Android: Definiciones Básicas y Desarrollo de Aplicaciones

Federico Cristina, Sebastián Dapoto, Fernando G. Tinetti[1]

III-LIDI, Facultad de Informática, UNLP
[1]Comisión de Inv. Científicas, Prov. de Buenos Aires

Reporte Técnico UNLP-PD-2012-1
Agosto 2012

Tabla de Contenidos

1 - Introducción

“Android es un sistema operativo móvil basado en Linux, que junto con aplicaciones middleware está enfocado para ser utilizado en dispositivos móviles como teléfonos inteligentes, tabletas, Google TV y otros dispositivos. Es desarrollado por la Open Handset Alliance, la cual es liderada por Google. “

http://es.wikipedia.org/wiki/Android

Originalmente diseñado para arquitectura ARM, actualmente hay ports no oficiales tanto para x86 como para MIPS. El código fuente se encuentra disponible bajo licencias de software libre (Apache o GNU General Public License versión 2). El mismo es generalmente liberado luego de cada nueva versión.

1.1 - Versiones (API Levels)

Existen distintas versiones de Android: las líneas 1.xx, 2.xx, 3.xx y 4.xx. La primera de éstas ya sin actualizaciones desde 2009. La línea 2.xx es la utilizada en dispositivos móviles, mientras que la 3.xx fue pensada específicamente para tablets.

Debido al avance en la capacidad de procesamiento de dispositivos móviles (y más específicamente en el caso de los teléfonos inteligentes), la línea 4.xx unifica su uso para cualquier dispositivo.

La Fig. 1 muestra la cantidad de dispositivos con Android por versión del sistema operativo, de acuerdo a la información provista por Open Signal Maps en Abril de 2012.

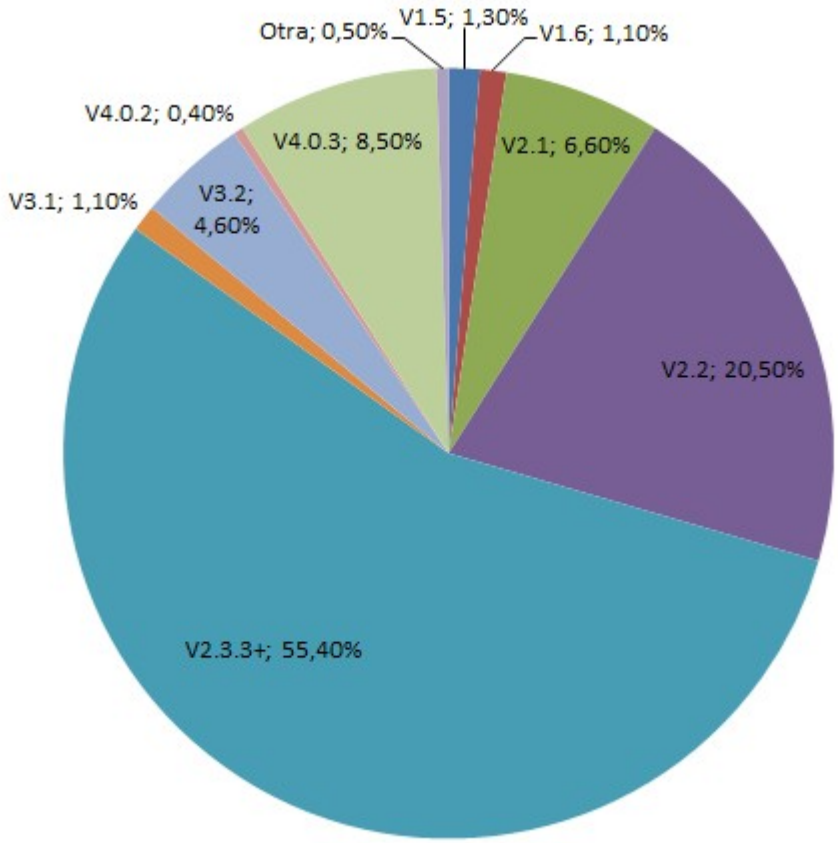

Figura 1. Distribución de Android según su versión. Información correspondiente al mes de Abril de 2012 según Open Signal Maps. http://opensignalmaps.com/reports/fragmentation.php

En lo que respecta a las políticas de actualización, las mismas pueden llegar a variar dependiendo varios factores, tales como fabricante del dispositivo, proveedora del servicio de telefonía móvil, etc. En ocasiones, las actualizaciones originalmente estipuladas luego no se cumplen por motivos externos a Google. Por ejemplo, el dispositivo Samsung Galaxy SII (originalmente con Android 2.3.3) tiene planificada su actualización a versión 4; la cual ya se efectivizó para equipos liberados. Sin embargo la actualización para dispositivos bajo carriers todavía se encuentra pendiente dado que cada compañía usualmente realiza una *adaptación* definitiva de la versión. En algunos casos, la actualización nunca llega de manera automática, sino que que es necesario "rootear" (término que se explica a continuación) el aparato e instalar una versión más actual de forma manual.

1.1.1 - Root en Android

Configurar un dispositivo Android en modo root permite tener un control completo del sistema. Los pasos para realizar esta tarea varía en función de cada dispositivo, pero en términos generales es necesario conectar el equipo a una PC, configurar Android en modo depuración

USB, reiniciar el equipo en modo descarga y ejecutar un programa específco desde una PC (como por ejemplo Odin) para efectivizar los cambios sobre el equipo.

1.2 - Incidencia en el Mercado

Actualmente Android es la plataforma que lidera el *market share* (cantidad de dispositivos con Android), por encima de iOS, Symbian OS, RIM OS y Windows Mobile. La Fig. 2 muestra los porcentajes de comercialización de teléfonos inteligentes según el sistema operativo instalado a partir del año 2007 hasta el 2011 según la información proporcionada en http://en.wikipedia.org/ wiki/Mobile_operating_system.

