Reconfiguración de la interacción en sistemas Android: adaptando las notificaciones al contexto.

17 de septiembre

2012

Deisson Leonardo Sánchez Toledo

Tesina de Máster

Directores: Vicente Pelechano Ferragud
Miriam Gil Pascual

UNIVERSIDAD POLITECNICA DE VALENCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Esta Memoria está dedicada a las cuatro personas por las cuales tengo tantas aspiraciones: mis padres Héctor y Rubiela, porque gracias a su gran esfuerzo, hoy puedo ver cumplida la meta que tenía de pequeño, en cuanto a nivel de educación superior se refiere, y mis hermanos Anderson y Jefferson, por su gran apoyo, no solo en el tiempo que he dedicado a esta Tesina, sino por todo el tiempo dedicado a mis estudios, ya que ellos han sido siempre mi mayor inspiración. Además, he de agradecer a mi hermano Anderson, por su ayuda, aportando sus grandes conocimientos de informática, sin los cuales me hubiese sido mucho más difícil alcanzar los objetivos de esta Tesina, en el tiempo planificado para ello.

También tengo que dar las gracias a mis amigos, puesto que siempre me han dado ánimo para alcanzar esta meta, y siempre han sabido brindarme el mayor apoyo para no desistir, en la lucha constante que requiere el conquistar tus objetivos.

A mi familia porque aunque está lejos siempre ha estado ahí.

Y como no! Tengo que dar las gracias a mi director Vicente Pelechano, y en especial a mi directora Miriam Gil, porque sin su idea y sin su predisposición, no hubiese sido posible realizar esta Tesina.

Contenido

Lista de imágenes

1. INTRODUCCIÓN

En un mundo en el que en la actualidad el uso de dispositivos móviles se ve cada vez más extendido entre la gran mayoría de la población, y en el cual dicha población exige poder estar informada de una forma estratégica, teniendo en cuenta el lugar y la actividad que se encuentren realizando en un determinado momento de su vida cotidiana, los Ingenieros y desarrolladores de nuevas tecnologías, nos vemos obligados a intentar desarrollar sistemas, que permitan informar de una forma adecuada a los usuarios, sobre todas aquellas cosas que tengan un grado de relevancia para su vida diaria.

Para la computación móvil el contexto lo es todo [*2*]. Los dispositivos móviles nos acompañan allá donde estemos durante la mayor parte del día. A diferencia de los ordenadores personales, o incluso los portátiles que los podemos trasladar con relativa facilidad, los dispositivos móviles están con nosotros en todo tipo de entornos, en los que confluyen múltiples parámetros, tales como, la localización del usuario, el sonido ambiente, o las necesidades o tareas a realizar [*5*]. Por ello las interfaces de estos dispositivos se tendrían que adaptar al usuario, para conseguir una interacción lo menos molesta posible para el mismo [*3*].

En este contexto, es fácil imaginar un mundo en el cual podamos recibir, en nuestros dispositivos móviles, notificaciones que nos puedan hacer más fácil nuestro día a día, como por ejemplo, avisos de nuestro frigorífico; informándonos de la falta de determinados alimentos, solicitudes de nuestra televisión; preguntándonos si queremos grabar nuestra serie favorita que está a punto de empezar, recordatorios de nuestra agenda; sobre una determinada cita, y así podríamos tener una larga lista, de todas las posibles cosas por las cuales podríamos ser informados mediante nuestro dispositivo móvil. Todas estas notificaciones pueden causar irritación en el usuario. Por tanto, cada una de dichas notificaciones, se debería presentar de la forma más adecuada, dependiendo del entorno en el que se encuentre el usuario.

Debido a que en la actualidad disponemos de dispositivos móviles, que cada vez están equipados con más recursos de interacción (GPS, acelerómetro, vibración, diversos recursos de notificación, etc.), podemos desarrollar servicios y/o aplicaciones, para que los usuarios puedan estar informados, y puedan hacerlo de la forma más adecuada dependiendo del contexto, con el fin de que esta información no se convierta en un inconveniente, sino que por el contrario, ayude a los usuarios a controlar de una forma más eficaz sus actividades. Según "Considerate computing" [*4*], en el contexto móvil, en el cual los usuarios están permanentemente conectados al entorno, dichos usuarios pueden ser interrumpidos constantemente, lo cual puede resultar insoportable. Los diferentes servicios proporcionados en los dispositivos móviles, deberían actuar de tal forma que los usuarios puedan recibir las notificaciones (de su mayor interés), de la manera más adecuada de acuerdo al contexto, para que no se sientan irritados por el

sistema. Por tanto, los sistemas móviles deberían ser capaces de solicitar la atención del usuario solo cuando sea realmente necesario, y se debería hacer de la forma más adecuada, dependiendo del entorno en el que éste se encuentre.

1.1. Establecimiento del problema

Resultaría muy interesante poder tener un sistema, en el cual pudiésemos ser informados sobre todas las cosas que suceden a nuestro alrededor, y sobretodo de aquellas cosas que tienen una mayor importancia para nosotros. Además, uno de los factores más relevantes a la hora de recibir dichas notificaciones, sería que el dispositivo tuviese un sistema que fuera sensible al contexto (context-aware), para de esa forma, asegurarnos que vamos a recibir nuestras notificaciones, en el momento indicado y de la forma más adecuada. Para ello, el sistema de notificaciones se debería poder ajustar dinámicamente, dependiendo del entorno en el que se encuentre el usuario.

El requerimiento del ajuste en tiempo real en la forma de mostrar las notificaciones, se basa en que cada tipo de notificación se debe mostrar de una forma determinada, dependiendo del usuario y del contexto en el que se encuentre el mismo. Por ejemplo, para un usuario que va conduciendo su coche solo, un servicio que proporcione avisos sobre la compra de alimentos, se debe mostrar de una forma que llame la atención del conductor (sin despistarlo), cuando éste pase cerca de un supermercado. Sin embargo, si el usuario viaja acompañado, la notificación puede ser presentada de una forma más discreta, dependiendo de la privacidad del mensaje.

En base esto, existe la necesidad de desarrollar un ***Reconfigurador,*** que permita adaptar las interfaces de usuario de dispositivos móviles, al contexto de uso en tiempo de ejecució, para conseguir la interacción adecuada en cada momento. En particular, nos centraremos en las interfaces de usuario de dispositivos Android, por ser una plataforma software de código abierto.
Además, todos los elementos de una interfaz de usuario de una aplicación Android, se construyen mediante una estructura de árbol, usando objetos *View* (Vistas) y *GroupView* (Grupo de vistas) (ver Imagen 1). Siguiendo el patrón de composición, un *View* es un componente de la interfaz de usuario, que dibuja algo; en la pantalla con la que el usuario puede interactuar. Un *ViewGroup* es un contenedor de objetos *View*, para definir el *Layout* de la interfaz. Android proporciona una colección tanto de objetos *View* como de *ViewGroup*, para ofrecer al programador controles de entrada (como botones o campos de texto) y varios modelos de *Layout* (Como *Linear* y *Relative Layout*). La interfaz de usuario de cada componente (siendo un componente una entidad compuesta por varios objetos *View*, como por ejemplo un *Dialog*) de una aplicación, se define usando una forma jerárquica de objetos *View* y *ViewGroup*, lo cual es lo ideal, para que el ***Reconfigurador*** pueda activar y desactivar los componentes genéricos (*StatusBar, Dialogs,* etc.), implementados para esta Tesina. A continuación se muestra la jerarquía de objetos *View* y *ViewGroup* empleados en Android.

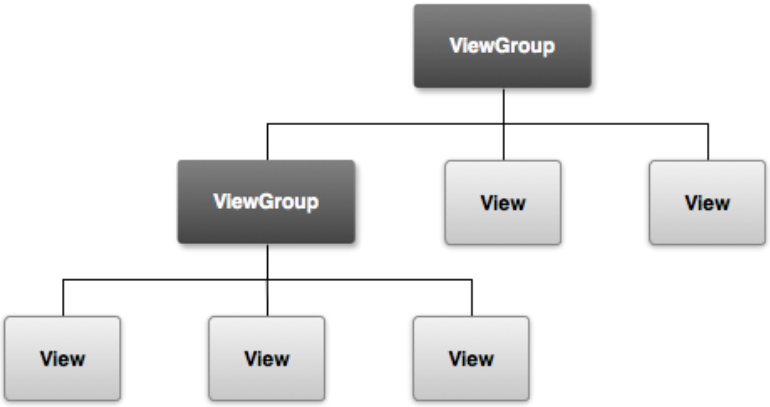

Imagen 1. Jerarquía de objetos View que definen la IU Android

Los objetivos que se persiguen en esta Tesina para conseguir esto son:

1. Desarrollo de unos componentes de interacción genéricos, que permitan su activación y desactivación en tiempo de ejecución.
2. Desarrollo de un ***Reconfigurador***, que permita presentar las notificaciones de la forma más adecuada, mediante la activación y desactivación de los componentes de interacción apropiados. Además estará integrado con *MoRE* [*1*] (Model-based Reconfiguration Engine), para dotarlo de una reconfiguración autónoma.
3. Validación del funcionamiento del ***Reconfigurador***, mediante su aplicación a un caso de estudio real.

1.2. Solución propuesta

Para alcanzar los objetivos planteados en el apartado anterior, lo primero que se ha de hacer es: estudiar la implementación básica de los componentes de interacción Android (Status bar, Toast, Sound, Vibration, etc.), para poder implementar dichos componentes de una forma genérica, para su utilización dentro del ***Reconfigurador***. Por otro lado, se han de estudiar las diversas formas existentes, para el trabajo con notificaciones push; dentro del entorno Android, para poder realizar la comunicación entre los servicios del entorno y el ***Reconfigurador***. Por último, se ha de profundizar en la creación de aplicaciones Android, para poder desarrollar una, en la cual, se pueda ver el funcionamiento de todo el sistema implementado.

A continuación se explica brevemente, las tres grandes partes en las que se ha dividido el desarrollo de esta Tesina:

1.2.1. Implementación de los componentes de interacción

Para conseguir que los componentes de interacción Android; se puedan utilizar de una forma genérica, éstos se han de implementar mediante un *Service* o una *Activity* Android (dependiendo del caso), para que se puedan invocar en tiempo de ejecución; mediante *Intents*. Además, a dichos componentes se les podrá pasar un número determinado de sub-componentes; según proceda, por ejemplo, un componente *Dialog*, necesita otros componentes como: texto, botones, etc.

1.2.2. Implementación del Reconfigurador

En la implementación del ***Reconfigurador*** se ven involucrados dos grandes procesos. El primero consiste en la recepción de las notificaciones de los diferentes servicios, para lo cual, se hace uso del *servidor de notificaciones* push; utilizado en los sistemas Android (*C2DM*). El segundo proceso involucrado, es el desarrollo de un controlador; que permite la integración con *MoRE* [*1*], el cual permite obtener un *JSON*, que contendrá la configuración con la que se debe mostrar la notificación; dependiendo del contexto. Una vez obtenida dicha configuración, el controlador se encargará de activar los componentes de interacción adecuados, que nos proporcione *MoRE* [*1*].

Por otro lado, para poder tener un sistema completo, y poder ver la funcionalidad del mismo, se ha de desarrollar un servicio web, para poder simular el envío de notificaciones al reconfigurador. Se desarrollará un servicio web *RESTful*, para poder enviar las notificaciones de la forma más apropiada. De esta forma, los servicios podrán enviar notificaciones; con tan solo enviar la información de la notificación vía HTTP. Para esta implementación se ha usado el Framework Restlet (http://www.restlet.org). Las notificaciones han sido formateadas en JSON, un formato ligero para el intercambio de datos. De esta forma, todos los componentes del sistema podrán entender el formato de la información.

1.2.3. Implementación de la Aplicación final

Para validar la funcionalidad del ***Reconfigurador***, se desarrollará una aplicación Android, para la gestión de las notificaciones. La aplicación consistirá en una interfaz *maestro-detalle*, que tendrá una vista de todas las notificaciones recibidas, así como una vista exclusiva de las notificaciones; que no han sido leídas. Al seleccionar una de las notificaciones, se podrá visualizar el detalle de la misma. Esta aplicación estará integrada con el ***Reconfigurador***, el cual se encargará de activar los componentes, con los que se ha de mostrar cada notificación.

Por otro lado, se han implementado servicios *REST*, que permitirán a la aplicación; hacer solicitudes como: obtener todas las notificaciones, obtener las notificaciones no leídas, marcar una notificación como eliminada entre otras.

1.3. Contexto de la Tesina

Esta Tesina está desarrollada en el contexto del Centro de Investigación en Métodos de Producción de Software (***ProS***), de la Universidad Politécnica de Valencia (www.pros.upv.es).

1.4. Estructura de la Tesina

La Tesina está dividida en siete grandes capítulos, incluido este capítulo introductorio. A continuación se detalla brevemente el contenido de cada uno de ellos:

1.4.1. Aplicaciones relacionadas

En este capítulo se describirán algunas aplicaciones, que tienen una funcionalidad similar a la desarrollada en esta Tesina, como por ejemplo: el *centro de notificaciones de Android* o la aplicación *Locale* entre otras. Además, se especificarán las diferencias entre dichas aplicaciones y la desarrollada en esta Tesina.

1.4.2. Tecnologías Involucradas

Se explicará, sin entrar en detalles complejos, el funcionamiento de los componentes de interacción de la plataforma Android, puesto que ha sido la elegida para el desarrollo de esta Tesina. También se describirá la estructura que toma el SO Android, para el desarrollo de aplicaciones. Además, se explicará el funcionamiento del *servidor de notificaciones* push, empleado en esta plataforma (*C2DM*).
Por otro lado, se dará una breve descripción, de las plataformas utilizadas en el proceso del envío de notificaciones al reconfigurador.

1.4.3. Visión global de la solución

A partir de una imagen que ilustra todos los procesos involucrados (arquitectura del sistema), se dará una visión generalizada, de la forma en la que se ha llevado a cabo el desarrollo de las distintas fases de la Tesina, para lograr cada uno de los objetivos especificados en el *apartado 1.1.*

1.4.4. Desarrollo de la propuesta

Se explicará detalladamente, cada una de las partes en las que se ha dividido el trabajo, para alcanzar los objetivos finales. Por tanto, este capítulo se dividirá en tres grandes sub-apartados:

- Implementación de los componentes genéricos.
- Implementación del ***Reconfigurador***.
- Implementación de la Aplicación de notificaciones.

Además, este capítulo tendrá un apartado final, en el que se expondrán todos aquellos aspectos de implementación más relevantes.

1.4.5. Un Caso Práctico

En este capítulo se expondrá un ejemplo de un caso en concreto; con tres escenarios diferentes, para mostrar la funcionalidad el sistema desarrollado, así como la forma de uso de la aplicación desarrollada.

1.4.6. Conclusiones

Se resumirán los aspectos más importantes, llevados a cabo durante la realización de esta Tesina, así como las contribuciones que se han hecho, al llevar a cabo la implementación de la misma.

2. APLICACIONES RELACIONADAS

En este capítulo se presenta una descripción de algunas aplicaciones que tienen una funcionalidad similar a la desarrollada en esta Tesina y una breve comparación para mostrar la novedad que se introduce. Las aplicaciones que se presentan en este capítulo son:

- Centro de notificaciones de Android
- Locale
- Pushover

2.1. Centro de notificaciones de Android

Imagen 2. Centro de notificaciones Android

El centro de notificaciones Android es muy sencillo, es una ventana que se muestra/oculta desplazando hacia abajo la barra de tareas (ver Imagen 2) y su funcionamiento es muy simple:

- Cuando llega una nueva notificación al dispositivo, se muestra un icono (relacionado con la aplicación que ha generado la notificación) en la barra de tareas.

- Además se crea un nuevo ítem en la ventana de notificaciones con el mensaje que contiene la notificación.
- La mayoría de veces al seleccionar la notificación en el centro de mensajes ésta desaparece de la lista y se abre la aplicación que ha enviado la notificación. Esto sucede siempre que el programador de la aplicación lo haya programado, es decir no es un proceso automático del sistema Android.

Como funcionalidad extra, este centro de notificaciones tiene una barra fija en la parte superior en la que se pueden activar/desactivar las funciones de WIFI, Bluetooth, GPS, Transferencia de datos y Rotación de pantalla.

2.1.1. Comparativa

El centro de notificaciones de Android no es sensible al contexto (context-aware), característica que si posee nuestro sistema. Dicha característica es el mayor logro que se alcanza con el desarrollo del *Reconfigurador* implementado en esta Tesina. El centro de notificaciones se limita a recibir las notificaciones en un determinado momento sin importar el contexto en el que se encuentre el usuario con su dispositivo móvil. Las notificaciones en Android siempre se muestran de la misma forma: Mediante un icono en la barra de tareas y un mensaje en la ventana de mensajes de Android, mientras que en esta Tesina las notificaciones se mostrarán de una forma u otra dependiendo de un fichero de configuración proporcionado de la integración con *MoRE* [*1*].

Por otro lado aunque los dispositivos móviles Android disponen de un centro de mensajes, este no llega a ser una aplicación en la que el usuario puede tener un listado en el que pueda controlar todas sus notificaciones. Además, en el centro de mensajes de las notificaciones Android, si el mensaje de la notificación es muy largo no se visualiza al completo, sin embargo en la aplicación creada en esta Tesina se dispone de una *Activity* en la que se muestra el detalle al completo de una notificación para poder visualizar todo el mensaje por largo que sea.

2.2. Locale

Locale es una aplicación móvil para dispositivos Android que permite cambiar perfiles del móvil en base a la localización. Los perfiles, que tendrán que ser definidos por el usuario son simplemente posiciones geográficas. La forma de funcionar es la siguiente:

- El usuario de la aplicación tiene que crear perfiles para que el dispositivo sepa en qué lugar se encuentra. Esto se hace indicando en el mapa un radio de la zona para la que se desea vincular al perfil.
- El usuario elegirá los componentes (*widgets*) del dispositivo que para un determinado perfil se desea activar o desactivar.
- La aplicación se encargará de realizar la activación o desactivación de los *widgets* cuando el dispositivo móvil se encuentre en uno de los perfiles definidos por el usuario

Mediante *Locale*, un usuario crea ***situaciones*** especificando ***condiciones*** bajo las cuales deben cambiar la configuración del dispositivo móvil (ver Imagen 3Imagen 3, Imagen 4). Por ejemplo la *situación* "en el colegio" avisa cuando la *condición* de localización es "la dirección del colegio" y cambia la configuración del volumen a vibración.

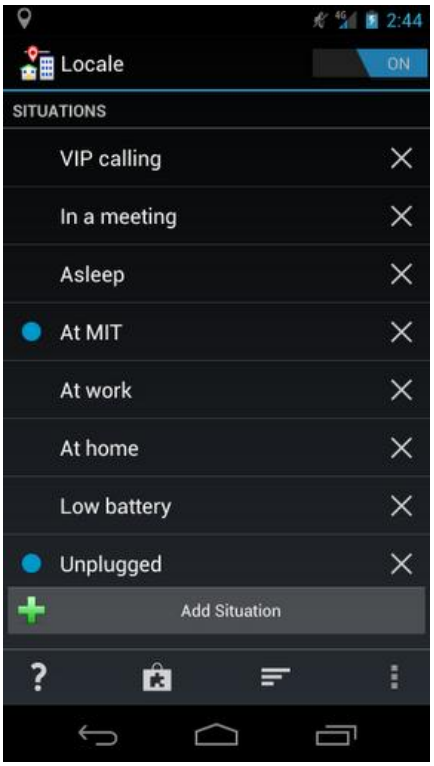

Imagen 3. Locale - Configuración de la localización

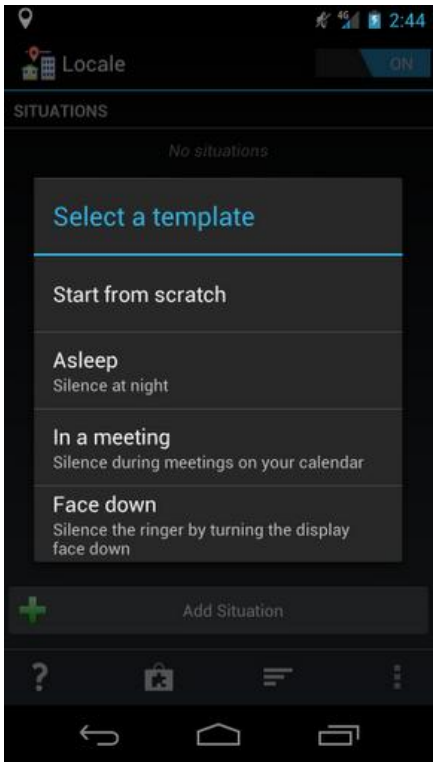

Imagen 4. Locale - Elección de una de las plantillas creadas

2.2.1. Comparativa

Esta aplicación se asemeja un poco al *Reconfigurador* desarrollado en esta tesina en cuanto a que puede activar o desactivar *widgets* dependiendo de la localización en la que se encuentre el dispositivo móvil. La diferencia con nuestro sistema es que nuestro sistema es context-aware y no sólo location-aware.
Se puede ver que la gran diferencia de la sensibilidad al contexto es que en *Locale* la "reconfiguración" se hace en base a unos perfiles que deben ser definidos previamente por el usuario, mientras que el *Reconfigurador* definido en esta Tesina lo hace automáticamente gracias a la integración con *MoRE* [*1*]. Por otro lado el *Reconfigurador* creado en esta Tesina se centra en activar componentes de interacción Android como por ejemplo: Sonido, Vibración, Dialogs, etc. mientras que *Locale* activa otro tipo de componentes como: WIFI, alarma, brillo de la pantalla entre otros.

Además en esta Tesina se ha implementado una pequeña aplicación en la que se puede tener un control de las notificaciones que se han recibido en todo momento, pudiendo incluso eliminarlas cuando ya no sean de gran importancia para el usuario.

2.3. Pushover

Pushover es una plataforma para enviar y recibir notificaciones (ver Imagen 5Imagen 5). En la parte del servidor proporciona un API HTTP para encolar los mensajes a enviarlos a los clientes. En la parte del cliente se reciben las notificaciones, se muestran al usuario y las almacena para poder visualizarlas (en una aplicación muy simple) en offline. En la aplicación del cliente existe una ventana que permite al usuario configurar el modo de recepción de las notificaciones manualmente (ver Imagen 6Imagen 6).

Esta propuesta es muy interesante ya que proporciona a los desarrolladores de aplicaciones la forma de integrar este sistema en cualquier aplicación tanto en sistemas Android como en sistemas IOS.

El gran inconveniente de esta plataforma es que el usuario final solo podrá recibir las notificaciones enviadas por aplicaciones que utilicen esta plataforma como sistema de envío de notificaciones y actualmente la lista es muy corta.

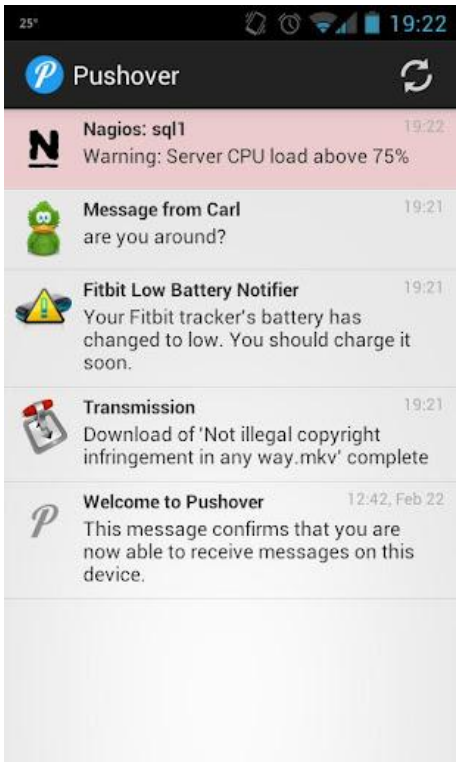

Imagen 5. Pushover - Pantalla de notificaciones

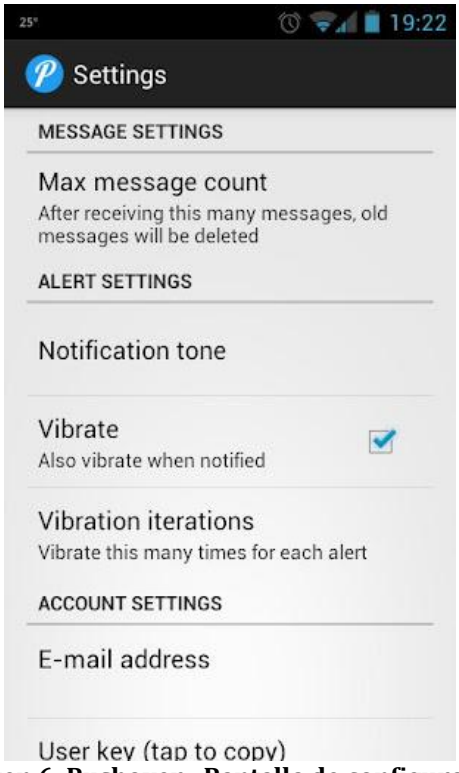

Imagen 6. Pushover - Pantalla de configuración

2.3.1. Comparativa

Como se puede apreciar, la gran similitud que tiene este sistema con el diseñado en esta Tesina es la forma de enviar los mensajes, ya que se hace mediante la comunicación con el sistema nativo de Android (*C2DM*) de una forma muy similar a como se ha desarrollado en esta Tesina. Además posee una interfaz que permite al usuario tener un control de las notificaciones recibidas, como se puede ver en la Imagen 5. Pese a esto se puede definir como una aplicación relativamente pobre respecto a prestaciones, ya que para lo único que sirve es para recibir notificaciones que han sido enviadas desde aplicaciones que tienen integrado este sistema.

Por otra parte no se dispone de ninguna funcionalidad que permita la recepción de las notificaciones de una forma especial, simplemente proporciona una interfaz para configurar de forma estática la recepción de las notificaciones, como se puede ver en la Imagen 6.

3. TECNOLOGÍAS INVOLUCRADAS

Para que todo aquel que lea esta Tesina, pueda tener unos conocimientos básicos sobre las tecnologías utilizadas en su desarrollo, en este capítulo, se dará una descripción de cada una de ellas.

3.1. Sistemas Android

Para el desarrollo de esta Tesina se ha elegido el sistema Android. Éste es un sistema operativo (SO) móvil basado en Linux, y que ha sido desarrollado por la Open Handset Alliance, la cual es liderada por la empresa Google. Se ha elegido este sistema, debido a que la utilización de su código es libre. La mayor parte de la documentación utilizada para el trabajo con este sistema, ha sido tomada de la página Web oficial, de desarrolladores Android [7].

La estructura del SO Android, se compone de aplicaciones, que se ejecutan en un framework Java de aplicaciones orientadas a objetos, sobre el núcleo de las bibliotecas de Java, en una máquina virtual (Dalvik), con compilación en tiempo de ejecución.
Actualmente este SO dispone de 10 versiones, siendo la más usada la versión 2.3 (*Gingerbread*), por lo que ha sido la elegida para el desarrollo de esta Tesina.

3.1.1. Fundamentos de una aplicación Android

Para el desarrollo de aplicaciones Android, se ha de instalar el plugin para eclipse (ADT – Android Development Tools). Además, se ha de tener el SDK (Set Development Kit) de Android, así como el JDK (Java development Kit).

Una aplicación Android está compuesta de código Java y ficheros XML, así como de los recursos externos que se puedan necesitar (imágenes, sonido, etc.). El compilador de Android, compila el código fuente de una aplicación junto con sus archivos de datos y recursos, y los empaqueta en un solo fichero con extensión ***.apk***. Dicho fichero es el que utilizan los dispositivos Android, para instalar las aplicaciones.

3.1.1.1. Componentes de una aplicación

Los componentes de una aplicación, son los bloques esenciales para la construcción de una aplicación Android: actividades (Activities), servicios

(Services), proveedores de contenido (Content Providers) y receptores de emisiones (Broadcast Receiver s). A partir de ahora, se hará referencia a ellos con su nombre en inglés, para no dar lugar a confusiones, ya que su nombre en castellano no está muy difundido. Cada uno tiene un propósito diferente y tiene un ciclo de vida distinto, que define, cómo se crean y se destruyen dichos componentes. A continuación se detallará cada uno de ellos.

