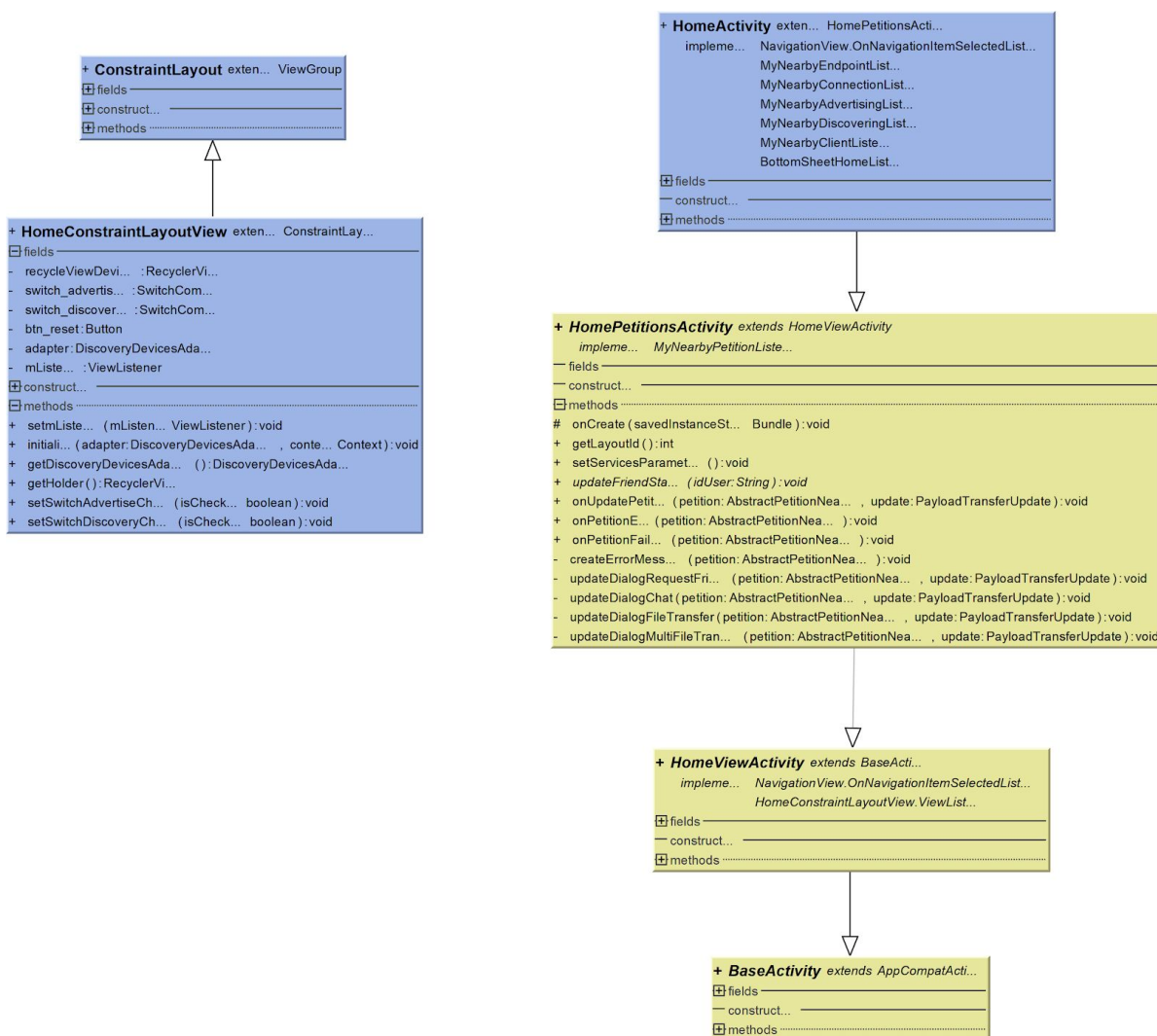


Ilustración 26: Diagrama de clases del módulo Home

Como hemos visto en el capítulo de la arquitectura el módulo home está compuesto por una vista, en este caso el **HomeConstraintLayoutView** y el controlador. El controlador está dividido en tres partes. Una primera que el **HomeViewActivity** que inicializa los componentes de la vista. La siguiente el **HomePetitionsActivity**, esta es la más importante y recibe las entradas de las peticiones y se encargará de actualizar la vista en base a estos mensajes. Finalmente el **HomeActivity** que se encarga del resto de funcionalidades de la vista. Estas son definidas por las diversas interfaces que definimos en la ilustración 20.

### 6.6.2.1 Interfaz

La pantalla de inicio puede verse de la siguiente manera :

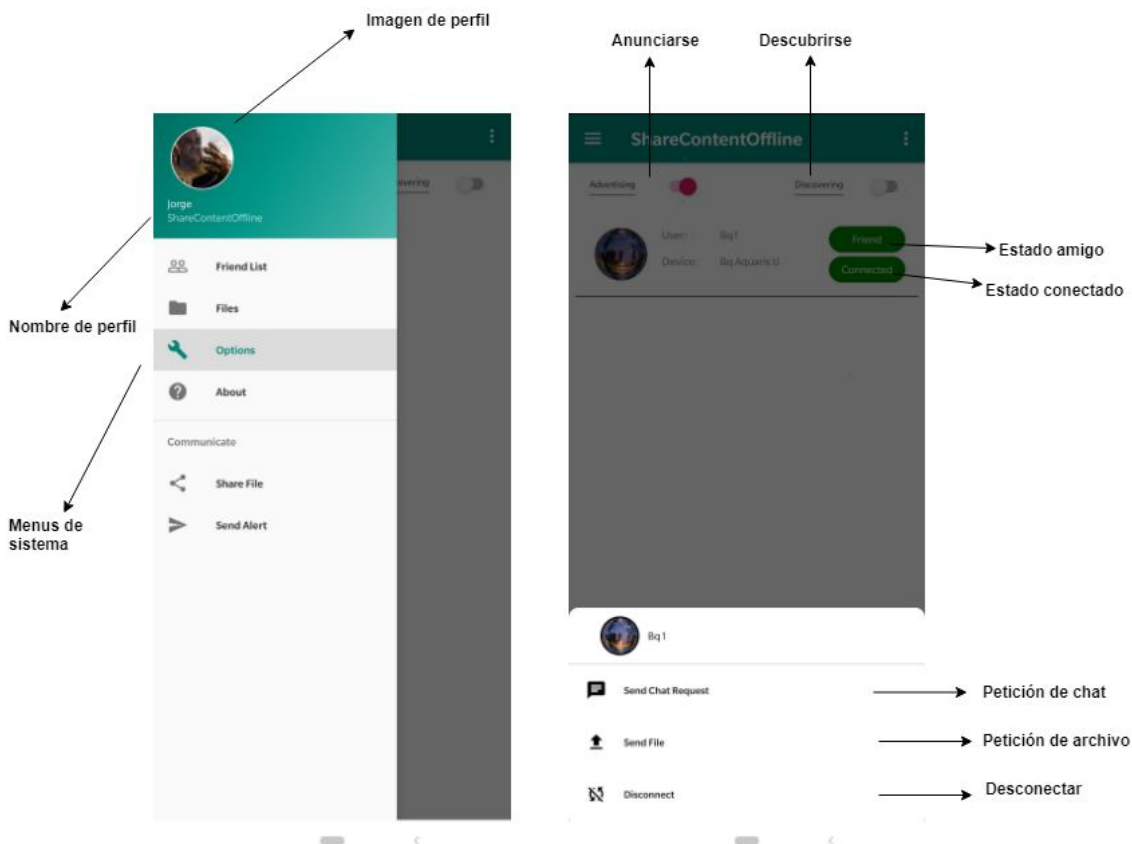


Ilustración 27: Vista y funcionalidad de la pantalla home

Como podemos observar la pantalla principal tiene bastante complejidad y permite realizar bastantes acciones.

Si desplegamos el menú lateral en la imagen de la izquierda :

- Imagen de perfil. Tocando sobre ella se puede cambiar. El tamaño máximo es 200 KB. Esto se hace para agilizar en la medida de lo posible la petición de amistad.
- Acceso a la lista de amigos. Se hablará de ella más adelante. Es un módulo aparte.
- Acceso a los archivos. Esto redirige a la carpeta donde se guardan los archivos transmitidos. Está alojado en Downloads/Nearby.
- About. Información general sobre el funcionamiento de la aplicación.
- Share File. Esto permite enviar un archivo múltiple a todos los amigos cercanos.
- Send Alert. Esto permite enviar una notificación a todos los amigos cercanos. Tiene un timeout para evitar el spam.

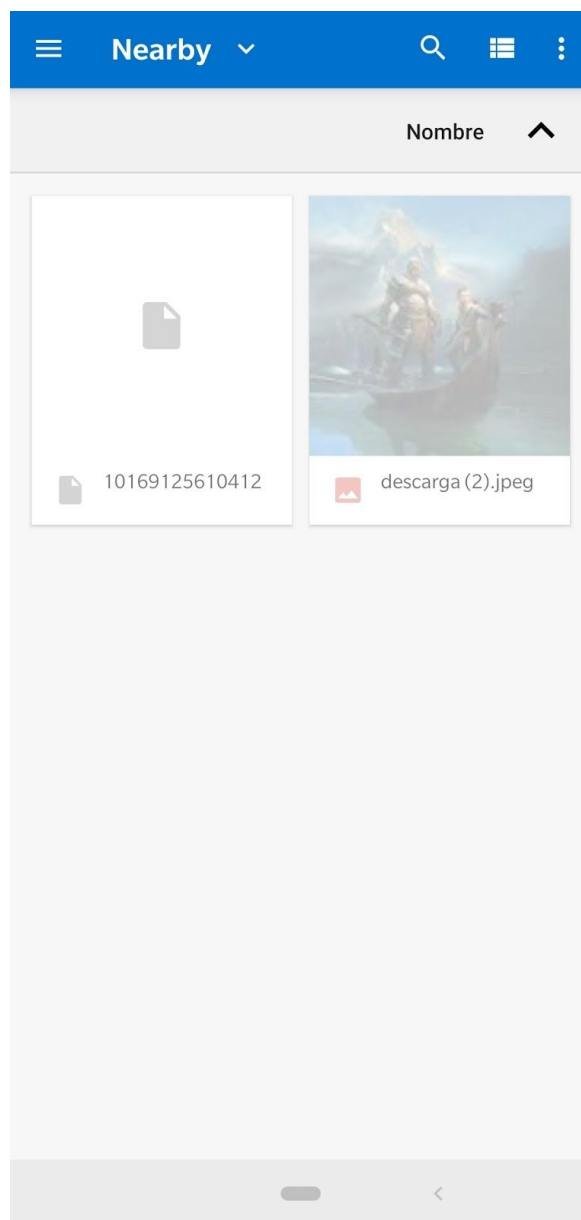
En la vista sin despegar en la imagen derecha :

- Dos primeros botones que permiten activar el anunciarte o descubrirte.
- Una lista rellena con los dispositivos. Se actualiza si aparecen nuevo o desaparece alguno previamente descubierto. Cada dispositivo actualiza además sus estados internos de amigo y conexión.
- Si pulsamos en un dispositivo, se abre un menú con las diferentes acciones a realizar. Cada una dispara una petición con el flujo comentado anteriormente. Exceptuando la desconexión, que realiza la acción que nombre indica.
- Un botón de reset justo debajo de la lista que permite borrar todos los dispositivos, y desconectarse si previamente estaban conectados.

