



**Universidad de Valladolid**

**Escuela de Ingeniería Informática (Segovia)**

**Grado en Ingeniería Informática de Servicios y  
Aplicaciones**

**Gestionando una Base de Datos Cinematográfica  
en un Dispositivo Android**

***Autor:* Jon Casado Gómez**

***Tutores:* Miguel Ángel Martínez Prieto  
Aníbal Bregón Bregón**



# Resumen

Cada día que pasa se publican en la Red miles de documentos nuevos. Con relativa frecuencia aparecen nuevas posibilidades de uso de Internet, y asiduamente se están inventando nuevos términos para poder entenderse en este mundo que no para de crecer. Una de estas posibilidades es la conocida como “Web de Datos”. Está es una simplificación de la Web Semántica, que proponía inicialmente superar las limitaciones de la red actual introduciendo descripciones explícitas del significado. Este tipo de datos semánticos se describen habitualmente mediante el lenguaje RDF. LinkedMDB es en la actualidad la mayor base de datos sobre cine en formato RDF que existe, este conjunto de datos puede ser consultado de forma online, obteniendo múltiples resultados, cada uno de los cuales posee gran cantidad de enlaces a otras páginas. Lo que lo diferencia es que contiene la información semántica que permite a las máquinas conocer el significado de cada uno de los enlaces, haciendo que éstas puedan extraer y filtrar información de manera más avanzada. Este proyecto pretende crear un método de consulta móvil para LinkedMDB que se beneficie de esta característica para obtener un acceso completo a él, almacenándolo localmente y poder consultarlo sin necesidad de disponer de acceso a la red. Para ello se utilizará el formato HDT que permite reducir la redundancia del lenguaje RDF, para favorecer el procesamiento por parte del dispositivo y la gestión de los datos. Además se realizará un estudio sobre el consumo de batería de las consultas realizadas a un conjunto de datos en formato HDT. Para poder llevar esto a cabo se desarrollarán dos aplicaciones iguales, pero implementadas con lenguajes diferentes, lo cual dará lugar a una nueva forma de ver las comparativas entre diferentes lenguajes, ya que en la actualidad se centran en la velocidad de consulta y no en su rendimiento en cuanto al consumo.



# Índice

Resumen.....	3
Índice.....	5
Índice de ilustraciones.....	9
Índice de tablas.....	13
Capítulo 1. Introducción.....	15
1.1. Descripción del proyecto.....	17
1.1.1. Motivación.....	17
1.1.2. Alcance.....	17
1.1.3. Objetivos .....	18
Capítulo 2. La Web de Datos.....	21
2.1. Introducción.....	23
2.2. La Web Semántica.....	25
2.3. Linked Data.....	27
2.4. RDF (Resource Description Framework).....	27
2.4.1. Ejemplo.....	29
2.5. SPARQL.....	31
2.6. HDT (Header, Dictionary, Triples).....	32
Capítulo 3. El Proyecto.....	35
3.1. Introducción.....	37
3.2. LinkedMDB.....	37
3.3. RDF/HDT.....	42
3.4. El proyecto: HDT-Java y HDT-Cpp.....	42
3.5. Estado del arte.....	43
3.5.1. Linked data from your pocket.....	43
3.5.2. HdTourist.....	45
3.5.3. HDT-it!.....	45
Capítulo 4. Análisis.....	47
4.2. Análisis.....	49
4.2.1. Objetivos.....	49
4.2.2. Restricciones.....	49
4.2.3. Especificación de requisitos funcionales.....	50
4.2.4. Especificación de requisitos no funcionales.....	51
4.2.5. Modelo de casos de uso.....	52
4.2.6. Modelo conceptual.....	63

Capítulo 5. Desarrollo del proyecto.....	69
5.1. Planificación, Presupuesto y Coste real.....	71
5.1.1. Estimaciones del proyecto .....	71
5.1.2. Planificación.....	71
5.1.3. Presupuesto.....	74
5.1.4. Coste real.....	78
5.1.5. Conclusiones.....	80
5.2. Diseño.....	81
5.2.1. Arquitectura lógica.....	81
5.2.2. Diagrama de clases.....	82
5.3. Implementación.....	86
5.3.1. Versiones de Android compatibles con el proyecto.....	86
5.3.2. Librería Java-HDT.....	87
5.3.3. Herramientas utilizadas.....	87
5.3.4. Programación de la Aplicación.....	92
5.4. Optimización.....	101
5.4.1. Adaptación a Java.....	102
5.4.2. Preparación de la librería.....	102
5.4.3. Identificar las clases y métodos de C++.....	104
5.4.4. Diseñar las clases de Java.....	107
5.4.5. Generar la declaración de las funciones de JNI .....	115
5.4.6. Escribir el cuerpo de las funciones de JNI.....	118
5.4.7. Adaptación a Android.....	126
Capítulo 6. Benchmarking .....	137
6.1. Introducción.....	139
6.2. Dispositivo móvil y cláusulas del benchmark.....	140
6.3. Pruebas.....	141
6.3.1. Pruebas SPO Java.....	142
6.3.2. Pruebas SPO C++.....	143
6.3.3. Conclusiones SPO Java vs C++.....	144
6.3.4. Pruebas SP? Java.....	145
6.3.5. Pruebas SP? C++.....	146
6.3.6. Conclusiones SP? Java vs C++.....	147
6.3.7. Pruebas S?O Java.....	150
6.3.8. Pruebas S?O C++.....	151
6.3.9. Conclusiones S?O Java vs C++.....	152

6.3.10. Pruebas S?? Java.....	154
6.3.11. Pruebas S?? C++.....	155
6.3.12. Conclusiones S?? Java vs C++.....	156
Capítulo 7. Manuales.....	159
7.1. Guía de instalación.....	161
7.2. Manual de usuario.....	164
7.2.1. Opción películas .....	165
7.2.2. Opción actores .....	166
7.2.3. Opción directores .....	167
7.2.4. Opción consulta avanzada .....	169
7.2.5. Opción realizar pruebas.....	170
Capítulo 8. Conclusiones .....	173
8.1. Conclusiones generales.....	175
8.2. Dificultades.....	175
8.3. Trabajo futuro.....	176
Capítulo 9. Bibliografía.....	179
Capítulo 10. Anexos.....	185
10.1. Contenido del CD.....	187
10.2. Entorno de desarrollo.....	187
10.3. Continuación de la Adaptación a Java.....	194
10.3.1. Crear la clase “main” del proyecto.....	194
10.3.2. Compilar y probar la librería nativa.....	195





# Índice de ilustraciones

Ilustración 1: Ejemplo de búsqueda en un navegador no semántico.....	22
Ilustración 2: Diferencia entre la web actual y la web semántica.....	24
Ilustración 3: Ejemplo de Grafo RDF.....	26
Ilustración 4: Ejemplo de grafo RDF que describe un recurso.....	28
Ilustración 5: HDT: Cabecera, Diccionario y Ternas.....	30
Ilustración 6: Ternas de HDT.....	31
Ilustración 7: Ejemplo sobre el funcionamiento general de ambas aplicaciones.....	35
Ilustración 8: Grafo de enlaces en Linked Movie DataBase.....	36
Ilustración 9: Estructura de una consulta a LinkedMDB.....	39
Ilustración 10: Arquitectura del proyecto Linked data from your pocket.....	42
Ilustración 11: Diferentes pantallas de la aplicación RDF Browser.....	42
Ilustración 12: Capturas de la aplicación HdTourist.....	43
Ilustración 13: Pantalla inicial de HDT-it!.....	44
Ilustración 14: Diagrama de casos de uso.....	51
Ilustración 15: Resultados de la consulta de una película específica en LinkedMDB.....	61
Ilustración 16: Resultados de la consulta de un actor concreto en LinkedMDB.....	62
Ilustración 17: Relación entre los resultados en LinkedMDB.....	63
Ilustración 18: Modelo de consulta SP? en forma de triple.....	64
Ilustración 19: Modelo de consulta S?? en forma de triple.....	64
Ilustración 20: Modelo de consulta ?P? en forma de triple.....	64
Ilustración 21: Modelo de consulta SPO en forma de triple.....	65
Ilustración 22: Planificación de etapas e iteraciones.....	72
Ilustración 23: Diagrama de Gantt.....	72
Ilustración 24: Planificación real de etapas e iteraciones.....	76
Ilustración 25: Planificación real del diagrama de Gantt.....	76
Ilustración 26: Arquitectura lógica.....	80
Ilustración 27: Diagrama de clases.....	82
Ilustración 28: Posición de JNI dentro de la JVM.....	89
Ilustración 29: Proceso de adaptación de C++ a Java.....	100
Ilustración 30: Resultado de la búsqueda de los makefiles.....	101
Ilustración 31: Funcionamiento de JNIEnv.....	116
Ilustración 32: Fichero antes de ser modificado.....	127
Ilustración 33: Fichero después de ser modificado.....	128
Ilustración 34: Cabecera ctype_base.h antes de la modificación.....	129

Ilustración 35: Cabecera ctype_base.h después de la modificación.....	129
Ilustración 36: Clase TripleListDisk.cpp antes.....	130
Ilustración 37: Clase TripleListDisk.cpp después.....	130
Ilustración 38: Makefile para generar la librería nativa con las toolchain de ARM.....	130
Ilustración 39: Dispositivo móvil.....	138
Ilustración 40: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta SPO.....	142
Ilustración 41: Diferencia de tiempos de ejecución entre la aplicación Java y C++ sobre la consulta SPO.....	143
Ilustración 42: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta SP?.....	146
Ilustración 43: Diferencia de tiempos de ejecución entre la aplicación Java y C++ de la consulta SP?.....	147
Ilustración 44: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta S?O.....	150
Ilustración 45: Comparación de tiempos de ejecución entre la aplicación Java y C++ de la consulta S?O.....	151
Ilustración 46: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta S??.....	154
Ilustración 47: Diferencia de tiempos de ejecución entre la aplicación Java y C++ de la consulta S??.....	155
Ilustración 48: Carpeta Dispositivo.....	159
Ilustración 49: Carpeta mnt.....	159
Ilustración 50: Carpeta sdcard.....	160
Ilustración 51: carpeta Download.....	160
Ilustración 52: Ejecutables de HDT.....	161
Ilustración 53: Permisos de instalación.....	161
Ilustración 54: Instalación completada.....	161
Ilustración 55: Pantalla bienvenida.....	162
Ilustración 56: Pantalla inicial.....	162
Ilustración 57: Pantalla Consulta Películas.....	163
Ilustración 58: Pantalla Resultado de la Consulta Películas.....	164
Ilustración 59: Pantalla Consulta Actores.....	165
Ilustración 60: Pantalla Consulta Directores.....	166
Ilustración 61: Pantalla Resultado de la Consulta Directores.....	166
Ilustración 62: Pantalla Consulta Avanzada.....	167
Ilustración 63: Pantalla Resultado de la Consulta Avanzada.....	168
Ilustración 64: Pantalla Realizar Pruebas.....	168

Ilustración 65: Pantalla Resultado de Realizar Pruebas.....	169
Ilustración 66: Versión de la librería C.....	187
Ilustración 67: Pantalla inicial del SDK Android.....	188
Ilustración 68: Ventana para agregar plugins de Eclipse.....	190
Ilustración 69: Pantalla para añadir el NDK a Eclipse.....	191



# Índice de tablas

Tabla 1: Estadísticas de Linked Movie DataBase.....	46
Tabla 2: Número de enlaces por propiedad en Linked Movie DataBase.....	47
Tabla 3 RF-01: Cargar un fichero en formato HDT.....	57
Tabla 4 RF-02: Seleccionar el tipo de consulta.....	57
Tabla 5 RF-03: Búsqueda de información.....	57
Tabla 6 RF-04: Visualización de los resultados de la búsqueda.....	57
Tabla 7 RF-05: Elección del test.....	58
Tabla 8 RF-06: Lectura del fichero con las consultas.....	58
Tabla 9 RF-07: Visualización de los resultados del test.....	58
Tabla 10 RNF-01: Espacio de almacenamiento.....	58
Tabla 11 RNF-02: Dispositivo móvil.....	59
Tabla 12 RNF-03: Dispositivo móvil para la ejecución de los test.....	59
Tabla 13 RNF-04: Adaptar la visualización a la resolución de pantalla.....	59
Tabla 14 CU-01: Visualizar la información de los actores .....	61
Tabla 15 CU-02: Visualizar el listado de directores.....	62
Tabla 16 CU-03: Obtener las películas dirigidas por un determinado director.....	63
Tabla 17 CU-04: Mostrar el catálogo de películas.....	64
Tabla 18 CU-05: Obtener información de una determinada película.....	65
Tabla 19 CU-06: Realizar una consulta avanzada.....	66
Tabla 20 CU-07: Visualizar los resultados de la búsqueda.....	67
Tabla 21 CU-08: Seleccionar test.....	68
Tabla 22 CU-09: Ejecutar pruebas.....	69
Tabla 23: Etapas e iteraciones del proyecto.....	79
Tabla 24: Presupuesto Hardware total.....	82
Tabla 25: Presupuesto Software total.....	83
Tabla 26: Presupuesto de Desarrollo total.....	84
Tabla 27: Presupuestos totales.....	84
Tabla 28: Coste real total del Hardware.....	86
Tabla 29: Coste real total del Desarrollo.....	87
Tabla 30: Coste real final .....	87
Tabla 31: Resultados medios obtenidos de las pruebas en Java de la Consulta SPO.....	150
Tabla 32: Resultados medios obtenidos de las pruebas en C++ de la Consulta SPO.....	151
Tabla 33: Resultados medios obtenidos de las pruebas en Java de la Consulta SP?.....	154
Tabla 34: Resultados medios obtenidos de las pruebas en C++ de la Consulta SP?.....	155

Tabla 35: Resultados medios obtenidos de las pruebas en Java de la Consulta S?O.....	158
Tabla 36: Resultados medios obtenidos de las pruebas en C++ de la Consulta S?O.....	159
Tabla 37: Resultados medios obtenidos de las pruebas en Java de la Consulta S??.....	162
Tabla 38: Resultados medios obtenidos de las pruebas en C++ de la Consulta S??.....	163

## **Capítulo 1. Introducción**





## 1.1. Descripción del proyecto

El siguiente documento presenta el sistema “Gestionando una Base de Datos Cinematográfica en un Dispositivo Android”, realizado por Jon Casado Gómez como parte del Trabajo Fin de Grado del Grado en Ingeniería Informática de Servicios y Aplicaciones.

### 1.1.1. Motivación

En un principio este proyecto fue ideado con la intención de proporcionar una aplicación diferente y novedosa en lo que se refiere a la consulta de información sobre *el séptimo arte*, es decir, el cine.

Esta diferencia de la que hablamos, es la de utilizar el formato HDT como herramienta de compresión de información. El uso de dicho formato nos limitará enormemente en cuanto al uso de otro tipo de tecnologías más estandarizadas, pero también nos proporcionará otras ventajas que otras herramientas no son capaces de ofrecernos.

Pero como en la actualidad ya hay diferentes aplicaciones que han podido aprovechar las capacidades de este formato, pensamos que hacer una aplicación que únicamente permitiese al usuario consultar información sobre el cine sería algo que no resultaría tan novedoso, por ello decidimos que en este proyecto incorporaríamos algo realmente distintivo, dicho elemento diferenciador consiste en desarrollar una aplicación con código nativo, es decir escrita en el lenguaje de programación C++. Con esto conseguiremos distinguir a nuestra aplicación del resto ya que la gran mayoría de las aplicaciones no incorpora este tipo de tecnología en la actualidad, la cual, pese a su dificultad de implantación, nos va a proporcionar ciertas ventajas como veremos posteriormente.

También, aprovechando la oportunidad que nos ofrece el desarrollo de este proyecto, decidimos establecer una comparativa entre la aplicación implementada con Java y la creada con C++. Dicha comparativa se realizará mediante una serie de consultas a nuestro conjunto de información en formato HDT, en donde mediremos los tiempos que tardan en ejecutarse las consultas y el consumo de batería que tienen estos sobre nuestro dispositivo móvil. Este tipo de estudio es algo diferente, puesto que por lo que hemos podido investigar, prácticamente todas las comparativas que hay en la actualidad sobre estos lenguajes de programación se centran en la velocidad de la consulta y no en su consumo de energía, por lo que además servirá de precedente para otros trabajos y nos arrojará resultados que nos permitirán establecer diferentes métricas.

### 1.1.2. Alcance

La Web de Datos enlazados es el mecanismo mediante el cual los datos estructurados con información semántica (conocidos como la Web Semántica) se vinculan e interconectan entre sí, de la misma forma que lo hacen los enlaces de las páginas web.

## Capítulo 1. Introducción

Actualmente, el gran potencial de la Web de Datos, en el ámbito de los dispositivos móviles, esta comenzando a ser explotado, aunque sigue existiendo un desconocimiento general sobre esta tecnología, las considerables tareas de promoción llevadas a cabo y las numerosas ventajas que proporciona su uso, hacen que muchos desarrolladores de gran importancia en el mundo de la telefonía móvil estén optando por comenzar a utilizar esta tecnología.

En nuestro caso mediante el formato binario HDT, vamos a conseguir reducir uno de los principales problemas de RDF. Dicho problema tiene que ver con la redundancia de información que RDF incluye en sus conjuntos de datos.

Otro problema que con el que nos encontramos es que la base de datos sobre cine que vamos a utilizar en este proyecto, LinkedMDB, que se encuentra en formato RDF, ocupa mucho espacio de almacenamiento, y al encontramos en un dispositivo móvil tendremos que tenerlo en cuenta. Aunque dicho problema no nos va a resultar un gran quebradero de cabeza, ya que con el uso de HDT también conseguiremos reducir enormemente el dicho coste, gracias al almacenamiento comprimido.

Por último, y como ya hemos especificado en el apartado anterior, una de las preguntas que ha dado pie al desarrollo de este proyecto ha sido la siguiente: *¿Cual será el consumo de batería que tendrán las consultas a un conjunto de datos en formato HDT a un dispositivo móvil?*

Esta pregunta nos sirve para establecer el alcance principal de este proyecto, el cual esta establecido en obtener unas métricas y comparativas, que serán analizadas minuciosamente y deberán proporcionar unos resultados tangibles.

### 1.1.3. Objetivos

Con el desarrollo de este proyecto pretendemos realizar una aplicación de consulta móvil para LinkedMDB, para el caso concreto de la consulta offline sobre información relacionada con el cine. Para ello el sistema tendrá que hacer uso de las capacidades del formato comprimido RDF, conocido como HDT, con el que podemos almacenar grandes cantidades de información semántica en el móvil, facilitando su recuperación y consulta en cualquier momento. También, como ya hemos destacado en los apartados anteriores, vamos a analizar el consumo de batería tras la ejecución de un numero determinado de consultas en el dispositivo móvil. Para poder establecer una comparativa, utilizaremos dos lenguajes diferentes, por un lado desarrollaremos una aplicación mediante el lenguaje Java, y por el otro desarrollaremos una aplicación que utilizara el lenguaje C++ a través de JNI.

Por tanto, los objetivos principales del proyecto son:

- Realizar un estudio comparativo, entre las aplicaciones que desarrollaremos, sobre el consumo de batería que se produce en el dispositivo móvil al ejecutar un determinado número de consultas sobre un conjunto de datos en formato HDT.

## *Capítulo 1. Introducción*

- Desarrollar dos aplicaciones móviles que permitan realizar diferentes tipos de consultas a un conjunto de datos en formato HDT, donde cada una de las aplicaciones se desarrollará de una forma diferente. Esta distinción será el lenguaje de programación que utilizarán para ser implementadas.
- Permitir consultar información cinematográfica al conjunto de datos LinkedMDB en formato HDT.

El nombre de la aplicación desarrollada, mediante el lenguaje Java, en este proyecto será **HDT-Java**, por otra parte, la aplicación desarrollada mediante C++, recibirá el nombre de **HDT-Cpp**.



## **Capítulo 2. La Web de Datos**



## 2.1. Introducción

En la actualidad la cantidad de información que se genera a diario sobrepasa lo que podamos llegar a pensar o incluso imaginar, Internet sigue creciendo a un ritmo vertiginoso y constantemente se mejoran los canales de comunicación con el fin de aumentar la rapidez de envío y recepción de datos.

La World Wide Web es quien se encarga de mostrar toda esta información, lo hace a través de la presentación de documentos escritos en HTML.

HTML es el lenguaje de marcado por excelencia y el encargado de crear el hipertexto en Internet. Es un lenguaje apropiado para adecuar el aspecto visual de un documento e incluir objetos multimedia en el texto como pueden ser imágenes, vídeos, música, flash, etc, pero se queda muy corto a la hora de **categorizar** los elementos que conforman dicho texto, y por tanto, es un lenguaje poco eficiente para ser interpretado por las máquinas.

A continuación mostramos un ejemplo de código HTML y explicaremos sobre el que es lo que está provocando el problema del que hablamos, es decir, por qué la Web está desarrollada para humanos y no para máquinas.

Algunas etiquetas de HTML y su significado:

`<p>` → Todo texto que se encuentra entre estas etiquetas es un párrafo.

`<em>` → Lo etiquetado debe ser mostrado con un formato diferente para ser enfatizado.

`<big>` → Lo etiquetado debe ser mostrado en un tamaño grande.

`<b>` → Lo etiquetado debe ser mostrado en negrita.

Ahora haciendo uso de estas etiquetas escribimos un pequeño párrafo el cual nos podríamos encontrar en cualquier lugar navegando por la Web:

```
<p>Segovia es un lugar <em> relajante </em> provincia de <b> Castilla y León </b> en el cual tenemos un clima <big> frío </big> de unos 5 °C.</p>
```

Para una persona el conocimiento que contiene ese párrafo es obvio, si lo viésemos desde una web estaríamos observando un párrafo normal y corriente en el que algunas palabras cobrarían mayor importancia que otras, unas al tener un tamaño mayor y otras por estar en negrita. Esto demuestra que HTML es un lenguaje diseñado para presentar información al ojo humano, pero para un ordenador ese mismo párrafo tiene un significado bien distinto:

<p>!#\$%\$/\$) &/ (/&%\$#<em>!"#%#!%#\$\$% </em> \$%\$/\$) &/ (/& <b> !"##!"% </b>!"#%#&/\$/#& &/\$/#<big> #"%#\$"%#%#% </big> !"#%!\$#&/#\$/&% \$#"/.</p>

Esto es lo que sucede en la máquina al momento de procesar el documento, como podemos observar el ordenador comprende correctamente todas las etiquetas del texto, pero no el contenido en sí mismo, es debido a este que no es posible realizar búsquedas complejas en los buscadores actuales. Esta situación se sigue dando a día de hoy, aun y con las mejoras que se han incluido en los buscadores web, para comprobar que esto sigue ocurriendo vamos a realizar una consulta compleja a través de Google:

Introducimos en el buscador: "Lugar de descanso fuera de Madrid que sea frío y que tenga monumentos".

El resultado de realizar esta consulta lo podemos ver en la Ilustración 1:



Ilustración 1: Ejemplo de búsqueda en un navegador no semántico

Esta búsqueda ejemplifica que, la gran mayoría de la información está disponible a los humanos y no a los ordenadores, por lo cual consultas como esas tienen muy poca cabida en la Web actual. Eso sí, es posible crear algoritmos que puedan hacer este trabajo, pero serían inviables en términos de consumo de recursos, solo responderían a un grupo reducido de preguntas y no serían muy efectivos.



Para tratar de resolver estos problemas se han puesto en práctica diferentes soluciones, mostramos a continuación alguna de ellas:

- **Posicionamiento Web (SEO)** - El posicionamiento Web es el conjunto de tareas dedicadas a la mejora de la visibilidad de un sitio web en los resultados de los diferentes buscadores. Empezó a utilizarse en 1990 cuando los motores de búsqueda comenzaron a indexar y catalogar Internet. Al principio estas tareas simplemente se basaban en indexar y extraer información de las páginas web, como las palabras que contenían, dónde estaban localizadas y su relevancia dentro del documento, además de todos los vínculos. Esta información sobre las palabras era provista por los administradores web en forma de *metatags*, lo cual provocó que las pocas restricciones en la utilización de estas *metatags*, fueran utilizadas por los administradores web para obtener un buen posicionamiento incluso aunque las búsquedas no estuvieran ni siquiera relacionadas con el contenido de las páginas. En la historia del posicionamiento web siempre ha habido un conflicto de intereses entre el interés de los buscadores por ofrecer contenido relevante y el interés de los administradores por posicionarse, que hasta la fecha no ha sido realmente solucionado.
- **Microformatos** - Son una tecnología similar a los datos enlazados, el objetivo común de ambos es extender la Web actual mediante datos estructurados. Los microformatos definen una serie de formatos de datos que se encuentran embebidos dentro de los documentos HTML por medio de atributos de clase. El punto débil de los microformatos es que se restringe la representación de los datos a un pequeño conjunto de entidades y atributos de estas entidades. Por este motivo los microformatos no son la mejor manera de compartir datos en la Web.
- **APIs Web** - Las APIs Web representan otra solución para el acceso a datos estructurados en la Web actual. Muchas de las mayores fuentes de datos como Amazon, eBay, Yahoo! y Google proporcionan acceso a sus datos vía APIs Web haciendo uso del protocolo HTTP. El número de aplicaciones especializadas que combinan datos de diferentes fuentes ha crecido increíblemente ya que ofrece beneficios indudables a los programadores. Sin embargo, el problema principal de estas APIs es que sus datos pueden ser accedidos de maneras muy heterogéneas y los datos obtenidos pueden tener múltiples formatos dependiendo de la fuente de datos que se utilice, lo cual supone un gran esfuerzo a la hora de integrar cada nuevo conjunto de datos en la aplicación que se esté desarrollando.

Debido a los problemas que acabamos de ver, y que afectan a la Web tal y como la conocemos, es en este punto es donde entra en juego la Web Semántica.

## 2.2. La Web Semántica

La **Web Semántica** es una extensión de la World Wide Web, cuyo precursor fue Tim Berners-Lee, su principal idea fue la de añadir metadatos semánticos y ontológicos a la WWW, esta información adicional permite devolver resultados más precisos ante una búsqueda de información, además se

debe proporcionar de manera formal para que pueda ser evaluada automáticamente por máquinas de procesamiento. Dichas máquinas son agentes inteligentes, programas que residen en las computadoras, que buscan información sin operadores humanos.

Por lo tanto, la Web Semántica abandona la idea de buscar a través de la simple comparación de caracteres, para avanzar hacia la búsqueda de conceptos.

Si representamos ambas versiones de la web en forma de grafo, vemos que la diferencia fundamental entre la Web actual y la Web semántica, reside en que en el caso de la **Web actual** los nodos siempre estarían compuestos por páginas HTML, y las relaciones entre estos documentos estarían siempre basados en hipervínculos simples sin mayor clasificación. Esto supone que no existiría ninguna diferencia entre un blog personal o una página de reserva de hoteles. El caso de la **Web semántica** es diferente, ya que cada nodo tendrá un dato de un tipo especificado y este nodo será la mínima unidad de información. Y cada unión entre estos nodos representará el tipo de relación que existe entre ellos.

Por esto, se considera que la Web actual está orientada al documento y la Web semántica a los datos.

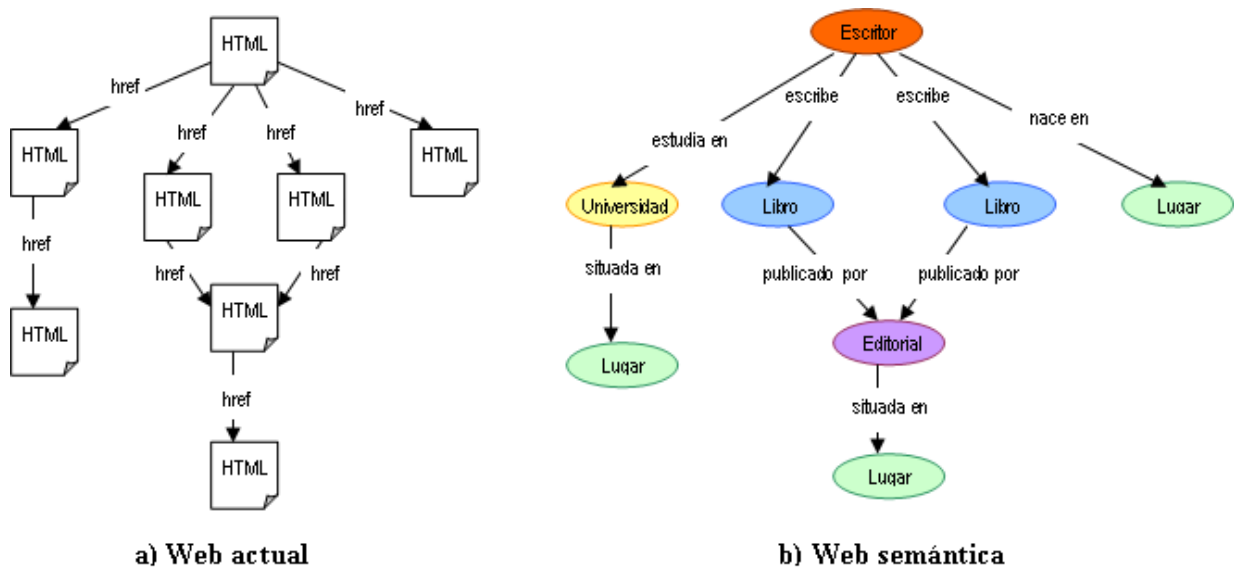


Ilustración 2: Diferencia entre la web actual y la web semántica

Concluyendo con esta introducción a la web semántica, uno de sus principales objetivos es el de poder construir aplicaciones más inteligentes que aprovechen la gran cantidad de datos que se exponen en la Web. El principal valor de dicho modelo es la filosofía **Linked Open Data**, una corriente promovida por el ya mencionado, inventor de la web y presidente del W3C, Tim Bernes Lee.

El objetivo de este modelo semántico es ampliar la web con una base de datos común mediante la publicación en la Web de bases de datos en el formato RDF y mediante el establecimiento de enlaces entre datos de diferentes fuentes.

## 2.3. Linked Data

Los datos enlazados, **Linked Data** o Web de Datos, es la forma que tiene la Web Semántica de vincular los distintos datos que están distribuidos en la Web, esto hace que los datos se referencien de igual forma a como lo hacen actualmente los enlaces en las páginas web. Esta vinculación añade el valor de que, tanto personas, como máquinas puedan explorar la web de datos pudiendo llegar a información relacionada que se hace referencia desde otros datos iniciales. En 2006 Tim Berners-Lee definió cuatro pilares básicos para la publicación de *Linked data*:

1. Utilizar **URIs** (Uniform Resource Identifier) que identifiquen los recursos de forma unívoca.
2. Seguir el protocolo **HTTP** para resolver la ubicación de los datos identificados mediante URIs.
3. Ofrecer información sobre los recursos utilizando **RDF**.
4. Incluir enlaces a otras **URIs**, facilitando el vínculo entre distintos datos distribuidos en la web.

Estos principios están definidos como reglas, pero en realidad son más bien recomendaciones o buenas prácticas para el desarrollo de la web semántica. Es posible publicar datos que cumplan sólo los tres primeros principios, pero el hecho de no aplicar el cuarto los convierte en menos visibles y en consecuencia, menos reutilizables.

## 2.4. RDF (Resource Description Framework)

Fue desarrollado por el W3C y actualmente RDF es el modelo estándar de intercambio de datos en la web semántica, se trata de un **framework** de descripción de recursos, creado inicialmente como lenguaje para añadir metadatos legibles en la Web, es decir, que se diseñó para conocimiento y no para datos.

El modelo lógico de RDF provee un método general y flexible para descomponer cualquier tipo de conocimiento en piezas más pequeñas, conocidas como **triples**, y siguiendo algunas reglas acerca del significado de dichas piezas. El principal objetivo es obtener lo que se conoce como un **grafo dirigido etiquetado**, en el que cada arista representa un hecho, o relación entre dos cosas. Un hecho representado de esta manera tiene tres partes: **Un sujeto, un predicado y un objeto**. El sujeto es lo que está al inicio de la arista y se le asocia con un nodo, el predicado es el tipo de arista, es decir la etiqueta, y une el nodo sujeto con el nodo objeto, que es lo que está al final de la arista. Un objeto de un triple puede funcionar como el sujeto de otro triple. Así mismo, RDF permite una forma de **reificación** (una declaración de una declaración), que significa que cualquier declaración de RDF puede ser usada como sujeto en un triple. RDF además no restringe la forma de serializar los triples, es por ello que soporta varios formatos posibles: RDF/XML, N3, Turtle, Json-LD, etc.

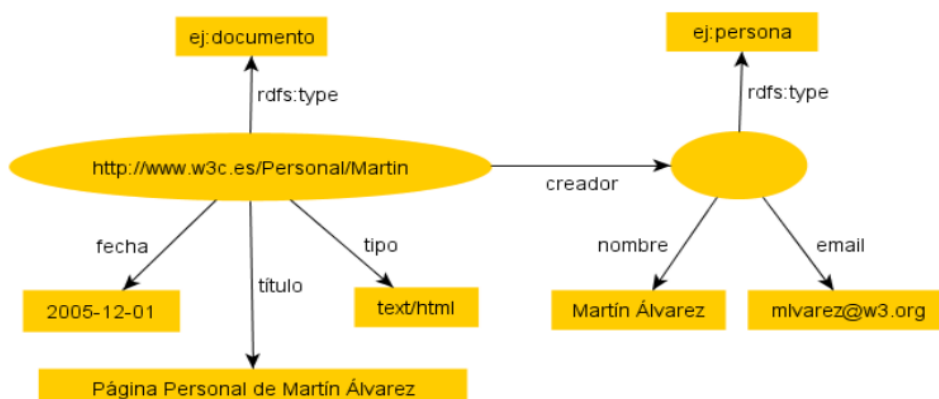


Ilustración 3: Ejemplo de Grafo RDF

Según el modelo de datos que propone RDF consiste en tres tipos de objetos:

- **Recursos:** Cualquier objeto de la Web identificable unívocamente mediante una URI, es decir, un identificador uniforme de recursos como una URL. **Un recurso** puede ser desde una parte de una página web, como por ejemplo, un elemento HTML dentro de un documento, hasta una colección de páginas, por lo que, en síntesis, un recurso es cualquier objeto de información.
- **Propiedades:** Aspectos específicos, atributos o relaciones utilizadas para describir recursos. Cada tipo de propiedad tiene sus valores específicos y define los valores permitidos, los tipos de recursos que puede describir y las relaciones que existen entre las distintas propiedades.
- **Descripciones:** Son el conjunto formado por un **recurso**, un nombre de **propiedad** y el **valor** de esa propiedad. Por lo tanto, una declaración está compuesta por **tres** partes individuales:
  - **Sujeto:** Recurso
  - **Predicado:** Propiedad
  - **Objeto:** Valor de la propiedad, que puede ser otro recurso, especificado por una URI, o un literal. Un literal es una cadena simple de caracteres definidos por XML, su contenido no es interpretado por RDF y en ocasiones contiene marcado XML adicional. Los literales se distinguen de los recursos gracias a que el modelo RDF no permite que los literales sean sujeto de una declaración.

Este conjunto de tres elementos, **sujeto-predicado-objeto**, se conoce como **terna** y es la unidad básica de información en RDF.

### 2.4.1. Ejemplo

El siguiente ejemplo se encuentra en la página oficial del World Wide Web Consortium, o W3C, en el podemos ver como se describe un recurso con diferentes tipos de declaraciones, las cuales tienen la siguiente forma:

El **sujeto** del recurso *http://www.w3.org/People/EM/contact#me*, que es una URI.

Los **objetos** son los siguientes:

- ◆ *Eric Miller* → con el **predicado** *cuyo nombre es*.
- ◆ *mailto:em@w3.org* → con el **predicado** *cuyo correo electrónico es*.
- ◆ *Dr.* → con el **predicado** *cuyo título es el de*.

Los predicados también tienen asociadas otras **URIs**:

- Cuyo nombre es: *http://www.w3.org/2000/10/swap/pim/contact#fullName*
- Cuyo correo electrónico es: *http://www.w3.org/2000/10/swap/pim/contact#mailbox*
- Cuyo título es: *http://www.w3.org/2000/10/swap/pim/contact#personalTitle*

Además, el sujeto tiene un **tipo** definido de RDF cuya URI es *http://www.w3.org/1999/02/22-rdf-syntax-ns#type*, que indica que es una persona *http://www.w3.org/2000/10/swap/pim/contact#Person*.

Por lo tanto, la descripción de este recurso puede expresarse con los siguientes triples:

- *http://www.w3.org/People/EM/contact#me*,  
*http://www.w3.org/2000/10/swap/pim/contact#fullName*,  
"Eric Miller"
- *http://www.w3.org/People/EM/contact#me*,  
*http://www.w3.org/2000/10/swap/pim/contact#mailbox*,  
*mailto:em@w3.org*
- *http://www.w3.org/People/EM/contact#me*,  
*http://www.w3.org/2000/10/swap/pim/contact#personalTitle*,  
"Dr."
- *http://www.w3.org/People/EM/contact#me*,  
*http://www.w3.org/1999/02/22-rdf-syntax-ns#type*,  
*http://www.w3.org/2000/10/swap/pim/contact#Person*

Y da lugar al siguiente grafo dirigido:

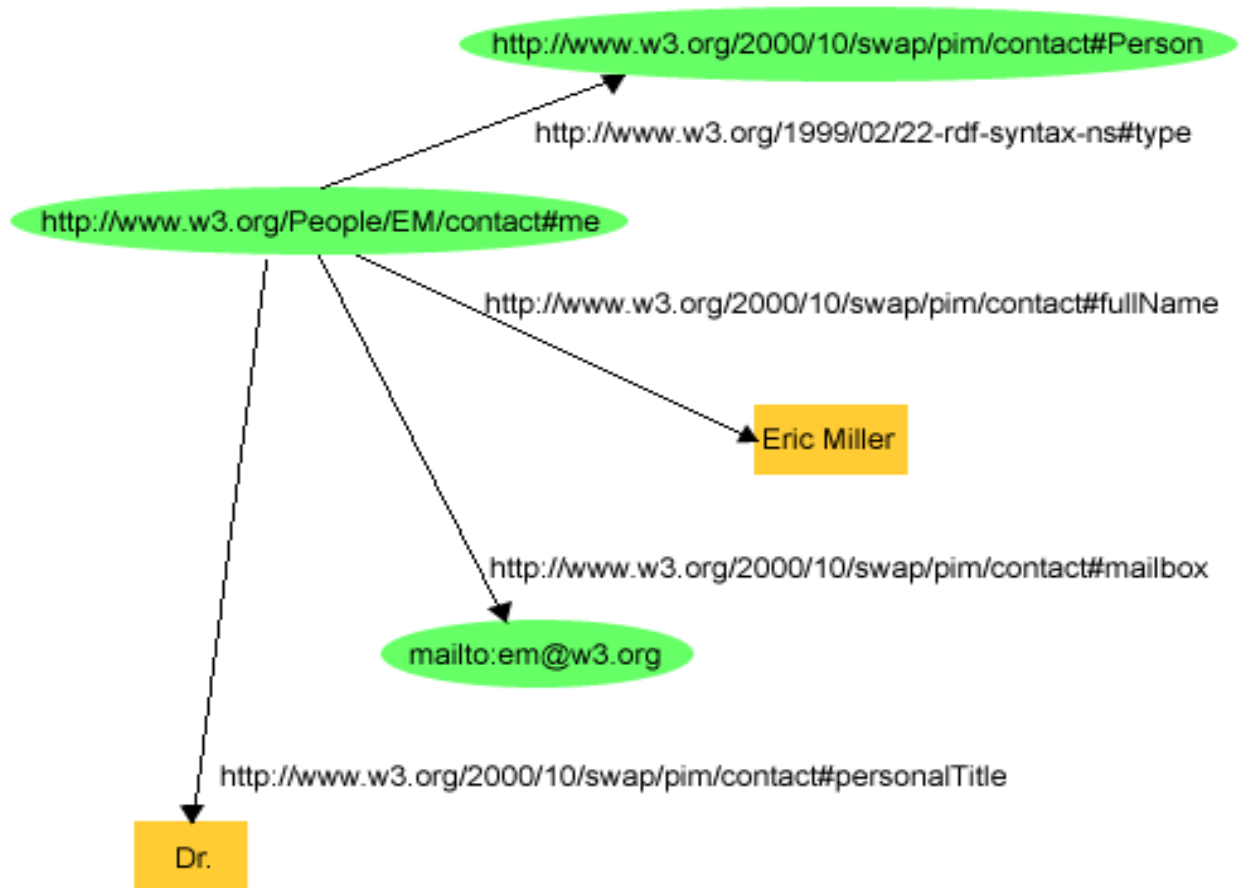


Ilustración 4: Ejemplo de grafo RDF que describe un recurso

El grafo de la Ilustración 4 se describiría de la siguiente forma, en el formato estándar **N-Triples**:

```
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/2000/10/swap/pim/contact#fullName>
"Eric Miller".
```

```
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/2000/10/swap/pim/contact#mailbox>
<mailto:em@w3.org>.
```

```
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/2000/10/swap/pim/contact#personalTitle>
"Dr.".
```

```
<http://www.w3.org/People/EM/contact#me>  
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  
<http://www.w3.org/2000/10/swap/pim/contact#Person>.
```

## 2.5. SPARQL

Proveniente del inglés SPARQL Protocol and RDF Query Language, es el lenguaje de consulta estandarizado para el formato RDF, concretamente, para grafos RDF. En 2004 paso a ser una recomendación del W3C. Este lenguaje de consulta esta basado en el emparejamiento de patrones, o *graph pattern matching*, es decir, que define las consultas en forma de patrones obligatorios u opcionales de grafo, junto con sus conjunciones y disyunciones, permitiendo también indicar los grafos sobre los que se va a realizar la consulta. Los resultados de las consultas SPARQL pueden ser conjuntos de resultados o grafos RDF. SPARQL a primera vista tiene una apariencia muy similar a SQL, pero tienen una gran diferencia, y es que mientras que SQL trabaja con datos que se encuentran en tablas, SPARQL lo hace con datos que están implementados en grafos.

