UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

I.T. Telecomunicación (Sonido e Imagen)

"Desarrollo de un sistema de control de stock para dispositivos Android"

TRABAJO FINAL DE CARRERA

Autor/es:
Marina Ferreró Ridaura

Director/es:
Dr. Jaime García Rupérez

GANDIA, 2013

A mi familia:
gracias por
vuestra paciencia.

Resumen

Stock it all es una aplicación para la plataforma Android con la que se puede controlar el stock tanto en un negocio como de forma particular, lanzando dicho sistema operativo al mundo empresarial. La aplicación fue diseñada para que funcionara con dispositivos de bajo rendimiento con el fin de reducir el impacto de la inversión, ya que habitualmente los dispositivos profesionales suelen tener precios elevados. Con esta aplicación es posible realizar tareas de gestión de unidades, pero también la consulta de los registros. Además se complementa con un elemento intermedio, un servidor que permite la conexión entre múltiples dispositivos, que a pesar de mantener la base de datos local, mediante sockets se mantendrán continuamente actualizados. En el caso concreto de este proyecto se ha realizado para la gestión de libros.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Con el desarrollo de la tecnología actual y la popularización de la telefonía móvil económica y socialmente, el uso del teléfono móvil con el mero hecho de comunicarse ha quedado relegado a un segundo plano. Con este proyecto se pretende demostrar que un smartphone puede usarse también en un ámbito profesional. Android es una plataforma donde los dispositivos son económicos con respecto a los profesionales y ofrece además una flexibilidad de uso prácticamente ilimitada.

1.1. Motivación

Al haber trabajado previamente con dispositivos profesionales para controlar el stock, se conoce de primera mano el bajo rendimiento no sólo de la aplicación, sino también del dispositivo adquirido a un precio desorbitado teniendo en cuenta las funciones y especificaciones ofrecidas. El poco acierto al escoger tanto el hardware como el software, reducía el rendimiento de la plantilla y producía una gran cantidad de errores que fácilmente se podrían haber reducido con un conjunto más depurado.

La llegada de Android abrió muchas puertas, pero a nivel profesional no parece ser de interés. Dispositivos con Windows (máquinas expendedoras de billetes de transporte) o Linux (en cajas registradoras de supermercados) acaparan gran parte del mercado. Con este proyecto se pretende mostrar que Android también está preparado para introducirse en el mundo profesional.

1.2. Objetivos

La aplicación de control de stock que a continuación se presenta, pretende controlar un stock tanto a nivel local para un único dispositivo, como para

un red de varios dispositivos conectados entre sí. Los objetivos marcados para el proyecto son:

- Que la aplicación sea capaz de mantener el correcto control de registros de forma local o remota contando a su vez con una interfaz simple e intuitiva.
- Que el acceso a la compra y venta de unidades se pueda realizar mediante la lectura del código de barras, simplificando de manera significativa la posibilidad de error y la facilidad de uso, o la escritura del mismo a elección del usuario.
- Mantener la base de datos en cada dispositivo local, para que la velocidad de consulta y actualización sea más rápida.
- Posibilidad de consulta de registros, que permita hacer una búsqueda concreta y/o se puedan mostrar todos los registros existentes.
- Dar la opción de trabajar en red y conectar varios dispositivos a un servidor Java remoto que comparta la información.
- Con el objetivo de simplificar la aplicación, acotar los tipos de registros disponibles a libros.

1.3. Metodología

No se ha utilizado ninguna metodología tradicional para la realización del proyecto, sin embargo si se ha seguido una estrategia particular para adquirir y desarrollar los conocimientos.

- Se han realizado tutoriales de dificultad progresiva, obtenidos de libros y artículos de internet, sobre los que se han implementando utilidades interesantes para la aplicación.
- Se ha mejorado el código según se dominaban nuevos conocimientos, realizando pruebas continuamente sobre dispositivos reales.
- Las pruebas han permitido la corrección y depuración de errores de código, el cual se ha tratado de mantener sencillo.

Una vez se obtuvieron los conocimientos de la herramientas ofrecidas tanto por el entorno de desarrollo como por el propio lenguaje de programación, se procedió a la escritura final del programa.

Capítulo 2

Estado de la tecnología

Un teléfono móvil es un dispositivo que recibe y origina llamadas a través de una red celular de telefonía móvil, lo cual permite la movilidad del usuario a lo largo del área que cubra dicha red.

En este capítulo se pretende dar a conocer la historia de los dispositivos y sus sistemas operativos.

2.1. Historia

Desde la aparición del primer dispositivo móvil en 1983 hasta el presente día, sus funciones y su uso han cambiado drásticamente.

El Dynatac 8000x desarrollado por Motorola de 33 centímetros de alto, autonomía de 1 hora y que funcionaba en AMPS (1G) se convirtió en el primer teléfono móvil de la historia. Véase Figura 2.1.

Figura 2.1: Motorola Dynatac 8000x

En IBM Simon Personal Communicator de 1992 puede considerarse el

primer smartphone ya que era un teléfono móvil, busca, fax y PDA todo en uno. Además incluía agenda, calendario, calculadora, pantalla táctil, entre otras aplicaciones. Véase Figura 2.2.

Figura 2.2: IBM Simon Personal Communicator

Con la llegada de las redes 2G, el gigante finlandés Nokia se convirtió en el mayor proveedor de teléfonos móviles en el mundo cuando en 1999 el Nokia 3210 y su sucesor el 3310 gozaron de gran popularidad. Se estima que el modelo 3210, figura 2.3, vendió 160 millones de unidades en todo el mundo, convirtiéndolo en uno de los terminales más exitosos de la historia mientras que el 3310 vendió 126 millones.

La aparición junto a GSM y DCS (voz) de GPRS y EDGE (estándares 2G para el uso de datos) supuso una adaptación de los terminales hacia la utilización de datos, a pesar de ser utilizado prácticamente en su totalidad a nivel profesional.

Figura 2.3: Nokia 3210

A pesar de todo, la sobrecarga de la red 2G obligó a buscar una nueva

alternativa, el 3G, lo que permitía que mediante un terminal móvil se pudiera establecer una conexión de datos suficientemente veloces para permitir VoIP y videollamada. Con ello llegaron los smartphone, impulsados por el Iphone 3G en 2008 con el que se podía navegar por internet, GPS, etcétera. Véase Figura 2.4.

Figura 2.4: IPhone 3G

Todas las características de los terminales móviles de la competencia palidecían al lado del Iphone. Los fabricantes de la competencia se unieron al sistema operativo de Google, Android. Un sistema operativo abierto basado en Linux, montado sobre dispositivos de última generación y que pudiera competir con el Iphone económica y socialmente.

2.2. Smartphone/tablet

La apertura en cuanto a la utilización de los teléfonos móviles, que en principio era fundamentalmente cursar llamadas, hasta el punto en el que nos encontramos ahora donde claramente ha sustituido a la PDA e incluso se dice que puede llegar a sustituir el ordenador, ha abierto una gran cantidad de puertas en muchos frentes. Una de las más explotadas es la del desarrollo de aplicaciones.

Un smartphone libre a dia de hoy se puede conseguir desde unos 85 euros, teniendo en cuenta que con un smartphone de gama baja se supera ampliamente cualquier terminal de hace 5 años. Visualmente las características más destacables de estos terminales suelen ser pantallas táctiles capacitivas de entre 3 y 5 pulgadas de diagonal.

El crecimiento en tamaño de los smartphone se debe al hecho de que cursar llamadas ha pasado a un segundo plano y se premia otro usos del terminal, tal como recibir/escribir e-mails, navegar por páginas web, jugar a

juegos, enviar mensajes cortos mediante diversas aplicaciones, visualización de fotos y vídeos, cámara digital, etc..

El hecho de que un smartphone sea un terminal portátil, provocó que se creara un segundo tipo de dispositivos inteligentes basados en la misma tecnología, las tablets. Véase Figura 2.5.

Figura 2.5: Tablets de 10.1 y 7 pulgadas y smartphone de 4.3 pulgadas

Las tablets son dispositivos de entre 7 y 10.1 pulgadas, que al fin y al cabo realizan las mismas funciones que un smartphone, ya que muchos modelos disponen de ranura para tarjeta SIM y así disfrutar de un plan de voz y datos como si de un smartphone se tratase.

El uso de las tablets es tema de discusión puesto que depende del ámbito en se utilice, ya que al disponer de un smartphone de unas pocas pulgadas más pequeño, la tablet puede no resultar útil. No obstante a nivel profesional la comodidad que ofrece este tipo de dispositivos es muy buena, como deja entrever el hecho de que el 80 % de los propietarios de una tablet la utilizan para trabajar.

2.3. Sistemas operativos para móviles

El sistema operativo es el software por el cual se maneja el hardware de nuestro dispositivo. A día de hoy los más populares son: iOS, Android, Blackberry y Windows Phone.

En el siguiente gráfico se observa que en los últimos años la fotografía ha cambiado radicalmente y Symbian, claro dominador durante la primera década del siglo, se ha hundido en favor de Android. Véase Figura 2.6.

2.3.1. Symbian OS

A pesar de que en la actualidad los únicos dispositivos que montan Symbian son los terminales residuales de otra época, debido al tremendo éxito

que tuvo en su momento, se debe nombrar.

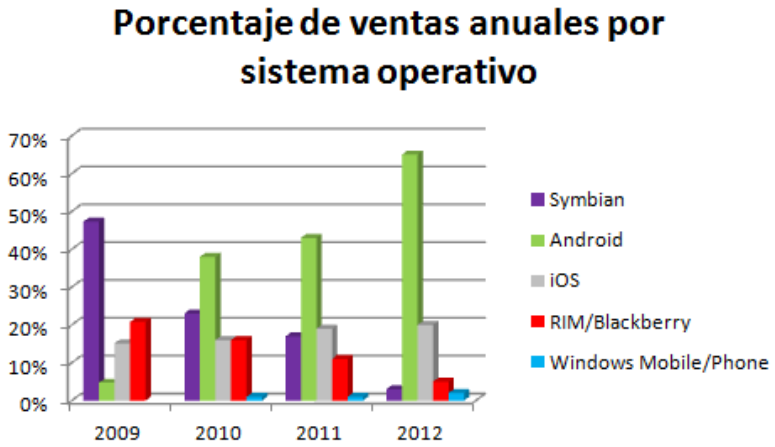

Figura 2.6: Gráfico de ventas de smartphones por sistema operativo

En 1998, Microsoft propuso a Nokia la creación de un sistema operativo para móviles con el fin de repetir el éxito de Windows en los ordenadores. No obstante el proyecto no fructiferó, y Nokia junto a Motorola, Sony Ericsson, Panasonic, Psion y Siemens crearon Symbian OS.

El problema es que los desarrolladores de Symbian no supieron reaccionar cuando Apple y Google sacaron sus respectivos sistemas operativos y terminales, perdiendo en 5 años toda la supremacía de la que gozaban.

En octubre de 2012 aparece la última actualización de Symbian, ya que no se van a desarrollar más versiones. Así, en verano de 2013 Nokia dejará de vender dispositivos con Symbian en beneficio de Windows Phone.

2.3.2. iOS

Apple siempre ha sido la alternativa a Windows en cuanto a ordenadores se refiere y siempre a la vanguardia en cuanto a dispositivos, ya fuera un ordenador portátil o un reproductor de música, aunque no a precios asequibles. Con la llegada del Iphone, se desató una revolución en cuanto a lo que todos conocíamos como teléfono móvil y al que a la competencia le costó reaccionar para ponerse al mismo nivel.

El concepto de la interacción con el terminal usando gestos múltiples supuso un gran cambio con respecto a los terminales de pantalla resistiva que por aquel entonces existían.

iOS se diseñó como un sistema operativo robusto y multitarea y la gran variedad de sensores en el terminal como acelerómetro, giroscopio, brújula,

sensor de proximidad, etcétera, abrió infinitas posibilidades a las aplicaciones de posicionamiento y a juegos. Adicionalmente se construyó un ecosistema para desarrolladores donde pudieran colgar sus propias aplicaciones, la App Store, algo que los otros fabricantes no habían podido conseguir.

Versión	Lanzamiento
iPhone OS 1.x	06/2007
iPhone OS 2.x	07/2008
iPhone OS 3.x	06/2009
iOS 4.x	06/2010
iOS 5.x	10/2011
iOS 6.x	09/2012
iOS 7.x	Otoño 2013

Tabla 2.1: Versiones de iOS

2.3.3. Android

Google, compró Android Inc en 2005 y ha desarrollado desde entonces el sistema operativo hasta convertirse en líder y gran rival de iOS. El gran titular entre ambas empresas son la gran cantidad de demandas cruzadas por plagio de patentes entre Apple y los grandes fabricantes de Android: Samsung y HTC entre otros. Incluso Google compró la filial de telefonía móvil de Motorola, Motorola Mobility, con el fin de hacerse con más de 17.000 patentes y proteger a sus fabricantes.

Se dice que al día se activan 1.5 millones de dispositivos en todo el mundo con Android, contando ya con 900 millones de dispositivos activos y con un crecimiento interanual del 400 %. Recientemente se ha lanzado la versión 4.3, señal de que Android no deja de crecer y evolucionar. A pesar de que no todos los dispositivos vayan a contar inmediatamente con esa versión, gracias a toda la comunidad de desarrolladores sin ánimo de lucro, se podrá gozar de la misma en prácticamente todos los dispositivos.