Cuando seleccionamos la opción de ver archivos saldría una pantalla como la siguiente:



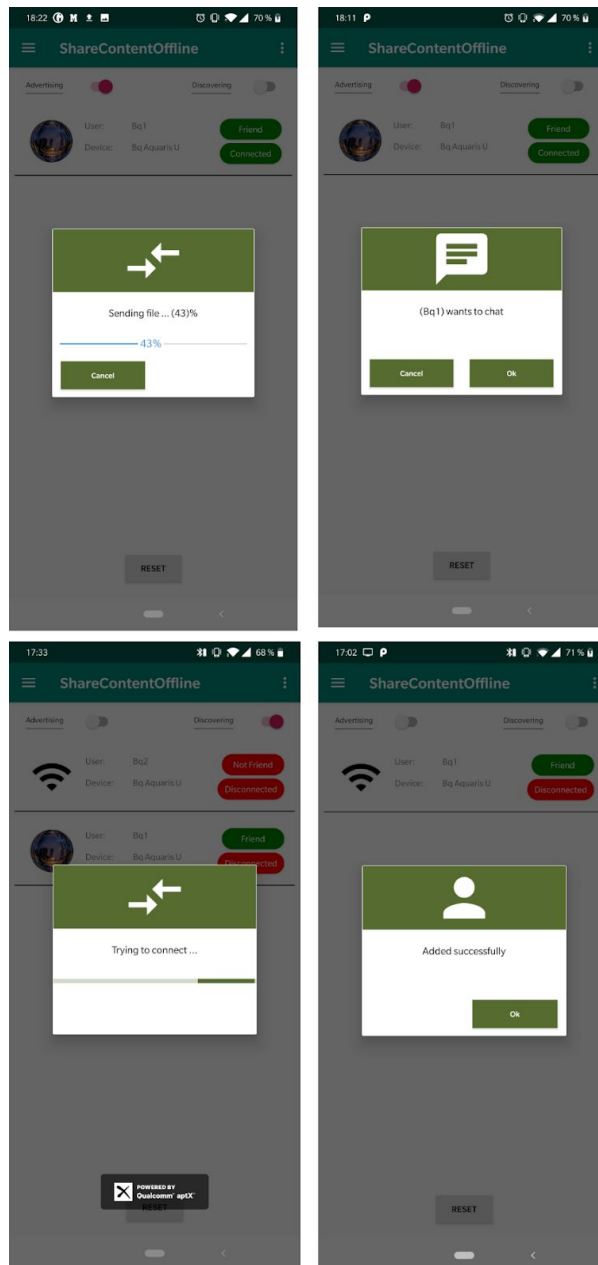
## Trabajo de fin de máster



*Ilustración 28: Opción files*

### 6.6.2.2 Dialogs y notificaciones

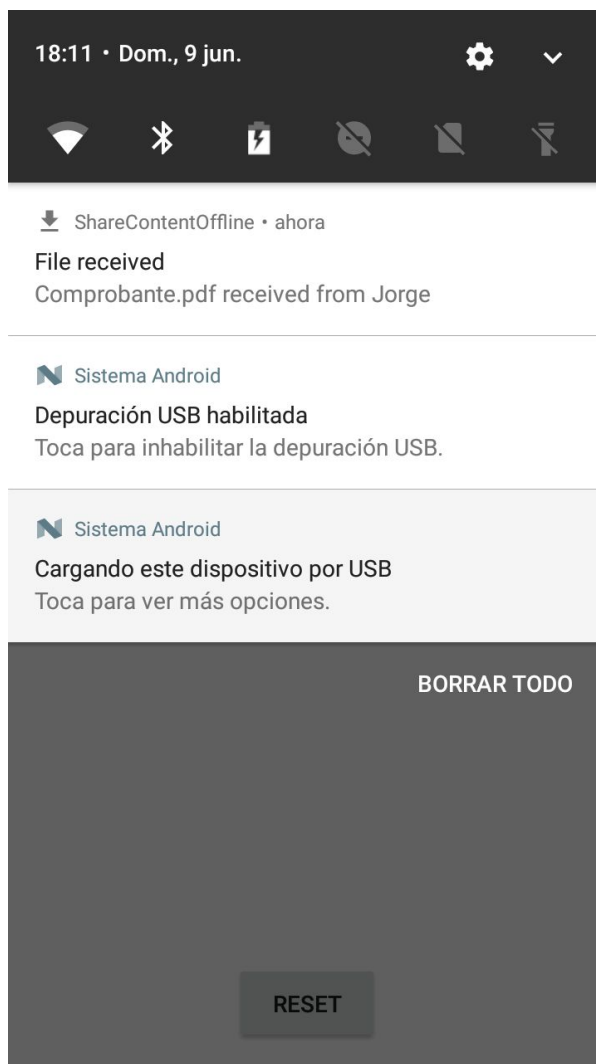
Vamos a ver algunos ejemplos de dialogs en función de las peticiones lanzadas.



*Ilustración 29: Ejemplos de diálogos en Home*

A continuación vamos a ver un ejemplo de cuando recibimos una notificación

## Trabajo de fin de máster



*Ilustración 30: Ejemplo de notificación en Home*

### 6.6.2.3 Controlador

En este apartado vamos a comentar el funcionamiento del controlador cuando recibe una petición. Es decir vamos a ver cómo se comporta la clase HomePetitionsActivity al ser la que extiende MyNearbyPetitionsNearby.

Esta clase implementa básicamente 3 métodos.

- OnUpdatePetition()

```
public void onUpdatePetition(AbstractPetitionNearby petition,
PayloadTransferUpdate update) {

    switch (petition.getType()) {
        case (RequestFriendPetition.REQUEST_FRIEND_TYPE):
            updateDialogRequestFriend(petition, update);
            break;
        case (Chatpetition.CHAT_TYPE):
            updateDialogChat(petition, update);
            break;
        case (FileExchangePetition.FILE_EXCHANGE_TYPE):
            updateDialogFileTransfer(petition, update);
            break;

        case (MultiFilesExchange.MULTIFILE_EXCHANGE_TYPE):
            updateDialogMultiFileTransfer(petition, update);
            break;

    }
}
```

Lo que realiza este método es obtener el tipo de petición y disparar una función que ejecuta acciones específicas en función del paso en el que se encuentre. Pongamos por ejemplo la función disparada si es una petición de tipo Chat.

```
private void updateDialogChat(AbstractPetitionNearby petition,
PayloadTransferUpdate update) {
    String text;
    int step = petition.getPetitionStep();
```

```
switch (step) {
    case Chatpetition.CHAT_START:
        dialogManager.createChatDialog();
        break;
    case Chatpetition.CHAT_TYPE:
        text = getString(R.string.chat_type,
petition.getEndpointUsername());
        dialogManager.createChatIncomeDialog(text, petition);
        break;
    case Chatpetition.CHAT_ACCEPT:
        Intent intent = new Intent(this, ChatActivity.class);
        intent.putExtra("endpointConnected", petition.getEndpointID());
        dialogManager.getActiveDialog().dismiss();
        startActivity(intent);
        break;
    case Chatpetition.CHAT_REJECT:

dialogManager.updateErrorDialog(getString(R.string.chat_rejected));
        break;
    case Chatpetition.TIMEOUT:

dialogManager.updateErrorDialog(getString(R.string.chat_timeout));
        break;
}
```

}

En esta función leerá el paso en el que nos encontramos y en función de este actualizará el Dialog dándole información al usuario. Se puede observar que el gestor de diálogos ya tiene funciones para crear el tipo de dialog que queramos.

- onPetitionEnd()

Esta función es la invocada cuando queremos acabar la petición. Puede ser porque hemos finalizado correctamente, porque se ha acabado el tiempo o porque el usuario la ha cancelado.

```
public void onPetitionEnd(AbstractPetitionNearby petition) {
    logI("ON PETITION END ");
}
```

```

        if (petition.getPetitionStep() == AbstractPetitionNearby.TIMEOUT &&
service.getPetition(petition.getEndpointID(), petition.getType()) != null)
{
    dialogManager.updateErrorDialog(getString(R.string.dialog_timeout,
petition.getEndpointUsername()));
    service.deletePetition(petition.getEndpointID(),
petition.getType());
} else if (petition.getPetitionStep() ==
AbstractPetitionNearby.CANCELED) {
    service.deletePetition(petition.getEndpointID(),
petition.getType());
} else {
    switch (petition.getType()) {

        case (RequestFriendPetition.REQUEST_FRIEND_TYPE):
            updateDialogRequestFriend(petition, null);
            RequestFriendPetition pet = (RequestFriendPetition)
petition;
            logD("FRIEND_ Se va a desconectar " +
pet.isDisconnectionOk());
            if (pet.isDisconnectionOk())
                service.disconnect(petition.getDestinationEndpoint());

            service.deletePetition(petition.getEndpointID(),
petition.getType());
            break;
        case (MultiFilesExchange.MULTIFILE_EXCHANGE_TYPE):
getBundle().remove(KConstantesShareContent.BUNDLE.IS_MULTI_FILE);
            service.deletePetition("0", petition.getType());
            updateDialogMultiFileTransfer(petition, null);

            break;
        case (FileExchangePetition.FILE_EXCHANGE_TYPE):
            updateDialogFileTransfer(petition, null);
            service.deletePetition(petition.getEndpointID(),
petition.getType());
            break;
        default:
            service.deletePetition(petition.getEndpointID(),
petition.getType());
    }
}
}
}

```


En esta función se comprueba primeramente si es un TimeOut o si es una petición cancelada por el usuario mostrando el mensaje correspondiente al usuario y borrando la petición.

Si no es así, se actúa de manera diferente en función del tipo de petición entrante. En una petición de amistad comprueba si puede desconectarse o si todavía está enviando un archivo. En un envío de archivo comprueba si eres el emisor o el receptor para desplegar una notificación de archivo recibido.

- onPetitionFailure()

Esta función se invoca cuando hay un error en alguna parte del procesamiento de la petición. Errores típicos pueden ser la desconexión de un dispositivo o null pointers imprevistos.

```
public void onPetitionFailure(AbstractPetitionNearby petition) {
    switch (petition.getType()) {
        case (RequestFriendPetition.REQUEST_FRIEND_TYPE):
            service.disconnect(petition.getDestinationEndpoint());
            service.deletePetition(petition.getEndpointID(),
petition.getType());
            createErrorMessage(petition);
            break;
        case (MultiFilesExchange.MULTIFILE_EXCHANGE_TYPE):
            getBundle().remove(KConstantesShareContent.BUNDLE.IS_MULTI_FILE);
            service.disconnect(petition.getDestinationEndpoint());
            service.deletePetition("", petition.getType());
            createErrorMessage(petition);
            break;
        case (FileExchangePetition.FILE_EXCHANGE_TYPE):
            if (petition.isStarterPetition())
                createErrorMessage(petition);
            break;
        default:
            service.deletePetition(petition.getEndpointID(),
petition.getType());
            createErrorMessage(petition);
            break;
    }
}
```



El comportamiento para cada petición es similar, lanzando un mensaje de error descriptivo y eliminando la petición.

### 6.6.3 Módulo lista de amigos

En este módulo podremos gestionar los usuarios aceptados como amigos. Bloqueando temporalmente si fuera necesario. Un usuario bloqueado no podrá conectarse ni recibir datos.

