TREBALL DE FI DE CARRERA

TÍTOL: Sistemes Linux Empotrats 2

AUTOR: Víctor González Pacheco

DIRECTOR: Cristina Barrado Muxí

DATA: 10 de julio de 2006

Títol: Sistemes Linux Empotrats 2

Autor: Víctor González Pacheco

Director: Cristina Barrado Muxí

Data: 10 de julio de 2006

Resumen

En este proyecto se desa demostrar la capacidad que tienen los sistemas Linux para añadir inteligencia a sistemas empotrados. Para demostrar esto se diseñará una aplicación que permita añadir funcionalidades a un sistema empotrado que antes no tenía.

Como sistema empotrado se utilizará un *router.* En este dispositivo se le instalará una distribución de Linux en la cual se hará correr la aplicación desarrollada. La aplicación será un monitorizador de red que permita escuchar el tráfico que circula por la red durante un examen de programacón.

Para ello será necesario introducir primero el sistema operativo Linux. Se explicarán sus principales características así como una breve introducción a su funcionamiento interno. Tras esto se realizará un descripción de los sistemas empotrados: sus clasificaciones y arquitecturas principales. También se realizará una descripción de los distintos configuraciones de desarrollo que existen en estos dispositivos.

Una vez conocidos el sistema operativo Linux y los sistemas empotrados, se realizará una introducción a las redes, ya que la aplicación hará un uso extensivo de ellas.

Finalmente se describirá la aplicación desarrollada y se analizarán los resultados obtenidos con ella.

Title: Embedded Linux Systems 2

Author: Víctor González Pacheco

Director: Cristina Barrado Muxí

Date: July, 10th 2006

Overview

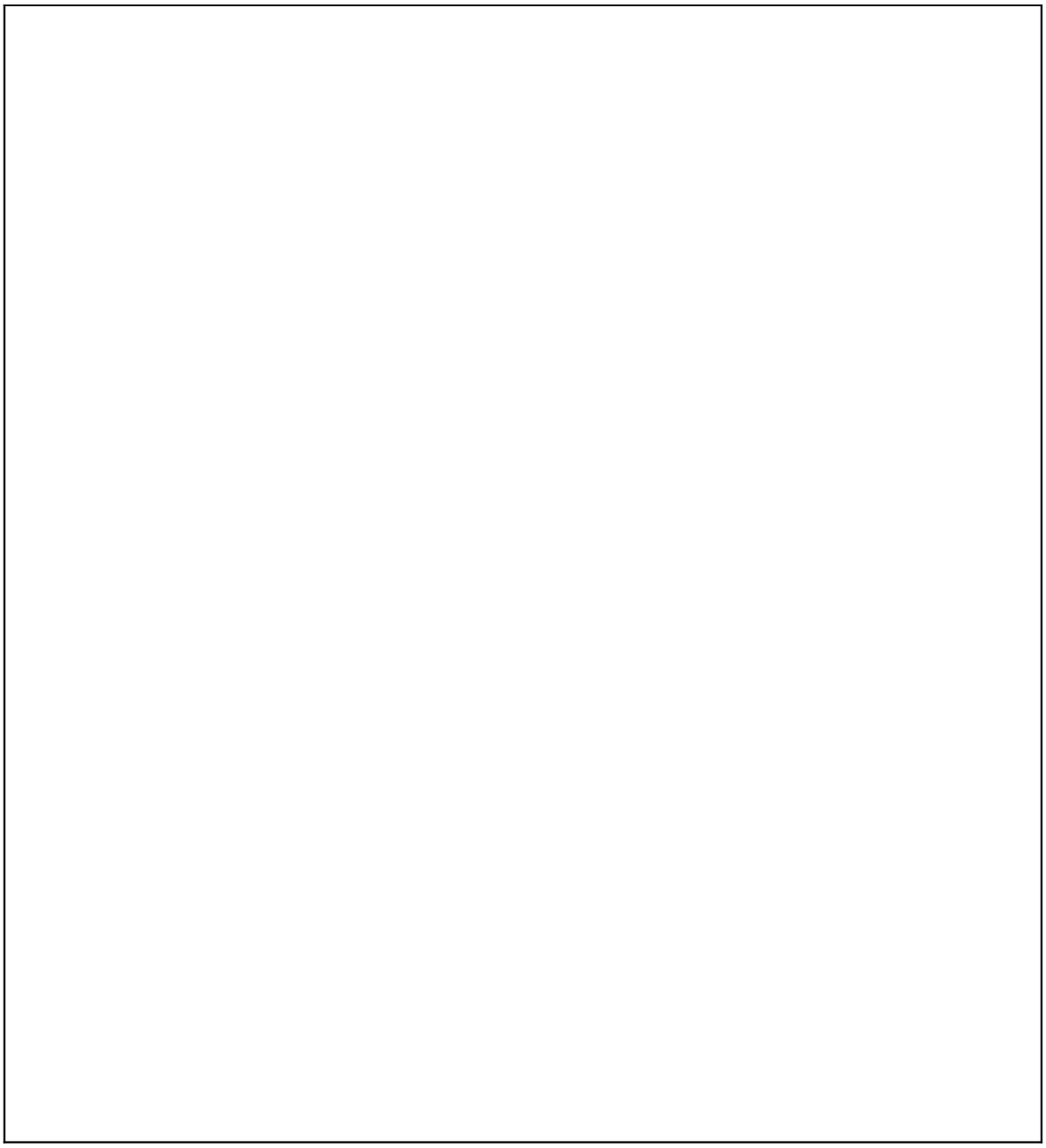
The main goal of this project is to demonstrate de capacity of Linux to give intelligence to the embedded systems. It will be designed an application wich will add extra functionalities to an embedded system.

The embedded system will be a router. In this device will be installed a Linux distribution on will run the developed application. That application will be a network sniffer wich listens on the net while a programming exam is done.

It will be necessary to introduce first de Linux operative system. It will be explained this main characteristics and a short introducction of its internals. After this, it will be relised a description of the embedded systems: their classifications and their main architectures. Also it will be done an enumeration of the different development configurations of this devices.

Once the Linux operative system and the embedded systems are known, it will be introduced to the networking.

Finally, the developed application it will be described and the results obtained with that application will be discussed.



Índice de contenido

CAPÍTULO 1. INTRODUCCIÓN....7

1.1.- Motivación del proyecto....7

1.2.- Objetivos del proyecto....8

1.3.- Planificación....8
1.3.1.- Fase 1: Búsqueda de información....8
1.3.2.- Fase 2: Instalación de una distribución Gentoo Linux....9
1.3.3.- Fase 3: Búsqueda y evaluación de un entorno de trabajo....10
1.3.4.- Fase 4: Pruebas del entorno de trabajo....10
1.3.5.- Fase 5: Redacción de la memoria del proyecto....11
1.3.6.- Fase 6: Elaboración de la presentación del proyecto....11
1.3.7.- Costes del proyecto....12

1.4.- Estructura de la memoria....12

CAPÍTULO 2. SISTEMAS OPERATIVOS LINUX....14

2.1.- Características Principales del Sistema....15

2.2.- Procesos y tareas....18

CAPÍTULO 3. LINUX EN SISTEMAS EMPOTRADOS....21

3.1.- Clasificación Sistemas Empotrados....21
3.1.1.- Tamaño....21
3.1.2.- Limitaciones temporales (Tiempo Real)....22
3.1.3.- Capacidades de comunicaciones....23
3.1.4.- Interacción con el usuario....23

3.2.- Arquitecturas de sistemas empotrados....24
3.2.1.- ARM 24
3.2.2.- PowerPC....24
3.2.3.- MIPS....25

3.3.- Sistemas de almacenamiento en sistemas empotrados....25

3.4.- Desarrollo en sistemas empotrados....26
3.4.1.- Configuración enlazada (linked setup):....27
3.4.2.- Configuración con almacenamiento extraíble (removable storage setup) 28
3.4.3.- Configuración autónoma (standalone setup)....28

CAPÍTULO 4. INTRODUCCIÓN A LAS REDES....30

4.1.- El modelo de referencia OSI....30

4.2.- El modelo de referencia TCP/IP....33

4.3.- El Modelo Híbrido....36

4.4.- Ejemplo de comunicación entre dos dispositivos situados en diferentes redes....38

4.5.- Redes en Linux....40

4.5.1.- Ejemplo de comunicación desde el punto de vista de linux.....41

CAPÍTULO 5. LAS LIBRERÍAS LIBPCAP.....46

5.1.- Funcionamiento de un monitorizador de red.....46

5.2.- Inicialización del programa.....48

5.3.- Filtros de captura.....50

5.4.- Bucle de captura.....54

5.5.- Extracción de datos.....56

5.6.- Procesado de datos.....58

CAPÍTULO 6. APLICACIÓN DESARROLLADA.....59

6.1.- El router Linksys WRT54GL.....59

6.2.- La aplicación.....62

CAPÍTULO 7. CONCLUSIONES.....65

7.1.- Valoraciones de Linux, empotrados y proyecto.....65

7.2.- Objetivos cumplidos.....66

7.3.- Seguimiento planificación.....67

7.4.- Mejoras y líneas futuras.....67

BIBLIOGRAFÍA.....69

REPERCUSIONES MEDIOAMBIENTALES.....70

ANEXO.....71

7.5.- Principales Estructuras.....71
Informacion de Interfaces.....71
Estadísticas.....72
Paquetes.....72
La estructura pcap_t.....73

Capítulo 1.

Hoy en día existen multitud de dispositivos electrónicos que realizan funciones muy específicas. Estos sistemas se les llama sistemas empotrados. Debido a los grandes avances en tecnología, estos dispositivos, han reducido su precio de manera que están al alcance de muchos consumidores. Actualmente, no es raro ver a gente con teléfonos móviles, agendas electrónicas, reproductores de audio digital, etc.

A su vez, el aumento de las prestaciones de estos dispositivos ha permitido que sean capaces de ejecutar tareas que requieren grandes cantidades de cálculo. Esto ha permitido que los controladores de estos dispositivos sean versiones en miniatura de los sistemas operativos de los ordenadores de sobremesa.

El sistema operativo Linux, gracias a su licencia GPL[1], es un sistema con una gran comunidad de desarrolladores que le están permitiendo ser portado a un gran número de plataformas. Algunas de estas plataformas están siendo sistemas empotrados.

Estos sistemas empotrados, al utilizar Linux, se benefician de muchas de sus ventajas. Las comunidades de desarrolladores consiguen añadir funcionalidades a estos productos que el fabricante no había imaginado o que no había podido desarrollar.

1.1.- Motivación del proyecto

Antes de comenzar este proyecto, tuve la oportunidad de descubrir el sistema operativo Linux y de introducirme en el ámbito de los sistemas empotrados. Esto ocurrió durante la realización de mis prácticas en empresa. Los pocos meses que duraron, sirvieron para despertar mi curiosidad sobre Linux. Cuestiones como su funcionamiento interno llamaron mi atención. Por otra parte, me surgieron cuestiones referentes a Linux en sistemas empotrados. ¿Que hay que hacer para “empotrar” Linux? ¿Que ventajas proporciona Linux con respecto a otros sistemas operativos?

Realizar este TFC me brindaba la posibilidad de resolver esas preguntas. Esa es la motivación de este proyecto: Profundizar mis conocimientos en sistemas empotrados y especialmente en Linux.

[1]General Public License. http://en.wikipedia.org/wiki/Gpl

1.2.- Objetivos del proyecto

Linux es un sistema operativo licenciado bajo la licencia GPL. La licencia GPL permite que el código del sistema operativo se encuentre al alcance de todo el que quiera. Esto ha permitido que cada usuario tenga la posibilidad de modificar el sistema adecuándolo a sus necesidades específicas. La posibilidad de instalar Linux en un sistema empotrado, permite que éste se beneficie de muchas de las ventajas del sistema operativo.

El objetivo del proyecto es realizar una aplicación con la que se pueda demostrar que Linux permite añadir funcionalidades extra a un sistema empotrado. Esta funcionalidad extra estará adaptada a nuestras necesidades.

1.3.- Planificación

En este capítulo se analizará la planificación realizada en las primeras fases del proyecto. El proyecto comenzó a finales de febrero y termina la primera semana de julio. Según la planificación el proyecto tendría una duración de unas 450 horas. El trabajo implica, en media, un compromiso por parte del alumno de 4 horas diarias.

La planificación divide el trabajo en 6 grandes apartados: búsqueda de información,instalación de una distribución Gentoo Linux[2], búsqueda y evaluación de un entorno de trabajo, pruebas con el entorno de trabajo elegido, redacción de la memoria del proyecto y preparación de la presentación del proyecto.

La finalización de cada uno de estos apartados significar la consecución de un hito del proyecto. Así mismo, para facilitar la consecución de los hitos, éstos se dividirán en hitos menores. De esta manera, para conseguir un hito mayor es necesario conseguir todos sus hitos menores. Igualmente, para el control y la ayuda al alumno, se acordó la celebración de reuniones quincenales.

A continuación se detallan las 6 diferentes fases de las que se compone el proyecto. Los diagramas de Gantt de cada una de las distintas fases se encuentran en el anexo. En el capítulo de conclusiones se presentan las principales desviaciones de la planificación.

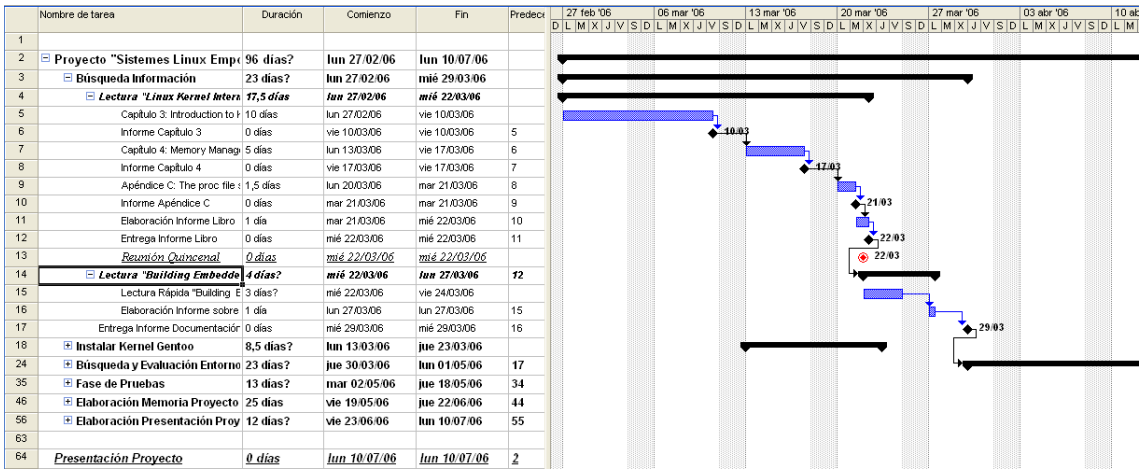
[2]http://www.gentoo.org

1.3.1.- Fase 1: Búsqueda de información

Según la planificación, esta fase duraría unas 86 horas realizadas entre el lunes 27 de febrero y el miércoles 29 de marzo.

Esta fase se divide en dos hitos principales: Lectura del libro "*Linux Kernel Internals*" y Lectura del libro "*Building Em bedded Linux System s*". A su vez cada una de estos hitos se divide en hitos menores. Cada uno de estos hitos menores o subtareas sería la finalización de un capítulo de los libros.

La distribución de trabajo estaba ideada para celebrar una reunión quincenal al final del primer libro. En esta reunión el alumno expondría los conocimientos adquiridos sobre el libro.



	Nombre de tarea	Duración	Comienzo	Fin	Predeci
1					
2	Proyecto "Sistemes Linux Empo	96 días?	lun 27/02/06	lun 10/07/06	
3	Búsqueda Información	23 días?	lun 27/02/06	mié 29/03/06	
4	*Lectura "Linux Kernel Intern*	*17,5 días*	*lun 27/02/06*	*mié 22/03/06*	
5	Capitulo 3: Introduction to I	10 días	lun 27/02/06	vie 10/03/06	
6	Informe Capitulo 3	0 días	vie 10/03/06	vie 10/03/06	5
7	Capitulo 4: Memory Manag	5 días	lun 13/03/06	vie 17/03/06	6
8	Informe Capitulo 4	0 días	vie 17/03/06	vie 17/03/06	7
9	Apéndice C: The proc file s	1,5 días	lun 20/03/06	mar 21/03/06	8
10	Informe Apéndice C	0 días	mar 21/03/06	mar 21/03/06	9
11	Elaboración Informe Libro	1 día	mar 21/03/06	mié 22/03/06	10
12	Entrega Informe Libro	0 días	mié 22/03/06	mié 22/03/06	11
13	*Reunión Quincenal*	*0 días*	*mié 22/03/06*	*mié 22/03/06*	
14	*Lectura "Building Embedde*	*4 días?*	*mié 22/03/06*	*lun 27/03/06*	*12*
15	Lectura Rápida "Building E	3 días?	mié 22/03/06	vie 24/03/06	
16	Elaboración Informe sobre	1 día	lun 27/03/06	lun 27/03/06	15
17	Entrega Informe Documentación	0 días	mié 29/03/06	mié 29/03/06	16
18	Instalar Kernel Gentoo	8,5 días?	lun 13/03/06	jue 23/03/06	
24	Búsqueda y Evaluación Entorno	23 días?	jue 30/03/06	lun 01/05/06	17
35	Fase de Pruebas	13 días?	mar 02/05/06	jue 18/05/06	34
46	Elaboración Memoria Proyecto	25 días	vie 19/05/06	jue 22/06/06	44
56	Elaboración Presentación Proy	12 días?	vie 23/06/06	lun 10/07/06	55
63					
64	*Presentación Proyecto*	*0 días*	*lun 10/07/06*	*lun 10/07/06*	*2*

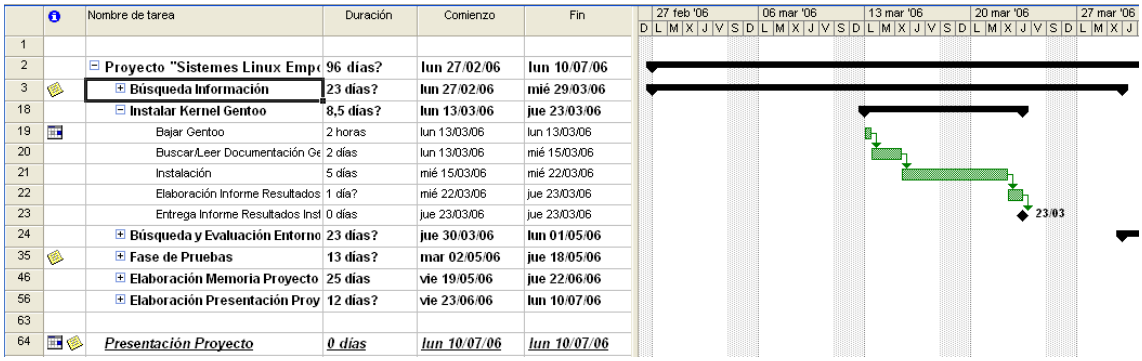
Figura 1.1: Diagrarm a de Gantt de la Fase 1.

1.3.2.- Fase 2: Instalación de una distribución Gentoo Linux

Esta fase tiene una duración de unas 32 horas realizadas entre los días 13 y 23 de marzo. De esta manera esta fase se realiza conjuntamente con la primera fase.

El hito principal de la fase es la instalación de una distribución Linux. La motivación de esta fase es que el alumno aprenda el proceso de instalación para conocer mejor el funcionamiento interno de Linux. La distribución elegida fue Gentoo Linux ya que es la que mejor permite este objetivo.

La fase se divide en diversos hitos que comprenden la lectura de documentación sobre la instalación, el proceso de instalación, etc.