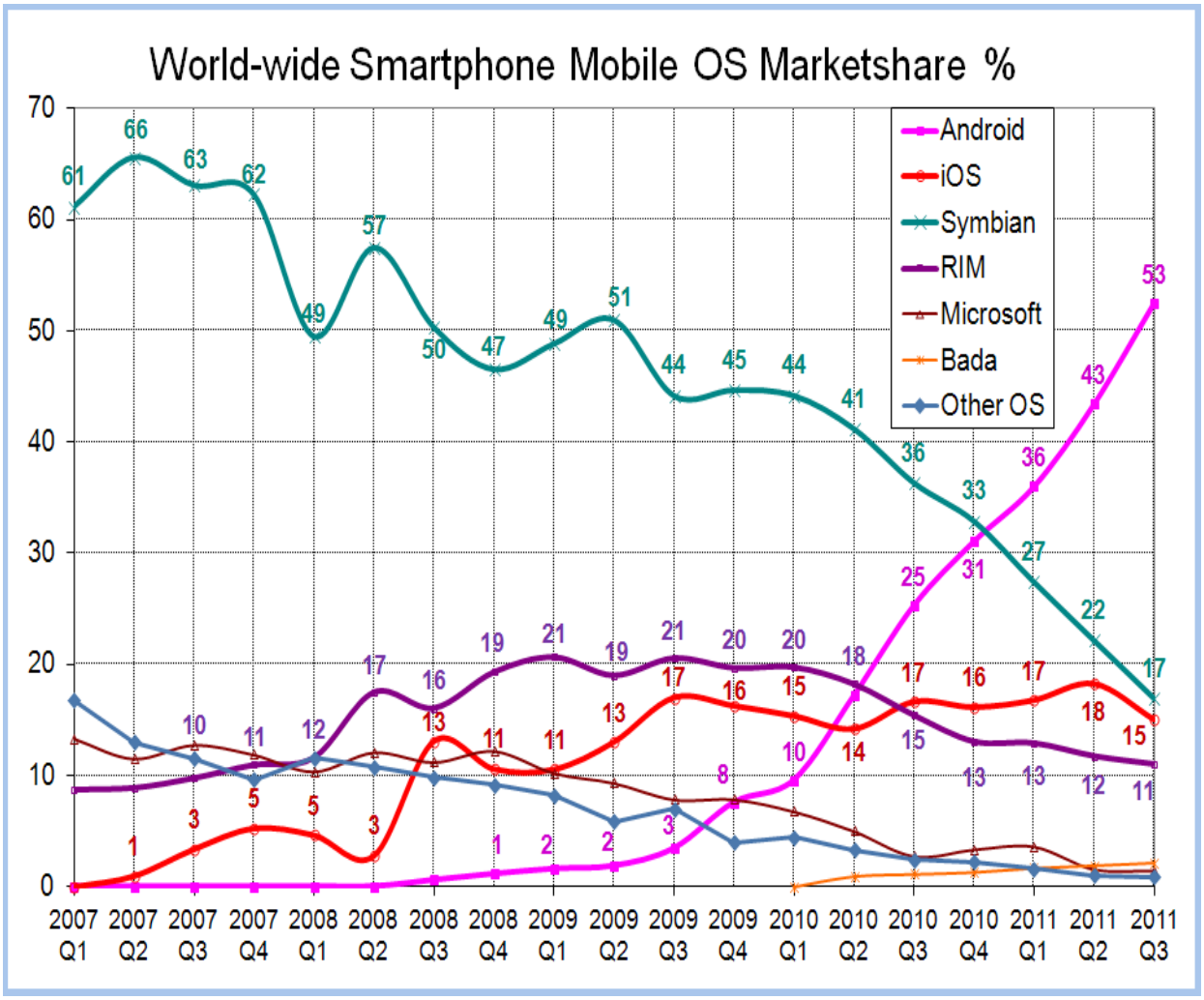

Figura 2. Evolución de ventas de dispositivos según su Sistema Operativo, según http://en.wikipedia.org/wiki/Mobile_operating_system (contemplando solo *smartphones*).

2 - Arquitectura

Como sistema operativo, Android está organizado en capas y bibliotecas con diferentes funcionalidades, servicios e interacciones con las demás partes o subsistemas del sistema operativo.

2.1 - Dalvik Virtual Machine (DVM)

El kernel de Android se encuentra basado en el de Linux, con librerías escritas en C, y aplicaciones corriendo sobre un framework de soporte para la máquina virtual Dalvik (DVM: Dalvik Virtual Machine) con compilación *Just-In-Time* (JIT), con traducción de *bytecodes* de Java (JIT a partir de Android 2.2).

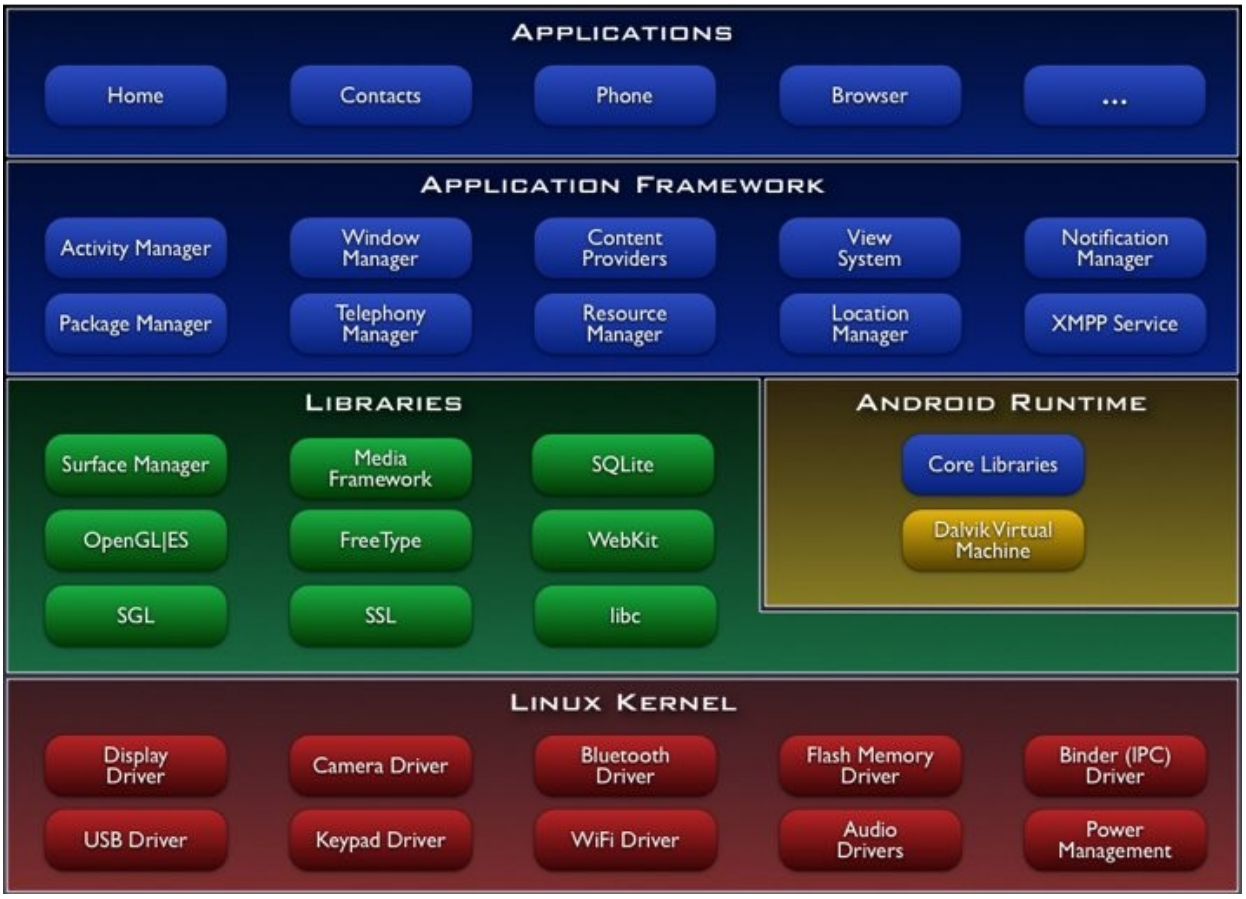

Figura 3. Arquitectura de componentes que conforman Android.
http://developer.android.com/guide/basics/what-is-android.html