Además de las formas básicas de consulta como SELECT, ASK, CONSTRUCT, DESCRIBE, este lenguaje también soporta otro tipo basado en patrones de triples (**triple patterns**) y permite que cualquiera de sus componentes sea una **variable**, dichas variables **recuperan** los **resultados** que satisfacen la consulta. El **triple pattern** es la unidad básica de consulta en SPARQL y su configuración es muy similar a la de **un triple RDF**, en la consulta debemos especificar un sujeto, un predicado y un objeto (S P O), en el caso de que desconozcamos uno de los términos, introduciremos una interrogante (?) para indicarlo. Como resultado de la búsqueda obtendremos los valores que coincidan con el grafo. A continuación mostramos los **ocho tipos** de consulta que podemos realizar con SPARQL utilizando los *triple patterns*:

- **S P O** – Indicamos los tres parámetros que queremos obtener. Estamos preguntando por un determinado triple.
- **S P ?** – Desconocemos el parámetro objeto, recuperaremos todos los valores ?O asociados al sujeto S mediante la propiedad P.
- **S ? O** – Desconocemos el parámetro predicado, recuperaremos todas las propiedades ?P que relacionan un sujeto S y un valor O.
- **? P O** – Desconocemos el parámetro sujeto, recuperaremos todos los sujetos ?S que tienen un valor O para una propiedad P.
- **S ? ?** – Recuperaremos todo lo que se dice sobre el sujeto S.
- **? P ?** – Recuperamos todos los pares sujeto-valor para una propiedad P dada.
- **? ? O** – Recuperamos todos los pares sujeto-propiedad cuyo valor es el objeto O.
- **? ? ?** – Recuperamos todos los triples de la colección.

## 2.6. HDT (Header, Dictionary, Triples)

El crecimiento de los Datos Enlazados trae consigo la elaboración de conjuntos de datos RDF cada vez de mayor tamaño. Actualmente existe la necesidad de intercambiar esta información, por lo que se hace necesario establecer unos criterios de compresión. En este caso es algo más complejo, ya que estos datos se serializan como texto de forma muy detallada. Esto implica que diferentes dispositivos con menos recursos, como por ejemplo los móviles, requieran de un formato más eficiente, por lo que se hace indispensable la creación de un nuevo formato para la publicación, intercambio y consumo de RDF.

Como respuesta a este problema nace **HDT**, un formato binario para la serialización de RDF en espacio comprimido. HDT es una estructura de datos compacta e indexada para **RDF**, permite ahorrar espacio de almacenamiento, y a su vez también nos ahorra mucho tiempo en las operaciones de búsqueda y navegación, puesto que no tiene la necesidad de descomprimir los datos para navegar por la información. Esto hace de HDT un formato ideal para almacenar y compartir conjuntos de datos RDF en la Web. Diferentes estudios recogen que HDT es capaz de reducir el espacio de almacenamiento hasta 15 veces en comparación con formatos similares como **NTriples** o **RDF/XML**.

Un conjunto de datos en formato HDT está formado por tres secciones lógicas, en la Ilustración 5 podemos ver dichas secciones, las cuales se dividen en: cabecera, diccionario y ternas. Estas últimas han sido especialmente diseñadas para trabajar con las particularidades del formato RDF.

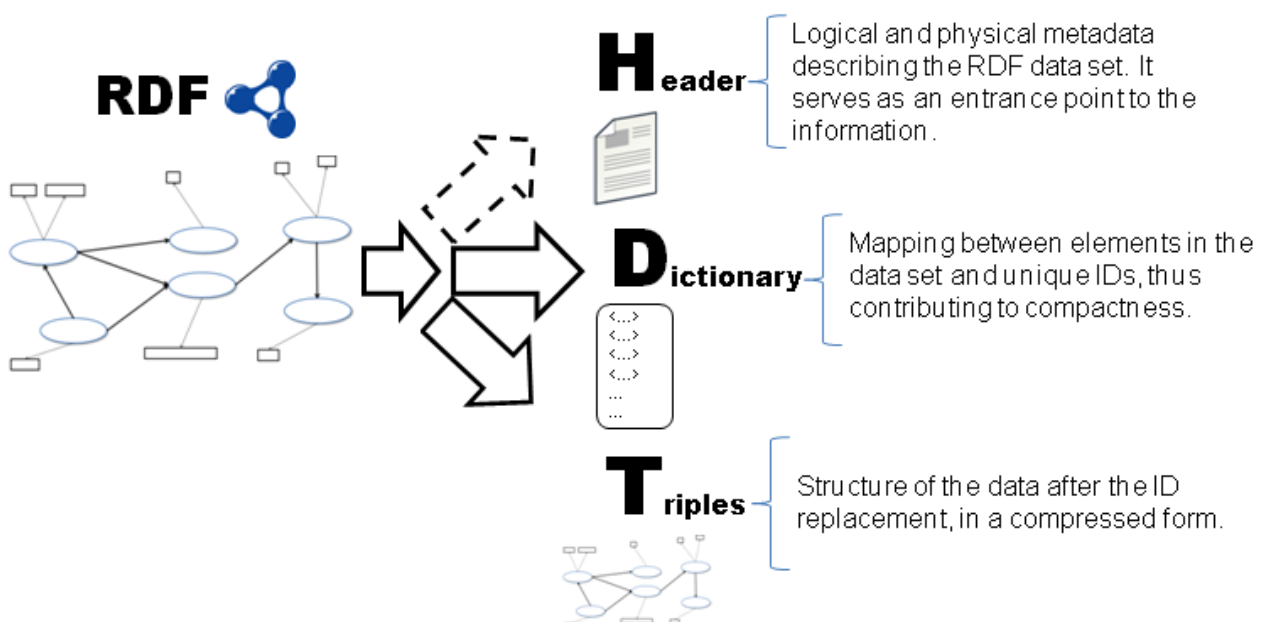


Ilustración 5: HDT: Cabecera, Diccionario y Ternas



**Cabecera (Header):** La cabecera contiene diferente información en forma de metadatos que describen el conjunto de datos RDF. Sirve como un punto de entrada para el consumidor, quien, de esta forma, puede obtener una aproximación de las propiedades clave del contenido antes incluso de recuperar el conjunto de datos completo.

**Diccionario (Dictionary):** El diccionario proporciona un vocabulario de todos los términos que se utilizan en el conjunto de datos RDF. En este catálogo cada término recibe un identificador único (ID), lo cual permite representar una terna con tres identificadores. Cada uno de estos identificadores hace referencia dentro del diccionario a su respectivo término sujeto/predicado/objeto. Este aspecto es clave a la hora de lograr una buena compresión, ya que evita que términos largos se repitan una y otra vez. Además, las cadenas de caracteres similares se guardan de forma consecutiva dentro del diccionario, circunstancia que se aprovecha para incrementar aún más la compresión.

**Ternas (Triples):** Las ternas RDF pueden verse ahora como tuplas de tres identificadores, esto hace que se “mapee” un grafo de relaciones entre los términos del conjunto de datos, tal y como se muestra en la Ilustración 6, el cual comprende las propiedades básicas de un grafo RDF, de esta forma se consiguen procedimientos más eficientes de representar esa información, tanto para reducir su tamaño, como para ofrecer mejores operaciones de búsqueda.

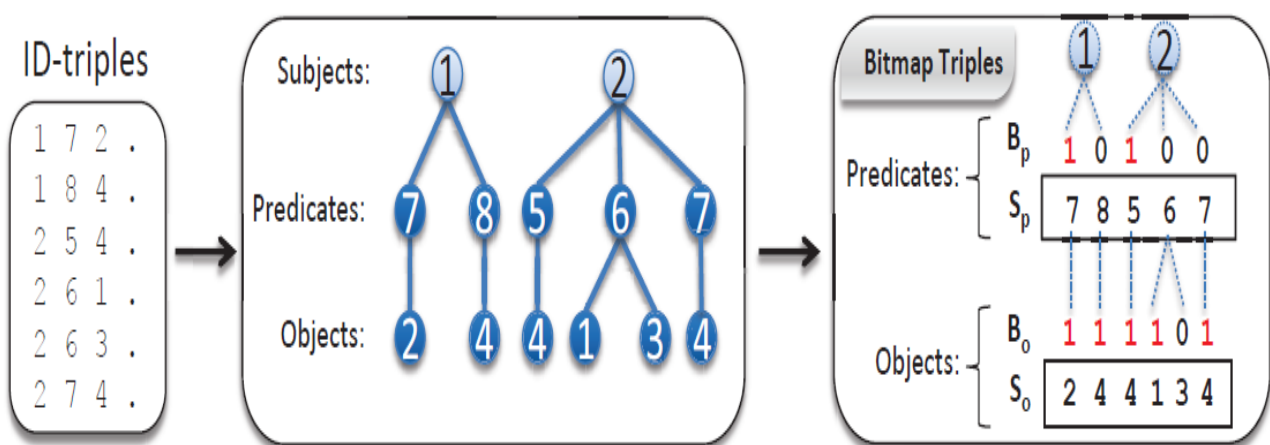


Ilustración 6: Ternas de HDT

Ahora que hemos visto que es HDT y para que sirve, destacamos algunas de sus ventajas:

- **El tamaño de los archivos es menor** que en otros formatos de serialización RDF. Esto supone un menor coste de ancho de banda en las transmisiones, lo cual se traduce en tiempos de descarga menores.
- **El archivo de HDT está indexado.** Utilizando este formato podemos realizar búsquedas con mayor rapidez, sin necesidad de realizar procesos previos de análisis e indexado.

## *Capítulo 2. La Web de Datos*

- **Alto rendimiento de consulta.** Normalmente, el cuello de botella que se produce en las bases de datos tiene relación con el acceso a disco. HDT solventa este problema utilizando unas técnicas de compresión internas que permiten que la mayor parte de los datos, o incluso de todo el conjunto de datos, se pueda mantener en la memoria principal, la cuál es varios ordenes de magnitud más rápida que los discos.
- **Altamente concurrentes.** HDT únicamente permite operaciones de **lectura**, por lo que puede manejar varias consultas simultáneamente haciendo uso de varios hilos de ejecución.
- **El formato es abierto** y está en proceso de estandarización por el W3C.
- **Las bibliotecas son de código abierto (LGPL).** Las librerías pueden ser adaptadas según el tipo de operaciones que se deseen realizar, para solucionar los problemas que puedan surgir del uso de las mismas, se apoyan en una comunidad en la que se pueden añadir mensajes sobre problemas detectados y aportar soluciones a los mismos.

## **Capítulo 3. El Proyecto**



### 3.1. Introducción

En este proyecto desarrollaremos dos aplicaciones dirigidas a la consulta de información sobre conjuntos de datos que se encuentren en el formato comprimido de RDF, es decir, en formato HDT. El conjunto de datos que utilizaremos en el proyecto será LinkedMDB, una base de datos sobre cine. Ambas aplicaciones serán desarrolladas para ejecutarse en un dispositivos móviles, por lo que el mencionado HDT nos ayudará a almacenar grandes cantidades de información semántica en el dispositivo, como también nos facilitará su recuperación y consulta en cualquier momento.

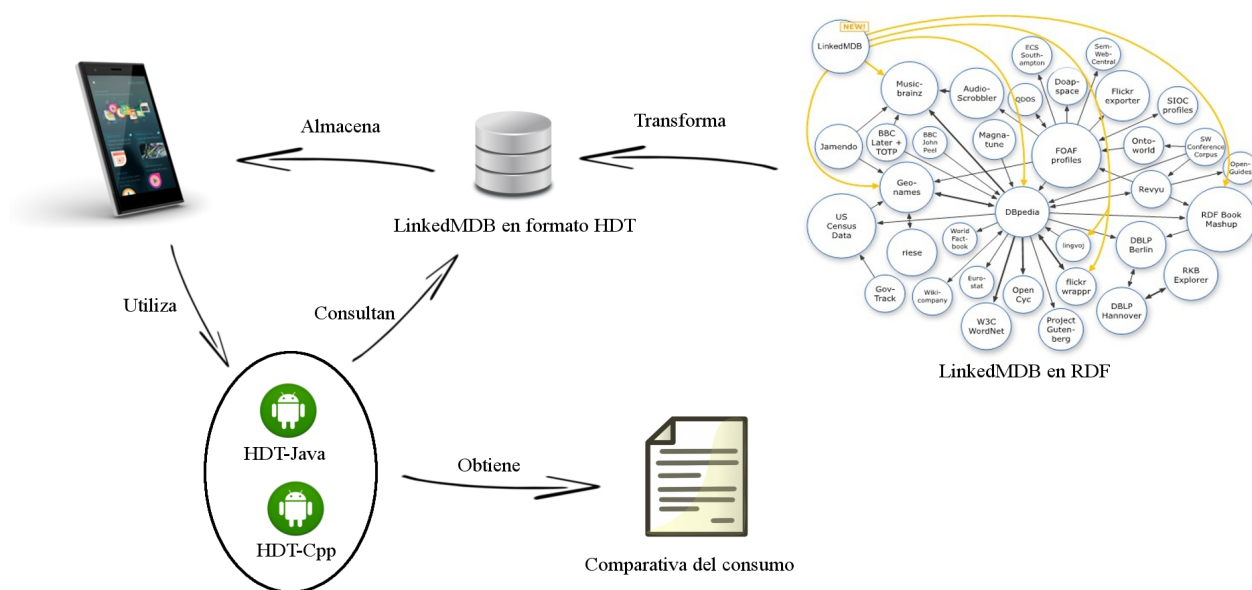


Ilustración 7: Ejemplo sobre el funcionamiento general de ambas aplicaciones

La Ilustración 7 muestra cómo un terminal móvil consulta, a través de las aplicaciones que vamos a desarrollar en este proyecto, al conjunto de datos de LinkedMDB que previamente tendremos que haber transformado al formato HDT. Una vez hecho esto, se hace necesario mover el conjunto de datos al dispositivo móvil, esto tendremos que hacerlo así porque la descarga de un *dataset* de LinkedMDB en formato HDT sería muy costosa, dado que el fichero puede llegar a ocupar más de 50 MB.

Como se puede ver también en la ilustración, no solo nos centraremos en el uso práctico de las aplicaciones sino también en su estudio, ya que mediante una serie de test estableceremos una comparativa sobre el consumo de batería que se produce en cada una.

### 3.2. LinkedMDB

**Linked Movie Data Base (LMDB)**, es la principal fuente de datos que vamos a utilizar en este proyecto. El proyecto LMDB, creado por Oktie Hassanzadeh y Mariano P. Consens, es el primer

proyecto de datos semánticos de libre acceso relacionados con el cine, y ganó el primer premio del LOD Triplication Challenge. Los enlaces entre datos han sido creados mediante la herramienta **ODDLinker** que hace uso de medio millón de atributos sobre alrededor de cuarenta mil películas y otras entidades relacionadas con las películas. En la Ilustración 8 pueden verse los enlaces de LMDb a otros conjuntos de datos pertenecientes al **Linking Open Data Project** entre los que se encuentran:

- Geo-names → Base de datos geográfica gratuita.
- DBpedia → Conocida como la Wikipedia de la Web Semántica.
- MusicBrainz → Base de datos musical de contenido abierto.
- FlickrWrappr → Extiende las capacidades de DBpedia añadiendo enlaces a fotos de flickr.
- Lingvoj → Permite definir el lenguaje de un recurso.
- RDF Book Mashup → Contiene información sobre libros, autores y artículos.

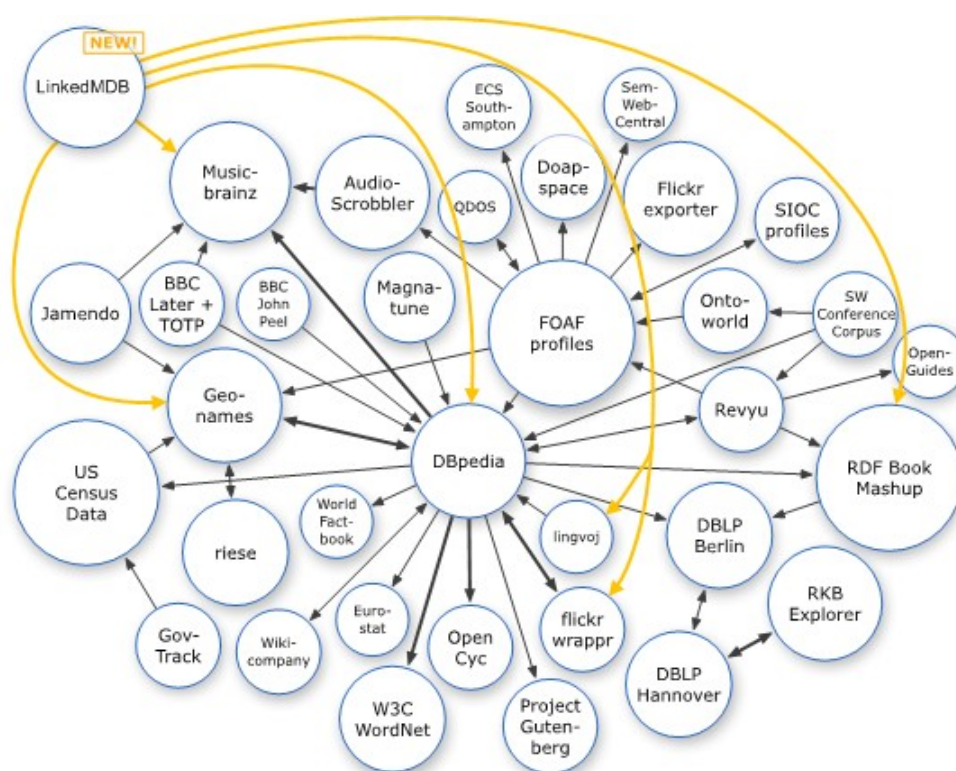


Ilustración 8: Grafo de enlaces en Linked Movie DataBase

Además también posee enlaces a importantes páginas web relacionadas con el cine tales como:

- IMDB (Internet Movie Data Base) - <http://www.imdb.com/>

- Rotten Tomatoes - <http://www.rottentomatoes.com/>
- FreeBase - <https://www.freebase.com/>

En LMDB el acceso a los datos se puede realizar de varias maneras, siendo la elegida para la realización de este proyecto la consulta a un conjunto de datos formateados mediante HDT.

Es importante tener en cuenta la cantidad de enlaces existentes de la fuente de datos que vamos a utilizar, esto nos va a servir para determinar la cantidad de resultados que vamos a obtener al realizar búsquedas de películas. En las tablas 1 y 2 se muestran las estadísticas totales de enlaces contenidas en el *dataset* del año 2010 de LMDB y las estadísticas de enlaces en función de la propiedad:

Número de triples publicados por LMDB	6.148.121
Número de enlaces a otras fuentes de datos de LOD	162.199
Número de referencias a sitios web de películas	541.810
Número de entidades en LMDB	503.242

Tabla 1: Estadísticas de Linked Movie DataBase

Destino	Propiedad	Número de enlaces
DBPedia	owl:sameAs	30 354
YAGO	owl:sameAs	30 354
flickrTM wrappr	dbpedia:hasPhotoCollection	30 354
RDF Book Mashup (Books)	movie:relatedBook	700
RDF Book Mashup (Authors)	rdfs:SeeAlso	12 990
MusicBrainz	owl:sameAs	2 207
GeoNames	foaf:based_near	27 272
GeoNames	owl:sameAs	272
lingvoj	movie:language	28 253
IMDb, Rotten Tomatoes, Freebase.com	foaf:page	541 810

Tabla 2: Número de enlaces por propiedad en Linked Movie DataBase

Ahora que hemos visto tanto los datos estadísticos como el número de enlaces que contiene LMDB, hemos creído oportuno no deshacernos de ningún tipo de enlace, puesto que todos ellos, en mayor o menor medida ofrecen información. Para que quede más claro cual es el mapa de definición de LMDB, a continuación vamos a ofrecer un listado con las propiedades que describen una película con sus respectivos identificadores de recurso (URIs). Esto lo conseguiremos gracias a una consulta a la aplicación HDT-it! que veremos posteriormente:

- Título <<http://purl.org/dc/terms/title>>
- Fecha <<http://purl.org/dc/terms/date>>
- Duración <<http://data.linkedmdb.org/resource/movie/runtime>>
- Idioma <<http://data.linkedmdb.org/resource/movie/language>>
- Actor <<http://data.linkedmdb.org/resource/movie/actor>>
- Director <<http://data.linkedmdb.org/resource/movie/director>>
- Productor <<http://data.linkedmdb.org/resource/movie/producer>>
- Escritor <<http://data.linkedmdb.org/resource/movie/writer>>



- Editor <<http://data.linkedmdb.org/resource/movie/editor>>
- Genero <<http://data.linkedmdb.org/resource/movie/genre>>
- Localizaciones del rodaje <[http://data.linkedmdb.org/resource/movie/featured\\_film\\_location](http://data.linkedmdb.org/resource/movie/featured_film_location)>
- Autores de la banda sonora <[http://data.linkedmdb.org/resource/movie/music\\_contributor](http://data.linkedmdb.org/resource/movie/music_contributor)>
- Distribuidora <[http://data.linkedmdb.org/resource/movie/film\\_distributor](http://data.linkedmdb.org/resource/movie/film_distributor)>
- Enlace a sitios web relacionados <<http://xmlns.com/foaf/0.1/page>>
- Edad recomendada <<http://data.linkedmdb.org/resource/movie/rating>>
- Precuela <<http://data.linkedmdb.org/resource/movie/prequel>>
- Secuela <<http://data.linkedmdb.org/resource/movie/sequel>>
- Performance <<http://data.linkedmdb.org/resource/movie/performance>>
- Identificador <<http://data.linkedmdb.org/resource/movie/filmid>>
- Esquema del RDF <<http://www.w3.org/2000/01/rdf-schema#label>>
- Tipo de RDF <<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>>

A continuación se muestra un grafo con la estructura de la consulta que se debe realizar a LMDB en el que se incluyen las propiedades que se utilizan para el enlazado con otras fuentes de datos:

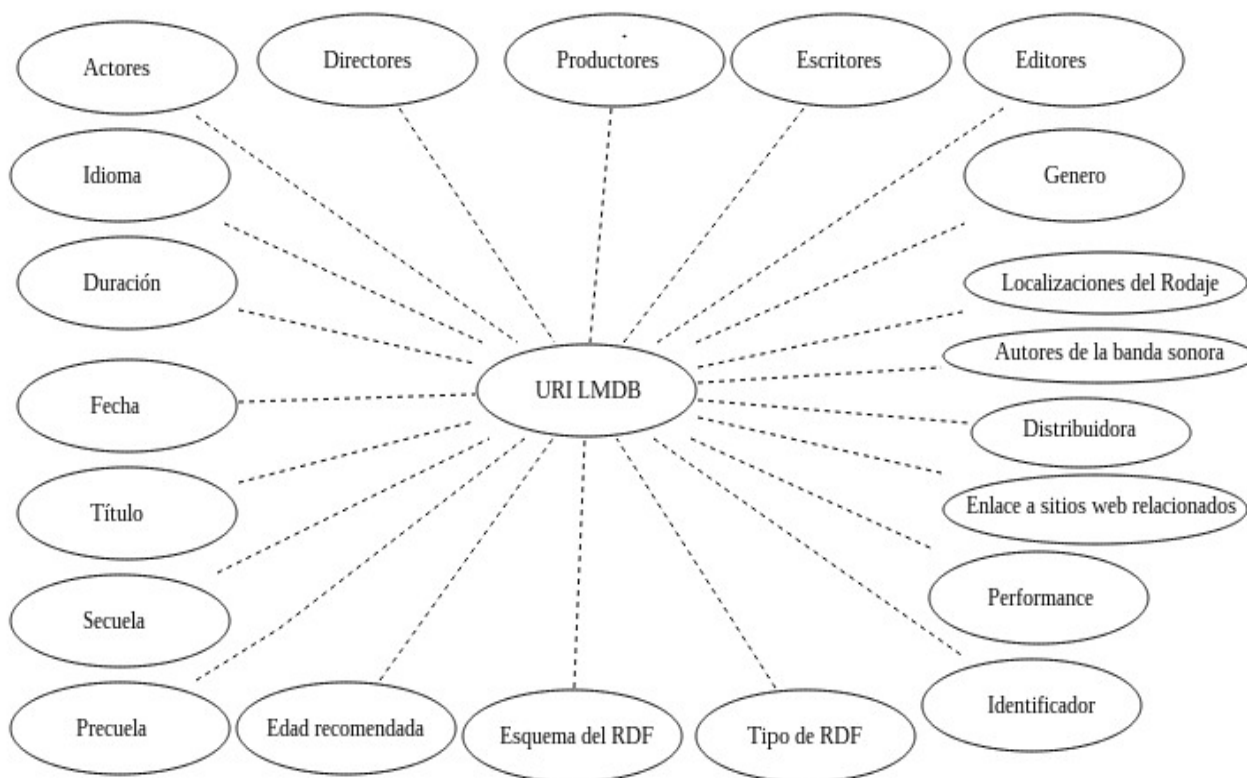


Ilustración 9: Estructura de una consulta a LinkedMDB

### **3.3. RDF/HDT**

El proyecto RDF/HDT, desarrollado por el equipo de investigación formado por Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutierrez, , Mario Arias Gallego y Axel Polleres, nace en el año 2009, a la par que la web semántica comenzó su auge. Su principal objetivo es el de ofrecer las utilidades necesarias para integrar la tecnología RDF/HDT en el desarrollo de diferentes aplicaciones y proyectos dentro del ámbito de la web semántica. En la actualidad es el principal valedor del formato HDT, y mantiene implementaciones de RDF/HDT, en C++ y Java, las cuales pueden ser reutilizadas bajo licencia LGPL.

De este proyecto obtendremos las herramientas que nos van a permitir implementar gran parte de nuestro proyecto actual, como acabamos de ver, si sumamos a las ventajas vistas en el apartado 2.6, el que las licencias que rodean a este proyecto sean libres, conseguimos que el desarrollo de nuestro proyecto, en cuanto a las herramientas de software se refiere, no nos suponga ningún coste.

### **3.4. El proyecto: HDT-Java y HDT-Cpp**

Lo que hace especial a este proyecto no solo es el formato HDT, si no el posterior estudio que se va a realizar sobre el consumo de batería que presenten las dos aplicaciones. Aunque alguno de los proyectos anteriores ya utilice HDT como formato, en ninguno de ellos se ha estudiado el efecto de dicho formato sobre el consumo energético que provoca su consulta en el dispositivo móvil.

Poniendo un ejemplo práctico de nuestro caso, el fichero LinkedMDB que guardaremos como HDT en el dispositivo móvil y que hemos generado, ocupa 52,4 megabytes, mientras que el dataset de esta misma fuente de datos en formato N-Triples ocupa 891,6 megabytes. Es decir, el fichero con formato HDT es 17 veces menor que el fichero con formato RDF. Esto es una ventaja a la hora de almacenarlo localmente en un dispositivo móvil, y constituye una de las principales razones de utilizar HDT en esta aplicación. La aplicación también incluirá diferentes tipos de consulta que el usuario podrá realizar sobre el conjunto de datos LinkedMDB

Como en la actualidad no se ha estudiado el rendimiento en cuanto al consumo energético que se produce al realizar consultas a un conjunto de datos en formato HDT, hemos creído oportuno aprovechar el desarrollo de este proyecto para establecer diferentes test y analizar sus resultados. Otro de los aspectos que hemos considerado es la velocidad, como se ha demostrado en numerosas pruebas el lenguaje de programación C++ es el más veloz en cuanto a su ejecución, este proyecto también servirá para ver si gracias a su gran optimización es capaz de consumir menos batería en un dispositivo Android, respecto al más que conocido lenguaje de programación Java.

### **3.5. Estado del arte**

Normalmente en este apartado se establece una comparativa entre las diferentes aplicaciones que se encuentran en el mercado y que trabajan en el mismo ámbito que la aplicación que estamos a punto de desarrollar. Este estudio previo permite, que tanto los desarrolladores, como los promotores del proyecto, puedan observar las cualidades y los defectos de dichas aplicaciones, y de esta forma actúen en consecuencia y añadan valores diferenciadores a su aplicación. Esta sencilla técnica los permite desmarcarse y por lo tanto, tener éxito con la implantación de su nueva aplicación.

En este proyecto, uno de los primeros problemas con los que nos topamos es esto mismo que acabamos de describir, puesto que, al no estar implantada ninguna aplicación que abarque los mismos temas que este proyecto, no podemos establecer una comparativa, y por tanto, dependemos únicamente de nuestro criterio para desarrollarlo.

A continuación mostraremos dos aplicaciones móviles que aprovechan las ventajas de HDT y en tercer lugar describiremos la propia herramienta desarrollada por el grupo RDF/HDT para realizar consultas a cualquier tipo de datos en formato HDT.

#### **3.5.1. Linked data from your pocket**

Es un framework que trata de proporcionar soporte a aquellas aplicaciones que deseen entregar su información en formato RDF. Permite que dichas aplicaciones exploten todas las capacidades del formato sin necesidad de conocer sus esquemas de antemano, es decir, que ofrece un sistema de explotación del formato RDF, uniforme. Esto hace que se puedan obtener cualquier tipo de datos de la web semántica de forma estandarizada, lo cual ayuda mucho al consumo de dicha información desde la plataforma Android.

Este framework ha sido desarrollado con las herramientas que proporciona el proyecto SWIP (Semantic web in the pocket), en la Ilustración 10 podemos ver la arquitectura global de dicho proyecto y todos los componentes que la forman:

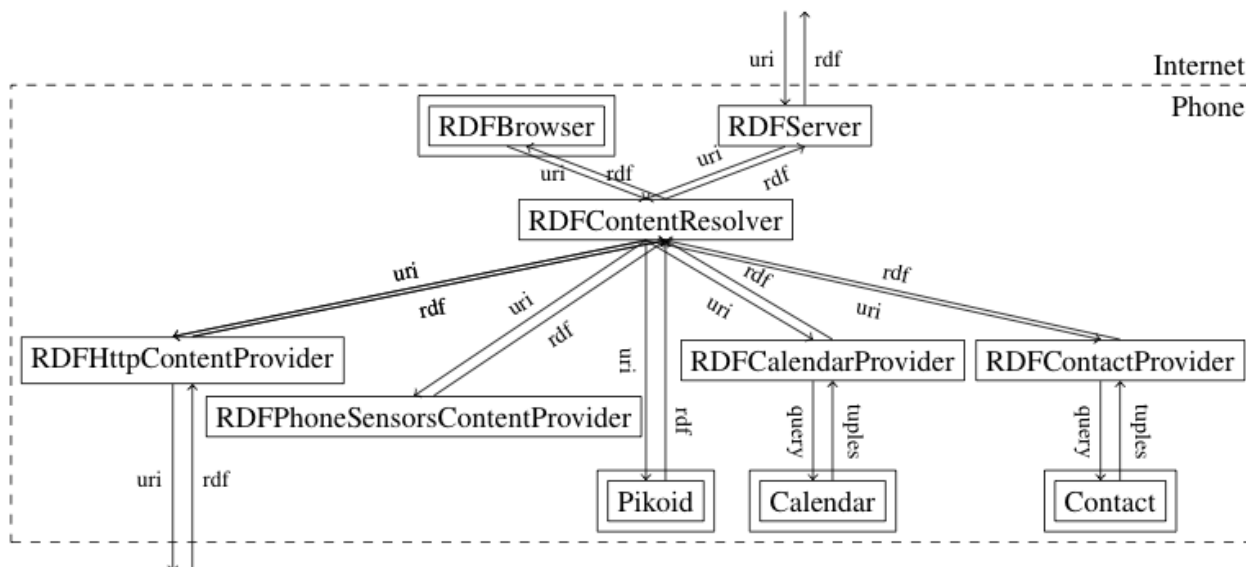


Ilustración 10: Arquitectura del proyecto *Linked data from your pocket*

Como vemos en la Ilustración los dos principales servicios que ofrece son: un servicio para compartir en la Web diferentes contenidos RDF, llamado RDFServer, en el que es necesario que el dispositivo móvil este conectado a Internet, y un buscador de información, RDFBrowser, que actúa como un cliente de la Web Semántica. Este ultimo servicio es muy interesante puesto que proporcionándole una URI, el buscador hace una petición HTTP que permite recibir toda la información que rodea al recurso, además si dicha información contiene otras URIs, se puede continuar con la navegación a través de ellas. En la Ilustración 11 podemos ver el aspecto que tiene la aplicación:

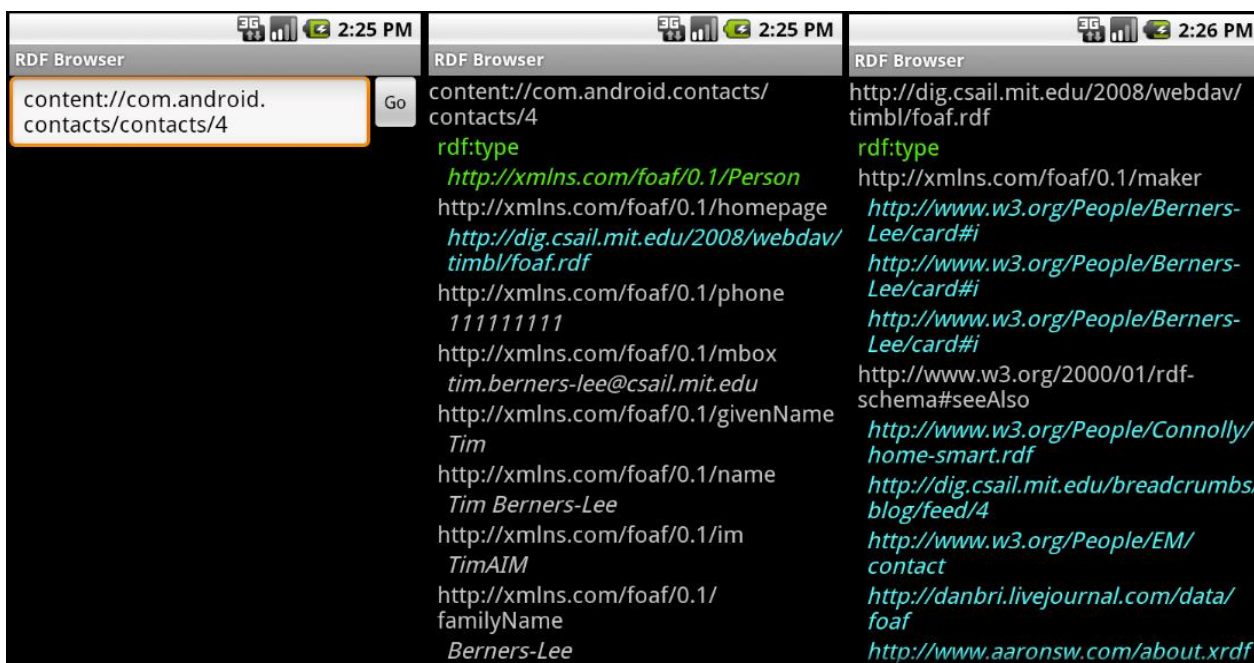


Ilustración 11: Diferentes pantallas de la aplicación RDF Browser

### 3.5.2. HdTourist

“HdTourist: Una Aplicación Ligera para el Consumo de Información Semántica en Android”, es un proyecto realizado por Elena Hervalejo Sevillano, en el que la funcionalidad básica que se pretendió conseguir fue obtener información de DBpedia, la Wikipedia de la Web Semántica. En concreto la información que se extrae de DBpedia tiene que ver con todo lo que rodea al urbanismo de las ciudades, esto tiene como principal objetivo ayudar a los turistas que visitan una ciudad extranjera. En la Ilustración 12 podemos ver algunas capturas de pantalla de la aplicación:

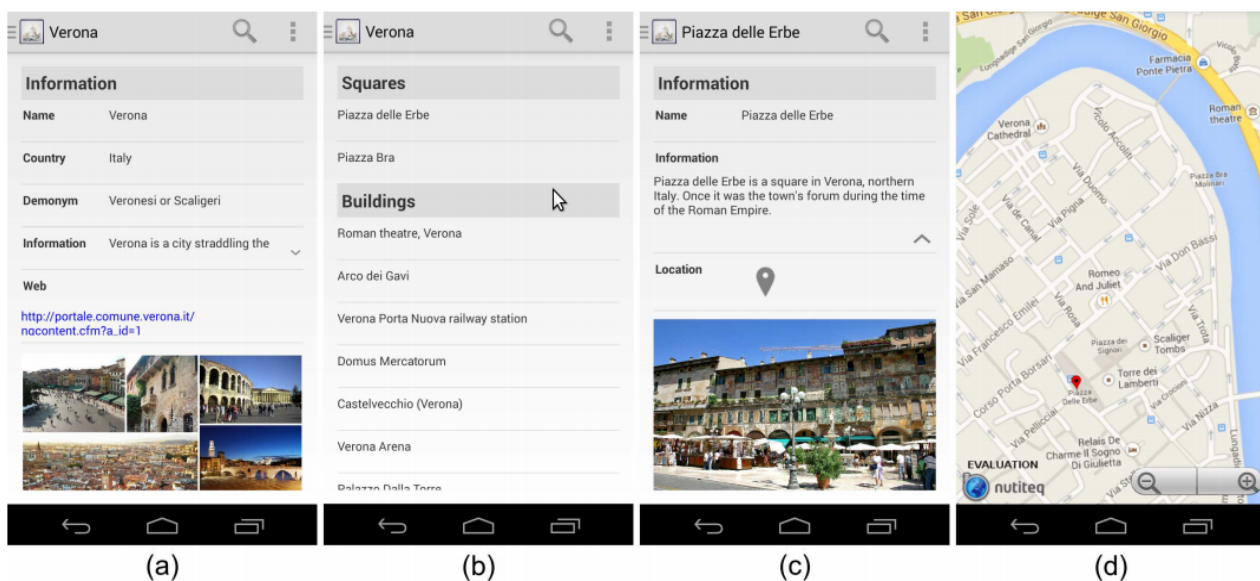


Ilustración 12: Capturas de la aplicación HdTourist

Para la recuperación de dicha información, la aplicación solicita los datos mediante una consulta a un *Middleware* sobre toda la información relativa a la ciudad que hayamos escogido. Dicho *Middleware* captará la información sobre la ciudad elegida contenidos de la Web de Datos relacionados, DBpedia, y le devolverá un fichero HDT con toda la información. Una vez descargado el fichero, el dispositivo móvil contendrá la información de manera local, es decir que podrá consumirla incluso de forma offline. Aunque se trate de una aplicación muy simple, nos muestra como los datos semánticos pueden ser almacenados y consultados en un dispositivo móvil, donde los recursos son muy limitados.

### 3.5.3. HDT-it!

HDT-It! Es una herramienta de software libre desarrollada por el propio proyecto RDF/HDT. Implementada con el lenguaje de programación C++, se apoya en la librería Raptor para proporcionar un conjunto de analizadores y serializadores que operan entre el formato HDT y las principales sintaxis de RDF.

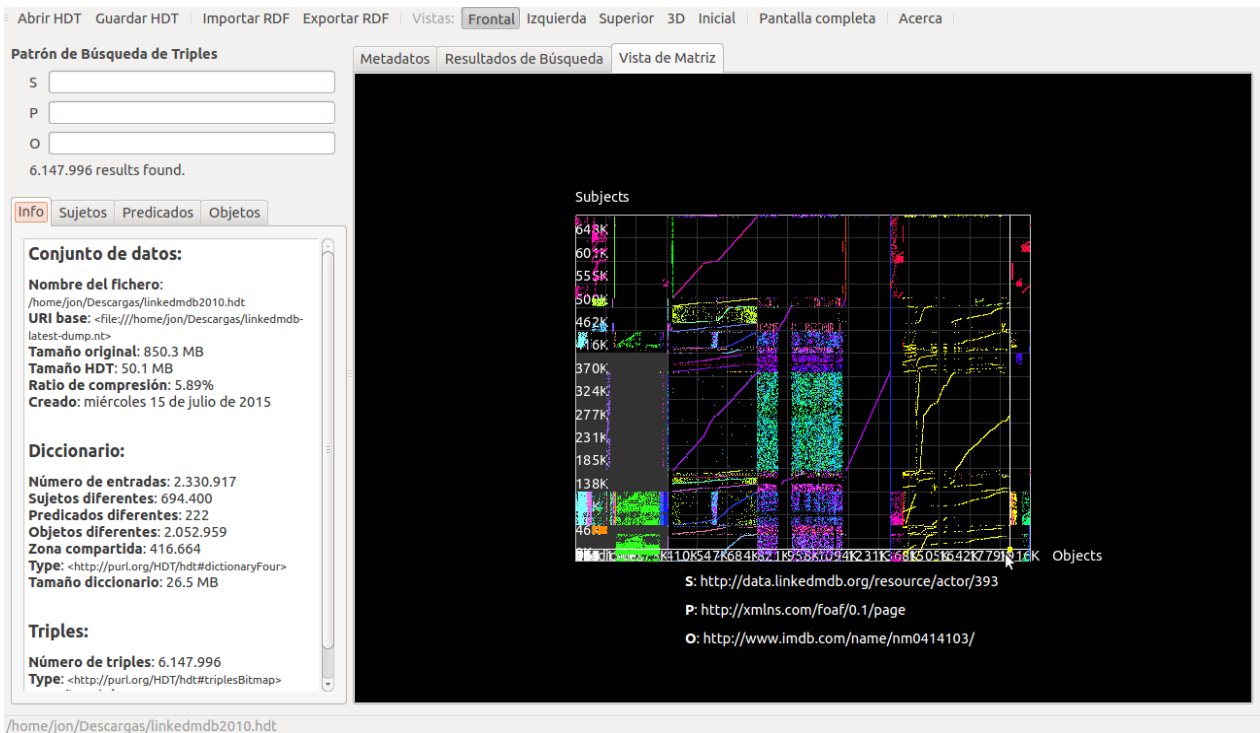


Ilustración 13: Pantalla inicial de HDT-it!

La Ilustración 13 muestra una imagen del programa una vez ha cargado un fichero, en la pestaña “Info” de la izquierda podemos ver toda la información que se puede extraer de la cabecera del conjunto de datos que acabamos de cargar. En la parte superior de este menú vemos como hay tres *input text* donde el usuario puede introducir diferentes parámetros para buscar cualquier patrón de triples, los resultados de la búsqueda se pueden mostrarán tanto en texto plano como en una vista en forma de matriz. Adicionalmente los resultados obtenidos pueden ser exportados en formato RDF o HDT, haciendo que la generación de subconjuntos de datos sea tremendamente sencilla.

HDT-it! muestra muchas de las ventajas que tiene utilizar el formato HDT, tanto para la consulta como para el almacenamiento, y ofrece al usuario una interfaz con la que poder interactuar con los diferentes conjuntos de datos que existen. Aprender a usar esta herramienta ha sido muy sencillo, y de ella hemos podido obtener diferentes conocimientos que hemos aplicado directamente en nuestro proyecto como se verá posteriormente.

