

Internet, un enfoque práctico



Universidad de Castilla-La Mancha

Escuela Superior de Informática
de Ciudad Real

29 de junio de 2021

Escuela Superior de Informática

e-mail esi@uclm.es

Teléfono 926 29 53 00

Web <http://www.esi.uclm.es>

© Los autores del documento. Se permite la copia, distribución y/o modificación de este documento bajo los términos de la licencia de documentación libre GNU, versión 1.1 o cualquier versión posterior publicada por la *Free Software Foundation*, sin secciones invariantes. Puede consultar esta licencia en <http://www.gnu.org>.

Índice general

Índice general	III
Índice de cuadros	XIII
Índice de figuras	XV
Índice de listados	XVII
Prefacio.	XXV
1. Shell	1
1.1. GNU Bash	2
1.2. Valor de retorno	3
1.3. Procesos	4
1.3.1. Trabajos	5
1.3.2. Otros modos de identificar procesos	7
1.4. Entrada, salida y salida de error	7
1.5. Redirección.	8
1.6. Paquetes.	11
1.6.1. Repositorios de paquetes	13
1.7. Catálogo de comandos	15
1.7.1. Ficheros y directorios	15
1.7.2. Sistema	17
1.7.3. Procesos	18

1.7.4. Usuarios y permisos	18
1.7.5. Servicios	19
2. Redes y protocolos	21
2.1. Tecnología Internet	21
2.2. Pila de protocolos	23
2.2.1. modelo OSI	24
2.2.2. Modelo TCP/IP	25
2.2.3. Modelo híbrido	25
2.3. Protocolos	25
3. Conectividad	27
3.1. Configuración básica	27
3.2. Interfaces de red.	28
3.2.1. <i>Loopback</i>	29
3.2.2. <i>eth0</i>	29
3.3. Encaminamiento básico	29
3.4. Conectividad, por capa	30
3.4.1. Conectividad a nivel de enlace	31
3.4.2. Conectividad a nivel de red	32
3.4.3. Conectividad a nivel de transporte.	36
3.4.4. Conectividad a nivel de aplicación	38
4. Pila de protocolos TCP/IP	41
4.1. Protocolos esenciales	42
4.1.1. Ethernet	42
4.1.2. IP.	45
4.1.3. UDP.	48
4.1.4. TCP.	48
4.2. Ejercicios	48
4.2.1. Enlace	48

4.2.2. Red	49
4.2.3. Transporte	49
5. Captura y análisis de tráfico de red.	51
5.1. tshark.	51
5.1.1. Advertencia de seguridad	52
5.1.2. Limitando la captura	53
5.1.3. Filtros de captura	53
5.1.4. Formato de salida	54
5.1.5. Filtros de visualización	56
5.1.6. Generación de estadísticas	57
5.2. Respuestas	61
6. Encapsulación de protocolos	63
6.1. En este capítulo...	63
6.2. Entorno	63
6.3. Situación inicial	64
6.4. Wireshark	64
6.4.1. Captura de paquetes	65
6.4.2. Interpretando de resultados	66
6.4.3. Filtrado	68
6.4.4. Seguir un stream.	69
7. Encaminamiento	71
7.1. En este capítulo...	72
7.2. Almacenamiento y reenvío.	72
7.2.1. Entrega directa e indirecta	73
7.3. Tabla de rutas	74
7.3.1. Un ejemplo	75
7.3.2. Tabla de rutas básica	76
7.4. Estático versus dinámico	78

8. Multicast.	79
8.1. IP multicast	80
8.2. Encaminamiento multicast.	80
8.2.1. Reenvío según ruta inversa.	82
8.2.2. Árboles de núcleo	82
9. Redes Privadas	85
9.1. Líneas alquiladas	85
9.2. Redes privadas TCP/IP	86
9.2.1. Direccionamiento privado	87
9.3. Conectividad en redes privadas	88
9.3.1. Traducción de Direcciones de Red (NAT)	89
9.4. Red Privada Virtual (VPN)	92
9.4.1. Acceso a VPN	92
9.4.2. Autenticación y gestión remota	93
9.5. Protocolos punto-a-punto	93
9.6. Protocolos tunelizados	94
9.6.1. PPP over IP.	95
10. Python.	97
10.1. Empezamos	97
10.2. Variables y tipos.	98
10.3. Tipos de datos	98
10.3.1. Valor nulo	98
10.3.2. Booleanos	98
10.3.3. Numéricos	99
10.3.4. Secuencias	99
10.3.5. Orientado a objetos.	101