Además en esta vista también seremos capaces de aceptar conexiones entrantes, aceptar peticiones de amistad y recibir notificaciones para ficheros entrantes. Para ellos se han implementado en el controlador las interfaces necesarias.



## Trabajo de fin de máster

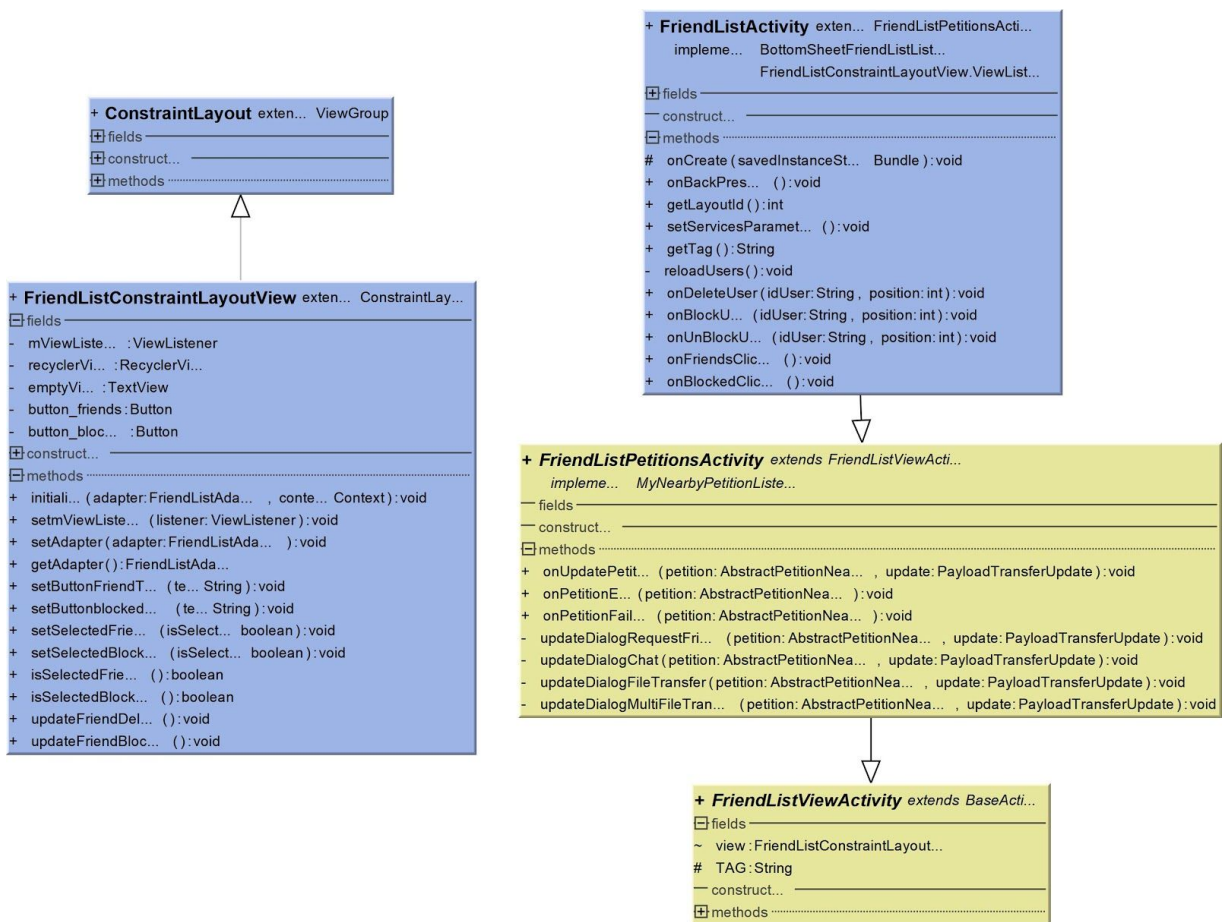
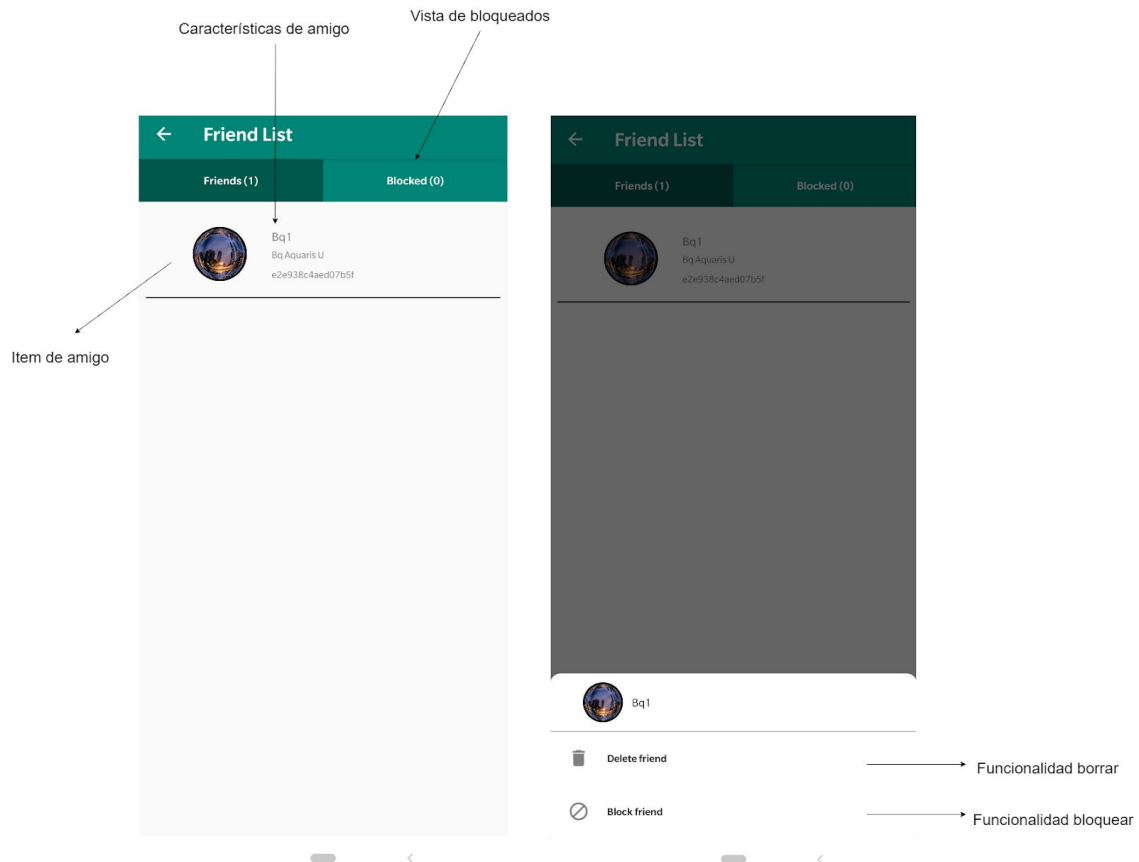


Ilustración 31: Diagrama de clases de la lista de amigos

### 6.6.3.1 Interfaz

La pantalla de amigos puede verse de la siguiente manera :



*Ilustración 32: Vista y funcionalidad de la pantalla de amigos*

Al contrario que la vista principal aquí podemos ver una vista muy sencilla y con menos funcionalidad:

- Si pulsamos en la pestaña de amigos (seleccionada por defecto), podremos ver los amigos que tenemos añadidos y que no han sido bloqueados.
- Si pulsamos en la pestaña de bloqueados iremos a la vista de bloqueados donde se muestran aquellos previamente bloqueados.
- Si pulsamos en un elemento de la lista se accionará un menú en función de la vista donde nos encontremos :
  - Si es un amigo sin bloquear, se podrá **bloquear** o **eliminar**.

- Si es un amigo bloqueado, se podrá **desbloquear** o **eliminar**.

### 6.6.3.2 Dialogs

Los dialogs son iguales a los mostrados en el apartado anterior.

### 6.6.3.3 Controlador

En este apartado vamos a comentar el funcionamiento del controlador cuando recibe una petición. Es decir vamos a ver cómo se comporta la clase FriendListPetitionsActivity al ser la que extiende MyNearbyPetitionsNearby.

- onPetitionUpdate()

Se comporta igual que en la vista anterior. Con la diferencia de que solo puede recibir peticiones entrantes, el nunca inicia peticiones. Para ellos genera dialogs o notificaciones. Veamos un ejemplo de notificación.

```
case (MultiFilesExchange.MULTIFILE_EXCHANGE_TYPE):
    getBundle().remove(KConstantesShareContent.BUNDLE.IS_MULTI_FILE);
    service.deletePetition("0", petition.getType());

    MultiFilesExchange multiPetition = (MultiFilesExchange) petition;
    nearbyPath =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS).getAbsolutePath() + "/Nearby" + "/" + multiPetition.getFileName();
    file = new File(nearbyPath);
    fileUri = NearbyDownloadsFileProvider.getUriForFile(this,
this.getApplicationContext().getPackageName() + ".fileprovider", file);
    myIntent = new Intent(Intent.ACTION_VIEW, fileUri);
    myIntent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    pendingIntent = PendingIntent.getActivity(this, 0, myIntent,
Intent.FILL_IN_ACTION);

    notification =
ApplicationNotificationManager.generateNotificationFileDownloaded(this,
multiPetition.getFileName(), petition.getEndpointUsername(),
pendingIntent);
    notificationManager.notify(ApplicationNotificationManager.ID_REQUEST,
notification);
    break;
```

- onPetitionEnd()

Mismo comportamiento que en la pantalla anterior. Borra las peticiones según se acaban.

- onPetitionFailure()

Mismo comportamiento que en la pantalla anterior.

#### 6.6.4 Módulo Chat

En este módulo vamos a ver la funcionalidad del chat. Más sencillo incluso que los anteriores pues lo único que se despliega es un teclado para enviar mensajes.

A nivel del resto de funcionalidades, desde la pantalla de chat se rechazan las peticiones de amistad entrantes. Sin embargo somos capaces de recibir notificaciones por archivos entrantes.