La Fig. 3 muestra la organización en capas de Android. La capa de más bajo nivel o más cercana al hardware es la basada en Linux y directamente sobre ella se podría decir que se

ubica la capa de “traducción” a lo que es propio de Android, donde se pueden identificar dos partes en esta capa:

- Bibliotecas: algunas de las cuales, quizás identificables como las clásicas de Linux, tal como libc y otras más *propias* o *adaptadas para* Android, como por ejemplo *Media Framework*.
- Android Runtime: es quizás la parte de esta segunda capa más específica de Android y diferenciada de lo que estaría a este nivel en un sistema operativo Linux, que contiene las bibliotecas más importantes y la DVM.

Ya las capas superiores son totalmente específicas de Android, aunque por supuesto no son esencialmente diferentes de lo que se define en un sistema operativo en capas de nivel de abstracción similares.

2.2 - Diferencias entre JVM vs DVM

Aunque desde el punto de vista de la aplicación no hay muchas diferencias entre la JVM y la DVM, arquitecturalmente ambas difieren de manera considerable. Sintéticamente, el código fuente Java es compilado a un archivo .class. Posteriormente, dicho archivo es procesado por *dexer* (una herramienta que es parte de la SDK de Android), a fin de generar el archivo con formato DEX que contiene los bytecodes Dalvik. Esto significa que la DVM sería similar a la JVM en términos prácticos (o *actuaría como tal*), gracias a la compilación *Just-In-Time* de bytecodes de Java a código ejecutable Dalvik o dex-code (*Dalvik Executable*).

Más propiamente, DVM es una máquina virtual de proceso tal como la JVM, con el agregado de tener un JIT para archivos de clase (.class) de Java o producidos por un compilador Java para una JVM estándar. La Tabla 1 muestra algunas de las diferencias más importantes entre la DVM y la JVM [1] [2].

	DVM	**JVM**
Arquitectura	Registros	Pila
Soporte S.O.	Android	Múltiples
Clases	Dex	Class
Pool de constantes	Por aplicación	Por clase
Aplicación	APK	JAR

Tabla 1. Diferencias entre DVM y JVM.

Referencias:

[1] http://www.momob.in/2010/general/bhavis/difference-of-dvm-and-jvm-in-the-android-world/
[2] http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html

3 - Desarrollo de Aplicaciones para Android en Java

Existen varias alternativas para desarrollar aplicaciones Android: en Eclipse desde una PC, o bien en AIDE desde una PC o directamente en un dispositivo móvil. A continuación se detallan los pasos para la creación y configuración de las mismas.

3.1 - Requisitos para Desarrollar desde Eclipse

El desarrollo desde Eclipse es el mecanismo más tradicional. Para ésto, es necesario contar con una serie de herramientas:

- Java Development Kit (JDK), actualmente versión 1.6 o superior
- Eclipse 3.5 o superior. También es posible utilizar NetBeans
- Android Standard Development Kit (SDK)
- Plugin Android Developer Tools (ADT) para Eclipse
- Uno o más dispositivos para ejecutar la aplicación. Las alternativas disponibles son:
    - Dispositivo real
    - Android Virtual Devices (AVD)
    - Android-x86

3.1.1 - Instalación y configuración

Siempre es recomendable seguir los pasos indicados en el sitio de desarrolladores Android (http://developer.android.com), debido a que las versiones van cambiando a lo largo del tiempo. En términos generales los pasos son los siguientes:

1. Descargar la JDK desde el site de Oracle e instalar.
2. Descargar Eclipse.
3. Descargar la SDK de Android e instalar, lo cual también nos facilitará las opciones para descargar los AVDs (la descarga de los AVDs requerirá un tiempo considerable).
4. Instalar el plugin ADT desde Eclipse (Help → Install New Software).

Más informacion en: http://developer.android.com/sdk/index.html

Una vez que están todos los componentes instalados, ya podremos crear aplicaciones y ejecutarlas en un dispositivo.

3.1.2 - Alternativas de depuración y ejecución

Como se detalló anteriormente, existen tres alternativas para depurar en tiempo real las aplicaciones desarrolladas: mediante un dispositivo real o dos alternativas de emulación, que

se detallan en los siguientes apartados.

3.1.2.1 - Dispositivo real

Idealmente se debe contar con un dispositivo real. Obviamente ésta es la mejor alternativa, dado que los emuladores presentan ciertos faltantes en sus funcionalidades (por ejemplo no cuentan con soporte de emulación WiFi), además que la performance se encuentra considerablemente degradada.

3.1.2.2 - Android Virtual Device

Para crear un AVD desde Eclipse se debe seleccionar la opción "**Window → AVD Manager...**" y luego la opción "**New...**". En la ventana de creación es necesario completar el nombre del nuevo dispositivo y las características con las que contará el mismo, tales como nivel de la API, tamaño de la SD Card, resolución de la pantalla, etc. (Ver Fig. 4). Una vez creado, se puede visualizar el nuevo dispositivo en la lista de AVDs disponibles, tal como se observa en la Fig. EE.

A su vez, es posible seleccionar sobre qué dispositivo se ejecutará una aplicación. Para realizar esto se debe elegir la opción "**Run → Run Configurations...**" y luego entrar en la pestaña "**Target**", en donde se puede visualizar la lista de dispositivos compatibles con la aplicación a ejecutar. Notar que si nuestra aplicación está configurada por ejemplo para una versión mínima del SDK igual a 9, entonces no se visualizarán los dispositivos que tienen una versión anterior a ésta.