## **Capítulo 4. Análisis**





## 4.2. Análisis

En esta etapa del proyecto de lo que se trata es de concretar y entender que es lo que tenemos que desarrollar y como lo vamos a hacer, así como el dominio en el que operará la aplicación que desarrollemos. En primer lugar, con lo que nos vamos a encontrar en esta fase, será con una descripción del problema que tenemos que resolver. Por lo tanto, uno de los objetivos de este apartado será identificar los elementos esenciales de la aplicación, algo que no será nada sencillo dado que debemos verlos tal y como lo haría un usuario final del programa. El resultado final deberá ser un modelo formal en el que se muestre brevemente los diferentes elementos de dominio, cosa que nos servirá como punto de inicio en la etapa de diseño.

### 4.2.1. Objetivos

Los principales objetivos de este proyecto son:

- Realizar un estudio comparativo, entre las aplicaciones que desarrollaremos, sobre el consumo de batería que se produce en el dispositivo móvil al ejecutar un determinado número de consultas sobre un conjunto de datos en formato HDT.
- Desarrollar dos aplicaciones móviles que permitan realizar diferentes tipos de consultas a un conjunto de datos en formato HDT, donde cada una de las aplicaciones se desarrollará de una forma diferente. Esta distinción será el lenguaje de programación que utilizarán para ser implementadas.
- Permitir consultar información cinematográfica al conjunto de datos LinkedMDB en formato HDT.

### 4.2.2. Restricciones

Uno de los requisitos del proyecto es que ambas aplicaciones se ejecuten en dispositivos móviles, por lo que tendremos que tener en cuenta tanto la memoria principal (RAM) del dispositivo, como la memoria de almacenamiento que disponga.

Por el contrario, no necesitaremos que el dispositivo móvil se encuentre conectado a la red, puesto que el *dataset* de LinkedMDB en formato HDT se proporcionará junto con el fichero de instalación de las aplicaciones, lo que hace podamos utilizar la aplicación tras su instalación.

### 4.2.3. Especificación de requisitos funcionales

<b>RF-01</b>	Cargar un fichero en formato HDT
<b>Descripción</b>	Cuando la aplicación se inicie deberá cargar en memoria principal el fichero HDT sobre el que se realizarán las consultas.
<b>Importancia</b>	Muy alta

Tabla 3 RF-01: Cargar un fichero en formato HDT

<b>RF-02</b>	Seleccionar el tipo de consulta
<b>Descripción</b>	La aplicación mostrará un listado de los tipos de consulta que se podrán realizar sobre el conjunto de datos y permitirá elegir entre cualquiera de ellas.
<b>Importancia</b>	Muy alta

Tabla 4 RF-02: Seleccionar el tipo de consulta

<b>RF-03</b>	Búsqueda de información
<b>Descripción</b>	Dependiendo del tipo de consulta elegida la aplicación deberá mostrar una serie de resultados o bien, permitir al usuario introducir los parámetros que desee y realizar la búsqueda en función a ellos.
<b>Importancia</b>	Muy alta

Tabla 5 RF-03: Búsqueda de información

<b>RF-04</b>	Visualización de los resultados de la búsqueda
<b>Descripción</b>	El sistema permitirá visualizar el resultado de la búsqueda que se haya realizado, y además deberá mostrarlos de manera entendible por el usuario.
<b>Importancia</b>	Muy alta

Tabla 6 RF-04: Visualización de los resultados de la búsqueda

<b>RF-05</b>	Elección del test
<b>Descripción</b>	El sistema mostrará una serie de botones. Cada uno de ellos permitirá realizar diferentes pruebas con distintos tipos predefinidos de búsquedas.
<b>Importancia</b>	Alta

Tabla 7 RF-05: Elección del test

<b>RF-06</b>	Lectura del fichero con las consultas
<b>Descripción</b>	La aplicación leerá un fichero con 500 tipos aleatorios de consultas, dependiendo del tipo de prueba elegido, deberá tener unos parámetros u otros.
<b>Importancia</b>	Media

Tabla 8 RF-06: Lectura del fichero con las consultas

<b>RF-07</b>	Visualización de los resultados del test
<b>Descripción</b>	La aplicación mostrará los resultados de la ejecución del test, en esta pantalla se deberá visualizar el tiempo que ha tardado la ejecución de dicho test y el número total de triples que ha obtenido.
<b>Importancia</b>	Media

Tabla 9 RF-07: Visualización de los resultados del test

#### 4.2.4. Especificación de requisitos no funcionales

<b>RNF-01</b>	Espacio de almacenamiento
<b>Descripción</b>	La aplicación requerirá de unos 55 MB de almacenamiento disponibles en el caso de la aplicación Java, y unos 60 MB en el caso de la aplicación C++.
<b>Importancia</b>	Baja

Tabla 10 RNF-01: Espacio de almacenamiento

<b>RNF-02</b>	Dispositivo móvil
<b>Descripción</b>	El dispositivo móvil donde se ejecutará la aplicación deberá disponer de una versión de Android 4.0.4 o superior, así como un procesador con al menos 1'5 GHz de potencia, y contar con 1 GB, o más, de memoria RAM.
<b>Importancia</b>	Muy Alta

Tabla 11 RNF-02: Dispositivo móvil

<b>RNF-03</b>	Dispositivo móvil para la ejecución de los test
<b>Descripción</b>	El dispositivo móvil, donde se ejecutarán los test para averiguar el consumo de batería, deberá disponer de una versión de Android 4.0.4 o superior, así como un procesador con al menos 1 GHz de potencia, y contar con 768 MB, o más, de memoria RAM.
<b>Importancia</b>	Muy Alta

Tabla 12 RNF-03: Dispositivo móvil para la ejecución de los test

<b>RNF-04</b>	Adaptar la visualización a la resolución de pantalla
<b>Descripción</b>	La aplicación solo podrá ser visualizada tanto de forma vertical, como de forma horizontal, ajustando su representación dinámicamente a como se encuentre el dispositivo en un determinado momento.
<b>Importancia</b>	Baja

Tabla 13 RNF-04: Adaptar la visualización a la resolución de pantalla

#### 4.2.5. Modelo de casos de uso

En este apartado describiremos los casos de uso que tendrán nuestras aplicaciones. Con ellos vamos a detallar los diferentes usos del sistema y como los usuarios podrán interactuar con las aplicaciones. En ellos se debe especificar paso por paso las diferentes acciones que realiza el sistema y el resultado que producirá cada una de ellas para un determinado actor.

#### 4.2.5.1. Identificación de actores

A continuación vamos a reconocer a aquellas entidades externas que puedan interactuar con la aplicación.

- **Usuario de la aplicación:** Este actor representa a un usuario normal de la aplicación. Este se encargará de la instalación, ejecución y uso de la aplicación. También se considerará que los usuarios están familiarizados con el uso de dispositivos móviles y sus aplicaciones. Por último, para algunas opciones de la aplicación, el usuario deberá tener conocimientos mínimos sobre el formato HDT.

#### 4.2.5.2. Diagrama de casos de uso

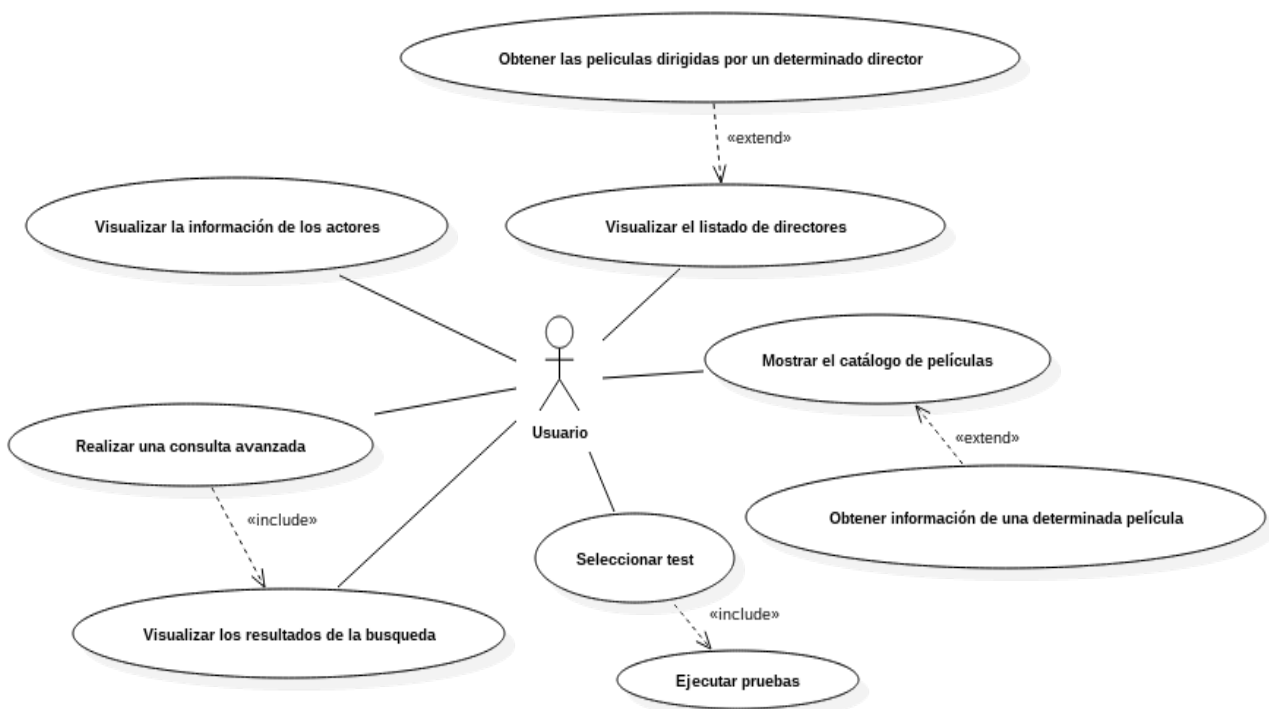


Ilustración 14: Diagrama de casos de uso

4.2.5.3. Especificación de casos de uso

CU-01	Visualizar la información de los actores	
Versión	1.0	
Dependencias	RF-01 RF-02 RF-03 RF-04	
Precondición		
Descripción	El sistema mostrará al usuario el listado completo de los actores que contiene el conjunto de datos, de los que se deberá mostrar tanto el nombre del actor como su identificador único.	
Secuencia Normal	<b>Paso</b>	<b>Acción</b>
	P1	El usuario selecciona la opción "Consulta Actores".
	P2	La aplicación realiza una consulta al conjunto de datos.
	P3	La aplicación muestra el listado completo de los actores con su identificador.
Postcondición	La aplicación muestra el nombre y el identificador de todos los actores que contiene el conjunto de datos.	
Importancia	Alta	
Prioridad	Media	
Comentarios	Ninguno	

Tabla 14 CU-01: Visualizar la información de los actores

CU-02	Visualizar el listado de directores	
Versión	1.0	
Dependencias	RF-01 RF-02 RF-03 RF-04	
Precondición		
Descripción	La aplicación mostrará al usuario el nombre de todos los directores que contiene el conjunto de datos.	
Secuencia Normal	<b>Paso</b>	<b>Acción</b>
	P1	El usuario selecciona la opción "Consulta Directores".
	P2	La aplicación realiza una consulta al conjunto de datos.
	P3	La aplicación muestra el listado completo de los directores.
Postcondición	La aplicación ha mostrado correctamente el nombre de todos los directores que contiene.	
Importancia	Alta	
Prioridad	Media	
Comentarios	Ninguno	

Tabla 15 CU-02: Visualizar el listado de directores

CU-03	Obtener las películas dirigidas por un determinado director	
Versión	1.0	
Dependencias	RF-01 RF-02 RF-03 RF-04	
Precondición	La aplicación deberá haber obtenido el listado de directores del caso de uso anterior (CU-02).	
Descripción	La aplicación permitirá al usuario seleccionar cualquier director del listado y mostrará las películas dirigidas por el director que se haya elegido.	
Secuencia Normal	<b>Paso</b>	<b>Acción</b>
	P1	El usuario busca un determinado director en el listado.
	P2	El usuario selecciona un director.
	P3	La aplicación realiza una consulta sobre el director elegido al conjunto de datos.
	P4	La aplicación muestra una lista con las películas que ha dirigido el director elegido.
Postcondición	La aplicación ha mostrado correctamente el nombre de todas las películas dirigidas por un determinado director.	
Importancia	Alta	
Prioridad	Media	
Comentarios	Ninguno	

Tabla 16 CU-03: Obtener las películas dirigidas por un determinado director



CU-04		Mostrar el catálogo de películas	
Versión	1.0		
Dependencias	RF-01 RF-02 RF-03 RF-04		
Precondición			
Descripción	La aplicación mostrará al usuario el nombre de todas las películas que contiene el conjunto de datos.		
Secuencia Normal	<b>Paso</b>	<b>Acción</b>	
	P1	El usuario selecciona la opción "Consulta Películas".	
	P2	La aplicación realiza una consulta al conjunto de datos.	
	P3	La aplicación muestra el catálogo completo de películas.	
Postcondición	La aplicación ha mostrado correctamente el nombre de todas las películas que contiene.		
Importancia	Alta		
Prioridad	Media		
Comentarios	Ninguno		

Tabla 17 CU-04: Mostrar el catálogo de películas

CU-05		Obtener información de una determinada película	
Versión	1.0		
Dependencias	RF-01 RF-02 RF-03 RF-04		
Precondición	La aplicación deberá haber obtenido el catálogo de películas del caso de uso anterior (CU-04).		
Descripción	La aplicación permitirá al usuario seleccionar cualquier película del catálogo y mostrará toda la información que rodea a la película que haya elegido.		
Secuencia Normal	<b>Paso</b>	<b>Acción</b>	
	P1	El usuario busca una determinada película en el listado.	
	P2	El usuario selecciona una película.	
	P3	La aplicación realiza una consulta sobre la película seleccionada al conjunto de datos.	
	P4	La aplicación muestra un listado con toda la información referente a dicha película. Dicha información se mostrará en forma de triple.	
Postcondición	La aplicación ha mostrado correctamente toda la información de un determinado actor.		
Importancia	Alta		
Prioridad	Media		
Comentarios	Ninguno		

Tabla 18 CU-05: Obtener información de una determinada película

CU-06	Realizar una consulta avanzada			
Versión	1.0			
Dependencias	RF-01 RF-02 RF-03 RF-04			
Precondición				
Descripción	La aplicación mostrará al usuario tres cuadros de texto en los que podrá introducir los parámetros de la consulta, en el formato <i>triple pattern</i> , que desee realizar al conjunto de datos.			
Secuencia Normal	<b>Paso</b>	<b>Acción</b>		
	P1	El usuario selecciona la opción "Consulta Avanzada".		
	P2	El usuario introduce los parámetros de la consulta.		
	P3	La aplicación comprueba que los datos introducidos son válidos.		
Postcondición	La aplicación obtiene los parámetros de la consulta			
Excepciones	<b>Paso</b>	<b>Acción</b>		
	P3	Los parámetros de la búsqueda no son válidos.		
		1	La aplicación muestra un mensaje indicando que los parámetros introducidos no producen ningún resultado.	
		2	Finaliza el caso de uso.	
Importancia	Alta			
Prioridad	Alta			
Comentarios	Ninguno			

Tabla 19 CU-06: Realizar una consulta avanzada

CU-07		Visualizar los resultados de la búsqueda	
Versión	1.0		
Dependencias	RF-01 RF-02 RF-03 RF-04		
Precondición	La aplicación deberá haber comprobado los parámetros de la búsqueda en el caso de uso anterior (CU-06).		
Descripción	La aplicación mostrará al usuario los resultados, en forma de triple, de los parámetros que ha introducido anteriormente.		
Secuencia Normal	<b>Paso</b>	<b>Acción</b>	
	P1	La aplicación realiza la consulta al conjunto de datos con los parámetros de consulta previamente introducidos.	
	P2	La aplicación muestra el listado de resultados obtenido de la consulta.	
Postcondición	La aplicación ha mostrado correctamente el resultado de todos los triples que coinciden con los parámetros introducidos por el usuario.		
Importancia	Alta		
Prioridad	Alta		
Comentarios	Ninguno		

Tabla 20 CU-07: Visualizar los resultados de la búsqueda

CU-08	Seleccionar test	
Versión	1.0	
Dependencias	RF-01 RF-05	
Precondición		
Descripción	La aplicación mostrará al usuario los diferentes tipos de test que puede realizar.	
Secuencia Normal	<b>Paso</b>	<b>Acción</b>
	P1	El usuario selecciona la opción "Realizar Pruebas".
	P2	La aplicación muestra los diferentes tipos de consulta que se pueden realizar.
Postcondición	La aplicación ha mostrado correctamente los tipos de pruebas que se pueden realizar en la aplicación.	
Importancia	Alta	
Prioridad	Alta	
Comentarios	Ninguno	

Tabla 21 CU-08: Seleccionar test

CU-09	Ejecutar pruebas	
Versión	1.0	
Dependencias	RF-01 RF-05 RF-06 RF-07	
Precondición	La aplicación deberá haber mostrado los tipos de pruebas que se pueden ejecutar en el caso de uso anterior (CU-08).	
Descripción	La aplicación permitirá al usuario ejecutar cualquier tipo de prueba y le mostrará los resultados de haber ejecutado un determinado tipo de test.	
Secuencia Normal	<b>Paso</b>	<b>Acción</b>
	P1	El usuario selecciona un determinado tipo de prueba.
	P2	La aplicación lee un fichero con una serie de consultas predefinidas.
	P3	La aplicación realiza todas las consultas obtenidas de dicho fichero.
	P4	La aplicación muestra los resultados de la ejecución de las consultas.
Postcondición	La aplicación ha leído y ejecutado correctamente el fichero, también deberá haber mostrado los resultados de la ejecución de la prueba.	
Importancia	Alta	
Prioridad	Alta	
Comentarios	Ninguno	

Tabla 22 CU-09: Ejecutar pruebas

### 4.2.6. Modelo conceptual

Este modelo sirve para describir la estructura de los datos y sus restricciones de integridad. Su principal objetivo es el de representar los elementos que forman dicha estructura y sus relaciones. En el apartado 3.2, donde se habla sobre el proyecto LinkedMDB, ya introducimos algunos conceptos básicos del modelo de datos sobre el que está creado dicho proyecto. En este punto vamos a ampliar más esa idea, para ello mostraremos algunos ejemplos de consultas que se podrán realizar en la aplicación al conjunto de datos.

#### 4.2.6.1. Ejemplos de consulta

Imaginemos que realizamos una consulta a LMDB sobre la película *The Lord of the Rings: The Return of the King*, obtendríamos el grafo dirigido y etiquetado en forma de triples que muestra la Ilustración 15:

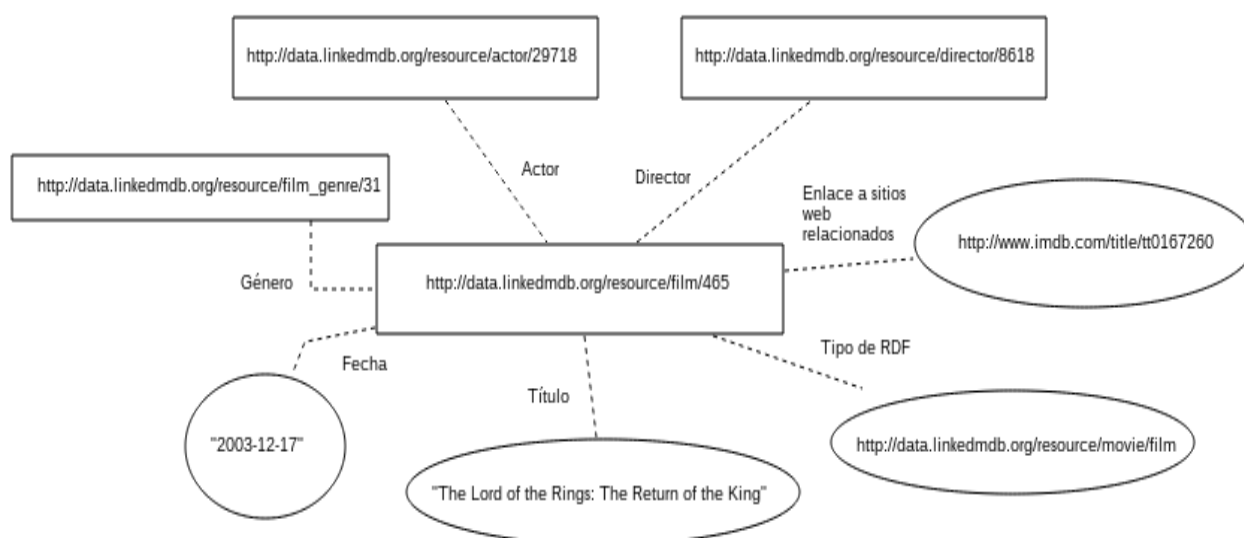


Ilustración 15: Resultados de la consulta de una película específica en LinkedMDB

Donde los rectángulos representan entidades que tendrían más relaciones entre sí dentro del conjunto de datos y las elipses representan nodos finales de un triple, es decir, resultados que solo podemos obtener de la consulta a una URI específica. Los predicados *Actor*, *Director*, *Título*, etc... son los mismos que los que ya se definieron en el apartado 3.2.

Para que resulte aun más claro, la Ilustración 16 muestra el grafo de triples que obtendríamos al realizar una consulta mediante la aplicación sobre el actor *Orlando Bloom*, cuyo identificador único dentro de nuestro conjunto de datos es <http://data.linkedmdb.org/resource/actor/29718>

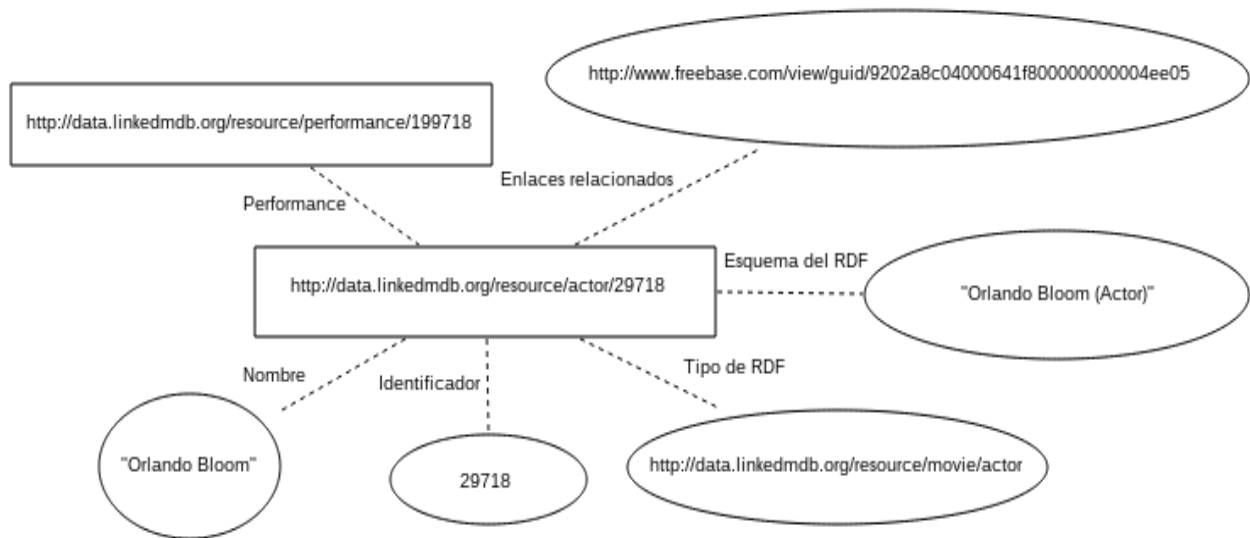


Ilustración 16: Resultados de la consulta de un actor concreto en LinkedMDB

En esta ocasión si que nos va a ser necesario especificar los predicados que aparecen en la Ilustración 16:

Nombre <[http://data.linkedmdb.org/resource/movie/actor\\_name](http://data.linkedmdb.org/resource/movie/actor_name)>

Identificador <[http://data.linkedmdb.org/resource/movie/actor\\_actorid](http://data.linkedmdb.org/resource/movie/actor_actorid)>

Tipo de RDF <<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>>

Esquema del RDF <<http://www.w3.org/2000/01/rdf-schema#label>>

Enlaces relacionados <<http://xmlns.com/foaf/0.1/page>>

Performance <<http://data.linkedmdb.org/resource/movie/performance>>

Ahora que hemos visto estos dos ejemplos, como sabemos, una de las grandes capacidades que tiene el formato HDT es la sencillez con la que la información se relaciona, en la Ilustración 17 se da una muestra de ello uniendo los dos grafos anteriores:



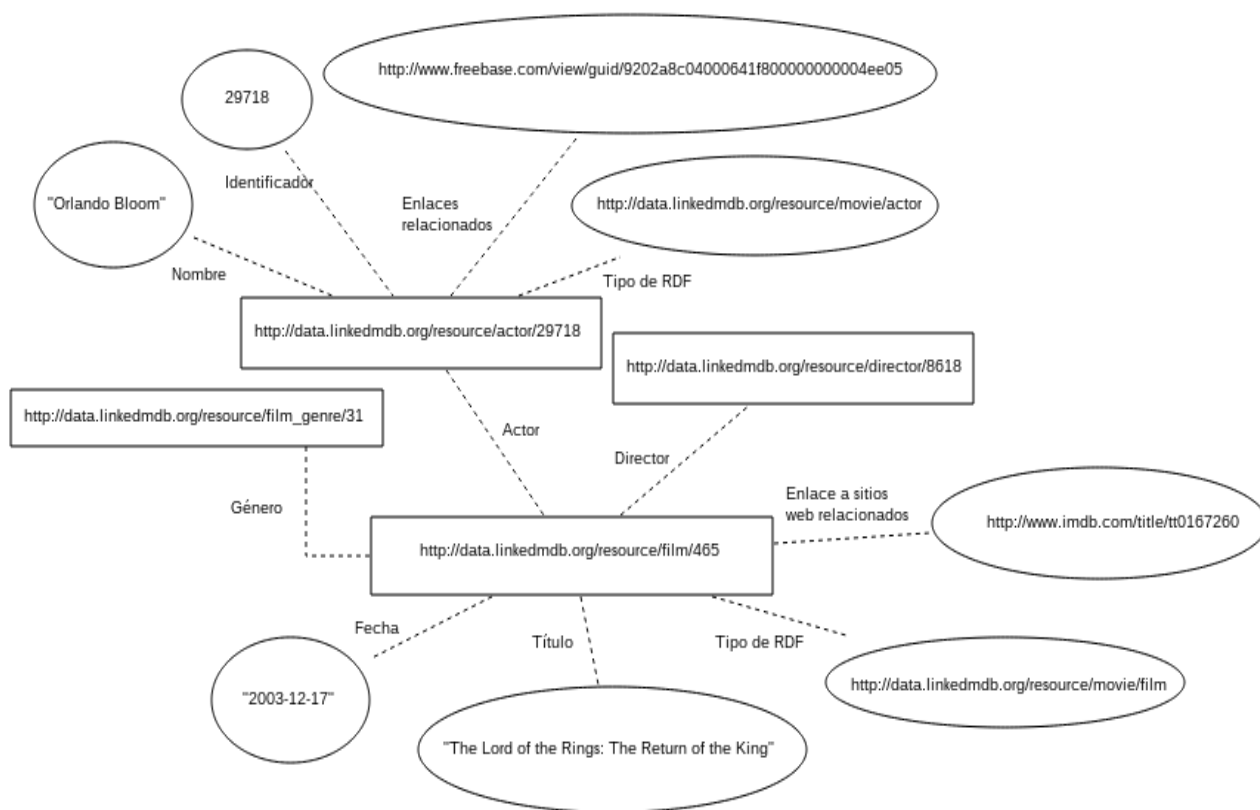
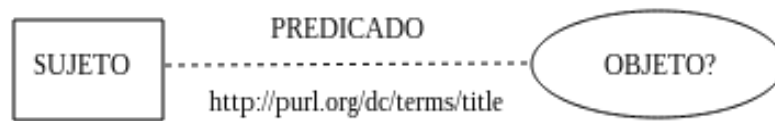


Ilustración 17: Relación entre los resultados en LinkedMDB

Una vez hemos visto de forma general las consultas que podemos realizar, y las respuestas que podemos obtener, vamos a tratar de concretar un poco más, por lo que, a continuación, **mostraremos de manera individual aquellas consultas que soportará nuestra aplicación**, y pondremos algunos ejemplos de los resultados que se podrán obtener.

Haciendo alusión al apartado 2.5 en el que hablábamos sobre SPARQL y los tipos de consulta, basados en *triple patterns*, que soporta, nuestra aplicación ofrecerá por defecto los siguientes tipos de consulta:

- **S P ?** → Consulta en la que desconocemos el parámetro objeto. Para nuestra aplicación realizaremos en primer lugar una consulta sobre todos los directores que contiene el conjunto de datos, esto lo conseguiremos mediante el predicado [http://data.linkedmdb.org/resource/movie/director\\_name](http://data.linkedmdb.org/resource/movie/director_name), una vez obtenido todo el listado, la aplicación permitirá seleccionar cualquiera de los resultados, una vez se haya elegido uno, la aplicación mostrará el nombre de aquellas películas que haya dirigido el director anteriormente elegido. Es decir, que para un cierto predicado P, el cual ha pasado a ser <http://purl.org/dc/terms/title>, necesario para que podamos obtener el título de la película, y un determinado sujeto S, en este caso el director elegido por el usuario, obtenemos como resultado lo dicho anteriormente. En la Ilustración 18 podemos ver el aspecto en forma de triple que tendrá dicha consulta.

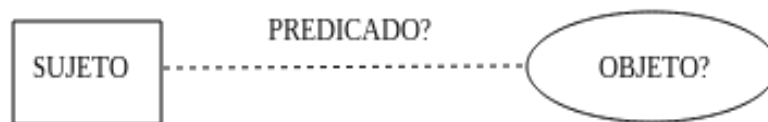


Por ejemplo:  
<http://data.linkedmdb.org/resource/director/8477>

Ilustración 18: Modelo de consulta SP? en forma de triple

Con este tipo de consulta recuperaremos el título de todas aquellas películas que hayan sido dirigidas por *Steven Spielberg*.

- **S ? ?** → Consulta en la que desconocemos tanto el parámetro Predicado como el Objeto. En esta consulta recuperaremos toda la información que rodea a una determinada película. Al igual que en el caso anterior, primero mostraremos una lista completa de todas las películas que contiene LinkedMDB, esto lo conseguiremos haciendo uso del predicado <http://purl.org/dc/terms/title>, una vez hecho esto, permitiremos al usuario elegir una determinada película, y una vez haya escogido, mostraremos toda su información. En la Ilustración 19 se puede ver un ejemplo de este tipo de consulta.



Por ejemplo:  
<http://data.linkedmdb.org/resource/film/5401>

Ilustración 19: Modelo de consulta S?? en forma de triple

Con el ejemplo de consulta anterior recuperaremos toda la información que rodea a la película *Gladiator*.

- **? P ?** → En este tipo de consulta desconocemos el Sujeto y el Objeto. Se trata de una consulta muy sencilla ya que nos devolverá como resultado todos aquellos triples que se encuentren unidos mediante un determinado Predicado, en nuestro caso vamos a recuperar el nombre de todos los actores. Además hemos decidido mostrar también el identificador único de cada actor para que el usuario lo conozca y pueda realizar posteriores búsqueda utilizando ese identificador como sujeto. En la Ilustración 20 se muestra el tipo de consulta que realiza la aplicación en formato *triple pattern*.

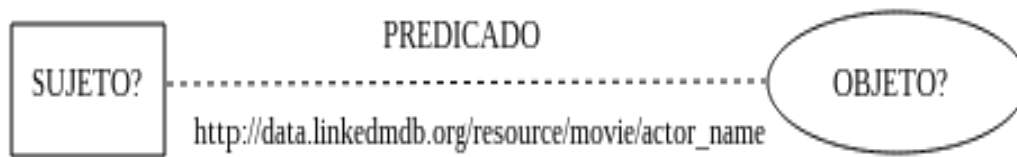


Ilustración 20: Modelo de consulta ?P? en forma de triple

- **S P O** → En este caso, la respuesta que deseamos obtener del conjunto de datos es la de un triple en concreto, para ello debemos indicar los tres parámetros, en caso de que no se encuentre ningún triple que coincida con la información introducida, no se mostrará ningún resultado. Este tipo de consulta está pensada para usuarios más avanzados que tienen conocimientos sobre cómo está definido el conjunto de datos de LinkedMDB. En la Ilustración 21 podemos ver un ejemplo de este tipo de consulta.

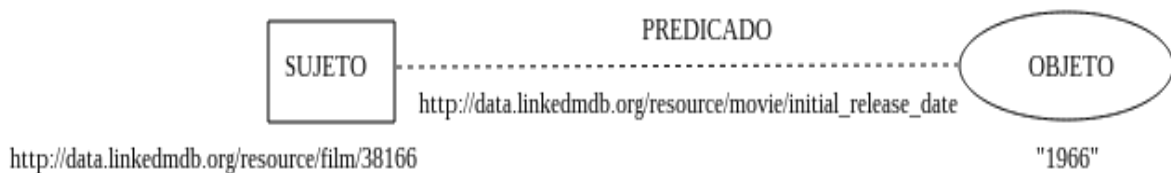


Ilustración 21: Modelo de consulta SPO en forma de triple

Para el resto de consultas (S ? O, ? P O, ?? O y ???) hemos decidido que la consulta avanzada que desarrollaremos para la aplicación se encargue de ellas, dado que el valor por separado de estas consultas no proporciona un aspecto diferencial a la hora de diseñar la aplicación.



## **Capítulo 5. Desarrollo del proyecto**



## **5.1. Planificación, Presupuesto y Coste real**

En esta fase estableceremos diferentes aspectos que serán clave para la correcta consecución del proyecto. Entre estos aspectos se encuentra una primera estimación del tiempo y personas necesarias para desarrollar nuestro proyecto, una planificación temporal en la que podremos observar los plazos que se han de cumplir, un presupuesto del proyecto y por último los costes reales que se han obtenido tras el desarrollo del proyecto.

### 5.1.1. Estimaciones del proyecto

El coste asociado de un proyecto depende de varios aspectos, se suelen dividir en tres grupos: componentes Hardware, las herramientas software y los recursos humanos.

En primer lugar se ha concretado que la imposición que se le impondrá a quien desarrolle este proyecto será que este sea entregado para la convocatoria ordinaria de Junio. Para ello se ha estimado que la duración total del proyecto será de aproximadamente 4 meses y medio, donde el desarrollo del mismo deberá ser realizado por dos personas. Durante esos cuatro meses y medio, hay unos 100 días laborales, en los cuales, los trabajadores tendrán una jornada de 8 horas, su salario será de 12 €/h, y necesitarán entorno a 800 horas para desarrollar el proyecto.

### 5.1.2. Planificación

Como ya hemos visto en el apartado de la estimación, el proyecto tendrá que estar terminado antes de la convocatoria ordinaria de Julio de 2015. Para establecer la planificación de este proyecto se ha optado por dividirlo en 5 etapas: Inicio, Elaboración, Desarrollo, Pruebas y Redacción. Cada una de estas etapas se ha seccionado en diferentes iteraciones en las que se realizan tareas relacionadas con la etapa a la que pertenecen. La tabla 23 muestra las etapas e iteraciones de nuestro proyecto:

Etapa	Iteración	Descripción
Inicio	Iteración 1	Investigación y Planificación.
Elaboración	Iteración 1	Análisis de requisitos y de restricciones. Modelo conceptual y modelos de análisis.
	Iteración 2	Diseño inicial de las aplicaciones. Modelos de diseño.
Desarrollo	Iteración 1	Implementación de la aplicación HDT-Java.
	Iteración 2	Implementación de la aplicación HDT-Cpp.
Pruebas	Iteración 1	Realización del benchmarking en Java.
	Iteración 2	Realización del benchmarking en C++.
Redacción	Iteración 1	Composición de la documentación y los manuales de uso. Entrega final del sistema.

Tabla 23: Etapas e iteraciones del proyecto

En cada etapa del proyecto tendrán lugar una serie de actividades, el siguiente listado muestra todas las que se van a realizar:

1. Etapa de inicio

- Iteración 1:

1. Constitución de los objetivos y el alcance del proyecto.
2. Investigación sobre la Web Semántica y RDF.
3. Estudio del conjunto de datos LinkedMDB.
4. Investigación y pruebas de LinkedMDB en formato HDT.
5. Aprendizaje del lenguaje C++.
6. Estudio de JNI.
7. Aprendizaje de NDK.
8. Establecimiento de la planificación.

2. Etapa de elaboración

- Iteración 1:

1. Obtención de requisitos.
2. Análisis de los requisitos.
3. Establecimiento de los requisitos.
4. Fijación de las restricciones.
5. Estudio del modelo conceptual de LinkedMDB



6. Elaboración del modelo conceptual.
  7. Creación de los modelos de análisis.
  - Iteración 2:
    1. Planteamiento de la arquitectura del proyecto.
    2. Diseño del aspecto de la aplicación.
    3. Estudio de los modelos de diseño.
    4. Elaboración de los diagramas de clases.
    5. Diseño del benchmarking.
3. Etapa de desarrollo
- Iteración 1:
    1. Preparación del entorno de desarrollo.
    2. Desarrollo de las funcionalidades de la aplicación Java.
  - Iteración 2:
    1. Preparación de las librerías.
    2. Creación de la librería nativa.
    3. Desarrollo de las funcionalidades de la aplicación C++.
4. Etapa de pruebas
- Iteración 1:
    1. Realización de pruebas en la aplicación Java.
    2. Análisis de los resultados.
    3. Establecimiento de métricas
  - Iteración 2:
    1. Realización de pruebas en la aplicación C++.
    2. Análisis de los resultados.
    3. Establecimiento de métricas.
5. Etapa de redacción
- Iteración 1:
    1. Finalización de la documentación.
    2. Elaboración de los manuales.
    3. Entrega del proyecto.

En las Ilustraciones 22 y 23 se muestra el resultado de trasladar las actividades que acabamos de describir al diagrama de Gantt:

Nombre	Duración	Inicio	Terminado
[-] Etapa de inicio	33 days	10/02/15 8:00	26/03/15 17:00
[+] Iteración 1	33 days	10/02/15 8:00	26/03/15 17:00
[-] Etapa de elaboración	27 days	27/03/15 8:00	4/05/15 17:00
[+] Iteración 1	16 days	27/03/15 8:00	17/04/15 17:00
[+] Iteración 2	11 days	18/04/15 7:00	4/05/15 17:00
[-] Etapa de desarrollo	28 days	5/05/15 8:00	11/06/15 17:00
[+] Iteración 1	13 days	5/05/15 8:00	21/05/15 17:00
[+] Iteración 2	15 days	22/05/15 7:00	11/06/15 17:00
[-] Etapa de pruebas	8 days	12/06/15 8:00	23/06/15 17:00
[+] Iteración 1	4 days	12/06/15 8:00	17/06/15 17:00
[+] Iteración 2	4 days	18/06/15 8:00	23/06/15 17:00
[-] Etapa de redacción	5 days	24/06/15 8:00	30/06/15 17:00
[+] Iteración 1	5 days	24/06/15 8:00	30/06/15 17:00

Ilustración 22: Planificación de etapas e iteraciones

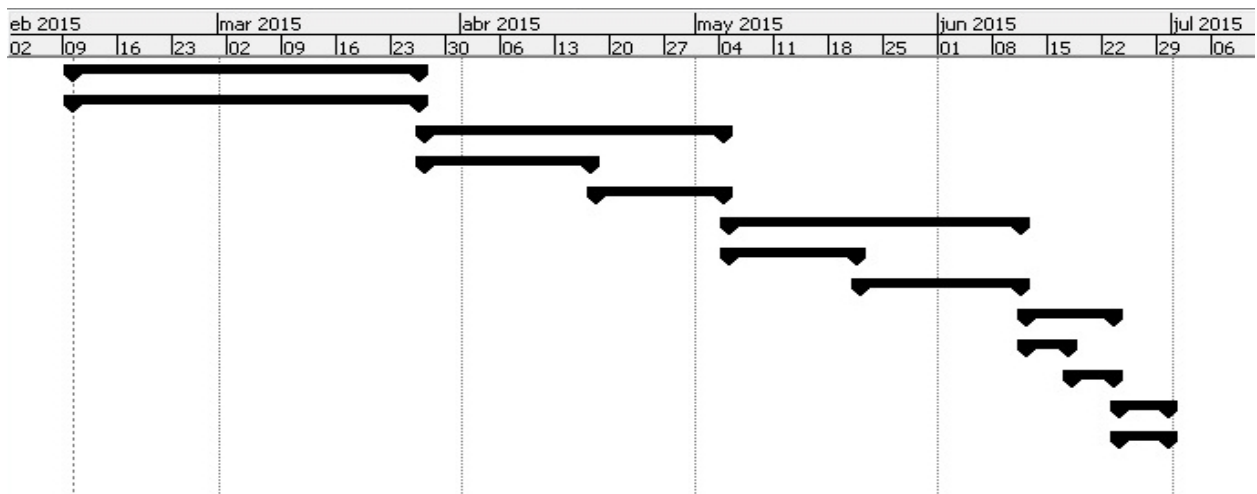


Ilustración 23: Diagrama de Gantt

### 5.1.3. Presupuesto

En esta fase debemos obtener una aproximación del presupuesto que supondrá realizar el desarrollo completo de nuestro proyecto. Para ello estimaremos el coste, tanto del Hardware, como del Software y por supuesto el del recurso humano, de forma proporcional, es decir, en base al uso que vayamos a hacer de ellos durante el tiempo que nos lleve desarrollar el proyecto. Como podemos observar en el diagrama de Gantt anterior el tiempo total de desarrollo de nuestro proyecto es de 101 días.

### 5.1.3.1. Presupuesto Hardware

Para el desarrollo de nuestro proyecto nos van a ser necesarios los siguientes elementos:

- ◆ Dos ordenadores personales con los que se realizarán los estudios previos de investigación, se implementarán las aplicaciones y se elaborará la documentación.
- ◆ Dos dispositivos móviles en los que se instalarán las aplicaciones y se realizarán las pruebas.
- ◆ Conexión a Internet, de fibra óptica, a través del cual obtendremos las diferentes herramientas software y podremos consultar información.

Para este proyecto utilizaremos como ordenador personal un portátil, en la actualidad la mayoría de este tipo de ordenadores tiene una vida estimada de unos 4 años, como puede verse en el diagrama de Gantt, la duración del proyecto es de **101 días**, por lo que su **porcentaje de uso del portátil será del 7%** aproximadamente.

Hemos cifrado también la vida útil de un *smartphone* en unos 2 años, por lo que en proporción habrá sido utilizado un **14%** aproximadamente.