## Trabajo de fin de máster

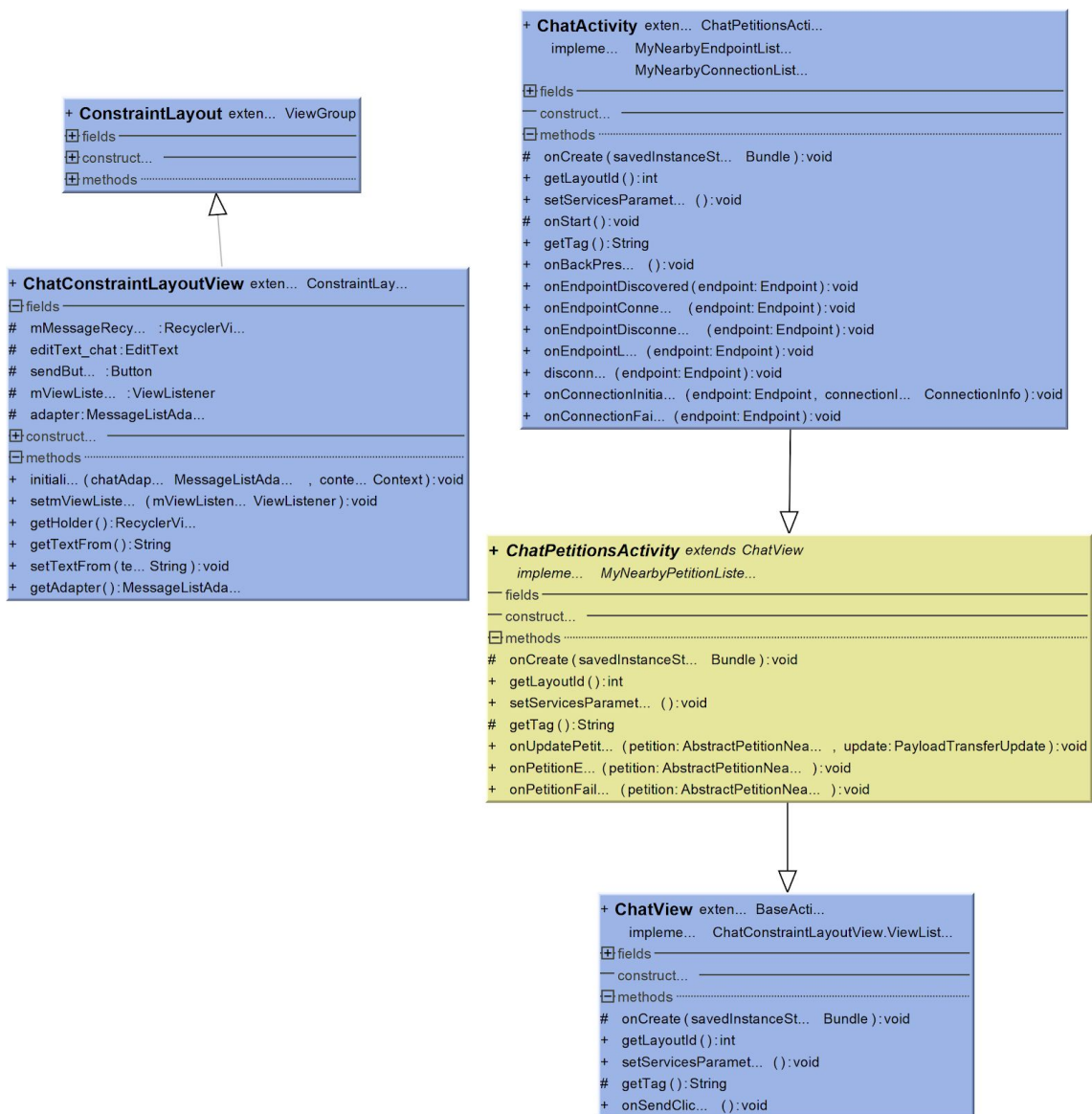


Ilustración 33: Diagrama de clases del módulo chat

#### 6.6.4.1 Interfaz

La pantalla de chat puede verse de la siguiente manera

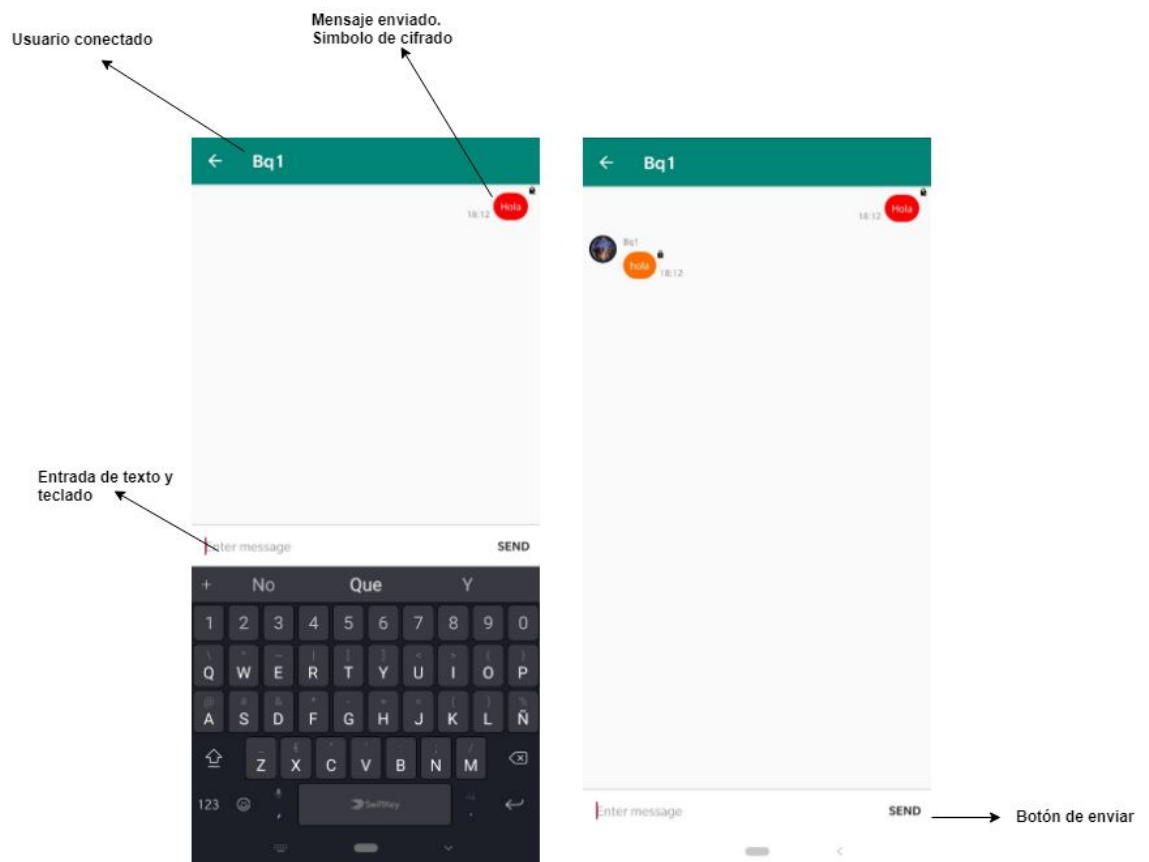


Ilustración 34: Vistas y funcionalidad de la pantalla de chat

Continuamos con una pantalla con funcionalidad reducida. El único elemento interactuable será el botón de enviar mensaje junto con el teclado.

#### 6.6.4.3 Dialogs

Los dialogs son iguales a los mostrados en el apartado anterior.

#### 6.6.4.4 Controlador

En este apartado vamos a comentar la funcionalidad del controlador.

- onUpdatePetition

```
@Override
public void onUpdatePetition(AbstractPetitionNearby petition,
PayloadTransferUpdate update) {

    if (petition.getType() == Chatpetition.CHAT_TYPE) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Chatpetition pet = (Chatpetition) petition;
                if (pet.getPetitionStep() == Chatpetition.CHAT_MESSAGE) {
                    UserMessage message = pet.getActualUserMessage();
                    message.setType(UserMessage.MESSAGE_RECEIVED);
                    if (message.isEncrypted()) {
                        String deEncryted =
cipher.decipherData(message.getMessage());
                        logI("__DESENCRIPTADO " + deEncryted);
                        message.setMessage(deEncryted.getBytes());
                    }
                    int number = mMessageAdapter.setMessage(message);
                    RecyclerView.ViewHolder holder =
mMessageAdapter.onCreateViewHolder(mainView.getHolder(),
VIEW_TYPE_MESSAGE_RECEIVED);
                    mMessageAdapter.onBindViewHolder(holder, number);
                    mMessageAdapter.notifyItemChanged(number);
                }
            }
        });
    }
}
```

Cuando llega un mensaje en la pantalla se recoge, se comprueba si se ha enviado cifrado y, si es así, se descifra y se pinta en pantalla. Si no viene cifrado se pinta en pantalla el mensaje que venga.

- onPetitionEnd()

Mismo comportamiento que en la pantalla anterior. Borra las peticiones según se acaban.

- `onPetitionFailure()`

Mismo comportamiento que en la pantalla anterior.

### 6.6.5 Módulo Opciones

Las opciones del sistema son una parte importante de la aplicación , pues nos permiten configurar la manera en que se comporta nuestro entorno ante los estímulos entrantes.

A diferencia de del resto de módulos, las opciones están implementadas de la manera indicada en la guía de desarrollo de Android <sup>18</sup> y por lo tanto no cuentan con una vista como tal. La manera en que se trata visualmente es idéntica para todos los dispositivos Android, a excepción, de los elementos propios que queramos añadir.

Hay multitud de elementos, desde ítems que capturan texto hasta switches para seleccionar si algo es verdadero o falso.

---

<sup>18</sup> "Settings Guide - Settings | Android Developers." <https://developer.android.com/guide/topics/ui/settings>. Se consultó el 10 jun.. 2019.