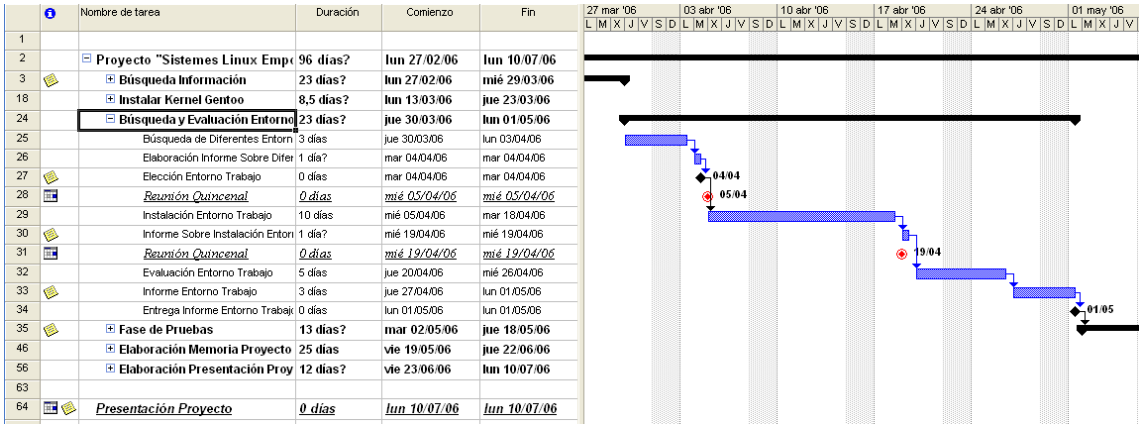
	Nombre de tarea	Duración	Comienzo	Fin
1				
2	Proyecto "Sistemes Linux Empo	96 días?	lun 27/02/06	lun 10/07/06
3	Búsqueda Información	23 días?	lun 27/02/06	mié 29/03/06
18	Instalar Kernel Gentoo	8,5 días?	lun 13/03/06	jue 23/03/06
19	Bajar Gentoo	2 horas	lun 13/03/06	lun 13/03/06
20	Buscar/Leer Documentación Ge	2 días	lun 13/03/06	mié 15/03/06
21	Instalación	5 días	mié 15/03/06	mié 22/03/06
22	Elaboración Informe Resultados	1 día?	mié 22/03/06	jue 23/03/06
23	Entrega Informe Resultados Inst	0 días	jue 23/03/06	jue 23/03/06
24	Búsqueda y Evaluación Entorno	23 días?	jue 30/03/06	lun 01/05/06
35	Fase de Pruebas	13 días?	mar 02/05/06	jue 18/05/06
46	Elaboración Memoria Proyecto	25 días	vie 19/05/06	jue 22/06/06
56	Elaboración Presentación Proy	12 días?	vie 23/06/06	lun 10/07/06
63				
64	Presentación Proyecto	0 días	lun 10/07/06	lun 10/07/06

Figura 1.2: Diagram a de Gant de la Fase 2.

1.3.3.- Fase 3: Búsqueda y evaluación de un entorno de trabajo

Esta fase tiene una duración prevista de unas 86 horas comprendidas entre el 30 de marzo y el 1 de mayo.

Durante esta fase se debe escoger un entorno de trabajo en el que trabajar. Durante la realización de la planificación aún no se sabía si sería un simulador o si se compraría algún sistema empotrado. En esta fase se celebrarían dos reuniones, para revisar lo realizado en otras fases (final de la fase 1 y toda la fase 2) y sobre los hitos conseguidos en esta fase.

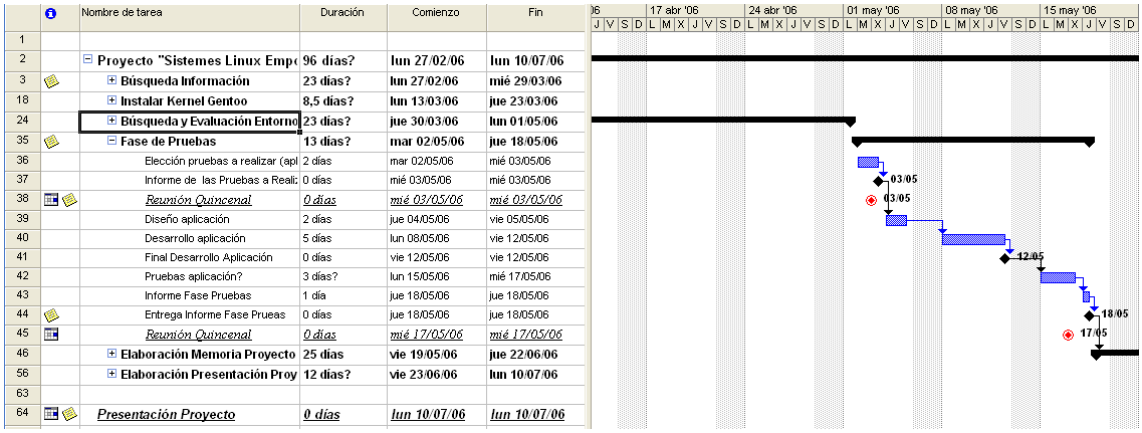


	Nombre de tarea	Duración	Comienzo	Fin
1				
2	Proyecto "Sistemes Linux Empo	96 días?	lun 27/02/06	lun 10/07/06
3	Búsqueda Información	23 días?	lun 27/02/06	mié 29/03/06
18	Instalar Kernel Gentoo	8,5 días?	lun 13/03/06	jue 23/03/06
24	Búsqueda y Evaluación Entorno	23 días?	jue 30/03/06	lun 01/05/06
25	Búsqueda de Diferentes Entorn	3 días	jue 30/03/06	lun 03/04/06
26	Elaboración Informe Sobre Difer	1 día?	mar 04/04/06	mar 04/04/06
27	Elección Entorno Trabajo	0 días	mar 04/04/06	mar 04/04/06
28	Reunión Quincenal	0 días	mié 05/04/06	mié 05/04/06
29	Instalación Entorno Trabajo	10 días	mié 05/04/06	mar 18/04/06
30	Informe Sobre Instalación Entorı	1 día?	mié 19/04/06	mié 19/04/06
31	Reunión Quincenal	0 días	mié 19/04/06	mié 19/04/06
32	Evaluación Entorno Trabajo	5 días	jue 20/04/06	mié 26/04/06
33	Informe Entorno Trabajo	3 días	jue 27/04/06	lun 01/05/06
34	Entrega Informe Entorno Trabajo	0 días	lun 01/05/06	lun 01/05/06
35	Fase de Pruebas	13 días?	mar 02/05/06	jue 18/05/06
46	Elaboración Memoria Proyecto	25 días	vie 19/05/06	jue 22/06/06
56	Elaboración Presentación Proy	12 días?	vie 23/06/06	lun 10/07/06
63				
64	Presentación Proyecto	0 días	lun 10/07/06	lun 10/07/06

Figura 1.3: Diagram a de Gantt de la Fase 3.

1.3.4.- Fase 4: Pruebas del entorno de trabajo

Esta fase consta de 72 horas comprendidas entre los días 2 y 18 de mayo. Los hitos principales de esta fase son la elección de una aplicación a desarrollar y el desarrollo de la aplicación elegida. Así mismo se preveen dos reuniones quincenales en esta fase. La primera para elegir la aplicación y la segunda para mostrar la aplicación desarrollada.

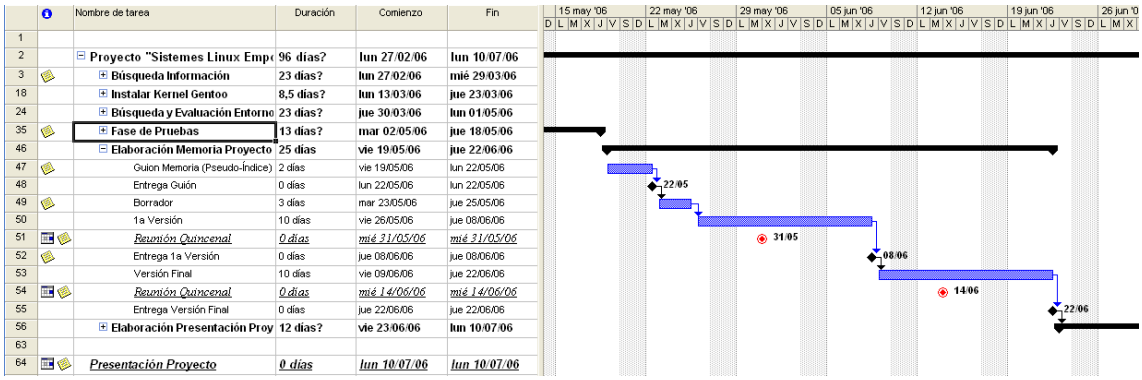


	Nombre de tarea	Duración	Comienzo	Fin
1				
2	Proyecto "Sistemes Linux Emp	96 días?	lun 27/02/06	lun 10/07/06
3	Búsqueda Información	23 días?	lun 27/02/06	mié 29/03/06
18	Instalar Kernel Gentoo	8,5 días?	lun 13/03/06	jue 23/03/06
24	Búsqueda y Evaluación Entorno	23 días?	jue 30/03/06	lun 01/05/06
35	Fase de Pruebas	13 días?	mar 02/05/06	jue 18/05/06
36	Elección pruebas a realizar (apl	2 días	mar 02/05/06	mié 03/05/06
37	Informe de las Pruebas a Reali	0 días	mié 03/05/06	mié 03/05/06
38	*Reunión Quincenal*	*0 días*	*mié 03/05/06*	*mié 03/05/06*
39	Diseño aplicación	2 días	jue 04/05/06	vie 05/05/06
40	Desarrollo aplicación	5 días	lun 08/05/06	vie 12/05/06
41	Final Desarrollo Aplicación	0 días	vie 12/05/06	vie 12/05/06
42	Pruebas aplicación?	3 días?	lun 15/05/06	mié 17/05/06
43	Informe Fase Pruebas	1 día	jue 18/05/06	jue 18/05/06
44	Entrega Informe Fase Prueas	0 días	jue 18/05/06	jue 18/05/06
45	*Reunión Quincenal*	*0 días*	*mié 17/05/06*	*mié 17/05/06*
46	Elaboración Memoria Proyecto	25 días	vie 19/05/06	jue 22/06/06
56	Elaboración Presentación Proy	12 días?	vie 23/06/06	lun 10/07/06
63				
64	*Presentación Proyecto*	*0 días*	*lun 10/07/06*	*lun 10/07/06*

Figura 1.4: Diagram a de Gantt de la Fase 4.

1.3.5.- Fase 5: Redacción de la memoria del proyecto

Esta fase dura unas 100 horas comprendidas entre los días 19 de mayo y 22 de junio. Durante esta fase se redacta la memoria del proyecto. Los distintos hitos y reuniones de esta fase sirven para el control del estado de la memoria.



	Nombre de tarea	Duración	Comienzo	Fin
1				
2	Proyecto "Sistemes Linux Emp	96 días?	lun 27/02/06	lun 10/07/06
3	Búsqueda Información	23 días?	lun 27/02/06	mié 29/03/06
18	Instalar Kernel Gentoo	8,5 días?	lun 13/03/06	jue 23/03/06
24	Búsqueda y Evaluación Entorno	23 días?	jue 30/03/06	lun 01/05/06
35	Fase de Pruebas	13 días?	mar 02/05/06	jue 18/05/06
46	Elaboración Memoria Proyecto	25 días	vie 19/05/06	jue 22/06/06
47	Guion Memoria (Pseudo-Índice)	2 días	vie 19/05/06	lun 22/05/06
48	Entrega Guión	0 días	lun 22/05/06	lun 22/05/06
49	Borrador	3 días	mar 23/05/06	jue 25/05/06
50	1a Versión	10 días	vie 26/05/06	jue 08/06/06
51	*Reunión Quincenal*	*0 días*	*mié 31/05/06*	*mié 31/05/06*
52	Entrega 1a Versión	0 días	jue 08/06/06	jue 08/06/06
53	Versión Final	10 días	vie 09/06/06	jue 22/06/06
54	*Reunión Quincenal*	*0 días*	*mié 14/06/06*	*mié 14/06/06*
55	Entrega Versión Final	0 días	jue 22/06/06	jue 22/06/06
56	Elaboración Presentación Proy	12 días?	vie 23/06/06	lun 10/07/06
63				
64	*Presentación Proyecto*	*0 días*	*lun 10/07/06*	*lun 10/07/06*

Figura 1.5: Diagram a de Gant de la Fase 5.

1.3.6.- Fase 6: Elaboración de la presentación del proyecto

Esta última fase dura unas 60 horas comprendidas entre el 23 de junio y el 10 de julio. Durante esta fase el alumno debe preparar la presentación del proyecto. La reunión y el hito de la fase son para el control del estado de la presentación. El hito final de esta fase y del proyecto entero será la presentación del proyecto ante un tribunal.

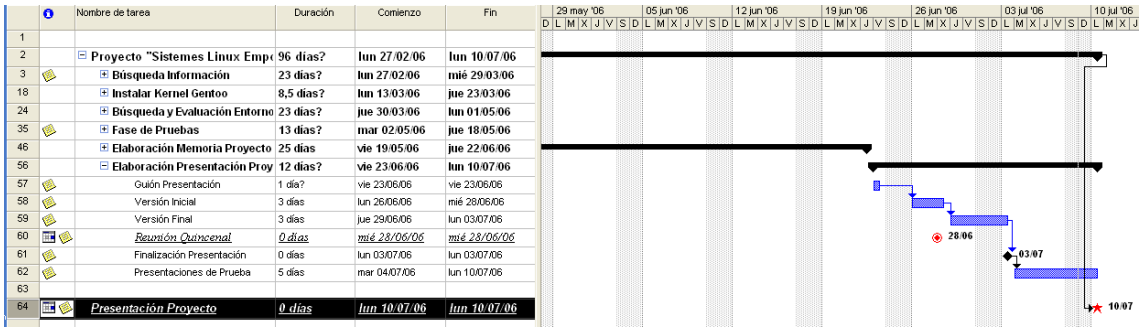


Figura 1.6: Diagram a de Gant de la Fase 6.

1.3.7.- Costes del proyecto

En este apartado se detallan los costes totales del proyecto asumiendo unos costes por hora a precio de mercado.

Concepto	Número	Coste	Horas	SubTotal
Router Linksys WRT54GL	1	70 €	NA	70 €
Portátil Sony Vaio S5M	1	1.500 €	NA	1.500 €
Ingeniero Téc. Telecomuni.	1	20€/hora	420	8.400 €
TOTAL				**9.970 €**

Tabla 1: Costes del TFC.

1.4.- Estructura de la memoria

Tras este capítulo introductorio,en el segundo capítulo se introducirá al sistema operativo Linux. Se explicarán sus principales características y se realizará una breve introducción al concepto de procesos y tareas.

En el tercer capítulo se presentará una introducción a los sistemas empotrados. Se definirá el concepto de sistema empotrado así como sus características

principales. Se introducirán las tres principales arquitecturas que se pueden encontrar en estos sistemas y se explicarán las diferentes opciones que hay para el desarrollo de aplicaciones en sistemas empotrados.

En el cuarto capítulo se realizará una introducción a las comunicaciones entre redes. Debido a que para el desarrollo de la aplicación se necesitan conceptos de redes, en este capítulo se analizarán los distintos modelos de comunicaciones que hay. Así mismo se estudiará un ejemplo de comunicación entre dos redes antes de explicar como maneja Linux los protocolos de red.

En el quinto capítulo introduciremos la herramienta principal en la que se basará la aplicación desarrollada. Esta herramienta será la librería *Libpcap*[3]*.* Estas librerías son ampliamente utilizadas por otros monitorizadores como *tcpdum p* y detectores de intrusos como *snort*.

En el sexto capítulo se describirá el *router* elegido donde se utilizará la aplicación desarrollada. También se explicará que distribución se instalará en el *router.* Mencionaremos sus principales características y se detallará el proceso de instalación en el *router*. Tras esto se describirá la aplicación desarrollada, se explicarán las alternativas estudiadas y el proceso de desarrollo que se siguió.

En el séptimo y último capítulo, finalmente, se presentarán las conclusiones una vez terminado el proyecto. Se evaluarán los resultados obtenidos tomando los objetivos iniciales como baremo y describiremos las líneas futuras que pueden seguir a este proyecto. También se realizará una evaluación de Linux y de los sistemas empotrados. Finalmente, se expondrá una valoración personal del proyecto.

[3] http://www.tcpdump.org/
http://en.wikipedia.org/wiki/Libpcap

Capítulo 2. Sistemas operativos Linux

Linux es un sistema operativo de propósito general basado en *Unix* cuya característica principal es que su código es libre. Originalmente desarrollado para PCs actualmente ha sido portado a numerosas arquitecturas como PowerPC, ARM, MIPS, etc.

El sistema operativo fue originalmente desarrollado en 1991 por el estudiante de informática finlandés Linus Torvalds. Linus publicó su sistema operativo bajo la licencia GPL (GNU Public License). Esta licencia permite que cualquiera pueda usar, copiar y modificar el sistema sin tener que pagar ningún cargo económico por ello. Esto permitió el código de Linux se publicara en internet, llegando al abasto de un gran número de internautas que comenzaron a contribuir a su desarrollo.

Hablar de Linux es sólo referirse al *Kernel,* el núcleo del sistema. El núcleo sólo es una interfaz que permite comunicar el *hardware* con los programas . Por lo que el *Kernel* sólo no forma el sistema operativo.

El proyecto GNU (GNU es el acrónimo de GNU is Not Unix) fue fundado, junto la *Free Software Foundation (FSF)* por Richard Stallman en 1983. El objetivo del proyecto era crear un sistema operativo basado en Unix totalmente libre. Para ello, la *FSF* publicó la licencia GPL y creó todos los componentes del sistema: librerías, compiladores, editores de texto, un shell (intérprete de comandos) basado en linux, etc. Sin embargo, a principios de los 90 aún no habían conseguido implementar un núcleo que soportara el sistema. Cuando apareció Linux se unieron todos los programas GNU, por lo que el sistema operativo creado se llamó GNU/Linux, si bien es más conocido únicamente como Linux.

Debido a la libertad que proporciona la GPL, se han creado numerosas distribuciones de GNU/Linux. Una distribución es un conjunto formado por programas GNU, programas de terceros y el *Kernel*. Entre las distribuciones más famosas se encuentran Debian, Gentoo, Red Hat, SuSe y Ubuntu.

Actualmente Linus Torvalds y un numeroso grupo de desarrolladores desarrollan el *Kernel* independientemente. El proyecto GNU sigue creando aplicaciones y adaptándolas al *Kernel* mientras que cada distribución se encarga de recopilar una serie de estas aplicaciones para formar lo que es la distribución en sí.

2.1.- Características Principales del Sistema

La gran cantidad de programas disponibles para Linux hacen que sus características y funcionalidades sean muy variadas, por lo que en este apartado sólo nos centraremos en las características que ofrece el *Kernel*.

Multitarea.

Linux soporta un sistema verdadero de multitarea preemtivo. Todos los procesos corren independientemente de los otros. No existe un proceso que se encarga de asignar tiempo de procesador a los demás procesos.

Multi-procesador.

Linux permite el uso de varios procesadores simultáneamente. Esto significa que se puede distribuir la carga de diversas tareas entre los procesadores. De esta manera también se consigue una multitarea en paralelo real.

Acceso Multi-usuario.

Linux permite que varios usuarios estén conectados en el sistema al mismo tiempo. Para evitar que un usuario realice tareas perjudiciales para otros usuarios o el sistema, se establece un sistema de permisos. Con este sistema un usuario puede establecer permisos de lectura, escritura y ejecución a otros usuarios. Existe un usuario principal en el sistema llamado *root.* Éste tiene permisos totales dentro del sistema.

***Kernel* Monolítico.**

Existen dos variantes principales de *kernels: kernels* Monolíticos y *m icrokernels.* Los *m icrokernels* proporcionan únicamente las funcionalidades mínimas como intercomunicación entre procesos y manejo de memoria. Éstos pueden ser implementados en archivos compactos y pequeños. Las demás funcionalidades del sistema operativo son procesos autónomos que se comunican con el *m icrokernel* mediante una interfaz bien definida. La principal ventaja de estos sistemas es que son muy fáciles de mantener. Los componentes individuales del sistema trabajan independientemente unos de otros, por lo que no pueden afectar a otros componentes y pueden ser reemplazados de

manera muy sencilla. El desarrollo de estos componentes es, también, muy simple. Sin embargo, al haber interfaces de comunicación entre los componentes y el *m icrokernel* en sí, estos tipos de *kernels* son ligeramente más ineficientes y lentos que los *kernels* monolíticos.

Por contra, los *kernels* monolíticos, como el de Linux, incluyen todas las funcionalidades en un sólo bloque. Su mantenimiento es más crítico que el de un *m icrokernel,* pero a cambio se gana en velocidad. En el desarrollo de Linux se dio más importancia a la eficiencia que a la elegancia. Esto hizo que los desarrolladores se decantaran por un *kernel* Monolítico en contra de un *Microkernel*.

