

Internet, un enfoque práctico

David Villa@uclm.es

9 de febrero de 2015

Índice general

Índice general	III
Índice de cuadros	IX
Índice de figuras	XI
Índice de listados	XIII
Prefacio	XVII
1. Shell	1
1.1. GNU (GNU is Not Unix) Bash	2
1.2. Valor de retorno	2
1.3. Gestión básica de procesos	3
1.4. Gestión de trabajos	4
1.5. Entrada, salida y salida de error estándar	5
1.6. Redirección	6
1.7. Catálogo de comandos	8
1.7.1. Ficheros y directorios	8
1.7.2. Sistema	10
1.7.3. Procesos	10
1.7.4. Usuarios y permisos	10
2. Conectividad	13
2.1. Modelos de referencia	13
2.2. Conectividad, por capa	14
2.3. Configuración básica	15
2.4. Interfaces de red	16
2.4.1. <i>Loopback</i>	16
2.4.2. <i>eth0</i>	17
2.4.3. <i>iproute2</i>	18

2.5.	Encaminamiento básico	18
2.6.	Conectividad a nivel de enlace	18
2.7.	Conectividad a nivel de red	20
2.7.1.	ping	20
2.7.2.	ping «extendido»	21
2.7.3.	Otros <i>ping</i>	22
2.7.4.	traceroute	22
2.7.5.	mtr	23
2.8.	Conectividad a nivel de transporte	23
2.8.1.	netcat	23
2.9.	Conectividad a nivel de aplicación	24
3.	Pila de protocolos TCP/IP	27
3.1.	Protocolos esenciales	28
3.1.1.	Ethernet	28
3.1.2.	IP	30
3.1.3.	UDP	32
3.1.4.	TCP	32
3.2.	Encaminamiento	32
3.2.1.	Entrega directa	32
3.2.2.	Entrega indirecta	32
3.2.3.	Encaminador por defecto	32
3.3.	Ejercicios	32
3.3.1.	Enlace	32
3.3.2.	Red	33
3.3.3.	Transporte	33
4.	Captura y análisis de tráfico de red	35
4.1.	tshark	35
4.1.1.	Advertencia de seguridad	36
4.1.2.	Limitando la captura	36
4.1.3.	Filtros de captura	37
4.1.4.	Formato de salida	37
4.1.5.	Filtros de visualización	39
4.1.6.	Generación de estadísticas	40
4.2.	Respuestas	44

5. Encapsulación de protocolos	45
5.1. Objetivos	45
5.2. Entorno	45
5.3. Situación inicial	45
5.4. Wireshark	46
5.4.1. Captura de paquetes	46
5.4.2. Interpretando de resultados	48
5.4.3. Filtrado	49
5.4.4. Seguir un stream	50
6. El lenguaje Python	53
6.1. Empezamos	53
6.2. Variables y tipos	53
6.3. Variables y tipos	54
6.3.1. Valor nulo	54
6.3.2. Booleanos	54
6.3.3. Numéricos	54
6.3.4. Secuencias	55
6.3.5. Orientado a objetos	56
6.4. Módulos	56
6.5. Estructuras de control	56
6.6. Indentación estricta	57
6.7. Funciones	57
6.8. Python is different	57
6.9. Hacer un fichero ejecutable	58
7. Codificación y representación de datos	59
7.1. Representación, sólo eso	59
7.2. Los enteros de Python	60
7.3. Caracteres	60
7.4. Tipos multibyte y ordenamiento	62
7.5. Cadenas de caracteres y secuencias de bytes	64
7.6. Empaquetado	65
7.7. Desempaquetado	67
8. Sockets BSD	69
8.1. Sockets	69
8.1.1. Historia	69

8.2.	Creación de un socket	70
8.2.1.	IPC	71
8.3.	Uso del socket	71
8.3.1.	Datos binarios	73
8.4.	Desconexión	73
8.4.1.	Cuando los sockets mueren	74
8.5.	Sockets no bloqueantes	74
8.5.1.	Rendimiento	75
8.6.	Créditos	76
9.	Modelo Cliente-Servidor	77
9.1.	Chat	78
9.1.1.	Paso 1: Mensaje unidireccional	78
9.1.2.	Paso 2: Lo educado es responder	80
9.1.3.	Paso 3: Libertad de expresión	81
9.1.4.	Paso 4: Habla cuando quieras	82
9.1.5.	Paso 5: Todo en uno	84
9.2.	Ejemplos rápidos de sockets	85
9.2.1.	Un cliente HTTP básico	85
9.2.2.	Un servidor HTTP	85
10.	Puertos y servicios	87
10.1.	netstat	87
10.1.1.	Visualizar la tabla de rutas	88
10.1.2.	Lista de interfaces de red	88
10.1.3.	Listar servidores	89
10.1.4.	Filtrar listado de sockets	89
10.1.5.	Otras opciones	89
10.2.	nmap	89
10.3.	IPTráf	90
10.4.	Referencias	92
11.	Sockets RAW	95
11.1.	Acceso privilegiado	96
11.2.	Tipos	96
11.3.	Sockets AF_PACKET:SOCK_RAW	97
11.3.1.	Construir y enviar tramas	98
11.3.2.	Implementando un arpíng	99

11.4. Sockets AF_INET:SOCK_RAW	101
11.4.1. Generando mensajes	102
11.4.2. Enviando	102
11.5. Ejercicios	103
12. Filtrado de paquetes	107
12.1. Introducción	107
12.2. Tablas y reglas	107
12.3. Políticas	108
12.4. Comandos básicos	108
12.4.1. Ver la configuración	109
12.4.2. Borrar todas las reglas de una tabla	109
12.5. Configuración de un router (con esquema 3)	109
12.5.1. Bloquear un puerto (en esquema 1)	110
12.5.2. Redireccionar un puerto del router hacia otro host (in- terno o externo)	110
12.5.3. Guardar la configuración de iptables	110
13. Escaneo de servidores y servicios	111
13.1. Descubrimiento de hosts	111
13.2. Escaneo de puertos	114
13.3. Identificación de servicios	116
13.4. Identificación del Sistema Operativo	116
A. Netcat	119
A.1. Sintaxis	119
A.2. Ejemplos	119
A.2.1. Un chat para dos	119
A.2.2. Transferencia de ficheros	120
A.2.3. Servidor de echo	120
A.2.4. Servidor de daytime	120
A.2.5. Shell remota estilo telnet	120
A.2.6. Telnet inverso	121
A.2.7. Cliente de IRC	121
A.2.8. Cliente de correo SMTP	121
A.2.9. HTTP	122
A.2.10. Streaming de audio	122
A.2.11. Streaming de video	122

A.2.12. Proxy	122
A.2.13. Clonar un disco a través de la red	123
A.2.14. Ratón remoto	123
A.2.15. Medir el ancho de banda	123
A.2.16. Imprimir un documento en formato PostScript	124
A.2.17. Ver «La Guerra de las Galaxias»	124
A.3. Otros «netcat»s	124

Referencias**125**

Índice de cuadros

4.1. tshark: ejemplos de filtros de captura	37
4.2. tshark: ejemplos de filtros de visualización	40
7.1. struct: especificación de ordenamiento	66
7.2. struct: especificación de formato	66

Índice de figuras

2.1.	Correspondencia entre los diferentes modelos de referencia . .	15
2.2.	Conector RJ-45 hembra de un equipo Ethernet	19
2.3.	Aspecto del programa mtr	23
2.4.	<i>Chromium</i> muestra la «página web» que devuelve ncat	25
3.1.	Pila de protocolos TCP/IP (Arquitectura de protocolos de Internet)	27
3.2.	Formato de la trama Ethernet	29
3.3.	Formato de la dirección MAC (Media Access Control) Ethernet [Fuente:Wikimedia Commons]	30
3.4.	Formato de las direcciones IP (Internet Protocol)	31
3.5.	Formato del paquete IP	31
5.1.	Situación inicial	46
5.2.	Ventana inicial	47
5.3.	Opciones de captura	47
5.4.	Ventana principal	48
5.5.	Wireshark: siguiendo un stream	50
7.1.	Ordenación de bytes	63
9.1.	Modelo Cliente/Servidor [Fuente:Wikimedia Commons] . . .	77
10.1.	91
10.2.	91
10.3.	92

Índice de listados

4.1. tshark capturando tráfico ICMP	35
4.2. Modo «verboso» de tshark	38
4.3. tshark permite elegir los campos a imprimir	39
7.1. Literales numéricos en Python	60
7.2. Conversión a representación binaria, octal y hexadecimal	60
7.3. Especificando la base en el constructor de int	61
7.4. Funciones de conversión de ordenamiento del módulo socket. . .	64
7.5. Codificación ASCII	64
7.6. Codificación UTF-8	65
7.7. struct: alternativas de ordenamiento	65
7.8. struct: empaquetado en diferentes tamaños	66
7.9. struct: empaquetando una cabecera Ethernet	67
7.10. struct: desempaquetando una cabecera Ethernet	67
9.1. Servidor de chat UDP (User Datagram Protocol) básico chat/udp_server.py	79
9.2. Client de chat UDP básico chat/udp_client.py	80
9.3. Servidor de chat UDP con respuesta chat/udp_server2.py	80
9.4. Cliente de chat UDP con respuesta chat/udp_client2.py	81
9.5. Servidor de chat UDP por turnos chat/udp_chat_server3.py	81
9.6. Cliente de chat UDP por turnos chat/udp_chat_client3.py	82
9.7. Servidor de chat UDP simultaneo chat/udp_chat_server_thread.py	82
9.8. Cliente de chat UDP simultáneo chat/udp_client_thread.py	83
9.9. Chat UDP (servidor y cliente) chat/udp_chat.py	84

9.10. Cliente HTTP básico	
examples/http_mini_client.py	85
9.11. Servidor HTTP	
examples/http_server.py	85

Listado de acrónimos

ADSL	Asymmetric Digital Subscriber Line
ANSI	American National Standards Institute
API	Application Program Interface
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
BSD	Berkeley Software Distribution
CIDR	Classless Interdomain Routing
CLI	Command Line Interface
CR	Carriage Return
DF	Don't Fragment
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EBCDIC	Extended Binary Coded Decimal Interchange Code
E/S	Entrada/Salida
EOL	End Of Line
ESI	Escuela Superior de Informática
FCS	Frame Check Sequence
FTP	File Transfer Protocol
GNU	GNU is Not Unix
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
IANA	Internet Assigned Numbers Authority
IBM	International Business Machines
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IGMP	Internet Group Management Protocol
IHL	Internet Header Length
IMAP	Internet Message Access Protocol
IOS	Internetwork Operating System

IP	Internet Protocol
IPC	Inter-Process Communication
IRC	Internet Relay Chat
ISO	International Organization for Standardization
KiB	Kibibyte (1 024 bytes)
LAN	Local Area Network
LED	Light Emitting Diode
LF	Line Feed
MiB	Mebibyte (1 024 KiB)
MAC	Media Access Control
MF	More Fragments
NIC	Network Interface Controller
OSI	Open Systems Interconnection
OUI	Organizationally Unique Identifier
PC	Personal Computer
PID	Process IDentifier
POO	Programación Orientada a Objetos
POP3	Post Office Protocol
POSIX	Portable Operating System Interface; UNIX
RFC	Request For Comments
ROM	Read Only Memory
RTC	Red Telefónica Conmutada
RTT	Round Trip Time
SMTP	Simple Mail Transport Protocol
SO	Sistema Operativo
SRT	Service Response Time
SSH	Secure SHell
TCP	Transmission Control Protocol
TCP/IP	Arquitectura de protocolos de Internet
ToS	Type of Service
TTL	Time To Live
TTY	TTY
UDP	User Datagram Protocol
URI	Universal Resource Identifier
UTF	Unicode Transformation Format
UTF-8	UTF de 8 bits
UUID	Universally Unique Identifier
VLC	Video LAN Client
WAN	Wide Area Network
WiFi	Wireless Fidelity
XML	Extensible Markup Language
WiFi	Wireless Fidelity

Prefacio

Las redes de comunicaciones, y especialmente Internet, se han convertido en una parte esencial de nuestras vidas. Internet ha cambiado radicalmente nuestra forma de comprar, viajar, hacer negocios, relacionarnos. Se trata de un cambio mucho más importante que la invención de la imprenta o la revolución industrial, supone un nuevo concepto de comunicación, un medio de difusión de información y conocimiento que no tiene precedentes. Y es la comunicación lo que nos convierte en seres sociales, en humanos. Las redes han cambiado lo que somos y cómo vivimos, y nada parece indicar que se trate de una moda pasajera.

Todo ello es motivo suficiente para que cualquier persona (y en particular cualquier técnico) se interese por el funcionamiento de las redes de computadores. Pocas cosas hoy en día influyen tanto en nuestra vida cotidiana. Entender cómo es y cómo funciona la red ayuda a tomar conciencia de sus posibilidades y oportunidades, pero también de sus debilidades y riesgos.

Este libro es una introducción muy práctica a las redes de computadores, concretamente a las basadas en la tecnología TCP/IP y por extensión a Internet. El enfoque que hemos seguido en este libro es un tanto particular. Se basa en dos tecnologías muy concretas:

- El sistema operativo GNU
- El lenguaje de programación Python

GNU

GNU es un sistema operativo estilo POSIX concebido y desarrollado bajo el concepto de software libre. Esto tiene dos implicaciones muy importantes:

- GNU incorpora todas las ventajas prácticas y técnicas de la tradición de la familia de sistemas UNIX. La tecnología base de Internet fue desarrollada sobre UNIX, concretamente BSD. Se podría decir que Internet forma parte del ADN de UNIX. GNU es heredero de todo ese legado.
- GNU está disponible para cualquier persona, en cualquier parte, sin ninguna limitación. Es *software libre* en estado puro, ideal para toda persona que quiera estudiar el funcionamiento de un sistema completo y productivo con todo detalle. Es tecnología *social* de la que aprender y enseñar, que se puede compartir, mejorar y, por supuesto, crear industria y riqueza. La comunidad de usuarios de GNU y todas sus distribuciones derivadas también es uno de sus puntos fuertes. Para seguir este documento se aconseja la distribución Debian GNU/Linux aunque, en esencia, cualquier otra puede ser perfectamente válida.

El entorno de trabajo y herramientas que se refieren en el presente texto corresponden al sistema operativo Debian GNU/Linux [deb]. Por simplicidad y para un máximo aprovechamiento te aconsejamos instalar esta distribución en tu computador, si bien cualquiera de las abundantes distribuciones basadas en GNU puede ser perfectamente válida para la realización de los ejemplos y ejercicios prácticos propuestos. También se asume que tu computador cuenta al menos con una interfaz de red Ethernet convenientemente configurada y que proporciona acceso a Internet sin restricciones importantes.

Python

Python es un lenguaje de programación interpretado, dinámico y orientado a objetos. A pesar de ser un lenguaje muy completo y potente su aprendizaje resulta sorprendentemente sencillo. Permite, incluso a un programador novato, ser productivo en muy poco tiempo, en comparación con otros lenguajes como Java, C++, C#, etc.

La librería estándar de Python respeta los nombres y conceptos básicos del API de programación de C de POSIX, lo que resulta muy útil para encontrar documentación y sobre todo facilita la *traducción* entre ambos lenguajes. Es muy útil como lenguaje de prototipado rápido, incluso aunque la versión final de la aplicación se vaya a implementar en otro lenguaje.

Python también es software libre, lo que significa que está disponible para cualquiera en cualquier plataforma. Está respaldado por una gran comunidad de usuarios y desarrolladores y suele ser fácil encontrar ayuda incluso en temas muy específicos.

Contenido de los capítulos

Shell

Una introducción al interprete de comandos.

Conectividad

Herramientas (y sus fundamentos) para la comprobación de conectividad de red de un computador.

Ejercicios

A lo largo del texto se proponen ejercicios, identificados en la forma «[EX]», siendo X un número creciente. La realización de estos ejercicios, normalmente muy prácticos, es importante para una adecuada comprensión y consolidación de los conceptos y manejo de las herramientas.

Sobre los ejemplos

Todos los ejemplos que aparecen en este documento (y algunos otros) están disponibles para descarga a través del repositorio mercurial en https://bitbucket.org/arco_group/python-net. Aunque es posible descargar estos ficheros individualmente o como un archivo comprimido, se aconseja utilizar el sistema de control de versiones mercurial¹.

¹<http://mercurial.selenic.com>

Si encuentra alguna errata u omisión es los programas de ejemplo, por favor, utilice la herramienta de gestión de incidencias (*issue tracker*) accesible desde https://bitbucket.org/arco_group/python-net/issues para notificarlo a sus autores.

Sobre este documento

Este documento está editado con GNU Emacs y tipografiado con \LaTeX en un sistema Debian GNU/Linux. Las figuras fueron realizadas con Inkscape.

Los fuentes de este documentos también se encuentran en un repositorio mercurial https://bitbucket.org/arco_group/net-book. Si quiere colaborar activamente en su desarrollo o mejora documento póngase en contacto con los autores.

Al igual que los ejemplos, también existe una herramienta de gestión de incidencias (pública) en la que puede notificar problemas o errores de cualquier tipo que haya detectado en el documento.

Shell

David Villa

El interprete de comandos (*shell* o también *consola*) es probablemente una de las herramientas más interesantes del sistema operativo GNU. Como con cualquier tecnología o herramienta que merezca la pena, es fácil encontrar grandes partidarios y detractores. Lo cierto es que la shell es una herramienta extraña, oscura y de aspecto absolutamente espartano en un mundo en el que las pantallas multi-táctiles o el reconocimiento de voz son algo habitual.

Ciertamente una interfaz de comandos —a menudo denominada CLI (Command Line Interface)— es mucho menos intuitiva¹ que una GUI (Graphical User Interface). Sin embargo, cuando la potencia (y por tanto la complejidad) de la herramienta aumenta, la interfaz gráfica resulta mucho más difícil de desarrollar, requiere tanto o más entrenamiento y su uso suele ser menos productivo que una interfaz de comandos. Claros ejemplos de ello podemos encontrarlos en programas como AutoCAD o Cisco IOS (Internetwork Operating System); aplicaciones con un alto grado de especialización, que a pesar de contar también con interfaz gráfica, suelen utilizarse mediante comandos, sobre todo por parte de los usuarios expertos.

Otra buena razón por la que la interfaz de comandos puede resultar más productiva es la facilidad para combinar conjuntos de sentencias en forma de macros, lo que además permite automatizar tareas repetitivas. Especificar o documentar un procedimiento, incluso de complejidad media, usando comandos es mucho más sencillo que dar instrucciones para utilizar menús, botones y listas desplegables, algo que a menudo requiere grabar en vídeo la secuencia de acciones —los llamados *screencast*.

«In the Beginning... Was the Command Line»² de Neal STEPHESON, es un ensayo crítico (y a ratos humorístico) sobre la influencia que los computadores, los sistemas operativos y sus distintas filosofías tienen sobre la sociedad actual, y especialmente sobre las personas con inquietudes técnicas. Uno de los temas trata precisamente sobre las diferencias entre interfaz gráfica e interfaz de comandos.

En todo caso, es importante entender que es un error subestimar o considerar anticuada una aplicación simplemente por el hecho de utilizar una interfaz de comandos. A menudo, las aplicaciones bien diseñadas definen un conjunto de acciones en una librería, que pueden ser utilizadas indistintamente desde *front-*

¹El manejo *intuitivo* se refiere a la posibilidad de realizar un uso productivo de una herramienta sin conocimiento ni instrucción previa. El adjetivo se puede aplicar tanto a la herramienta como al usuario.

²http://en.wikipedia.org/wiki/In_the_Beginning..._Was_the_Command_Line

*ends*³ gráficos, línea de comandos, o incluso desarrollando otras aplicaciones a medida. Ejemplos de este tipo de aplicaciones pueden ser VLC, FIXME:Amarok, etc. Incluso servicios de Internet tan conocidos con HTTP, SMTP o FTP pueden ser utilizados mediante comandos textuales, tal como veremos en capítulos posteriores.

1.1. GNU Bash

Después de esta introducción, el capítulo se centra en Bash —el interprete de comandos por excelencia de todo sistema operativo GNU. Bash es probablemente una de las shells más famosas y potentes, aunque obviamente no es la única. Los sistemas POSIX han conocido una larga variedad de shell desde principios de los años 70. Aunque bash tiene características muy interesantes (como la «complección» automática contextual) primero debemos abordar la cuestiones básicas que todo usuario avanzado debe conocer.

Por otra parte, hay una colección de programas (agrupados con el nombre de GNU *coreutils*) que realizan acciones relativamente sencillas, pero que están diseñados de modo que se puedan combinar para realizar operaciones de complejidad nada trivial de forma bastante simple, una vez que se entiende un concepto clave: la redirección de la entrada/salida.

Estas y otras aplicaciones junto con el lenguaje de programación «C-Shell» proporcionan una herramienta con infinitas posibilidades para todo tipo de tareas, que pueden ir desde la administración de sistemas hasta las más sofisticadas técnicas de pen-testing, pasando por el desarrollo de aplicaciones de todo tipo.

1.2. Valor de retorno

En los sistemas POSIX se asume que cuando un programa acaba devuelve un valor entero. Este valor es recogido por su proceso padre (que puede ser una shell) y ofrece información muy útil para saber cómo ha ido su ejecución.

Un programa que realiza su función sin ningún inconveniente importante debería devolver un valor 0, mientras que un valor distinto indica que hubo un problema. El valor concreto puede utilizarse para señalar la causa concreta.

Éste es el motivo por el que el estándar del lenguaje C dice que toda función `main()` debe devolver un entero y que por defecto su valor de retorno debería ser cero. Según eso, el «hola mundo» correcto en C es el siguiente:

```
1 #include <stdio.h>
2 int main(int argc, char* argv[]) {
3     puts("hello world\n");
4     return 0;
5 }
```

Por ejemplo, el siguiente listado muestra la ejecución del comando `ls /`, que lista el contenido del directorio raíz:

```
david@amy:~$ ls /
```

³Se denomina *front-end* a la capa de software que proporcionad una interfaz —del tipo que sea— a un programa o librería que ofrece su funcionalidad por medio de una serie de funciones o clases (API)

bin	etc	lib	mnt	root	selinux	tmp	vmlinuz
boot	home	lost+found	opt	run	srv	usr	
dev	initrd.img	media	proc	sbin	sys	var	

La shell bash creó un proceso en el cuál ejecutó el programa `ls` con el argumento `/`, después esperó a que el programa terminase y recogió su valor de retorno. Ese valor lo almacenó en una «variable de entorno» llamada «`?`». Se puede comprobar el valor de una variable utilizando el comando `echo` y colocando el símbolo «`$`» delante del nombre de la variable, tal que:

`echo`
escribe la cadena de texto
que le se indique como
argumento, expendiendo las
variables que incluya
(identificadores precedidos
del símbolo «`$`»).

```
david@amy:~$ echo $?
0
```

Pero solo es aplicable al comando inmediatamente anterior. Una ejecución incorrecta retornaría un código de error, como se puede comprobar en el siguiente ejemplo:

```
david@amy:~$ ls noexiste
ls: cannot access noexiste: No such file or directory
david@amy:~$ echo $?
2
```

1.3. Gestión básica de procesos

El proceso es una abstracción del SO para ejecutar un programa conforme a determinados parámetros de seguridad, prioridad y privilegios de acceso a recursos. Un usuario puede ver los procesos del terminal al que está conectado con el comando `ps`

```
david@amy:~$ ps T
  PID TTY          TIME CMD
 4970 pts/2    00:00:00 bash
 5107 pts/2    00:00:01 emacs
 5164 pts/2    00:00:00 bash
 6021 pts/2    00:00:01 firefox-bin
 6204 pts/2    00:00:00 ps
```

La primera columna indica el PID (Process IDentifier) de cada proceso. La segunda (TTY) el terminal al que están asociados. la tercera (TIME), el tiempo de procesador otorgado al proceso y por último (CMD) el programa que se está ejecutando.

El programa `ps` dispone de una amplia variedad de opciones para filtrar qué procesos se desean listar, qué atributos o en qué formato. Por ejemplo, el siguiente comando muestra las relaciones de jerarquía:

```
david@amy:~$ ps f
  PID TTY          STAT TIME  COMMAND
 4970 pts/2    Ss   0:00  bash
 5107 pts/2    T    0:01  \_  emacs
 5164 pts/2    S    0:00  \_  bash
 6021 pts/2    Sl   0:01  \_  /usr/lib/iceweasel/firefox-bin
 6283 pts/2    R+   0:00  \_  ps f
 4592 pts/0    Ss   0:00  bash
 4755 pts/0    S+   0:28  \_  emacs shell.tex
```

En esta ocasión aparece una columna adicional (STAT) que indica el estado del proceso. La primera letra: R (*running*), T (*stopped*) y S (*sleep*). La segunda: s (*session leader*), l (*multi-hilo*), + (*en primer plano*).

La operación más habitual que se suele hacer sobre un proceso existente es matarlo (con el comando `kill`).

1.4. Gestión de trabajos

La shell crea un nuevo proceso hijo⁴ para ejecutar cada comando que se le pide. El comportamiento normal de la shell es esperar a que ese proceso termine antes de permitir la introducción de un nuevo comando —fácil de comprobar con el programa `sleep`. Esto se llama ejecución en «primer plano» o «foreground».

Sin embargo la shell puede no esperar (si se le pide), permitiendo incluso que el comando quede en ejecución indefinidamente. A eso se le llama ejecución en «segundo plano» o «background». Para conseguir que la shell ejecute un comando en segundo plano basta con añadir el símbolo «ampersand» ('&') al final de la línea de comando:

```
david@amy:~$ emacs &
[1] 19777
david@amy:~$
```

Como se aprecia en el listado anterior, la shell imprime una línea con dos números y luego queda disponible para introducir el siguiente comando. El primer número [entre corchetes] identifica al «trabajo» en segundo plano (proceso hijo de la shell). El segundo número es el PID de dicho proceso, que lo identifica dentro del SO completo. Una shell puede ejecutar múltiples trabajos en segundo plano. La forma más sencilla de ver cuáles son es el comando `jobs`:

```
david@amy:~$ firefox &
[2] 20847
david@amy:~$ jobs
[1]-  Running                  emacs &
[2]+  Running                  firefox &
```

Si el usuario ejecuta el comando `fg` (*foreground*), el último comando ejecutado (marcado con el símbolo '+') pasará a primer plano, bloqueando la shell. Si se le da un número de trabajo será ése el que pase a primer plano.

```
david@amy:~$ fg %1
emacs
```

Si la shell se encuentra bloqueada en espera de la finalización de un comando en primer plano, el usuario puede pararlo pulsando Control-Z. Partiendo de la situación anterior:

```
david@amy:~$ fg %1
emacs
^Z
```

kill

A pesar de su nombre, sirve para enviar una señal a un proceso. La señal por defecto, si no se indica otra, es SIGTERM. Puede ver la lista de todas las señales con `kill -l`.

emacs

GNU Emacs es un potente editor para todo tipo de lenguajes de programación, altamente configurable gracias al interprete de lenguaje Lisp que incorpora.

⁴La propia shell también se ejecuta en un proceso, que puede haber sido creado a su vez por otra shell...


```
[1]+  Stopped                  emacs
david@amy:~$ jobs
[1]+  Stopped                  emacs
[2]-  Running                  firefox &
david@amy:~$
```

Un trabajo parado puede volver a estado de ejecución en primer plano si se introduce el comando `fg` o segundo plano si se introduce `bg` (*background*).

```
david@amy:~$ jobs
[1]+  Stopped                  emacs
[2]-  Running                  firefox &
david@amy:~$ bg
[1]+  emacs &
david@amy:~$
```

El usuario puede enviar una señal a cualquier trabajo usando el comando `kill` indicando el número de trabajo y opcionalmente un número de señal (por defecto `SIGTERM`).

```
david@amy:~$ kill -SIGKILL %2
[4]+  Killed                  firefox
```

1.5. Entrada, salida y salida de error estándar

En los sistemas POSIX los programas interactúan con el SO únicamente mediante ficheros —o de abstracciones que ofrecen el mismo interfaz de uso. Es decir, la lectura o escritura desde y hacia cualquier disco, pantalla, teclado, tarjeta de sonido o cualquier otro periférico se realiza en términos de primitivas `read/write` de forma similar a un fichero.

Todo proceso —programa en ejecución— dispone desde su inicio de tres descriptores de fichero abiertos:

- la entrada estándar (o «`stdin`») con el descriptor 0,
- la salida estándar (o «`stdout`») con el descriptor 1 y
- la salida de error estándar (o «`stderr`») con el descriptor 2.

Eso significa que cuando el programa trata de escribir algo (con la función `puts()` en C o `System.out.println()` en Java) lo hará escribiendo sobre su salida estándar. Del mismo modo, cuando el programa trata de leer o genera un mensaje de error, esas operaciones se realizarán respectivamente sobre su entrada y salida de error estándar.

Normalmente, la entrada estándar está ligada al teclado, mientras que la salida y salida de error estándar están ligadas a la pantalla (teclado y pantalla se conocen comúnmente por «consola»). Esta asociación entre la consola y los descriptores de fichero estándar la decide precisamente la shell, y mediante las órdenes correspondientes, el usuario puede alterar esa asociación para que la entrada y la salida de datos de un programa queden ligadas con otra cosa, tal como un fichero en disco o incluso otro programa...

La ejecución anterior del comando `ls /` (el listado de ficheros del directorio raíz) apareció en pantalla debido a que su salida estándar corresponde por defecto a la consola.

1.6. Redirección

Es posible alterar la salida estándar de cualquier programa para que utilice un fichero en disco simplemente con el símbolo ‘>’ (mayor-que) seguido del nombre del fichero:

```
david@amy:~$ ls -l / > /tmp/root-dir
david@amy:~$
```

Repito: la «redirección de salida» consigue que **cualquier** programa pueda almacenar su resultado en un fichero sin que el desarrollador de dicho programa tenga que hacer absolutamente nada para soportarlo. Todo gracias a la shell.

Dos cosas han cambiado respecto a la ejecución anterior del comando `ls`: la primera es que se ha especificado la opción `-l` que hace que se muestren permisos, propietario y fecha de modificación de cada fichero o directorio. La segunda es que esta vez *nada* ha aparecido en la consola. La lista de los nombres de fichero ha sido almacenada en el fichero `/tmp/root-dir`.

Puede ver el contenido de dicho fichero utilizando el programa `cat`:

```
david@amy:~$ cat /tmp/root-files
total 98
drwxr-xr-x  2 root root  4096 Aug 15 00:34 bin
drwxr-xr-x  4 root root  2048 Aug 17 22:22 boot
drwxr-xr-x 17 root root  3560 Aug 26 10:20 dev
drwxr-xr-x 193 root root 12288 Aug 25 18:29 etc
[...]
```

La redirección simple (>) crea siempre un fichero nuevo. Si existe un fichero con el mismo nombre, su contenido se pierde y es substituido por los nuevos datos. Pero existe otro tipo de redirección de salida que añade el contenido *al final* del fichero especificado. Se indica con dos símbolos mayor-que

```
david@amy:~$ date > /tmp/now
david@amy:~$ date >> /tmp/now
david@amy:~$ cat /tmp/now
Sat Aug 25 18:27:37 CEST 2012
Sat Aug 25 18:27:41 CEST 2012
```

Volviendo al fichero `root-files`, veamos cómo podríamos filtrar sus líneas de modo que aparezcan únicamente las que contengan la cadena «Jun» (en principio los ficheros modificados en junio). Para ello se puede utilizar el comando `grep` del siguiente modo:

```
david@amy:~$ grep Jun /tmp/root-files
drwxr-xr-x  6 root root  4096 Jun  6 17:58 home
lrwxrwxrwx  1 root root    32 Jun 27 13:09 initrd.img
lrwxrwxrwx  1 root root    28 Jun 27 13:09 vmlinuz
```

`grep`, como muchos otros programas, utiliza su entrada estándar como fuente de datos si no se le dan argumentos. Por tanto, utilizando la «redirección de entrada» (con el carácter ‘<’) se puede lograr lo mismo ejecutando:

```
david@amy:~$ grep Jun < /tmp/root-files
drwxr-xr-x  6 root root  4096 Jun  6 17:58 home
```

/tmp

El directorio `/tmp` se utiliza por las aplicaciones para ficheros *temporales* en los que almacenar logs o datos de sesión. El contenido de este directorio se elimina en el arranque del SO.

cat

Lee el contenido de los ficheros que se le indiquen como argumentos y lo escribe en su salida estándar. Si no se le dan argumentos, leerá de su entrada estándar.

date

Escribe en su salida estándar la fecha y hora actual con la zona horaria configurada.

grep

Escribe en su salida estándar aquellas líneas que coincidan con un criterio especificado. Lee de los ficheros indicados como argumentos (o de su entrada estándar en su defecto).

```
lrwxrwxrwx 1 root root 32 Jun 27 13:09 initrd.img
lrwxrwxrwx 1 root root 28 Jun 27 13:09 vmlinuz
```

La diferencia es que en este caso el comando `grep` no tiene constancia de estar leyendo realmente de un fichero en disco. Resulta irrelevante.

