ETS de Ingenierías Informática y de Telecomunicación

Android

Estructura y principales componentes – Grupo C

2016/17

Integrantes:

Guillermo Gómez Trenado,
Sonsoles Jiménez Gómez
Juan Salvador Molina Martín,
Miguel Ángel Rispal Martínez,
Juan Carlos Ruiz García

Android

Índice

1. Estructura del sistema

Android es una pieza de software que va más allá de las características usuales de un sistema operativo, de código libre, **basado en Linux no POSIX** y diseñado como una pila donde cada capa se comunica con la anterior hasta llegar al kernel, todo optimizado para dispositivos pequeños con la eficiencia siempre en mente y pensado para controlar desde la forma en que el usuario interactúa con las aplicaciones hasta la integración del hardware. [1]
Analizaremos cada capa, cómo funciona y cuál es su forma de comunicarse. Hay que aclarar, que el Android que los usuarios solemos encontrar en los móviles es un sistema operativo que por lo general incluye partes de software libre y partes de software propietario, nosotros nos limitaremos a analizar lo que se conoce como **AOSP o Android Open Source Project**, sin incluir las modificaciones estructurales o superficiales de los vendedores así como las piezas de software propietario de Google que son sin embargo parte esencial de la experiencia de usuario Android (Google Mobile Services y Google Apps). [2] [3]

1.1. PCI

Antes de nada, definiremos las **tres formas de comunicación entre procesos** que encontramos en un sistema Android, y son las únicas, pues Android **no soporta System V IPC**[4] definidas en el estándar POSIX, por dos motivos, seguridad y eficiencia. Estas tres son Binder, la interfaz nativa de Java y el uso de llamadas al sistema.

1.1.1. Binder

Basado en **OpenBinder**, éste es un componente de sistemas para el desarrollo de servicios en sistemas operativos orientado a objetos, y es el **sustituto** del resto de herramientas de comunicación entre procesos (IPC) como **semáforos, memoria compartida y paso de mensajes**. La API Android tiene **dos formas de comunicación entre procesos** que abstraen la implementación de Binder en el sistema Android, **paso de mensajes a través de *Intent*** para la comunicación asíncrona y ***ContentResolvers/ContentProviders para la comunicación síncrona***. Utiliza un *Binder driver* parte del kernel y oculta la información a los procesos intervinientes, siguiendo la filosofía de Android -*it just works*- el funcionamiento es transparente a los desarrolladores. [5]

1.1.2. JNI (Java Native Interface)

Es una **interfaz nativa** de programación que permite a los programas ejecutados dentro de una máquina virtual de Java **interactuar con el código nativo** (escrito en C/C++), así como cargar librerías compartidas enlazadas dinámicamente. Todo esto se produce de forma relativamente eficiente y segura. [6] [7]

1.1.3. Llamadas al sistema

Es la forma usual de **permitir a los procesos interactuar con el kernel** del sistema operativo, como ocurre en la mayoría de sistemas operativos, incluidos los sistemas Unix y Unix-like. Esta forma de comunicación está reservada para el código nativo que analizaremos más adelante.

1. (Android[1], 2016)
2. (Android[2], 2016)
3. (Android[3], 2016)
4. (Android[4], 2016)
5. (Gargenta, 2013)
6. (Android[5], 2016)
7. (Oracle, 2016)

1.2. Arquitectura de la plataforma

1.2.1. Aplicaciones del sistema

Son las **piezas de código prescindibles** que se ejecutan por encima del sistema operativo, **escritas en Java** y distribuidas en **paquetes APK** que incluyen los recursos, el manifiesto de la aplicación y un archivo .dex, contenedor de las definiciones de clases (equivalente a los archivos .class). Estas aplicaciones **se ejecutan en el entorno de ejecución ART sustituto de las antiguas máquinas virtuales Dalvik** que acompañaron a Android hasta 2013 con Android KitKat.
Las aplicaciones se comunican entre sí y con la capa inferior a través de las abstracciones de la **API** basadas en Binder.