2.3.4. Windows Phone

Windows Phone es el sistema operativo que sustituye desde 2010 al antiguo Windows Mobile, y completamente incompatible con los dispositivos previos. Windows Mobile estuvo basado en el núcleo de Windows CE y presente como sistema operativo de muchos terminales móviles, como por ejemplo una amplia gama de dispositivos Motorola. Windows Phone en la actualidad lo podemos encontrar en smartphones HTC, Samsung, Huawei y en exclusiva en teléfonos Nokia (desde su asociación en 2011).

Versión	Lanzamiento
PocketPC 2002	10/2002
Windows Mobile 2003	06/2003
Windows Mobile 2003 Second Edition	03/2004
Windows Mobile 5.0	05/2005
Windows Mobile 6	02/2007
Windows Mobile 6.1	04/2008
Windows Mobile 6.5	05/2009
Windows Phone 7	11/2010
Windows Phone 8	10/2012

Tabla 2.2: Versiones de Windows Mobile y Windows Phone

Microsoft también ha estado presente en la guerra de patentes entre Google y Apple, llegando a obtener mayor beneficio por las patentes que HTC utilizaba ilegalmente, que por su propio sistema operativo.

2.3.5. Blackberry OS

Es el sistema operativo del fabricante canadiense de teléfonos móviles homónimo, antes RIM. El conjunto del dispositivo y el sistema operativo está orientado al uso profesional como gestor de correo electrónico y agenda, pudiendo ser sincronizados con diversos software para ordenador. En los últimos años la venta de dispositivos Blackberry ha caído en picado, llegando a salir del Top 10 de fabricantes de smartphones.

2.3.6. Ubuntu Edge/Ubuntu Touch

Ubuntu, una distribución de Linux basada en Debian y desarrollada por los socios de Canonical, ha decidido entrar en el mundo de los smartphone con su propio sistema operativo, Ubuntu Touch (versión para móviles de Ubuntu Desktop). Independientemente a ello aparece Ubuntu Edge, un proyecto financiado mediante crowdfunding y que a pesar de recoger 12 millones de dólares, no alcanzó la cifra establecida de 32 millones. El proyecto trataba de fabricar el smartphone más potente posible, al precio de 800 dolares, para que funcionara con dos sistemas operativos: Android y Ubuntu Touch, o conectarlo a una pantalla y usar como un pc normal.

2.3.7. Firefox OS

Es un sistema operativo lanzado en 2013, de código abierto y basado en el núcleo de Linux. Este proyecto esta apoyado por Mozilla Corporation

y Telefónica entre otros con el fin de competir frente al resto de sistemas operativos propietarios, haciendo en 2012 una demostración de Firefox OS en un dispositivo con Android. El primer Smartphone con Firefox OS fue el ZTE Open y se comercializa desde el 2 de Julio de 2013 en España de las manos de Telefónica.

2.4. Google

Google Inc, se fundó en 1998 como un motor de búsqueda en internet, y se ha ido especializando a lo largo de los años en servicios de internet con un gran catálogo de productos y servicios fuera de su ámbito inicial. Este amplio crecimiento se debe a la adquisición de muchas compañías con productos interesantes que luego Google seguiría desarrollando y comercializando, lo que les llevaría al gran éxito que hoy en día disfrutan. Algunos de los productos más populares de Google son:

- Android, sistema operativo para móviles.
- Gmail, gestor de correo.
- Google Chrome, navegador Web.
- Google Drive, almacenamiento en la nube.
- Google Maps, Google Earth y Google Street View. Localización mundial a vista de satélite, mapa o desde la misma localización.
- Youtube, visor online de vídeos.

A pesar de todo, no todos los productos de Google han sido rentables y con el tiempo muchos se han unificado a otros productos o simplemente han desaparecido. El último de ellos el lector de RSS, Google Reader.

Otro de los productos interesantes de la compañía, es su gama Nexus de smartphones y tablets. Los smartphone han sido fabricados por HTC, Samsung y LG y las tablets por Asus. Se tratan de dispositivos de gama alta a un precio competitivo enfocados a los desarrolladores, aunque disponibles para todos.

El último dispositivo de Google, Google Glass, es un ordenador que se puede llevar puesto como un accesorio y muestra información en una pequeña pantalla como si de un smartphone con manos libres se tratase. Las "gafas" funcionarían bajo Android e incorporarán una cámara, touchpad localizado en una lado para controlar el dispositivo, Wi-Fi, Bluetooth, giroscopio, acelerómetro, etcétera. Véase Figura 2.7.

Figura 2.7: Google Glass

Capítulo 3

Entorno de desarrollo

Para desarrollar una aplicación es necesario disponer del entorno de desarrollo adecuado, y específicamente para Android, existen programas como Eclipse y Android Studio.

Eclipse es un software de programación al que se le pueden integrar distintos entornos de desarrollo, mientras que Android Studio es ya un entorno de desarrollo integrado (IDE). No obstante, Android Studio fue lanzado en Mayo de 2013 y no todas las funciones están operativas.

En este capítulo se pretende enseñar los entresijos de Android, así como el software para desarrollar la aplicación y los conocimientos para adecuar nuestra aplicación a las distintas herramientas disponibles.

3.1. Plataforma Android

A pesar de que Google compró la empresa Android Inc, en 2005, no fue hasta que en 2007 se presentó el Iphone cuando se desarrolló Android tal y como lo conocemos en la actualidad. La idea de Google siempre ha sido vender la mayor cantidad de dispositivos con Android, ya fueran terminales de gama alta o de gama baja, convirtiéndolo en un sistema operativo de código abierto (aunque Google tiene que aprobar un dispositivo antes de permitir el uso de Android en el mismo) lo que permite un impulso extra a la plataforma desde la comunidad de desarrolladores.

Por otra parte, la tienda Google Play permite el acceso a los usuarios a miles de aplicaciones, tanto de pago como gratuitas, y que convierten al terminal móvil en un dispositivo con funciones prácticamente ilimitadas.

La estrategia de Google ha funcionado perfectamente y en algunos países como en España la cuota de mercado es estratosférica. Véase Figura 3.1.

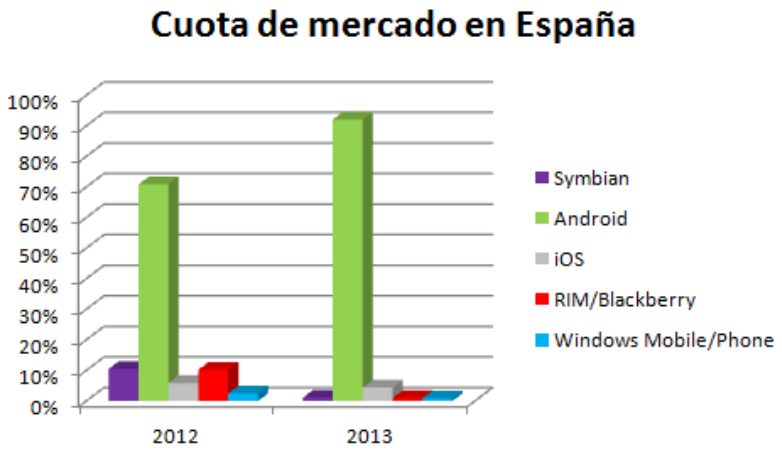

Figura 3.1: Porcentaje de dispositivos vendidos según sistema operativo

3.1.1. Arquitectura

Los componentes fundamentales que contiene un terminal con Android son:

- Aplicaciones básicas instaladas escritas en Java, como correo electrónico, navegador, etcétera.
- Librerías de bajo nivel escritas en C/C++.
- Framework para los desarrolladores con el fin de simplificar el uso de componentes entre aplicaciones.
- Conjunto de librerías base del lenguaje Java, así como la máquina virtual Dalvik para que cada aplicación tenga su propio proceso.
- Núcleo de Linux. Pilar base en cuestión de software sobre el que se asienta todo el sistema operativo de Android. El núcleo se encarga de la seguridad, gestión de drivers, gestión de memoria, gestión de procesos, etcétera.

3.1.2. Componentes de la aplicación

Una parte fundamental de todas las aplicaciones de Android son los componentes que la forman y una forma de entrar y/o hacer uso de la misma.

A continuación se enumeran los cuatro tipo de componentes y su propósito.

- Activity. Un Activity representa una pantalla en la aplicación, es decir, tiene una interfaz de usuario sobre la que se interacciona.
- Intent. Se utiliza para moverse entre activities. Describe lo que la aplicación quiere hacer sobre una acción y sobre dónde.
- Service. Es un código que no utiliza interfaz gráfica. Además se puede mantener en funcionamiento aunque se cambie de activity.
- Content provider. Permite a las aplicaciones guardar y usar datos de otras aplicaciones.

3.1.3. Ciclo de vida de una actividad

A lo largo de la vida de una actividad esta puede presentar varios estados. El control del mismo lo realiza el sistema, pero se puede programar para que la actividad reaccione de una forma u otra según el estado en el que se encuentra. Véase Figura 3.2.

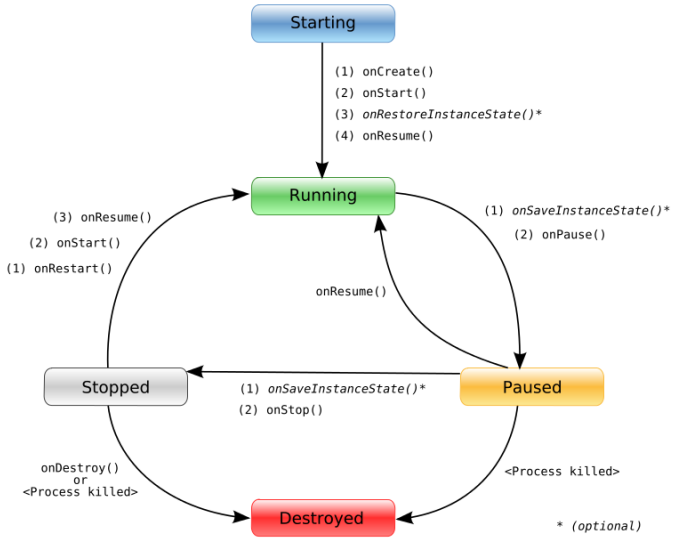

Figura 3.2: Ciclo de vida de una actividad en Android

- onCreate(). El sistema llama a este método cuando la actividad se inicia por primera vez, por ejemplo, para crear la interfaz de usuario.
- onStart(). Indica que el activity va ser visualizable al usuario.

- onResume(). Se llama a este método cuando la actividad puede interaccionar con el usuario.
- onPause(). Indica cuando una actividad va a pasar a un segundo plano, probablemente porque se ha lanzado una nueva actividad.
- onStop(). Se utiliza cuando la actividad no es visible y puede que nos ea usada en un tiempo.
- onRestart(). La actividad vuelve a un primer plano después de haber estado "parada".
- onDestroy(). Se llama antes de que se destruya una actividad.

Es importante saber que las actividades que están en segundo plano pueden ser detenidas por el sistema si no hay suficiente memoria para desarrollar otra actividad en primer plano.

3.2. IDE Eclipse con ADT Bundle y SDK Android

Eclipse es un software multiplataforma y de código abierto que se utiliza para desarrollar aplicaciones. Fue desarrollado en sus inicios por IBM, aunque en la actualidad es desarrollado por una organización independiente cuyo objetivo no es obtener beneficios económicos.

Para el desarrollo de la aplicación se ha utilizado la versión Eclipse Java EE IDE for Web Developers, véase figura 3.3, y además deberemos tener instalado Java en nuestro ordenador.

Pero como se ha comentado anteriormente, Eclipse no es un programa de desarrollo nacido para desarrollar aplicaciones para Android por lo que será necesario instalar unos plugins para ello.

Se trata del plugin ADT Bundle. Es un paquete proporcionado por Google con Eclipse IDE y los programas y herramientas necesarios para el desarrollo. Dos de esos programas son Android SDK Tools y Android Platform-tools.

- SDK Tools es un componente que contiene todo lo necesario para desarrollar y depurar la aplicación.
- Platform-tools contiene las librerías necesarias para desarrollar una aplicación en Android para una o varias versiones concretas de Android.

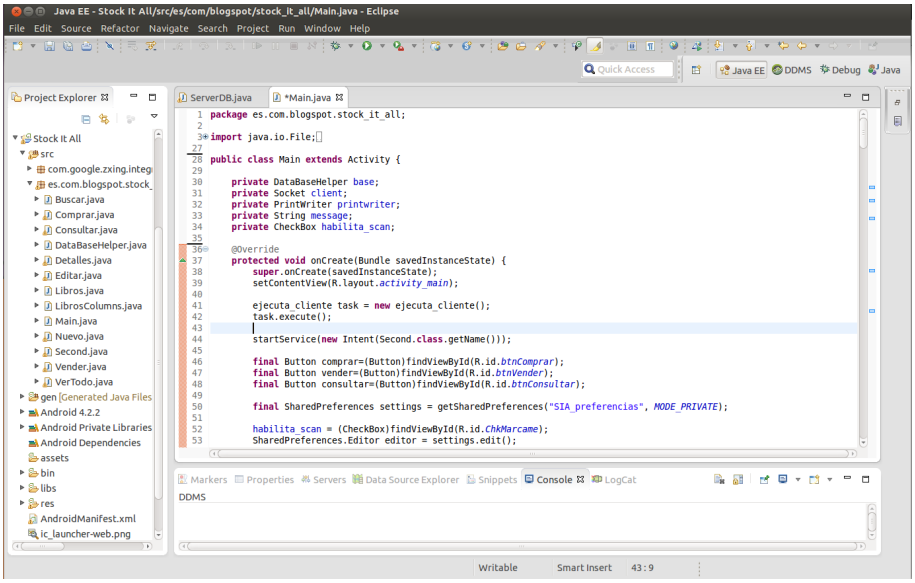

Figura 3.3: Eclipse Java EE IDE for Web Developers

A través de Android SDK Manager podremos gestionar los componentes citados anteriormente para actualizar librerías y descargar nuevas versiones de Android.