3.1.1.1.1. Activities

Una *Activity* representa una pantalla con una interfaz de usuario, es decir, es la forma que tienen los sistemas Android, para representar las interfaces gráficas, con las que los usuarios del sistema podrán interactuar. Una ventana suele abarcar el tamaño de la pantalla, pero se pueden crear; para que sean más pequeñas; e incluso para que sean flotantes, es decir, que estén sobre otra *Activity* que se esté visualizando de fondo.

Una aplicación suele constar de un conjunto de *Activities*; que están estrechamente relacionadas. Normalmente se tiene una *Activity* principal, que se muestra al ejecutar por primera vez la aplicación. Cada *Activity* de la aplicación puede ejecutar otra *Activity*, para mostrar diferentes acciones de la funcionalidad de la aplicación. Cada vez que se inicializa una nueva *Activity*, la *Activity* previa pasará a estar parada (estado *stopped*), pero el sistema preservará dicha *Activity* en una pila ("*back stack*"). Cuando se inicializa una nueva Activity, se introduce en la pila y aparece en primer plano. La pila cumple el mecanismo: "El último en entrar es el primero en salir" ("*last in, first out*"), por tanto, cuando el usuario presiona la tecla de volver (*back button*), la *Activity* que se estaba mostrando sale de la pila y es destruida (estado *destroyed*), con lo que la aplicación muestra la última *Activity* que está en la pila, y esta última, pasa de estar parada (*stopped*) a estar reanudada (estado *resumed*).
Debido a que una *Activity* puede estar en diferentes estados, durante su ciclo de vida, existe un conjunto de métodos que se pueden sobrecargar, para realizar acciones en cada uno de los estados en los que se encuentre. Por ejemplo, cuando una *Activity* pasa al estado *stopped*, debería liberar los objetos que consumen demasiados recursos, como por ejemplo: conexiones de red y de bases de datos, pero cuando pase de nuevo al estado *resumed*, se deberían volver a establecer dichas conexiones, y se deberían reanudar las acciones interrumpidas.

A continuación, se muestra una imagen con todos los estados y el flujo de ejecución de los mismos, en los que puede estar una *Activity* durante su ciclo de vida.

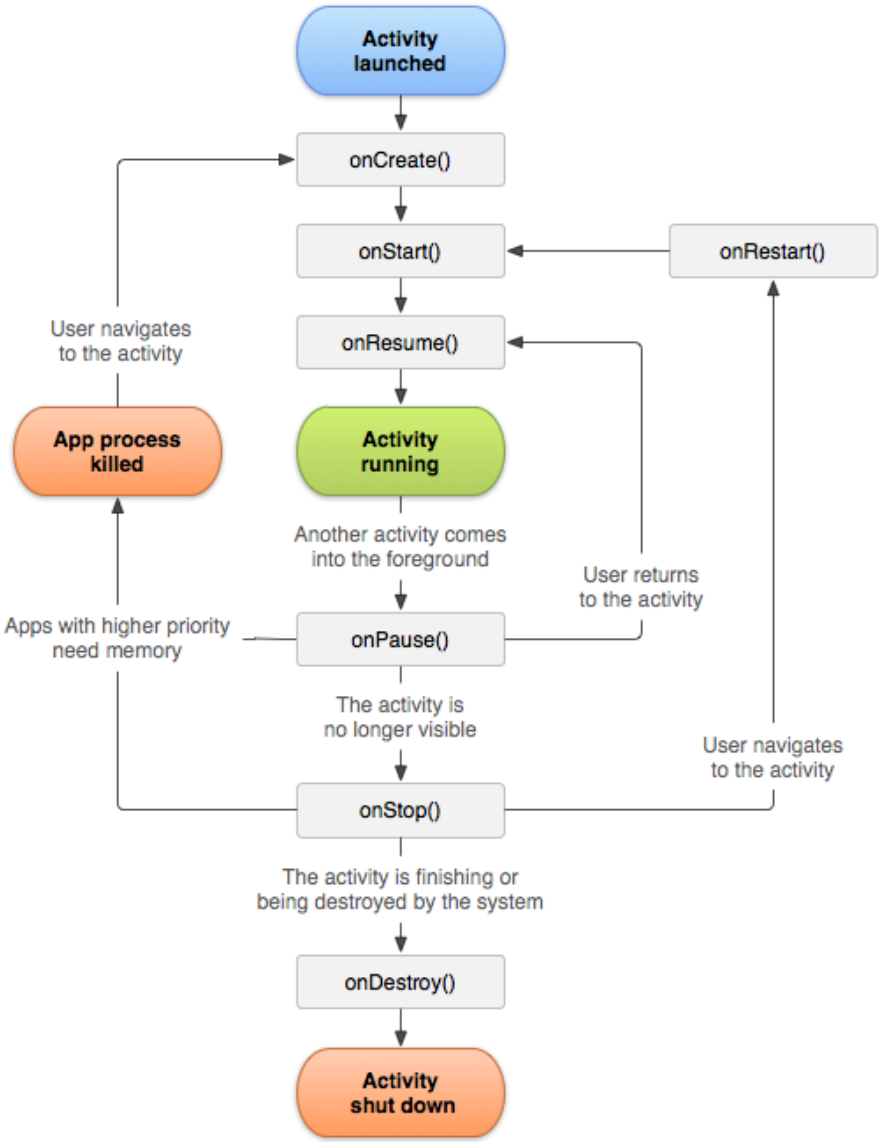

Imagen 7. Activity lifecycle

onCreate(): Se llama cuando se ejecuta una *Activity* por primera vez. Aquí se deben crear las vistas, enlazar los datos a las listas, etc.

onRestart(): Se llama después de que una *Activity* haya sido parada, justo antes de que se inicialice de nuevo, y siempre seguido por *onStart*().

onStart(): Se llama siempre antes de que la *Activity* se haga visible al usuario. Seguida por *onResume*(); si la *Activity* pasa a primer plano o por *onStop*(); si va a permanecer oculta.

onResume(): Se llama justo antes de que la *Activity* comience la interacción con el usuario. En este punto, la *Activity* está en la cima de la pila. Siempre va seguida por *onPause*().

onPause(): Se llama cuando el sistema va a reanudar otra *Activity*. Este método se usa normalmente para guardar datos, parar animaciones y otras cosas que pueden consumir CPU. Esto se debe hacer bastante rápido, puesto que la siguiente *Activity* no se reanudará, hasta que ésta no termine sus acciones. Es seguida por *onResume*(); si la *Activity* vuelve al primer plano, o por *onStop*(); si se hace invisible al usuario.

onStop(): Se llama cuando la *Activity* ya no va a ser visible al usuario, esto puede pasar porque vaya a ser destruida o porque otra *Activity* va a pasar a ocupar el primer plano. Puede ser seguido por *onRestart*(); si la *Activity* vuelve a interactuar con el usuario o por *onDestroy*(); si la *Activity* va a desaparecer.

onDestroy(): Se llama antes de que la *Activity* se destruya. Es la última llamada que recibe una *Activity*. Puede haber sido llamada porque la *Activity* ha sido finalizada (invocación del método *finish*()) o porque la está destruyendo el sistema, por falta de espacio. Se puede saber cuál de los dos escenarios es el que está ocurriendo, con el método *isFinishing*().

Creación de una Activity:

Para crear una *Activity*, se ha de crear una subclase de ***Activity***. En la subclase, se ha de implementar el método *onCreate*(), que es el que se llamará cuando se inicialice la aplicación. Además, se deberán implementar todos los demás métodos que sean necesarios (*onStop*(), *onResume*(), *onPause*(), etc.).

La interfaz de usuario de una *Activity* está formada por una jerarquía de vistas (objetos derivados de la clase *View*). Cada vista controla un espacio rectangular específico, dentro del la ventana de la *Activity*, y puede responder a las interacciones de los usuarios. Por ejemplo, una vista puede ser un botón, que realice una acción cuando sea presionado.

Android proporciona un número de vistas, que se pueden usar para diseñar y organizar los *Layouts*. Los ***Layouts*** son vistas derivadas de un *ViewGroup*, que proporcionan un único *Layout* modelo, para la vista de sus hijos, en Android se tienen los *Linear Layout*, *Relative Layout*, *Grid Layout* entre otros. Por otro lado, en un *Layout* se pueden añadir ***widgets***, que son vistas que proporcionan elementos visuales e interactivos para la ventana, como pueden ser: los botones, campos de texto, checkbox o una simple imagen.

La forma más común de definir un *Layout* usando vistas, es haciendo uso de un fichero *XML* y guardarlo en la carpeta de recursos de la aplicación. De esta manera, se puede tener de forma separada, las interfaces de usuario, del código fuente; que define el comportamiento de las *Activities*. Sin embargo, también es posible definir las vistas de un *Layout* en el propio código Java.

3.1.1.1.2. Services

Un *Service* es un componente de aplicación, que puede ejecutar operaciones de larga ejecución en *background* y no proporciona una interfaz de usuario. Otro componente de la aplicación puede iniciar un *Service*, y éste continuará ejecutándose en *background*, incluso si el usuario cambia a otra aplicación.

Un *Service* estará "iniciado", cuando un componente de la aplicación (como una *Activity*) lo inicialice; invocando el método *startService*(). Una vez iniciado, un *Service* puede estar ejecutándose en *background* indefinidamente, incluso si el componente que lo inició ha sido destruido. Normalmente, un *Service* iniciado ejecuta una operación simple, y no retorna nada al componente que lo ha invocado. Por ejemplo, puede descargar o subir un fichero a la red. Cuando la operación se termina, el *Service* se para solo.

Hay que tener en cuenta, que un *Service* se ejecuta en el *Thread* principal, es decir, cuando se inicia; no se crea un nuevo *Thread*, por tanto, si el acometido de un *Service* es realizar tareas que consuman mucha *CPU*, como por ejemplo, transferencias de red, se debería crear un nuevo *Thread* para evitar errores *ANR* (Aplicación No Responde).

Creación de un Service:

Para crear un *Service* se debe crear una subclase de ***Service***. En su implementación, se deben sobrescribir algunos métodos, para controlar algunos aspectos claves del ciclo de vida de un *Service*. Los métodos más importantes a sobrescribir son:

onStartCommand(): El sistema llama este método, cuando otro método inicia un *Service* mediante el método *startService*(). Una vez que se ejecuta este método, se inicia el *Service* en *background* indefinidamente, es responsabilidad del programador, parar el *Service* cuando haya terminado su acometido, llamando al método *stopSelf*() o al método *stopService*().

onBind(): El sistema llama a este método cuando otro componente se quiere ligar a este *Service*, llamando al método *bindService*(). Este método se ha de implementar siempre, pero si no se desea hacer ligamientos; simplemente se devuelve *null*.

onCreate(): El sistema llama a este método cuando el *Service* se crea por primera vez, este se ejecuta antes de llamar a los dos métodos citados anteriormente.

onDestroy(): El sistema llama a este método cuando el *Service* no se va a usar más, y va a ser destruido. Un servicio debe implementar este método para limpiar cualquier recurso, como por ejemplo, Threads, registered listeners,

Receivers, etc. Esta es la última llamada que recibe un *Service*.

La ejecución más común de un *Service*, se realiza mediante la llamada al método *onStartCommand*(), dicho método devuelve un *Integer*. El *Integer*, es un valor que describe como debe continuar el servicio, en el caso de que el sistema lo destruya. Para ello, se tienen los siguientes *Integers* de retorno:

START_NOT_STICKY

Si el sistema mata el *Service* antes del retorno del método *onStartCommand*(), no se recrea el *Service* al menos de que hayan *Intents* pendientes de enviar. Esta es la opción más segura de evitar ejecutar un *Service* cuando no sea necesario, y cuando la aplicación simplemente quiera reiniciar un trabajo no finalizado.

START_STICKY

Si el sistema mata el *Service* antes del retorno del método *onStartCommand*(), se recrea el *Service* y llama a *onStartCommand*() pero no reenvía el último *Intent*, en su lugar, el sistema llama a *onStartCommand*() con un *Intent* null, a menos de que hubieran *Intents* pendientes para iniciar el *Service*, en cuyo caso, dichos *Intents* serían enviados. Esto es útil para media players (o *Services* parecidos), que no ejecutan órdenes (*commands*), sino que se ejecutan indefinidamente y esperan a un trabajo (*job*).

START_REDELIVER_INTENT

Si el sistema mata el *Service* antes del retorno del método *onStartCommand*(), se recrea el *Service* y se llama a *onStartCommand*() con el último *Intent* que fue enviado al *Service*. Los *Intents* pendientes serán enviados de uno en uno. Esto es adecuado para *Services* que ejecutan trabajos activamente, que deben ser reanudados inmediatamente, como puede ser la descarga de ficheros.

A continuación se muestra una imagen con el ciclo de vida de un *Service*.

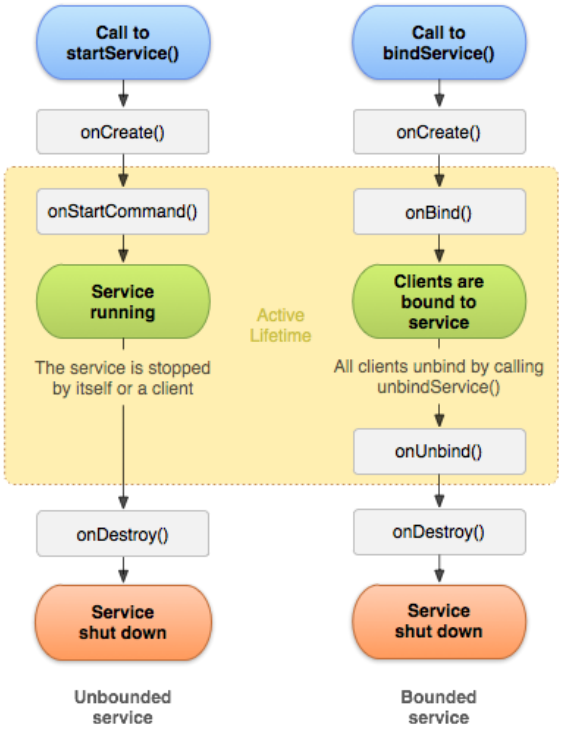

Imagen 8. Service Lifecycle

3.1.1.1.3. Content Providers

Un *Content Provider*, es el encargado de gestionar un conjunto de datos de la aplicación, que se han de compartir. Los datos pueden ser almacenados en: un sistema de archivos, una base de datos SQLite, en la Web o en cualquier otro lugar persistente, a la que la aplicación pueda acceder.

Mediante un *Content* Provider, otras aplicaciones pueden hacer consultas o incluso modificar los datos (si el *Content Provider* lo permite). Por ejemplo, el sistema Android proporciona un *Content Provider* que gestiona la información de los contactos del usuario, por tanto, una aplicación que tenga los permisos adecuados, podrá consultar parte del *Content Provider* (como *ContactsContract.Data*), para leer y escribir información sobre una determinada persona. Los *Content Provider* también son adecuados para leer y escribir datos que son propios de una aplicación, y no deben ser compartidos.

Un *Content Provider* se ha de implementar como una subclase de ***ContentProvider***, y debe implementar un conjunto estándar de *APIs*, que habilitan otras aplicaciones para ejecutar transacciones.

Cuando se desea acceder a los datos de un *Content Provider*, se ha de hacer mediante el uso de un objeto *ContentResolver*, que esté dentro del contexto de la aplicación, para realizar la comunicación con el *Provider* como un cliente. El funcionamiento es el siguiente: el objeto *ContentResolver* se comunica con el objeto *Provider* (una instancia de la clase que implementa *ContentProvider*). Por su parte el objeto *Provider* recibe las solicitudes de los datos desde los clientes, ejecuta las acciones solicitadas y devuelve los resultados.

3.1.1.1.4. Broadcast Receiver

Un *Broadcast Receiver*, es un componente que responde a la emisión de anuncios/notificaciones en todo el sistema. Existen muchos *Broadcasts* que los origina el sistema, como por ejemplo: *Broadcasts* que avisan que la pantalla se ha apagado, que la batería está baja, o que se ha capturado una imagen. Las aplicaciones por su parte también pueden iniciar *Broadcasts*, por ejemplo, para informar a otras aplicaciones que ya se han descargado los datos en el dispositivo, y pueden ser usados.

Aunque los *Broadcast Receivers* no muestran una interfaz de usuario, estos pueden crear una notificación *StatusBar*, para alertar al usuario cuando ocurra un evento *Broadcast*. Sin embargo, los *Broadcast* son usados más comúnmente, como una puerta ("*Gateway*") a otros componentes, y tienen como propósito, realizar el menor trabajo posible. Por ejemplo, este puede instanciar un *Service*, para que ejecute algún trabajo basado en el evento.

Un *Broadcast Receiver* se implementa como una subclase de *BroadcastReceiver*, y cada *Broadcast* se envía mediante un *Intent*.

3.1.1.2. Activación de componentes (Intents)

Tres de los cuatro componentes, *Activities*, *Services* y *Broadcast Receivers*, se activan mediante un mensaje asíncrono llamado ***Intent***. Los Intents enlazan componentes en tiempo de ejecución. Se puede pensar en ellos, como los mensajeros que solicitan una acción de otros componentes. Un *Intent* se crea mediante un objeto ***Intent***, el cual define un mensaje, para activar un componente específico.

Para *Activities* y *Services*, un *Intent* define la acción a ejecutar, por ejemplo, para ver o enviar algo. Para los *Broadcast Receivers* los *Intents* simplemente definen el anuncio que se está emitiendo, por ejemplo, un *Broadcast* para indicar que la batería está baja, incluye solo un *string* de acción conocido, que indica "batería baja". El otro tipo de componente, *Content Provider*, no se activa mediante *Intents* sino que se activa, cuando es el objetivo de una solicitud, realizada por un *ContentResolver*.

En un *Intent* también se pueden pasar datos básicos (Integer, Boolean, String, etc.) a los componentes, con métodos básicos como *putExtra*(), también se pueden pasar objetos personalizados, pero para ello, hay que hacer que dichos objetos implementen la clase *Parcelable*.

3.1.1.3. Fichero Manifest

El ***manifest*** en un fichero *XML* que se ha de crear para cada aplicación, dicho fichero se debe llamar: ***AndroidManifest.xml***. Éste contendrá, la declaración de cada uno de los componentes, de los que está compuesta la aplicación a la que pertenece. Otras de las cosas que se han de declarar en el *manifest* son:

- Todos los permisos de usuario que requiera la aplicación, como por ejemplo, acceso a Internet, acceso para la lectura de los contactos del usuario, etc.
- El API Level mínimo requerido por la aplicación, esto es, la versión mínima de Android, para la cual la aplicación es operativa.
- Las características de hardware y software, requeridos por la aplicación (Cámara, Bluetooth, etc.).
- Librerías API que se deben enlazar con la aplicación (diferentes al API de Android), como por ejemplo, las librerías de Google Maps.

3.1.1.4. Recursos de una aplicación

Una aplicación no está compuesta solo por código. Se requieren recursos que están separados del código fuente, tales como: imágenes, archivos de audio y todo aquello relativo a la presentación visual de la aplicación.
Para cada uno de los recursos que se incluyen en un proyecto Android, el "SDK build tools" define un único entero de identificación (ID), dicho ID será usado para hacer referencia a cada recurso, desde el código de la aplicación o desde otros recursos definidos en ficheros *XML*.

Uno de los aspectos más importantes de proporcionar los recursos separados del código fuente, es la capacidad de proporcionar recursos alternativos, para diferentes configuraciones de dispositivos.

En un proyecto Android tenemos una carpeta raíz llamada ***res***, en la que se almacenarán todos los recursos de la aplicación. Dentro de este directorio, estarán otros subdirectorios, para cada uno de los diferentes recursos posibles, que puede contener una aplicación. Dentro de los más comunes tenemos:

drawable:	Para la definición de gráficos con bitmaps o XML
color:	Para la definición de colores
layout:	Para la definición de los layouts que contendrán las interfaces de usuario.
values:	Para definir strings, estilos, etc.

En Android existen unos calificadores, que se añaden a los nombres de las carpetas; que contienen los recursos, para especificar algunos casos especiales. Por ejemplo, se puede tener una carpeta *res/drawable-hdpi*, para especificar que los recursos contenidos en la misma, serán utilizados en dispositivos con pantallas de alta densidad. Otro ejemplo, sería la carpeta *res/values-en/string.xml*, que contendría todos los strings utilizados en el idioma Inglés. Como se puede intuir, esto es muy útil, para desarrollar aplicaciones en diferentes idiomas, y además, que puedan ser usados en dispositivos de diferentes tamaños.

3.1.2. Interfaces de usuario (Layouts)

Como se ha explicado en el apartado de *Activities*, las interfaces de usuario, se suelen crear mediante ficheros *XML* que crean los *Layouts* (arquitectura de las

interfaces de usuario), en los que se pondrán cada uno de los *widgets* que interactuarán con el usuario.

En Android, existen varios tipos de Layouts, que permiten al usuario desplegar los widgets de una forma determinada, entre los más comunes tenemos:

LinearLayout:	Es un *ViewGroup* que alinea a todos sus hijos en una única dirección, vertical u horizontal.
RelativeLayout:	Es un *ViewGroup* que muestra las vistas de los hijos en posiciones relativas.
GridView:	Es un *ViewGroup* que muestras los ítems en dos dimensiones, cuadrícula enrollable. Éste tipo de Layouts es muy útil para mostrar imágenes.

Por otro lado, existen otros componentes de interacción tales como *Dialogs*, *Notifications*, *Toast*, etc. Estos componentes no se explicarán en este punto debido a que hacen parte del primer objetivo de esta Tesina, por tanto, se explicarán en el capítulo de desarrollo (*Componentes genéricos*).

Cuando se desea mostrar un listado (***ListView***); cuyos ítems (cada elemento de la lista) no son elementos básicos (strings, int, etc.), se ha de crear un *Layout* especial para poder mostrar cada uno de dichos ítems. Además, para poder mostrar dichos ítems en la lista, se ha de crear un ***Adapter***, cuyo funcionamiento se explica a continuación.

3.1.2.1. Adapters

Un objeto *Adapter* actúa como puente entre un *AdapterView* y los datos que contendrá esta vista. El *Adapter* proporciona acceso a los datos de los ítems, y también es responsable de crear una vista para cada ítem, del conjunto de datos. Por tanto, un *Adapter* es la vía mediante la cual un *ListView* recibe sus datos.

El *Adapter* es asignado al *ListView* mediante el método ***setAdapter*** que posee la misma. Por otra parte, se ha de implementar el método ***setOnItemClickListener***() de la lista que se desea mostrar, para indicar a la aplicación las acciones a realizar, cuando se seleccione un ítem de la lista.

El Adapter ha de ser implementado en una clase que extienda de alguna de las siguientes clases:

- BaseAdapter
- ArrayAdapter
- CursorAdapter

Para finalizar, dentro de la clase creada para implementar el *Adapter*, se ha de sobrescribir el método ***getView***(), en el cual se *inflará* el *Layout* definido, para representar la interfaz gráfica de cada ítem, es por tanto, en este método, donde se "dibujan" tantos ítem como elementos tenga la lista asociada.

3.1.3. Notificaciones push para sistemas Android

Android utiliza un sistema propio para el envío de notificaciones push, dicho sistema se denomina ***C2DM*** (Cloud to Device Messaging). C2DM es un servicio que ayuda a los desarrolladores en el envío de datos, desde los servidores a las aplicaciones; sobre dispositivos Android.
Las principales características de este servicio son:

- Permite que los servidores puedan enviar mensajes ligeros a las aplicaciones Android. El servicio de mensajería no está diseñado para el envío de grandes cantidades de datos, sin embargo, puede ser usado para informar a la aplicación que hay nuevos datos en el servidor, para que la aplicación pueda recuperarlos.
- No se necesita crear una aplicación que se esté ejecutando siempre, para recibir mensajes. El sistema despertará la aplicación, mediante la emisión de un *Intent*, cuando llegue el mensaje, siempre y cuando, se haya configurado en la aplicación el respectivo *Broadcast Receiver*, así como los permisos adecuados.
- Se requiere la utilización de una versión de Android posterior a la 2.2.
- Utiliza una conexión existente para los servicios de Google, lo cual obliga a los usuarios a configurar su cuenta de Google en sus dispositivos.

Para poder trabajar con el servidor de C2DM se necesitan los siguientes datos:

Sender ID: Es una cuenta de correo de Gmail, que estará asociada al desarrollador de la aplicación. Éste identificador será usado en el proceso de registro, para identificar una aplicación Android, a la cual se le permite enviar mensajes al dispositivo.

Application ID: Aplicación para la cual se hace el registro para la recepción de mensajes. La aplicación se identifica mediante el nombre del paquete del *manifest*. De esta forma, se asegura que los mensajes son enviados a la aplicación correcta.

Registration ID: un identificador emitido por los servidores del C2DM a la aplicación Android, para permitirles recibir mensajes. Cuando la aplicación obtenga este identificador, lo debe enviar al servidor de la aplicación, el cual lo utilizará para identificar cada dispositivo que ha registrado la aplicación, para la recepción de mensajes. En otras palabras, un identificador de registro está vinculado a una aplicación particular, que se estará ejecutando en un determinado dispositivo.

Cuenta de usuario de Google: Para que el C2DM pueda funcionar, el dispositivo tiene que estar logueado; con una cuenta de Google.

Sender Auth Token: Es un token de autorización, que se almacenará en el servidor de la aplicación, y que permite a éste, acceder a

los servicios de Google. Este token se ha de incluir en la cabecera de la solicitud POST, para el envío de mensajes.

Para poder hacer uso del C2DM, se han de declarar en el *manifest* de la aplicación los siguientes permisos:

- Permiso para que solo la aplicación en cuestión, pueda recibir los mensajes del servidor del C2DM:

```
<permission
    android:name="pros.activities.permission.C2D_MESSAGE"
    android:protectionLevel="signature" />

<uses-permission
    android:name="pros.activities.permission.C2D_MESSAGE" />
```

- Permiso para el registro y la recepción de mensajes desde el C2DM:

```
    <uses-permission
        android:name="com.google.android.c2dm.permission.RECEIVE"
    />
```

La descripción del código necesario para interactuar con el C2DM, se especificará en el capítulo de desarrollo (*Recepción de las notificaciones*).

3.2. REST

REST [21], define un conjunto de principios arquitectónicos por los cuales se diseñan *servicios web*, haciendo énfasis en los recursos del sistema, incluyendo, cómo se accede al estado de dichos recursos, y cómo se transfieren por *HTTP* hacia clientes escritos en diversos lenguajes. *REST* ha emergido en los últimos años como el modelo predominante para el diseño de servicios. De hecho, *REST* ha logrado un impacto tan grande en la web, que prácticamente ha desplazado a *SOAP* y las interfaces basadas en *WSDL*, por tener un estilo bastante más simple de usar.

REST no tuvo mucha atención, cuando Roy Fielding lo presentó por primera vez en el año 2000, en la Universidad de California, durante la charla académica "*Estilos de Arquitectura y el Diseño de Arquitecturas de Software basadas en Redes*", la cual analizaba un conjunto de principios arquitectónicos de software, para usar la Web como una plataforma de procesamiento distribuido. Ahora, años después de su presentación, comienzan a aparecer varios frameworks *REST* y se ha convertido en una parte integral de Java 6 a través de JSR-311. Los cuatro principios de *REST* son:

1. **Utiliza los métodos HTTP de manera explícita.**

Una de las características claves de los servicios web *REST*, es el uso explícito de los métodos *HTTP*, siguiendo el protocolo definido por *RFC 2616*. Por ejemplo, *HTTP GET* se define como un método productor de datos, cuyo uso está pensado para que las aplicaciones cliente obtengan recursos, busquen datos de un servidor web, o ejecuten una consulta, esperando que el servidor web la realice y devuelva un conjunto de recursos.