1.2.2. Java API Framework

Compartida tanto por las aplicaciones de usuario como las del sistema**, incluyen mecanismos para interactuar con el sistema de forma eficiente y cómoda para los desarrolladores**, incluyen entre otras el sistema de vistas de la interfaz de usuario, el gestor de recursos, gestor de notificaciones, proveedor de contenidos (que permite la comunicación entre aplicaciones), servicios de batería y wifi, y el gestor de actividades, que es la pieza clave del ciclo de vida de las aplicaciones en Android y que aunque por cuestiones de espacio no describimos aquí destacamos sólo que las aplicaciones en Android se definen como *Actividades* con una finalidad muy concreta y ligera que se van llamando unas a otras y pasando información, cada actividad con distintas etapas en su ciclo de vida así como con distintas implicaciones especialmente a nivel de uso de recursos y de estadios de carga. [8]
La API se comunica entre ella también mediante el uso de Binder, y con la capa inferior a través de JNI.

1.2.3. Librerías nativas de C/C++

Esto son los **cimientos de la mayoría de funcionalidades que se encuentran en la API Java**, la capa de abstracción sobre el hardware que veremos a continuación (**HAL**) se basa en estas librerías nativas, así como el entorno de ejecución ART. Aquí encontramos por ejemplo la librería de renderizado web y el motor JavaScript Webkit, la implementación de OpenGL para Android y otras librerías de reproducción de medios. También se pueden utilizar algunas de estas librerías directamente introduciendo porciones de código en C/C++ en las aplicaciones Android a través del uso de la palabra clave *native* en la declaración de los métodos. El método de comunicación es Binder, y otros recursos usuales en sistemas Unix-like como son el uso de cauces con nombre y comunicación por sockets. [9]

1.2.4. Android Runtime

Antiguamente la forma en la que **se ejecutaban las aplicaciones era a través del uso de Dalvik Virtual Machine (Dalvik VM)**, que **iba compilando pequeñas porciones de código en tiempo de ejecución** y optimizándolas a medida que se ejecutaba y analizaba su comportamiento. Es en definitiva una reformulación de la JVM optimizada para dispositivos poco potentes enfocado al uso de memoria y batería. **Progresivamente se cambió éste modelo por el uso de Android Run-time (ART)** que **compila en el momento de instalación las aplicaciones (AOT compilation)** y produce un **archivo ELF** que es ejecutado posteriormente, esto ha sido posible en parte por la mejora de la capacidad de los procesadores móviles, que permiten compilaciones de programas completos en tiempos razonables, así pues, aunque repercute negativamente en la espera el momento de la instalación, la mejora en la ejecución es evidente así como el ahorro en el uso de la batería. [10]
ART funciona como cualquier compilador y enlazador optimizado para estos dispositivos.

8. (Android[6], 2016)
9. (Android[7], 2016)
10. (Android[8], 2016)

1.2.5. Capa de abstracción de hardware (HAL)

La **HA**L define un **estándar de comunicación entre Android y las características específicas de cada hardware**, y permite a los servicios de Android ser agnóstico sobre el uso del hardware, delegando la responsabilidad en los fabricantes que integran nuevos componentes. La mayoría de sistemas Unix-Like utilizaban una tecnología similar del mismo nombre, pero migraron hacia UDEV entre 2008 y 2010 cuando el proyecto fue descontinuado. Al final la lógica es siempre la misma, introducir una capa extra entre la capa hardware y la capa nativa, abstrayendo la lógica y dejando el funcionamiento del hardware transparente a los desarrolladores.
Los módulos son **librerías enlazadas dinámicamente** (.so) con su pertinente *makefile* y son cargados por el sistema operativo sólo cuando son necesarios liberando así la memoria, funcionamiento similar a los módulos de carga dinámica Linux. Son desarrollados en C. [11]

1.3. Linux Kernel

Android carga el kernel de Linux como pilar del sistema siendo **versiones modificadas** de los mismos adaptadas a dispositivos móviles. Las adaptaciones normalmente consisten en eliminación de características prescindibles o que suponen un **riesgo para la seguridad** [12], o en optimizaciones para el uso de memoria y batería. La primera versión integrada fue el Linux Kernel 2.6 con Android Cupcake en 2009 y actualmente, en Android Nougat utilizan la versión 4.4.1. [13]

En definitiva **Android se configura como un sistema Linux restringido** en su potencial para ofrecer una plataforma segura y fiable, permitiendo garantizar un **uso homogéneo** del sistema en un entorno donde el desarrollo de aplicaciones es libre, impedir en la medida de lo posible usos irresponsables de la memoria o el procesador (repercutiendo en la duración de la batería) y crear un contexto cómodo de desarrollo donde se libera al programador de preocupaciones relativas al SO a cambio de sacrificar cierta libertad en pro de un entorno micro-gestionado y relativamente dogmático.