## Trabajo de fin de máster

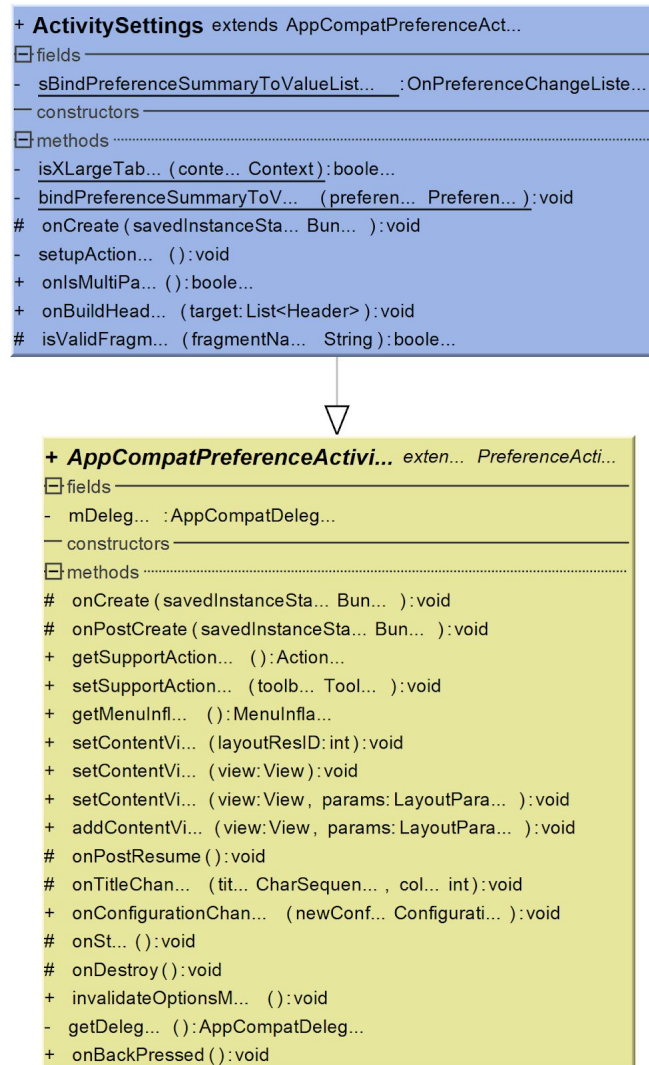
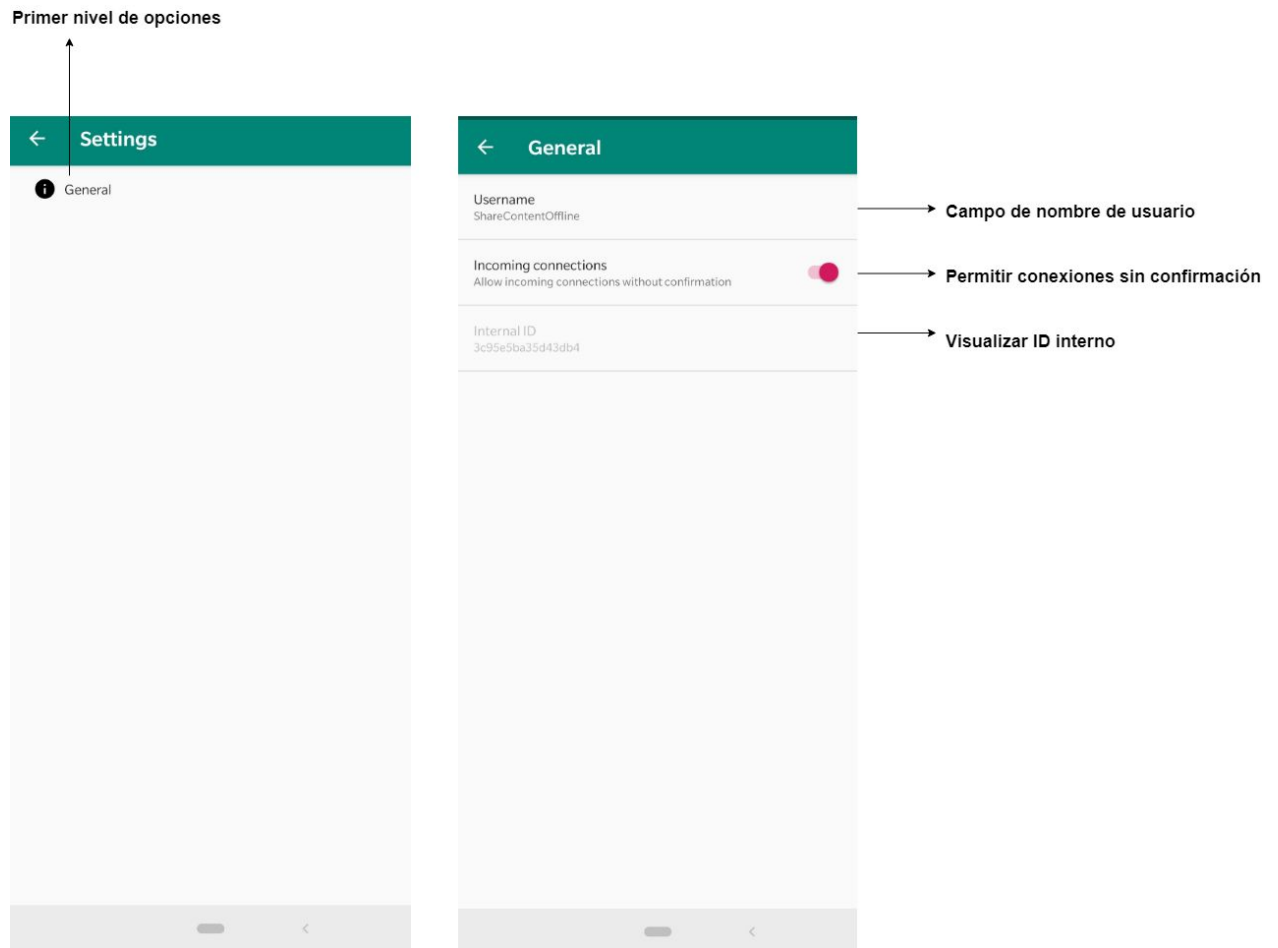


Ilustración 35: Diagrama de clases del módulo de opciones

#### 6.6.5.1 Interfaz

La pantalla de opciones puede verse de la siguiente manera



*Ilustración 36: Vistas y funcionalidades de la pantalla de opciones*

Como podemos observar podemos configurar en la aplicación 3 opciones :

- El nombre de usuario. Aquí sería donde se establecería nuestro nombre de usuario. Por defecto si es la primera vez que accedemos será ShareContentOffline.
- Permitir conexiones. Esta opción al activarse deja de mostrar un diálogo de confirmación cuando viene una conexión entrante. De esta manera podríamos recibir ficheros 1 a 1 sin necesidad de estar conectados. Así como peticiones de chat.
- Id del dispositivo. Se muestra el ID interno del sistema.

Cada opción es un elemento Preference en el sistema de android. Un elemento preference se configura mediante un fichero .xml . Veamos el ejemplo del nombre de usuario, que se configura mediante un EditTextPreference.

```
<EditTextPreference
    android:capitalize="words"
    android:defaultValue="@string/pref_default_display_name"
    android:inputType="textCapWords"
    android:key="@string/key_username"
    android:summary="@string/summary_general_username"
    android:maxLines="1"
    android:selectAllOnFocus="true"
    android:singleLine="true"
    android:title="@string/pref_title_display_name" />
```

Como podemos ver podemos configurar diversas opciones. Entre ellas se encuentran el título, la descripción, el número máximo de líneas, el tipo de entrada que acepta, el valor por defecto...

#### 6.6.5.3 Dialogs

No existen dialogs personalizados en este módulo.

#### 6.6.5.4 Controlador

La actividad controladora es generada por el sistema.

## 6.7 Servicios de seguridad

Como hemos comentado, uno de los principales objetivos del proyecto era la implantación de servicios de seguridad en el sistema. La aplicación está basada en los entornos de proximidad, es decir, no requerimos infraestructura de comunicaciones para funcionar. Pero esto no implica que un tercero pueda acceder a los datos que estamos enviando mediante los canales de este entorno de proximidad comprometiendo la confidencialidad de la información que transmitimos.

Además debemos asegurar cierta autenticidad en las conexiones que realizamos, es decir, debemos asegurar que nos conectamos con aquel que sabemos quien es y en el que confiamos.

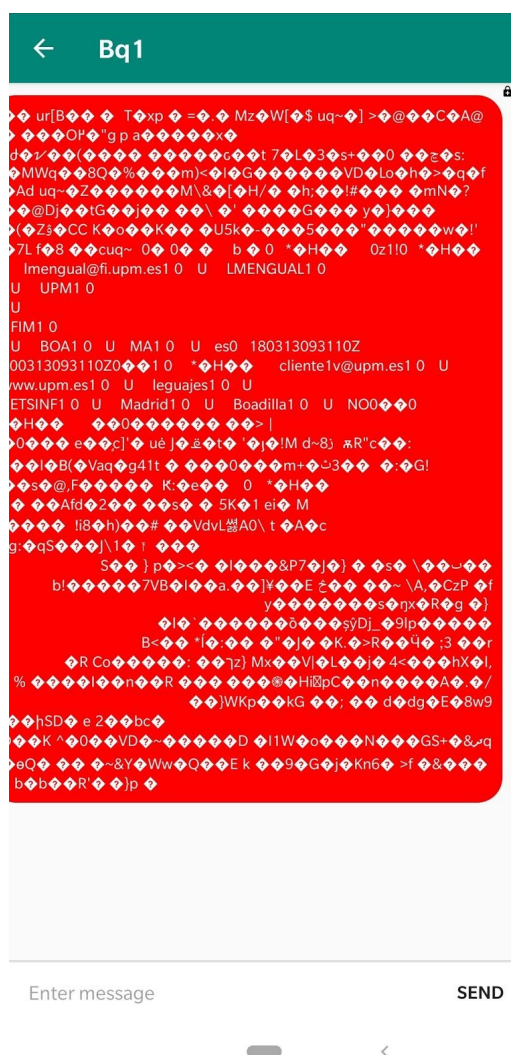
Para ello se ha implementado un servicio de seguridad. Se puede definir como un servicio de comunicación que garantiza la seguridad de los sistemas de información y de la información que viaja por las redes de datos y está formado por mecanismos de seguridad. En nuestro caso tenemos dos.

Para asegurar autenticidad, se ha implementado **la lista de amigos**, y es por ello que solo podemos realizar conexiones y enviar información a usuarios con los que previamente nos hemos identificado y registrado en nuestro sistema local. De esta manera, y si la implementación es correcta, no podrás aceptar peticiones de un usuario que no se encuentre en tu lista de amigos, y deberás solicitar una petición de amistad si deseas conectarte a un dispositivo. Podría considerarse como una política de seguridad implantada visualmente.

Para asegurar la integridad de los datos viajando a través de las redes de proximidad se ha implementado un **protocolo de cifrado** que explicaremos a continuación basado en el uso de claves AES, Hash y certificados de clave pública y privada. En un principio se iba a aplicar a cualquier dato saliente de la aplicación, pero al final y como comentamos más atrás, solo lo vamos a implementar en los mensajes que salen desde la pantalla de chat.

## 6.7.1 Integridad de los datos

En la siguiente imagen podemos observar un ejemplo de mensaje cifrado donde no aplicamos el descifrado.



*Ilustración 37: Integridad de los datos*

1. Cogemos el mensaje que introduce el usuario y lo ciframos con una clave AES de 128 bits.
2. Esta clave AES se cifra con la clave pública del certificado de la aplicación. Teóricamente debería ser la clave pública del certificado del dispositivo con el que nos conectamos pero nos hemos quedado sin tiempo.

## Trabajo de fin de máster

3. Hacemos un Hash del mensaje cifrado con la clave privada de quien envía el mensaje.
4. Además enviaremos el certificado de quien envía el mensaje. En este caso no es necesario al usar el mismo en ambos.

Una vez llega el mensaje al receptor se sigue el proceso inverso :

1. Recuperamos el mensaje cifrado, la clave AES cifrada con la clave pública de B, el hash del mensaje cifrado y el certificado de firma.
2. Extraemos la clave AES descifrando el segundo bloque con la clave privada del usuario sacada del fichero.
3. Desciframos el mensaje cifrado.
- 4 Comprobamos la firma utilizando la clave pública del certificado del 4 bloque.

## 7. Conclusiones

A lo largo del trabajo se han cumplido, no sin gran esfuerzo, los objetivos del trabajo. Pero más allá de eso hemos cumplido un objetivo personal que era el adentrarnos en el mundo del desarrollo de aplicaciones en Android y confío en extender esta base y desarrollar aplicaciones más complejas en un futuro no muy lejano.

Con respecto al planteamiento del trabajo, sabíamos de antemano que iba a ser complicado llegar a desarrollar todo lo que nos habíamos propuesto. Funcionalmente si hemos llegado a los objetivos pero a nivel visual nos hemos quedado cortos de tiempo para realizar un lavado a las diferentes interfaces, y más ahora que ha salido la nueva librería de Material Design de Google donde se unifican un poco todos los conceptos y patrones a seguir en el diseño de pantallas en sistemas móviles y que me habría encantando explorar.

En el apartado del cifrado, y como hemos comentado, ha sido una lástima no poder ofrecer certificados individuales a cada usuario cuando inicias la aplicación. La idea era que cada uno con su propio certificado, fuese intercambiado con el usuario receptor en el momento de añadirse como amigos, y a partir de ahí establecer conexiones seguras. Es más, en la aplicación está implementado el envío de múltiples archivos más allá de la imagen de perfil. Pero otra vez nos quedamos sin tiempo, pues esto requiere un servidor generando y ofreciendo certificados firmados por una autoridad de certificación válida lo cual nos quitaba bastante tiempo y no hemos

llegado a dicho objetivo. Además, para esto se requería el uso de internet lo cual habría complicado un poco el objetivo del proyecto.

En conclusión estoy muy satisfecho con el trabajo realizado, a pesar de la cantidad de baches encontrados. Lo más probable es que desarrolle algo en un futuro no muy lejano para android por pura satisfacción personal, y este trabajo me ha ayudado muchísimo a entender cómo funcionan las bases de Android aún sabiendo que apenas he rascado una parte de sus posibilidades y que, además, con cada nueva versión de sistema operativo cambian muchísimas cosas.