REST hace que los desarrolladores usen los métodos *HTTP* explícitamente, de manera que resulte consistente con la definición del protocolo. Este principio de diseño básico, establece una asociación *uno-a-uno*, entre las operaciones de crear, leer, actualizar y borrar, y los métodos *HTTP*. De acuerdo a esta asociación:

- Se usa *POST* para crear un recurso en el servidor
- Se usa *GET* para obtener un recurso
- Se usa *PUT* para cambiar el estado de un recurso o actualizarlo
- Se usa *DELETE* para eliminar un recurso

2. No mantiene estado.

Los servicios web *REST* necesitan escalar, para poder satisfacer una demanda en constante crecimiento. Se usan *clusters* de servidores con balanceadores de carga y alta disponibilidad, *proxies*, y *gateways* para conformar una topología útil/práctica, que permita transferir peticiones de un equipo a otro, para disminuir el tiempo total de respuesta de una invocación al *servicio web*. El uso de servidores intermedios para mejorar la escalabilidad, hace necesario que los clientes de servicios web *REST*, envíen peticiones completas e independientes, es decir, se deben enviar peticiones que incluyan todos los datos necesarios, para cumplir el pedido, de manera que los componentes en los servidores intermedios, puedan re-direccionar y gestionar la carga, sin mantener el estado localmente entre las peticiones.

Una petición completa e independiente, hace que el servidor no tenga que recuperar ninguna información de contexto o estado, al procesar la petición. Una aplicación o cliente de servicio web *REST* debe incluir, dentro del encabezado y del cuerpo *HTTP* de la petición, todos los parámetros, contexto y datos que necesita el servidor, para generar la respuesta. De esta manera, el no mantener estado, mejora el rendimiento de los servicios web y simplifica el diseño e implementación de los componentes del servidor, ya que la ausencia de estado en el servidor, elimina la necesidad de sincronizar los datos de la sesión, con una aplicación externa.

3. Despliega URIs con forma de directorios

Desde el punto de vista del cliente de la aplicación que accede a un recurso, la *URI* determina qué tan intuitivo va a ser el servicio web *REST*, y si éste va a ser utilizado tal como fue pensado al momento de diseñarlo. La tercera característica de los servicios web *REST* es justamente sobre las *URIs*.

Las *URIs* de los servicios web *REST* deben ser intuitivas, hasta el punto de que sea fácil adivinarlas. Se ha de pensar en las *URI* como una interfaz auto-documentada, que necesita de muy poca o ninguna, explicación o referencia, para que un desarrollador pueda comprender a lo que apunta, y a los recursos derivados relacionados.

Una forma de lograr este nivel de usabilidad, es definir *URIs* con una estructura al estilo de los directorios. Este tipo de *URIs* es jerárquica, con una única ruta raíz, y va abriendo ramas a través de las sub-rutas, para exponer las áreas principales del servicio. De acuerdo a esta definición, una *URI* no es solamente una cadena de

caracteres delimitada por barras, sino más bien, un árbol con padres e hijos organizados como nodos, por ejemplo:
http://www.miservicio.com/webservice/JSONApi/retrieveNotifications.php

4. Transfiere XML, JSON o ambos.

Estos son los dos formatos en los que un servicio web *REST*, puede mostrar los datos devueltos por el servidor.

3.3. JSON

JSON (*JavaScript Object Notation – Notación de Objetos de JavaScript*) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para las personas, mientras que para las máquinas es simple; interpretarlo y generarlo. Está basado en un subconjunto del lenguaje de programación *JavaScript, Standard ECMA-262 3rd Edition - Diciembre 1999*. JSON es un formato de texto, que es completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades, hacen que *JSON* sea un lenguaje ideal para el intercambio de datos. *JSON* está constituido por dos estructuras:

- Una colección de pares de ***nombre/valor***. En varios lenguajes es conocido como un **objeto**, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
- Una lista ordenada de ***valores***. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan, de una forma u otra. Es razonable que un formato de intercambio de datos, que es independiente del lenguaje de programación, se base en estas estructuras.

3.3.1. Representación de un JSON

Un ***objeto***, es un conjunto desordenado de pares ***nombre/valor***. Un *objeto* comienza con { (llave de apertura), y termina con } (llave de cierre). Cada nombre es seguido por : (dos puntos), y los pares *nombre/valor* están separados por , (coma).

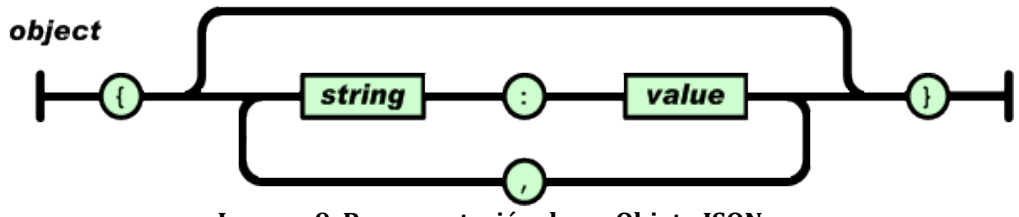

Imagen 9. Representación de un Objeto JSON

Un ***array*** es una colección de valores. Comienza con [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por , (coma).

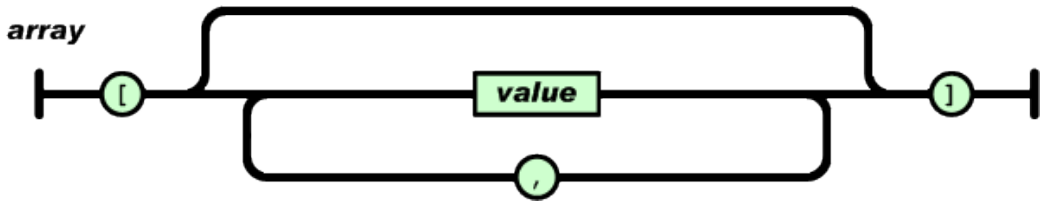

Imagen 10. Representación de un Array JSON

Un ***valor***, puede ser una **cadena de caracteres** con comillas dobles, un **número**, true, false o null, un objeto o un array. Estas estructuras pueden anidarse.

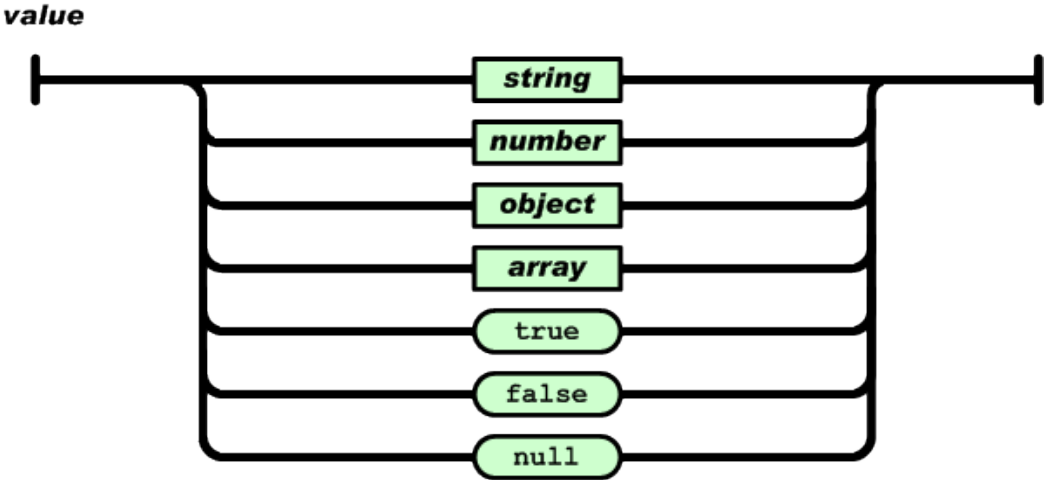

Imagen 11. Representación de un Valor JSON

Una ***cadena de caracteres***, es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter está representado por una cadena de caracteres; de un único carácter. Una cadena de caracteres en JSON, es parecida a una cadena de caracteres C o Java.

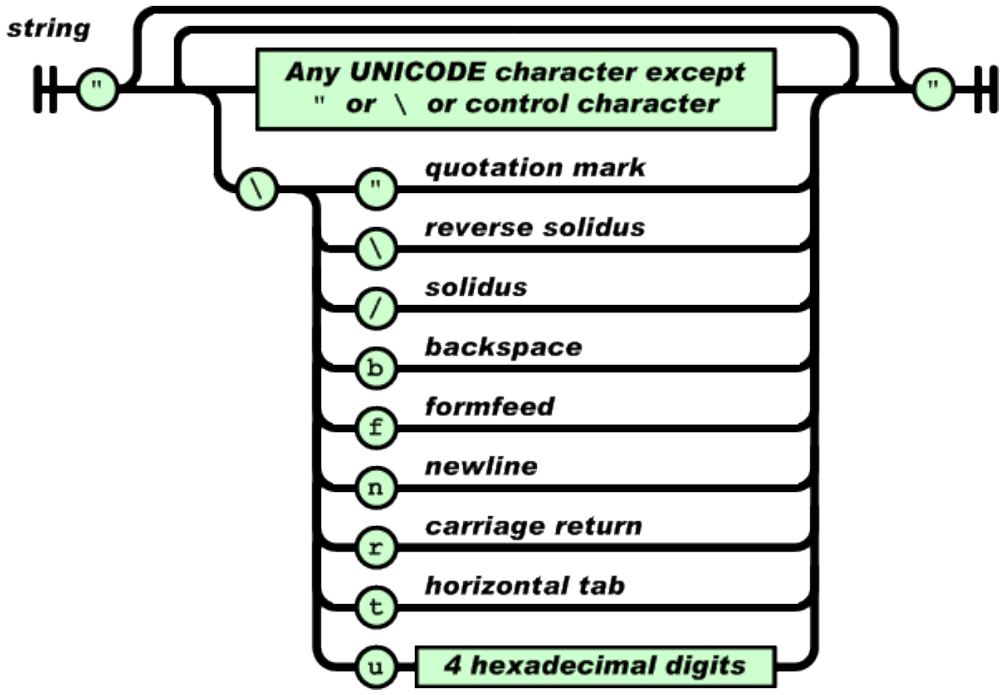

Imagen 12. Representación de una cadena de caracteres JSON

Un ***número***, es similar a un número C o Java, excepto que no se usan los formatos octales y hexadecimales.

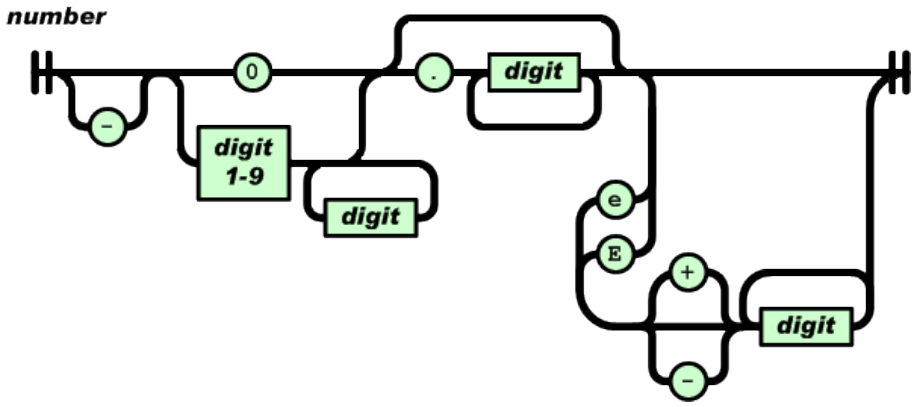

Imagen 13. Representación de un número JSON

Los espacios en blanco pueden insertarse entre cualquier par de símbolos.

Exceptuando pequeños detalles de *decodificación*, esto describe completamente el lenguaje. A continuación, se muestra un ejemplo de uno de los *JSON* devueltos, en una de las llamadas al servicio web utilizado en esta Tesina:

```
[{"id":"42","serviceID":"s6","service":"Weather","serviceIcon":"Weat
herServiceIcon.png","nTitle":"Sunny","nText":"Today is
sunny","readed":"1","creationDate":"2012-08-14 at 15:29:22"},
{"id":"41","serviceID":"s6","service":"Weather","serviceIcon":"Weath
erServiceIcon.png","nTitle":"Sunny","nText":"Today is
sunny","readed":"1","creationDate":"2012-08-13 at 08:01:12"}]
```

3.4. MoRE

MoRE (Model-based Reconfiguration Engine) [*1*] es un motor de reconfiguración, que proporcionan capacidades de autoconfiguración a un sistema.

MoRE está basado en los principios de la Computación Autonómica. Los diseñadores pueden proporcionar reglas de adaptación a *MoRE*, para indicar cuándo se debe reconfigurar la arquitectura. Si hay un cambio en el contexto que active una regla, *MoRE* ejecuta un conjunto de cambios en la arquitectura, y luego computa cuáles componentes se deben habilitar y cuáles no. Finalmente, ejecuta las acciones correspondientes, para obtener el estado deseado. La reconfiguración se ejecuta de una forma transaccional, habilitando el rollback de las acciones, en caso de fallo de la reconfiguración.

Se ha elegido *MoRE*, ya que es un motor genérico, que puede ser personalizado por medio de modelos. Esto hace posible su integración, dentro de la aproximación expuesta en esta Tesina, mediante el aprovechamiento de modelos existentes en el sistema. Para utilizar *MoRE* en una aplicación particular, Los diseñadores deben

proporcionar las reglas de adaptación y la descripción de la arquitectura, por medio de modelos. Más información sobre la infraestructura de *MoRE* se puede consultar en [6].

4. VISIÓN GLOBAL DE LA SOLUCIÓN

En este capítulo se presenta la **arquitectura del sistema** desarrollado en esta Tesina, así como una explicación general de su funcionamiento.

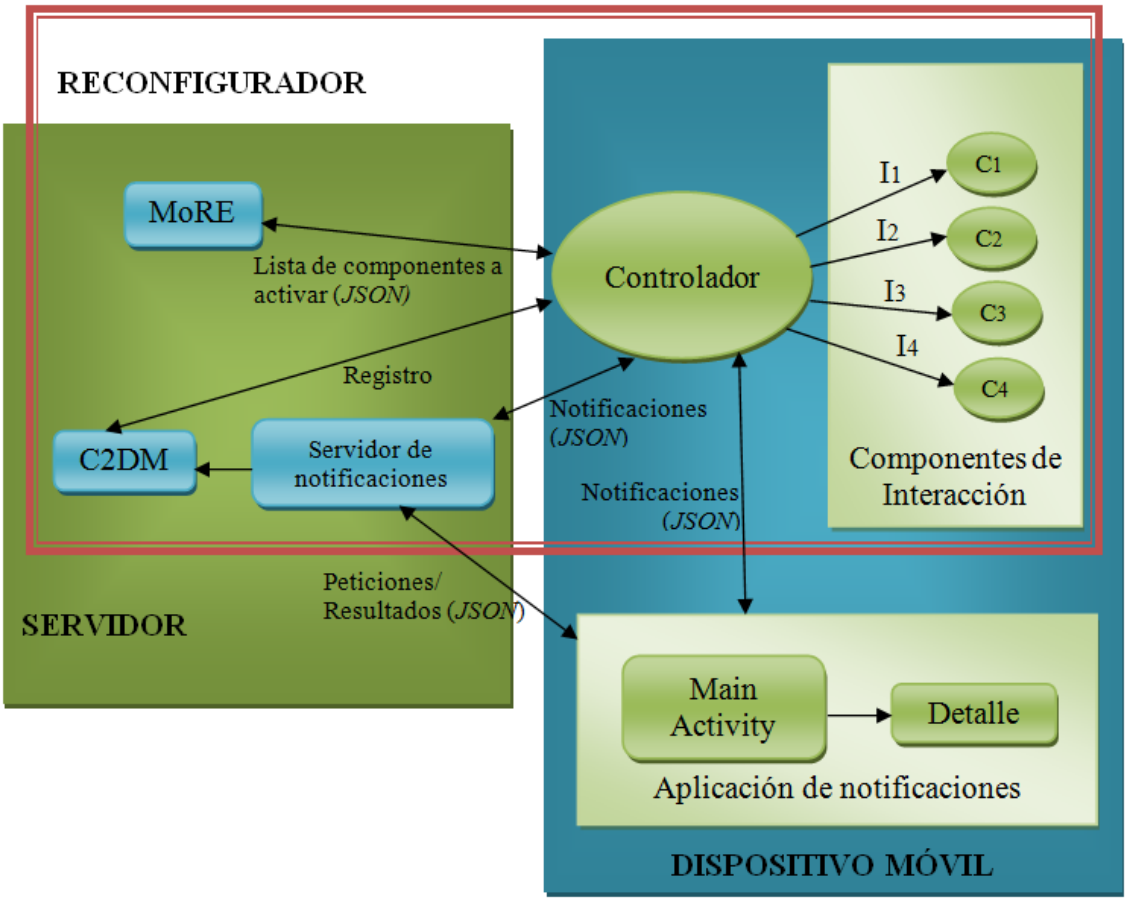

Imagen 14. Arquitectura del sistema implementado

A continuación se explica brevemente, cada una de las partes de las que se compone el sistema.

4.1. Componentes de interacción

Los componentes de interacción, son aquellos componentes Android que se utilizarán en esta Tesina, para avisar al usuario de la llegada de una nueva notificación,

dichos componentes son: *StatusBar*, *Dialog*, *Toast*, *Speech*, *Vibration*, *Lights* y *Sound*. Estos componentes se activan mediante *Intents*. En el capítulo de desarrollo se detalla el funcionamiento de cada uno de ellos, así como la forma en la que han sido implementados.

4.2. Reconfigurador

Como se puede ver en la Imagen 14; de la arquitectura del sistema ***Reconfigurador***, éste, está compuesto por componentes, que se encuentran tanto en el servidor como en el dispositivo móvil.

Para la recepción de las notificaciones en el dispositivo móvil, se hace uso del *servidor de notificaciones* push utilizado en los sistemas Android (*C2DM*). Se han creado unos servicios mediante *PHP* (en el *servidor de notificaciones*), con los que se puede enviar una notificación a la aplicación. Para ello, lo primero que se hace es crear una notificación en el *servidor de notificaciones*, y luego se realiza una conexión con el *C2DM*, siendo éste el encargado de notificar a la aplicación (mediante el *controlador*), que se ha recibido una nueva notificación. La aplicación recibirá la señal de que hay una nueva notificación, y la leerá.
Por otra parte, se dispone de un *controlador*, encargado de recibir las notificaciones y hacer las peticiones al motor de reconfiguración *MoRE* [*1*], el cual le devolverá un *JSON* con la configuración que indicará; qué componentes se deberán activar, para mostrar la notificación. Una vez recibida la configuración, el *controlador* se encargará de activar los *componentes de interacción* indicados, mediante *Intents* de activación, y la notificación se mostrará en el dispositivo de la forma indicada.

4.3. Aplicación de notificaciones

Se ha creado una aplicación Android, que permite ver el funcionamiento global del sistema. Al ejecutar la aplicación por primera vez, se mostrará en la pantalla del dispositivo una *Activity* (*Main Activity*), en la que se mostrarán todas las notificaciones. En este momento se ejecutan dos acciones:

- La primera acción, es el registro de la aplicación en el *C2DM*. Para ello, la aplicación crea un *Intent* de comunicación con este servicio, para registrar la aplicación, y así, permitir la recepción de notificaciones push. El *C2DM* devolverá a la aplicación un identificador de registro. Debido a que dicho identificador será utilizado para el envío de las notificaciones, por parte del *C2DM*, la aplicación realiza una llamada al *servidor de notificaciones*, para almacenar el citado identificador en un fichero.
- La segunda acción, es la llamada al *servidor de notificaciones*, para obtener todas las notificaciones. Cuando la aplicación realiza una petición *REST* al *servidor de notificaciones*, por ejemplo, para obtener todas las notificaciones, el *servidor de notificaciones* realiza las siguientes acciones:

  - Realiza una conexión con la base de datos en la que están almacenadas las notificaciones.

- Realiza un *select* sobre la base de datos para obtener los datos que la aplicación le ha solicitado, en este caso todas las notificaciones.
- Convierte cada uno de los registros devueltos por la *select*, en formato *JSON*.
- Finalmente devuelve a la aplicación el *JSON* con todas las notificaciones.

Una vez la aplicación recibe el *JSON* con las notificaciones, las procesa y las muestra en pantalla.

La aplicación también dispone de una vista, en la que se pueden mostrar solo las notificaciones que no han sido leídas, así como una vista de edición, en la que el usuario puede eliminar las notificaciones. Al seleccionar alguna de las notificaciones, se mostrará una nueva pantalla que muestra el detalle de la notificación seleccionada. Además, la aplicación está provista con la funcionalidad; de marcar una notificación como leída y como eliminada.

Para hacer la aplicación más interactiva con el usuario, se han creado *Dialogs*, para que el usuario pueda elegir, si realmente quiere eliminar una notificación o si desea salir de la aplicación. También está provista de objetos *Toast*, para informar al usuario de las acciones realizadas.

Cuando la aplicación recibe una nueva notificación, ésta aparecerá automáticamente, en la vista que muestra todas las notificaciones.

5. DESARROLLO

Como se ha especificado en la introducción, para cumplir los objetivos de esta Tesina de Máster, su implementación ha sido desarrollada en tres grandes fases, por tanto este capítulo está dividido en tres sub-apartados. En cada uno de ellos se explica de una forma detallada, como se han implementado cada uno de los subsistemas; que componen el sistema global desarrollado. Además, se expone, en un cuarto sub-apartado, los aspectos más relevantes, de la forma en la que se ha implementado el sistema que engloba todo el ***Reconfigurador***.

Para evitar poner grandes cantidades de código (en este documento), relacionadas con la explicación de las implementaciones realizadas, en este capítulo se omitirán los comentarios y la parte de código relacionada con el tratamiento de errores. Por tanto, si se desea ver el código completo, habrá que remitirse a los ficheros de implementación.

5.1. Componentes genéricos

Para cumplir el primero de los objetivos de esta Tesina, lo primero que se ha realizado es un estudio en profundidad, sobre cómo crear los componentes (*widgets*) Android de una forma abstracta, para que puedan ser activados desde cualquier parte del sistema (en este caso, desde un controlador, que se creará para dicho objetivo).

Los componentes (*widgets*) Android que se han implementado de una forma genérica, para lograr los objetivos de esta Tesina son:

- Grupo de Alertas:
  - Vibration
  - Lights
  - Sound
- AlertDialog
- StatusBar
- Toast
- Speech

Para poder desarrollar, de una forma genérica, los componentes citados anteriormente, se han tenido que implementar mediante *Services* o *Activities*, dependiendo de su funcionalidad. Mediante *Services* aquellos componentes para los que no es necesario crear una interfaz de usuario, y mediante *Activities* para aquellos que la necesitan, como es el caso de los Dialogs.

Android proporciona los *Intents* para poder activar componentes, por tanto, esta será la forma como los componentes genéricos (creados para esta Tesina) podrán ser activados, desde cualquier parte del sistema. Además, será mediante estos mismos *Intents*, con los que cada componente, podrá recibir los datos adicionales que necesitan, para poder ejecutarse. A continuación se explicará el funcionamiento de cada uno de ellos, así como la forma como se han desarrollado, para hacerlos genéricos.

5.1.1. Grupo de Alertas

Este grupo está compuesto por tres componentes: *Vibration*, *Lights* y *Sound*. Debido a que estos componentes no necesitan una interfaz de usuario, han sido implementados mediante un *Service*. Los tres componentes se han implementado en un único *Service* (***Alertas***), por tanto, para poder ejecutar cada uno de ellos, se deberá especificar en el *Intent* de activación, cuál de los tres componentes es el que se desea activar, y por tanto, este atributo será lo primero que se leerá, antes de poder activar cada uno de ellos.

A continuación, antes de especificar los detalles de implementación de cada una de las Alertas, se muestra el código que se ejecuta al inicio del *Service*, creado para la implementación de estos componentes.

```
tipoAlerta = Intent.getIntExtra("type", 0);
if(tipoAlerta == VIBRATION)
    runVibration();
else if(tipoAlerta == LIGHTS)
    showLights();
else if(tipoAlerta == SOUND){
    sonidoValue = Intent.getStringExtra("sound");
    playSound();
}
```

5.1.1.1. Vibration

La vibración (*Vibration*) como su propio nombre indica, es el componente que permitirá que el dispositivo vibre, cuando se reciba una nueva notificación. Éste no necesita de ningún parámetro adicional, lo único que se ha hecho es invocar a la vibración del sistema, y se le ha especificado un tiempo de 1seg (duración de vibración).

Para la utilización de este componente se ha tenido que activar el siguiente permiso en el *manifest*:

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

El código necesario para la activación de este componente es el siguiente:

```
Vibrator mVibrator = (Vibrator)
          getSystemService(Context.VIBRATOR_SERVICE);
long milliseconds = 1000;
   mVibrator.vibrate(milliseconds);
```

5.1.1.2. Lights

Las luces (*Lights*) son un componente visual, que se muestran en forma de luz parpadeante, para informar al usuario de la llegada de una nueva notificación.

Éstas, se han de mostrar a través del flash de la cámara del dispositivo, por tanto en el *manifest*, a parte del permiso de las luces, se ha tenido que poner el permiso; para la utilización de la cámara:

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

En este caso lo que se ha hecho es crear un *Thread*, para que la aplicación "duerma" durante 0.5seg, y así poder dar una sensación de parpadeo de las luces.

El código de implementación de este componente es el siguiente:

```
Camera mycam = Camera.open();
Parameters p = mycam.getParameters();
p.setFlashMode(Parameters.FLASH_MODE_TORCH);
mycam.setParameters(p);
mycam.startPreview();
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
    e.printStackTrace();
}
    p.setFlashMode(Parameters.FLASH_MODE_OFF);
mycam.stopPreview();
mycam.release();
```

Como se puede ver en el código, para desconectar y liberar la cámara, se realiza la llamada al método *release*().

5.1.1.3. Sound

El sonido (*Sound*), como se deduce de su nombre, es uno de los componentes sonoros, proporcionado por el Sistema Android, para notificar al usuario de un dispositivo, sobre la recepción de un nuevo mensaje. Para este caso se hace uso de los Threads en Android, para ello se ha usado como referencia la entrada: "Threads en Android, Mostrar ProgressDialog durante GET Request" [14], del blog de Jesús Gallardo.

Este componente ha sido implementado mediante la clase *MediaPlayer*. Como archivo de sonido se le pasa la alarma por defecto del sistema, aunque éste ha sido implementado para que pueda recibir un archivo de sonido como parámetro adicional.

```
sonidoValue = Intent.getStringExtra("sound");
```

A continuación se muestra el código necesario para su activación:

```
Uri myUri =
RingtoneManager.getDefaultUri(RingtoneManager.TYPE_ALARM);
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_ALARM);
try {
    mediaPlayer.setDataSource(getApplicationContext(),
    myUri);
} catch (IllegalArgumentException e) {
    …Tratamiento de errores
}
try {
    mediaPlayer.prepare();
} catch (IllegalStateException e) {
    …Tratamiento de errores
}
mediaPlayer.start();
WaitThread dormir = new WaitThread();
dormir.start();

while (continuar) {
}
mediaPlayer.stop();
```

Como se puede ver, se ha creado un *Thread* ***WaitThread***(), que permite que el sonido proporcionado se reproduzca durante 5 seg. Dicho *Thread* se ha implementado de la siguiente manera:

```
private class WaitThread extends Thread {
    public void run() {
        try {
            synchronized(this){
                wait(5000);
            }
        } catch (InterruptedException e) {
            …Tratamiento de errores
        }
    continuar = false;
    }
}
```

Para finalizar, decir que para este componente, no hace falta declarar ningún permiso especial en el *manifest*.