2.Procesos del sistema

De manera predeterminada, todos los componentes de una misma aplicación se ejecutan en el **mismo proceso** y la mayoría de las aplicaciones **no deben** cambiar esto.
Android puede decidir finalizar un proceso en algún momento, cuando la memoria es baja y la requieren otros procesos que son inmediatamente más necesarios para lo que el usuario desea. Cada aplicación tiene su **propio proceso** y su propia **VM** (*Virtual machine*). [14]
Las reglas que se usan para decidir qué procesos finalizar se analizan a continuación.

2.1. Ciclo de vida de los procesos

El sistema Android trata de mantener el proceso de una aplicación el mayor tiempo posible, pero, finalmente, necesita quitar los procesos viejos a fin de recuperar memoria para procesos nuevos o más importantes. Para determinar que procesos mantener y cuales finalizar, el sistema utiliza una **jerarquía de importancia**. Los procesos con la importancia más baja se eliminan primero, luego, los siguientes con la importancia más baja y así sucesivamente. La jerarquía de importancia tiene cinco niveles. [15]

11. (Android[9], 2016)
12. (Gargenta, 2013)
13. (Android[10], 2016)
14. (Android[11], 2016), Processes
15. (Android[11], 2016), Lifecycle

- ***Proceso en primer plano.***
  Es un proceso que es necesario para lo que el usuario está haciendo en ese momento. Por lo general existen unos pocos procesos en primer plano en un momento determinado. **Se cancela solamente como último recurso** cuando la memoria está tan lenta que no todos pueden seguir ejecutándose.

- ***Proceso visible.***
  Es un proceso que no tiene **ningún** componente en primer plano, pero que puede afectar lo que el usuario ve en pantalla. Un proceso visible se considera **extremadamente importante** y no se finalizará a menos que sea necesario para mantener todos los procesos en primer plano.

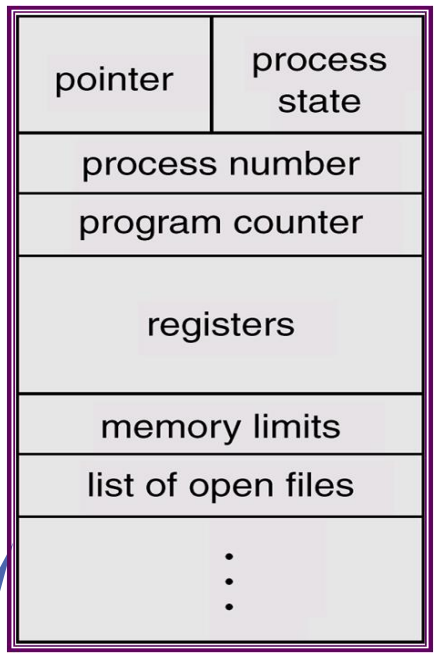
- ***Proceso de servicio.***
  Son procesos ejecutados por un servicio iniciado con el método *startService().* Aunque los procesos de servicio no están vinculados directamente con nada de lo que el usuario ve, generalmente **hacen cosas que le interesan al usuari**o (reproducir música en segundo plano, descargar datos de red) por lo que el sistema los mantiene en ejecución a menos que no haya suficiente memoria.

- ***Procesos de segundo plano.***
  Son procesos que tienen una actividad que actualmente **no es visible** para el usuario. Estos procesos no tienen un efecto directo en la experiencia del usuario, y el sistema puede finalizarlos en cualquier momento para recuperar memoria para un proceso en *primer plano, visible o de servicio*. Por lo general se los mantiene en una **lista LRU** (*Less Recent Used o Menos usado recientemente*) para garantizar que el proceso que el usuario vio más recientemente sea el último en finalizarse.