# Bibliografía

1. "Android, el sistema operativo." [https://www.android.com/intl/es\\_es/](https://www.android.com/intl/es_es/).
2. "Especificación de requisitos de software - Wikipedia, la enciclopedia ...." Se consultó el junio 3, 2019.  
[https://es.wikipedia.org/wiki/Especificaci%C3%B3n\\_de\\_requisitos\\_de\\_software](https://es.wikipedia.org/wiki/Especificaci%C3%B3n_de_requisitos_de_software).
3. "3. Técnicas para Identificar Requisitos Funcionales y No Funcionales ...." Se consultó el junio 3, 2019.  
<https://sites.google.com/site/metodologiareq/capitulo-ii/tecnicas-para-identificar-requisitos-funcionales-y-no-funcionales>.
4. "Técnicas para Identificar Requisitos Funcionales y No Funcionales ...." Se consultó el junio 3, 2019.  
<https://sites.google.com/site/metodologiareq/capitulo-ii/tecnicas-para-identificar-requisitos-funcionales-y-no-funcionales>.
5. "Nearby | Google Developers." <https://developers.google.com/nearby/>. Se consultó el 4 jun.. 2019.
6. "Bluetooth de baja energía - Wikipedia, la enciclopedia libre."  
[https://es.wikipedia.org/wiki/Bluetooth\\_de\\_baja\\_energ%C3%ADa](https://es.wikipedia.org/wiki/Bluetooth_de_baja_energ%C3%ADa). Se consultó el 4 jun.. 2019.

7. "Permissions overview | Android Developers."  
<https://developer.android.com/guide/topics/permissions/overview>. Se consultó el 4 jun.. 2019.
8. "Strategies | Nearby Connections API | Google Developers." 11 abr.. 2019,  
<https://developers.google.com/nearby/connections/strategies>. Se consultó el 4 jun.. 2019.
9. "How to retrieve an Unique ID to identify Android devices ? - Medium." 9 feb.. 2017,  
<https://medium.com/@ssaurel/how-to-retrieve-an-unique-id-to-identify-android-devices-6f99fd5369eb>. Se consultó el 7 jun.. 2019.
10. Modelo–vista–controlador - Wikipedia, la enciclopedia libre."  
<https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador>.  
Se consultó el 6 jun.. 2019.
11. "MVP and MVC in Android - part 1 » - TechYourChance." 22 abr.. 2018,  
<https://www.techyourchance.com/mvp-mvc-android-1/>. Se consultó el 6 jun.. 2019.
12. "Android Architecture: Part 10, The Activity Revisited - Android Developer." 9 jul.. 2012,  
<http://www.therealjoshua.com/2012/07/android-architecture-part-10-the-activity-revisited/>. Se consultó el 6 jun.. 2019.
13. "Manifiesto de la app | Android Developers."  
<https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419>.  
Se consultó el 12 jun.. 2019.
14. "Parcelable | Android Developers."  
<https://developer.android.com/reference/android/os/Parcelable>. Se consultó el 7 jun.. 2019.
15. "how to persist an object in android? - Stack Overflow." 31 mar.. 2014,  
<https://stackoverflow.com/questions/22755655/how-to-persist-an-object-in-android>. Se consultó el 7 jun.. 2019.
16. "Settings Guide - Settings | Android Developers."  
<https://developer.android.com/guide/topics/ui/settings>. Se consultó el 10 jun.. 2019.

# Anexo

Aquí podemos encontrar las clases más importantes de la aplicación.

## Clase de servicios

```
public class MyNearbyConnectionService {

    private static MyNearbyConnectionService instance;
    private String name = "";

    private static final String TAG = "APP_SHARE_FILE";
    private String SERVICE_ID = "com.example.jorge.androidapp";
    private ConnectionsClient mConnectionsClient;
    private Context context;

    private final Map<String, Endpoint> mDiscoveredEndpoints = new HashMap<>();
    private final Map<String, Endpoint> mPendingConnections = new HashMap<>();
    private final Map<String, Endpoint> mEstablishedConnections = new HashMap<>();
    private final PetitionManager manager;
    private boolean mIsConnecting = false;
    private boolean mIsDiscovering = false;
    private boolean mIsAdvertising = false;
    private Strategy currentStrategy = Strategy.P2P_CLUSTER;

    public MyNearbyConnectionService(Context context) {
        this.context = context;
        this.mConnectionsClient = Nearby.getConnectionsClient(context);
        this.manager = new PetitionManager(this);
        Timber.tag(TAG);
    }
}
```

```
}

public static MyNearbyConnectionService getInstance(Context context) {
    if (instance == null) {
        synchronized (MyNearbyConnectionService.class) {
            if (instance == null) {
                instance = new MyNearbyConnectionService(context);
            }
        }
    }
    return instance;
}

private MyNearbyAdvertisingListener mNearbyAdvertisingListener;
private MyNearbyDiscoveringListener mNearbyDiscoveringListener;
private MyNearbyEndpointListener mNearbyEndpointListener;
private MyNearbyConnectionListener mNearbyConnectionListener;
private MyNearbyPetitionListener mNearbyPetitionListener;

public void setMyNearbyAdvertisingListener(MyNearbyAdvertisingListener
myNearbyAdvertisingListener) {
    mNearbyAdvertisingListener = myNearbyAdvertisingListener;
}

public void setMyNearbyDiscoveringListener(MyNearbyDiscoveringListener
myNearbyDiscoveringListener) {
    mNearbyDiscoveringListener = myNearbyDiscoveringListener;
}

public void setMyNearbyEndpointListener(MyNearbyEndpointListener
mNearbyEndpointListener) {
    this.mNearbyEndpointListener = mNearbyEndpointListener;
}

public void setMyNearbyConnectionListener(MyNearbyConnectionListener
mNearbyConnectionListener) {
    this.mNearbyConnectionListener = mNearbyConnectionListener;
}

public void setMyNearbyPetitionListener(MyNearbyPetitionListener
mNearbyPetitionListener) {
    this.mNearbyPetitionListener = mNearbyPetitionListener;
}

public MyNearbyPetitionListener getmNearbyPetitionListener() {
    return mNearbyPetitionListener;
}
```

```

    }

    private final ConnectionLifecycleCallback mConnectionLifecycleCallback =
        new ConnectionLifecycleCallback() {
            @Override
            public void onConnectionInitiated(@NonNull String endpointId,
            ConnectionInfo connectionInfo) {
                logD(String.format("onConnectionInitiated(endpointId=%s,
            endpointName=%s)",
                                endpointId, connectionInfo.getEndpointName()));
                Endpoint endpoint = new Endpoint(endpointId,
            connectionInfo.getEndpointName());
                mPendingConnections.put(endpointId, endpoint);
                mNearbyConnectionListener.onConnectionInitiated(endpoint,
            connectionInfo);
            }

            @Override
            public void onConnectionResult(String endpointId,
            ConnectionResolution result) {
                logD(String.format("onConnectionResponse(endpointId=%s,
            result=%s)", endpointId, result));

                // We're no longer connecting
                mIsConnecting = false;
                if (!result.getStatus().isSuccess()) {
                    logW(
                        String.format(
                            "Connection failed. Received status %s.",
                            MyNearbyConnectionService.toString(result.getStatus())));

                    mNearbyConnectionListener.onConnectionFailed(mPendingConnections.remove(endpointId)
                    , null);

                    return;
                }

                connectedToEndpoint(mPendingConnections.remove(endpointId));
            }

            @Override
            public void onDisconnected(String endpointId) {
                if (!mEstablishedConnections.containsKey(endpointId)) {
                    logW("Unexpected disconnection from endpoint " +
            endpointId);

                    manager.onEndpointDisconnected(endpointId);
                    return;
                }
            }
        }
    }

```

```

        }

        disconnectedFromEndpoint(mEstablishedConnections.get(endpointId));

    }

};

/**
 * Callbacks for payloads (bytes of data) sent from another device to us.
 */
private final PayloadCallback mPayloadCallback =
    new PayloadCallback() {
        @Override
        public void onPayloadReceived(@NonNull String endpointId, Payload
payload) {
            if (mEstablishedConnections.get(endpointId) != null)
                manager.addPayload(mEstablishedConnections.get(endpointId),
payload);
        }

        @Override
        public void onPayloadTransferUpdate(@NonNull String endpointId,
PayloadTransferUpdate update) {
            if (mEstablishedConnections.get(endpointId) != null) {
                manager.addPayloadUpdate(endpointId, update);
            } else {
                manager.addOutgoingPayloadUpdate(endpointId, update);
            }
        }
    };

public void startAdvertising() {
    mIsAdvertising = true;
    final String localEndpointName = getName();

    AdvertisingOptions.Builder advertisingOptions = new
AdvertisingOptions.Builder();
    advertisingOptions.setStrategy(getCurrentStrategy());

    mConnectionsClient
        .startAdvertising(
            localEndpointName,
            getServiceId(),
            mConnectionLifecycleCallback,
            advertisingOptions.build())
        .addOnSuccessListener(

```

```

        new OnSuccessListener<Void>() {
            @Override
            public void onSuccess(Void unusedResult) {
                logV("Now advertising endpoint " +
localEndpointName + getCurrentStrategy().toString());
                mNearbyAdvertisingListener.onAdvertisingStarted();
            }
        })
        .addOnFailureListener(
            new OnFailureListener() {
                @Override
                public void onFailure(@NonNull Exception e) {
                    mIsAdvertising = false;
                    logW("startAdvertising() failed.", e);
                    mNearbyAdvertisingListener.onAdvertisingFailed();
                }
            }
        ));
    }

    public void stopAdvertising() {
        logV("Stop advertising");
        mIsAdvertising = false;
        mConnectionsClient.stopAdvertising();
        mNearbyAdvertisingListener.onAdvertisingStopped();
    }

    public boolean isAdvertising() {
        return mIsAdvertising;
    }

    public void acceptConnection(final Endpoint endpoint) {
        mConnectionsClient
            .acceptConnection(endpoint.getId(), mPayloadCallback)
            .addOnFailureListener(
                new OnFailureListener() {
                    @Override
                    public void onFailure(@NonNull Exception e) {
                        logW("acceptConnection() failed.", e);
                    }
                }
            ));
    }

    public void rejectConnection(Endpoint endpoint) {
        mConnectionsClient
            .rejectConnection(endpoint.getId())
            .addOnFailureListener(
                new OnFailureListener() {
                    @Override

```



```

        public void onFailure(@NonNull Exception e) {
            logW("rejectConnection() failed.", e);
        }
    });
}

public void startDiscovering() {
    mIsDiscovering = true;
    mDiscoveredEndpoints.clear();
    logV("Now Discovering endpoints");
    DiscoveryOptions.Builder discoveryOptions = new DiscoveryOptions.Builder();
    discoveryOptions.setStrategy(getCurrentStrategy());
    mConnectionsClient
        .startDiscovery(
            getServiceId(),
            new EndpointDiscoveryCallback() {
                @Override
                public void onEndpointFound(String endpointId,
DiscoveredEndpointInfo info) {
                    logD(
                        String.format(
                            "onEndpointFound(endpointId=%s,
serviceId=%s, endpointName=%s)",
                            endpointId, info.getServiceId(),
info.getEndpointName()));

                    if (getServiceId().equals(info.getServiceId())) {
                        Endpoint endpoint =
mDiscoveredEndpoints.get(endpointId);
                        if (endpoint == null) {
                            Endpoint discovered = new
Endpoint(endpointId, info.getEndpointName());
                            mDiscoveredEndpoints.put(endpointId,
discovered);

mNearbyEndpointListener.onEndpointDiscovered(discovered);
                        }
                    }
                }
            }

            @Override
            public void onEndpointLost(String endpointId) {
                logD(String.format("onEndpointLost(endpointId=%s)",
endpointId));

                Endpoint endpointLost =
mDiscoveredEndpoints.get(endpointId);
                manager.onEndpointLost(endpointLost);
                mDiscoveredEndpoints.remove(endpointId);
            }
        }
    );
}