En el caso de la conexión a Internet tendremos que calcular su precio no en base al uso, si no al tiempo total incluidos festivos y días no laborales, lo que hace que calculemos el coste de la conexión a Internet durante 5 meses.

Hardware	Uso (%)	Coste total (€)	Coste en base al uso (€)
Ordenador personal	7	$570 * 2 = 1.140$	80
Dispositivo móvil	14	$200 * 2 = 400$	56
Conexión a Internet	5 (meses)	20	100

Tabla 24: Presupuesto Hardware total

Por lo tanto la suma de los costes totales del Hardware es **236 €**.

### 5.1.3.2. Presupuesto Software

Para el desarrollo de la aplicación necesitaremos las siguientes herramientas:

- Ubuntu
- LibreOffice
- StarUML
- OpenProj
- Eclipse
- Android
- Android SDK
- Android NDK
- LibHDT-Java
- LibHDT-Cpp

Software	Uso (%)	Coste total (€)	Coste en base al uso (€)
Ubuntu	8,15	0	0
LibreOffice	8,15	0	0
StarUML	8,15	0	0
OpenProj	8,15	0	0
Eclipse	8,15	0	0
Android	8,15	0	0
Android SDK	8,15	0	0
Android NDK	8,15	0	0
LibHDT-Java	8,15	0	0
LibHDT-Cpp	8,15	0	0

Tabla 25: Presupuesto Software total

## Capítulo 5. Desarrollo del proyecto

Como podemos observar el coste total del Software es **0 €**, esto reducirá notablemente el presupuesto total del proyecto y se debe principalmente al uso de herramientas y sistemas operativos de libre distribución.

### 5.1.3.3. Presupuesto de desarrollo

Como podemos ver en el apartado de la estimación, este proyecto ha sido pensado para ser realizado por 2 personas, por lo tanto, el presupuesto de desarrollo es el siguiente:

El número de días trabajados según el diagrama de Gantt es de 101, a 8 horas cada día, hacen un total de 808 horas.

Personal	Tiempo (h)	Coste (€/h)	Coste Total (€)
Ingeniero	808	12	$808 * 12 = 9.696$

Tabla 26: Presupuesto de Desarrollo total

Como el proyecto ha sido desarrollado por dos personas, el coste total de los recursos humanos es de  $9696 * 2$ , lo que hacen **19392 €**.

### 5.1.3.4. Presupuesto total

El presupuesto total es la suma de los 3 presupuestos calculados anteriormente:

Presupuesto	Coste (€)
Hardware	236
Software	0
Desarrollo	19.392

Tabla 27: Presupuestos totales

Lo que hace un **coste total de 19628 €**.

5.1.4. Coste real

Como ya sabemos, los plazos de entrega en un proyecto no siempre pueden ser cumplidos, en este caso en concreto se estimo que el proyecto debería ser realizado por dos personas, pero realmente este proyecto ha sido desarrollado únicamente por un estudiante de Grado en Ingeniería Informática de Servicios y Aplicaciones, motivo principal por el que no se han podido cumplir los plazos de entrega establecidos. Por ello a continuación, lejos de estimaciones previas, mostraremos la planificación real que se ha seguido seguido para el desarrollo de este proyecto. Las Ilustraciones 24 y 25 muestran el diagrama de Gantt asociado a un nuevo plazo de entrega, fijado para la convocatoria extraordinaria de Septiembre, y seguidamente calcularemos el presupuesto en base a dicha fecha y con los cambios requeridos.

[-] Etapa de inicio	51 days	10/02/15 8:00	21/04/15 17:00
[-] Iteración 1	51 days	10/02/15 8:00	21/04/15 17:00
[-] Etapa de elaboración	34 days	22/04/15 7:00	8/06/15 17:00
[-] Iteración 1	20 days	22/04/15 7:00	19/05/15 17:00
[-] Iteración 2	14 days	20/05/15 7:00	8/06/15 17:00
[-] Etapa de desarrollo	44 days	9/06/15 8:00	7/08/15 17:00
[-] Iteración 1	16 days	9/06/15 8:00	30/06/15 17:00
[-] Iteración 2	28 days	1/07/15 7:00	7/08/15 17:00
[-] Etapa de pruebas	10 days	10/08/15 8:00	21/08/15 17:00
[-] Iteración 1	5 days	10/08/15 8:00	14/08/15 17:00
[-] Iteración 2	5 days	17/08/15 8:00	21/08/15 17:00
[-] Etapa de redacción	6 days	24/08/15 8:00	31/08/15 17:00
[-] Iteración 1	6 days	24/08/15 8:00	31/08/15 17:00

Ilustración 24: Planificación real de etapas e iteraciones

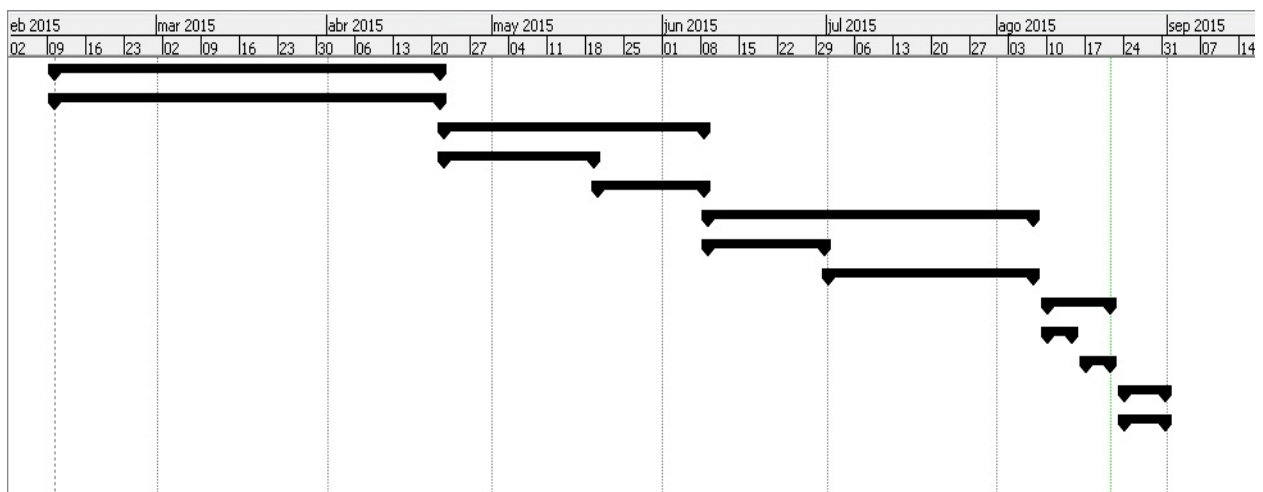


Ilustración 25: Planificación real del diagrama de Gantt

Como podemos ver en las Ilustraciones anteriores este el total de días necesarios para desarrollar el proyecto asciende a **145 días**.

#### 5.1.4.1. Coste real Hardware

Dado que vamos a utilizar los mismos elementos descritos en el apartado 5.1.3.1, únicamente estableceremos los nuevos porcentajes de uso que vamos a dar a los componentes Hardware.

Puesto que el número de días ha aumentado hasta los 145 días, **el porcentaje de uso que le daremos al portátil se incrementará hasta el 10% aproximadamente.**

Del mismo modo sucede con el terminal móvil, cuyo **porcentaje de uso aumenta hasta el 20%.**

En el caso, el uso de la conexión a Internet gira entorno a los 7 meses.

Hardware	Uso (%)	Coste total (€)	Coste en base al uso (€)
Ordenador personal	10	570	57
Dispositivo móvil	20	200	40
Conexión a Internet	7 (meses)	20	140

Tabla 28: Coste real total del Hardware

Por lo tanto, el coste real del hardware es de **237 €.**

#### 5.1.4.2. Coste real Software

El coste real del software es el mismo que el que aparece en el apartado 5.1.3.2, por lo tanto, para evitar repetirnos, solamente indicamos que el coste total del software también es de **0 €.**

#### 5.1.4.3. Coste real desarrollo

Como el número de personas que realizan el proyecto ha variado, el coste real de desarrollo es el siguiente:

El número de días trabajados según el diagrama de Gantt anterior es de 145, a 8 horas cada día, hacen un total de **1160 horas.**

Personal	Tiempo (h)	Coste (€/h)	Coste Total (€)
Ingeniero	1.160	12	1.160 * 12 = 13.920

Tabla 29: Coste real total del Desarrollo

Por lo tanto el coste real de los recursos humanos es de **13920 €**.

#### 5.1.4.4. Coste real total

El coste real total es la suma de los 3 costes reales calculados anteriormente:

Coste Real	Coste (€)
Hardware	237
Software	0
Desarrollo	13.920

Tabla 30: Coste real final

Lo que hace un **coste total de 14157 €**.

#### 5.1.5. Conclusiones

Podemos destacar varios aspectos de la diferencia entre el presupuesto que se estimó antes de realizar el proyecto y el coste real del mismo.

En primer lugar, como puede apreciarse el presupuesto de hardware es prácticamente igual aunque el número de elementos utilizados sea el doble en el presupuesto frente al coste real. El motivo de que ambos costes totales de hardware sean similares es la diferencia de tiempo en la que se utilizan, puesto que en la estimación del presupuesto su uso era de 101 días frente a los 145 días del coste real.

La diferencia en los costes totales entre el presupuesto y el coste real también se ve afectada por el supuesto anterior.

La conclusión que podemos sacar es muy simple, cuanto mayor sea el número de trabajadores en un proyecto mayor será su coste total de desarrollo, pero también más se reducirán los tiempos de entrega.



## 5.2. Diseño

En esta fase vamos a convertir los requisitos definidos en la etapa de análisis en la aplicación software que queremos desarrollar. A lo largo de este apartado iremos transformando el modelo conceptual previamente descrito, en las piezas de software necesarias para dar forma a nuestra aplicación.

### 5.2.1. Arquitectura lógica

En la actualidad el **modelo–vista–controlador (MVC)**, se utiliza en una gran cantidad de aplicaciones. Se trata de un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación, de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Algunas de las claves que hacen que esta arquitectura sea tan ampliamente utilizada, son la reutilización de código y la separación de conceptos, aspectos que posteriormente nos facilitarán la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

Como cabía esperar, el sistema operativo Android también se ha hecho eco de este patrón, por lo que las aplicaciones que vamos a desarrollar en este proyecto también lo utilizarán, aunque con una pequeña diferencia, y es que existe un problema relacionado con el código nativo (Java) de Android, dicho problema tiene que ver con la gestión del **Controlador** sobre las Vistas, es decir, que podemos encontrarnos con vistas en las que se mezcle la interfaz con el acceso a los datos, cosa que obviamente no queremos que suceda. Este problema unido además a que el Controlador tiene capacidades de gestión de eventos, hace que la arquitectura MVC de Android elimine la parte del Controlador gestionada por el propio sistema, pasando a utilizar una arquitectura propia conocida como **modelo-vista-presentador (MVP)**. A continuación describiremos cada una de sus partes:

- ◆ **Vista:** Incluye las interfaces de usuario (Activities, recursos, layouts, etc...), adicionalmente contendrá una referencia a la capa presentador. La vista tiene como tarea llamar a un método del presentador cada vez que se realice una acción sobre la interfaz, normalmente pulsar un botón, un elemento de una lista, etc.
- ◆ **Presentador:** Interactúa de intermediario entre la vista y el modelo, incluye nuestras clases de negocio, es decir, que contiene las clases con métodos que albergan la lógica de nuestra aplicación. Recupera los datos del modelo y se los devuelve a la vista formateados, y a diferencia del MVC típico, también decide qué ocurre cuando se interactúa con la vista.
- ◆ **Modelo:** Es el proveedor de los datos que queremos mostrar en la vista, también incluye la definición lógica del modelo de datos con el que vamos a trabajar.

Además de este patrón de arquitectura, utilizaremos las siguientes librerías:

## Capítulo 5. Desarrollo del proyecto

- **Java HDT library:** Es la librería que nos va a permitir realizar consultas HDT, la podemos encontrar en la página oficial del proyecto RDF/HDT y en ella están contenidas todas las herramientas necesarias para poder desarrollar la parte del modelo de la aplicación.
- **C++ HDT library:** Proporcionada por el mismo grupo que la librería de Java, esta librería es algo diferente a la de Java ya que nos provee una API con la que podemos utilizar los archivos de programación que contiene, no como en el caso de Java donde la librería esta lista para ser usada. Para hacer uso de esta librería en el sistema operativo Android serán necesarios algunos cambios, los cuales veremos en los siguientes apartados.

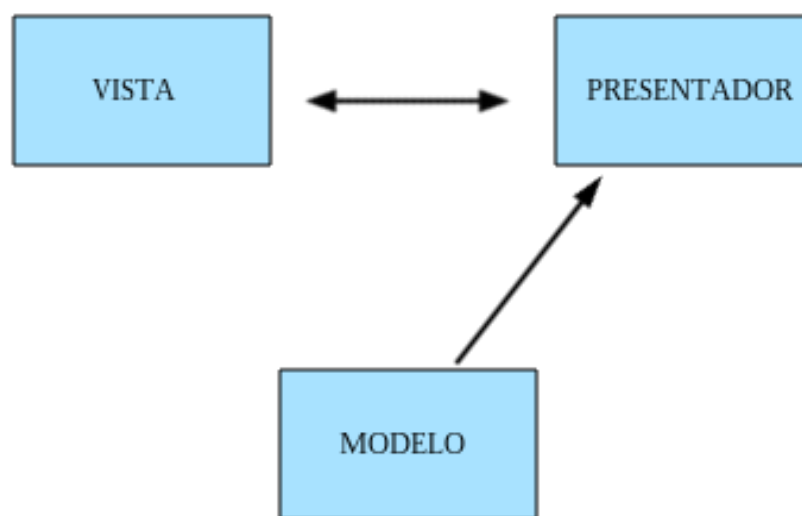


Ilustración 26: Arquitectura lógica

### 5.2.2. Diagrama de clases

Se trata de un diagrama que describe la estructura de un sistema mostrando las clases del mismo, sus atributos, las operaciones que puede realizar y las relaciones entre sus objetos. Su principal objetivo es el de facilitar la comprensión del sistema que vamos a desarrollar, por ello, para tener un mayor entendimiento de dichos diagramas, se hace necesario describir las clases y conceptos básicos que rodean al sistema operativo Android.

- **Activity:** Es una clase del paquete de Android, esta formada por una parte lógica (un archivo Java) que se crea para poder manipular, interactuar y colocar el código de esa actividad, y una parte gráfica, la cual contiene todos los elementos que estamos viendo en la pantalla y que son declarados mediante etiquetas. Podríamos llegar a decir que todas las pantallas de una aplicación son Activities.
- **SplashActivity:** Es una pantalla de carga que se ejecuta en el inicio del ciclo de vida de la aplicación, básicamente sirve para mejorar la inmersión del usuario en nuestra aplicación. En nuestra *splash screen* mostraremos un “logo” diseñado por nosotros específicamente para la aplicación, y tras una breve pausa, pasaremos a la siguiente pantalla.

## *Capítulo 5. Desarrollo del proyecto*

- **Main Thread:** Con la ejecución de una aplicación en el dispositivo móvil, Android le asocia un hilo de ejecución propio conocido como **Main Thread**. Este hilo principal se encarga de soportar la carga de procesamiento de las operaciones que realice la aplicación que se está ejecutando y los componentes que forman la interfaz de usuario de dicha aplicación. En el momento de que cualquier tarea del Main Thread bloquee la ejecución durante más de 5 segundos, se producirá un error grave que llevará al cierre de la aplicación. Por regla general los componentes de la vista no pueden ser modificados por hilos que no sean el Main Thread.
- **Adapter:** El principal objetivo de un adapter es el de rellenar los objetos de la vista con los datos que se deben mostrar. Entre las capacidades de un adaptador está el acceso a la colección de datos y la responsabilidad de crear una vista para cada elemento extraído de dicha colección.
- **Application:** Este objeto representa el estado común de una aplicación, es decir, que cuando haya alguna parte de la aplicación Android funcionando, el objeto “Application” será creado. Normalmente este tipo de clase se utiliza para guardar diferentes tipos de datos que vayan a ser utilizados en muchas Activities. Para el caso concreto de este proyecto, en esta clase guardaremos el fichero HDT que hemos cargado en memoria.
- **Intent:** Un intent es un elemento básico de comunicación entre los distintos componentes de Android. Se pueden entender como los mensajes o peticiones que son enviados entre los distintos componentes de una aplicación o entre distintas aplicaciones. Mediante un intent se puede mostrar una actividad desde cualquier otra, iniciar un servicio, enviar un mensaje broadcast, iniciar otra aplicación, etc.
- **Layout:** Sirve para definir la estructura visual que tendrá el usuario de la aplicación que desarrollaremos, divide en espacios o campos la pantalla del móvil y ayuda a facilitar la distribución de los elementos.

Ahora que hemos explicado estos conceptos básicos sobre Android, pasamos a describir el diagrama de clases de la aplicación y algunas de sus operaciones:



## Capítulo 5. Desarrollo del proyecto

Para aclarar este diagrama de clases, vamos a describir algunas de las operaciones que podemos realizar a través de las aplicaciones, pero antes que esto hay que destacar que aunque en el diagrama no aparezcan unidas entre si, las clases **HDTManager**, **HDT**, **IteratorTripleString**, **TripleString** y **HDTApplication** están relacionadas prácticamente con todas las clases que forman el proyecto, pero para simplificar la visión del diagrama y permitir que se entienda de manera más sencilla, las hemos eliminado.

Al iniciar la aplicación lo primero con lo que nos encontraremos será con la clase **SplashActivity** perteneciente a la capa vista y la cual representa una pantalla de carga con el logo característico de nuestra aplicación, simplemente sirve para introducir de una manera más formal al usuario en nuestro programa.

Una vez pasada esta pantalla de carga la siguiente clase con la que nos encontramos es con la **Pantalla\_inicio** en la que se producen numerosas operaciones, entre ellas se encuentra la carga del conjunto de datos que utilizaremos durante todo el ciclo de vida de la aplicación, para que este fichero sea cargado correctamente debe estar situado en un emplazamiento valido dentro del dispositivo móvil y además encontrarse en formato HDT. Una vez la aplicación haya encontrado y cargado en memoria el fichero HDT guardará la dirección de memoria en la que se encuentra dicho fichero cargado, esto lo conseguimos mediante el uso de la clase **ApplicationHDT**.

En esta primera pantalla de la aplicación nos encontraremos con una lista de tipos de consulta que podemos realizar en la aplicación, para poder mostrar esta lista y que la aplicación reaccione al pulsar cualquiera de sus elementos es necesario utilizar un *adapter* personalizado, en este caso utilizaremos el **AdapterConsulta** para conseguir tal propósito. Como este listado de consultas contiene imágenes necesitamos de la creación de una clase que guarde las referencias a las imágenes de cada objeto que haya en la lista y además contenga un título para cada uno de los objetos, con este objetivo se crea la clase **Consulta**.

Como ejemplo de uso, vamos a pulsar el botón perteneciente a la consulta avanzada del listado que aparece en la **Pantalla\_inicio**, el siguiente *Activity* con el que nos encontramos es con la **Pantalla\_consulta\_avanzada**, en ella nos encontramos con tres *input box* en los que podremos introducir los parámetros de búsqueda que deseemos, una vez los introduzcamos y pulsemos el botón que aparece, se enviarán a través de un *Intent* los parámetros que hayamos introducido y la búsqueda se realizará en la clase **Pantalla\_resultado** quien se encargará de realizar la consulta, por consiguiente necesitará obtener el conjunto de datos que cargamos en memoria en la pantalla de inicio, para ello llamara a un método implementado en la clase **ApplicationHDT**. Por último mostrará los resultados de forma personalizada mediante el **AdapterResultado**.

Otro ejemplo de uso de la aplicación sería seleccionar la opción consulta actores, la cual nos llevará hasta la clase **Pantalla\_actor**, quien, en este caso, se encarga tanto de la búsqueda, como de la representación en la vista de los resultados, con ayuda de la clase **AdapterActor**. Esta clase es capaz de hacer ambas cosas porque la consulta esta predefinida, es decir, que esta clase se iniciará cuando seleccionemos su opción en la **Pantalla\_inicio**, y durante su ejecución realizará la búsqueda de todos los actores que contiene el conjunto de datos y los mostrará por pantalla con su nombre e identificador único. Al igual que en el caso anterior para realizar la búsqueda será necesario que la **Pantalla\_actor** obtenga el conjunto de datos cargado en memoria a través de la clase

**ApplicationHDT** y a su vez utilice las clases **HDTManager**, **HDT**, **IteratorTripleString** y **TripleString**.

Para finalizar con los ejemplos de uso, elegiremos la opción realizar pruebas presente en la *Pantalla\_inicio*. Como podemos apreciar en la Ilustración 27 la clase **TestConsumo** contiene tanto los adaptadores como las operaciones para realizar la búsqueda. El motivo de hacer esto así es para que el formateo de la vista que recibe el usuario no las llamadas entre clases afecten al rendimiento y las mediciones de las pruebas. Esta clase se encuentra a la espera de que el usuario seleccione un el tipo de test que desea realizar, para ello define una serie de botones que están en modo de escucha y que cada uno de ellos ejecutará una serie de operaciones, por ejemplo, al pulsar el botón perteneciente al test SPO, realizaremos 500 búsquedas a nuestro conjunto de datos, para ello la clase **leerá un fichero** con dichas consultas. Para medir los resultados de los test la clase define varias variables que miden tanto el tiempo de ejecución que ha tardado la aplicación en resolver todas las búsquedas, como el número de las mismas. Al igual que en los dos casos anteriores para poder ejecutar las pruebas es necesario que recoja el conjunto de datos cargado en memoria de la clase **ApplicationHDT** y que utilice los métodos de búsqueda de las clases **HDTManager**, **HDT**, **IteratorTripleString** y **TripleString**.

### 5.3. Implementación

En esta etapa vamos a transcribir, al lenguaje de programación Java, toda la fase de diseño seguida en el apartado anterior. Este apartado contiene las versiones de Android que soporta nuestra aplicación, el uso de la librería del proyecto RDF/HDT, las herramientas que vamos a utilizar y en último lugar el código de aquellas clases de Java que hemos considerado muy relevantes para el funcionamiento del proyecto.

Antes de continuar para comenzar con esta fase del proyecto nos ha sido necesario preparar nuestro ordenador para desarrollar la aplicación. Los pasos que hemos seguido para ello se pueden encontrar en el apartado 10.2 de los Anexos.

#### 5.3.1. Versiones de Android compatibles con el proyecto

Debido a las múltiples versiones de Android, ha sido necesario elegir la versión mínima de Android que soportará nuestra aplicación y la versión más moderna con la que el proyecto será compatible.

Para que nuestra aplicación funcione en el mayor número de dispositivos posible, debemos elegir la versión mínima que permita implementar las características requeridas por los requisitos. Esta versión será *Ice Cream Sandwich v4.0.4 API 15*, uno de los motivos que nos ha llevado a elegir esta versión es que implementa muchas funcionalidades básicas, y que como veremos en el capítulo siguiente, esta versión nos va a ser necesaria para el desarrollo de la aplicación mediante el lenguaje C++.

Para que nuestra versión tenga mayor compatibilidad, actualizaremos el SDK de Android cada vez que salga una nueva versión.

### 5.3.2. Librería Java-HDT

Como ya comentamos en el apartado 5.2.1, vamos a utilizar la librería Java desarrollada por el proyecto RDF/HDT, dicha librería recibe el nombre de *hdt-lib.jar*, y contiene las clases **HDT**, **HDTManager**, **IteratorTripleString** y **TripleString** que hemos visto anteriormente en el diagrama de clases. A la hora de utilizarla nos encontramos con un pequeño problema y es que si queremos hacer uso de adaptadores personalizados tenemos que modificar la clase *TripleString* que se encuentra dentro de la librería. Para ello tenemos que obtener el código fuente a través del repositorio de *Google Code* que nos proporciona **RDF/HDT**, y modificar dicha clase, a la que añadimos una variable de tipo **long** llamada **id**, que nos servirá para determinar la posición de un objeto *TripleString* dentro de un *ArrayList*. Para poder obtener los valores de esta variable también nos ha sido necesario añadir los conocidos métodos **Get** y **Set** a esta clase. Una vez hecho este cambio ya podremos mostrar los resultados de una búsqueda de forma personalizada. Para evitar complicaciones adicionales junto con este proyecto incorporaremos la librería *hdt-lib.jar* modificada.

### 5.3.3. Herramientas utilizadas

En este apartado describiremos brevemente las principales aplicaciones, herramientas y entornos de programación que nos han ayudado a desarrollar este proyecto.

#### 5.3.3.1. Ubuntu

Ubuntu es un sistema operativo basado en GNU/Linux. Sus principales objetivos son la facilidad de uso y mejorar la experiencia de usuario. Está compuesto de múltiple software normalmente distribuido bajo una licencia libre o de código abierto. Incluye diferentes funcionalidades que lo hacen ser un sistema operativo dirigido para desarrolladores. La facilidad de instalación de nuevas herramientas con el uso de la terminal mediante los repositorios, y las actualizaciones de seguridad que son desarrolladas por la comunidad, hacen que este sistema operativo sea una herramienta indispensable a la hora de trabajar en el desarrollo de un proyecto.

Cada seis meses se publica una nueva versión de Ubuntu. Las versiones LTS (Long Term Support), se liberan cada dos años y reciben soporte durante cinco años, para este proyecto nos hemos decantado por utilizar la **versión 12.04 LTS**.

## *Capítulo 5. Desarrollo del proyecto*

El proyecto lo comenzamos haciendo uso del sistema operativo Windows 7, pero debido a la gran cantidad de errores y problemas que nos dio a la hora de trabajar con diferentes librerías, sobre todo con las escritas en C++, decidimos probar Ubuntu dado que tiene soporte específico para dicho lenguaje. Al principio comenzamos usándolo a través de una máquina virtual, pero como vimos que el soporte de las herramientas del proyecto RDFHDT era mucho mejor incluso en una máquina virtual que en Windows, esto propició que decidiésemos utilizar Ubuntu como entorno operativo para el resto del proyecto.

### 5.3.3.2. Eclipse

Es el entorno de programación donde desarrollaremos gran parte de nuestro proyecto, algunos de los motivos que nos han llevado a elegir este programa son: que se trata de software libre, dispone de un Editor de texto con un analizador sintáctico, la compilación es en tiempo real, tiene control de versiones con **CVS**, integración con **Ant**, asistentes para creación de proyectos, clases, tests, etc., y **refactorización**.

Definido como una plataforma de desarrollo, está basado en Java, y compuesto por un conjunto de herramientas multiplataforma para desarrollar aplicaciones. Normalmente ha sido utilizado para desarrollar entornos de desarrollo integrados (IDE), como el famoso IDE de Java, conocido como JDT (Java Development Toolkit o Herramientas de Desarrollo de Java). Eclipse fue desarrollado por IBM y actualmente se encuentra en manos de la Fundación Eclipse.

El aspecto clave que define a Eclipse es que esta basado en módulos (plug-ins) que proporcionan una gran variedad de funcionalidades, a diferencia de otros entornos de programación donde todas las funcionalidades vienen incluidas desde un principio, las necesite el usuario o no. Este mecanismo de módulos hace de Eclipse una plataforma muy liviana, en la que el usuario decide que módulos incluir, concretamente nosotros haremos uso de un módulo para programar en el lenguaje C++.

### 5.3.3.3. Android

Es un sistema operativo basado en Linux pensado para teléfonos móviles inteligentes (smartphones) y tablets, sus principales características se basan en tener un núcleo de sistema operativo libre, gratuito y multiplataforma.

Android era un sistema operativo para móviles prácticamente desconocido hasta que en 2005 Google lo compró. Fue presentado en 2007 junto la fundación del Open Handset Alliance, una asociación de compañías de hardware, software y telecomunicaciones para avanzar en los estándares abiertos de los dispositivos móviles, se lanzó junto con el SDK para que los programadores empezaran a crear sus aplicaciones para este sistema.



Algunas de sus principales componentes y características son las siguientes:

- **Aplicaciones:** El sistema operativo proporciona todas las interfaces necesarias para desarrollar aplicaciones que accedan a las funciones del teléfono (como el GPS, las llamadas, la agenda, etc.) de una forma sencilla en el lenguaje de programación Java. Incluye algunas aplicaciones base como son: un gestor de llamadas, soporte de mensajería instantánea, calendario, un cliente de correo electrónico, navegador web, mapas, etc.
- **Entorno de trabajo de aplicaciones:** Gracias a que las herramientas de programación son gratuitas y que el sistema es completamente libre, los desarrolladores tienen acceso completo a los mismos APIs del framework usados por las aplicaciones base, esto, sumado a que la arquitectura está diseñada para simplificar la reutilización de componentes, hace de Android un sistema en el que los costes para lanzar un teléfono o una aplicación sean muy bajos, lo cual también provoca que la cantidad de aplicaciones disponibles sea ingente, haciendo que se extienda casi sin límites la experiencia del usuario.
- **Bibliotecas:** Android incluye un conjunto de bibliotecas de C/C++ usadas por varios componentes del sistema. Algunas son: System C library, OpenGL para los gráficos 3D, bibliotecas de medios, SQLite, etc.
- **Runtime de Android:** Es un set de bibliotecas básico, que proporciona la gran mayoría de las funciones disponibles pertenecientes al lenguaje Java. Cada aplicación Android ejecuta su propio proceso y en el instancia su propia máquina virtual **Dalvik**. La forma en la que esta máquina virtual ha sido escrita, permite que se puedan ejecutar varias máquinas virtuales simultáneamente, de manera eficiente, en el mismo dispositivo. Los archivos que se ejecutan en Dalvik tienen la extensión **.dex**, Dalvik Executable, la cual está optimizada para utilizar la menor cantidad de memoria posible.
- **Núcleo Linux:** El kernel de Android utiliza Linux para muchos de los servicios principales del sistema como seguridad, gestión de memoria, gestión de procesos, conexiones de red y los diferentes controladores. El núcleo también actúa como una capa de abstracción entre el hardware y el resto del software.

Estas características sumadas a que cualquiera puede bajarse el código fuente, inspeccionarlo, compilarlo e incluso cambiarlo, da una gran seguridad a los usuarios, ya que algo que es abierto permite detectar fallos más rápidamente. Y también a los fabricantes, pues pueden adaptar mejor el sistema operativo a los terminales.

### 5.3.3.4. Android SDK

Un SDK, o Software Development Kit, es generalmente un conjunto de herramientas de desarrollo de software que le permite al programador crear aplicaciones para un sistema concreto, en este caso Android. Un SDK se asemeja a una interfaz de programación de aplicaciones o API, e incluye una serie de herramientas, entre las que comúnmente se encuentra el soporte de detección de errores de

programación. Los SDK también suelen incluir códigos de ejemplo y notas técnicas de soporte, además de documentación adicional que ayuda a clarificar ciertos puntos. Por lo tanto, el SDK de Android es un conjunto de herramientas de desarrollo utilizadas para crear y desarrollar aplicaciones para la plataforma Android mediante el lenguaje de programación Java, y en su instalación incluye lo siguiente:

- Librerías necesarias.
- Debugger.
- Emulador.
- Documentación de las interfaces de programación de aplicaciones (APIs).
- Códigos de ejemplo.
- Tutoriales para el Sistema operativo Android.

### 5.3.3.5. JNI

Es un framework de programación que permite a aplicaciones escritas en Java ejecutar aplicaciones nativas, estas pueden ser desde bibliotecas, a funciones escritas en lenguajes como C, C++ o ensamblador. Así mismo también es posible la **situación inversa**, puesto que se trata de una interfaz bidireccional, es decir, que es posible la ejecución de código Java desde código nativo. Estas funcionalidades permiten a los programadores reutilizar desarrollos en código nativo, ahorrando tiempo en el desarrollo de tareas específicas que, en el caso de que no existiera JNI, habría que **reprogramar**. JNI proporciona una interfaz estandarizada para el acceso a aplicaciones nativas independientemente de la implementación de la máquina virtual, suele ser utilizado para recursos de bajo nivel de la plataforma como funcionalidades de sonido, ficheros o lectura de datos. Antes de entrar más en profundidad con JNI, es necesario definir algunos conceptos fundamentales para entender el papel que juega en Java:

- **Plataforma Java**: es el conjunto formado por la **máquina virtual** de Java (JVM o Java Virtual Machine) y su API. Esta API consiste en una serie de clases predefinidas que realizan un gran número de tareas y sirven de base para la implementación de aplicaciones.
- **Entorno huésped (host environment)**: representa el sistema operativo, un conjunto de librerías nativas y el juego de instrucciones de la CPU. Las **aplicaciones nativas** se escriben en lenguajes de programación nativos como C/C++, se compilan en código máquina y se enlazan con otras librerías nativas. Las aplicaciones nativas son **dependientes** del entorno huésped, al contrario que las aplicaciones Java, que pueden ejecutarse en cualquier plataforma Java estándar. Esto se debe a que las plataformas Java se sitúan encima del entorno huésped, abstrayendo a las aplicaciones del entorno en el que se están ejecutando.

En la Ilustración 28 mostramos el contexto en el que se sitúa JNI:

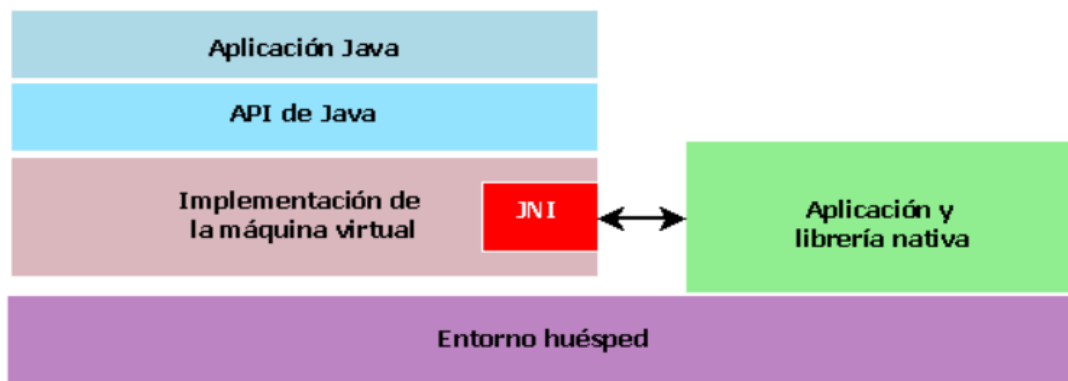


Ilustración 28: Posición de JNI dentro de la JVM

JNI integra código Java y código nativo, soportando **dos tipos de código nativo**:

- **Librería nativa:** es una colección de funciones implementadas en código nativo y compiladas para una determinada máquina y sistema operativo. Es el tipo de código nativo al que puede llamar una aplicación Java desde JNI usando los llamados **métodos nativos**. Estos métodos permiten encapsular la llamada a librerías nativas dentro de métodos Java.
- **Aplicación nativa:** JNI permite, a través de su interfaz de invocación, que una aplicación nativa englobe una implementación de la máquina virtual. De este modo, puede ejecutar aplicaciones Java.

### 5.3.3.6. Android NDK

Es un conjunto de herramientas que nos permite desarrollar aplicaciones nativas, es decir, sin hacer uso de la máquina virtual, esto hace que la ejecución de la aplicación sea en cierto modo más rápida, ya que pasará a ejecutarse directamente en el procesador. Permite a los desarrolladores reutilizar código escrito en C/C++ introduciéndolo en las aplicaciones a través de **JNI** (Java Native Interface).

Hay que ser precavido con el uso del NDK puesto que puede ser útil y beneficioso para ciertos tipos de aplicaciones, bien sea por la necesidad de eficiencia o por la reutilización de código C, pero aun así existen aplicaciones en las que el uso de código nativo en Android no se traduce en una notable mejora de rendimiento, aumentando sin embargo notablemente la complejidad de la aplicación.

Las aplicaciones que suelen utilizar el NDK son aquellas que harán un uso intensivo de la CPU, como por ejemplo motores de videojuegos, procesado de señal o como en nuestro caso consultas masivas a una base de datos. En general, se trata de optimizar al máximo costosas operaciones matemáticas.

Algunas de las aplicaciones más conocidas que utilizan el Android NDK son WhatsApp, Instagram, Skype o TuneIn Radio, muchas de ellas hacen uso de librerías ya desarrolladas en lenguaje C/C++ y aprovechan el NDK para no tener que escribirlas de nuevo en Java, como sucede en nuestro caso.

### 5.3.4. Programación de la Aplicación

En este apartado comentaremos las clases, funciones y ficheros que conforman la aplicación , así como algunas partes de configuración de la misma. Destacaremos aquellas partes del código que sean relevantes, evitando así extender demasiado esta sección.

#### 5.3.4.1. Clase Pantalla\_inicio.java

De esta clase, que extiende de *Activity*, destacaremos el método que carga un fichero en formato HDT en el hilo principal de la aplicación y se lo envía a la clase **ApplicationHDTJ** para que este pueda ser utilizado por cualquier clase de la aplicación. El método recibe el nombre de *cargarFicheroHDT* y su código es el siguiente:

```
private boolean cargarFicheroHDT () {

    /*Creamos un objeto de la clase aplicacion al que pasaremos el
    *conjunto de datos cargado si lo encontramos en la memoria del
    *movil*/

    final ApplicationHDTJ globalVariable = (ApplicationHDTJ)
    getApplicationContext();

    try{

        HDT hdt =
        HDTManager.loadHDT("/mnt/sdcard/Download/linkedmdb2010.hdt",
        null);

        //si llegamos hasta este punto significará que la aplicacion
        //ha encontrado y cargado el fichero y por tanto podremos hacer
        //un "set" del fichero cargado a *la clase ApplicationHDTJ

        globalVariable.setHDT(hdt);

        return true;

    }
```

```
catch(Exception noHayFichero){

    /*Cuando entramos aqui implica que la aplicacion no ha *encontrado
    el fichero por lo que en vez de gestionar la *exception retornamos
    un false para que envíe un mensaje de *error*/

    Toast toast = Toast.makeText(getApplicationContext(), ";Fichero NO
    cargado!", Toast.LENGTH_LONG);

    toast.show();

    return false;

}

} //Fin del método cargarFicheroHDT
```

Dentro de esta misma clase, en el método *onCreate*, propio de un *Activity*, comprobaríamos si el conjunto de datos ha sido cargado o no.

#### 5.3.4.2. Clase ApplicationHDTJ.java

Esta clase es clave para el funcionamiento de nuestra aplicación ya que, a través de ella, conseguimos mejorar el rendimiento. Esto se debe a que si no tuviéramos implementada esta clase cada vez que fuésemos a realizar una consulta a nuestro conjunto de datos tendríamos que cargar en memoria el fichero y por lo tanto estaríamos desaprovechando recursos. Esta clase extiende de *Application*, y el código por el que esta formada es el siguiente:

```
public class ApplicationHDTJ extends Application {

    HDT hdt;

    public void setHDT(HDT biblio) {

        this.hdt = biblio;

    }

    public HDT getHDT() {

        return this.hdt;

    }

}
```

```
}
```

Como vemos, la clase únicamente posee una variable tipo **HDT** cuyo nombre es *hdt* y dos métodos. El primero de ellos, **setHDT**, sirve para almacenar la dirección de memoria en la que se ha cargado el conjunto de datos en formato HDT, si nos fijamos este método recibe como parámetro una variable tipo HDT, la cual recibe el nombre de *biblio*, que vendría a ser el fichero que hemos cargado en la *Pantalla\_inicio*, está *biblio* recibida por parámetro la asociamos a la variable declarada en esta clase llamada *hdt*. El método **getHDT** devuelve el valor que hemos asociado a la variable *hdt*, es decir, que devolverá la posición de memoria en la que está cargado el fichero HDT.

#### 5.3.4.3. Clase Pantalla\_resultado.java

Esta clase, que extiende de *Activity*, es la encargada de mostrar los resultados de la búsqueda que ha realizado un usuario. A continuación, destacaremos las partes del código más importantes de esta clase:

```
final ApplicationHDTJ globalVariable = (ApplicationHDTJ)
getApplicationContext();
```

```
HDT hdt = globalVariable.getHDT();
```

Con este trozo de código obtenemos el conjunto de datos cargado en memoria en la *Pantalla\_inicio*, una vez hecho esto, podemos realizar consultas a dicho conjunto de datos. En primer lugar, para realizar la búsqueda, tendremos que recibir los parámetros introducidos por el usuario, para ello definiremos tres variables que van a almacenar la información que se ha añadido al *Intent* en la clase anterior, y que ahora somos capaces de recuperar gracias al siguiente código:

```
String sujCad = getIntent().getExtras().getString("sujeto");
```

```
String preCad = getIntent().getExtras().getString("predicado");
```

```
String objCad = getIntent().getExtras().getString("objeto");
```

Ahora que ya conocemos los parámetros de la búsqueda podemos pasar a realizarla, para ello utilizaremos las clases **IteratorTripleString.java** y **TripleString.java** que nos proporciona la librería **hdt-lib.jar**, el siguiente código es el encargado de realizar dicha operación:

```
try{
```

```
IteratorTripleString it = hdt.search(sujCad,preCad, objCad);
```

```
while(it.hasNext())
```

```
{  
  
TripleString ts = it.next();  
  
//Rellenamos el adapter con los resultados  
adapter.add(ts);  
  
}  
  
//Incorporamos la informacion en el list view  
listaRes.setAdapter(adapter);  
  
}  
  
catch (Exception e) {  
  
errorBusq = (TextView) findViewById(R.id.errorBusqueda);  
  
errorBusq.setText("Los parámetros que ha introducido no producen  
ningún resultado.");  
  
e.printStackTrace();  
  
}
```

Otro aspecto a tener en cuenta es que para realizar la consulta sobre nuestro conjunto de datos es necesario que gestionemos posibles excepciones, de ahí las cláusulas *try/catch*, como podemos ver en el código nos aprovechamos de esta situación para en el caso de que la búsqueda no devuelva ningún resultado le indiquemos al usuario que los parámetros que ha introducido no coinciden con ningún dato almacenado en nuestro fichero HDT.