Kernel Modular

El *kernel* de Linux permite añadir módulos que extiendan sus funcionalidades. Los módulos del *kernel* son ficheros que contienen código que permite extender el *kernel* que está corriendo. Estos módulos son normalmente utilizados para dar soporte a nuevo *hardware*, sistemas de ficheros, añadir llamadas al sistema, etc. Cuando la funcionalidad ya no se necesita, se puede desactivar el módulo ahorrando, de esta manera, memoria.

Independencia de la arquitectura.

Esto no significa que el mismo sistema funcione en dos máquinas de distinta arquitectura, si no que el esfuerzo de las comunidades desarrolladoras ha permitido migrar Linux a diferentes arquitecturas. La mayor parte del código de Linux es directamente portable, mientras que sólo una pequeña parte ha de reescribirse. Esto ha permitido portar a un gran número arquitecturas. Hoy en día no existe ningún otro sistema operativo portado a tantos tipos de *hardware.*

Requerimientos de memoria dinámicos.

Sólo las partes que necesita un programa para funcionar son cargadas en la memoria. Cuando un nuevo proceso es creado, no realiza una petición de memoria instantáneamente. Utiliza la memoria del proceso padre hasta que desea modificar algún dato. Cuando el hijo accede a una parte de la memoria del padre en modo de escritura, primero copia esta sección a su espacio para luego

modificarlo. Este concepto se conoce como *copy-on-write*.

Memoria Virtual

Siempre puede pasar que el sistema se quede sin memoria RAM disponible. Cuando ocurre esto, Linux busca páginas de 4Kb de memoria que puedan ser liberadas. Las páginas que ya se encuentran guardadas en el disco duro son descartadas. El resto son copiadas en un espacio de memoria del disco. Este espacio de memoria es llamado memoria SWAP. Cuando una de las páginas almacenadas en la memoria SWAP quiere ser accedido, esta página es recargada a la memoria principal. Este proceso se llama paginado.

Memoria protegida.

Linux utiliza mecanismos de protección de memoria que previenen que un proceso pueda acceder al área de memoria de otro proceso o a la memoria del *kernel*. De este modo un programa erróneo no puede romper el sistema.

Librerías compartidas.

Las librerías son colecciones de rutinas que necesitan los programas para el procesado de datos. Existe un gran número de librerías estándar que son utilizadas por más de un proceso al mismo tiempo. Linux tan sólo carga en memoria estas librerías una sola vez aunque varios procesos estén utilizándolas. Como estas librerías son cargadas sólo cuando un proceso las necesita, éstas son conocidas como librerías dinámicas.

Sistemas de ficheros.

Linux soporta una gran variedad de sistemas de ficheros. El más utilizado en la actualidad por Linux es el *Extended File System (Ext3)*. Otros sistemas de ficheros soportados por Linux son el sistema de ficheros de MS-DOS, el sistema de ficheros VFAT, de Windows 95, el sistema de ficheros ISO, para acceder a CD-ROMs. El sistema de ficheros de Windows NT sólo se soporta en modo lectura, ya que aún no se ha conseguido escribir sin perder la estabilidad del sistema de ficheros.

Mención a parte són los sistemas de ficheros que permiten el acceso a dispositivos MTD[4] como las memorias Flash. Los sistemas empotrados utilizan en gran medida estos dispositivos, por lo que los mencionaremos en el capítulo 3.

TCP/IP.

El rápido crecimiento de Linux no podría haberse realizado si no hubiera sido de la colaboración de un gran número de programadores que colaboraron (y colaboran) trabajando desde la red. Esto se ha notado en el propio sistema operativo, que proporciona unas excelentes capacidades para todo tipo de comunicaciones por red.

2.2.- Procesos y tareas

Visto desde el punto de vista de un proceso que está corriendo en Linux, el *kernel* es un proveedor de servicios. Los procesos individuales existen independientemente sin que puedan afectar o ser afectados por otros procesos. Dicho de otra manera, el área de memoria de un proceso está protegida ante la modificación que pudiera hacerle otro proceso.

Hay que comentar, sin embargo, que desde el punto de vista del *kernel* esto es diferente. Sólo un programa (el sistema operativo) está funcionando en el ordenador. Este programa puede acceder a todos los recursos del sistema. Las diferentes tareas son llevadas a cabo por co-rutinas, lo que implica que cada tarea decide cuando pasar el control a otra tarea. Una consecuencia de esto es que un error de programación en el *kernel* podría bloquear el sistema entero. Una tarea podría acceder a los recursos de otras tareas y modificarlos

Cuando una tarea tiene privilegios, se dice que está en Modo Sistema, mientras que cuando no los tiene, ésta se encuentra en Modo Usuario. Alguien que mira desde fuera hacia el *kernel*, ve esta tarea como un proceso. Desde el punto de vista de un proceso se está trabajando en tiempo real. Durante este trabajo denominaremos por igual a tarea y proceso y no se hará distinción de ninguno de los dos ya que ambos se referirán al mismo concepto.

[4]Mermoy Technology Devices.

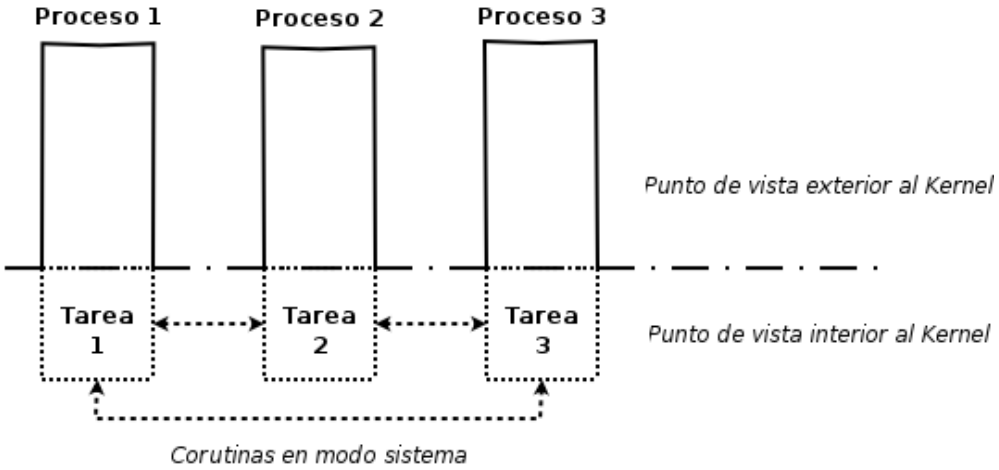


Figura 2.1: Puntos de vista interior y exterior al kernel

Una tarea puede encontrarse en varios estados. La Figura 2.2 muestra los estados más importantes. Las flechas indican los posibles cambios que se pueden realizar estando en un determinado proceso.

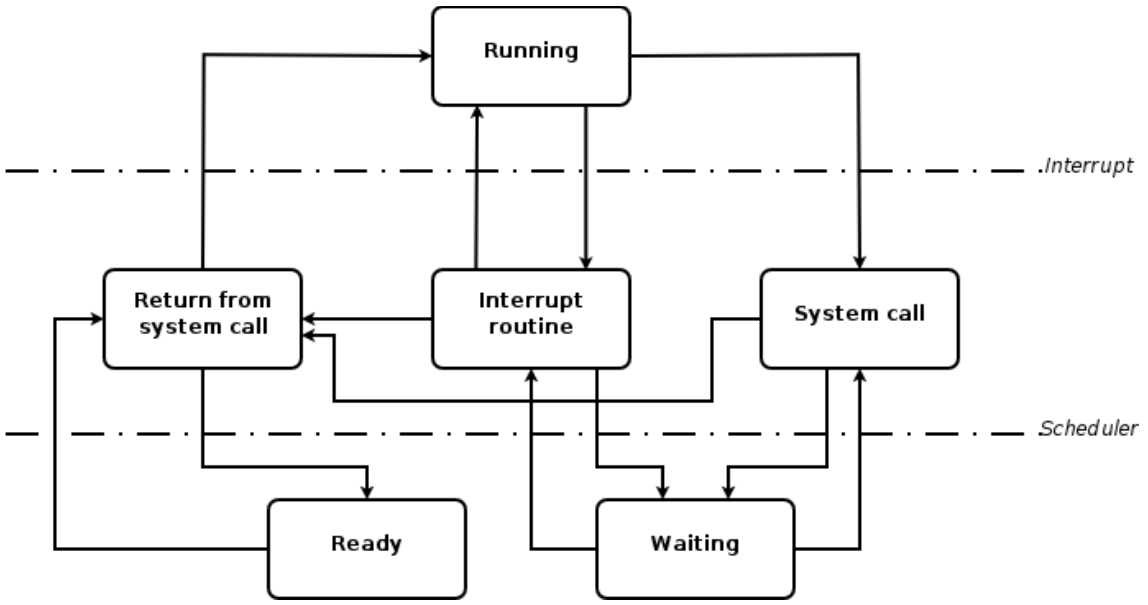


Figura 2.2: Estados de un proceso.

Corriendo (*Running*)

La tarea está activa y corriendo en Modo Usuario. En este caso el proceso avanza por el código del programa de manera normal y corriente. Solo se puede salir de este estado mediante una interrupción o una llamada del sistema (System Call). Las llamadas al sistema son interrupciones especiales.

En una interrupción el procesador cambia a Modo Sistema (*System Mode*) *y* comienza a ejecutarse la rutina de la interrupción generada.

Rutina de Interrupción (*Interrupt Routine*)

Las rutinas de interrupción se activan cuando el *hardware* activa una señal de excepción. Estas señales pueden ser provocadas porque se ha pulsado una tecla en el teclado o porque se ha recibido un paquete de la red, por ejemplo.

Llamadas del sistema (*System Calls*)

Las llamadas al sistema son generadas por interrupciones de *sofware*. Una llamada al sistema es capaz de suspender una tarea para que espere un evento determinado.

Esperando (*Waiting*)

En este estado, el proceso está esperando a un evento externo. Sólo continuará su trabajo cuando ocurra dicho evento.

Retorno de una llamada del sistema (Return from system call)

Este estado se adopta automáticamente después de cada llamada del sistema y de algunas interrupciones. En este punto el programador de tareas (*scheduler*) puede cambiar el estado del proceso a *Ready* y activar otro proceso.

Preparado (*Ready*)

El proceso compite con los demás procesos por tiempo de procesador. Éste está ocupado con otro proceso en ese momento, así que todos los procesos en espera de procesador están en estado *ready.*

Capítulo 3. Linux en Sistemas Empotrados

Un sistema empotrado es un sistema de uso específico cuya computadora se encuentra encapsulada en el dispositivo que controla. A diferencia de los ordenadores de propósito general, como los PC, un sistema empotrado realiza tareas muy específicas para las cuales ha sido construido.

En el núcleo de cualquier sistema empotrado se encuentran uno o más microprocesadores programados para realizar un número pequeño de tareas. Esto contrasta con los ordenadores de propósito general, los cuales son capaces de realizar una gran variedad de tareas mediante el uso de de diferentes aplicaciones software. Sin embargo, debido al aumento de las prestaciones del *hardware* y de su consiguiente reducción de costes, estos dispositivos, hoy en día, son capaces de realizar un gran número de funcionalidades. Véase el ejemplo de una PDA (*Personal Digital Assistant*). Actualmente, estos dispositivos realizan funciones de agenda electrónica, tienen soporte para distintos tipos de redes de comunicaciones, son capaces de funcionar como teléfono, etc.

3.1.- Clasificación Sistemas Empotrados

Debido al gran número de ámbitos en los que se pueden encontrar sistemas empotrados, existen muchas maneras de clasificarlos. En esta sección se analizarán las más relevantes: tamaño, limitaciones temporales, capacidades de comunicaciones y grado de interacción con el usuario.

3.1.1.- Tamaño

El tamaño de un sistema empotrado puede ser determinado por dos factores. El tamaño físico y las capacidades del *hardware* del dispositivo. Con relación al tamaño existen dispositivos de grandes dimensiones, como clusters y otros muy pequeños como un reloj con Linux empotrado construido por IBM.

En relación a las prestaciones del *hardware*, ésta es una clasificación difícil de realizar, ya que año tras año estas prestaciones mejoran considerablemente. Un equipo que se considera potente en la actualidad, probablemente será considerado como un sistema de pocas prestaciones al cabo de unos años[5].

[5]Esto no significa que el sistema quede obsoleto. La NASA, por ejemplo utiliza aún sistemas *hardware* que fueron desarrollados hace más de una década, pero que han sido ampliamente probados y se conoce su fiabilidad. Consideran que no es necesario arriesgar sus equipos o sus tripulaciones por utilizar dispositivos que no han sido lo suficientemente probados.

Los tres principales indicadores de las prestaciones que ofrece un sistema son la velocidad de su CPU6, el tamaño de su memoria RAM7 y la capacidad de su sistema de almacenamiento permanente. Según estos indicadores un sistema empotrado puede ser considerado pequeño, mediano o grande[8].

Se considera pequeño un sistema con una CPU poco potente, con unos de 2 MB de ROM9 y 4MB de RAM. Estos sistemas sólo son capaces de realizar pequeñas tareas que no requieran un gran consumo de recursos.

Un sistema mediano se caracteriza por tener una CPU de potencia media y tener alrededor de 32 MB de memoria ROM y 64 MB de memoria RAM. Muchos dispositivos orientados al consumo se encuentran en esta categoría. Esto incluye a dispositivos como PDAs, Móviles, reproductores multimedia, equipamiento de redes (*routers*), etc. Algunos de estos dispositivos incluyen sistemas de almacenamiento permanente. Normalmente consisten en dispositivos sin partes móviles como memorias flash. Los dispositivos de esta categoría ya son lo suficientemente potentes como para realizar varias tareas pequeñas o una tarea que requiera muchos recursos.

Un sistema grande se caracteriza por disponer de una o varias CPUs combinadas con grandes cantidades de memoria RAM y de almacenamiento masivo. Usualmente, estos sistemas se utilizan en entornos que requieren un gran número de cálculos y de mucha complejidad. Un simulador de vuelo es un buen ejemplo de estos sistemas.

3.1.2.- Limitaciones temporales (Tiempo Real)

Existen dos tipos de limitaciones temporales en cuanto a clasificación de sistemas empotrados: severas y leves. Las limitaciones temporales severas requieren que el sistema reaccione ante un determinado evento en un tiempo definido. Si el sistema tarda más en reaccionar de lo que el intervalo de tiempo indica, pueden ocurrir situaciones peligrosas. Pongamos por ejemplo una máquina cizalladora controlada por un sistema empotrado. Si un trabajador introduce accidentalmente su mano en la máquina, la máquina tiene que parar antes de que sea demasiado tarde. El sistema no puede esperar a que otro proceso libere la CPU, o a que un archivo se esté paginando a la memoria swap, ya que no se cumpliría el requisito de tiempo. Estos sistemas se llaman sistemas de tiempo real estricto (*hard real- tim e system*).

Las limitaciones temporales leves son aquellas en las que no es crítico que el sistema reaccione en un tiempo determinado. Un ejemplo de este tipo de

[6]Central Processing Unit. http://en.wikipedia.org/wiki/Cpu
[7]Random Access Memory. http://en.wikipedia.org/wiki/Random_access_memory
[8]Esta clasificación ha sido extraída del libro "*Building Embedded Linux Systems, Ed. O'Reilly*"
[9]Read Only Memory. http://en.wikipedia.org/wiki/Read-only_memory

limitaciones es el *stream ing* de audio o vídeo, donde la pérdida de una o varias imágenes suele ser tolerado por el usuario. En estos casos el sistema asegura que el intervalo de reacción se cumplirá un cierto porcentaje de veces. Se dice que son sistemas de tiempo real suave (*soft real-tim e system*).

Linux es un sistema que no ofrece *real-tim e.* Al ser un sistema de propósito general, está costando muchos esfuerzos a la comunidad desarrolladora conseguir tiempo real en Linux. Según [4]: “La versión del *Kernel* 2.4 no ofrece ni siquiera tiempo real suave. En la versión 2.6 del *Kernel* se ha conseguido acercarse algo a *soft real-tim e*, pero sigue sin serlo. Estas mejoras se han conseguido debido a la *preem ptividad* del *Kernel.* En la versión 2.4 el *Kernel* una tarea podía expulsar a otra de su estado *Running* sólo si ésta última se encontraba en modo usuario. En el *Kernel* 2.6 se ha conseguido que una tarea pueda expulsar a otra se encuentre en Modo *Kernel* o en Modo Usuario. También se han conseguido progresos en los *tim ers* del sistema, pasando de una resolución de 10ms en el *Kernel* 2.4 a una resolución de 1ms en el *Kernel* 2.6, lo que ha significado una mejora en los tiempos de respuesta.

3.1.3.- Capacidades de comunicaciones

Las capacidades para la comunicación de los sistemas empotrados son otro de los factores que permiten clasificarlos. Actualmente, estos dispositivos tienen muchas capacidades para comunicarse con otros similares, incluso aunque sean de arquitecturas distintas. El avance tecnológico junto con el abaratamiento del *hardware* y la estandarización han permitido que los sistemas empotrados dispongan de chips que permitan controlar distintos tipos de comunicaciones. Hoy en día no es raro ver como dos PDAs de distintas arquitecturas se comunican entre sí mediante protocolos como Bluetooth o incluso mediante WiFi.

Como reseña importante, adelantaremos que una de las razones por las que Linux está entrando fuerte en este sector es gracias a sus capacidades para las comunicaciones por redes.

3.1.4.- Interacción con el usuario

El grado de interacción con el usuario varía de un sistema a otro. En algunos sistemas, como SmartPhones (un híbrido entre PDA y teléfono móvil), se centran en la interacción con el usuario ofreciendo interfaces gráficas en pantallas a color. Otros sistemas, como controladores industriales el único interfaz que proporcionan son un conjunto de LEDs (*Light Em m iting Diode*) e interruptores.

3.2.- Arquitecturas de sistemas empotrados

Al contrario que en el mercado de los ordenadores personales, existen una gran variedad de arquitecturas de sistemas empotrados. Un gran número de fabricantes se dedican a fabricar microprocesadores y chips para estos sistemas. En este apartado se analizarán las principales arquitecturas que hay hoy en día. Estas arquitecturas son ARM, Power PC y MIPS.

3.2.1.- ARM

ARM (Advanced RISC[10] Machine) es una familia de procesadores producida por ARM Holdings Ltd. Al contrario que otros fabricantes de procesadores, esta empresa no manufactura sus procesadores. ARM se dedica a diseñar los núcleos de sus CPUs para sus clientes. Éstos pagan una licencia por el diseño y se encargan ellos mismos de la producción.

Esta manera de fabricar chips puede llevar a pensar que en el mercado existen una gran cantidad de arquitecturas ARM totalmente incompatibles entre ellas. Esto no es cierto, ya que estos chips comparten el mismo set de instrucciones, lo que hace que el software de todas las variantes sea totalmente compatible.

Esto no significa que todas las variantes de ARM se programen de la misma manera. Sólo los binarios resultantes tras la compilación son idénticos para todos los procesadores ARM.

Actualmente muchos fabricantes como Toshiba, Intel o Samsung desarrollan plataformas ARM. Esta arquitectura es muy popular en muchos tipos de aplicaciones por lo que es una arquitectura muy extendida. Además esta arquitectura está muy bien soportada por Linux: existen al menos 10 CPUs ARM soportadas por este sistema operativo.

3.2.2.- PowerPC

La arquitectura PowerPC (PPC) nació de la colaboración entre IBM, Motorla y Apple. De las ideas de las tres firmas se creó la arquitectura POWER (*Perform ance Optim ization With Enhaced RISC*). Esta arquitectura es

[10]La arquitectura computacional RISC (Reduced Instruction Set Computer) tiene la filosofía de utilizar conjuntos de instrucciones pequeños y simples, al contrario que la arquitectura CISC (Complex Intruction Set Computer), que es la utilizada por los procesadores x86 de los ordenadores de sobremesa.

conocida principalmente por su uso en los ordenadores Machintosh de Apple, pero se utiliza también en estaciones de trabajo y sistemas empotrados.

Al igual que ARM, existe un gran soporte para esta arquitectura en Linux. Existen una gran variedad de CPUs PowerPC en las que Linux funciona correctamente. Adicionalmente, un gran número de aplicaciones para procesadores x86 (arquitectura de los PCs de escritorio) disponibles para la arquitectura PPC. También hay soporte para Java y aplicaciones como OpenOffice han sido portadas a PowerPC. La comunidad Linux es muy activa en muchas áreas de desarrollo para esta arquitectura.