Si se desea almacenar el resultado obtenido en otro fichero basta con utilizar de nuevo la redirección de salida. Es posible combinar redirecciones de entrada y salida en el mismo comando:

```
david@amy:~$ grep Jun < /tmp/root-files > /tmp/Jun-files
```

El contenido de ese fichero está ordenado por los nombres de los ficheros (la última columna). Veamos cómo reordenar esa lista en función del tamaño (quinta columna) utilizando el comando `sort`:

sort
 Ordena las líneas de los ficheros que se le indiquen como argumento (o de su entrada estándar) y las escribe en su salida estándar. Se pueden especificar diferentes criterios de ordenación mediante opciones.

```
david@amy:~$ sort --numeric-sort --key=5 /tmp/Jun-files
lrwxrwxrwx 1 root root 28 Jun 27 13:09 vmlinuz
lrwxrwxrwx 1 root root 32 Jun 27 13:09 initrd.img
drwxr-xr-x 6 root root 4096 Jun 6 17:58 home
```

Para lograr este resultado se han creado dos ficheros temporales (`root-files` y `Jun-files`). Si lo único que interesa es el resultado final, hay una forma más sencilla y rápida de conseguir lo mismo sin necesidad de crear ficheros intermedios: la tubería.

La tubería (*pipe*) conecta la salida estándar de un proceso directamente con la entrada de otro, de modo que todo lo que el primer programa escriba podrá ser leído inmediatamente por el segundo. Para indicar a la shell que cree una tubería se utiliza el carácter `|` (AltGr-1 en el teclado español). El comando para obtener directamente los ficheros del directorio raíz modificados en junio ordenados por tamaño sería:

```
david@amy:~$ ls -l / | grep Jun | sort -n -k5
lrwxrwxrwx 1 root root 28 Jun 27 13:09 vmlinuz
lrwxrwxrwx 1 root root 32 Jun 27 13:09 initrd.img
drwxr-xr-x 6 root root 4096 Jun 6 17:58 home
```

Nótese que `grep` y `sort` no tienen nombres de fichero en sus argumentos, y por tanto leen de sus respectivas entradas estándar.



La mayoría de los programas CLI disponen de opciones que modifican su comportamiento aportando gran versatilidad. Las opciones normalmente presentan dos formas equivalentes: un guión y una letra (`-h`) o dos guiones y una palabra (`--help`).

El formato largo es el adecuado cuando se quiere comandos auto-explicativos (como en este documento) o cuando se escribe un *script* (un programa en un fichero de texto que será interpretado por la shell). El formato corto es más conveniente cuando el usuario escribe comando directamente en la consola, por el ahorro de tiempo obviamente.

Por supuesto es posible combinar la tubería y la redirección en el mismo comando. En este caso el resultado del comando anterior se almacena en un fichero, del mismo modo que se hacía con los comandos simples:

```
david@amy:~$ ls -l / | grep Jun | sort -n -k5 > /tmp/root-jun-files
-order-by-size
```

Es fácil apreciar el gran potencial de este sencillo mecanismo. Cualquier sistema POSIX (en especial GNU) dispone de una gran cantidad de pequeños programas especializados —como los que se han introducido aquí— que pueden combinarse mediante redirección para cubrir una gran variedad de necesidades puntuales de una forma rápida y eficiente.

1.7. Catálogo de comandos

A continuación se introducen brevemente algunos de los programas más comunes y útiles cuando se trabaja con la shell.

1.7.1. Ficheros y directorios

pwd (*print working directory*)

escribe el nombre del directorio actual.

cp (*copy*)

dados un nombre de fichero existente y un directorio o nombre de fichero, copia el primero en el segundo. Si el fichero destino existe, lo sobrescribe.

mv (*move*)

es equivalente a cp pero borra el fichero original.

rm (*remove*)

elimina el fichero indicado.

mkdir (*make dir*)

crea un directorio.

rmdir (*remove dir*)

borra un directorio vacío.

cut escribe en su salida partes de las líneas de entrada según sus opciones:

```
david@amy:~$ date
Sun Aug 26 12:47:35 CEST 2012
david@amy:~$ date | cut --delimiter=" " --fields=2
Aug
```

head

escribe a su salida los primeros bytes o líneas de su entrada o del fichero indicado como argumento.

```
david@amy:~$ head --lines=3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
```

- ln** (*link*)
crea enlaces a otros ficheros o directorios.
- md5sum**
calcula (o comprueba) la suma MD5 del fichero indicado (o de su entrada estándar).
- mkfifo**
crea una tubería ligada al sistema de ficheros con un nombre.
- nl** (*number lines*)
lee líneas de un fichero y las escribe a su salida precedidas del número de línea.
- seq** (*sequence*)
escribe un rango de enteros. Puede usarse como variable de control de un bucle `for`.
- split**
divide un fichero en trozos del tamaño indicado.
- stat**
produce información detallada de ficheros.
- tac** (*reverse cat*)
escribe a su salida líneas de su entrada en orden inverso.
- tail**
escribe a su salida los últimos bytes o líneas de su entrada o del fichero indicado como argumento. Con la opción `-f/-follow` monitoriza el fichero mostrando el nuevo contenido tan pronto como se añade. Muy útil para hacer el seguimiento de un fichero de log.
- tee** crea una «te». Lee de su entrada estándar y lo escribe a su salida estándar y al fichero indicado al mismo tiempo.
- test**, [
realiza comprobaciones lógicas sobre ficheros, cadenas de texto y datos numéricos. Se utiliza habitualmente como condición en estructuras de control `if`, `while`, etc.
- touch**
cambia la fecha de un fichero (por defecto ahora). Si el fichero indicado no existe, lo crea vacío.
- tr** (*translate*)
lee líneas de su entrada y las escribe a su salida reemplazando *tokens*⁵ tal como lo indiquen sus opciones.
- uniq**
lee líneas de su entrada y las escribe a su salida, pero omitiendo líneas consecutivas idénticas.
- wc** (*word count*)
cuenta letras, palabras y líneas del fichero indicado (o de su entrada estándar) y escribe los totales en su salida.

⁵Un *token* es cualquier combinación de caracteres que cumpla una expresión regular concreta.

yes escribe «y» y un salto de línea continuamente a su salida. Se utiliza para contestar afirmativamente a cualquier pregunta que haga un programa por línea de comando:

```
david@amy:~$ touch kk
david@amy:~$ yes | rm --interactive --verbose kk
rm: remove regular empty file 'kk'? removed 'kk'
```

1.7.2. Sistema

df muestra información sobre el uso de los sistemas de ficheros montados en el computador.

du calcula el espacio de disco utilizado por ficheros y directorios.

uname
muestra información del sistema:

```
david@amy:~$ uname -a
Linux amy 3.2.0-3-686-pae #1 SMP Mon Jul 23 03:50:34 UTC 2012
i686 GNU/Linux
```

sync
escribe a disco inmediatamente las operaciones pendientes sobre ficheros.

1.7.3. Procesos

nice
ejecuta un programa fijando un nivel de prioridad.

nohup
ejecuta un comando ignorando las señales para su finalización. Permite que un proceso creado por una shell sobreviva a la propia shell.

sleep
realiza una pausa.

true, false
simplemente retornan 0 y 1, los valores que corresponden a una ejecución correcta e incorrecta respectivamente.

top

1.7.4. Usuarios y permisos

chmod (*change mode*)
cambia los permisos de lectura, escritura y ejecución de un fichero o directorio.

chown (*change owner*)
cambia el propietario (usuario y grupo) de un fichero.

chgrp (*change group*)
cambia el grupo propietario de un fichero.

groups
escribe los nombres de los grupos a los que pertenece el usuario.

- id** escribe los identificadores del usuario y los grupos a los que pertenece.
- who** muestra los usuarios conectados junto con los terminales que utilizan y la hora de la conexión (*login*).
- whoami**
muestra el nombre del usuario conectado.

Conectividad

David Villa

La *conectividad* se refiere a la capacidad de un proceso, computador o equipo de comunicaciones para intercambiar información con un igual. Las herramientas disponibles para la verificación de conectividad resultan extremadamente útiles para resolver una gran cantidad de averías y problemas que pueden presentarse en redes de comunicaciones de cualquier tipo.

Aprender a manejar estas herramientas y conocer sus fundamentos teóricos es muy importante para cualquier profesional relacionado con el desarrollo, implantación y explotación de infraestructuras de comunicaciones.

En este capítulo

Después de este capítulo el lector debería ser capaz de responder satisfactoriamente a las siguientes cuestiones:

- Qué es y cuál es la utilidad de los modelos de referencia OSI, TCP/IP e híbrido.
- Qué es la conectividad a nivel de enlace, red, transporte y aplicación.
- Qué son y cómo se configuran las interfaces de red de un computador.
- Cómo se utilizan, cómo funcionan y cuáles son las bases teóricas en las que se fundamentan las herramientas aplicables a la comprobación de conectividad.

2.1. Modelos de referencia

El modelo de referencia OSI —definido por la ISO— se utiliza para el estudio, diseño e implementación de cualquier tecnología de comunicaciones. El modelo describe las interfaces, protocolos y servicios que proporciona cada una de las siete capas: aplicación, presentación, sesión, transporte, red, enlace y física. Cada capa (por medio de los protocolos correspondientes) se centra en resolver problemas y ofrecer funcionalidades concretas, y ofrece servicios a la capa inmediatamente superior y demanda servicios de la inmediatamente inferior. De ese modo se consigue aislar y desacoplar sus funciones simplificando cada elemento, y haciéndolo reemplazable.

A continuación se explica brevemente el objetivo de cada capa:

1. Física. Define las características eléctricas, mecánicas y temporales requeridas en una tecnología de comunicación de datos particular.

2. Enlace. Se ocupa del intercambio de mensajes entre nodos directamente conectados (vecinos). Cuando usa un medio de difusión suele proporcionar un sistema de *direccionamiento físico*.
3. Red. Proporciona soporte para comunicaciones *extremo a extremo*, es decir, que puede implicar dispositivos intermediarios. Permite enviar mensajes individuales de tamaño variable y define un sistema de *direccionamiento lógico*.
4. Transporte. Proporciona un canal de comunicación libre de errores entre procesos o usuarios finales. Incluye un mecanismo de multiplexación y un sistema de direccionamiento de procesos.
5. Sesión. Permite crear sesiones entre hosts remotos y se ocupa de la sincronización.
6. Presentación. Define la representación canónica de los datos (reglas de codificación) y su semántica asociada.
7. Aplicación. Incluye protocolos especializados según la aplicación.

El modelo TCP/IP se definió años después de las primeras implementaciones y trata de formalizar y estandarizar para garantizar interoperabilidad entre los distintos fabricantes. Este modelo define únicamente cuatro capas:

1. Host a red. Asume que existen los mecanismos necesarios para conseguir que un paquete IP pueda ser enviado desde un host a sus vecinos, pero realmente no aborda la problemática específica que ello implica.
2. Inter-red. Proporciona los mecanismos de interconexión de redes y encaминamiento de paquetes. Define el protocolo IP (Internet Protocol), que proporciona un servicio de entrega sin conexión.
3. Transporte. Define dos protocolos de transporte: TCP que proporciona un servicio confiable y orientado a conexión y UDP que únicamente proporciona multiplexación.
4. Aplicación. Incluye los protocolos para los servicios comunes, tales como: DNS, SMTP, HTTP, FTP, etc.

Por último, el modelo híbrido es una mezcla de los modelos OSI y TCP/IP eliminando «lo que sobra» (las capas de presentación y sesión son aplicables a relativamente pocas aplicaciones) y añadiendo «lo que falta» (la capa de enlace es decisiva para comprender el funcionamiento de la arquitectura de red).

La figura 2.1 muestra la correspondencia entre los tres modelos anteriores:

2.2. Conectividad, por capa

Tomando como guía el modelo de referencia OSI se puede considerar la conectividad a distintos niveles o capas, aunque por norma general se habla de conectividad sólo en los niveles de enlace, red y transporte, y marginalmente en el nivel de aplicación. Veamos qué implica cada uno de ellos:

Aplicación

Se tiene conectividad a nivel de aplicación cuando dos computadores pueden establecer una comunicación por medio de una aplicación final.

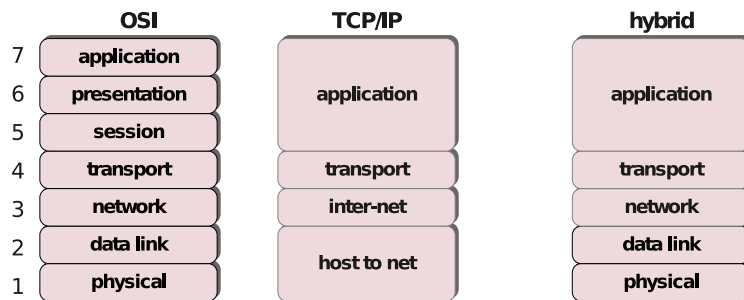


FIGURA 2.1: Correspondencia entre los diferentes modelos de referencia

Suele utilizarse `telnet` aunque cualquier otro programa que emplee un protocolo de aplicación puede servir para demostrar que existe conectividad.

Transporte

La conectividad a nivel de transporte implica la posibilidad de crear un servidor TCP o UDP y la de conectar un cliente desde otro computador. Se trata de una comunicación extremo a extremo, es decir, dos computadores que se comunican a través de una inter-red cualquiera. La herramienta más sencilla y adecuada para comprobarlo es `netcat`.

Red Conectividad a nivel de red se refiere únicamente a la capacidad de una pareja de computadores para intercambiar paquetes IP a través de una red o inter-red. Para ello se utilizan habitualmente los programas `ping` y `traceroute`.

Enlace

Se habla de conectividad a nivel de enlace cuando se verifica que el dispositivo es capaz de intercambiar tramas con sus nodos vecinos inmediatos. En este caso las herramientas pueden ser muy variadas puesto que dependen del protocolo y tecnología concreta: Ethernet, X.25, frame relay, etc.

Como paso previo al uso de las herramientas mencionadas, el lector aprenderá algunas nociones básicas sobre la configuración de su computador, especialmente lo referente a sus interfaces de red y servicios básicos de sistema operativo.

2.3. Configuración básica

Lo primero que necesita un computador es un nombre, y elegir un buen nombre no siempre es una tarea sencilla [Lib90]. El comando `hostname`, ejecutado en una consola, muestra el nombre asignado al computador, sin incluir su dominio.

```
$ hostname
storm
```

Si dispones de privilegios de administrador (cuenta de root) puedes utilizar `hostname` para cambiar el nombre del computador. Para que este cambio

permanezca después de reiniciar debes editar el fichero `/etc/hostname` (lo cual requiere también permisos de administrador).

Puedes conseguir algo más de información sobre tu computador con el comando `uname -a`.

```
$ uname -a
Linux storm 2.6.35 #1 SMP Sat Sep 11 20:42:05 UTC 2010 i686 GNU/
Linux
```

La información que aparece corresponde con:

- Nombre del núcleo: «Linux»
- Nombre del nodo: «storm»
- Versión del núcleo: «2.6.25»
- Fecha y hora en que fue compilado el núcleo
- Tipo de arquitectura: «i686»
- Nombre del sistema operativo: «GNU/Linux»

2.4. Interfaces de red

La forma más sencilla de ver con qué interfaces de red cuenta tu computador es con el comando `ifconfig` (del paquete `net-tools`). Se trata de una herramienta pensada para el administrador. Por esa razón no se encuentra en la ruta de búsqueda de comandos, a menos que estés usando la cuenta de administrador. Aún así puede ser usada por usuarios comunes, pero habrás de ejecutarla indicando la ruta completa, tal como se muestra:

```
$ /sbin/ifconfig
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128  Scope:Host
            UP LOOPBACK RUNNING  MTU:16436  Metric:1
            RX packets:430205 errors:0 dropped:0 overruns:0 frame:0
            TX packets:430205 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:109639240 (104.5 MiB)  TX bytes:109639240 (104.5 MiB)

eth0        Link encap:Ethernet  HWaddr 00:22:34:6b:c5:47
            inet addr:161.67.101.12  Bcast:161.67.101.255  Mask:255.255.255.0
            inet6 addr: fe80::225:32ee:cf53:c429/64  Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:21037026 errors:0 dropped:0 overruns:0 frame:0
            TX packets:10171510 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:4206810852 (3.9 GiB)  TX bytes:750085525 (715.3 MiB)
            Interrupt:17
```

En la salida puedes ver dos secciones claramente diferenciadas, una especie de «ficha» de datos para cada una de las dos interfaces de red que ha encontrado: `lo` y `eth0`.

2.4.1. Loopback

La interfaz `lo` (*loopback*) es una interfaz de red un tanto especial ya que no está ligada a ningún dispositivo físico de comunicaciones. Se dice que es

una interfaz «virtual», A pesar de ello se puede utilizar para cualquier servicio basado en TCP/IP teniendo en cuenta que su ámbito de trabajo está limitado al propio computador tal como aparece en la línea 4: `Scope:Host`.

La interfaz loopback suele tener la dirección IP 127.0.0.1 (línea 3) independiente del tipo de computador o sistema operativo. El nombre simbólico para esa dirección es «localhost».

La línea 5 muestra otros datos interesantes:

- La interfaz de red está habilitada por parte del administrador: UP.
- La interfaz está activa y puede ser utilizada para transmitir paquetes: RUNNING.
- Es de tipo bucle: LOOPBACK.
- Tiene una MTU de 16436 bytes.

2.4.2. eth0

La interfaz eth0 identifica a una NIC (Network Interface Controller) de tipo Ethernet (línea 11). Por ese motivo la interfaz:

- Tiene una dirección física (una MAC): 00:22:34:6b:c5:47.
- Tiene una dirección IP (161.67.101.12) y una dirección de broadcast (161.67.101.255).
- Permite direccionamiento BROADCAST y MULTICAST (línea 14).
- Tiene una MTU de 1500 bytes (inherente a Ethernet).

También se muestran datos sobre la operación de la interfaz (aplicable también a la interfaz loopback):

RX packets

(líneas 6 y 15) Cantidad total de paquetes recibidos. Se indica también en cuántos falló el CRC (errors), cuántos fueron descartados (dropped), cuántos provocaron un desbordamiento del buffer de recepción (overruns).

TX packets

(líneas 7 y 16) Cantidad de paquetes transmitidos. Como en el caso anterior se indican paquetes erróneos, descartados o que provocaron overrun.

collisions

(línea 17) Número de colisiones cuando la interfaz de red opera en modo half duplex.

txqueuelen

(línea 17) En el tamaño de la cola de transmisión (en bytes)

frame

(línea 15) Errores de entramado.

carrier

(línea 16) Fallos en la detección de la portadora.

2.4.3. iproute2

iproute2 [Kuz] es otro paquete de software que permite visualizar y manipular la configuración de las interfaces de red (entre otras muchas cosas). En realidad, el comando *ip* (proporcionado por el paquete *iproute2*) es el reemplazo para *ifconfig*, *route* y otras herramientas para administración de la red.

El comando equivalente a *ifconfig* si usas *iproute2* es *ip addr show* o simplemente *ip addr*.

```

1 $ ip addr
2 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
3     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4     inet 127.0.0.1/8 scope host lo
5     inet6 ::1/128 scope host
6         valid_lft forever preferred_lft forever
7 2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state
8     UP qlen 1000
9     link/ether 00:22:34:6b:c5:47 brd ff:ff:ff:ff:ff:ff
10    inet 161.67.101.12/24 brd 161.67.101.255 scope global eth0
11    inet6 fe80::225:32ee:cf53:c429/64 scope link
        valid_lft forever preferred_lft forever

```

2.5. Encaminamiento básico

Al igual que los encaminadores (*routers*), los computadores también tienen una tabla de rutas. Esa tabla indica qué debe hacer con los paquetes que *salen*. Puedes ver, haciendo uso del comando *route* qué aspecto tiene la tabla de rutas de tu computador.

```

$ /sbin/route -n
Kernel IP routing table

```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
161.67.101.0	0.0.0.0	255.255.255.0	U	0	0	0	eth0
0.0.0.0	161.67.101.1	0.0.0.0	UG	0	0	0	eth0

Puedes ver que la tabla que devuelve tiene únicamente dos filas. La primera indica que todo el tráfico dirigido a la red 161.67.101.0/24 debe enviarse directamente (Gateway = 0.0.0.0) al destino a través de la interfaz *eth0*.

La segunda fila dice que el tráfico dirigido a cualquier otra parte (Destination = 0.0.0.0) debe enviarse al encaminador 161.67.101.1 a través de la interfaz *eth0*. Éste es el encaminador por defecto, que en el caso de ser una red local también se llama encaminador local, pasarela (*gateway*) o puerta de enlace.

El comando *ip* también se puede utilizar para visualizar la tabla de rutas, concretamente debes ejecutar *ip route show*:

```

$ ip route show
161.67.101.0/24 dev eth0 proto kernel scope link src
    161.67.101.12
default via 161.67.101.1 dev eth0

```

2.6. Conectividad a nivel de enlace

Esta sección asume que tu computador tiene una conexión Ethernet correctamente configurada y funcionando. Y es esa interfaz la que vamos a comprobar

en esta y posteriores secciones. Normalmente el equipo estará conectado a un conmutador (*switch*), ya sea directamente o a través de una roseta en la pared. Físicamente el conector RJ45 hembra suele presentar dos LEDs (ver Figura 2.2).

El LED de la izquierda aparece encendido si se ha podido establecer el enlace de datos con el vecino (el conmutador). Es habitual también que indique el modo (velocidad) a la que ha realizado la negociación: verde para 10 Mbps, naranja para 100 Mbps y amarillo para 1 Gbps.

El LED de la derecha (de color ámbar) parpadea cada vez que se transmite o recibe una trama.

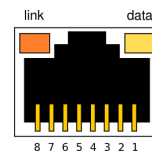


FIGURA 2.2: Conector RJ-45 hembra de un equipo Ethernet

Aparte de la comprobación visual anterior, es posible utilizar software que interroga al controlador de dispositivo. En el caso de Ethernet puedes utilizar el programa `ethtool`. Lo siguiente es la salida de dicho comando para una interfaz Ethernet de un PC conectada a un conmutador.

```

1  # ethtool eth0
2  Settings for eth0:
3  Supported ports: [ TP ]
4  Supported link modes:   10baseT/Half 10baseT/Full
5                           100baseT/Half 100baseT/Full
6                           1000baseT/Half 1000baseT/Full
7  Supports auto-negotiation: Yes
8  Advertised link modes:  10baseT/Half 10baseT/Full
9                           100baseT/Half 100baseT/Full
10                          1000baseT/Half 1000baseT/Full
11  Advertised pause frame use: No
12  Advertised auto-negotiation: Yes
13  Speed: 100Mb/s
14  Duplex: Full
15  Port: Twisted Pair
16  PHYAD: 1
17  Transceiver: internal
18  Auto-negotiation: on
19  MDI-X: Unknown
20  Supports Wake-on: g
21  Wake-on: g
22  Current message level: 0x000000ff (255)
23  Link detected: yes

```

Como puedes comprobar se trata de una tarjeta Gigabit (línea 4) que está conectada a un conmutador (línea 13)¹ de 100 Mbps (línea 12) y el enlace de datos está correctamente establecido (línea 22).

E 2.01 Desconecta el cable Ethernet, vuelve a ejecutar `ethtool` y compara con la salida del comando cuando el cable estaba conectado ¿Qué datos han cambiado? ¿Qué significan?

¹Sólo se puede establecer un enlace *full duplex* si se trata de Ethernet conmutada, es decir, el PC está conectado a un conmutador (*switch*).

E 2.02 Manteniendo el cable desconectado, ejecuta el comando `ip addr` y compara con la salida del comando cuando el cable estaba conectado. ¿Qué ha cambiado? ¿Qué significa?

2.7. Conectividad a nivel de red

Una vez que has verificado que tu computador puede mantener un enlace de datos con su vecino (el switch en este caso) puedes pasar a comprobar si es posible enviar (y recibir) tráfico IP a otro PC, ya sea un vecino o un computador al otro lado del planeta.

2.7.1. ping

La solución para comprobar que efectivamente puedes enviar tráfico a un destino remoto es que él te responda con una confirmación. Esa es una de las funciones del protocolo ICMP (Internet Control Message Protocol) [Pos81a]. Concretamente nos interesa uno de sus tipos, el «Echo Request» (comúnmente llamado «ping»).

El «ping» es un mensaje que se encapsula dentro de un paquete IP y se envía al destino indicado: la dirección que aparezca en la cabecera del paquete IP. Cuando el computador destino lo recibe, fabrica un «ICMP Echo Reply» y lo envía de vuelta al remitente.

Cuando tu equipo recibe la respuesta puedes estar seguro de que el tráfico de paquetes entre ese destino y tu computador es posible y, obviando la presencia de cortafuegos u otros mecanismos de control, debería funcionar para cualquier otro protocolo que se encapsule sobre IP. Una consecuencia interesante de este funcionamiento es que es posible calcular el RTT (Round Trip Time) (Round Trip Time) o «tiempo de vuelo», es decir, el tiempo transcurrido desde que envió la petición hasta que recibió la respuesta.

Prueba a ejecutar ping hacia tu pasarela de enlace:

```
1 $ ping 161.67.101.1
2 PING 161.67.101.1 (161.67.101.1) 56(84) bytes of data.
3 64 bytes from 161.67.101.1: icmp_req=1 ttl=255 time=2.02 ms
4 64 bytes from 161.67.101.1: icmp_req=2 ttl=255 time=0.730 ms
5 64 bytes from 161.67.101.1: icmp_req=3 ttl=255 time=1.66 ms
6 ^C
7 — 161.67.101.1 ping statistics —
8 3 packets transmitted, 3 received, 0% packet loss, time 2000ms
9 rtt min/avg/max/mdev = 0.730/1.471/2.023/0.546 ms
```

Por defecto, el programa enviará mensajes «ping» de forma indefinida (uno por segundo) hasta que pulses Control-C (eso es lo que ha pasado en la línea 6).

Las líneas 3 a 5 aparecen al recibirse el mensaje de respuesta. Para cada una se muestra el número de secuencia (`icmp_req`), el TTL (Time To Live) del paquete IP y el RTT (en milisegundos). La máquina destino está muy cerca (es un vecino) por lo que ronda los 2 ms.

Cuando el programa termina muestra unas estadísticas sobre lo ocurrido (línea 8). Aparece el número de peticiones enviadas, respuestas recibidas, porcentaje de peticiones perdidas (sin respuesta) y el tiempo total que ha llevado.

En la última línea aparecen los RTT mínimo, medio, máximo y la desviación estándar.

- E 2.03** Ejecuta ping hacia «localhost» y hacia la IP de tu propio equipo. ¿Hay alguna diferencia? ¿Por qué?
- E 2.04** Ejecuta ping hacia otros equipos de tu misma red. Pide a un compañero que te diga cuál es la dirección IP de su equipo. Toma nota de los resultados obtenidos.
- E 2.05** Ejecuta ping hacia equipos remotos como uclm.es, google.com, microsoft.com, ibm.com, whitehouse.gov, traffic2.kci.net.nz, etc. Toma nota de los resultados. ¿Por qué no hay resultados para algunos de ellos?

Estás son algunas opciones interesantes para modificar el comportamiento del programa:

- c** Envía la cantidad de peticiones que se indiquen y termina.
 - i** Permite fijar el intervalo entre envíos. Se especifica en segundos y admite decimales.
 - A** Modo adaptativo, calcula el intervalo en función del RTT medido.
 - f** Envía la siguiente petición tan pronto como vuelve la respuesta anterior hasta un máximo de 100 por segundo. Requiere privilegios de administrador.
- E 2.06** Ejecuta ping hacia destinos cercanos y lejanos con distintos valores para el parámetro **-i**. ¿Cuál es la diferencia? ¿Por qué crees que determinados valores precisan de privilegios de administrador?
 - E 2.07** Ejecuta ping hacia destinos cercanos y lejanos con distintos valores para el parámetro **-A**. ¿Cuál es la diferencia? ¿Por qué crees que determinados valores precisan de privilegios de administrador?

2.7.2. ping «extendido»

Algunas implementaciones de ping proporcionan opciones para características avanzadas. En realidad, esas opciones se corresponden con parámetros de los mensajes, tales como el valor del TTL, el tamaño de la carga del mensaje ICMP, etc. Veamos algunos:

- p** Permite indicar el contenido de los bytes de relleno de la carga del paquete ICMP.
 - R** Graba y muestra la ruta seguida por la petición y su respuesta.
 - s** Indica el tamaño (en bytes) de la carga útil del mensaje de petición. El tamaño por defecto es 56 bytes.
 - t** Fija el valor del TTL de la cabecera IP en los mensajes de petición.
- E 2.08** Ejecuta ping hacia destinos cercanos y lejanos con distintos valores para el parámetro **-s**. ¿Cuál es la diferencia? ¿Por qué crees que determinados valores precisan de privilegios de administrador?

2.7.3. Otros *ping*

Algunas variantes del programa *ping* disponibles en sistemas GNU:

fping

La diferencia principal respecto al *ping* convencional es que permite especificar múltiples destinos que serán comprobados secuencialmente, de modo que se puede utilizar para determinar cuáles de ellos están accesibles.

oping

También permite especificar múltiples destinos pero las peticiones son enviadas en paralelo.

2.7.4. *traceroute*

Cuando un encaminador recibe un paquete IP decrementa el valor del campo TTL de su cabecera; si ese valor es cero el encaminador descarta el paquete e informa al emisor de este hecho. En concreto, le envía un mensaje ICMP de tipo *Time Exceeded* con el código 0:«TTL expired in transit».

traceroute es un programa que aprovecha este mecanismo para descubrir la ruta que sigue un paquete IP hacia su destino. *traceroute* comienza enviando un mensaje ICMP *Echo* o bien un segmento UDP o TCP encapsulado en un paquete IP con un valor de TTL=1. Eso provoca que la pasarela de enlace descarte el paquete, informe del error y con ello revele su dirección IP (aparece como dirección origen del mensaje de error). A continuación, se repite la operación con TTL=2 obteniendo la dirección del segundo encaminador, y así sucesivamente hasta alcanzar el destinatario.

A continuación aparece el resultado de una ejecución de *traceroute* dirigida a rediris.es desde la ESI de Ciudad Real:

```

1  $ traceroute rediris.es
2  traceroute to rediris.es (130.206.13.20), 30 hops max, 60 byte packets
3  1  161.67.101.1 (161.67.101.1)  0.694 ms  1.015 ms  1.256 ms
4  2  172.16.160.5 (172.16.160.5)  1.153 ms  1.465 ms  1.710 ms
5  3  ro-vlan170.ctic.cr.red.uclm.es (172.16.160.22)  0.514 ms  0.517 ms  0.571 ms
6  4  GE1-2-0.EB-CiudadReal0.red.rediris.es (130.206.200.1)  0.803 ms  0.939 ms  1.073 ms
7  5  CLM.S01-1-1.EB-IRIS4.red.rediris.es (130.206.250.133)  4.529 ms  4.568 ms  4.656 ms
8  6  XE3-0-0-264.EB-IRIS6.red.rediris.es (130.206.206.133)  4.806 ms  4.318 ms  4.332 ms
9  7  www.rediris.es (130.206.13.20)  4.456 ms  4.465 ms  4.518 ms

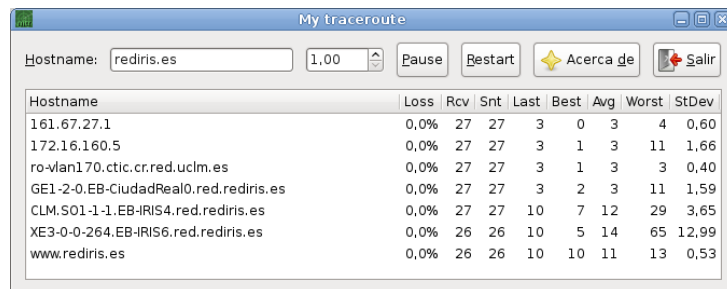
```

El primer resultado es para un TTL=1 (línea 3) y corresponde a la puerta de enlace (161.67.101.1). Al lado aparecen Los tiempos RTT para cada uno de los tres paquetes enviados. Para cada encaminador aparece primero su nombre simbólico (si tiene) y después su dirección IP. Como se puede apreciar la ruta completa requiere 7 saltos, siendo el último el destino solicitado.

E 2.09 Ejecuta *traceroute* hacia equipos remotos como uclm.es, google.com, microsoft.com, ibm.com, whitehouse.gov, traffic2.kci.net.nz, etc. Toma nota de los resultados. ¿Cuántos saltos se necesitaron para cada uno de ellos? ¿Hay partes comunes en las rutas? ¿Por qué?

2.7.5. mtr

`mtr` es muy similar a `traceroute` pero puede utilizarse con interfaz gráfica, tal como se muestra en la Figura 2.3.



Hostname	Loss	Rcv	Snt	Last	Best	Avg	Worst	StDev
161.67.27.1	0,0%	27	27	3	0	3	4	0,60
172.16.160.5	0,0%	27	27	3	1	3	11	1,66
ro-vlan170.ctic.cr.red.uclm.es	0,0%	27	27	3	1	3	3	0,40
GE1-2-0.EB-CiudadReal0.red.rediris.es	0,0%	27	27	3	2	3	11	1,59
CLM.S01-1-1.EB-IRIS4.red.rediris.es	0,0%	27	27	10	7	12	29	3,65
XE3-0-0-264.EB-IRIS6.red.rediris.es	0,0%	26	26	10	5	14	65	12,99
www.rediris.es	0,0%	26	26	10	10	11	13	0,53

FIGURA 2.3: Aspecto del programa `mtr`

2.8. Conectividad a nivel de transporte

En el nivel de transporte, tener conectividad implica poder conectar a un servidor, ya sea TCP o UDP, y poder transmitir datos satisfactoriamente en ambos sentidos. En determinadas situaciones, debido a las políticas de acceso de los cortafuegos, tu equipo podrá ejercer el papel de cliente (el que inicia la conexión) y no el de servidor (el que espera la conexión).