#### 5.3.4.4. Clase TestConsumo.java

Como ya comentamos en el apartado 5.2.2, Esta clase agrupa en si misma tanto los adaptadores como las operaciones de búsqueda, esto se hace así para mejorar el rendimiento de las consultas, evitando que se pierdan recursos y tiempo en llamadas a otros activities o adaptadores que realicen la búsqueda o formateen la salida. Esta clase extiende de *Activity* y de ella destacaremos los métodos *leerFichero* y *ejecutarPruebas* que veremos a continuación:

```
public void leerFichero(InputStream flujo) {
```

```
try

{

lector = new BufferedReader(new InputStreamReader(flujo));

String sujeto="", predicado="", objeto="";

//Leemos la primera línea del fichero, con lo que iniciamos
//el proceso de lectura

String texto = lector.readLine();

while(texto!=null)

{

//Creamos un objeto de este tipo para ser capaces de dividir
//líneas del código respecto a los delimitadores que contiene
el //fichero de consultas

StringTokenizer delimitador = new StringTokenizer(texto,";");

sujeto = delimitador.nextToken().trim();

predicado = delimitador.nextToken().trim();

objeto = delimitador.nextToken().trim();

//Una vez hemos dividido la línea que se acaba de leer,
//comprobamos que ninguno de los parámetros sea cero

if(sujeto.equals("0"))

{

sujeto = "";

}

if(predicado.equals("0"))

{
```



```
predicado = "";

}

if(objeto.equals("0"))

{

objeto = "";

}

//Una vez realizado esto, podemos crear un nuevo objeto tipo
//TripleString, en //el almacenamos el sujeto, el predicado y
el //objeto en un objeto de tipo TripleString y guardamos sus
datos //en el ArrayList para leerlo posteriormente

try{

TripleString ts = new TripleString();

//Almacenamos en este nuevo TripleString los valores leidos

ts.setSubject(sujeto);

ts.setPredicate(predicado);

ts.setObject(objeto);

arrayConsultas.add(ts);

}

catch(Exception e){

e.printStackTrace();

}

//Una vez añadido el objeto al array, pasamos a la siguiente
//línea y el programa la leera si existe

texto=lector.readLine();
```

```
}  
  
} // Fin del bucle de lectura  
  
catch (Exception ex)  
  
{  
  
ex.printStackTrace();  
  
}  
  
finally  
  
{  
  
try {  
  
if(flujo!=null)  
  
{  
  
flujo.close();  
  
}  
  
}  
  
catch (IOException e) {  
  
e.printStackTrace();  
  
}  
  
}  
  
} // Fin del método leerFichero
```

Este método permite leer un archivos de texto en formato *txt*. Para nuestro caso tendrá que leer ficheros en los que cada línea tiene el siguiente aspecto:

```
http://data.linkedindb.org/resource/film/7522;0;http://data.linkedinm  
db.org/resource/performance/4143;1
```

## Capítulo 5. Desarrollo del proyecto

Como podemos observar la línea esta delimitada por “;” esto hace que por cada línea que leamos tengamos que dividirla de forma que sea valida para poder realizar las consultas. Una vez que hemos dividido la línea en Sujeto, Predicado y Objeto, la aplicación esta lista para realizar la consulta, pero como queremos evitar ensuciar el tiempo que se tarda en ejecutar una consulta con el de lectura del fichero, esta operación la realizaremos en otro método, por ello creamos un nuevo objeto de tipo *TripleString* y le asociamos las variables sujeto, predicado y objeto que acabamos de obtener de la lectura. El paso siguiente es introducir el *TripleString* en un *ArrayList* y si no ha surgido ningún problema, continuar con la lectura del archivo de texto hasta que el método no encuentre líneas con texto para leer. También podemos ver como al final de la línea aparece un “1”, este número nos indica el número de resultados, en forma de triple, que deberíamos obtener tras realizar la consulta. También cabe destacar que en el código **comprobamos** que cada vez que se lea un “0” debemos cambiarlo por un **espacio vacío**, ya que esto le indica a la herramienta de búsqueda que ese **parámetro de la búsqueda es desconocido**. Los ficheros con las consultas que va a leer la aplicación se encuentran en la carpeta */res/raw/*.

A continuación mostramos el código del método *ejecutarPruebas*:

```
public void ejecutarPruebas () {

//Definimos las variables que mediran el tiempo que tardaran //en
realizarse las consultas y los Strings que recogeran la
//informacion del TripleString

long time_start, time_end, tiempoCons;

String sujetoConsulta = "", predicadoConsulta = "", objetoConsulta
= "";

//Una vez leído el ArrayList esta lleno con todas las
//consultas a realizar a continuación tenemos que recorrerlo
//e ir realizando las consultas
//Comenzamos a medir el tiempo con la lectura del primer
//elemento del Array

time_start = System.currentTimeMillis();

for(int i = 0;i < arrayConsultas.size();i++)

{

sujetoConsulta = arrayConsultas.get(i).getSubject().toString();

predicadoConsulta =
arrayConsultas.get(i).getPredicate().toString();

objetoConsulta = arrayConsultas.get(i).getObject().toString();
```

```
try{

IteratorTripleString its =
hdt.search(sujetoConsulta,predicadoConsulta, objetoConsulta);

while(its.hasNext())

{

TripleString ts = its.next();

}

}

catch(Exception e){

e.printStackTrace();

}

}

//Paramos el tiempo y recogemos el resultado

time_end = System.currentTimeMillis();

//Calculamos el tiempo en milisegundos

tiempoCons = time_end - time_start;

//Mostramos los resultados

numCons.setText(arrayConsultas.size() + " consultas realizadas");

tiempoConsulta.setText(tiempoCons + " milisegundos");

tTotalConsultas = tTotalConsultas + tiempoCons;

tiempoTotal.setText("Tiempo total de todas las consultas: " +
tTotalConsultas + " ms");
```

## Capítulo 5. Desarrollo del proyecto

```
//Para finalizar con este método vaciamos el arrayConsultas y
de //esta forma evitamos que se mezclen consultas y generamos
//tiempos mas reales

arrayConsultas.clear();

} //Fin del metodo ejecutarPruebas
```

Este método básicamente recorre el *ArrayList* con las consultas que añadimos mediante el método *leerFichero* y se encarga de realizar una búsqueda a través de las clases que proporciona la librería **hdt-lib.jar**. Su peculiaridad reside en que mientras se recorre el *ArrayList* y se ejecutan las consultas se esta cronometrando el tiempo que esta tardando la aplicación en realizar dicha tarea, la cual, una vez completa, se mide la diferencia entre el tiempo final y el inicial y se muestran los resultados por pantalla. Por ultimo se borra el contenido del *Array* evitando así que en la siguiente ejecución de las consultas el *Array* no este vacío y por tanto se realicen más consultas de las que se debería.

Para terminar con este apartado vamos a exponer una línea concreta del *AndroidManifest*. Dicha línea de código es la siguiente:

```
android:largeHeap="true"
```

Con el uso de esta clausula hacemos que nuestra aplicación solicite a Dalvik que le conceda un hilo de ejecución mayor, es decir, que le estamos informando a la máquina virtual de que nuestra aplicación va a utilizar procesos que requieren más memoria de lo habitual, por lo que, en caso de que la aplicación tarde más tiempo de lo normal en realizar una tarea, Dalvik no cortará la ejecución de dicha tarea y la gestionará de forma especial.

### 5.4. Optimización

En este apartado del proyecto vamos a desarrollar la aplicación HDT-Cpp, veremos todas las dificultades que entraña desarrollar una aplicación de este tipo, y explicaremos detalladamente cada una de las partes de esta aplicación.

El principal motivo que nos anima a implementar esta aplicación, es la posibilidad de realizar una comparativa del consumo de batería en dos aplicaciones desarrolladas para un mismo dispositivo pero que incorporan tecnologías muy diferentes. Otro de los aspectos que nos lleva a implementar esta parte son los numerosos estudios que demuestran las aplicaciones desarrolladas con el lenguaje Java son mucho más lentas que aquellas que utilizan C++ en alguna de sus partes, con esta aplicación también descubriremos si esta máxima es cierta para todos los casos.

### 5.4.1. Adaptación a Java

Para comenzar a implementar la aplicación desarrollada mediante el lenguaje C++ y que se ejecutará desde un dispositivo móvil con el sistema operativo Android, cuyo lenguaje nativo es Java, nos va a ser necesario realizar una adaptación previa, es decir, que vamos a tener que crear unos envoltorios conocidos como *wrappers* capaces de intercambiar información entre la librería desarrollada en C++ y diferentes clases implementadas en Java.

El motivo de este paso previo es tratar de reducir la complejidad del problema ya que adaptar una librería, escrita completamente en C++, directamente a Android, con la cantidad de particularidades que recoge el sistema operativo, sería un proceso demasiado tedioso y complicado, además de que es un paso inevitable puesto que es necesario ajustar los métodos de la librería C++ a las llamadas que realicemos sobre ellos desde Java. La Ilustración 29 muestra todo el proceso que vamos a llevar a cabo en este apartado:

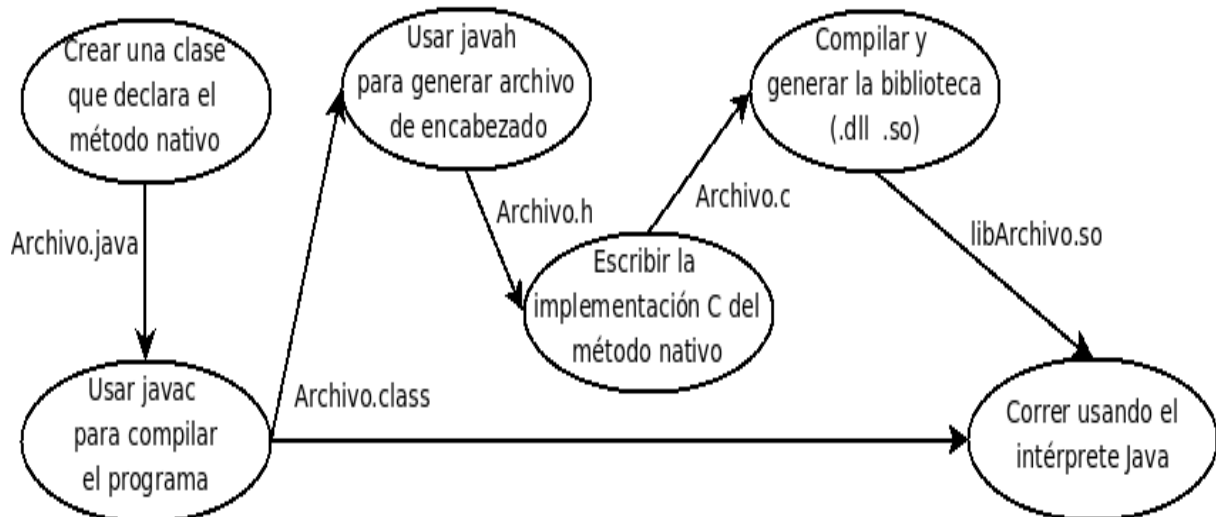


Ilustración 29: Proceso de adaptación de C++ a Java

### 5.4.2. Preparación de la librería

Al igual que la librería Java HDT, podemos obtener la librería C++ HDT de la misma página del proyecto *RDF/HDT*. Una vez que hemos obtenido la librería es necesario que sepamos que no esta preparada para ser adaptada a Java mediante JNI, puesto que por defecto en su compilación era creada con el único objetivo de proporcionar el soporte necesario para si misma. Si queremos que la librería C++ HDT pueda ser utilizada por terceros, debemos añadir a todos los *makefiles* de la librería, la opción “-fPIC” en la variable *CPPFLAGS*.

Los *makefiles* son ficheros de texto, que utiliza la herramienta *make*, para llevar a cabo la gestión de la compilación de programas. Se podrían entender como los guiones de la película que quiere hacer *make*. Todos los *makefiles* están ordenados en forma de reglas, especificando qué es lo que

hay que hacer para obtener un módulo en concreto. Las instrucciones escritas en este fichero se llaman dependencias.

El objetivo de la variable `CPPFLAGS` es indicar al compilador una serie de opciones que ha de tener en cuenta al generar la librería. Dicha opción del *makefile*, **-fpic** (for position-independent code), debemos utilizarla para que la librería se compile de forma que pueda ser utilizada en librerías compartidas, aunque esta opción no es muy recomendable, nos es necesaria para continuar con el desarrollo del proyecto. Además, esta opción requiere un soporte especial, y es que, **solo funciona en determinadas máquinas**. Otro aspecto a considerar es que en sistemas de **32 bits** tendremos que utilizar **-fpic** y en **64 bits -fpic**.

Para realizar esta modificación de forma más rápida esta opción, haremos uso del buscador de Ubuntu. Nos colocamos sobre el directorio raíz de la librería C++, e introducimos la palabra *makefile*. El resultado de la búsqueda deberá ser algo parecido a lo que muestra la Ilustración 30:

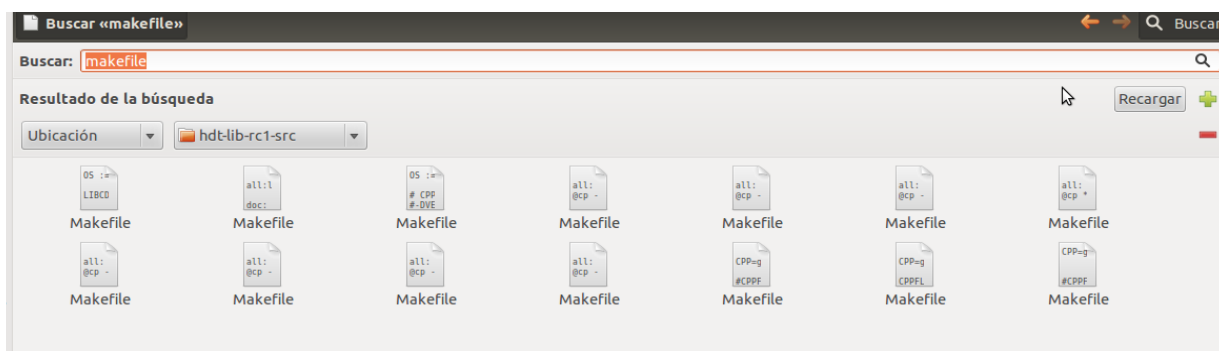


Ilustración 30: Resultado de la búsqueda de los makefiles

Ahora que tenemos localizados todos los ficheros, los abrimos, y en todos aquellos en los que aparezca la variable “`CPPFLAGS`” o “`FLAGS`” le añadimos la opción **-fpic**.

Ahora que ya hemos añadido estas opciones, el siguiente paso es compilar las librerías. En primer lugar tendremos que compilar la librería CDS, una librería adicional que se encuentra dentro del directorio de la librería C++ y de la cual depende. Dentro del directorio de la librería HDT, nos movemos hasta la carpeta *deps*, y a continuación nos metemos en la carpeta *libcds-v1.0.12*. Una vez ahí, ejecutamos las ordenes **make clean** y **make**, con esto tendremos la librería *libcds.a* lista para que pueda ser utilizada por terceros. El siguiente paso sera hacer lo mismo, pero con la librería C++ HDT, así que nos movemos hasta el directorio raíz y ejecutamos las mismas ordenes, una vez hecho esto obtendremos la librería estática *libhdt.a*.

A continuación mostraremos los pasos que hemos seguido para poder completar el proceso de adaptación de la librería C++ HDT a Java.

### 5.4.3. Identificar las clases y métodos de C++

En primer lugar vamos a identificar las clases y métodos de C++ que necesitamos *wrappear* para que la librería pueda ser utilizada desde Java. Estos métodos y clases podemos encontrarlos en las cabeceras de la librería, también conocidas como **API**. Dichas cabeceras se encuentran en la carpeta *include*, y como también nos va a ser necesario el código fuente de algunos de los métodos y clases, utilizaremos la librería *libhdt.a* en la compilación consiguiendo así la implementación.

Hemos identificado las siguientes **cuatro cabeceras**: *HDT*, *HDTManager*, *Iterator* y *SingleTriple*. En dichas cabeceras vienen declaradas diferentes clases y métodos que a continuación especificaremos cuales son los que vamos a tener que *wrappear*:

- En la cabecera **HDT** nos tendremos que centrar en la clase HDT. Es una de las clases más importantes para el desarrollo de la adaptación, dado que es la clase principal de la librería C++ HDT y proporciona un acceso abstracto a un objeto de tipo HDT. En ella encontramos la declaración de un método de tipo *IteratorTripleString*, el cual nos permite realizar una consulta en formato *triple pattern*.
- En la cabecera **HDTManager** solo esta declarada la clase HDTManager. Se trata de otra clase importante, ya que es la encargada de **mapear**, es decir, cargar en memoria principal, un fichero HDT. El método *mapHDT* es mucho más rápido que el de la versión de Java llamado *loadHDT*, además utiliza menos memoria, pero puede resultar un poco más lento en el tiempo de acceso.
- La cabecera **Iterator** contiene diferentes tipos de clases, nosotros nos centraremos en la clase *IteratorTripleString*. Esta clase es la encargada de recorrer el fichero HDT cargado en memoria, en busca de los valores que, dentro de dicho fichero, coincidan con los parámetros introducidos en la búsqueda. Para el desarrollo de este proyecto necesitaremos *wrappear* los métodos *hasNext* y *next*, declarados en esta misma clase.
- Dentro de la cabecera **SingleTriple** deberemos fijarnos en la clase *TripleString*, ya que es la forma de representar la información que obtendremos de la clase *IteratorTripleString*. La clase *IteratorTripleString* nos devolverá un triple que coincida con los parámetros de búsqueda y la clase *TripleString* se encargara de devolver los resultados divididos en **sujeto**, **predicado** y **objeto**.

A continuación mostraremos un resumen del código que contienen las cabeceras, destacando los métodos que vamos a *wrappear*.

#### 5.4.3.1. Cabecera HDT

```
class HDT : public RDFAccess
```



```
{  
  
public:  
  
//Otros métodos  
  
virtual IteratorTripleString *search(const char *subject, const  
char *predicate, const char *object) = 0;  
  
IteratorTripleString *search(TripleString &pattern) {  
  
return search(pattern.getSubject().c_str(),  
pattern.getPredicate().c_str(), pattern.getObject().c_str());  
  
}  
  
//Otros métodos  
  
};
```

#### 5.4.3.2. Cabecera HDTManager

```
class HDTManager {  
  
public:  
  
//Otros métodos  
  
static HDT *mapHDT(const char *file, ProgressListener  
*listener=NULL);  
  
//Otros métodos  
  
};
```

#### 5.4.3.3. Cabecera IteratorTripleString

```
class IteratorTripleString {  
  
public:
```

```
//Otros métodos

virtual bool hasNext() {
return false;
}

virtual TripleString *next() {
return NULL;
}

//Otros métodos
};
```

#### 5.4.3.4. Cabecera TripleString

```
class TripleString
{
private:
std::string subject;
std::string predicate;
std::string object;

public:
//Otros métodos

TripleString(std::string subject, std::string predicate,
std::string object) {
this->subject = subject;
this->predicate = predicate;
```

```
this->object = object;

}

std::string &getSubject() {

return subject;

}

std::string &getPredicate() {

return predicate;

}

std::string &getObject() {

return object;

}

//Otros métodos

};
```

Todas estas cabeceras comparten el espacio de nombres, o *namespace*, **hdt**. Un espacio de nombres es una zona separada donde se pueden declarar y definir objetos, funciones y en general, cualquier identificador de tipo, clase, estructura, etc; al que se asigna un nombre o identificador propio. Son muy útiles puesto que ayudan a evitar problemas con identificadores en grandes proyectos o cuando se usan bibliotecas externas. Nos permite, por ejemplo, que existan objetos o funciones con el mismo nombre, declarados en diferentes ficheros fuente, siempre y cuando se declaren en distintos espacios con nombre.

Otro aspecto que debemos considerar es que algunos de los métodos que vamos a *wrappear* están declarados como *virtual*. La clausula *virtual* vendría a ser un equivalente en C++ a las interfaces de Java. Aunque no nos va a suponer un gran problema, debemos saberlo, ya que una interfaz no contiene la implementación del método.

#### 5.4.4. Diseñar las clases de Java

En esta fase diseñaremos un conjunto de clases en Java que estarán estrechamente relacionadas con los métodos identificados en la etapa anterior. Es decir que estas clases Java incluirán además de sus

propias funciones y variables unos métodos especiales que reciben el nombre de **nativos**. Dichos métodos nativos invocarán, mediante el *wrapper* que crearemos posteriormente, a los métodos que se encuentran implementados en la librería **libhdt.a**.

#### 5.4.4.1. Método de búsqueda

Para este proceso ha sido imprescindible entender como íbamos a realizar la operación de búsqueda, para ello nos hemos apoyado en un método ya implementado, el cual podemos encontrar en el fichero *search.cpp* dentro de la carpeta *examples* del directorio raíz de la librería C++ HDT. El siguiente código corresponde a dicho método de búsqueda:

```
#include <iostream>

#include <HDTManager.hpp>

using namespace std;

using namespace hdt;

int main(int argc, char *argv[]) {

HDT *hdt = HDTManager::mapHDT("data/test.hdt");

IteratorTripleString *it = hdt-
>search("http://example.org/uri3","", "");

while(it->hasNext()){

TripleString *triple = it->next();

cout << "Result: " << triple->getSubject() << ", " << triple-
>getPredicate() << ", " << triple->getObject() << endl;

}

delete it; // Remember to delete iterator to avoid memory leaks!

delete hdt; // Remember to delete instance when no longer needed!

}
```

Como podemos ver, en el método se utilizan todas las clases que identificamos anteriormente. En primer lugar nos topamos con la carga el fichero en memoria principal a través de la clase

## Capítulo 5. Desarrollo del proyecto

**HDTManager**, si nos fijamos bien, cuando se carga el fichero, este devuelve un objeto de tipo **HDT**. Esto deberemos tenerlo en cuenta tanto a la hora de desarrollar el *wrapper* de JNI, como cuando estemos declarando los métodos nativos en la parte de Java.

De igual forma nos pasará a la hora de realizar la búsqueda de un determinado triple sobre el fichero, ya que la consulta devolverá un objeto de tipo **IteratorTripleString** y por tanto deberá ser recogido por un objeto del mismo tipo. Sucederá lo mismo cuando queramos mostrar por pantalla el resultado del triple que hemos encontrado con el iterador.

A continuación vamos a mostrar la el código final de las clases Java que vamos a crear, y al final de todas ellas explicaremos un concepto un tanto abstracto pero vital para que la aplicación funcione correctamente, este concepto del que hablamos recibe el nombre de *handle*.

### 5.4.4.2. Clase HDTManager.java

```
package hdt.proyecto.cpp;

public class HDTManager{

    static {

        System.loadLibrary("hdt-jni");

    }

    private long handler;

    private native long cargarHDTNativo(String file);

    public HDTManager(){} //Constructors sin parametrizar

    public HDT cargarHDT (String file){

        long handle = cargarHDTNativo(file);

        return new HDT(handle);

    }

}
```

Como vemos, al principio del código estamos cargando una librería dinámica, la cual generaremos al final del proceso de adaptación a Java. El motivo de hacer la carga en esta clase y no en otra esta

## Capítulo 5. Desarrollo del proyecto

relacionado con el método de búsqueda que veíamos anteriormente, y es que la primera clase de la que utilizamos un método nativo es HDTManager, concretamente la carga del fichero HDT, por lo tanto es lógico que carguemos aquí dicha librería dinámica.

Seguidamente nos encontramos con la variable *handler*, posteriormente hablaremos más en profundidad de esto, pero ahora mismo podemos decir de ella que va a almacenar una dirección de memoria en la que se podrá encontrar un puntero del lenguaje C++, dicho puntero podrá tener múltiples valores, desde un objeto del tipo *TripleString*, hasta, como es el caso, un fichero HDT cargado en memoria. Por lo tanto esta variable es la encargada de almacenar la información, en formato entendible por Java, de los resultados de la ejecución de los métodos implementados en la librería nativa que obtendremos al final de este proceso de desarrollo. Otro aspecto a destacar es que la variable *handler* debe ser de tipo **long**, esto se debe a que la longitud de las direcciones de memoria entre C++ y Java varía notablemente.

Después de esta variable nos encontramos con la declaración de un método nativo, el motivo de hacerlo aquí es muy simple, pues es la forma que tenemos de decirle al compilador que esta clase contiene métodos cuya implementación se encuentra en otro lugar, en este caso, en una librería nativa. Como vemos el método envía una variable de tipo **String** en su llamada, este String pertenece a la ruta dentro de nuestro dispositivo móvil donde tenemos alojado el conjunto de datos HDT sobre el que vamos a realizar las operaciones de consulta.

Por último nos encontramos con un método cuyo tipo pertenece a la clase HDT, esto es algo extraño, pero si recordamos el método de búsqueda anterior cuando cargábamos un fichero HDT en memoria nos devolvía un objeto de tipo HDT, de ahí que tengamos que hacerlo de esta forma. Por lo tanto cuando creemos un objeto de la clase HDTManager y utilicemos el método *cargarHDT* realmente este llamará a un método nativo *cargarHDTNativo* el cual devolverá a Java un puntero C++, transformado por una clase que veremos posteriormente, y cuya dirección de memoria sera entendible por Java, por lo que podrá ser manipulado por este lenguaje.

### 5.4.4.3. Clase HDT.java

```
package hdt.proyecto.cpp;

public class HDT{

private long handler;

//Una vez que hemos obtenido el handle, podemos establecer los
métodos que queremos utilizar

public native long buscarNativo(String sujeto, String predicado,
String objeto);
```

```
//Sirve para volver a la dirección de memoria inicial donde se
encontraba cargado el fichero HDT

public native void reiniciarHDT();

public HDT(long handle){

handler = handle;//Obtenemos el handle del fichero cargado

}

public IteratorTriple buscar(String sujeto, String predicado,
String objeto){

long handle = buscarNativo(sujeto, predicado, objeto);

return new IteratorTriple(handle);

}

}
```

La única diferencia notable en esta clase respecto a la anterior es el método *reiniciarHDT* cuya función es imprescindible para poder realizar varias búsquedas al conjunto de datos en un mismo ciclo de vida de la aplicación. Debido a que C++ no tiene recolector de basura nos es necesario implementar este método para que realice esta operación, en realidad aquí no estamos eliminando la referencia al puntero de C++ que contiene nuestro fichero HDT cargado, si no algo bastante más complejo. Cuando llamamos a este método en realidad le estamos devolviendo la dirección de memoria inicial en la que se cargó el fichero, porque como podremos comprobar posteriormente, todas las clases implementadas en Java contienen la variable *handler* la cual recibe una dirección de memoria interpretable por Java, pues bien, dado que todas las clases reciben dicha dirección el valor de la misma cambiará y es por esto que debemos devolverle a su valor inicial y de este modo poder realizar varias consultas sobre el mismo conjunto de datos cargado en memoria.

#### 5.4.4.4. Clase IteratorTriple.java

```
package hdt.proyecto.cpp;

public class IteratorTriple{

private long handler;

public native boolean hasNextNativo();
```

```
public native long nextNativo();

public native void eliminarIterator();

public IteratorTriple(long handle) { //Constructor
    handler = handle;
}

public boolean hasNext() {
    boolean var = hasNextNativo();
    return var;
}

public TripleString next() {
    long handle = nextNativo();
    return new TripleString(handle);
}
}
```

De nuevo no encontramos grandes diferencias entre este código y el de la clase anterior, salvo el método *eliminarIterator* que en esta ocasión, cuando se le invoca, si que elimina la dirección de memoria que está almacenando la variable *handle*. También nos fijamos en el método *hasNext* que llama a un método implementado en la librería nativa y el cual devuelve a Java una respuesta booleana, es decir, *true* o *false*, en función del si el conjunto de datos sobre el que se está realizando la consulta contiene o no los parámetros introducidos para realizar la búsqueda.

#### 5.4.4.5. Clase TripleString.java

```
package hdt.proyecto.cpp;

public class TripleString{

    private long handler;
```



```
String sujeto, predicado, objeto;

public native String getSujetoNativo();

public native String getPredicadoNativo();

public native String getObjetoNativo();

//Constructor sin parametrizar

public TripleString() {

    this.sujeto = "";

    this.predicado = "";

    this.objeto = "";

}

public TripleString(long handle) {

    handler = handle;

    //De la siguiente forma, cuando creemos un objeto
    //TripleString de Java, se asignaran los sujetos, predicados
    //y objetos nativos de forma automatica a las variables de java

    this.sujeto = getSujetoNativo();

    this.predicado = getPredicadoNativo();

    this.objeto = getObjetoNativo();

}

public void setSujeto(String sujeto) {

    this.sujeto = sujeto;

}

public String getSujeto() {
```

```
return sujeto;

}

public void setPredicado(String predicado) {

this.predicado = predicado;

}

public String getPredicado() {

return predicado;

}

public void setObjeto(String objeto) {

this.objeto = objeto;

}

public String getObjeto() {

return objeto;

}

}
```

Esta clase no contiene ninguna novedad respecto a las clases anteriores, lo más destacable es que cuando se crea un objeto pasándole una dirección de memoria, que para este caso almacenará un puntero a un objeto de tipo `IteratorTripleString`, es capaz de extraer de dicho iterador los valores del sujeto, el predicado y el objeto. Una vez ha obtenido dicho valor de las llamadas nativas y para evitar posibles pérdidas de información, asigna a variables de tipo `String`, ya en Java, el valor que ha obtenido.

Como se ha ido comentando a lo largo de estas cuatro clases, quién se encarga de la parte crítica para que la aplicación funcione correctamente es la clase *handle*, posteriormente veremos el código de dicha clase, pero por el momento, vamos a profundizar en la explicación sobre como se va a transferir información entre Java y C++.

Para cada una de estas clases desarrolladas en Java, debemos crear una **instancia**, que tendrá que manejar un objeto C++ con su clase correspondiente de la librería nativa. Siempre que llamemos a un método nativo de una de las clases de Java, dicha llamada deberá contener otra llamada a su

método correspondiente en C++. Para hacer esto posible, necesitamos que la instancia de la clase Java que creamos, recuerde que objeto de C++ esta manejando.

Como hemos ido comentando durante el desarrollo de las cuatro clases anteriores, para dicho propósito necesitamos que todas las clases de Java contengan una variable que recibe el nombre de *handler*. Esta variable contendrá un valor que, dependiendo de la clase en donde nos encontremos, significara una cosa u otra. Cuando se realice la llamada a la función escrita en la librería C++, el *handler* almacenará un valor que podrá ser interpretado como un puntero a un objeto C++, mientras que en la parte desarrollada en Java, cuando se llame a un método nativo, el puntero que se guardaba en C++, se convertirá en un valor entero capaz de ser interpretado por Java.

Debemos asegurarnos que el *handler* que tenemos en Java sea capaz de almacenar un tipo entero lo suficientemente grande como para poder guardar un puntero de C++, este es el principal motivo que nos lleva a definir como **long** el tipo de la variable, siendo capaz de almacenar hasta 64 bits.

### 5.4.5. Generar la declaración de las funciones de JNI

En este apartado vamos a generar de forma automática las cabeceras de las funciones de nuestro *wrapper*.

Para cada uno de los métodos que hemos declarado como nativos en nuestras clases de Java, necesitaremos una implementación del mismo en el *wrapper* de JNI. El contenido de dicha función estará escrito en C++, y contendrá un identificador único para cada método y clase de Java. Este identificador permitirá a JNI **mapear** correctamente los argumentos de Java y unirlos con los de C++.

Podríamos escribir estas cabeceras manualmente, pero el kit de desarrollo de Java, o JDK, incluye la herramienta *javah*.

**Javah** es un mecanismo que genera de forma automática los encabezados de C necesarios para implementar el código nativo, de aquellos métodos que se hayan declarado como tal, en una determinada clase de Java. El fichero que generaremos tendrá una extensión *.h*, es decir, el de una cabecera escrita en C, y que no deberemos modificar. La forma en la que generaremos dicho fichero variará dependiendo del sistema operativo en el que nos encontremos. Para la creación de dichas cabeceras, es necesario que las clases que acabamos de crear en Java estén compiladas, es decir, que se hayan generado los ficheros con extensión *.class* de Java. Como el desarrollo de la adaptación lo estamos haciendo de forma manual, esta parte la realizaremos mediante línea de comandos, por lo que las ordenes que vamos a utilizar son las siguientes:

```
javac hdt/proyecto/cpp/*.java
```

Donde *hdt/proyecto/cpp/* es el paquete donde se encuentran las clases Java del proyecto.

Y acto seguido pasamos a generar las cabeceras:

```
javah -jni hdt.proyecto.cpp.HDT hdt.proyecto.cpp.HDTManager
hdt.proyecto.cpp.IteratorTriple hdt.proyecto.cpp.TripleString
```

A continuación mostramos un breve ejemplo del código de una de las cabeceras que acabábamos de generar:

```
/* DO NOT EDIT THIS FILE - it is machine generated */

#include <jni.h>

/* Header for class hdt_proyecto_cpp_HDTManager */

#ifndef _Included_hdt_proyecto_cpp_HDTManager
#define _Included_hdt_proyecto_cpp_HDTManager

#ifdef __cplusplus

extern "C" {

#endif

/*
 * Class:      hdt_proyecto_cpp_HDTManager
 * Method:     cargarHDTNativo
 * Signature:  (Ljava/lang/String;)J
 */

JNIEXPORT jlong JNICALL
Java_hdt_proyecto_cpp_HDTManager_cargarHDTNativo
    (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}

#endif
```

```
#endif
```

Como podemos ver al principio del código, se está utilizando la cabecera *jni.h*, la cual podemos encontrarla entre los archivos del JDK y que contiene las definiciones de todas las funciones, tipos y macros que se necesitan para implementar un método nativo. Dos de las definiciones más importantes que contiene son *#define* y *typedef*, estas cláusulas ocultan parte de la complejidad de asignar tipos de Java para tipos nativos.

Un aspecto que cabe destacar es el uso de la condición *extern "C"*, que indica al compilador que la implementación del método declarado en la cabecera será con código C++. Java utiliza esta condición de forma automática, pero si se diera algún tipo de error en la compilación o el durante el uso de algún tipo de método, es en lo primero que nos tendríamos que fijar.

El modificador **JNIEXPORT**, realiza una tarea muy importante, y es que se encarga de hacer que las funciones nativas aparezcan en la tabla dinámica que se crea al construir la librería nativa. Sin el uso de esta palabra clave, JNI no será capaz de encontrar la implementación de los métodos nativos, y por lo tanto, se generará un error en tiempo de ejecución.

La palabra clave **JNICALL** indica que se trata de una llamada al método escrito en la clase *HDTManager*, la cual se encuentra en el paquete *hdt/proyecto/cpp*. El paso de parámetros mediante *JNIEnv* y *jobject*, es otro de los aspectos clave en JNI, a continuación explicamos para que sirven estos parámetros:

- *JNIEnv* → Es el puntero que utiliza JNI para permitir a los métodos nativos el uso de sus funcionalidades. El puntero *JNIEnv* está asociado a una **estructura** que almacena **información** propia del **hilo** de **ejecución** donde se están efectuando las operaciones. Mediante el uso de este puntero la máquina virtual nos garantiza que se pasará el mismo puntero a todos los métodos nativos que realicen llamadas durante el ciclo de vida del hilo, esto hace que los métodos nativos puedan usar el puntero *JNIEnv* como un **ID** que permanece único mientras el hilo exista.
- *jobject* → Se trata de un argumento cuyo valor varía dependiendo de si es un método de instancia o un método de clase (estático). Si es un método de instancia, actuará como un puntero *this* al objeto Java. Por el contrario si es un método de clase, ejercerá como referencia *jclass* a un objeto que representa la clase en la cual están definidos los métodos estáticos.

La Ilustración 31 muestra el funcionamiento de la variable *JNIEnv*:

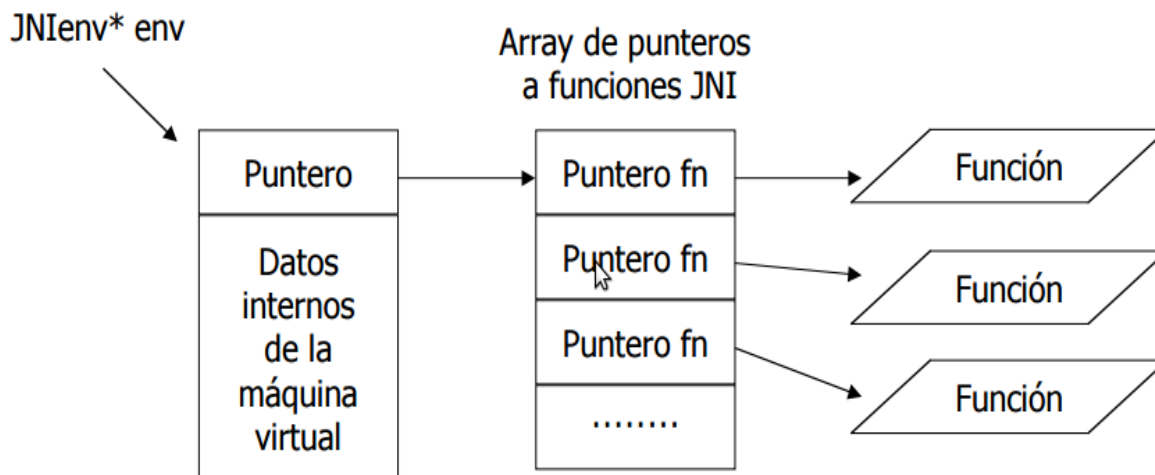


Ilustración 31: Funcionamiento de JNIEnv

#### 5.4.6. Escribir el cuerpo de las funciones de JNI

En este apartado vamos a escribir el cuerpo del *wrapper* de JNI. En el implementaremos cada método y convertiremos los argumentos de Java a argumentos de C++, quienes llamarán a la librería dinámica que generaremos posteriormente y que será la encargada de devolvernos los resultados.

En primer lugar mostraremos la implementación de la cabecera encargada de recoger y establecer los *handler* que hemos establecido en cada clase de Java. El principal objetivo de esto es que podamos hacer un seguimiento del objeto nativo correspondiente a cada objeto Java. Posteriormente mostraremos el código de todas las clases que albergan la implementación de los métodos generados por *javah*, y que formarán el *wrapper* de nuestra aplicación. El nombre de dichas clases es *HDTManagerW*, *HDTw*, *HDTiteratorW* y *HDTtripleW*, todas ellas con extensión *cpp* y con la letra W en su nombre. Esto último para que puedan ser identificadas rápidamente.

##### 5.4.6.1. Cabecera handle.h

```
#ifndef _HANDLE_H_INCLUDED_
#define _HANDLE_H_INCLUDED_

jfieldID static getHandleField(JNIEnv *env, jobject obj)

{
```

```
jclass c = env->GetObjectClass(obj);

// "J" es el tipo de asignación para long:

return env->GetFieldID(c, "handler", "J");

}

template <typename T>

T *getHandle(JNIEnv *env, jobject obj)

{

jlong handle = env->GetLongField(obj, getHandleField(env, obj));

return reinterpret_cast<T *>(handle);

}

template <typename T>

void setHandle(JNIEnv *env, jobject obj, T *t)

{

jlong handle = reinterpret_cast<jlong>(t);

env->SetLongField(obj, getHandleField(env, obj), handle);

}

#endif
```

De esta clase debemos destacar tanto el método *setHandle*, como el método *getHandleField*. Este último es una función estática cuyo tipo pertenece a *jfieldID*, y el cual es usado únicamente por los métodos *getHandle* y *setHandle* declarados en esta misma cabecera. El tipo *jfieldID* permite devolver el identificador de una instancia de una determinada clase, es decir, que nos permite obtener el identificador único que le proporciona JNIEnv a una determinada variable de una clase. Este método básicamente obtiene la posición de memoria que esta ocupando la variable *handler* que declaramos en pasos anteriores en nuestras clases Java y le asocia un identificador único. Cuando el método *setHandle* sea llamado desde alguno de los métodos de las clases del *wrapper*, este recibirá como parámetros los punteros JNIEnv y jobject, pero el que realmente nos interesará será el último parámetro que recibirá este método, pues será un objeto creado en el propio *wrapper* y cuyo valor será el resultado de la llamada a una función de la **librería nativa**. Si nos fijamos en el código de la

## Capítulo 5. Desarrollo del proyecto

cabecera *handle*, cuando el método *setHandle* recibe dicho objeto lo **reinterpreta** mediante la clausula *reinterpret\_cast* y se lo asocia a la variable recogida por el método *getHandleField*, quien, como broche final, se lo devuelve a Java.

El método *getHandle* obtiene el valor de la variable *handler* de una determinada clase de Java, lo reinterpreta de forma que sea entendible por C++, es decir, que convierte su valor a un puntero y lo utiliza para poder seguir haciendo operaciones a través de los métodos implementados en el *wrapper*.

### 5.4.6.2. Implementación de HDTManagerW.cpp

```
#include "hdt_proyecto_cpp_HDTManager.h"//Cabecera generada por
javah

#include <HDTManager.hpp>

#include <HDT.hpp>

#include "handle.h"

#include <iostream>

using namespace hdt;

using namespace std;

HDT *hdt1;

HDTManager *inst;
```

Como podemos ver, la primera línea del código hace referencia a la cabecera que contiene las funciones que hemos generado anteriormente mediante *javah*. Las siguientes líneas también incluyen otras cabeceras, pero en este caso pertenecen a la API de la librería *libhdt.a*. También debemos incluir la cabecera “handle” para poder hacer uso de sus métodos.

Otro aspecto a tener en cuenta es el uso de los *namespaces* propios de la librería *libhdt.a* y que nos evitan problemas con los identificadores.

Por último vemos la declaración de dos objetos propios de la clase del *wrapper*. Los declaramos de forma global para sí, por ejemplo, llamásemos al método en repetidas ocasiones no tendría que volver a cargarse el fichero en memoria.



## Capítulo 5. Desarrollo del proyecto

El siguiente paso es incluir los prototipos de las cabeceras e implementarlos. El método *cargarHDTNativo* quedará de la siguiente forma:

```
jlong  
  
Java_hdt_proyecto_cpp_HDTManager_cargarHDTNativo (JNIEnv *env,  
jobject obj, jstring file){  
  
    const char *kstr = env->GetStringUTFChars(file, 0);  
  
    hdt1 = inst->mapHDT(kstr);  
  
    setHandle(env, obj, hdt1);  
  
    return (jlong)hdt1;  
  
}
```

Como ya sabemos el método debe ser de tipo *long* para que Java sea capaz interpretar el valor de la dirección de memoria donde está almacenado.

También nos es necesario transformar el *String* que recibimos de Java, en nuestro caso, el *String* contiene la ruta del dispositivo móvil donde se encuentra el fichero HDT. A través del uso del método *GetStringUTFChars* transformaremos dicho *String* a un conjunto de caracteres que C++ será capaz de interpretar.

Una vez transformado el *String* el siguiente paso es cargar el fichero mediante la llamada a la función encargada de ello en la librería nativa, en este caso se trata del método *mapHDT* situado en la clase *HDTManager*, y que nos devolverá un objeto de tipo *HDT*.