10.4.	Módulos	101
10.5.	Estructuras de control	101
10.6.	Indentación estricta	102
10.7.	Funciones	102
10.8.	Python <i>is different</i>	103
10.9.	Hacer un fichero ejecutable	103
10.10.	<i>Type checking</i>	103
11.	Serialización	105
11.1.	Representación, sólo eso.	105
11.2.	Los enteros de Python	107
11.3.	Caracteres	107
11.4.	Tipos multibyte y ordenamiento	109
11.5.	Cadenas de caracteres y secuencias de bytes.	112
11.6.	Empaquetado	113
11.7.	Desempaquetado	116
12.	Sockets BSD	117
12.1.	Sockets	117
12.1.1.	Historia	117
12.2.	Creación de un socket	118
12.2.1.	IPC	119
12.3.	Uso del socket.	119
12.3.1.	Datos binarios.	122
12.4.	Desconexión	122
12.4.1.	Cuando los sockets mueren.	123
12.5.	Sockets no bloqueantes	123
12.5.1.	Rendimiento	125
12.6.	Créditos	125

13. Modelo Cliente-Servidor.	127
13.1. Chat	129
13.1.1. Paso 1: Mensaje unidireccional	129
13.1.2. Paso 2: Lo educado es responder	131
13.1.3. Paso 3: Libertad de expresión.	132
13.1.4. Paso 4: Habla cuando quieras.	134
13.1.5. Paso 5: Todo en uno	136
13.2. Chat UDP con <code>select()</code> .	137
13.3. Chat UDP con <code>asyncio</code>	138
13.4. Chat TCP	138
13.5. Puertos ocupados	138
13.6. Ejemplos rápidos de sockets	139
13.6.1. Un cliente HTTP básico	139
13.6.2. Un servidor HTTP	139
14. Puertos y servicios	141
14.1. <code>netstat</code>	142
14.1.1. Visualizar la tabla de rutas.	142
14.1.2. Lista de interfaces de red	143
14.1.3. Listar servidores	143
14.1.4. Filtrar listado de sockets.	143
14.1.5. Otras opciones	143
14.2. <code>nmap</code>	144
14.3. <code>IPTraf.</code>	145
14.4. Referencias	145
15. Sockets RAW.	149
15.1. Acceso privilegiado.	150
15.2. Tipos	151

15.3. Sockets AF_PACKET:SOCK_RAW	151
15.3.1. Construir y enviar tramas	153
15.3.2. Implementando un arpíng	154
15.4. Sockets AF_INET:SOCK_RAW	157
15.4.1. Capturando mensajes	157
15.4.2. Enviando	158
15.5. Ejercicios	159
16. Filtrado de paquetes	163
16.1. Introducción	163
16.2. Tablas y reglas	163
16.3. Políticas	164
16.4. Comandos básicos	165
16.4.1. Ver la configuración	165
16.4.2. Borrar todas las reglas de una tabla	165
16.5. Configuración de un router (con esquema 3).	165
16.5.1. Bloquear un puerto (en esquema 1)	167
16.5.2. Redireccionar un puerto del router hacia otro host (interno o externo)	167
16.5.3. Guardar la configuración de iptables	167
I Seguridad	169
17. Escaneo de servidores y servicios	171
17.1. Descubrimiento de hosts	171
17.2. Otros protocolos	174
17.3. Escaneo de puertos	175
17.4. Identificación de servicios	178
17.5. Identificación del Sistema Operativo	179
18. Herramientas de criptografía	181
18.1. Cálculo de resumen	181

18.2. Cifrado simétrico	182
18.3. Cifrado asimétrico	183
18.3.1. Creación del par de claves	183
18.3.2. Anillo de claves	184
18.3.3. Cifrado	185
18.3.4. Descifrado	186
18.4. Autenticación mediante firma digital.	186
18.4.1. Relaciones de confianza	188
19. Autenticación.	189
19.1. Espacio de claves	190
19.2. Fuerza bruta	191
19.3. Ataques de diccionario	192
19.4. Rainbow tables	192
19.5. Caso de estudio: Microsoft LAN Manager.	193
19.6. Autenticación remota.	194
 II Servicios básicos	 197
20. SSH	199
20.1. Shell segura	199
20.2. Configuración.	200
20.3. Acceso con clave pública	200
20.4. Autenticación mediante certificado	201
20.4.1. Un certificado para el usuario.	202
20.5. Copia de ficheros con SCP	204
 A. Netcat	 207
A.1. Sintaxis	207
A.2. Ejemplos.	208
A.2.1. Un chat para dos.	208

