UNIVERSIDAD
REY JUAN CARLOS

Ingeniería Superior de Telecomunicación

Curso Académico 2011/2012

Proyecto Fin de Carrera

Sistema transparente de descarga de libros para e-readers basado en FUSE

Autor: César López Ramírez

Tutor: Dr. Gregorio Robles

Índice general

1. Introducción **13**

1.1. Contexto del PFC 13

1.2. Motivaciones 14

1.3. Introducción al almacenamiento en la nube 14

1.4. El sistema Terabox de Telefónica 17

1.5. Otros sistemas de almacenamiento en la nube 18

1.6. El e-reader Irex Iliad 20

2. Objetivos **25**

3. Diseño e implementación **29**

3.1. Estructura del capítulo 29

3.2. Fase 1: Usar FUSE en Irex Iliad 29

3.2.1. Introducción a FUSE 29

3.2.2. Justificación del uso de FUSE 30

3.2.3. FUSE en el Iliad 31

3.2.4. Compilando FUSE mediante compilación cruzada 32

3.2.5. Activar el paquete de desarrollo en el Irex Iliad 34

3.2.6. Instalación de la shell y del servidor SSH 35

3.2.7. Instalación y prueba de FUSE en el libro electrónico 35

3.2.8. Resultados de la primera fase 36

3.3. Fase 2: Usar el API de Terabox en el libro 36

3.3.1. Introducción al API de Terabox 37

3.3.2. Compilación del API de Terabox para el ordenador de escritorio 37

3.3.3. Estructura general de un programa con el API de Terabox 38
3.3.4. Prueba del API de Terabox para el ordenador de escritorio 38
3.3.5. Compilación del API de Terabox para el dispositivo Iliad 39
3.3.6. Prueba del API de Terabox en el libro electrónico 39
3.4. Fase 3: Usar FUSE en el libro 40
3.4.1. Implementar operaciones de FUSE 41
3.4.2. Integrar FUSE con el API de Terabox 43
3.4.3. Primeras pruebas de la integración de FUSE con el API de Terabox . . 45
3.4.4. Análisis de las pruebas de integración FUSE - Terabox 47
3.5. Fase 4: Implementar un sistema básico de caché 47
3.5.1. Descripción pormenorizado del funcionamiento de la operación de la caché 48
3.5.2. Pruebas entre distintas maneras de descargarse los archivos de Terabox 50
3.6. Fase 5: Implementación del caso ”offline” 51
3.6.1. Introducción a la funcionalidad necesaria 51
3.7. Fase 5.1: Comprobar la conectividad a Internet 52
3.7.1. Descripción de la implementación de la comprobación de Internet . . . 54
3.8. Fase 5.2.- Representación de la estructura de directorios 56
3.8.1. Descripción de la implementación 58
3.9. Fase 5.3.- Almacenamiento de la estructura de directorios 62
3.9.1. Introdución al concepto de la serialización 62
3.9.2. Descripción de la serialización 62
3.9.3. Elección del mecanismo de almacenamiento: GNU dbm 63
3.9.4. Implementación de la serialización 64
3.10. Fase 6.- Integración con la interfaz gráfica del Irex Iliad 69
3.10.1. Descripción de la interfaz del Irex Iliad 69
3.10.2. Implementación de la integración en la interfaz gráfica 72
3.11. Fase 7.- Integración de la aplicación con el módem 3G 78

4. Conclusiones **83**

4.1. Evaluación de los objetivos 83
4.2. Lecciones aprendidas 87

4.3. Trabajos futuros 87

A. Apéndice 1: Sintaxis del comando AT +CGDCONT **89**

Bibliografía **91**

Índice de figuras

1.1. Esquema de un sistema cloud-computing genérico 16
1.2. Imagen de un Irex Iliad 21

3.1. Arquitectura de FUSE 31
3.2. Pila de protocolos de libddvs 37
3.3. Diagrama de estados de un proceso en Linux 53
3.4. Esquema de un arbol n-ario 57
3.5. Interfaz gráfica del Irex Iliad 70
3.6. Icono e-book PDF 71
3.7. Icono e-book PDF (no disponible) 71
3.8. Icono e-book Mobipocket (no disponible) 71
3.9. Icono e-book TXT (no disponible) 72

Índice de cuadros

1.1. Comparativa de sistemas de almacenamimento en la nube 20

3.1. Resumen de los tipos de e-book 73
3.2. Cambios en el fichero de propiedades de la conexión 3G (vodafone) 79

A.1. Sintaxis de los parámetros del comando +CGDCONT 89

Resumen

Este proyecto fin de carrera consiste en una aplicación móvil embebida del sistema de almacenamiento en la nube Terabox de Telefónica. El dispositivo móvil objeto de la aplicación es el Irex Iliad, uno de los primeros dispositivos e-readers comerciales y que cuenta con gran soporte para el desarrollo. Con esta aplicación se persigue que de manera totalmente transparente para el usuario se vean libros electrónicos que no están físicamente en un primer momento en el e-reader. Mediante un modem 3G adosado a un puerto USB se descargan los libros en el momento de su visualización y se quedan almacenados en una caché por si posteriormente se quieren volver a visualizar y no hay conectividad 3G o no se desea utilizar ésta. El nombre interno de la aplicación es ”Terabook” que viene la unión de las palabras Terabox y e-book.

Capítulo 1

Introducción

1.1. Contexto del PFC

Este proyecto se realizó en las dependencias de Telefónica I+D. El proyecto empezó en diciembre de 2009. Para ello se realizó una beca de prácticas en empresas de 6 meses con un desempeño diario de 4 horas. La división concreta en la que se realiza la beca es la número 6112 dirigida por Joaquín López (joaquin@tid.es). El tutor del proyecto de cara a Telefónica es Javier Puga (jgarcia@tid.es). Además se contó con el apoyo del trabajador externo Roberto Pérez Cubero (robj.perez@gmail.com). Este último fue el que realizó la demostración de la tecnología desarrollada por este PFC a directivos de Telefónica Móviles con resultados bastante satisfactorios. La división en esos momentos estaba inmersa en un proyecto de alta repercusión llamado internamente ”Pincho infinito” o por su posible nombre comercial ”3G Box”.[1] La beca finalizó en mayo de 2010 pero el proyecto se continuó probando y desarrollando.

En este proyecto se buscaba tener un pendrive que almacenara la información en el sistema de almacenamiento on-line Terabox de Telefónica. El hecho de que hiciera esto de manera transparente para el usuario hace que éste experimentara tener un pendrive una capacidad ”infinita” (de ahí el nombre interno). Dentro de la división fui considerado como un trabajador más con el mismo nivel de exigencia que el resto de los empleados. Aunque parte de las soluciones empleadas para ”3G Box” se podían utilizar en el proyecto, decidimos no compartir la información con el fin de que me enfrentara a los problemas de nuevas. De esta manera es posible que pudiera encontrar soluciones distintas a las que habían empleado ellos en ”3G Box” y en el

[1] `http://www.elmundo.es/elmundo/2010/02/18/navegante/1266487640.html`

caso de que éstas fueran mejores se podrían utilizar en este proyecto. Por otro lado al constituir éste un Proyecto Fin de Carrera en el que tuviera que desarrollar mis propias soluciones, maximizaba las posibilidades académicas del proyecto. Este proyecto se enmarcaba en la línea de la división de ofrecer servicios basados en Terabox para libros electrónicos. Los servicios para libros electrónicos constituían una posible línea de negocio con un gran potencial en un futuro. En el momento de escribir esta memoria la división había sido disuelta y su personal reasignado a otros proyectos y no tengo noticia de que ”3G Box” se haya convertido en un producto comercial. Las razones para esto último serán tratadas en el apartado de ”Conclusiones”. Sin embargo sí han salido clientes de Terabox para iOS y Android.

1.2. Motivaciones

Dentro de las opciones posibles que me ofrecía Telefónica I+D, este proyecto era el más atractivo por el hecho de tener que programar en Linux en un entrno embebido y también por tener unos objetivos definidos claros. Por otro lado compartía la visión de la división de que los e-readers son un campo con mucho futuro. Una de mis motivaciones era poder hacer una aplicación de código libre, sin embargo, esto no pudo ser posible.

1.3. Introducción al almacenamiento en la nube

El almacenamiento en la nube es un modelo de almacenamiento en red y en línea donde los datos se almacenan en varios servidores virtuales, por lo general organizados por terceros, en lugar de ser alojados en servidores dedicados. Estas compañías operan grandes centros de datos y las personas que necesitan guardar información compran o arriendan capacidad de almacenamiento a estas empresas y lo utilizan para sus necesidades. Los operadores de los centros de datos virtualizan los recursos de acuerdo a los requerimientos del cliente y permiten que los clientes puedan utilizar este servicio para almacenar archivos u objetos de datos. Físicamente, la información puede extenderse a lo largo de varios servidores.

Los servicios de almacenamiento en la nube pueden accederse a través de una interfaz de programación de aplicaciones (API), o a través de una interfaz de usuario basada en la web.

Ventajas de almacenamiento en la nube (extraído de la referencia bibliográfica [1]):

- Las compañías sólo necesitan pagar por el almacenamiento que realmente utilizan.

- Las empresas no necesitan instalar dispositivos físicos de almacenamiento en sus centros de datos o en las oficinas, lo que reduce los costos de IT y hosting.

- Las tareas de mantenimiento, tales como la copia de seguridad, la replicación de datos, y la compra de dispositivos adicionales de almacenamiento es ahora responsabilidad de un proveedor de servicios, permitiendo a las organizaciones a centrarse en su negocio principal.

Desventajas o principales problemas:

- La seguridad de los datos almacenados y los datos en tránsito pueden ser una preocupación cuando se almacenan datos sensibles en un proveedor de almacenamiento en la nube.

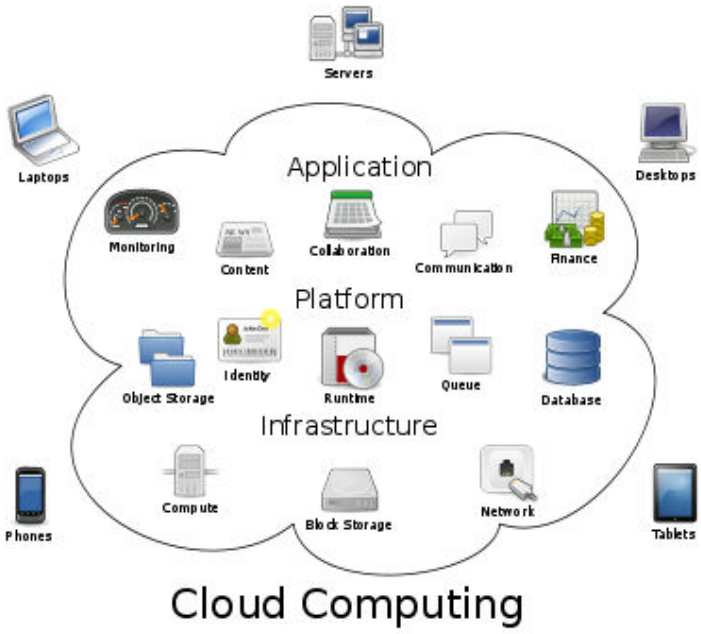
- El rendimiento puede ser menor comparado al almacenamiento local.

- La fiabilidad y la disponibilidad depende de la disponibilidad de red y en el nivel de las precauciones tomadas por el proveedor de servicios.

- Los usuarios con determinados requisitos de registro, tales como los organismos públicos que deben conservar los registros electrónicos de acuerdo a la ley, pueden tener complicaciones con el uso de la computación en nube.

En la figura 1.1 podemos ver como el concepto de cloud computing se puede usar en diferentes niveles de funcionalidad. Además por su propia naturaleza facilita la ubicuidad como también podemos ver en la figura.

Figura 1.1: Esquema de un sistema cloud-computing genérico



Los sistemas de almacenamiento en la nube se pueden clasificar (Cáp. 15.2 de la referencia [1]) en administrados o “managed” en inglés y sin administración o “unmanaged”. En los sistemas sin administración el proveedor del servicio pone su capacidad de almacenamiento disponible a los usuarios, pero define la naturaleza de este almacenamiento, como puede ser utilizado y por qué aplicaciones. Las opciones que el usuario tiene que administrar en está categoría de almacenamiento están tremendamente limitadas. No obstante el almacenamiento no administrado es fiable, relativamente barato de usar y particularmente fácil de usar.

El almacenamiento administrado es principalmente usado por desarrolladores para su uso en aplicaciones construidas usando servicios web. El almacenamiento en la nube administrado se provee y se ofrece como un disco en crudo. Se deja al usuario montar y desmontar el disco, formatearlo, hacer particiones y permitir que los recursos de almacenamiento estén disponibles a otros usuarios. Ejemplos de este tipo de almacenamiento en la nube son Rackspace, Amazon S3 y Google Storage.

En este proyecto nos centraremos en Terabox que es un servicio de almacenamiento sin administración.

1.4. El sistema Terabox de Telefónica

El sistema Terabox de Telefónica permite guardar los datos, compartirlos con cualquier personas y acceder a estos desde cualquier lugar. Por tanto elimina la necesidad de usar discos duros externos o pendrives. También permite tener los datos a salvo en situaciones tales como: rotura del ordenador, fallo del disco duro, borrado accidental de ficheros o pérdida de pendrive. Al utilizar la tecnología de cifrado SSL, los datos viajan seguros por Internet. El acceso a la información se vuelve ubicuo al poder acceder a la misma desde el ordenador de casa, el portátil o el móvil. La compartición de los datos se vuelve muy simple al poder crear álbumes de fotos, carpetas compartidas o simplemente poder enlazar a los ficheros almacenados desde cualquier punto de Internet. Además se pueden compartir las carpetas para que solo tenga acceso un determinado grupo, configurar permisos de lectura / escritura y fijar fecha de vencimiento. Uno de los inconvenientes principales de la alta capacidad del almacenamiento online es la organización de la información. Terabox soluciona esto mediante la creación de álbumes. Estos permiten ordenar fotos, vídeos o música de manera simple y sencilla. La capacidad de 100 GB es gratuita para clientes de banda ancha, 200 GB costaría 1,5€ al mes, 500 GB 3€ al mes y la capacidad máxima de un terabyte (de ahí viene el nombre de Terabox) 5€ al mes. Tiene soporte oficial Windows, Mac OS, iOS y Android. Por último Movistar ofrece atención al usuario mediante el 1004 así como mediante el sistema de Atención Técnica Online. En cualquiera de las dos opciones un operador resuelve las dudas y preguntas del usuario. La funcionalidad incluida en las aplicaciones móviles (iOS y Android) incluye:

- Gestión de ficheros: moverlos, borrarlos, renombrarlos, etc.
- Acceso en cualquier momento y lugar de documentos, fotos, vídeos, música, etc.
- Subida de fotos y vídeos directamente desde el móvil
- Creación de álbumes personales con fotos y vídeos o listas de reproducción con archivos musicales.

1.5. Otros sistemas de almacenamiento en la nube

Dropbox El cliente de Dropbox permite a los usuarios dejar cualquier archivo en una carpeta designada. Ese archivo es sincronizado en la nube y en todas las demás computadoras del cliente de Dropbox. Los archivos en la carpeta de Dropbox pueden entonces ser compartidos con otros usuarios de Dropbox o ser accedidos desde la página Web de Dropbox. Asimismo, los usuarios pueden grabar archivos manualmente por medio de un navegador web. Si bien Dropbox funciona como un servicio de almacenamiento, se enfoca en sincronizar y compartir archivos. Tiene soporte para historial de revisiones, de forma que los archivos borrados de la carpeta de Dropbox pueden ser recuperados desde cualquiera de las computadoras sincronizadas. También existe la funcionalidad de conocer la historia de un archivo en el que se esté trabajando, permitiendo que una persona pueda editar y cargar los archivos sin peligro de que se puedan perder las versiones previas. El historial de los archivos está limitado a un período de 30 días, aunque existe una versión de pago que ofrece el historial ilimitado. El historial utiliza la tecnología de delta encoding. Para conservar ancho de banda y tiempo, si un archivo en una carpeta Dropbox de un usuario es cambiado, Dropbox sólo carga las partes del archivo que son cambiadas cuando se sincroniza. Si bien el cliente de escritorio no tiene restricciones para el tamaño de los archivos, los archivos cargados por medio de la página Web están limitados a un máximo de 300 MB cada uno. Dropbox permite elegir entre tres tipos de cuentas: la primera llamada ”Basic” es gratuita; la segunda, llamada ”Pro50”, y la tercera, llamada ”Pro100”, son de pago. Las diferencias radican en la cantidad de espacio disponible para poder utilizar, mientras la ”Basic” dispone de 2Gb, la ”Pro50” dispone de 50Gb y la ”Pro100” de 100Gb. Los precios de las dos cuentas de pago son 9,99 dólares al mes para la cuenta ”Pro50” y 19,99 dólares al mes para la cuenta ”Pro100”. Dropbox utiliza el sistema de almacenamiento S3 de Amazon para guardar los archivos. La sincronización de Dropbox usa transferencias SSL y almacena los datos mediante el protocolo de cifrado AES-256. Si bien no existe un API oficial, es posible acceder a la información de Dropbox desde otras aplicaciones. Existen plugins para Drupal, WordPress y Joomla! La aplicación, tanto el servidor como las versiones de escritorio de los clientes, están escritos en Python. Es una aplicación privativa salvo la interfaz de usuario de Linux que se integra en Nautilus

y es de código abierto. No obstante necesita para funcionar un demonio que también es privativo. Tiene versiones móviles oficiales para iOS, Android y Blackberry así como versiones de escritorio para Windows, Linux y MacOS. Además la comunidad de usuarios ha creado mash-ups como: Envío de archivos a Dropbox via Gmail; uso de Dropbox para sincronizar logs de mensajería instantánea, administración de BitTorrent, administración de contraseñas, ejecución de aplicaciones remotas y servicio gratuito de alojamiento web.

iCloud Es un sistema de almacenamiento nube o cloud computing de Apple Inc. Anunciado el 6 de junio 2011 en la Conferencia de Desarrolladores Globales de Apple (WWDC por sus siglas en inglés). El servicio permite a los usuarios almacenar datos, como archivos de música en servidores remotos para descargar en múltiples dispositivos como iPhones, iPods, iPads y las computadoras personales que funcionen con Mac OS X (Lion o más reciente) o Windows de Microsoft (Windows Vista o más reciente). El sistema basado en la nube permite a los usuarios almacenar música, fotos, aplicaciones, documentos, enlaces favoritos de navegador, recordatorios, notas, iBooks y contactos, además de servir como plataforma para servidores de correo electrónico de Apple y los calendarios. Cada cuenta tiene 5 GB de almacenamiento gratuito, el contenido comprado de Apple iTunes se almacena de forma gratuita sin interferir en esos 5 GB. Todos los archivos de música comprada a través de iTunes se descargan automáticamente a cualquier dispositivo registrado, por ejemplo, iPhones y computadoras. Cuando un usuario se registra un nuevo dispositivo, todo el contenido de iTunes se puede descargar automáticamente.

Skydrive Forma parte de los servicios de Microsoft llamados Windows Live. SkyDrive permite a los usuarios subir archivos de una computadora y almacenarlos en línea (nube), y acceder a ellos desde un navegador web. El servicio utiliza Windows Live ID para controlar el acceso a los archivos del usuario, y permite mantener la confidencialidad de los archivos, compartir con contactos o compartirlos con el público en general. Los archivos que se comparten públicamente no requieren una cuenta de Windows Live ID para acceder. El servicio ofrece 25 GB de almacenamiento personal gratuito con un tamaño máximo de ficheros de 300 MB. El servicio está construido usando HTML5 y los archivos pueden ser subidos usando ”drag and drop ”. Se puede integrar con otros productos de Microsoft como Hotmail, Office Web/Desktop apps, Bing y Windows Live Groups. Además tiene

soporte móvil para iOS y Windows Phone.

iFolder Está desarrollada por Novell inc. Tiene como novedad frente al resto el ser una aplicación de código abierto tanto el servidor como los clientes. Aunque existen otras aplicaciones open-source para almacenamiento en la nube, iFolder es la única que se integra en Windows, Linux y Mac OS. iFolder trabaja con la idea de compartir carpetas, donde las carpetas son marcadas como compartidas y los contenidos de otras carpetas son sincronizados con otros ordenadores a través de una red, tanto directamente entre ordenadores mediante peer-to-peer, como a través de un servidor. Esto permite un uso individual para compartir los propios archivos entre diferentes ordenadores o con otros usuarios. La última versión soporta SSL, Active Directory, LDAP, acceso web y un asistente para recuperación de claves.