A continuación lo que tendremos que hacer es, almacenar el valor del puntero C++ que contiene el fichero cargado, en el objeto de Java que ha llamado a este método, para ello invocamos al método *setHandle*.

### 5.4.6.3. Implementación de HDTw.cpp

```
#include "hdt_proyecto_cpp_HDT.h"//Cabecera generada por javah  
  
#include <HDT.hpp>  
  
#include <HDTManager.hpp>  
  
#include <iostream>
```

```
#include "handle.h"

using namespace std;

using namespace hdt;

HDT *hdt2;

IteratorTripleString *it

jlong

Java_hdt_proyecto_cpp_HDT_buscarNativo(JNIEnv *env, jobject obj,
jstring sujeto, jstring predicado, jstring objeto)

{

hdt2 = getHandle<HDT>(env,obj);

const char *ksuj = env->GetStringUTFChars(sujeto, 0);

const char *kpre = env->GetStringUTFChars(predicado, 0);

const char *kobj = env->GetStringUTFChars(objeto, 0);

it = hdt2->search(ksuj,kpre,kobj);

setHandle(env,obj,it);

return (jlong)it;

}

void

Java_hdt_proyecto_cpp_HDT_reiniciarHDT (JNIEnv *env, jobject obj){

setHandle(env, obj, hdt2);

it = NULL;

}
```

## Capítulo 5. Desarrollo del proyecto

Una de las diferencias más significativas respecto al caso anterior es que estamos llamando al método *getHandle*, que como sabemos, recogemos el valor de la variable establecida en Java anteriormente. También hay que destacar que el método *buscarNativo* recibe por parámetro unos *Strings* de Java que en este caso serán los parámetros introducidos por el usuario para realizar la búsqueda sobre el conjunto de datos, al igual que sucede con el método que vimos anteriormente, C++ no puede interpretar el tipo *String*, por lo que es necesario que transformemos las cadenas de caracteres. Una vez hecho esto, realizamos la búsqueda en formato de *triple pattern* y volvemos a realizar un *setHandle*.

La otra principal diferencia es el método *reiniciarHDT*, el uso de este método se debe a que C++ a diferencia de Java no tiene un recolector de basura automático y por lo tanto cada vez que queramos volver a realizar una búsqueda sobre el conjunto de datos cargado en memoria, aunque eliminemos los iteradores, como veremos posteriormente, es necesario guardar la dirección que ocupa el fichero HDT durante todo el ciclo de vida de la aplicación, es por ello que antes de poner a *NULL* el puntero de la clase *IteratorTripleString*, hagamos un *setHandle* del objeto *hdt2*, el cual volverá a obtener la dirección de memoria inicial donde se cargo el fichero HDT.

### 5.4.6.3. Implementación de HDTiteratorW.cpp

```
#include "hdt_proyecto_cpp_IteratorTriple.h"//Cabecera generada
por javah

#include <vector>

#include <string>

#include <fstream>

#include <HDT.hpp>

#include <HDTManager.hpp>

#include <SingleTriple.hpp>

#include "handle.h"

using namespace hdt;

IteratorTripleString *it2;

TripleString *triple;
```

```
jboolean  
  
Java_hdt_proyecto_cpp_IteratorTriple_hasNextNativo (JNIEnv *env,  
jobject obj){  
  
    it2 = getHandle<IteratorTripleString>(env,obj);  
  
    return it2->hasNext();  
  
}  
  
jlong  
  
Java_hdt_proyecto_cpp_IteratorTriple_nextNativo (JNIEnv *env,  
jobject obj){  
  
    triple = it2->next();  
  
    return (jlong)triple;  
  
}  
  
void  
  
Java_hdt_proyecto_cpp_IteratorTriple_eliminarIterator (JNIEnv  
*env, jobject obj){  
  
    it2 = NULL;  
  
}
```

Como vemos, el método *eliminarIterator*, a diferencia del método visto en la implementación anterior, pone a *NULL* la dirección de memoria a la que apunta el puntero, sin utilizar previamente el método *setHandle*, por lo que una vez ha terminado de realizar la búsqueda, elimina dicho puntero y se encuentra preparado para recibir el siguiente.

#### 5.4.6.4. Implementación de HDTtripleW.cpp

```
#include "hdt_proyecto_cpp_TripleString.h"//Cabecera generada por  
javah  
  
#include <vector>
```

```
#include <string>

#include <fstream>

#include <HDT.hpp>

#include <HDTManager.hpp>

#include <SingleTriple.hpp>

#include "handle.h"

using namespace hdt;

TripleString *triple2;

jstring

Java_hdt_proyecto_cpp_TripleString_getSujetoNativo (JNIEnv *env,
jobject obj){

triple2 = getHandle<TripleString>(env,obj);

return env->NewStringUTF(triple2->getSubject().c_str());

}

jstring

Java_hdt_proyecto_cpp_TripleString_getPredicadoNativo (JNIEnv
*env, jobject obj){

return env->NewStringUTF(triple2->getPredicate().c_str());

}

jstring

Java_hdt_proyecto_cpp_TripleString_getObjetoNativo (JNIEnv *env,
jobject obj){

return env->NewStringUTF(triple2->getObject().c_str());

}
```

## Capítulo 5. Desarrollo del proyecto

De este código podemos destacar que al contrario que nos pasaba cuando recibíamos *Strings* de Java y teníamos que transformarlos a tipos que fuesen legibles por C++, en esta ocasión, tendremos que hacer el procedimiento opuesto y transformar una cadena de caracteres de C++ a **String** y retornarla a Java mediante el método *NewStringUTF*.

Alcanzado este punto ya seríamos capaces de generar una librería nativa que podría ser utilizada a través de una máquina virtual Java, pero como el objetivo de este proyecto es conseguir que dicha librería pueda ser utilizada desde un dispositivo móvil, nos van a ser necesarios una serie de fases adicionales para completar la adaptación de la librería C++ HDT a Android. Como no queremos alargar en exceso esta etapa del proyecto, la adaptación completa para Java se podrá ver en el apartado 10.3 del Anexo.

### 5.4.7. Adaptación a Android

En esta etapa utilizaremos todo el desarrollo realizado en la fase anterior, y apoyándonos sobre diferentes herramientas conseguiremos generar una librería nativa que podrá ser utilizada desde cualquier dispositivo móvil que utilice un procesador ARM. En primer lugar explicaremos cuales son las herramientas que vamos a utilizar y cual es el motivo que nos ha llevado a emplearlas. Además debido a la aplicación de estos nuevos mecanismos, serán necesarios algunos cambios sobre la librería C++ HDT.

#### 5.4.7.1. Preparación del entorno

Entramos de lleno en el mundo del NDK, es decir, en el **desarrollo nativo**, lo primero que necesitamos saber es que el compilador por defecto para C++ de Ubuntu, es decir, el famoso *g++*, no es un mecanismo válido para generar una librería nativa que pueda ser usada desde Android. Por lo tanto, vamos a necesitar un conjunto de herramientas conocido como *standalone toolchains*, las cuales están incluidas en el directorio de instalación del NDK.

Como ya hemos dicho las *standalone toolchains* son un conjunto de herramientas cuyo objetivo es el de compilar diferentes ficheros o librerías de forma que estas puedan ser utilizadas por Android de forma directa. Además ofrecen un soporte completo para las APIs de Android, lo que las dota de una gran flexibilidad. En un principio esta herramienta era sumamente compleja de manejar, y solo las utilizaban los propios desarrolladores de Android, pero con la evolución y las ventajas que aporta el desarrollo nativo, en cada nueva API, se ha ido facilitando el acceso a dichas herramientas, a la par se le han ido añadiendo más funcionalidades, operaciones más complejas, se les han pulido errores y se han incorporado nuevas cabeceras y librerías. Esto hace que, poniendo como ejemplo nuestro proyecto, si tratamos de compilar un proyecto con las *standalone toolchains* pertenecientes a la API 8, es muy probable que las cabeceras incluidas en esa API no tengan el soporte necesario para ejecutar nuestro proyecto en un terminal como a los que actualmente tenemos acceso. Por lo tanto para nuestro proyecto nos hemos decidido por utilizar las cabeceras y librerías que nos ofrece la API 18 de Android.

## Capítulo 5. Desarrollo del proyecto

Otra decisión importante a llevar a cabo a la hora de elegir una determinada *standalone toolchain*, es la elección de la **ABI (Application Binary Interface)**, o interfaz binaria de aplicación, con la que vamos a compilar nuestras librerías.

Una ABI es un **intermediario** entre dos programas, uno de los cuales normalmente es una librería o un sistema operativo, a nivel del lenguaje de máquina. Determina los detalles que deben cumplirse para que una de las partes pueda llamar a funciones de la otra, además también define en que formato binario se va a transferir la información entre dichos componentes.

Nosotros, para el desarrollo de nuestro proyecto hemos elegido la **ABI de ARM**. El motivo de esta elección es muy sencillo, y es que la gran mayoría de terminales móviles utilizan esta gama de microprocesadores, por lo que compilar nuestras librerías con las *toolchains* pertenecientes a esta ABI hará que sean multiplataforma, en lo que se refiere a la capacidad de ser ejecutadas en cualquier terminal que contenga un microprocesador de esta clase.

En resumen, para este proyecto nos hemos decantado por utilizar las *standalone toolchain* de ARM para la API 18. Además de incluir todos los cambios realizados sobre las APIs anteriores a esta, también incluye otras propiedades muy interesantes como el soporte de excepciones de C++ y la posibilidad de utilizar RTTI, o Run-Time Type Information. Esta versión también soporta diferentes librerías que contienen cabeceras necesarias para la compilación de nuestra aplicación.

Ahora que ya hemos entrado un poco en el contexto de las herramientas que nos son necesarias para proseguir con la adaptación del proyecto, mostramos a continuación el comando que vamos a utilizar para extraer la *standalone toolchain* de la API 18.

En primer lugar tendremos que situarnos en el directorio donde hemos descomprimido el NDK, y a continuación, introducir el siguiente comando:

```
build/tools/make-standalone-toolchain.sh --platform=android-18
--system=linux-x86_64 --toolchain=arm-linux-androideabi-4.9
--install-dir=/home/jon/Android/arm-API-18
```

En la opción `-system` debemos indicar el sistema operativo donde estemos ejecutando la instrucción, teniendo especial cuidado a la hora de determinar si nuestro sistema operativo es de 32 bits o de 64.

Con la opción `-install-dir` le indicamos al *script* donde tiene que situar la carpeta de instalación de la herramienta en nuestro sistema.

La opción `-toolchain` nos permite elegir con que versión del compilador queremos trabajar, en este caso hemos elegido la más reciente, es decir, la 4.9.

Otro aspecto a destacar es que estas herramientas incluyen un **SYSROOT** propio, el cual contiene todas la librerías y cabeceras necesarias para compilar nuestro código o librería escrita en C++ en base a la API que hayamos elegido.

## Capítulo 5. Desarrollo del proyecto

Una vez hecho esto, debemos añadir algunas variables al **PATH** para poder hacer uso de las herramientas que acabamos de instalar, editando el fichero *bashrc* como ya se explico, añadimos las siguientes líneas:

```
export PATH=${PATH}:/home/jon/Android/arm-API-18/bin

export CXX=arm-linux-androideabi-g++

export AR=arm-linux-androideabi-ar
```

La primera línea hace referencia a la carpeta *bin* del directorio donde acabamos de instalar la *toolchain*.

Las otras dos variables sirven para poder utilizar los compiladores propios de la herramienta en los *makefiles* de nuestro proyecto.

### 5.4.7.2. Preparación de las librerías

Una vez hemos completado la fase anterior, podemos comenzar a preparar la librería C++ HDT para que pueda ser compilada por las herramientas que hemos visto anteriormente. El motivo de que tengamos que preparar la librería se debe a que las *standalone toolchain* incluyen un soporte muy específico de cabeceras y librerías por lo que hay cabeceras y métodos básicos que no incluyen y que para que puedan ser utilizados en nuestro proyecto tendremos que modificar algunos ficheros. Para poder generar la librería nativa vamos a necesitar compilar tanto la librería *libhdt.a*, como la librería *libcds.a*.

En primer lugar vamos a compilar *libcds.a*, que como ya se especifico anteriormente, *libhdt.a* depende de ella. Al igual que hicimos en el apartado 5.4.2, tenemos que modificar todos los *makefiles* que contiene la librería. Una vez los hemos abierto, en todos aquellos en los que aparezca una variable definida como **CPP** y cuyo valor sea **g++**, deberemos cambiarlo por los compiladores de nuestra *toolchain*, es decir, que el valor de la variable **CPP**, tiene que pasar a ser **arm-linux-androideabi-g++**.

Adicionalmente, necesitamos que en todos aquellos *makefiles* en los que se utilice el comando **ar**, el cual sirve para unificar todos los ficheros objeto generados en la compilación, lo cambiemos por la herramienta propia de la *toolchain*. Por lo tanto, tenemos que modificar el *makefile* y añadirle una variable, es muy sencillo, en una línea vacía introducimos el nombre que tendrá la variable, en nuestro caso **AR**, después de el nombre de la variable introducimos el símbolo igual (=) y le asignamos el valor de la herramienta, **arm-linux-androideabi-ar**.

Una vez hecho esto podemos proceder a tratar de compilar las librerías, como hicimos anteriormente mediante las ordenes *make clean* y *make*.



Durante la compilación nos aparece un problema que, al parecer, proviene de la cabecera *TableOffsetRRR.h*, el mensaje de error es el siguiente:

```
'ushort' does not name a type
```

Este tipo de error suele estar relacionado con la interpretación que está haciendo el compilador sobre el **tipo** de una variable. Pasamos a comprobar la cabecera que nos produce el error y en ella podemos observar que hay varios métodos que utilizan este tipo de simbología, pero no encontramos en ningún lugar del archivo la definición de dichas variables, lo que nos lleva a pensar que el problema podría provenir de alguna cabecera de la que *TableOffsetRRR.h* dependa. Por lo tanto pasamos a comprobar las cabeceras de las que depende, para ello nos situamos en la parte superior de *TableOffsetRRR.h* y vemos como incluye la cabecera *libcdsBasics.h*, por lo que el siguiente paso será examinar esta nueva cabecera.

Una vez que hemos abierto el fichero perteneciente a la cabecera *libcdsBasics.h*, la Ilustración 32 muestra las dos declaraciones de los tipos que nos estaban dando problemas:

```
33 #include <sys/types.h>
34 #include <iostream>
35 #include <fstream>
36 #include <cstdlib>
37 #include <cmath>
38 #include <string>
39 #include <sstream>
40 #include <cassert>
41 #include <stdint.h>
42
43 #ifdef WIN32
44 typedef unsigned int uint;
45 typedef unsigned short ushort;
46 #endif
47
48
49 namespace cds_utils
50 {
51
52     using namespace std;
53     typedef unsigned char uchar;
54
55     /** mask for obtaining the first 5
56     const uint mask31 = 0x0000001F;
```

Ilustración 32: Fichero antes de ser modificado

El uso de la clausula **typedef** trata de hacer la vida un poco más fácil al programador cambiando la longitud de la palabra clave **unsigned short** por **ushort**, algo que a la hora de programar le hará ir un poco más rápido. Sin embargo, dicha definición se encuentra entre unas **condiciones**, las cuales nuestro compilador actual no es capaz de interpretar, a diferencia de g++ que sí puede hacerlo, por lo que sin más rodeos, para subsanar este error, bastará con eliminar las condiciones entre las que se encuentra la declaración de los **typedef**. La Ilustración 33 muestra el aspecto final del archivo:

```

33 #include <sys/types.h>
34 #include <iostream>
35 #include <fstream>
36 #include <cstdlib>
37 #include <cmath>
38 #include <string>
39 #include <sstream>
40 #include <cassert>
41 #include <stdint.h>
42
43
44 typedef unsigned int uint;
45 typedef unsigned short ushort;
46
47
48
49 namespace cds_utils
50 {
51
52     using namespace std;
53     typedef unsigned char uchar;
54
55     /** mask for obtaining the first 5
56     const uint mask31 = 0x0000001F;

```

Ilustración 33: Fichero después de ser modificado

Una vez completados estos cambios, volvemos a tratar de compilar la librería, en esta ocasión aparecerán varios mensajes de aviso, o *warnings*, de los que no debemos preocuparnos en exceso, puesto que no afectarán al correcto funcionamiento de nuestra librería. Cuando se complete la compilación, ya tendremos nuestra librería *libcds.a* lista para ser utilizada desde un procesador ARM, y también para formar parte de una librería nativa que será utilizada desde un dispositivo móvil. Ahora que hemos completado la adaptación de la librería CDS, pasamos a adaptar *libhdt.a*.

De igual forma que hemos hecho anteriormente, modificamos el *makefile* cambiando la variable *CPP* y añadiendo la variable *AR*. A continuación, procedemos a compilar, y de nuevo nos surge un error, esta vez relacionado con la librería **Bionic**. Antes de pasar a analizar el error, hay que destacar que la librería *Bionic* es la librería estándar de C desarrollada por Google para el sistema operativo Android, la cual incluye otras librerías como *libc*, *libdl*, *libm*, y *libpthread*.

Continuando con el error, el mensaje que nos da el compilador es el siguiente:

```

ctype_base.h: error: #error Bionic header ctype.h does not define
either _U nor _CTYPE_U

```

En primer lugar vamos a comprobar de que no se trate de el mismo problema que nos surgió anteriormente, por lo que, nos dirigimos a la carpeta donde tenemos instalada la *standalone toolchain* y procedemos a revisar la cabecera *ctype\_base.h*. Una vez la hemos analizado por completo, no observamos error alguno, lo que nos lleva a pensar que el problema pueda deberse a que hayamos descrito erróneamente en el *makefile* la carpeta donde el compilador tiene que buscar la librería **Bionic**. Ciertamente en el *makefile* estábamos haciendo uso de las cabeceras y las librerías que utiliza el compilador de *g++*, la cuales no son validas para compilar con nuestra *standalone toolchain*, por lo que eliminaremos del *makefile* los valores de las variables **INCLUDES** y **LIB** que contengan las cabeceras y librerías que están provocando el problema. A continuación

mostramos el resultado final de los ficheros, la Ilustración 34 pertenece al *makefile* inicial y la Ilustración 35 es el resultado del *makefile* que hemos modificado para hacer uso de las *standalone toolchain*.

```
9
10 CPP=g++
11 FLAGS=-fPIC -O3 -Wno-deprecated
12 INCLUDES=-I $(LIBCDS_PATH)/includes/ -I /usr/local/include -I ./include -I /opt/local/include -I /usr/include
13 LDFLAGS=
14 DOXYGEN=doxygen
15 DEFINES=
16 LIB=$(LIBCDS_PATH)/lib/libcdfs.a -L/usr/local/lib -lstdc++
17
18 ifeq ($(RAPTOR_SUPPORT), true)
19 DEFINES:=$(DEFINES) -DUSE_RAPTOR
20 LIB:=$(LIB) -lraptor2
21 endif
22
23 ifeq ($(KYOTO_SUPPORT), true)
24 DEFINES:=$(DEFINES) -DUSE_KYOTO
25 LIB:=$(LIB) -lkyotocabinet
26 endif
27
28 ifeq ($(LIBZ_SUPPORT), true)
29 DEFINES:=$(DEFINES) -DUSE_LIBZ
```

Ilustración 34: Cabecera *ctype\_base.h* antes de la modificación

```
9
10 CPP=arm-linux-androideabi-g++
11 AR=arm-linux-androideabi-ar
12 FLAGS=-fPIC -O3 -Wno-deprecated
13 INCLUDES=-I $(LIBCDS_PATH)/includes/ -I ./include
14 LDFLAGS=
15 DOXYGEN=doxygen
16 DEFINES=
17 LIB=$(LIBCDS_PATH)/lib/libcdfs.a -lstdc++
18
19 ifeq ($(RAPTOR_SUPPORT), true)
20 DEFINES:=$(DEFINES) -DUSE_RAPTOR
21 LIB:=$(LIB) -lraptor2
22 endif
23
24 ifeq ($(KYOTO_SUPPORT), true)
25 DEFINES:=$(DEFINES) -DUSE_KYOTO
26 LIB:=$(LIB) -lkyotocabinet
27 endif
28
29 ifeq ($(LIBZ_SUPPORT), true)
```

Ilustración 35: Cabecera *ctype\_base.h* después de la modificación

Una vez hechos estos cambios volvemos a tratar de compilar la librería, pero de nuevo nos aparece otro error, esta vez más parecido a que nos surgió con la compilación de *libcdfs.a*. El mensaje que nos aparece es el siguiente:

```
src/triples/TripleListDisk.cpp: error: 'mkostemp' was not declared
in this scope
```

Para solucionarlo vamos a revisar la clase *TripleListDisk.cpp*. Examinando el fichero, podemos observar como de nuevo aparecen las condiciones que vimos en el primer error, por lo que probaremos a borrarlas como hicimos anteriormente y volveremos a compilar.

Sorprendentemente, el error persiste, lo que nos va a obligar a eliminar una línea del código responsable del error. Decidir eliminar una línea de un archivo que pertenece a un proyecto tan grande puede traer consigo consecuencias muy perjudiciales, por lo que debemos revisar por completo si la eliminación de esa línea afectaría a otras partes de la librería *libhdt.a*. Una vez hemos

## Capítulo 5. Desarrollo del proyecto

comprobado que borrar esa línea no va a traer consigo mayores problemas, procedemos a eliminarla. El trozo de código en cuestión es el siguiente: *fd = mkostemp*. Cabe destacar que antes de optar por esta solución, probamos diferentes opciones, y el resultado siempre era el mismo error. El estado inicial del fichero se muestra en la Ilustración 36, mientras que el resultado final pertenece a la Ilustración 37.

```
61     v.assign( s.begin(), s.end() );
62
63 #ifdef __APPLE__
64     fd = mkstemp(&v[0]);
65 #elif defined(WIN32)
66
67 #else
68     fd = mkostemp(&v[0], O_RDWR | O_CREAT | O_TRUNC);
69 #endif
70
71     if (fd == -1) {
72         perror("Error open");
73         throw "Error open";
74     }
75
76     fileName.assign( &v[0] );
77
```

Ilustración 36: Clase TripleListDisk.cpp antes

```
59     std::string s("triplelistdiskXXXXXX");
60     std::vector<char> v(100);
61     v.assign( s.begin(), s.end() );
62
63
64     fd = mkstemp(&v[0]);
65
66
67     if (fd == -1) {
68         perror("Error open");
69         throw "Error open";
70     }
71
72     fileName.assign( &v[0] );
73
74     cout << "TripleListDisk: " <<fileName << endl;
75     cout << "\t: " <<&v[0] << endl;
76     cout << "\t: " <<s << endl;
77
```

Ilustración 37: Clase TripleListDisk.cpp después

Una vez lo hemos eliminado, la librería se compilará correctamente, por lo que con este último paso ya tendremos ambas librerías compiladas y preparadas para ser utilizadas en Android, a través eso si de una librería nativa. El desarrollo de esta librería nativa lo afrontaremos en la siguiente etapa.

### 5.4.7.3. Creación de la librería nativa

Llegados a esta fase del proceso estamos listos para generar la librería nativa que utilizaremos en el dispositivo móvil.

En primer lugar crearemos un archivo *makefile* en el que tendremos que las diferentes carpetas y archivos donde se encuentran los códigos necesarios para formar la librería nativa. La Ilustración 38 muestra el estado final del *makefile*:

```

1 |
2 |
3 | CXX=arm-linux-androideabi-g++
4 |
5 | LIBRARY := libhdt-jni.so
6 | OBJFILES := src/HDTManagerW.o src/HDTw.o src/HDTiteratorW.o src/HDTtripleW.o
7 | INCLUDES := -I"/usr/lib/jvm/java-7-openjdk-amd64/include" -I"/usr/lib/jvm/java-7-openjdk-amd64/include/linux/" -I./hdt-lib/includes/
8 | CXXFLAGS := -fPIC -O3 $(INCLUDES)
9 | LIB=./hdt-lib/lib/libhdt.a ./hdt-lib/lib/libcdfs.a
10 |
11 | $(LIBRARY): $(OBJFILES)
12 |     $(CXX) -shared -o $@ $^ ${LIB}
13 |
14 | clean:
15 |     @echo " [CLN] Removing object files"
16 |     @rm -f $(OBJFILES) src/*~ *~

```

Ilustración 38: Makefile para generar la librería nativa con las toolchain de ARM

La variable **CXX** indica el compilador que utilizaremos, como podemos observar se corresponde con el utilizado en la etapa anterior. La variable **LIBRARY** sirve para definir el nombre que tendrá nuestra librería nativa, en nuestro caso hemos elegido **libhdt-jni.so**. La variable **OBJFILES** deberá indicar los *object files* producto de la compilación de las clases del *wrapper*, estos archivos contienen **metadatos** muy importantes para el *linkado* y el *debug* final de la librería. La clausula **INCLUDES** le dice al compilador el lugar donde debe buscar las cabeceras necesarias para compilar la librería. **CXXFLAGS** indica diferentes condiciones a tener en cuenta durante el proceso de compilación. La variable **LIB** le dice al compilador de que librerías depende la nueva librería nativa. Por último le indicaremos al compilador que queremos que nuestra librería nativa sea dinámica, para ello, utilizaremos la clausula *shared*.

Una creado este *makefile*, estaremos listos para generar nuestra librería nativa, ejecutamos las ordenes *make clean* y *make* y ya tendremos la librería lista para ser utilizada en Android.

#### 5.4.7.4. Proyecto HDT-Cpp

Ahora que ya tenemos nuestra librería nativa preparada, solo nos queda crear un nuevo proyecto en Eclipse, añadirle las clases Java desarrolladas en el apartado 5.4.4 y la librería nativa. Debemos prestar especial atención al nombre del paquete que le damos a este nuevo proyecto pues debe coincidir exactamente con el nombre del paquete que le dimos en la adaptación a Java, si recordamos, el nombre del paquete del proyecto es **hdt.proyecto.cpp**, por lo que a este nuevo proyecto tendremos que darle el mismo nombre. Hay un problema asociado a esto, y es que, si llamamos a métodos nativos desde clases que se encuentren en un paquete diferente a este, la librería nativa no será capaz de encontrar la variable *handler* y por tanto se generara un error en tiempo de ejecución, este es el principal motivo de que este proyecto, a diferencia del desarrollado en Java, solamente contenga un único paquete.

El siguiente paso será copiar las clases con extensión *.java* que ya tenemos desarrolladas a este nuevo proyecto, algo tan sencillo como copiar y pegar, salvo con la clase *TripleString.java*, a la que tendremos que añadir la variable *id* y los *getters* y *setters* específicos para ella, al igual que hicimos con el proyecto Java, salvo que en este caso será mucho más sencillo puesto que solo tenemos que

## Capítulo 5. Desarrollo del proyecto

modificar una clase Java y en el otro caso tuvimos que modificar un archivo `.class` que se encontraba dentro de una librería.

Una vez copiadas las clases Java al proyecto HDT-Cpp, tendremos que añadir la opción de desarrollo nativo, que incluye el NDK, a nuestro nuevo proyecto Android. Esta herramienta nos facilita el comienzo de esta fase, ya que crea de forma automática un directorio al que asigna el nombre de `jni`, el cual contendrá diferentes ficheros necesarios para implementar la parte nativa del proyecto. Dentro de ese mismo directorio también genera el `makefile` propio de un proyecto Android que incluye código nativo, dicho `makefile` recibe el nombre de **`Android.mk`**. Este fichero sirve para describir qué otros ficheros, códigos y librerías va a necesitar nuestra librería nativa para ser construida. Es realmente un pequeño `makefile` como los que vistos anteriormente, pero específico para el desarrollo nativo en Android. La particularidad de este `makefile` es que se divide en **módulos**, y en cada uno de ellos se deben incluir diversas variables que indicaran al compilador que partes debe tener en cuenta para generar la librería nativa.

Continuando con el uso del NDK en nuestro proyecto, para poder hacer uso de la utilidad que ofrece, tenemos que hacer click derecho sobre la carpeta raíz del proyecto, seleccionar la opción **`Android Tools`**, y a continuación clickar sobre la opción **`Add Native Support`**. Hecho esto, aparece una ventana en la que debemos indicar el nombre que le damos a nuestra librería nativa, en nuestro caso no importará el nombre que le demos puesto que ya hemos generado nuestra propia librería.

Una vez Hecho esto, se generan automáticamente un fichero con extensión `.cpp` y una librería nativa, procedemos a eliminarlos y copiamos nuestra librería nativa `libhdt-jni.so`, a la carpeta `jni`. Para que nuestro proyecto haga uso de nuestra librería nativa nos va a ser necesario modificar el fichero **`Android.mk`**. Para nuestro caso particular, tendremos que definir un **modulo** específico en el que le digamos al compilador del NDK, conocido como **`ndk-build`**, que la librería que vamos a utilizar en la aplicación ya está preparada y **no necesita ser creada ni recompilada**. Para decirle esto al compilador vamos a hacer uso de la cláusula **`Prebuilt Shared Library`**, propia de los **`Android.mk`**.

Una **librería preconstruida**, a ojos del NDK, es una librería que ha sido compilada con las `standalone toolchains` que el mismo proporciona, como es nuestro caso. El uso de librerías preconstruidas tiene dos grandes beneficios:

- Podemos distribuir las librerías a otros desarrolladores, sin necesidad de distribuir nuestro código.
- El uso de versiones preconstruidas acelera el proceso de desarrollo de la aplicación.

Ahora que ya vemos como podemos incluir nuestra librería compartida en el proyecto, pasamos a mostrar el resultado final del **`Android.mk`**:

```
LOCAL_PATH := $(call my-dir)
```

```
include $(CLEAR_VARS)
```

```
LOCAL_MODULE := hdt

LOCAL_SRC_FILES := libhdt-jni.so

include $(PREBUILT_SHARED_LIBRARY)
```

Donde el aspecto más destacable de este fichero es la variable **LOCAL\_SRC\_FILES**, que como vemos, apunta directamente a nuestra librería nativa que se encuentra en el directorio *jni*. Gracias a este modulo conseguimos que las herramientas del NDK indiquen a la aplicación Android que debe utilizar la librería nativa durante su ejecución cuando esta sea requerida.

Nos va a ser necesario un paso adicional antes de terminar con este proceso, vamos a tener que crear un nuevo archivo, el cual recibe el nombre de **Application.mk**, que además debe situarse en la misma carpeta en la que se encuentre el *Android.mk*.

Este nuevo archivo sirve principalmente para ayudarnos a establecer diversas opciones que afectarán a todos los módulos que hallamos descrito en el *Android.mk*. En nuestro caso vamos a definir en el *Application.mk* la variable **APP\_ABI**, la cual nos permitirá definir la interfaz binaria de aplicación en la que va ser posible utilizar nuestra librería nativa, y por lo tanto, los procesadores en los que se podrá ejecutar la aplicación HDT-Cpp. En nuestro caso los valores que toma dicha variable son **armeabi**, y **armeabi-v7a**.

Antes de poder ejecutar este nuevo proyecto en un terminal móvil, tendríamos que repetir la etapa 5.3.4, en la que implementábamos todas las clases del proyecto Java, afortunadamente, como ya tenemos esas clases desarrolladas únicamente tendremos que copiarlas a este nuevo proyecto y ajustarlas de forma que se adapten al nuevo nombre del paquete y que las clases sean capaces de utilizar la librería nativa que hemos desarrollado. Por ejemplo, en este proyecto tendremos que cargar la librería nativa en el primer *Activity* que se ejecute en el terminal, por lo tanto, el *la* clase **Pantalla\_inicio** deberá incluir el código correspondiente para realizar dicha carga. El código al que nos referimos es el siguiente:

```
static {

System.loadLibrary("hdt-jni");

}
```

Tampoco debemos olvidarnos de gestionar la basura, cuando llamemos a métodos nativos implementados en nuestra librería, mediante los métodos *reiniciarHDT* y *eliminarIterador*.

Una vez hallamos añadido el resto de las clases del proyecto HDT-Java a este, estaremos listos para ejecutar la aplicación HDT-Cpp en un dispositivo móvil.





## **Capítulo 6. Benchmarking**



## 6.1. Introducción

Uno de los principales motivos que nos ha llevado a realizar esta parte del proyecto es que actualmente en el mundo de Internet es muy complicado encontrar un *benchmark* que utilice variables relacionadas con el consumo, puesto que prácticamente todos tienen que ver con la velocidad y el rendimiento de los lenguajes sobre los que se realizan las pruebas. Por ello, ante una oportunidad de ser pioneros en lo que a las comparativas se refiere, hemos decidido realizar un estudio del consumo de energía de ambas aplicaciones realizando una serie de consultas a nuestro conjunto de datos **LinkedMDB**. Este *benchmark* no solo servirá de precedente en el mundo de los dispositivos móviles, si no que también ofrecerá una imagen de cuanto de bueno es utilizar el formato HDT en la tecnología móvil de hoy en día.

El *benchmark* es una técnica utilizada para medir el rendimiento de un sistema o un componente del mismo. Formalmente puede entenderse que un *benchmark* es el resultado de la ejecución de un programa informático o un conjunto de programas en una máquina, con el objetivo de estimar el rendimiento de un elemento concreto, y poder comparar los resultados con máquinas similares. Los *benchmarks* se dividen en dos grandes grupos:

- **Benchmarks sintéticos:** ponen a prueba diversos componentes, sometiéndolos a una carga de trabajo elevada para comprobar la velocidad y eficiencia con la que se completan dichas cargas. Estas aplicaciones no van más allá de su objetivo, el de producir un valor numérico que es comparable a otro tomado anteriormente. Su ventaja es clara: están orientados a componentes específicos, y **son fácilmente replicables**. Por ejemplo, un *benchmark sintético* obtendrá un valor idéntico o muy similar, si no hay servicios o aplicaciones en segundo plano que influyan en el rendimiento, en dos smartphones iguales, y por esa misma razón son una herramienta muy válida para establecer referencias de rendimiento.
- **Benchmarks de aplicación:** estas pruebas se ejecutan con aplicaciones reales que los usuarios utilizan en su día a día y que permiten comprobar el rendimiento de una plataforma con esa aplicación en concreto. La codificación de vídeo, la ejecución de código JavaScript de los navegadores web, el renderizado de imágenes en 3D son algunos ejemplos de este tipo de *benchmark*. El problema es que los valores de salida de estas pruebas pueden verse **afectados por numerosos parámetros** al comprobarse el rendimiento de plataformas en conjunto y no de componentes particulares.

En este proyecto vamos a realizar un *benchmark de aplicación*, es decir, pondremos a prueba el consumo de energía de un *smartphone* con una carga de trabajo muy elevada, concretamente mediremos el consumo de energía que tiene realizar un lote de 500 consultas, cuyo número de resultados variará dependiendo del tipo de lote que se ejecute, y repetiremos este proceso hasta acabar la batería del terminal móvil. Iremos anotando el número de veces que hemos repetido las consultas al mismo tiempo que controlamos el consumo de energía.

## 6.2. Dispositivo móvil y cláusulas del benchmark

El *smartphone* con el que vamos a realizar esta comparativa, incorporará ambas versiones de la aplicación que hemos desarrollado en este proyecto, y como cabría esperar, el teléfono móvil incorporará el sistema operativo Android.

El terminal móvil donde vamos a realizar las pruebas es el modelo **Samsung Galaxy Trend GT-S7560**, que aparece en la Ilustración 39. A continuación mostraremos algunas de sus características, las cuales las hemos obtenido de la pagina oficial de Samsung:

- La **versión de Android** que esta instalada en el terminal es la **4.0.4**, la cual pertenece a la **API 15** y que recibe el nombre de ***Ice Cream Sandwich***.
- La batería del dispositivo es de Li-Ion con una capacidad de 1500 mAh, lo que le permite tener una autonomía de unas 30 horas de llamada.
- El **procesador** es de un único núcleo con una velocidad de **1 GHz**, y posee una memoria **RAM de 768 MB**.



Ilustración 39: Dispositivo móvil

Hay dos aspectos que debemos destacar de este *smartphone*. En primer lugar, el teléfono ha sido “reseteado” y por lo tanto, tiene los valores por defecto de fábrica. En segundo lugar, la marca Samsung incluye en sus dispositivos un modo de **ahorro de energía automático** que hace que cuando el terminal móvil llegue al 5% de batería entre en modo economizador, este aspecto es muy importante puesto que nos condicionará hasta el punto en el que vamos a poder medir el consumo de batería, ya que al entrar en el modo de ahorro de energía el dispositivo establece el brillo de la pantalla al mínimo, y por lo tanto los resultados que obtendríamos desde ese 5% de batería hasta

## Capítulo 6. Benchmarking

agotarla variarían enormemente respecto a los valores tomados con el brillo de la pantalla al máximo. Por lo tanto, mediremos el consumo de energía de las consultas hasta que el *smartphone* llegue al 5% de su capacidad de batería. Dicho esto, a continuación vamos a establecer las condiciones en la que se encontrará el terminal móvil a la hora de realizar el *benchmark*.

La intención es que el móvil no este ejecutando ningún proceso en segundo plano y de esta forma obtengamos los resultados más reales posibles, para ello vamos a eliminar las siguientes aplicaciones y servicios:

- Conexión a Internet: Redes Wifi y Redes móviles.
- Aplicaciones en segundo plano: Play Store, Google Maps, Reproductor de Música, etc.

Hay algunos procesos en segundo plano que no podemos eliminar, dado que son aplicaciones propias del sistema y quitarlas supondría múltiples errores en el dispositivo. Por ello a continuación mostramos aquellos procesos que si que se encontrarán en ejecución durante el proceso de pruebas:

- Reloj, DSMLawmo, MAPServiceSamsung, SecPhone y Teclado.

En total el dispositivo esta **haciendo uso de 240 MB** de memoria RAM, lo que significa que hay aproximadamente **400 MB libres**.

Por último, para realizar las pruebas, vamos a situar el **brillo de la pantalla al máximo**.

### 6.3. Pruebas

Para nuestro *benchmark* solo vamos a estimar 4 escenarios diferentes, estos se corresponden con los tipos de consulta de los que vamos a realizar las pruebas. Los lotes de consulta sobre los que se va a realizar el estudio pertenecen a los patrones **SPO, SP?, S?O y S??**.

Como ya se ha comentado anteriormente los indicadores que vamos a medir son el consumo de batería tras la ejecución de el lote de consultas, cuyo número de resultados va a variar según el tipo de patrón que ejecutemos, como con la ejecución de un único lote no obtendríamos suficientes datos, repetiremos el proceso hasta llegar al 5% de la batería del móvil. Durante el proceso de prueba iremos recogiendo el consumo de batería, el tiempo total que ha tardado el dispositivo en realizar las consultas y el numero de repeticiones que llevamos del lote de consultas. **Entre cada ejecución de un lote de consultas dejaremos un intervalo de tiempo de 6 segundos.**

A continuación mostraremos la media de las pruebas realizadas sobre un tipo, redondeada hacia arriba, y al final de ellas analizaremos brevemente los valores obtenidos.

### 6.3.1. Pruebas SPO Java

La tabla 31 muestra la media obtenida tras haber repetido en dos ocasiones esta prueba en la aplicación HDT-Java:

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
500	100	1,014
5.000	99	0,5798
25.000	98	0,526
50.000	95	0,5245
100.000	89	0,51942
250.000	80	0,51674
500.000	66	0,516906
750.000	50	0,515948
1.000.000	32	0,515725
1.250.000	10	0,5154976
1.311.000 (media)	5	0,514994661

Tabla 31: Resultados medios obtenidos de las pruebas en Java de la Consulta SPO

La batería, según la media, ha tardado 5 horas y 6 minutos en descargarse.

6.3.2. Pruebas SPO C++

La tabla 32 muestra los valores medios de las pruebas sobre la consulta SPO en la aplicación HDT-Cpp:

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
500	100	0,39
5.000	99	0,293
25.000	98	0,2844
50.000	95	0,28622
100.000	93	0,28775
250.000	85	0,288948
500.000	72	0,302654
750.000	56	0,312009333
1.000.000	39	0,320529
1.250.000	16	0,335404
1.373.500 (media)	5	0,389145249

Tabla 32: Resultados medios obtenidos de las pruebas en C++ de la Consulta SPO

La batería ha tardado de media entorno a 5 horas y 14 minutos en descargarse.

### 6.3.3. Conclusiones SPO Java vs C++

Dado que la aplicación de C++, es capaz de hacer 75.000 consultas más, de media, que la aplicación Java, ya nos hace pensar que una consumirá mas batería que la otra, puesto que, un numero superior de consultas se traduce en una duración de la batería mayor. La Ilustración 40 muestra, de forma conjunta, el consumo de ambas aplicaciones.

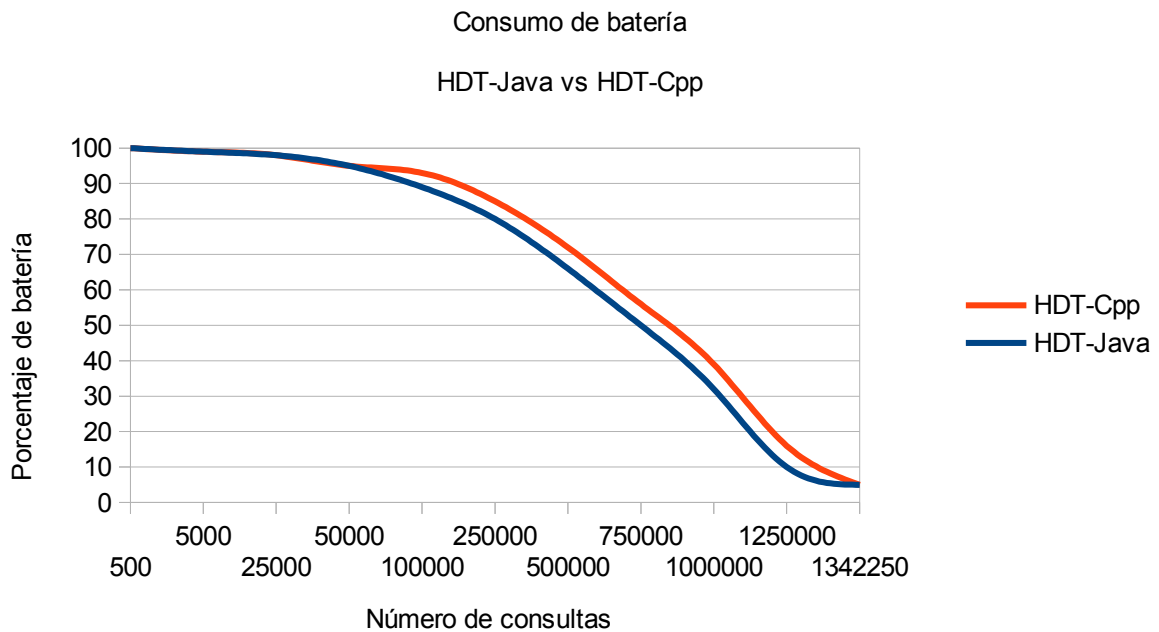


Ilustración 40: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta SPO

También hemos establecido una comparativa entre los tiempos de ejecución de cada aplicación, aunque actualmente existen una gran cantidad de *benchmark* de este tipo, lo realizaremos para asegurarnos de que en este caso también es más rápido C++ que Java. El resultado de esta comparación puede verse en la Ilustración 41.