5.1.2. Dialogs

Aunque en las notificaciones creadas para mostrar la funcionalidad del ***Reconfigurador***, solo existe como componente de este tipo, el *AlertDialog*, Android proporciona cuatro tipos de *Dialog*, y todos han sido implementados de forma genérica en esta Tesina. Debido que los *Dialogs* necesitan de una interfaz de usuario, y además necesitan proporcionar la funcionalidad de interacción con el usuario, estos han tenido que ser implementados mediante una *Activity*. Al igual que el grupo de alertas, todos los *Dialogs* han sido implementados en una sola clase (***DialogComponent***), por tanto, para poder ejecutar cada uno de ellos hay que indicar en el *Intent* de activación, un atributo para saber cuál es la que se desea activar. Por ello, lo primero que se tiene en la *Activity* que los implementa, es lo siguiente:

```
Intent = getIntent();
tipo = Intent.getIntExtra("tipoDialog", 0);
```

Cada uno de los *Dialogs* implementados, ha de recibir, mediante el *Intent* que los activa, los datos necesarios para su correcta ejecución. En los apartados siguientes, se detallan los atributos necesarios para cada uno de ellos. Como referencia de la recepción de dichos atributos, a continuación se muestra como se reciben los datos del *AlertDialog*, que por su parte, como se ha dicho antes, es el único tipo de *Dialog* que se ha tenido en cuenta, para probar el funcionamiento del ***Reconfigurador*** creado en esta Tesina.

```
switch (tipo) {
case ALERT_DIALOG_ID:
    idNot = Intent.getStringExtra("notID");
    titulo = Intent.getStringExtra("title");
    mensaje = Intent.getStringExtra("message");
    positiveButton = Intent.getStringExtra("positiveBtn");
    negativeButton = Intent.getStringExtra("negativeBtn");
    neutralButton = Intent.getStringExtra("neutralBtn");
    showDialog(ALERT_DIALOG_ID);
break;
    …Otros Dialogs
```

Cuando se trabaja con *Dialogs* en una *Activity* se ha de implementar el método *onCreateDialog()*, para poder implementar la funcionalidad de los mismos. Dicho método se ejecutará después de invocar a un *Dialog*, mediante el método *showDialog*().

```
protected Dialog onCreateDialog(int id) {
    Dialog dialog;
    switch(id) {
    case ALERT_DIALOG_ID:
        return mAlertDialog();
    …Otros Dialogs
```

Los Dialogs proporcionados por Android y que han sido desarrollados de forma genérica, son los siguientes:

5.1.2.1. AlertDialog

Un *AlertDialog* es un tipo de *Dialog* que permite la interacción con el usuario, para que éste pueda realizar acciones tales como: elegir elementos de una lista, preguntar al usuario si desea realizar una determinada acción, proporcionando de funcionalidad a los botones, entre otras muchas posibilidades.

Debido a que este tipo de *Dialog* puede recibir diferentes atributos, para la realización de esta Tesina, se han dividido en tres tipos diferentes, puesto que la implementación de cada uno de ellos; requiere de algunos cambios (aunque son muy pequeños). Dichos tipos son:

- *AlertDialog* que puede gestionar hasta 3 botones.
- *AlertDialog* que puede gestionar una lista de ítems seleccionables

- *AlertDialog* que puede gestionar una lista con *checkboxes* o *radio buttons*.

Para crear un *AlertDialog* se ha de usar la subclase *AlertDialog.Builder*, con la que se puede definir todas las propiedades del *Dialog*.

5.1.2.1.1. AlertDialog con botones

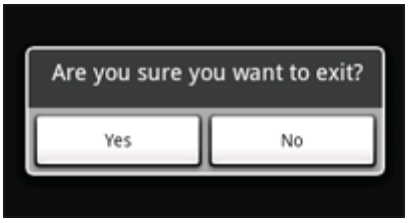

Imagen 15. AlertDialog con botones

Para este tipo de *Dialog* se ha de pasar a la *Activity* que lo implementa los siguientes datos:

- Un título
- Un mensaje de Texto
- El texto de uno, dos o tres botones

5.1.2.1.2. AlertDialog con lista

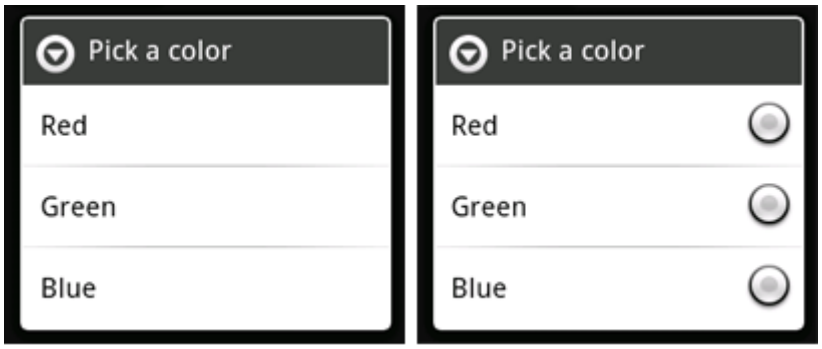

Imagen 16. AlertDialog con lista

Para este tipo de *Dialog* se ha de pasar a la *Activity* que lo implementa los siguientes datos:

- Un título
- Un Array con los datos de la lista a mostrar

El *Alert Dialog* que se muestra como una lista pero que contiene *checkboxes* o *radio buttons*, se ha implementado como un componente diferente, debido a que su funcionalidad es distinta.

El *Builder* (creado para la implementación de cada uno de los diferentes *AlertDialog* especificados anteriormente), proporciona unos métodos que permite asignar los datos adicionales, necesarios para cada *Dialog* (recibidos en la activación). A continuación se muestra la asignación de los atributos del *AlertDialog*.

```
builder = new AlertDialog.Builder(this);
builder.setTitle(titulo);
builder.setMessage(mensaje);
builder.setOnKeyListener(new
    DialogInterface.OnKeyListener() {
    @Override
    public boolean onKey(DialogInterface dialog, int keyCode,
    KeyEvent event) {
        if(keyCode == KeyEvent.KEYCODE_BACK) {
            …Funcionalidad al pulsar el botón de retorno
              (back button)
            return true;
        }
        return false;
    }
});
if(positiveButton.length() != 0){
    builder.setNegativeButton(negativeButton, new
    DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            …Funcionalidad al pulsar este botón
        }
    });
}
if(negativeButton.length() != 0)
    …Funcionalidad
if(neutralButton.length() != 0)
    …Funcionalidad
alert = builder.create();
alert.show();
```

Se puede ver que aparte de asignar los valores recibidos en la activación, a los diferentes componentes del *Dialog*, se ha de implementar la funcionalidad de sus botones, por ejemplo, ir a otra *Activity*, cerrar el *Dialog*, etc.
Para finalizar, se utiliza la llamada al método *show*(), para mostrar el *Dialog* al usuario.

5.1.2.2. ProgressDialog

Un *ProgressDialog* es el tipo de *Dialog* que se ha de emplear; cuando se necesita mostrar al usuario una pequeña ventana, que indique que se está ejecutando una acción, como por ejemplo, la descarga o la subida de un fichero de Internet.

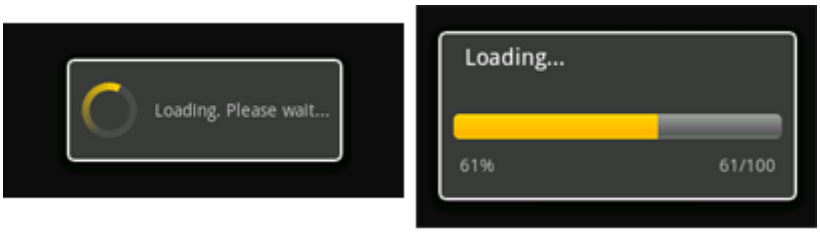

Imagen 17. ProgressDialog

Existen dos tipos de *ProgressDialog*. Los que se muestran en forma de barra y los que se muestran en forma de rueda. Al igual que se ha hecho con los *AlertDialog*, estos se han creado como dos componentes distintos, debido a que su implementación es distinta. Aunque la implementación de los dos es distinta, solo se les ha de pasar un dato: *el mensaje*. Ambos se crean con la clase *ProgressDialog*.

```
progressDialog = new ProgressDialog(this);
```

La implementación de la asignación de los atributos, se hace de una forma similar como se hace para los *AlertDialog*:

```
progressDialog.setProgressStyle
    (ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setMessage(titulo);
```

Para cada uno de estos *Dialogs* se ha creado un *Thread*, que se ejecutará para realizar el proceso del paso del tiempo, que tienen que estar visibles. Dichos *Threads* se crean en el método *onPrepareDialog*(), el cual se ha sobrescrito, ya que este método se ejecuta antes de que el *Dialog* sea mostrado al usuario.

5.1.2.3. DatePickerDialog

Imagen 18. DatePicker Dialog

Éste es un *Dialog* que permite al usuario seleccionar una fecha. Este tipo de *Dialog* no requiere ningún dato adicional. Para su creación se ha de crear un objeto *DatePickerDialog*. Para completar el proceso, se sobrescribe el método *onDateSet*(), para que una vez que se haya elegido la fecha y se pulse el botón *Set*, los valores de fecha seleccionados, sean enviados vía un *Intent*, al componente

que haya hecho la llamada al *Dialog* en cuestión. El código de su implementación es el siguiente:

```
DatePickerDialog mDatePickerDialog;
final Calendar c = Calendar.getInstance();
int mYear = c.get(Calendar.YEAR);
int mMonth = c.get(Calendar.MONTH);
int mDay = c.get(Calendar.DAY_OF_MONTH);
mDatePickerDialog = new DatePickerDialog(this,
    mDateSetListener, mYear, mMonth, mDay);
mDatePickerDialog.setOnKeyListener(new
    DialogInterface.OnKeyListener() {

    @Override
    public boolean onKey(DialogInterface dialog, int keyCode,
        KeyEvent event) {
        if(keyCode == KeyEvent.KEYCODE_BACK) {
            …Funcionamiento al pulsar el botón de retorno
            return true;
        }
        return false;
    }
});
return mDatePickerDialog;

private DatePickerDialog.OnDateSetListener
    mDateSetListener = new
    DatePickerDialog.OnDateSetListener() {
        public void onDateSet(DatePicker view, int year, int
        monthOfYear, int dayOfMonth) {
            …Funcionamiento al pulsar el botón Set
        }
};
```

5.1.2.4. TimePickerDialog

Imagen 19. TimePicker Dialog

Este es un *Dialog* que permite al usuario seleccionar una hora. Su implementación se ha realizado de la misma forma que se ha hecho el *DatePikerDialog*, con la diferencia que el objeto a crear es un *TimePickerDialog* y el método a sobrescribir el *onTimeSet*.

5.1.3. StatusBar

Una *StatusBar* es un componente que informa al usuario, de la ocurrencia de un determinado evento en el sistema, de la siguiente forma:

- Añade un icono a la barra de estado del sistema. Opcionalmente, dependiendo de la implementación, se mostrará un número sobre el icono, si la misma notificación ha sido generada varias veces.
- Añade un mensaje en la ventana de notificaciones. Este mensaje puede estar compuesto por varios elementos, tales como, la hora, un título, etc. Cuando el usuario selecciona este mensaje, Android lanza un *Intent*, que ha sido definido por la notificación (normalmente, lanza una *Activity*).

Además, se puede configurar la notificación, para que alerte al usuario con un sonido, una vibración y luces parpadeantes en el dispositivo.

Imagen 20. Status Bar con una notificación

Imagen 21. Ventana de notificaciones

Debido a que un *StatusBar* no necesita una interfaz gráfica, porque la proporciona el propio sistema, la implementación de este componente genérico, se ha hecho mediante un *Service* (***StatusBarNotification***). Los datos que se le han de pasar mediante el *Intent* de activación son:

- Icono
- Título
- Mensaje

- La clase destino, es decir, la clase que contiene el *Activity* al que se ha de ir, cuando se seleccione la notificación en la ventana de notificaciones.
- Un id de notificación.
- El número de veces que se ejecuta la misma notificación. Este se ha de informar para que si se ejecuta la misma notificación varias veces, esta no aparezca duplicada en la barra de estado sino que por el contrario aparezca una sola vez. De esta forma, sobre el icono aparecerá un número que indicará el número de veces que ha sido ejecutada dicha notificación.
- El sonido
- La vibración
- Las luces

Puesto que el sonido, la vibración y las luces se han tratado como componentes individuales, cuando se ejecute una *StatusBar* en esta Tesina dichos datos nunca estarán informados, por tanto nunca se activarán. Por lo cual, si por ejemplo, se desea activar la vibración, se habrá de activar el componente genérico creado para realizar dicha acción. A continuación se muestra el código de la recepción de los atributos que la componen:

```
iconValue = Intent.getIntExtra("icon", 0);
tittleValue = Intent.getStringExtra("title");
textValue = Intent.getStringExtra("message");
nomClase = Intent.getStringExtra("clase");
if(!nomClase.isEmpty()){
    targetClass = Intent.getStringExtra("clase").getClass();
}
NOTIF_ID = Intent.getIntExtra("id", 1);
numNot = Intent.getIntExtra("cont", 1);
ttValue = Intent.getCharSequenceExtra("tickerText");

if(Intent.getStringExtra("sound")==null)
    sonidoValue = "";
else
    sonidoValue = Intent.getStringExtra("sound");

vibracionValue = Intent.getBooleanExtra("vibration", false);
lucesValue = Intent.getBooleanExtra("flashLight", false);
```

Para crear el *StatusBar*, se ha creado una instancia de la clase *NotifcationManager*, para poder notificar al usuario cuando se ejecute la notificación. Por otro lado, para crear la notificación propiamente dicha, se ha creado un objeto *Notificacionk*, al que se le han de pasar todos los datos informados por el *Intent*. Para finalizar se pasa la notificación creada al *NotificationManager* y esta es mostrada en el dispositivo realizando la llamada al método *notify*(). A continuación, se muestra el código necesario para su activación:

```
String ns = Context.NOTIFICATION_SERVICE;
mNotificationManager = (NotificationManager)
    getSystemService(ns);

long when = System.currentTimeMillis(); // notification time
notification = new Notification(iconValue, ttValue, when);

Context context = getApplicationContext();
if(nomClase.isEmpty())
```

```
    notificationIntent = new Intent();
else{
    notificationIntent = new Intent(context,
        DetailActivity.class);
}

PendingIntent contentIntent = PendingIntent.getActivity(context,
0, notificationIntent, 0);

notification.setLatestEventInfo(context, tittleValue, textValue,
    contentIntent);

notification.number = numNot;
notification.flags = Notification.FLAG_AUTO_CANCEL;

if(!sonidoValue.isEmpty()){
    if( sonidoValue.equals("DEFAULT_SOUND") ){
        notification.defaults |= Notification.DEFAULT_SOUND;
    }
    else{
        notification.sound = Uri.parse(sonidoValue);
    }
}

if(vibracionValue==true){
    notification.defaults |= Notification.DEFAULT_VIBRATE;
}

if(lucesValue==true){
    notification.defaults |= Notification.DEFAULT_LIGHTS;
}

mNotificationManager.notify(NOTIF_ID, notification);
```

Se puede ver que los atributos como la vibración, luces, etc. solo se activan si vienen informadas. Como referencia, para saber cómo abrir una Activity al pulsar sobre una notificación, en el centro de notificaciones del dispositivo, se ha tomado la entrada: "Show activity only once when clicking the Notification icon" [20], de la Web stackoverflow.com.

5.1.4. Toast

Un *Toast* es un mensaje que aparece en la superficie de la ventana, y que sirve para informar al usuario sobre algún evento ocurrido en el sistema. Éste solo ocupa el espacio requerido por el mensaje en la pantalla, y la *Activity* actual permanece visible, e interactiva al usuario. La notificación aparece y desaparece automáticamente, y no acepta eventos de interacción.

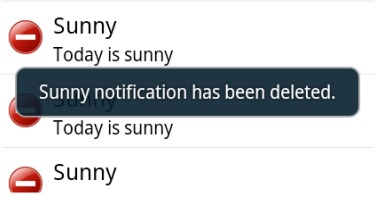

Imagen 22. Toast Notification

Debido a que este tipo de componente no permite la interacción con el usuario, y por tanto, no se ha de proporcionar una interfaz de usuario condicionada, la implementación de este componente genérico se ha realizado mediante un *Service* (***ToastNotification***).

El único dato que se le ha de pasar es el *mensaje*. Para mostrarlo, se ha creado un objeto de la clase *Toast*, y se muestra al usuario llamando al método *show*(). Su implementación es la más sencilla de todos los componentes genéricos creados y se muestra a continuación:

```
toastText = Intent.getStringExtra("message");
Toast toast = Toast.makeText(getApplicationContext(), toastText,
Toast.LENGTH_LONG);
toast.show();
```

5.1.5. Speech

El Speech es una funcionalidad que poseen los sistemas Android, y que se construye, a partir de una clase proporcionada por el sistema, para equipar una aplicación con dicha funcionalidad. Se ha utilizado como referencia el tutorial: "Android Text to Speech" [19], de la Web androidhive.info.

Éste componente se ha creado mediante un *Service*, ya que no se necesita mostrar ninguna interfaz. El Service creado para este propósito es: *SpeechAlertService*. El *service* implementa la interfaz *OnInitListener*, ya que es en esta interfaz, en donde está definido el método que se ejecuta al crear el objeto *TextToSpeech*, utilizado para crear el Speech.

```
public class SpeechAlertService extends Service implements
  Constantes, OnInitListener {
```

Para crear el *Speech*, lo único que se ha de hacer es crear un objeto *TextToSpeech*, y ejecutar el método *speak*(); en el método *onInit*(), todo lo demás, son comprobaciones del idioma y demás. Además, como es lógico, se necesitará el mensaje de la notificación para reproducirlo, éste se recibirá del mediante el intent de activación. El código se muestra a continuación:

```
private TextToSpeech tts;
String texto;

@Override
public int onStartCommand(Intent intent, int flags, int
startId){
    texto = intent.getStringExtra("mensaje");
    tts = new TextToSpeech(this, this);
    return START_NOT_STICKY;
}

@Override
public void onInit(int status) {
    if (status == TextToSpeech.SUCCESS) {
        int result = tts.setLanguage(Locale.US);
        if (result == TextToSpeech.LANG_MISSING_DATA
            || result == TextToSpeech.LANG_NOT_SUPPORTED){
```

```
                Toast.makeText(getApplicationContext(), "This Language
                    is not supported", Toast.LENGTH_LONG).show();
            }
            else
                tts.speak(texto, TextToSpeech.QUEUE_FLUSH, null);
        } else{
            Toast.makeText(getApplicationContext(), "Initilization
                Failed!", Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public void onDestroy() {
        // Don't forget to shutdown tts!
        if (tts != null) {
            tts.stop();
            tts.shutdown();
        }
        super.onDestroy();
    }
```

Para finalizar, al destruir el servicio, se liberan los recursos usados por el motor del *TextToSpeech*.

5.2. Reconfigurador

El ***Reconfigurador*** es el componente que permite recibir una notificación y mostrarla al usuario de la forma más indicada. Dicha forma es recibida de la integración con *MoRE* [*1*], mediante un fichero de configuración.

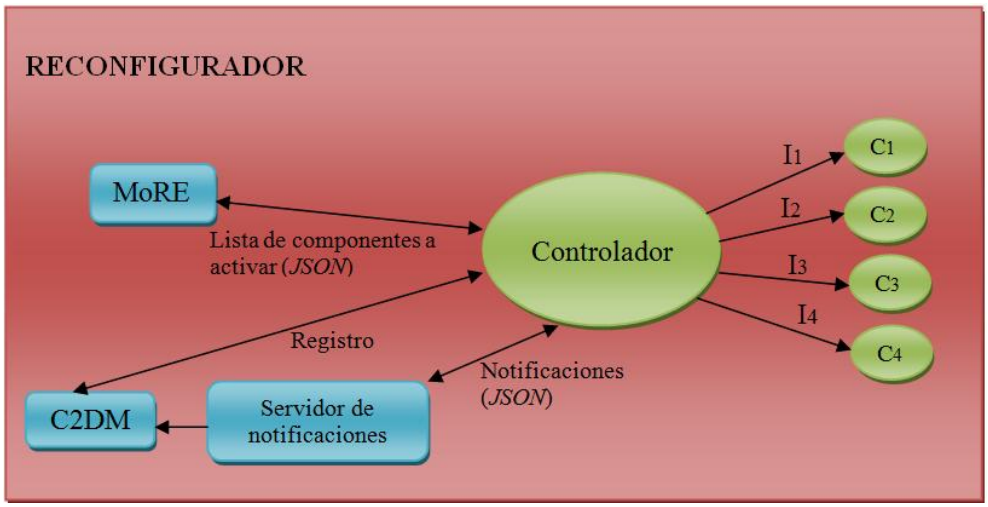

Imagen 23. Arquitectura del Reconfigurador

La implementación se ha dividido en los siguientes dos procesos:

1. Servidor de notificaciones
2. Implementación del *controlador*, el cual incluye la integración con *MoRE* [*1*] y la activación de los componentes de interacción.

A continuación se especificarán los detalles de implementación, de cada uno de los procesos listados anteriormente.

5.2.1. Servidor de notificaciones

Debido a que el servidor de notificaciones sirve, tanto para enviar nuevas notificaciones, como para atender peticiones de la *aplicación de notificaciones*, este apartado se divide en los siguientes dos sub-apartados:

- Envío de una nueva notificación
- Servicios *REST*

5.2.1.1. Envío de una nueva notificación

Puesto que una notificación no se puede recibir si antes no ha sido enviada, para la realización de esta Tesina, se ha creado un servicio web *RESTful* para el envío de nuevas notificaciones.

Existe una interfaz web, que ha sido adaptada para el trabajo en esta Tesina, que permite el envío de notificaciones:

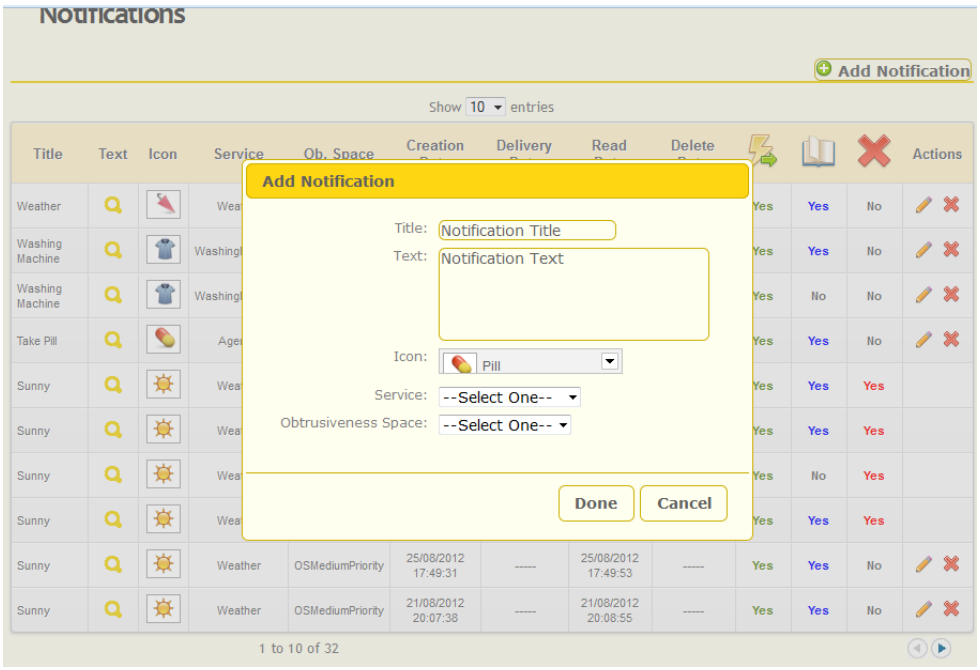

Imagen 24. Interfaz Web para el envío de notificaciones

Se ha creado un servicio web definido en el fichero ***sendMessage.php*** que se ejecutará al pulsar el botón ***Done***.

```
if(op=='ADD'){
    $.ajax({
        type: "POST",
        url: "sendMessage.php",
```

El fichero ***sendMessage.php*** realiza las siguientes acciones:

1. Crea una notificación con los datos; con los que se ha trabajado en esta Tesina.

```
$notDao = DAOFactory::getNotificationDAO();
$not = new Notification();
$not->id = $n + 1;
$not->nTitle = $_POST["title"];
$not->nText = $_POST["text"];
$not->serviceId = $_POST["serviceID"];
$not->obtrusivenessSpaceId = $_POST["osID"];
$not->icon = $_POST["icon"];
$not->deleted = 0;
$not->sent = 1;
$not->readed = 0;

$mysqldate = date('Y-m-d');
$mysqlTime = date('H:i:s');
$not->creationTime = $mysqlTime;
$not->creationDate = $mysqldate;
$not->deleteDate = '0000-00-00';
$not->readDate = '0000-00-00';
$not->deliveryDate = '0000-00-00';
$not->deleteTime = '00:00:00';
$not->readTime = '00:00:00';
$not->deliveryTime = '00:00:00';
$not->considerateSystemID =
    $_SESSION['considerateSystem']->id;
```

Dichos datos representan lo siguiente:

nTitle: Título de la notificación
nText: Mensaje de la notificación
serviceId: Identificador del servicio (se llama servicio porque cada notificación representa un servicio prestado).
obtrusivenessSpaceId: Identificador del espacio de molestia
icon: Icono de la notificación
deleted: (0/1) indica si la notificación está eliminada o no.
Sent: (0/1) indica si la notificación ha sido enviada o no.
Readed: (0/1) indica si la notificación ha sido leída o no.
creationTime: Hora de creación
creationDate: Fecha de creación
deleteDate: Fecha de eliminación
readDate: Fecha de lectura
deliveryDate: Fecha de entrega
readTime: Hora de lectura
deliveryTime: Hora de envío
considerateSystemID: Identificador del sistema (Este identificador se utiliza porque el sistema está pensado para poder trabajar tanto con dispositivos Android como con dispositivos Iphone).