A.2.2. Transferencia de ficheros.	208
A.2.3. Servidor de echo	208
A.2.4. Servidor de daytime	209
A.2.5. Shell remota estilo telnet	209
A.2.6. Telnet inverso	209
A.2.7. Cliente de IRC.	209
A.2.8. Cliente de correo SMTP	210
A.2.9. HTTP	210
A.2.10.Streaming de audio.	211
A.2.11.Streaming de video	211
A.2.12.Proxy	211
A.2.13.Clonar un disco a través de la red	211
A.2.14.Ratón remoto	212
A.2.15.Medir el ancho de banda.	212
A.2.16.Imprimir un documento en formato PostScript	213
A.2.17.Ver «La Guerra de las Galaxias»	213
A.3. Otros «netcat»s	213
Referencias	215

Índice de cuadros

- 5.1. tshark: ejemplos de filtros de captura 54
- 5.2. tshark: ejemplos de filtros de visualización 57
- 7.1. Ejemplo: Tabla de rutas de router R0 76
- 9.1. Bloques IP reservados para direccionamiento privado 88
- 9.2. Tabla NAT para el envío de la figura 9.5 91
- 9.3. Tabla NAPT para el envío de la figura 9.6 91
- 11.1. struct: especificación de ordenamiento 114
- 11.2. struct: especificación de formato 115

Índice de figuras

- 2.1. Correspondencia entre los diferentes modelos de referencia 26
- 3.1. Conector RJ-45 hembra de un equipo Ethernet 31
- 3.2. Aspecto del programa mtr 36
- 3.3. *Chromium* muestra la «página web» que devuelve ncat 39
- 4.1. Pila de protocolos TCP/IP 41
- 4.2. Formato de la trama Ethernet 43
- 4.3. Formato de la dirección Media Access Control (MAC) Ether-
net [Fuente:Wikimedia Commons] 44
- 4.4. Formato de las direcciones Internet Protocol (IP) 45
- 4.5. Formato del paquete IP 46
- 4.6. Formato del segmento TCP 48
- 6.1. Situación inicial 64
- 6.2. Ventana inicial 65
- 6.3. Opciones de captura 66
- 6.4. Ventana principal 67
- 6.5. Wireshark: siguiendo un stream 70
- 7.1. Topología parcial que se deduce de la tabla de rutas 7.1 76
- 8.1. Árbol sumidero para R y árboles podados para los grupos
G1 y G2 81

9.1. Red privada que utiliza líneas alquiladas Esta imagen tiene una licencia Creative Commons Attribution-Share Alike 4.0. Disponible en <https://commons.wikimedia.org> . . . 86

9.2. Red privada híbrida 86

9.3. Conexiones de un router ADSL doméstico Esta imagen tiene una licencia Creative Commons Attribution-Share Alike 3.0. Disponible en <https://commons.wikimedia.org> . . 89