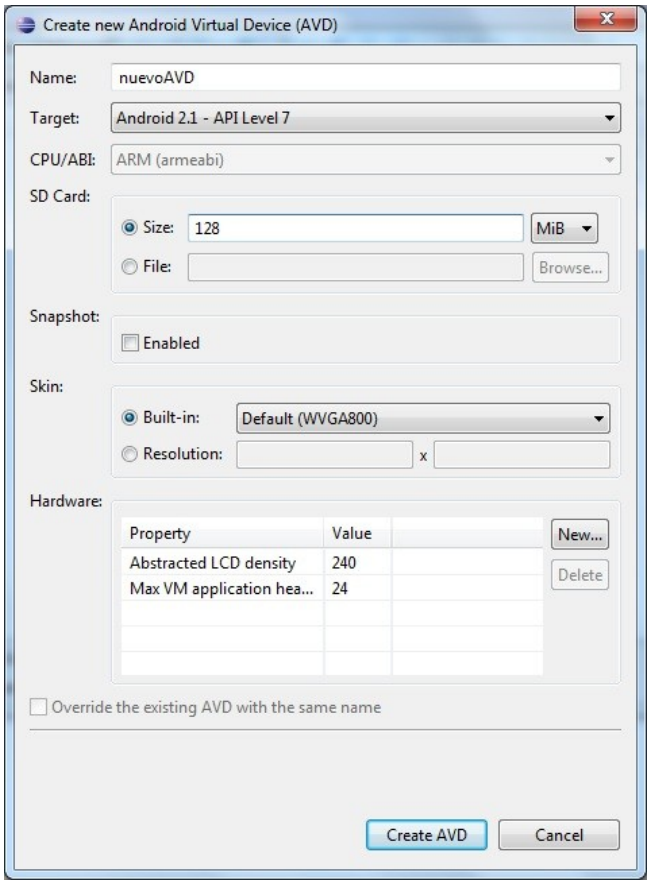

Figura 4. Creación de un AVD en Eclipse.

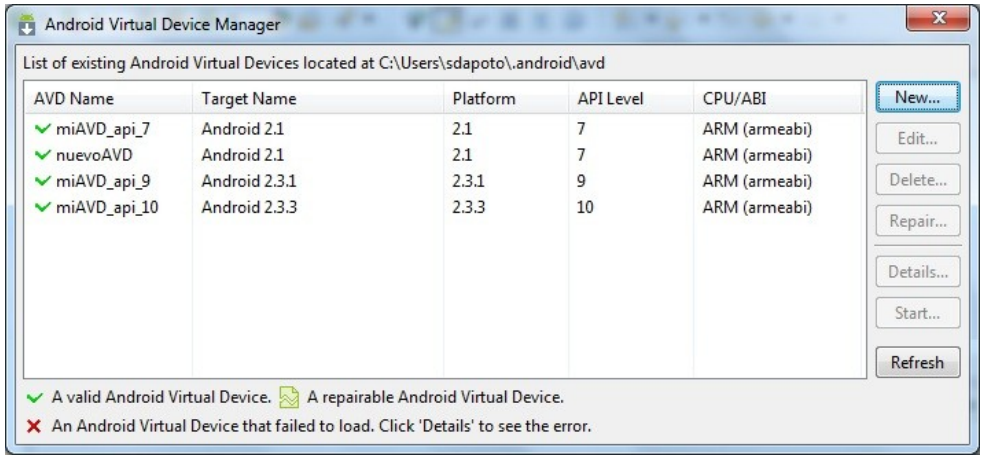

Figura 5. Lista de AVDs disponibles.

3.1.2.3 - Android-x86

Para instalar la plataforma Android-x86 se deben seguir una serie de pasos que se enumeran a continuación.

1. Descargar e instalar VirtualBox, el cual servirá para instalar Android x86 en una máquina virtual.
2. Descargar Android x86. Existen varias versiones de Android que se pueden utilizar. La versión 2.2 de Android x86 es suficiente.
3. Crear una nueva máquina virtual e instalar Android x86 como Sistema Operativo (http://androidspin.com/2011/01/24/howto-install-android-x86-2-2-in-virtualbox/)
4. Configurar como "**puente**" el adaptador de red de la máquina virtual. Esto es necesario para que al dispositivo se le asigne un IP real y entonces sea posible conectarse al mismo para ser utilizado desde Eclipse.

Una vez que estemos ejecutando la máquina virtual con Android x86, se debe realizar la conexión al dispositivo. Esto se lleva a cabo desde línea de comando haciendo uso de la herramienta **Android Debug Bridge (ADB)**. Esta herramienta se encuentra en el subdirectorio 'platform-tools' del SDK (ej: C:\ruta_SDK\platform-tools). Para realizar la conexión se debe ejecutar el siguiente comando:

> adb connect IPdispositivo

Para averiguar el IP que le fue asignado a nuestra máquina virtual, es útil el comando '**netcfg**' que se puede usar desde la consola de android. Para entrar a la consola se debe presionar las teclas **<alt+F1>** y para salir las teclas **<alt+F7>**.

En cuanto el dispositivo virtual esté conectado aparecerá dentro de la "lista de dispositivos en ejecución" en el entorno Eclipse, pudiendo ser elegido para depurar nuestras aplicaciones.

3.2 - Requisitos para Desarrollar desde AIDE

AIDE es un entorno de desarrollo Android para Android, el cual permite la creación de aplicaciones directamente en el dispositivo móvil. Aunque las características del mismo son un tanto reducidas (no provee un editor visual de ventanas, ni un debugger en tiempo real), es una alternativa simple e interesante para dar los primeros pasos.

3.2.1 - Instalación y configuración de AIDE en un equipo móvil

Los pasos para esta alternativa se reducen simplemente a acceder al Market y descargar e instalar la aplicación.

3.2.2 - Instalación y configuración de AIDE desde PC

Alternativamente, podemos instalar AIDE en una PC. Esta solución puede ser útil en caso de no contar con un dispositivo móvil, o bien si queremos utilizar un teclado físico en caso que el

dispositivo con el que contamos no lo provee.

Para llevar a cabo la instalación se debe primeramente instalar una máquina virtual con el sistema operativo Android x86 siguiendo los pasos vistos en la sección 3.1.2.3. Una vez que estemos ejecutando Android x86 se procede con la instalación de AIDE. Hacerlo desde el market en la máquina virtual puede traer complicaciones, con lo cual es más fácil descargar el APK de AIDE desde el navegador de Android, alojado en apktops, e instalarlo. (http://www.apktops.com/aide-android-java-ide-1-0-beta16.html).

4 - Desarrollo de Aplicaciones para Android en C/C++

Además de desarrollo en Java, es posible implementar partes del código en C/C++. Para ésto, es necesario utilizar la Android NDK o Native Development Kit:
http://developer.android.com/sdk/ndk/index.html
similar a la JNI de Java:
http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html.