3.3. Niveles API

Un nivel API es un valor entero que identifica la revisión del entorno de trabajo ofrecida para una versión de Android. Cada nuevo nivel API está diseñado con compatibilidad hacia atrás y puede introducir o actualizar funcionalidades. Las más antiguas dejan de usarse en favor de las nuevas para niveles API más altos, pero no se eliminan con el fin de que las aplicaciones existentes puedan seguir usándolas. Por ejemplo la funcionalidad del menú ha sido sustituida por la Action Bar introducida a partir del nivel 11. Eso significa que el desarrollador puede implementar el menú como siempre lo ha hecho, pero en los niveles inferiores al 11 tendremos que pulsar el botón menú para acceder al mismo mientras que desde el 11 hacia adelante, nos aparecerá una barra en la zona superior de la pantalla.

El objetivo de indicar el nivel mínimo de API soportado, es para comunicar al sistema que la aplicación está utilizando herramientas desarrolladas a partir de ese nivel y de esa forma, no permitiendo que se instale en dispositivos con niveles API inferiores evitando fallos. En la tabla 3.1 se muestra la relación de niveles con las versiones de Android.

Nombre	Versión	Nivel API	Lanzamiento
(sin nombre)	1.0	API 1	23/09/2008
(sin nombre)	1.1	API 2	9/02/2009
Cupcake	1.5	API 3, NDK 1	30/04/2009
Donut	1.6	API 4, NDK 2	15/09/2009
Eclair	2.0	API 5	26/10/2009
Eclair	2.0.1	API 6	3/12/2009
Eclair	2.1	API 7, NDK 3	12/01/2010
Froyo	2.2.x	API 8, NDK 4	26/10/2009
Gingerbread	2.3 - 2.3.2	API 9, NDK 5	6/12/2010
Gingerbread	2.3.3 - 2.3.7	API 10	9/02/2011
Honeycomb	3.0	API 11	22/02/2011
Honeycomb	3.1	API 12, NDK 6	10/05/2011
Honeycomb	3.2.x	API 13	15/07/2011
Ice Cream Sandwich	4.0.1 - 4.0.2	API 14, NDK 7	19/10/2011
Ice Cream Sandwich	4.0.3 - 4.0.4	API 15, NDK 8	16/12/2011
Jelly Bean	4.1.x	API 16	9/07/2012
Jelly Bean	4.2.x	API 17	13/11/2012
Jelly Bean	4.3	API 18	24/07/2013

Tabla 3.1: Relación de niveles API con versiones Android

Cuando hay que desarrollar una aplicación hay que pensar cual va a ser el rango de dispositivos que podrán ser capaces de utilizarla. Según el gráfico siguiente, figura 3.4, si ponemos como SDK mínimo API 8 accederemos al 98.7 % de los dispositivos con Android. No obstante la elección no puede basarse únicamente en este apartado, puesto a que a mayor nivel API mínimo decidido, obtendremos un mayor número de herramientas a nuestro alcance.

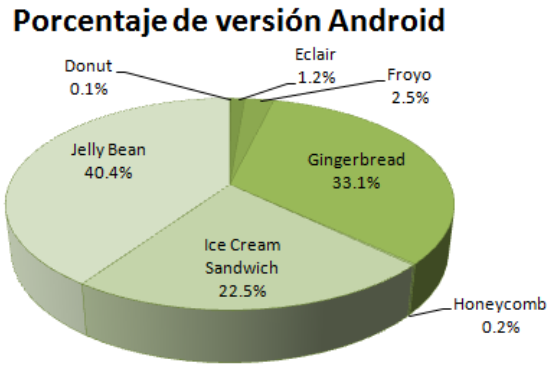

Figura 3.4: Porcentaje de dispositivos Android según versión

3.4. AVD

En el momento de probar una aplicación de Android, y si no disponemos de un dispositivo físico o este no tiene la versión necesaria, se puede utilizar un dispositivo virtual (AVD). Para configurar el AVD se debe tener instalada la versión en la que se pretende probar la aplicación, así como otras herramientas. Todos ellos se obtienen del paquete Android Platform-tools. La configuración se realiza a través de un programa llamado AVD manager que mediante una interfaz gráfica permite gestionar y crear distintos dispositivos virtuales.

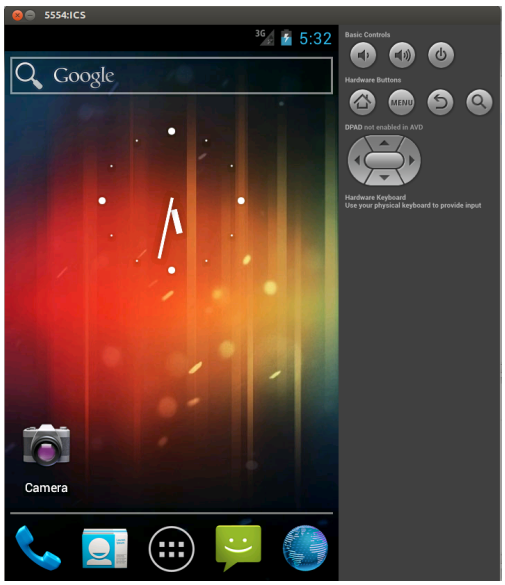

Figura 3.5: Dispositivo virtual Android

A pesar de todo, un ADV tiene ciertas limitaciones como por ejemplo el uso de la cámara. Se consiguió hacer funcionar la cámara con la aplicación propia de Android en el dispositivo virtual, pero no fue posible utilizarla en la aplicación desarrollada. Como la utilidad de la cámara es muy importante para la aplicación, ya que mediante ella se captura el código de barras simplificando el trabajo, fue necesario obtener un dispositivo físico para realizar las pruebas.

3.5. Estructura de una aplicación Android

Una vez creado un proyecto Android con Eclipse, nos encontraremos con la siguiente estructura de carpetas:

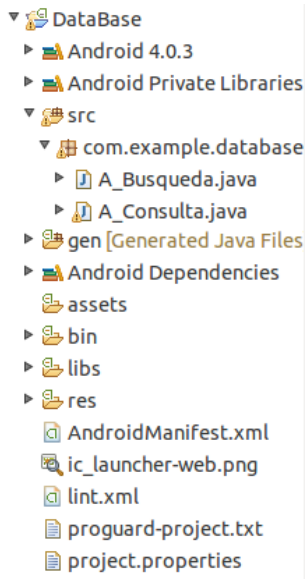

Figura 3.6: Estructura de una aplicación Android

Detalladamente cada una de las carpetas contiene los siguientes ficheros:

- Carpeta src. Contiene el código de nuestra aplicación, ya sean actividades con interfaz gráfica, servicios de ejecución en segundo plano, clases auxiliares, etcétera.

- Carpeta res. En ella se encuentran todos los recursos externos que nuestra aplicación va a necesitar, tales como imágenes clasificadas por resolución, los ficheros de diseño gráfico según la orientación, menús, cadenas de texto, etcétera.

- Carpeta gen. Son archivos generados automáticamente. Cabe destacar la clase R que contendrá unas constantes asignadas de forma automática de todos los recursos de la carpeta res.

- Carpeta assets. Se guardarán ficheros auxiliares como en la carpeta res, pero para acceder a estos se utilizará su ruta y no su constante en la clase R.

- Carpeta bin. Aquí se encuentran los elementos compilados, entre ellos el *.apk que será el instalador de la aplicación.

- Carpeta libs. Por si es necesario utilizar librerías auxiliares.