```

```

mNearbyEndpointListener.onEndpointLost(endpointLost);

        }
    },
    discoveryOptions.build())
    .addOnSuccessListener(
        new OnSuccessListener<Void>() {
            @Override
            public void onSuccess(Void unusedResult) {
                mNearbyDiscoveringListener.onDiscoveryStarted();
            }
        })
    .addOnFailureListener(
        new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
                mIsDiscovering = false;
                logW("startDiscovering() failed.", e);
                mNearbyDiscoveringListener.onDiscoveryFailed();
            }
        });
    });

}

public void stopDiscovering() {
    mIsDiscovering = false;
    mConnectionsClient.stopDiscovery();
    mNearbyDiscoveringListener.onDiscoveryStopped();
}

public boolean isDiscovering() {
    return mIsDiscovering;
}

public void disconnect(Endpoint endpoint) {
    if (endpoint != null && mEstablishedConnections.get(endpoint.getId()) !=
null) {
        mConnectionsClient.disconnectFromEndpoint(endpoint.getId());
        mEstablishedConnections.remove(endpoint.getId());
        mNearbyEndpointListener.disconnect(endpoint);
    } else {
        logV("SE INTENTA DESCONECTAR Y NO EXISTE");
    }
}

public void disconnectFromAllEndpoints() {
    for (Endpoint endpoint : mEstablishedConnections.values()) {
        mConnectionsClient.disconnectFromEndpoint(endpoint.getId());
    }
    mEstablishedConnections.clear();
}

```

```

    }

    public void stopAllEndpoints() {
        mIsAdvertising = false;
        mIsDiscovering = false;
        mIsConnecting = false;
        mDiscoveredEndpoints.clear();
        mPendingConnections.clear();
        mEstablishedConnections.clear();
    }

    public void connectToEndpoint(final Endpoint endpoint) {
        logV("Sending a connection request to endpoint " + endpoint);
        mIsConnecting = true;
        logV("Name of the endpoint " + getName() + "&" + endpoint.getType());
        mConnectionsClient
            .requestConnection(getName() + "&" + endpoint.getType(),
                endpoint.getId(), mConnectionLifecycleCallback)
            .addOnFailureListener(
                new OnFailureListener() {
                    @Override
                    public void onFailure(@NonNull Exception e) {
                        logW("requestConnection() failed.", e);
                        mIsConnecting = false;
                        ApiException apiException = ((ApiException) e);
                        mNearbyConnectionListener.onConnectionFailed(endpoint, apiException);
                    }
                });
    }

    public final boolean isConnecting() {
        return mIsConnecting;
    }

    public void connectedToEndpoint(Endpoint endpoint) {
        logD(String.format("connectedToEndpoint(endpoint=%s)", endpoint));
        mEstablishedConnections.put(endpoint.getId(), endpoint);
        mNearbyEndpointListener.onEndpointConnected(endpoint);
    }

    public void disconnectedFromEndpoint(Endpoint endpointDisconnected) {
        logD(String.format("disconnectedFromEndpoint(endpoint=%s)",
            endpointDisconnected));
        mEstablishedConnections.remove(endpointDisconnected.getId());
        mNearbyEndpointListener.onEndpointDisconnected(endpointDisconnected);
        manager.onEndpointLost(endpointDisconnected);
    }
}

```

```

public Set<Endpoint> getDiscoveredEndpoints() {
    return new HashSet<>(mDiscoveredEndpoints.values());
}

public Set<Endpoint> getConnectedEndpoints() {
    return new HashSet<>(mEstablishedConnections.values());
}

public void send(Payload payload) {
    List<String> keys = new ArrayList<>(mEstablishedConnections.keySet());
    sendMultiple(payload, keys);
}

public void sendOne(Payload payload, String endPointID) {

    mConnectionsClient
        .sendPayload(endPointID, payload)
        .addOnFailureListener(
            new OnFailureListener() {
                @Override
                public void onFailure(@NonNull Exception e) {
                    logW("sendPayload() failed.", e);
                }
            }
        );
}

/**
 * Sends a {@link Payload} to target connected endpoints.
 *
 * @param payload The data you want to send.
 */
public void sendMultiple(Payload payload, List<String> endpoints) {

    mConnectionsClient
        .sendPayload(endpoints, payload)
        .addOnFailureListener(
            new OnFailureListener() {
                @Override
                public void onFailure(@NonNull Exception e) {
                    logW("sendPayload() failed.", e);
                }
            }
        );
}

public static String toString(Status status) {

```

```

        return String.format(
            Locale.ENGLISH,
            "[%d]%",
            status.getStatusCode(),
            status.getStatusMessage() != null
                ? status.getStatusMessage()
                :
            ConnectionsStatusCodes.getStatusCodeString(status.getStatusCode()));
    }

    public static boolean hasPermissions(Context context, String... permissions) {
        for (String permission : permissions) {
            if (ContextCompat.checkSelfPermission(context, permission)
                != PackageManager.PERMISSION_GRANTED) {
                return false;
            }
        }
        return true;
    }

    @CallSuper
    public void logV(String msg) {
        Timber.tag(TAG);
        Timber.v("MyNearbyConnectionsService ____ " + msg);
    }

    @CallSuper
    public void logI(String msg) {
        Timber.tag(TAG);
        Timber.i("MyNearbyConnectionsService ____ " + msg);
    }

    @CallSuper
    public void logD(String msg) {
        Timber.tag(TAG);
        Timber.d("MyNearbyConnectionsService ____ " + msg);
    }

    @CallSuper
    public void logW(String msg) {
        Timber.tag(TAG);
        Timber.w("MyNearbyConnectionsService ____ " + msg);
    }

    @CallSuper
    public void logW(String msg, Throwable e) {

```

```

        Timber.tag(TAG);
        Timber.w(e, "MyNearbyConnectionsService ____ " + msg);
    }

    @CallSuper
    public void logE(String msg, Throwable e) {

        Timber.tag(TAG);
        Timber.e(e, "MyNearbyConnectionsService ____ " + msg);
    }

    public String getName() {
        if (!name.equals("")) {
            return name + "&" + getAndroidId();
        }
        String manufacturer = Build.MANUFACTURER;
        String model = Build.MODEL;
        if (model.startsWith(manufacturer)) {
            return capitalize(model);
        } else {
            return capitalize(manufacturer) + " " + model + "&" + getUser_name() +
"&" + getAndroidId();
        }
    }

    public String getAndroidId() {
        return Settings.Secure.getString(context.getContentResolver(),
            Settings.Secure.ANDROID_ID);
    }

    public String getUser_name() {
        SharedPreferences shared =
PreferenceManager.getDefaultSharedPreferences(context);
        return shared.getString(context.getString(R.string.key_username),
KConstantesShareContent.DEFAULT.DEFAULT_DISPLAY_NAME);
    }

    private String capitalize(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }
        char first = s.charAt(0);
        if (Character.isUpperCase(first)) {
            return s;
        } else {
            return Character.toUpperCase(first) + s.substring(1);
        }
    }
}

```

```

public Strategy getCurrentStrategy() {
    return currentStrategy;
}

public void setCurrentStrategy(Strategy strat) {
    this.currentStrategy = strat;
}

protected String
getServiceId() {
    return SERVICE_ID;
}

public void startPetition(Endpoint endpoint, int type, Object... values) {

    switch (type) {
        case (RequestFriendPetition.REQUEST_FRIEND_TYPE):
            RequestFriendPetition request = new RequestFriendPetition(this,
endpoint);

            request.setConnect(true);
            manager.startPetition(endpoint, request);
            break;
        case (Chatpetition.CHAT_TYPE):
            Chatpetition chatpetition = new Chatpetition(this, endpoint);
            manager.startPetition(endpoint, chatpetition);
            break;
        case (FileExchangePetition.FILE_EXCHANGE_TYPE):
            FileExchangePetition fileExchangePetition = new
FileExchangePetition(this, endpoint);
            fileExchangePetition.setUri((Uri) values[0]);
            manager.startPetition(endpoint, fileExchangePetition);
            break;
        case (MultiFilesExchange.MULTIFILE_EXCHANGE_TYPE):
            MultiFilesExchange multiFilesExchange = new
MultiFilesExchange(this, (ArrayList<Endpoint>) values[0]);
            multiFilesExchange.setUri((Uri) values[1]);
            manager.startPetition(null, multiFilesExchange);
            break;
    }

}

public AbstractPetitionNearby getPetition(String id, int type) {
    return manager.getPetition(id, type);
}

```

```
public void deletePetition(String endpointID, int type) {  
  
    manager.deletePetition(endpointID, type);  
}  
  
}
```

## Clase Manejadora de peticiones

```
public class PetitionManager {  
  
    private HashMap<String, ArrayList<AbstractPetitionNearby>>  
petitions;  
    private MultiFilesExchange multiPetition;  
    private MyNearbyConnectionService service;  
  
    public PetitionManager(MyNearbyConnectionService service) {  
        this.petitions = new HashMap<>();  
        this.service = service;  
    }  
  
    public void startPetition(Endpoint endpoint, @NonNull  
AbstractPetitionNearby petition) {  
        if (endpoint != null) {  
            String endpointId = endpoint.getId();  
            initilizeEndpoint(endpointId);  
            service.logV("START PETITIONN");  
  
            if (petitions.get(endpointId) != null) {  
                petitions.get(endpointId).add(petition);  
                petition.startPetition();  
            }  
        } else {  
            multiPetition = (MultiFilesExchange) petition;  
            multiPetition.startPetition();  
        }  
    }  
  
    private void initilizeEndpoint(String endpointId) {
```



```

        if (petitions.get(endpointId) == null)
            petitions.put(endpointId, new ArrayList<>());
    }

    public void deletePetition(String endpointId, int type) {
        if (endpointId.equals("0")) {
            this.multiPetition = null;
        } else {
            ArrayList<AbstractPetitionNearby> currentPetitions =
petitions.get(endpointId);
            AbstractPetitionNearby found = null;
            if (currentPetitions != null) {
                for (AbstractPetitionNearby pet : currentPetitions) {
                    if (pet.getType() == type)
                        found = pet;
                }
                if (found != null) {
                    service.logI("ELIMINADA PETICION " +
found.getType());
                    currentPetitions.remove(found);
                }
            }
        }
    }

    public void addPayload(Endpoint endpoint, Payload payload) {
        String endpointId = endpoint.getId();
        long payloadId = payload.getId();
        boolean processed = false;
        payloadId = Utils.getStatusFromPayloadId(payloadId);
        if (String.valueOf(payloadId).startsWith("4")) {
            if (multiPetition == null) {
                multiPetition = new MultiFilesExchange(service,
endpoint);
                multiPetition.addPayload(payload);
            } else {
                multiPetition.addPayload(payload);
            }
        } else {

            ArrayList<AbstractPetitionNearby> currentPetitions =