Dicha NDK es una herramienta complementaria a la SDK que permite desarrollar partes de una aplicación - idealmente las críticas en cuanto a performance - en código nativo. La misma provee los encabezados y librerías que permiten crear *activities*, manejar periféricos, acceder a recursos de otros aplicaciones, etc. programando en C/C++. El modelo fundamental de aplicación Android no cambia. Incluso en estos casos, la aplicación final será empaquetada en un archivo .apk y correrá dentro de una máquina virtual en el dispositivo.

Vale destacar que con la utilización de código nativo se pierde la portabilidad y no resulta en una mejora de la performance de manera directa, dependiendo en gran parte del contexto de la aplicación. Algunos ejemplos donde puede ser necesario desarrollar código nativo son: tiempo real, procesamiento de señales, simulaciones, etc. Este tema excede la finalidad del presente documento.

5 - Ciclo de Vida de Aplicaciones

La Fig. 6 presenta el gráfico del ciclo de vida general de una aplicación Android, en donde la clase inicial de la misma extiende de la clase *Activity*.

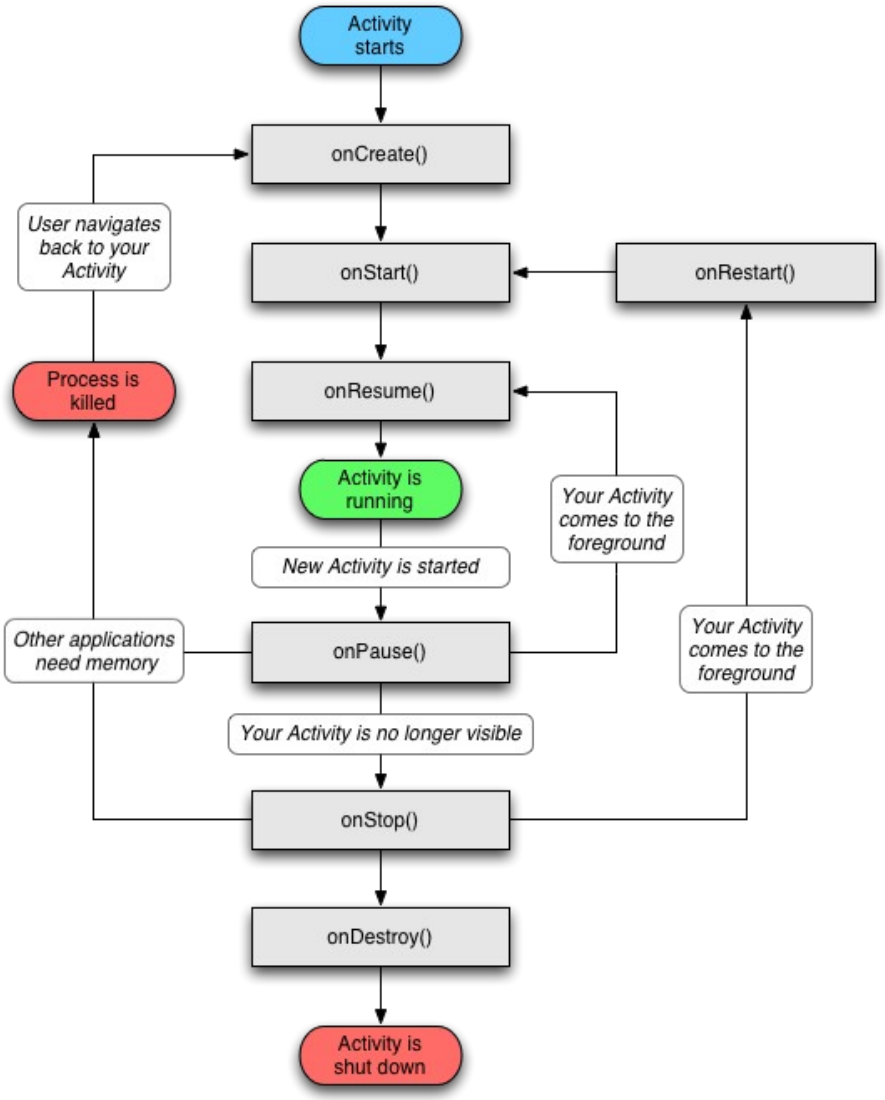

Figura 6. Ciclo de Vida http://www.androidjavadoc.com/1.0_r1_src/android/app/Activity.html

Los eventos definidos por la clase **Activity** son los siguientes:

- onCreate(): invocado cuando la actividad es creada por primera vez.
- onStart(): invocado cuando la actividad se hace visible para el usuario.
- onResume(): invocado cuando la actividad comienza a interactuar con el usuario.
- onPause(): invocado cuando la actividad actual se pausa y la actividad anterior se reanuda.
- onStop(): invocado cuando la actividad ya no es visible para el usuario.

- onDestroy(): invocado antes de que el sistema destruya la actividad (ya sea de manera manual o por el sistema para conservar memoria).
- onRestart(): invocado cuando la actividad ha sido detenida y se está reiniciando.

Cada uno de estos eventos es invocado a fin de poder ejecutar la lógica en cuestión en el momento correspondiente. De todas maneras, no siempre es necesario redefinir todos los métodos. De manera simplificada, podría ser suficiente la implementación de los métodos ***onCreate()***, ***onResume()***, ***onPause()***, aunque esto variará según las necesidades en cada caso.

6 - Ejemplo: Desarrollo de una Aplicación Android en Eclipse

6.1 - Creación y configuración inicial del proyecto

A continuación se enumeran los pasos necesarios para crear y configurar un proyecto android en el entorno Eclipse. A modo de ejemplo se crea el proyecto de la aplicación “Hello World” que se implementará mas adelante.