2.8.1. netcat

La herramienta por excelencia para probar conectividad TCP o UDP es `netcat`. Este programa puede funcionar como cliente o servidor (dependiendo de las opciones que se le indiquen) y una vez establecida la comunicación permite que otro programa proporcione la fuente y el sumidero de los datos.

En la configuración más simple `netcat` lee datos de su *entrada estándar* (normalmente el teclado) y los envía al computador remoto. Al mismo tiempo, los datos que recibe desde la red los envía a su *salida estándar* (normalmente la consola).

Como existen varias implementaciones de `netcat`, no siempre compatibles, se aconseja `ncat` (del paquete `nmap`), una versión moderna y con un desarrollo activo. A pesar de usar `ncat` es los siguientes ejemplos, nos referiremos a esta herramienta por su nombre genérico: `netcat`.

Por ejemplo, para probar que es posible establecer una comunicación entre los computadores `storm` y `magneto` puedes ejecutar los siguientes comandos. En `storm`:

```
pepa@storm:~$ ncat -l 9000
```

Esto arranca un *servidor* (-l) TCP en el puerto 9000 y queda a la espera de conexiones. Para probarlo, en `magneto` ejecuta:

```
paco@magneto:~$ ncat 161.167.101.12 9000
```

Esto ejecuta un *cliente* que conecta a `magneto` asumiendo que esa (161.67.101.12) es su dirección IP.

El programa (en ambos extremos) queda a la espera de que se introduzcan datos a través de la entrada estándar; de modo que simplemente teclea algo y pulsa **enter**. Si la conexión TCP se ha establecido correctamente, lo que has escrito debería aparecer en el otro computador. Esto puede ocurrir en cualquier momento y de forma simultánea.

E 2.10 Prueba el ejemplo anterior con dos terminales pero conectando el cliente a la dirección [localhost](#). Prueba también utilizando tu computador y el de algún compañero.

E 2.11 Añade la opción `-u` a ambos extremos (cliente y servidor). ¿Qué ocurre si el primero en escribir es el servidor? ¿Ocurría lo mismo con el caso anterior usando TCP.

`netcat` es una herramienta extraordinariamente flexible. Se puede utilizar para cosas tan dispares como clonar un disco duro remoto, enviar *streaming* multimedia o ejecutar una shell remota. A continuación aparecen algunas de las opciones de `ncat` (hay diferencias respecto a otras implementaciones de `netcat`):

- 6** Usar IP versión 6 en lugar de IP versión 4.
- e/-c** Ejecuta el comando indicado. Dicho comando hereda el socket conectado y lo utiliza como su E/S estándar. Eso permite convertir cualquier programa de terminal en un servidor o cliente TCP/UDP. `-c` ejecuta un comando de shell en lugar de un fichero ejecutable.
- m** Número máximo de conexiones simultáneas.
- o/-x** Volcar la sesión (tráfico enviado/recibido) a un fichero. `-x` lo vuelca en formato hexadecimal.
- proxy**
Permite realizar la conexión a través del proxy indicado.

2.9. Conectividad a nivel de aplicación

Si has comprobado la conectividad a nivel de transporte para un protocolo y puerto concreto, lo más probable es que la aplicación que utiliza dicho puerto funcione correctamente.

Por ejemplo, si `netcat` ejecutado en la máquina [magneto](#) puede conectar, enviar y recibir datos a otro `netcat` que funciona como servidor en el puerto 80 de la máquina [storm](#) lo habitual será que esos `netcat` puedan ser reemplazados por un servidor y un navegador web y todo funcione perfectamente.

La única excepción se presenta cuando la red cuente con cortafuegos del nivel de aplicación configurados para impedir o filtrar protocolos de aplicación concretos.

Por tanto, `netcat` también puede utilizarse para comprobar la conectividad a nivel de aplicación. Como ejemplo, prueba a conectar con el servidor web de [insecure.org](#) (los creadores de `ncat`):

```
$ echo "GET /" | ncat insecure.org 80
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
```

```
<HEAD>
<TITLE>
  Nmap — Free Security Scanner For Network Exploration & Security
  Audits.
</TITLE>
[...]
</BODY>
</HTML>
```

El comando va precedido de `echo "GET /"`. Esto corresponde con el mensaje HTTP para solicitar la página raíz. El resultado, como era de esperar, es la página web que corresponde al fichero `index.html` del sitio web de insecure.org.

Se puede utilizar `netcat` del mismo modo para interactuar con cualquier protocolo de aplicación textual, como pueden ser SMTP, POP3 (Post Office Protocol) o IRC (Internet Relay Chat).

Para el caso concreto de HTTP existen algunos programas de terminal muy potentes, concretamente `wget` y `curl`. Están pensados como clientes HTTP genéricos pero no son navegadores (no renderizan HTML) y resultan muy adecuados para probar que determinados contenidos están accesibles o no, entre otras muchas cosas.

En el siguiente ejemplo puedes ver cómo usar `wget` para descargar el logotipo de google directamente desde su localización.

```
$ wget http://www.google.com/images/google_sm.gif
```

En esos ejemplos se comprueba que un servidor web remoto funciona correctamente, pero también puede ser necesario probar un cliente. Puedes hacerlo utilizando `netcat` como servidor «web» del siguiente modo:

```
$ ncat -l 8000 -c date
```

Si pones la dirección `localhost:8000` en la barra de direcciones de tu navegador verás que muestra la hora actual, es decir, el resultado de ejecutar `date` en un terminal.

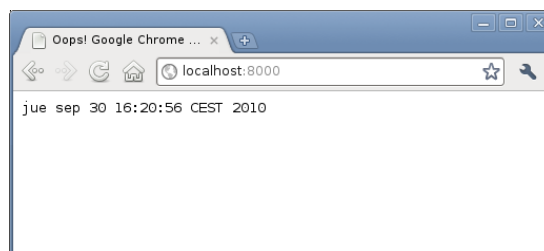


FIGURA 2.4: *Chromium* muestra la «página web» que devuelve `ncat`

En realidad en este caso no entra en juego ningún protocolo de aplicación, dado que `netcat` coloca el texto que devuelve `date` directamente sobre un segmento TCP. Sin embargo, el navegador web lo representa sin problema como se puede comprobar en la Figura 2.4.

Pila de protocolos TCP/IP

David Villa

Una pila de protocolos es el software que implementa las comunicaciones en una red de computadores conforme a un modelo de referencia. Se denomina «pila de protocolos»¹ porque está formado por un conjunto de protocolos «apilados» según las especificaciones del modelo correspondiente.

En el caso de la pila de protocolos TCP/IP los protocolos más importantes (agrupados por capas) se muestran en la siguiente figura. Nótese que el modelo de referencia TCP/IP no especifica los protocolos de la capa de enlace. De los que aparecen en la figura sólo ARP (Address Resolution Protocol) se puede considerar un protocolo TCP/IP²

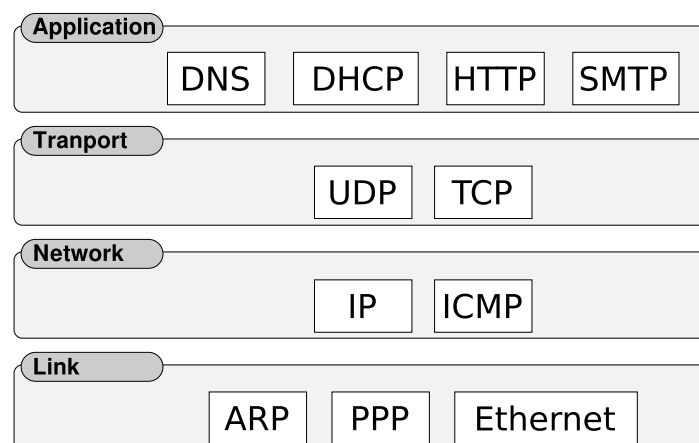


FIGURA 3.1: Pila de protocolos TCP/IP

Un protocolo define las pautas y normas específicas que se deben aplicar para efectuar una comunicación correcta³. El protocolo define normalmente el formato y tamaño de los mensajes, el significado de cada campo, los rangos de valores admisibles, etc.

En este capítulo se introduce el formato y descripción básica de los protocolos más relevantes, las relaciones entre ellos y el mecanismo de encapsulación de mensajes. La implementación de estos protocolos puede ser muy variopinta: Los protocolos de enlace se implementan típicamente en el hardware de la NIC, los de red y transporte en el subsistema de red en el núcleo del sistema operativo y los de aplicación están disponibles en forma de librerías, o en

¹En inglés: *protocol stack*

²Debido a su función, muchos autores consideran ARP como un protocolo de la capa de red.

³Similar a la acepción que se aplica en las relaciones diplomáticas.

las propias aplicaciones si son de uso muy específico. Pero todo eso puede cambiar de un sistema a otro, por ejemplo los routers de gama media/alta implementan la mayoría de su funcionalidad (incluidos los protocolos de red) por medio de ASIC (Application-Specific Integrated Circuit) para conseguir mayor productividad y menor consumo.

En este capítulo

Al acabar este capítulo el lector debería ser capaz de abordar satisfactoriamente las siguientes cuestiones:

- Comprensión básica del formato y funcionamiento de los protocolos Ethernet, IP, ICMP, ARP, UDP y TCP.
- La encapsulación de mensajes y relaciones entre los protocolos básicos.
- Direccionamiento lógico y físico.

3.1. Protocolos esenciales

Esta sección describe brevemente los protocolos más importantes relacionados con TCP/IP.

3.1.1. Ethernet

La capa de enlace de datos se encarga de llevar mensajes desde un computador o equipo de comunicaciones hasta sus vecinos⁴.



Para TCP/IP, la capa de enlace simplemente se encarga —por medio del protocolo correspondiente— de llevar paquetes IP entre vecinos de la misma red.

Dependiendo del medio físico se distingue entre medios:

punto a punto que conectan pares de vecinos, y

de difusión (*broadcasting*) que permiten enviar la misma información a un conjunto determinado de vecinos, puesto que el medio es compartido por todos ellos.

Los medios de difusión son típicos de las redes LAN aunque se da en otros ámbitos, como son las comunicaciones satelitales.

Las tecnologías LAN más utilizadas con diferencia en la actualidad son Ethernet IEEE 802⁵ y WiFi⁶, que salvando las distancias, podríamos ver como una «ethernet inalámbrica».

Ethernet cubre los detalles tecnológicos de la transmisión de señales (la capa física) pero también la funcionalidad, codificación y formato de los mensajes (la capa de enlace) que es el tema que se aborda aquí. El formato de las tramas Ethernet es el que aparece en la figura 3.2.

⁴dispositivos cercanos que pueden ser alcanzados sin ayuda de intermediarios

⁵<http://wikipedia.org/wiki/Ethernet>

⁶<http://wikipedia.org/wiki/Wifi>

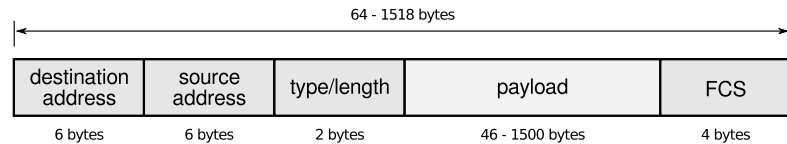


FIGURA 3.2: Formato de la trama Ethernet

Como muestra la figura 3.2, la trama Ethernet incluye:

- Las direcciones MAC de la tarjeta emisora y receptora,
- El tipo del contenido de la trama (normalmente el protocolo al que corresponde),
- La carga útil (de tamaño variable) y
- El FCS (Frame Check Sequence), una especie de suma de comprobación que permite al receptor comprobar si la trama es correcta.

Obviando todos los detalles del funcionamiento de Ethernet, el servicio que presta a la capa de red es bastante simple: Lleva la trama hasta el computador al que esté conectada la NIC con la dirección destino que aparece en la cabecera.

Esa dirección es un número grabado en la memoria ROM de la NIC. Ese número se llama «dirección MAC» y se asume que no puede haber dos tarjetas Ethernet con el mismo número⁷. En concreto la dirección MAC es un número de 6 bytes que se suele representar como lista de cifras en hexadecimal separadas por el carácter ':', por ejemplo: 00:22:34:6b:c5:47.

Dirección física

Las direcciones que están grabadas en la memoria (normalmente ROM del dispositivo electrónico) se denominan «direcciones físicas» o «direcciones hardware».

La dirección MAC está formada por dos partes:

OUI (Organizationally Unique Identifier) (24 bits) Identifica al fabricante de la tarjeta. Este prefijo debe solicitarse a la IANA (Internet Assigned Numbers Authority). La lista de todos los OUI asignados actualmente se puede consultar en su web⁸.

NIC Es un número de serie que el fabricante garantiza que es único en su producción.

Podemos ver la trama Ethernet como un contenedor en el que meter una secuencia de bytes arbitraria (el campo *payload*) y Ethernet se encargará de llevar esos bytes hasta el computador destino. La NIC emisora calcula el FCS y lo añade al final. Al llegar a sus destino, la NIC receptora realiza el mismo cálculo, si corresponde con el valor FCS la trama es correcta y será entregada a la capa de red del computador destino. Pero si el resultado de ese cálculo no corresponde, la trama será descartada. Y eso es todo... Ethernet no pide una retransmisión, no hay temporizadores ni reconocimientos. Si una trama se corrompe durante su viaje, se descarta sin más. Obviamente habrá muchas aplicaciones que no puedan tolerar ese comportamiento, pero ese tipo de confiabilidad es tarea de los protocolos de transporte, no de Ethernet.

⁷Este tipo de identificación se denomina «identificador único global» o UUID (Universally Unique Identifier)

⁸<http://www.iana.org/assignments/ethernet-numbers>

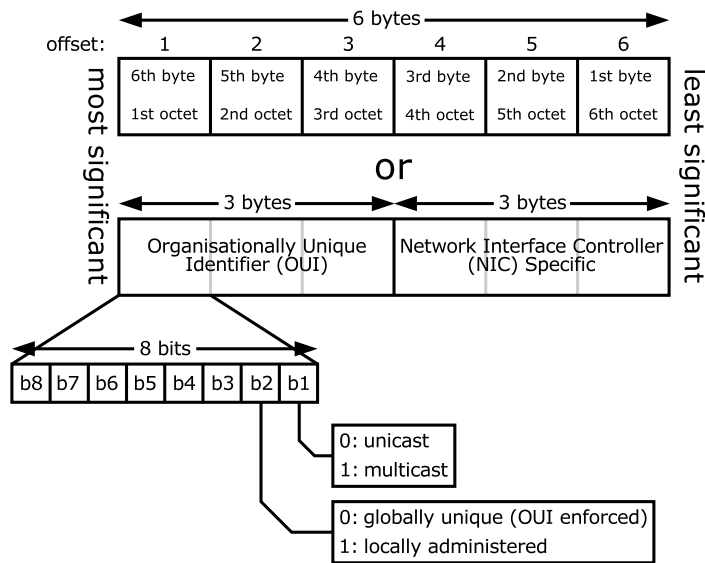


FIGURA 3.3: Formato de la dirección MAC Ethernet [Fuente:Wikimedia Commons]

Aparte de la entrega para un único destinatario (*unicast*) Ethernet soporta otros dos métodos de entrega:

broadcast Permite enviar un mensaje a todos los vecinos conectados a la misma LAN. Para ello se utiliza como destino una dirección especial que tiene todos sus bits a uno; en hexadecimal FF:FF:FF:FF:FF:FF.

multicast Permite enviar un mensaje a un subconjunto de los vecinos. Para ello se utilizan direcciones que tienen prefijos reservados.

3.1.2. IP

La gran aportación de Internet está definida en su propio nombre: inter-red. La capacidad de interconectar limpiamente redes con distintas tecnologías LAN y WAN es su principal valor. Lo consigue gracias a tres cosas:

- Un esquema de direccionamiento jerárquico global: El direccionamiento IP.
- Un protocolo de red único, que han de implementar tanto los nodos terminales como los dispositivos de interconexión de redes (los routers). Ese protocolo es IP.
- Un sistema de encaminamiento que permite llevar paquetes desde cualquier punto de la interred a cualquier otro.

La dirección IP es un número de 32 bits que habitualmente se representa con los valores en decimal de los 4 bytes que la forman separados por puntos, por ejemplo, 161 . 67 . 136 . 169. A diferencia de las direcciones MAC las direcciones IP tienen una estructura jerárquica, es decir, todos dispositivos conectados a una red comparten el mismo prefijo. Las redes pequeñas (con pocos computadores) tienen prefijos más largos.

Este direccionamiento jerárquico es lo que permite que los routers puedan hacer su trabajo de forma más eficiente ya que las tablas de rutas no necesitan incluir las direcciones de todos los posibles destinos, sólo los prefijos comunes. la figura 3.4 muestra el formato de las direcciones IP.

FIGURA 3.4: Formato de las direcciones IP

IP es un protocolo de red, es decir, es capaz de llevar un paquete IP de cualquier punto de Internet a cualquier otro. A esto se llama entrega extremo a extremo (*end-to-end*). Pero IP es un protocolo *best-effort*, es decir, que tampoco garantiza que pueda hacer llegar el paquete a su destino.

Como en Ethernet, aparte del direccionamiento unicast, hay otros modos:

broadcast Es posible direccionar, al menos en teoría, todos los hosts de una red si todos los bits del *host-id* tienen valor uno.

multicast Se puede direccionar un grupo arbitrario de hosts utilizando direcciones de grupo (aquellas cuyo primer byte es 240). Este mecanismo no es trivial ya que implica la subscripción activa de los hosts a los grupos y requiere soporte en los routers que deben propagar la información de membresía. Para ello se utiliza un protocolo específico llamado IGMP (Internet Group Management Protocol) [Fen97].

El formato del paquete IP [Pos81b], que aparece en la figura 3.5 es también bastante simple.

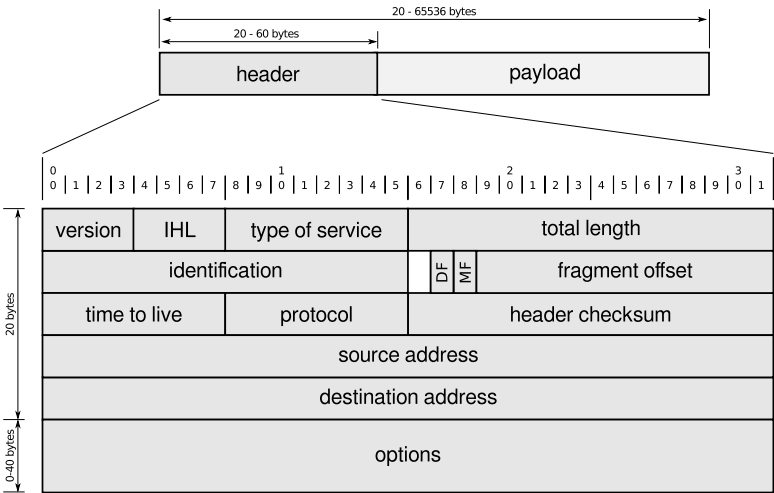


FIGURA 3.5: Formato del paquete IP

A continuación se describe brevemente el significado de cada campo:

Version

Versión del protocolo. Siempre contiene el valor 4.

IHL Internet Header Length, es decir, el tamaño de la cabecera, pero expresado en palabras de 4 bytes. Eso significa que el valor mínimo posible es 5 (una cabecera estándar de 20 bytes) y el máximo es 15 (una cabecera de 60 bytes).

ToS Type of Service contiene información útil para priorizar el tráfico en función de su naturaleza.

Total length

La longitud total del paquete IP incluyendo cabecera y carga útil. Como es un entero de 16 bits implica que el tamaño máximo de un paquete IP es 65 535 bytes.

Identification

Es un número que identifica a todos los fragmentos que proceden un mismo paquete IP original.

DF Don't Fragment le indica a los routers que no deben fragmentar este paquete.

MF More Fragments indica que éste no es el último fragmento.

Fragment offset

Indica qué posición ocupa este fragmento en el paquete original.

Time to live

Indica cuantos routers puede atravesar este paquete (saltos) antes de ser descartado.

Protocolo

Indica el protocolo al que corresponde la carga útil del paquete.

Header checksum

Una suma de comprobación para verificar que la cabecera no ha sufrido cambios inesperados.

Source address

La dirección IP del host que crea el paquete.

Destination address

La dirección IP del destinatario del paquete.

3.1.3. UDP**3.1.4. TCP****3.2. Encaminamiento****3.2.1. Entrega directa****3.2.2. Entrega indirecta****3.2.3. Encaminador por defecto****3.3. Ejercicios****3.3.1. Enlace**

E 3.01 Escribe un programa Python que realice el entramado de un flujo de datos, utilizando delimitadores con relleno de bytes siguiendo la especificación de SLIP. El tamaño de trama debe ser configurable. Para probar el programa se aconseja utilizar ficheros binarios como fuentes

y sumideros de datos. Para poder probar que el programa es correcto se debe realizar también otro programa que haga el «desentramado».

- E 3.02** Sobre el ejercicio anterior, implementar un algoritmo de detección de errores usando CRC32.
- E 3.03** Sobre el ejercicio **E 3.01** , implementar un algoritmo de corrección de errores Hamming.
- E 3.04** Escribe un programa que acepte una dirección MAC Ethernet por línea de comandos e imprima el tipo de dirección y el OUI al que corresponde.
- E 3.05** Escribe un programa que dado un fichero binario que contiene una trama Ethernet (en formato binario) imprima el tipo de trama, las direcciones origen y destino, el valor del campo tipo y el tamaño en bytes del payload.

3.3.2. Red

- E 3.06** Escribe un programa que a partir de un fichero binario que contiene un paquete IP, imprima en pantalla el valor de cada campo de la cabecera en un formato adecuado.
- E 3.07** Escriba un programa que a partir de una captura de paquetes IP, calcule el checksum y compruebe si corresponde con el capturado.
- E 3.08** Escriba un programa que acepte por línea de comandos una IP y una máscara en notación CIDR, por ejemplo "161.67.27.18/26." imprima en su salida:
 - La dirección de red a la que pertenece esa IP.
 - La dirección de broadcast de esa red.
 - La lista completa de las IPs que pueden ser asignadas a hosts (una por línea).
- E 3.09** Escriba un programa que acepte una dirección IP por línea de comandos e informe si esa dirección es pública, privada o multicast.
- E 3.10** Dada una tabla de rutas expresada del siguiente modo:

```

1      table = [
2          # destination  mask      next-hop  iface
3          ('120.2.10.0', 24,      '0.0.0.0', 'e0'),
4          ('30.12.0.0',  22,      '30.12.0.1', 'e1'),
5          ('160.8.0.0',  20,      '160.8.0.1', 'e0'),
6          ('0.0.0.0',    0,       '120.2.10.1', 'e0')
7      ]

```

Escriba un programa Python que a partir de una dirección IP indique cuál es la interfaz de salida.

3.3.3. Transporte

- E 3.11** Servidor de procesos que implemente internamente servidores de echo, daytime, time y discard tanto UDP como TCP. Debe poder utilizar servidores estándar de ftp, tftp y telnet.

- E 3.12** Escanner de puertos, es decir, un programa que permite averiguar qué puertos tiene abiertos/cerrados/filtrados una máquina remota. Ejemplo de invocación:

```
$ scanner.py www.uclm.es 20-2000.
```

- E 3.13** Servidor Echo concurrente (TCP o UDP). Ejemplo de invocación:

```
$ echo_server.py --tcp 2000
```

- E 3.14** Servidor Daytime concurrente (TCP o UDP). Ejemplo de invocación:

```
$ daytime_server.py --udp 2000
```

- E 3.15** Servidor Time concurrente (TCP o UDP). Ejemplo de invocación:

```
$ time_server.py --udp 2000
```

- E 3.16** Servidor Discard concurrente (TCP o UDP).

```
$ discard_server.py --tcp 2000
```

- E 3.17** Cliente que explote la vulnerabilidad SYN Flood.

- E 3.18** Programa que implemente ARP spoofing. El programa debe aceptar por línea de parámetros la IP del objetivo y del host que le suplanta.

Captura y análisis de tráfico de red

David Villa

Cuando una aplicación de red no funciona como se espera, tener la posibilidad de observar los mensajes que aparecen en la red por su causa, puede ser determinante para localizar la causa. Además de la depuración y optimización de protocolos y aplicaciones de red, capturar y visualizar el tráfico de red es muy útil para entender cómo funcionan los procesos de direccionamiento, encapsulación y conectividad a nivel de enlace y red.

Para lograrlo es necesario un programa que capture, diseccione y filtre los mensajes a fin de encontrar cuáles de ellos son relevantes y cuál es su contenido. Este tipo de programas se denominan *sniffers* o analizadores de protocolos.

4.1. tshark

Uno de esos analizadores de protocolos es *tshark*. Es la versión para línea de comando del programa *wireshark* (que también veremos) y su uso es similar a *tcpdump*. La forma más sencilla de entender lo que hace es verlo en funcionamiento. El listado 4.1 muestra *tshark* capturando el tráfico ICMP correspondiente al comando:

```
$ ping ietf.org &
```

LISTADO 4.1: *tshark* capturando tráfico ICMP

```
1 $ sudo tshark -i eth0 -f icmp
2 Capturing on wlan0
3 0.000000 192.168.2.49 -> 64.170.98.30 ICMP 98 Echo (ping) request id=0x405d, seq=10/2560, ttl=64
4 0.218770 64.170.98.30 -> 192.168.2.49 ICMP 98 Echo (ping) reply id=0x405d, seq=10/2560, ttl=60
5 1.000830 192.168.2.49 -> 64.170.98.30 ICMP 98 Echo (ping) request id=0x405d, seq=11/2816, ttl=64
6 1.224505 64.170.98.30 -> 192.168.2.49 ICMP 98 Echo (ping) reply id=0x405d, seq=11/2816, ttl=60
7 2.000749 192.168.2.49 -> 64.170.98.30 ICMP 98 Echo (ping) request id=0x405d, seq=12/3072, ttl=64
8 2.219841 64.170.98.30 -> 192.168.2.49 ICMP 98 Echo (ping) reply id=0x405d, seq=12/3072, ttl=60
9 6 packets captured
```

El significado de los argumentos es:

-i eth0

Capturar tráfico de la interfaz *eth0*.

-f icmp

Filtrar el protocolo ICMP.

-c 6

Capturar únicamente 6 paquetes y terminar.

La información de salida está organizada en columnas, que corresponden con:

1. Instante de captura, respecto al primer mensaje.

2. Dirección IP origen.
3. Dirección IP destino.
4. Protocolo de la carga útil.
5. Tamaño total del mensaje.
6. Tipo de mensaje.
7. Identificador ICMP.
8. Número de secuencia ICMP.
9. TTL.

Mediante estos argumentos (y muchos otros) es posible afinar la captura con un alto grado de detalle. Además de capturar tráfico de una interfaz de red e imprimir un resumen como el anterior, *tshark* también puede leer tráfico almacenado en un fichero (`-r`) y, por supuesto, también crear este tipo de ficheros para un análisis posterior (`-w`).

4.1.1. Advertencia de seguridad

Quizá haya advertido en la línea 1 del listado 4.1 el uso del comando *sudo*. Esto se debe a que para capturar tráfico «crudo» de la interfaz de red se requieren privilegios especiales. El comando *sudo* sirve para ejecutar un comando con los privilegios de otro usuario (normalmente *root*). Sin embargo, esa es una mala solución. Ejecutar cualquier programa con privilegios de administrador es siempre un riesgo de seguridad, de hecho, así lo indica el propio *tshark*:

```
Running as user "root" and group "root". This could be dangerous.
```

Existe una alternativa. El núcleo Linux permite asignar «capacidades» (*capabilities*) concretas a un programa (un fichero binario). Las capacidades requeridas son *CAP_NET_ADMIN* y *CAP_NET_RAW*. Para aplicarlas al programa que realiza las capturas se debe ejecutar:

```
$ sudo setcap cap_net_raw,cap_net_admin=eip /usr/bin/dumpcap
```

Esta solución permite a cualquier usuario del sistema ejecutar el programa *dumpcap* con los citados privilegios. Una alternativa más conservadora es otorgarlos sólo a los miembros de un nuevo grupo. Consulte <http://wiki.wireshark.org/CaptureSetup/CapturePrivileges> para más detalles sobre esta última opción.

4.1.2. Limitando la captura

Si no se le indica lo contrario, *tshark* continuará capturando tráfico indefinidamente, ya sea mostrando el resumen en consola o almacenándolo en un fichero.

Es posible limitar la captura de varias formas:

-c N un número de paquetes (que cumplen los filtros).

-a duration:N

durante un número de segundos.

-a filesize:N

un tamaño de fichero, indicando en KiB.

-a files:N

un número de ficheros.

El siguiente comando captura todo el tráfico de red de la interfaz wlan0 y lo almacena en un fichero llamado wlan0.pcap hasta un máximo de 1 MiB.

```
$ tshark -i wlan0 -w wlan0.pcap -a filesize:1024
```

Durante el proceso, tshark mostrará en consola el número de paquetes capturados hasta el momento.

4.1.3. Filtros de captura

Como se ha indicado, el argumento `-f` sirve para especificar un filtro de captura. El formato de estos filtros constituye todo un lenguaje en sí mismo. La tabla 4.1 contiene unos cuantos ejemplos extraídos de <http://wiki.wireshark.org/CaptureFilters>. Puede encontrar información detallada en la página de manual de `pcap-filter`.

Filtro	Significado
ip	tráfico IP
ip or arp	IP o ARP
tcp and not dst port 80	cualquier protocolo sobre TCP excepto el dirigido al puerto 80 (HTTP)
host 192.168.0.1	hacia o desde la IP indicada
net 192.168.0.0/24	hacia o desde la red indicada
net 192.168	
src host 192.168.0.1	procedente del host o red indicado
src net 192.168	
port 22	TCP o UDP hacia o desde el puerto 22
tcp dst port 22	hacia el puerto 22 TCP (SSH)
dst host 10.0.0.1 and port 443 and http	tráfico HTTP con destino al puerto 443 del host 10.0.0.1
not ether dst FF:FF:FF:FF:FF:FF	tramas Ethernet no broadcast

CUADRO 4.1: tshark: ejemplos de filtros de captura

4.1.4. Formato de salida

Por defecto el programa muestra una línea que resume la información más importante de cada mensaje, tal como aparece en los ejemplos anteriores. Pero existen otras muchas posibilidades para obtener información sobre los mensajes. La opción `-V` muestra todos los detalles de cada mensaje (y cada capa) en un formato de texto legible (vea el listado 4.2).

LISTADO 4.2: Modo «verboso» de tshark

```

1  $ tshark -f icmp -c 1 -V
2  Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
3      Interface id: 0
4      WTAP_ENCAP: 1
5      Arrival Time: Feb 12, 2013 16:59:13.856520000 CET
6      [Time shift for this packet: 0.000000000 seconds]
7      Epoch Time: 1360684753.856520000 seconds
8      [Time delta from previous captured frame: 0.000000000 seconds]
9      [Time delta from previous displayed frame: 0.000000000 seconds]
10     [Time since reference or first frame: 0.000000000 seconds]
11     Frame Number: 1
12     Frame Length: 98 bytes (784 bits)
13     Capture Length: 98 bytes (784 bits)
14     [Frame is marked: False]
15     [Frame is ignored: False]
16     [Protocols in frame: eth:ip:icmp:data]
17     Ethernet II, Src: Intel_ef:d4:96 (c4:85:08:ef:d4:96), Dst: Cisco_3a:c9:40 (00:64:40:3a:c9:40)
18     Destination: Cisco_3a:c9:40 (00:64:40:3a:c9:40)
19     Address: Cisco_3a:c9:40 (00:64:40:3a:c9:40)
20     ....0. .... = LG bit: Globally unique address (factory default)
21     ....0. .... = IG bit: Individual address (unicast)
22     Source: Intel_ef:d4:96 (c4:85:08:ef:d4:96)
23     Address: Intel_ef:d4:96 (c4:85:08:ef:d4:96)
24     ....0. .... = LG bit: Globally unique address (factory default)
25     ....0. .... = IG bit: Individual address (unicast)
26     Type: IP (0x0800)
27     Internet Protocol Version 4, Src: 120.12.17.214 (120.12.17.214), Dst: 12.22.58.30 (12.22.58.30)
28     Version: 4
29     Header length: 20 bytes
30     Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
31     0000 00.. = Differentiated Services Codepoint: Default (0x00)
32     ....00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable Transport) (0x00)
33     Total Length: 84
34     Identification: 0x0000 (0)
35     Flags: 0x02 (Don't Fragment)
36     0... .... = Reserved bit: Not set
37     1... .... = Don't fragment: Set
38     ..0. .... = More fragments: Not set
39     Fragment offset: 0
40     Time to live: 64
41     Protocol: ICMP (1)
42     Header checksum: 0x415c [correct]
43     [Good: True]
44     [Bad: False]
45     Source: 120.12.17.214 (120.12.17.214)
46     Destination: 12.22.58.30 (12.22.58.30)
47     [Source GeoIP: Unknown]
48     [Destination GeoIP: Unknown]
49     Internet Control Message Protocol
50     Type: 8 (Echo (ping) request)
51     Code: 0
52     Checksum: 0x1fb8 [correct]
53     Identifier (BE): 27502 (0x6b6e)
54     Identifier (LE): 28267 (0x6e6b)
55     Sequence number (BE): 10557 (0x293d)
56     Sequence number (LE): 15657 (0x3d29)
57     Timestamp from icmp data: Feb 12, 2013 16:59:13.000000000 CET
58     [Timestamp from icmp data (relative): 0.856520000 seconds]
59     Data (48 bytes)
60
61     0000  8c 11 0d 00 00 00 00 00 10 11 12 13 14 15 16 17  .....
62     0010  18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27  ..... !"#%&'
63     0020  28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37  ()*+,-./01234567
64         Data: 8c110d0000000000101112131415161718191a1b1c1d1e1f...
65         [Length: 48]

```

LISTADO 4.3: tshark permite elegir los campos a imprimir

```
$ tshark -f "tcp and dst port 80" -T fields -e eth.src -e ip.src -e eth.dst -e ip.dst
Capturing on wlan0
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      198.252.206.25
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      65.121.208.106
c4:85:08:ef:d4:96      120.12.17.214      08:20:12:3d:f4:32      120.12.27.170
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      198.252.206.25
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      173.194.34.196
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      173.194.34.196
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      173.194.34.196
c4:85:08:ef:d4:96      120.12.17.214      00:64:40:3a:c9:40      108.160.160.160
```

En el listado 4.2 hay 4 secciones bien delimitadas:

1. *Frame I* (línea 2) muestra información sobre el mensaje completo sin entrar en el formato del mensaje. Esta información se refiere al tamaño e instante de tiempo de la captura principalmente.
2. *Ethernet II* (línea 17) muestra información de la trama Ethernet detallando todos sus campos (vea §3.1.1).
3. *Internet Protocol* (línea 27) muestra los detalles del paquete IP (vea §3.1.2).
4. *Internet Control Message* (línea 49) contiene la información de la carga útil del paquete IP, que en este caso es un mensaje ICMP.