3.6. Android Manifest

Este archivo XML no se encuentra dentro de ninguna de las carpetas comentadas antes y aún así es uno de los más importantes, ya que en él se declaran todas las funciones de las que va a hacer uso nuestra aplicación, es decir, se declaran los activities, los services, los permisos, se define la actividad principal, etcétera.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="es.com.blogspot.stock_it_all"
4      android:versionCode="1"
5      android:versionName="1.0" >
6      <uses-permission android:name="android.permission.INTERNET" >
7      </uses-permission>
8      <uses-sdk
9          android:minSdkVersion="7"
10         android:targetSdkVersion="17" />
11     <application
12         android:allowBackup="true"
13         android:icon="@drawable/ic_launcher"
14         android:label="@string/app_name"
15         android:theme="@style/AppTheme" >
16         <activity
17             android:name="es.com.blogspot.stock_it_all.Main"
18             android:label="@string/app_name" >
19             <intent-filter>
20                 <action android:name="android.intent.action.MAIN" />
21                 <category android:name="android.intent.category.LAUNCHER" />
22             </intent-filter>
23         </activity>
24         <service
25             android:name="es.com.blogspot.stock_it_all.Second"
26             android:label="@string/app_name"
27             android:process=":remote" >
28             <intent-filter>
29                 <action android:name="es.com.blogspot.stock_it_all.Second" />
30             </intent-filter>
31         </service>
32     </application>
33 </manifest>
```

Figura 3.7: Android Manifest

Dentro del archivo nos encontramos las siguientes etiquetas:

- <manifest>. En esta etiqueta definimos la versión de desarrollo de nuestra aplicación y la versión y el paquete con el que referenciamos el proyecto en Google Play.

- <application>. Definimos las actividades y servicios y dentro de cada uno de ellos el nombre de la clase, la etiqueta que aparecerá en la parte superior de la Action Bar. También se definirá que activity será la principal y con cual se iniciará la aplicación.

- <uses-sdk>. Se determina el rango de versiones soportadas por nuestra aplicación.
- <uses-permissions>. Son los permisos que necesita la aplicación para poder ejecutarse y que el usuario deberá aceptar antes de instalar la misma. Por ejemplo si se quiere hacer uso de internet o acceso a una tarjeta de almacenamiento externa, se tendrán que definir un los permisos de antemano. Para consultar la lista completa de permisos ver el anexo A.

Para la aplicación de control de stock desarrollada se han usado los siguientes permisos:

- INTERNET para poder conectar por Wi-Fi los distintos dispositivos.
- READ_PHONE_STATE para enviar al servidor el número único de nuestro dispositivo.
- WRITE_EXTERNAL_STORAGE para poder hacer un backup de la base de datos en la tarjeta SD.

Capítulo 4

Software para control de stock

La aplicación desarrollada consiste en poder llevar a cabo de forma ordenada un control de registros, y poder añadir y extraer unidades así como realizar búsquedas. Este proyecto esta particularizado para el control de libros.

Además del uso local para un dispositivo único, la aplicación permite la conexión a un servidor remoto escrito en Java que guardará su propia copia de la base da datos, y que además servirá como conexión intermedia de múltiples dispositivos gestionando que todos ellos se mantengan actualizados al momento de su primera conexión y mientras la aplicación este en uso. Además el hecho de que los dispositivos tengan la base de datos local permite que las actualizaciones y consultas se realicen con mayor velocidad.

En este apartado se va a estudiar el código clave de la aplicación.

4.1. Dispositivos de testeo

La aplicación esta desarrollada para dispositivos de entre 5 y 7 pulgadas. Para testear la aplicación se han utilizado las dos tablets de la figura 2.5.

A pesar de que la aplicación se ha desarrollado para dispositivos modestos como la tablet de 7 pulgadas comentada, se ha utilizado otra tablet de 10 pulgadas debido a que el dispositivo de 7 pulgadas no dispone de cámara, aunque el de 10 no reúna las especificaciones de tamaño ideales sí dispone de ella. Además se trata de un dispositivo de gama media más moderno, por lo que la velocidad de procesado es mayor.

Para consultar una tabla con las especificaciones de ambos dispositivos, ver apéndice B.

El servidor Java se ha montado sobre un ordenador portátil Intel Core

i3 con Ubuntu 12.04 y se ha utilizado Eclipse para su funcionamiento.

Por lo tanto para probar el correcto funcionamiento del servidor remoto ha sido necesario el uso de ambas tablets de forma simultanea.

4.2. Estructura de la aplicación

Las pantallas a las que el usuario tendrá acceso en la aplicación las componen los siguientes procesos representados en el siguiente diagrama de flujo, figura 4.1.

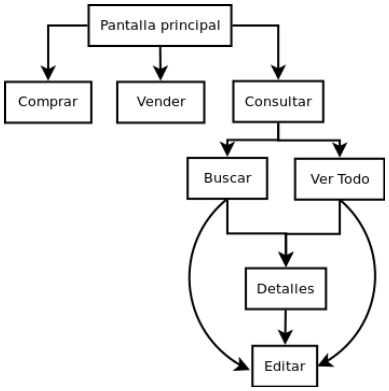

Figura 4.1: Estructura de la aplicación

Aunque visualmente solo se aprecien las actividades anteriormente representadas, cuando se crea la actividad principal esta llama a un servicio (una actividad sin interfaz gráfica) que se mantendrá en ejecución en segundo plano mientras tengamos la aplicación abierta.

4.3. Funcionamiento de la aplicación

La aplicación al iniciarse creará un base de datos local (si no existiera) y después en caso de encontrar conexión con el servidor remoto, procederá a actualizar los registros editados o incluidos previamente por otros dispositivos y que se hallan registrados en la base de datos local del servidor. Una vez la actualización termina, la aplicación continuará en escucha de posibles actualizaciones por parte de otros dispositivos de las cuales el servidor le informará de forma instantánea.

Por otro lado, el dispositivo que inicia el proceso de actualización realizando cambios sobre su propia base de datos, notificará al servidor del cambio. Este actualizará su base de datos propia e irá notificando al resto de

dispositivos uno a uno para que hagan lo propio con su base de datos local.

Las actualizaciones permitidas son la eliminación o inclusión de registros y la edición de campos o únicamente de unidades.

Es importante conocer que todos los cambios se realizan primero sobre la base de datos local para agilizar el proceso en cada dispositivo y después se notifica en segundo plano.

Por otra parte se ha incluido en el menú la posibilidad de escribir la ip del servidor y a continuación intentar la nueva conexión, borrar la base de datos por si es necesario reiniciarla y hacer una copia de seguridad a la tarjeta SD.

4.4. Pantalla principal

Desde la pantalla principal podremos acceder a todas las funciones de la aplicación. Pero antes de permitir al usuario el manejo completo, es necesario automatizar ciertas tareas.

Cuando se crea la actividad Main.java, es decir, cuando el usuario lanza la aplicación, se ejecuta un clase llamada *ejecuta_ cliente* que es una extensión de Asyntask, para ponerse en contacto con el servidor y hacerle saber que se ha conectado y cual es su Android ID, número único de cada dispositivo Android. Si esta conexión no tiene éxito la aplicación se podrá seguir usando, pero los cambios que se realicen solo estarán en la base de datos local, por lo que no se recomienda.

La clase *ejecuta_ cliente()* se estudiará más adelante en la sección 4.9.

Por otro lado, se llama a la función DataBaseHelper.java donde se creará la base de datos si no existiera.

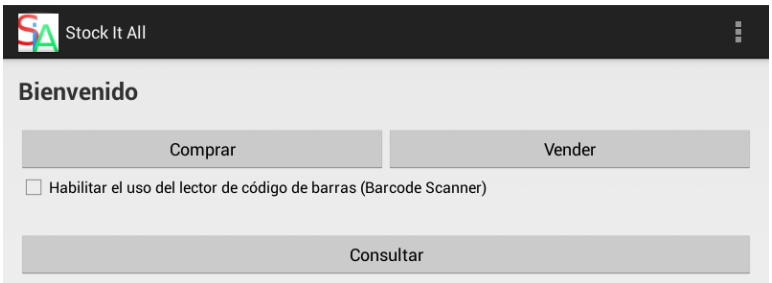

Figura 4.2: Interfaz principal

Visualmente hay 3 botones: comprar, vender y consultar, así como un *checkbox* al que responderán las actividades comprar y vender. Véase Figura 4.2. Si esta marcado, habilitaremos el escáner para la introducción de códigos de barras, si no se tendrá que introducir de forma manual. La de-

cisión del usuario al respecto, se mantendrá a lo largo de toda la sesión a no ser que se cambie.

Cuando se sale de la aplicación, únicamente desde la actividad principal, se notificará al servidor un mensaje para que gestione adecuadamente la salida de este dispositivo.

4.4.1. Tarea en segundo plano

Como se ha comentado en el punto anterior, al iniciar la aplicación se lanzará la tarea en segundo plano *Second.java*. En la figura 4.3, se observa el esquema de la misma.

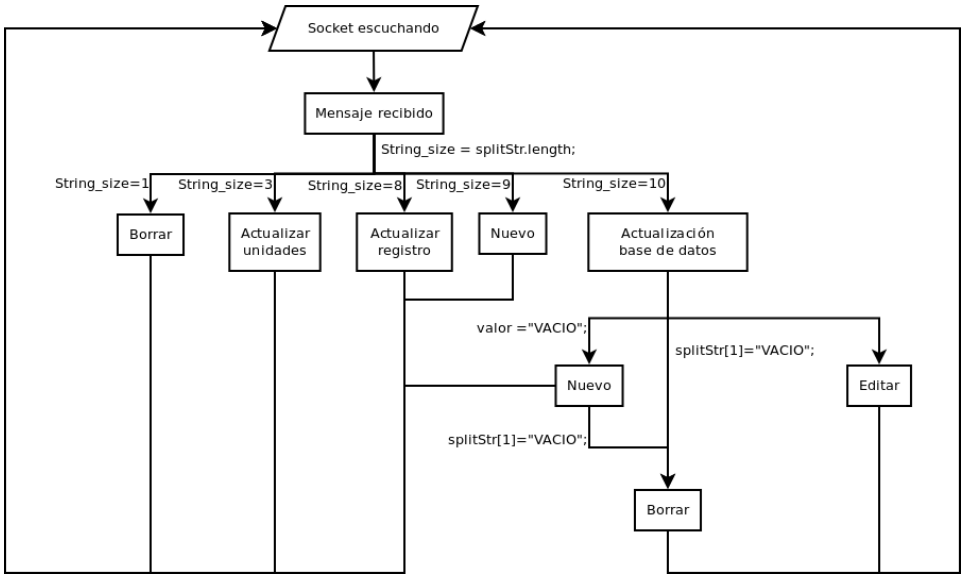

Figura 4.3: Diagrama de flujo del servicio en segundo plano.

La función de este servicio es la de mantener un socket de escucha abierto, independientemente de la actividad en la que el usuario se encuentre.

```
@Override
    public void onCreate() {
      super.onCreate();

          try {
              serverSocket = new ServerSocket(4444);

          } catch (IOException e) {
           e.printStackTrace();
          }
          tarea1 = new MiTareaAsincrona();
```

```
        tarea1.execute();
    }
```

Este socket permanecerá en escucha hasta que reciba un mensaje del servidor remoto Java. Se capturará dicho mensaje y según el tamaño del array extraído se realizará una acción u otra. Estas acciones se realizarán en el método actualizaDB(String) y podrán ser:

- **Tamaño 1.** Borrar un registro según su ID.
- **Tamaño 3.** Sumar o restar unidades.
- **Tamaño 8.** Edición de un registro.
- **Tamaño 9.** Introducción de un nuevo registro.
- **Tamaño 10.** Actualizará los registros que el servidor le envie. Solo puede deberse en la primera conexión.

Una vez se termina la actualización de la base de datos, el socket se mantendrá a la espera de nuevos mensajes.

Para poder mantener una tarea en segundo plano se ha utilizado la clase Asyntask, que permite ejecutar varias tareas asíncronas sin bloquear la interfaz gráfica. Como Asyntask solo se puede utilizar una vez, ha sido necesario el uso de un *handler* para poder usar el mismo hilo varias veces.

```
private class MiTareaAsincrona extends AsyncTask<Void,
Integer, Boolean> {

        Runnable myRunner = new Runnable(){
              public void run() {
                 new MiTareaAsincrona().execute();
         }};


         @Override
         protected Boolean doInBackground(Void... params) {


              try {

            // Se acepta a conexión
                 clientSocket = serverSocket.accept();

                 //Se lee el mensaje
```

```
                inputStreamReader = new InputStreamReader(
                clientSocket.getInputStream());

                bufferedReader = new BufferedReader(
                inputStreamReader);

                message = bufferedReader.readLine();

                //Se cierra el socket
                clientSocket.close();


                  // Actualizar DB
                  actualizaDB(message);


              }
               catch (IOException e) {
               e.printStackTrace();

              }
           return true;
        }

        @Override
        protected void onPostExecute(Boolean result) {
           if(result)

               myhandler.post(myRunner);

               Toast.makeText(getApplicationContext(),
               "Base de datos actualizada",
               Toast.LENGTH_SHORT).show();

        }

        @Override
        protected void onCancelled() {
           Toast.makeText(getApplicationContext(),
           "Actualización cancelada", Toast.LENGTH_SHORT).
           show();
        }
    }
```

4.4.2. SQLite

Android incorpora el paquete de sqlite para la creación y gestión de base de datos. Para ello tendremos que crear una clase que sea una extensión de SQLiteOpenHelper, en este caso la clase se llama *DataBaseHelper.java*.

Junto a esta clase obligatoriamente se deben crear tres subclases:

- **DataBaseHelper** Se define el nombre de la base de datos y la versión, muy importante para la subclase *onUpgrade*.
- **onCreate** Si la base de datos ya esta creada no se ejecutará, pero si no lo esta se creará con las columnas definidas.
- **onUpgrade** En esta subclase se decide que hacer si las versiones no coinciden. En nuestro caso se borrará la base de datos y se creará una nueva con los parámetro de *onCreate*.

4.4.3. Menu

Dentro del menú hay cuatro opciones: Nuevo, Borrar DB, Copiar DB a la SD y Configuración. La primera se refiere a añadir un nuevo registro y se estudiará en la sección 4.8. La segunda, Borrar DB, se refiere al borrado completo de la base de datos. Esta acción es irreversible.

```
try{

    File file = new File("data/data/es.com.blogspot.
    stock_it_all/databases/registro.db");

    file.delete();

}catch(Exception ex){

}
```

La tercera opción se trata de hacer un *backup* de la base de datos a una tarjeta externa SD. Al método copiarDB se le pasan las rutas de origen y de destino y se copia de la siguiente forma:

```
public static void copiarDB(File src, File dst) throws
IOException{

    //Comprueba que exista el ficher en la tarjeta SD
    //Si no existe lo crea
```

```
    File rutaSD = new File("sdcard", "Stock-It-All");
    if(!rutaSD.exists())
        rutaSD.mkdirs();

     FileChannel inChannel = new FileInputStream(src)
     .getChannel();
     FileChannel outChannel = new FileOutputStream(dst)
     .getChannel();

     try
     {
         inChannel.transferTo(0, inChannel.size(),
         outChannel);
     }
     finally
     {
         if (inChannel != null)
             inChannel.close();
         if (outChannel != null)
             outChannel.close();
     }
}
```

Por último en la opción de configuración podremos configurar la IP del servidor remoto.

La configuración del servidor se quedará guardada en las preferencias de la aplicación para no tener que configurarlo cada vez que se inicia. Al abrir el cuadro de dialogo primero extraeremos la información guardada previamente y la mostraremos.

```
// Se abren las preferencias
final SharedPreferences settings = getSharedPreferences
("SIA_preferencias", MODE_PRIVATE);

String ip_serv = settings.getString("IpServidorJava",
 "192.168.1.64");

Ip.setText(ip_serv);
```

Con este código se extrae la información existente en el cuadro de texto, se guarda en las preferencias para futuros usos y se ejecuta de nuevo la clase *ejecuta_cliente* para intentar una nueva conexión con el servidor.

```
Ip.getText();
```

```
SharedPreferences.Editor editor = settings.edit();
editor.putString("IpServidorJava", String.valueOf(Ip.
getText()));

editor.commit();

ejecuta_cliente task_2 = new ejecuta_cliente();
task_2.execute();
```

4.5. Comprar y vender unidades

A esta actividad se accede desde el menú principal. Ambas son lo mismo, solo que en comprar se añaden unidades y en vender se sustraen, por lo que solo se va a explicar *Comprar.java*.

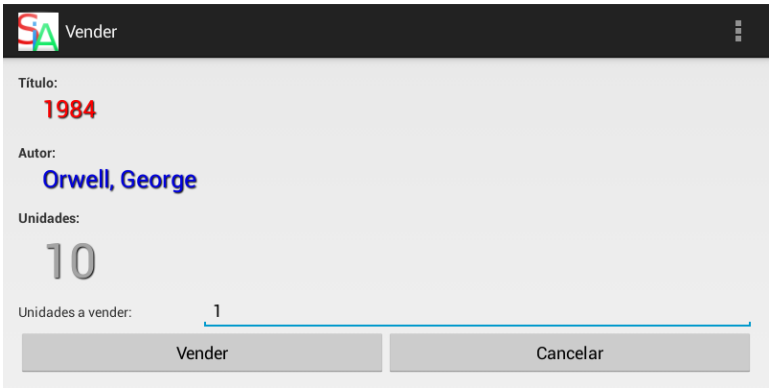

Figura 4.4: Interfaz para vender unidades

Como se ha comentado con anterioridad, el cuadro que habilita el escáner solo afecta a estas dos actividades. Al iniciarlas se comprobará el estado, si esta marcado se lanzará un *intent* hacia el programa Barcode Scanner, que realizará el escaneado del código de barras. Para ello se ha añadido al proyecto las librerías *IntentIntegrator.java* e *IntentResult.java* proporcionadas por el equipo Zxing (desarrolladores de la aplicación Barcode Scanner).