2. Una vez que se ha creado la notificación, ésta se codifica al formato *JSON*.

```
$mensaje = json_encode($not);
```

3. Seguidamente, se lee el fichero *registerID.txt*, que contendrá el identificador de registro, proporcionado por el *C2DM* a la *aplicación de notificaciones*.

```
$manejador = fopen("registerID.txt","r");
$registrationID = fgets($manejador);
fclose($manejador);
```

4. Después de esto, se hace la llamada al *C2DM*, para obtener el token necesario, para poder enviar la notificación al ***controlador***.

```
$tooken = googleAuthenticate("cuenta_google", "contraseña",
$source="Company-AppName-Version", $service="ac2dm");
```

5. Con esto tenemos los datos requeridos, para poder enviar la notificación al ***controlador***, y se hace de la siguiente manera:

```
$re=sendMessageToPhone($tooken, $registrationID,
"collapsed", json_encode($not));
```

6. Para finalizar, una vez hecho esto, se inserta el registro creado en la base de datos (tabla *Notification*).

```
$newNotID = $notDao->insert($not);
```

5.2.1.2. Servicios REST

Se han creado los siguientes servicios *REST*, para poder proporcionar una mayor funcionalidad, a la aplicación de notificaciones:

5.2.1.2.1. retrieveNotifications

Propósito: Devolver a la aplicación todas las notificaciones.
Realiza un *select* de todas las notificaciones (almacenadas en la tabla *Notifications* de la base datos), que hayan sido enviadas (campo ***sent=1***) y que no hayan sido eliminadas (campo ***deleted=0***). Devuelve dichas notificaciones en un *JSON*.

```
<?php
…Declaraciones

$cs = $_SESSION['considerateSystem'];
$notDAO = DAOFactory::getNotificationDAO();
$searchResults = $notDAO->queryAllOfCSOrderBy('creation_date
DESC, creation_time DESC');
$count = count($searchResults);
$res = array();
for ($i = 0; $i < $count; $i++) {
    $thisNot = $searchResults[$i];
    if (!$thisNot->deleted&&$thisNot->sent) {
```

```
            $index = count($res);
            $res[$index]['id'] = $thisNot->id;
            $res[$index]['serviceID'] = $thisNot->serviceId;
            $res[$index]['service'] =
                $cs->getServiceByID($thisNot->serviceId)->name;
            $res[$index]['serviceIcon'] =
                $cs->getServiceByID($thisNot->serviceId)->icon;
            $res[$index]['nTitle'] = $thisNot->nTitle;
            $res[$index]['nText'] = $thisNot->nText;
            $res[$index]['readed'] = $thisNot->readed;
            $res[$index]['creationDate'] =
                $thisNot->creationDate." at ".
                $thisNot->creationTime;
        }
    }
    print(json_encode($res));
    ?>

    public function queryAllOfCSOrderBy($orderColumn) {
        $sql = 'SELECT * FROM Notification WHERE
            considerateSystemID = ? ORDER BY ' . $orderColumn;
        $sqlQuery = new SqlQuery($sql);
        if (!isset($_SESSION)) {
            session_start();
        }
        $sqlQuery->set($_SESSION['csID']);
        return $this->getList($sqlQuery);
    }
```

5.2.1.2.2. setNotificationReaded

Propósito: Marcar una notificación como leída.
Realiza un *update* (en la tabla *Notifications* de la base datos), del servicio que se pasa como parámetro de entrada (se pasa el id de la notificación), y se pone el atributo ***readed=1***. Además, actualiza los campos de fecha y hora de lectura, y finalmente devuelve un *JSON* con el resultado *OK*.

```
    <?php
    include_once '../include_dao.php';
    $notDAO=DAOFactory::getNotificationDAO();
    $notID=$_REQUEST['notID'];
    $not=$notDAO->load($notID);
    $not->readed=1;
    $not->readDate=date('Y-m-d');
    $not->readTime=date('H:i:s');
    $notDAO->update($not);
    $result=  array("result"=>"OK");
    print (json_encode($result));
    ?>

    public function load($id){
        $sql = 'SELECT * FROM Notification WHERE id = ?';
        $sqlQuery = new SqlQuery($sql);
        $sqlQuery->setNumber($id);
        return $this->getRow($sqlQuery);
    }
```

5.2.1.2.3. setNotificationDeleted

Propósito: Marcar una notificación como eliminada.
Realiza un *update* (en la tabla *Notifications* de la base datos) del servicio que se pasa como parámetro de entrada (se pasa el id de la notificación), y se pone el atributo ***deleted=1***. Además, actualiza los campos de fecha y hora de eliminación, y finalmente devuelve un *JSON* con el resultado *OK*.

```
<?php
include_once '../include_dao.php';
$notDAO=DAOFactory::getNotificationDAO();
$notID=$_REQUEST['notID'];
$not=$notDAO->load($notID);
$not->deleted=1;
$not->deleteDate=date('Y-m-d');
$not->deleteTime=date('H:i:s');
$notDAO->update($not);
$result=  array("result"=>"OK");
print (json_encode($result));
?>
```

```
public function load($id){
    $sql = 'SELECT * FROM Notification WHERE id = ?';
    $sqlQuery = new SqlQuery($sql);
    $sqlQuery->setNumber($id);
    return $this->getRow($sqlQuery);
}
```

5.2.2. Implementación del controlador

Como se puede ver en la arquitectura del ***Reconfigurador*** (ver Imagen 23), y como se ha dicho en apartados anteriores, el *controlador* tiene dos funciones. La primera es la integración con *MoRE* [*1*], para poder recibir el *JSON* con la configuración y el segundo es el proceso de activación de los componentes necesarios, para una determinada notificación y un determinado contexto, según indique el *JSON* obtenido de la integración con *MoRE* [*1*]. A continuación se explicará, por separado, como se han implementado; cada uno de los procesos citados anteriormente.

5.2.2.1. Integración con MoRE

Lo primero que se ha de hacer es recibir una nueva notificación, para luego realizar la integración con *MoRE* [*1*], y así poder obtener el *JSON* con la configuración.

5.2.2.1.1. Recepción de notificaciones

Como ya se ha dicho en apartados anteriores, la recepción de mensajes se hace mediante la comunicación con el servidor del *C2DM*. Se ha tomado como ayuda para el desarrollo de esta parte de la Tesina, el tutorial sobre *C2DM* de Lars Vogel [8].

Para ello, lo primero que se ha hecho es declarar en el *manifest*, el *Intent* que permite la recepción de mensajes; por parte del *C2DM*, de la siguiente manera:

```
<Intent-filter>
    <action
     android:name="com.google.android.c2dm.Intent.RECEIVE" />
    <category android:name="pros.activities" />
</Intent-filter>
```

Seguidamente, en el mismo *Broadcast Receiver* (***C2DMReceiver***), creado para la recepción del identificador de registro, se crea un *Intent* que estará en escucha, para la recepción de mensajes, así:

```
if(Intent.getAction().equals
 ("com.google.android.c2dm.Intent.RECEIVE"))
    handleMessage(context, Intent);
```

Dicho *Intent* se ejecutará, cuando el *C2DM* envíe un nuevo mensaje a la *aplicación de notificaciones*. En el método *handleMessage*(*context*, *Intent*) se hará lo siguiente:

- Se lee el atributo ***payload*** del *Intent* activado, en este atributo estará el mensaje (la notificación) enviado por el *C2DM*.

```
final String payload = Intent.getStringExtra("payload");
```

- Se llama al método que obtiene la configuración, de cómo se ha se mostrar la notificación. Esto se explicará en el siguiente apartado (*Obtención de la configuración*).

5.2.2.1.2. Obtención de la configuración

La integración con *MoRE* [*1*], consiste en la implementación de un servicio *REST* (***getConfigurationForService.php***), que interactúa con este motor de reconfiguración, y que obtiene un fichero *JSON* con la configuración, que indicará qué componentes de interacción se han de activar, para una determinada notificación, en un determinado contexto. La detección del contexto queda fuera del alcance de esta Tesina. De esta tarea se encarga *MoRE*.

Una vez que se ha recibido el mensaje del *C2DM*, en el método *handleMessage*(*context*, *Intent*) del *BroadcastReceiver*, se hace la llamada al método ***obtenerConfiguración*** que hace lo siguiente:

1. Convierte el mensaje recibido en un objeto *Notification*.

```
notData = Utilities.convertStringToNotification(payload);
```

2. Realiza la llamada al fichero *getConfigurationForService.php*, del servidor de la aplicación.

```
String SERVICE_ID = notData.getServiceId();
String DEVICE_OPERATIVE_SYSTEM = "2";
```

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet();
HttpResponse response;
String url = URLbase +
"JSONApi/getConfigurationForService.php" +
    "?serviceID=" + SERVICE_ID + "&osID=" +
    DEVICE_OPERATIVE_SYSTEM;
httpget.setURI( new URI(url) );
response = httpclient.execute(httpget);
```

3. El resultado de dicha llamada será un *JSON*, que contendrá la información de cuáles son los componentes que se han de activar, para mostrar la notificación.

```
if (entity != null) {
    InputStream instream = entity.getContent();
    String notConfig =
        Utilities.convertStreamToString(instream);
```

4. Una vez recibida la respuesta, se convertirá a un objeto *ConfigNotification*, para poder trabajar con cada uno de sus atributos.

```
configData =
 Utilities.convertStringToConfigNotification(notConfig);
```

El servicio REST, realiza un *select* de la tabla *CurrentServicesStates* de la base de datos. En dicha tabla se encuentra la configuración que guarda *MoRE* [*1*] para cada una de las notificaciones y sus diferentes contextos. Finalmente devuelve en un *JSON*, la configuración obtenida. La implementación de servicio *REST* es la siguiente:

```
<?php
include_once '../include_dao.php';
$confs=DAOFactory::getCurrentServicesStatesDAO()
    ->queryByServiceIdAndOSID($_REQUEST['serviceID'],
    $_REQUEST['osID']);
$res=array();
if(count($confs)>0){
     $res['sound']=$confs[0]->sound;
     $res['speech']=$confs[0]->speech;
     $res['vibration']=$confs[0]->vibration;
     $res['light']=$confs[0]->light;
     $res['statusBar']=$confs[0]->statusBar;
     $res['toast']=$confs[0]->toast;
     $res['dialog']=$confs[0]->dialog;
     $res['desktopWidget']=$confs[0]->desktopWidget;
}
echo json_encode($res);
?>

public function queryByServiceIdAndOSID($serviceID, $osID){
    $sql = 'SELECT * FROM CurrentServicesStates WHERE
        service_id = ? AND device_os= ?';
    $sqlQuery = new SqlQuery($sql);
    $sqlQuery->set($serviceID);
    $sqlQuery->set($osID);
```

```
        return $this->getList($sqlQuery);
    }
```

5.2.2.2. Activación de componentes

Una vez que se ha recibido el *JSON* con la configuración, el último proceso realizado por el *controlador*, consiste en la activación de los componentes, indicados en dicho fichero de configuración.

Al llegar a este punto, se tienen dos objetos creados, un objeto *Notification* (***notData***), que contiene todos los datos de la notificación recibida del *C2DM*, y un objeto *ConfigNotification* (***configData***), que contiene los atributos que indican los componentes a activar.

Se ha creado un manejador (*HandlerActivity*), implementado como una *Activity* sin interfaz, para realizar las llamadas al *Service de activación de componentes* (en adelante ***el Service***), que como su propio nombre indica, permite la activación de cada uno de los componentes, indicados en el *JSON* de configuración. La llamada a dicho manejador es la siguiente:

```
Intent inten = new Intent(context, HandlerActivity.class);
inten.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK );
inten.putExtra("notData", notData);
inten.putExtra("configData", configData);
context.startActivity(inten);
```

Dicha *Activity* lo único que hará es invocar al *Service*, tantas veces, como componentes se tengan que activar.

```
Intent mIntent = getIntent();
notData = mIntent.getParcelableExtra("notData");
configData = mIntent.getParcelableExtra("configData");
```

Como se ha dicho antes, se ha implementado un *Service* (***MainService***), que activa cada uno de los componentes genéricos, mediante *Intents*, que se crearán (en la *Activity HandlerActivity*) dependiendo del valor de los atributos del objeto *ConfigNotification*. Por su parte, el objeto *Notification*, sirve para pasar los atributos necesarios al componente que lo requiera. A continuación se presenta un ejemplo, para la activación del componente *StatusBar*:

```
if(configData.getStatusBar().equals("2") ){
    Log.d(TESIS, "Se ha de activar el Statusbar con icono");
    Intent mIntent = new Intent(getApplicationContext(),
        MainService.class);
    mIntent.putExtra(COMPONENT_ID, STATUS_BAR);//Tipo de Alerta
    mIntent.putExtra("icon", notData.getIconID() );
    mIntent.putExtra("title", notData.getnTitle() );
    mIntent.putExtra("message", notData.getnText() );
    mIntent.putExtra("clase", "" );
    mIntent.putExtra("tickerText", "" );
    mIntent.putExtra("sound", "" );
    mIntent.putExtra("vibration", false );
    mIntent.putExtra("flashLight", false );
    startService(mIntent);
```

```
}
…Implementación de la activación de los demás componentes
```

configData es una instancia de la clase *ConfigNotification*. Como se puede ver, lo primero que se hace es obtener el valor del atributo, que identifica si se ha de activar el componente (*configData.getStatusBar()*). Si dicho atributo tiene el valor indicado para activar el componente (en este caso el valor de activación es 2), se crea un *Intent* que invoca el *Service* ***MainService***, así:

```
Intent mIntent = new Intent(getApplicationContext(),
   MainService.class);
```

Posteriormente, se pasa al *Service* (mediante el *Intent* creado), los datos necesarios para la activación del componente, accediendo a los datos de la notificación (***notData*** es una instancia de la clase *Notification*), por ejemplo, con:

```
mIntent.putExtra("icon", notData.getIconID() );
```

Se obtiene el identificador de la notificación, y se pasa al *Service* en el atributo ***icon***.

Se puede ver, que hay algunos valores que se informan "*vacíos*", esto se hace, porque los componentes genéricos, han sido diseñados, para que funcionen de la misma forma que se hace; si se crea una instancia directa de los mismos, por tanto, se debe poder pasar a los mismos, todos los datos necesarios, para que se puedan mostrar de todas las formas posibles, por ejemplo, un *Dialog* puede mostrarse con o sin título, puede tener activado uno, dos y/o hasta tres botones, y así cada uno de los componentes.
En este caso de ejemplo (*StatusBar*), debido que al activar el componente no se deben activar funcionalidades como el sonido, la vibración, etc. los atributos que representan dichas funcionalidades se pasan al ***controlador*** vacías ("" para atributos de tipo *String*, y *false* para atributos de tipo *boolean*) así:

```
mIntent.putExtra("sound", "" );
mIntent.putExtra("vibration", false );
```

Para finalizar, se ejecuta el *Service*:

```
startService(mIntent);
```

Este proceso se repite, para cada uno de los componentes indicados en el objeto de configuración.

5.2.2.2.1. Implementación de Service (MainService)

Como se ha dicho anteriormente, se ha implementado un *Service de activación de los componentes* llamado ***MainService.*** En él se ejecutan las siguientes acciones:

1. Una vez es activado mediante un *Intent*, recibe los datos enviados en el mismo. Un dato común entre todos los componentes, es su identificador (*id*). Con este dato, mediante una sentencia *switch*, se realiza la llamada al componente que se desea activar, de la siguiente forma:

```
mIntent = Intent;
componente = mIntent.getIntExtra(COMPONENT_ID,
    VIBRATION);
switch (componente) {
    case VIBRATION:
        CallComponents.callAlertas(getApplicationContext()
        ,VIBRATION, sonido);
    break;
    …Implementación demás componentes
```

Como se puede ver en el anterior ejemplo, una vez se ha identificado el componente a activar, se hace una llamada al método específico para activar el componente, dicho método se encuentra implementado en la clase CallComponents. En el ejemplo, el método es *callAlertas*.

2. Algunos componentes como los *Dialogs* y el *Toast*, entre otros, necesitan datos adicionales (obligatorios) para poder ser ejecutados (mensaje, título, etc.). Como se ha dicho antes, estos datos se pasan al *Service*, mediante el *Intent* que lo ha activado. Por tanto, para componentes como estos, antes de realizar la llamada al método que lo ejecuta, se han de obtener los datos requeridos, en variables locales, ya que estos se tendrán que pasar como parámetros de entrada, al método de activación. A continuación se muestra el código para la activación del *StatusBar*:

```
mIconValue = mIntent.getIntExtra("icon", 0);
mTittleValue = mIntent.getStringExtra("title");
mTextValue = mIntent.getStringExtra("message");
mNomClase = mIntent.getStringExtra("clase");
mTickerText = mIntent.getCharSequenceExtra("tickerText");
mSonido = mIntent.getStringExtra("sound");
mVibracion = mIntent.getBooleanExtra("vibration", false);
mFlashLight = mIntent.getBooleanExtra("flashLight",
false);

CallComponents.callSBNotification(getApplicationContext(),
    mIconValue, mTittleValue, mTextValue, mNomClase, 1,
    mNumNot, mTickerText, mSonido, mVibracion,
    mFlashLight);
mNumNot++;
```

Con esto queda implementado el *Service*. Los métodos creados para la activación de los componentes, se especifican en el punto: Clase CallComponentes.

5.3. Aplicación de notificaciones

Para poder mostrar la utilidad del ***Reconfigurador*** implementado en esta Tesina, se ha creado una aplicación de notificaciones Android (en adelante "***la aplicación***"). En este apartado se explica al detalle la funcionalidad de la misma, así como todos los pasos que se han ejecutado, para su correcta implementación.

Como se ha especificado en apartados anteriores, la aplicación se ha implementado para que funcione en los dispositivos con la versión Android 2.3 (*Gingerbread*). Se ha elegido esta versión del SO Android, debido a que actualmente es la más extendida entre los usuarios de esta plataforma, y además, permite la implementación de los recursos necesarios, para alcanzar los objetivos de esta Tesina.

A continuación se muestra un gráfico con la arquitectura de la aplicación, y luego se explica cómo se ha implementado cada parte.

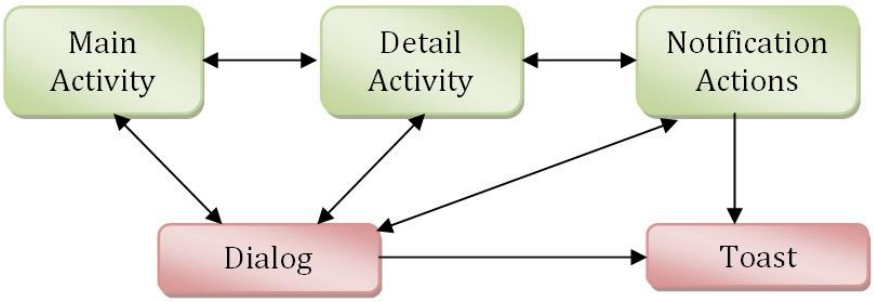

Imagen 25. Arquitectura de la aplicación

Como se muestra, la aplicación está compuesta por tres *Activities* (en verde), con las que el usuario podrá interactuar, para ver las notificaciones obtenidas. Además, proporciona algunos *Dialog* y *Toast* (en rojo), para informar al usuario, de las acciones que se desean realizar o se han realizado. A partir de ahora, se especificará la funcionalidad de cada *Activity*, así como la explicación de la forma en la que se han implementado.

5.3.1. Main Activity

Esta *Activity* es la que se muestra al ejecutar por primera vez la aplicación. Se ha implementado en la clase ***AllNotsActivity***. La interfaz gráfica de esta *Activity* está definida en el *Layout* ***main.xml***, y se encuentra dividida en 3 partes:

- Una barra superior
- Una barra inferior
- Un ListView que se ubicará entre las dos barras

Las dos barras, han sido implementadas en dos interfaces independientes (en dos *Layouts* distintos), y se han incluido en la vista de esta *Activity*, mediante el *tag include*, de la siguiente forma:

```
<include
   android:id="@+id/top_actionbar1"
   layout="@layout/top_actionbar1" />

<ListView
   android:id="@+id/listView"
   style="@style/allListStyle"
   android:layout_width="match_parent"
   android:layout_height="313dp"
   android:layout_weight="0.69" >
</ListView>
```

```
<include
   android:id="@+id/bottom_actionbar1"
    layout="@layout/bottom_actionbar1" />
```

A continuación se explicará la implementación de cada una de las tres interfaces; en las que se divide esta *Activity*.

5.3.1.1. Interfaz barra superior

Imagen 26. Main Activity - Barra superior

La barra superior de la *Main Activity*, está dividida en tres Tabs (***All***, ***Unreaded*** y ***Edit***). Cada uno de ellas servirá para cambiar la vista de la *Activity*, sin tener que crear una *Activity* nueva para cada vista. Actualmente, para las aplicaciones implementadas, para versiones superiores a la versión 3.0 de Android, ya existe un componente llamado *ActionBar*, que implementa esta barra, pero debido a que actualmente los móviles más usados aún usan la versión 2.3, se ha tenido que implementar esta barra de esta forma.

El Tab ***All*** servirá para indicar que se están mostrando todas las notificaciones, el Tab ***Unreaded*** indicará que se están visualizando solo las notificaciones que no han sido leídas, y para finalizar, el Tab ***Edit*** indicará que se van a editar las notificaciones. Este último Tab, mostrará la edición de las notificaciones que se estén visualizando antes de pulsar dicho Tab, es decir, si antes de pulsar el Tab ***Edit*** estaba pulsado el Tab ***All***, entonces se editarán todas las notificaciones, por tanto, en el *ListView* se mostrarán las mismas notificaciones que se muestran al pulsar el Tab ***All***. Si por el contrario, antes de pulsar el Tab ***Edit*** estaba pulsado el Tab ***Unreaded***, entonces se editarán las notificaciones que no han sido leídas, por tanto, en el *ListView* se mostrarán las mismas notificaciones que se muestran al pulsar el Tab ***Unreaded***.

El Layout de esta barra está implementado en el fichero ***top_actionbar1.xml***. Está implementado como un *RelativeLayout* con tres botones que se distribuyen horizontalmente. Se ha implementado de tal forma, que cada vez que se pulse uno de sus Tabs, este cambie de color (tanto el fondo como la letra), para resaltar la vista que se está visualizando en cada momento.

5.3.1.2. Interfaz barra inferior

Imagen 27. Main Activity - Barra inferior

La barra inferior está compuesta por dos botones, ubicados en cada uno de los extremos de la misma. Solo está provisto de funcionalidad el botón de la izquierda, y solo para cuando estén seleccionados los Tabs ***All*** y ***Unreaded*** (de la

barra superior), en cuyo caso, servirá para recargar las notificaciones, haciendo la llamada pertinente al servidor. Por su parte, cuando se seleccione el Tab ***Edit*** (de la barra superior), esta barra se ocultará.

La implementación de esta barra se encuentra en el fichero ***bottom_actionbar1.xml***, y al igual que la barra superior, está implementada con un *RelativeLayout*. Sin embargo, debido a que el contenido de los botones no es texto sino una imagen, estos se han implementado como *ImageButton*. Al pulsar los botones, el fondo de los mismos cambia de color (azul), para indicar al usuario que está pulsando el botón, pero en este caso, el color de fondo desaparece cuando el usuario deja de pulsar el botón.

5.3.1.3. Interfaz del ListView

Para el desarrollo de esta parte de la aplicación, se ha tomado como referencia el ejemplo expuesto en la Web Codehenge.com [9], así como el tutorial: "Android ListView and ListActivity" [15] del blog de Lars Vogel.

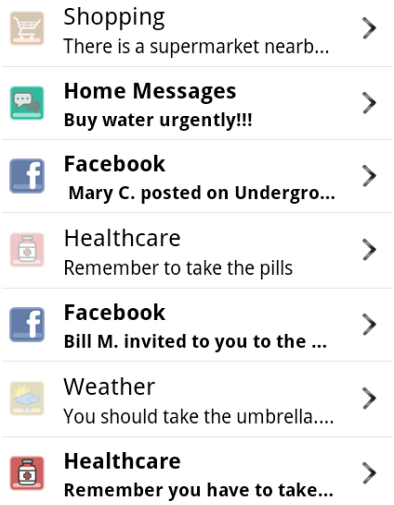

Imagen 28. Main Activity - ListView

Como se puede ver en el punto *4.3.1 (Main Activity)*, el *ListView* está implementado en el propio layout del *Main Activity*, sin embargo, aquí simplemente se define la vista en la que se mostrarán las notificaciones, pero debido a que, como se puede ver en la Imagen 28, cada ítem (elemento) de la lista está compuesto por varios componentes, se ha de crear un *Layout* para especificar la forma que tendrá cada uno de los ítems.

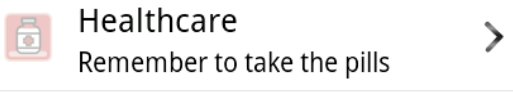

Imagen 29. Main Activity - ListView - Ítem no Edit

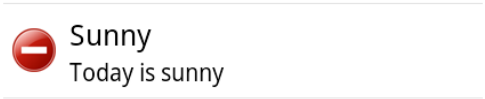

Imagen 30. Main Activity - ListView - Ítem Edit

Debido a que el funcionamiento de los Tabs *All* y *Unreaded* no es el mismo que el del Tab *Edit*, se han tenido que crear dos interfaces diferentes, una para cuando se muestren las notificaciones (pulsando el Tab *All* o *Unreaded*), y otra cuando se quieran editar las mismas (pulsando Tab *Edit*). La interfaz para los ítem de *no Edit* se ha implementado en el fichero ***allnots_item.xml***. Ésta ha sido implementada mediante un *RelativeLayout* que contiene dos *ImageView*, uno para el icono de la notificación y otro para representar la flecha que indica que se puede seleccionar el ítem. Para poner el nombre y la descripción de la notificación, se han creado dos *TextView*. La interfaz para los ítems del *Edit* está implementada en el fichero ***allnots_item_edit.xml.*** La implementación es la misma que para los otros ítems, con la diferencia que no está la flecha, ya que estos ítems no son seleccionables, y la imagen, que en el otro tipo de ítem representa el icono de la notificación, en este *Layout* se reemplaza por un botón (*Button*), que al ser pulsado, realizará la llamada al servidor, para marcar una notificación como eliminada.

5.3.1.4. Funcionamiento e implementación

Lo primero que se explicará en este apartado, es como se registra la aplicación en el *C2DM*, debido a que este es el primer procedimiento automático que realiza, luego se explicará cómo se han implementado todos los componentes que interactúan en esta Activity (Tabs y botones), y se detallará el funcionamiento de cada uno de ellos.

5.3.1.4.1. Registro de la aplicación de notificaciones en el C2DM

Este proceso se realizará cuando se inicie *la aplicación*, y se hace debido a que el C2DM asigna a cada aplicación, instalada en un determinado dispositivo, un identificador, para no tener problemas en el envío de una nueva notificación.
Lo primero que se ha hecho, es declarar en el *manifest* un *Intent* de registro al servidor del *C2DM*, así:

```
<Intent-filter>
    <action
        android:name=
            "com.google.android.c2dm.Intent.REGISTRATION"/>
    <category android:name="pros.activities" />
</Intent-filter>
```

Se ha creado un *Broadcast Receiver* (***C2DMReceiver***) en el que se tendrá un *Intent* de registro, que estará en escucha:

```
if (Intent.getAction().
    equals("com.google.android.c2dm.Intent.REGISTRATION"))
handleRegistration(context, Intent);
```

Al ejecutar *la aplicación* por primera vez, se realiza una solicitud de registro al servidor del *C2DM*, el cual responderá activando el *Intent* detallado anteriormente. En el método *handleRegistration(context, Intent)* se hace lo siguiente:

- Se recibe del *C2DM* (mediante el *Intent* activado) el atributo ***registration_id***, el cual contiene el identificador de registro, asignado por el servidor del *C2DM* a la aplicación.

```
String registration =
    Intent.getStringExtra("registration_id");
```

- Se crea un nuevo *Thread*, en el que se enviará dicho identificador al servidor de la aplicación, para guardarlo en un fichero. Esto se hace debido a que el identificador de registro, es uno de los datos obligatorios (por parte del *C2DM*), para poder enviar una nueva notificación a la aplicación.

```
SendRegIDThread myThread = new
    SendRegIDThread(registration);
myThread.start();
```

En el *Thread* se hace lo siguiente:

```
HttpClient client = new DefaultHttpClient();
HttpPost post = new HttpPost(URLbase +
       "sendRegisterID.php");

   List<NameValuePair> nameValuePairs = new
ArrayList<NameValuePair>(1);

nameValuePairs.add(new BasicNameValuePair
    ("registrationid", registrationID));
post.setEntity(new UrlEncodedFormEntity(nameValuePairs));

HttpResponse response = client.execute(post);
```

Por su parte el fichero ***sendRegisteID.php*** realiza las siguientes acciones:

```
<?php
    $fichero = 'registerID.txt';
    $clave = $_POST["registrationid"];
    file_put_contents($fichero, $clave);
    chmod($fichero,0777);
?>
```

5.3.1.4.2. Tab All

Cuando se inicia la aplicación se abre esta *Activity* con el Tab ***All*** seleccionado. Lo primero que se hace es una llamada al servidor, para obtener todas las notificaciones. La llamada se hace al fichero ***retrieveNotifications.php***, el cual devuelve a la aplicación un *JSON*, con todas las notificaciones de la tabla ***Notification*** de la base de datos, cuyos atributos ***deleted=0 y sent=1***.

Las notificaciones obtenidas se guardan en el atributo ***AllNots***, el cual está definido como un *ArrayList* de *RetrieveNotification*.

```
private  ArrayList<RetrieveNotification> AllNots;
AllNots = getDataFromServer(GET_ALL_NOTS);
```

En el método *getDataFromServer*() se han implementado dos *Toast*. Uno para informar al usuario que no hay notificaciones a mostrar, y otro para informar que ha habido un error en la conexión de la red:

```
private ArrayList<RetrieveNotification>
getDataFromServer(int tipo) {

    …Declaración de variables

    try {
        …Llamada al servidor
        HttpEntity entity = response.getEntity();
        if (entity != null) {
            …Obtención y conversión de datos
        }
        else{
            Toast.makeText(getApplicationContext(), " No hay
                notificaciones a mostrar ",
                Toast.LENGTH_LONG).show();
        }
    } catch (ClientProtocolException e) {
        …Procesamiento de errores
    } catch (IOException e) {
        Toast.makeText(getApplicationContext(),
        ConectionError,
            Toast.LENGTH_LONG).show();
    } catch (URISyntaxException e) {
        …Procesamiento de errores
    }
    return datosFromServer;
}
```

Seguidamente, a partir de la lista obtenida (***AllNots***), se calcula una nueva lista en la que se almacenarán únicamente las notificaciones que no han sido leídas ***UnreadedNots***. Dichas notificaciones serán aquellas cuyo parámetro ***readed=0***.