Por último, a partir de la línea 61, aparece un volcado de la carga útil del mensaje ICMP en tres columnas, que corresponden respectivamente al *offset*, valores en hexadecimal y ASCII de cada palabra de 16 bytes.

Normalmente esto es demasiada información y lo interesante es aislar exactamente los campos relacionados con el problema que se esté tratando en cada caso. Para lograrlo se debe indicar el formato de salida por campos (`-T fields`) y la lista con los campos concretos, que pueden depender de las condiciones de filtrando. El ejemplo del listado 4.3 filtra los mensajes TCP dirigidos al puerto 80 (presunto HTTP) y muestra únicamente las direcciones MAC e IP origen y destino.

Los nombres de los campos posibles son los mismos que para los «filtros de visualización» (que se tratan más adelante).

E 4.19 En el listado 4.3 las direcciones MAC e IP origen son siempre las mismas porque los mensajes salen del computador que está haciendo la captura. Sin embargo, la misma dirección MAC destino aparece junto a direcciones IP diferentes. ¿Cómo es posible?

Además, es posible generar una representación de la captura en otros formatos, como XML o PostScript.

4.1.5. Filtros de visualización

A partir de la captura realizada es posible realizar un segundo filtrado que determina qué mensajes se muestran al usuario. El formato de estos filtros es diferente al de los filtros de captura, y también mucho más potente.

Como ejemplo, el siguiente listado muestra las peticiones HTTP que soliciten

una URI que contenga la cadena 'favicon.ico', el icono que los navegadores asocian al guardar una página como favorito (*bookmark*).

```
$ tshark -R 'http.request.uri contains "favicon.ico"'
Capturing on wlan0
 9.666903 192.168.2.49 -> 93.184.220.111 HTTP 380 GET /i/favicon.ico?m=1317424629g HTTP
/1.1
82.079964 192.168.2.49 -> 217.148.71.165 HTTP 365 GET /favicon.ico HTTP/1.1
87.771618 192.168.2.49 -> 23.37.161.157 HTTP 633 GET /favicon.ico HTTP/1.1
11 packets dropped
3 packets captured
```

El programa proporciona literalmente decenas de miles de filtros. Puede consultarlos en la referencia en línea¹ o en la página de manual para wireshark-filter. Estos filtros están organizados jerárquicamente siendo el primer componente el nombre de un protocolo. En el ejemplo anterior, `http.request.uri` se refiere a la URI que aparece en la petición (GET) de los mensajes HTTP². La tabla 4.2 muestra algunos ejemplos representativos similares a <http://wiki.wireshark.org/DisplayFilters>.

Filtro	Significado	chuleta filtros de visualización
<code>icmp.type == 8</code>	Peticiones ICMP Echo	
<code>tcp.port == 25</code>	segmentos TCP hacia o desde el puerto 25	
<code>tcp.dstport == 25</code>	únicamente los dirigidos al puerto 25	
<code>arp or icmp</code>	ICMP or ARP de cualquier tipo	
<code>ip.addr == 192.168.2.0/24</code>	generado o dirigido por un computador de la red indicada	
<code>!(ip.dst == 127.0.0.1)</code>	no dirigido a la interfaz <i>loopback</i>	

CUADRO 4.2: tshark: ejemplos de filtros de visualización

Al igual que los filtros de captura es posible utilizar operadores lógicos y relacionales que permiten escribir expresiones muy elaboradas³.

tshark puede procesar una captura previa almacenada en un fichero (opción '-r'), y aplicando distintos filtros de visualización sin afectar al fichero. De ese modo se puede afinar un filtro o estudiar distintos aspectos de la misma captura.

4.1.6. Generación de estadísticas

tshark genera una gran variedad de estadísticas útiles⁴. En esta sección se incluyen solo algunas de las posibilidades.

¹<http://www.wireshark.org/docs/dfref/>

²<http://www.wireshark.org/docs/dfref/t/tcp.html>

³http://www.wireshark.org/docs/wsug_html_chunked/ChWorkBuildDisplayFilterSection.html

⁴Puede ver todas las estadísticas disponibles con `tshark -z help`

proto,colinfo,filter,field

Añade columnas a la salida habitual. Por ejemplo:

```
$ tshark -z proto,colinfo,tcp.srcport,tcp.srcport
5.508997 108.160.160.160 -> 192.168.2.49 HTTP 245 HTTP/1.1 200 OK (text/plain)
    tcp.srcport == 80
8.928316 173.194.78.125 -> 192.168.2.49 TCP 471 [TCP segment of a reassembled PDU]
    tcp.srcport == 5222
17.660298 173.194.78.125 -> 192.168.2.49 TCP 199 [TCP segment of a reassembled PDU]
    tcp.srcport == 5222
```

icmp,srt[,filter]

Calcula las estadísticas SRT (Service Response Time) del tráfico ICMP echo de forma similar al comando ping. Un ejemplo:

```
$ tshark -z icmp,srt
[...]
10024 packets captured

=====
ICMP Service Response Time (SRT) Statistics (all times in ms):
Filter: <none>

Requests  Replies  Lost      % Loss
1784      1783      1         0,1%

Minimum   Maximum   Mean      Median    SDeviation   Min Frame Max Frame
55,228    818,109   62,742    56,894    37,105       7831     5373
=====
```

io,phs[,filter]

Contabiliza mensajes y bytes de todos los mensajes (conformes al filtro) desglosando los resultados según su encapsulamiento.

```
$ tshark -z io,phs -q -c 100
Capturing on wlan0
100 packets captured

=====
Protocol Hierarchy Statistics
Filter:

eth                                     frames:100 bytes:18503
  ip                                   frames:98 bytes:18289
    icmp                             frames:32 bytes:3136
      udp                             frames:34 bytes:3760
        dns                           frames:32 bytes:3408
          db-lsp-disc                  frames:2 bytes:352
            tcp                        frames:32 bytes:11393
              ssl                      frames:14 bytes:10189
                tcp.segments           frames:2 bytes:1157
          llc                          frames:2 bytes:214
            data                       frames:2 bytes:214
=====
```

io,stat,interval,filter,filter

Contabiliza mensajes y bytes en intervalos del tiempo indicado. Permite indicar una cantidad arbitraria de filtros que serán representados en columnas independientes.

```
$ LANG=C tshark -nzio,stat,0.5,udp,"tcp.dstport==80",
[...]
794 packets captured

=====
| IO Statistics |
| Interval size: 0.5 secs |
| Col 1: udp |
| 2: tcp.dstport==80 |
| 3: Frames and bytes |
|-----|
| Interval | 1 | 2 | 3 |
| Interval | Frames | Bytes | Frames | Bytes | Frames | Bytes |
|-----|
| 0.0 <> 0.5 | 2 | 211 | 2 | 132 | 30 | 9012 |
| 0.5 <> 1.0 | 0 | 0 | 9 | 498 | 18 | 1044 |
| 1.0 <> 1.5 | 2 | 211 | 0 | 0 | 4 | 407 |
| 1.5 <> 2.0 | 0 | 0 | 7 | 378 | 14 | 798 |
| 2.0 <> 2.5 | 6 | 806 | 11 | 1597 | 24 | 3376 |
| 2.5 <> 3.0 | 17 | 1780 | 61 | 11138 | 133 | 42969 |
|-----|
```

io,stat,interval,func

Aplica funciones de agregación (COUNT, SUM, MIN, MAX, AVG y LOAD) que se pueden aplicar a campos cualesquiera para generar columnas en la tabla de resultados. El siguiente ejemplo cuenta el número de paquetes IP, y calcula la media y la suma de los tamaños de esos paquetes en intervalos de 2 segundos.

```
$ LANG=C tshark -nzio,stat,2,"COUNT(ip)ip","AVG(ip.len)ip.len","SUM(ip.len)ip.len"
[...]
803 packets captured

=====
| IO Statistics |
| Interval size: 2 secs |
| Col 1: COUNT(ip)ip |
| 2: AVG(ip.len)ip.len |
| 3: SUM(ip.len)ip.len |
|-----|
| Interval | 1 | 2 | 3 |
| Interval | COUNT | AVG | SUM |
|-----|
| 0 <> 2 | 7 | 61 | 432 |
| 2 <> 4 | 0 | 0 | 0 |
| 4 <> 6 | 107 | 115 | 12354 |
| 6 <> 8 | 486 | 267 | 130039 |
| 8 <> 10 | 126 | 497 | 62628 |
| 10 <> 12 | 20 | 45 | 904 |
| 12 <> 14 | 38 | 40 | 1544 |
|-----|
```

follow,proto,mode,filter[,range]

Muestra el contenido de un flujo de mensajes entre dos computadores, es decir, concatena la carga útil de los segmentos sucesivos que corresponden a la misma sesión o conexión. De este modo, si por ejemplo se está transmitiendo un fichero podría recuperarse a partir de la captura. Los protocolos soportados son TCP y UDP. Y puede hacer el volcado en tres modos diferentes: ascii, hex y raw.

```
$ tshark -z follow,tcp,hex,192.168.2.12:47147,129.42.58.216:80
```

conv,type[,filter]

Muestra las «conversaciones», es decir, computadores que intercambian mensajes entre sí de forma recurrente. El parámetro *type* puede ser uno de: eth, fc, fddi, ip, ipv6, ipx, tcp, tr, udp.

```
1 $ tshark -z conv,tcp
2 =====
3 TCP Conversations
4 Filter:<No Filter>
5
6 | <- | -> | Total | Start | Durat |
7 | F B | F B | F B | | |
8 192.168.2.49:38216 <-> 73.233.104.123:80 1 74 2 140 3 214 0,002912 0,2126
9 192.168.2.49:58949 <-> 92.184.220.111:80 1 66 2 128 3 194 0,002801 0,1134
10 192.168.2.49:58948 <-> 92.184.220.111:80 1 66 2 128 3 194 0,002754 0,1118
11 192.168.2.49:58947 <-> 92.184.220.111:80 1 66 2 128 3 194 0,002704 0,1027
12 192.168.2.49:58946 <-> 92.184.220.111:80 1 66 2 128 3 194 0,002660 0,1015
13 192.168.2.49:56569 <-> 92.100.127.144:80 1 74 2 140 3 214 0,002398 0,0727
14 192.168.2.49:38838 <-> 95.172.94.11:80 1 62 2 128 3 190 0,002319 0,0952
15 192.168.2.49:38837 <-> 95.172.94.11:80 1 62 2 128 3 190 0,002270 0,0935
16 =====
```

E 4.20 En el listado anterior se pueden apreciar 4 conversaciones distintas con las mismas IP origen y destino, pero distinto puerto origen (líneas 7–10). Más abajo (líneas 12–13) vuelve a ocurrir con un computador remoto diferente. ¿A qué corresponde este comportamiento?

4.2. Respuestas

E 4.19 FIXME

E 4.20 Dado que el puerto remoto es el 80 es probable que se trate de peticiones HTTP. Este comportamiento es típico de los navegadores web modernos. Al cargar una página utilizan conexiones (sockets) adicionales para descargar en paralelo contenido adicional enlazado en la página, como pueden ser iconos, hojas de estilo, etc.

Encapsulación de protocolos

Miguel Ángel Martínez

Hasta el momento se ha explicado cómo se conectan los distintos componentes que existen en una red, sin embargo no se ha abordado cómo y qué datos se introducen en los paquetes que se envían a través de la red. Este proceso es conocido como *encapsulación*, y en esta sección se mostrará una herramienta que permite inspeccionar los paquetes de la red observando cómo se lleva a cabo ese proceso.

A lo largo de esta sección el lector podrá apreciar la utilidad de este tipo de herramientas en multitud de campos. Por ejemplo resultan de especial interés en la depuración (de protocolos, aplicaciones, etc.) o bien aprendiendo de ellos y en la seguridad (detectando ataques, permitiendo la monitorización de tráfico, etc.).

Para una comprensión completa de esta sección es necesario que el lector esté familiarizado con el modelo OSI, ya que se muestra cómo se van introduciendo los datos pertenecientes a cada nivel en un paquete.

5.1. Objetivos

Tras finalizar esta sección el lector debe adquirir los siguientes conocimientos:

- Visualizar y comprender el concepto de encapsulación.
- Comprender la pila de protocolos.
- Manejar una herramienta de monitorización del tráfico red.

5.2. Entorno

Al igual que las secciones anteriores, el entorno de trabajo elegido es un sistema operativo GNU, concretamente Debian [deb], por lo que los aspectos de configuración mencionados en este documento se refieren a dicho sistema operativo. Conviene destacar que *wireshark*, la herramienta utilizada en esta sección, también está disponible para otros sistemas operativos.

Para llevar a cabo los ejercicios propuestos será imprescindible la utilización de al menos una interfaz de red que permita al computador estar conectado a Internet sin limitaciones importantes.

5.3. Situación inicial

Lo primero que el lector debe comprender es lo que pretendemos hacer, es decir, debe ser consciente de dónde está su computador, a que red está conectado

y qué tráfico puede monitorizar. En la Figura 5.1 se muestra una estructura típica de una red en la que hay dos grupos de hosts conectados mediante dos LAN Ethernet a dos switches. Éstos a su vez están conectados a un router que permite el acceso a Internet.

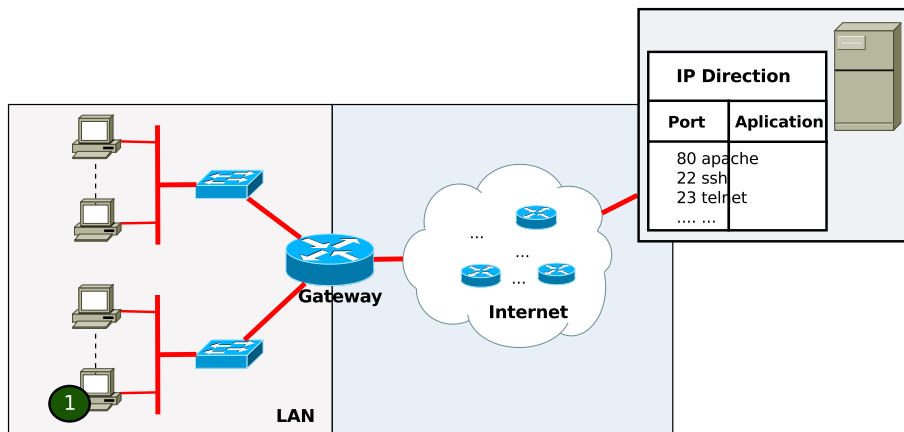


FIGURA 5.1: Situación inicial

E 5.21 Suponiendo que estamos en host 1 ¿Qué tráfico podríamos ser capaces de inspeccionar?

5.4. Wireshark

Evidentemente lo primero que debemos hacer, en caso de no tener la herramienta, es instalarla:

```
$ sudo aptitude install wireshark
```

Para poder sacar el máximo partido a esta herramienta es necesario que sea capaz de gestionar las interfaces de red de la computadora, por lo que es necesario ejecutarla en modo superusuario: `sudo wireshark`.

Al ejecutarlo aparece la ventana principal con un aspecto similar al de la Figura 5.2 (aunque puede variar dependiendo de la plataforma). Esta ventana la podemos dividir en cuatro partes empezando desde la parte superior, encontrando la barra de menú (que contiene todas las posibles opciones), una botonera con las opciones más comunes, tales como la opción de iniciar/parar la captura, configurar las interfaces o abrir/guardar archivos de capturas. Debajo de la botonera se encuentra una sección para el manejo de filtros, y finalmente, en el espacio en el que se mostrarán las capturas se muestran más atajos para las opciones más comunes.

5.4.1. Captura de paquetes

Llegados a este punto nos interesa empezar a capturar paquetes, sin embargo primero debemos seleccionar la interfaz que queremos utilizar para la captura. Para ello iremos a la opción `Capture → Options (C-k)`, obteniendo una ventana similar a la de la Figura 5.3. Desde este diálogo se pueden especificar tanto interfaces de captura, como el modo de almacenamiento de las mismas o incluso filtros (se verán más adelante).

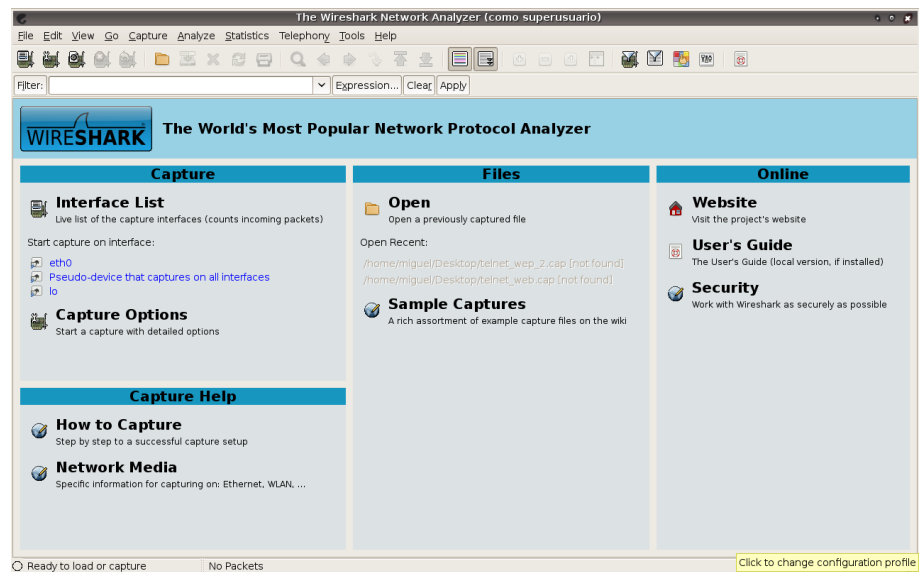


FIGURA 5.2: Ventana inicial

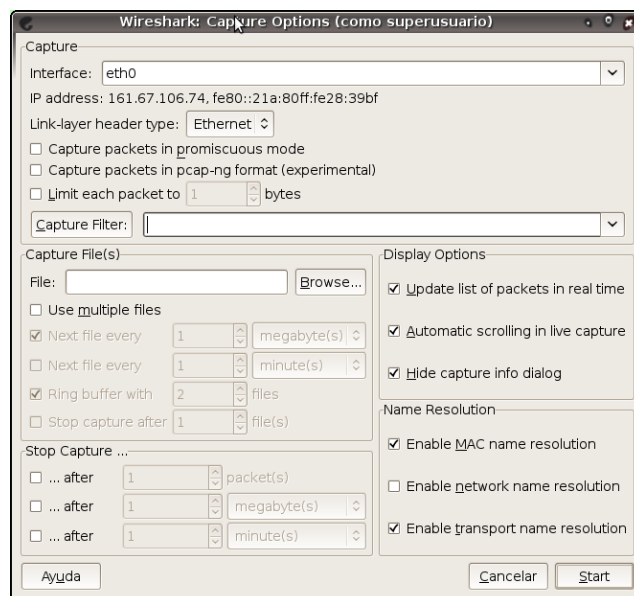


FIGURA 5.3: Opciones de captura

Una vez seleccionada la interfaz de red podemos empezar a recuperar los paquetes (simplemente pulsa el botón de start) y en tiempo real se irán mostrando en la ventana principal. Una vez que hayas recabado los paquetes que te interesen para la captura.

E 5.22 Cierra todos los programas que estén haciendo uso de la red (mensajería instantánea, navegador web, P2P, etc.). Abre una terminal y deja preparada una dirección para hacer ping, por ejemplo `web.mit.edu`. Empieza a capturar paquetes y acto seguido ejecuta el ping, tras un par de segundos para el ping y la captura ¿Qué protocolo aparece el primero (en el contexto de la ejecución de este programa)? ¿Por que aparece? ¿Qué protocolo utiliza ping?

5.4.2. Interpretando de resultados

Durante la captura la herramienta ha ido mostrando datos en tres partes del panel inferior, mostrando un aspecto parecido al de la Figura 5.4. Como se puede apreciar, la parte superior de este panel muestra los paquetes capturados, la parte intermedia muestra una disección del paquete seleccionado en la parte superior, mientras que la parte inferior muestra el contenido en hexadecimal del paquete. Nótese que la parte del paquete que se selecciona en el panel de disección aparece remarcada en el panel que muestra el contenido en hexadecimal.

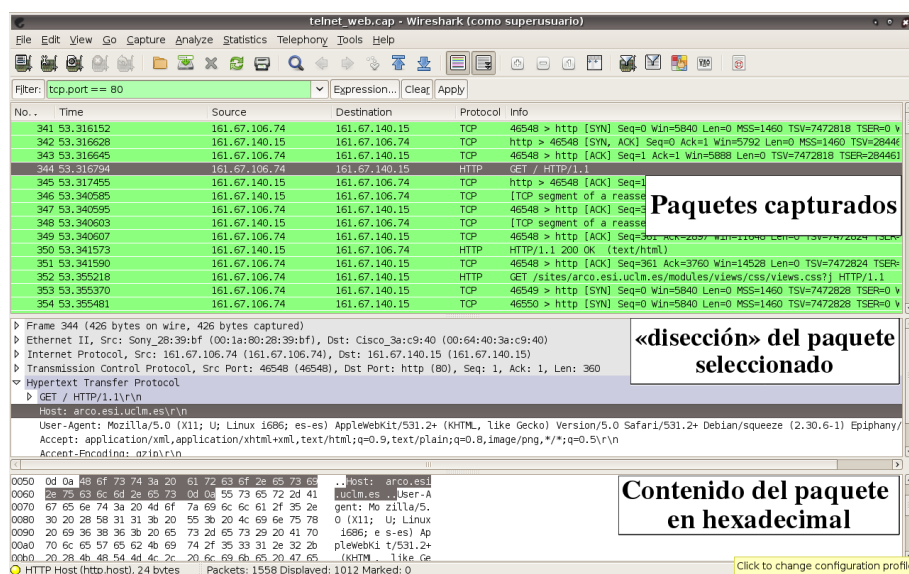


FIGURA 5.4: Ventana principal

En la parte en la que se muestran los paquetes podemos apreciar distintas columnas. La primera es el número de paquete (conteo que lleva de la captura), la segunda es el momento en el que se captó el paquete. A continuación se muestran columnas que representan información de los niveles de red y aplicación, tales como las direcciones IP origen y destino, el protocolo y algo de información adicional.

En la parte central podemos ver los datos que contiene el paquete seleccionado. Resulta especialmente interesante comprobar cómo se van encapsulando los protocolos, y para ello esta vista es magnífica.

Podemos ir desplegando las partes del paquete por protocolos, viendo cómo unos se encapsulan dentro de otros. Por ejemplo, seleccionamos el campo `frame` veremos, que en la pantalla inferior, se remarca todo el paquete en hexadecimal. Esto implica que al hablar de `frame` hablamos de la trama completa que viaja por la red. Se propone al lector la tarea de comprobar las posiciones que la encapsulación de cada protocolo va ocupando dentro de la trama.

Por último, la ventana inferior muestra el tamaño en bytes del elemento seleccionado en la ventana central.

E 5.23 Realiza una captura de un par de segundos. Busca un paquete con el protocolo ARP. ¿Cual es la MAC destino del paquete? ¿Por qué? ¿Cual es el tamaño de la dirección de respuesta?

E 5.24 Averigua cuál es la IP de tu máquina. Cierra todos los programas que estén haciendo uso de la red (mensajería instantánea, navegador web, P2P, etc). Abre una terminal y deja preparado dos `wget` a dos direcciones web distintas, por ejemplo:

```
$ wget web.mit.edu; wget arco.esi.uclm.es
```

Empieza a capturar, y acto seguido ejecuta esa línea y termina la captura. ¿Cuál es la dirección IP del servidor web del MIT? ¿Y la del servidor Arco? Examina la capa de enlace de un paquete que hayas recibido del MIT y de otro recibido de Arco ¿Qué conclusión obtienes?

5.4.3. Filtrado

Durante los procesos de captura anteriores hemos parado otras herramientas que utilizan la red para minimizar el tráfico que capturamos y que las capturas sean más limpias, sin embargo, una de las principales utilidades de *wireshark* es que es capaz de filtrar las capturas incluso en tiempo real.

Este documento no pretende ilustrar de forma exhaustiva el manejo de filtros, pero si mostrar al lector la utilidad de esta técnica. Si deseas crear filtros más sofisticados te recomendamos encarecidamente consultar: `man wireshark-filter`.

El lector se estará preguntando qué es un filtro, pues bien, simplemente es una expresión regular que deben satisfacer los paquetes que estamos recuperando. Por ejemplo, si únicamente queremos recuperar paquetes que utilicen el protocolo IP deberíamos utilizar el filtro *ip*.

Ahora lo que queremos es que *wireshark* aplique este filtro. Para esto tenemos justo debajo de la botonera un lugar reservado para los filtros. Este espacio permite escribir los filtros y los va comprobando (que no aplicando) en tiempo real. Es decir, si el filtro es una expresión regular válida el campo se verá en color verde, si no en color rojo. Una vez que la expresión regular sea correcta podemos aplicar el filtro, obteniendo sólo aquellos paquetes que satisfacen la expresión.

El filtro que acabamos de mostrar es trivial, sin embargo, se puede afinar todo lo que queramos. La expresión regular permite operadores lógicos como `and`, `or`, `not`, operadores de comparación `==`, `!=`, etc.

Wireshark también permite preguntar por los campos característicos de cada

protocolo. Por ejemplo, sabemos que un paquete IP contiene una dirección destino; con *wireshark* podríamos utilizar la expresión `ip.src==18.9.22.69` para filtrar los paquetes IP que procedan de esa dirección IP.

Hasta el momento hemos utilizado los filtros una vez habíamos capturado una serie de paquetes, sin embargo es posible capturar únicamente los paquetes que cumplan ese filtro. Para ello podemos definir el filtro en la pantalla de opciones de captura, aunque la notación varía ligeramente (ver los ejemplos que se proponen desde la herramienta).

E 5.25 Escriba y pruebe un filtro que permita recuperar únicamente el tráfico de una comunicación TELNET[PR83].

E 5.26 Escriba y pruebe un filtro que permita recuperar únicamente los paquetes que contienen HTML[DVGD96] que interpreta el navegador web. (Pista: slice en filtros)

5.4.4. Seguir un stream

Cuando realizamos una petición a un servidor web se establece una conexión TCP por la que nuestro navegador solicita al servidor una página HTML. El trasiego de datos entre el cliente y el servidor constituye un flujo. Éste concepto flujo es el que se conoce como stream.

Básicamente lo que vamos a hacer en esta sección es visualizar cómo interactúan un cliente y un servidor de una forma mucho más clara. Para ello es suficiente con que nos situemos sobre un paquete y seleccionemos *Analyze* → *Follow Stream*. Obtendremos una ventana como la que se muestra en la Figura 5.5. En ella podemos ver los mensajes del cliente y el servidor en colores diferentes, pudiendo comprobar cómo se comportan cada uno en base a dichos mensajes.

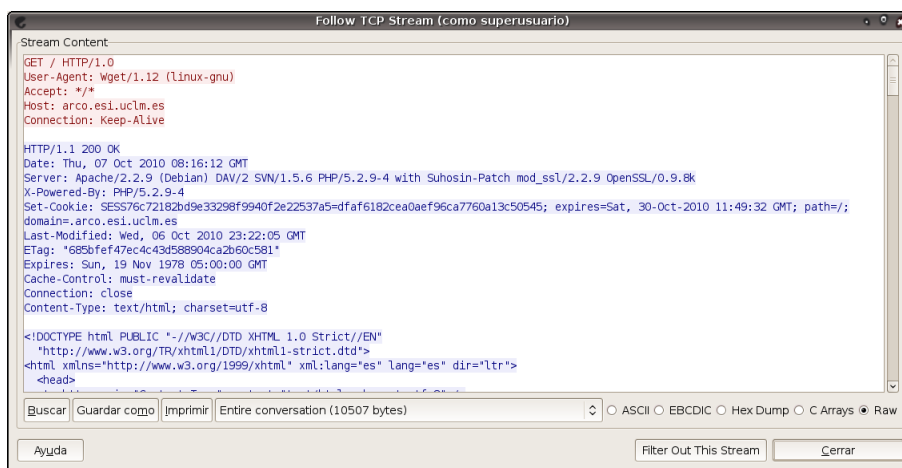


FIGURA 5.5: Wireshark: siguiendo un stream

E 5.27 Empieza a capturar tráfico y utiliza el navegador para conectarte por ejemplo a `www.uclm.es`. Analiza el flujo HTTP ¿Qué primitivas se utilizan para solicitar y recibir la página web?

E 5.28 Utilizando un archivo de capturas que contenga una conexión TELNET, o generando una tú mismo, realice un seguimiento del stream TELNET.

¿Qué conclusiones obtiene?

El lenguaje Python

David Villa

Python es un lenguaje interpretado, interactivo y orientado a objetos. Se le compara con C++. Java o Perl. Tiene variables, clases, objetos, funciones y todo lo que se espera de un lenguaje de programación moderno. Cualquier programador puede empezar a trabajar con Python con poco esfuerzo. Según muchos estudios, Python es uno de los lenguajes más fáciles de aprender y que permiten al novato ser productivo en tiempo record.

Python es perfecto como lenguaje «pegamento» y para prototipado rápido de aplicaciones (aunque se hagan luego en otros lenguajes). Pero también se pueden hacer programas serios como Zope, mailman, etc. Dispone de librerías para desarrollar aplicaciones multihilo, distribuidas, bases de datos, con interfaz gráfico, juegos, gráficos 3D y una larga lista.

Este pequeño tutorial presupone que el lector tiene nociones básicas de programación con algún lenguaje como C, Java o similar.

6.1. Empezamos

Nada como programar en un lenguaje para aprender a usarlo. El intérprete de Python se puede utilizar como una shell (lo que se llama modo interactivo), algo que viene muy bien para dar nuestros primeros pasos.

```
$ python3
Python 3.3.4rc1 (default, Jan 27 2014, 11:28:31)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

6.2. Variables y tipos

Las variables no se declaran explícitamente, se pueden usar desde el momento en que se inicializan. Y normalmente no hay que preocuparse por liberar la memoria que ocupan, tiene un «recolector de basura» (como Java).

```
$ python
>>> a = "hola"
```

En el modo interactivo se puede ver el valor de cualquier objeto escribiendo simplemente su nombre:

```
>>> a
```

```
'hola'
```

Python es un lenguaje de tipado fuerte pero dinámico. Eso significa que no se puede operar alegremente con tipos diferentes (como se hace en C).

```
>>> a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Esa operación ha provocado que se dispare la excepción `TypeError`. Las excepciones son el mecanismo más común de notificación de errores.

Pero se puede cambiar el tipo de una variable sobre la marcha.

```
>>> a = 3
```

6.3. Variables y tipos

6.3.1. Valor nulo

Una variable puede existir sin tener valor ni tipo, para ello se asigna el valor `None`:

6.3.2. Booleanos

Lo habitual:

```
>>> a = True
>>> a
True
>>> not a
False
>>> a and False
False
>>> 3 > 1
True
>>> b = 3 > 1
>>> b
True
```

6.3.3. Numéricos

Python dispone de tipos enteros de cualquier tamaño que soportan todas las operaciones propias de C, incluidas las de bits. También hay reales como en cualquier otro lenguaje. Ambos tipos se pueden operar entre si sin problema. Incluso tiene soporte para números complejos de forma nativa.

```
>>> a = 2
>>> b = 3.0
>>> a + b
5.0
>>> b - 4j
(3-4j)
```

En algunos lenguajes existe el tipo carácter (`char`) que puede manejarse como datos numéricos ya que permiten operaciones aritméticas. En Python,

sin embargo, se utilizan cadenas de caracteres de tamaño 1, y no permiten operaciones aritméticas.