En el cuadro 1.1 se puede ver una comparativa de alguno de los parámetros relevantes para este proyecto de estos sistemas de almacenamiento en la nube.

Sistema	Plataformas	Capacidad (gratis)	Lenguaje de programación	¿Open source?
Dropbox	Windows, Mac, Linux, iOS, Android, Blackberry	2 Gb	Python	Solo interfaz
iCloud	Mac (Lion), iOS	5 Gb	Objective C	No
Skydrive	Windows, iOS, Windows Phone	25 Gb	HTML 5	No
iFolder	Windows, Mac, Linux	Depende de la implementación	Mono	Sí

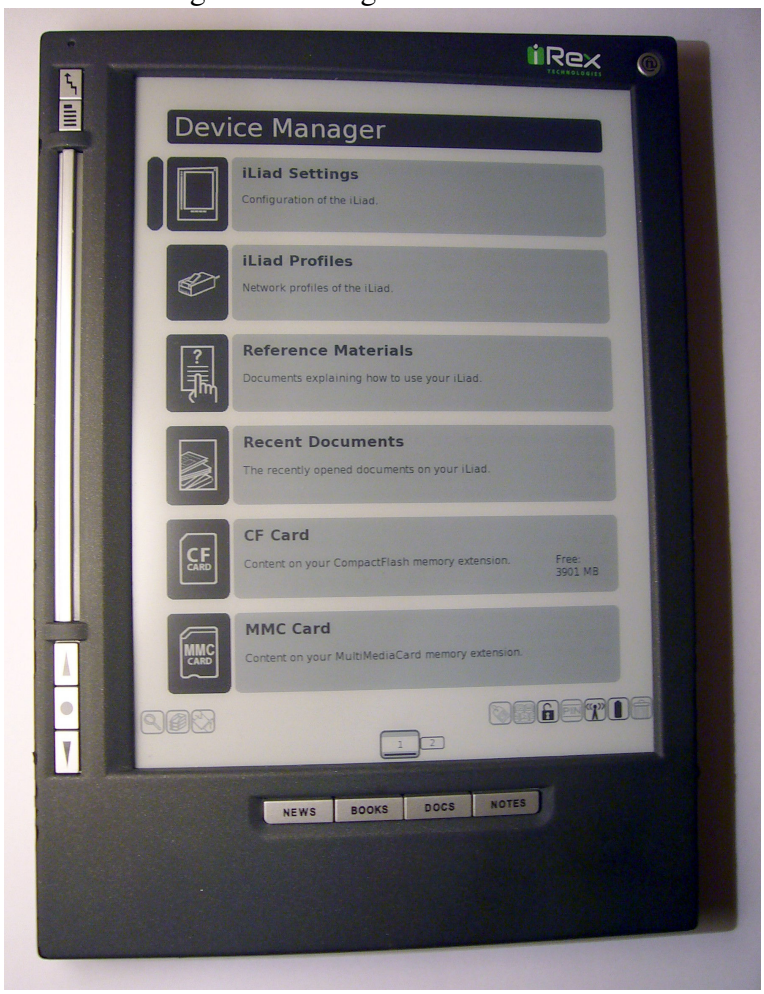
Cuadro 1.1: Comparativa de sistemas de almacenamimento en la nube

1.6. El e-reader Irex Iliad

El modelo de libro electrónico para el que se desarrolla la apliación es el Iliad fabricado por iRex Technologies. El iLiad es un dispositivo de lectura electrónica (e-reader) que puede

ser usado para lectura o edición de documentos. Al igual que otros modelos como Barnes and Noble Nook, Sony Reader o Amazon Kindle, hace uso de la tecnología de tinta electrónica. En 2010, las ventas de Iliad terminaron cuando iRex entró en bancarrota.

Figura 1.2: Imagen de un Irex Iliad



Las especificaciones del dispositivo son las siguientes:

- Pantalla de 20,6 cm de tinta electrónica
- Resolución de 768x1024 píxeles (160dpi)
- 16 niveles de gris
- Puerto USB para almacenamiento externo
- Bahía (ranura) de memoria CompactFlash Type II

- Bahía tarjetas MMC MultiMediaCard
- Entrada de audio stereo de 3.5 mm
- Red inalámbrica WiFi 802.11g LAN
- Puerto de red LAN 10/100MB
- 390g de peso
- Procesador 400MHz Intel XScale
- 64 MB RAM
- 256 MB memoria flash interna (128 para el usuario, 128 reservados para el sistema)
- Sistema operativo basado en Linux con kernel 2.4
- Kit de desarrollo disponible, con lo que la funcionalidad puede ser extendida fácilmente

Mide 155x216x16 mm, el tamaño de un documento A5. La pantalla usa una matriz activa electroforética, la cual emplea la tecnología de tinta electrónica ”E - Ink Vizplex Imaging Film” fabricada por ”E ink Corporation”. Debajo de la pantalla de tinta electrónica está una tableta gráfica por Wacom que requiere un estilete para la entrada de datos. Cuando se introdujo en el mercado, el Iliad tenía el tamaño de pantalla más grande de todos los libros electrónicos.

El iLiad puede mostrar documentos en diferentes formatos, incluyendo PDF, Mobipocket, XHTML y texto plano. También puede mostrar imágenes JPEG, BMP y PNG, pero no en color. Como desde el 3 de mayo de 2007 se soporta el formato Mobipocket, hay un sistema de DRM disponible para la plataforma. El distribuidor del Iliad es iRex Technologies una spin-off de Philips. Se empezó a anunciar en diciembre de 2005 para lanzarse en abril de 2006 pero el lanzamiento finalmente se retrasó hasta julio y desde entonces hubo numerosas actualizaciones de software. Su precio de salida eran 650€ que luego se rebajaron a 600€.

Debido a su sistema operativo Linux abierto, el Iliad es capaz de ejecutar aplicaciones de terceros creadas para él. Los desarrolladores y usuarios que quieren ejecutar o crear aplicaciones pueden pedir el acceso shell al fabricante. Los desarrolladores han sido capaces de mejorar la funcionalidad del dispositivo portando visualizadores cono el FBReader, y programas como

Abiword y Stardict. Con versiones creadas por la comunidad del programa iPDF es posible la visualización de PDFs a pantalla completa. Usuarios independientes también han conseguido portar navegadores web pero con funcionalidad limitada y bastantes fallos.

iRex como compañía tiene una relación bastante fría con los desarrolladores de código abierto. La mayoría de las quejas hacia iRex van en la línea de la lentitud con la que libera los SDK's y otras informaciones del dispositivo. Programas más lúdicos, como sudokus, de reproducción de audio y calendarios han ido apareciendo rápidamente. Sin embargo también es verdad que parte del software interno del Iliad ha sido liberado y que herramientas creadas por la comunidad como el calibrador del estilete han sido incorporadas al software oficial.

La causa que el fabricante aduce para su bancarrota son los retrasos en la homologación del dispositivo en los EEUU por parte de la FCC.

El modelo de libro empleado en este proyecto salió en 2006. Actualmente ya no se fabrica y ni siquiera sigue existiendo el fabricante. Entre los modelos que hoy por hoy se distribuyen en España cabe reseñar los siguientes (los modelos tipo tableta con pantalla LCD no están considerados):

- Sony Reader
- Papyre
- Cybook
- Kindle
- Movistar ebook, FNACbook y Casa Del Libro/Tagus

Esta nueva generación de e-readers tienen mejoras tales como una mayor capacidad de almacenamiento y una tecnología de tinta electrónica más avanzada que junto a procesadores de mayor velocidad permiten un refresco superior y tiempos de respuesta más cortos.

Capítulo 2

Objetivos

El objetivo principal del proyecto fin de carrera es crear una aplicación para el e-reader Irex Iliad que consiga que la descarga de e-books de una aplicación de almacenamiento online se haga de manera transparente para el usuario

Para alcanzar el objetivo principal, identificamos los siguientes subobjetivos:

1. **Representación de la estructura de archivos de Terabox en un sistema de archivos local.**

   En el sistema de almacenamiento en la nube Terabox está organizado en una estructura de árbol en la que existen archivos y directorios. Esta estructura estará representada en un sistema de archivos local. Además de la estructura en sí, estará disponible la información de cada archivo que nos proporciona Terabox como son su tamaño y su fecha de creación.

   No existirá una correspondencia uno a uno entre lo que existe en el sistema de archivos local y lo que hay en Terabox en todo momento. Esto es debido a que no siempre tendremos conexión a Internet. Sin embargo el sistema de archivos local deberá actualizarse tan pronto como esta conexión exista.

   Además, cuando se quiera acceder a uno de esos archivos, en el caso de que exista conexión a Internet, se descargará el contenido y éste estará disponible para el usuario.

2. **Almacenamiento temporal de los archivos en una caché**

   Una vez descargados los archivos, estos permanecerán almacenados en un determinado directorio local del Irex Iliad. Este directorio estará alojado físicamente en un lugar que

tenga suficiente capacidad como ello. Elegiremos como lugar la Compact Flash, pues es el sistema de almacenamiento del Iliad que ofrece una mayor capacidad. Los archivos disponibles en esta caché tendrán una doble función. Cuando exista conexión a Internet acelerarán el acceso a los mismos al evitar tener que descargarlos de nuevo. Cuando esta conexión no exista, permitirán el acceso a los mismos ya que de otra manera sería imposible.

Por último esta caché podrá estar presente o podrá no estarlo. Además podría borrarse, por necesitar el espacio en la Compact Flash por ejemplo, en cualquier momento.

3. **Acceso a la unidad virtual en modo OFFLINE desde el inicio**

   La aplicación debe funcionar desde el inicio, montándose este sistema de archivos en un directorio accesible pulsando una sola tecla del Irex Iliad. Elegimos que esta tecla sea la primera de la botonera inferior, la tecla NEWS.

   Al pulsar el botón del Irex Iliad llamado NEWS se lista el directorio */mnt/free/newspapers*. Es ahí donde se monta el sistema de archivos de nuestra aplicación. Utilizando la interfaz estándar del Iliad, se deberán listar los últimos libros a los que se tuvo acceso la última vez que hubo conexión.

   Deberá visualizarse la diferencia entre los libros que están disponibles y los que no. Por disponibles entenderemos los libros que aun estando en modo OFFLINE (es decir, sin conexión a Internet) podemos leer porque han sido descargados previamente y por tanto están en la caché. Los libros que no están disponibles en el modo OFFLINE son aquellos que estuvieron disponibles la última vez que estuvimos ONLINE (es decir, conectados a Internet).

4. **Apertura de un libro en modo OFFLINE**

   El menú de libros de nuestra aplicación aparecerá al pulsar la tecla NEWS. Ahí se mostrarán los archivos y directorios existentes en Terabox la última vez que se pudo acceder a Internet.

   Un libro que está en caché deberá estar simbolizado por tener el icono habitual. Al hacer doble click sobre él deberá visualizarse con un tiempo de respuesta similar al que tendría un libro que está en local.

Un libro que no está en la caché deberá estar simbolizado con un icono partido por la mitad. Al hacer doble click sobre él deberá visualizarse un mensaje notificando que el libro no está disponible. Además en la descripción del menú principal deberá aparecer un aviso debajo del nombre del libro.

5. **Paso al modo ONLINE al enchufar el módem USB**

   La aplicación estará preparada para trabajar con alguno de los módem USB 3G marca Huawei existentes en la división. En el momento de que se inserte ese módem, el e-reader está en condiciones de poderse conectar a Internet. Esto tiene que ocurrir en un tiempo razonable y sin que el usuario tenga que hacer ninguna acción adicional.

   Además la aplicación tiene que actualizar su árbol de archivos y directorios con lo que en ese momento esté presente en Terabox. Nuevamente esto tiene que ocurrir en una cantidad de tiempo que no sea excesiva.

   No habrá ningún cambio en la pantalla del e-reader si el usuario no está visualizando el menú de libros disponibles en la aplicación. La única prueba de que nos hemos conectado a Internet será que el indicador LED del módem pasa de azul oscuro a azul celeste.

6. **Listado y apertura de archivos en modo ONLINE**

   Como ya estamos en modo ONLINE, los iconos deberán haber cambiado, sustituyéndose los iconos característicos del caso ”no disponible ” por el icono habitual. También en este momento deberán aparecer en los menús los cambios en la estructura del árbol de directorios. Habrá archivos nuevos disponibles y otros que hayan sido borrados del servidor habrán desaparecido. Lógicamente también pueden existir directorios nuevos y directorios borrados.

   Tendremos nuevamente dos casos en los libros mostrados, los que están descargados en la caché y los que están ”disponibles” pero hay que descargarlos de Terabox. Estos dos casos deben estar diferenciados en el menú que lista los libros de una manera integrada en la interfaz.

   El libro que está descargado deberá abrirse inmediatamente como ocurre en el caso OFFLINE. El que se tiene que descargar, deberá hacerlo tras la espera de la descarga, que nuevamente no deberá demorarse demasiado. Por último es necesario aclarar que cuando

nos referimos a tiempos *razonables* o *no excesivos* no tenemos un canon establecido que nos diga cuando es demasiado, sino que simplemente lo dejamos al juicio subjetivo de los componentes de la división.

7. **Paso al modo OFFLINE al desenchufar el módem USB**

   Al desconectar el módem USB, la aplicación debe detectar que ya no existe conexión a Internet y pasará de nuevo al modo OFFLINE. Esto tiene que ocurrir en un tiempo razonable desde el mismo momento de la desconexión del módem.

   También debería detectarse otras situaciones distintas a la desconexión del módem USB pero que supongan una pérdida de la conexión, como por ejemplo quedarse sin cobertura.

   Además, una vez consumado el paso al modo OFFLINE, todo el menú aparecerá modificado reflejando los archivos que están disponibles y los que no. Al ser este proyecto un prototipo no consideraremos el caso ”límite” de apertura de archivos que necesitan ser descargados estando todavía en modo ONLINE aunque el módem esté desconectado ya.

   Por último, no debería interrumpirse al usuario en ningún modo al desconectarse el módem. Con interrumpir me refiero a cualquier interferencia en la experiencia de usuario al leer un libro, por ejemplo. Esto iría en contra de la ”transparencia” que poníamos como objetivo principal.

Capítulo 3

Diseño e implementación

3.1. Estructura del capítulo

La manera de estructurar este capítulo será ir explicando en orden cronológico cada una de las etapas de desarrollo que se fueron haciendo en el proyecto. A medida que vayan apareciendo también se describirán las tecnologías que se vayan usando en cada etapa.

3.2. Fase 1: Usar FUSE en Irex Iliad

El objetivo de esta fase era comprobar que la librería FUSE pudiese funcionar en el libro. Tal y como comento previamente en la sección de ”Contexto del PFC” la información que recibí de mis compañeros de división era la mínima imprescindible para empezar el proyecto. En esta fase me dijeron que existía una librería que se llamaba FUSE y que me podía servir de ayuda. Antes de nada me parece pertintente introducir qué es FUSE.

3.2.1. Introducción a FUSE

FUSE son las iniciales de Filesystem in Userspace. Es un módulo de núcleo para sistemas operativos de tipo Unix que permite a usuarios sin privilegios de root crear sus propios sistemas de ficheros sin necesidad de recurrir a implementar un driver del kernel. Esto se consigue ejecutando el código del sistema de archivos en el espacio de usario mientras que el módulo FUSE proporciona solo un ”puente” a las interfaces del kernel en sí mismas. El módulo del kernel

está liberado bajo los términos de la ”GNU General Public License” (GPL), y la librería con la ”GNU Lesser General Public License” (LGPL), por lo que FUSE es software libre. El sistema FUSE fue originariamente parte de A Virtual Filesystem (AVFS), pero ahora es un proyecto independiente dentro de la forja Sourceforge.net. FUSE está disponible para Linux, FreeBSD, NetBSD, OpenSolaris y Mac OS X. Se incluyó en el kernel de Linux oficial a partir de la versión 2.6.14.

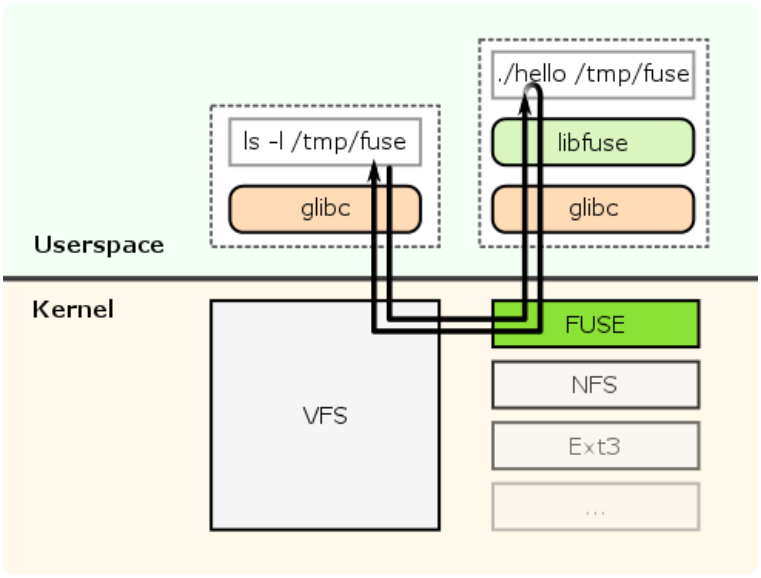
FUSE es particularmente útil para escribir sistemas de archivos virtuales. De manera distinta a los sistemas de archivos tradicionales que esencialmente guardan datos y los obtienen directamente de discos físicos, los sistemas de archivos virtuales no guardan datos por sí mismos realmente. Actúan como una vista o traducción de un sistema de archivos existente o dispositivo de almacenamiento. En principio cualquier recurso disponible para FUSE puede ser exportado como un sistema de archivos.

En la figura 3.1 podemos ver como FUSE se integra con la aplicación de usuario por un lado y con el kernel de Linux por el otro. El proceso lo inicia una aplicación que utiliza un sistema de archivos a través de las funciones de *GLIBC* (parte izquierda superior de la figura). Esta operación con el sistema de archivos se abstrae a través de *VFS* (parte izquierda inferior), de manera que es independiente del tipo de sistema de archivos utilizado. Aquí en vez de usar un sistema de archivos *ext3* por ejemplo, es FUSE quien gestiona la operación con el sistema de archivos (parte derecha inferior). Pero hasta aquí FUSE no aporta nada respecto a lo que sería desarrollar un driver del kernel para un sistema de archivos específicos. La ventaja de la arquitectura estriba en que ahora FUSE delega esta funcionalidad en una aplicación en el espacio de usuario que a través de la librería *libfuse* implementa toda la funcionalidad del sistema de archivos.

3.2.2. Justificación del uso de FUSE

Gracias a FUSE se puede hacer un ”puente” entre el API de Digidata que utiliza Terabox y el sistema de archivo de Linux de forma que podamos usar sin problema la interfaz de usuario habitual del e-reader. El usuario se meterá en un determinado directorio y a partir de ahí FUSE nos da un control total de los archivos de ese directorio y cómo se obtienen estos en realidad. Para esta fase solo necesitamos comprobar si funciona FUSE, con lo que ver si el programa de ejemplo que viene incluido con la librería proporciona la funcionalidad esperada es suficiente.

Figura 3.1: Arquitectura de FUSE



3.2.3. FUSE en el Iliad

En la documentación de FUSE [1] te informan que el soporte para las versiones de la 2.x a la 2.5.x empieza con la versión del kernel 2.4.21. Sin embargo la versión del kernel que trae el Iliad es la 2.4.19. El hecho de que esté tan cercano una de otra (solo dos versiones de diferencia) hizo que en un primer momento intentara adaptar el kernel 2.4.21 para que funcionara en el Iliad. Para ello cogí los parches que creó Irex sobre el kernel mainstream 2.4.19 e intenté aplicarlos al 2.4.21. Como evidentemente los archivos variaban tuve que modificar cada archivo manualmente en base a los parches. Esto me llevó bastante tiempo pero no obtuve resultados. Tras las modificaciones manuales compilé el kernel mediante el método de compilación cruzada que luego se explicará y el kernel modificado no arrancaba en el Iliad. Esto me planteó abandonar FUSE y hacer directamente un módulo del kernel que implementase la funcionalidad deseada. Comentando el problema en la división me sugirieron que buscara en versiones anteriores de FUSE hasta que encontrara alguna que fuera compatible con el kernel 2.4.19. Analizándolo a posteriori me doy cuenta que la idea de intentar modificar el kernel de Linux era bastante descabezada debido a su elevada complejidad. Finalmente hice caso a mis compañeros de división y efectivamente vi que la única versión anterior a la 2.x que quedaba en la forja era la versión 1.9 de FUSE. Por tanto ésta era la versión candidata a utilizar en el libro. Para probar su compatibilidad con el Irex habría que compilarlo mediante compilación cruzada.