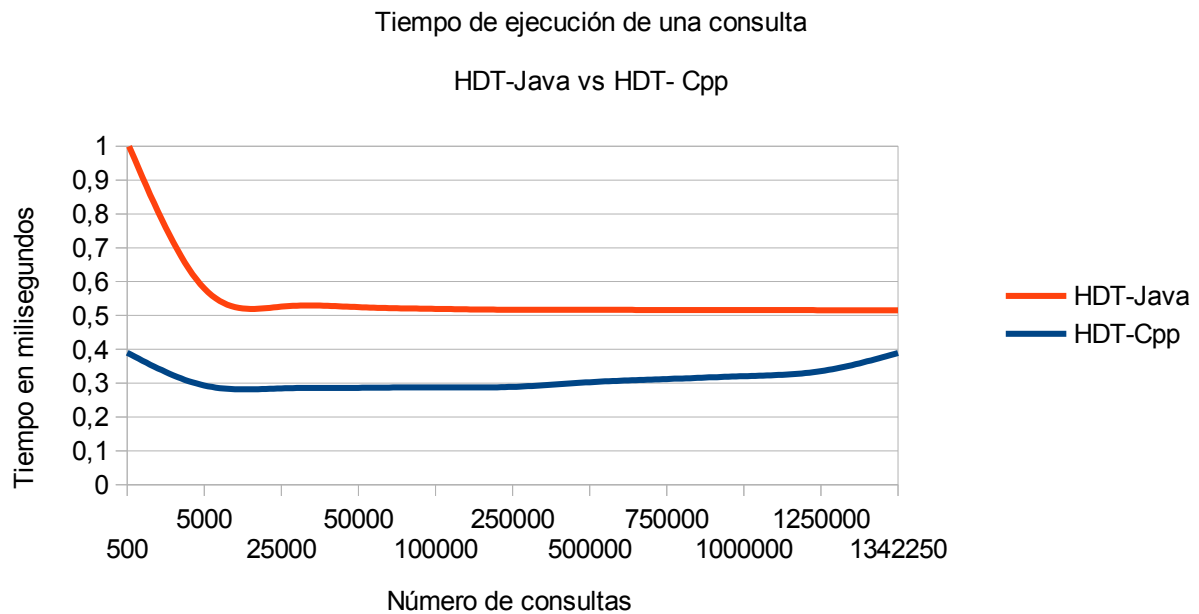


Ilustración 41: Diferencia de tiempos de ejecución entre la aplicación Java y C++ sobre la consulta SPO

Por lo tanto podemos concluir que en este caso que se cumple la máxima de que una aplicación que incorpora funciones nativas es más rápida que una desarrollada únicamente con Java, como muestra la Ilustración 41, la aplicación Cpp es más rápida, aunque como podemos ver tiene una tendencia creciente al final, y en ningún momento parece mantenerse constante, mientras que la aplicación Java, aunque mucho más lenta al principio, se mantiene constante en los tiempos de ejecución hasta el final. Como conclusión final, podemos demostrar mediante estos resultados que el consumo de la aplicación C++ es menor que el de Java.

### 6.3.4. Pruebas SP? Java

En este caso el número de consultas que vamos a realizar cada vez que ejecutemos un lote será de 564. La tabla 33 refleja la media de los resultados obtenidos de las pruebas en la aplicación Java del tipo de consultas SP?:

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
564	100	0,703900709
5.640	99	0,410638298
28.200	98	0,381276596
56.400	96	0,382464539
112.800	93	0,382269504
282.000	83	0,382677305
564.000	70	0,384445035
846.000	56	0,385208038
1.128.000	42	0,386093085
1.410.000	21	0,386598582
1.637.292 (media)	5	0,385567144

Tabla 33: Resultados medios obtenidos de las pruebas en Java de la Consulta SP?

La batería ha tardado, de media, 5 horas y 21 minutos en descargarse.

6.3.5. Pruebas SP? C++

La tabla 34 refleja la media de los resultados obtenidos de las pruebas en la aplicación C++ del tipo de consultas SP?:

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
564	100	0,226950355
5.640	99	0,24822695
28.200	98	0,240921986
56.400	97	0,246258865
112.800	94	0,246214539
282.000	86	0,248404255
564.000	72	0,257955674
846.000	57	0,259159574
1.128.000	43	0,281578014
1.410.000	25	0,29779078
1.652.520 (media)	5	0,329541549

Tabla 34: Resultados medios obtenidos de las pruebas en C++ de la Consulta SP?

La batería ha tardado, de media, 5 horas y 23 minutos en descargarse.

### 6.3.6. Conclusiones SP? Java vs C++

Al igual que sucede con las consultas SPO, en este caso también consume menos batería la aplicación HDT-Cpp, pero como se puede ver en la Ilustración 42 la diferencia de consumo entre ambas aplicaciones es mucho menor.

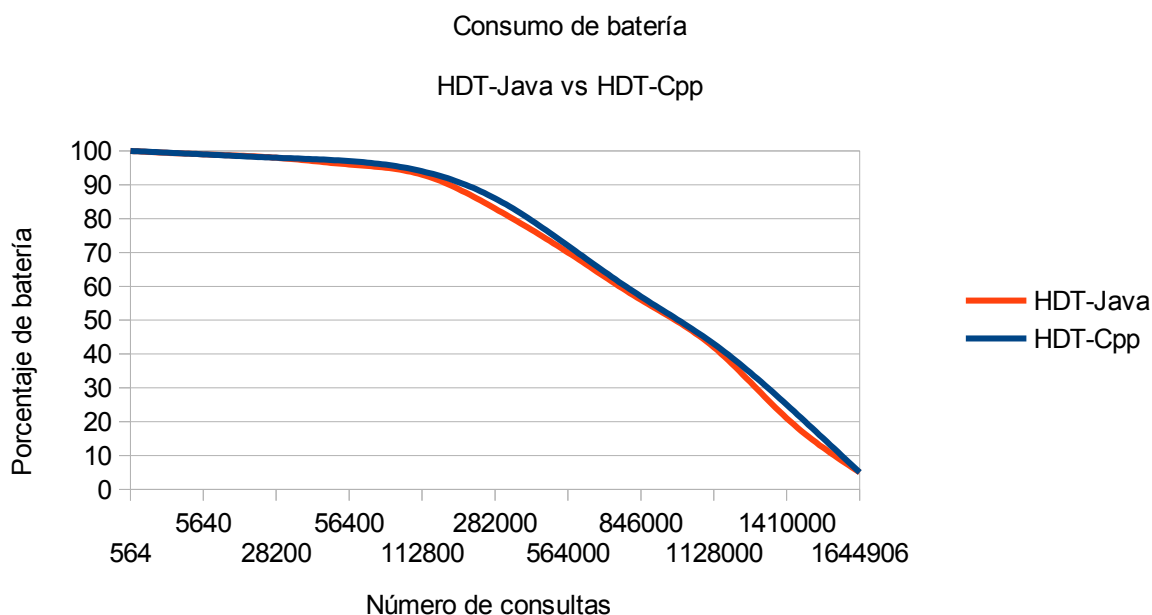


Ilustración 42: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta SP?

Sin embargo, en lo que al tiempo de ejecución se refiere, como muestra la Ilustración 43, la aplicación HDT-Cpp sigue siendo bastante más rápida que HDT-java. También podemos observar que sucede lo mismo que en el caso de las consultas SPO, y es que la aplicación Java tiene unos tiempos de ejecución más constantes que la aplicación Cpp.

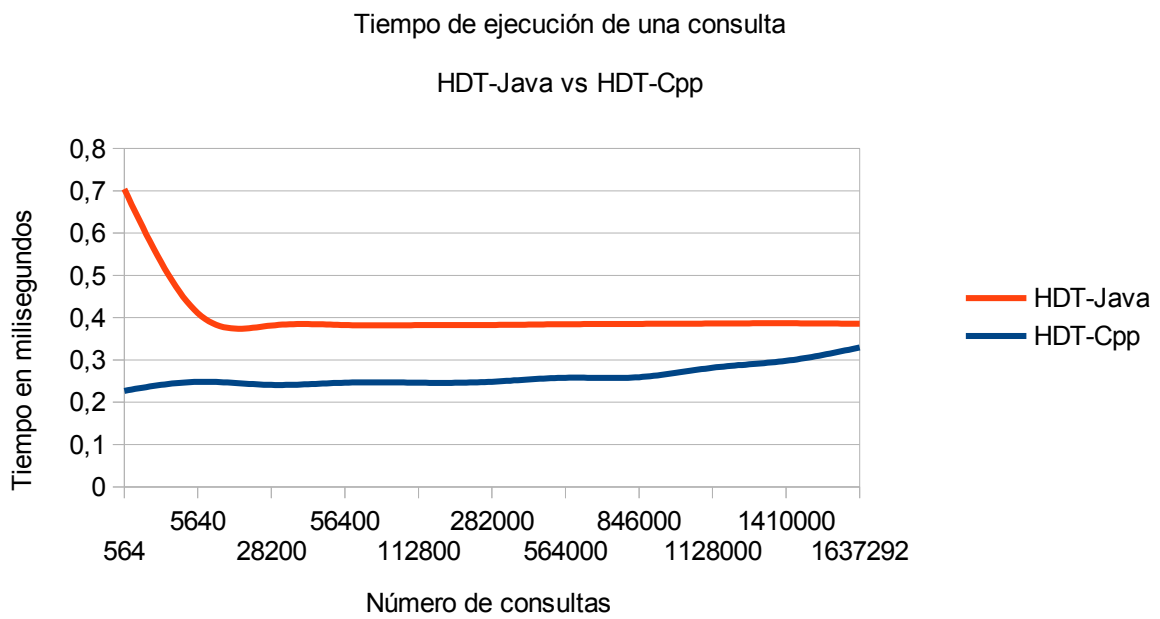


Ilustración 43: Diferencia de tiempos de ejecución entre la aplicación Java y C++ de la consulta SP?

### 6.3.7. Pruebas S?O Java

En este caso cada lote de búsquedas realizará 517 consultas, lo que le sitúa en una posición intermedia, en lo que a consultas se refiere, entre las dos pruebas anteriores. A continuación comprobaremos si esta posición intermedia hace que su consumo se sitúe también entre medias de los resultados anteriores, la tabla 35 refleja los resultados obtenidos:

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
517	100	0,959381044
5.170	99	0,593423598
25.850	98	0,558181818
51.700	96	0,555241779
103.400	92	0,553046422
258.500	82	0,55232882
517.000	68	0,555976789
775.500	53	0,556602192
1.034.000	41	0,556834236
1.292.500	19	0,558439845
1.545.572 (media)	5	0,556639845

Tabla 35: Resultados medios obtenidos de las pruebas en Java de la Consulta S?O

La batería ha tardado aproximadamente una media de 5 horas y 19 minutos en descargarse.

6.3.8. Pruebas S?O C++

La tabla 36 refleja la media de los resultados obtenidos de las pruebas en la aplicación C++ del tipo de consultas S?O:

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
517	100	0,348162476
5.170	99	0,292843327
25.850	98	0,29106383
51.700	97	0,287736944
103.400	94	0,285191489
258.500	86	0,283003868
517.000	74	0,274533849
775.500	61	0,27229529
1.034.000	45	0,294609671
1.292.500	22	0,334609671
1.574.007 (media)	5	0,36838108

Tabla 36: Resultados medios obtenidos de las pruebas en C++ de la Consulta S?O

La batería ha tardado de media entorno a 5 horas y 22 minutos en descargarse.

### 6.3.9. Conclusiones S?O Java vs C++

Como viene sucediendo a lo largo de todas las pruebas el consumo de la aplicación HDT-Cpp es menor que el de la aplicación HDT-Java, tal y como muestra la Ilustración 44.

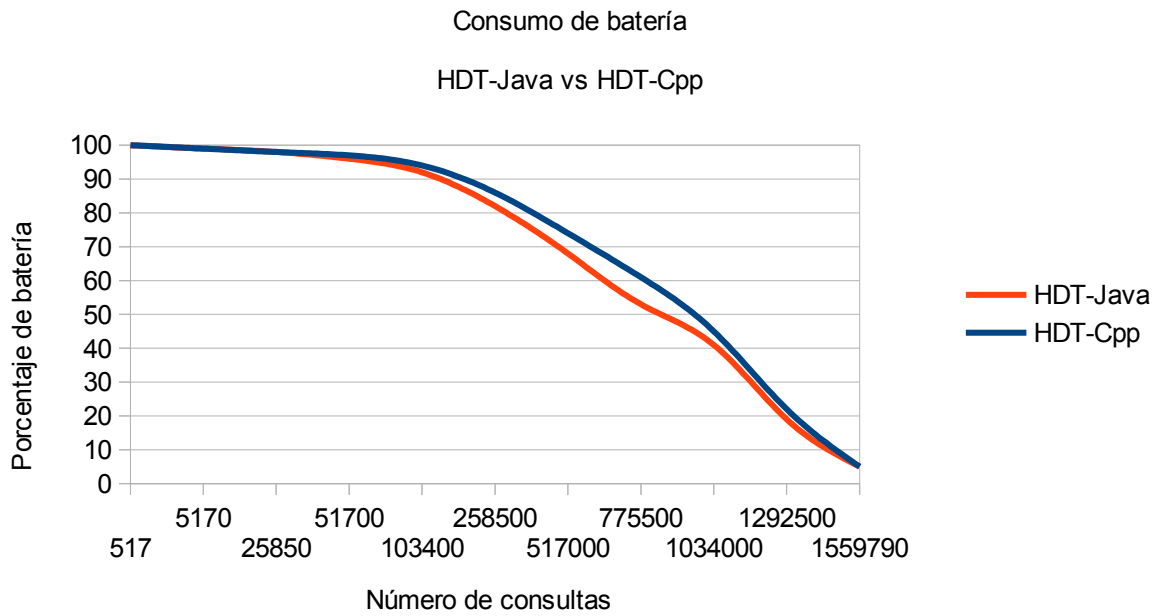


Ilustración 44: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta S?O

Como podemos observar, la aplicación Cpp sigue estando por debajo en lo que al consumo se refiere de la aplicación Java, lo mismo sucede con los tiempos ejecución de una consulta como veremos en la siguiente Ilustración 45.



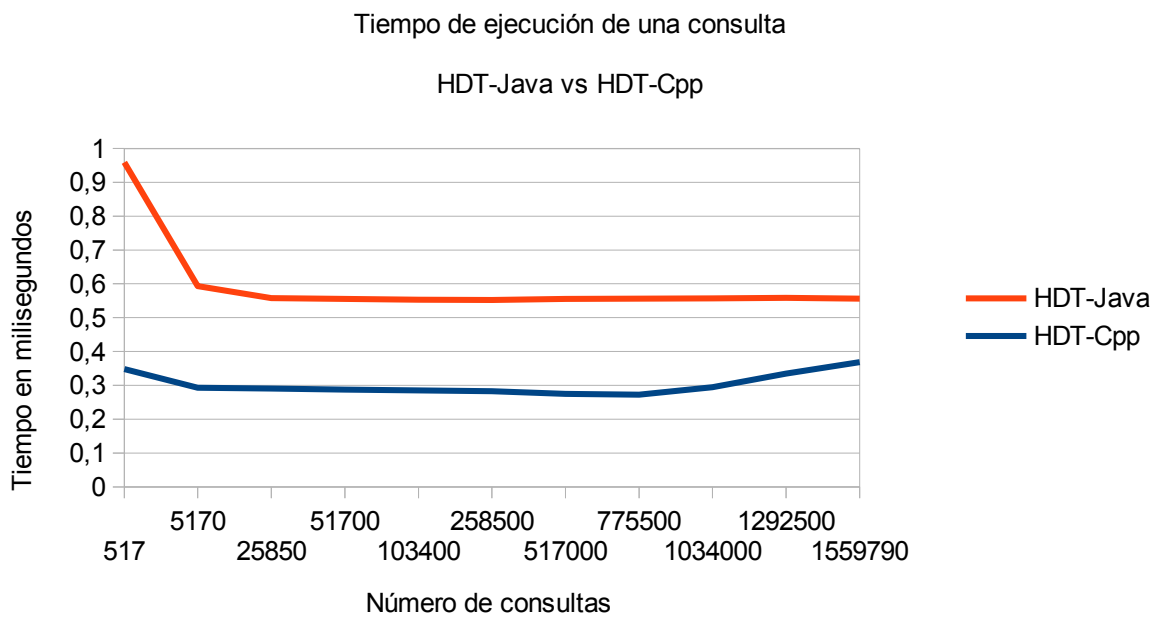


Ilustración 45: Comparación de tiempos de ejecución entre la aplicación Java y C++ de la consulta S?O

### 6.3.10. Pruebas S?? Java

Por último vamos a pasar a realizar las pruebas en el que cada lote de consultas devuelve más resultados, en concreto, con la ejecución de cada lote se van a realizar 4549 búsquedas. Como veremos a continuación realizar un número tan elevado de consultas va a suponer un mayor consumo de batería. La tabla 37 muestra los resultados medios obtenidos en las pruebas de la aplicación Java.

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
4.549	100	0,121125522
45.490	99	0,098483183
227.450	97	0,09449989
454.900	94	0,093715102
909.800	91	0,093627171
2.274.500	80	0,093738844
4.549.000	64	0,093874698
6.823.500	47	0,09397406
9.098.000	35	0,09440976
11.372.500	12	0,09415177
11.427.088 (media)	5	0,094147608

Tabla 37: Resultados medios obtenidos de las pruebas en Java de la Consulta S??

La batería ha tardado de media entorno a 4 horas y 59 minutos en desgastarse.

### 6.3.11. Pruebas S?? C++

La tabla 38 muestra los resultados medios obtenidos en las pruebas de la aplicación C++.

Número de resultados obtenidos	Estado de la batería tras la ejecución del lote (%)	Tiempos de ejecución de una consulta (ms)
4.549	100	0,123323807
45.490	99	0,130424269
227.450	93	0,131272807
454.900	86	0,128729391
909.800	74	0,128469993
2.274.500	64	0,130372829
4.549.000	50	0,130714003
6.823.500	36	0,131724775
9.098.000	19	0,166533634
10.869.836 (media)	5	0,171585017

Tabla 38: Resultados medios obtenidos de las pruebas en C++ de la Consulta S??

La batería ha tardado de media aproximadamente 4 horas y 32 minutos en descargarse.

### 6.3.12. Conclusiones S?? Java vs C++

Como reflejan las tablas de los apartados anteriores, la aplicación HDT-Cpp ha sido la gran perjudicada, puesto que con este número tan elevado de consultas tanto su consumo como su tiempo de ejecución se han disparado. La Ilustración 46 muestra ambas gráficas juntas y podremos ver más claramente como en esta ocasión el consumo de Java es mucho menor al de la aplicación C++, llegando incluso esta última a no alcanzar la ejecución de 2500 lotes de consulta, cosa que hará que en las ilustraciones aparezca con un punto menos en su representación.

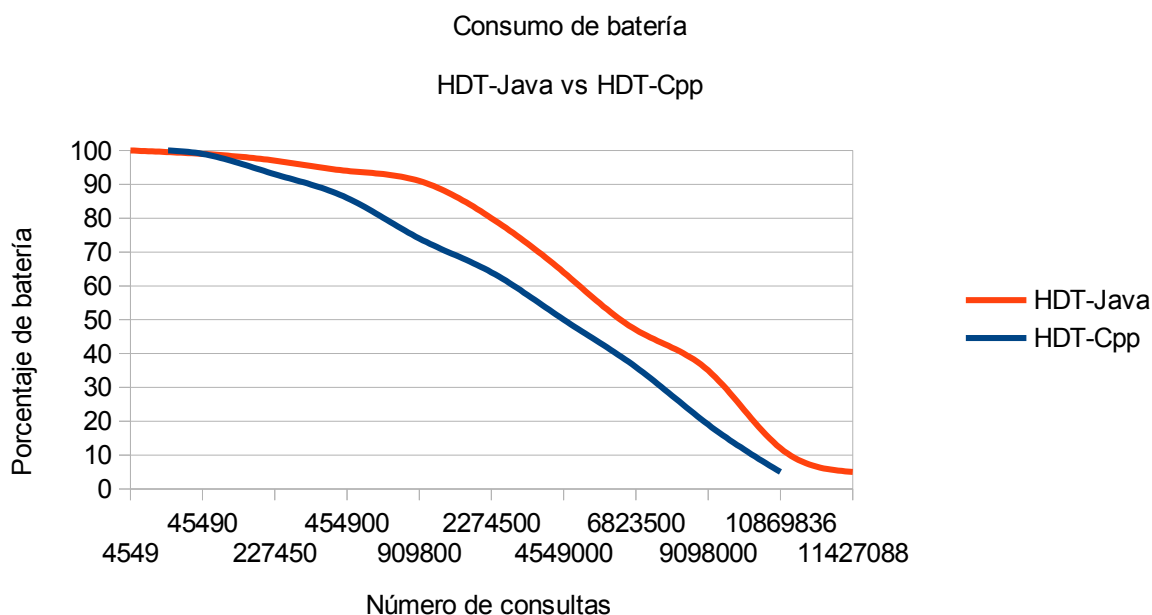


Ilustración 46: Comparación del consumo entre la aplicación Java y la de C++ sobre la consulta S??

Este resultado es sorprendente puesto que en las pruebas anteriores la aplicación HDT-Cpp había sido siempre superior a la aplicación desarrollada en Java. Esto nos lleva a pensar que quizás cuando se trata de cargas de trabajo pequeñas la aplicación HDT-Cpp es mucho más efectiva, mientras que ante situaciones con una carga de trabajo mayor es el lenguaje propio de Android quien mejor gestiona tanto el consumo como el tiempo de ejecución. El resultado de los tiempos de ejecución se muestra en la Ilustración 47.

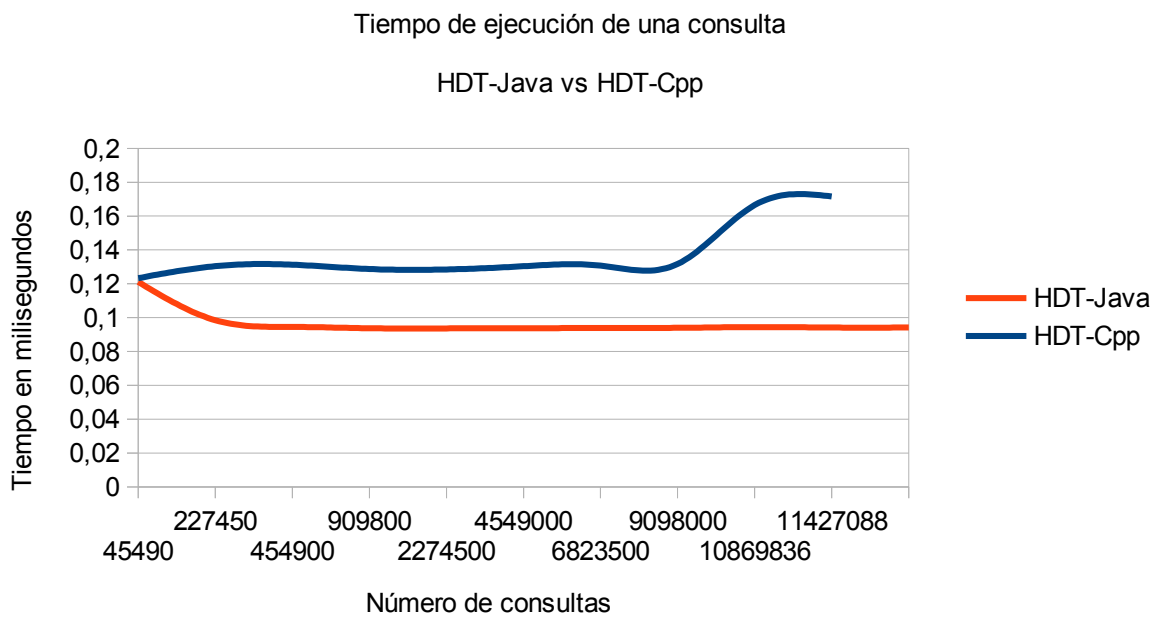


Ilustración 47: Diferencia de tiempos de ejecución entre la aplicación Java y C++ de la consulta S??



## **Capítulo 7. Manuales**





## 7.1. Guía de instalación

En primer lugar, tendremos que tener localizado nuestro conjunto de datos, LinkedMDB, en una ruta específica en nuestro dispositivo móvil para poder realizar las consultas. Dicha ruta es la siguiente: `/mnt/sdcard/Download/linkedmdb2010.hdt`. Donde `sdcard` indica que se trata del espacio de almacenamiento interno del dispositivo móvil, y en el, tendremos que situar el fichero en la carpeta `Download`. Las Ilustraciones 48,49,50 y 51 muestran como llegar hasta dicha carpeta, utilizando el administrador de archivos *ES File Explorer*, el cual se encuentra disponible en el *Play Store* de Android.

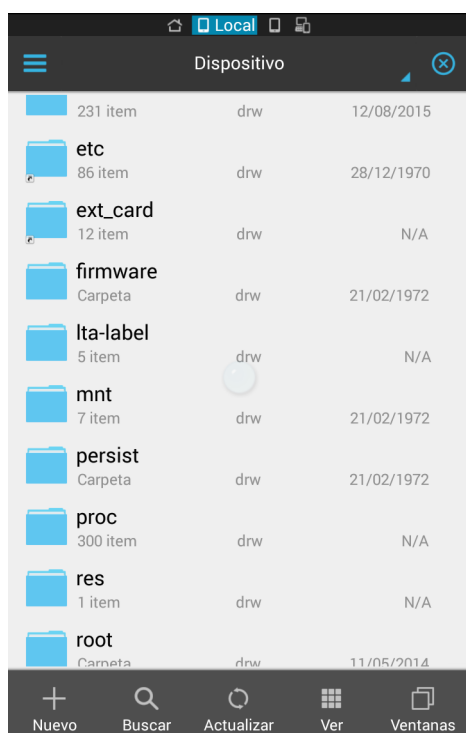


Ilustración 48: Carpeta Dispositivo

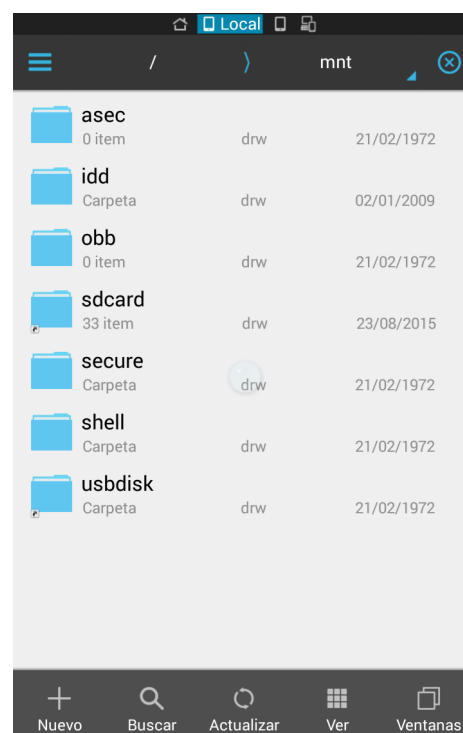


Ilustración 49: Carpeta mnt

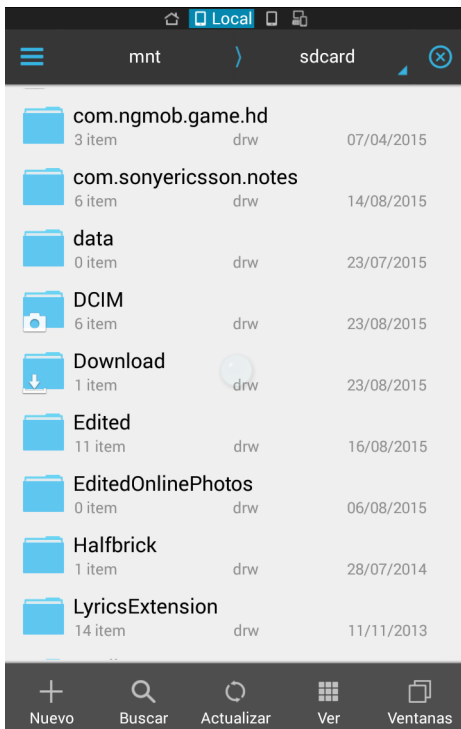


Ilustración 50: Carpeta sdcard

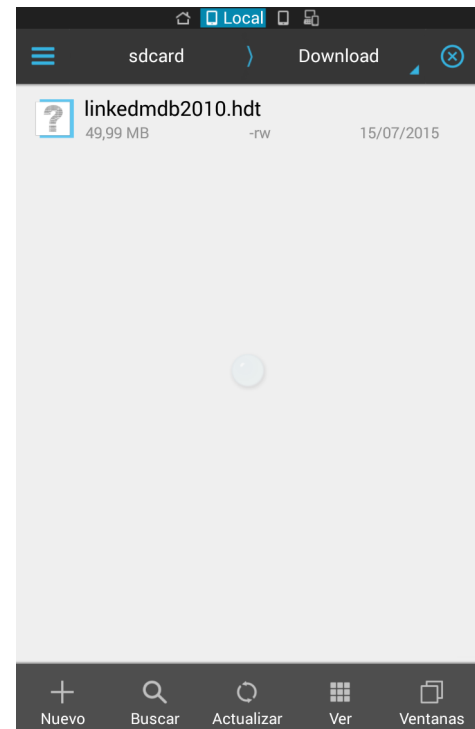


Ilustración 51: carpeta Download

Una vez hemos situado el fichero HDT en la carpeta correcta, estamos listos para instalar nuestras aplicaciones en el dispositivo móvil. Hay que destacar que ambas aplicaciones soportan errores, es decir, que si no hubiésemos colocado correctamente el fichero nos avisarían con un mensaje de alerta y no permitirían realizar ningún tipo de consulta. Dado que nuestras aplicaciones no se encuentran disponibles en el *Play Store*, tendremos que copiar los archivos de instalación (.apk) al dispositivo móvil. Al contrario que en el caso anterior, en esta ocasión no será necesario situar los archivos de instalación en ninguna ruta concreta del dispositivo móvil.

Antes de comenzar con la instalación tendremos que cerciorarnos de que el sistema Android de nuestro dispositivo nos permita instalar aplicaciones de fuentes desconocidas, para ello haremos lo siguiente: Nos vamos a **Ajustes**, y ahí hacemos click en la opción **Seguridad**, una vez dentro buscamos la opción **Origenes Desconocidos**, y en caso de que este desmarcada la activamos.

Hecho esto y una vez hayamos copiado los archivos de instalación que se adjuntan con esta memoria al dispositivo, procederemos a su ejecución. Para realizar esta tarea nos situaremos, mediante la misma aplicación que utilizamos en el caso anterior, en la carpeta donde los hayamos copiado dichos archivos de instalación y los ejecutaremos. En este caso instalaremos la aplicación **HDT-Cpp**, pero la instalación de la aplicación **HDT-Java** se realiza de la misma forma a esta. Cuando la instalación comience nos tendrá que aparecer algo parecido a lo que muestra la Ilustración 53:

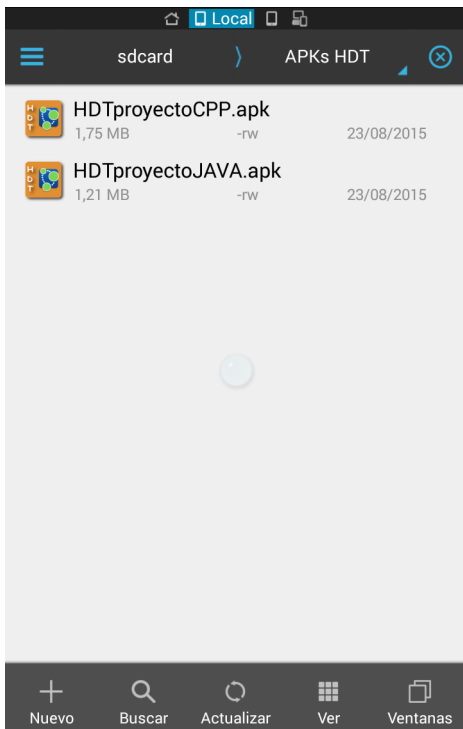


Ilustración 52: Ejecutables de HDT

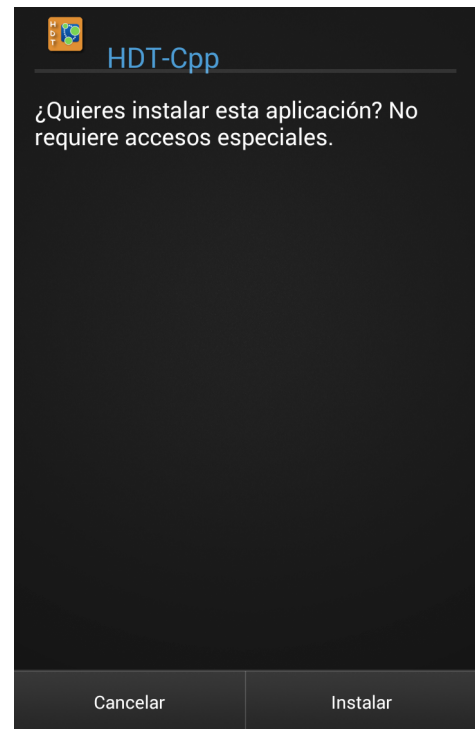


Ilustración 53: Permisos de instalación

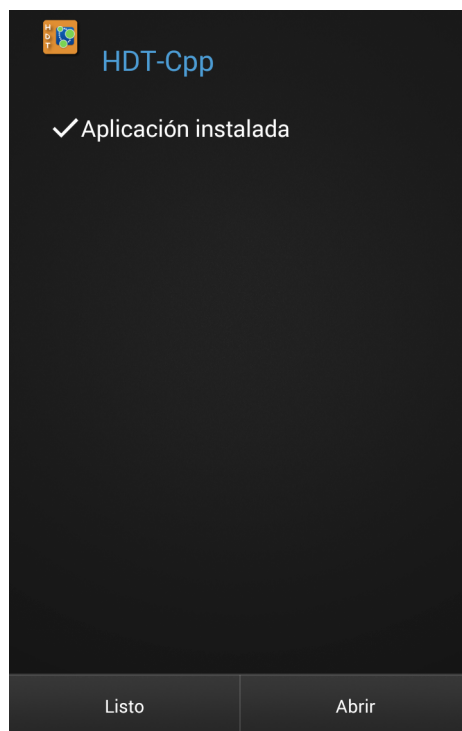


Ilustración 54: Instalación completada

## 7.2. Manual de usuario

Cada vez que iniciemos la aplicación aparecerá la pantalla de bienvenida que refleja la Ilustración 55:

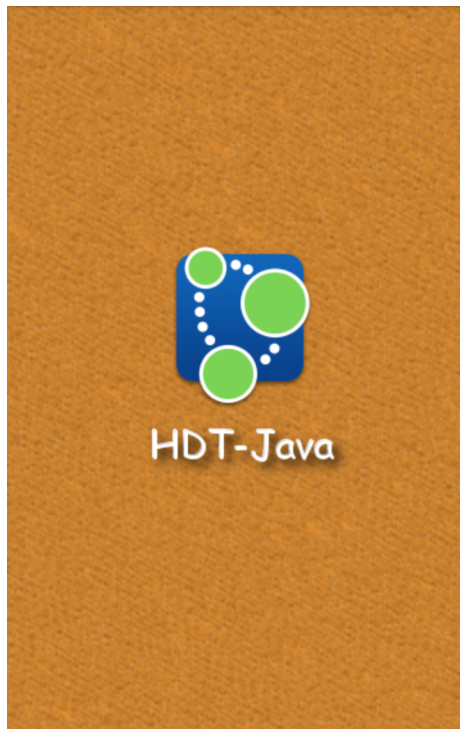


Ilustración 55: Pantalla bienvenida

Después de esta breve pantalla de bienvenida, lo siguiente que mostrará la aplicación será el menú de consultas que podemos realizar al conjunto de datos. La Ilustración 56 muestra el aspecto de esta pantalla:



Ilustración 56: Pantalla inicial

Cada elemento del menú realiza una acción diferente, por ello vamos a separar en diferentes secciones cada una de las opciones de la lista.

### 7.2.1. Opción películas

Cuando pulsamos sobre el elemento *Consulta Peliculas* la pantalla a la que nos lleva la aplicación es la que muestra la Ilustración 57:



Ilustración 57: Pantalla Consulta Películas

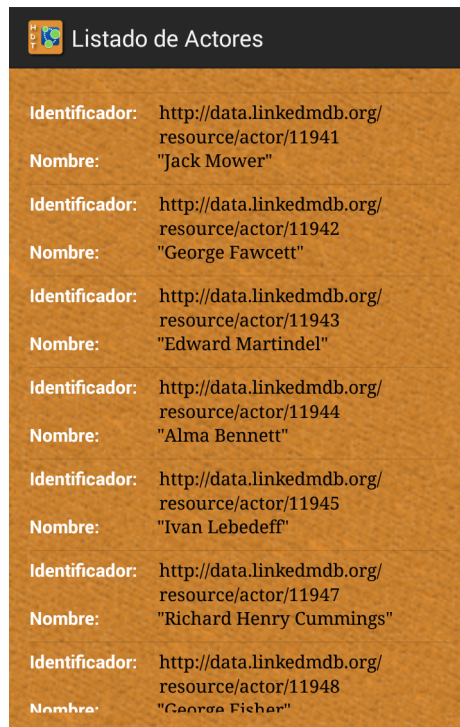
Como vemos, en esta pantalla se muestra el catálogo completo de las películas que tiene el conjunto de datos *LinkedMDB*, **permitiendo al usuario seleccionar cualquiera de ellas**. Para poner un ejemplo de uso, a continuación vamos a elegir la película *Godzilla*, y de ella obtendremos todos los resultados en formato *triple pattern* que contiene nuestro conjunto de datos sobre dicha película. La Ilustración 58 presenta la pantalla que obtendremos para dicho ejemplo:

Resultado de la Consulta	
Objeto:	<a href="http://data.linkedmdb.org/resource/writer/2700">http://data.linkedmdb.org/resource/writer/2700</a>
Sujeto:	<a href="http://data.linkedmdb.org/resource/film/12">http://data.linkedmdb.org/resource/film/12</a>
Predicado:	<a href="http://data.linkedmdb.org/resource/movie/writer">http://data.linkedmdb.org/resource/movie/writer</a>
Objeto:	<a href="http://data.linkedmdb.org/resource/writer/2701">http://data.linkedmdb.org/resource/writer/2701</a>
Sujeto:	<a href="http://data.linkedmdb.org/resource/film/12">http://data.linkedmdb.org/resource/film/12</a>
Predicado:	<a href="http://purl.org/dc/terms/date">http://purl.org/dc/terms/date</a>
Objeto:	"1954-11-03"
Sujeto:	<a href="http://data.linkedmdb.org/resource/film/12">http://data.linkedmdb.org/resource/film/12</a>
Predicado:	<a href="http://purl.org/dc/terms/title">http://purl.org/dc/terms/title</a>
Objeto:	"Godzilla"
Sujeto:	<a href="http://data.linkedmdb.org/resource/film/12">http://data.linkedmdb.org/resource/film/12</a>
Predicado:	<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">http://www.w3.org/1999/02/22-rdf-syntax-ns#type</a>
Objeto:	<a href="http://data.linkedmdb.org/resource/movie/film">http://data.linkedmdb.org/resource/movie/film</a>

Ilustración 58: Pantalla Resultado de la Consulta Películas

### 7.2.2. Opción actores

Seleccionando la opción *Consulta Actores* de la pantalla de inicio, la aplicación nos enviará a la pantalla que muestra la Ilustración 59:



Listado de Actores	
Identificador:	http://data.linkedmdb.org/resource/actor/11941
Nombre:	"Jack Mower"
Identificador:	http://data.linkedmdb.org/resource/actor/11942
Nombre:	"George Fawcett"
Identificador:	http://data.linkedmdb.org/resource/actor/11943
Nombre:	"Edward Martindel"
Identificador:	http://data.linkedmdb.org/resource/actor/11944
Nombre:	"Alma Bennett"
Identificador:	http://data.linkedmdb.org/resource/actor/11945
Nombre:	"Ivan Lebedeff"
Identificador:	http://data.linkedmdb.org/resource/actor/11947
Nombre:	"Richard Henry Cummings"
Identificador:	http://data.linkedmdb.org/resource/actor/11948
Nombre:	"George Fisher"

Ilustración 59: Pantalla Consulta Actores

La única tarea de esta pantalla es mostrar el listado completo de actores que contiene el conjunto de datos *LinkedMDB*, este tipo de consulta está pensado para que el usuario busque a su actor/actriz favorito y obtenga su identificador único, con el que será capaz de realizar consultas más específicas sobre el o ella.