3.2.3.- MIPS

La arquitectura MIPS (*Microprocessor without Interlocked Pipeline Stages*) surgió obra de un proyecto de John Hennessey para la universidad de Stanford. Esta arquitectura es famosa por ser la base de las videoconsolas Nintendo 64 y Sony Playstation 1 y 2, aunque también se puede encontrar en muchos sistemas empotrados.

Al igual que ARM, la compañía que diseña las CPUs MIPs, *MIPs Technologies Inc.* vende sus diseños a terceros. Pero, a diferencia de ARM, existen varias implementaciones de sets de instrucciones que difieren según su versión. Compañías como IDT, Toshiba, Alchemy y LSI disponen de implementaciones de 32 bits de MIPS. Existen diseños MIPS de 64 bits que son propiedad de las compañías IDT, LSI, NEC, QED, *SandCraft* y Toshiba.

La primera versión de Linux en arquitecturas MIPS se realizó para dar soporte a estaciones de trabajo basadas en MIPS. Más tarde se comenzó a portar Linux a sistemas empotrados MIPS. Comparado con otras distribuciones como ARM o PowerPC, el uso de Linux en arquitecturas MIPS se encuentra muy limitado. Muy pocas de las grandes distribuciones de Linux han sido portadas a esta arquitectura.

3.3.- Sistemas de almacenamiento en sistemas empotrados

Una característica importante en estos sistemas es como almacenan la información de manera permanente, si es que pueden. Cuando el fabricante de un sistema empotrado quiere añadirle un sistema de almacenamiento permanente, normalmente eligen dispositivos de rápido acceso a los datos y que no tengan partes móviles. Esto último es por razones de seguridad y consumo energético. La solución más utilizada es el uso de memorias flash, también llamadas MTD (*Mem ory Technology devices*).

Las memorias flash son un tipo de memoria no volátil. Esto significa que no necesita alimentación eléctrica para mantener la información. Este tipo de memorias ofrece un acceso rápido a los datos, si bien no tanto como las memorias RAM de los PCs. Además, las memorias flash ofrecen una mayor resistencia a los golpes y vibraciones que los discos duros, por lo que son idóneas para dispositivos móviles como cámaras digitales, reproductores de audio digital y teléfonos móviles.

Estas memorias, sin embargo, tienen una serie de desventajas con respecto a los discos duros magnéticos. El principal es que estas memorias tienen número finito de ciclos de escritura. Normalmente este número se encuentra alrededor del millón de escrituras. Para solucionar esto se requieren algoritmos como el *wear leveling*, que reparte las escrituras de manera homogénea por toda la memoria. Otro de los problemas comunes de estas memorias es el *bit flipping*, el cambio espontáneo del valor de un bit. Existen mecanismos como el uso de algoritmos ECC (*Error Correcting Code*) que permiten corregir estos errores.

La necesidad de estos mecanismos implica el uso de sistemas de ficheros especiales, específicamente preparados para este tipo de memorias. Algunos de estos sistemas de ficheros son JFFS (*Journaling Flash File System*), YAFFS (*Yet Another Flash File System*), SquashFS y CramFS. Existe otro sistema de ficheros, que además permite la compresión de los datos hasta el momento de su lectura, el JFFS2 *(Journaling Flash File System version 2)*. Éste es adoptado en numerosos sistemas empotrados que disponen de poca capacidad de almacenamiento.

El uso de estos sistemas de ficheros no es estrictamente necesario. También existe la posibilidad de utilizar otros sistemas de ficheros clásicos como *Ext3* o *ReiserFS*. Sin embargo, si se quiere utilizar uno de estos sistemas de ficheros, es necesario utilizar una interfaz por encima de éstos, que implemente los algoritmos de *wear leveling y ECC.* Esta capa se la conoce como FTL (*Flash Transaction Layer)*.

3.4.- Desarrollo en sistemas empotrados

Crear software en un sistema empotrado no difiere mucho de como se hace en un ordenador normal y corriente. Para el proceso de desarrollo se requieren herramientas similares al proceso normal: compilador, ensamblador y debugador. Sin embargo, desarrollar software para sistemas empotrados difiere en algunos puntos, al proceso de desarrollo en PCs convencionales.

Generalmente, cuando se quiere desarrollar software para sistemas empotrados, hay que compilar el programa en un PC convencional. Esto suele

pasar en sistemas empotrados con muy pocas prestaciones donde la CPU del sistema empotrado no tiene potencia para el compilado. De esta manera aparecen dos participantes en el proceso de desarrollo: el *target* (objetivo) y el *host* (anfitrión). El *target* es el sistema empotrado en sí, el sistema donde correrá el programa desarrollado. El *host* es la máquina donde se desarrolla y compila el software que está desarrollando. Para compilar en el *host* es necesario utilizar un compilador especial, llamado compilador cruzado (*cross com piler*). El compilador cruzado es un compilador que se ejecuta en el host y que genera binarios preparados para funcionar en la arquitectura del *target*.

Dependiendo de las prestaciones del *target*, de sus capacidades de comunicaciones y de su grado de interacción con el programador se pueden encontrar diferentes tipos de configuraciones. Estas pueden ser Configuración enlazada (*linked setup*), configuración con almacenamiento extraíble (*rem ovable storage setup*) y configuración autónoma (*standalone setup*).

3.4.1.- Configuración enlazada (*linked setup*):

La configuración enlazada es la más común de todas. En esta configuración, el target se encuentra permanentemente enlazado al host mediante un cable físico. Este enlace suele ser, típicamente un cable serie o un enlace Ethernet. En esta configuración no hay ningún intercambio de sistemas de almacenamiento físico entre *host* y *target.* Todas las transferencias de información se realizan mediante el enlace.

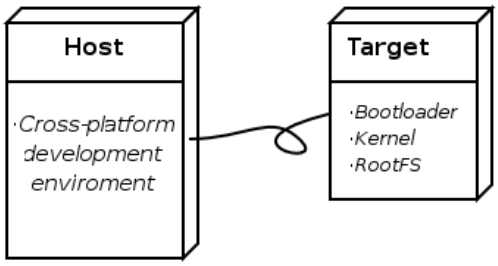


Figura 3.1: Linked Setup.

Como se ve en la Figura 3.1, el *host* contiene el compilador cruzado, mientras que el *target* contiene un *bootloader* (sistema de arranque del sistema operativo), un *kernel* y un sistema de archivos mínimo. En este tipo de configuraciones, el *kernel* puede ser cargado desde el *host* mediante el protocolo TFTP11 y el sistema de archivos puede encontrarse en un dispositivo de almacenamiento en el *target* o estar montado remotamente mediante NFS12 en el *host*. Esta segunda opción es muy utilizada ya que evita tener que

[11]Trivial File Transfer Protocol. http://en.wikipedia.org/wiki/Tftp
[12]Network File System. http://en.wikipedia.org/wiki/Network_File_System

estar haciendo copias del software desarrollado entre *host* y *target* constantemente.

En estas configuraciones es posible utilizar el enlace para debugar el programa. Sin embargo, es más común utilizar otro enlace para realizar este propósito. En muchos sistemas empotrados se utiliza un cable Ethernet y un enlace serie (RS232). El enlace Ethernet se utiliza para descargar el ejecutable, el *kernel* y el sistema de ficheros. El enlace serie RS232, al ser más lento, se utiliza para debugar.

3.4.2.- Configuración con almacenamiento extraíble (*removable storage setup*)

En esta configuración, no existe un enlace físico entre el *host* y el *target*. El programa desarrollado se compila en algún sistema de almacenamiento en el *host*, por ejemplo una memoria flash. Tras esto, el sistema de almacenamiento es transferido al *target.* y es utilizado para arrancar el dispositivo.

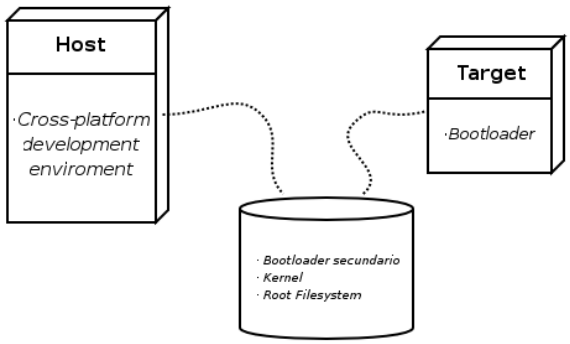


Figura 3.2: Rem ovable Storage Setup.

Como en la configuración enlazada, el *host* contiene la plataforma de desarrollo con el compilador cruzado. El *target* contiene un *bootloader* mínimo. El resto de componentes se encuentran en el sistema de almacenamiento.

Esta configuración se suele utilizar en sistemas empotrados que no tienen capacidades de comunicación por red, o en las fases iniciales de desarrollo de un sistema que si las tendrá.

3.4.3.- Configuración autónoma (*standalone setup*)

En esta configuración, el *target* contiene todo el software requerido para arrancar, operar y desarrollar software adicional. Esta configuración es muy similar a la utilizada en los PCs o en las estaciones de trabajo, donde los programas se desarrollan y compilan en la misma máquina objetivo.

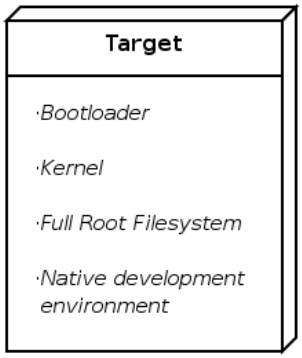


Figura 3.3: Stand Alone Setup.

Al contrario que en las otras configuraciones, en la configuración autónoma no es necesario disponer de una herramienta cross-compiladora ya que todas las herramientas corren directamente en el sistema empotrado. Por la misma razón tampoco es necesario establecer ningún método de transferencia de información entre el *host* y el *target.*

Esta configuración se utiliza en sistemas empotrados de altas prestaciones, los cuales tienen potencia suficiente para realizar compilaciones.

Capítulo 4. Introducción a las redes.

En el ámbito de las redes de comunicaciones son necesarias una serie de reglas que permitan la comunicación entre dispositivos. Asimismo es necesario que estas reglas sean las mismas para todo tipo de redes. Si no fuera así, no sería posible la comunicación entre distintas redes.

Estas reglas preestablecidas o protocolos, están descritas en lo que se llaman modelos de referencia, los cuales los podemos definir como una pila de protocolos organizados en capas, cada una construida encima de otra a la cual proporciona funcionalidades extra. Los dos modelos más importantes son el Modelo de Referencia OSI (*Open System s Interconnection*) y el Modelo de Referencia TCP/IP.

4.1.- El modelo de referencia OSI

El modelo de Referencia OSI fue creado por la ISO (*International Standars Organization*) como un primer paso hacia la estandarización de los diversos protocolos de red existentes. Su primera versión data de 1983 y en 1995 fue revisado. Su nombre real es Modelo de Referencia OSI de la ISO. Nosotros nos referiremos a él como el Modelo OSI. Este modelo no ha tenido gran éxito de implantación, sin embargo define a nivel teórico como debería ser una comunicación.

El modelo OSI dispone de 7 capas o niveles. Cada una de estas capas proporciona un nivel de abstracción diferente y realiza una función bien definida. Es importante tener en cuenta que el Modelo OSI no es una arquitectura de red en sí ya que no especifica los servicios y protocolos que serán usados. El modelo tan sólo define cual es la misión de cada capa. La ISO ha producido diversos protocolos que operan en cada una de estas capas.

En la Figura 4.1 se pueden ver las capas del Modelo OSI. A continuación se describirán las diferentes funciones que realiza cada una de estas capas comenzando por la capa inferior y acabando en la superior.

Figura 4.1.- *El Modelo OSI.*

En el Nivel 1 o Capa Física se transmiten tiras de bits en un canal de comunicaciones. En esta capa se describen características electromagnéticas de la comunicación: voltaje de la señal, frecuencia, potencia de transmisión, etc. También se describen características mecánicas de los dispositivos de comunicación: número de pines, tipo de cables, etc.

El Nivel 2 o Capa de Enlace de Datos se encarga de transformar tiras de bits en información libre de errores. Para ello separa las tiras de bits en bloques llamados tramas de datos y los transmite secuencialmente. Si el servicio proporciona fiabilidad ante errores, el receptor de los datos puede confirmar los datos recibidos con una trama de reconocimiento de datos. En esta capa se puede incluir un mecanismo regulador de tráfico para evitar que el transmisor sature de información al receptor. Normalmente este mecanismo de regulación de tráfico se encuentra integrado con el mecanismo de manejo de errores. En redes de tipo *broadcast* (redes en las que todos los usuarios pueden escuchar la información que se envían dos usuarios, como por ejemplo las redes Ethernet), se debe implementar una subcapa que maneje el acceso de los usuarios a la red.

El Nivel 3 o Capa de Red se encarga de ofrecer una comunicación homogénea entre distintas redes sin importar cual sea su arquitectura. En esta capa se deben resolver los problemas de comunicación entre redes de distintas arquitecturas y de como hacer llegar los paquetes de información desde origen a destino. Este problema se puede resolver mediante rutas estáticas en redes que no cambian su topología o mediante enrutamiento dinámico en rutas que

varían su topología. La Capa de Red debe proporcionar también calidad de servicio que permita evitar congestión en la red y que asegure unos valores mínimos de retardo, tiempo de transmisión, *jitter*[13], etc.

El Nivel 4 o Capa de Transporte tiene como misión establecer la comunicación extremo a extremo entre dos usuarios. Esta capa también debe determinar que tipo de servicio se provee a la Capa de Sesión. Los dos tipos de servicios más usados en esta capa son una comunicación punto a punto orientada a conexión que entrega los mensajes en orden y libres de errores[14] y una comunicación punto a punto que no garantiza que los paquetes lleguen a destino ni que lo hagan en orden. La Capa de transporte es una capa que proporciona comunicación real punto a punto. En las capas inferiores los protocolos se comunican entre cada máquina y su vecino más cercano. En una comunicación normal emisor y destinatario suelen estar separados por diversos *routers.* La diferencia entre la comunicación en las capas 1 a 3 y ésta en las capas 4 a 7 se muestra en la Figura 4.2.

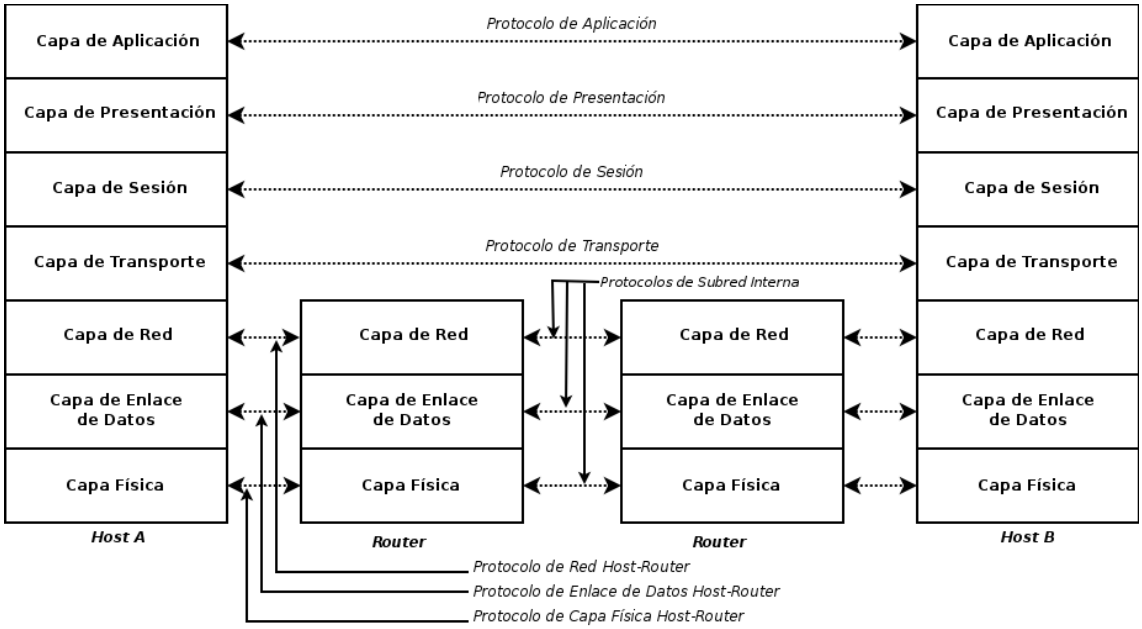


Figura 4.2.- *Diferencia entre protocolos punto-punto y protocolos extrem o-extrem o.*

El nivel 5 o Capa de Sesión permite a los usuarios de distintas máquinas establecer sesiones entre ellos. Mediante el uso de sesiones se pueden proporcionar distintos servicios como son el control de dialogo (control de turnos al transmitir) o la sincronización (uso de marcadores para recuperar el punto en que se quedó una transmisión larga de datos que ha sido interrumpida) entre otros.

[13]Variación del retardo.
[14]Asegurar que no se reciban errores es imposible. Por una comunicación libre de errores se entiende a aquella en la que los errores producidos no son percibidos por el usuario.

El nivel 6 o Capa de Presentación es la responsable de la entrega y el formateo de la información a la capa de aplicación. El motivo es un posterior proceso de la información. Básicamente se encarga de la sintaxis y la semántica de la información transmitida.

El nivel 7 o Capa de Aplicación contiene los protocolos que las aplicaciones realizan para comunicarse entre sí. Ejemplos de estos protocolos son los protocolos HTTP, FTP, Telnet15, etc.

4.2.- El modelo de referencia TCP/IP

El Modelo de Referencia TCP/IP fue creado por el Departamento de Defensa de los Estados Unidos (DoD) al gestarse el proyecto ARPANET, el embrión de Internet. El modelo fue creado con el propósito de interconectar diversas redes entre sí. Al contrario que en el Modelo OSI, primero se crearon los protocolos que permitieran esta interconexión y, a medida que fueron apareciendo nuevos tipos de redes, se creó el modelo en sí. Su primera versión data de 1974 y se la conoce como Modelo de Referencia TCP/IP. Su última revisión se realizó en 1988.

El Departamento de Defensa de los Estados Unidos de América diseñó este modelo otorgándole una gran importancia a la capacidad de que la red siguiera interconectada aún cuando cayeran diversos nodos o, incluso, diversas subredes. Otro de sus principales objetivos era disponer de una arquitectura flexible que soportara el uso de muchos tipos de aplicaciones, desde la simple transmisión de datos hasta comunicaciones de voz en tiempo real. Todos estos requerimientos llevaron a la creación de una red de conmutación de paquetes cuyo modelo de referencia recibe el nombre de sus dos protocolos más importantes: TCP (*Transfer Control Protocol*) e IP (*Internet Protocol*).

[15]No hay que confundir los protocolos de aplicación con las aplicaciones en sí. Existen numerosas aplicaciones que reciben el mismo nombre que el protocolo de nivel 7 que utilizan. Un claro ejemplo de este caso es la aplicación Telnet.

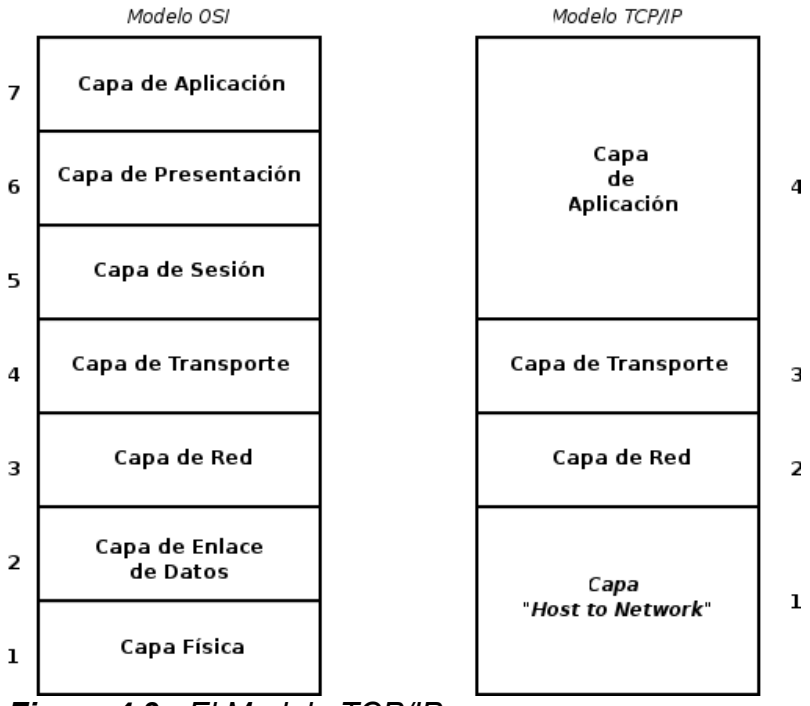


Figura 4.3.- *El Modelo TCP/IP.*

La Capa de Internet es la que se encarga de permitir la comunicación entre *hosts* situados en diferentes subredes. Estos *hosts* inyectan paquetes en la red los cuales viajan independientemente hasta llegar a su destino. La Capa de Internet define un formato oficial de paquetes y un protocolo, también oficial, llamado IP. IP no ofrece calidad de servicio, por lo que no garantiza que un paquete llegue a destino en un determinado intervalo ni que sea entregado a su destino. Tampoco garantiza que los paquetes lleguen en orden, que lleguen corruptos o duplicados. La única garantía que ofrece IP es que la cabecera del paquete es correcta. El motivo de que este protocolo no ofrezca ningún tipo de calidad de servicio es para liberar de carga a los *routers* que comunican las distintas redes. Si se desea una cierta calidad en el servicio se deben utilizar protocolos de capas superiores que si la ofrezcan. El enrutado de paquetes es uno de los mayores retos de la capa. La función principal de IP es entregar los paquetes enviados a su destino evitando congestión. La Capa de Internet es muy similar, en cuanto a funcionalidades, a la Capa de Red del Modelo OSI.