```
private  ArrayList<RetrieveNotification> UnreadedNots;
UnreadedNots.clear();
RetrieveNotification not;

Iterator<RetrieveNotification> ite = AllNots.iterator();
while (ite.hasNext()) {
    not = (RetrieveNotification) ite.next();
    if(not.getReaded().equals(NOT_UNREADED))
        UnreadedNots.add(not);
}
```

En este momento, debido a que está seleccionado el Tab *All*, se realiza la acción para mostrar en el *ListView* todas las notificaciones (lista *AllNots*). Tanto para mostrar todas las notificaciones como para mostrar solo las que no se han leído, se ha implementado el Adapter ***NotsAdapter***.

```
NotsAdapter adapter2 = new NotsAdapter(this,
R.layout.allnots_item, lista);
notsListView.setAdapter(adapter2);
```

Como se puede ver, cuando se crea el *Adapter*, se le pasa la lista que se desea mostrar (en este caso ***AllNots***), así como el *Layout* diseñado para representar la interfaz de cada ítem de la lista. Para mostrar la lista, ésta debe invocar al método *setAdapter*(), pasándole como argumento el *Adapter* creado.

El ***NotsAdapter*** ha sido creado como una extensión de la clase *ArrayAdapter<RetrieveNotification>:*

```
public class NotsAdapter extends
    ArrayAdapter<RetrieveNotification>
    implements Constantes {
```

En el método *getView*() de esta clase, se *infla* el *Layout* definido, de la siguiente forma:

```
LayoutInflater inflater = mContext.getLayoutInflater();
rowView  = inflater.inflate(R.layout.allnots_item, null);
```

Además, en este mismo método, se comprueba qué tipo de notificación se está mostrando (leída o no leída), esto se hace para poder dibujar las notificaciones que no han sido leídas; con un color más fuerte, y con una tonalidad más clara; las notificaciones que ya se han leído.

```
if(not.getReaded().equals(NOT_UNREADED)){
    holder.title.setTextAppearance(getContext(),
R.style.textStyle1_1);
    holder.text.setTextAppearance(getContext(),
R.style.textStyle2_2);
    holder.icon.setAlpha(200);
}
else if(not.getReaded().equals(NOT_READED)){
    holder.title.setTextAppearance(getContext(),
R.style.textStyle1);
    holder.text.setTextAppearance(getContext(),
R.style.textStyle2);
    holder.icon.setAlpha(70);
}
```

Con todo esto, la primera pantalla que se muestra al usuario es la siguiente:

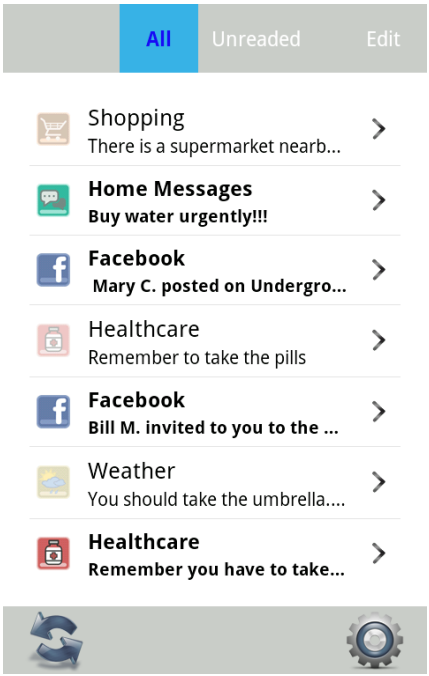

Imagen 31. Main Activity - All Nots

En esta ventana el usuario puede interactuar con la aplicación de la siguiente forma:

- Puede *pulsar el Tab **Unreaded***, con lo que en el *ListView* se mostrarán solo las notificaciones no leídas.
- Puede *pulsar el Tab **Edit***, en cuyo caso en el *ListView* seguirán apareciendo todas las notificaciones, pero no se podrán seleccionar. Además, en lugar de mostrar el icono de la notificación, se mostrará un botón, para poder marcar la notificación como eliminada.
- Puede *pulsar el botón de refresco* (botón izquierdo de la barra inferior), para volver a llamar al servidor, para obtener todas las notificaciones, por si ha habido algún cambio.
- Puede seleccionar cualquiera de las notificaciones, lo cual implicará la ejecución del *Detail Activity*.

5.3.1.4.3. Tab Unreaded

Al pulsar este Tab se llama a un método que permite cambiar la apariencia de la barra superior, para indicar que a partir de ese momento, en el *ListView*, se están visualizando solo las notificaciones que no se han leído. Además, se comprueba que la lista que contiene solo este tipo de notificaciones no esté vacía, y si lo está, se vuelve a llamar al servidor para obtener todas las notificaciones, y se vuelve a rellenar la lista ***UnreadedNots***. Para mostrar estas notificaciones, se realiza la llamada al mismo *Adapter* que para el caso anterior (todas las notificaciones), y se obtiene la siguiente pantalla:

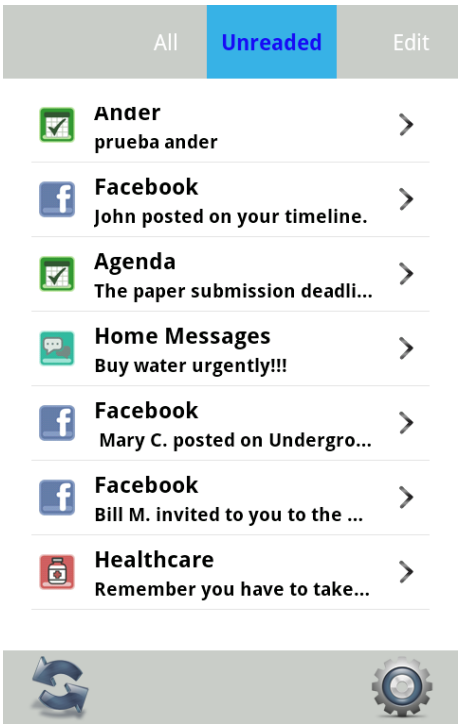

Imagen 32. Main Activity - Unreaded Nots

Como se aprecia en la Imagen 32, en el *ListView*, todas las notificaciones tiene una tonalidad oscura (no hay ninguna de color claro), ya que solo se están mostrando las notificaciones que no se han leído.

En esta vista, el usuario puede realizar las mismas interacciones con la aplicación, expuestas en el apartado anterior, con la excepción, de que si se pulsa el botón *Edit*, en el *ListView* no aparecerán todas las notificaciones, sino solo las que no se han leído. Y por su puesto, se puede volver a *pulsar el Tab All* para volver a mostrar todas las notificaciones.

5.3.1.4.4. Tab Edit

El Tab *Edit* es el que permitirá mostrar las notificaciones (tanto leídas como no leídas), en modo de edición. Como se ha dicho en los dos apartados anteriores, se mostrará en el *ListView* la lista *AllNots* o *UnreadedNots*, dependiendo de cuál sea la lista que se esté visualizando antes de pulsar este Tab.

Para mostrar las listas en modo de edición, se ha implementado el *Adapter* ***NotEditAdapter***, al cual se le pasará el *Layout* diseñado para este propósito, y la lista apropiada para cada caso, así:

```
case EDIT_TAB_SELECTED:
    NotEditAdapter adapter1 = new NotEditAdapter(this,
    R.layout.allnots_item_edit, lista);
    notsListView.setAdapter(adapter1);
    break;
```

Al igual que para los dos casos anteriores, lo primero que se hace al pulsar este Tab, es cambiar la apariencia de la barra superior, para indicar que este Tab ha sido seleccionado. A continuación se puede ver como se muestra la aplicación, cuando se selecciona este Tab, para los dos tipos de listas proporcionados:

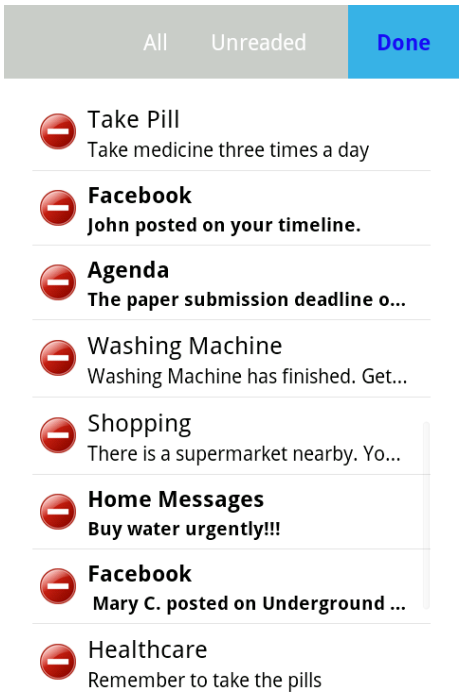

Imagen 33. Main View - Edit All nots

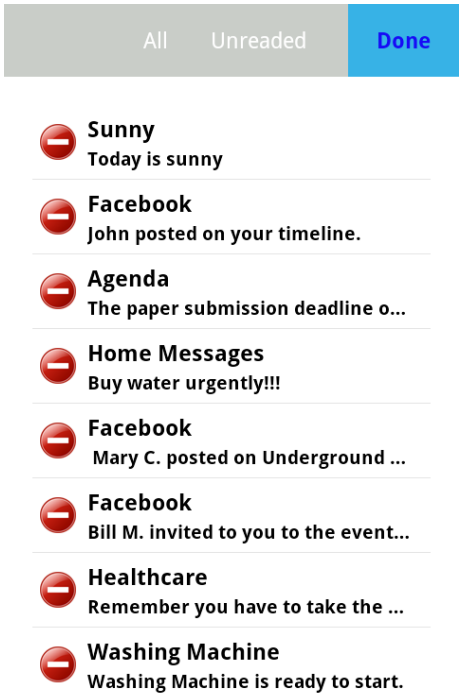

Imagen 34. Main view - Edit Unreaded nots

Cuando se selecciona este Tab, su nombre cambia a ***Done***, y si se vuelve a pulsar, su nombre vuelve a cambiar a ***Edit***, y retorna la lista que se estaba

mostrando antes de ser pulsado. También se puede ver que en esta vista, al igual que en las dos anterior (Tabs *All* y *Unreaded*), las notificaciones no leídas están dibujadas con un color más resaltado; que las notificaciones que ya han sido leídas. Además, se puede ver que al seleccionar este Tab se oculta la barra inferior.

En el método *getView* de este *Adapter* se ha implementado la funcionalidad del botón de eliminar, así:

```
delBtn.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        AlertDialog.Builder builder = new
            AlertDialog.Builder(mContext);
        builder.setMessage(dialogdel1 + not.getnTitle() +
            dialogdel2)
        .setCancelable(false)
        .setPositiveButton("Yes", new
            DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            String resul = not.deleteNot(mContext);
            if(resul.equals("OK") ){
                Toast.makeText(mContext, not.getnTitle() +
                    notDeleted, Toast.LENGTH_LONG).show();
                remove(not);
            }
            else
                Toast.makeText(mContext, not.getnTitle() +
                    notNoDeleted, Toast.LENGTH_LONG).show();
                dialog.cancel();
            }
        })
        .setNegativeButton("No", new
            DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog,int id){
                dialog.cancel();
            }
        });
        AlertDialog alert = builder.create();
        alert.show();
    }
});
```

Este código da como resultado, que cuando se pulse el botón eliminar de una notificación, aparezca en pantalla el siguiente *Dialog*:

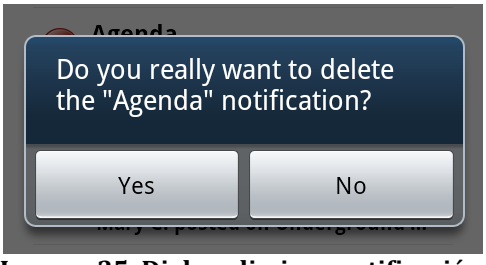

Imagen 35. Dialog eliminar notificación

Si se pulsa el botón **No**, se elimina el *Dialog*, y se sigue mostrando la misma lista. Si por el contrario se pulsa el botón **Yes**, se realiza la llamada al método *deleteNot*(), definido en la clase *RetrieveNotification*. Si la llamada al servidor para eliminar la notificación es OK, se muestra el *Toast* de la Imagen 36, al usuario, informando que se ha eliminado la notificación, y además, la notificación se elimina de la lista, por lo que cuando desaparezca el *Dialog*, ya no se visualizará la notificación en el *ListView*.

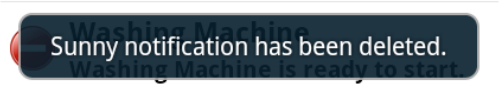

Imagen 36. Toast info notificación eliminada

5.3.1.4.5. Botón de refresco

La barra inferior solo se muestra cuando están seleccionados los Tabs *All* y *Unreaded*. Este botón es el que aparece en la parte izquierda de la mencionada barra. Cuando es pulsado, se llama al servidor, para obtener todas las notificaciones, y así poder refrescar los listados, que contienen tanto todas las notificaciones (*AllNots*) como las no leídas (*UnreadedNots*). Si ha habido algún cambio en las listas, se vuelve a ejecutar el *Adapter NotsAdapter*, con la lista adecuada en cada momento, para mostrar las notificaciones con los datos correctos.

5.3.2. Detail Activity

Para mostrar el detalle de una notificación se ha creado la *Activity* ***DetailActivity***. Esta *Activity* muestra el detalle de una notificación. La interfaz gráfica definida para la misma se ha implementado en el *Layout* ***detalle.xml***. Al igual que la *Main Activity*, ésta, está dividida en 3 partes:

- Una barra superior
- Una barra inferior
- Un RelativeLayout en medio de las dos barras, para mostrar el detalle de la notificación.

En este caso, las barras también han sido implementadas mediante dos *Layouts* diferentes, y han sido incluidas en el *Layout detalle*, mediante la etiqueta <*include*/>.

```
<include
   android:id="@+id/top_actionbar2"
    layout="@layout/top_actionbar2" />

…Definición del RelativeLayout para el detalle

<include
   android:id="@+id/bottom_actionbar2"
    layout="@layout/bottom_actionbar2" />
```

A continuación se explicará la implementación de cada una de las interfaces implementadas, paras las barras superior e inferior, así como la implementación del RelativeLayout incluido en Layout *detalle*, para mostrar el detalle de una notificación.

5.3.2.1. Interfaz barra superior

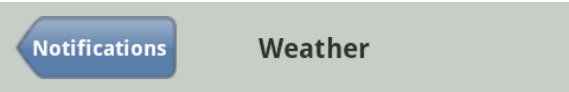

Imagen 37. Detail Activity - Barra superior

La interfaz de esta *Activity* está definida como un *RelativeLayout* horizontal definido en el fichero ***top_actionbar2.xml***.
Ésta contiene un botón que permite volver al *Main Activity*, pudiéndose dar los siguientes 3 casos:

- Si para mostrar el detalle, se ha seleccionado una notificación de la vista de todas las notificaciones (***All***), al pulsar el botón ***Notifications***, se volverá a dicha vista, con la peculiaridad que si la notificación no había sido leída (mostrada con un color más oscuro), al volver, se mostrará como leída (con un color más claro).
- Si para mostrar el detalle, se ha seleccionado una notificación de la vista de las notificaciones no leídas (***Unreaded***), al pulsar el botón ***Notifications***, se volverá a la mencionada vista, con la peculiaridad de que la notificación que se había seleccionado antes, ya no estará en la lista (no se mostrará), debido a que ya se ha leído, y como bien especifica su nombre, en esa vista se muestran solo las notificaciones que no han sido leídas.
- Como última opción, tenemos el caso de que se muestre el detalle sin haber abierto el *Main Activity*. Este caso ocurre cuando llega una nueva notificación, y bien sea mediante el *StatusBar* o mediante el *Dialog* mostrado, se decide mostrar el detalle. En este caso al pulsar el botón ***Notifications***, se mostrará la vista de todas las notificaciones.

El otro componente que tiene este *Layout* es un *TextView*, que sirve para mostrar el título de la notificación.

Como peculiaridad, decir que el botón ***Notifications*** cambia de tonalidad, cuando es pulsado, para mostrar al usuario que se ha realizado dicha acción.

5.3.2.2. Interfaz barra inferior

Imagen 38. Detail Activity - Barra inferior

El Layout de esta barra está definido como un *RelativeLayout*, en el fichero ***bottom_actionbar2.xml***, y como se puede apreciar, solo está compuesto por un botón, y debido a que es una imagen, se ha definido como un *ImageButton*.

Al pulsar dicho botón, se ejecutará una nueva *Activity* que simula un *Dialog* (***NotActionDialog***), para manipular las acciones que se pueden realizar sobre una notificación.

5.3.2.3. Interfaz del detalle

Imagen 39. Detail Activity - Interfaz del detalle

Esta Interfaz ha sido implementada en el propio *Layout* ***detalle.xml***. Como se ha dicho, se ha creado con un *RelativeLayout*, ya que este tipo de *Layout* permite distribuir los componentes de una forma heterogénea. Los componentes que la integran son:

- Un *ImageView* para mostrar la imagen de la notificación.
- Un *TextView* para mostrar el título de la notificación, al lado derecho de la imagen.
- Se han creado dos *LinearLayout*. Uno para mostrar el texto de la notificación; con su respectivo título, y otro para mostrar la fecha en la que se ha creado la notificación, con su respectivo título también.

5.3.2.4. Funcionamiento e implementación

En este apartado se explica cómo se ha implementado la funcionalidad de esta *Activity*, y cuál es su función.

El *Detail Activity* es la interfaz que permite al usuario; visualizar algunos de los datos de una notificación, más específicamente, el nombre, el texto que muestra la notificación como tal, y la fecha en la que se ha creado. Esta *Activity* puede haber sido invocada desde el *Main Activity*, desde el mensaje de un *StatusBar* o de un *Dialog*, que se hayan activado al recibir una nueva notificación.

Lo primero que se hace en esta *Activity* es recibir, mediante el *Intent* que ha iniciado, el objeto *RetrieveNotification*, que contiene los datos de la notificación, para la que se desea mostrar el detalle, así:

```
RetrieveNotification not;
Intent intent = getIntent();
not = intent.getParcelableExtra("mNot");
```

Acto seguido, si la notificación no había sido leída, se realiza una llamada al servidor, para marcarla como leída.

```
if(not.getReaded().equals(NOT_UNREADED)){
    ReadedThread mreadedThread = new ReadedThread(not);
    mreadedThread.start();
}
```

La llamada al servidor se ha hecho en un nuevo *Thread*, para no sobrecargar al sistema. Como se puede ver, al *Thread* se le pasa la notificación, y este obtiene el identificador de la misma, para realizar la llamada al servidor.

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet();
String url = URLbase +
    "JSONApi/setNotificationReaded.php?notID=" +
    not.getId();
httpget.setURI( new URI(url) );
httpclient.execute(httpget);
```

El siguiente paso es leer los atributos adecuados de la notificación, para asignarlos a cada uno de los componentes de la vista, así:

```
//Título de la barra superior
TextView titleTV = (TextView)
    findViewById(R.id.titleDetailID);
titleTV.setText(not.getnTitle() );

//Imagen de la notificación
ImageView mImagenView = (ImageView)
    findViewById(R.id.imageView1);
mImagenView.setImageResource(not.getImageID() );

//Título que acompaña a la imagen
TextView serviceName = (TextView)
    findViewById(R.id.serviceName);
serviceName.setText(not.getnTitle() );

//Texto de la notificación
TextView textTV = (TextView) findViewById(R.id.textDetail);
textTV.setText(not.getnText() );

//Fecha y hora
TextView tituloTV = (TextView)
    findViewById(R.id.TextView01);
tituloTV.setText(not.getcreationDate() );
```

Para terminar la implementación de esta *Activity*, se han creado unos métodos que se ejecutarán al pulsar el botón ***Notifications*** de la barra superior, y el botón ***NotAction*** de la barra inferior. Antes de especificar el funcionamiento y la implementación de estos botones, se muestra el aspecto general de esta *Activity*:

Imagen 40. Detail Activity

5.3.2.4.1. Botón Notifications

Al pulsar este botón lo único que se hace es finalizar esta *Activity*, para que se pueda mostrar al usuario de nuevo la *Main Activity*.

```
public void notificationsBtnClick(View view) {
    finish();
}
```

Este método ha sido asignado al botón en cuestión, en la implementación del *Layout* donde se ha definido.

```
android:onClick="notificationsBtnClick"
```

5.3.2.4.2. Botón NotAction

Este botón es el que se encuentra en la barra inferior de la *Activity*. Al igual que el botón ***Notifications***, a éste se le ha asignado un método dentro de la *Activity*, en la implementación de su *Layout*, así:

```
android:onClick="notActionBtnClick"
```

En este método se realiza la llamada a la *Activity* ***NotActionDialog***, y se le pasa la notificación.

```
public void notActionBtnClick(View view) {
    Intent intent = new Intent(getApplicationContext(),
        NotActionDialog.class);
    intent.putExtra("delNot", not);
    startActivityForResult(intent, NOT_DELETED);
}
```

La llamada a la nueva *Activity* se hace mediante el método *startActivityForResult*(), por tanto, esta *Activity* espera que la *Activity NotActionDialog* le devuelva un resultado. Esto se ha hecho para cerrar esta Activity, si al volver de dicha *Activity* la notificación se ha eliminado, esto se hace debido a que no tiene sentido mostrar el detalle de una notificación que ha sido eliminada. Esta *Activity* tiene implementado el método *onActivityResult*(), ya que éste es el que se ejecuta, cuando la *Activity* a la que se ha llamado le devuelve un resultado.

```
@Override
protected void onActivityResult(int requestCode, int
    resultCode, Intent data) {
    if(resultCode == RESULT_OK  & requestCode ==
        NOT_DELETED){
            finish();
    }
}
```

5.3.3. Notification Actions

Solo hay una acción que se puede realizar con una notificación, y es, marcarla como eliminada. Para este propósito se ha creado una *Activity* que simula un *Dialog* que implementa esta acción. Esta *Activity* será ejecutada cuando se pulsa el botón que se encuentra en la barra inferior de la *Activity*, que muestra el detalle de una notificación (*Detail Activity*). Se ha tomado como referencia, para el desarrollo de esta Activity, los siguientes enlaces:

- Entrada: "How to make a dialog slide from bottom to middle of screen in android" [11], del sitio Web stackoverflow.com.
- Entrada: Creating an Android Activity with no UI" [18], del blog geekyouup.blogspot.com.es.

La *Activity* en cuestión es ***NotActionDialog***. Su interfaz gráfica está definida en el *Layout* ***not_action_dialog***. Dicha interfaz está implementada mediante un *LinearLayout* ,que contiene un *TextView* para mostrar una descripción de la *Activity*, y dos *Button*, uno para implementar el botón que permite eliminar la notificación, y otro para salir de la *Activity* sin realizar ninguna acción.

El aspecto de esta *Activity* es el siguiente:

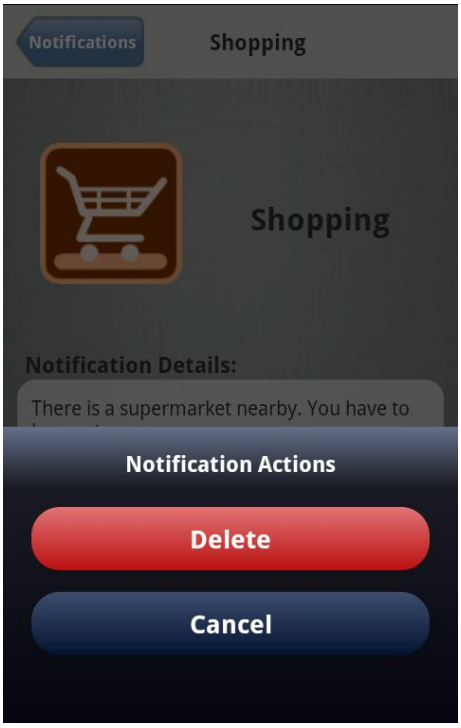

Imagen 41. Notification Action Activity

Como se puede apreciar, aunque se ha implementado como una *Activity*, su apariencia es como la de un *Dialog* personalizado, ya que se puede ver la *Activity* que ha realizado la llamada de fondo.

5.3.3.1. Funcionamiento e implementación

Esta *Activity* proporciona la funcionalidad de eliminar una notificación. Lo primero que se hace cuando se inicializa es leer la notificación del *Intent* que la ha activado.

```
RetrieveNotification not;
Intent intent = getIntent();
not = intent.getParcelableExtra("delNot");
```

Se ha asignado un método dentro de la *Activity* a cada uno de los botones.

- Método asignado al botón ***Delete***:
  `android:onClick="onClickeliminarBtn"`

- Método asignado al botón ***Cancel***:
  `android:onClick="onClickcancelarBtn"`

Al pulsar el botón ***Cancel*** lo único que se hace es eliminar la *Activity*.

```
public void onClickcancelarBtn(View view){
    finish();
}
```

Por su parte, al pulsar el botón ***Delete*** Se crea un *Dialog*, para preguntar al usuario si realmente desea eliminar la notificación, este *Dialog* es el mismo que se muestra en la Imagen 35. Con la diferencia de la implementación de los botones.

```
public void onClickeliminarBtn(View view){
    AlertDialog.Builder builder = new
        AlertDialog.Builder(mContext);
    builder.setMessage(dialogdel1 + not.getnTitle() +
        dialogdel2)
    .setCancelable(false)
    .setPositiveButton("Yes", new
        DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            String resul = not.deleteNot(mContext);
            dialog.cancel();
            resultado(resul);
        }
    })
    .setNegativeButton("No", new
        DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            dialog.cancel();
        }
    });
    AlertDialog alert = builder.create();
    alert.show();
}
```

Como se puede ver, al hacer clic en el botón **Yes**, se ejecuta el método ***deleteNot***() de la notificación, y el resultado del mismo, es lo que se devuelve al *Detail Activity*.

```
private void resultado(String resul) {
    if(resul.equals("OK") )
        Toast.makeText(getApplicationContext(),
            not.getnTitle() + notDeleted,
            Toast.LENGTH_LONG).show();
    else
        Toast.makeText(getApplicationContext(),
            not.getnTitle() + notNoDeleted,
            Toast.LENGTH_LONG).show();
    Intent res = new Intent();
    res.putExtra("respuesta", resul);
    setResult(RESULT_OK, res);
    finish();
}
```

Por tanto, si la notificación es eliminada con éxito, se cierra esta *Activity* y se devuelve una respuesta afirmativa: *setResult*(*RESULT_OK*, *res*). Además, se muestra un *Toast* que indica si la notificación ha sido eliminada o no.

5.4. Aspectos importantes de implementación

En este apartado se indicarán todos aquellos aspectos más relevantes que se han tenido en cuenta para la implementación del sistema que engloba el ***Reconfigurador*** diseñado para esta Tesina.

5.4.1. Interfaz CONSTANTES

Se ha creado una Interfaz denominada ***CONSTANTES***, en la que se han declarado todos los atributos utilizados en la implementación del sistema.
Esta interfaz contiene atributos como:

- Nombres de los servicios (notificaciones).
- *Strings*, que contienen los mensajes a mostrar en los *Dialogs*.
- Identificadores de los componentes genéricos.
- Identificadores de los Tabs de la aplicación final.
- Identificadores de los iconos de las notificaciones, etc.