[1] `http://sourceforge.net/apps/mediawiki/fuse/index.php?title=OperatingSystems`

3.2.4. Compilando FUSE mediante compilación cruzada

Para que FUSE funcione son necesarios dos elementos: En primer lugar el módulo del kernel *fuse.o* y en segundo lugar la librería *libfuse.so.2.0.0*. Para generar cada uno de estos archivos utilizaremos una **toolchain** diferente. Una **toolchain** es el conjunto de herramientas de programación necesarias para crear tanto aplicaciones como sistemas operativos. En este proyecto se emplea la **GNU Toolchain** que es un componente vital en el desarrollo Linux, BSD y como es nuestro caso el software para sistemas embebidos (que en este proyecto se basa en Linux también). Las herramientas forman un sistema integrado donde los resultados de uno son la entrada de otro, de ahí lo de cadena (chain). Los componentes de la **GNU toolchain** son:

- GNU Make - automatización de la estructura y de la compilación.
- GNU Compiler Collection (GCC) – compiladores para varios lenguajes. En este proyecto de estos lenguajes utilizaremos solo el lenguaje C.
- GNU Binutils – enlazador, ensamblador y otras herramientas.
- GNU Debugger (GDB) - un depurador interactivo.
- GNU build system (autotools) – Autoconf, Autoheader, Automake, Libtool - generadores de makefiles.

La **compilación cruzada** consiste en compilar (generar el archivo ejectuable) de una plataforma desde otra distinta. En nuestro caso tenemos un ordenador de escritorio con su sistema operativo (GNU Linux), su arquitectura (i686 o x86-64) y además todas sus propias librerías que pueden ser distintas a la de la plataforma para la que generamos los ejecutables (que en este caso es arquitectura ARM, Linux también pero versiones de librería bastante anteriores). Todo esto se tiene en cuenta en la compilación cruzada. En un primer lugar en mi división me recomendaron el uso de ”Scratchbox” que simplifica este asunto mediante un sistema de virtualización, sin embargo no conseguí generar ejecutables válidos para el iLiad. Leyendo el foro de desarrolladores de ”Mobileread” que es donde escriben los miembros de la comunidad de desarrolladores de Irex Iliad me enteré que el fabricante ofrece un **toolchain** para compilar el kernel y otro diferente (que incluye todas las librerías que utiliza el e-reader) para aplicaciones de usuario. Una vez localizados estos **toolchains** en la página del fabricante, empezé con la compilación del módulo del kernel de FUSE. Los pasos fueron los siguientes:

1. Descomprimir el **toolchain** del núcleo tras bajárnoslo de la página del fabricante.
2. Ajustamos el PATH para que el sistema busque en primer lugar el PATH actual y en segundo lugar en el directorio bin del **toolchain** del núcleo. La idea es que use las herramientas disponibles en el equipo y después busque las específicas del **toolchain** que llevaran el prefijo *arm-linux*
3. Nos bajamos las fuentes del kernel del libro y las descomprimimos
4. Aplicamos un parche que lo adapta a la versión del software 2.11 (la última disponible)
5. Copiamos la configuración correcta adaptada al libro a *.config*
6. Configuramos el kernel con la configuración de *.config*
7. Creamos los archivos y dependencias
8. Crear el kernel
9. Crear los módulos del kernel
10. Descomprimimos las fuentes de FUSE
11. Configuramos FUSE especificando donde tenemos compilado el kernel con rutas absolutas y además el prefijo que llevan los programas de la toolchain. La orden queda como sigue: *./configure –with-kernel=/tmp/linux-2.4.19-rmk7-pxa2-irex1 –host=arm-linux*
12. Creamos el ejecutable ejecutando *make* dentro del directorio ”kernel” de las fuentes de FUSE.

La siguiente etapa sería compilar la librería de FUSE. Para ello necesitamos la **toolchain** que incluye todas las librerías instaladas en el libro electrónico. Con ella seguimos los siguientes pasos:

1. Bajarse y descomprimir la **toolchain** para aplicaciones de usuario
2. Ajustar el PATH, pero esta vez poner el directorio en el que está la **toolchain** antes del resto del PATH (al contrario que en el caso del kernel, ya que en este caso vamos a dar prioridad a las herramientas de la **toolchain**)

3. Movemos *ipkg* (el gestor de paquetería embebido que usa el Irex) a donde espera estar */usr/lib*.

4. Ajustamos la variable de entorno PKG_CONFIG_PATH para que apunte al directorio donde se guardan las configuraciones de este programa que nos ofrece la **toolchain**. **pkg-config** es un software que provee una interfaz unificada para llamar a bibliotecas instaladas cuando se está compilando un programa a partir del código fuente.

5. Desde el directorio principal de fuse lo configuramos de manera similar al caso anterior pero con un pequeño matiz *./configure –with-kernel=/home/cesar/Desktop/linux-2.4.19-rmk7-pxa2-irex1 –host=arm-linux –prefix=/usr/local/arm/oe/arm-linux/* y es que ahora especificamos el directorio que se tomará como referencia sea el de la **toolchain** de usuario.

6. Bajamos a los directorios *lib*, *util* y *example* y vamos ejecutando *make* que nos generará la librería, una utilidad que nos permite desmontar los sistemas de usario fuse y el ejemplo que usaremos para probar la funcionalidad respectivamente.

3.2.5. Activar el paquete de desarrollo en el Irex Iliad

El Iliad cada vez que se formatea hay que activarle el paquete de desarrollo. Esto es lo que te permite ejecutar cualquier programa desde él. Para este proyecto este paso resulta esencial. En primer lugar hay que solicitar la opción al fabricante. Éste te suministra un login y un password para entrar en Irexnet. Es importante chequear previamente que estamos conectados a Internet visualizando el icono WiFi. Esta información la introduces en la página 4 del apartado Iliad Settings dentro del menú Device Manager que aparece pulsando el segundo botón de la esquina superior izquierda. Después pulsamos el botón de la esquina superior derecha para conectarnos a Irexnet y registrar la MAC de nuestro dispositivo Iliad. Ya desde el equipo de escritorio nos vamos a MyIrex[2] y elegimos la MAC de nuestro dispositivo Iliad. Tras ver en la pantalla del ordenador de escritorio la confirmación del envío del paquete, pulsamos el botón de la esquina superior derecha nuevamente durante tres segundos y comenzará la descarga e instalación del paquete de desarrollo.

[2] `https://myirex.irexnet.com/index.php/developer`

3.2.6. Instalación de la shell y del servidor SSH

De estos dos pasos el único estrictamente necesario es el primero, ya que nos permitirá tener una shell operativa en el e-reader. Sin embargo resulta poco práctica utilizar el estilete y el teclado virtual para cualquier comando que se quiera introducir dentro de un entorno de desarrollo. Por ello instalaremos el servidor SSH con el fin de poder acceder al e-reader en modo consola desde el ordenador de escritorio. También será muy útil para mandar archivos al dispositivo directamente a través del protocolo seguro SCP sin tener que utilizar tarjetas de memoria. Cabe reseñar que es imprescindible haber activado previamente el paquete de desarrollo. Ésta es una ”protección” que pone el fabricante. Nos bajamos con el ordenador de escritorio el *mrxvt_iliad_xshell_0.5.0* que es un emulador de terminal optimizado para el Iliad desde p. ej. el foro de Mobileread. Gracias a este programa ya podremos introducir instrucciones en el Iliad con el estilete y el teclado virtual. Para guardarlo en el Iliad lo guardamos en una Compact Flash o Multimedia Card (ambos formatos de tarjeta son soportados por el Iliad). Introducimos esta tarjeta en el e-reader y pulsamos la tecla del ”Device Manager”que está en la segunda tecla de la esquina superior izquierda y que previamente utilizamos para activar el paquete de desarrollo. En este menú nos metemos en el apartado correspondiente de la tarjeta de memoria que hayamos introducido y tras ver el icono de terminal virtual hacemos click en ”Install to internal memory” y en sucesivas ejecuciones ”Run from internal memory”.

Para la instalación del servidor SSH tenemos que localizar en el foro de Mobileread el archivo *unbreakableinst.zip* y lo guardamos en el Iliad usando la Compact Flash o el cable USB (En el menú ”Iliad Settings” se puede configurar que tengamos acceso a la Compact Flash a través del cable USB). Este archivo comprimido contiene el paquete con el servidor SSH así como un script de instalación. Además contiene los archivos necesarios para que aparezca correctamente en la interfaz del Iliad. Más adelante en este mismo proyecto también se modificarán estos archivos para tener una correcta representación en la interfaz del Iliad. Ejecutamos el script y ya está disponible el acceso SSH al libro electrónico.

3.2.7. Instalación y prueba de FUSE en el libro electrónico

Una vez conseguido el acceso SSH resulta muy sencillo transferir los archivos de FUSE al libro mediante SCP. Para ello hay que utilizar o bien la interfaz ethernet cableada que se enchu-

fará al mismo router que el ordenador de escritorio (se autoconfigurará mediante DHCP), o bien mediante WiFi. Lo único que hay que saber es la dirección IP asignada al e-reader, que se puede conseguir mediante el comando *ifconfig* en la terminal virtual que antes hemos instalado. El par usuario/clave para acceder es *root:rootme*. Ahora solo queda copiar los archivos compilados de FUSE *kernel/fuse.o*, *lib/.libs/libfuse.so.2.0.0*, *example/.libs/hello* al directorio con los módulos del kernel corespondiente al sistema de archivos, al directorio con las librerías y a un directorio del PATH respectivamente. Posteriormente habrá que ejecutar *ldconfig* mediante SSH para que termine de instalar la libería de FUSE. En esta URL[3] está documentado la función de *ldconfig*. La versión de Linux que tiene el Iliad no tiene soporte para montar los módulos automáticamente así que habrá que ejecutar *insmod* para cargar el módulo de FUSE explícitamente. Una vez cargado se puede ejecutar la aplicación hello que simula un sistema de archivos con un único archivo ”hello” de contenido ”Hello world!”. Como conseguimos tener esta funcionalidad correctamente, tenemos cumplida la primera fase del proyecto.

3.2.8. Resultados de la primera fase

Como resultados tenemos:

- El Iliad preparado para ejecutar aplicaciones externas directamente o a través de SSH.
- El entorno de compilación cruzada funcionando correctamente.
- El sistema de FUSE (módulo del kernel + librería) operativo en el libro.

3.3. Fase 2: Usar el API de Terabox en el libro

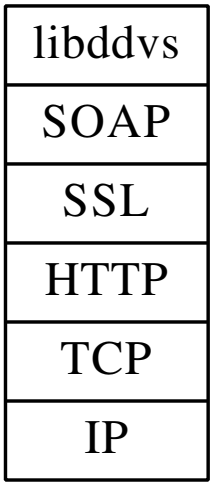
La funcionalidad de **Terabox** se ofrece a partir de una serie de funciones (API) que proveen los servicios de almacenaje online. El reto está en implementar estos servicios en un entorno embebido. Aunque esto se hizo cronológicamente después de probar FUSE (fase 1), nada impedía haberlo hecho antes y que esto hubiese supuesto la primera etapa. Sin embargo lo hicimos así siguiendo el principio de hacer las fases menos complejas primero. Los objetivos de esta etapa es que las funciones que vamos a utilizar del API de Terabox estén funcionando en el libro.

[3] `http://wiki.linuxquestions.org/wiki/Library`

3.3.1. Introducción al API de Terabox

El API de Terabox se proporciona a partir de la librería *libddvs* que proporciona el fabricante Digidata. Concretamente utilizamos su versión 0.4.4 que pese a no ser la última ni la que se usa en otros proyectos de la división fue la recomendada por mis compañeros. Internamente utiliza una implementación de la librería libre multiplataforma para implementar web services sobre C/C++ *gSOAP*. El uso de *gSOAP* está ampliamente extendido en la industria. El protocolo que utiliza por debajo es HTTPS (HTTP segurizado con la capa de SSL por encima). Esta pila de protocolos se puede observar en la figura 3.2.

Figura 3.2: Pila de protocolos de libddvs



3.3.2. Compilación del API de Terabox para el ordenador de escritorio

En la división en la que trabajaba, este API lo utilizaban en su versión para Windows dentro del proyecto mencionado en la introducción del ”Pincho infinito”. Por tanto en primer lugar había que probar el API en Linux dentro de un entorno más favorable como es el del ordenador de escritorio y ver su funcionamiento mediante el código de prueba que nos ofrece el fabricante y tras conseguirlo pasar a ejecutar las pruebas en el libro. Esta misma estrategia de probar el código en el escritorio y en el libro se repetirá en posteriores etapas. Compilar para el ordenador resulta inmediato siguiendo el procedimiento habitual de *configure* (sin ninguna opción especial) más *make*. Lo que sí ocurrió fue que apareció un fallo inesperado compilando el archivo *api/sharing.c* de tal forma que la parte correspondiente a este archivo en el Makefile no se generaba bien. El procedimiento para solucionarlo fue ver lo que se iba ejecutando real-

mente para otros archivos fuente y por inspección deducir cómo era la orden correspondiente que tenía que intercalar para que continuase ejecutándose el *make*. La librería se instala de la manera convencional con *make install*

3.3.3. Estructura general de un programa con el API de Terabox

En primer lugar se suele definir una macro para comprobar si las funciones del API han fallado que si es así se liberan los recursos y se salen del programa (en el código la denominamos *ERROR_CHECK*). Después hay que inicializar la conexión introduciendo el usuario y contreseña mediante la orden *ddvs_init*. En la división teníamos una cuenta de Terabox que utilizábamos para el desarrollo. Con la extinción de la división también dejó de funcionar esta cuenta. Si los comandos utilizados en el API devuelven algún resultado se le suele pasar un puntero para que escriban el resultado en él. Finalmente tanto si se ha producido un error como si el programa termina normalmente se liberan los recursos con *ddvs_clean*.

3.3.4. Prueba del API de Terabox para el ordenador de escritorio

Una vez compilado el API elegí una batería de pruebas que fueran significativas para comprobar que el API funcionara correctamente. Estas pruebas están basadas en los ejemplos de la librería que incluye el fabricante. Las operaciones referidas a funcionalidad no incluida en los objetivos (la escritura en remoto p.ej.) y que aparecía en los ejemplos del fabricante ha sido eliminada. En la compilación hay que añadir la opción *-lddvs* para incluir la librería del API.

- Hacer un ping usando *ddvs_ping*.
- Probar que existe un determinado archivo que hemos subido anteriormente (usando la interfaz web de Terabox p. ej.) mediante *ddvs_file_exists*.
- Probar que existe un determinado directorio que hemos subido creado previamente mediante *ddvs_directory_exists*.
- Descargar un archivo que está en remoto a un directorio local subido anteriormente mediante *ddvs_download_file* y que éste se mantiene igual.
- Leer una porción de un archivo subido anteriormente mediante *ddvs_read* de manera correcta.

Todas las pruebas funcionaron correctamente en esta etapa.

3.3.5. Compilación del API de Terabox para el dispositivo Iliad

En este caso realicé los siguientes pasos que suponen una novedad respecto la compilación para el ordenador de escritorio:

1. Comprobar que en directorio de librerías de la **toolchain** existen las librerías requeridas por el API que son *libopenssl* y *libcrypto*

2. Ajustar el PATH para poner el directorio binario de la **toolchain** antes del PATH existente.

3. Ajustar el PATH de *pkg_config* al correspondiente de la **toolchain**.

4. En el caso de compilar en arquitectura x86-64 había que añadir la variable de entorno *ac_cv_func_malloc_0_nonnull* puesta a *yes* para solucionar ciertos problemas que aparecen al reservar memoria con *malloc*.

5. En la orden *configure* hay que poner la opción *–host=arm-linux* y *–build=i686* o *–build=x86-64* que especificará la arquitectura del libro y la arquitectura en la cual estamos compilando respectivamente. También especificamos el directorio de referencia donde se encontrarán las librerías de la **toolchain** con *–prefix=/usr/local/arm/oe/arm-linux*

6. Tras compilar con *make* ya solo queda copiar la librería generada *libddvs.so.0.0.0* al directorio */usr/lib* del libro electrónico y ejecutar *ldconfig* mediante SCP y SSH respectivamente.

3.3.6. Prueba del API de Terabox en el libro electrónico

Son las mismas pruebas que en el caso del ordenador de escritorio y que describimos previamente. Para probar el API de Terabox hace falta conexión a Internet. El Iliad se puede conectar mediante cable Ethernet o bien con WiFi. No obstante esta aplicación está pensada para conectarse mediante módem 3G, aunque esta forma de conexión la dejaremos para fases sucesivas. Para compilar aprovechamos que ya hemos configurado el PATH para incluir los binarios de la **toolchain**. La orden de compilación será entonces *arm-linux-gcc -L/usr/local/arm/oe/arm-linux/lib -I/usr/local/arm/oe/arm-linux/include -lssl -lcrypto -ldl -lddvs* seguida del archivo de

prueba. En este comando especificamos donde se encuentrar las librerías y las cabeceras de la **toolchain** además de enlazar las librerías necesarias para el API. En este caso sí nos encontramos con un error que aparece en todas las pruebas excepto en la primera del ping. Como no tenía suficiente información del error, compilé el programa GDB para el libro electrónico para investigar donde aparecía. Cabe reseñar que para poder depurar un programa con GDB es necesario especificar al compilar que se incluyan las trazas de depuración con la opción *-g* de GCC. Un compañero de división me dejó que utilizará un servidor que tenían, que puesto entre medias de una comunicación SSL te dejaba ver el contenido de los mensajes. Como SOAP manda mensajes XML ver estos mensajes hace que puedas ver en qué punto de la comunicación se produce el error. Gracias a estas dos técnicas vi que el problema era que habían caducado los certificados del iLiad. El motivo era que la fecha del libro estaba mal configurada y la fecha que tenía puesta estaba fuera del periodo de validez de los certificados. La validez de los certificados se comprueba durante la negociación SSL, cuando el servidor envía su certificado X.509 en la fase inmediatamente posterior al saludo del servidor (**ServerHello**). Una vez solucionado este problema, las pruebas funcionaron con normalidad cumpliéndose los objetivos de la etapa 2.

3.4. Fase 3: Usar FUSE en el libro

Una vez probado que el API de FUSE y la librería de Digidata (api de Terabox) funcionan en el libro, nos podemos centrar en la implementación del programa en sí. Para ello empezaremos con desarrollar las funciones u operaciones de FUSE. Ésta es la manera de interactuar con la aplicación a través del VFS como hemos explicado anteriormente y por tanto lo lógico es empezar por ahí. También elegimos hacer un código ”dual”que funcionara tanto en el libro como en el ordenador de escritorio. Esto permitirá acotar los problemas que puedan deberse a trabajar en un entorno embebido como son entre otras cosas el tener un procesador de velocidad muy baja, poca memoria disponible así como una conexión a Internet menos estable debida al módem 3G. También facilitará posible trabajo futuro con modelos de e-readers que soporten las nuevas versiones de FUSE.

Para que una aplicación FUSE funcione hacen falta cuatro elementos principalmente:

- Incluir las cabeceras de FUSE *fuse.h*

- Linkar el programa contra la librería de FUSE con la opción de **gcc** *-lfuse*

- Definir la estructura *fuse_operations* con punteros a las funciones que implementen cada operación y que veremos en detalle en el siguiente apartado.

- Ejecutar la función *fuse_main* con la estructura de las operaciones y los argumentos de la función main. De ahí se extraen opciones como ejecutar con *-d* para entrar en modo debug que posteriormente utilizaremos o el parámetro fundamental de la aplicación FUSE que es el punto de montaje del sistema de archivos.