9.4. Ejemplo de configuración Network Address Translation (NAT) en una red doméstica 90

9.5. Ejemplo de Source NAT (SNAT) 90

9.6. Ejemplo de NAPT 91

9.7. Encapsulado PPTP 96

9.8. Encapsulado L2TP 96

11.1. Ordenación de bytes 110

13.1. Modelo Cliente/Servidor [Fuente:Wikimedia Commons] 127

14.1. 146

14.2. 147

14.3. 148

19.1. Recuperación de claves Microsoft LAN Manager con ophcrack . . 194

19.2. Edición de la plantilla para una ataque de fuerza bruta con Burb Suite 195

Índice de listados

- 5.1. tshark capturando tráfico ICMP 51
- 5.2. Modo «verboso» de tshark 55
- 5.3. tshark permite elegir los campos a imprimir 56
- 11.1. Literales numéricos en Python 106
- 11.2. Conversión a representación binaria, octal y hexadecimal 106
- 11.3. Especificando la base en el constructor de int 107
- 11.4. Averiguar el ordenamiento de bytes con Python. 110
- 11.5. Funciones de conversión de ordenamiento del módulo socket. . . 111
- 11.6. Codificación ASCII 112
- 11.7. Codificación UTF-8 113
- 11.9. struct: empaquetado en diferentes tamaños 114
- 11.8. struct: alternativas de ordenamiento 114
- 11.10 struct: empaquetando una cabecera Ethernet 115
- 11.11 struct: desempaquetando una cabecera Ethernet 116
- 13.1. Servidor de chat UDP básico
 - udp-chat/server1.py 129
- 13.2. Cliente de chat UDP básico
 - udp-chat/client1.py 131
- 13.3. Servidor de chat UDP con respuesta
 - udp-chat/server2.py 132
- 13.4. Cliente de chat UDP con respuesta
 - udp-chat/client2.py 132
- 13.5. Servidor de chat UDP por turnos
 - udp-chat/server3.py 132

13.6. Cliente de chat UDP por turnos	
udp-chat/client3.py	133
13.7. Servidor de chat UDP simultaneo	
udp-chat/server4.py	134
13.8. Cliente de chat UDP simultáneo	
udp-chat/client4.py	135
13.9. Chat UDP multihilo (servidor y cliente)	
udp-chat/chat-thread.py	136
13.10 Chat UDP (servidor y cliente) con select()	
udp-chat/chat-select.py	137
13.11 Cliente HTTP básico	
examples/http_mini_client.py	139
13.12 Servidor HTTP	
examples/http_server.py	139
15.1. Sniffer básico con AF_PACKET:SOCK_RAW	
raw/sniff-all.py	151
15.2. Sniffer AF_PACKET:SOCK_RAW filtrando ARP	
raw/sniff-arp.py	152
15.3. Enviando una trama Ethernet con AF_PACKET:SOCK_RAW	
raw/send-wrong-eth.py	153
15.4. Sniffer de mensajes UDP con AF_INET:SOCK_RAW	
raw/sniff-udp.py	157
15.5. Sintetizando un mensaje UDP con AF_INET:SOCK_RAW	
raw/send-udp.py	158

Listado de acrónimos

AAA	Authentication, Authorization and Accounting
ACK	Acknowledgement
ADSL	Asymmetric Digital Subscriber Line
AH	Authentication Header
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
CA	Certificate Authority
CAD	Computer-Aided Design
CD	Continuous Delivery
CHAP	Challenge Handshake Authentication Protocol
CI	Continuous Integration
CIDR	Classless Interdomain Routing
CIFS	Common Internet File System
CLI	Command Line Interface
CPU	Central Processing Unit
CR	Carriage Return
CUPS	Common Unix Printing System
DF	Don't Fragment
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System

EAP	Extensible Authentication Protocol
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
FTTH	Fiber To The Home
GNU	GNU is Not Unix
GUI	Graphical User Interface
GRE	Generic Routing Encapsulation
HFC	Hybrid Fiber-Coaxial
HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
IANA	Internet Assigned Numbers Authority
IBM	International Business Machines
ICMP	Intenet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IGMP	Internet Group Management Protocol
IHL	Internet Header Length
IMAP	Internet Message Access Protocol
IP	Internet Protocol
IPSec	Internet Protocol Security
IPv4	IP versión 4
IPv6	IP versión 6
IPC	Inter-Process Communication
IPv6	IP versión 6
IPX	Internetwork Packet Exchange
ISN	Initial Sequence Number
IRC	Internet Relay Chat
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization

ISP	Internet Service Provider
JSON	JavaScript Object Notation
KiB	Kibibyte (1 024 bytes)
L2TP	Layer 2 Tunneling Protocol
LAC	L2TP Access Concentrator
LAN	Local Area Network
LCP	Link Control Protocol
LED	Light Emitting Diode
LF	Line Feed
LM	Microsoft LAN Manager
LNS	L2TP Network Server
MD5	Message-Digest Algorithm 5
MiB	Mebibyte (1 024 KiB)
MAC	Media Access Control
MF	More Fragments
MTU	Maximum Transmission Unit
NAPT	Network Address Port Translation
NAS	Network Access Server
NAT	Network Address Translation
NCP	Network Control Protocol
NIC	Network Interface Controller
OSI	Open Systems Interconnection
OUI	Organizationally Unique Identifier
PAC	PPTP Access Concentrator
PAP	Password Authentication Protocol
PC	Personal Computer
PDU	Protocol Data Unit
PID	Process IDentifier
PNS	PPTP Network Server
POO	Programación Orientada a Objetos
POP	Post Office Protocol
POP3	Post Office Protocol