### 7.2.3. Opción directores

La Ilustración 60 muestra la pantalla resultado de seleccionar la opción *Consulta Directores* de la pantalla de inicio:



Ilustración 60: Pantalla Consulta Directores

En esta pantalla se mostrará un listado de todos los directores que contiene el fichero *LinkedMDB* al que estamos realizando las consultas. Esta pantalla también nos permite seleccionar un director cualquiera, y tras haber elegido uno, la aplicación nos enviara a otra pantalla donde se mostrarán las películas que ha dirigido el director escogido. Para poner un ejemplo practico vamos a mostrar el resultado que obtendríamos de seleccionar como director al afamado cantante *Bob Dylan*, la Ilustración 61 muestra las películas que ha dirigido:

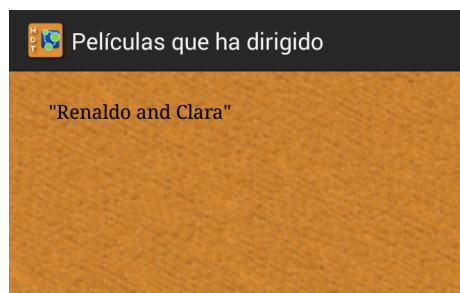


Ilustración 61: Pantalla Resultado de la Consulta Directores



### 7.2.4. Opción consulta avanzada

Al elegir la opción *Consulta Avanzada* en la pantalla de inicio, se presentará la pantalla que muestra la Ilustración 62:

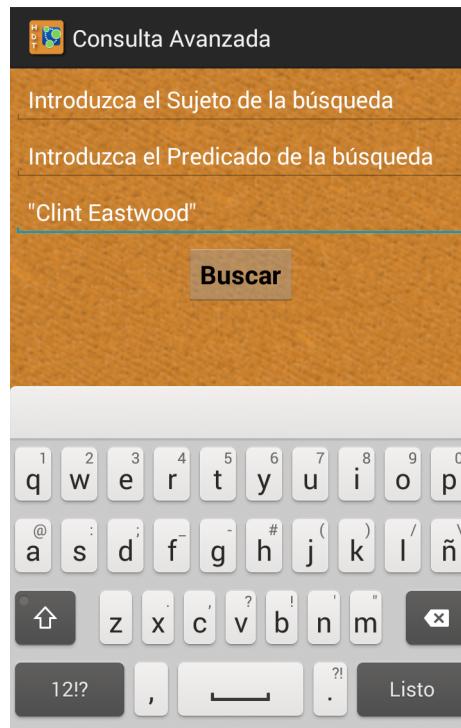


Ilustración 62: Pantalla Consulta Avanzada

Como se puede ver en la Ilustración 62, esta pantalla nos permite introducir cualquiera de los tres parámetros de un *triple pattern*, es decir, que podemos realizar una consulta introduciendo el sujeto, el predicado o el objeto en cualquier combinación, por lo que esto nos permite realizar cualquier tipo de búsqueda siempre y cuando conozcamos el modelo conceptual de *LinkedMDB*, ya que hay que introducir el parámetro exacto a como esta descrito en el conjunto de datos. Para este caso pondremos como ejemplo la búsqueda que aparece en la Ilustración 62, relacionada con el actor *Clint Eastwood*, para ello, introduciremos su nombre como objeto de la consulta, y el resultado que vamos a obtener se muestra en la Ilustración 63:

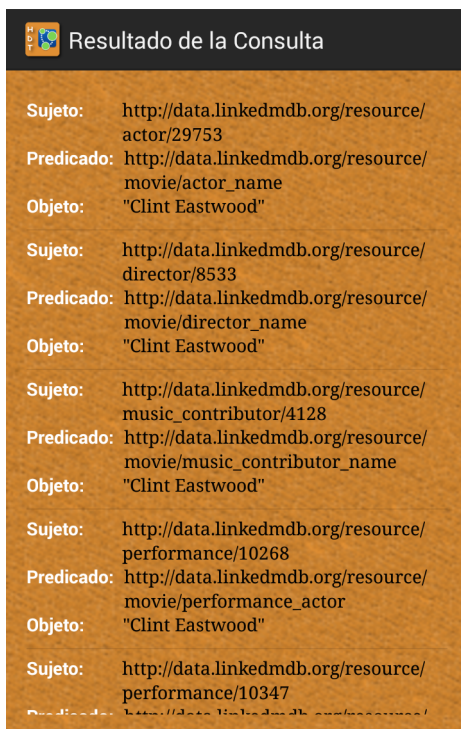


Ilustración 63: Pantalla Resultado de la Consulta Avanzada

### 7.2.5. Opción realizar pruebas

Cuando elijamos la opción *Realizar Pruebas* de la pantalla de inicio, la aplicación nos enviará a la pantalla que muestra la Ilustración 64:

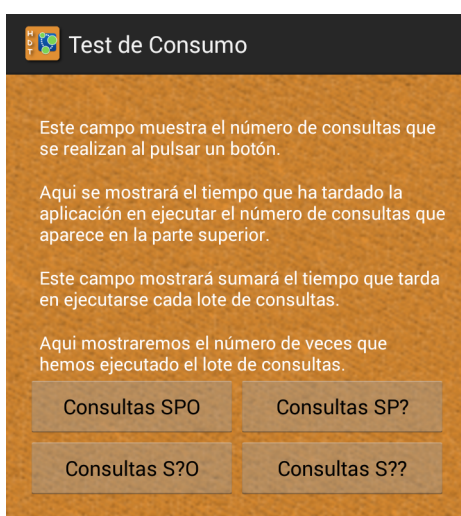


Ilustración 64: Pantalla Realizar Pruebas

Esta pantalla nos permitirá realizar diferentes tipos de pruebas en la aplicación. Estas pruebas se basan en realizar lotes con 500 consultas predefinidas al conjunto de datos. La Ilustración 65 muestra el resultado de ejecutar un lote de consultas del tipo SPO.

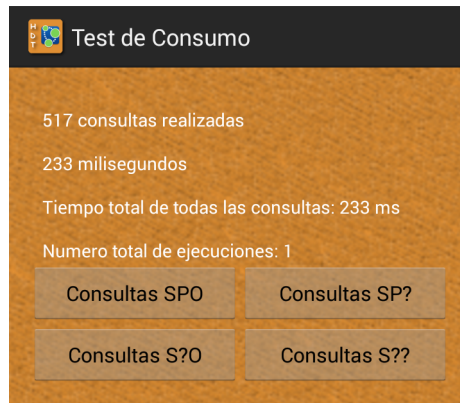


Ilustración 65: Pantalla Resultado de Realizar Pruebas



## **Capítulo 8. Conclusiones**



## **8.1. Conclusiones generales**

En primer lugar, destacar que se han cumplido todos los objetivos que se propusieron al comienzo de este trabajo de fin de grado. Se han conseguido además varios resultados interesantes sobre los lenguajes de programación Java y C++ y como afectan sus códigos de consulta al consumo de batería de un terminal móvil, aspecto que prácticamente nadie antes había tenido en cuenta y que nos sitúa como referencia en este aspecto.

Con la terminación de este proyecto hemos conseguido tener una aplicación estable que permite consultar cualquier tipo de información sobre el mundo cinematográfico en apenas unos segundos y con la ventaja de tener dicha aplicación disponible en cualquier gracias a los terminales móviles.

Personalmente este Trabajo de Fin de Grado me ha servido para darme cuenta de que la informática es mucho más que diseñar una página web, o estar “picando” código constantemente, y es que con el avance tecnológico que existe a día de hoy, y la velocidad a la que se expande, hacen de la informática un mundo en el que te encuentras en constante desconocimiento. Esta palabra, desconocimiento, es la que en mi opinión ha marcado este TFG, puesto que esta rodeado de tecnologías solo conocidas en el mundo de la informática y cuya aplicación o estudio requiere mucho tiempo y disciplina. En este aspecto tanto la Web Semántica, como el formato HDT eran soluciones tecnológicas totalmente irreconocibles para mí, al igual que el lenguaje de programación C++, el framework JNI, la herramienta NDK y el desarrollo nativo en Android en su conjunto.

El desarrollo de este proyecto también nos ha servido para poder ver tanto las ventajas como las desventajas de utilizar librerías nativas en Android, y es que como podemos comprobar en el apartado de las pruebas, cuando se trata de realizar pequeños conjuntos de consultas u operaciones poco complejas el lenguaje C++ no tiene rival, sin embargo cuando la carga de trabajo comienza a aumentar, al menos en el entorno en el que hemos trabajado nosotros, este lenguaje empieza a flaquear. Esto hemos podido comprobarlo en el último caso de ejecución de las pruebas, cuando hemos realizado un conjunto de 4549 consultas la aplicación HDT-Cpp ha comenzado a dar problemas en lo que al consumo se refiere, puesto que ha tardado casi 20 minutos menos en acabar descargar la batería que el lenguaje Java. Como conclusión a este punto, los resultados obtenidos de las pruebas nos llevan a pensar que el desarrollo nativo es muy eficaz para operaciones simples, pero que no está tan bien optimizado como el lenguaje Java cuando se trata de grandes cargas de trabajo o operaciones más complejas.

Pues bien, en resumen este proyecto me ha ayudado a ser un poco menos ignorante en lo que respecta a todo lo que acabo de nombrar y, sobre todo, a superarme ante las adversidades, que no han sido pocas, que se pueden presentar en medio del desarrollo de una aplicación.

## **8.2. Dificultades**

En este apartado vamos a detallar brevemente las dificultades que nos hemos encontrado durante el desarrollo de este TFG:

## Capítulo 8. Conclusiones

- En primer lugar empezar a desarrollarlo a través del sistema operativo Windows, fue prácticamente un mes y medio tirado a la basura intentando añadir al sistema diferentes funcionalidades para que fuese capaz de ejecutar y compilar el lenguaje de programación C++.
- Por otra parte el desconocimiento de las librerías implementadas por el proyecto RDFHDT hizo que tuviese que dedicar bastante tiempo a ver como trabajaban y realizaban sus operaciones.
- Al igual que sucedió con las librerías, ocurrió con el formato HDT y la transformación del conjunto de datos LinkedMDB a este formato. Entender el modelo conceptual de este conjunto y comprenderlo también fue complejo.
- Sin duda alguna la parte más complicada del proyecto fue la adaptación de la librería C++ a Java a través de JNI, no solo por el hecho de tener que aprender conceptos básicos de un nuevo lenguaje como es C++, si no también por utilizar una tecnología tan poco conocida como lo es JNI y de la que existen muy pocos ejemplos y documentación. Entender como se comunicaba con Java a través de la JVM y como establecía el paso de parámetros fue algo tedioso y enormemente desalentador.
- Por último, el desarrollo de dos aplicaciones Android, que aunque parecidas en su apariencia, tienen varias diferencias en su implementación, fue otro aspecto a tener en cuenta puesto que trabajan de diferentes formas y hay que tener especial cuidado en el caso de C++ y su recolección de basura en Java.

### 8.3. Trabajo futuro

Las aplicaciones entregadas con este proyecto podrían ampliar sus funcionalidades, a continuación mostramos una serie de aspectos en los que se podría trabajar:

- ◆ Implementar un buscador que permita reducir el conjunto de resultados ofrecido por las pantallas de Consulta Películas, Actores y Directores de forma que sea más fácil localizar un determinado resultado.
- ◆ Para la consulta avanzada hacer que los cuadros de búsqueda se autocompleten, es decir, que cuando el usuario este escribiendo el parámetro de su consulta, se le ofrezcan las coincidencias que contiene el conjunto de datos sobre las letras que esta introduciendo.
- ◆ Otra funcionalidad interesante sería que la aplicación permitiese al usuario elegir la ruta en la que se encuentra el conjunto de datos sobre el que realizaremos las consultas.
- ◆ Quizás la mejora más notable sería que la ejecución de las consultas se ejecutase en *background* tanto en Java como en la adaptación de C++, pero esto sería algo sumamente



## *Capítulo 8. Conclusiones*

complejo debido a la forma que tiene Android de gestionar el hilo principal de la aplicación y las *AsyncTask*.

- ◆ Por último subir la aplicación al Play Store haría que esta fuese accesible por mucha más gente lo cual daría lugar a un gran *feedback* que ayudaría a ir aumentando el tamaño del proyecto y añadir diferentes funcionalidades.



## **Capítulo 9. Bibliografía**

- [1] Javier D. Fernández, Miguel A. Martínez Prieto, Mario Arias, Claudio Gutiérrez, Sandra Álvarez-García, Nieves R. Brisaboa  
Lightweighting the Web of Data through Compact RDF/HDT  
14th Conference of the Spanish Association for Artificial Intelligence (CAEPIA 2011)  
Página(s): 483-493  
Disponible en: <http://dataweb.infor.uva.es/wp-content/uploads/2011/07/caepia2011.pdf>  
Último acceso: Febrero 2015
- [2] Miguel A. Martínez Prieto y Javier D. Fernández  
Aprendiendo a nadar en el diluvio de datos  
Disponible en: <http://dataweb.infor.uva.es/wp-content/uploads/2012/03/curso2.pdf>  
Último acceso: Mayo 2015
- [3] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez  
HDT-it: Storing, Sharing and Visualizing Huge RDF Datasets  
10th International Semantic Web Conference (ISWC 2011), 2011.  
Disponible en: <http://dataweb.infor.uva.es/wp-content/uploads/2011/10/iswc2011.pdf>  
Último acceso: Julio 2015
- [4] Javier Fernández, Miguel Martínez-Prieto, Claudio Gutiérrez y Axel Polleres  
Binary RDF Representation for Publication and Exchange (HDT)  
W3C Member Submission, 2011.  
Disponible en: <http://www.w3.org/Submission/2011/SUBM-HDT-20110330/>  
Último acceso: Marzo 2015
- [5] RDF/HDT  
Página oficial del proyecto HDT  
Disponible en: <http://www.rdfhdt.org/>  
Último acceso: Agosto 2015
- [6] Aníbal Bregón Bregón  
Listas y Adaptadores  
Apuntes de la asignatura Plataformas Software Móviles  
Último acceso: Junio 2015
- [7] Javier García de Jalón, José Ignacio Rodríguez, José María Sarriegui y Alfonso Brazález  
Aprenda C++ como si estuviera en primero  
Disponible en: <http://mat21.etsii.upm.es/ayudainf/aprendainf/Cpp/manualcpp.pdf>  
Último acceso: Julio 2015
- [8] Chris Cannam  
Wrapping a C++ library with JNI  
Publicado el 21 de Enero de 2012  
Disponible en: <http://thebreakfastpost.com/2012/01/21/wrapping-a-c-library-with-jni-introduction/>  
Último acceso: Agosto 2015
- [9] Fernando López Hernández  
JNI: Java Native Interface  
Disponible en: <http://macprogramadores.org/documentacion/jni.pdf>  
Último acceso: Agosto 2015

## Capítulo 9. Bibliografía

- [10] Funciones de JNI  
JNI Functions  
Disponible en:  
<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp5901>  
Último acceso: Junio 2015
- [11] Especificación de JNI  
Java Native Interface Specification  
Publicada el 16 de mayo de 1997  
Disponible en:  
<http://publib.boulder.ibm.com/html/as400/v4r5/ic2931/info/java/rzaha/javaapi/jni/spec/jniTOC.doc.html>  
Último acceso: Mayo 2015
- [12] Consejos sobre JNI  
JNI Tips  
Disponible en: <http://developer.android.com/training/articles/perf-jni.html>  
Último acceso: Mayo 2015
- [13] Peter Sestoft  
Numeric performance in C, C# and Java  
Publicado el 19 de Febrero de 2010  
Disponible en: <http://www.itu.dk/people/sestoft/papers/numericperformance.pdf>  
Último acceso: Agosto 2015
- [14] Grupo Network Theory  
Una introducción a GCC  
Disponible en: <http://www.davidam.com/docu/gccintro.es.html#Examinado-archivos-compilados>  
Último acceso: Abril 2015
- [15] Jeffrey T. Pollock  
“Semantic Web for Dummies”  
Publicado en Marzo del 2009  
Último acceso: Marzo 2015
- [16] Josh Tauberer  
What is RDF and what is it good for?  
Disponible en: <https://github.com/JoshData/rdfabout/blob/gh-pages/intro-to-rdf.md#>  
Último acceso: Mayo 2015
- [17] Eva Méndez Rodríguez  
RDF: Un modelo de metadatos flexible para las bibliotecas digitales del próximo milenio  
Disponible en: <http://www.cobdc.org/jornades/7JCD/1.pdf>  
Último acceso: Marzo 2015

## Capítulo 9. Bibliografía

- [18] Web Semántica  
Guía breve sobre la web semántica  
Disponible en: <http://www.w3c.es/Divulgacion/GuiasBreves/WebSemantica>  
Último acceso: Febrero 2015
- [19] María Jesús Lamarca Lapuente  
Hacia la Web Semántica  
Publicado en Mayo de 2011  
Disponible en: [http://www.hipertexto.info/documentos/web\\_semantica.htm](http://www.hipertexto.info/documentos/web_semantica.htm)  
Último acceso: Marzo 2015
- [20] Web Semántica  
¿Aún no comprendes la Web Semántica?  
Disponible en: <https://semantizandolaweb.wordpress.com/2011/11/01/aun-no-comprendes-la-web-semantica/>  
Último acceso: Abril 2015
- [21] MARIO PÉREZ ESTESO  
Guía para Android NDK  
Disponible en: <https://geekytheory.com/category/geeky-theory-2/tutoriales-2/android-2/tutoriales-android/android-ndk/>  
Último acceso: Febrero 2015
- [22] NDK  
Introducción al NDK de Android  
Disponible en: <http://elbauldelprogramador.com/introduccion-al-ndk-de-android/>  
Último Acceso: Marzo 2015
- [23] SKOS Simple Knowledge Organization System Primer, Grupo W3C  
SPARQL Lenguaje de consulta para RDF Publicado el 18 de agosto de 2009  
Disponible en: <http://skos.um.es/TR/rdf-sparql-query/>  
Último acceso: Junio 2015
- [24] Jérôme David, Jérôme Euzenat y Maria-Elena Rosoiu  
Linked data from your pocket  
Disponible en: <ftp://ftp.inrialpes.fr/pub/exmo/publications/david2012a.pdf>  
Último acceso: Abril 2015
- [25] Miguel Pérez  
El impacto de las nuevas tecnologías en la era post-PC  
Publicado el 14 de mayo de 2013  
Disponible en: <http://blogthinkbig.com/nuevas-tecnologias-era-post-pc/>  
Último acceso: Marzo 2015

## Capítulo 9. Bibliografía

- [26] Guía oficial de las APIs de Google  
Disponibile en: <http://developer.android.com/guide/index.html>  
Último acceso: Marzo 2015
- [27] Benchmarking  
A performance comparision redux: Java, C, and renderscript on the Nexus 5  
Disponibile en: <http://www.learnopengles.com/a-performance-comparison-redux-java-c-and-renderscript-on-the-nexus-5/>  
Último acceso: Agosto 2015
- [28] Miguel Ángel  
ListView y ArrayAdapter en Adnroid  
Disponibile en: <https://miguelangellv.wordpress.com/2012/05/05/listview-y-arrayadapter-en-android/>  
Último acceso: Abril 2015
- [29] Crear un SplashScreen en Android  
Disponibile en: <https://amatellanes.wordpress.com/2013/08/27/android-crear-un-splash-screen-en-android/>  
Último acceso: Junio 2015





## **Capítulo 10. Anexos**



## 10.1. Contenido del CD

El contenido del CD que se adjunta con este documento es el siguiente:

- Documento en formato PDF cuyo nombre es *Memoria\_TFG* y que corresponde a este mismo documento. Dentro de este mismo documento se encuentra la Guía de instalación y el Manual de Usuario de la aplicación.
- Dos archivos instalables de las aplicaciones, uno perteneciente a la aplicación Java y que recibe el nombre *HDTproyectoJAVA.apk* y el otro, propio de la aplicación C++, cuyo nombre es *HDTproyectoCPP.apk*
- Un directorio llamado *Software* que contiene el código de los proyectos desarrollados, dentro de este directorio hay dos subdirectorios, uno llamado *HDTproyectoJAVA*, cuyo contenido es el código fuente de la aplicación Java, y el otro llamado *HDTproyectoCPP*, que contiene además del código fuente de la aplicación C++ la librería nativa desarrollada.
- Un directorio llamado *Librerías* cuyo contenido son, la librería *hdt-lib.jar* modificada para este proyecto, y dos carpetas, una con el nombre *HDT-Java*, que contiene todos los elementos necesarios para generar una librería nativa adaptada a Java, y la otra, cuyo nombre es *HDT-Android*, cuyo contenido son los archivos necesarios para generar la librería nativa adaptada a Android.
- El conjunto de datos de **LinkedMDB** del año 2010 en formato HDT preparado para realizar consultas sobre el.

## 10.2. Entorno de desarrollo

En este apartado vamos a detallar la instalación de las herramientas y aplicaciones que nos han sido necesarias para llevar a cabo este proyecto

Para poder desarrollar una aplicación para un dispositivo móvil, nos va a ser necesario preparar nuestro ordenador de forma que sea capaz de realizar dicha tarea, cabe destacar que el desarrollo de este proyecto lo hemos realizado íntegramente en una instalación “limpia” de Ubuntu, cuya versión es la 12.04 LTS de 64 bits.

En primer lugar es necesario que dispongamos de Java en nuestro ordenador, puesto que es un requisito necesario para la instalación de las siguientes herramientas. En nuestro caso optamos por instalar la versión 7 de *OpenJDK*, esta se divide en dos partes la JRE, o “Java Runtime Environment” y el JDK, o “Java Development Kit”. A continuación explicaremos brevemente cada una:

- **JRE** es el entorno en tiempo de ejecución de Java está conformado por una Máquina Virtual de Java o JVM, un conjunto de bibliotecas Java y otros componentes necesarios para que

una aplicación escrita en lenguaje Java pueda ser ejecutada. El JRE actúa como un "intermediario" entre el sistema operativo y Java.

- **JDK** es un software que nos provee de las herramientas de desarrollo necesarias para la creación de programas en Java, también incluye un compilador para Java.

Para instalar ambas herramientas abrimos una terminal e introducimos las siguientes instrucciones:

```
sudo apt-get install openjdk-7-jre
```

```
sudo apt-get install openjdk-7-jdk
```

Ahora que ya tenemos Java en nuestro ordenador, el siguiente paso será instalar Eclipse, para ello nos dirigimos hasta su página oficial y descargamos la versión que deseemos, en nuestro caso escogimos la versión "Eclipse Luna for java developers". Es importante que nos fijemos en si nuestro sistema operativo es de 32 o 64 bits antes de descargarnos la herramienta.

Una vez lo descargamos, tendremos que descomprimirlo en alguna carpeta que nos permita utilizar el entorno de manera global, en nuestro caso lo descomprimos en la carpeta "/opt/" para ello introducimos la siguiente orden en la terminal:

```
cd /opt/ && sudo tar -zxvf ~/Descargas/eclipse-*.tar.gz
```

Por ultimo para poder hacer uso de esta herramienta fácilmente, creamos un acceso directo, para ello vamos a generar un fichero, en el que especificaremos algunas opciones referentes a la aplicación. Introduciremos en la terminal la siguiente instrucción:

```
gksudo gedit /usr/share/applications/eclipse.desktop
```

En el fichero que se acaba de abrir tendremos que introducir lo siguiente:

```
[Desktop Entry]
Name=Eclipse
Type=Application
Exec=/opt/eclipse/eclipse
Terminal=false
Icon=/opt/eclipse/icon.xpm
Comment=Integrated Development Environment
NoDisplay=false
Categories=Development;IDE;
Name[en]=Eclipse
```

En el primer inicio de Eclipse, nos preguntan el lugar donde queremos almacenar todos nuestros proyectos, el directorio en cuestión recibe el nombre de "workspace", por comodidad el nuestro se encuentra en la carpeta personal.

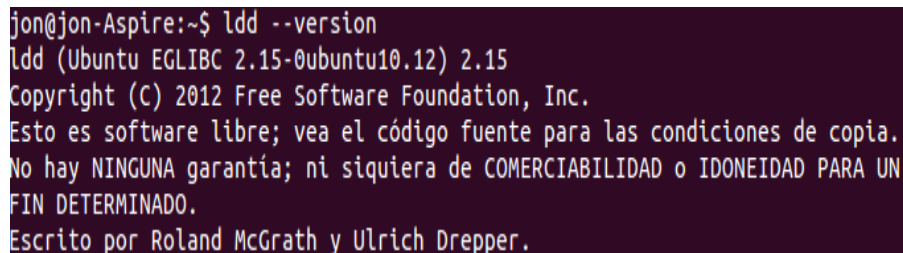
Ahora que disponemos de Eclipse, nos vamos a centrar en la instalación del SDK de Android. En primer lugar deberemos asegurarnos de que nuestro sistema dispone de los siguientes requisitos mínimos para poder ejecutar la herramienta:

- GNU C Library (glibc) 2.15 o superior.
- 2 GB RAM mínimo, 4 GB RAM recomendado.
- 400 MB de espacio libre en el disco duro.
- 1280 x 800 resolución mínima de pantalla.
- OpenJDK versión 7.

Para comprobar la versión de la librería de C que estamos ejecutando, basta con introducir en la terminal la siguiente instrucción:

```
ldd --version
```

En la Ilustración 66 mostramos la salida que se ha producido tras la ejecución de dicha orden:

A terminal window showing the output of the 'ldd --version' command. The output text is: 'ldd (Ubuntu EGLIBC 2.15-0ubuntu10.12) 2.15', 'Copyright (C) 2012 Free Software Foundation, Inc.', 'Esto es software libre; vea el código fuente para las condiciones de copia.', 'No hay NINGUNA garantía; ni siquiera de COMERCIABILIDAD o IDONEIDAD PARA UN FIN DETERMINADO.', and 'Escrito por Roland McGrath y Ulrich Drepper.'

```
jon@jon-Aspire:~$ ldd --version
ldd (Ubuntu EGLIBC 2.15-0ubuntu10.12) 2.15
Copyright (C) 2012 Free Software Foundation, Inc.
Esto es software libre; vea el código fuente para las condiciones de copia.
No hay NINGUNA garantía; ni siquiera de COMERCIABILIDAD o IDONEIDAD PARA UN
FIN DETERMINADO.
Escrito por Roland McGrath y Ulrich Drepper.
```

Ilustración 66: Versión de la librería C

Ahora que ya sabemos que cumplimos todos los requisitos mínimos para poder instalar el SDK, podemos proseguir.

En primer lugar tendremos que descargar el SDK de la página oficial de Google, podemos descargarlo a través del enlace que viene a continuación, como vamos a incorporarlo a Eclipse, elegiremos la versión “**stand-alone**”:

<https://developer.android.com/sdk/index.html>

Cuando iniciemos la descarga, nos dirigirán a una guía de instalación. Si nos fijamos, hay un problema con Ubuntu, y es que si vamos a instalar el SDK en un sistema operativo de 64 bits, es necesario que instalemos algunos paquetes adicionales. Para las versiones 13.10 de Ubuntu y posteriores tenemos que instalar los paquetes *libncurses5:i386*, *libstdc++6:i386*, y *zlib1g:i386*. Para versiones anteriores a la 13.10, tendremos que instalar el paquete *ia32-libs*. Como en nuestro

caso nos encontramos en Ubuntu 12.04, abriremos una terminal e introduciremos el siguiente comando:

```
sudo apt-get install ia32-libs
```

Ahora que ya tenemos todo listo, el siguiente paso es descomprimir el archivo que acabamos de descargar donde nosotros queramos. Una vez hecho esto nos moveremos, con la terminal, hasta la carpeta que acabamos de descomprimir, concretamente tendremos que meternos en el directorio “**tools**”, una vez que nos encontremos ahí, introduciremos el comando “**./android**”, nos tendrá que aparecer una ventana igual que la de la Ilustración 67:

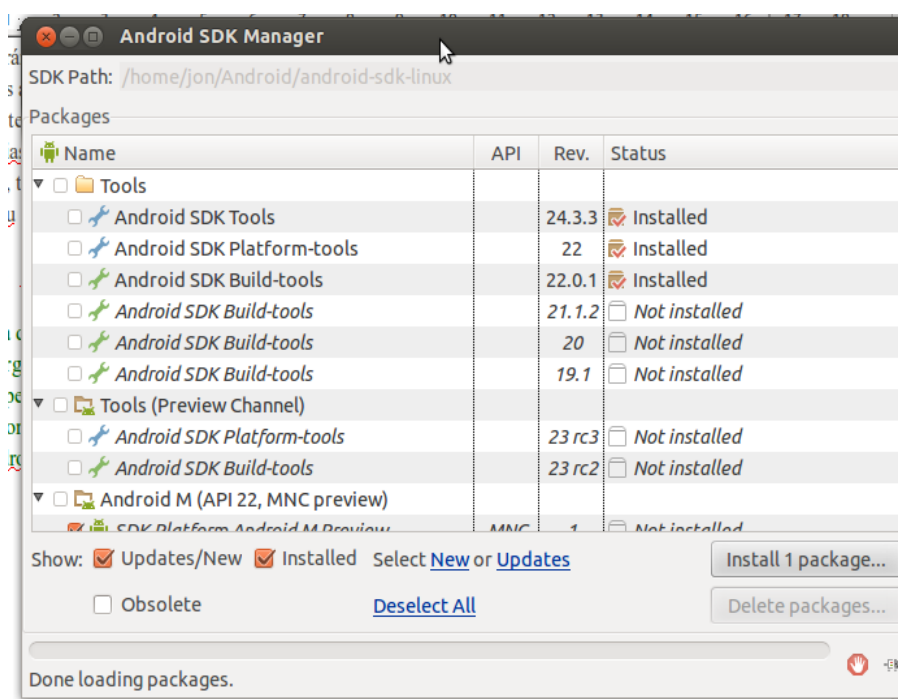


Ilustración 67: Pantalla inicial del SDK Android

A través de esta utilidad, llamada **SDK Manager**, podremos descargarnos las diferentes herramientas y plataformas que proporciona Google para el desarrollo de aplicaciones Android.

Es tan sencillo como seleccionar aquellos componentes que deseamos instalar en nuestra maquina y pulsar el botón “**Install**”. Hay una serie de paquetes que tienen que ser instalados de forma obligatoria, estos son, “**SDK Tools**”, “**SDK Platform-tools**” y “**SDK Platform**”. También es necesario instalar la última API que se haya implantado, ya que al ser la más reciente, será la versión que contenga los últimos cambios y por tanto, Eclipse la utilizará para compilar y depurar nuestro proyecto. Como ya sabemos, también nos será necesario instalar los paquetes pertenecientes a la **API 19** y la **API 15**, puesto que serán las APIs en las que se ejecutará nuestra aplicación.

Una vez descargados todos los paquetes, podemos cerrar el SDK Manager. El siguiente paso será añadir al **PATH**, la ruta de las carpetas “**tools**” y “**platform-tools**”, que podemos encontrar en el directorio donde hayamos descomprimido el SDK. Nosotros introducimos la siguiente orden en la terminal.

```
gedit ~/.bashrc
```

E introducimos al final del archivo las siguientes líneas:

```
export PATH=${PATH}:~/Android/android-sdk-linux/tools  
export PATH=${PATH}:~/Android/android-sdk-linux/platform-tools
```

Con este último paso, ya podemos pasar a añadir todo lo que acabamos de descargar e instalar, a nuestro entorno de desarrollo Eclipse.

Para realizar esta tarea será necesario un *plugin*, conocido como **ADT**, o **Android Developer Tools**. Este *plugin* proporciona un entorno integrado en el que desarrollar aplicaciones de Android, básicamente extiende las capacidades de Eclipse para que este pueda construir nuevos proyectos para Android, elaborar una interfaz de usuario de aplicación, depurar la aplicación, y añadir el firmado, si lo deseamos, de los paquetes de aplicaciones (archivos APK) para su distribución.

Abrimos Eclipse y en la barra de herramientas superior nos dirigimos a la pestaña “Help”, dentro de ella seleccionamos la opción “Install New Software”. Una vez nos encontramos ahí tendremos que añadir un nuevo repositorio, cuando hagamos clic en el botón “Add” nos aparecerá una ventana en la que introduciremos el nombre que vamos a proporcionar al repositorio y la dirección URL desde la que se descargará el *plugin*, la dirección que introduciremos es la siguiente:

```
https://dl-ssl.google.com/android/eclipse/
```

Una vez hecho esto hacemos clic en “Ok”, en la ventana que nos aparecía antes elegiremos el repositorio que acabamos de añadir del desplegable que aparece en la parte superior. Una vez le hemos elegido, nos encontraremos con la ventana que se muestra en la Ilustración 68:

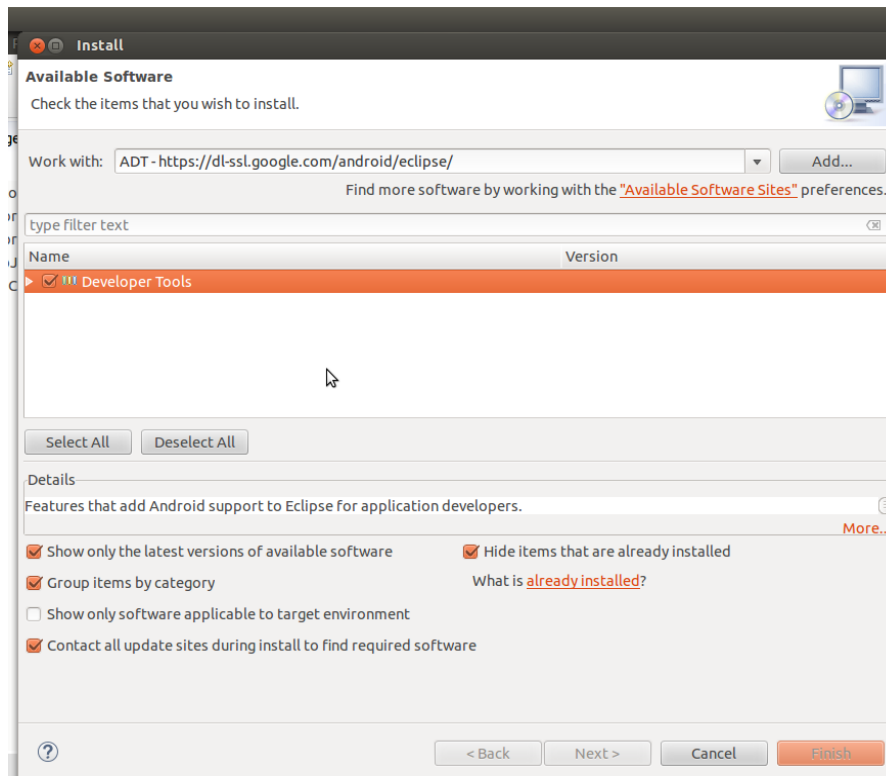


Ilustración 68: Ventana para agregar plugins de Eclipse

Seleccionamos el “checkbox” de “Developer Tools” y pulsamos en “Next”, en la siguiente ventana veremos la lista de herramientas que van a ser descargadas. Por ultimo aceptaremos los términos de la licencia y haremos click en “Finish”.

Durante la instalación nos aparecerá un aviso de seguridad diciéndonos que la autenticidad o validez del software no puede ser establecida, omitiremos este mensaje y proseguiremos con la instalación. Una vez se complete reiniciaremos Eclipse.

Cuando Eclipse se reinicie, nos aparecerá una ventana de bienvenida a Android, en ella se nos da la posibilidad de elegir un SDK existente o descargarlo, en nuestro caso seleccionaremos la opción “Use existing SDK”, en ella tendremos que especificar la ruta exacta del directorio donde se encuentre la carpeta del SDK, una vez seleccionada, haremos click en “Next” y de esta forma habremos acabado con la instalación del SDK en Eclipse.

El siguiente mecanismo que tendremos que instalar recibe el nombre de **CDT, o C/C++ Development Tooling**, se trata de una herramienta que proporciona al entorno Eclipse el soporte completo de los lenguajes C y C++. Además añade la creación y gestión de varias “toolchains”, el estándar de make para construir proyectos, y varias herramientas más para la programación y compilación.



Para instalar el CDT, seguiremos los mismos pasos que con el ADT, por lo que para evitar repetir la misma información, únicamente vamos a proporcionar el repositorio de donde se puede obtener dicha herramienta para la versión Luna de Eclipse:

<http://download.eclipse.org/tools/cdt/releases/8.6>

Por último, la herramienta que instalaremos a continuación es esencial para poder desarrollar la aplicación con el lenguaje C++, hablamos de la herramienta **NDK, o Native Development Kit**. Podemos descargarnos esta herramienta a través del siguiente enlace:

<https://developer.android.com/ndk/downloads/index.html>

Una vez lo hayamos descargado tendremos que dirigirnos hasta la carpeta que contenga la descarga desde el terminal, e introducir las siguientes instrucciones con permisos de superusuario:

```
chmod a+x android-ndk-r10d-linux-x86_64.bin
```

```
./android-ndk-r10d-linux-x86_64.bin
```

Una vez hayamos ejecutado dichas instrucciones obtendremos una carpeta que contendrá todos los archivos del NDK descomprimidos. El siguiente paso será el de añadir la ruta de esta carpeta a Eclipse. Abrimos Eclipse y en la barra de herramientas superior elegimos la opción “Window” y dentro de ella seleccionamos “Preferences”. Cuando nos en el menú de las preferencias desplegamos la pestaña Android, y elegimos la opción NDK donde deberemos escribir la ruta donde hayamos descomprimido la carpeta. Quedando de la forma que muestra la Ilustración 69:

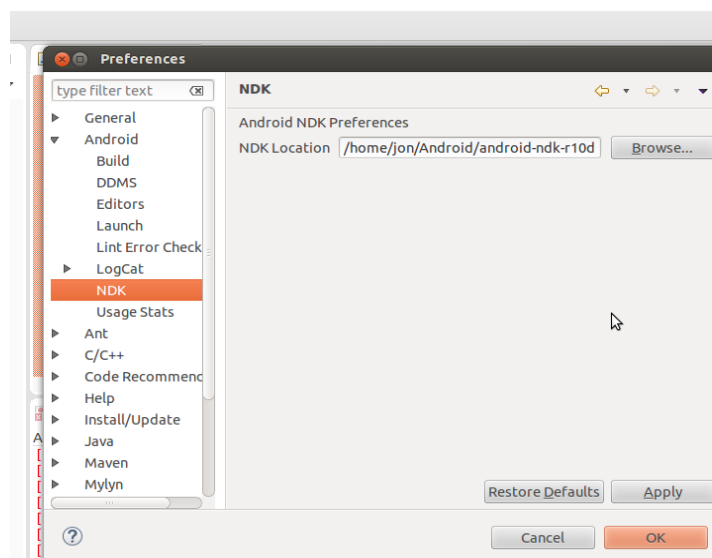


Ilustración 69: Pantalla para añadir el NDK a Eclipse

Por último añadiremos la carpeta del NDK al PATH, para que podamos hacer uso de sus herramientas de forma global, al igual que hicimos con el SDK, modificaremos el fichero “bashrc” y añadiremos la siguiente línea:

```
export PATH=${PATH}:~/Android/android-ndk-r10d
```

### 10.3. Continuación de la Adaptación a Java

En este apartado continuaremos con el proceso de adaptación a Java de la librería nativa, una vez completemos los dos pasos que nos quedan, seremos capaces de probar las capacidades de nuestra librería nativa desde cualquier dispositivo que disponga de una JVM.

#### 10.3.1. Crear la clase “main” del proyecto

Para poder comprobar todo el código que hemos realizado, nos será necesaria una clase que llamaremos “test”, y en la cual crearemos una búsqueda realizando llamadas a los métodos de cada clase Java que hemos creado anteriormente, quedando de forma muy similar al código de búsqueda que encontramos en los códigos fuente de la librería C++.

La clase test de Java tendrá el siguiente aspecto:

```
import java.util.Scanner;
public class test{
public static void main (String[] args){
Scanner in = new Scanner(System.in);
String suj, pred, obj;
String file =
"/home/jon/Escritorio/ProyectoHDTcJNI/jniHDT/biblioteca/linkedmdb.
hdt";
HDTManager loader = new HDTManager();
HDT hdt = loader.cargarHDT(file);
System.out.println("Por favor, introduzca el sujeto de la
búsqueda, escriba un 0 si desea introducir \"\");
suj = in.nextLine();
System.out.println("Por favor, introduzca el predicado de la
búsqueda, escriba un 0 si desea introducir \"\");
pred = in.nextLine();
System.out.println("Por favor, introduzca el objeto de la
búsqueda, escriba un 0 si desea introducir \"\");
```

```

obj = in.nextLine();
if(suj.equals("0")){
suj = "";
}
if(pred.equals("0")){
pred = "";
}
if(obj.equals("0")){
obj = "";
}
IteratorTriple its = hdt.buscar(suj,pred,obj);
while(it.hasNext()){
TripleString triple = its.next();
System.out.println("Sujeto: " + triple.getSujeto());
System.out.println("Predicado: " + triple.getPredicado());
System.out.println("Objeto: " + triple.getObjeto());
}
} //Fin del main
}

```

Como podemos ver nos es necesario importar el paquete *util.Scanner* para introducir datos por teclado y poder realizar la búsqueda. Surge un problema, y es que si el usuario desconoce el tipo de información que tiene que introducir, por ejemplo, en el sujeto, debe introducir un espacio en blanco. Como la clase *Scanner* no permite que las variables *suj*, *pred* y *obj* tengan como valor un campo vacío, es necesario que comuniquemos al usuario que si quiere omitir la inserción de información del sujeto, introduzca un cero y el programa le asignara unas dobles comillas.

### 10.3.2. Compilar y probar la librería nativa

Pasamos ya a este punto final, en el que tenemos que compilar y enlazar nuestro *wrapper* JNI con la librería C++. Para hacer esta compilación más sencilla, vamos a crear un *makefile*, de forma que cada vez que hagamos un cambio únicamente tengamos que ejecutar este archivo. Cabe destacar que para cada ordenador habrá que revisar diferentes aspectos de este fichero, en nuestro caso queda de la siguiente forma:

```
LIBRARY := libhdt-jni.so
```

```

OBJFILES := src/HDTManagerW.o src/HDTw.o src/HDTiteratorW.o
src/HDTtripleW.o

INCLUDES := -I"/usr/lib/jvm/java-7-openjdk-amd64/include"
-I"/usr/lib/jvm/java-7-openjdk-amd64/include/linux" -I./hdt-
lib/includes/

CXXFLAGS := -fPIC -O3 $(INCLUDES)

LIB=./hdt-lib/lib/libhdt.a

$(LIBRARY): $(OBJFILES)

$(CXX) -shared -o $@ $^ ${LIB}

clean:
@echo " [CLN] Removing object files"
@rm -f $(OBJFILES) src/*~ *~

```

De este *makefile* hay que hacer especial hincapié en la variable **INCLUDES**, en la cual se establecen las rutas que contienen las cabeceras necesarias para la compilación de nuestro *wrapper* y la incorporación de la librería C++, *libhdt.a*, que contiene el código necesario para poder utilizar los métodos nativos.

A continuación ejecutamos la orden *make* donde se encuentre nuestro *makefile*, compilamos la clase test mediante la orden *javac hdt/proyecto/cpp/\*.java*, y por último ejecutamos la clase test con la siguiente orden:

```
java -Djava.library.path=../ -classpath . hdt.proyecto.cpp
```

Si todo ha funcionado correctamente debería aparecer en la terminal una serie de preguntas que nos soliciten introducir el sujeto, el predicado y el objeto de la búsqueda que queramos realizar, y acto seguido debería aparecer por pantalla los resultados que LinkedMDB contenga que esten relacionados con los parámetros introducidos.