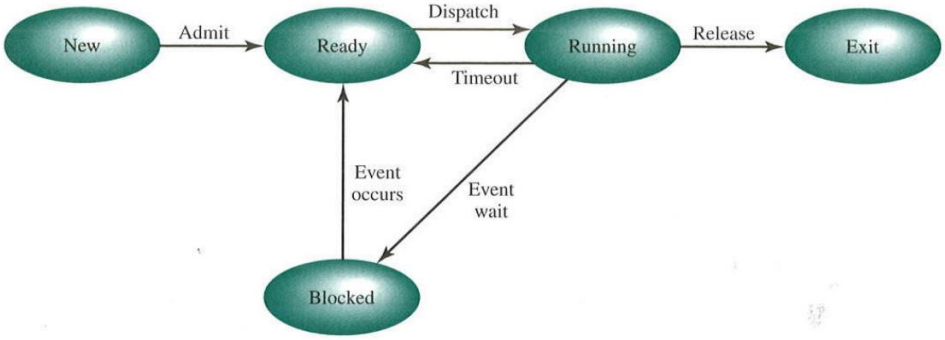
- ***Procesos vacíos.***
  Son procesos que **no tienen ningún componente de la aplicación activo**. El único motivo para mantener este tipo de procesos activos es por fines de la memoria caché.

2.2. Gestión de procesos

La gestión de procesos es **una operación típica del sistema**, la cual envuelve complejas estructuras y algoritmos de datos, pero no va mucho más allá del nivel de gestión de la estructura típica de procesos de datos. [16]
Android tiene una base de nivel de control de estructuras parecido a la siguiente imagen.



Esta estructura de datos es gestionada mediante un **estándar de gestión de procesos similar al de Linux.**



16. (Colt's Blog, 2015)

2.3. Comunicación entre procesos

Android ofrece un mecanismo para comunicación entre procesos (IPC) con llamadas de procedimiento remoto (RPC), en el que una actividad u otro componente de la aplicación llama a un método y los resultados se devuelven al emisor. [17]

3. Gestión de memoria

Android es un Sistema operativo **basado en Linux**, que maneja la mayoría de tareas. Utiliza bibliotecas nativas en C abiertas. Todas las operaciones básicas del sistema operativo como de E/S, gestión de memoria, gestión de dispositivos, etc. son manejados por el de **kernel de Linux**. [18]

3.1. ¿Cómo usa la memoria Android para cada aplicación?

Al igual que Java y .NET, **Android utiliza el entorno de ejecución** y la **máquina virtual** para gestionar la memoria de la aplicación. A diferencia de los dos citados anteriormente, el entorno de ejecución de Android **también maneja los tiempos de vida del proceso**.

Android asegura que la respuesta de la aplicación sea **correct**a, deteniendo y matando a los procesos que obstaculizan la fluidez si fuera necesario y libera recursos para las aplicaciones con mayor prioridad.

3.1.1. La máquina virtual de Android

Dalvik Virtual Machine es una máquina virtual basada en registros que ha sido optimizada para asegurar que un dispositivo puede *ejecutar múltiples instancias* de manera eficiente. Se basa en el kernel de Linux para la gestión de memoria de bajo nivel.

Uno de los elementos clave de Android es la **máquina virtual de Dalvik**. En lugar de utilizar una tradicional máquina virtual Java (VM), tales como Java ME (Java Mobile Edition*),* **Android utiliza su propia máquina *virtual*** diseñada para asegurar que la multitarea se ejecuta de manera eficiente en un único dispositivo.

3.1.2. ¿Cómo funciona Dalvik?

Cada aplicación Android **se ejecuta en un proceso independiente dentro de su propia instancia de Dalvik**, renunciando a toda responsabilidad de la memoria y la gestión de procesos. [19]

Todo el hardware de Android y acceso a los servicios del sistema se gestiona mediante Dalvik como un nivel intermedio. Mediante el uso de una máquina virtual para organizar la ejecución de aplicaciones, los desarrolladores tienen una capa de abstracción para asegurarse de que nunca tendrán que preocuparse de una aplicación de hardware en particular. [20]

3.2. Low Memory Killer Levels

Cuando una **aplicación** que se está corriendo en primer plano requiere más **memoria para la ejecución de un** proceso, el sistema cierra las aplicaciones que se ejecutan en **segundo plano para evitar el uso de la memoria** secundaria, que por lo general es escasa en los dispositivos móviles. [21]

17. (Android[11], 2016), IPC
18. (Riquelme, 2012)
19. (ReadTheDocs, 2012)
20. (Riquelme, 2012)
21. (About Things, 2015)

4. Sistema de Archivos

Los **objetos File** se usan para leer o escribir enormes cantidades de datos de principio a fin sin omisiones. En Android, el **sistema de archivos tiene cierto parecido con los sistemas de archivos basados en discos**, es decir, diseñado para el almacenar los archivos en una unidad de disco que puede estar conectada directa o indirectamente a la computadora [22], que utilizan otras plataformas.