3.4.1. Implementar operaciones de FUSE

La funcionalidad de FUSE se basa en ir implementando las operaciones que están relacionadas con las llamadas al sistema virtual de ficheros (VFS). Lo primero que hay que tener en cuenta es que como comentamos en la sección anterior, la versión de FUSE que utilizamos en el libro no es la más reciente sino la última compatible con el kernel del e-reader. En la versión del código para el ordenador de escritorio sí que trabajaremos con la última versión disponible. El motivo es poder aislar posibles problemas que pudiera haber en la versión antigua de FUSE y que se hubieran corregido en versiones posteriores. La manera de separar el código dependiente de la versión de FUSE es mediante el uso de preprocesador C (*cpp*). Concretamente definiremos la variable de compilación ILIAD que usaré junto con las directivas de compilación condicional *ifdef*, *ifndef* y *endif*. Si está definida ILIAD se compilará el código para el libro electrónico en el caso de que no, el código para el ordenador de escritorio será compilado. Las funciones que hay que implementar son:

get_attr(const char* path, struct stat* stbuf) Devuelve los atributos del archivo. El archivo se especifica en el *path* y los atributos se escriben en *stbuf*. El formato de estos atributos es el de la función *stat* de UNIX. De estos atributos el primero que utilizamos es *st_mode* que nos indica en primer lugar si es un archivo o un directorio. Existen otros tipos de archivos pero no los utilizaremos porque ni Terabox los soporta ni es necesario para la funcionalidad del proyecto. A continuación se fijan también los permisos. El API de Terabox no tiene un sistema de asignar permisos a (usuario/grupo/todos) y nosotros necesitamos especificar que los archivos no se pueden escribir (no está contemplada esa

funcionalidad en el proyecto). Además los directorios tienen que tener el flag de ejecución activado. La lectura obviamente tiene que estar activada. Respecto a distinguir entre usuario, grupo y todos al ser un entorno monousuario no tiene sentido la distinción y dejaremos todos los flags iguales. El tamaño del archivo sí que nos lo da el API y en el mismo formato de número utilizado por *st_size* (Unsigned long de 64 bits) así que no hay ningún problema. Por último las fechas de modificación y acceso no las proporciona el API, con lo que las dejaremos todas iguales a las de creación que sí nos la da el API. Nuevamente el formato vuelve a ser el mismo (time_t).

getdir(const char *path,fuse_dirh_t buf, fuse_dirfil_t filler) Esta operación se usa para obtener el contenido de un directorio. Los argumentos de esta operación de FUSE cambian con la versión (de hecho cambia hasta el nombre que pasa a ser readdir) así que usaremos compilación condicional para distinguir el caso de cuando compilamos para el ILIAD y cuando compilamos para el ordenador de escritorio. El funcionamiento de esta operación es simple, se trata únicamente de ir llamando a la función *filler* con el nombre del archivo o directorio. Esta función *filler* nos la pasa FUSE por parámetro y no es necesaria implementarla. No hace falta especificar más, el número de argumentos de fillers (todos a NULL, excepto el citado nombre) vuelve a cambiar con la versión con lo que usaremos de nuevo compilación condicional. Cabría hacerse la pregunta que cómo se distinguen archivos normales de directorios, la respuesta es corta, en caso de necesitarse esa información el sistema llamará a la operación anterior *get_attr*. Es necesario introducir manualmente los directorios .^es^peciales". y ..

open(const char *path, int flags) Se usa esta operación cuando se quiere abrir un fichero. Vuelven a cambiar los argumentos con la versión. En este caso lo que nos interesa que son los *flags* con los que se abre el fichero, esto es si se abre para leerlo o también para escribirlo. En este último caso hay que mostrar un error porque nuestro sistema no soporta escribir en los archivos. Esta variable de flags está dentro de una estructura *fuse_file_info* en las versiones modernas de fuse. También se mostrará un error en el caso de que el archivo no exista.

read(const char *path, char *buf, size_t size, off_t offset) Es la operación para leer en el fichero especificado por *path*. Se leerá el número de bytes especificado por *size* a partir del

punto que nos determine el argumento *offset*. El resultado de la lectura se escribe en *buf*. La versión moderna de FUSE tiene un parámetro adicional *fuse_file_info* que no se usa en esta aplicación.

flush(const char *path) Se llama cuando se cierran los archivos con el fin de mostrar errores que no se habían mostrado todavía. Puede haber varios *flush* por cada *open* y nada garantiza que se vaya a ejecutar esta operación con seguridad en cada cierre. Es obligatorio definirla, pero no hace nada en nuestra aplicación. De nuevo aparece el mismo parámetro adicional que en *read* para las versiones nuevas de FUSE.

3.4.2. Integrar FUSE con el API de Terabox

Una vez descritas las operaciones de FUSE que vamos a utilizar ya solo queda probarlas con Terabox. Para ello vamos a restringir funcionalidad respecto a la especificada en los objetivos del programa. El objetivo es tener una aplicación más simple para ir probando funcionalidad según un esquema iterativo. Estas restricciones son:

- Estaremos siempre ”online”. No trabajaremos con archivos temporales guardados en el Iliad y supondremos que siempre hay conexión a Internet.
- La interfaz será la terminal. Las pruebas de esta fase se harán a través del cliente SSH, evitando tener que tratar con los menús del libro que como veremos más adelante necesitan una funcionalidad específica.

Una vez hechas estas limitaciones veremos que funcionalidad del API necesitamos en cada operación. Las funciones del API de Digidata/Terabox tienen la peculiaridad de que la reserva de memoria para las estructuras que se utilizan en la devolución de datos de las funciones del API se hace dentro de las mismas funciones. Habitualmente en C en estos casos es la función que llama la que suele tener la responsabilidad de reservar la memoria necesaria (típicamente con el método *malloc*) y posteriormente liberarla (con *free* p. ej.), para cualquier atributo que usen tanto en entrada como en salida las otras funciones que se vayan llamando. En este API la reserva de memoria se hace internamente y su liberación se produce al llamar a la función de finalización *ddvs_cleanup*. El motivo de esto es el uso por debajo de gSOAP que implementa ”garbage collection”.

get_attr Aquí necesitamos conocer los atributos de un determinado archivo o directorio. Con las limitaciones presentes en esta etapa no sabemos a priori si el archivo que nos pasan en la variable *path* es un archivo o un directorio. Esto es así porque en este momento no guardamos en memoria ninguna representación del arbol de archivos como luego sí que ocurrirá en fases sucesivas. Así que lo que se hará será en primer lugar probar si es un archivo y sino probar si es un directorio. En caso de que las dos pruebas tengan resultado negativo concluiremos que el archivo/directorio no existe y devolvermos el error estándar ENOENT. Estamos hablando en todo momento de extraer los atributos de un archivo que está en remoto, dentro del sistema de almeacenamiento en la nube Terabox. Para comprobar si es un directorio y sacar sus atributos en el caso de que exista utilizaremos la función del API *ddvs_get_directory_details*, mientras que para el caso de los atributos del archivo usaremos *ddvs_get_file_details*. En la siguiente tabla vemos la correspondencia entre los campos que devuelve *get_attr* y lo que devuelve la función del API corresondiente.

Campo	Archivo	Directorio
st_mode	S_IFREG *or* 0444	S_IFDIR *or* 0755
st_nlink	1	2
st_size	Size	Size
st_atime	CreationDate	CreationDate
st_ctime	CreationDate	CreationDate
st_atime	CreationDate	CreationDate

get_dir Para esta operación necesitaremos una función que nos diga los contenidos de un determinado directorio. Esta función en el API de Digidata / Terabox será *ddvs_directory_contents*. Dado un directorio situado en un determinado path, nos devuelve una estructura con todos los archivos que cuelgan de ese directorio además de todos los subdirectorios. También nos da la información de cada archivo/directorio, pero sólo nos interesan en esta operación los nombres. FUSE simplifica mucho todo al tener solamente que ir llamando a la función *filler* que nos la pasan con parámetro con cada uno de los nombres de archivo/directorio. Otra cosa que tenemos que hacer es chequear si hay un error en la llamada a esta función, que ocurrirá si no existe un directorio en ese path. En ese caso habrá que devolver el error estándar ENOENT. Este error lo arrojan diferentes funciones que trabajan con sistemas de archivos y significa exactamente esto, que no existe el archivo.

open Además la comprobación comentada anteriormente de mirar si solo se quiere abrir el archivo para lectura, la existencia en sí de un determinado archivo la comprobamos con la función del API *ddvs_file_exists* que nos devuelve *ddvs_true* si existe o *ddvs_false* en caso contrario. Si el problema es que el archivo no existe se devuelve el error antes mencionado de ENOENT. En caso de que el archivo exista y se quiera abrir para escribir en él, el error a arrojar será EROFS. Este error lo utilizan diferentes llamadas al sistema cuando se intenta escribir en un sistema de archivos de solo lectura. Si se intenta abrir como lectura un directorio en cuyo caso lanzamos el error EISDIR que significa exactamente esto.

read La solución directa para implementar esta función es utilizar la función del API cuya funcionalidad es más parecida a esta operación de FUSE. La función *ddvs_read* tiene una correspondencia total entre las entradas y las salidas de esta operación de FUSE. Con esta operación no deberían producirse errores de leer ficheros cuyo path no existe, ya que a esta función le precede siempre una llamada a *open* que ya comprueba que la ruta exista. En caso de que se produjera cualquier error en el proceso de lectura remota (típicamente errores con la red, o usuario/contraseña de Terabox errónea) se lanza el error EIO que informa de cualquier error de bajo nivel y es adecuado para este tipo de errores al producirse en un nivel por debajo del sistema de archivos en sí mismo.

flush No se implementa.

3.4.3. Primeras pruebas de la integración de FUSE con el API de Terabox

Con las limitaciones antes comentadas, empezamos las pruebas tanto en el libro como en el ordenador de escritorio. Como estamos en la terminal en ambos casos, visualizaremos un libro electrónico en formato TXT con editor de texto estándar como es **vim**. El libro electrónico es un libro de tamaño algo mayor al medio megabyte. Como tras esta prueba sospechamos que puede haber un problema, añadimos una prueba adicional. En esta última prueba, abrimos un fichero PDF de 4 megabytes de tamaño desde el visor estándar del ordenador de escritorio y el menú normal del libro electrónico utilizando el estilete. Debido a que todavía no hemos considerado el caso de trabajar con los menús de Iliad la visualización de los contenidos del directorio no será igual a la que vemos con un directorio local (en la tarjeta Compact Flash, p. ej. del libro). Esto es debido a que para los menús el Iliad escribe una serie de ficheros en

el directorio y aquí estamos en un entorno de solo lectura. Este problema lo solucionaremos más tarde. Sin embargo la visualización del PDF en sí no debería dar ningún problema. Los resultados son los siguientes:

Listado de directorios Para ello realizamos la prueba con el comando del shell *ls*. Las operaciones FUSE involucradas en esta prueba son *get_dir* y también *get_attr*. Esto es porque *ls* primero mira cuales son los miembros de un directorio y posteriormente mira sus atributos. Los resultados son satisfactorios aunque aparecen ficheros que parecen ser archivos especiales de Terabox y que por tanto no deberían aparecer en el listado que devuelve *get_dir*. Buscando en la interfaz web de Terabox vemos que estos ficheros no aparecen con lo que confirmamos que son archivos especiales. Son fácilmente localizables porque todos empiezan con un asterisco (*), con lo que el trabajo de su eliminación en posteriores etapas será sencillo.

Visualización del fichero TXT El resultado de las pruebas en el libro electrónico es diferente al del ordenador de escritorio:

Prueba en el ordenador de escritorio La visualización del fichero era algo más lenta que lo habitual, pero podemos considerarlo razonable dado que no entorpece demasiado la experiencia de usuario.

Prueba en el Irex Iliad El fichero tarda en cargar bastante más que en el ordenador de escritorio y está en el límite de lo que el usuario estaría dispuesto a esperar. Con esta prueba vemos que si con un fichero de texto de tamaño relativamente pequeño comparado con un PDF estamos al límitede lo esperado por el usuario, con un PDF probablemente tendremos problemas con lo que añadimos a la prueba la apertura de un fichero PDF de tamaño grande.

Visualización del fichero PDF La diferencia en este caso entre el ordenador de escritorio y el libro electrónico es crítica y nos habla de un error en el diseño:

Prueba en el ordenador de escritorio El fichero tarda en cargarse en torno a algo menos de un minuto, pese a ser un tiempo considerable lo damos por aceptable por no ser excesivo.

Prueba en el Irex Iliad En este caso el fichero tarda un total de seis minutos lo que resulta totalmente inaceptable y denota que algo estamos haciendo mal.

3.4.4. Análisis de las pruebas de integración FUSE - Terabox

La prueba de la visualización del fichero PDF de cuato megabytes nos dio unos tiempos de carga muy elevados lo que supone un problema. Resulta que el uso habitual del libro electrónico es la visualización de ficheros PDF en torno a ese tamaño. Para hacerme una idea de lo que estaba ocurriendo, puse la aplicación en modo ”debug” mediante la opción *-d* al invocarla en la línea de comandos. De esta manera veo por pantalla cada operación que va cursando la aplicación. Lo que ocurre es se están realizando una gran cantidad de operaciones *read* ya que van leyendo fragmentos en torno a 128 kilobytes. En este momento teníamos un modo de operación ”síncrono” (1 petición read FUSE se corresponde con 1 petición read a Terabox) que estaba produciendo un gran coste en recursos. El motivo es que cada llamada al *read* del API Terabox conlleva entre otras cosas preparar los mensajes SOAP que se envían, parsear los que se reciben, deserializar datos... que en un entorno embebido con pocos recursos hacen el esquema tal y como está hasta ahora inviable. La solución está en implementar un sistema de caché de manera que si empezamos a leer un fichero almacenemos localmente parte del resto del fichero ya que es probable que en otras llamadas a la operación read de FUSE nos pidan esa información. De esta manera nos ahorramos llamadas al API de Terabox que es lo que ralentiza la aplicación. La idea es parecida al concepto de ”readahead”(Sección 18.4 de la referencia [3]) que usa el kernel de Linux, solo que en vez de hacer caché en memoria de archivos en disco se hace caché en disco de archivos que vienen de la red. Dado que hemos comprobado en esta fase la importancia de la caché, resulta interesante que esto sea lo siguiente en implementar.

3.5. Fase 4: Implementar un sistema básico de caché

Como hemos comprobado con los resultados de la pasada fase, resulta de vital importancia tener un sistema de almacenamiento temporal que nos permita ahorrarnos llamadas al API de Terabox, mejorando por tanto los tiempos de carga de los ficheros por las razones antes comentadas. Existen decisiones de diseño que deben quedar clarificadas en esta fase como es la manera óptima de bajarse el fichero. Se puede descargar el fichero entero, o bien bajárselo por bloques

de una cantidad determinada. Dado que no existe una forma de conseguir este parámero a priori usando cálculos teóricos lo determinaremos experimentalmente. La manera más fácil de hacer esto es medir los tiempos de carga de un determinado fichero PDF (el usado anteriormente por ejemplo) cambiando las maneras de descargarse el fichero y viendo cuál es el tiempo más corto como veremos en detalle en esta fase.

3.5.1. Descripción pormenorizado del funcionamiento de la operación de la caché

Lo primero de todo es fijar un directorio temporal donde se vayan almacenando los archivos según se descargan. Para identificarlos lo más sencillo es conservar la estructura de directorios que tienen remotamente, de esta manera los tendremos identificados de manera unívoca. Este directorio temporal desde el que colgarán los archivos temporales debe estar preestablecido previamente. Lo lógico es colocarlo donde esté montada la Compact Flash y así no saturar la memoria interna del libro que es bastante limitada. Éste es el único requerimiento y dado que el Iliad la monta normalmente en el mismo directorio, es suficiente establecer mediante un *#define* este directorio que colgará de donde esté montada la CompactFlash. Esta forma de establecerlo me parece correcta, dado que se conservará de una ejecución a otra (no es necesario por tanto pasarlo como parámetro cuando se ejecuta la aplicación) y en el caso de que hubiera que cambiarlo lo tenemos localizado mediante esta definición de macro del preprocesador de C.

Otra cuestión a resolver es cómo comprobar que un determinado archivo está presente en la caché y también determinar si el directorio al que pertenece está en la caché. Lo segundo podría ocurrir sin necesidad de lo primero. Para esto utilizaremos un comando UNIX estándar como es *test* que con la opcion *-f* nos permite comprobar la existencia de un archivo y con *-d* la de un directorio. Lo interesante de este comando es que al devolver cero si se cumple la condición o uno en caso contrario nos permite, a través de su invocación con la función *system* (que ejecuta comandos externos en un programa C), comprobar fácilmente estas condiciones con un simple *if*.

Después tenemos la cuestión de determinar la forma óptima de descargar el fichero. Esto se puede hacer de una vez con *ddvs_download_file* o bien ir leyendo consecutivamente bloques de

un determinado tamaño con la función que utilizábamos en el esquema síncrono de la anterior fase, *ddvs_read*. Habrá que implementar las dos opciones e ir comentando la opción que no usamos. También lo podía haber hecho con directivas de compilación condicional pero dado lo concreto de la prueba no vi necesario añadir está complejidad adicional al código.

Una vez consideradas estas cuestiones previas podemos pasar a la implementación en sí misma. La operación FUSE en la que meteremos esta funcionalidad es la de *read* aunque también lo podíamos haber introducido en la operación *open*, tras haber comprobado que se quiere abrir el como lectura. Una cosa ocurre inmediatamente después de otra y no existe beneficio apreciable en los tiempos de carga al introducir la funcionalidad de la caché en la operación *open*. Empezaremos por la construcción de los paths temporales de la caché y el ”real” de Terabox. El primero será donde guardaremos el archivo y ya tenemos definida con la macro el directorio del que cuelga. Será por tanto solo cuestión de añadir el path relativo que nos lo pasan por parámetro en la operación *fuse*. Al estar programando en C hay cuestiones a tener en cuenta al construir estas variables, como son calcular primero su longitud total sumando las longitudes del path relativo y del directorio raíz de la caché mediante *strlen* y reservar memoria suficiente con *malloc*. Respecto del directorio ”real” de Terabox resulta muy interesante no trabajar con el directorio raiz, sino con un determinado subdirectorio. Esto es así porque el usuario puede tener aparte de libros electrónicos, que será lo que vaya a usar en el Iliad con nuestra aplicación, otros archivosque no tiene sentido que estén accesibles desde la aplicación. La solución vuelve a ser que los datos de la aplicación cuelguen de un directorio específico de Terabox. Todo lo que hemos usado para el caso de la construcción del path de la caché temporal vuelve a ser útil para construir el path de Terabox. Me refiero a definir con una macro de compilación el directorio de Terabox desde donde cuelgan los archivos de la aplicación y a la construcción de este path ”real”.

Una vez que tenemos el path de la caché temporal hay que comprobar si está ese archivo presente de la forma anteriormente comentada en esta misma subsección. Si es así es que ya lo hemos descargado en la caché y pasamos directamente a leer de la Compact Flash. Para ello primero abrimos el fichero de la caché con *open*, pasándole el flag del acceso de lectura, *O_RDONLY*. Para usar esta función es necesario incluir el fichero de cabeceras **fcntl.h**. Una vez abierto el fichero pasamos a leerlo con la función *pread*. La correspondencia entre los argumentos de la operación FUSE *read* y esta última función es casi directa, salvando que

cambiamos el path que nos pasan como argumento (y que hemos usado para construir el path de la caché) por el descriptor del fichero de la caché devuelto por *open*. El resultado que nos devuelve *pread*, que son el número de bytes leidos es también lo que devuelve la operación *read* de FUSE.

En el caso de que el archivo no esté presente en la caché tenemos que bajárnoslo de Terabox. Implementeremos dos formas de bajárnoslo. La primera es bajárnoslo por bloques mediante *ddvs_read*. Para esta implementación lo primero es abrir para escritura el archivo correpondiente de la caché que será donde nos descargaremos el archivo remoto y cuya ruta hemos establecido previamente. Una vez bajado el bloque lo escribimos en el fichero temporal de la caché con *fwrite*. Esta función está pensada para escribir arrays con elementos de un tamaño dado. En este caso será un solo bloque de tamaño igual al número de bytes leidos. Repetimos está doble acción de lectura de la red / escritura en la caché hasta que el número de bytes leidos sean menor que el tamaño del bloque, lo que quiere decir que hemos llegado al final del fichero. Llegados a este punto cerraremos el fichero porque no vamos a escribir más en él y así se vacían los buffers de escritura. La función que se usa para esto es *close*. La segunda forma es directamente con la función *ddvs_download_file* en la que especificando tanto la ruta de la caché como la "real" de Terabox nos descargamos directamente el fichero a la ubicación correspondiente de la caché.