```
SharedPreferences settings = getSharedPreferences
("SIA_preferencias", MODE_PRIVATE);

        String Hab_Scan = settings.getString("Habilita_Scan",
         "NO");

        if (Hab_Scan.equals("YES")){
```

```
IntentIntegrator scanIntegrator = new
IntentIntegrator(this);

scanIntegrator.initiateScan();

}else{

    insertarCodigo();

}
```

Si no estuviera el programa instalado en nuestro dispositivo, se nos llevaría a Google Play para descargarlo. Por otro lado, si la casilla estuviera desmarcada el código se introduciría manualmente.

Si el código introducido obtiene coincidencias en la base de datos se mostrará: título, autor y unidades actuales. Bajo estos datos se mostrará un cuadro de introducción de texto donde por defecto aparece una unidad, aunque se puede modificar. Véase Figura 4.4.

En el caso de *Vender.java*, se comprueba que el número de unidades a vender sea superior al número de unidades disponibles. Si es así no se permite que continúe la acción.

Una vez se guardan los cambios, se envía esta información al servidor remoto para que lo comparta con los otros dispositivos mediante la función *ejecuta_ cliente()*. La diferencia radica en el hecho de que al comprar/vender enviaremos un String con 3 datos separados por espacios: ID, Comprar/Vender y el nuevo número de unidades.

4.6. Consulta de registros

En esta pantalla podemos hacer dos cosas: hacer una búsqueda o ver todos los registros. Véase Figura 4.5.

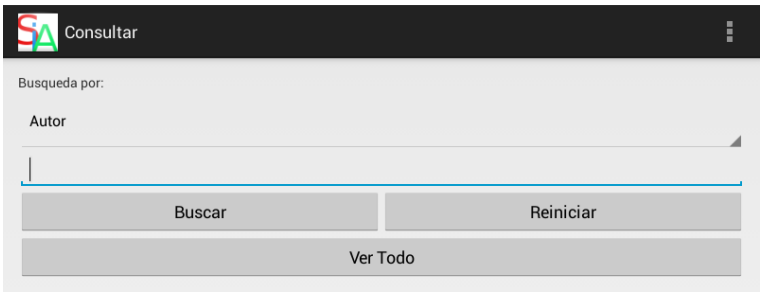

Figura 4.5: Interfaz de consulta

Para hacer una búsqueda en particular, hay un *control spinner* que nos permitirá elegir qué buscar. Los campos seleccionables son: autor, título y código de barras.

```
//Referencia al control
final Spinner cmbOpciones = (Spinner)findViewById(R.id.
CmbOpciones);


//Datos en la lista desplegable
final String[] datos= new String[]{"Autor","Titulo",
"Código de Barras (Manual)", "Código de Barras (Scanner)"};

//Declaramos el adaptador y el layout con la lista desplegada
ArrayAdapter<String> adaptador =new ArrayAdapter<String>(
this,android.R.layout.simple_spinner_dropdown_item, datos);

//Layout cuando la lista no esta desplegada
adaptador.setDropDownViewResource(android.R.layout.
simple_spinner_dropdown_item);

// Asignación del adaptador al control
cmbOpciones.setAdapter(adaptador);
```

Para el código de barras podremos seleccionar una búsqueda manual donde se introducirá en el cuadro de texto inferior o lanzar el escaner. Es muy importante que el texto introducido sea exacto al registro que se busca.

4.6.1. Buscar

La actividad Buscar.java es una extensión de ListActivity, es decir va a ser una lista. Al crear la actividad en el caso de Buscar.java se recogerán los datos a buscar desde Consultar.java, se realizará una consulta y mostrará coincidencias, si las hubiera, mediante el método mostrarLibros(). Este método llama a getLibrosConsultados() en el cual se realizará la consulta.

Para lograr todo eso lo primero es crear un ArrayList con el resultado de la consulta. Según el parámetro de búsqueda se usará una de las *queries* siguientes:

```
if("Titulo".equals(lblMensaje)) {

    c = db.query("libro", campos, "titulo=?", args, null,
    null, "autor");

} else if("Autor".equals(lblMensaje)) {
```

```
    c =db.query("libro", campos, "autor=?", args, null,
    null, "titulo");

} else if("Código de Barras (Manual)".equals(lblMensaje)){

    c =db.query("libro", campos, "cod_barras=?", args, null,
    null, "autor");

} else if("Código de Barras (Scanner)".equals(lblMensaje)){

     c =db.query("libro", campos, "cod_barras=?", args, null,
     null, "autor");
}
```

Y de ellas extraeremos los parámetros: id, título, autor, editorial, número de páginas, género, edición, año y unidades.

```
if (c.moveToFirst()) //Nos aseguramos de que exista un registro
//Recorremos el cursor hasta que no haya más registros
 do {
    libros.add(new Libros(c.getInt(c.getColumnIndex(
    LibrosColumns.ID)),  c.getString(c.getColumnIndex(
    LibrosColumns.TITULO)), c.getString(c.getColumnIndex(
    LibrosColumns.AUTOR)),c.getString(c.getColumnIndex(
    LibrosColumns.EDITORIAL)),c.getInt(c.getColumnIndex(
    LibrosColumns.NUM_PAG)),c.getString(c.getColumnIndex(
    LibrosColumns.GENERO)),c.getInt(c.getColumnIndex(
    LibrosColumns.EDICION)),c.getInt(c.getColumnIndex(
    LibrosColumns.AÑO)),c.getInt(c.getColumnIndex(
    LibrosColumns.UNIDADES))));

 } while(c.moveToNext());
```

Una vez tenemos nuestro ArrayList completo se creará un *Hashmap* con los datos del mismo. Se reemplazarán los espacios, ya que la consulta de registros con este carácter en una base de datos sqlite es irregular se utilizará una palabra comodín, es este caso *BLANKSPACE*.

```
String titulo_esp = libro.getTitulo();
titulo_esp = titulo_esp.replace("BLANKSPACE", " ");
```

Una vez se tienen todos los datos tal cual queremos mostrar se crear una SimpleAdapter y se muestran los campos título, autor y editorial. Si hubieran varios registros que compartieran el texto introducido en la búsqueda, aparecerían todas las coincidencias en la lista.

```
final String[] from = { LibrosColumns.TITULO,
LibrosColumns.AUTOR , LibrosColumns.COD_BARRAS,
LibrosColumns.EDITORIAL};

final int[] to = new int[] { R.id.titulo , R.id.autor ,
R.id.editorial};


SimpleAdapter ListAdapter = new SimpleAdapter(this ,
HLibros, R.layout.mostrar, from, to);

setListAdapter(ListAdapter);
```

Además si se mantiene una presión larga sobre el elemento deseado, aparecerá un menú contextual con las opciones de borrar y editar.

```
@Override
public void onCreateContextMenu(ContextMenu menu,
View v, ContextMenuInfo menuInfo) {

    super.onCreateContextMenu(menu, v, menuInfo);

    if (v.getId() == android.R.id.list){

        AdapterView.AdapterContextMenuInfo info =
        (AdapterView.AdapterContextMenuInfo) menuInfo;

        @SuppressWarnings("unchecked")
        HashMap<String, String> item = (HashMap<String,
        String>) this.getListAdapter().getItem(info.position);

        menu.setHeaderTitle(item.get(LibrosColumns.TITULO) +
         " por " + item.get(LibrosColumns.AUTOR));

        String[] menuItems = {"Borrar","Actualizar","Cancelar"};

        for (int i = 0; i < menuItems.length; i++) {
        menu.add(Menu.NONE, i, i, menuItems[i]);
        }
    }
}
```

Con una presión corta se accederá al detalle del registro.

4.6.2. Ver Todo

La actividad VerTodo.java, al igual que Buscar.java, es una extensión de *ListActivity*. Cabe comentar que Vertodo.java es independiente de los campos introducidos en la pantalla de consulta y mostrará todos los registros de la base de datos.

Por lo tanto, a diferencia de Buscar.java, la consulta será sobre toda la base de datos y los registros se ordenarán por orden alfabético con respecto al autor.

```
int i = 0;
String[] campos = new String[] {"_id", "titulo",
"autor", "editorial", "num_paginas", "genero",
"edicion", "año","unidades"};

SQLiteDatabase db = base.getReadableDatabase();

//Query a todos los campos y ordenados por autor
c = db.query("libro",campos,null,null,null,null,"autor");
```

Por otra parte, y al igual que la actividad de buscar, el usuario puede interactuar con la pantalla mediante pulsaciones cortas y largas con los resultado descritos en el apartado anterior.

4.6.3. Borrar un registro

La eliminación de un registro se realizará mediante el acceso al menú contextual sobre la consulta que se desee. Es común para ambas actividades y se realiza con el método eliminarRegistro(**int** position).

```
public void eliminarRegistro(int position){

    SQLiteDatabase db = base.getWritableDatabase();
    db.delete("libro", BaseColumns._ID + "=" + position, null);
    db.close();

    message = new StringBuilder().append(position).toString();
    ejecuta_cliente task = new ejecuta_cliente();
    task.execute(message);

    mostrarLibros();
}
```

A continuación se ejecuta el envío de la información actualizada al servidor remoto mediante AsynTask, creando un String con la ID a borrar como único dato.

4.6.4. Detalles

Es una actividad a la que se puede acceder desde Buscar.java y desde Consultar.java. Aquí solamente se observan todos los campos detallados de cada registro, el usuario no tiene ningún tipo de interacción con ellos. Véase Figura 4.6.

Figura 4.6: Interfaz donde se muestran los detalles del registro

No obstante, desde el menú se podrá acceder a la edición de algunos de ellos.

4.7. Edición de registros existentes

En la edición de registros se muestran todos los campos editables y los valores obtenidos en las consultas previas. Se permite al usuario la edición de los campos bajo restricciones de formato. Véase Figura 4.7.

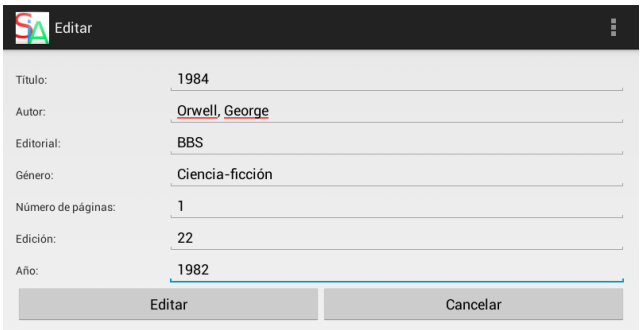

Figura 4.7: Interfaz de edición

Una vez la edición se completa y es correcta se enviará al servidor.

```
//Se recogen los valores y se formatean
String titulo_esp = String.valueOf(titulo.getText());
titulo_esp = titulo_esp.replace(" ", "BLANKSPACE");
int año_esp = Integer.parseInt(año.getText().toString());

//Abrimos la DB como escritura
SQLiteDatabase db=base.getWritableDatabase();

//Se crea un ContentValues y se rellena.
ContentValues valores = new ContentValues();

//Con el espacio reemplazado
valores.put(LibrosColumns.TITULO, titulo_esp);
valores.put(LibrosColumns.AUTOR, autor_esp);
valores.put(LibrosColumns.EDITORIAL, editorial_esp);
valores.put(LibrosColumns.GENERO, genero_esp);
valores.put(LibrosColumns.EDICION, edicion_esp);
valores.put(LibrosColumns.NUM_PAG, num_pag_esp);
valores.put(LibrosColumns.AÑO, año_esp);


int row = Integer.parseInt(idem);

db.update("libro", valores, BaseColumns._ID + "=" + row,
 null);

db.close();


message = new StringBuilder().append(row).append(" ")
.append(titulo_esp).append(" ").append(autor_esp).
append(" ").append(editorial_esp).append(" ").append(
genero_esp).append(" ").append(edicion_esp).append(" ")
.append(num_pag_esp).append(" ").append(año_esp).
toString();

ejecuta_cliente task = new ejecuta_cliente();
task.execute(message);
```

El mensaje enviado al servidor tendrá 8 campos que serán: ID, título, autor, editorial, género, número de páginas, edición y año.

4.8. Añadir nuevos registros

Para agregar un nuevo registro se accede desde el menú en cualquiera de las actividades. Esto nos abrirá una nueva pantalla con todos los campos en blanco. Véase Figura 4.8.

Figura 4.8: Interfaz para añadir nuevos registros.

Para añadir el código de barras, se puede escribir a mano o si se presiona el botón *Scan* se abrirá el programa Barcode Scanner para escanear el código automáticamente.

Por otro lado, con el fin de que un registro se agregue correctamente se deben rellenar todos los campos, y además, con el fin de evitar registros duplicados se hará una consulta a la base de datos mediante el método *duplicados(String, String, String)* que devolverá si el registro ya forma parte de la base de datos o no. En caso negativo se procederá a la introducción del mismo y a la notificación al servidor.