Todas los componentes Android utilizados en la implementación del sistema implementan esta interfaz. A continuación se puede ver un ejemplo:

```
public class C2DMReceiver  extends BroadcastReceiver  implements
Constantes {
```

5.4.2. Deserialización JSON

Todos los mensajes recibidos, tanto por parte del servidor de la aplicación como por parte del servidor del *C2DM*, están en formato *JSON*. Como referencia se ha tomado la entrada: "Deserializing from JSON into PHP, with casting" [16] de la Web stackoverflow.com. Para poder trabajar con los datos de cada mensaje de una forma individualiza en código *Java*, se han creado las siguientes clases:

5.4.2.1. Clase RetrieveNotification

Esta clase servirá para crear instancias de los mensajes obtenidos de las peticiones realizadas al servidor de la aplicación, como por ejemplo, para obtener todas las notificaciones.
Esta clase implementa la interfaz *Parcelable*, ya que en el sistema es necesario pasar objetos instancia de esta clase, entre *Activities*.

```
public class RetrieveNotification implements Constantes,
Parcelable{
```

Los atributos que contiene esta clase son:

```
private String id;            - ID de la notificación en la
                                tabla Notification de la base
                                de datos.
private String serviceID;     - Identificador del servicio
private String service;       - Nombre del servicio
private String serviceIcon;   - Icono del servicio
private String nTitle;        - Título
private String nText;         - Mensaje del servicio
private String readed;        - Indicador de lectura (0/1)
private String creationDate;  - Fecha de creación de la
                                notificación
```

En esta clase también se ha implementado el método que permite marcar una notificación como eliminada. Este método realiza la llamada al fichero ***setNotificationDeleted.php*** del servidor de la aplicación, y se le pasará como parámetro, el identificador de la notificación que se desea marcar como eliminada.

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet();
HttpResponse response;
String url = URLbase +
    "JSONApi/setNotificationDeleted.php?notID=" + id;
httpget.setURI( new URI(url) );
response = httpclient.execute(httpget);
```

5.4.2.2. Clase Notification

Esta clase ha sido diseñada y creada para convertir a un objeto *Java*; las notificaciones enviadas desde el servidor del *C2DM*. Esta clase implementa la clase *Parcelable*, ya que se necesitará pasar objetos de esta clase entre algunos componentes, diseñados en el desarrollo de esta Tesina.

Los atributos de esta clase, por tanto, serán los mismos que tiene el objeto creado para enviar una notificación desde el servicio web, dichos datos son:

```
private String id;
private String serviceId;
private String obtrusivenessSpaceId;
private String creationDate;
private String creationTime;
private String deliveryDate;
private String deliveryTime;
private String readDate;
private String readTime;
private String deleteDate;
private String deleteTime;
private String nTitle;
private String nText;
private String icon;
private String sent;
private String readed;
private String deleted;
private String considerateSystemID;
```

Esta clase también proporciona los métodos para obtener cada uno de los atributos declarados anteriormente (*getters*).

5.4.2.3. Clase ConfigNotification

Esta clase se ha creado para convertir a un objeto *Java* el *JSON* recibido del servidor de la aplicación, con la configuración de cómo se deberá mostrar una notificación. Los atributos de esta clase son los siguientes:

```
private String id;
private String sound;
private String speech;
```

```
private String vibration;
private String light;
private String statusBar;
private String toast;
private String dialog;
private String desktopWidget;
```

Cada uno ellos tendrá un número que indicará si se debe o no activar el componente al que representa. Un "2", y para algunos también un "3", indicará que el componente ha de ser activado.

Esta clase, al igual que las anteriores, proporciona los métodos para obtener cada uno de sus atributos (*getters*), y también ha sido implementada la Interfaz *Parcelable*, para poder transportar objetos de esta clase, entre los componentes desarrollados en esta Tesina.

5.4.3. Métodos globales

Para tener una mayor independencia en la utilización de algunos métodos, se han creado dos clases, con la implementación de aquellos métodos que pueden ser llamados desde cualquier parte del sistema (además, se pueden llamar tantas veces como se necesiten). A continuación se detallan los métodos implementados en cada una de ellas.

5.4.3.1. Clase Utilities

Se ha creado esta clase para implementar los métodos necesarios para el procesamiento de la información obtenida de los servidores. Como ayuda, para la implementación de estos métodos, se han tomado las siguientes referencias:

- Tutorial: "How-to: Android as a RESTful Client" [10]
- Entrada: "Make an HTTP request with Android" [11], de la Web stackoverflow.com.
- Entrada: "Reading a json login response with Android SDK" [12], de la Web instropy.com.
- Entrada: GSON throwing "Expected BEGIN_OBJECT but was BEGIN_ARRAY" [13], de la Web stackoverflow.com.

Los métodos que la componen son:

5.4.3.1.1. Método convertStreamToString

Este método sirve para convertir los mensajes recibidos del servidor en un *String* para su posterior utilización. Recibe como parámetro el objeto *Stream* que devuelve el servidor.

```
public static String convertStreamToString(InputStream is) {
   BufferedReader reader = new BufferedReader(new
       InputStreamReader(is));
    StringBuilder sb = new StringBuilder();
```

```
        String line = null;
        try {
            while ((line = reader.readLine()) != null) {
                sb.append(line + "\n");
            }
        } catch (IOException e) {
            …Procesamiento de errores
        } finally {
            try {
                is.close();
            } catch (IOException e) {
                …Procesamiento de errores
            }
        }
        return sb.toString();
    }
```

5.4.3.1.2. Método convertStringToArrayRetrNot

Este método convierte un *String* (el que contiene un conjunto de notificaciones) en un *Array* de objetos *RetrieveNotification*. Para ello, se utiliza la clase *Gson*, que permite convertir un *JSON* en el objeto indicado.

```
    public static ArrayList<RetrieveNotification>
        convertStringToArrayRetrNot(String men) {
        ArrayList<RetrieveNotification> notsList = new
            ArrayList<RetrieveNotification>();
        try {
            Gson gson = new Gson();
            JsonParser parser = new JsonParser();
            JsonArray Jarray = parser.parse(men).getAsJsonArray();
                for(JsonElement obj : Jarray ){
                RetrieveNotification not = gson.fromJson(obj ,
                    RetrieveNotification.class);
                    notsList.add(not);
                }
         } catch (IllegalStateException e) {
                …Procesamiento de errores
         }catch (JsonParseException  e) {
                …Procesamiento de errores
         }
         int numNotifications = notsList.size();
         return notsList;
     }
```

5.4.3.1.3. Método convertStringToConfigNotification

Este método convierte un *String* en un objeto *ConfigNotification*. Es usado para convertir al objeto indicado, el mensaje que se recibe del servidor (con la configuración con la que se tiene que mostrar la notificación). Esto se hace para poder tener los atributos individuales de la configuración, los cuales servirán para indicar a la aplicación, que componentes se han de activar, a la hora de mostrar la notificación al usuario.

```
    public static ConfigNotification
        convertStringToConfigNotification(String men) {
```

```
        ConfigNotification config = null;
    try {
        Gson gson = new Gson();
        JsonParser parser = new JsonParser();
        JsonElement obj = parser.parse(men);
        config = gson.fromJson(obj ,
            ConfigNotification.class);
    } catch (IllegalStateException e) {
        …Procesamiento de errores
    }catch (JsonParseException  e) {
        …Procesamiento de errores
    }
    return config;
}
```

5.4.3.1.4. Método convertStringToNotification

Este método convierte el *String* que contiene los datos de la notificación recibida del *C2DM* a la clase *Notification*. Este objeto obtenido tendrá todos los atributos que tiene una notificación en la base de datos (tabla *Notification*).

```
public static Notification convertStringToNotification
    (String men) {
    Notification not = null;
    try {
        Gson gson = new Gson();
        JsonParser parser = new JsonParser();
        JsonElement obj = parser.parse(men);
        not = gson.fromJson(obj , Notification.class);
    } catch (IllegalStateException e) {
        …Procesamiento de errores
    }catch (JsonParseException  e) {
        …Procesamiento de errores
    }
    return not;
}
```

5.4.3.1.5. Método convertStringToRespuesta

Este método se utiliza para convertir un *String* a un objeto *Respuesta*. Dicho *String* contiene los datos de las respuestas de las llamadas al servidor de la aplicación de los métodos que solo devuelven un *OK*, en caso de que la operación se haya realizado correctamente, como por ejemplo, cuando se llama al método que marca una notificación como leída.

```
public static Respuesta convertStringToRespuesta
    (String men){
    Respuesta resp = new Respuesta();
    try {
        Gson gson = new Gson();
        JsonParser parser = new JsonParser();
        JsonElement obj = parser.parse(men);
        resp = gson.fromJson(obj , Respuesta.class);
    } catch (IllegalStateException e) {
        …Procesamiento de errores
    }catch (JsonParseException  e) {
```

```
            …Procesamiento de errores
        }
        return resp;
    }
```

5.4.3.1.6. Método getNotification

Método que obtiene una notificación (objeto *RetrieveNotification*) realizando la llamada al fichero *queryNotification.php* del servidor de la aplicación, y pasándole como parámetro el identificador de la notificación.

```
    public static RetrieveNotification getNotification
        (String id){
        RetrieveNotification not = new RetrieveNotification();
        HttpClient httpclient = new DefaultHttpClient();
        HttpGet httpget = new HttpGet();
        HttpResponse response;
        try {
            String url = URLbase +
                "JSONApi/queryNotification.php?notID=" + id;
            httpget.setURI( new URI(url) );
            response = httpclient.execute(httpget);
            HttpEntity entity = response.getEntity();
            if (entity != null) {
                InputStream instream = entity.getContent();
                String datosRecibidos =
                    Utilities.convertStreamToString(instream);
                not =
                Utilities.convertStringToRetrNot(datosRecibidos);
            }
        } catch
            …Tratamiento de errores
        }
        return not;
    }
```

5.4.3.1.7. Método convertStringToRetrNot

Método que convierte un *String* en un objeto *RetrieveNotification*. Este método ha sido creado para las llamadas que se realizan al servidor de la aplicación, y cuyo resultado es la obtención de una única notificación, por ejemplo, el método citado anteriormente (*getNotification*).

```
    public static RetrieveNotification convertStringToRetrNot
        (String men) {
        RetrieveNotification not = new RetrieveNotification();
        try {
            Gson gson = new Gson();
            JsonParser parser = new JsonParser();
            JsonElement obj = parser.parse(men);
            not = gson.fromJson(obj , RetrieveNotification.class);
        } catch
            … Tratamiento de errors
        }
        return not;
    }
```

5.4.3.2. Clase CallComponents

Esta clase contiene todos los métodos que crean los *Intents*, y realizan la llamada a cada uno de los componentes genéricos creados. Todos los métodos siguen la siguiente estructura:

- Creación del *Intent* de activación del componente genérico.
- Paso de parámetros mediante el *Intent* creado.
- Llamada al *Service* o *Activity* que lo ejecuta.

Los métodos que la componen son los siguientes:

5.4.3.2.1. Método callToastNotification

Realiza la llamada al *Service* ***ToastNotification*** que ejecuta el componente ***Toast***. Como dato adicional solo se le ha de pasar el mensaje:

```
mIntent.putExtra("message", mensaje);   //Texto a mostrar
```

5.4.3.2.2. Método callSBNotification

Realiza la llamada al *Service* ***StatusBarNotification*** que ejecuta el componente ***StatusBar***. Los datos que se le han de pasar son:

```
mIntent.putExtra("icon", icono);
mIntent.putExtra("title", titulo);
mIntent.putExtra("message", mensaje);
mIntent.putExtra("clase", objetivo);
mIntent.putExtra("id", idNot);
mIntent.putExtra("cont", numNot);
mIntent.putExtra("tickerText", tickerText);
mIntent.putExtra("sound", sonido);
mIntent.putExtra("vibration", vib);
mIntent.putExtra("flashLight", lights);
```

5.4.3.2.3. Método callAlertDialog

Realiza la llamada a la *Activity* ***DialogComponent*** que ejecuta el componente ***Dialog***. Los datos necesarios que se le han de pasar son:

```
in.putExtra("notID", notID);
in.putExtra("tipoDialog", tipo);
in.putExtra("title", titulo);
in.putExtra("message", mensaje);
in.putExtra("positiveBtn", positiveButton);
in.putExtra("negativeBtn", negativeButton);
in.putExtra("neutralBtn", neutralButton);
in.putExtra("list", lista);
```

5.4.3.2.4. Método callAlertas

Realiza la llamada al *Service* ***Alertas*** que ejecuta los componentes ***Vibration***, ***Lights*** o ***Sound***, dependiendo del tipo que se pase mediante el *Intent*. De estos tres componentes, el único que puede necesitar un dato auxiliar es el *Sound*, aunque para la implementación del ***Reconfigurador*** desarrollado en esta Tesina, este dato se pasará "*vacío*".

```
mIntent.putExtra("type", tipo);
mIntent.putExtra("sound", sonido);
```

5.4.4. Llamadas al servidor de la aplicación

Todas las llamadas realizadas al servidor se han hecho mediante solicitudes *HTTP* de la siguiente forma:

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet();
HttpResponse response;
String url = "URL_indicada"
httpget.setURI( new URI(url) );
response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
InputStream instream = entity.getContent();
```

Después de esto, siempre se llama al método *convertStreamToString* (implementado en la clase *Utilities*), para convertir el mensaje recibido en un *String*, y así poder convertirlo posteriormente en el objeto indicado, para su posterior utilización dentro del sistema.

5.4.5. Pasar objetos entre componentes

Como se ha explicado en el capítulo de Tecnologías involucradas, los componentes que hacen parte de esta Arquitectura se activan mediante *Intents*, además, si se han de pasar parámetros entre dichos componentes, también se hace mediante los mismos *Intents* de activación. En Android, el paso de parámetros mediante *Intents* está implementado, pero solo para los casos básicos (*int*, *string*, *boolean*, etc.), sin embargo, si se desea pasar entre componentes un objeto de una clase diseñada en la aplicación desarrollada, se puede hacer que dicha clase implemente la interfaz *Parcelable*, y así de ese modo, poder pasar dicho objeto mediante el *Intent* creado de un componente a otro. Como referencia se ha tomado la entrada: "Passing custom objects between android activities" [17], del blog sohailaziz.com. A continuación, se puede ver como se pasa un objeto de la clase *RetrieveNotification* desde un componente, y como se recibe en otro:

Envío:

```
RetrieveNotification item = (RetrieveNotification)
    notsListView.getItemAtPosition(position);
Intent IntentDetail = new Intent(getApplicationContext(),
    DetailActivity.class);
IntentDetail.putExtra("mNot", item);
```

```
startActivity(IntentDetail);
```

Recepción:

```
Intent Intent = getIntent();
RetrieveNotification not = Intent.getParcelableExtra("mNot");
```

5.4.5.1.1. Hacer clase Parcelable

A continuación se explica a través de la clase *RetrieveNotification* cómo se hace una clase *Parcelable*.

1. Lo primero que se ha de hacer es crear un *constructor*, que tenga como parámetro un atributo *Parcel*.

```
public RetrieveNotification(Parcel source) {
    String[] data = new String[8];
    source.readStringArray(data);
    id = data[0];
    serviceID = data[1];
    service = data[2];
    serviceIcon = data[3];
    nTitle = data[4];
    nText = data[5];
    readed = data[6];
    creationDate = data[7];
}
```

2. Se han de sobrescribir los dos métodos siguientes:

```
@Override
public int describeContents() {
return 0;
}

@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeStringArray(new String[] { id, serviceID,
        service, serviceIcon, nTitle, nText, readed,
        creationDate } );
}
```

3. Para finalizar, hay que crear un atributo que debe tener el nombre CREATOR (siempre en mayúsculas), de la siguiente forma:

```
public static final
Parcelable.Creator<RetrieveNotification> CREATOR =
    new Parcelable.Creator<RetrieveNotification>() {

    @Override
    public RetrieveNotification createFromParcel
        (Parcel source) {
            return new RetrieveNotification(source);
    }

    @Override
```

```
        public RetrieveNotification[] newArray(int size) {
            return new RetrieveNotification[size];
        }
    };
```

Este mismo proceso se ha realizado en todos las clases creadas, para la representación de las notificaciones y de la configuración, con el objetivo de poder pasar instancias de los mismos, entre los diferentes componentes del sistema desarrollado.

6. UN CASO PRÁCTICO

En este capítulo se presenta un caso de estudio, para ver la funcionalidad del sistema de reconfiguración de notificaciones, diseñado en esta Tesina. En éste, también se muestran todas las posibles formas de interacción que puede tener el usuario, con la aplicación desarrollada como centro gestor de notificaciones.

Como caso de estudio, se presenta un usuario que dispone de un teléfono móvil Android. El usuario es un empresario que trabaja como director de su propia empresa. La notificación que recibirá estará relacionada con la compra de un regalo por el aniversario con su mujer. Para mostrar el funcionamiento del sistema desarrollado se explicará, cómo el usuario recibirá dicha notificación, para cada uno de los siguientes tres escenarios:

1. El usuario se encuentra conduciendo hacia su trabajo, acompañado por su mujer y sus hijos, y pasa cerca de una joyería.
2. El usuario se encuentra conduciendo **solo**, de vuelta a casa, y pasa cerca de una joyería.
3. El usuario se encuentra reunido en su trabajo con unos clientes muy importantes.

A continuación se explicará en un tres sub-apartados, la forma de recepción de la notificación, así como la posible interacción del usuario con la aplicación.

6.1. Escenario 1

El usuario se encuentra conduciendo hacia su trabajo, acompañado por su mujer y sus hijos, y pasa cerca de una joyería.

En este caso el sistema detecta, gracias a su integración con *MoRE* [*1*], que el usuario viaja acompañado, por tanto, debido a la privacidad del mensaje a notificar, el sistema de reconfiguración envía un *JSON* de configuración, para que se activen solo los siguientes componentes:

- StatusBar
- Dialog (sin botones)

Como se puede apreciar, todos los componentes son silenciosos, para no llamar la atención de los acompañantes, debido a la privacidad de la notificación.

En este caso, cuando el usuario encienda la pantalla de su dispositivo móvil se encontrará con la siguiente pantalla:

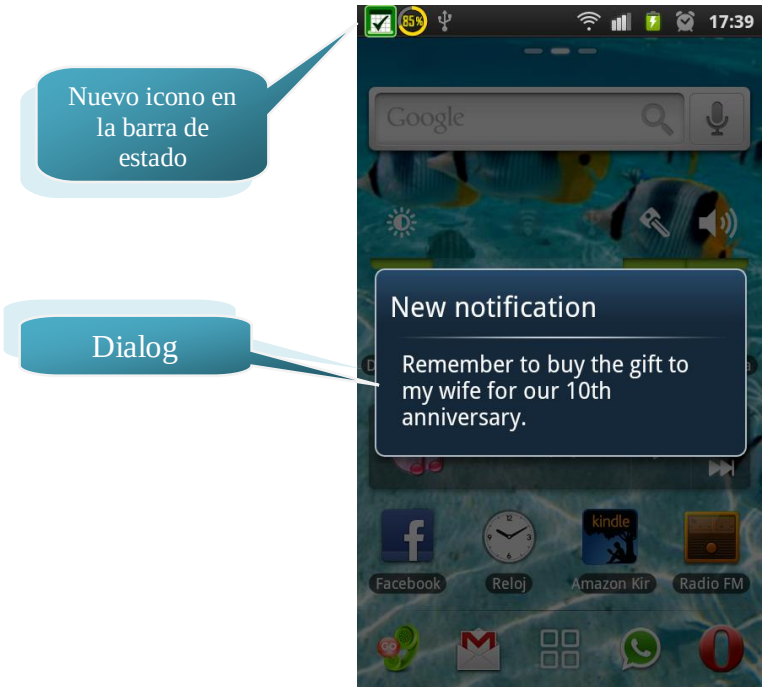

Imagen 42. CE - Escenario 1 - Notificación recibida

Como se puede ver, en el frontend del dispositivo aparece un Dialog informativo, y en la barra de estado aparece un icono que simboliza la agenda. Dichos componentes son los que el reconfigurador ha decidido activar, por el contexto en el que se encuentra el usuario, en el momento de la recepción.

En este caso, el usuario puede eliminar el Dialog del frontend pulsando el botón de retroceso. Además, si quiere ver el detalle de la notificación, puede abrir la ventana de notificaciones del dispositivo; deslizando la barra de estado hacia abajo, entonces, visualizará la notificación de la siguiente manera:

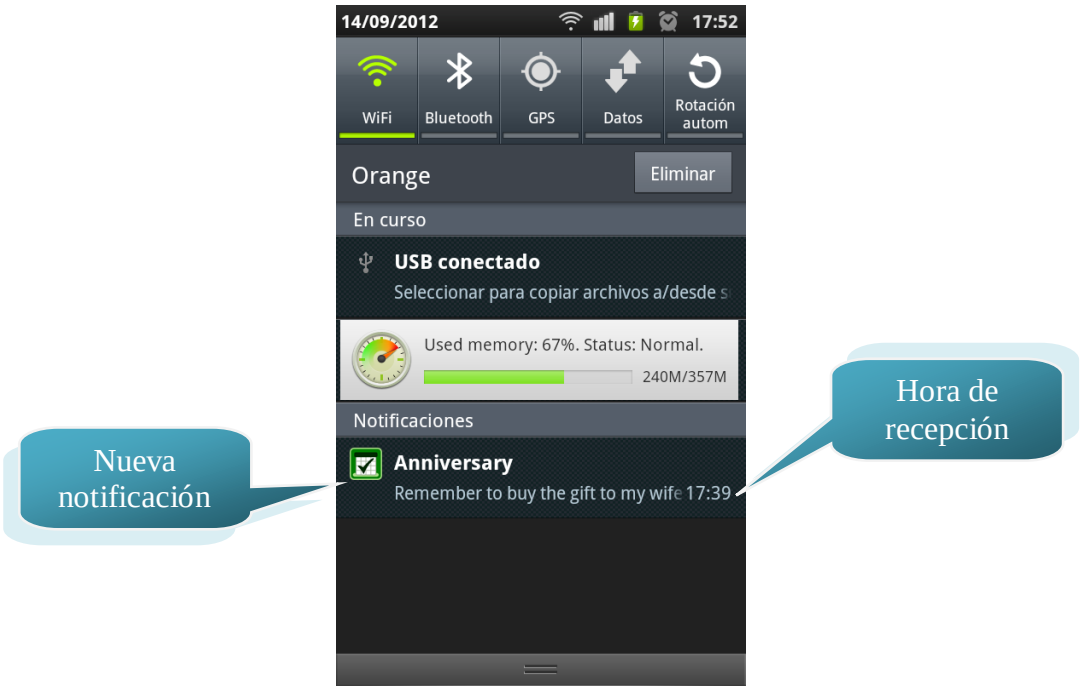

Imagen 43. CE - Escenario 1 - Ventana de notificaciones Android

Como se puede ver, aparecerá una notificación más en el centro de notificaciones de Android. En la vista de la notificación, se puede ver la hora de recepción del mensaje. Se puede apreciar que si el mensaje es muy largo, no se puede leer al completo, por tanto, si el usuario desea ver la notificación con más detalle, solo tiene que pulsar la notificación, y aparecerá la ventana de la aplicación diseñada, en la que se muestra el detalle de la notificación. La pantalla en cuestión es la siguiente:

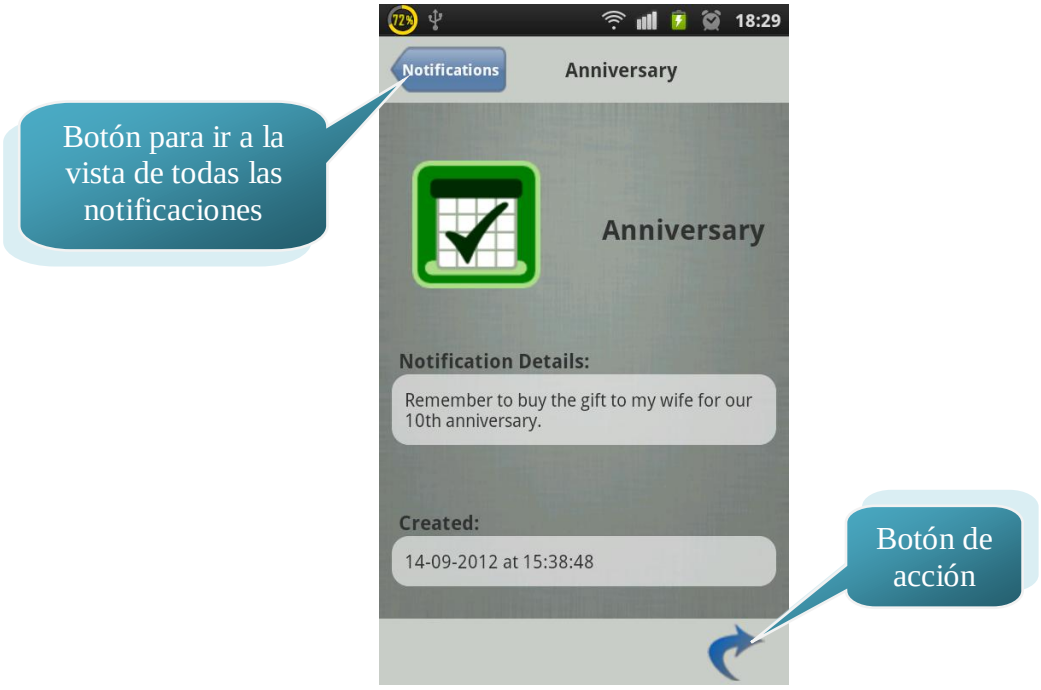

Imagen 44. CE - Escenario 1 - Detalle de la notificación

En este momento, el usuario se encuentra dentro de la aplicación diseñada para el control de las notificaciones, por tanto, puede realizar todas las acciones que contempla

la misma. A continuación se explicarán todas las opciones de las que dispone el usuario para interactuar con la aplicación.

6.1.1. Eliminar notificación

En la aplicación, se dispone de dos formas de eliminar una notificación. Partiendo de que el usuario se encuentra en la ventana que muestra el detalle de la notificación recibida, a continuación, se explica en dos apartados distintos, cada una de las dos formas que tiene el usuario de eliminar la notificación recibida.

6.1.1.1. Primera opción de eliminación de una notificación

La primera opción para eliminar la notificación, es pulsar el "botón de acción", de la barra inferior de la pantalla de detalle (ver Imagen 44). Este llevará al usuario a la siguiente pantalla:

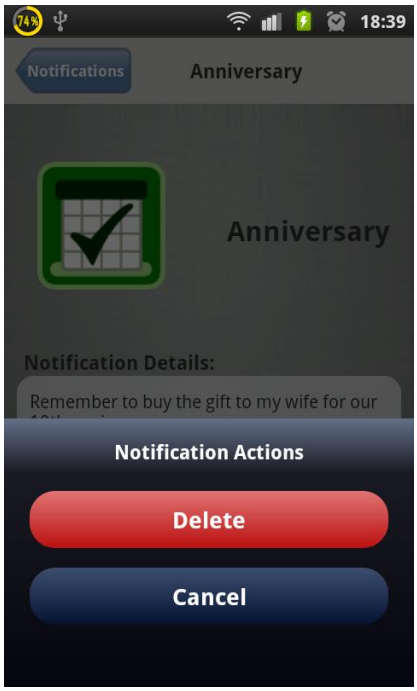

Imagen 45. CE - Escenario 1 - Acción de la notificación

Un aspecto muy importante, es que al visualizar esta pantalla, se puede ver de fondo, el detalle de la notificación, por tanto, el usuario puede saber qué notificación es la que está a punto de eliminar.

En esta pantalla el usuario tiene dos opciones:

1. Pulsar el botón "Cancel", con lo que se vuelve a tener en el frontend la pantalla del detalle (ver Imagen 44).
2. Pulsar el botón "Delete", con lo que aparecerá el siguiente Dialog de confirmación de la eliminación:

Imagen 46. CE - Escenario 1 - Dialog confirmar eliminación desde pantalla de acción

En este instante, el usuario tiene la posibilidad de decidir si realmente quiere eliminar la notificación. Esto se ha hecho para evitar eliminar una notificación por una decisión apresurada (y algunas veces equivocada), por parte del usuario.