3.5.2. Pruebas entre distintas maneras de descargarse los archivos de Terabox

El procedimiento de esta prueba que nos determinará la manera óptima de bajarse los archivos del sistema de almacenamiento en la red es probar los dos métodos que acabo de presentar. En el caso de ir descargándolo por bloques probé con diferentes tamaños múltiplos de 512 kilobytes desde esta cantidad hasta 10 megabytes. La elección fue totalmente arbitraria, siendo los únicos requisitos que nos moviáramos en el entorno típico de tamaño de ficheros PDF's de libros, comics, etc. y quedara un número de pruebas razonable. La solución que escogiera decidí que no iba a depender del tamaño real del fichero. Aunque es posible que si considerara este factor se podría optimizar el tiempo de descarga, esto introducía una complejidad adicional para un parámetro que no iba a mejorar de manera apreciable. El tamaño del archivo de pruebas era entorno a 8 megas, por ser un ficharo lo suficientemente grande por un lado para poder probar

con bastantes tamaños inferiores, y por otro lado más pequeño que el límite superior para que probar que ocurre cuando el tamaño de bloque es superior al del archivo. El resultado óptimo fue establecer un tamaño de bloque de 5 megabytes, que daba ligeramente mejores tiempos que utilizar la función que te lo descargaba de una sola vez.

3.6. Fase 5: Implementación del caso ”offline”

3.6.1. Introducción a la funcionalidad necesaria

Por caso ”offline” me refiero a lo que ocurre cuando no existe acceso a Internet. Estamos en un entorno móvil y por tanto puede ocurrir que no exista cobertura 3G por estar en el metro o por estar en una región aislada o simplemente que el usuario tenga una cierta tarifa de datos y no le interese que se opere con el módem 3G aun siendo posible la conexión. La primera cuestión que hay que resolver es saber qué cambia en la aplicación cuando no se tiene conexión a Internet. La respuesta es evidente, no se pueden realizar llamadas al API de Terabox. Hay que ver los objetivos para ver cómo influye esto en la funcionalidad del programa y partir de ahí ver los cambios que debe haber en las funciones del programa. Leyendo los objetivos tenemos que estando en este modo se deben mostrar ”los archivos disponibles la última vez que hubo conexión”. Hay que idear por tanto un sistema que conserve todos los archivos disponibles cuando hay conexión para que estén disponibles en el estado ”offline”. Lo siguiente es determinar cuál es esa información exactamente. Por archivos nos referimos solo a los nombres de archivos, ya que el archivo en sí solo estará disponible en el caso de que estuviera en la caché, que como hemos visto en la fase anterior, esto es equivalente a decir que lo hemos visualizado/descargado previamente. Son por tanto los nombres de los archivos lo que hay que conservar. Esto traducido a funciones FUSE es la funcionalidad de *get_dir* que devuelve los nombres de archivo/directorio existentes. Pero esta función tiene como parámetro el directorio donde residen los archivos. Esto hace que habrá que conservar la estructura de directorios. Por último viendo la aplicación en modo ”debug” en las fases anteriores vemos que toda operación de mostrar archivos, conlleva también la operación FUSE para obtener sus atributos que es *get_attr*. Esto es así porque hay que saber parámatros como el tamaño, la fecha de creación o simplemente si es un archivo o un directorio a la hora de mostrar un listado de archivos. Juntando todo tenemos que necesita-

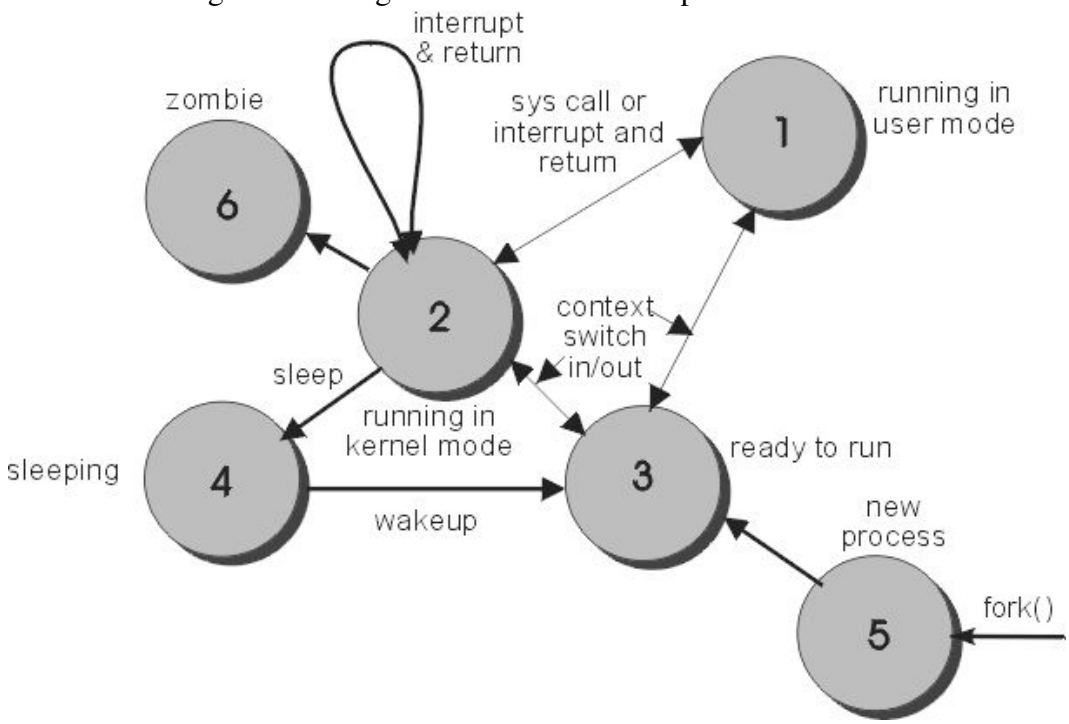
remos almacenar la estructura de archivos/directorios junto con sus características básicas. Por último tenemos que idear la manera de que la información se conserve tras cerrar la aplicación. El requisito de ”los archivos disponibles la última vez que hubo conexión” no se limita a que esa última conexión fuera en la misma ejecución del programa. Podría darse el caso de que el usuario apagase el libro electrónico entre medias.

Dada la complejidad de todo lo que supone trabajar con la aplicación en modo offline he visto conveniente dividir la descripción de esta implementación en varias secciones por separado.

3.7. Fase 5.1: Comprobar la conectividad a Internet

Lo que necesitamos saber es si hay conectividad a Internet con el fin de poder acceder a Terabox. Lo primero es determinar en qué momento se realizará esta comprobación. Tenemos que nuestra aplicación no está ejecutándose permanentemente. La aplicación estará en estado de ”sleep” mientras no lleguen llamadas a las funciones que implementan operaciones de FUSE. Este estado de ”sleep” es uno de los estados en el que puede estar un proceso en Linux y se utiliza para que una aplicación que no tiene que hacer nada no consuma recursos de sistema. En la figura 3.3 se visualizan los diferentes estados en que puede estar un proceso en Linux.

Figura 3.3: Diagrama de estados de un proceso en Linux



Por tanto si queremos sondear de manera periódica la conexión a Internet no podremos hacerlo en el mismo proceso que la aplicación principal. La única alternativa posible a la utilización de un nuevo proceso es comprobar esta conexión en cada llamada al API. Esto supondría añadir un retardo considerable a la aplicación dado que las pruebas de conexión llevan bastante tiempo. Una prueba de conexión a Internet puede consisitir únicamente en descargar una página web con un comando estándar de linux como *wget*. Dado que vamos a tener dos procesos diferentes hay que diseñar una forma de comunicación entre ellos de manera que el proceso que comprueba la conexión pueda avisar a la aplicación principal de que está disponible o no esta conexión a Internet. En Linux existen diferentes mecanismos del kernel para la comunicación entre procesos como son las tuberías, semáforos, la memoria compartida o las colas de mensajes. Sin embargo en nuestro caso no necesitamos tanta funcionalidad, dado que solo queremos avisar que hay conexión a Internet o bien que no existe ésta. Para esto es ideal el uso de las señales.

Las señales son una forma limitada de comunicación entre procesos que se utiliza en los sistemas tipo UNIX y en todos los sistemas operativos compatibles con el estándar POSIX. Es una notificación asíncrona mandada por un proceso o entre hilos del mismo proceso con el fin de notificar que ha ocurrido un evento. Cuando se envía una señal, el sistema interrumpe el

flujo de ejecución normal del proceso que recibe la señal. La ejecución se puede interrumpir en cualquier instrucción no atómica. Si el proceso tiene registrado previamente un manejador para la señal, se ejecuta esa rutina. En caso contrario el manejador normal de la señal es ejecutado. Las señales las envía el sistema a un proceso cuando una combinación de teclas como CTRL-C son pulsadas, por iniciativa del kernel cuando se produce un determinado evento, cuando se produce una excepción hardware o manualmente desde el intérprete de comandos con *kill* o *killall*. Éste ultimo comando será el que utilizaremos en nuestra aplicación. La manipulación de señales es susceptible de condiciones de carrera. Como las señales son asíncronas, otra señal (incluso del mismo tipo) puede ser enviada al proceso durante la ejecución de la rutina de manejado de la señal. Esto lo tenderemos en cuenta en nuestra aplicación.

Existen diferentes señales cada una asociada a un evento específico. Sin embargo hay dos señales SIGUSR1 y SIGUSR2 que se dejan para eventos definidos por el usuario. Esto viene de manera ideal a la aplicación porque solo tenemos dos eventos que comunicar a la aplicación: que existe conectividad a Internet y que ésta no existe. Asignamos SIGUSR1 a la presencia de conexión a Internet y SIGUSR2 al caso contrario.

3.7.1. Descripción de la implementación de la comprobación de Internet

En primer lugar veremos la implementación de la aplicación independiente que sondea periódicamente la conexión a Internet. Al estar funcionando permanentemente estará dentro de un bucle infinito implementado con *while(1)*. Para que no consuma excesivos recursos elegiremos un tiempo de medio minuto entre sondeo y sondeo. Este tiempo lo esperará gracias a la función *sleep(30)*. La elección de este tiempo nos proporciona una solución de compromiso entre estar continunamente sondeando, con el gasto de recursos del sistema que esto conlleva y sondear poco frecuentemente, donde el sistema tardaría demasiado en darse cuenta que la conexión a Internet está disponible, haciendo esperar al usuario un tiempo elevado.

Dentro de cada iteración del bucle infinito, en primer lugar nos intentamos descargar la página con *wget*. La página a descargar elegimos que sea la homepage de la empresa matriz de donde se realizizó el proyecto (Telefónica). Daba por hecho que la probabilidad que ésta esté fuera de servicio es muy baja. Además es bastante ligera en cuanto a tamaño con lo que no descargará muchos datos. Cabe recordar que estamos en un entorno móvil y es posible que al usuario se le esté tarificando por bytes descargados con lo que no tiene mucho sentido que se

estén consumiendo constatemente datos que no corresponden a nada que haya pedido el usuario. Elegir correctamente el parámetro del *sleep* también influye en este aspecto, pues determina la periodicidad de esta descarga.

Si el fichero finalmente se descarga se escribirá en el directorio de trabajo como ”index.html”. Para comprobar su existencia usaremos el comando estándar UNIX de *test* con la opción de *-s* que además de la existencia del archivo nos comprueba que éste tenga más de 0 bytes. Estamos hablando de una detección binaria, si se descarga algo habrá Internet y sino no habrá. No consideramos los casos en los que haya errores de conexión de temporales que puedan influenciar en lo que se descarga porque aumenta la complejidad y estos no son muy comunes. Los dos comandos irán en la misma llamada *system* concatenando uno con otro mediante el punto y coma (*;*) y comprobando que devuelve cero. Lo que se devuelve dependerá exclusivamente del resultado del segundo comando (la llamada a *test*) con lo que solo devolverá un cero en el caso de que que se haya podido descargar la homepage.

Como paso posterior es importante borrar el resultado de la descarga (el fichero index.html), para asegurarnos que si existe es porque se ha descargado en esa iteración y no en alguna anterior. Para finalizar, con el resultado de la iteración ya solo falta mandar la señal correspondiente a la aplicación principal. Para esto último podemos utilizar, como hemos comentado antes *kill* o *killall*. Con el primer comando necesitamos almacenar en algún lugar el PID de la aplicación principal, cosa que nos evitamos con la segunda opción ya que solo necesitamos saber el nombre de la aplicación, junto con la señal a enviar, obviamente. Con el nombre de la aplicación me refiero al nombre que se le ha dado al archivo ejecutable.

Una vez terminada la aplicación auxiliar que comprueba Internet hay que volver a la aplicación principal para implementar los manejadores de señal correspondientes a las dos señales que se utilizan. Necesitamos tener una variable global que defina el estado de la conexión. Dado que esta variable se modificará en los manejadores y para evitar los problemas de lectura sucia y condiciones de carrera que se pueden dar la definiremos con un tipo especial que se utiliza para estos casos. Este tipo es el llamado *sig_atomic_t*. Los manejadores lo único que harán es ponerla a uno o a cero dependiendo si llega el evento de que hay conexión o el evento opuesto, respectivamente.

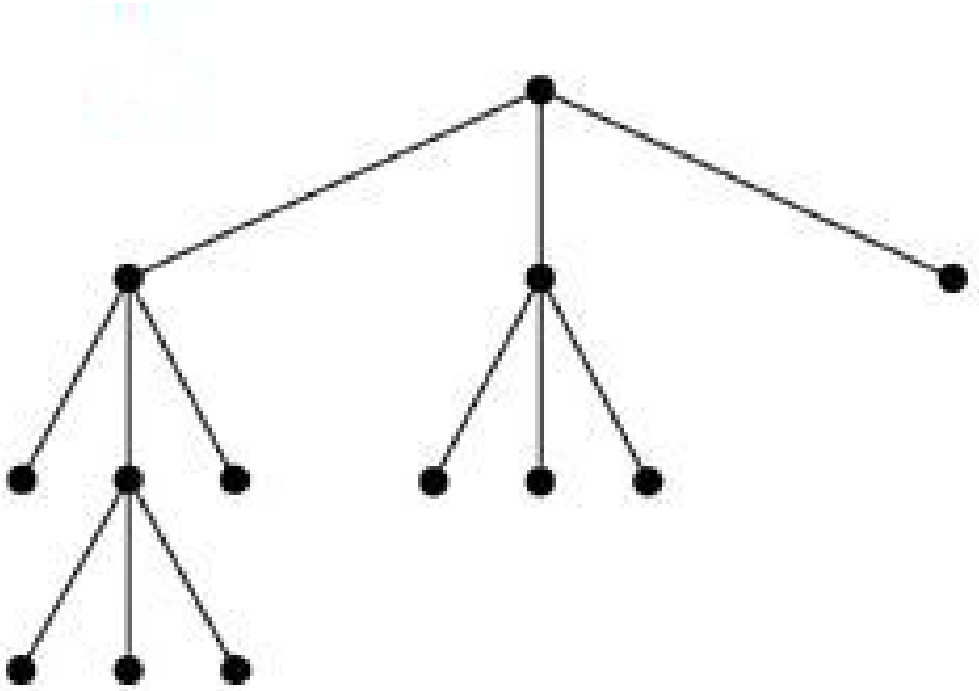
3.8. Fase 5.2.- Representación de la estructura de directorios

Dado que en el caso ”offline” no tenemos acceso a Terabox, necesitamos tener la información de los archivos disponibles de manera que podamos acceder a ella cuando la necesitemos. La primera solución para esto que podría ocurrirsenos es replicar la estructura de archivos de Terabox en un directorio temporal. Esta solución sería válida de no ser del problema surgido por la coexistencia de archivos que están en la caché con archivos que no lo están. Por ello necesitamos una alternativa que nos permita caracterizar esta circunstancia. Para ello parece conveniente conservar lo hecho en la etapa anterior y determinar si un archivo está en caché comprobando si está presente en un determinado directorio temporal. Pero para la estructura de archivos en sí necesitamos una estructura de datos que permita tener esto en memoria.

Para ello recurrimos en primer lugar a alguna librería que nos ofrezca estructura de datos y que ya esté presente en el libro. Dado que el e-reader usa GNOME, cabe utilizar la librería **glib** que es una librería de carácter general que implementa diversas funciones no gráficas. Lo único que hay que percartarse es la versión específica que usa el libro, que al igual que pasaba con el kernel, será una versión bastante más antigua que la actual en el momento de desarrollar la aplicación. La versión que hay en el libro es la 2.06 frente a la 2.22 que es la actual en el momento de la implementación. GLib implementa una amplia gama de tipos de datos con lo que era muy posible que alguno de ellos sirviera para representar un árbol de directorios y archivos básico.

GLib tiene dos tipos de estructuras de datos que implemantan árboles. Son los árboles binarios balanceados y los los árboles n-arios. Dado que de un directorio cuelgan un número indeterminado de archivos y directorios, la segunda estructura resulta más conveniente. La estructura tiene el aspecto de la figura 3.4

Figura 3.4: Esquema de un arbol n-ario



Después hay que crear una estructura de datos que irá alojada en cada nodo del árbol y que contendrá la información necesaria de cada archivo / directorio. Lo primero que debe tener esta estructura es un indicador de si el nodo es un archivo o un directorio. Cabría preguntarse si este dato es necesario sabiendo que de los directorios cuelgan más archivos y otros directorios. Sin embargo en esta estructura de datos cada nodo hoja puede ser potencialmente un nodo padre de otros nodos y puede darse el caso de que que el directorio esté vacío con lo que no habría forma de diferenciarlo. El resto de información lo sacamos de la información que necesitamos devolver en la operación *get_attr*.

Por último habrá que repensar los estados ya que ahora tenemos un estado adicional correspondiente al caso de que pasamos de offline a online y todavía no hemos reconstruido en memoria el estado del árbol de directorios actual que nos devuelve la llamada a Terabox. También impondremos la asunción de que los archivos en Terabox no varían mientras no pasemos de online a offline de nuevo, o bien no apaguemos el libro electrónico. Esta asunción no limita la funcionalidad de la aplicación ya que mientras se está usando el libro no debería haber cambios en los archivos de Terabox, ya que se está en un entorno de movilidad lejos del PC de escritorio que es donde se suben los libros. De todas formas en el caso de que estos cambios se producieran bastaría con reiniciar el libro, o como implementaremos en fases venideras, desconectar el

módem 3G USB y volverlo a conectar.

3.8.1. Descripción de la implementación

En primer lugar habrá que modificar el manejador de la señal correspondiente al evento de estar online para que considere el caso de que cambiamos de offline a online. Asociaremos a este modo adicional el entero ”dos”. Ahora por tanto las transiciones de estado cuando se recibe la señal de estar ”online” quedan según esta tabla.

Estado actual	Estado nuevo
0	2
1	1
2	2

Operación getdir

El siguiente paso es partir de la primera operación a la que se llama en orden cronológico que es *getdir*. En ese momento suponemos que estamos en el estado ”dos” (si estamos en el cero no podemos hacer nada porque no tenemos conexión ni información previa y en el estado ”uno” no podemos estar sin haber pasado antes por el ”dos” ya que partimos del estado de desconexión y todavía no hemos regenerado en memoria la estructura de directorios). Lo primero de todo es conectarse a Terabox para obtener la información de la estructura de directorio. Dado que podemos estar online u offline, cambiamos el momento de conectarnos del inicio a la aplicación a este preciso momento que es cuando realmente necesitamos acceder y hemos comprobado que podemos hacerlo porque estamos online.

Una vez conectados pasamos a extraer la información de los archivos y directorios, que lo hacemos en una función auxiliar llamada *regenerate_dir*. En esta función necesitaremos utilizar la estructura de datos con la información de cada archivo y directorio que por tanto es necesario definir. Esta estructura tendrá en primer lugar la información de si es un fichero o un directorio. Esto lo codificamos como como un tipo *enum* que sólo puede tener dos valores: *file* o *dir*. Después tendrá los atributos nombre, tamaño y fecha de creación en el mismo formato que comparten FUSE y Terabox. Tras definir esta estructura ya podemos empezar con la función *regenerate_dir* que comenzará creando el nodo raíz que se corresponde con el directorio raíz

desde donde cuelga toda la estructura de archivos de Terabox. Lo primero de todo es reservar memoria para la creación de la estructura de datos que posteriormente la rellenaremos con una cadena vacía como nombre, un tamaño de 0 bytes (a estos datos no se va a acceder nunca, de hecho ni siquiera rellenamos la parte de la fecha de creación) y lo esencial que es marcar que es un directorio. Tras tener la estructura de datos definida, creamos el nuevo árbol n-ario con la llamada a la función *g_node_new* pasando como parámetro esta estructura con la información de archivos o directorio que a partir de ahora llamaremos ”member”. Resultará conveniente tener este nodo raíz almacenado en una variable global ya que esta estructura va a ser utilizada en diferentes funciones de la aplicación.