Es importante tener en cuenta que en las capas superiores a IP la información viaja extremo a extremo, mientras que a partir de este nivel lo hace punto a punto. Si nuestro paquete debe atravesar varias redes, tendrá que atravesar los diversos nodos que comunican a éstas hasta llegar a su destino. Para solucionar este problema se utilizan los *routers*, los cuales tienen la información de la topología de las redes que interconectan. Los *routers* están organizados en una jerarquía, en la cual los *routers* superiores tienen información de más redes que los inferiores. De esta manera, cuando un *router* tiene que

encaminar un paquete a una red que no conoce, envía el paquete a su *router* superior.

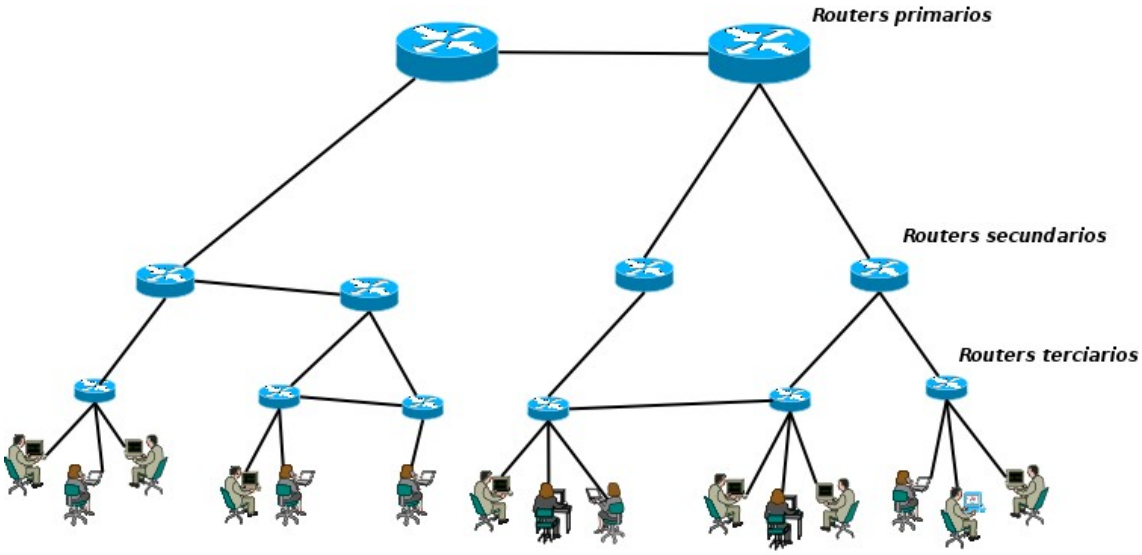


Figura 4.4.- *Jerarquía de routers de tres niveles. Los routers superiores conocen m ás redes que los inferiores.*

La capa superior a la Capa de Internet es la Capa de Transporte. Ésta es muy similar a la Capa de Transporte del Modelo OSI ya que proporciona transferencia de datos transparente extremo a extremo de la comunicación. Esta capa añade diversas funcionalidades a la capa de red como son la capacidad de múltiples comunicaciones entre dos o más *hosts* y la detección de errores extremo a extremo. El Modelo TCP/IP proporciona dos protocolos principales diferentes, donde cada uno está preparado para un tipo de comunicaciones. El primero es el protocolo TCP (*Transfer Control Protocol*), el cual proporciona recuperación de errores extremo a extremo, ordenación de paquetes, control del flujo y congestión de la comunicación. Además asegura la transferencia completa de datos extremo a extremo. El segundo protocolo, el protocolo UDP *(User Datagram Protocol),* es más ligero que TCP y no incluye ninguna de las funcionalidades que éste proporciona, por lo que deja a la capa de aplicación la decisión de cuales son necesarias para la comunicación. Ambos protocolos son ampliamente utilizados debido a la gran diversidad de tipos de comunicaciones existentes hoy en día.

El Modelo TCP/IP no utiliza capas de Sesión y Presentación. En el momento que se diseñó el modelo, no se percibió la necesidad de ellas, lo cual parece una decisión acertada, ya que actualmente pocas aplicaciones hacen uso de estas capas.

La Capa de Aplicación es análoga a la capa del mismo nombre del Modelo OSI. Contiene los protocolos de más alto nivel que interaccionan con las

aplicaciones utilizadas por los usuarios. Ejemplos de estos protocolos pueden ser HTTP16, FTP17, etc.

El Modelo TCP/IP deja un gran vacío bajo la Capa de Internet. La llamada capa *"host-to-network"* queda muy abandonada por el modelo TCP/IP. No se creó ningún protocolo para esta capa y solo se especifica que el fabricante debe proporcionar algún mecanismo para que los paquetes IP sean inyectados en la red.

4.3.- El Modelo Híbrido

El Modelo de Referencia OSI no ha conseguido gran éxito en cuanto a su implementación. Es un modelo con una gran base teórica pero que presenta serias dificultades para llevar a cabo su implementación, principalmente, debido a problemas de flexibilidad. En el Modelo TCP/IP ocurre el caso contrario. A pesar de presentar serias lagunas en cuanto al modelo en sí, es un modelo altamente flexible que ha sido utilizado con mucho éxito.

En cuanto a desarrollo, el Modelo OSI primero creó el modelo en sí para a continuación crear los protocolos que se usarían en él. En el caso del Modelo TCP/IP este proceso se produjo al revés. Primero aparecieron los protocolos para más tarde crearse el modelo que los uniera.

El Modelo OSI apareció cuando los protocolos del Modelo TCP/IP ya llevaban tiempo funcionando exitosamente en gran número de redes. Sin embargo, y a pesar del buen funcionamiento de sus protocolos, el modelo deja muchos vacíos en cuanto a su estructura, sobre todo en que debe haber debajo de la Capa de Internet.

Puede observarse que allí donde falla un modelo es fuerte el otro, lo que ha llevado a que generalmente se use un modelo de referencia que reúne las mejores características de ambos. A este modelo se le conoce, popularmente, como Modelo Híbrido. No hay ninguna organización detrás de este modelo, simplemente es el que los fabricantes y desarrolladores utilizan. Al ser el modelo más extendido actualmente es el que utilizaremos en este trabajo. Por herencia histórica, y por ser los protocolos TCP e IP los más importantes, nosotros nos referiremos a este modelo como Modelo TCP/IP o Modelo Híbrido indistintamente.

Como puede verse en la Figura 4.5, el modelo utiliza 5 capas. Las dos primeras, las capas Física y de Enlace de Datos realizan las mismas funciones que sus homólogas del Modelo OSI. Existen numerosos protocolos que operan

[16]HyperText Transfer Protocol. http://en.wikipedia.org/wiki/Http
[17]File Transfer Protocol. http://en.wikipedia.org/wiki/Ftp

en estas capas, los cuales dependen del tipo de red en el que se alojen los *hosts*. De todos ellos, el protocolo más conocido es el protocolo Ethernet, una variante del protocolo IEEE 802.3. Las tres capas superiores, las capas de Red, Transporte y Aplicación tienen las mismas funcionalidades que las capas de Internet, Transporte y Aplicación respectivamente del modelo TCP/IP. En cuanto a protocolos, la Capa de Red, también llamada Capa de Internet, aloja al protocolo IP como protocolo principal. En la Capa de Transporte encontramos a los protocolos TCP y UDP, mientras que en la de aplicación encontramos los típicos que ya podíamos encontrar en los modelo OSI y TCP/IP.

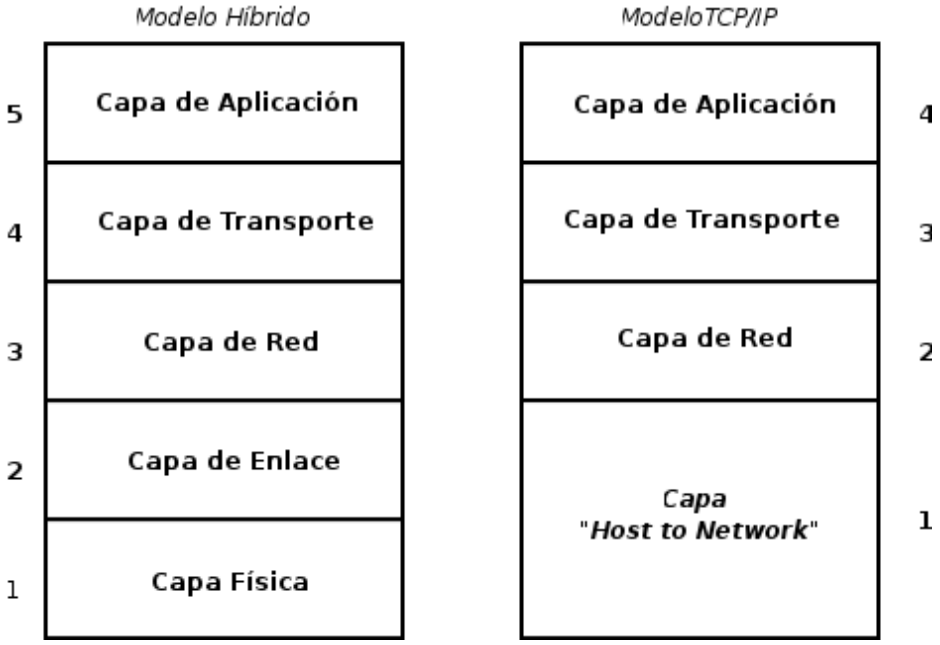


Figura 4.5.- *El Modelo Híbrido junto al Modelo TCP/IP.*

4.4.- Ejemplo de comunicación entre dos dispositivos situados en diferentes redes.

Para este trabajo consideraremos que todas las comunicaciones, a no ser que se especifique lo contrario, se realizan utilizando el modelo TCP/IP híbrido en redes Ethernet[18]. Para el ejemplo de este apartado supondremos una comunicación bajo estas condiciones y que utiliza el protocolo de transporte TCP. Para simplificar el ejemplo, supondremos también que en la comunicación no existe fragmentación de paquetes en la capa IP ni en la de Enlace. En este ejemplo se verá como evoluciona un paquete desde la capa de aplicación hasta que es enviado a la red.

[18]Debido a las escasas diferencias entre el protocolo IEEE 802.3 y el protocolo Ethernet, se utilizarán ambos nombres para referirnos al protocolo Ethernet ya que éste se encuentra más generalizado.

Imaginemos un usuario desea establecer comunicación con un servidor web situado en otra red. Para ello utilizaremos el protocolo de aplicación HTTP. La capa de aplicación entregará el mensaje a la capa de transporte, donde se añadirá una cabecera al mensaje que contiene información de control. Entre esta información de control se encuentran los puertos de origen y destino, que permiten diferenciar distintos flujos de información entre las dos máquinas. Otro campo de la cabecera añadida en la capa de transporte es el campo *checksum* , que permite detectar si se ha producido un error en el mensaje entre el punto de origen y el de destino. En este caso, el protocolo de transporte usado es TCP, la cabecera incluye diversos campos adicionales. De los campos utilizados por las cabeceras TCP, el más destacable es el número de secuencia. Con este campo se puede detectar la pérdida o la recepción de paquetes desordenados. TCP implementa mecanismos de retransmisiones para los paquetes perdidos y mecanismos de control de flujo para evitar que un *host* con una conexión de red muy rápida no sature a otro con una más lenta. Todos estos mecanismos hacen uso de los números de secuencia.

Una vez se ha añadido la cabecera de transporte, esta capa entrega el mensaje a la capa IP, donde se añade una cabecera cuyos campos más importantes son las direcciones de origen y destino del paquete.

Tras esto, el paquete será entregado a la capa de Enlace, donde se le añadirá una cabecera que contiene entre otros la dirección origen y destino para el protocolo de red utilizado, en este caso Ethernet. Es importante diferenciar entre las direcciones de nivel IP y las direcciones de nivel de Enlace. En el nivel IP, el sistema de direcciones es global y permite identificar y dirigirse a una máquina con su dirección, mientras que en el nivel Ethernet, el sistema de direcciones local, es decir usado por usuarios dentro de una misma red.

Finalmente el paquete se envía a la capa física donde será transmitido por la red hacia su destino.

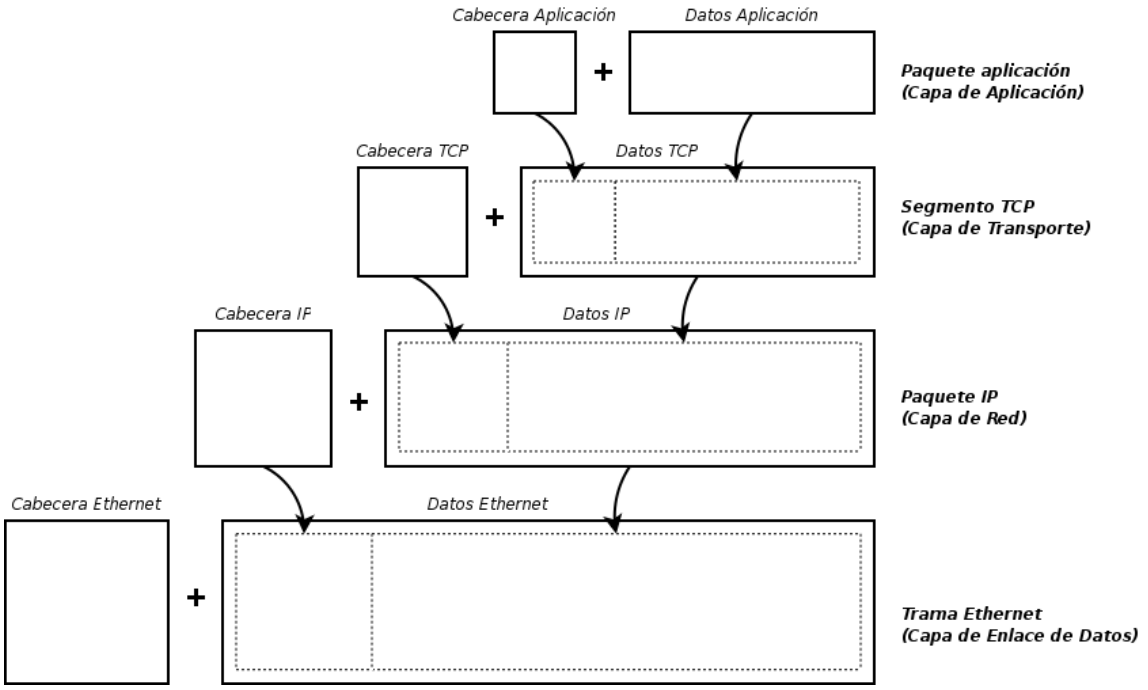


Figura 4.6: Los paquetes y cabeceras capa a capa.

4.5.- Redes en Linux

Como ha podido verse en las secciones anteriores, las comunicaciones entre redes se realizan mediante el uso de modelos estructurados en capas. En este capítulo nos centraremos en como Linux realiza una comunicación por red utilizando el Modelo Híbrido.

Desde el punto de vista de una aplicación, la comunicación se realiza mediante el uso de *sockets*, los cuales definen los dos extremos de una comunicación y permiten el intercambio de información entre estos dos extremos. Para permitir el uso de la red a una aplicación, Linux ofrece la interfaz *BSD socket interface*. La interfaz *BSD Socket* ofrece a los procesos las funciones necesarias para la comunicación vía red.

Debajo de la Interfaz *BSD Socket* se encuentra la capa *INET Socket*, que maneja los puntos extremos de la comunicación para los protocolos TCP y UDP. Esta capa se encuentra representada por la estructura de datos *sock.* A los *sockets* de la capa *INET Socket* se les conoce como *INET Sockets.*

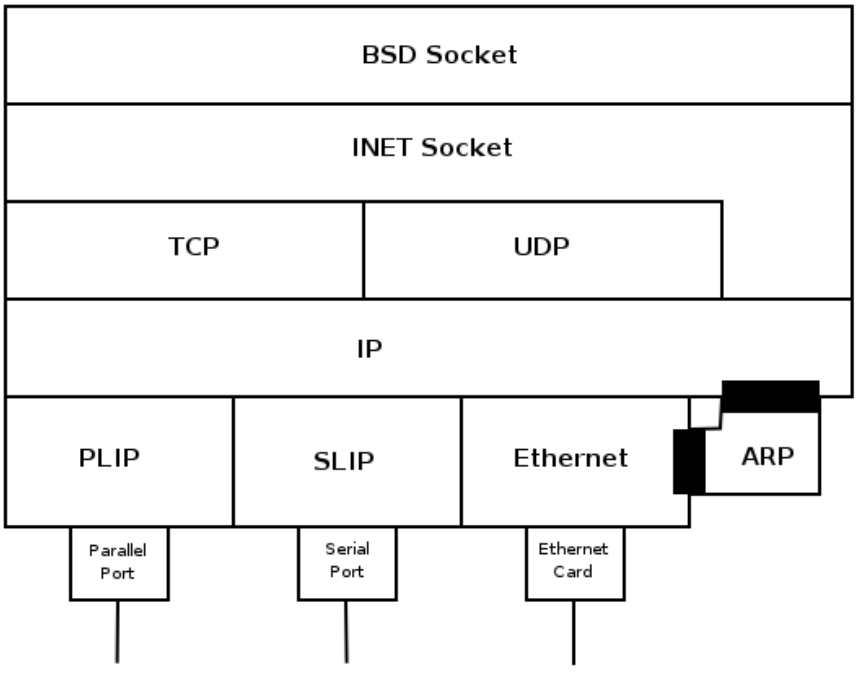
La capa que se encuentra debajo de la Capa *INET Socket* viene determinada por el tipo de *socket* que se haya utilizado en la Capa *INET Socket.* Concretamente se pueden encontrar las capas TCP, UDP o la capa IP directamente. La capa TCP implementa el Protocolo TCP, la capa UDP hace lo propio con el Protocolo UDP y en la capa IP se encuentra el código del

Protocolo IP. En la Capa IP confluyen todos los flujos de comunicación de las capas superiores.

Por debajo de la Capa IP se encuentran los dispositivos de red, a los cuales la Capa IP les pasa los paquetes. Estos dispositivos son los que se encargan de el transporte físico de la información. En la Figura 4.7 puede verse que existen diferentes protocolos de nivel 2 en función del tipo de red a la que quiera conectarse.

El dibujo muestra protocolos y dispositivos físicos para los Puertos Serie y Paralelo y para Redes Ethernet. Existen numerosos tipos de redes a las que Linux es capaz de conectarse ya que implementa los protocolos de nivel 2 necesarios para utilizar estas redes y existen drivers para manejar los dispositivos físicos que permiten la conexión a estas redes. Algunos de estos tipos de redes son IEEE 802.11x, también llamadas redes WiFi (Wireless Fidelity), Fibra Óptica (FDDI), Infrarojos (IrDa), Bluetooth, etc.

En la Figura 4.7 puede verse un esquema de las distintas capas.



Fuente: *Linux Kernel Internals. Ed.Addison Wesley*

Figura 4.7.- *Estructura de capas desde el punto de vista de Linux.*

4.5.1.- Ejemplo de comunicación desde el punto de vista de linux

Para comprender mejor la implementación de red, seguiremos los datos que son enviados a través de la red desde un proceso (aplicación) A a un proceso B. Asumiremos que ambos procesos ya han creado los *sockets* y que están conectados mediante las funciones ***connect()*** y ***accept()***. En el ejemplo asumiremos una comunicación mediante el Protocolo TCP y que la información circulará por una red Ethernet.