4.1. Almacenamiento interno vs almacenamiento externo

Android dispone de dos áreas para el almacenamiento.

Estos nombres vienen dados porque **al crearse Android** la mayoría de dispositivos disponían de una **memoria no volátil** integrada (no necesita energía [23]) que servía como memoria interna y un medio de almacenamiento extraíble como memoria externa.

Sin embargo, algunos dispositivos tienen **dividido el espacio** de almacenamiento permanente en **internas y externas**, así que incluso sin un medio de almacenamiento extraíble, sigue habiendo dos espacios de almacenamiento.

- El **almacenamiento interno** se caracteriza por estar siempre disponible; solo las aplicaciones pueden acceder a los archivos guardados; si se desinstala una aplicación el sistema elimina todos los archivos relacionados.
- El **almacenamiento externo**, dado que el usuario puede montarlo como almacenamiento USB e incluso eliminarlo del dispositivo, no está siempre disponible; puede ser leído por cualquier usuario; si de desinstala una aplicación, el sistema solo elimina sus archivos si están guardados en el directorio *getExternalFilesDir()*.

4.2. Archivos públicos y privados

A su vez, en el **almacenamiento externo existen dos categorías** de archivos, **archivos públicos y privados**.

- Los **archivos públicos** deben estar disponibles para el usuario y aplicaciones. Aunque el usuario desinstale la app, los archivos deben seguir estando disponibles para el usuario. Las fotografías son un ejemplo de este tipo de archivos.
- Los **archivos privados** pertenecen a sus respectivas aplicaciones y deben ser eliminados si se desinstala su app dado que carecen de valor fuera de su aplicación. [24]

Tal y como cualquier otro sistema operativo como Windows, **Android usa aplicaciones distintas para abrir tipos particulares de archivos**.

Un **sistema de archivos** puede tener implementado **restricciones** para ciertas operaciones que se realizan sobre los archivos, como leer, escribir o ejecutar. Los conjuntos de estas restricciones se conocen como '**Permisos de acceso'** *(Access permissions)*.

Las instancias de la clase File son inmutables. Una vez creado, el nombre de ruta abstracto representado por un objeto File no volverá a cambiar.

En Android, **el sistema de archivos usa siempre UTF-8** para codificar los nombres de archivos. [25]

La **jerarquía de archivos** en Android es una **versión modificada de la jerarquía de archivos tradicional de Linux** y solo tiene una raíz. [26]

22. (Wikipedia[1], 2016)
23. (Wikipedia[2], 2016)
24. (Android[12], 2016)
25. (Android[12], 2016)
26. (ATA, 2013)

5. Seguridad del sistema [27]

La seguridad es un aspecto importante en cualquier sistema ya que si descargáramos una aplicación maliciosa podría robar nuestra información personal y ser trasladada a la red.
Por esto, se propone en Android un esquema de seguridad que protege a los usuarios sin imponer un sistema centralizado y sin ser controlado por una única empresa, como es el caso de iOS. [EGLA]

Su seguridad se basa en **tres pilares fundamentales**:

1. Como Android está **basado en Linux**, por lo tanto, se **aprovechará la seguridad que incorpora este sistema operativo**. Una de las principales ventajas es que pueden impedir que las aplicaciones tengan acceso directo al hardware o interfieran con recursos de otras aplicaciones.
2. Todas las aplicaciones deben ser **firmada con un certificado digital** que identifique a su autor. Esto nos garantiza que el fichero de la aplicación no ha sido modificado. Si se modificara la aplicación, debe ser firmada de nuevo y esto solo lo puede hacer el autor.
3. Si se quiere que una aplicación tenga acceso a partes del sistema que compromete la seguridad del sistema, hemos de usar un **modelo de permisos**, de forma que se conozca los riesgos antes de instalar una aplicación.

Para proteger el acceso a recursos utilizados por otras aplicaciones, se crea **una cuenta de usuario Linux nueva por cada paquete de formato .apk** en el sistema. Cualquier dato almacenado por la aplicación será asignado a su usuario Linux, por lo que no tendrá acceso a otras aplicaciones.