El siguiente paso es comenzar un proceso recursivo para ir recorriendo todo el árbol de archivos y directorios que cuelga de Terabox e ir replicándolo en memoria. Para ello utilizaremos la función auxiliar *fill_dir_tree_recur* pasándole el nodo raíz. Esta función lo primero que hace es llamar a la función que nos proporciona el contenido de un directorio en Terabox que es *ddvs_directory_contents*. A esta función se le pasa el path correspondiente de Terabox como entrada y deja los datos en una estructura llamada *vault__Directory_Contents*. Para obtener el path de Terabox necesitamos en primer lugar construir el path a partir del nodo en el que estamos actualmente y después añadir como prefijo del directorio concreto de Terabox en el que estarán los archivos correspondientes al libro. Esto último lo hace la función *getRealPath*. Para la construcción del path en sí usamos la función *getPath* que a partir del nodo del árbol en el que nos encontramos (el que se ha pasado como parámetro a la función recursiva) va construyendo el path de la siguiente manera: se van concatenando al nombre del nodo, los nombres de los nodos padre separados por el signo de barra (/) hasta llegar al nodo raíz.

Una vez construido el path de Terabox, la llamada a *ddvs_directory_contents* nos devuelve la estructura *vault__Directory_Contents* que tiene dos arrays, uno con los ficheros que contiene el directorio y otro con los subdirectorios. Tras esto es el momento de bajar al directorio local temporal correspondiente al nodo, que no es más que calcular el path de nodo con *getPath* como hemos hecho antes solo que esta vez de añadimos el prefijo del directorio local temporal donde se irán almacenando los archivos de la caché. La idea es que en este directorio local se replique la estructura de directorios de Terabox y que se aproveche este momento en el que se está recorriendo el árbol entero para ir creando cada subdirectorio local. Esto lo hacemos recorriendo el array con los subdirectorios (*vault__ArrayOfDirectory*), excluyendo los directorios

que empiezen por un asterisco (*) por ser directorios especiales y llamando la función recursiva en cada uno de ellos, habiendo creando previamente el nodo hijo mediante la función auxiliar *fill_dir*. Esta última función crea el nodo hijo a partir de su entrada en el array *vault__Directory* y el subdirectorio correspondiente en la caché local temporal. La manera de hacerlo es crear una nueva estructura *member* con la información del directorio (nombre, tamaño y fecha de creación), crear como ya hemos dicho el subdirectorio de la caché (dado que antes nos hemos situado en el subdirectorio padre nos olvidamos del path absoluto y solo especificamos el nombre del subdirectorio con *mkdir*). Por último añadimos el nodo correspondiente al subdirectorio colgando del nodo padre con el contenido de la estructura *member* que acabamos de crear. La función de **GLib** que se usa para esto es *g_node_append_data*. La función auxiliar devuelve el nodo hijo recién creado de manera que se pueda volver a llamar a función recursiva con el nodo del subdirectorio y se repita todo el proceso. Una vez recorridos todos los subdirectorios, es el turno de recorrer los archivos, caso que lógicamente es mucho más simple ya que solo hay que crear las estructuras *member* con la información existente en las entradas del array de archivos en sus correspondientes nodos que insertaremos en su nodo padre como hemos hecho previamente con los directorios.

Una ver regenerada en memoria la estructura de archivos y directorios, podemos pasar a la funcionalidad en sí misma de la operación en la que estamos, *get_dir*, que no es otra que mostrar los contenidos de un determinado directorio. La novedad respecto a fases anteriores es que ahora en vez de buscar directamente en Terabox, buscamos en la estructura de datos que tenemos en memoria (el árbol n-ario). Buscaremos ahí independientemente de que estemos online u offline ya que hemos hecho la suposición de que los contenidos de Terabox no varían mientras no se cambie de modo. En primer lugar, necesitamos una forma de localizar un nodo dado de la estructura a partir del parámetro de la operación FUSE que es el path del directorio. Para ello utilizaremos la función auxiliar *get_node_by_path*.

Esta función lo que hace es ir troceando el path en fragmentos delimitados por la barra (/) y, empezando por el nodo raíz va buscando subdirectorios cuyo nombre sea igual al fragmento que acaba de trocear. Para ello utilizamos la función de **GLib** *g_node_children_foreach* que va llamando a una función dada con cada uno de los nodos hijos. Con la opción *G_TRAVERSE_ALL* especificamos que visite todos los nodos hijos. Necesitamos introducir en una estructura todos los parámetros que necesite la función auxiliar que se va a ir llamando. Esta estructura

tendrá dos atributos, el primero es *foundChild* y se rellenará si se ha encontrado el nodo a buscar. El otro parámetro es *str* que es el fragmento del path que hay que ir buscando. La función auxiliar tan solo tiene que comprobar que no se ha encontrado ya al nodo que se busca y sino es así comparar el nombre del nodo con *str*. Si la comparación es positiva se iguala *foundChild* al nodo en cuestión. De esta manera la siguiente iteración se hará con el siguiente fragmento de path y se visitarán los hijos del nodo encontrado por *foundChild*. Si alguno de los fragmentos no sirve para encontrar un nodo la función devolverá *NULL* y al no encontrarse el directorio se enviará el error correspondiente de que no existe entrada con ese nombre: *ENOENT*. Si lo que se encontrara es un archivo en vez de un directorio, entonces el error será *ENOTDIR*. Si finalmente se encuentra el nodo correspondiente al directorio lo que hay que hacer es recorrer todos sus nodos hijos y apuntar sus nombres siguiendo los mecanismos que nos ofrece FUSE. La función de **GLib** a utilizar vuelve a ser la misma que usábamos para buscar el nodo, ya que a nivel de estructura de datos hacemos lo mismo, visitar todos los nodos hijo. Esta función era *g_node_children_foreach* y ahora utilizaremos una nueva función objetivo y una nueva estructura para pasársela como argumento. La estructura solo contiene el buffer y la función ”filler” que escribe en ese buffer; ya que lo único que hay que hacer cuando visitamos cada nodo hijo es escribir su nombre usando esos parámetros de la estructura tal y como hacíamos en fases anteriores. Para finalizar se devuelve un ”0” si se ha conseguido realizar correctamente la operación y no hay errores por tanto.

Operación getattr

La siguiente operación en orden cronológico es obtener los atributos de cada archivo o directorio ya que de *get_dir* solo se obtienen los nombres a secas. Conceptualmente no aporta mucho respecto a la operación anterior (*get_dir*) ya que al igual que esta operación primero hay que encontrar el nodo en la estructura de datos correspondiente (con la función auxiliar *get_node_by_path*). Una vez encontrado este nodo en su estructura *member* tendremos la información que antes obteníamos directamente del API de Terabox sin mayor variación respecto a fases anteriores.

Operación open

En esta fase en vez de usar el API de Terabox, como hacíamos en fases anteriores, hay que buscar el nodo correspondiente al path dado en la estructura de datos. Este paso es el mismo que en el resto de operaciones de esta fase y se realiza con la función auxiliar *get_node_by_path*.

Operación read

No hay ninguna variación en esta fase.

3.9. Fase 5.3.- Almacenamiento de la estructura de directorios

3.9.1. Introdución al concepto de la serialización

La serialización (o marshalling en inglés) consiste en un proceso de codificación de un objeto (programación orientada a objetos) en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML o JSON, entre otros. La serie de bytes o el formato pueden ser usados para crear un nuevo objeto que es idéntico en todo al original, incluido su estado interno (por tanto, el nuevo objeto es un clon del original). La serialización es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones. Varios lenguajes de programación orientados a objeto soportan la serialización de forma directa. El lenguaje que utilizamos, C estándar, no está entre ellos.

3.9.2. Descripción de la serialización

Pese a no tener un método directo soportado por el lenguaje para serializar, existe una solución sencilla a poco que uno se ponga a pensar sobre el problema concreto en nuestra aplicación, al menos en lo que la estructura se refiere. Estamos continuamente pasando del path al nodo de la estructura de datos. Existe por tanto una relación unívoca entre el path y un nodo dado de la

estructura y éste puede ser la forma más correcta de serializar la estructura del árbol. Algo más complejo es a priori la serialización del contenido de cada nodo, pero no lo es tanto si tenemos en cuenta que todos los campos de la estructura *member* tienen tamaño fijo exceptuando el nombre. En realidad el nombre como tal no lo serializaremos aisladamente, ya que para conservar la estructura necesitamos el path entero y el nombre está incluido al final del path. En la siguiente tabla se resume la información a serializar por nodo:

Campo	Tamaño
path	*variable*
size	sizeof(ULONG64)
type	sizeof(fileType)
creationDate	sizeof(time_t)

3.9.3. Elección del mecanismo de almacenamiento: GNU dbm

Dado que sabemos la estructura de la información a almacenar, cabría la posibilidad de escribirlo todo en un archivo en formato crudo un nodo detrás de otro. Sin embargo la existencia del path como campo de tamaño variable (string) puede complicar un poco el manejo de la información. Esto lo podemos solucionar con una librería para C ligera (requisito necesario para el entorno embebido) de base de datos simple como es **GNU dbm**. *dbm* fue el primero de una familia de mecanismos de almacenamiento, originariamente escrito por Ken Thompson y liberado por AT&T en 1979. El nombre es un acrónimo de ”database manager”. *dbm* guarda datos arbitrarios por el uso de una clave única (una clave primaria) en ”recipientes” (buckets en inglés) de tamaño fijo y usas técnicas de hashing para permitir un acceso rápido de los datos a partir de la clave. El esquema de hashing es una forma de ”hashing extendible” en el que el esquema de hashing se expande a medida que nuevos recipientes son añadidos a la base de datos, significando esto que, cuando está prácticamente vacía, la base de datos empieza con un recipiente, el cual se divide cuando se llena. Los dos recipientes hijos resultantes se volverán a dividir a si mismos cuando se llenen, de tal forma que la base de datos crece a medida que las claves se añaden. Dado que **dbm** y sus derivados son bases de datos pre-relacionales, en la práctica pueden ofrecer una solución más útil para almacenamiento de alta velocidad buscado por clave que no requiere el sobrecoste computacional de conexiones y preparación de peticiones. Esto se equilibra con el hecho de de que generalmente sólo pueden ser abiertas para escribir por un

único proceso a la vez. **GDBM** o **GNU dbm** es un sucesor de **dbm** con una licencia libre para el proyecto **GNU**. Añade soporte para datos de longitud arbitraria: anteriormente todos los datos tenían que tener una longitud máxima. Ésta última capacidad es la que justifica su elección como mecanismo de almacenamiento en esta aplicación.

3.9.4. Implementación de la serialización

Una vez elegido el mecanismo de almacenamiento veremos como diseñar la implementación. **GDBM** hace que se pueda recuperar la información a partir de una clave. Además, se almacena el tamaño de esa información. Para simplificar la asignación de claves almacenaremos toda la información de cada nodo junta y su clave será un entero positivo. El tamaño del campo path lo obtendremos indirectamente restando del tamaño del nodo serializado total que nos lo da **GDBM**, el tamaño de la suma del resto de los campos. El hecho de asignar a cada nodo una clave numérica nos permite establecer un orden que garantiza que los nodos padres se creen siempre antes que los nodos hijos. Así garantizamos que al deserializar podremos insertar siempre el nodo hijo colgando del nodo padre, ya que éste habrá sido creado previamente. Por último pondremos una dupla (clave, dato) extra para simplificar el proceso cuya clave será ”SIZE” y cuyo valor será el número de nodos en total con el fin de saber cuándo parar de deserializar nodos.

Leyendo el API de GDBM cabría plantearse usar las instrucciones *gdbm_firstkey* y *gdbm_nextkey* para sacar todos los nodos ahorrándonos el uso de la variable ”SIZE”, ya que cuando *gdbm_nextkey* no devolviera nada es que no existen más nodos. Sin embargo en ese mismo API se especifica que no se garantiza que el orden de inserción se respete a la hora de extraer los nodos con este método. Por tanto descartamos este método alternativo

Serialización

El momento de iniciar el proceso será cuando la aplicación FUSE se cierre. Esto ocurrirá cuando se salga de la función *fuse_main*. Inmediatamente después de la llamada a esta función colocaremos la llamada a la función que realiza esta tarea de serialización que es *save_dir_tree*. Aquí comenzaremos con la inicialización de la base de datos mediante la función *gdbm_open*. El primer parámetro de esta función es el nombre de la base de datos (en realidad

el archivo donde se alojará ésta dentro del e-reader) y lo definiremos en una macro de precompilación. Así tendremos la ubicación de este archivo localizada en un solo lugar al principio del código fuente por si posteriormente queremos modificarla. El siguiente parámetro es el tamaño de bloque, dado que los datos que vamos a manejar son de tamaño extremadamente pequeño, no merece la pena cambiar este parámetro de su valor por defecto dependiente de la plataforma que se establece asigándole un ”0”. En realidad cualquier valor para este parámetro inferior a ”256” sería igualmente válido, como se especifica en la página del manual de **GDBM**. El siguiente parámetro *GDBM_NEWDB* especifica que se cree una nueva base de datos aunque exista previamente otra anterior con el mismo nombre. Éste será el caso de nuestra aplicación que, salvo la primera vez que se ejecute la aplicación (o que cambiemos la ubicación de la base de datos en la macro de compilación) tendremos una base de datos anterior que necesitaremos sobreescribir completamente dado que la serialización se hace de manera íntegra (todos los nodos de la estructura de directorios) cada vez que se cierra la aplicación en base al estado de esta estructura en memoria únicamente y sin utilizar información anterior. A continuación tenemos el modo con el se creará el nuevo fichero de la base de datos. Este parámetro tiene la misma forma que el *chmod* **UNIX** y lo estableceremos en ”0666” que permite la lectura y escritura por todo el mundo. Estamos en un entorno embebido monousuario y por tanto no es necesario preocuparse por los permisos de los usuarios. Lo único importante es que los permisos de lectura estén activos con el fin de poder abrir la base de datos en el proceso de deserialización, así como obviamente los permisos de escritura para poder escribir en este proceso de serialización. El último parámetro es un puntero a una función para el tratamiento de errores fatales que lo dejaremos a NULL, dado que consideraremos que no se producirán errores en el proceso de serialización/deserialización.

El siguiente paso es recorrer todos los nodos de la estructura. Para ello necesitamos una estructura con toda la información que necesita la función auxiliar que se va llamando con cada nodo. Estos parámetros son la estructura vinculada a la base de datos abierta (necesaria para poder escribir en ella) *GDBM_FILE* y que la hemos referenciado como resultado del paso anterior. El otro parámetro es el contador de nodos que se va incrementando a medida que vamos serializando los nodos. Es ahora cuando lo ponemos a cero.

Una vez establecidos los argumentos de la función auxiliar, empezamos el recorrido por la estructura usando la función *g_node_traverse*. El primer parámetro es la estructura que tenemos

almacenada en la variable global *tree*. El segundo es el orden que como ya hemos comentado hay que visitar primero el nodo padre y después los hijos. Esto lo aseguramos poniendo este parámetro igual a *G_PRE_ORDER*. Con el tercer parámetro especificaremos que visite nodos hoja (nodos que no tienen hijos, que en esta fase serán directorios vacíos y archivos) y nodos no hoja (nodos con hijos, directorios no vacíos). La manera de hacer esto es usando el parámetro *G_TRAVERSE_ALL* que no es más que un **OR** lógico entre los flags *G_TRAVERSE_LEAVES* (para que se visiten los nodos hoja) y *G_TRAVERSE_NON_LEAVES* (lo mismo para los nodos no hoja). El cuarto parámetro especifica el número de niveles del árbol n-ario a recorrer. Esto se refiere a la profundidad en el árbol, donde 1 será el nodo raíz, 2 sus hijos y así sucesivamente. Lo pondremos a "-1" para que no exista limitación en la profundidad. Los últimos argumentos son la función auxiliar y la estructura de argumentos auxiliares que ya hemos explicado. Lo que hará esta función del API de **GLib** es ejecutar esta función auxiliar con cada nodo de la estructura y los parámetros auxiliares.

La función auxiliar *save_dir_tree_iter* comienza recuperando la base de datos y el contador de los nodos que llevamos serializados hasta ahora de la estructura de argumentos auxiliares. Cabe reseñar que actualizaremos el contador a la vez que lo recuperamos de la estructura usando el operador de preincremento (++). Alternativamente se podía haber utilizado el operador de postincremento inicializandolo a "1" en vez de a "0". También recupera la estructura *member* del nodo en cuestión que es la que contiene sus características. Una vez hecho esto construimos el path recorriendo la estructura desde la raíz hasta el nodo y concatenando cada nombre con la barra (/). De esta manera serializamos dónde va situado el nodo dentro del árbol n-ario. Además reutilizamos la función de creación del PATH que nos sirve, además, para localizar la ubicación del archivo en Terabox y en la caché temporal. Es ahora cuando tenemos todos los datos necesarios para serializar el nodo. Es por tanto el momento de calcular el tamaño que ocupará en la base de datos. Para ello necesitamos calcular el tamaño del path, que logicamente cambiará con cada nodo. Al ser un string, basta con utilizar la función ANSI C *strlen* para obtenerlo. El tamaño total será, pues, la suma del tamaño del path con la de los tamaños fijos de cada campo de información del nodo por separado. Una vez que tenemos el tamaño ya podemos reservar la memoria necesaria para poner toda la información tal cual será serializada. La manera de operar con la base de datos es mediante la dupla de estructuras *datum key* (clave) y *data* (datos). Cada una de estas estructuras tienen un puntero *dptr* hacia donde está la información en sí (la

clave y los datos) y una variable *dsize* igual al tamaño de esta información. El puntero que nos devuelve *malloc* con la reserva de memoria que acabamos de hacer directa lo igualaremos a *data.dptr*. El tamaño también lo acbamos de calcular. Respecto a la clave, para sacar el puntero a los datos basta con obtener la direcciónd de memoria del contador. Respecto al tamaño del contador cabe reseñar que será el más el del entero sin signo con más capacidad posible de los tipos estándar de C. Me parecía absurdo limitar la capacidad del número de nodos que se pueden serializar (y con ello el número máximo de archivos y directorios en la aplicación) cuando es tan pequeño el tamaño que aún así iban a ocupar estas claves en la base de datos más aún cuando ésta estará previsiblemente alojada en la tarjeta Compact Flash cuyo tamaño será relativamente grande. Este tipo es *unsigned long long*. El siguiente paso es copiar la información del nodo toda seguida en la porción de memoria que hemos reservado para los datos a serializar. Empezaremos con el path que lo copiaremos al principio del todo sin terminarlo con el carácter 0. El motivo es que siempre sabremos de antemano el tamaño del path y por tanto no necesitamos marcar el final con un carácter especial. Esto lo conseguiremos utilizando *strncpy* en vez de *strcpy* aunque podríamos haber utilizado *memcpy* igualmente. Los siguientes datos se copian directamente con *memcpy* especificando los tamaños de cada campo y teniendo cuidado de incrementar el puntero al punto donde se copian el tamaño del campo anterior. El último paso es llamar a *gdbm_store* para culminar la serialización con las estructuras *datum* de la clave y los datos y la base de datos que abrimos al principio. El hecho de establecer como flag *GDBM_INSERT* resulta irrelevante, pues hemos abierto la base de datos con la opción *GDBM_NEWDB* y por tanto no habrá datos previos en la base de datos. El uso del contador nos garantiza que las claves no se repetirán. Finalmente, esta función auxiliar debe devolver *FALSE* siempre para que se puedan visitar todos los nodos.

Una vez recorridos todos los nodos es el momento de escribir en la base de datos el número de nodos totales bajo la clave ”SIZE”. Para finalizar la serialización se cierra la base de datos con la función *gdbm_close*.

Deserialización

La deserialización comenzará al principio de la aplicación justo antes del método que inicia la recepción de llamadas a las operaciones FUSE *fuse_main*. Para ella utilizaremos la función *load_dir_tree*.