Los datos son enviados desde el proceso A al proceso B. Estos datos se encuentran almacenados en un *buffer* de longitud *Length*. El proceso A contiene el siguiente fragmento de código:

write(socket, data.Lengh);

que utiliza la función del *Kernel* ***sys_write()***. Esta comprueba ciertas condiciones como que el acceso de lectura/escritura en el área de memoria referenciada por los datos pueda ser realizado.

La operación de escritura para los *sockets BSD* es ***sock_write()***, que sólo se encarga de funciones administrativas. Luego los parámetros de la operación de escritura son transferidos a una estructura *m essage*. Como el *socket* es de la familia *AF_INET* (Utilizará el protocolo IP), ***sock_write()*** llamará a la función ***inet_sendmsg()***, a la cual le pasa como parámetros el puntero a la estructura de datos del *socket BSD*, el puntero a la estructura del mensaje, la longitud de los datos y una indicación de si está permitido bloquear la función (*blocking flag*) además de una serie de *flags* adicionales.

La función ***inet_sendmsg()*** extrae un puntero a la estructura *socket INET.* Este puntero lo ha extraído de los datos del *socket BSD* que se le han pasado. En este ejemplo, la estructura *socket INET* contiene los datos esenciales utilizados en las capas TCP e IP. El puntero *prot* de esta estructura referencia al vector de operaciones de la implementación TCP. La función llama a la función ***tcp_sendrasg()*** de este vector de operaciones, pasándole parámetros como el puntero al *socket INET*, el puntero a la estructura *m essage,* el tamaño de los datos, el flag de *blocking* y los *flags* adicionales. Estos *flags* son utilizados para indicar prioridad de transferencia "*out of band*" a los datos. En este ejemplo no es el caso, por lo que no son necesarios.

Hasta este momento, los datos únicamente han atravesado los diferentes niveles de abstracción. En la función ***tcp_sendmsg()***, comienza la comunicación propiamente dicha. Primero se realizan diversos tests para comprobar diferentes condiciones de errores. Estas condiciones pueden ser que el *socket* no esté preparado para enviar datos. Tras esto se realiza la comunicación en sí mediante la función ***do_tcp_sendmsg()***. Esta función inicializa la estructura *sk_buff* mediante la operación de protocolo ***build_header()*** para completar la cabecera del paquete.En este ejemplo la operación de protocolo es ***ip_build_header()***. Una vez las cabeceras de las

capas inferiores son inicializadas, la cabecera del protocolo TCP es añadida mediante la función ***tcp_build_header()***.

La operación de TCP de envío copia ahora los datos provenientes del espacio de memoria del proceso al segmento TCP. En este punto se calcula el campo *checksum* . Para la optimización de este proceso se utiliza la función ***csum_partial_copy_formuser()***, la cual realiza ambas operaciones en un sólo paso. Si la longitud de los datos copiados excede el valor típico del *Maxim um Segm ent Size (MSS)*, éstos son divididos en varios paquetes. También es posible, para bloques de datos muy pequeños, recoger varias operaciones de envío en un sólo segmento TCP.

El paquete, almacenado en la estructura *sk_buff*, es transferido mediante la función ***tcp_send_skb()***. Ésta añade información específica de protocolo a la cabecera. En este punto se llama a la función de operación protocolo ***queue_xmit()*** que sea necesaria. En este caso es la función ***ip_queue_xmit()***. Luego se almacena el paquete en una cola de espera de paquetes preparados para la transferencia.

En este momento la función ***ip_queue_xmit()*** añade a la cabecera IP campos que sólo pueden ser añadidos en este momento, como son el campo *checksum* de la cabecera IP. Después de esto entrega el paquete a la función ***dev_queue_xmit()***. En este ejemplo se asume que el dispositivo utilizado es una tarjeta Ethernet del tipo WD8023.

La función ***dev_queue_xmit()*** finalmente, llama a la función ***do_dev_queue_xmit()***. Esta segunda función utiliza el puntero ***hard_start_xmit()***. Para la tarjeta WD8023 este puntero referencia a la función ***ei_start_xmit()***, la cual entrega los datos al adaptador de red. En este punto la tarjeta transmite los datos por la red Ethernet.

En el otro extremo de la comunicación el proceso B está esperando recibir datos del proceso A. Para realizar esto, ejecuta una operación de lectura:

read(socket,data.Length);

Esta llamada atraviesa diferentes niveles de abstracción mediante las funciones ***sys_read()***, ***sock_read()***, ***inet_rcvmsg()*** y ***tcp_rcvmsg()***. Si el buffer de recepción del *socket INET* está vacío, el proceso es bloqueado. El proceso se desbloquea cuando se reciben datos. Una vez el proceso se ha desbloqueado o si ya hay datos en el buffer de recepción, éstos son copiados en el espacio de usuario en la memoria el proceso. En este punto consideramos que el proceso se bloquea esperando datos. A continuación analizaremos el proceso desde que la tarjeta de red Ethernet recibe los datos hasta que se desbloquea el proceso.

La trama Ethernet transmitida es recibida por la tarjeta de red del ordenador de destino. Al igual que antes, consideraremos que el adaptador de red de la máquina receptora es una tarjeta Ethernet WD8023.

Tras recibir la trama Ethernet, la tarjeta de red lanza una interrupción. Ésta es manejada por la función ***ei_interrupt()***. Si la transferencia vía Ethernet ha sido completada sin errores, la función ***ei_receive()*** será llamada con una referencia al adaptador de red. ***ei_receive()*** utiliza una operación *block_input* para escribir el paquete en un *buffer* recién creado. En nuestro ejemplo esta operación es realizada por la función ***wd_block_input()***. Al igual que en la fase de envío, este *buffer* incluye espacio para la estructura *sk_buff*. Esta estructura se inicializa por la función ***ei_receive()*** tras la llamada a ***wd_block_input()***.

Una vez hecho esto, la función ***netif_rx()*** es llamada con el paquete como argumento. Esta lo añade a la *backlog list*. Existe sólo una sola lista de este tipo en todo el sistema. Esta lista contiene todos los paquetes recibidos por el sistema. Todas las funciones utilizadas para la recepción de paquetes que se describen a continuación son ejecutadas por la interrupción.

La función ***netif_rx()*** marca la rutina *bottom -half* en la máscara *bh_m ask*. La función *net_bh()* es llamada por ***do_bottom_half()*** con el marcador de máscara activado. La función ***do_bottom_half*** es llamada después de las llamadas de sistema y las interrupciones lentas (normales). La llamada no se realiza si una interrupción ha interrumipido a otra interrupción o a la misma ***do_bottom_half()***.

La función ***net_bh()*** mueve el puntero *union h* de la esctructura *sk_buff* al principio del paquete de protocolo, justo después de la cabecera Ethernet. En este caso el protocolo es IP. Tras esto se llama a la función ***ip_rcv()***. En esta función se comprueba que la cabecera IP es correcta y las rutinas para el manejo de las opciones de IP son ejecutadas si son necesarias. En este punto, si la máquina receptora fuera un *router* probablemente tendría que reenviar el paquete hacia su destino. Este proceso de direccionamiento se realiza mediante la función ***ip_forward()***. Si el paquete recibido estuviera fragmentado son reconstruidos mediante ***ip_defrag()***. Asumimos que este no es nuestro caso.

El puntero se desplaza ahora al final de la cabecera IP, por lo que apunta al inicio de la cabecera del siguiente protocolo. Este protocolo está especificado en el campo *protocol* de la cabecera IP. El paquete que había sido enviado era del protocolo TCP. En este punto se llama a la función de recepción de protocolo apropiada. Para TCP es ***tcp_rcv()***. Ésta llama a la función ***get_tcp_sock()*** para determinar a que *INET socket* pertenece el segmento TCP. Esto lo hace mediante referencia a las direcciones y puertos de origen y destino. Tras una serie de tests de consistencia se llama a ***tcp_data()***. Esta función entra el buffer *sk_buff* en la lista de datos recibidos por el *socket*. Si se han recibido datos nuevos (podría darse el caso de que lleguen paquetes

replicados), se envía el paquete apropiado de reconocimiento (ACK) y la operación de *INET Socket* ***data_ready()*** es llamada. Hasta la llamada de la función ***data_ready(),*** todas las acciones relativas a recibir paquetes han sido llevadas por el *Kernel* fuera de del flujo de programa de cualquier proceso. ***data_ready()*** levanta todos los procesos esperando a algún evento en el *socket*. En este caso el proceso que se levanta es el proceso B ya que tiene los datos listos para ser recogidos.

Tras esto el proceso de envío de datos entre el proceso A y el proceso B ha finalizado. Como ha podido observarse, se han atravesado diferentes capas del sistema operativo. Los datos han sido copiados sólo cuatro veces: del área de usuario del proceso A a la memoria del *Kernel*, de la memoria del *Kernel* al adaptador de red, de la tarjeta de red de la segunda máquina al espacio de memoria del *Kernel* y de la memoria del *Kernel* al área de memoria de usuario del proceso B.

En la implementación de Linux de TCP/IP se ha tenido mucho cuidado de realizar el mínimo número de operaciones de copia. Se puede apreciar que la implementación de red es muy embrollada: existen una gran cantidad de funciones dependientes unas de otras, y no es fácil explicar a que capa pertenecen. Ante esta implementación primó la eficiencia y la velocidad de operación y sobre una estructura bien ordenada.

Capítulo 5. Las librerías Libpcap

Las librerías *Libpcap* son unas librerías escritas en lenguaje C y publicadas bajo licencia *open source* que permiten programar un monitorizador de paquetes en redes locales de manera rápida y sencilla. Están escritas para soportar tanto sistemas operativos Windows como Linux. Nosotros nos nos centraremos en su versión Linux.

5.1.- Funcionamiento de un monitorizador de red

Un monitorizador de red es un programa de escucha de una red y se encarga de recoger todos los paquetes que circulan por ésta.

Para poder escuchar la información que circula por una red, se deben de reunir, como mínimo, dos requisitos. El primero es que la red sea de tipo *broadcast.* El segundo es que la tarjeta de red del usuario que quiera realizar la escucha se encuentre en modo *m onitor*.

Las redes de tipo *broadcast* son aquellas en la que la información que envía un usuario a otro se expande por toda la red, llegando a todas las máquinas conectadas pero que solo la procesan los destinatarios. Cuando un usuario de estas redes envía una trama de información, todas las máquinas leen la dirección de destino de su cabecera. Si al comparar la dirección de destino con la suya ven que no son iguales, descartan la trama. Si ambas direcciones son iguales se procesará el paquete. El tipo de redes *broadcast* más conocido en la actualidad son las redes Ethernet.

Si se desea procesar una trama de información que no va dirigida a nosotros es necesario especificarle a nuestra tarjeta de red que lo haga. Para ello hay que activarle el *flag* de *m onitor*. En otras palabras, una máquina con el *flag* de *m onitor* activado procesa todos las tramas que recibe independientemente si van dirigidas a ella o no. Cuando una máquina tiene activado el *flag* de *m onitor* se dice que trabaja en modo promiscuo.

En la Figura 5.1 puede observarse un escenario que cumple las dos condiciones anteriores. Por un lado la red de la figura es un medio *broadcast* ya que todas las máquinas conectadas reciben la trama de información enviada y sólo la máquina destinataria de la trama la procesa. Por otro lado, la máquina que está monitorizando el tráfico puede recibir tramas que no van dirigidas a ella ya que trabaja en modo promiscuo.

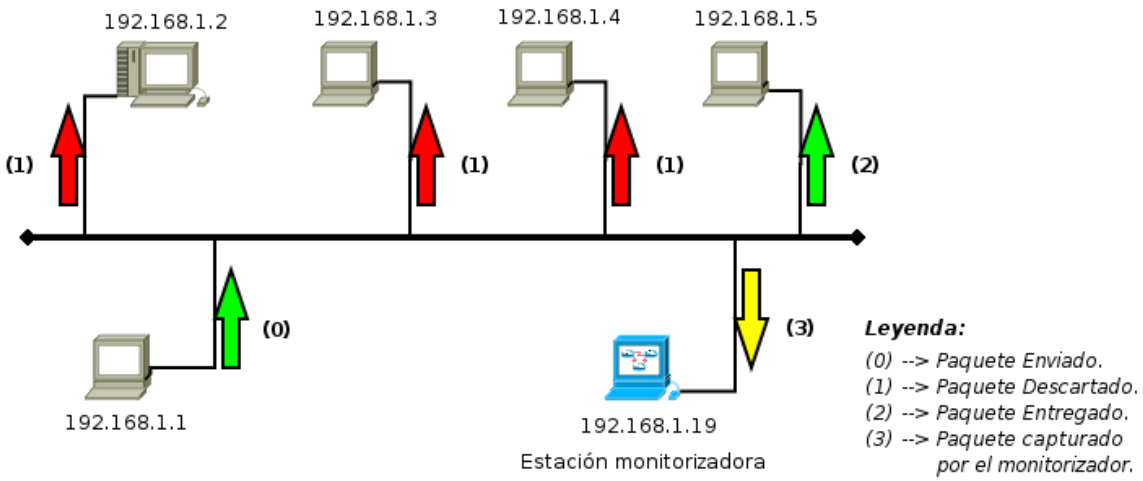


Figura 5.1.- *Red en la que una máquina monitoriza el tráfico que fluye por ésta.*

Cuando una máquina recibe una trama Ethernet, ésta se procesa y se entrega a las capas superiores hasta llegar a la capa de aplicación. A partir de esta capa, la aplicación correspondiente procesa los datos recibidos. En el caso de un monitor de red, la información que éste procesa no sólo son los datos de aplicación, sino todos los datos recibidos incluyendo las cabeceras de las capas de Enlace, Red, Transporte y Aplicación. De esta manera se puede obtener información sobre quien envía datos en una red, de que tipo son, a que puertos van dirigidos, etc.

El funcionamiento básico de un monitorizador programado con las librerías *libpcap* se muestra en la Figura 5.2.

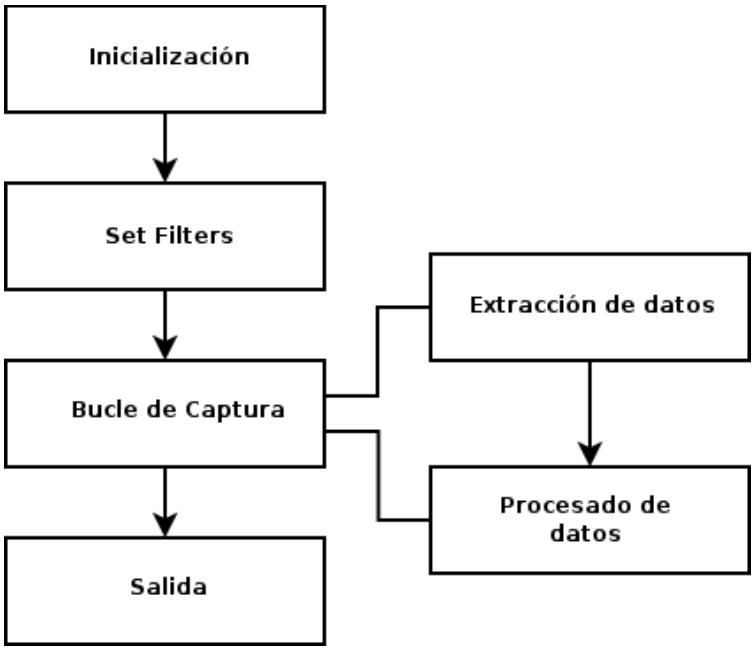


Figura 5.2.- *Bucle de un m onitorizador Libpcap*

Primero se inicializa el programa. En esta fase se asigna una interfaz (tarjeta) de red a monitorizar, y se ejecutan ciertas funciones necesarias para la correcta inicialización del programa. En la fase de activación de filtros, se activan los filtros de captura que permitirán decidir de qué máquinas o redes capturamos tráfico o qué tipo de tráfico deseamos capturar. La tercera fase es la fase de captura, donde se realiza el bucle que escucha la red hasta recibir algún paquete. Cuando se recibe información entramos en la fase de extracción de datos, en la cual se realiza el proceso de extraer la información de las cabeceras y del *payload* (datos útiles) del paquete. Una vez extraída, entramos en la fase de procesado de datos, donde podemos mostrarlos por pantalla o guardarlos en un fichero entre otras funciones. Tras procesar los datos volvemos al bucle del programa, el cual puede acabar si cumple cierta condición, como por ejemplo que se hayan recibido un número de paquetes determinado o que haya pasado cierto tiempo. Una vez se sale del bucle el programa entra en la fase de salida, en la cual finaliza el programa.

5.2.- Inicialización del programa

La fase de inicialización del programa engloba a las funciones capaces de obtener información del sistema: qué interfaces de red instaladas, configuración de estas interfaces (Dirección IP, Máscara de Red), etc.

Las funciones más relevantes de esta fase son las siguientes:

char *pcap_lookupdev(char *errbuff)

Esta función busca el primer dispositivo de red válido para captura y devuelve un puntero a éste. En caso de error devuelve NULL y una descripción del error en la cadena *errbuff*

int pcap_lookupnet(char *device, bpf_u_int32 *netp,bpf_u_int32 *maskp, char *errbuf)

Una vez obtenido el nombre de una interfaz válida podemos consultar su dirección de red (no su dirección IP) y su máscara de subred. *device* es un puntero a una cadena de caracteres que contiene el nombre de una interfaz de red válida. *Netp* y *m askp* son dos punteros a *bpf_u_int32* en los que la función almacenará la dirección de red y la máscara de red respectivamente. Si se produce un error la función devuelve -1 y una descripción del error en la variable *errbuff*.

int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)

Esta función devuelve todas las interfaces de red que pueden ser abiertas para captura de datos. Puede darse el caso de que existan interfaces de red que no puedan ser abiertas (por ejemplo por no tener permisos). Con esta función sólo aparecen las interfaces que han sido abiertas.

Para llamar a esta función es suficiente pasarle un puntero sin inicializar de tipo *pcap_if_t*. En la sección Estructuras de datos usadas por Libpcap hay más información sobre está su descripción detallada. La función transforma ese puntero en una lista enlazada que contendrá cada una de las interfaces y sus datos asociados (dirección y máscara de red). En caso de error la función devuelve -1 y una descripción del error producido en *errbuff*.

int pcap_datalink(pcap_t *p)

Esta función devuelve el protocolo de enlace de datos asociado a una interfaz de red. Existen numerosos valores, cada uno asociado a un tipo de red. Los que nosotros utilizaremos son:

DLT_EN10MB Ethernet (10Mbps, 100Mbps, 1000Mbps)

DLT_IEEE802_11 IEEE 802.11 wireless LAN

5.3.- Filtros de captura

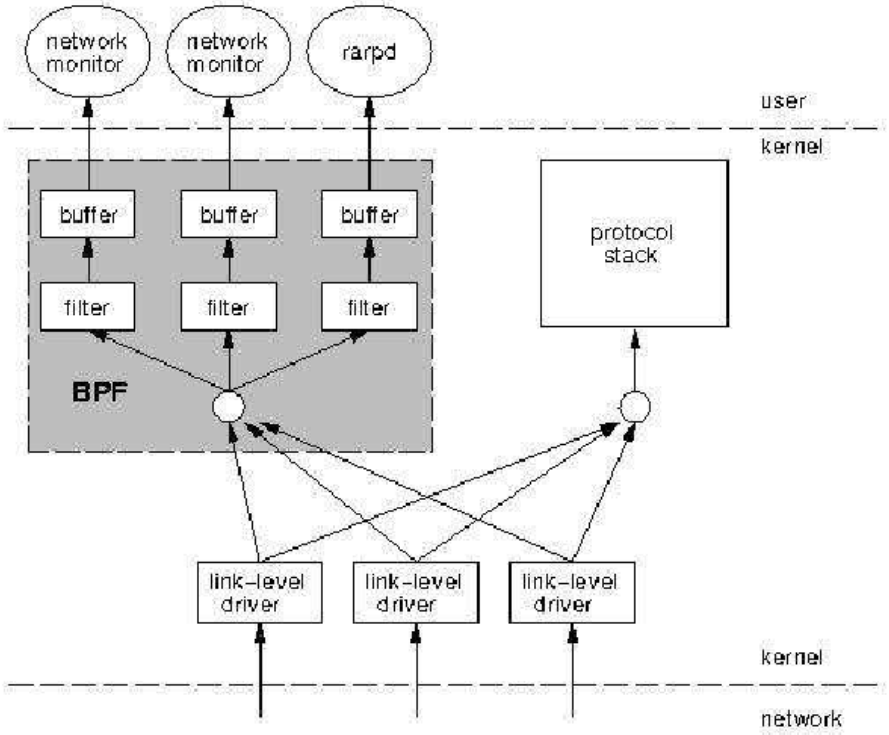
Como se ha comentando anteriormente, libpcap es una librería de funciones que ejecuta un monitor en espacio de usuario (*user space*). Sin embargo la captura de datos debe realizarse en la zona del Kernel (*Kernel space*). Esto significa que es necesario un mecanismo capaz de traspasar esta frontera y este mecanismos debe ser eficiente y seguro ya que cualquier fallo en las capas interiores del espacio del Kernel puede ocasionar una degradación en el rendimiento de sistema.