6.3.4. Secuencias

La secuencia más simple y habitual es la cadena de caracteres (tipo `str`). Las cadenas admiten operaciones como la suma y la multiplicación:

```
>>> cad = 'hola '  
>>> cad + 'mundo '  
'hola mundo '  
>>> cad * 3  
'hola hola hola '
```

Las tuplas son una agrupación de valores similar al concepto matemático homónimo. Se pueden empaquetar y desempaquetar varias variables (incluso de tipo diferente).

```
>>> x = cad, 'f', 3.0  
>>> x  
('hola ', 'f', 3.0)  
>>> v1, v2, v3 = x  
>>> v2  
'f'
```

Las tuplas, al igual que las cadenas, son inmutables, es decir, una vez construida no se puede modificar.

```
>>> cad = 'hola '  
>>> cad[0] = 'm'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment
```

Si se quiere un cambio hay que crear una nueva instancia a partir de la anterior:

```
>>> cad.upper()  
'HOLA '  
>>> cad  
'hola '
```

Las Listas son similares a los vectores o arrays de otros lenguajes, aunque en Python se pueden mezclar tipos. La lista si es un tipo mutable.

```
>>> x = [1, 'f', 3.0]  
>>> x[0]  
1  
>>> x[0] = None  
>>> x  
[None, 'f', 3.0]
```

Las listas también se pueden «sumar» y «multiplicar»:

```
>>> x = [1, 'f', 3.0]  
>>> x + ['adios']  
[1, 'f', 3.0, 'adios']  
>>> x * 2  
[1, 'f', 3.0, 1, 'f', 3.0]
```

Cualquier secuencia (listas, tuplas y cadenas) se puede indexar del modo habitual, pero también desde el final con número negativos o mediante «rodajas» (*slicing*):

```
>>> cad = 'holamundo'
>>> cad[-1]
'o'
>>> cad[1:6]
'olamu'
>>> cad[:3]
'hol'
>>> cad[3:]
'amundo'
>>> cad[-6:-2]
'amun'
```

Los diccionarios son tablas asociativas (*hash maps*). Tanto las claves como los valores pueden ser de cualquier tipo, y de tipos diferentes en el mismo diccionario:

```
>>> notas = {'antonio':6, 'maria':9}
>>> notas['antonio']
6
```

Python dispone de otros tipos de datos nativos como conjuntos (*set*), pilas, colas, listas multidimensionales, etc.

6.3.5. Orientado a objetos

Casi todo en Python está orientado a objetos incluyendo las variables de tipos básicos, y hasta los literales!

```
>>> cad = 'holamundo'
>>> cad.upper()
'HOLAMUNDO'
>>> 'ADIOS'.lower()
'adios'
```

6.4. Módulos

Los módulos son análogos a las *librerías* o *paquetes* de otros lenguajes. Para usarlos se utiliza la sentencia `import`:

```
>>> import sys
>>> sys.exit(1)
```

6.5. Estructuras de control

Están disponibles las más habituales: `for`, `while`, `if`, `if else`, `break`, `continue`, `return`, etc. Todas funcionan de la forma habitual excepto `for`. El `for` de Python no es un `while` disfrazado, como ocurre la mayoría de lenguajes. Este `for` hace que, en cada iteración, la variable de control tome los valores que contiene una secuencia, es decir, que itere sobre ella; algo parecido al `for_each` de C++, pero mucho más compacto:

```
>>> metales = ['oro', 'plata', 'bronce']
>>> for m in metales:
...     print(m)
...
oro
plata
bronce
```

6.6. Indentación estricta

En muchos lenguajes se aconseja «tabular» (indentar) de determinada manera para facilitar su lectura. En Python este estilo es obligatorio porque el esquema de indentación define realmente los bloques. Es una importante peculiaridad de este lenguaje. Se aconseja tabulación blanda de 4 espacios. Mira el siguiente fragmento de un programa Python:

```
1 total = 0
2 aprobados = 0
3 notas = {'antonio':6, 'maria':9}
4 for i in notas.values():
5     total += i
6     if i >= 5:
7         aprobados += 1
8
9 print('La media es:', total / len(notas))
```

El cuerpo del bloque `for` queda definido por el código que está indentado un nivel debajo de él. Lo mismo ocurre con el `if`. La sentencia que define el bloque siempre lleva un `:` al final. No se necesitan llaves ni ninguna otra cosa para delimitar los bloques.

6.7. Funciones

Nada mejor para explicar su sintaxis que un ejemplo:

```
1 def factorial(n):
2     if n == 0:
3         return 1
4
5     return n * factorial(n-1)
6
7 print(factorial(10))
```

6.8. Python is different

Aunque Python se puede utilizar como un lenguaje convencional (tipo Java), siempre hay una manera más «pythónica» de hacer las cosas. Por ejemplo, el soporte de programación funcional que incluye permite hacer la función factorial de un modo diferente:

```
1 from functools import reduce
2 factorial = lambda x: reduce(int.__mul__, range(1, x+1), 1)
3 print(factorial(10))
```

Muy aconsejable leer [Python Is Not Java](#)

6.9. Hacer un fichero ejecutable

Lo normal es escribir un programa en un fichero de texto -con extensión .py utilizando algún buen editor (como emacs). Después ese fichero se puede ejecutar como cualquier otro programa. Para eso, la primera línea del fichero debe tener algo como:

```
1 | #!/usr/bin/python
```

Y no olvides darle permisos de ejecución al fichero:

```
$ chmod +x fichero.py
```

Codificación y representación de datos

David Villa

El único modo posible de procesar y almacenar datos en un computador actual es el código binario. Pero, salvo unas pocas excepciones, rara vez resulta suficientemente expresivo para representar información útil para las personas. Por ese motivo se utilizan distintas formas de interpretar el código binario en función del tipo de dato que se desea: entero, cadena, flotante, etc. Lo importante es recordar que sea cual sea su representación en un lenguaje de programación de alto nivel determinado, en la memoria o registros de la computadora o **en la red**, todo dato es a fin de cuentas una secuencia de bits (o de bytes).

7.1. Representación, sólo eso

Una de las excepciones a las que alude el párrafo anterior es la *programación de sistemas*, es decir, aquellas programas que consumen directamente servicios del SO. El binario resulta útil para manejar campos de bits, banderas binarias o máscaras, muy comunes cuando se manipulan registros de control, operaciones de E/S, etc.

Es importante manejar adecuadamente datos con estas representaciones. Como las personas, la mayoría de los lenguajes de programación utilizan la base decimal para expresar todo tipo de cantidades. Pero también ofrecen mecanismos para realizar cambios de base.

Python permite expresar literales numéricos en varios formatos. En todos los ejemplos se representa el número 42; y en todos los casos, si se asigna a una variable, se está creando un entero (tipo `int`) con el mismo valor. Puede comprobarlo fácilmente en el listado 7.1.

binario: `0b101010`
decimal: `42`
octal: `0o52`
hexadecimal: `0x2A`

Asimismo ofrece funciones para convertir entre bases: `bin()`, `oct()` y `hex()`; pero una consideración importante a tener en cuenta es que estas funciones devuelven cadenas (`str`) y se utilizan precisamente para ofrecer *representaciones* diferentes del mismo dato. Observe las comillas simples en los valores de retorno en el listado 7.2 que indican claramente que se trata de cadenas.

Opcionalmente, el constructor de la clase `int` acepta un número expresado como cadena de caracteres pudiendo además indicar la base (incluso con bases tan exóticas como 23). Véalo en el listado 7.3.

LISTADO 7.1: Literales numéricos en Python

```

1  >>> 0b101010
2  42
3  >>> 0o52
4  42
5  >>> 0x2A
6  42

```

LISTADO 7.2: Conversión a representación binaria, octal y hexadecimal

```

1  >>> bin(42)
2  '0b101010'
3  >>> oct(42)
4  '0o52'
5  >>> hex(42)
6  '0x2A'

```

Todo programador debe tener claro que 42, 052, 0x2A o 101010 no son más que representaciones diferentes del mismo dato, y que el computador lo manejará **siempre** en su forma binaria.

7.2. Los enteros de Python

El tipo `byte` es el más simple de cualquier lenguaje de programación y corresponde con una secuencia de 8 bits. Suele ser un entero sin signo, es decir, puede representar números enteros en el rango [0, 255]. El lenguaje Python, por su naturaleza dinámica, solo tiene un tipo de datos para enteros: `int`¹. En Python, un entero puede ser arbitrariamente largo puesto que el *runtime* se encarga de gestionar la memoria necesaria:

```

1  >>> googol = 10 ** 100
2  >>> type(googol)
3  <class 'int'>

```

Sin embargo, hay ocasiones, como cuando se serializan datos en un fichero o una conexión de red, en los que es necesario manejar explícitamente el tamaño de los datos. Lo veremos en las siguientes secciones.



En el lenguaje C no existe el tipo `byte`. En su lugar se suele utilizar `unsigned char`, dado que el tipo `char` de C es en realidad un entero de 8 bits que admite literales de carácter. Sin embargo, el tipo `byte` de Java es un entero de 8 bits *con* signo.

7.3. Caracteres

La codificación de caracteres más simple (y una de las más antiguas) consiste en asignar un número a cada carácter del alfabeto. ASCII fue creado por ANSI en 1963 como una evolución de la codificación utilizada anteriormente en telegrafía. Es un código de 7 bits (128 símbolos) que incluye los caracteres

¹Estamos hablando de Python 3.0.

LISTADO 7.3: Especificando la base en el constructor de int

```

1 >>> int('42')
2 42
3 >>> int('52', 8)
4 42
5 >>> int('1J', 23)
6 42

```

alfa-numéricos de la lengua inglesa (mayúsculas y minúsculas) y la mayoría de los signos de puntuación y tipográficos habituales. Además incluye caracteres de control para indicar salto de línea, de página, tabulador, etc. Más tarde IBM creó el código EBCDIC, similar a ASCII aunque de 8 bits (256 símbolos).



El carácter «retorno de carro» (CR, código 10 o 0x0A), indica que debe colocarse el cursor en la primera columna (en el borde izquierdo), mientras que el carácter de «avance de línea» (LF, código 13 o 0x0D) indica situar el cursor en la siguiente línea. Claramente alude a las máquinas de escribir, teletipos e impresoras en los que la máquina debe situar su cabezal para comenzar a escribir la siguiente línea. Las computadoras, por analogía, utilizaban la secuencia CR-LF para indicar la misma operación en un terminal. De hecho sigue siendo de este modo en los sistemas operativos de Microsoft. Por contra, los creadores de los sistemas UNIX entendieron que el salto de línea sin retorno de carro no tenía sentido en una consola y por tanto se utiliza únicamente el carácter CR para conseguir el mismo efecto. En muchos lenguajes de programación se representa con el carácter de control '\n' y se denomina «nueva línea» o EOL.

Prácticamente todos los lenguajes de programación incluyen funciones elementales para manejar la conversión entre bytes (números de 8 bits) y sus caracteres equivalentes. El siguiente fragmento de código Python lo demuestra mediante las funciones `ord()` y `chr()`.

```

1 >>> ord('a')
2 97
3 >>> chr(97)
4 'a'
5 >>> ord('0')
6 48
7 >>> ord('\0')
8 0
9 >>> chr(0)
10 '\x00'
11 >>> ord('\n')
12 10
13 >>> ord(' ')
14 32

```

Fíjese en la línea 5 que el código equivalente al **carácter** '0' es 48, mientras que (línea 7) el carácter equivalente al código 0 es el carácter '\x00'. Es especialmente importante tener claro que los caracteres numéricos **no son** equivalentes a los valores que representan. También resulta digno de mención

que la secuencia ‘\n’ es *un solo carácter* ya que la barra es lo que se conoce como un «carácter de escape», es decir, cambia el significado del siguiente carácter, en este caso significa «nueva línea» como hemos visto.

De modo similar, la secuencia ‘\x’ indica que los siguientes dígitos deben entenderse como un código hexadecimal. La función `chr()` devuelve una secuencia de este tipo cuando no existe un carácter «imprimible» asociado al código indicado. El siguiente listado muestra un ejemplo de la equivalencia entre una cadena y su representación numérica.

```
1 >>> for i in '\xd2a3\n':
2 ...     print ord(i)
3 ...
4 210
5 97
6 51
7 10
```

La cadena de la línea 1 está compuesta por los caracteres ‘\xd2’, ‘a’, ‘3’ y ‘\n’. La cadena de caracteres de Python (la clase `str`) es un tipo inmutable, es decir, no se puede modificar su contenido. Para añadir o cambiar alguno de los elementos de la cadena, es necesario crear una nueva a partir de la primera.

Sin embargo, existe el tipo `bytearray`, que permite almacenar una secuencia de bytes, modificar su contenido (acepta tanto caracteres como números) y se puede obtener fácilmente la lista de caracteres o secuencia de números equivalente:

```
1 >>> buf = bytearray('abcd', 'ascii')
2 >>> buf[0] = 20
3 >>> buf
4 bytearray(b'\x14bcd')
5 >>> buf.decode()
6 '\x14bcd'
7 >>> bytes(buf)
8 b'\x14bcd'
9 >>> list(buf)
10 [20, 98, 99, 100]
```

7.4. Tipos multibyte y ordenamiento

Como sabemos, un byte solo puede representar 256 valores. Obviamente casi cualquier programa o algoritmo, por simple que sea, necesita manejar enteros mayores, reales en coma flotante y otros tipos de datos que no pueden almacenarse en un solo byte. El más sencillo de estos tipos es el `short` o entero de 16 bits². Aquí aparece una cuestión interesante: el ordenamiento de bytes (*endianness* o *byte-order*), o lo que es lo mismo ¿en qué orden se deberían colocar en memoria los dos bytes que lo forman?

Dependiendo de la respuesta a esta pregunta se distingue entre *little endian* y *big endian*. *Little endian* significa que el byte *menos* significativo se coloca primero en memoria (tiene una dirección menor) mientras que en *big endian* es el byte de *mayor* peso el que se coloca primero en memoria. La figura 7.1 los muestra para un dato de 4 bytes.

Para que el computador realice las operaciones (aritméticas, lógicas, etc.)

²aunque `short` no existe como tipo nativo en Python

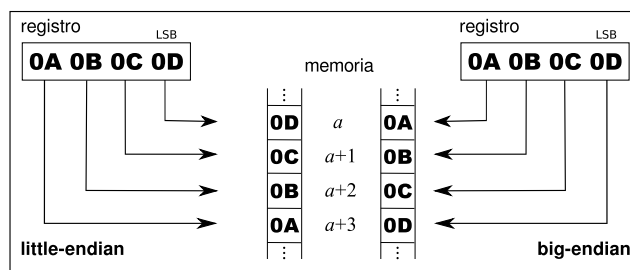


FIGURA 7.1: Ordenación de bytes

que correspondan sobre el dato, es esencial que los programas manipulen la memoria de acuerdo al ordenamiento de la arquitectura correspondiente.

Algo parecido ocurre con la red. Cuando se coloca un dato multi-byte *en el cable*³ también debe respetarse un ordenamiento concreto. En particular, los protocolos de la pila TCP/IP imponen que siempre ha de utilizarse ordenamiento *big-endian* [RP94].

Eso significa que las arquitecturas *little-endian* (típicamente Intel y AMD) deben convertir sus datos multi-byte antes de enviarlos a la red, es decir, al escribirlos sobre un socket. Para evitar que los programas tengan que comprobar por sí mismos qué tipo de ordenamiento utiliza el computador en el que se están ejecutando, las librerías de sockets proporcionan funciones que realizan la conversión. Nótese que en un computador *big-endian* estas funciones no harán nada⁴, pero deben usarse a pesar de ello porque de ese modo los programas serán portables, es decir, se podrán ejecutar en máquinas de diferente arquitectura sin modificaciones.

Estas funciones, tomadas de las llamadas al sistema POSIX homónimas, son:

- `socket.ntohs()`. Convierte un entero de 16 bits (*short*) del ordenamiento de la red al del host.
- `socket.ntohl()`. Convierte un entero de 32 bits (*long*) del ordenamiento de la red al del host.
- `socket.htons()`. Convierte un entero de 16 bits (*short*) del ordenamiento del host al de la red.
- `socket.htonl()`. Convierte un entero de 32 bits (*long*) del ordenamiento de host al de la red.

E 7.01 Escriba un programa Python que determine el tipo de ordenamiento que utiliza el computador que lo ejecuta.

E 7.02 Escriba la versión C del ejercicio anterior.

³del inglés *on the wire*

⁴retornan el mismo valor que les pasa como parámetro

El siguiente listado muestra unos ejemplos de uso de estas funciones en un computador *little-endian*.

LISTADO 7.4: Funciones de conversión de ordenamiento del módulo `socket`.

```

1 >>> socket.htons(32)
2 8192
3 >>> socket.htonl(32)
4 536870912
5 >>> socket.htons(0)
6 0

```

Veamos de nuevo, en hexadecimal, la primera transformación para observar fácilmente los 2 bytes que lo forman:

```

1 >>> hex(32)
2 '0x20'
3 >>> hex(socket.htons(32))
4 '0x2000L'

```

Se puede ver claramente que al convertir el valor `0x20` desde *big-endian* se coloca en el primer byte del entero de 16 bits. Si el computador receptor fuese *little-endian* no habría cambio alguno.

7.5. Cadenas de caracteres y secuencias de bytes

En Python-3 las cadenas de caracteres (el tipo `str`) utilizan Unicode. Sin embargo, este tipo de datos no se puede leer o escribir de un fichero (salvo que sea de texto), ni enviar o recibir de un socket. Todas esas operaciones requieren secuencias de bytes (el tipo `bytes`). Convertir una cadena a una secuencia de bytes requiere aplicar una codificación (un *encoding*). El siguiente listado ilustra la diferencia entre ambos tipos de datos:

LISTADO 7.5: Codificación ASCII

```

1 >>> string = "hello world"
2 >>> type(string)
3 <class 'str'>
4 >>> sequence = bytes(string, 'ascii')
5 >>> type(sequence)
6 <class 'bytes'>
7 >>> string
8 'hello world'
9 >>> sequence
10 b'hello world'

```



Python puede utilizar muchos sistemas de codificación *encodings* diferentes. El más habitual en los sistemas POSIX es UTF-8.

En la cadena de caracteres, cada símbolo representa un carácter, mientras que en la secuencia de bytes cada símbolo representa un byte. Y ¿cuál es la diferencia? Parece que las líneas 8 y 10 son iguales salvo por la `'b'` que precede a la secuencia de bytes. Esto se debe a que se ha utilizado una codificación `'ascii'`. ASCII codifica cada carácter con un único byte, por eso coinciden. Veamos otro ejemplo más ilustrativo:

LISTADO 7.6: Codificación UTF-8

```

1 >>> string = "ñandú"
2 >>> sequence = bytes(string, 'ascii')
3 UnicodeEncodeError: 'ascii' codec can't encode character [...]
4 >>> sequence = bytes(string, 'utf-8')
5 >>> string
6 'ñandú'
7 >>> sequence
8 b'\xc3\xba\xc3\xba'
9 >>> len(sequence)
10 7

```

Esta vez, al intentar codificar la cadena “ñandú” con el *encoding* ASCII se produce un error (línea 2), porque ese sistema de codificación no puede representar la letra ‘ñ’ ni la ‘ú’. El *encoding* UTF-8 sí es capaz de codificar esos caracteres, pero requiere 2 bytes por cada uno. Por eso, la secuencia equivalente requiere 7 bytes (línea 10) a pesar de que la cadena solo tiene 5 caracteres. Nótese que la conversión puede lograrse utilizando los constructores de ambos tipos (`bytes()` y `str()`) o los métodos `encode()` y `decode()` respectivamente:

```

1 >>> bytes('ñandú', 'utf-8')
2 b'\xc3\xba\xc3\xba'
3 >>> 'ñandú'.encode('utf-8')
4 b'\xc3\xba\xc3\xba'
5 >>> str(b'\xc3\xba\xc3\xba', 'utf-8')
6 'ñandú'
7 >>> b'\xc3\xba\xc3\xba'.decode('utf-8')
8 'ñandú'

```

7.6. Empaquetado

El módulo `struct` de la librería estándar de Python puede hacer transformaciones en la representación de los datos de un modo mucho más general. La función `struct.pack()` (*empaquetar*) convierte datos nativos de Python a una secuencia de bytes (tipo `bytes`) según la especificación de tamaño y ordenamiento que se le indique. Por ejemplo, el siguiente listado *empaqueta* el entero 5 como un entero de 32 bits con ordenamiento *big-endian* y después como *little-endian*:

```

1 >>> struct.pack('>i', 5)
2 b'\x00\x00\x00\x05'
3 >>> struct.pack('<i', 5)
4 b'\x05\x00\x00\x00'

```

El primer parámetro de `pack()` es la especificación de la conversión. Hay dos conjuntos de símbolos: uno para especificar ordenamiento y otro para especificar formato. La tabla 7.1 muestra los símbolos para ordenamiento.

El siguiente listado muestra el resultado de aplicar ambos ordenamientos al mismo dato en un computador *little-endian*, pero con un entero de 16 bits.

LISTADO 7.7: `struct`: alternativas de ordenamiento

```

1 >>> struct.pack('>h', 5)
2 b'\x00\x05'
3 >>> struct.pack('<h', 5)
4 b'\x05\x00'

```

@	ordenamiento nativo del computador (realiza alineamiento)
=	ordenamiento nativo
<	<i>little endian</i>
>	<i>big endian</i>
!	ordenamiento de la red (<i>big-endian</i>)

CUADRO 7.1: `struct`: especificación de ordenamiento

x	relleno (alineado al siguiente dato)
c	carácter (<code>char</code>)
b	byte con signo
B	byte sin signo
?	booleano/ <code>char</code>
h	entero de 16 bits con signo
H	entero de 16 bits sin signo
i	entero de 32 bits con signo
I	entero de 32 bits sin signo
q	entero de 64 bits con signo (nativo)
Q	entero de 64 bits sin signo (nativo)
f	<code>float</code>
d	<code>double</code>
s	cadena de caracteres (un número previo indica tamaño)
P	entero que puede almacenar una dirección de memoria
q ó Q	entero equivalente al <code>long</code> <code>long</code> de C en la misma arquitectura.

CUADRO 7.2: `struct`: especificación de formato

La tabla 7.2 corresponde a la especificación de formato.

El listado 7.8 muestra el resultado de aplicar los diferentes formatos al mismo dato en un computador *little-endian*.

Pero lo verdaderamente interesante de `struct` es que la cadena de formato puede especificar un número arbitrario de campos, que corresponden a parámetros sucesivos de la función `pack()`. Veamos un ejemplo empaquetando la cabecera de un mensaje ARP sobre una trama Ethernet (vea §3.1.1).

El valor para los campos de la trama será:

LISTADO 7.8: `struct`: empaquetado en diferentes tamaños

```

1  >>> struct.pack('b', 5)
2  b'\x05'
3  >>> struct.pack('?', 5)
4  b'\x01'
5  >>> struct.pack('h', 5)
6  b'\x05\x00'
7  >>> struct.pack('i', 5)
8  b'\x05\x00\x00\x00'
9  >>> struct.pack('l', 5)
10 b'\x05\x00\x00\x00\x00\x00\x00\x00'
11 >>> struct.pack('f', 5)
12 b'\x00\x00\xa0@'
13 >>> struct.pack('d', 5)
14 b'\x00\x00\x00\x00\x00\x00\x14@'

```

MAC destino

FF:FF:FF:FF:FF:FF, es decir, se trata de una trama broadcast.

MAC origen

C4:85:08:ED:D3:07.

tipo 0x0806, puesto que la trama a construir corresponde al protocolo ARP.

En el listado 7.9 se puede ver cómo construir dicha cabecera, la secuencia de bytes se obtiene y su equivalente numérico. La cadena de formato ('!6s6sh') indica que debe codificarse con ordenamiento de red ('!') y está compuesto de dos cadenas de 6 bytes ('6s') y un entero de 16 bits sin signo ('h'). Lo interesante es que la secuencia resultante siempre tendrá una longitud de 14 bytes independientemente del valor de sus tres argumentos.

LISTADO 7.9: struct: empaquetando una cabecera Ethernet

```

1 >>> header = struct.pack('!6s6sh', b'\xFF' * 6, b'\xC4\x85\x08\xED
  \xD3\x07', 0x0806)
2 >>> header
3 b'\xff\xff\xff\xff\xff\xff\xc4\x85\x08\xed\x07\x08\x06'
4 >>> list(header)
5 [255, 255, 255, 255, 255, 255, 196, 133, 8, 237, 211, 7, 8, 6]
```

E 7.03 Tomando como entrada los datos (3, 4, 5), escribe la cadena de formato para obtener la secuencia b'\x03\x00\x04\x00\x05' con ordenamiento *big-endian*.

E 7.04 Tomando como entrada los datos (3, 4, 5), escribe la cadena de formato para obtener la secuencia b'\x03\x00\x00\x00\x04\x05' con ordenamiento *big-endian*.

7.7. Desempaquetado

La contrapartida de `pack()` es `struct.unpack()`. Esta función toma una cadena de formato con las mismas reglas que `pack()` y una secuencia de bytes, que puede haber sido obtenida con `file.read()`, `socket.recv()` o cualquier otra función orientada a lectura de flujos (*streams*). La función `unpack()` retorna una tupla con los valores que corresponden a cada uno de los campos especificados en la cadena de formato.

El listado 7.10 realiza la función inversa al anterior. Es decir, a partir de la secuencia de bytes devuelve una tupla con las dos direcciones MAC y el tipo de la trama. La línea 5 simplemente corrobora que el tercer valor de la tupla (2054) coincide efectivamente con el valor hexadecimal 0x0806.

LISTADO 7.10: struct: desempaquetando una cabecera Ethernet

```

1 >>> header
2 b'\xff\xff\xff\xff\xff\xff\xc4\x85\x08\xed\x07\x08\x06'
3 >>> struct.unpack('!6s6sh', header)
4 (b'\xff\xff\xff\xff\xff\xff', b'\xc4\x85\x08\xed\x07', 2054)
5 >>> hex(2054)
6 '0x806'
```

E 7.05 Tomando como entrada la secuencia b'\x0aax\0\x45\x027', ¿cuál es la cadena de formato para obtener la secuencia (b'\nax\x00', 17666, 55)?

Sockets BSD

Gordon Mc Millan
[Traducción: David Villa]

Los sockets se usan casi en cualquier parte, pero son una de las tecnologías peor comprendidas. Este documento es una panorámica de los sockets. No se trata de un tutorial - debe poner de su parte para hacer que todo funcione. No cubre las cuestiones puntuales (y hay muchas), pero espero que le dé un conocimiento suficiente como para empezar a usarlos decentemente.

8.1. Sockets

Sólo se van a tratar los sockets INET (es decir, IPv4), pero éstos representan el 99 % de los sockets que se usan. Y sólo se hablará de los STREAM sockets - a menos que realmente sepa lo que estás haciendo (en cuyo caso este documento no le será útil), conseguirá un comportamiento mejor y más rendimiento de un STREAM socket que de cualquier otro. Intentaré desvelar el misterio de qué es un socket, así como las cuestiones relativas a cómo trabajar con sockets bloqueantes y no bloqueantes. Pero empezaré hablando sobre sockets bloqueantes. Necesita saber cómo trabajan los primeros antes de pasar a los sockets no bloqueantes.

Parte del problema para entender qué es «socket» es que esa palabra se puede utilizar para distintas cosas con diferencias sutiles, dependiendo del contexto. Lo primero de todo, hay que hacer una distinción entre socket cliente - un extremo de la conversación, y un socket «servidor», que es más como un operador de una centralita. La aplicación cliente (tu navegador, por ejemplo) usa exclusivamente sockets «cliente»; el servidor web con el que habla usa tanto sockets «servidor» como sockets «cliente».

8.1.1. Historia

De las diferentes formas de IPC (Inter Process Communication), los sockets son con mucho la más popular. En una plataforma dada, probablemente hay otras formas de IPC más rápidas, pero para comunicaciones inter-plataforma, los sockets son casi la única elección.

Se inventaron en Berkeley como parte de la variante BSD de UNIX. Se extendieron muy rápidamente junto con Internet. Y con razón - la combinación de los sockets con INET hace que la comunicación entre máquinas cualesquiera sea increíblemente sencilla (al menos comparada con otros esquemas).

8.2. Creación de un socket

Grosso modo, cuando pulsa un enlace para visitar una página, su navegador web hace algo parecido a lo siguiente:

```
1 # crea un socket INET de tipo STREAM
2 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3
4 # ahora se conecta al servidor web en el puerto 80 (HTTP)
5 s.connect(("www.python.org", 80))
```

Cuando la conexión se completa, el socket se usa para enviar una petición para el texto de la página. El mismo socket puede leer la respuesta, y después se destruye. Sí, así es, se destruye. Los sockets cliente normalmente sólo se usan para un sólo intercambio (o una pequeña secuencia de ellos).

Lo que ocurre en el servidor es un poco más complejo. Primero el servidor web crea un «socket servidor».

```
1 # create a STREAM INET socket
2 mastersocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3
4 # bind socket in the HTTP port
5 mastersocket.bind((socket.gethostname(), 80))
6
7 # and define the backlog
8 mastersocket.listen(5)
```

Un par de cosas a tener en cuenta: cuando se usa `socket.gethostname()` el socket debería ser visible desde el exterior. Si hubiera usado `s.bind(", 80)` o `s.bind('localhost', 80)` o `s.bind('127.0.0.1', 80)` también tendría un socket servidor, pero solo sería accesible desde la misma máquina.

Una segunda cuestión: los puertos con números bajos normalmente están reservados para servicios «bien conocidos» (HTTP; SNMP, etc.). Si estás experimentando, usa un número alto (por encima del 1024).

Por último, el argumento de `listen` le dice al socket que quiere encolar un máximo de 5 peticiones de conexión (lo normal) antes de rechazar conexión externas. Si el resto del código está bien escrito, debería ser suficiente.

Bien, ahora tiene un socket servidor, escuchando en el puerto 80. Ahora entra en el bucle principal del servidor web.

```
1 while 1:
2     # acepta conexiones externas
3     (childsocket, address) = mastersocket.accept()
4
5     # ahora se trata el socket cliente
6     # en este caso, se trata de un servidor multihilado
7     ct = client_thread(childsocket)
8     ct.run()
```

Realmente, hay tres modos en los que este bucle puede trabajar: creando un hilo para manejar `childsocket`, creando un nuevo proceso para manejar `childsocket` o reestructurar la aplicación para usar sockets no bloqueantes, y multiplexar entre el socket servidor y uno de los `childsockets` activos usando `select()`. Lo verá más adelante. Ahora lo importante es entender que esto es *todo* lo que hace un socket servidor. No envía ningún dato. No recibe ningún

dato. Simplemente produce un socket cliente. Un `childsocket` se crea en respuesta a otro socket cliente remoto que invoca `connect()` al host y puerto al que está vinculado el socket servidor. Tan pronto como se crea el socket cliente, se vuelve a escuchar en espera de nuevas conexiones. Los dos clientes son libres de hablar - usan un puerto temporal que se reciclará cuando la conversación termine.

8.2.1. IPC

Si necesita IPC (Inter-Process Communication) rápida entre dos procesos en una misma máquina, debería echar un vistazo a las tuberías o la memoria compartida. Si decide usar sockets `AF_INET`, utilice un servidor vinculado a «localhost». En la mayoría de plataformas, esto implica un atajo a través de varias capas del código de red y puede ser bastante rápido.

8.3. Uso del socket

Lo primero que debe tener en cuenta, es que el socket «cliente» del navegador web y el socket cliente del servidor web son idénticos. Es decir, es una conversación entre iguales. O por decirlo de otra manera, *como diseñador, debe decidir qué reglas de etiqueta utilizar en la conversación*. Normalmente el socket que conecta comienza la conversación, enviando una petición, o puede que una señal de inicio. Pero eso es una decisión de diseño - no es una norma de los sockets.

Ahora hay dos conjuntos de primitivas para usar en la comunicación. Puede usar `send()` y `recv()`, o puede transformar su socket cliente en algo similar a un fichero y usar `read()` y `write()`. Esta última forma es la que usa Java en sus sockets. No vamos a tratar ese tema aquí, excepto para advertir de la necesidad de usar `flush()` en los sockets. Hay ficheros «buffereados», y un error habitual es escribir algo, y a continuación leer la respuesta. Si no se hace `flush()`, puede que tenga que esperar para siempre, porque la petición sigue en el buffer de salida y nunca llegó a ser escrita (o enviada) realmente.

Ahora llegamos al mayor tropiezo de los sockets: `send()` y `recv()` operan sobre buffers de red. No tratan necesariamente todos los bytes que se les entrega (o se espera de ellos) porque ellos están más preocupados de gestionar los buffers de red. En general, retornan cuando los buffers de red asociados se han llenado (`send()`) o vaciado (`recv()`). Es en esos momentos cuando informan de cuántos bytes han tratado. Es tu responsabilidad invocar de nuevo hasta que el mensaje haya sido aceptado completamente.

Cuando una llamada a `recv()` devuelve 0 bytes, significa que el otro lado ha cerrado (o está cerrando) la conexión. No recibirá más datos en esta conexión. Sin embargo, puede estar autorizado para enviar datos con éxito; veremos esto más adelante.

Un protocolo como HTTP usa un socket para una sola transferencia. El cliente envía una petición, lee la respuesta. Es decir. El socket es descartado. Esto significa que un cliente puede detectar el fin de una respuesta recibiendo 0 bytes.