Una vez en esta función, creamos la estructura *member* del nodo raíz que tendrá como nombre la cadena vacía (””), como tipo directorio (lógicamente) y un tamaño de 0 bytes. Esta estrucura se la pasamos a la función *g_node_new* y el árbol n-ario quedará creado con el nodo raíz. Después abrimos la base de datos igual que habíamos hecho al inicio de la serialización, pero esta vez lo hacemos con el parámetro *GDBM_READER* ya que esta vez abrimos la base de datos únicamente para leer de ella. Después hay que comprobar que efectivamente hemos abierto la base de datos, ya que es posible que sea la primera vez que ejecutamos la aplicación o que por alguna razón hayamos borrado la base de datos entre dos ejecuciones de la aplicación. En caso de que esto suceda hay q salir de la función porque no se puede realizar la deserialización. En caso de que ésta si pueda ser posible, empezaremos a obtener datos de la base de datos. La función que usaremos para ello será *gdbm_fetch* que pasándole como parámetro un *datum* clave, te devuelve el *datum* dato correspondiente a esa clave. Lo primero que obtendremos será el número de nodos, que habíamos guardado con la clave ”SIZE”. Cabe reseñar que el tamaño de esta clave será 4 bytes, porque hemos introducido esta clave sin el carácter especial ”\0” al final. Una vez que tenemos el número de nodos vamos iterando hasta llegar a ese número con un bucle *for*. En cada iteración la clave será el número de nodo y una vez obtenido el nodo serializado lo pasamos a la función auxiliar *load_node*.

Esta función crea en primer lugar la estructura *member* que contendrá la información del nodo con un *malloc*. Después se saca el tamaño del path restando del tamaño total del nodo serializado (contenido en la estructura *datum* que se pasa como parámetro), el tamaño del resto de los campos menos el path y que tienen un tamaño fijo. A continuación reservaremos memoria para el path, pero dejando una posición de memoria más con el fin de introducir el carácter especial ”\0” con el que terminan los strings normalmente y que habíamos suprimido en la serialización. Inicializamos un puntero auxiliar que va a ir apuntando a la posición de cada dato dentro del nodo serializado al principio de los datos (*data.ptr*). En esta posición estará el path, lo copiamos al fragmento de memoria que acabamos de reservar y ponemos un carácter especial ”\0” al final. El siguiente paso será separar del path el nombre del nodo del directorio donde se encuentra. Para ello usaremos la función auxiliar *getName* que nos apuntará al carácter posterior a la última barra (/) y almacenaremos esa posición en **name*. Éste será el nombre del nodo. Después sustituiremos la última barra del path por el carácter especial ”\0”. Con esta operación tenemos el directorio en el string **path* y el nombre del nodo en **name*. Llamando

a la función *get_node_by_path* con el path obtenemos el nodo padre (el directorio donde se encuentra) del que estamos deserializando. Con esta implementación reutilizamos el código de la función *get_node_by_path*. Continuamos copiando el nombre del nodo al lugar correspondiente de la estructura *member*. Lo mismo hacemos con el resto de los campos de la estructura, incrementando el puntero el tamaño del campo anterior para obtener donde se encuentra el dato serializado. Una vez que tenemos todos los datos en la estructura, insertamos el nuevo nodo en el nodo padre con *g_node_append_data*.

Una vez que hemos iterado sobre todos los nodos cerramos la base de datos con *gdbm_close* y con ello culminamos el proceso de deserialización.

3.10. Fase 6.- Integración con la interfaz gráfica del Irex Iliad

3.10.1. Descripción de la interfaz del Irex Iliad

En la figura 3.5 podemos ver el aspecto que tiene el menú en el que se muestran los libros existentes en un directorio dado. A partir de este menú el usuario haciendo doble click con el estilete sobre un libro puede abrirlo para su lectura. Por cada libro tenemos el icono del libro blanco sobre negro en la columna de la izquierda, el nombre del libro en la columna central y por último una descripción del libro en la columna de la derecha. La información de la descripción la genera el libro creando un directorio del mismo nombre que el archivo y copiando el libro dentro de ese directorio. Además crea un fichero ”manifest.xml” donde guarda la información correspondiente a esta descripción, el archivo que se abrirá tras hacer doble click y el icono que se muestra en el menú.

En primer lugar vemos que el e-reader escribe en el sistema de archivos para generar todo esto y nuestra aplicación no soporta la escritura, con lo que no podrá construir los archivos necesarios para la interfaz gráfica. Sin embargo, nada nos impide que nuestra aplicación genere por sí misma estos archivos y personalicemos la interfaz adecuándola a nuestras necesidades. Además podíamos distinguir los casos que, estando offline, exista el archivo en la caché o no. Para cada uno de estos casos es posible diseñar una interfaz específica.

Lo que haremos será cambiar el icono para el caso de que el archivo no esté en la caché y estemos offline, quede claro que aunque dicho archivo esté en Terabox, no está disponible en

Figura 3.5: Interfaz gráfica del Irex Iliad

ese momento en el libro y por tanto no se puede visualizar. Si aún así hacemos doble clic sobre el libro ”no disponible”, podemos conseguir que se muestre un mensaje de error. Para ello lo que habrá que hacer es ir modificando el fichero ”manifest.xml” y diseñar iconos adecuados para el caso de que el archivo no está disponible. El siguiente icono se mostrará en el caso de un libro PDF (figura 3.6):

Figura 3.6: Icono e-book PDF

En el caso de que no esté disponible se mostrará este otro icono elaborado especialmente para esta aplicación (figura 3.7):

Figura 3.7: Icono e-book PDF (no disponible)

El hecho de que aparezca partido simboliza gráficamente que el libro no está disponible, a modo de un enlace roto. Prepararemos iconos especiales (partidos) también para el formato con DRM Mobipocket (.prc) (figura 3.8):

Figura 3.8: Icono e-book Mobipocket (no disponible)

y para el formato (.TXT) (figura **??**):

Figura 3.9: Icono e-book TXT (no disponible)

Además para el caso en la tercera columna del menú pondremos un mensaje avisando de la no disponibilidad. Por último estableceremos que en el caso de que aún así el usuario quisiera abrir el archivo se abra un documento con una única frase ”El archivo no está disponible porque el Irex Iliad no está conectado a Internet”. Este mensaje que explica completamente la situación resulta demasiado largo para ponerlo directamente en el menú, donde pondremos un mensaje más escueto: ”No disponible”.

Respecto del caso de tener conexión a Internet pero el archivo no está en la caché, mostraremos en la tercera columna el mensaje: ”El archivo tardará unos segundos en descargarse..”. Esto lo haremos porque, aunque finalmente podrá leer el libro, lógicamente los tiempos de carga serán mucho mayores y esto interferirá en la experiencia de usuario.

3.10.2. Implementación de la integración en la interfaz gráfica

Comenzamos la descripción de la implementación con el primer hecho que ocurre cronológicamente, el inicio de la aplicación. Habitualmente lo que ocurrirá será que habrá una base de datos con la información de la estructura de archivos y se deserializará. En este proceso, que ya hemos descrito previamente, es donde introducimos la descripción para la interfaz gráfica del Iliad. Concretamente es en la función *load_node* tras haber añadido el nodo al árbol con *g_node_append_data*. Lo primero que hacemos es comprobar si el nodo se corresponde a un e-book. Para ello simplemente nos fijamos en la extensión para ver si es alguno de los tres tipos de libro que hemos considerado en la sección anterior y que resumimos en el cuadro 2.1. El caso del archivo MobiPocket, aunque como hemos visto anteriormente habíamos creado un icono especial para él, finalmente no lo consideramos para simplificar la implementación y es tratado como si fuera un archivo de texto.

Extensión	Tipo de libro	¿Implementado?
.TXT	Archivo de texto	Sí
.PDF	Archivo PDF	Sí
.PRC	Archivo MobiPocket	No

Cuadro 3.1: Resumen de los tipos de e-book

La implementación de estos comprobadores de extensiones es sencilla, buscamos con *strchr* la posición del punto con el que comienza la extensión y a partir de ahí comparamos con los tres caracteres correspondientes.

Una vez determinado si el archivo en cuestión corresponde a un e-book hay que ver si éste está almacenado en la caché o no. Para ello utilizamos la función *check_availability_of_file* en la que en primer lugar obtenemos la ruta de la caché en la que debería estar el e-book. Para ello intentaremos reutilizar al máximo las funciones que hemos implementado previamente como estrategia de diseño.Así, encontramos el directorio temporal a partir del nodo del árbol correspondiente al e-book, que acabamos de crear, usando la función *getPath*.[4] A continuación le concatenamos el nombre que para simplificar se lo pasamos por parámetro en vez de utilizar la función *getName*. Por ejemplo un archivo **Quijote.pdf** dentro del directorio **/CervantesBooks** dará lugar a **/CervantesBooks/Quijote.pdf/Quijote.pdf**. Una vez construido la ruta del archivo añadimos el prefijo correspondiente a la ruta de la caché con *getTemporalPath* y comprobamos la existencia del archivo temporal usando el comando *test -f* que hemos descrito anteriormente.

Caso de archivo no disponible

En el caso de que el archivo no esté disponible en la cache pasamos el nodo del e-book a la función *fill_not_available_file*. En esta función obtenemos en primer lugar el directorio en el que se aloja el archivo. Para ello obtenemos la ruta del directorio padre. Siguiendo el ejemplo anterior sería **/CervantesBooks**. Después añadimos el prefijo de la caché y ya tenemos la ruta donde se alojarán los metadatos y el icono. Tras situarnos en el directorio padre con *chdir*

[4]Para clarificar esto es necesario recordar que el e-reader crea un directorio de nombre igual al e-book e introduce el archivo en este directorio. En este directorio se alojará el archivo del e-book en sí mismo (en el caso de que esté descargado), además del archivo de metadatos *manifest.xml*, el icono correspondiente y el archivo de error. Lógicamente todo esto tendra su correspondiente nodo en el árbol n-ario.

podemos crear el directorio correspondiente al e-book usando rutas relativas.[5] Utilizaremos para ello el comando *mkdir* con la opción *-p* ya que ello nos permite ignorar el caso de que el directorio esté creado previamente.

A continuación copiaremos el icono correspondiente, que será o bien el del PDF o el del archivo TXT en su versión partida por la mitad. El caso de los archivos Mobipocket decidimos no implementarlo como hemos comentado anteriormente. También copiaremos el archivo PDF con el mensaje de error, ya que estamos en el caso **no disponible**. Para ello utilizaremos la macro de compilación correspondiente al comando para copiar el icono correpondiente y que hemos definido al principio del archivo fuente. Aquí en primer lugar definimos las rutas donde se encuentran los iconos. En primer lugar definimos el nombre de los iconos con *ERROR_ICO_PDF* y *ERROR_ICO_TXT* y el nombre del archivo de error *ERROR_PDF*. Después a esto se concatena (volviendo a usar macros del preprocesador) el directorio en el que se encuentra cada archivo. Con este esquema obtenemos máxima flexibilidad para poner archivo en el sitio que consideremos oportuno. Por ejemplo, tenemos una ruta para el caso de que estemos compilando para el Iliad y otra ruta distinta si es la versión para el ordendador de escritorio. Por último concatenamos cada ruta con el comando *cp* y el directorio destino (el actual *./*). Todo este proceso se hace en la etapa del preprocesador. Con estos comandos ya generados solo queda ejecutarlos dentro de la función con *system*.

El siguiente paso es borrar el archivo de metadatos antiguo, usando *rm* con la opción *-f* para considerar el caso de que el archivo de metadatos no existiera. Tras ello escribimos el archivo de metadatos en el que que iremos intercalando el texto que no varía con los tags XML que influyen en la representación de la interfaz que queremos mostrar. Este texto fijo estará almacenado en macros del preprocesador al principio del archivo con el fin de tenerlo lo más agrupado posible. Este esquema pese a ser algo engorroso evita la utilización de librerías XML que suponían invertir un tiempo adicional para aprender a utilizarlas. Especificaremos con el tag **Title** que el nombre del archivo aparezca donde pone *Book Title* ... en la figura 3.5). Mediante la modificación del tag **Description** conseguiremos que el mensaje que aparece en la columna de la derecha, sea en este caso *El archivo no está disponible*. Con el tag **image** especificamos el icono(que dependerá de si es un e-book PDF o TXT y que estará partido para representar la no

[5]El hecho de situarnos en el directorio del e-book y usar rutas relativas simplificará la implementación al no tener que estar continuamente construyendo rutas absolutas.

disponibilidad) y con **ItemSize** el tamaño del archivo que tenemos almacenado en el nodo del libro del árbol n-ario. Por último necesitamos especificar qué se va a abrir en realidad cuando pinchemos con el estilete. Será el archivo de error que hemos copiado previamente y cuya ruta tenemos en una macro del preprocesador propia y al no variar se combinará con el texto fijo durante el preprocesado. El tag XML en el que se introduce es **startpage**.

Una vez escrito el archivo de metadatos es el momento de reflejar estos cambios en el árbol n-ario. En primer lugar buscamos el nodo correspondiente a un archivo de metadatos anterior. La manera de hacerlo será mediante la ruta pasada a la función *get_node_by_path*. Lógicamente el archivo de metadatos colgará del directorio correspondiente al e-book del que proporciona la información. Usando el ejemplo que estamos utilizando en este capítulo la ruta del archivo de metadatos será **/CervantesBooks/Quijote.pdf/manifest.xml**. En esta ocasión la implementación elegida es escribir una variable mediante *sprintf*. Esta es la solución más sencilla para construir la ruta. En el caso de que encontremos el nodo correspondiente al archivo de metadatos (que es el caso de que éste existiera previamente) tenemos que modificar la información de este archivo que contiene el nodo. La manera de obtener la información del archivo será mediante la llamada a la función *stat* de la que extraeremos el tamaño y la fecha de creación del archivo *manifest.xml*. Una vez extraídos estos datos los actualizamos en el nodo del árbol correspondiente al archivo de metadatos. El hecho de que exista previamente este archivo, supone también que estamos en la transición del caso online al caso offline (el suceso contrario supondría que estamos cargando los datos desde la base de datos al inicio de la aplicación). Por ello debemos borrar el nodo correspondiente al archivo del e-book en sí porque ya no está disponible. La estrategia vuelve a ser la misma, construir la ruta con *sprintf* y con ella localizar el nodo gracias a la función *get_node_by_path*. Continuando con el ejemplo la ruta será ahora **/CervantesBooks/Quijote.pdf/Quijote.pdf**. La manera de eliminar dicho nodo será con la función del API *g_node_destroy*.

En el caso de que no hubiera un archivo de metadatos anterior simplemente tendremos que generar los nodos del nuevo archivo de metadatos, el archivo del mensaje error y el icono. Cabe reseñar que este es el caso del inicio de la carga de datos de la base de datos del inicio del programa, que es la situación desde la que hemos partido esta sección. Para ello utilizaremos una función auxiliar llamada *replicate_file* que insertará en un nodo padre dado un nuevo nodo con la información de un archivo, especificando su nombre mediante ruta relativa. La información

la obtendrá usando la función *stat* y ésta será lo que necesitamos para rellenar la estructura *member*: tamaño y fecha de creación. Además habrá dos datos que ya conocemos que son el nombre de archivo (lo pasamos por parámetro) y el tipo que si llamamos a esta función siempre será un archivo y no un directorio.

Caso de archivo disponible

En este caso partimos que en la carga de datos desde la base de datos nos encontramos con que el archivo del e-book está ya presente en la caché. Lo que hacemos en este caso a través de la función *fill_available_file_from_db* es rellenar una estructura *vault__File* que es la que devuelve el API de Terabox con el fin de reutilizar la función *fill_available_file* ya que a todos los efectos es equivalente tener el archivo en caché con que esté disponible online. Lo único que cambia es lo que comentábamos anteriormente de mostrar un mensaje advirtiendo que el archivo tardará un poco más en abrirse en el caso de que el archivo no esté en caché y haya que descargarlo de Terabox. Pero dado que solo difiere en esto se puede comprobar de nuevo en la función si el archivo está presente y así ahorrarnos la necesidad de implementar dos funciones distintas.

La implementación de esta función *fill_available_file* comienza creando una estructura *member* en la que rellenaremos el nombre, el tamaño y fecha de creación con la información que nos pasan en el argumento *File* del tipo *vault__File*. Este argumento se habrá rellenado o bien con el contenido de la base de datos al principio de la ejecución en la función *load_node* o bien con el listado de los contenidos de un directorio que nos proporciona el API de Terabox (mediante la función *ddvs_directory_contents*) en la función *fill_dir_tree_recur*. Rellenamos esta función al principio porque es independiente de si el archivo es un e-book o no.

El siguiente paso es comprobar si existe un directorio en la caché correspondiente al archivo y si es así borrar en su interior el archivo de metadatos, el icono y el mensaje de error. De esta manera consideramos el caso de la transición de offline a online. La comprobación la hacemos construyendo el comando con *sprintf* y usando el comando *test -d* cuyas ventajas hemos explicado al comienzo del capítulo.

Al igual que en el caso del archivo no disponible hay que considerar el caso de que el archivo sea un e-book para introducir la información de la interfaz gráfica. Lo primero que hacemos es situarnos en el directorio de la caché correspondiente al directorio en el que está el

libro. Para ello referenciamos al nodo padre del árbol n-ario y obtenemos su path. En el ejemplo que estamos utilizando, éste será **/CervantesBooks**. A continuación creamos el directorio del libro usando la opción *-p* porque éste podría estar creado previamente.

Una vez creado el directorio escribimos el archivo de metadatos, que tendrá algunas diferencias respecto al caso del archivo no disponible. Los tags **Title** y **ItemSize** vuelven a ser iguales al nombre del archivo y su tamaño respectivamente. Sin embargo habrá cambios en los siguientes tags:

Image : Estará vacío ya que al estar disponible el archivo se debe mostrar el icono que muestra el e-book por defecto

startpage : Ahora será directamente el archivo del libro al estar disponible.

Description : En el caso de que esté disponible en la caché este tag estará vacío. Si por el contrario no lo está y hay que bajárselo de Terabox contendrá el mensaje **El archivo tardará unos segundos en descargarse...**

Tras crear el archivo de metadatos hay que crear su nodo correspondiente en el árbol. Reutilizando el código y dado que el archivo ya está creado usaremos la función *replicate_file*. El último nodo que crearemos será el del archivo del libro en sí. Ahora no podremos usar la función auxiliar anterior porque puede que esté físicamente presente o puede que no. Para salvar esto obtendremos los datos de la estructura *File* que hemos recibido como parámetro en esta función y lo escribiremos en la estructura *member* del nodo directamente. El nodo al que nos referimos tendrá la ruta **/CervantesBooks/Quijote.pdf/Quijote.pdf** en el ejemplo. Por último queda especificar que el nodo del directorio del libro (**/CervantesBooks/Quijote.pdf** en el ejemplo) es un directorio en realidad cambiando su tipo en la estructura *member* a *dir*. Es importante acabar volviendo al directorio padre (mediante *chdir* para que el esquema de rutas relativas siga funcionando.

La función *fill_available_file* termina implementando el caso de un archivo genérico que no es un e-book. Entonces en vez de fijar el tipo de la estructura *member* a *dir* lo fijaremos a *file* ya que en este caso es un archivo y no un directorio.

3.11. Fase 7.- Integración de la aplicación con el módem 3G

El objetivo de esta fase es que la aplicación funcione en un entorno móvil. Este entorno se caracteriza por conectarse a Internet a través de un módem 3G. Una de las razones que llevaron a elegir el Irex Iliad como entorno de desarrollo es que había habido gente que había conseguido hacer funcionar este tipo de módems en el dispositivo.

Concretamente nos centramos en el funcionamiento de los módems 3G de marca Huawei. En la división teníamos disponibles los modelos E220 y E156G. Comenzamos haciendo pruebas con el primero, pero el procedimiento que vamos a explicar en esta sección no consiguió establecer la conexión y hubo que probar con el segundo modelo con el que sí fue posible, aunque haciendo algunas modificaciones en el código. El motivo sospecho que fue que en la división se utilizaban estos módems para desarrollo modificando el firmware habitualmente y es posible que estas modificaciones hayan dejado este modelo inservible para su uso en otras aplicaciones como la de este proyecto.