En el caso de no usar un filtro de captura, todos los paquetes que fueran recibidos traspasarían la frontera del espacio del kernel para llegar al espacio de usuario. Esto es muy ineficiente si solo se desea capturar un cierto tipo de paquetes (P.ej. los que van dirigidos al puerto 21). La solución a esta ineficiencia es establecer un filtro en la zona Kernel que solo deje pasar los paquetes que nos interese capturar. Ésta es la principal labor de un *Packet/Socket Filter*.

Prácticamente cada sistema operativo posee un sistema de filtrado propio. Nosotros nos centraremos en el más extendido en Linux, el BSF (BSD Packet Filter).

El funcionamiento de BPF se basa en dos componentes: el *Network Trap* y el *Packet Filter*. El *Network Tap* se encarga de recopilar los paquetes desde el driver del dispositivo de red y de entregárselos a las aplicaciones a las que estén destinados. El *Packet Filter* tiene por función decidir si el paquete debe ser aceptado o no. En caso afirmativo también se encarga de decidir qué cantidad de información del paquete le entrega a la aplicación. Esto se hace porque no tiene sentido que ésta reciba información de, por ejemplo, las cabeceras Ethernet.

En la Figura 5.3 puede apreciarse el esquema general del funcionamiento de BPF. Cuando la interfaz de red recibe un paquete, el driver de red comprueba que este paquete vaya dirigido a la máquina. En caso afirmativo lo entrega a la pila de protocolos (*protocol stack*), si no es así descartará el paquete. Si cuando se recibe un paquete el BPF se encuentra activo, dicho paquete será procesado por el filtro en vez de ser enviado a la pila de protocolos o descartado.



Fuente: Aprendiendo a programar con Libpcap, A.López Monge.

Figura 5.3.- *Funcionam iento de BPF.*

El BPF se encarga de comparar el paquete con cada uno de los filtros establecidos, entregando una copia de dicho paquete a los *buffers* de las aplicaciones cuyo filtro se ajuste al contenido del paquete. Si no existe ninguna coincidencia filtro-paquete, se devuelve el paquete al *driver*, el cual actuará como si el BPF no estuviera activo. Todo este proceso sucede sin que el paquete salga de la zona Kernel.

Existen procesos que pueden estar interesados en consultar cada uno de los paquetes que circulan por la red, lo que echa por tierra todos los intentos de maximización del rendimiento introducidos por el BPF. Una manera de aumentar la eficiencia es el uso de *buffers*, lo que permite entregar varios paquetes de una sola vez, evitando así cambios continuos de zona. Para mantener la secuencialidad en que se reciben los paquetes, BPF utiliza una marca de tiempo, tamaño y offset a cada uno de los paquetes del grupo.

BPF utiliza su propio lenguaje para la programación de sus filtros. Las librerías Libpcap implementan una interfaz que permite utilizar los filtros BPF mediante un lenguaje más amigable al usuario. Una vez escrito el filtro en el lenguaje Libpcap, estas librerías se encargan de compilar el filtro a lenguaje BPF para que pueda ser aplicado. El lenguaje de alto nivel desarrollado por Libpcap ha

sido popularizado por *TCPdum p* gracias al cual se ha convertido en el estándar para definir filtros de captura.

La expresión utilizada para definir un filtro contiene una serie de primitivas y tres posibles modificadores a las mismas. La expresión puede ser verdadera o falsa. En caso de ser verdadera el paquete se entrega a la zona de usuario, mientras que si es falsa se devuelve al controlador de red. Los tres modificadores posibles son:

tipo: Puede ser *host*, *net* o *port*, indicando respectivamente a una máquina, una red completa o un puerto concreto. Por defecto se asume el tipo host

dir: Especifica el sentido del tráfico que se desea capturar. Las dos opciones son *src* y *dst*. Por defecto se captura el tráfico que circula en ambos sentidos (*src or dst*).

proto: En este caso se especifica el protocolo que se desea capturar. Las opciones son *tcp*, *udp*, *ip*, *ether* (en este caso se capturan tramas a nivel de enlace), *arp* o *rarp* (peticiones *reverse-arp*).

Todas las expresiones anteriores pueden ser combinadas con la ayuda de operadores lógicos: negación (! *not*), concatenación (&& *and*) o adición (|| *or*).

Algunos ejemplos de estas expresiones serían:

- *192.168.1.1 and proto tcp and (dst port 21 or dst port 22)*
- *(dst net 192.168 or dst host 127.0.0.1) and proto udp and port 55500*

Si se desea más información sobre el lenguaje de filtros de libpcap puede consultarse el manual de *tcpdum p.*

$> m an tcpdum p

Las funciones específicas que libpcap proporciona para la fase de filtrado son las siguientes:

int pcap_compile(pcap_t *p, struct bpf_program *fp,char *str, int optimize, bpf_u_int32 netmask)

Esta función se encarga de compilar un programa filtrado en formato *tcpdum p* (*char* str*) en su BPF equivalente (*bpf_u_int32 netm ask*). n*etm ask* es la máscara de red local que puede obtenerse mediante

la función *pcap_lookupnet()*. Si se produce un error la función retorna -1. Para una descripción más detallada de lo sucedido se puede emplear la función *pcap_geterr().*

int pcap_setfilter(pcap_t *p, struct bpf_program *fp)

Esta función se utiliza para aplicar el filtro ya compilado. Para ello hay que pasarle como parámetro a esta función el resultado de compilar el filtro con *pcap_com pile()*. En caso de error la función devuelve -1 y puede obtenerse una descripción más detallada con *pcap_geterr()*.

5.4.- Bucle de captura

Una vez se ha compilado y aplicado el filtro de captura el programa entra en su siguiente fase, el bucle de captura. En esta fase se espera la llegada de algún paquete. Cuando éste llega se activa una función que permite capturarlo.

Durante la fase del bucle es posible utilizar varias funciones que permiten distintos modos de funcionamiento del programa durante esta fase. Las principales diferencias entre estas funciones son el número de paquetes que se quieren capturar, el modo de captura (modo promiscuo o normal) y la manera en que se definen las funciones de llamada o *Callbacks* (la función invocada cada vez que se captura un paquete). A continuación se describen estas funciones con más detalle.

pcap_t *pcap_open_live (char *device, int snaplen, int promisc, int to_ms, char *errbuf)

Para que el programa pueda entrar en el bucle de captura hay que obtener un descriptor de tipo *pcap_t*, por lo cual se utiliza esta función.

El parámetro *char *device* es el nombre del dispositivo de red en el que se realizará la captura. Si este parámetro es entrado con los valores ANY o NULL se fuerza la captura en todos los dispositivos disponibles.

El argumento *int snaplen* indica el número máximo de *bytes* que serán capturados.

El siguiente parámetro, *int prom isc*, indica el modo de captura. Si su valor es 0 se realizará una captura en modo normal, cualquier otro valor indica que la captura se realizará en modo promiscuo.

El parámetro *int to_m s* especifica cuantos milisegundos se quiere que el Kernel agrupe paquetes antes de entregárselos a la zona de usuario. Esto se hace por razones de rendimiento, ya que como se ha explicado antes, realizar esta operación para cada paquete sería muy costoso.

La función devuelve NULL en caso de producirse un error, para el cual puede encontrarse una descripción en el parámetro *errbuf.*

int pcap_dispatch (pcap_t *p, int cnt, pcap_handler callback, u_char *user)

Con esta función se capturan y procesan los paquetes. El parámetro *cnt* especifica el número máximo de paquetes a procesar antes de salir del bucle. Si *cnt vale* -1 se capturará indefinidamente.

El argumento *callback* es un puntero a la función que se invocará para procesar el paquete. El puntero es de tipo *pcap_handler()*, al cual se le pasan los dos siguientes parámetros:

Puntero *u_char*: Aquí es donde se encuentra el paquete. Más adelante se realizará una explicación más detallada de este parámetro.

Estructura *pcap_pkthdr*: Definida con detalle en la sección de Estructuras. La función devuelve el número de paquetes capturados o -1 en caso de producirse un error. Para mostrar un mensaje más detallado del error pueden utilizarse las funciones *pcap_perrorr()* y *pcap_geterr()*.

int pcap_loop (pcap_t *p, int cnt, pcap_handler callback, u_char *user)

Esta función es muy parecida a *pcap_dispatch()*. La principal diferencia reside en que *pcap_loop()* no finaliza cuando se produce un error por *tim eout*. En caso de error la función devuelve un número negativo y 0 si el número de paquetes indicados en *cnt* se ha completado con éxito.

u_char *pcap_next(pcap t *p, struct pcap_pkthdr *h)

Esta función lee un único paquete y devuelve un puntero a *u_char* con su contenido. Esta función es una alternativa a las anteriores. Con esta función no existe la necesidad de declarar ninguna función de *callback*.

5.5.- Extracción de datos

Una vez se captura un paquete se entra en la fase de extracción de datos. En esta fase se tienen que separar datos útiles de cabeceras para que luego puedan ser procesados. Hasta este momento, el paquete que se ha entregado es un conjunto de *bytes* en bruto llamados *RAW bytes*. Para poder obtener información inteligible es necesario realizar la función que la pila de protocolos hubiera hecho si el paquete no hubiera sido capturado por el filtro. Esto significa que es necesario interpretar las cabeceras que tiene el paquete, lo cual no puede realizarse si no se conocen como están estructuradas las cabeceras. En esta sección se explicará de manera genérica algunos de los campos de éstas, pero si se quiere realizar una lectura más profunda de éstas se pueden consultar los RFC y estándares que las definen. Los más importantes son:

RFC 768 (UDP)

RFC 791 (IP)

RFC 792 (ICMPv4)

RFC 793 (TCPv4)

RFC 826 (ARP)

Cuando se recibe un paquete la primera cabecera que se extrae es la de la capa más baja para luego ir subiendo capa a capa. En este caso la primera cabecera a extraer es la de Ethernet.

El primer dato que hay que conocer de la cabecera de una trama Ethernet es su tamaño: 14 bytes. Los campos más importantes de esta cabecera son la dirección ethernet origen (*ether_shost*), la dirección ethernet destino (*ether_dhost*) y el tipo de paquete que porta. Los tres tipos de paquetes más típicos son:

ETHERTYPE_IP (paquete IP)

ETHERTYPE_ARP (paquete tipo ARP)

ETHERTYPE_RARP (paquete de tipo RARP)

En cuanto a las direcciones, Linux proporciona una serie de funciones que permiten transformar direcciones Ethernet de 48 bits a texto legible. Si se desea más información sobre estas funciones puede consultarse el fichero */usr/include/netinet/ether.h*.

Una vez conocida la estructura de la cabecera Ethernet es posible extraerla del paquete *u_char* packet*. Para ello solo hay que desplazarse por el *u_char* packet* 14 *bytes*. El resto de información del paquete dependerá del valor que tenga el campo *ether_type*, por lo que es necesario consultarlo. Nosotros continuaremos el ejemplo suponiendo que el valor del campo *ether_type* es ETHERTYPE_IP, por lo que el paquete será de tipo IP.

En Linux, la cabecera IP se encuentra definida en */usr/include/netinet/ip.h.* Si se consulta este archivo puede observarse que la cabecera IP puede variar de tamaño. La longitud de la cabecera IP se encuentra en el campo *ip_hl*. Otros campos importantes de la cabecera IP son la direcciónes de origen y destino (*saddr* y *daddr* respectivamente) y el protocolo que se encuentra dentro del *payload* de IP (*protocol*). Los protocolos más importantes indicados por este campo son:

ICMP	(*protocol* = *1*)
TCP	(*protocol* = *6*)
UDP	(*protocol* = 17)

Si suponemos que el paquete recibido contenía un datagrama UDP, el valor del campo *protocol* será 17. Para extraer el datagrama tendremos que recorrer *ip_hl bytes* del paquete *u_char* packet*. En este caso obtendremos el datagrama UDP. La cabecera UDP tiene un tamaño fijo de 8 bytes, por lo que si se quiere extraer el *payload* que contiene un datagrama UDP solo hay que avanzar 8 bytes en el paquete *u_char* packet*.

Para el caso de otros protocolos indicados por el campo *protocol* de la cabecera IP el proceso de extracción sería similar. Los pasos a seguir serían los siguientes:

1.- Extracción de la cabecera Ethernet (14 primeros *bytes* del *u_char* packet*).

2.- Consulta del campo *ether_type* de la cabecera Ethernet para saber si el paquete es IP.

3.- Si el paquete es IP, consulta del campo *protocol* de la cabecera IP.

4.- Consulta del archivo indicado por *protocol* (alojado en */usr/include/netinet/*)

5.- Consulta del tamaño de la cabecera del protocolo. Si existe *payload*, éste viene a continuación de la cabecera del protocol.

5.6.- Procesado de datos

En la fase de procesado de datos ya se dispone, tanto de las cabeceras del paquete recibido como del paquete en sí. Con estos datos se pueden realizar las funciones que se deseen, desde realizar estadísticas del tráfico que fluye por la red hasta espiar la información de los usuarios de ésta. Al ser tan extensas las posibles aplicaciones existentes, éstas no se tratarán en este apartado. Sin embargo si que se explicarán las funciones que ofrece Libpcap para el almacenamiento de estos datos en ficheros y el análisis de éstos.

pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname)

Esta función se utiliza para abrir un fichero en el que se guardarán los datos que sean capturados. Si no hay problemas la función abrirá el fichero *char* fnam e* en modo escritura y devolverá un puntero a un descriptor de tipo *pcap_dum per_t*. En este descriptor es donde se podrán volcar los datos de la captura. Si se produce un error la función retornará NULL. Mediante la función *pcap_geterr()* se puede ver una descripción de dicho error.

Las siguientes funciones se ejecutan una vez se ha realizado la captura.

pcap_t pcap_open_offline(char* fname, char *errbuf))

Esta función permite abrir un fichero en modo lectura cuyo contenido es una captura de paquetes en formato tcp*dum p. fnam e* indica la ruta del fichero. La función devuelve NULL en caso de error. Para consultar éste se usa *errbuf*.

void pcap_dump_close(pcap_dumper_t *p)

Si se ha terminado con el fichero de salida, éste puede cerrarse con la función *pcap_dum p_close.*

Capítulo 6. Aplicación desarrollada

Uno de los objetivos de este proyecto ha sido la realización de una aplicación que demostrara la capacidad del sistema operativo Linux para añadir inteligencia a los sistemas empotrados. Linux es un sistema operativo que permite mejorar las prestaciones de un equipo gracias a la inteligencia que proporciona.

La intención para esta sección era instalar una distribución de Linux en un sistema empotrado y luego, debido a que Linux está publicado bajo la licencia GPL, realizar alguna modificación que mejorara sus prestaciones. De esta manera se comprobaría la capacidad que tiene el sistema para mejorar sus prestaciones con un esfuerzo mínimo, tanto temporalmente como económicamente.

El proceso de desarrollo se dividió en tres partes: elección del sistema empotrado, instalación de una distribución Linux y desarrollo de la aplicación. Si bien las dos primeras partes no eran propiamente desarrollo, si produjeron una gran influencia en la aplicación a desarrollar. Esto fue especialmente cierto en la elección del sistema. Dependiendo del tipo de sistema empotrado a elegir (móvil, PDA, *router*) se elegiría una aplicación u otra.

Debido a que se tenía una ligera experiencia en el manejo de *routers*, se eligió como entorno de trabajo un *router.* Esto implicaba que la aplicación a desarrollar tendría que ver con las redes de comunicaciones. Finalmente se optó por realizar un monitorizador de tráfico que capturara tan sólo las cabeceras de los paquetes que circularan por la red. Este monitorizador podría ser utilizado, por ejemplo durante un examen de programación para comprobar que los alumnos no se están comunicando entre ellos o accediendo a sitios de Internet para conseguir información que les ayudara a responder a las preguntas del examen.

Como reto se intentaría que esta aplicación estuviera integrada en el *Kernel* de Linux. Con esto se conseguiría un sistema lo más eficiente posible.

En este capítulo, antes de analizar la aplicación en sí, se analizará la plataforma elegida, el *router* Linksys WRT54GL, un *router* famoso entre las comunidades *wireless* porque permite la instalación de distribuciones Linux, lo que mejora ostensiblemente sus prestaciones. Tras analizar el *router*, y la distribución elegida para éste, se explicará la aplicación desarrollada en sí.

6.1.- El router Linksys WRT54GL

Como el alumno tuvo contacto con *routers* inalámbricos y Linux durante sus prácticas en empresa, se consideró interesante ampliar su formación eligiendo un *router* para la realización de la aplicación. Por lo tanto la plataforma donde realizar la aplicación tendría que ser un *router* inalámbrico y que soportara alguna distribución de Linux en su interior. Además, se tendría en cuenta que se hubieran desarrollado varias distribuciones de Linux para el producto elegido y éstas tuvieran soporte por parte de la comunidad. Esto era para asegurar que se trabajara con la distribución más adecuada a las necesidades del proyecto. No serviría, por ejemplo, una distribución que no tuviera información actualizada.

El departamento de Arquitectura de Computadores de la UPC, disponía de un presupuesto de 100€ para la compra de equipamiento de este TFC. Este presupuesto reducía sensiblemente las posibilidades de elección. Sin embargo, todavía quedaban un buen número de routers inalámbricos que soportaran Linux en su interior. Todos los *routers* que se encajaban en la descripción se encuentran en la tabla siguiente. Se proporcionan también sus principales características fisicas y su precio. En la segunda tabla se muestra la o las distribuciones que soportan el *router* y el *Kernel* sobre el que corre su distribución. Se buscaron también distribuciones con licencia GPL.

	Arquitectura	CPU	RAM	Almacenamiento	Precio (sin IVA)
D-Link DSL-G604T	MIPS	150MHz	16MB	Flash 4MB	90,00 €
Linksys WRT54GL	MIPS	200MHz	16MB	Flash 4MB	63,00 €
3Com 3CR860-95	MIPS	133MHz	32MB	Flash 2x8MB	120,00 €
Asus WL-500g	MIPS	125MHz	16MB	Flash 4MB	86,00 €
Asus WL-300g	MIPS	125MHz	16MB	Flash 4MB	79,00 €
Asus WL-HDD	MIPS	125MHz	16MB	Flash 4MB	73,00 €

	Kernel	Distribución	Libre
D-Link DSL-G604T	2.4.17	Monta Vista	GPL
Linksys WRT54G	2.4.5	OpenWRT, Ddrt, varias	GPL
3Com 3CR860-95	2.4.17	Debian, OpenWRT	GPL
Asus WL-500g	2.4/2.6	Varias	GPL
Asus WL-300g	Idem	Varias	GPL
Asus WL-HDD	Idem	Varias	GPL

Tabla 2: Características de los diferentes routers.

La elección final fue el router *Linksys* WRT54GL. Los motivos principales de la elección fueron tres: el precio, las prestaciones del *hardware* y el soporte ofrecido por la comunidad. Como puede verse, el *router* Linksys WRT54GL era el que menor precio tenía, y que mejores

prestaciones ofrecía19. Además, éste era el *router* con más distribuciones Linux que hay actualmente en el mercado.

La distribución elegida para el *router* fue OpenWRT20, ya que ésta disponía de una extensa documentación en su página web *wiki21*. OpenWRT es una distribución GNU/Linux que utiliza BusyBox22 como paquete principal. Según sus propios desarrolladores, "*Busybox es la navaja suiza de los Linux em potrados".* Busybox es una recopilación de prácticamente todas las aplicaciones de una distribución Linux convencional pero en un sólo archivo de muy poco tamaño. Concretamente, en el *router* ocupa 538KB. Este paquete es ampliamente utilizado en distribuciones Linux en sistemas empotrados de prestaciones medias o bajas. Gracias a este paquete, se puede trabajar en el *router* como si se estuviera en la *shell* de un sistema Linux convencional.