Pero si planea reutilizar su socket para sucesivas transferencias, debes tener claro que no hay un EOT (*End of Transfer*) en un socket. Repito: si un `send()` o `recv()` indica que ha tratado 0 bytes, la conexión se ha roto. Si la conexión

no se ha roto, debería esperar en un `recv()` para siempre, porque el socket nunca dirá que no queda nada más por leer (por ahora). Ahora, si piensa un poco en ello, se dará cuenta de una cuestión fundamental de los sockets: los mensajes deben tener longitud fija, o estar delimitados, o indicar su longitud (mucho mejor) o acabar cerrando la conexión. La elección corresponde únicamente al diseñador (aunque hay algunas maneras mejores que otras).

Suponiendo que no quiere terminar la conexión, la solución más simple es utilizar mensajes de longitud fija.

```

1 class mysocket:
2     '''solo para demostracion - codificado asi por claridad, no
      por eficiencia'''
3     def __init__(self, sock=None):
4         self.sock = sock or socket.socket(socket.AF_INET, socket.
      SOCK_STREAM)
5
6     def connect(host, port):
7         self.sock.connect((host, port))
8
9     def mysend(msg):
10        totalsent = 0
11        while totalsent < MSGLEN:
12            sent = self.sock.send(msg[totalsent:])
13            if sent == 0:
14                raise RuntimeError("conexión interrumpida")
15            totalsent = totalsent + sent
16
17    def myreceive():
18        msg = bytes()
19        while len(msg) < MSGLEN:
20            chunk = self.sock.recv(MSGLEN - len(msg))
21            if chunk == b'':
22                raise RuntimeError("conexión interrumpida")
23            msg = msg + chunk
24        return msg

```

El código de envío de este ejemplo se puede usar para casi cualquier esquema de intercambio de mensajes - en Python puede enviar cadenas, y puede usar `len()` para obtener la longitud (incluso si tiene caracteres `\0`). Normalmente, es el código de recepción el que es más complejo. Y en C, no es mucho peor, excepto que no puede usar `strlen()` si el mensaje contiene `\0`.

la mejora más fácil es hacer que el primer carácter del mensaje sea un indicador del tipo del mensaje, y hacer que ese tipo determine la longitud. Ahora hay dos `recv()`: el primero lee (al menos) el primer carácter de modo que se puede averiguar el tamaño, y el segundo en un bucle para leer el resto. Si decide elegir el método del delimitador, recibirá un bloque de tamaño arbitrario, (de 4096 a 8192 normalmente es una buena elección para los tamaños de buffer de la red), y esperar a recibir el delimitador.

Una cuestión a tener en cuenta: si el protocolo conversacional permite múltiples mensajes consecutivos (sin ningún tipo de respuesta), y se le indica a `recv()` un bloque de tamaño arbitrario, puede acabar leyendo un trozo del siguiente mensaje. Necesitará guardarlo hasta que lo necesite.

Indicar la longitud del mensaje por medio de un prefijo (por ejemplo, 5 caracteres numéricos) es más complejo, porque (lo crea o no) puede no recibir esos cinco caracteres en un mismo `recv()`. Si todo va bien, funcionará; pero ante una carga alta de red, el programa fallará a menos que use dos bucles para la recepción: el primero determinará la longitud, el segundo leerá el mensaje.

Horrible. Algo similar ocurre cuando descubra que `send()` no siempre puede enviar todo en una sola pasada. Y a pesar de haber leído esto, es posible que sufra este problema de todos modos!

Para no extenderme demasiado (y preservar mi posición privilegiada) estas mejoras se dejan como ejercicio para el lector.

8.3.1. Datos binarios

Es perfectamente posible enviar datos binarios a través de un socket. El mayor problema es que no todas las máquinas usan los mismos formatos para datos binarios. Por ejemplo, un chip Motorola representa un entero de 16 bit con el valor 1 como dos bytes 0x00 0x01 (*big endian*). Intel y DEC, sin embargo, usan ordenamiento invertido - el mismo 1 es 0x01 0x00 (*little endian*). Las librerías de socket tienen funciones para convertir enteros de 16 y 32 bits - `ntohl()`, `htonl()`, `ntohs()`, `htons()` donde «n» significa red(*network*), «h» significa «host», «s» significa corto(*short*) y «l» significa «largo»(*large*). Cuando el ordenamiento de la red es el mismo que el del host, estas funciones no hacen nada, pero cuando la máquina tiene ordenamiento invertido, intercambian los bytes del modo apropiado.

En estos tiempos de máquinas de 32 bits, la representación ASCII de datos binarios normalmente ocupa menos espacio que la representación binaria. Se debe a que en una sorprendente cantidad de veces, todos esos «longs» tienen valor 0, o 1. La cadena «0» serían dos bytes, mientras que en binario serían cuatro. Por supuesto, eso no es conveniente con mensajes de longitud fija. Decisiones, decisiones.

8.4. Desconexión

Estrictamente hablando, se supone que se debe usar `shutdown()` en un socket antes de cerrarlo con `close()`. `shutdown()` es un aviso al socket del otro extremo. Dependiendo del argumento que se pasa, puede significar «No voy a enviar nada más, pero seguiré escuchando», o «No estoy escuchando». La mayoría de las librerías de sockets, no obstante, son tan usadas por programadores que descuidan el uso de esta muestra de buena educación que normalmente `close()` es `shutdown()`. De modo que en la mayoría de los casos, no es necesario un `shutdown()` explícito.

Una forma para usar `shutdown()` de forma efectiva es en un intercambio tipo HTTP. El cliente envía una petición y entonces ejecuta `shutdown(1)`. Esto le dice al servidor «Este cliente ha terminado de enviar, pero aún puede recibir». El servidor puede detectar «EOF» al recibir 0 bytes. Puede asumir que ha completado la petición. El servidor envía una respuesta. Si el envío se completa satisfactoriamente entonces, realmente, el cliente estaba aún a la escucha.

Python lleva el `shutdown` automático un paso más allá, cuando el recolector de basura trata un socket, automáticamente hará el `close()` si es necesario. Pero confiar en eso es un muy mal hábito. Si un socket desaparece sin cerrarse, el socket del otro extremo puede quedar bloqueado indefinidamente, creyendo que este extremo simplemente es lento. Por favor, cierre los sockets cuando ya no los necesite.

8.4.1. Cuando los sockets mueren

Probablemente lo peor que puede pasar cuando se usan sockets bloqueantes es lo que sucede cuando el otro extremo cae bruscamente (sin ejecutar un `close()`). Lo más probable es que el socket de este extremo «se cuelgue». `SOCKSTREAM` es un protocolo fiable, y esperará durante mucho, mucho tiempo antes de abandonar la conexión. Si está usando hilos, el hilo completo estará muerto en la práctica. No hay mucho que pueda hacer. En tanto que no haga alguna cosa absurda, como mantener un bloqueo (lock) mientras hace una lectura bloqueante, el hilo no está consumiendo realmente muchos recursos. No intente matar el hilo - parte del motivo por que los hilos son más eficientes que los procesos es que no incluyen la sobrecarga asociada a reciclarlo automático de recursos. En otras palabras, si intenta matar el hilo, es probable que fastidie el proceso completo.

8.5. Sockets no bloqueantes

Si ha comprendido todo lo anterior, ya sabe más que de lo que necesita saber sobre la mecánica de los sockets. Usará casi siempre las mismas funciones, y del mismo modo. Sólo es eso, si lo hace bien no tendrá problemas.

En Python, se usa `socket.setblocking(0)` para hacer un socket no bloqueante. En C, es más complicado, (necesita elegir entre BSD modo `O_NONBLOCK` o el casi idéntico Posix modo `O_NDELAY`, que es completamente diferente de `TCP_NODELAY`), pero es exactamente la misma idea. Eso se hace después de crear el socket, pero antes de usarlo. Realmente, si está un poco chiflado, puede cambiar entre un modo y otro.

La diferencia más importante es que `send()`, `recv()`, `connect()` y `accept()` puede volver sin haber hecho nada. Hay, por supuesto, varias alternativas. Puede comprobar el valor de retorno y el código de error y generalmente volverse loco. Si no lo cree, debería intentarlo alguna vez. Su aplicación pronto será grande, pesada y llena de errores. De modo que pasemos de las soluciones absurdas y hagámoslo bien.

Use `select`.

En C, escribir código para `select()` puede ser complicado. En Python, está chupado, pero es lo suficientemente parecido a la versión C como para que si entiende `select()` en Python, no tenga muchos problemas en C.

```

1      listo_para_leer, listo_para_escribir, en_error = \
2          select.select(lectores_potenciales,
3                        escritores_potenciales,
4                        errores_potenciales,
5                        timeout)

```

A `select()` se le pasan tres listas: la primera contiene todos los sockets de los que quiere intentar leer; la segunda todos los sockets en los que quiere intentar escribir, y por último (normalmente vacía) aquellos en los que quiere comprobar se ha producido un error. La llamada a `select()` es bloqueante, pero se le puede indicar un timeout. Generalmente es aconsejable indicar un timeout - indique un tiempo largo (digamos un minuto) a menos que tenga una buena razón para hacer otra cosa.

Al retornar, devuelve tres listas. Son las listas de sockets en los que realmente

se puede leer, escribir y tienen un error. Cada una de esas listas es un subconjunto (puede que vacío) que la lista correspondiente que se paso como parámetro en la llamada. Y si pone un socket en más de una lista de entrada, sólo estará (como mucho) en una de las listas de retorno.

Si un socket está en la lista de salida de «legibles», puede estar seguro que al invocar `recv` recibirá algo. Lo mismo es aplicable a la lista de «escribibles». Puede que esto no es lo que quería, pero algo es mejor que nada. (Realmente, cualquier socket razonablemente sano retornará como «escribible» - únicamente significa que hay espacio disponible en el buffer de salida de red.

Si se tiene un socket «servidor», se debe poner en la lista de lectores_potenciales. Si retorna en la lista de «legibles», una invocación a `accept()` funcionará casi con toda seguridad. Si se crea un socket nuevo para conectar a algún sitio, debe ponerse en la lista de escritores_potenciales. Si aparece en la lista de «escribibles», existen ciertas garantías de que haya conectado.

Hay un problema muy feo con `select()`: Si en alguna de las listas de entrada hay un socket que ha muerto de mala manera, `select()` fallará. Entonces necesitará recorrer todos los sockets (uno por uno) de las tres listas y hacer `select([sock], [], [], 0)` hasta que encuentre al responsable. El timeout 0 significa que debe retornar inmediatamente.

En realidad, `select()` puede ser útil incluso con sockets bloqueantes. Es un modo para determinar si quedará bloqueado - el socket retorna como «legible» cuando hay algo en el buffer. Sin embargo, esto no ayuda con el problema de determinar si el otro extremo ha terminado, está ocupado con otra cosa.

Advertencia de portabilidad En `\UNIX{}`, `select()` funciona tanto con sockets como con ficheros. No intente esto en Windows. En Windows, `select()` sólo funciona con sockets. También debe notar que en C, muchas de las opciones avanzadas con sockets son diferentes en Windows. De hecho, en Windows normalmente se usan hilos (que funcionan muy bien) con sockets. Si quiere rendimiento, su código será muy diferente en Windows y en `\UNIX{}`.

8.5.1. Rendimiento

No hay duda de que el código más rápido es el que usa sockets no bloqueantes y `select()` para multiplexarlos. Se puede enviar una cantidad inmensa de datos que saturen una conexión LAN sin que suponga una carga excesiva para la CPU. El problema es que una aplicación escrita de este modo no puede hacer mucho más - necesita estar lista para generar bytes en cualquier momento.

Asumiendo que se supone que su aplicación hará algo más que eso, utilizar hilos es la solución óptima, (y usar sockets no bloqueantes es más rápido que usar sockets bloqueantes). Desafortunadamente, el soporte de hilos en los `\UNIX`'es varia en API y calidad. Así que la solución habitual en `\UNIX{}` es crear un subproceso para manejar cada conexión. La sobrecarga que eso supone es significativa (y en Windows ni lo haga - la sobrecarga por la creación de procesos es enorme en Windows). También implica que, a menos que cada subproceso sea completamente independiente, necesitarás usar otra forma de IPC, como tuberías, o memoria compartida y semáforos, para comunicar el padre y los procesos hijos.

Finalmente, recuerde que a pesar de que los sockets bloqueantes son algo más lentos que los no bloqueantes, en muchos casos son la solución «correcta».

Después de todo, si su aplicación reacciona ante los datos que recibe de un socket, no tiene mucho sentido complicar la lógica sólo para que la aplicación pueda esperar en un `select()` en lugar de en un `recv()`.

8.6. Créditos

Este documento está basado en «Socket Programming HOWTO» de Gordon Mc Millan, disponible en <http://docs.python.org/howto/sockets.html>. Su licencia (Python Software Foundation) permite la realización de trabajos derivados como éste a condición de mantener dicha licencia. Por esta razón, este capítulo concreto mantiene la licencia PSF¹

¹<http://docs.python.org/license.html>

Modelo Cliente-Servidor

David Villa
Fernando Rincón

El modelo cliente-servidor es probablemente el enfoque más simple, y más popular, para la construcción de aplicaciones distribuidas en la red. La mayoría de los protocolos clásicos de **aplicación** de Internet (HTTP, FTP, IMAP, SMTP, etc.) están basados en este modelo, y a día de día, la mayoría de aplicaciones que utilizamos siguen aplicando la misma arquitectura básica.

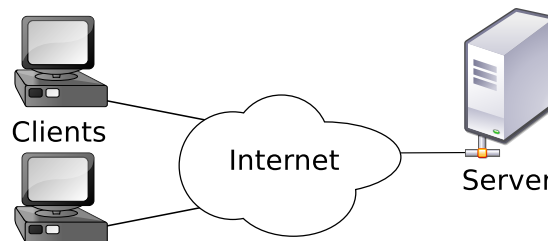


FIGURA 9.1: Modelo Cliente/Servidor [Fuente:Wikimedia Commons]

En este modelo se asumen dos roles bien diferenciados:

- El **servidor** es la parte pasiva. Permanece inactiva en espera de una petición para realizar una tarea, servicio o proporcionar acceso a un recurso a través de la red.
- El **cliente** es la parte activa. Establece una conexión o envía una petición a un servidor para que éste realice la tarea especificada o bien le proporcione acceso a un recurso remoto.

Cliente y *servidor* son **roles** que puede tomar una aplicación, es decir, no definen tipos de aplicaciones. Es cierto que normalmente se habla de aplicaciones diferentes, por ejemplo: el servidor y el cliente web (navegador). Sin embargo, nada impide y de hecho es bastante habitual, que una aplicación que es servidora para un propósito concreto sea a su vez cliente de un segundo servidor.

E 9.01 Nombre los tres servidores y clientes HTTP más usados en el mundo (y cite la fuente).

También es habitual referirse a los computadores como *servidores* o *clientes* para denotar que su utilidad principal es la de alojar programas con el sesgo correspondiente. Aunque la arquitectura de un computador utilizado en producción como *servidor* no difiere en absoluto de uno usado como *cliente* es fácil

encontrar algunas características diferenciadoras: el servidor suele disponer de un mayor ancho de banda, está situado en un *rack* y no dispone de pantalla, teclado, ratón u otros periféricos habituales, ya que ninguna persona lo utiliza para realizar tareas propias de escritorio.

Sin embargo, en un entorno de desarrollo las aplicaciones servidoras se ejecutan en los computadores personales de los programadores. Incluso en un entorno doméstico es relativamente habitual que los usuarios ejecuten en sus máquinas servidores de diversa índole, aunque por supuesto lo normal es que sus aplicaciones realicen el rol de *cliente*.

El modelo cliente-servidor implica un patrón de comunicación característico conocido como *petición-respuesta*. En éste, el cliente realiza una petición —que puede incluir un identificador de un recurso y una operación— y el servidor devuelve una respuesta indicando un resultado o bien un código que indica si la operación se pudo realizar satisfactoriamente o se produjo un error. El formato de estos mensajes de petición y respuesta está especificado en un protocolo de aplicación. Por supuesto, existe una gran cantidad de variantes, pero esa es normalmente la pauta.

Otro aspecto interesante es que el modelo asume la existencia de múltiples clientes que operan sobre un único servidor. Eso determina la forma en la que se construye este último, ya que son necesarios mecanismos que gestionen accesos por parte de diferentes *usuarios* (sean personas o no) a un recurso (p.ej. una impresora). Este tipo de recurso *compartido* se denomina de *tiempo compartido*. Pero es muy habitual que el recurso tenga una granularidad mucho menor (p.ej. registros de una base de datos) en cuyo caso se convierte en un problema de acceso concurrente, que obliga a gestionar los accesos de modo que se pueda asegurar la integridad de los recursos (datos en este caso).

9.1. Chat

Una de las aplicaciones de red más simples que se puede programar es un *chat*, es decir, un programa que permite a dos personas intercambiar mensajes de texto a través de la red. En esta sección se aborda paso a paso la construcción de una aplicación de chat con complejidad creciente.

9.1.1. Paso 1: Mensaje unidireccional

En este primer paso el objetivo es el siguiente:

- Implementar un servidor UDP que ha recibir un único mensaje, imprimirlo en consola y terminar.
- Implementar un cliente UDP que debe enviar la cadena "hello" al servidor anterior y terminar.

Servidor

Las tareas que ha de realizar el servidor son muy simples:

1. Crear un socket UDP.
2. Vincular dicho socket a un puerto libre.
3. Esperar un datagrama que transporta un mensaje de texto.

4. Imprimir el mensaje en consola.

Estas tareas corresponden línea a línea con el listado 9.1.

LISTADO 9.1: Servidor de chat UDP básico
chat/udp_server.py

```

1 import socket
2
3 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4 sock.bind('', 12345)
5 message, peer = sock.recvfrom(1024)
6 print(message)

```

La línea 3 crea el objeto `sock`, que es una instancia de la clase `socket.socket`. Los argumentos del constructor son `socket.AF_INET` (que indica que es un socket de la *familia* de protocolos de Internet y `SOCK_DGRAM`, que indica que debe utilizar un protocolo de transporte tipo datagrama. El protocolo de transporte de TCP/IP que cumple con eso es UDP, de modo que `sock` es un socket UDP.

Para que los clientes puedan referirse a este servidor necesitan indicar un puerto. Eso se consigue pidiendo al SO, mediante el método `bind()`, que asocie el socket recién creado a un puerto determinado (que debe estar libre). En realidad el argumento de `bind()` es una tupla formada por una dirección IP (o un nombre de dominio) y un número de puerto. Lo más sencillo es indicar la IP "0.0.0.0" o "", que le dice al SO que vincule este socket en todas las direcciones IP asociadas a todas las interfaces de red de este computador. De no ser así, habría que indicar la IP concreta que tiene asignada el computador, y eso no es siempre una tarea sencilla...

- E 9.02** ¿Cómo se puede averiguar si un puerto determinado está libre u ocupado (vinculado a un servidor activo) en el propio computador? Si está ocupado ¿cómo puedes averiguar qué programa es el responsable?
- E 9.03** ¿Cómo se puede averiguar si hay un servidor vinculado a un puerto en un computador remoto?
- E 9.04** Escriba un programa Python que cree un socket UDP e intente vincularlo a un puerto ocupado ¿qué error se obtiene?
- E 9.05** La dirección IP "0.0.0.0" es una dirección especial según indica la RFC3330 [IAN02]. ¿Cuál es su propósito?
- E 9.06** ¿Por qué cree que el texto dice que averiguar la dirección IP asignada al computador puede no ser una tarea sencilla?

La línea 5 invoca el método `recvfrom()` indicando que está dispuesto a leer un máximo de 1024 bytes, y retorna dos valores: el primero (`message`) es la carga útil del datagrama UDP (o al menos los 1024 primeros bytes). El segundo valor (`peer`) es una tupla que identifica al cliente que ha enviado el mensaje. Esa tupla tiene el mismo formato que el argumento del método `bind()`, es decir, una dirección IP expresada como cadena y un entero que indica el puerto del socket del cliente.

Para ejecutar este programa simplemente:

```
$ python udp_chat_server.py
```

El programa queda inmediatamente bloqueada a la espera de un mensaje de un cliente. En concreto el programa está bloqueado en la invocación del método `recvfrom()`.

Cliente

Antes de escribir un programa cliente en Python es interesante probar que el servidor de la sección anterior funciona como se espera. Una herramienta extremadamente útil para este tipo de tareas es `netcat` (ver anexo A).

El siguiente comando le dice a `ncat` que cree un socket UDP y envíe la cadena "hola" al servidor que está escuchando en el puerto 12345 (el de la sección anterior). En este instante el servidor debería imprimir la cadena "hola" y terminar.

```
$ echo hola | ncat --udp --send-only 127.0.0.1 12345
```

Ahora que se ha comprobado que el servidor funciona es momento de escribir un cliente que imite la funcionalidad del comando anterior. Es muy simple, lo puede ver en el listado 9.2.

LISTADO 9.2: Client de chat UDP básico
chat/udp_client.py

```
1 import socket
2
3 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4 sock.sendto('hola', ('127.0.0.1', 12345))
```

La línea 1 de este listado crea el socket UDP (idéntica a la del servidor) y la línea 2 envía un datagrama con el contenido "hola" al servidor que escucha en el puerto 12345. Como se ha dicho, su ejecución debería tener el mismo efecto que el comando con `netcat`.

E 9.07 Captura el tráfico de red, ejecuta de nuevo servidor y cliente, y busca el mensaje que contiene la cadena "hola" ¿Cuántos mensajes están involucrados en la comunicación? ¿por qué?

E 9.08 Utiliza `netcat` para probar el cliente, es decir, identifica el comando adecuado para substituir el servidor UDP mediante un comando `netcat`. ¿Cuál es ese comando?

9.1.2. Paso 2: Lo educado es responder

El servidor del paso anterior sólo imprime el mensaje recibido. Una pequeña modificación le permitirá devolver el saludo.

La modificación en el servidor es simple (una línea de código). Utilizando la dirección del cliente que se obtenía como valor de retorno del método `recvfrom()` la aplicación puede a su vez utilizar el método `sendto()` y enviar un mensaje de respuesta al cliente (ver listado 9.3).

LISTADO 9.3: Servidor de chat UDP con respuesta
chat/udp_server2.py

```

1  import socket
2
3  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4  sock.bind(('', 12345))
5  message, peer = sock.recvfrom(1024)
6  print(message)
7  sock.sendto('qué tal?', peer)

```

El cliente del paso anterior terminaba inmediatamente después de enviar su mensaje. Ahora debe esperar la respuesta. Como es lógico basta con imitar lo que hacer el servidor, usar el método `recvfrom()` (ver listado 9.4).

LISTADO 9.4: Cliente de chat UDP con respuesta
chat/udp_client2.py

```

1  import socket
2
3  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4  sock.sendto('hola', ('127.0.0.1', 12345))
5  message, peer = sock.recvfrom(1024)
6  print(message)

```

E 9.09 Ejecuta servidor y cliente y comprueba que efectivamente el servidor responde y el cliente imprime el mensaje en su consola.

E 9.10 Prueba servidor y cliente por separado con `netcat`.

9.1.3. Paso 3: Libertad de expresión

En este paso los usuarios que ejecutan cliente y servidor tendrán realmente la oportunidad de conversar. Para ello, ambos programas deben leer de consola lo que el usuario teclee y enviarlo hacia su interlocutor. Además, podrán enviar cuantos mensajes quieran. La conversación se mantendrá hasta que cualquiera de ellos envíe la cadena "bye". El listado 9.5 muestra el código completo del servidor:

LISTADO 9.5: Servidor de chat UDP por turnos
chat/udp_chat_server3.py

```

1  import socket
2  QUIT = b'bye'
3
4  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5  sock.bind(('', 12345))
6
7  while 1:
8      msg_in, peer = sock.recvfrom(1024)
9      print(msg_in.decode())
10
11     if msg_in == QUIT:
12         break
13
14     msg_out = input().encode()
15     sock.sendto(msg_out, peer)
16
17     if msg_out == QUIT:
18         break

```

La diferencia principal respecto a las versiones anteriores es el bucle `while`. Este bucle termina si el usuario introduce la cadena "bye" o bien es recibida

a través del socket. Para leer una cadena de texto de la consola se utiliza la función `input()`. El código del cliente (listado 9.6) es muy similar.

LISTADO 9.6: Cliente de chat UDP por turnos
chat/udp_chat_client3.py

```

1  import socket
2  QUIT = b'bye'
3
4  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5  server = ('', 12345)
6
7  while 1:
8      msg_out = input().encode()
9      sock.sendto(msg_out, server)
10
11     if msg_out == QUIT:
12         break
13
14     msg_in, peer = sock.recvfrom(1024)
15     print(msg_in.decode())
16
17     if msg_in == QUIT:
18         break

```

La única diferencia es que, lógicamente, sólo el servidor ejecuta `bind()`. El servidor empieza esperando un mensaje entrante mientras el cliente empieza leyendo de teclado y enviando al servidor.

E 9.11 Ejecuta servidor y cliente en computadores diferentes. ¿por qué no funciona? Haz las modificaciones necesarias para lograr que funcione.

E 9.12 Reemplaza el servidor por `netcat` y explica las diferencias.

E 9.13 Reemplaza el cliente por `netcat` y explica las diferencias.

9.1.4. Paso 4: Habla cuando quieras

La versión de la sección anterior tiene un problema grave. Tanto el cliente como el servidor tienen dos puntos diferentes en los que el programa queda bloqueado: la función `input()` para leer de consola y el método `recvfrom()` para leer del socket. Eso implica que cada interlocutor debe esperar a que su interlocutor envíe un mensaje antes de poder escribir de nuevo.

Ese es un problema clásico y recurrente. El programa necesita atender al mismo tiempo dos (en este caso) o más fuentes de datos asíncronas, es decir, que pueden enviar datos en cualquier momento. En este tipo de programas se suele asociar un bloque de código (un *manejador*) a un evento asíncrono (no predecible). Este enfoque se denomina *programación dirigida por eventos*¹ y es típico de las interfaces gráficas de usuario.

Existen varias formas de solucionar ese problema. La que se propone aquí consiste en atender una de las entradas (la consola) en un hilo adicional mientras que el hilo principal se utiliza para atender la entrada desde el socket. El servidor aparece en el listado 9.7.

LISTADO 9.7: Servidor de chat UDP simultaneo
chat/udp_chat_server_thread.py

¹http://es.wikipedia.org/wiki/Programación_dirigida_por_eventos

```

1  import socket
2  import _thread
3  server = ('', 12345)
4  QUIT = b'bye'
5
6  class Chat:
7      def __init__(self, sock, peer):
8          self.sock = sock
9          self.peer = peer
10
11         _thread.start_new_thread(self.sending, ())
12         self.receiving()
13
14     def sending(self):
15         while 1:
16             msg = input().encode()
17             self.sock.sendto(msg, self.peer)
18
19             if msg == QUIT:
20                 break
21
22     def receiving(self):
23         while 1:
24             msg, peer = self.sock.recvfrom(1024)
25             print(msg.decode())
26
27             if msg == QUIT:
28                 self.sock.sendto(QUIT, self.peer)
29                 break
30
31 if __name__ == '__main__':
32     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
33     sock.bind(server)
34     msg, client = sock.recvfrom(0, socket.MSG_PEEK)
35     Chat(sock, client)

```

El programa está compuesto por una clase Chat con tres métodos, incluyendo el constructor. El método `sending()` se ocupa de leer líneas de texto de la consola y enviarlas a través del socket. El método `receiving()` lee líneas de texto del socket y las imprime en la consola. En ambos casos, si el mensaje leído o recibido es "bye" la función termina. En el caso de la función `receiving()` además devuelve el mensaje para que el hilo de recepción del otro extremo también termine. El constructor simplemente copia el socket y la dirección del otro extremo como atributos, crea el hilo para la tarea de envío y ejecuta la tarea de recepción.

En cuanto a la función principal, la llamada a `recvfrom()`, en la línea 33, se utiliza únicamente para obtener la dirección del cliente, pero no lee nada del buffer del socket (gracias al flag `MSG_PEEK`), y la línea 34 que crea una instancia de la clase Chat pasándole el socket y la dirección del cliente.

El cliente solo difiere en la creación del socket. Simplemente crea el socket (pero no lo vincula), crea una instancia de la clase Chat (la misma del servidor) y le pasa dicho socket y la dirección del servidor. Puede verlo en el listado 9.8.

LISTADO 9.8: Cliente de chat UDP simultáneo
chat/udp_client_thread.py

```

1  import socket
2  from udp_server_thread import Chat, server
3
4  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5  Chat(sock, server)

```

Sigue habiendo un pequeño problema de uso. Como el punto de entrada del usuario y el lugar donde se escriben los mensajes que se reciben es el mismo (la salida estándar), es fácil que se mezclen dificultando el uso del programa. Para solucionarlo bastaría con que las tareas de recepción y envío escribieran en partes diferentes de la pantalla, o quizá construir un pequeño GUI, pero ésta es una cuestión estética que excede el ámbito de este ejemplo.

- E9.14** Una vez que tengas cliente y servidor en ejecución, envía algo (escribe y pulsa ENTER) en el servidor antes que escribir nada en la consola del cliente. ¿Por qué no llega el mensaje?
- E9.15** Escribe una versión TCP de cliente y servidor.
- E9.16** Cuando se usa netcat como chat se pueden enviar y recibir mensajes en cualquier momento ¿cómo lo hace?
- E9.17** Implementa una solución alternativa que utilice `select()` o `poll()` en lugar de crear un hilo para cada entrada asíncrona.

9.1.5. Paso 5: Todo en uno

El servidor y el cliente del paso anterior utilizan la misma clase Chat para resolver la mayor parte del problema. Lo único diferente entre servidor y cliente es el código de la función principal (lo que está fuera de clase Chat). Eso significa que podríamos crear un único programa que se comparte como servidor o cliente en función de un parámetro de línea de comandos. Puedes verlo en el listado 9.9.

LISTADO 9.9: Chat UDP (servidor y cliente)
chat/udp_chat.py

```

1
2 import sys
3 import socket
4 import _thread
5 server = ('', 12345)
6 QUIT = b'bye'
7
8 class Chat:
9     def __init__(self, sock, peer):
10         self.sock = sock
11         self.peer = peer
12
13         _thread.start_new_thread(self.sending, ())
14         self.receiving()
15
16     def sending(self):
17         while 1:
18             msg = input().encode()
19             self.sock.sendto(msg, self.peer)
20
21             if msg == QUIT:
22                 break
23
24     def receiving(self):
25         while 1:
26             msg, peer = self.sock.recvfrom(1024)
27             print(msg.decode())
28
29             if msg == QUIT:
30                 self.sock.sendto(QUIT, self.peer)
31                 break
32
33 def error():

```



```

34     print(__doc__ % sys.argv[0])
35     sys.exit()
36
37 if __name__ == '__main__':
38     if len(sys.argv) != 2:
39         error()
40
41     mode = sys.argv[1]
42     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
43
44     if mode == '--server':
45         sock.bind(server)
46         msg, client = sock.recvfrom(0, socket.MSG_PEEK)
47         Chat(sock, client)
48
49     elif mode == '--client':
50         Chat(sock, server)
51
52     else:
53         error()

```

9.2. Ejemplos rápidos de sockets

9.2.1. Un cliente HTTP básico

LISTADO 9.10: Cliente HTTP básico
examples/http_mini_client.py

```

1 import socket
2 s = socket.socket()
3 s.connect(('insecure.org', 80))
4 s.send(b'GET /\n')
5 print(s.recv(2048))

```

9.2.2. Un servidor HTTP

LISTADO 9.11: Servidor HTTP
examples/http_server.py

```

1 from http.server import HTTPServer, SimpleHTTPRequestHandler
2
3 server = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
4 print("Open http://{0}:{1}".format(*server.socket.getsockname()))
5 try:
6     server.serve_forever()
7 except KeyboardInterrupt:
8     server.server_close()

```

o simplemente:

```
$ python3 -m http.server
```


Puertos y servicios

David Villa

Junto con las herramientas que ya se han mostrado (ping, traceroute o wireshark) existen otro tipo de aplicaciones que permiten obtener la configuración de equipos desde el punto de vista de los servicios de red, qué puertos TCP o UDP tienen abiertos, etc.

En concreto se describen las siguientes herramientas:

netstat

Sirve para monitorizar conexiones de red, tablas de encaminamiento, estadísticas de interfaces, grupos multicast, etc.

nmap

Se trata de una potente herramienta de exploración de red y escaneo de puertos.

IPTraf

Es un monitor protocolos TCP/IP.

Antes de pasar a ver un mini-tutorial de estas herramientas en detalle, debes saber que la lista de puertos asignados a servicios se puede encontrar en la RFC 1700. Estos puertos tienen asociados servicios específicos y no deberían ser usados por otros programas, al menos por debajo del 1024.

El objetivo de este tema es comprender y aprender a manejar un conjunto de herramientas útiles para:

- Depurar y optimizar aplicaciones de red en desarrollo.
- Evaluar la configuración de un equipo desde el punto de vista de la seguridad.
- Descubrir, diagnosticar y solucionar problemas de seguridad o accesibilidad.