Para empezar hay que explicar que estos módems al igual que otros modelos como los de Alcatel empiezan funcionando como si fueran una unidad de almacenamiento. El objetivo es poner a disposición del usuario los drivers del módem y una vez que estén instalados estos se envía un comando al módem para que empiece a funcionar como tal. Estos drivers son únicamente para entorno Windows.

Para el entorno Linux que es el que usa el Irex Iliad existe soporte de estos módems a partir de la versión del kernel 2.6.20 del kernel. Sin embargo la versión que utiliza el Irex Iliad es la 2.4.19. Habiendo descartado intentar compilar versiones superiores del kernel en las primeras fases de este proyecto, solo queda ver las soluciones que otros han implementado para hacer este módem compatible con la versión de Linux del Iliad. Básicamente lo que es necesario es enviar los comandos para que el módem conmute de dispositivo de almacenamiento a módem 3G.

La solución encontrada [6] está hecha para funcionar con el modelo E220 que por otro lado está bastante más extendido. La modificación consistirá en cambiar el identificador de producto de esa aplicación por el que usa el E156G. Para obtener ese identificador fue suficiente con insertar el modem E156G en el Irex Iliad y obtener este parámetro mediante el comando *lsusb*.

[6] `http://linux.frankenberger.at/dat/huaweiAktBbo.c`

Gracias a ello se pudo ver que había que cambiar el *0x1003* del E220 por por el *0x1406* del E156G. La manera de compirarlo será con la **toolchain** de las aplicaciones de usuario que hemos utilizado previamente. Únicamente hay que tener en cuenta que habrá que linkarlo contra la librería *libusb*. Esta librería está presente en el e-reader así que no es necesario compilarla.

El siguiente paso es adaptar los scripts para realizar la conexión 3G con el proveedor de Internet que utilizaremos. Partimos de unos scripts específicos para Irex Iliad pero hechos para funcionar con Vodafone Rumanía [7] y que obviamente habrá que cambiar para que funcionen con Movistar. Los cambios se determinarán simplemente por inspección visual cambiando las referencias a Vodafone por Movistar y teniendo en cuenta que el resto de los parámetros de la negociación PPP y los comandos AT no variarán.En primer lugar habrá que modificar el fichero con las propiedades de la conexión 3G. En este fichero variarán el usuario y el password utilizados. En el cuadro 3.2 se resumen las modificaciones.

Cuadro 3.2: Cambios en el fichero de propiedades de la conexión 3G (vodafone)

Campo	Dato antiguo	Dato nuevo	Significado
name	vodafone	movistar.es	Identificador local de la conexión
user	internet.vodafone.ro	movistar	Nombre de usuario
password	vodafone	movistar	Contraseña

Una vez modificado hay que copiarlo al e-reader. Elegimos alojarlo en la CompactFlash porque tendrá espacio de sobra. Por tanto toda referencia al alojamiento de los scripts habrá que modificarla para que apunten al directorio elegido de la Compact Flash.

También habrá que modificar el comando AT inicial en el que se especifica el APN para configurarlo con el APN de Movistar (ver apéndice A). Para ello cambiaremos la línea:

'OK' 'AT+CGDCONT=1,"IP","internet.vodafone.ro" '

por la línea:

'OK' 'AT+CGDCONT=1,"IP","movistar.es" '

Una vez modificados los scripts de la conexión, crearemos un programa auxiliar para gestionar cuando tenemos conexión y cuando no. Este programa empieza cargando el programa principal. Dado que habitualmente tendrá que cargar la estructura del árbol de la base de datos, resulta conveniente introducir un retardo de 10 segundos para que le de tiempo a realizar esta

[7] `http://www.mobileread.com/forums/attachment.php?attachmentid=3279&d=1177352538`

carga. Una vez hecho esto el programa se introduce en un bucle infinito que se repetirá cada 10 segundos.

Lo primero que se hace en este bucle es comprobar la conectividad a Internet como ya hemos descrito previamente. Después comprobamos en primer lugar si está conectado el módem 3G/USB. Para ello se escanea los buses USB presentes en el sistema con *usb_find_busses* y a continuación los dispositivos presentes en cada bus con *usb_find_devices*. Una vez hecho esto podemos buscar el módem 3G/USB dentro de las listas enlazadas que representas los buses y los dispositivos usando la función auxiliar *find_device*. Los identificadores de fabricante y producto a los que antes nos referíamos servirán para encontrar al módem y estarán almacenados en macros del preprocesador para tenerlos bien localizados en el caso de un posible cambio de modelo de módem. En el caso de que encontremos presente el módem lo que hacemos es activar la conexión 3G. Lo haremos en un hilo aparte porque fue la única manera de conseguir que el comando *pppd* siguiera funcionando ya que si lo lanzábamos en *background* se quedaba zombie y la conexión no funcionaba. El comando pppd es el gestor de la conexión a Internet con el módem 3G a través del protocolo PPP. Para funcionar necesita que estén presentes unos determinados módulos del kernel. Este hilo auxiliar esos módulos en el orden que se especifica a continuación. El orden es importante puesto que unos necesitan que estén cargados previamente los anteriores:

1. *insmod ./usbserial.o vendor=VENDOR product=PRODUCT* Configura el módem como un puerto serie. Esto hará que la interfaz sea similar a la tradicional de los módems telefónicos que es para los que se diseñó el sistema que usa *pppd*

2. *insmod ./slhc.o* Es el módulo con rutinas para comprimir paquetes TCP (transmitidos por interfaces serie lentas)

3. *insmod ./ppp_generic.o* Es el módulo que gestiona la capa PPP genérica.

4. *insmod ./ppp_async.o* Es el modulo que provee la encapsulación y entramado para enviar y recibir tramas PPP sobre líneas serie asíncronas.

5. *insmod ./ppp_deflate.o* Es la interfaz para la compresión y decompresión *Deflate* usadas por Gzip para la interfaz PPP

6. *insmod ./bsd_comp.o* Es la interfaz para la compresión y decompresión tipo *BSD* para PPP.

7. *insmod ./ppp_synctty.o* Es un driver de canal TTY que puede ser usado con drivers de dispositivo TTY (por ejemplo usbserial) que estén orientados a trama.

Este hilo auxiliar ejecutará a continuación el programa que conmuta el módem para que funcione como tal (y no como un dispositivo de almacenamiento). A este programa ya nos hemos referido al principio de la sección. El siguiente paso es configurar los servidores DNS. Aquí optamos por usar los servidores DNS genéricos que provee Google porque tienen la ventaja respecto a los específicos de la conexión de Movistar, que al funcionar independientemente del proveedor a Internet, van a seguir funcionando en el caso de que más adelante nos conectemos mediante WiFi en vez de la conexión 3G. Aunque sea más lógico configurar los servidores DNS con posterioridad al establecimiento de la conexión con el módem, el hecho de que el comando *pppd* no termine mientras dura la conexión hace más simple configurar los DNS previamente. Finalizamos este hilo auxiliar llamando a este comando mediante *sbin/pppd file vodafone*. **vodafone** es el archivo que contiene la información de la conexión 3G y que ya hemos descrito. El hecho que nombre a una marca de la competencia no importa en un entorno de desarrollo como el que estamos y lo dejamos así por el hecho de cambiar solo lo imprescindible de la configuración que obtuvimos del foro de Mobileread. Obviamente estas referencias deberían eliminarse en un entorno de producción.

Tras la llamada a este hilo auxiliar en el hilo principal esperamos 50 segundos ya que es el tiempo suficiente para que se termine de establecer la conexión 3G. Este tiempo lo hemos determinado introduciendo los comandos del hilo auxiliar manualmente y viendo cuál es el momento en el que la conexión estaba realmente disponible. Una vez pasado ese tiempo cambiamos la ruta IP por defecto a la que acabamos de configurar mediante el comando *route del default; route add default dev ppp0*. Por último pondremos a uno la variable *connection_established* para saber que hemos levantado la conexión PPP.

Si por el contrario no hemos encontrado el módem conectado al e-reader y además teníamos una conexión PPP levantada (de ahí la necesidad de la variable *connection_established*) es el momento de señalizar al programa principal de que no hay conexión enviandole la señal correspondiente con *killall -SIGUSR2 terabook*. Además habrá que finalizar el demonio *pppd* con

killall pppd.

Por último habrá un segundo caso en el que tendremos que volver a enviar la señal de desconexión (*SIGUSR2*) y matar al demonio *pppd* que será si no hemos podido descargarnos la homepage de Telefónica. Si por el contrario lo hemos conseguido entonces sabremos con seguridad que hay conexión a Internet con lo que enviaremos la señal de conexión a Internet (*SIGUSR1*).

Cabe destacar que utilizando compilación condicional tenemos una versión de este programa auxiliar para el ordenador de escritorio que omite toda la operación con el modem 3G.

Para finalizar esta última fase estableceremos el lugar desde donde se lanzará el programa. Inspeccionando los scripts de inicio resulta lógico utilizar el que se encarga de montar los archivos situado en la ruta */home/etc/init.d/mountall.sh*. Desde allí en primer lugar montamos la partición:

/home/root/terabook /mnt/free/newspapers

y posteriormente lanzamos el programa que gestiona la conectividad a Internet.

Capítulo 4

Conclusiones

4.1. Evaluación de los objetivos

Para evaluar el cumplimiento del objetivo principal nos pararemos primero en cada objetivo secundario.

La integración en la interfaz del Irex Iliad es total, dado que utilizamos los distintos campos del menú en el que se listan los libros para notificar los distintos estados en el que pueden estar estos.

Respecto al segundo objetivo secundario, el grado de cumplimiento no es tan alto como el anterior. El tiempo de las transiciones online - offline aunque no es excesivo sí resulta algo lento. Lo mismo ocurre con el tiempo de descarga de los libros, que para un PDF de 4 megas se eleva a 40 segundos. Está dentro de lo aceptable por el usuario según nuestro criterio pero también resulta algo lento.

Dado que los dos objetivos secundarios se cumplen (aunque en distinta medida) podemos concluir que se cumple el objetivo principal de aplicación transparente para el usuario.

1. **Representación de la estructura de archivos de Terabox en un sistema de archivos local**

   Gracias a la implementación del árbol n-ario, conseguimos guardar al 100 por ciento la información de la estructura de Terabox. Ignoramos otros aspectos como la información de compartición, pero esto es algo que no mencionábammos en este objetivo.

   Respecto a la correspondencia entre el sistema de archivos y lo que hay en Terabox, la

implementación tarda un poco desde que inserta el módem USB, hasta que se conecta realmente a Internet. Después la actualización no se hace en cuanto nos conectamos a Internet, sino la siguiente vez que se muestran los contenidos de un determinado directorio. Esto hace que los tiempos de respuesta al pulsar la tecla NEWSPAPERS sean más altos, pero simplifica enormemente la implementación. Aunque a priori se pudiera pensar que podría dar algún problema adicional por el hecho del caso en que se quiera acceder a un archivo saltándose el mostrar el contenido del directorio en la práctica del funcionamiento normal del e-reader esto no ocurre nunca.

2. **Almacenamiento temporal de los archivos en una caché**

   La aplicación funciona perfectamente tanto cuando están los archivos en caché, como cuando no están. Al comprobarse en la implementación justo en el momento de leer de un archivo la existencia de una copia la caché, salvamos los problemas que podría dar el hecho de vaciar la caché.

   Respecto al lugar elegido para la caché, aunque no puede modificarse en tiempo de ejecución, sí que está localizado mediante una macro del preprocesador, lo que hace sencillo que en el futuro este lugar pueda ser modificado en tiempo de compilación.

3. **Acceso a la unidad virtual en modo OFFLINE desde el inicio**

   Este objetivo es quizás uno de los mejor resuelto, ya que al pulsar la tecla NEWS aparece el listado de libros casi sin retardo apreciable. Esto se debe a que en este caso se carga la base de datos al principio de la aplicación encontrándose la estructura de árbol en memoria cuando se muestran los contenidos del directorio.

   Quedaría por resolver cómo notificar al usuario que podrían exisitir libros nuevos u incluso que parte de los que sí aparecen podrían no estar ya disponibles, aunque no me parece algo que sea muy relevante.

   También me parece interesante hacer notar que en las pruebas posteriores a la creación del programa hemos optado en ocasiones con un funcionamiento alternativo en el que que se montaba la unidad virtual cuando se enchufaba el módem USB y nos conectábamos a Internet ejecutando un script (la manera de hacerlo era simplemente pinchando con el estilete sobre un icono). Esto se hizo por similitud en el funcionamiento con otro proyecto

de la división llamado ”pincho infinito” pero personalmente no me gusta este esquema por diferentes razones. En primer lugar, el hecho de tener presente en todo momento el módem USB puede resultar engorrosa dado el espacio adicional que éste ocupa. Además el hecho de tener un que ejecutar un script va en contra del objetivo principal del proyecto de hacer una aplicación ”transparente”.

4. **Apertura de un libro en modo OFFLINE**

   Con el uso de los iconos partidos para representar los libros que no están disponibles también conseguimos cumplir lo que especificábamos en este objetivo. El hecho de utilizar también el campo de la descripción del menú para identificar a los ”no disponibles”, aunque pueda resultar redundante, clarifica este aspecto a los usuarios que no están familiarizados con la interfaz.

   La apertura de un libro que está en caché tiene efectivamente un tiempo de respuesta similar al de un libro normal y no existe entre ellos ninguna diferencia en la práctica. Respecto a la apertura de un libro ”no disponible” quizás el hecho de abrir un PDF con solo una frase mostrando el error no sea la mejor solución pero cumple con el criterio de ser transparente y por ello lo damos por válido.

5. **Paso al modo ONLINE al enchufar el módem USB**

   Este es uno de los objetivos más flojos. Esto es así por el hecho de tardar bastante tiempo desde que enchufamos el módem hasta que realmente nos conectamos a Internet. Dado que estamos hablando de una conexión con un módem 3G pensamos que aún así el usuario estaría dispuesto a tolerar este retardo. Sin embargo, a la vista de la implementación es posible que todavía se pudiera optimizar algo más el tiempo.

   Respecto a la notificación usando el cambio de color del LED del módem 3G, cumple completamente el objetivo de ser ”transparente” ya que no involucra cambio alguno en la pantalla de tinta electrónica del Irex Iliad.

6. **Listado y apertura de archivos en modo ONLINE**

   De nuevo y como ocurre con el objetivo anterior los tiempos vuelven a ser los principales ”peros” que nos encontramos al evaluar este objetivo. En primer lugar y al contrario de lo que ocurre cuando se inicia la aplicación, al realizar la transición al modo ONLINE el

árbol n-ario no se regenera hasta que no se lista alguno de los directorios. Esto introduce el retardo que se experimenta al pulsar la tecla NEWS. Pese a todo, este retardo vuelve a ser algo aceptable.

Respecto al listado de los libros en el menú cabe reseñar que esta vez el uso del campo descripción, al contrario que en el modo OFFLINE, no es redundante, y sí que aporta información al distinguir los libros que están en la caché de los que se tienen que descargar. Dado que esta diferencia no es algo funcional sino simplemente consiste en tener que esperar algo más no parecía adecuado el cambiar el icono como hacíamos en caso OFFLINE.

El segundo escollo que nos encontramos en este objetivo es el elevado tiempo de descarga que nos encontramos al abrir un libro que necesita ser descargado de Terabox. Para un archivo PDF DE 4 megabytes, la carga asciende a 40 segundos. Pese a ser alto, este tiempo lo damos por aceptable por dos razones. En primer lugar porque el usuario está avisado por el mensaje en el menú que la carga del archivo va a llevar tiempo. En segundo lugar, porque el hecho de tener que esperar por descargarse un archivo de Internet es algo al que usuario estará habituado y por tanto le podrá parecer admisible.

7. **Paso al modo OFFLINE al desenchufar el módem USB**

   La aplicación pasa al modo OFFLINE en cualquier situación que suponga una desconexión de Internet. En el caso de fallos de la conexión, la aplicación intentará volver a conectarse periódicamente manteniéndose en el modo OFFLINE hasta que se establezca de nuevo la conexión.

   Respecto a la transición al modo OFFLINE, se introduce un retardo al pulsar la tecla NEWS que es superior al que se produce en las situaciones referidas en todos los objetivos anteriores. Aunque, a juicio siempre de los integrantes de la división, que fuimos los que hicimos las pruebas, no nos pareció demasiado tiempo. El motivo es la construcción de los metadatos y la copia de iconos que implica la transición al modo OFFLINE.

Para resumir, salvando los tiempos de respuesta que pese a ser ”tolerables” son bastante altos en las situaciones antes mencionadas, por lo demás la aplicación cumple perfectamente con el objetivo principal de este proyecto fin de carrera.

4.2. Lecciones aprendidas

Este proyecto me ha servido para consolidar mis conocimientos de programación en C que se concretan en aspectos como manejo de punteros, estructuras y del preprocesador. También he aumentado mis conocimientos de programación en Linux con técnicas como la de la compilación cruzada.

También he aprendido lecciones de los errores cometidos, como intentar adaptar por mis medios un kernel superior para el e-reader. En lo sucesivo seré más cauto al plantearme soluciones ad-hoc. Además he aprendido como en entornos embebidos pueden aparecer cuellos de botella en situaciones en las fuera de estos entornos no aparecen.

En la metodología de trabajo al ser éste el primer proyecto de envergadura al que me he enfrentado he experimentado lo conveniente que es una planificación por fases con objetivos concretos para proyectos de este tipo.

Por último el hecho de enfrentarme yo solo a un proyecto grande desde cero ha fortalecido mis capacidades de automotivación y gestión del trabajo. Sin embargo, también quiero señalar que los pocos contactos que tuve con mi compañeros de división fueron muy productivos ya que aparte de brindarme su valiosa experiencia en ocasiones tuve que defender ante ellos mis propias soluciones y esto fue muy enriquecedor.

4.3. Trabajos futuros

En primer lugar sería interesante comentar el código y gestionar correctamente las asignaciones de memoria, ya que hay casos en que no liberamos correctamente ésta y esto podría causar problemas.

El paso lógico siguiente es implementar la escritura en el sistema de archivos. Esto entraña una gran complejidad pero aportaría funcionalidad para tareas posibles en el Iliad como escribir notas en los e-books.

Por último habría que portar el programa para que funcionara en el modelo de e-reader que comercializa Movistar.

Apéndice A

Apéndice 1: Sintaxis del comando AT +CGDCONT

Extraído de la referencia bibliográfica número [2]

Cuadro A.1: Sintaxis de los parámetros del comando +CGDCONT

Comando	Respuesta posible
+CGDCONT=[cid [,PDP_type [,APN [,PDP_addr [,d_comp [,h_comp [,pd1 [,. . . [,pdN]]] [,IPv4AddrAlloc]]]]]]]	OK / ERROR

Description

The set command specifies PDP context parameter values for a PDP context identified by the (local) context identification parameter, *cid*. The number of PDP contexts that may be in a defined state at the same time is given by the range returned by the test command.

For EPS the PDN connection and its associated EPS default bearer is identified herewith. For EPS the *PDP_addr* shall be omitted.

A special form of the set command, *+CGDCONT= cid* causes the values for context number *cid* to become undefined.

The read command returns the current settings for each defined context.

The test command returns values supported as a compound value. If the MT supports several PDP types, *PDP_type*, the parameter value ranges for each *PDP_type* are returned on a separate line.

Defined values

cid: a numeric parameter which specifies a particular PDP context definition. The parameter is local to the TE-MT interface and is used in other PDP context-related commands. The range of permitted values (minimum value = 1) is returned by the test form of the command.

PDP_type: a string parameter which specifies the type of packet data protocol

X.25	ITU-T/CCITT X.25 layer 3 (Obsolete)
IP	Internet Protocol (IETF STD 5)
IPV6	Internet Protocol, version 6 (IETF RFC 2460)
IPV4V6	Virtual *PDP_type* introduced to handle dual IP stack UE capability.
OSPIH	Internet Hosted Octect Stream Protocol (Obsolete)
PPP	Point to Point Protocol (IETF STD 51)

NOTE: Only IP, IPV6 and IPV4V6 values are supported for EPS services.

APN: a string parameter which is a logical name that is used to select the GGSN or the external packet data network. If the value is null or omitted, then the subscription value will be requested.

Bibliografía

[1] Barrie Sosinksy. *Cloud Computing Bible*. John Wiley & Sons. 2011

[2] Varios autores. *Especificación técnica 3GPP-TS 27.007 V9.2.0 (2009-12)*. 3rd Generation Partership Project. 2009

[3] Wolfgang Maurer *Professional Linux Kernel Architecture*. Wrox. 2008.