```
//Buscar duplicados al añadir
public String duplicados(String cod_barras, String
titulo_dup, String autor_dup){

        SQLiteDatabase db = base.getReadableDatabase();
        ArrayList<Libros> libros = new ArrayList<Libros>();
        Cursor c;

        String[] campos = new String[] {"cod_barras",
        "titulo", "autor"};

        String[] args = new String[] {cod_barras};
```

```
        c = db.query("libro", campos, "cod_barras=?", args,
        null, null, null);

        c.moveToFirst();
        if (c.moveToFirst()) {
            do {

                libros.add(new Libros(c.getInt(c.getColumnIndex(
                LibrosColumns.COD_BARRAS)),c.getString(c.
                getColumnIndex(LibrosColumns.TITULO)),c.
                getString(c.getColumnIndex(LibrosColumns.AUTOR))));

            } while(c.moveToNext());
        }

        //Se comprueba que el Arraylist no este vacío.
        //Si lo esta es que no hay coincidencias.
        if (libros.size() == 0) {

            //CLEAN no se han encontrado coincidencias
            result = String.valueOf("CLEAN");


        } else {

        //El código de barras es único, no se comprueba más
        result = String.valueOf("FOUND");

        }

    libros.clear();
    c.moveToFirst();
    return result;
}
```

El mensaje tendrá un total de 10 elementos siendo estos: Código de barras, ID, título, autor, editorial, género, número de páginas, edición, año y unidades.

4.9. Conexión entre dispositivos

La conexión entre dispositivos se realiza mediante sockets y con un servidor Java de mediador. El objetivo es tener la base de datos de forma local

en cada dispositivo para que el acceso a la misma sea más rápido y a la vez mantener todas las bases de datos del resto de dispositivos conectados al servidor actualizados. En la figura 4.9 el dispositivo 1 realiza un cambio en su base de datos y mediante la clase *ejecuta_ cliente* realizará la conexión y el consiguiente envío de información.

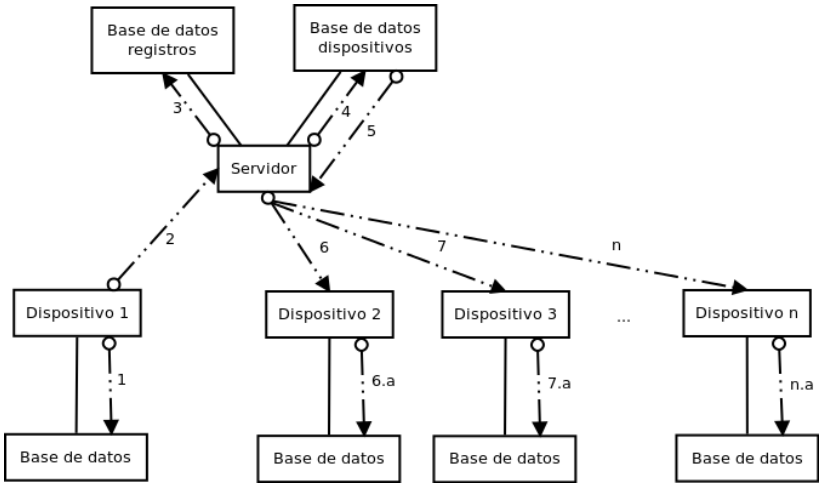

Figura 4.9: Esquema de la conexión.

El servidor actualizará su base de datos propia, consultará en otra base de datos los dispositivos conectados y les enviará de forma progresiva la información actualizada. Para ello contamos con la tarea en segundo plano *Second.java*, descrita en el apartado 4.4.1, que estará en constante escucha.

Todas las actividades donde se manipula de una forma u otra la base de datos disponen de un socket cliente que se pondrá en contacto con el servidor y notificará los cambios. Esta será la clase *ejecuta_ cliente* que es una extensión de AsynTask para no bloquear la interfaz gráfica.

```
private class ejecuta_cliente extends AsyncTask<Void,
String, Void>{

    @Override
    protected Void doInBackground(Void... params) {

        String android_id = Secure.getString(
        getApplicationContext().getContentResolver(),
        Secure.ANDROID_ID);

        message = new StringBuilder().append(android_id)
        .append(" ").append("null").toString();
```

```
        try {

            final SharedPreferences settings =
            getSharedPreferences("SIA_preferencias",
             MODE_PRIVATE);

              String ip_serv = settings.getString(
              "IpServidorJava","192.168.1.64");

            //Conecta con el servidor
            client = new Socket(ip_serv, 5555);

            printwriter = new PrintWriter(client.
            getOutputStream(), true);

            printwriter.write(message);
           printwriter.flush();
            printwriter.close();

            client.close();    //Cierra el socket

           } catch (UnknownHostException e) {
              e.printStackTrace();
            } catch (IOException e) {
              e.printStackTrace();
              //No se encuentra el servidor
              publishProgress("No se encuentra
              el servidor");
            }
        return null;
     }
    @Override
    protected void onProgressUpdate(String... progress) {
         String No_Server = progress[0];

         Toast.makeText(getApplicationContext(), No_Server,
          Toast.LENGTH_LONG).show();
    }
}
```

La diferencia entre los *ejecuta_cliente()* de cada actividad reside en el mensaje que el método envía.

4.10. Clases auxiliares

Se han definido dos clases auxiliares globales para toda la aplicación.

- **LibrosColumns.java** define el nombre las columnas y es una extensión de *BaseColumns*. Si en un momento determinado se necesitará cambiar el nombre de alguna de ellas, bastaría con hacerlo aquí.
- **Libros.java** describe métodos para el acceso rápido de los registros una vez se ha accedido a la base de datos y permite una fácil extracción de los valores determinados de cada columna definida.

4.11. Servidor Java remoto

Para poder crear la conexión entre los dispositivos ha sido necesario implementar un servidor Java. La implementación es muy similar a la de la aplicación, aunque en este caso no es necesario definir ninguna IP, ya que es la aplicación Android la que se pone en contacto inicialmente con el servidor y este la guarda para un uso futuro en un ArrayList con todas las IP de todos los dispositivos conectados.

En la figura 4.10 se puede ver esquematizado el funcionamiento del servidor Java.

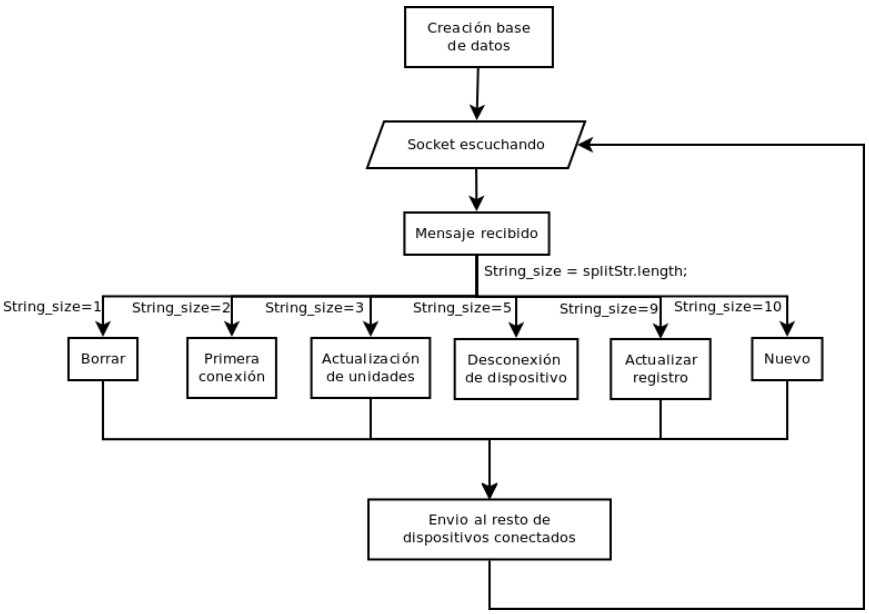

Figura 4.10: Diagrama de flujo del servidor Java.

Por lo tanto, al igual que la aplicación, se mantiene un puerto de escucha continuamente abierto a la espera de alguna conexión. Cuando esta se realiza, y según el tamaño del ArrayString recibido, se realizará una acción u otra equivalente a lo visto en el servicio Second.java de la aplicación Android.

```
//Divido el String en array String por espacios
String[] splitStr = message.split("\\s+");

//Recupero la IP y el puerto
ip_socket = clientSocket.getRemoteSocketAddress().toString();

//Aislo la IP
ip_socket2 = ip_socket.substring(1);
String[] ip = ip_socket2.split(":");

int String_size = splitStr.length;
```

Se clasifican de la siguiente forma:

- **Tamaño 1.** Borrar un registro.
- **Tamaño 2.** Nuevo dispositivo.
- **Tamaño 3.** Actualización de las unidades.
- **Tamaño 5.** Un dispositivo se desconecta y se elimina del ArrayList.
- **Tamaño 8.** Edición de un registro.
- **Tamaño 10.** Introducción de un nuevo registro.

Esta clasificación se realiza para la actualización interna de la base de datos del servidor que no es compatible con la de la aplicación, aunque se puede consultar con cualquier programa escrito para ello. Para ello se ha utilizado la librería *sqlite jdbc*.

4.11.1. Creación de la base de datos

Al iniciar la aplicación se llamará a la clase *DBAccess()* que comprobará si las tablas están creadas. En caso negativo se crearán las dos bases de datos necesarias. La primera será *dispositivo* donde se llevará un registro de cada dispositivo conectado y la última fecha de modificación. La segunda es *libro*, que será la base de datos donde se guardará todos los registros.

```
public class DBAccess{
```

```
    public static void main() throws ClassNotFoundException,
    SQLException, ParseException{

        Class.forName("org.sqlite.JDBC");
        Connection conn = DriverManager.getConnection("jdbc:
        sqlite:/home/marina/Documentos/java_server.db");

        DatabaseMetaData dbm = conn.getMetaData();

        // Comprobar si dispositivos existe, sino se crea.
        ResultSet tables = dbm.getTables(null, null,
        "DISPOSITIVO", null);

        if (tables.next()) {

            System.out.println("La tabla ya existe");


        } else {

            conn.prepareStatement("create table dispositivo(
            android_id,ip,fecha)").execute();

            conn.prepareStatement("create table libro(id,titulo,
            autor,    editorial,num_paginas,genero,edicion,año,
            cod_barras,unidades,fecha)").execute();

            System.out.println("Tabla creada");

        }


        tables.close();
        conn.close();
    }

}
```

Cabe destacar que en cada una de las conexiones que realicemos a la base de datos, se debe dar la ruta completa de la base de datos local del ordenador, en este caso:

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:
/home/marina/Documentos/java_server.db");
```

4.11.2. Registro de dispositivos

El registro de dispositivos se realiza cuando se obtiene un tamaño de *String* de 2. Anteriormente se habrá creo un *ArrayList* llamado *dispositivos* donde se va a almacenar los dispositivos activos en esta sesión. Para ello se recorrera el ArrayList en busca de coincidencias de ip, si no halla alguna se agregará.

Por otra parte, se extraerá la última fecha de modificación que llevó a cabo este dispositivo mediante el método *get_ Old_ Date()*:

```
public static String get_Old_Date(String[] splitStr) throws
ParseException, ClassNotFoundException, SQLException{


        Class.forName("org.sqlite.JDBC");
        Connection conn = DriverManager.getConnection("jdbc:
        sqlite:/home/marina/Documentos/java_server.db");

        conn.setAutoCommit(false);
        PreparedStatement ps = conn.prepareStatement("SELECT
        fecha FROM dispositivo WHERE android_id =?");

        ps.setString(1, splitStr[0]);

        ResultSet rs = ps.executeQuery();

        resultado = "01/01/2000 00:00:00";

        while(rs.next()) {

            resultado = rs.getObject(1).toString();

             }

        rs.close();
        ps.close();
        conn.close();

        return resultado;

}
```

A continuación se procederá a la actualización de la tabla *dispositivo*, donde se hará una consulta sobre la identidad Android:

```
PreparedStatement ps = conn.prepareStatement("SELECT *
```

```
FROM dispositivo WHERE android_id =?");

ps.setString(1, splitStr[0]);

ResultSet rs = ps.executeQuery();
```

Si se hallan coincidencias se actualizará la fecha.

```
String ip_last = splitStr[1];
String nom = dateNow;

ps = conn.prepareStatement("update dispositivo set
fecha='" +nom+ "' where android_id =?");

ps.setString(1, splitStr[0]);
ps.executeUpdate();

ps = conn.prepareStatement("update dispositivo set
ip='" +ip_last+ "' where android_id =?");

ps.setString(1, splitStr[0]);
ps.executeUpdate();
ps.close();

conn.close();
System.out.println("Actualizado");
```

Si por el contrario no las hay, se añadirá el dispositivo a la tabla.

```
ps = conn.prepareStatement("insert into dispositivo
values(?,?,?)");

ps.setString(1, splitStr[0]);
ps.setString(2, splitStr[1]);
ps.setString(3, dateNow);
ps.addBatch();
ps.executeBatch();
conn.commit();

System.out.println("Añadido");

ps.close();
conn.close();
```

La variable *dateNow* se obtiene al lanzar *get_Date()*, que obtendrá la fecha actual y la formateará como "dd/MM/yyyy HH:mm:ss".

```
public static String get_Date() throws ParseException{
```

```
    Calendar currentDate = Calendar.getInstance();

    SimpleDateFormat sdf = new SimpleDateFormat("
    dd/MM/yyyy HH:mm:ss");

    dateNow = sdf.format(currentDate.getTime());

    return dateNow;

}
```

A continuación se debe comprobar si el dispositivo tiene actualizaciones pendientes de la base de datos desde su última conexión. Para ello lo que se hace es tomar la fecha obtenida con *get_Old_Date()* y comparar la ultima fecha de modificación de cada registro. Si la fecha del registro es más nueva que la de la última conexión del dispositivo, se añade a un *ArrayString*.

```
private static String value;
ArrayList<String> mensajes = new ArrayList<String>();

PreparedStatement ps = conn.prepareStatement("SELECT * FROM
libro");

ResultSet rs = ps.executeQuery();

int nuevos=0;
int size=0;
while(rs.next()) {

    nuevos = DispositivosDB.compare_Date(resultado, rs.
    getObject(11).toString());

   //Comparar fechas y añadir

    if (nuevos==1) {

      //Fecha guardada más antigua

      size = mensajes.size();

      value = new StringBuilder().append(rs.getObject(1).
      toString()).toString();

        for (int i=2; i<11; i++){
```

```
            //Hasta 11 porque no envio la fecha
            //Se van agregando todos los campos
            value = new StringBuilder().append(value).append
            (" ").append(rs.getObject(i).toString()).toString();

        }
            mensajes.add(size, value);
    }
```

La comparación de fechas se realiza mediante *Date.after(Date when)*. Finalmente se envía String a String cada actualización mediante el método de cliente-servidor por sockets utilizado a lo largo de toda la aplicación. El tamaño final del String con la información debe ser 10 y se enviarán todos los datos excepto la fecha de modificación.