Como es lógico, debido a que se dispone de dos botones, el usuario dispone de nuevo, de dos opciones:

1. Si pulsa el botón "No", se cierra el Dialog de confirmación de la eliminación (ver Imagen 46), y se vuelve a tener en el frontend la pantalla de acciones (ver Imagen 45).
2. Si pulsa el botón "Yes", la notificación se eliminará, y se cerrará la aplicación. Como modo informativo, el usuario visualizará un Toast, que le indicará si la notificación ha sido eliminada correctamente. A continuación, se puede ver que la aplicación se ha cerrado, y se puede visualizar el Toast de notificación.

Imagen 47. CE - Escenario 1 - Confirmación de la eliminación

Si por algún problema en la red, la notificación no se pudiese eliminar, el mensaje del Toast sería distinto, como es de esperar, informaría que la notificación no se habrá eliminado.

Si el usuario en este momento decidiera abrir la aplicación de notificaciones, vería que ésta no está en el listado, debido a que la notificación ha sido eliminada previamente.

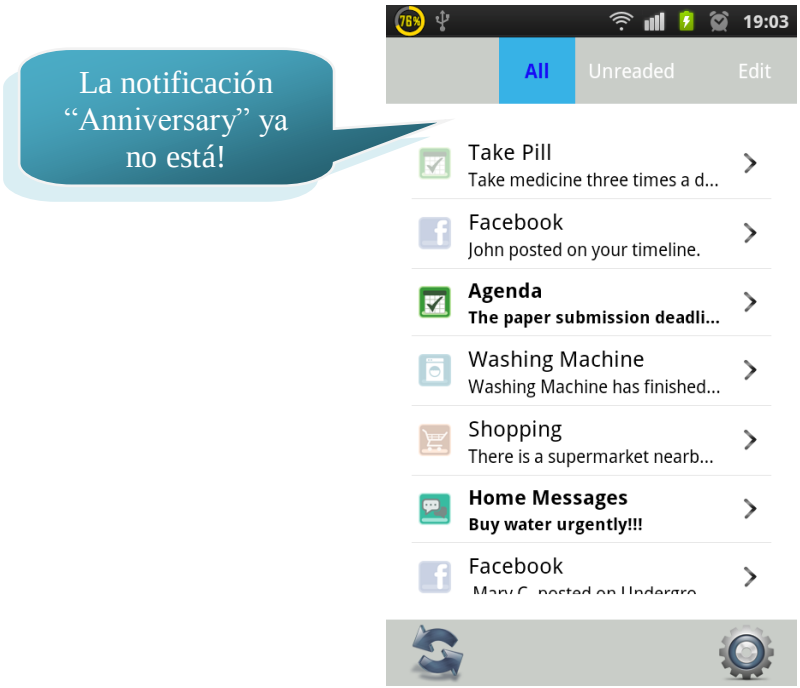

Imagen 48. CE - Escenario 1 - Todas las notificaciones (con eliminación)

La última notificación que recibe el usuario, siempre aparece como primer elemento de la lista, y como se puede apreciar; la notificación "Anniversary" no está en la lista.

6.1.1.2. Segunda opción de eliminación de una notificación

La segunda opción de eliminación, es mediante la vista de edición de la pantalla principal. Para ello, y suponiendo que el usuario se encuentra en la pantalla de detalle (ver Imagen 44), el usuario deberá pulsar el botón "Notifications". Con esto, se mostrará la pantalla donde se visualizan todas las notificaciones. En este caso, la notificación recibida aparecerá en el listado (en la primera posición), como se muestra a continuación.

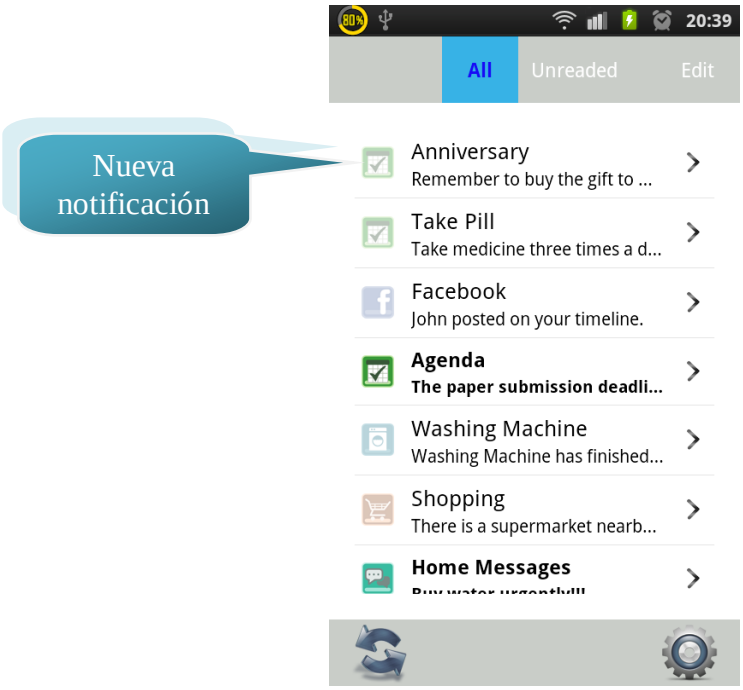

Imagen 49. CE - Escenario 1 - Todas las notificaciones (sin eliminación)

Como se puede ver, se muestra la pantalla con la vista de todas las notificaciones, donde el primer elemento de la lista es la última notificación que ha recibido el usuario (*Anniversary*). Como peculiaridad, se puede ver que las notificaciones que ya han sido leídas (se ha visualizado el detalle), se muestran en un color más opaco, y las que no han sido leídas, se muestran con un color más vivo.

Para eliminar la notificación, lo primero que se ha de hacer el pulsar el Tab Edit, con lo que se visualizarán todas las notificaciones, pero en lugar de mostrar el icono de la notificación, se mostrará un botón que permite eliminar la notificación. La pantalla en cuestión es la siguiente:

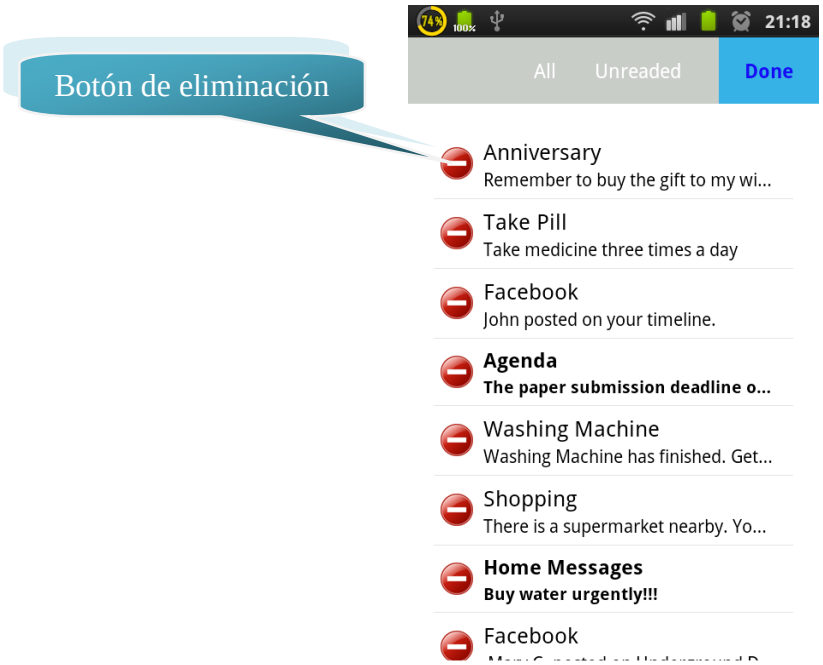

Imagen 50. CE - Escenario 1 - Pantalla de edición

Lo primero que puede apreciar el usuario, es que el Tab "Edit" cambia de nombre a "Done". Esto le indica que está es una pantalla de edición, y que una vez que haya eliminado la/las notificación/es que desee, podrá volver a la pantalla en la que se encontraba antes de pulsar el Tab "Edit". También se puede ver que la barra inferior se oculta, ya que en esta pantalla solo debe estar operativa la función de eliminar las notificaciones.

En este caso, para eliminar la notificación, el usuario deberá que pulsar el "botón de eliminación". Una vez presionado, al igual que para la primera opción de eliminación, aparecerá un Dialog de confirmación.

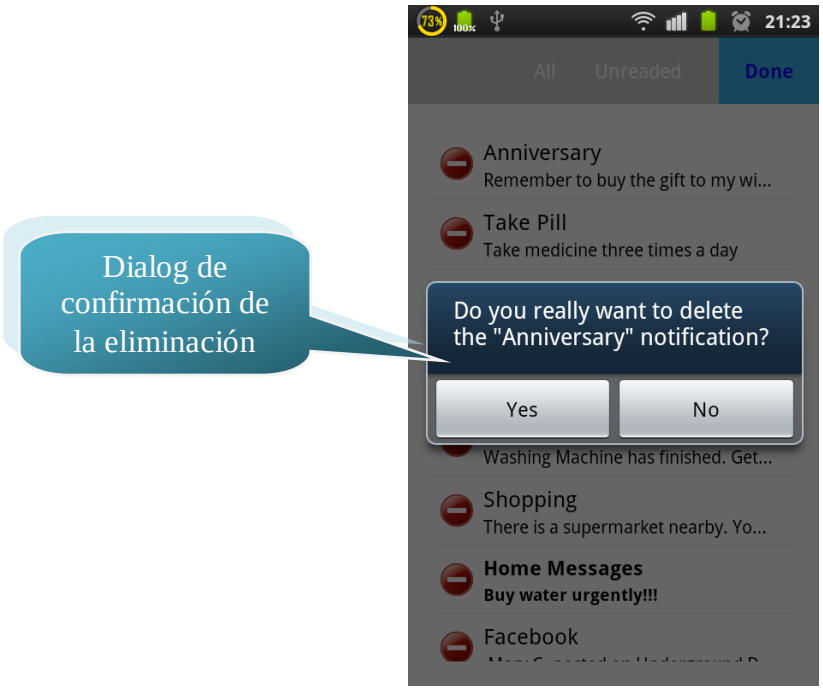

Imagen 51. CE - Escenario 1 - Dialog confirmar eliminación desde Tab de edición

Al igual que para la primera opción, en este instante, el usuario tiene la posibilidad de decidir si realmente quiere eliminar la notificación, y de nuevo se tienen dos opciones:

1. Si pulsa el botón "No", se cierra el Dialog de confirmación de la eliminación (ver Imagen 51), y se vuelve a tener en el frontend la pantalla de edición (ver Imagen 50).
2. Si pulsa el botón "Yes" se realizarán las siguientes opciones:

    - Se eliminará la notificación
    - Se cerrará el Dialog de confirmación
    - Se mostrará un Toast de notificación de la eliminación
    - Se volverá a la pantalla de edición, por tanto, la siguiente pantalla que verá el usuario será la siguiente:

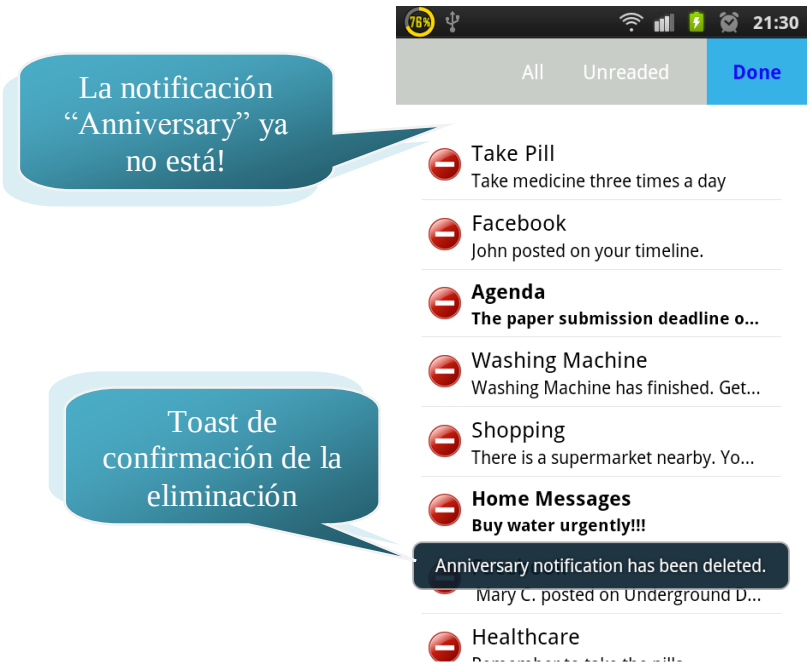

Imagen 52. CE - Escenario 1 - Pantalla de edición después de una eliminación

En este instante, el usuario se percatará que la notificación ha sido eliminada debido al Toast de información, y debido a que la notificación ya no aparecerá en la lista. Para finalizar, el usuario puede pulsar el Tab Edit (renombrado Done) con lo que volverá a la vista de todas las notificaciones (ver Imagen 48). Además el Tab "Edit", volverá a tener el mismo nombre (dejará de llamarse "Done").

En este punto, suponiendo que el usuario ya ha realizado la acción deseada, y no desea interactuar más con la aplicación, para salir de la misma, solo tiene que pulsar el botón de retroceso del dispositivo, con lo que se mostrará el siguiente Dialog de confirmación, de salida de la aplicación:

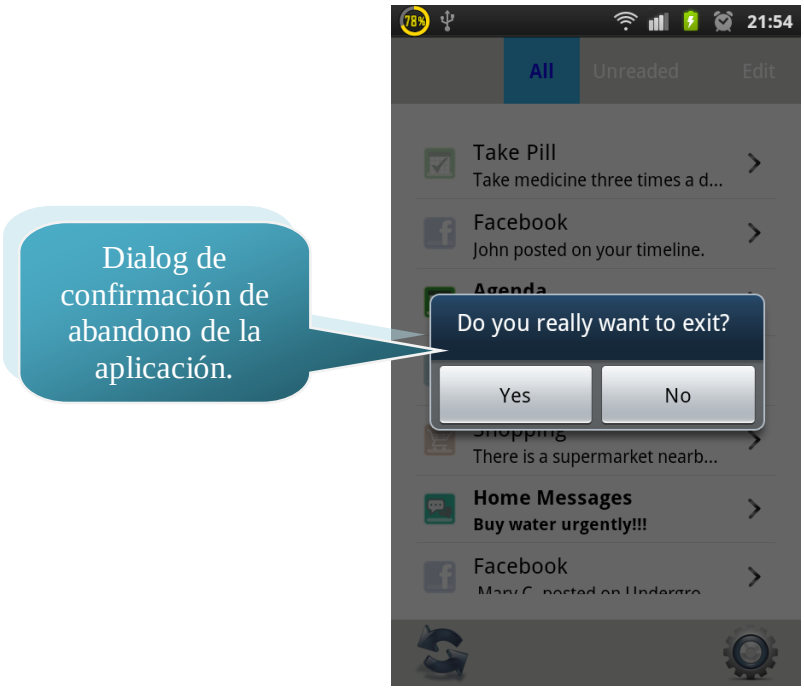

Imagen 53. CE - Escenario 1 - Dialog confirmar abandono de aplicación

El usuario tendrá de nuevo dos opciones para elegir:

1. Si pulsa el botón "No", se volverá a tener como frontend la pantalla de todas las notificaciones (ver Imagen 48).
2. Si pulsa el botón "Yes" la aplicación se cerrará.

6.2. Escenario 2

El usuario se encuentra conduciendo **solo** de vuelta a casa, y pasa cerca de una joyería.

En este caso, el sistema detecta, de nuevo gracias a su integración con *MoRE* [*1*], que el usuario viaja solo, y reconfigura el sistema enviando un *JSON* de configuración, para que se activen los siguientes componentes:

- Speech
- Dialog (con botones)

Como se puede apreciar, uno de los dos componentes a activar en este caso, a diferencia de los activados en el Escenario 1, es sonoro. Esto es debido a que el usuario no está en compañía de nadie y puede recibir una notificación privada como ésta, sin temor a que nadie descubra sus propósitos.

La notificación se mostrará en el dispositivo móvil de la siguiente forma:

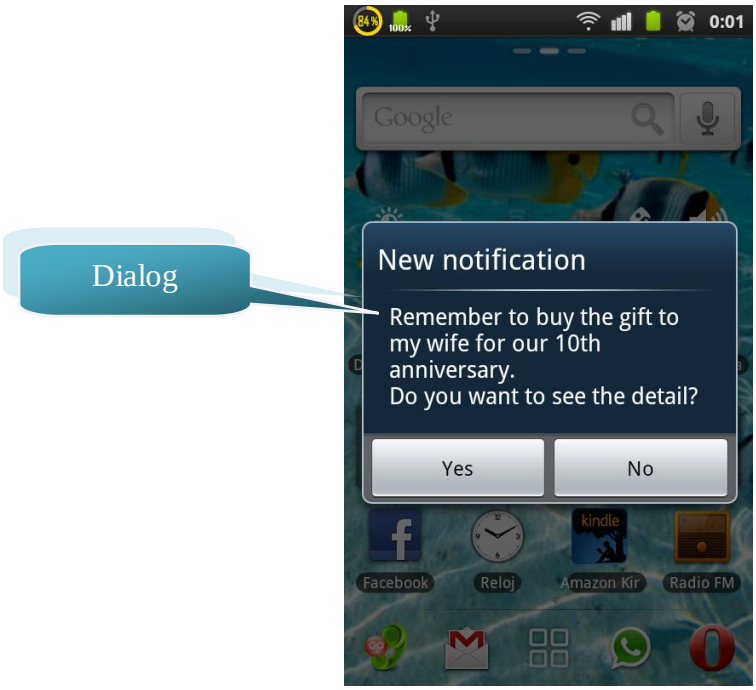

Imagen 54. CE - Escenario 2 - Notificación recibida

Para este escenario solo se presenta un componente visual (un Dialog). A diferencia del Dialog mostrado en el Escenario 1 (ver Imagen 42), el Dialog que se muestra en este Escenario, posee botones que permite al usuario interactuar con el sistema de la siguiente manera:

1. Si pulsa el botón "No", el Dialog se cierra.
2. Si pulsa el botón "Yes" se abre el detalle de la notificación (ver Imagen 44), permitiendo al usuario realizar las mismas acciones explicadas para el Escenario 1, después de mostrar la Imagen 44.

En la realización de este caso de estudio, se ha comprobado que al recibir la notificación, con el fichero de configuración generado para este contexto, el móvil reproduce el mensaje del servicio, mediante el Speech.

6.3. Escenario 3

El usuario se encuentra reunido en su trabajo con unos clientes muy importantes.

Para este caso, debido a que el contexto en el que se encuentra el usuario, requiere que no se le moleste, la notificación debe llegar al dispositivo de una forma silenciosa, pero dejando constancia de que ha llegado una nueva notificación, por tanto, para este caso, solo se activará el StatusBar.

De esta forma el usuario no será interrumpido por la recepción de la notificación, sin embargo, una vez haya terminado la reunión y el usuario mire su dispositivo, sabrá que tiene un nuevo mensaje, por el icono que se encuentra en la barra de estado. La pantalla que vería sería la siguiente:

Imagen 55. CE - Escenario 3 - Notificación recibida

En este caso, si posteriormente el usuario desea gestionar sus notificaciones, abriendo la aplicación de gestión de notificaciones, la primera pantalla que verá será la siguiente:

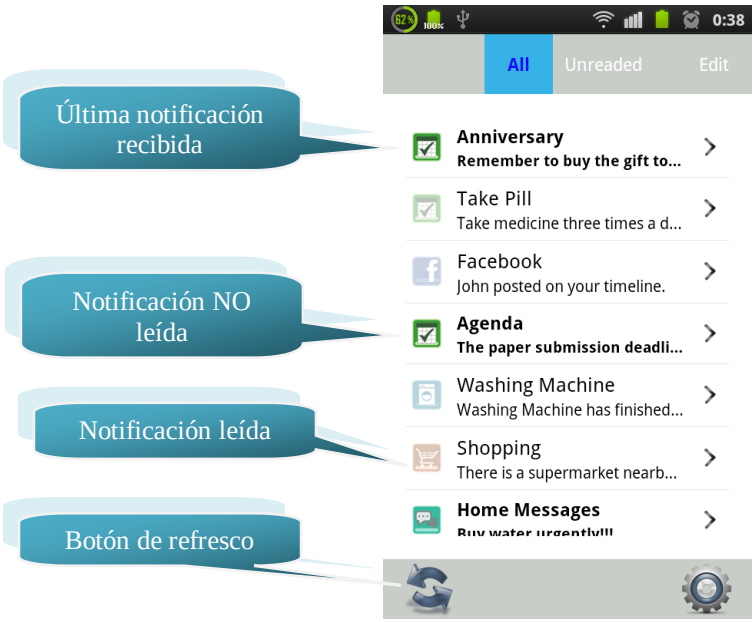

Imagen 56. CE - Escenario 3 - Pantalla inicial aplicación

En esta primera vista, el usuario tendrá una lista con todas las notificaciones que han recibido, pudiendo diferenciar entre las que ha leído y las que no, puesto que las notificaciones que ya se han leído, aparecen con un color más atenuado.

Si el usuario desea visualizar solo las notificaciones que no han sido leídas, basta con pulsar el Tab “Unreaded”, con lo que obtendrá la siguiente vista:

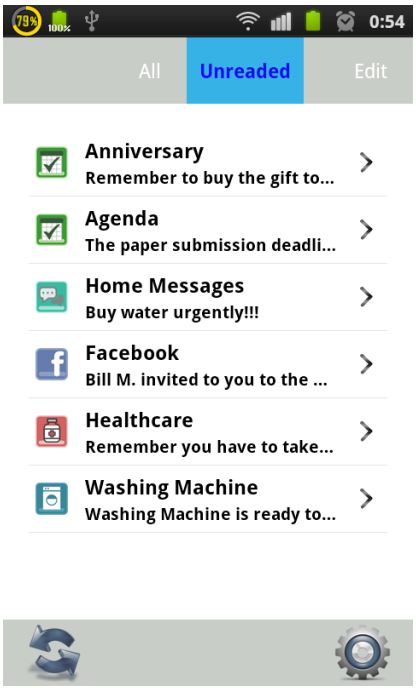

Imagen 57. CE - Escenario 3 - Notificaciones NO leídas

Como se puede ver, en esta pantalla solo aparecen las notificaciones que no han sido leídas. Si todas las notificaciones estuviesen leídas, al pulsar el Tab "Unreaded", aparecería la lista vacía, y un Toast de información de la siguiente forma:

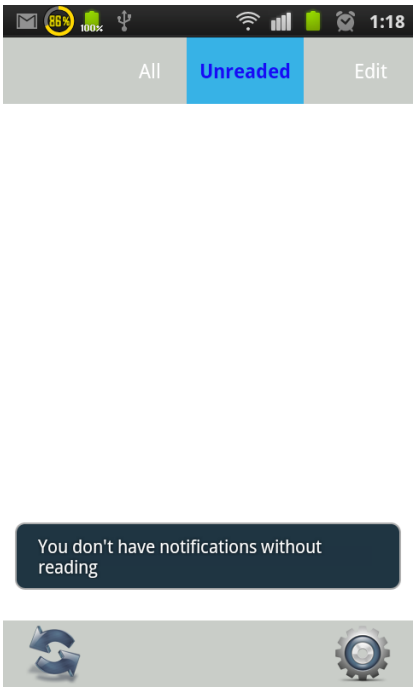

Imagen 58. CE - Escenario 3 - Sin notificaciones pendientes de leer

En este mismo entorno, si el usuario decide pulsar el Tab "Edit" se mostrará la siguiente pantalla, informando que no hay elementos a editar:

Imagen 59. CE - Escenario 3 - Sin notificaciones para editar

Para finalizar, se dispone del botón de refresco, para las vistas de los Tabs "All" y "Unreaded", que permite obtener las notificaciones del servidor, por si por algún motivo no se estuviesen visualizando todas. Además, si el usuario selecciona una notificación, se mostrará el detalle de la misma (ver Imagen 44).

En este entorno, el usuario también puede realizar todas las interacciones para la eliminación de notificaciones, explicadas para el Escenario 1.

Un último detalle que se puede apreciar de la pantalla principal (ver Imagen 56, Imagen 57, Imagen 59), es que cada vez que se pulsa un Tab, se cambia el color del mismo, para que el usuario sepa en todo momento qué es lo que está visualizando.

Con estos tres escenarios, se ha mostrado al completo toda la funcionalidad del sistema desarrollado, así como todas las posibles interacciones que el usuario puede realizar sobre la aplicación diseñada; como centro de gestión de las notificaciones recibidas.

7. CONCLUSIONES

Gracias a la tecnología de dispositivos móviles que existe actualmente, se ha podido diseñar e implementar, un sistema que mediante la integración con *MoRE* [*1*], permite la ***RECONFIGURACIÓN*** de notificaciones (en dispositivos de telefonía móvil), en tiempo de ejecución, teniendo en cuenta el contexto en el que se encuentre el usuario.

Para este propósito, se ha trabajado con el SO Android, por lo que antes de empezar con la implementación de cada uno de los subsistemas, que han permitido alcanzar los objetivos de esta Tesina, se ha realizado un estudio previo de la arquitectura Android (base, composición, estructura y desarrollo de aplicaciones).

Se ***han implementado de forma genérica los componentes básicos Android*** (*Status Bar*, *Toast*, *Dialog*, *Sound*, *Vibration* y *Lights*), que permiten avisar al usuario de la llegada de una nueva notificación.

Por otro lado, se ha implementado un ***controlador***, que a través del *JSON* recibido de la llamada al servidor (Integración con *MoRE* [*1*]), es capaz de activar y desactivar los componentes indicados en dicho *JSON*.

Para tener un centro de gestión de las notificaciones recibidas, y para mostrar la operatividad del sistema implementado, se ha ***desarrollado una Aplicación de notificaciones Android maestro-detalle***. En este ámbito, también se han implementado unos *servicios web*, que permiten interactuar con el servidor, desde el cual se pueden obtener las notificaciones que se han enviado a la aplicación.

8. REFERENCIAS

1. C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. Computer, 42(10):37–43, 2009.
2. Nadav Savio, Jared Braiterman. Design Sketch: The Context of Mobile Interaction. International Journal of Mobile Marketing, June 2007.
3. M. Gil, P. Giner, V. Pelechano. Personalization for unobtrusive service interaction. Personal and Ubiquitous Computing Volume 16 Issue 5, June 2012 Pages 543-561.
4. W. W. Gibbs. Considerate computing. Scientific American, 292(1):54–61, 2005.
5. Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonckt J (2003) A unifying reference framework for multi-target user interfaces. Interact Comput 15(3):289–308.
6. Cetina, C. Achieving Autonomic Computing through the Use of Variability Models at Run-time. Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2010.
7. http://developer.android.com/intl/es/guide/components/index.html
8. http://www.vogella.com/articles/AndroidCloudToDeviceMessaging/article.html
9. http://codehenge.net/blog/2011/05/customizing-android-listview-item-layout/
10. http://senior.ceng.metu.edu.tr/2009/praeda/2009/01/11/a-simple-restful-client-at-android/
11. http://stackoverflow.com/questions/3505930/make-an-http-request-with-android
12. http://www.instropy.com/2010/06/14/reading-a-json-login-response-with-android-sdk/
13. http://stackoverflow.com/questions/9598707/gson-throwing-expected-begin-object-but-was-begin-array
14. http://www.jesusgallardo.es/2011/06/02/threads-en-android-mostrar-progressdialog-durante-get-request/
15. http://www.vogella.com/articles/AndroidListView/article.html
16. http://stackoverflow.com/questions/658469/deserializing-from-json-into-php-with-casting
17. http://www.sohailaziz.com/2012/04/passing-custom-objects-between-android.html
18. http://geekyouup.blogspot.com.es/2010/04/creating-android-activity-with-no-ui.html
19. http://www.androidhive.info/2012/01/android-text-to-speech-tutorial/
20. http://stackoverflow.com/questions/9437423/show-activity-only-once-when-clicking-the-notification-icon
21. http://www.dosideas.com/noticias/java/314-introduccion-a-los-servicios-web-restful.html