10.1. netstat

`netstat` muestra información sobre los subsistemas de red en GNU/Linux. El uso más sencillo (sin opciones) muestra el estado de todos los sockets abiertos, tanto de la familia Internet como Unix, pero puede hacer mucho más. Como es habitual en los programas de consola, la opción `-h` ofrece información resumida de todas sus opciones:

```
usage: netstat [-veenNcCF] [<Af>] -r
       netstat \{-V|--version|-h|--help\}
       netstat [-vnNcaeol] [<Socket> ...]
```

```

netstat { [-veenNac] -i | [-cnNe] -M | -s \}

-r, --route                display routing table
-i, --interfaces           display interface table
-g, --groups               display multicast group
                           memberships
-s, --statistics           display networking statistics (
                           like SNMP)
-M, --masquerade           display masqueraded connections

-v, --verbose              be verbose
-n, --numeric              don't resolve names
--numeric-hosts            don't resolve host names
--numeric-ports            don't resolve port names
--numeric-users            don't resolve user names
-N, --symbolic             resolve hardware names
-e, --extend               display other/more information
-p, --programs             display PID/Program name for
                           sockets
-c, --continuous          continuous listing

-l, --listening            display listening server sockets
-a, --all, --listening    display all sockets (default:
                           connected)
-o, --timers               display timers
-F, --fib                  display Forwarding Information
                           Base (default)
-C, --cache                display routing cache instead of
                           FIB

<Socket>={-t|--tcp\} \{-u|--udp\} \{-w|--raw\} \{-x|--unix\} --
ax25 --ipx --netrom
<AF>=Use '-6|-4' or '-A <af>' or '--<af>'; default: inet
List of possible address families (which support routing):
inet (DARPA Internet) inet6 (IPv6) ax25 (AMPR AX.25)
netrom (AMPR NET/ROM) ipx (Novell IPX) ddp (Appletalk DDP)
x25 (CCITT X.25)

```

Mediante unos cuantos ejemplos sencillos es sencillo entender y aprender el uso y posibilidades más habituales del programa.

10.1.1. Visualizar la tabla de rutas

netcat se suma a la lista de programas que ofrecen información resumida de la tabla de rutas del nodo.

```

$ netstat -r
Kernel IP routing table
Destination      Gateway          Genmask          Flags   MSS Window  irtt  Iface
localnet         *                255.255.255.0    U        0 0           0 eth0
default          161.67.212.1    0.0.0.0          UG        0 0           0 eth0

```

10.1.2. Lista de interfaces de red

Ofrece información detallada de todas las interfaces del nodo en formato de tabla.

```

$ netstat -i
Kernel Interface table
Iface  MTU Met  RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0   1500 0      47774      0      0      0      7991      0      0      0 BMRU
lo     16436 0       280      0      0      0       280      0      0      0 LRU

```

10.1.3. Listar servidores

Éste es el uso más habitual que se le pide a `netstat`. Muestra una lista de los sockets vinculados (LISTEN) de todas las familias.

```
$ netstat -l
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:51413                :::*                     LISTEN
[...]
Active UNIX domain sockets (only servers)
Proto RefCnt Flags       Type       State         I-Node  Path
unix    2      [ ACC ]     STREAM    LISTENING     6669    /var/run/dbus/system
[...]
```

10.1.4. Filtrar listado de sockets

Las diferentes opciones permiten filtrar qué tipo de socket se desea que aparezcan en la lista. Así, `-u` se refiere a los sockets UDP y `-t` a los TCP. Por tanto para ver un listado de los servidores TCP y UDP activos has de ejecutar:

```
$ netstat -ltu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:51413                :::*                     LISTEN
tcp        0      0 *:ssh                   :::*                     LISTEN
tcp        0      0 *:ipp                   :::*                     LISTEN
[...]
```

10.1.5. Otras opciones

Algunas otras opciones interesantes son:

-p Muestra el PID del proceso que ha creado el socket. Resulta muy útil para determinar qué programa es el responsable de haber abierto el puerto correspondiente. El uso de esta opción requiere privilegios de administrador.

-n/--numeric

No «resuelve» los nombres de puertos y hosts. Es decir, mostrará direcciones IP en lugar de nombres de dominio y números de puerto en lugar de nombres de protocolos.

10.2. nmap

`nmap` es un escáner de puertos remoto potente y flexible (entre otras cosas). Se podría decir que `nmap` es la contrapartida remota de `netstat`, en el sentido de que permite conocer qué puertos tiene abiertos un host.

Puede resultar muy útil para servicios está ofreciendo un host cualquiera o para determinar qué componente de la red es el responsable de un problema de conectividad a nivel de enlace. En particular permite saber si un puerto está «filtrado», es decir, si un cortafuegos está impidiendo el tráfico hacia o desde determinado puerto.

El uso más básico consiste en indicar una dirección IP o nombre de dominio de un host remoto:

```
$ nmap uclm.es

Starting Nmap 5.21 ( http://nmap.org ) at 2010-12-02 18:14 CET
Nmap scan report for uclm.es (172.20.96.8)
Host is up (0.0021s latency).
Hostname uclm.es resolves to 4 IPs. Only scanned 172.20.96.8
rDNS record for 172.20.96.8: dcto01.uclm.es
Not shown: 986 closed ports
PORT      STATE SERVICE
88/tcp    open  kerberos-sec
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
389/tcp   open  ldap
445/tcp   open  microsoft-ds
464/tcp   open  kpasswd5
593/tcp   open  http-rpc-epmap
636/tcp   open  ldapssl
1025/tcp  open  NFS-or-IIS
1029/tcp  open  ms-lsa
1093/tcp  open  unknown
2301/tcp  open  compaqdiag
2381/tcp  open  unknown
3389/tcp  open  ms-term-serv

Nmap done: 1 IP address (1 host up) scanned in 1.21 seconds
```

Entre las muchas opciones que soporta nmap se pueden destacar las siguientes:

-p{rango}

Permite indicar qué rango de puertos se desea escanear. Ej: \$ nmap -p22-80 uclm.es.

-sU Para escanear puertos UDP. Por defecto solamente escanea puertos TCP.

-sP Envía un mensaje ping a todas las máquinas del bloque indicado y muestra un listado con aquellas que parecen estar activas. Por ejemplo \$ nmap -sP 161.67.136.*.

-O Intenta determinar el sistema operativo del host. Ej: # nmap -O www.uclm.es.

-iL Permite especificar un fichero que contiene una lista de hosts sobre los que se desea realizar el escaneado.

10.3. IPTraf

Genera estadísticas en tiempo real sobre diversos protocolos de red y transportes, tales como IP, ICMP, UDP, TCP, etc. IPTraf tiene dos modos de operación, uno interactivo por medio de un interfaz de ventanas sencillo, y otro en línea de comandos.

En la siguiente figura se muestra el modo interactivo, en el que posible navegar por los distintos menús para configurar la herramienta y seleccionar el tipo de estadísticas que se pretende visualizar.

Con las flechas arriba/abajo se pueden elegir las distintas opciones, y mediante la tecla ENTER se ejecutan. Si desea ver las estadísticas IP, seleccione la interfaz que corresponda:

y se visualizan las estadísticas:

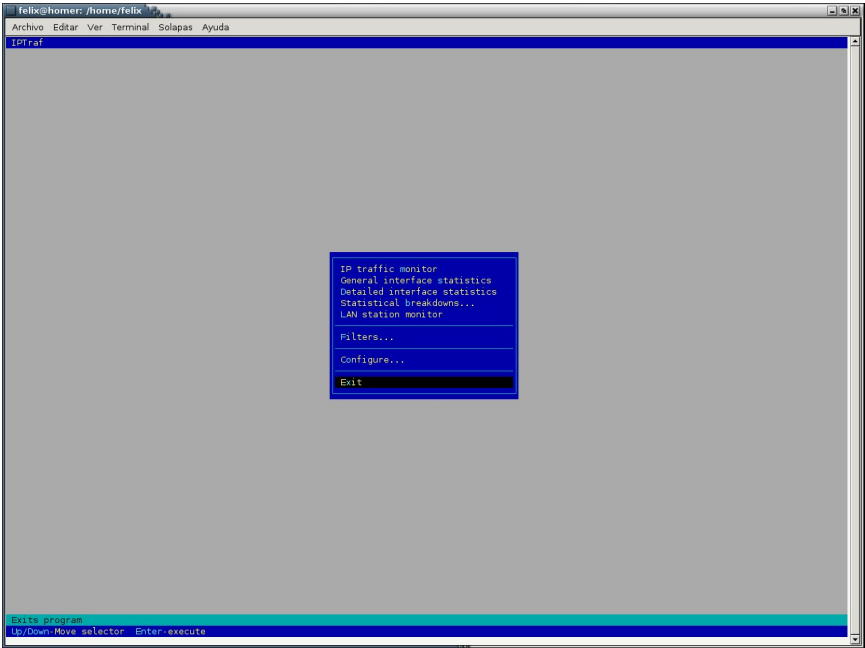


FIGURA 10.1

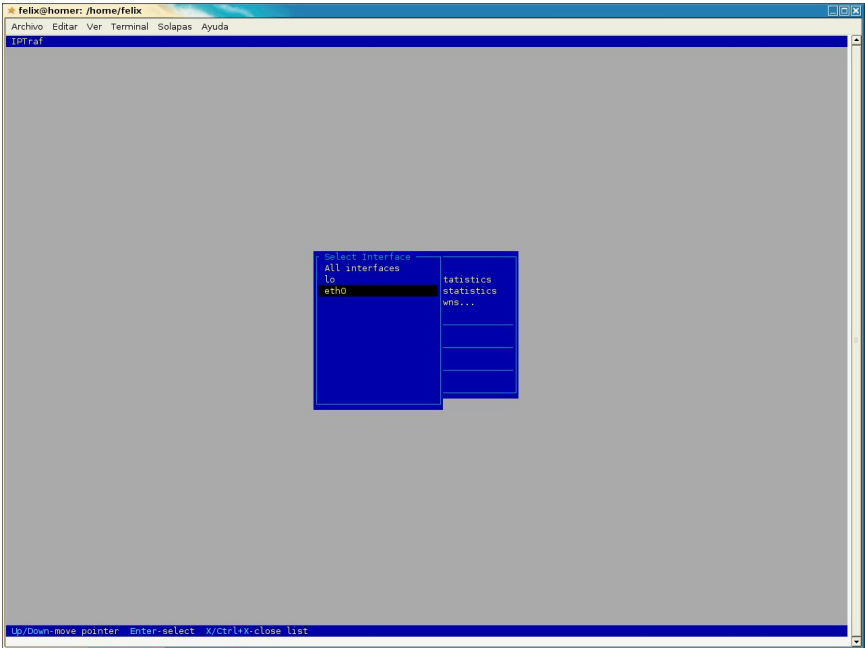


FIGURA 10.2

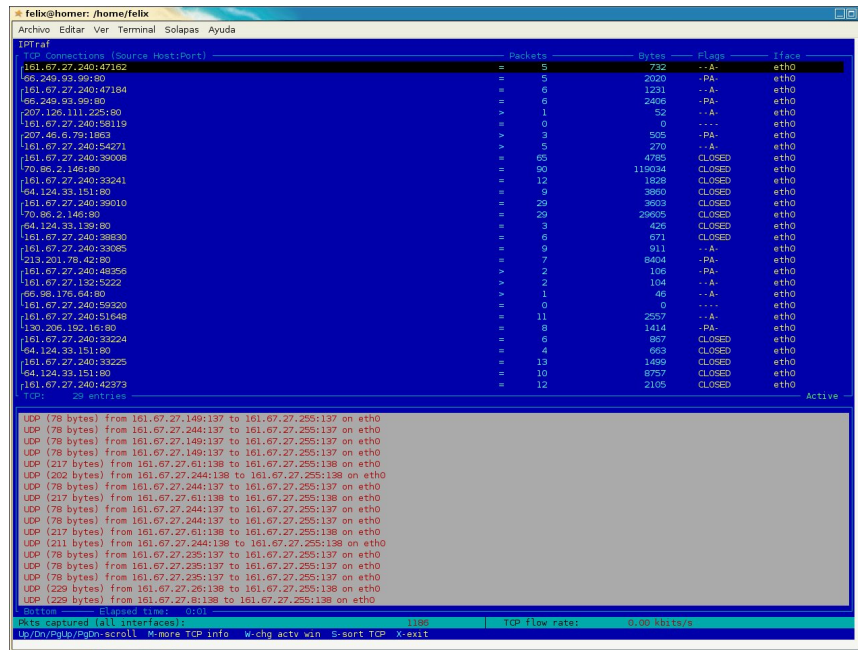
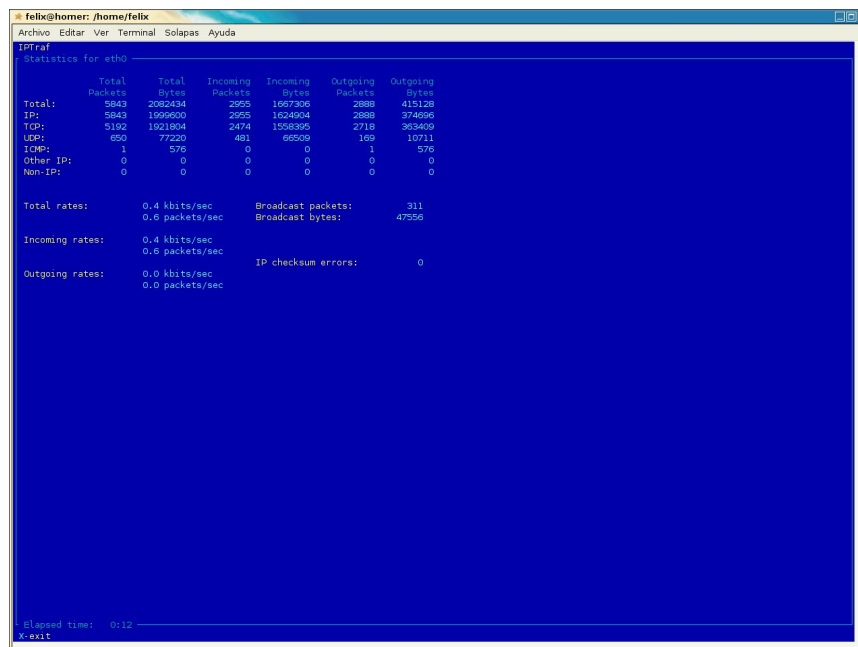


FIGURA 10.3

Donde se pueden ver las estadísticas para el interfaz seleccionado y por conexión. También se pueden visualizar estadísticas generales para todas las interfaces detectadas o estadísticas por interfaz como en la siguiente figura:



De igual forma se pueden establecer y definir filtros para los distintos protocolos.

10.4. Referencias

- [Página oficial de nmap.](#)

- [Página oficial de IPTraf.](#)

Sockets RAW

David Villa

Los sockets más usados con diferencia son los TCP seguidos de los UDP. Sin embargo, hay muchos otros tipos de socket. Este capítulo es una introducción muy práctica a los «sockets RAW», concepto a veces traducido como «conector directo».

El término «raw» en informática suele hacer referencia al acceso directo a un recurso, o al menos con menor intervención de terceros (normalmente el sistema operativo)¹. Este acceso directo tiene tres implicaciones principales:

- Mayor flexibilidad, al no estar limitado por las reglas o normas que impongan las capas de alto nivel que ofrece el sistema operativo.
- Acceso privilegiado, debido precisamente a que dichas posibilidades tienen un impacto directo sobre la seguridad del sistema y la privacidad de sus usuarios.
- Menos soporte, ya que son precisamente las capas del sistema operativo que se dejan a un lado las que simplifican el manejo del recurso. El «modo RAW» conlleva un nivel de abstracción mucho menor y por tanto, más complejidad técnica.

Estas tres cuestiones se pueden aplicar casi a cualquier dispositivo que permita un acceso «raw», sea un periférico USB, una consola o como en este caso un socket.

Con los sockets AF_INET:SOCK_STREAM o AF_INET:SOCK_DGRAM no es posible acceder (para leer o escribir) a las cabeceras de ninguno de los protocolos de TCP/IP, ya sea IP, ICMP, ARP o TCP. Esos sockets únicamente permiten indicar cuál será la carga útil de los segmentos TCP o UDP y solo indirectamente se puede influir en algunos de los campos de sus cabeceras: puerto origen y destino y poco más.

Algunas veces (no frecuentemente) se necesita ofrecer servicios que implican a protocolos de capas 2 y 3, o a las cabeceras de tramas, paquetes y segmentos, que normalmente quedan fuera de la vista del programador. Algunos programas de este tipo pueden ser el programa ping, traceroute, arping o un *sniffer* cualquiera. Entonces ¿cómo se han estos programas? La respuesta, como podrás adivinar, pasa por los sockets RAW.

¹El adjetivo «raw» (crudo) se utiliza como contraposición a «cooked» (cocinado).

11.1. Acceso privilegiado

Como decía un poco más arriba, el uso de un socket RAW requiere de los privilegios correspondientes, concretamente, se requieren permisos de superusuario. Solo el root o un programa ejecutado con sus privilegios² tendrá permiso para crear sockets RAW.

Si tu interfaz de red es una tarjeta Ethernet, únicamente las tramas broadcast, multicast o que vayan dirigidas específicamente a su dirección MAC serán capturadas y entregadas al subsistema de red. Sin embargo, si pretendes utilizar un socket RAW, es muy probable que te interese recibir todo el tráfico que llegue a la interfaz de red de tu computador, y no sólo el mencionado. Para lograr eso es necesario activar el «modo promiscuo» de la NIC. Eso se puede lograr con ip:

```
# ip link set eth0 promisc on
```

O bien con ifconfig:

```
# ifconfig eth0 promisc
```

De modo análogo se puede saber si una interfaz está en modo promiscuo con:

```
$ ip link show eth0
2: eth0: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc \
    pfifo_fast state UP qlen 1000
    link/ether 00:1b:c2:32:71:32 brd ff:ff:ff:ff:ff:ff
```

Y el comando análogo con ifconfig:

```
$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:1e:c9:34:7e:92
          inet addr:192.168.2.4  Bcast:192.168.2.255  Mask
          :255.255.255.0
          inet6 addr: fe80::21e:c9ff:fe34:7e92/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric
          :1
          RX packets:160791 errors:0 dropped:0 overruns:0 frame:0
          TX packets:121923 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:177101459 (168.8 MiB)  TX bytes:18361567 (17.5
          MiB)
          Memory:fdfe0000-fe000000
```

11.2. Tipos

Lo primero que hay que tener en cuenta es que hay dos tipos básicos de socket raw, y que la decisión de cuál utilizar depende totalmente del objetivo y requisitos de la aplicación que se desea:

Familia AF_PACKET

Los sockets raw de la familia AF_PACKET son los de más bajo nivel y permiten leer y escribir cabeceras de protocolos de cualquier capa.

²bit SUID

Familia AF_INET

Los sockets raw AF_INET delegan al sistema operativo la construcción de las cabeceras de enlace y permiten una manipulación «compartida» de las cabeceras de red.

En las próximas secciones veremos en detalle la utilidad y funcionamiento de ambas familias.

11.3. Sockets AF_PACKET:SOCK_RAW

Son los sockets raw más flexibles y de más bajo nivel, y representan la elección obligada si el objetivo es crear un *sniffer* o algo parecido. Precisamente el siguiente listado es un *sniffer* básico que imprime por consola las tramas Ethernet completas recibidas por cualquier interfaz y de cualquier protocolo.

```

1  #!/usr/bin/python
2  # -*- mode:python; coding:utf-8; tab-width:4 -*-
3
4  import socket
5
6  ETH_P_ALL = 3
7
8  sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
9                      socket.htons(ETH_P_ALL))
10
11 while 1:
12     print "--\n", repr(sock.recvfrom(1600))

```

Y a continuación puedes ver cómo ejecutar este programa, que podríamos llamar con el original nombre de `sniff-all.py`, y su salida:

```

$ sudo ./sniff-all.py
---
('\xff\xff\xff\xff\xff\xff\xa8\x92,\xce\xcd\xb3\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01\
  xa8\x92,\xce\xcd\xb3\xac\x13\xb0\x00\x00\x00\x00\x00\x00\xac\x13\xb0\x01', ('wlan0',
  2054, 1, 1, '\xa8\x92,\xce\xcd\xb3'))
---
('\xff\xff\xff\xff\xff\xff\xd8*~1\x11\xae\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01\xd8*~1\
  x11\xae\xa1C\xfcK\x00\x00\x00\x00\x00\x00\xa1C\xfc\x01',
  ('wlan0', 2054, 1, 1, '\xd8*~1\x11\xae'))
---

```

Haciendo modificaciones mínimas a este programa es posible filtrar el tráfico en dos aspectos:

Tipo de trama

Es decir, el código que identifica protocolo encapsulado como carga útil.³ Para ello se utiliza el tercer campo del constructor de socket.

La interfaz de red

Se logra vinculando el socket a una interfaz de red concreta por medio del método `bind()`.

El uso de ambos «filtros» quedan demostrado en el siguiente programa, llamado `sniff-arp.py`. Sólo muestra mensajes ARP recibidos por la interfaz `eth0`:

³<http://www.iana.org/assignments/ethernet-numbers>

```

1  #!/usr/bin/python
2  # -*- mode:python; coding:utf-8; tab-width:4 -*-
3
4  import socket
5
6  ETH_P_ARP = 0x0806
7
8  sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
9                      socket.htons(ETH_P_ARP))
10 sock.bind(("eth0", ETH_P_ARP))
11
12 while 1:
13     print "--\n", repr(sock.recv(1600))

```

Y el programa en acción:

```

$ sudo ./sniff-arp.py
--
'\xff\xff\xff\xff\xff\xff>m\x84y\x1d\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01>m\x84y\x1d\x
a1C\x11<\x00\x00\x00\x00\x00\x00\xa1C\x11\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00'
--
('xc4\x85\x08B=\xee\x00d@:\xc9@\x08\x00EP\x00T\xfe\x92\x00\x009\x01\x1e-\xd4j\xdd\x15\
xa1C\x11\xd6\x00\x00\x00\x00\x00\x00\x00\x05\x8a\xff\x1cQ\x00\x00\x00\x00z2\x02\x00\x00\x00\x00
\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&\'()
*+,-./01234567', ('wlan0', 2048, 0, 1, '\x00d@:\xc9@'))
--

```

De este modo tan sencillo es posible realizar un *sniffer* completamente a medida de las necesidades concretas. Pero todo esto sólo sirve para leer tramas. Ahora verás como enviar, lo que abre un interesante mundo de posibilidades.

Si quieres identificar el origen del paquete puedes utilizar el método `recvfrom()` en lugar de `recv()`. En ese caso el valor de retorno es una tupla que incluye, entre otras cosas, el nombre de la interfaz (p.ej «eth0») y la dirección MAC origen como una secuencia de bytes.

11.3.1. Construir y enviar tramas

El mismo socket creado en los ejemplos anteriores se puede utilizar para enviar datos. Para sintetizar un paquete, es decir, construir cabeceras de acuerdo a las especificaciones, se utiliza normalmente el módulo `struct` (consulte el capítulo 7) para más información sobre dicho módulo.

El siguiente listado envía una cabecera Ethernet cuyos campos son:

Destino

FF:FF:FF:FF:FF:FF

Origen

00:01:02:03:04:05

Protocolo

0x0806 (ARP)

```

1  #!/usr/bin/python
2  # -*- mode:python; coding:utf-8; tab-width:4 -*-
3
4  import socket, struct
5
6  ETH_P_ARP = 0x0806
7

```

```

8 sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
9                       socket.htons(ETH_P_ARP))
10 sock.bind(("eth0", ETH_P_ARP))
11
12 sock.send(struct.pack("!6s6sh", "\xFF"*6,
13                               "\x00\x01\x02\x03\x04\x05", ETH_P_ARP))

```

Si ejecutas *wireshark* y capturas la trama enviada verás que la marca con un «malformed packet», y con razón: es sólo una cabecera ¡no tiene carga útil! Y eso lógicamente contradice todas las normas del protocolo Ethernet. Resumiendo, este programa no sirve para nada, sólo para que veas que se puede construir y enviar lo que quieras a la red, incluso aunque sea un completo sinsentido.

11.3.2. Implementando un arping

Aunque hay muchas variantes, el programa *arping* envía una petición ARP Request y espera la respuesta correspondiente. En esta sección veremos una implementación que sirve para ilustrar el uso de los sockets raw de la familia AF_PACKET.

Generando mensajes

El programa necesita enviar mensajes ARP Request, que irán encapsulados en tramas Ethernet. Una forma de implementar esta tarea (llamado a veces «sintetizar paquetes») y aprovechar la POO (Programación Orientada a Objetos) es escribir una clase por cada tipo de mensaje. Por tanto, la clase para generar el mensaje ARP Request es algo tan sencillo como esto:

```

1 class Ether:
2     def __init__(self, hwsrc, hwdst):
3         self.hwsrc = hwsrc
4         self.hwdst = hwdst
5         self.payload = None
6
7     def set_payload(self, payload):
8         self.payload = payload
9         payload.frame = self
10
11     def render(self):
12         retval = struct.pack("!6s6sh", self.hwdst, self.hwsrc,
13                               self.payload.proto) + self.payload.
14                               render()
15
16         return retval + (60-len(retval)) * "\x00"

```

Lo único a destacar de la clase *Ether* es el método *render()* que se encarga de generar la representación binaria de los datos que corresponden a la cabecera, concretamente dirección MAC destino, MAC origen, protocolo (el que indique el payload) y a continuación el payload propiamente dicho. Esos datos se empaquetan en binario gracias a *struct.pack()*⁴ indicando que se trata de son 2 secuencias de 6 bytes (6s6s) y un entero de 16 bits (h). La última línea de ese método calcula y concatena el relleno (*padding*) necesario para que la trama alcance el tamaño mínimo necesario de 60 bytes.

La clase para generar mensajes ARP Request es incluso más sencilla:

⁴Ver <http://docs.python.org/library/struct.html#format-characters>

```

1 class ArpRequest:
2     proto = ETH_P_ARP
3
4     def __init__(self, psrc, pdst):
5         self.psrc = socket.inet_aton(psrc)
6         self.pdst = socket.inet_aton(pdst)
7         self.frame = None
8
9     def render(self):
10        return struct.pack("!HHbbH6s4s6s4s", 0x1, 0x0800, 6, 4, 1,
11                               self.frame.hwsrc, self.psrc, "\x00",
12                               self.pdst)

```

Leyendo mensajes

La otra funcionalidad importante del programa es reconocer los mensajes que se obtendrán como respuesta si todo va bien. Se trata de discretizar el valor de cada campo representándolo en un formato adecuado. Esa tarea se suele llamar «disección de paquetes». Como en el caso anterior, una buena forma de hacer esto es delegar el reconocimiento (*parsing*) de cada tipo de mensaje en una clase específica. Hace falta una clase para reconocer tramas Ethernet y otra para reconocer mensajes ARP Reply.

La clase para reconocer tramas Ethernet puede ser algo tan sencillo como esto:

```

1 class EtherDissector:
2     def __init__(self, frame):
3         try:
4             (self.hwdst,
5              self.hwsrc,
6              self.proto) = struct.unpack("!6s6sh", frame[:14])
7         except struct.error:
8             raise DissectionError
9
10        self.payload = frame[14:]

```

El constructor acepta por parámetro una secuencia de bytes, es decir, la trama tal como se lee del socket. Los valores que «desempaqueta» con `struct` y que estarán accesibles como atributos públicos son: dirección MAC destino, MAC origen, protocolo y payload.

El disector del mensaje ARP Reply, llamado `ArpReplyDissector`, es también muy sencillo:

```

1 class ArpReplyDissector:
2     def __init__(self, msg):
3         self.msg = msg
4
5         if struct.unpack("!H", self.msg[6:8])[0] != ARP_REPLY:
6             raise DissectionError
7
8         try:
9             (self.hwsrc,
10              self.psrc,
11              self.hwdst,
12              self.pdst) = struct.unpack("!6s4s6s4s", msg[8:28])
13         except struct.error:
14             raise DissectionError

```

El constructor de la clase acepta una secuencia de byte, que corresponden

con la carga útil de una trama. Como antes, los valores de todos los campos quedan disponibles como atributos de la instancia. Si algún campo o formato no corresponde, el constructor lanza la excepción `DissectionError`.

Programa principal

Solo queda escribir la función principal, la que realmente crea, lee y escribe en el socket. Aparece en el siguiente listado:

```

1  def main(ipsrc, ipdst):
2      print("Request: Who has {0}? Tell {1}".format(ipdst, ipsrc))
3
4      sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
5                          socket.htons(ETH_P_ARP))
6      sock.bind(("eth0", ETH_P_ARP))
7
8      frame = Ether(sock.getsockname()[-1], BROADCAST)
9      frame.set_payload(ArpRequest(ipsrc, ipdst))
10
11     sock.send(frame.render())
12
13     while 1:
14         eth = EtherDissector(sock.recv(2048))
15
16         try:
17             arp_reply = ArpReplyDissector(eth.payload)
18             if arp_reply.hwdst == frame.hwsrc:
19                 print("Reply: {0} is at {1}".format(
20                     ipdst, display_mac(arp_reply.hwsrc)))
21                 break
22         except DissectionError:
23             print(".")

```

La función `main()` acepta las direcciones IP del host origen y destino respectivamente (línea 1). Primero crea y vincula el socket a la interfaz `eth0` (líneas 4-6). A continuación crea una trama Ethernet con destino *broadcast* y origen la MAC de `eth0` (línea 8), y le fija como payload una instancia de `ArpRequest`. El método `send()` envía la trama en su formato binario (línea 11).

El bucle `while` espera la respuesta. En cada iteración se lee y disecciona una trama (línea 14). Si esa trama contiene un mensaje ARP Reply, es decir, si `ArpReplyDissector` no lanza la excepción `DissectionError`, se comprueba además que esa sea la respuesta ARP que se espera y no otra (línea 18). Si es así se imprime la dirección IP del destino y la dirección MAC asociada a esa IP, que es el objetivo final del programa (líneas 19-20).

Puedes encontrar el listado completo en el fichero `src/raw/arping.py`.

11.4. Sockets AF_INET:SOCK_RAW

A pesar de la flexibilidad y potencia de los sockets `AF_PACKET`, no siempre son la mejor elección ya que el programador debe parsear y generar el contenido de todas las cabeceras. Eso puede ser bastante latoso cuando entra en juego el cálculo de checksums u otros datos no tan directos.

Los sockets `AF_INET:SOCK_RAW` pueden ser una buena alternativa si solo te interesa «tocar» las cabeceras de transporte, dejando al sistema operativo todo el trabajo relacionado con las de enlace, y opcionalmente las de red.

11.4.1. Generando mensajes

El siguiente programa imprime por consola todos los paquetes IP que contengan un segmento UDP:

```

1  #!/usr/bin/python
2  # -*- mode:python; coding:utf-8; tab-width:4 -*-
3
4  import socket
5
6  sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
7                      socket.getprotobyname("udp"))
8  while 1:
9      print '--\n', repr(sock.recv(1600))

```

La función `getprotobyname()` devuelve el número de protocolo⁵ a partir de su nombre (línea 4). Es interesante destacar que el resultado del método `recv()` es el paquete IP completo, incluyendo cabecera (línea 7).

Como en el caso de los socket `AF_PACKET` puedes identificar el origen del paquete (su dirección IP) sin tener que parsear la cabecera IP. Para lograrlo utiliza el método `recvfrom()` en lugar de `recv()`. En ese caso el valor de retorno es una tupla con la forma (datos, dirección), teniendo en cuenta que la dirección es su vez una tupla (IP, 0).

11.4.2. Enviando

Para enviar datos sobre este tipo de socket debes utilizar el método `sendto` indicando la dirección destino. El siguiente programa envía un segmento UDP válido que contiene el texto «hello Inet»:

```

1  #!/usr/bin/python
2  # -*- mode:python; coding:utf-8; tab-width:4 -*-
3
4  import socket, struct
5
6  sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
7                      socket.getprotobyname("udp"))
8
9  payload = "hello Inet"
10 udp_pkt = struct.pack('!4h', 0, 2000, 8+len(payload), 0) + payload
11 sock.sendto(udp_pkt, ('127.0.0.1', 0))

```

Puedes comprobar su funcionamiento ejecutando un servidor UDP en el puerto 2000 gracias a `ncat`. En un terminal ejecuta:

```
$ ncat -l -p 2000
```

Y en otro terminal, pero en la misma máquina, ejecuta:

```
$ ./udp-send.py
```

Si todo ha ido bien, en el primer terminal debería aparecer el texto «Hello Inet».

⁵<http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml>

IP_HDRINCL

Como has podido comprobar en el ejemplo anterior, es posible enviar un segmento sin tener que construir la cabecera IP, únicamente la UDP. Sin embargo, puede haber ocasiones en las que el programador necesite «tocar» también la cabecera IP. Eso se consigue con la opción `IP_HDRINCL`.

La ventaja respecto al socket `AF_PACKET` es doble: no hay que molestarse con la cabecera de enlace, y además el SO puede rellenar por ti algunos de los campos menos cómodos si así lo quieres. Esos campos son:

- El checksum.
- La dirección IP origen (si el programador pone ceros).
- El identificador del mensajes (si el programador pone ceros).
- El campo de longitud total.

Esta opción, como la gran mayoría debe fijarse explícitamente después de creado el socket por medio del método `setsockopt()`, tal como se indica:

```
1 sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
```

Esto resulta muy útil cuando quieres utilizar el socket para enviar distintos protocolos, y por tanto necesitas tener acceso al campo *proto*. Para poder hacer eso necesitas crear un socket de un *protocolo* especial: `IPPROTO_RAW`. Ese tipo de socket se crea con:

```
1 sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
2                       socket.IPPROTO_RAW)
```

Aunque tiene un pequeño inconveniente. No se puede leer de este tipo de socket, tendrás que crear un socket adicional para poder leer los mensajes entrantes.