1. Seleccionar “**File** → **New** → **Project…**”. Una vez dentro del menú, elegir la opción “**Android Project**” dentro de la carpeta de Android. (Fig. 7)

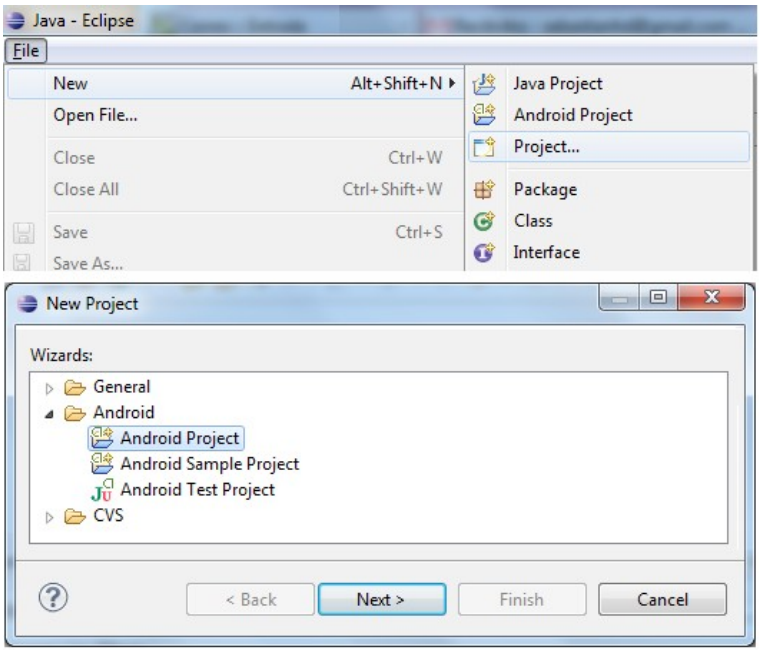

Figura 7. Creación de un nuevo proyecto Android.

Nota: durante el tiempo de elaboración de este reporte, la imagen y las opciones de selección del tipo de proyecto cambiaron. Es muy probable que durante los siguientes meses vuelvan a cambiar, pero conceptualmente las opciones de creación de proyectos se mantienen básicamente invariantes.

2. Como se observa en la Fig. 8, es necesario completar una serie de datos para la configuración inicial del proyecto (entre paréntesis se indican los valores cargados en el ejemplo, aunque pueden variar según sea necesario):

- Nombre al proyecto (“**HelloWorld**”).
- Versión de Android a utilizar (**2.3.1**).
- Nombre de la aplicación (“**HelloWorld**”).
- Nombre del package. La convención recomendada es utilizar el nombre del dominio en orden inverso, seguido del nombre del proyecto

(“**com.programaciondistribuida.HelloWorld**”)

- Nombre de la actividad principal (“**MainActivity**”)
- Versión mínima del API (“**9**”)

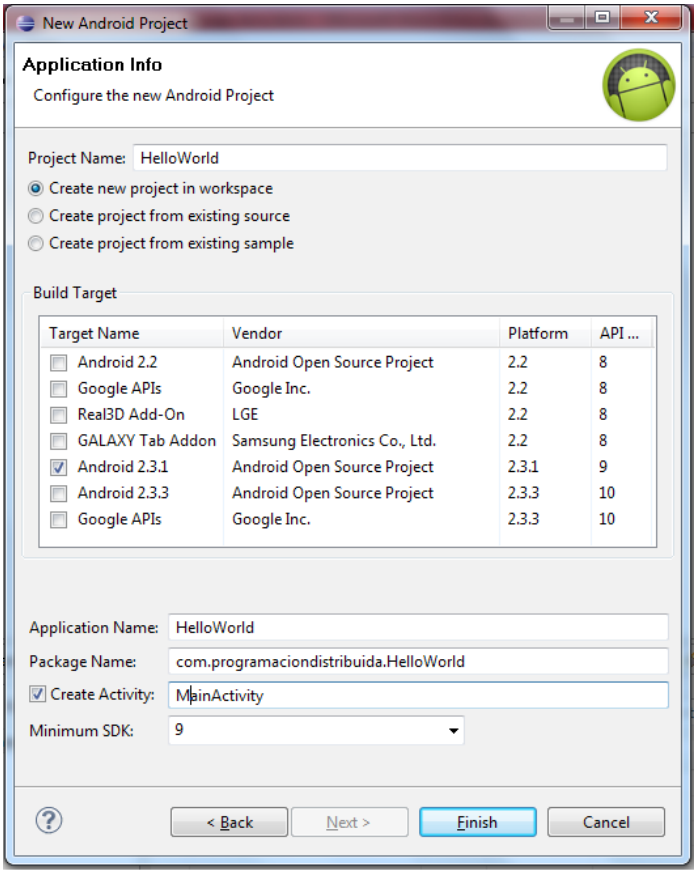

Figura 8. Configuración inicial del proyecto Android.

3. Al presionar sobre el botón “**Finish**” se creará la estructura de la aplicación.

6.2 - Componentes principales

En este momento ya es posible visualizar la estructura de nuestra aplicación android. La misma cuenta con un conjunto de carpetas y archivos que se detallarán a continuación:

- **src**: contiene los archivos de código fuente (.java) del proyecto. En nuestro ejemplo existe el archivo “MainActivity.java”.
- **gen**: contiene el archivo “R.java”, autogenerado por el compilador.
- **Android 2.3.1**: contiene la librería de android 2.3.1 (“android.jar”).
- **assets**: contiene todos los “assets” usados por la aplicación, tales como HTML, archivos de texto, bases de datos, etc.
- **bin**: contiene las clases compiladas (.class) de nuestra aplicación.
- **res**: contiene los recursos usados por la aplicación. Posee las siguientes subcarpetas:

    - **drawable-hdpi, drawable-mdpi, drawable-ldpi**: contiene imagenes tales como los iconos de la aplicación en diferentes resoluciones (high, medium, low respectivamente).
    - **layout**: contiene el archivo “main.xml” en donde se encuentra especificada la interfaz de usuario (UI) en formato XML.
    - **values**: contiene el archivo “strings.xml”. El mismo sirve para definir todos los strings que contenga nuestra aplicación, para luego ser referenciados. Es recomendable utilizarlo, ya que si en algún momento se necesita traducir la aplicación a otro idioma, solo es necesario reemplazar el archivo strings.xml con otro equivalente en el nuevo idioma.
- **AndroidManifest.xml**: este es el archivo de manifiesto de nuestra aplicación. En el mismo se especifica los permisos requeridos por la aplicación, su orientación (*landscape* o *portrait*), el API mínimo necesario, la clase principal, etc..
- **proguard.cfg**: archivo de configuración autogenerado por ProGuard para optimizar/ ofuscar el código de la aplicación. No se debe modificar.
- **project.properties**: archivo de configuración autogenerado por Android Tools. No se debe modificar.

6.3 - Implementación de “Hello World”

Continuando con el proyecto creado anteriormente, se detalla el código a implementar en cada componente para finalizar la aplicación “Hello World”.