5.1. Esquema de permisos en Android[28]

Para proteger ciertos recursos, se sigue una pauta o **esquema de permisos**. Todas las aplicaciones están obligadas a declarar su intención de usarlos. Si una aplicación intentara acceder a un recurso del que no hay solicitado permiso, se produciría una excepción de permiso. A la hora de instalar una aplicación, podremos examinar la lista de permisos que nos solicita la aplicación. Los permisos se pueden **clasificar según nivel de peligro o el tipo de permiso** (ubicación, almacenamiento, etc....). Nombraremos unos de los cientos que hay.

5.1.1. Permisos Marshmallow

En los **dispositivos anteriores a Marshmallow**, los usuarios concedían los **permisos en el momento de la instalación**, en el caso en el que no estemos de acuerdo con ellos no lo instalamos. Pero una vez instalada la aplicación, puede realizar acciones asociadas a esto permisos como desee incluso dejando al usuario indefenso ante posibles abusos. Pero en la **versión 6 o Marshmallow** divide los permisos en peligrosos y normales ya nombrados anteriormente. Ahora en la instalación de una aplicación en los permisos normales se hace como en la versión anterior pero los permisos peligrosos no son concedidos durante la instalación y **la aplicación consultará al usuario si quiere conceder un permiso peligroso en el momento en el que se utiliza**.

5.1.2. Permisos definidos por el programador en Android

Además de los permisos definidos por el sistema, **los desarrolladores vamos a poder crear nuevos permisos** para restringir el acceso a elementos de nuestro software.

Para definir un nuevo permiso utilizaremos el ***tag <permission>*** en el fichero ***AndroidManisfest.xml***

27. (ADSLZone, 2016)
28. (EGLA, 2016)

El atributo ***android:name*** indica el nombre del permiso este debe estar dentro del mismo dominio que nuestra aplicación.
Ejemplo de ello sería:

android:name=".....".
android:description=".....".
android:label=".....".
android:permissionGroup=".....".
android:protectionLevel=".....".

En el atributo **permissionGroup** es opcional y es útil para agrupar las aplicaciones que tienen un coste al usuario. En el atributo android:protectionLevel informa al sistema de cómo debe ser informado el usuario y se dan cuatro casos:

- ✓ **Normal**. El usuario no es advertido cuando se instala la aplicación.
- ✓ **Dangerous**. El usuario es advertido en el proceso de instalación.
- ✓ **Signature**. Solo se da el permiso a aplicaciones firmadas con la misma firma digital que la aplicación que declara el permiso.
- ✓ **SignatureOrSystem**. Igual al Signature, pero además puede ser usado por el sistema.

5.2. Seguridad en la memoria

Esta función **llegó en la versión Android 4.1**. Dado que la mayoría de las vulnerabilidades se explotan a través de poder ejecutarlas en la memoria del sistema operativo.
Se añadirán, en concreto, **dos nuevas opciones de configuración**:

- CONFIG_DEBUG_RODATA
- CONFIG_CPU_SW_DOMAIN_PAN

Las **principales ventajas** son:

- La primera **permitirá a los desarrolladores controlar qué segmentos de la memoria son modificables y ejecutables**. Con esto, los desarrolladores pueden limitar la memoria a la que pueden acceder las aplicaciones, y reducir la memoria de la que disponen los posibles atacantes en el caso de que la aplicación sea maliciosa.
- La segunda opción **limita cuánto puede acceder el kernel al espacio de usuario**. Los *exploits* normalmente toman primero el control de la memoria del espacio del usuario, y esperan a que interactúe con el kernel para acceder a ese espacio de memoria. Con esto, se limita la posibilidad de que código malicioso acabe llegando al kernel, lo cual supondría exponer el móvil al control total de un atacante.