La distribución OpenWRT contiene también un sistema de ficheros específico para memorias flash. El sistema de ficheros es JFFS223, que aporta compresión de datos hasta el momento de su lectura. De esta manera se consigue un mayor ahorro de espacio en el sistema.

Otra de las características de la distribución es la inclusión de una interfaz web que permite la configuración del dispositivo. Esta característica, sin embargo, no es interesante para el proyecto, pues no será necesario acceder a sus parámetros de configuración para el desarrollo de la aplicación.

El *router* Linksys incorpora de serie un software muy limitado en comparación con el de OpenWRT. A pesar de ello, su *bootloader* permite arrancar y permite cargar versiones más actuales mediante una configuración enlazada. El enlace físico es un cable Ethernet. El envío de datos al *router (*el *target* en esta configuración enlazada) se realiza mediante el protocolo TFTP.

Para instalar una distribución en el *router*, es necesario bajarse a nuestro *host* la distribución que queremos instalar. En la página web de los desarrolladores podemos encontrar el archivo *.bin que contiene la distribución. Una vez descargada, hay que cargar este archivo en el *router* mediante el protocolo TFTP. Una vez hecho esto ya se tiene el *target* actualizado con una distribución Linux corriendo en su interior.

En el caso de la distribución OpenWRT, los pasos a seguir para actualizar la versión del sistema operativo del *router* fueron los siguientes:

[19]El router *3Com 3CR860-95* ofrecía mejores prestaciones que el modelo de Linksys y tenía un excelente soporte por parte de la comunidad desarrolladora. Sin embargo, excedía el presupuesto del departamento. Por esta razón fue descartado..
[20]http://openwrt.org/
[21]http://wiki.openwrt.org/
[22]http://en.wikipedia.org/wiki/Busybox
[23]Journaled Flash File System. http://en.wikipedia.org/wiki/Jffs2

- Primero se descarga la distribución al *host*. La página de descarga es la siguiente: http://downloads.openwrt.org/whiterussian/newest/
- A continuación se desenchufa el *router (target).*
- El siguiente paso es arrancar el cliente TFTP en nuestro *host*.
    - Se abre una conexión con el *router.* Normalmente la dirección de éste es 192.168.1.1
    - Se activa el modo binario (*octet*).
    - Se le ordena al cliente TFTP que reenvíe el archivo hasta que tenga éxito.
    - Subir el archivo.
- Encender el *router.* En este momento se tiene el cliente TFTP intentando constantemente establecer la conexión. Lo primero que realiza el bootloader del router es comprobar si alguien está intentando establecer conexión por el cable Ethernet. Como es así, el bootloader descarga la distribución mediante TFTP. Una vez descargada, el *bootloader* arranca la distribución descargada.
- En este momento el router ya tiene la distribución OpenWRT instalada y arrancada, por lo que podemos acceder a él vía *ssh24.*

Los comandos exactos del cliente TFTP son los siguientes:

```
atftp
connect 192.168.1.1
mode octet
trace
timeout 1
put openwrt-xxx-x.x-xxx.bin
```

6.2.- La aplicación

El objetivo de desarrollar una aplicación en un sistema empotrado era demostrar que éstos pueden mejorar sus prestaciones, o conseguir características de las que antes no disponían. Para ello se propuso realizar una aplicación monitorizadora que sólo capturara cabeceras de paquetes. El motivo de monitorizar sólo las cabeceras de los paquetes y no toda su información es por razones temporales. No era necesario tener que realizar una aplicación que decodificara todos los campos de las cabeceras para descubrir que protocolos

[24]Secure Shell Protocol (SSH). http://en.wikipedia.org/wiki/Ssh

viajan dentro de cada paquete. Sólo monitorizando cabeceras ya se le habría añadido una funcionalidad extra al *router* que antes no tenía.

De esta manera se desarrollaría una aplicación que con el *router* monitorizara el tráfico que hay en una red. Por ejemplo en una clase durante un examen. Se propuso el reto de intentar integrar esta aplicación en el propio *kernel* de Linux. Al incrustar la aplicación en el *kernel* no sólo se demostraría que es posible añadir funcionalidades extras, sino que es posible modificar el sistema en sí, adaptándolo de una manera más optimizada a las necesidades de la aplicación. En el caso del *router,* realmente no es necesario un grado tan alto de optimización, pero podría darse el caso de tener un sistema empotrado que si requiriera de dicha optimización. Por ejemplo porque el sistema tiene menos prestaciones.

Durante el desarrollo de la aplicación surgieron diversos problemas como la diferencia entre el código descrito en la bibliografía estudiada por el alumno y el código real. Numerosas funciones cambiaban de nombre y lugar, principalmente debido al cambio de la estructura entre los *kernels* 1.2 de la bibliografía y el *kernel* 2.4 del *router*. Principalmente las estructuras teórica y práctica eran muy parecidas, pero las diferencias fueron suficientes como para que surgieran numerosos errores de compilación.

Estos retrasos llevaron a cancelar el intento de realizar la aplicación a nivel de *kernel* e intentar seguir con la misma aplicación pero a un nivel más alto. Desde un principio se consideró importante cumplir el objetivo de demostrar que se podían mejorar o ampliar las funcionalidades del *router.* Conseguir este objetivo a nivel de *kernel* era sólo un objetivo secundario, por lo que se decidió centrarse en cumplir el objetivo principal.

Una vez tomada esta decisión, se decidió continuar con la misma aplicación, pero haciendo uso de las librerías *Libpcap.* El programa se desarrolló y probó primero en el portátil de alumno. Tras comprobar su funcionamiento adecuado en el portátil, se procedió a cross-compilar la aplicación con el SDK25 de OpenWRT. Esta plataforma de desarrollo fue obtenida de la página de descargas del proyecto OpenWRT.

El uso de librerías dinámicas produjo algunos problemas que no se produjeron en el portátil, por lo que se optó por compilar el programa utilizando librerías estáticas. Esto implica que el programa ocupa más espacio que si es compilado dinámicamente, ya que debe incluir todas las librerías que utiliza. Afortunadamente el *router* disponía de memoria suficiente para el programa, por lo que se pudo subir la versión compilada estáticamente y lograr que funcionara sin ningún problema.

El programa captura las cabeceras de paquetes y muestra los siguientes datos:

[25]Software Development Kit. http://en.wikipedia.org/wiki/SDK

- Número de paquete capturado.
- Dirección MAC Origen.
- Dirección MAC Destino.
- Tipo de protocolo de nivel 3.
- TTL del paquete IP (si es IP)
- Dirección IP Origen.
- Dirección IP Destino
- Protocolo de nivel de transporte que contiene el paquete IP

A continuación puede verse una captura del programa en funcionamiento.

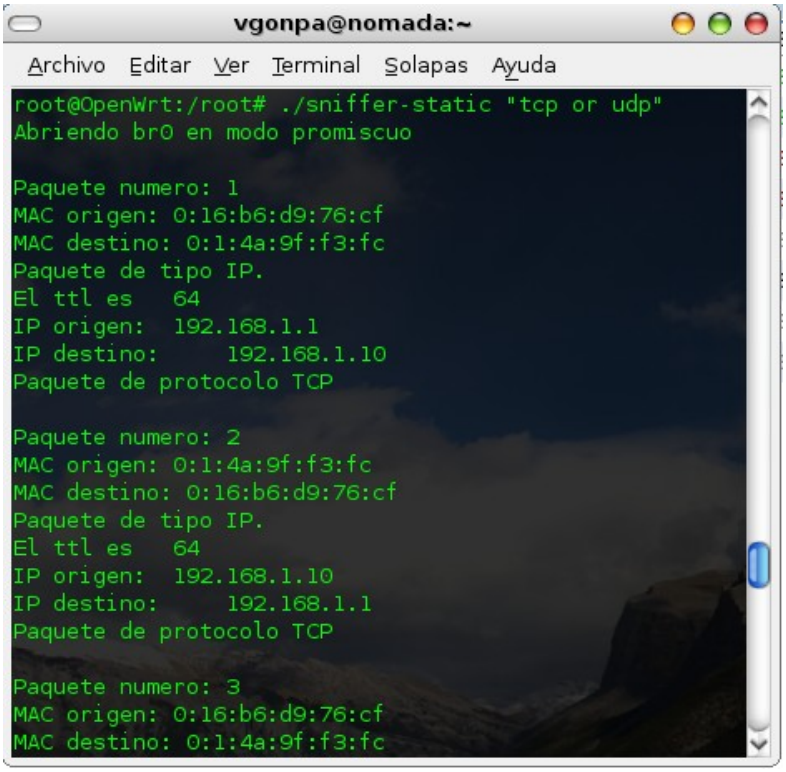


Tabla 3: La aplicación en funcionam iento.

Capítulo 7. Conclusiones

Tras la implementación de la aplicación, se evalúan los resultados obtenidos. Se valorará qué objetivos se han cumplido y cuales no, así como los motivos que han impedido conseguirlos. Del mismo modo se valorará si las tecnologías utilizadas han ofrecido los resultados esperados y en caso de no ser así se determinarán algunas acciones que podrían tomarse para mejorar su rendimiento. Finalmente se plantearán posibles mejoras y otras líneas que han quedado abiertas en este proyecto.

7.1.- Valoraciones de Linux, empotrados y proyecto.

El sistema operativo Linux se ha mostrado como una herramienta muy potente y con una gran capacidad de mejora de cara al futuro. El hecho de que exista una gran comunidad desarrolladora tras este sistema implica que exista una gran documentación Linux. De esta manera, Linux no es una caja negra de la que no se conoce su interior ni su funcionamiento. Ha sido relativamente sencillo conocer sus mecanismos internos, lo que implica que es muy sencillo conocer las posibilidades que ofrece. Esto también lleva a que al conocer un sistema internamente se puede maximizar el rendimiento al trabajar con éste.

Gracias a la GPL, esta comunidad también genera un gran número de aplicaciones libres que permiten la reutilización de software. No es necesario desarrollar algo que ya existe y además no es necesario pagar por una licencia de uso, sencillamente hay que descargarlo y utilizarlo. Durante la realización del proyecto se ha podido comprobar este hecho. La aplicación no salía adelante, pero el uso de las librerías *Libpcap* permitió que se pudiera finalizar la aplicación con éxito.

Si añadimos a esto que la licencia GPL permite la modificación del software, las posibilidades se multiplican. Es posible modificar una aplicación para adaptarla a nuestras necesidades. Esto último es especialmente importante en el caso de los sistemas empotrados, donde los recursos del sistema suelen ser escasos. Un claro ejemplo de esto es la posibilidad de realizar nuestra aplicación a nivel de Kernel. A pesar de no haberse conseguido, es posible realizar esto, por lo que se minimizaría el tiempo de CPU.

Los sistemas empotrados son, hoy en día, sistemas diseñados para realizar tareas muy específicas. Sin embargo, el aumento de la tecnología permite estos sistemas aumenten sus prestaciones. Esto permite una mayor integración de sistemas, con lo cual un mismo sistema ofrece más funcionalidades.

Una posibilidad interesante para estos sistemas es que los fabricantes publiquen sus especificaciones. Al igual que en el caso de Linux, esto permitiría que los consumidores exprimieran al máximo las posibilidades del sistema que han comprado. Un ejemplo es el *router* que se ha utilizado durante el proyecto. Utilizar un sistema Linux empotrado, ha permitido que los usuarios sean capaces de conocer las especificaciones internas del *router.* Esto ha hecho posible que los *routers* de Linksys, que utilizan Linux, por un valor de 60€ realicen las mismas tareas que *routers* valorados en 600€[26]. Que a la empresa le interese publicar o no sus especificaciones es otro asunto, pero desde mi punto de vista, creo que beneficiaría al consumidor.

Mi valoración personal del proyecto ha sido extremadamente satisfactoria. He podido conocer mejor el entorno de los sistemas empotrados así como el funcionamiento de Linux. Sobre este sistema operativo he aumentado considerablemente mis conocimientos.

Aprendí mucho sobre el funcionamiento interno del *Kernel*, un gran descubrimiento para mí. Pude comprobar que no es tan lejano y complicado como parece.

Durante mi asistencia a la conferencia de Alejandro Lucero sobre Linux en Sistemas empotrados en Madrid, descubrí muchas de las razones por las que utilizar Linux desde el punto de vista empresarial. Pude comprobar como Linux no es sólo un sistema operativo utilizado por poca gente, sino que se está convirtiendo en una verdadera alternativa, muy atractiva, para el uso empresarial. Éste era un campo que al ser totalmente desconocido para mí, me pareció sumamente interesante.

7.2.- Objetivos cumplidos

Los objetivos iniciales del proyecto eran, primero construir una aplicación que añadiera funcionalidades a un sistema empotrado y, segundo, que esta aplicación funcionara a nivel de *Kernel*.

El primer objetivo ha sido cumplido satisfactoriamente. Se ha desarrollado una aplicación que permite monitorizar las cabeceras de los paquetes que circulan por una red. Esta funcionalidad no la tenía el sistema empotrado elegido, el *router* Linksys WRT54GL. De esta manera se ha demostrado como Linux permite añadir funcionalidades a un sistema con un propósito muy específico.

El segundo objetivo, lamentablemente no se ha cumplido. La razón principal por la que no se ha conseguido implementar la aplicación a nivel de *Kernel* es

[26]http://www.lifehacker.com/software/router/hack-attack-turn-your-60-router-into-a-600-router-178132.php

por cumplir la planificación dada. De esta manera no se ha demostrado que no se pueda realizar esta tarea. Si el tiempo para desarrollar esta tarea hubiera sido mayor, seguramente se podría haber completado con éxito. Sin embargo, al no ser el objetivo principal del proyecto se prefirió dejarla en un puesto secundario, para que no interfiriera con el primer objetivo del proyecto.

7.3.- Seguimiento planificación

La planificación inicial del proyecto tenía previsto que éste durara entre el 27 de febrero y el 10 de julio de 2006. Al pasar los meses, diversos factores han intervenido a que esta planificación tuviera que ser modificada.

El primer condicionante fueron los cambios en las fechas de las reuniones debido a los numerosos imprevistos que obligaron a adelantar o retrasar algunas reuniones quincenales. Otro condicionante fue el hecho de que la planificación inicial era aproximada y sobre unas tareas desconocidas. Muchas tareas tardaron tiempos diferentes de lo esperado.

Diversos imprevistos técnicos como la falta de un equipo donde instalar la distribución Gentoo Linux provocaron un retraso en la realización de la Fase 2. Por otro lado, esta fase duró más de lo esperado ya que se realizó conjuntamente con el estudiante Carles Delgado. La dificultad para cuadrar los horarios de los dos alumnos hizo que esta fase se fuera retrasando.

7.4.- Mejoras y líneas futuras

La realización de este proyecto ha dejado diversas líneas abiertas. Una o varias de estas líneas podrían ser el punto de partida de un nuevo proyecto.

La primera de ellas sería añadir más protocolos analizables por la aplicación. Es posible incluir la información de todos los campos de las diferentes cabeceras de los paquetes. También se podrían procesar datos de los protocolos de aplicación, los cuales no se procesan en esta versión de la aplicación.

También queda abierto que la aplicación esté integrada en el *kernel*. Si bien esto no se ha realizado aún, se cree que es posible. Al funcionar a este nivel, la aplicación funcionaría de una manera más eficiente, de esta manera se podría utilizar en dispositivos con menos recursos.

La tercera vía de desarrollo sería demostrar que es posible portar la aplicación a otras distribuciones e incluso a otros sistemas empotrados. De momento la aplicación ha funcionado correctamente en el portátil del alumno, con un *Kernel* 2.6.16 y en el *router* que utiliza un *Kernel* 2.4.5.

Bibliografía

[1] M. Beck, H. Böme, M. Dziadzka, U Kunitz, R. Magnus, D. Verworner en *Linux Kernel Internals 2nd Ed.*, Addison-Wesley Eds., pp. 1-379, Harlow (1997).

[2] K. Yaghmour, en *Building Em bedded Linux System s.* Ed. O'Reilly pp. 23-94 (2003).

[3] A. S Tanenbaum en *Com puter Networks 4th Edition.* Ed. Prentice Hall pp. 21-45 , New Jersey (2003).

[4] A. Lucero en *Sem inario de Linux en Sistem as Em potrados.* 26-IV-2006 Escuela Politécnica Superior, UAM, Madrid.

[5] The Wikipedia Editors. *Wikipedia, varios artículos*. Varias fechas. URL<http://en.wikipedia.org/wiki/Main_Page>

[6] OpenWRT developers. *OpenWRT wiki*. Varias fechas. URL<http://wiki.openwrt.org/>

Repercusiones medioambientales

En este TFC se pueden comentar los siguientes aspectos del uso de sistemas empotrados y Linux:

Linux añade inteligencia a los sistemas empotrados. Esta inteligencia permite que los sistemas empotrados realicen funcionalidades extra. Al añadir nuevas funcionalidades mediante el uso de software, se aprovechan mejor los recursos *hardware.* Un claro ejemplo de integración mediante software es el caso de los *sm artphones.* Estos dispsitivos, mediante el uso de diversas aplicaciones, consiguen tener las funcionalidades de una PDA y un teléfono móvil. Si bien no todos los *sm artphones* utilizan Linux en su interior si que permiten asegurar que el uso de inteligencia en *software* permite un ahorro en *hardware.*

Por otra parte, la licencia GPL permite que un programador adapte una aplicación a sus necesidades. Por ejemplo, el hecho de incluir la aplicación monitorizadora en el *Kernel* de Linux permite que ésta consuma menos recursos del sistema. Al consumir menos CPU, se está consumiendo menos energía. Esto implica que un buen uso del software puede repercutir, indirectamente, en un mejor aprovechamiento de los recursos energéticos.

ANEXO

7.5.- Principales Estructuras

A continuación se detallan las principales estructuras utilizadas por las funciones de las librerías Libpcap. Estas estructuras de datos se encuentran en el fichero */usr/include/pcap.h*.

Informacion de Interfaces

```
/*
    La llamada a pcap_findalldevs, construye una lista enlazada de pcap_if
(pcap_if=pcap_if t), en la cual se recoge toda la información de cada una de las
interfaces de red instaladas.
*/
struct pcap_if {
        struct pcap_if *next;                  // enlace a la siguiente definición de
interfaz
        char *name;                           // nombre de la interfaz (eth0,wlan0. . .)
        char *description;                          // descripción de la interfaz o
NULL
        struct pcap_addr *addresses;          // lista enlazada de direcciones
asociadas a esta interfaz
        u_int flags;                          // PCAP_IF_ interface flags
};

/*
    Una interfaz puede tener varias direcciones, para contenerlas todas se crear
una lista con un pcap_addr por cada una
*/

struct pcap_addr {
        struct pcap_addr *next;               // enlace a la siguiente dirección
        struct sockaddr *addr;                      // dirección
        struct sockaddr *netmask;             // máscara de red
```

```
        struct sockaddr *broadaddr;                    // dirección de broadcast
para esa dirección
        struct sockaddr *dstaddr;                  // dirección de destino
};
```

Estadísticas

```
/*
     Mediante la llamada a pcap_stats obtenemos la siguiente estructura
información estadística
*/

struct pcap_stat {
        u_int ps_recv;          // numero de paquetes recibidos
        u_int ps_drop;          // numero de paquetes descartados
        u_int ps_ifdrop;        // descartes por cada interfaz (aun no soportado)
};
```

Paquetes

```
/*
  Cada paquete va precedido por esta cabecera genérica
*/

struct pcap_pkthdr {
        struct timeval_ts;              // time stamp (marca de tiempo)
        bpf_u_int32 caplen;             // tamaño del paquete al ser capturado
        bpf_u_int32 len;                        // tamaño real del paquete en el fichero;
};

struct pcap_file_header {
        bpf_u_int32 magic;
        u_short version_major;
        u_ short version_minor;
        bpf_int32 thiszone;                     // corrección de GMT a local
        bpf_u_int32 sigfigs;                    // precisión de los timestamps
```

```
    bpf_u_int32 snaplen;        // tamaño máximo salvado de cada paquete
    bpf_u_int32 linktype;       // tipo de DataLink
};
```

La estructura pcap_t

La estructura pcap_t que aparece en muchas funciones, es una estructura opaca para el usuario. Por lo tanto no es necesario conocer sus contenidos, basta con saber que ahí Libpcap guarda sus variables internas.