- **AndroidManifest.xml**: En este archivo se indica el package, la versión de la aplicación, su nombre, y la clase encargada de iniciar la aplicación (main launcher). Los valores indicados con arroba referencian a otros archivos (por ejemplo @string/app_name referencia al archivo ***strings.xml***).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.programaciondistribuida.HelloWorld"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="9" />
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

- **main.xml**: es posible crear la interfaz gráfica desde código, o bien generar un archivo XML donde se defina la misma. Para esta última opción se puede utilizar el *Graphical Layout*, el cual es una herramienta visual de diseño, parte del ADT. En este ejemplo se definieron dos *TextView* y un *Button* (notar que no es obligatorio el uso de ***strings.xml***).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
      <TextView
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/hello" />
      <TextView
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Esta es la aplicación de prueba Hello World" />
</LinearLayout>
```

- **strings.xml**: Este archivo almacenará los textos en formato clave/valor.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, MainActivity!</string>
    <string name="app_name">HelloWorld</string>
</resources>
```

- **MainActivity.java**: Ampliaremos el método onCreate() a fin de incorporar dinámicamente un botón, el cual además presentará un mensaje al ser clickeado:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Button aButton = new Button(getBaseContext());
    aButton.setText("Y este es un botón de prueba");
    aButton.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
         Toast.makeText(getBaseContext(), "Clicked", Toast.LENGTH_LONG).show();
        }
    });
    LinearLayout layout = (LinearLayout)findViewById(R.id.layout);
    layout.addView(aButton);
}
```

Al ejecutar la aplicación en el emulador veremos una pantalla similar a la de la Fig. 9. La Fig. 10 muestra el resultado de clickear el botón de la aplicación.

Figura 9. Ejecución de la aplicación

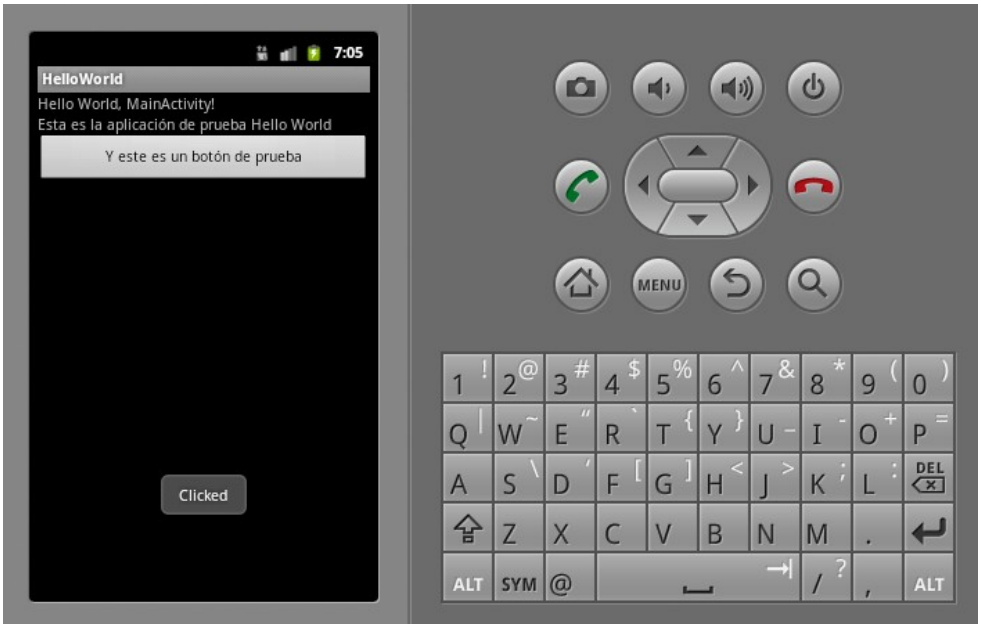

Figura 10. Mensaje al clickear el botón

Al ejecutar la aplicación en el emulador veremos una pantalla similar a la de la Fig. 9. La Fig. 10 muestra el resultado de clickear el botón de la aplicación.

6.4 - Instalación Mediante APK

Alternativamente a la ejecución de la aplicación desde Eclipse, es posible realizar una instalación tradicional de ésta mediante el formato de archivo APK.

Primeramente se debe generar el APK definitivo de la aplicación. Desde Eclipse, se debe acceder a **"File → Export → Export Android Application"**, y completar la información requerida (Fig. 11). Una vez finalizados los pasos de exportación, se contará con el APK listo para ser instalado en el dispositivo.

Para realizar la instalación del APK solo se necesita poder acceder al mismo y descargarlo desde donde esté alojado (web server, mail, dropbox, etc.).

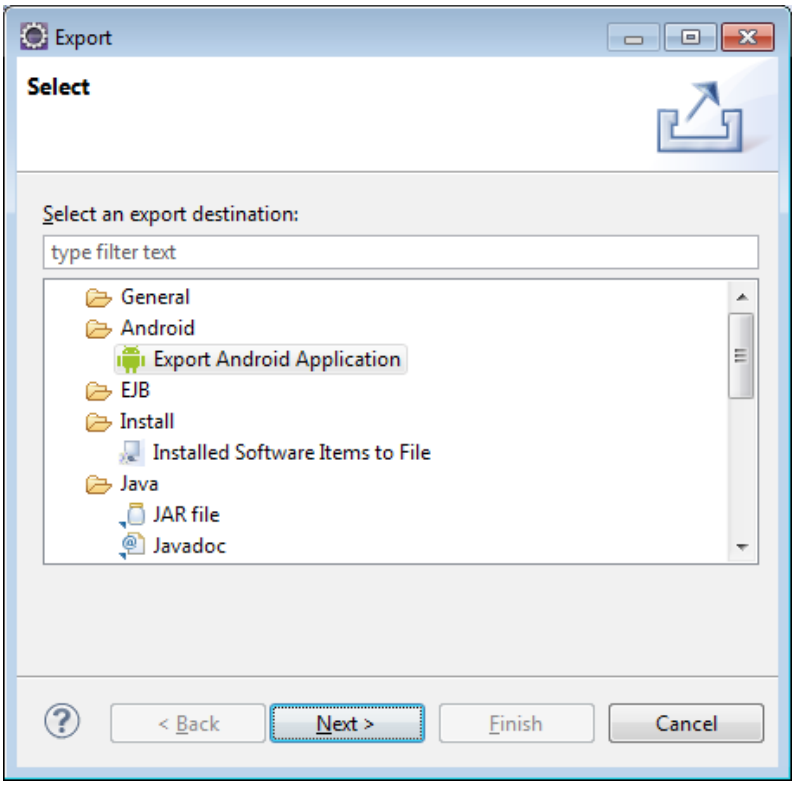

Figura 11. Generación de APK de instalación.

7 - Comunicación entre Dispositivos

7.1 - Sockets en Java

Jave provee las clases principales para la comunicación mediante Sockets:

- TCP
  - **java.net.ServerSocket:** Esta clase implementa sockets de tipo servidor, el cual espera *requests* a través de la red.
  - **java.net.Socket:** Esta clase implementa sockets de tipo cliente, el cual realiza *requests* hacia un ServerSocket.
  - **java.io.InputStream, java.io.OutputStream:** Usadas para el envío de datos entre el cliente y el servidor (y viceversa).
- UDP
  - **java.net.DatagramSocket:** Esta clase es usada para enviar y recibir *datagram packets*.. No se garantiza que los paquetes lleguen a destino ni que lleguen en el mismo orden en que fueron enviados.
  - **java.net.DatagramPacket:** Esta es la clase que se va a enviar y recibir como mensaje.
  - **java.net.MulticastSocket:** Es un DatagramSocket con capacidades adicionales de multicast. Es útil para enviar y recibir multicast packets.

7.2 - Ejemplo Sockets TCP

El siguiente ejemplo instancia dos threads encargados de realizar una conexión cliente-servidor mediante sockets TCP: Server y Client. El primero quedará en espera hasta que el segundo realice la conexión correspondiente.

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class TestTCP {

  static final int PORT = 8765;
  static final String HOST = "localhost";

  public static void main(String[] args)
  {
       new TestTCP();
  }


  public TestTCP()
  {
    Server server = new Server();
    Thread serverThread = new Thread(server);
```

```
    serverThread.start();

    Client client = new Client();
    Thread clientThread = new Thread(client);
    clientThread.start();
  }

  public class Server implements Runnable {
    public void run() {
      try {
        System.out.println("[Server] Esperando mensaje...");
        ServerSocket serverConn = new ServerSocket(PORT);
        Socket socket = serverConn.accept();
        ObjectInputStream fromBuffer = new ObjectInputStream(socket.getInputStream());
        String datos = (String)fromBuffer.readObject();
        fromBuffer.close();
        socket.close();
        serverConn.close();
        System.out.println("[Server] Recibido: " + datos);
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

  public class Client implements Runnable {
    public void run() {
      try {
        Socket socket = new Socket(HOST, PORT);
        ObjectOutputStream toBuffer = new
ObjectOutputStream(socket.getOutputStream());
        System.out.println("[Client] Enviando mensaje");
        toBuffer.writeObject("MENSAJE");
        toBuffer.flush();
        toBuffer.close();
        socket.close();
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

}
```