6. Bibliografía

· (Android[1], 2016), Android Interfaces and Architecture. (2017). Android. Accedido el 9 de Enero, 2017, en https://source.android.com/devices

· (Android[2], 2016), Google Mobile Services. (2016). Android. Accedido el 9 de Enero, 2017, en https://www.android.com/gms/

· (Android[3], 2016) Android Source. Frequently Asked Questions. (2017). Android. Accedido el 9 de Enero, 2017, en https://source.android.com/source/faqs.html

· (Android[4], 2016). Documentación oficial sobre la NDK de Android , Accedido el 9 de Enero, 2017, en https://android.googlesource.com/platform/ndk/+/4e159d95ebf23b5f72bb707b0cb1518ef96b3d03/docs/system/libc/SYSV-IPC.TXT

· (Gargenta, 2013). Deep Dive into Android IPC/Binder Framework at Android Builders Summit 2013. Aleksandar Gargenta. Marakana Inc. Accedido el 9 de Enero, 2017, en https://events.linuxfoundation.org/images/stories/slides/abs2013_gargentas.pdf

· (Android[5], 2017). JNI Tips, Android Developers. (2017). Android. Accedido el 9 de Enero, 2017, en https://developer.android.com/training/articles/perf-jni.html

· (Oracle, 2016). Java SE Documentation. (2016). Oracle. Accedido el 9 de Enero, 2017, en http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp725

· (Android[6], 2016). Android Developers. Platform Architecture. (2016) Android. Accedido el 9 de Enero, 2017, en https://developer.android.com/guide/platform/index.html

· (Android[7], 2016). Android Developers, OS. (2016). Android. Accedido el 9 de Enero, 2017, en https://developer.android.com/reference/android/system/Os.html

· (Android[8], 2016). Android Open Source Project, ART and Dalvik. (2016). Android. Accedido el 9 de Enero, 2017, en https://source.android.com/devices/tech/dalvik/index.html

· (Android[9], 2016) Android Open Source Project, Android Interfaces and Architecture. (2016). Android. Accedido el 9 de Enero, 2017, en https://source.android.com/devices/index.html

· (Android[10], 2016) Android Kernel-Headers, Git at Google (2016). Android. Accedido el 9 de Enero, 2017, en https://android.googlesource.com/platform/external/kernel-headers/+refs

· (Android[11], 2016) Android Developers, Processes. (2016). Android. Accedido el 9 de Enero, 2017, en https://developer.android.com/guide/components/processes-and-threads.html

· (Colt's Blog, 2015). Android OS – Process and the Zygote!. (2015). Colt Doe. Accedido el 9 de Enero, 2017, en http://coltf.blogspot.com.es/p/android-os-processes-and-zygote.html

· (Riquelme, 2012). Gestión de memoria en Android, Sozpic. (2012), Alfonso Riquelmen. Obtenido el 9 de Enero, 2017, en https://www.sozpic.com/gestion-de-memoria-en-android/

· (ReadTheDocs, 2012) Android OS, detalles técnicos. (2012). ReadTheDocs. Obtenido el 9 de Enero, 2017, en http://androidos.readthedocs.io/en/latest/data/detalles_tecnicos

· (About Things, 2015). Android OS, Gestión de memoria (2015) About Things. Obtenido el 9 de Enero, 2017, en http://abth.co/articulo/memandroid~Gesti%C3%B3n-de-memoria-en-Android.html

· (Wikipedia[1], 2016). Sistema de archivos. (2016). Wikipedia. Obtenido el 9 de Enero, 2016, en https://es.wikipedia.org/wiki/Sistema_de_archivos

· (Wikipedia[2], 2016). Sistema de archivos. (2016). Wikipedia. Obtenido el 9 de Enero, 2016, en https://es.wikipedia.org/wiki/Memoria_no_vol%C3%A1til

· (Android[12], 2016). Android Developers, Saving Files. (2016). Android. Obtenido el 9 de Enero, 2016, en https://developer.android.com/training/basics/data-storage/files.html

· (Android[13], 2016). Android Developers, File. (2016). Obtenido el 9 de Enero, 2016, en https://developer.android.com/reference/java/io/File.html

· (ATA, 2013). Understanding the Android File Hierarchy. (2013) All things Android. Obtenido el 9 de Enero, 2016 en http://www.all-things-android.com/content/understanding-android-file-hierarchy

·(ADSLZone, 2016). Android 7.0 Nougat contará con las últimas mejoras de seguridad del kernel de Linux. (2016). ADSLZone. Obtenido el 9 de Enero, 2017, en http://www.adslzone.net/2016/07/29/android-7-0-nougat-contara-las-ultimas-mejoras-seguridad-del-kernel-linux/

· (EGLA, 2016). El Gran Libro de Android 5ª (2016). Jesús Tomás Gironés.