4.11.3. Borrar un registro

Se produce cuando se recibe un String de tamaño 1. Como no queremos eliminar por completo todo el registro, para poder saber cuando se ha eliminado, se colocara la palabra "VACIO" en todos los campos excepto en el ID y en la fecha de modificación.

```
ps = conn.prepareStatement("DELETE FROM libro
WHERE id = ? " );

ps.setString(1, args);

ps.executeUpdate();

String splitStr[] = {"VACIO","VACIO","VACIO","VACIO",
"VACIO","VACIO","VACIO","VACIO","VACIO",args};

NuevoDB.main(splitStr);
```

4.11.4. Actualización de registro

Al recibir un mensaje de tamaño 3 se procederá a actualizar las unidades y la fecha según el ID recibido.

```
ps = conn.prepareStatement("update libro set
unidades='" +splitStr[2]+ "' where id =?");

ps.setString(1, splitStr[0]);
ps.executeUpdate();
```

```
ps = conn.prepareStatement("update libro set
fecha='" +NewDate+ "' where id =?");

ps.setString(1, splitStr[0]);
ps.executeUpdate();
```

Si por el contrario el mensaje recibido tiene tamaño 8, el proceso es análogo a la actualización de unidades, excepto que se realiza sobre todos los campos.

4.11.5. Nuevo registro

En principio se recibe un String con 10 campos, ya que necesitamos añadir el ID del registro a la base de datos del servidor, puesto que al contrario que con la base de datos que gestiona Android, en la base de datos del servidor el ID no es un campo auto-incremental.

```
ps = conn.prepareStatement("insert into libro
values(?,?,?,?,?,?,?,?,?,?,?)");

ps.setString(1, splitStr[9]); //ID
ps.setString(2, splitStr[1]); //Titulo
ps.setString(3, splitStr[2]); //Autor
ps.setString(4, splitStr[3]); //Editorial
ps.setString(5, splitStr[5]); //Número páginas
ps.setString(6, splitStr[4]); //Genero
ps.setString(7, splitStr[6]); //Edición
ps.setString(8, splitStr[7]); //Año
ps.setString(9, splitStr[0]); //Código de barras
ps.setString(10, splitStr[8]); //Unidades
ps.setString(11, NewDate); //Fecha
ps.addBatch();
ps.executeBatch();
conn.commit();

System.out.println("Añadido");

ps.close();
conn.close();
```

Por ello antes de reenviar el mensaje al resto de dispositivos es necesario reconstruir el String con 9 campos, eliminando el ID.

4.11.6. Envío al resto de dispositivos

A la hora de repetir la información al resto de dispositivos, primero comprobamos que el dispositivo que envía la información no sea el único dispositivos conectado y en caso negativo se va leyendo el ArrayList con todas las IP del resto de dispositivos y se envía la información.

```
for( int i = 1 ; i < size; i++ ){

    if (dispositivos.get(i).equals(ip[0])){

        //No se hace nada

        System.out.println("No hay más dispositivos
        conectados");

      }else{

        clientSocket = new Socket(dispositivos.get(i),
         4444);

        PrintWriter salida = new PrintWriter(new
        OutputStreamWriter(clientSocket.getOutputStream()),true);

        System.out.println(clientSocket.
        getRemoteSocketAddress().toString());

        salida.println(message);

        clientSocket.close();
        System.out.println("Envio a otros dispositivos");
        DeleteDB.actualiza(dispositivos.get(i));
      }
}
```

Además cada vez que se envía una actualización a un dispositivo, se actualizará a su vez la última fecha de modificación en la tabla *dispositivo* por si acaso hubiera un corte de conexión repentino entre el servidor y el dispositivo y este último no pudiera enviar el mensaje de notificación de desconexión.

4.11.7. Salida de dispositivos

Cuando un dispositivo sale de la aplicación notifica al servidor su marcha con un mensaje de tamaño 5. El servidor deberá borrar ese dispositivo del

array donde conserva las IPs para reenviar los mensajes.

```
dispositivos.remove(i);
```

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

En el proyecto se ha demostrado que la programación para dispositivos con Android prácticamente no tiene límites y se ha llevado la utilidad al campo de la productividad empresarial.

La aplicación permite el uso de forma local para un único dispositivo, así como de varios dispositivos conectados por Wi-Fi a un servidor remoto dentro de la misma red. Para las pruebas se ha utilizado un punto de acceso sin conexión a internet lo cual permite a su vez que la conexión sea segura al no tener contacto con el exterior, a pesar de ser vulnerable a ataques en el radio de la cobertura Wi-Fi disponible. Se recomienda que antes de utilizar la aplicación se decida si el uso va a ser remoto o local, puesto que los cambios que se realicen en los dispositivos sin estar conectados al servidor no se enviarán y podrían provocar conflictos internos.

El modelo utilizado de cliente-servidor es entre dispositivos Android y un servidor Java, ya que a día de hoy el consumo de batería de los dispositivos no hubiera permitido el uso continuado de un servidor con Android, a pesar de que hubiera simplificado el código de forma notable, y la utilización de un dispositivo virtual no es una opción adecuada debido a la lentitud de procesado, a pesar de que el rendimiento de la aplicación en los dispositivos utilizados es bueno teniendo en cuenta que la aplicación esta enfocada a dispositivos de gama baja.

Para la lectura del código de barras se ha utilizado un programa externo. El manejo y procesado de imágenes en Android es complejo y se ha aprovechado una aplicación de código abierto para simplificar el código. Además es una parte importante de la aplicación y el hecho de utilizar *Barcode Scanner* permite tener las ultimas mejoras y correcciones en nuestro

terminal realizadas por expertos.

El entorno de trabajo no ha simplificado la tarea de familiarización y desafortunadamente el hecho de instalarlo sobre Ubuntu lo ha dificultado más. Han sido necesarias varias instalaciones para conseguir el correcto funcionamiento de todas las herramientas y los servidores de descarga son muy lentos, por lo que esta parte del trabajo ha resultado tediosa.

En cuanto a la base de datos, cuando se introducía un registro con algún espacio no había ningún problema si se mostraba todo el contenido de la base de datos, no obstante si se hacia una búsqueda sobre ese elemento solo se buscaba la parte correspondiente hasta el primer espacio dando lugar a resultados erróneos e incompletos. Esto se solucionó reemplazando los espacios por una palabra comodín antes de su introducción a la base de datos.

Cuando se decidió introducir el servidor remoto fue necesario implementar la tarea en segundo plano que se mantuviera en constante escucha. Crear un *service*, una actividad sin interfaz gráfica, no era suficiente puesto que la aplicación se quedaba colgada. Fue necesario extender el método a Asyntask. Cuando todo se implementó correctamente se descubrió que AsynTask únicamente funcionaba una vez, por lo que fue necesario a su vez incluir un handler para gestionar distintos hilos del método.

Otro problema con el servidor remoto fue que la base de datos creada con *jdbc* no era compatible con la de Android. En principio la idea era que cada vez que un dispositivo se conectara, se le enviara la base de datos del servidor que él mismo actualizaba. Esto evitaba tener que llevar un registro de salida de cada dispositivo. Como consecuencia de ello, el desarrollo del servidor se volvió más complejo y la respuesta entre los dispositivos y el servidor más lenta.

5.2. Líneas futuras

A pesar de todas las precauciones tomadas a lo largo del desarrollo de la aplicación, sería interesante tener en cuenta las siguientes líneas que podría adoptar el futuro desarrollo de la misma.

5.2.1. Integración total de lectura del código de barras

El hecho de incluir el programa *Barcode Scanner* de esta manera en la aplicación nos permitirá tener las últimas actualizaciones del mismo. No obstante, obliga al usuario a descargar una aplicación adicional.

Una de las futuras mejoras sería la integración completa de una aplicación

de lectura de código de barras, que permitiera integrarla en la interfaz propia de la aplicación. Además se podría escoger el uso de la cámara frontal del dispositivo en caso de contar también con cámara trasera, ya que ahora mismo no esta soportado.

5.2.2. Mayor flexibilidad local-remoto

En el punto de desarrollo en el que se encuentra la aplicación se recomienda utilizarla exclusivamente de forma local o remota y no compaginar ambas, a no ser que de forma local solo se utilice a modo de consulta.

Esto se debe a que en la aplicación no hay implementado ningún método que permita conocer al servidor y por lo tanto al resto de dispositivos los posibles cambios realizados por otro dispositivo cuando se encontraba *offline*.

La implementación de esto trae consigo nuevos problemas a tener en cuenta, como por ejemplo que varias instrucciones se contradigan, por lo tanto sería un punto a estudiar detenidamente.

Además no existe modo de comprobar si la transferencia de datos se ha realizado con éxito por lo que si se produce un error, no hay forma de saberlo si no se repasan los logs del servidor.

5.2.3. Envío de la base de datos

En muchas ocasiones y por mucho cuidado que se tenga, la informática puede jugar malas pasadas. Es por ello que sería interesante implementar una función que pidiera a petición del usuario la base de datos al servidor. Para ello seria necesario guardar una copia compatible y actualizada en el servidor. No obstante, la falta de compatibilidad entre la base de datos creada en Android y la que crea el conector Lava *jdbc* no permite con facilidad este intercambio.

Además, la transferencia inicial en la cual se actualizan los datos puede llevar un tiempo considerable según el uso de cada dispositivo, ya que la transferencia se hace registro a registro. Sería interesante que a partir de cierto número de registros a actualizar se enviara directamente la base de datos actualizada.

5.2.4. Ampliación de tipos de registro

En un principio la aplicación esta diseñada para el control de libros, lo cual limita de forma radical el uso de la aplicación. Una futura línea de trabajo sería que el usuario pudiera escoger entre varios tipos de registro, como CDs o DVDs o incluso crear las columnas y la temática que deseara.

Para ello se haría uso de la clase auxiliar LibrosColumns.java que permite el uso del nombre de las columnas en todas las funciones únicamente definiendo en esta clase el nombre de cada una de ellas.

5.2.5. NFC

La cámara de fotos lleva integrándose en dispositivos portátiles casi 10 años y sigue teniendo problemas por ejemplo, a la hora de enfocar. La lectura de código de barras que se realiza en la aplicación en ocasiones se vuelve tediosa debido a ello.

La reciente aparición de NFC (*Near Field Communication*) que es una variación del RFID (*Radio-frequency identification*), permite el intercambio de información de forma rápida y eficaz. El problema es que hoy en día el número de dispositivos con esta tecnología es muy escaso y normalmente suelen ser terminales de gama alta-media. Además sería necesario disponer de un elemento pasivo que contuviera la información de cada registro físicamente, lo cual supone un gasto extra y que solo sería rentable si el precio del objeto fuera considerable y valiera la pena invertir en la etiqueta NFC.

El NFC en la actualidad se esta utilizando en la vida cotidiana, por ejemplo es las tarjetas de transporte urbano e incluso tarjetas de crédito. Consiste en la comunicación de dos dispositivos a una distancia casi inexistente en la que el dispositivo activo genera por inducción un campo magnético en el que el elemento pasivo y el activo intercambian información, pudiendo llegar a los 300Mbits/s.