7.3 - Ejemplo Sockets UDP

El siguiente ejemplo es similar al anterior, pero utilizando UDP en lugar de TCP. Notar que en este ejemplo se trabaja mediante array de bytes, el envío es no bloqueante, y se está utilizando *MulticastSockets* a fin de que los hosts destinatarios puedan ser varios en lugar de solo uno.

```java
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class TestUDP {

  public static final int PORT_UDP = 9998;
  public static final String GROUP_IP = "230.0.0.1";

  public static void main(String[] args)  {
    new TestUDP();
  }

  public TestUDP()  {
    Server server = new Server();
    Thread serverThread = new Thread(server);
    serverThread.start();

    Client client = new Client();
    Thread clientThread = new Thread(client);
    clientThread.start();
  }

  public class Client extends Thread {
    public void run() {
      try
      {
        // demoramos a fin de que el server esté esperando a recibir
        Thread.sleep(1000);
        byte[] buf = new byte[256];
        buf = "MENSAJE".getBytes();
        InetAddress group = InetAddress.getByName(GROUP_IP);
        DatagramPacket packet = new DatagramPacket(buf, buf.length, group, PORT_UDP);
        MulticastSocket socket = new MulticastSocket(PORT_UDP);
        System.out.println("[Client] Enviando mensaje");
        socket.send(packet);
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }
  }
```

```
  public class Server extends Thread {
    public void run() {
      try
      {
        MulticastSocket socket = new MulticastSocket(PORT_UDP);
        InetAddress group = InetAddress.getByName(GROUP_IP);
        socket.joinGroup(group);

        DatagramPacket packet;
        byte[] buf = new byte[256];
        packet = new DatagramPacket(buf, buf.length);
        System.out.println("[Server] Esperando mensaje...");
        socket.receive(packet);
        String received = new String(packet.getData());
        System.out.println("[Server] Recibido: " + received);
        socket.leaveGroup(group);
        socket.close();
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

}
```

7.4 - Sockets en Android

La API de Sockets para Android es igual a la de Java, con lo cual se simplifica el desarrollo de aplicaciones heterogéneas, dado que es posible implementar una única librería en común que contenga la funcionalidad de acceso y uso de sockets.

Por ejemplo, en Eclipse es posible crear un proyecto en común que será utilizado tanto por aplicaciones Java tradicionales como por aplicaciones Android. Luego, este proyecto puede ser referenciado desde desde otro proyecto, en la configuración *Java Build Path* de este último.

7.4.1 Multicasts en Android

Cabe destacar una consideración especial en lo que respecta a recepciones multicast. A menos que se configure explícitamente otra cosa, las mismas son filtradas, básicamente por razones de consumo de energía del equipo:

> "Normally the Wifi stack filters out packets not explicitly addressed to this device. Acquring a MulticastLock will cause the stack to receive packets addressed to multicast addresses. Processing these extra packets can cause a noticable battery drain and should be disabled when not needed."

Fuente: http://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.html

Para poder recibir paquetes multicast, es necesario obtener el lock correspondiente. Esta tarea se generalmente se realiza en tres pasos: 1) Instanciar y configurar el multicastLock en el

método ***onCreate()***. 2) Requerir en el método ***onResume()***. 3) Liberar en el método ***onPause()***.

```
public void onCreate(Bundle savedInstanceState) {
    ...
    WifiManager wm = (WifiManager)getSystemService(Context.WIFI_SERVICE);
    WifiManager.MulticastLock multicastLock = wm.createMulticastLock("myDebugTag");
}

protected void onResume() {
    super.onResume();
    if (multicastLock!=null && !multicastLock.isHeld())
        multicastLock.acquire();
}

protected void onPause() {
    super.onPause();
    if (multicastLock!=null && multicastLock.isHeld())
        multicastLock.release();
}
```

Apéndice A: Práctica

1. Proponer una solución que involucre la instanciación dinámica y carga de datos de un objeto desde una aplicación cliente Android y su posterior envío a un servidor; indicándole a este último el o los métodos que debe ejecutar, para luego retornar el valor resultante.
2. Implementar mediante sockets TCP un *“control remoto IP”* para dispositivo móvil encargado de comandar un servicio (corriendo por ejemplo en un equipo desktop) como el de reproducción de música (acciones de reproducción, pausa, etc.) o navegación de fotografías (siguiente, anterior, etc.).
3. Desarrollar utilizando sockets UDP un *“notificador de hosts conectados en una LAN”*, teniendo en cuenta que no será necesario contar con un servidor central para esta tarea, sino que cada dispositivo móvil deberá mantener actualizada su tabla de hosts conectados.