11.5. Ejercicios

A continuación se propone una lista de pequeñas herramientas de captura que pueden ser utilizadas para análisis, monitorización y validación de tráfico, y detección de anomalías. Los programas resultantes deberían funcionar correctamente al menos en una plataforma GNU/Linux.

E 11.01 Para una red Ethernet, escriba un programa que cuente el número de tramas que aparecen en el enlace con la granularidad temporal indicada como parámetro (en minutos) y lo imprima en consola tal como se indica. La primera columna es el tiempo en segundos del inicio del intervalo y la segunda en el número de tramas que han aparecido en la red en dicho intervalo:

```
$ ./frame-count.py 2
Slot size: 120s
0: 12
120: 14
240: 150
Capture time: 281.2s
```

E 11.02 Para una red Ethernet, escriba un programa que cuente el número de paquetes de cada protocolo (niveles red y transporte) durante el tiempo que esté en ejecución. Ejemplo de uso:

```
$ ./package-type-count.py
Capture time: 123.4s
ARP: 50
IP: 500
UDP: 80
TCP: 420
```

E 11.03 Para una red Ethernet, escriba un programa que calcule una estadística del tamaño de las tramas (en bytes) que aparecen en la red, con la granularidad indicada en bytes. Ejemplo de uso:

```
$ ./frame-sizes.py 300
Capture time: 120.2s
46 - 300: 340
301 - 600: 62
601 - 900: 10
901 - 1200: 140
1201 - 1500: 970
```

E 11.04 Para una red Ethernet, escriba un programa que calcule una estadística de la utilización y lo exprese como porcentaje de la capacidad del enlace. Debe realizarse con la granularidad indicada (en minutos). Ejemplo de uso:

```
$ ./utilization.py 2
Slot size: 120s
0: 22%
120: 35%
240: 2%
Capture time: 341.2s
```

E 11.05 Para una red Ethernet, escriba un programa que calcule la utilización (tamaño total de tramas Ethernet) y el throughput (considerando payload de segmentos UDP y TCP). Ejemplo de uso:

```
$ ./bandwidth.py
Capture time: 380.6s
Utilization: 1280 Kbps
Throughput: 992 Kbps
```

E 11.06 Escriba un programa que calcule el *tiempo medio de inactividad* (Average Idle Time) de un enlace durante el tiempo de ejecución del programa.

E 11.07 Escriba un programa que mida la utilización que un host hace de un enlace (dada su dirección MAC) durante el tiempo de ejecución del programa.

E 11.08 Escriba un programa que capture tramas Ethernet de una interfaz indicada como argumento e imprima por pantalla las direcciones origen y destino, el campo tipo y tamaño del payload de cada trama que reciba.

E 11.09 Escriba un programa que capture paquetes IP de una interfaz de red

indicada como argumento e imprima en pantalla el valor de los campos más importantes de la cabecera en un formato adecuado.

Filtrado de paquetes

David Villa

12.1. Introducción

Lo más normal para un usuario medio es utilizar iptables para definir reglas de firewall en su router. Para simplificar la explicación de los ejemplos que aparecen en la receta voy a suponer una configuración concreta. Es importante tenerla en cuenta cuando tengas que hacer los cambios pertinentes para aplicarla a tu configuración concreta. Es ésta:

Una conexión a Internet (eth0)

Debe ser algún tipo de conexión a Internet, que puede ser RTC (Red Telefónica Conmutada), ADSL, cable-modem, etc. Supondremos para esta receta que usas un modem ADSL con conexión Ethernet.

Una conexión a tu red local (eth1)

Será una tarjeta de red Ethernet 10/100. Esto permitirá que puedas enchufarle un conmutador y conectar múltiples computadores si quieres. Esta interfaz debes configurarla de forma estática, con una dirección IP privada, por ejemplo 192.168.0.1.

Esta configuración se corresponde con nuestra otra receta sobre compartir la conexión, de modo que es más sencillo seguir aquí desde aquella.

12.2. Tablas y reglas

Hay 3 tablas (o cadenas), que indican qué tipo de operación puede hacer el router con los paquetes.

FILTER En esta tabla hay reglas que dicen qué hacer con los paquetes, pero sin modificarlos. Esta es la tabla por defecto, si no se indica otra.

Se puede definir 3 tipos de reglas:

INPUT Para paquetes cuyo destino es un socket de la propia máquina.

OUTPUT Para paquetes generados por la propia máquina.

FORWARD Para paquetes que llegan a la máquina, pero cuyo destino es una máquina distinta.

NAT Implica a los paquetes que requieren crear nuevas conexiones. Normalmente implica algún tipo de traducción de dirección, ya sea dirección o puerto (modifican el paquete). También se puede definir 3 tipos de reglas:

PREROUTING El paquete se modifica en cuanto llega a la máquina.

POSTROUTING El paquete se modifica después de decidir su destino (una vez rutado).

OUTPUT El paquete se ha generado en la propia máquina y se modifica antes de decidir su destino.

MANGLE Implica modificaciones más sofisticadas del paquete, que van más allá de su dirección. Excede el alcance de esta receta.

Además de éstas que vienen de serie, el usuario puede crear otras.

12.3. Políticas

Existen dos políticas básicas para administrar un firewall.

Restictiva (DROP) Todo lo que no está explícitamente permitido, está prohibido.

Permisiva (ACCEPT) Todo lo que no está explícitamente prohibido, está permitido. Es la política por defecto en iptables.

La política se determina para cada tipo de regla por separado. Por ejemplo, para establecer una política restrictiva para FORWARD sería:

```
# iptables -P FORWARD DROP
```

Evidentemente, iptables tienes muchas opciones que dan mucho juego, mucho más de lo que cualquier mortal pueda imaginar, no es tan complejo y sofisticado como CISCO IOS pero tampoco le falta tanto. Por eso vamos a acotar un poco el asunto y vamos a hablar sólo de 3 «esquemas» que se suelen usar para configuraciones sencillas:

- Utilizar política por defecto (permisiva) y denegar explícitamente cada servicio que NO se desea.
- Utilizar política restrictiva y aceptar explícitamente cada servicio que se desea.
- Utilizar política por defecto, indicar qué servicios se desean explícitamente y **por último** denegar todo lo demás. Este esquema, a pesar de usar una política permisiva, prohíbe todo lo que no esté explícitamente aceptado.

Lógicamente los esquemas 1 y 3 son iguales salvo porque en el 3 no tiene sentido definir reglas de denegación específicas.

Es importante tener en cuenta que las reglas se aplican secuencialmente. Una vez que el paquete coincide con una regla de la lista ya no se le aplica ninguna más. Eso significa que si se prohíbe un servicio, después el inútil permitir un subconjunto de lo prohibido.

12.4. Comandos básicos

Este es un resumen de comandos habituales simplificados.

12.4.1. Ver la configuración

```
# iptables -L [-t tabla] [-v]
```

12.4.2. Borrar todas las reglas de una tabla

```
# iptables -F [-t tabla]
```

12.5. Configuración de un router (con esquema 3)

Esta es una configuración sencilla pero completa de un router, utilizando el esquema 3.

Enrutar tráfico de la red local a Internet aplicando NAT. Recuerda activar también el forwarding.

```
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Aceptar todo el tráfico ICMP.

```
# iptables -A INPUT -p ICMP -j ACCEPT
```

Aceptar todo el tráfico de conexiones ya establecidas (ESTABLISHED) o relacionado con otras conexiones (RELATED).

```
# iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Aceptar cualquier conexión que proceda de la red interna:

```
# iptables -A INPUT -p tcp -i eth1 -j ACCEPT
```

Aceptar conexiones web (80) desde cualquier origen:

```
# iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

Aceptar todo el tráfico desde el loopback:

```
# iptables -A INPUT -i lo -j ACCEPT
```

Descartar el resto:

```
# iptables -A INPUT -j DROP
```

Si quieres permitir algo más debes insertar la nueva regla antes del DROP. Por ejemplo: Aceptar conexiones HTTPS desde cualquier sitio:

```
# iptables *-I INPUT 7* -p tcp --dport https -j ACCEPT
```

Por supuesto también puedes editar directamente el fichero en el que guardes la configuración de iptables (ver guardar configuración). El contenido de dicho

fichero para la configuración fijada con estos comandos sería (no se muestran las tablas vacías):

```

1  *filter
2  :INPUT ACCEPT [696743:123302283]
3  :FORWARD ACCEPT [591745:477869546]
4  :OUTPUT ACCEPT [801266:279702677]
5  -A INPUT -p ICMP -j ACCEPT
6  -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
7  -A INPUT -i eth1 -j ACCEPT
8  -A INPUT -p tcp --dport 80 -j ACCEPT
9  -A INPUT -i lo -j ACCEPT
10 -A INPUT -p tcp --dport https -j ACCEPT
11 -A INPUT -j DROP
12 COMMIT
13 *nat
14 :PREROUTING ACCEPT [64615:3723067]
15 :POSTROUTING ACCEPT [101:14282]
16 :OUTPUT ACCEPT [10260:530443]
17 -A POSTROUTING -o eth0 -j MASQUERADE
18 COMMIT

```

12.5.1. Bloquear un puerto (en esquema 1)

Esto es útil si el router da un servicio a la red local que no quieres que se vea desde Internet, por ejemplo, el servidor FTP.

```
# iptables -A INPUT -i eth0 -p tcp --dport 20:21 -j DROP
```

12.5.2. Redireccionar un puerto del router hacia otro host (interno o externo)

Esto se conoce comúnmente como Port forwarding. Por ejemplo, al conectar al puerto 80 del router, se accederá realmente al servidor web de una máquina de la red interna:

```
# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j DNAT
--to-destination 192.168.0.15:80
```

12.5.3. Guardar la configuración de iptables

Para guardar las reglas activas (las que has ido definiendo hasta ahora) ejecuta:

```
Router:~# iptables-save > /etc/iptables.up.rules
```

Y ahora hay que hacer que esa configuración se cargue automáticamente al levantar la interfaz de red principal (la externa). Para ello, edita de nuevo el fichero `/etc/network/interfaces` para que la entrada de `eth0` quede así:

```

1  auto eth0
2  iface eth0 inet dhcp
3      pre-up iptables-restore < /etc/iptables.up.rules

```

Escaneo de servidores y servicios

David Villa

El primer paso al realizar una auditoría o examen externo es identificar los computadores activos en la red, cuáles son los servicios que ofrecen y las versiones concretas del sistema operativo y aplicaciones que ejecutan.

Esta información es crucial para localizar los riesgos de seguridad y descubrir puntos débiles, ya sea por errores de configuración o administración, o por errores explotables.

13.1. Descubrimiento de hosts

El descubrimiento de hosts trata de determinar qué direcciones IP de la red objetivo están asociadas a computadores activos y por tanto susceptibles de presentar riesgos de seguridad. También es de gran valor cualquier información que permita averiguar cuál es la topología de la red y la existencia de encaminadores, cortafuegos o proxies intermedios.

Descubrir hosts en la red local (LAN) es, en principio, más sencillo pues se dispone de más información. El dispositivo más importante de la LAN es el encaminador local o «pasarela de enlace». Si DHCP está disponible, es el propio servidor el que ofrece esa información dado que es esencial para que los computadores de la LAN puedan comunicarse con el exterior.

Para obtener esa información desde el computador del auditor basta utilizar el comando `ip`:

```
$ ip route show
default via 161.67.0.1 dev wlan0 proto static metric 1024
161.67.0.0/24 dev wlan0 proto kernel scope link src 161.67.0.28
169.254.0.0/16 dev wlan0 scope link metric 1000
```

La primera línea corresponde con la ruta por defecto y por tanto indica la IP del encaminador local. Pero ¿cómo saber qué otros computadores están activos en esta misma LAN? Básicamente existen dos estrategias:

- El escaneo pasivo consiste en analizar el tráfico que los otros computadores envían a la LAN. La información más valiosa se puede obtener gracias al protocolo ARP. Lo usual es que todos los computadores envíen regularmente peticiones ARP para averiguar las direcciones físicas de los computadores con los que se comunica.
- El escaneo activo consiste en tratar de enviar tráfico a los computadores vecinos y comprobar que, directa o indirectamente, existe una respuesta.

El escaneo **pasivo** obviamente se puede realizar con un sniffer de propósito general:

```
$ tshark -a duration:7 -n -f arp -i wlan0
Capturing on 'wlan0'
 1  0.000000 2c:d0:5a:f2:46:11 -> bcast ARP 56 Gratuitous ARP for 172.19.148.201 (Req)
 2  0.614331 30:39:26:06:f4:8a -> bcast ARP 56 Gratuitous ARP for 172.19.209.48 (Req)
 3  0.615345 b4:52:7d:d6:32:cd -> bcast ARP 56 Gratuitous ARP for 172.19.182.10 (Req)
 4  1.433630 30:a8:db:9a:04:0a -> bcast ARP 56 Gratuitous ARP for 172.19.149.149 (Req)
 5  2.457601 30:a8:db:9a:04:0a -> bcast ARP 56 Gratuitous ARP for 172.19.149.149 (Req)
 6  2.665341 b4:52:7d:d6:32:cd -> bcast ARP 56 Gratuitous ARP for 172.19.182.10 (Req)
 7  5.836938 fc:25:3f:93:b8:de -> bcast ARP 56 Gratuitous ARP for 172.19.210.79 (Req)
7 packets captured
```

No hay ninguna garantía de que todas las peticiones correspondan con máquinas activas porque puede haber un computador tratando de resolver una IP no asignada. Sin embargo, las peticiones «gratuitas» sí tienen una alta fiabilidad, supiendo por supuesto que no hay ningún programa malicioso generando mensajes falsos. En cualquier caso, este tipo de información siempre se debe considerar como indicios y no como evidencias.

E 13.10 Las técnicas descritas en este capítulo suelen aplicarse a un rango de direcciones (expresado en notación CIDR (Classless Interdomain Routing), por ejemplo, 161.67.0.0/24). Escribe un programa que genere todas las direcciones posibles en un rango expresado de este modo.

E 13.11 Escribe un programa que utilice sockets *raw* para listar las direcciones IP que aparecen en las peticiones ARP capturadas.

El escaneo **activo** más sencillo se puede conseguir gracias a ICMP ping. Basta con generar todas las direcciones IP posibles en el rango de la LAN (161.67.0.1 - 161.67.0.254 en nuestro caso) y comprobar si hay respuesta. El siguiente comando aplica esa técnica:

```
$ nmap -sP 161.67.0.0/24 | head -n 16
Starting Nmap 6.47 ( http://nmap.org ) at 2014-09-30 20:20 CEST
Nmap scan report for 161.67.0.1
Host is up (0.0024s latency).
Nmap scan report for android-e210d576490f503b.uclm.es (161.67.0.24)
Host is up (0.089s latency).
Nmap scan report for amy.uclm.es (161.67.0.28)
Host is up (0.000034s latency).
Nmap scan report for android-e90184c47f5cc9a.uclm.es (161.67.0.29)
Host is up (0.14s latency).
Nmap scan report for iphoneduelangel.uclm.es (161.67.0.33)
Host is up (0.033s latency).
Nmap scan report for iphone-de-ana.uclm.es (161.67.0.44)
Host is up (0.16s latency).
Nmap scan report for android-4a1c6871cd9134a9.uclm.es (161.67.0.56)
Host is up (0.026s latency).
```

E 13.12 Escribe un script C-Shell que utilice el comando ping para listar las direcciones IP de todos los hosts activos en el rango indicado.

Sin embargo, es sencillo configurar un computador para ignorar las peticiones ICMP. Pero aún así, en el comando anterior nmap ha necesitado resolver las direcciones físicas de todos los computadores a los que ha enviado la petición ping, de modo que todos los que estén activos tienen que haber contestado a la petición ARP. Dicho de otro modo, podemos concluir que las direcciones conte-

nidas en la tabla ARP son computadores activos aunque no hayan contestado al mensaje ping.

```
$ /usr/sbin/arp -n | head -n 10
Address      HWtype  HWaddress  Flags Mask  Iface
161.67.0.31  ether   24:ec:99:71:93:f1  C          wlan0
161.67.0.154 ether   74:e5:43:ad:15:34  C          wlan0
161.67.0.100 ether   dc:f1:10:52:00:8c  C          wlan0
161.67.0.5   ether   1c:af:05:f6:b6:5d  C          wlan0
161.67.0.33  ether   04:f7:e4:f3:b5:a9  C          wlan0
161.67.0.189 ether   e0:c9:7a:73:1e:5f  C          wlan0
161.67.0.56  ether   60:be:b5:3a:1c:a0  C          wlan0
161.67.0.124 ether   7c:c5:37:da:d6:23  C          wlan0
161.67.0.29  ether   f0:27:65:6a:33:a2  C          wlan0
```

Sin embargo, todo lo anterior solo es aplicable cuando se pretenden descubrir vecinos activos (computadores en la misma LAN). Si se está analizando una red diferente ARP no aplica y si los computadores objetivo ignoran el mensaje ICMP Echo es necesario aplicar técnicas alternativas:

TCP SYN

Consiste en enviar un segmento TCP vacío que lleva activo el flag SYN. Si el computador objetivo contesta, ya sea con SYN/ACK o RST es que está activo. Si no hay ningún tipo de respuesta se asume que el computador no está activo o simplemente la IP no está asignada a ningún host.

TCP connect

La técnica TCP SYN requiere privilegios especiales ya que implica sintetizar un paquete mediante un socket *raw*. Cuando el usuario que lanza el escaneo no tiene estos privilegios se puede recurrir a un intento de conexión con un socket TCP. Aunque el resultado no es tan fiable puede funcionar bien en la mayoría de los casos.

TCP ACK

De forma similar a TCP SYN, el computador del auditor envía un segmento TCP con el flag ACK activado. Si el computador objetivo está activo se espera que devuelva un mensaje RST.

IP proto

Se trata de enviar un paquete IP con diferentes identificadores de protocolo. Ante cualquier respuesta, incluyendo un error de «protocolo inalcanzable», se considera que el computador objetivo está vivo.

UDP ping

Construye y envía un segmento UDP a un puerto del computador objetivo. Si el puerto está cerrado lo más probable es obtener como respuesta un mensaje de error ICMP, generalmente «puerto inalcanzable». La ausencia de respuesta se puede interpretar como computador activo.

ICMP queries

Utiliza tipos de mensajes ICMP alternativos al ping, tales como petición de marca de tiempo o máscara de red.

El motivo para utilizar segmentos TCP con distintos flags es el de maximizar la posibilidad de que estos segmentos efectivamente lleguen al computador objetivo a pesar de que existan cortafuegos intermedios.

E 13.13 El programa *nmap* proporciona soporte para todas las técnicas de

descubrimiento anteriormente descritas. Determina qué parámetros de llamada corresponden a cada una de ellas y demuestra con una captura de tráfico que efectivamente es así.

13.2. Escaneo de puertos

El escaneo de puertos (o servicios) trata de determinar el estado de cada puerto de un computador, para cada uno de los protocolos de transporte. Se consideran los siguientes estados:

abierto

Un puerto abierto es aquel en el que hay un proceso en ejecución vinculado y aceptando mensajes entrantes (conexiones en el caso de TCP). Un puerto abierto es la forma en la que un computador ofrece servicios a la red, pero también puede implicar la existencia de un riesgo de seguridad susceptible de ser analizado.

cerrado

Cualquier puerto que no tiene una proceso asociado se considera cerrado si es conforme a la especificación del protocolo correspondiente. Por ejemplo, si es un puerto TCP «libre» debería devolver un segmento con el flag RST cuando un computador cliente trata de abrir una nueva conexión. Un puerto cerrado ofrece una respuesta.

filtrado

Si no se obtiene ningún tipo de respuesta al tratar de acceder a un puerto remoto es indicativo de que un dispositivo intermedio (o quizá el propio computador objetivo) está descartando el tráfico. Es decir, se trata de algún tipo de filtro impuesto por un contafuegos. También se considera que un puerto está filtrado si se obtiene un mensaje de error ICMP del tipo «destino inalcanzable» procedente de un encaminador intermedio, probablemente por la misma razón. Nótese que las reglas de filtrado del cortafuegos pueden aplicarse sobre orígenes específicos, es decir, el puerto puede aparecer filtrado para el computador del auditor, pero podría estar accesible y abierto para computador de otras redes.

El siguiente listado muestra una salida típica de nmap al tratar de descubrir el estado de los puertos de un servidor público:

```
$ sudo nmap -sS www.google.com
Starting Nmap 6.47 ( http://nmap.org ) at 2014-10-01 17:28 CEST
Nmap scan report for www.google.com (173.194.40.115)
Host is up (0.024s latency).
rDNS record for 173.194.40.115: par10s09-in-f19.1e100.net
Not shown: 998 filtered ports
PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https
```

Cuando se utiliza un socket BSD para acceder a un puerto remoto (con la llamada al sistema `connect()` en el caso de TCP) solo se puede determinar si el puerto está abierto (se estableció la conexión) o hubo un problema, pero resulta complicado evaluar qué problema es y qué implica en relación al estado del puerto. Con puertos UDP es más complicado aún. Las técnicas de escaneo de puertos se basan en su mayoría en la fabricación de mensajes con características

muy concretas que ayudan a determinar el estado de los puertos de forma más precisa. Es por ese motivo que los escáneres de puertos utilizan sockets «raw», algo que requiere privilegios de administrador.

A continuación se resumen algunas de las técnicas para escaneo más comunes.

TCP SYN

Es la más común y normalmente muy efectiva. Es la misma técnica utilizada para la detección de computadores activos y permite diferenciar entre un puerto abierto (devuelve un segmento SYN+ACK) un puerto cerrado (devuelve RST) o filtrado en otro caso.

TCP connect

Equivalente también a la técnica utilizada en la detección de computadores y con las mismas premisas que la técnica TCP SYN, solo puede la puede ejecutar un usuario sin privilegios.

TCP ACK

Cuando se activa el flag ACK no se puede determinar si el puerto está abierto o cerrado (la respuesta será RST en ambos casos) pero sí se puede determinar si está filtrado o no. Pero esta técnica resulta interesante en combinación con TCP SYN porque puede determinar si existe hay un cortafuegos con estado operando en la ruta, ya que lo habitual es que las reglas de filtrado permitan segmentos SYN, pero descarten segmentos ACK.

TCP Window

Utiliza un detalle de implementación de algunas pilas TCP. Los segmentos procedentes de puertos cerrados indican un tamaño de ventana igual a cero, mientras que los abiertos indican un valor diferente.

TCP flags

La especificación de TCP indica que el SO debe enviar un mensaje RST si un puerto cerrado recibe un segmento que no contiene ninguno de los flags SYN, RST y ACK. Sin embargo hay otros tres flags (FIN, PSF y URG) que pueden influir en este comportamiento. Cambiando el valor de dichos flags, el escáner puede tratar de obtener una respuesta.

TCP Mainmon

Algunas implementaciones de TCP descartan un segmento FIN+ACK en puertos abiertos mientras que contestan con un segmento RST para los puertos cerrados.

Idle Se trata de una técnica que utiliza un computador adicional (denominado *zombie*) para realizar el escaneo. Se basa en el hecho de que el identificador utilizado para identificador de los datagramas IP (que sirve para reconocer todos los fragmentos del mismo datagrama) crece de forma secuencial cada vez que el computador envía uno nuevo. La técnica consiste en averiguar el identificador actual del *zombie*, después se envía un segmento al computador objetivo indicado la IP del *zombie* como dirección origen y por último volver a solicitar el identificador del *zombie*. Si ambos números difieren en más de uno significa que el puerto consultado está abierto. La justificación se basa en el detalle de que ante un puerto abierto, el computador objetivo habrá respondido con

un segmento SYN+ACK hacia el *zombie* y éste a su vez le habrá enviado un segmento RST. Sin embargo, si el puerto está cerrado, el computador objetivo habrá enviado un segmento RST, y en este caso el *zombie* lo habrá descartado sin generar nada.

- E 13.14** Escribe un programa que implemente la técnica TCP SYN y que acepte una dirección IP y un rango de puertos a escanear.
- E 13.15** Escribe un programa que verifique el funcionamiento de la técnica de escaneo *Idle*.
- E 13.16** Determina qué opción de *nmap* permite utilizar cada una de las técnicas descritas y ejecuta un ejemplo funcional.

13.3. Identificación de servicios

13.4. Identificación del Sistema Operativo

ANEXOS

Netcat

David Villa

Netcat es una de las herramientas más potentes y flexibles que existen en el campo de la programación, depuración, análisis y manipulación de redes y servicios TCP/IP. Es un recurso imprescindible para profesionales relacionados con las comunicaciones en Internet, expertos en seguridad de redes o hackers. Este capítulo incluye varios ejemplos de uso de *GNU netcat*.

Aunque *netcat* puede hacer muchas cosas, su función principal es en realidad muy simple:

- Crea un socket «conectado» al destino indicado si es cliente, o en el puerto indicado, si es servidor.
- Una vez conectado, envía por el socket todo lo que lea de su entrada estándar y envía a su salida estándar todo lo que reciba por el socket. Todo ello sin modificar el contenido de los mensajes que maneja.

Resumiendo, es un programa que puede actuar como servidor o cliente «para cualquier cosa». Algo tan simple resulta ser extraordinariamente potente y flexible como vas a ver e continuación. Por simplicidad se utilizan conexiones locales aunque, por supuesto, se pueden utilizar entre máquinas diferentes.

A.1. Sintaxis

```
nc [-options] hostname port[s] [ports]
nc -l -p port [-options] [hostname] [port]
```

Parámetros básicos:

- l** modo 'listen', queda a la espera de conexiones entrantes.
- p** puerto local.
- u** crear un socket UDP.
- e** ejecuta el comando dado después de conectar.
- c** ejecuta órdenes de shell (equivalente `/bin/sh -c [comando]` después de conectar.

A.2. Ejemplos

A.2.1. Un chat para dos

Servidor

```
$ nc -l -p 2000
```

Cliente

```
$ nc localhost 2000
```

A.2.2. Transferencia de ficheros

La instancia de nc que *escucha* recibe el fichero. El receptor ejecuta:

```
$ nc -l -p 2000 > fichero.recibido
```

Y el emisor:

```
$ nc localhost 2000 < fichero
```

A.2.3. Servidor de **echo**

Ponemos un servidor que ejecuta cat de modo que devolverá todo lo que se le envíe.

```
$ nc -l -p 2000 -e /bin/cat
```

Y en otra consola:

```
$ nc localhost 2000
hola
hola
...
```

A.2.4. Servidor de **daytime**

Exactamente lo mismo que el ejemplo anterior pero ejecutando date en lugar de cat.

```
$ nc -l -p 2000 -e /bin/date
```

en otra consola:

```
$ nc localhost 2000
lun feb 23 21:26:48 CET 2004
```

A.2.5. Shell remota estilo **telnet**

Servidor:

```
$ nc -l -p 2000 -e /bin/bash
```

Cliente:

```
$ nc localhost 2000
```

A.2.6. Telnet inverso

En esta ocasión es el cliente quien pone el terminal remoto
Servidor

```
$ nc -l -p 2000
```

Cliente

```
$ nc server.example.org 2000 -e /bin/bash
```

A.2.7. Cliente de IRC

```
$ *nc irc.freenode.net 6666*
NOTICE AUTH :*** Looking up your hostname...
NOTICE AUTH :*** Found your hostname, welcome back
NOTICE AUTH :*** Checking ident
NOTICE AUTH :*** No identd (auth) response
*NICK nadie*
*USER nadie nadie nadie :nadie*
:kubrick.freenode.net 001 nadie :Welcome to the freenode IRC
      Network nadie
:kubrick.freenode.net 002 nadie :Your host is kubrick.freenode.net [
      kubrick.freenode.net/6666], running version hyperion-1.0.2b
[...]
```

y a partir de ahí puedes introducir cualquier comando de IRC:

- LIST
- JOIN #canal
- PART #canal
- PRIVMSG #canal :mensaje
- WHO #canal
- QUIT

A.2.8. Cliente de correo SMTP

Podemos usar netcat para enviar correo electrónico por medio de un servidor SMTP, utilizando el protocolo directamente:

```
~$ nc mail.servidor.com
220 mail.servidor.com ESMTP Postfix
HELO yo
250 mail.servidor.com
MAIL FROM:guillermi@microchhof.com
250 Ok
RCPT TO:manolo@cocaloca.es
250 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
Aviso: su licencia ha caducado. Me deben una pasta.
.
250 Ok: queued as D44314A607
QUIT
221 Bye
```

A.2.9. HTTP

Es sencillo conseguir un cliente y un servidor HTTP rudimentarios.

Servidor

```
$ nc -l -p http -c "cat index.html"
```

Al cual podemos conectar con cualquier navegador HTTP, como por ejemplo firefox.

Cliente

```
$ echo "GET /" | nc www.google.com 80 > index.html
```

A.2.10. Streaming de audio

Un sencillo ejemplo para hacer *streaming* de un fichero .mp3:

Servidor

```
$ nc -l -p 2000 < fichero.mp3
```

y para servir todos los .mp3 de un directorio:

```
$ cat *.mp3 | nc -l -p 2000
```

Cliente

```
$ nc server.example.org 2000 | madplay -
```

A.2.11. Streaming de video

Servidor

```
$ nc -l -p 2000 < pelicula.avi
```

Cliente

```
$ nc server.example.org 2000 | mplayer -
```

A.2.12. Proxy

Sirva para redirigir una conexión a otro puerto u otra máquina:

```
$ nc -l -p 2000 -c "nc example.org 22"
```

El trafico recibido en el puerto 2000 de esta máquina se redirige a la máquina example.org:22. Permite incluso que la conexión entrante sea UDP pero la redirección sea TCP o viceversa!

A.2.13. Clonar un disco a través de la red

Esto se debe usar con muchísima precaución. ¡Si no estás 100 % seguro, no lo hagas! No digas que no te avisé.

Es este ejemplo voy a copiar un pen drive USB que está conectado al servidor a un fichero en el cliente y después lo voy a montar para acceder al contenido.

Servidor

```
$ dd if=/dev/sda1 | nc -l -p 2000
```

Cliente

```
$ nc server.example.org 2000 | dd of=pendrive.dump
$ mount pendrive.dump -r -t vfat -o loop /mnt/usb
```

A.2.14. Ratón remoto

Es decir, usar el ratón conectado a una máquina para usar el entorno gráfico de otra. El ejemplo está pensado para Xorg.

Servidor

```
# nc -l -p 2000 < /dev/input/mice
```

Cliente

Editar el fichero `@/etc/X11/xorg.conf@` y modificar la configuración del ratón para que queda así:

```
1 Section "InputDevice"
2     Driver      "mouse"
3     ...
4     Option      "Device"      "/tmp/fakemouse"
5     ....
6 EndSection
```

```
$ mkfifo /tmp/fakemouse
$ nc server.example.org 2000 > /tmp/fakemouse
# /etc/init.d/gdm restart
```

A.2.15. Medir el ancho de banda

Servidor

```
$ nc -l -p 2000 | pv > /dev/null
```

Cliente

```
$ nc server.example.org 2000 < /dev/zero
```

A.2.16. Imprimir un documento en formato PostScript

Funciona en impresoras que soporten el estándar AppSocket/JetDirect, que son la mayoría de las que se conectan por Ethernet.

```
$ cat fichero.ps | nc -q 1 nombre.o.ip.de.la.impresora 9100
```

A.2.17. Ver «La Guerra de las Galaxias»

```
$ nc towel.blinkenlights.nl 23
```

A.3. Otros «netcat»s

cryptcat

<http://farm9.org/Cryptcat/> - netcat cifrado.

socat

<http://www.dest-unreach.org/socat/> - Cuando netcat te queda corto.

socket

<http://packages.debian.org/sid/socket>

ncat <http://nmap.org/ncat/>

Referencias

- [deb] Debian GNU/Linux. url: <http://www.debian.org/>.
- [DVGD96] C. Davis, P. Vixie, T. Goodwin, y I. Dickinson. A Means for Expressing Location Information in the Domain Name System. RFC 1876 (Experimental), Enero 1996. url: <http://www.ietf.org/rfc/rfc1876.txt>.
- [Fen97] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236 (Proposed Standard), Noviembre 1997. Obsoleted by RFC 3376. url: <http://www.ietf.org/rfc/rfc2236.txt>.
- [IAN02] IANA. Special-Use IPv4 Addresses. RFC 3330 (Informational), Septiembre 2002. url: <http://www.ietf.org/rfc/rfc3330.txt>.
- [Kuz] Alexey N. Kuznetsov. IP Command Reference.
- [Lib90] D. Libes. Choosing a name for your computer. RFC 1178 (Informational), Agosto 1990. url: <http://www.ietf.org/rfc/rfc1178.txt>.
- [Pos81a] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), Septiembre 1981. Updated by RFCs 950, 4884. url: <http://www.ietf.org/rfc/rfc792.txt>.
- [Pos81b] J. Postel. Internet Protocol. RFC 791 (Standard), Septiembre 1981. Updated by RFC 1349. url: <http://www.ietf.org/rfc/rfc791.txt>.
- [PR83] J. Postel y J. K. Reynolds. Telnet Protocol Specification. RFC 854 (Standard), Mayo 1983. Updated by RFC 5198. url: <http://www.ietf.org/rfc/rfc854.txt>.
- [RP94] J. Reynolds y J. Postel. Assigned Numbers. RFC 1700 (Historic), Octubre 1994. Obsoleted by RFC 3232. url: <http://www.ietf.org/rfc/rfc1700.txt>.