Apéndice A

Lista de permisos Android Manifest

Objeto	Constantes	Descripción
String	ACCESS_CHECKIN_PROPERTIES	Allows read/write access to the "properties"table in the checkin database, to change values that get uploaded.
String	ACCESS_COARSE_LOCATION	Allows an app to access approximate location derived from network location sources such as cell towers and Wi-Fi.
String	ACCESS_FINE_LOCATION	Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
String	ACCESS_LOCATION_EXTRA_COMMANDS	Allows an application to access extra location provider commands
String	ACCESS_MOCK_LOCATION	Allows an application to create mock location providers for testing
String	ACCESS_NETWORK_STATE	Allows applications to access information about networks
String	ACCESS_SURFACE_FLINGER	Allows an application to use SurfaceFlinger's low level features.
String	ACCESS_WIFI_STATE	Allows applications to access information about Wi-Fi networks
String	ACCOUNT_MANAGER	Allows applications to call into AccountAuthenticators.
String	ADD_VOICEMAIL	Allows an application to add voicemails into the system.
String	AUTHENTICATE_ACCOUNTS	Allows an application to act as an AccountAuthenticator for the AccountManager

String	BATTERY_STATS	Allows an application to collect battery statistics
String	BIND_ACCESSIBILITY_SERVICE	Must be required by an AccessibilityService, to ensure that only the system can bind to it.
String	BIND_APPWIDGET	Allows an application to tell the AppWidget service which application can access AppWidget's data.
String	BIND_DEVICE_ADMIN	Must be required by device administration receiver, to ensure that only the system can interact with it.
String	BIND_INPUT_METHOD	Must be required by an InputMethodService, to ensure that only the system can bind to it.
String	BIND_NOTIFICATION_LISTENER_SERVICE	Must be required by an NotificationListenerService, to ensure that only the system can bind to it.
String	BIND_REMOTEVIEWS	Must be required by a RemoteViewsService, to ensure that only the system can bind to it.
String	BIND_TEXT_SERVICE	Must be required by a TextService
String	BIND_VPN_SERVICE	Must be required by an VpnService, to ensure that only the system can bind to it.
String	BIND_WALLPAPER	Must be required by a WallpaperService, to ensure that only the system can bind to it.
String	BLUETOOTH	Allows applications to connect to paired bluetooth devices
String	BLUETOOTH_ADMIN	Allows applications to discover and pair bluetooth devices
String	BRICK	Required to be able to disable the device (very dangerous!).
String	BROADCAST_PACKAGE_REMOVED	Allows an application to broadcast a notification that an application package has been removed.
String	BROADCAST_SMS	Allows an application to broadcast an SMS receipt notification.
String	BROADCAST_STICKY	Allows an application to broadcast sticky intents.

String	BROADCAST_WAP_PUSH	Allows an application to broadcast a WAP PUSH receipt notification.
String	CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed.
String	CALL_PRIVILEGED	Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.
String	CAMERA	Required to be able to access the camera device.
String	CHANGE_COMPONENT_ENABLED_STATE	Allows an application to change whether an application component (other than its own) is enabled or not.
String	CHANGE_CONFIGURATION	Allows an application to modify the current configuration, such as locale.
String	CHANGE_NETWORK_STATE	Allows applications to change network connectivity state
String	CHANGE_WIFI_MULTICAST_STATE	Allows applications to enter Wi-Fi Multicast mode
String	CHANGE_WIFI_STATE	Allows applications to change Wi-Fi connectivity state
String	CLEAR_APP_CACHE	Allows an application to clear the caches of all installed applications on the device.
String	CLEAR_APP_USER_DATA	Allows an application to clear user data.
String	CONTROL_LOCATION_UPDATES	Allows enabling/disabling location update notifications from the radio.
String	DELETE_CACHE_FILES	Allows an application to delete cache files.
String	DELETE_PACKAGES	Allows an application to delete packages.
String	DEVICE_POWER	Allows low-level access to power management.
String	DIAGNOSTIC	Allows applications to RW to diagnostic resources.
String	DISABLE_KEYGUARD	Allows applications to disable the keyguard
String	DUMP	Allows an application to retrieve state dump information from system services.

String	EXPAND_STATUS_BAR	Allows an application to expand or collapse the status bar.
String	FACTORY_TEST	Run as a manufacturer test application, running as the root user.
String	FLASHLIGHT	Allows access to the flashlight
String	FORCE_BACK	Allows an application to force a BACK operation on whatever is the top activity.
String	GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service
String	GET_PACKAGE_SIZE	Allows an application to find out the space used by any package.
String	GET_TASKS	Allows an application to get information about the currently or recently running tasks.
String	GET_TOP_ACTIVITY_INFO	Allows an application to retrieve private information about the current top activity, such as any assist context it can provide.
String	GLOBAL_SEARCH	This permission can be used on content providers to allow the global search system to access their data.
String	HARDWARE_TEST	Allows access to hardware peripherals.
String	INJECT_EVENTS	Allows an application to inject user events (keys, touch, trackball) into the event stream and deliver them to ANY window.
String	INSTALL_LOCATION_PROVIDER	Allows an application to install a location provider into the Location Manager.
String	INSTALL_PACKAGES	Allows an application to install packages.
String	INTERNAL_SYSTEM_WINDOW	Allows an application to open windows that are for use by parts of the system user interface.
String	INTERNET	Allows applications to open network sockets.
String	KILL_BACKGROUND_PROCESSES	Allows an application to call killBackgroundProcesses(String).
String	LOCATION_HARDWARE	Allows an application to use location features in hardware, such as the geofencing api.

String	MANAGE_ACCOUNTS	Allows an application to manage the list of accounts in the AccountManager
String	MANAGE_APP_TOKENS	Allows an application to manage (create, destroy, Z-order) application tokens in the window manager.
String	MASTER_CLEAR	Not for use by third-party applications.
String	MODIFY_AUDIO_SETTINGS	Allows an application to modify global audio settings
String	MODIFY_PHONE_STATE	Allows modification of the telephony state - power on, mmi, etc.
String	MOUNT_FORMAT_FILESYSTEMS	Allows formatting file systems for removable storage.
String	MOUNT_UNMOUNT_FILESYSTEMS	Allows mounting and unmounting file systems for removable storage.
String	NFC	Allows applications to perform I/O operations over NFC
String	PERSISTENT_ACTIVITY	This constant was deprecated in API level 9. This functionality will be removed in the future; please do not use. Allow an application to make its activities persistent.
String	PROCESS_OUTGOING_CALLS	Allows an application to monitor, modify, or abort outgoing calls.
String	READ_CALENDAR	Allows an application to read the user's calendar data.
String	READ_CALL_LOG	Allows an application to read the user's call log.
String	READ_CONTACTS	Allows an application to read the user's contacts data.
String	READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
String	READ_FRAME_BUFFER	Allows an application to take screen shots and more generally get access to the frame buffer data.
String	READ_HISTORY_BOOKMARKS	Allows an application to read (but not write) the user's browsing history and bookmarks.
String	READ_INPUT_STATE	This constant was deprecated in API level 16. The API that used this permission has been removed.

String	READ_LOGS	Allows an application to read the low-level system log files.
String	READ_PHONE_STATE	Allows read only access to phone state.
String	READ_PROFILE	Allows an application to read the user's personal profile data.
String	READ_SMS	Allows an application to read SMS messages.
String	READ_SOCIAL_STREAM	Allows an application to read from the user's social stream.
String	READ_SYNC_SETTINGS	Allows applications to read the sync settings
String	READ_SYNC_STATS	Allows applications to read the sync stats
String	READ_USER_DICTIONARY	Allows an application to read the user dictionary.
String	REBOOT	Required to be able to reboot the device.
String	RECEIVE_BOOT_COMPLETED	Allows an application to receive the ACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.
String	RECEIVE_MMS	Allows an application to monitor incoming MMS messages, to record or perform processing on them.
String	RECEIVE_SMS	Allows an application to monitor incoming SMS messages, to record or perform processing on them.
String	RECEIVE_WAP_PUSH	Allows an application to monitor incoming WAP push messages.
String	RECORD_AUDIO	Allows an application to record audio
String	REORDER_TASKS	Allows an application to change the Z-order of tasks
String	RESTART_PACKAGES	This constant was deprecated in API level 8. The restartPackage(String) API is no longer supported.
String	SEND_RESPOND_VIA_MESSAGE	Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls.
String	SEND_SMS	Allows an application to send SMS messages.

String	SET_ACTIVITY_WATCHER	Allows an application to watch and control how activities are started globally in the system.
String	SET_ALARM	Allows an application to broadcast an Intent to set an alarm for the user.
String	SET_ALWAYS_FINISH	Allows an application to control whether activities are immediately finished when put in the background.
String	SET_ANIMATION_SCALE	Modify the global animation scaling factor.
String	SET_DEBUG_APP	Configure an application for debugging.
String	SET_ORIENTATION	Allows low-level access to setting the orientation (actually rotation) of the screen.
String	SET_POINTER_SPEED	Allows low-level access to setting the pointer speed.
String	SET_PREFERRED_APPLICATIONS	This constant was deprecated in API level 7. No longer useful, see addPackageToPreferred(String) for details.
String	SET_PROCESS_LIMIT	Allows an application to set the maximum number of (not needed) application processes that can be running.
String	SET_TIME	Allows applications to set the system time.
String	SET_TIME_ZONE	Allows applications to set the system time zone
String	SET_WALLPAPER	Allows applications to set the wallpaper
String	SET_WALLPAPER_HINTS	Allows applications to set the wallpaper hints
String	SIGNAL_PERSISTENT_PROCESSES	Allow an application to request that a signal be sent to all persistent processes.
String	STATUS_BAR	Allows an application to open, close, or disable the status bar and its icons.
String	SUBSCRIBED_FEEDS_READ	Allows an application to allow access the subscribed feeds ContentProvider.
String	SUBSCRIBED_FEEDS_WRITE	
String	SYSTEM_ALERT_WINDOW	Allows an application to open windows using the type TYPE_SYSTEM_ALERT, shown on top of all other applications.

String	UPDATE_DEVICE_STATS	Allows an application to update device statistics.
String	USE_CREDENTIALS	Allows an application to request authtokens from the AccountManager
String	USE_SIP	Allows an application to use SIP service
String	VIBRATE	Allows access to the vibrator
String	WAKE_LOCK	Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming
String	WRITE_APN_SETTINGS	Allows applications to write the apn settings.
String	WRITE_CALENDAR	Allows an application to write (but not read) the user's calendar data.
String	WRITE_CALL_LOG	Allows an application to write (but not read) the user's contacts data.
String	WRITE_CONTACTS	Allows an application to write (but not read) the user's contacts data.
String	WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.
String	WRITE_GSERVICES	Allows an application to modify the Google service map.
String	WRITE_HISTORY_BOOKMARKS	Allows an application to write (but not read) the user's browsing history and bookmarks.
String	WRITE_PROFILE	Allows an application to write (but not read) the user's personal profile data.
String	WRITE_SECURE_SETTINGS	Allows an application to read or write the secure system settings.
String	WRITE_SETTINGS	Allows an application to read or write the system settings.
String	WRITE_SMS	Allows an application to write SMS messages.
String	WRITE_SOCIAL_STREAM	Allows an application to write (but not read) the user's social stream data.
String	WRITE_SYNC_SETTINGS	Allows applications to write the sync settings
String	WRITE_USER_DICTIONARY	Allows an application to write to the user dictionary.

Apéndice B

Especificaciones de los dispositivos de testeo

	Asus MeMO Pad Smart 10”	**Tablet 7 - 4G - 1**
Sistema operativo	Android 4.2.1	Android 2.1
Pantalla	Pantalla 10.1” LED WXGA 1280x800, Capacitiva multitáctil 10 puntos	Pantalla 7” 800x400. Resitiva.
CPU	NVIDIA Tegra3 Quad-Core, 1.2 GHz	Telechips 600MHz
Memoria	1GB	512MB
Almacenamiento	16GB eMMC	4GB
Conectividad inalámbrica	WLAN 802.11 a/b/g/n, Bluetooth V3.0+EDR +A2DP	WLAN 802.11 a/b/g, Bluetooth V2
Cámara	Cámara frontal de 1.2 MP, Cámara trasera de 5 MP	N/A
Interfaz	1 x Micro USB, 1 x Micro HDMI, 1 Auriculares / Micrófono, 1 x Micro SD	1 x Mini USB, 1 x Micro HDMI, 1 Auriculares / Micrófono, 1 x Micro SD
Sensores	G-Sensor, Giroscopio, E-compass, Sensor de luz ambiental, GPS	G-Sensor
Dimensiones	263 x 180.8 x 9.9 mm	186 x 115 x 14 mm
Peso	580 g	341 g

Bibliografía

TOMÁS GIRONÉS, J (2012). *El Gran Libro de Android.* Marcombo; Edición: Segunda.

BURNETTE, E. (2010). *Hello Android.* The Pragmatic Programmers; Edición: Tercera.

HASEMAN, C. (2008). *Android Essentials.* Apress; Edición: Primera.

MURPHY, M.L. (2010). *Android Programming Tutorials.* CommonsWare; Edición: Primera.

MURPHY, M.L. (2009). *Beginning Android.* Apress; Edición: Primera.

GÓMEZ, S. *Curso Programación Android, 2013. http://www.sgoliver.net/* (20 de abril de 2013)

VÉLIZ, J. *Base de Datos SQLite3 para Android por Eclipse. <http://coderwar.com/2012/05/base-de-datos-sqlite3-para-android-por-eclipse/>* (15 de enero de 2013)

HERNÁNDEZ SALINAS, G. *Cómo instalar SQLITE en Eclipse IDE. <http://baro3495.blogspot.com.es/2012/09/como-instalar-sqlite-en-eclipse-ide.html/>* (17 de julio de 2013)

Cool2k. *HowTo: Realizar consultas a una base de datos sqlite desde java. <http://cool2k.wordpress.com/2009/06/05/howto-realizar-consultas-a-una-base-de-datos-sqlite-desde-java/>* (17 de julio de 2013)

ZXING. *Multi-format 1D/2D barcode image processing <http://code.google.com/p/zxing/>* (20 de abril de 2013)