```

```

        petitions.get(endpointId);
        if (currentPetitions != null) {
            for (AbstractPetitionNearby pet : currentPetitions) {
                if (!processed &&
                    pet.isPayloadFromPetition(payloadId)) {
                    processed = true;
                    pet.addPayload(payload);
                }
            }
        }

        if (!processed) {
            AbstractPetitionNearby petition =
                createPetition(endpoint, payload.asBytes());
            if (petition != null) {
                initializeEndpoint(endpointId);
                petitions.get(endpointId).add(petition);
                petition.addPayload(payload);
            }
        }
    }

}

    public void addPayloadUpdate(String endpointId,
        PayloadTransferUpdate payloadUpdate) {
        long payloadId =
            Utils.getStatusFromPayloadId(payloadUpdate.getPayloadId());
        boolean processed = false;
        if (String.valueOf(payloadId).startsWith("4")) {
            if (multiPetition != null)
                multiPetition.addUpdatePayload(payloadUpdate);
        } else {
            ArrayList<AbstractPetitionNearby> currentPetitions =
                petitions.get(endpointId);
            if (currentPetitions != null) {
                for (AbstractPetitionNearby pet : currentPetitions) {
                    if (!processed &&
                        pet.isPayloadFromPetition(payloadId)) {
                        processed = true;

```

```

        pet.addUpdatePayload(payloadUpdate);
    }
}

}

}

}

}

    public void addOutgoingPayloadUpdate(String endpointId,
PayloadTransferUpdate payloadUpdate) {
    long payloadId = payloadUpdate.getPayloadId();
    boolean processed = false;
    if (String.valueOf(payloadId).startsWith("4")) {
        if (multiPetition != null)
            multiPetition.onOutgoingPayloadUpdate(payloadUpdate);
    } else {
        ArrayList<AbstractPetitionNearby> currentPetitions =
petitions.get(endpointId);
        if (currentPetitions != null) {
            for (AbstractPetitionNearby pet : currentPetitions) {
                if (!processed &&
pet.isPayloadFromPetition(payloadId)) {
                    processed = true;
                    pet.onOutgoingPayloadUpdate(payloadUpdate);
                }
            }
        }
    }
}

}

}

}

    private AbstractPetitionNearby createPetition(Endpoint endpoint,
byte[] bytes) {

        AbstractPetitionNearby pet =
AbstractPetitionNearby.getInstance(bytes, service, endpoint);
        service.logI("CREADA PETICION " + (pet != null ?
pet.getType() : "Fallo"));
        return pet;
    }
}

```

```
public AbstractPetitionNearby getPetition(String endpointId, int
type) {
    if (type == MultiFilesExchange.MULTIFILE_EXCHANGE_TYPE)
        return multiPetition;
    AbstractPetitionNearby result = null;
    ArrayList<AbstractPetitionNearby> currentPetitions =
petitions.get(endpointId);
    if (currentPetitions != null) {
        for (AbstractPetitionNearby pet : currentPetitions) {
            if (pet.getType() == type) {
                result = pet;
            }
        }
    }
    return result;
}

public void onEndpointLost(Endpoint endpoint) {
    ArrayList<AbstractPetitionNearby> currentPetitions =
petitions.get(endpoint.getId());
    if (currentPetitions != null)
        for (AbstractPetitionNearby pet : currentPetitions) {
            pet.onEndpointLost(endpoint);
        }
}

public void onEndpointDisconnected(String endpointID) {
    ArrayList<AbstractPetitionNearby> currentPetitions =
petitions.get(endpointID);
    if (currentPetitions != null)
        for (AbstractPetitionNearby pet : currentPetitions) {
            pet.onEndpointDisconnected(endpointID);
        }
}
}
```

## Clase Base

```
public abstract class BaseActivity extends AppCompatActivity {

    protected int REQUEST_CODE_REQUIRED_PERMISSIONS = 1;

    private Thread.UncaughtExceptionHandler androidDefaultUEH;
    private String TAG = "APP_SHARE_FILE";
    protected DialogManager dialogManager;
    protected boolean isAutoConnect = false;

    private NearbyConnectionsService nService;
    protected MyNearbyConnectionService service;

    protected String CHANNEL_ID = "1";

    protected NotificationManagerCompat notificationManager;

    private static final String[] REQUIRED_PERMISSIONS =
        new String[]{
            Manifest.permission.BLUETOOTH,
            Manifest.permission.BLUETOOTH_ADMIN,
            Manifest.permission.ACCESS_WIFI_STATE,
            Manifest.permission.CHANGE_WIFI_STATE,
            Manifest.permission.ACCESS_COARSE_LOCATION,
            Manifest.permission.READ_EXTERNAL_STORAGE,
            Manifest.permission.WRITE_EXTERNAL_STORAGE
        };

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        createNotificationChannel();
        /*Barra de color blanco*/
        int color = ContextCompat.getColor(this, R.color.white);
        getWindow().setNavigationBarColor(color);
    }
}
```

```

        /*Seteamos el view*/
        setContentView(getLayoutId());
        /*recuperamos el servicio*/
        NearbyApplication appState = ((NearbyApplication)
this.getApplication());
        service = appState.service;
        /*Toolbar*/
        setSupportActionBar(findViewById(R.id.toolbar));
        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            getSupportActionBar().setDisplayShowHomeEnabled(true);
        }

        /*Bindeamos las vistas*/
        ButterKnife.bind(this);

        /*Uncaught Exceptions*/
        androidDefaultUEH =
Thread.getDefaultUncaughtExceptionHandler();
        Thread.setDefaultUncaughtExceptionHandler(handler);

        /*set service parameters*/
        setServicesParameters();
        notificationManager = NotificationManagerCompat.from(this);
        /*Logs*/
        Timber.tag(TAG);
        /*Dialogs */
        this.dialogManager = new DialogManager(this);
    }

    @Override
    protected void onStart() {
        super.onStart();

        String[] privatePermissions = getRequiredPermissions();

        /*Pedimos permisos*/

        if (!hasPermissions(this, privatePermissions)) {
            if (!hasPermissions(this, privatePermissions)) {
                if (Build.VERSION.SDK_INT < 23) {

```

```
        ActivityCompat.requestPermissions(  
            this, privatePermissions,  
REQUEST_CODE_REQUIRED_PERMISSIONS);  
    } else {  
        requestPermissions(privatePermissions,  
REQUEST_CODE_REQUIRED_PERMISSIONS);  
    }  
}  
}  
  
    //Conexiones  
    isAutoConnect = Utils.isAutoConnections(this);  
}  
  
    @Override  
    protected void onStop() {  
        super.onStop();  
    }  
  
    public abstract int getLayoutId();  
  
    public abstract void setServicesParameters();  
  
    private void onServiceConnectedStarted() {  
        setServicesParameters();  
    }  
  
    protected Bundle getBundle() {  
        NearbyApplication appState = ((NearbyApplication)  
this.getApplication());  
        return appState.getBundle();  
    }  
  
    private Context getContext() {  
        return this;  
    }  
  
    private ServiceConnection mConnection = new ServiceConnection() {  
  
        @Override  
        public void onServiceConnected(ComponentName className,  

```

```

        IBinder bindService) {
            NearbyConnectionsService.MyNearbyBinder binder =
(NearbyConnectionsService.MyNearbyBinder) bindService;
            nService = binder.getService();
            service = nService.getConnections();
            onServiceConnectedStarted();
            Utils.createToastShort(getContext(), "Service started");
        }

@Override
public void onServiceDisconnected(ComponentName arg0) {
}

};

protected boolean isMyServiceRunning(Class<?> serviceClass) {
    ActivityManager manager = (ActivityManager)
getSystemService(Context.ACTIVITY_SERVICE);
    for (ActivityManager.RunningServiceInfo service :
manager.getRunningServices(Integer.MAX_VALUE)) {
        if
(serviceClass.getName().equals(service.service.getClassName())) {
            return true;
        }
    }
    return false;
}

private void createNotificationChannel() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        CharSequence name =
ApplicationNotificationManager.DEFAULT;
        String description =
getString(R.string.channel_description);
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel channel = new
NotificationChannel(CHANNEL_ID, name, importance);
        channel.setDescription(description);
        NotificationManager notificationManager =
getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
}
}

```



```
private SharedPreferences getSharedPreferences() {
    return PreferenceManager.getDefaultSharedPreferences(this);
}

@Override
public boolean onSupportNavigateUp() {
    onBackPressed();
    return true;
}

protected void writeToPreferences(String key, String value) {
    SharedPreferences sharedPref = getSharedPreferences();
    SharedPreferences.Editor editor = sharedPref.edit();
    editor.putString(key, value);
    editor.apply();
}

protected String readFromPreferences(String key) {
    SharedPreferences sharedPref = getSharedPreferences();
    return sharedPref.getString(key, "");
}

private Thread.UncaughtExceptionHandler handler = new
Thread.UncaughtExceptionHandler() {
    public void uncaughtException(Thread thread, Throwable ex) {
        Log.e("SharedContentApp", "Uncaught exception is: ", ex);
        androidDefaultUEH.uncaughtException(thread, ex);
    }
};

protected boolean hasPermissions(Context context, String...
permissions) {
    for (String permission : permissions) {
        if (ContextCompat.checkSelfPermission(context,
permission)
            != PackageManager.PERMISSION_GRANTED) {
            return false;
        }
    }
}
```

```

        return true;
    }

    @CallSuper
    @Override
    public void onRequestPermissionsResult(
        int requestCode, @NonNull String[] permissions, @NonNull
int[] grantResults) {
        if (requestCode == REQUEST_CODE_REQUIRED_PERMISSIONS) {
            for (int grantResult : grantResults) {
                if (grantResult == PackageManager.PERMISSION_DENIED)
{
                    Toast.makeText(this,
R.string.error_missing_permissions, Toast.LENGTH_LONG).show();
                    finish();
                    return;
                }
            }
            recreate();
        }
        super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
    }

    public String[] getRequiredPermissions() {
        return REQUIRED_PERMISSIONS;
    }

    @Override
    protected void onResume() {
        super.onResume();

        isAutoConnect = Utils.isAutoConnections(this);
    }

    protected boolean isAppInBackground() {
        NearbyApplication appState = ((NearbyApplication)
this.getApplication());
        return
appState.getAppLifecycleListener().isAppInBackGround();
    }

```

```
protected abstract String getTag();

@CallSuper
public void logV(String msg) {
    Timber.tag(TAG);
    Timber.v(getTag() + msg);
}

@CallSuper
public void logD(String msg) {
    Timber.tag(TAG);
    Timber.d(getTag() + msg);
}

@CallSuper
public void logW(String msg) {
    Timber.tag(TAG);
    Timber.w(getTag() + msg);
}

@CallSuper
public void logI(String msg) {
    Timber.tag(TAG);
    Timber.i(getTag() + msg);
}

@CallSuper
public void logI(String msg, Object... args) {
    Timber.tag(TAG);
    Timber.i(getTag() + msg, args);
}

@CallSuper
public void logW(String msg, Throwable e) {
    Timber.tag(TAG);
    Timber.w(e, getTag() + msg);
}

@CallSuper
public void logE(String msg, Throwable e) {
    Timber.tag(TAG);
    Timber.e(e, getTag() + msg);
}
```

```
}

@CallSuper
public void logI(String msg, Throwable e) {
    Timber.tag(TAG);
    Timber.i(e, getTag() + msg);
}
}
```