POSIX	Portable Operating System Interface; UNIX
PPP	Point to Point Protocol
PPTP	Point-to-Point Tunneling Protocol
PSDN	Public Switched Data Network
RDP	Remote Desktop Protocol
RADIUS	Remote Authentication Dial In User Service
REST	REpresentational State Transfer
RFC	Request For Comments
ROM	Read Only Memory
RPF	Reverse Path Forwarding
RSA	Rivest, Shamir y Adleman
RTC	Red Telefónica Conmutada
RTT	Round Trip Time
SCP	Secure Copy Protocol
SDSL	Symmetric Digital Subscriber Line
SHA	Secure Hash Algorithm
SMB	Server Message Block
SMTP	Simple Mail Transport Protocol
SNAT	Source NAT
SO	Sistema Operativo
SRT	Service Response Time
SSH	Secure SHell
VNC	Virtual Network Computing
TCP	Transmission Control Protocol
TCP/IP	Arquitectura de protocolos de Internet
TLS	Transport Layer Security
ToS	Type of Service
TTL	Time To Live
TTY	TeleTYpewriter
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URI	Universal Resource Identifier

UTF	Unicode Transformation Format
UTF-8	UTF de 8 bits
UUID	Universally Unique Identifier
VLC	Video LAN Client
VPN	Virtual Private Network
WAN	Wide Area Network
WiFi	Wireless Fidelity
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

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 precisamente 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 con la que 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.

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 porque este será el entorno que se asume cuando se habla de configuración o comandos de sistema. Por supuesto, es posible usar 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 o WiFi convenientemente configurada y que proporciona acceso a Internet sin restricciones relevantes.

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 asimilación 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 git en <https://github.com/UCLM-ARCO/net-book-code.git>. Aunque es posible descargar estos ficheros individualmente o como un archivo comprimido, se aconseja utilizar el sistema de control de versiones git¹.

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://github.com/UCLM-ARCO/net-book-code/issues> para notificarlo a sus autores.

Sobre este documento

Este documento está tipografiado con L^AT_EX en un sistema Debian GNU/Linux. Las figuras y la mayoría de los diagramas están realizados con inkscape.

Los fuentes de este documento también se encuentran en un repositorio git <https://github.com/UCLM-ARCO/net-book>. Si quiere colaborar activamente en su desarrollo o mejora 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.

¹<https://git-scm.com>

Capítulo 1

Shell

David Villa

El interprete de comandos (*shell*) es probablemente una de las herramientas más interesantes del sistema operativo GNU/Linux o POSIX en general. Una shell es esencialmente un programa interactivo que permite al usuario ejecutar otros programas. 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 Command Line Interface (CLI)— es mucho menos intuitiva¹ que una GUI. Sin embargo, cuando la capacidad y el nivel de detalle (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. Ejemplos muy evidentes de esto se pueden encontrar en programas de diseño Computer-Aided Design (CAD) con literalmente miles de opciones repartidas en innumerables menús. 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*.

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.

«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 usar 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-ends*³ gráficos, línea de comandos, o incluso desarrollando otras aplicaciones a medida. Un ejemplo de este tipo de aplicaciones puede ser VLC. Incluso servicios de Internet tan conocidos como HTTP, SMTP o FTP pueden ser utilizados mediante comandos de texto, tal como veremos en capítulos posteriores.

1.1. GNU Bash

Bash es probablemente una de las shells más famosas y potentes disponibles en los sistemas GNU, aunque obviamente no es la única. Los sistemas POSIX han conocido una larga variedad de shell's desde principios de los años 70. Aunque bash tiene características muy interesantes (como la «complección» automática contextual) primero debemos abordar las cuestiones básicas que todo usuario avanzado debería 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.

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

³Se denomina *front-end* a la capa de software que proporciona 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)

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 satisfactoriamente debería devolver siempre un valor 0. Puede devolver un valor distinto (1, 2, etc) para indicar que hubo un problema concreto.

Éste es el motivo por el que el estándar del lenguaje C dice que toda función `main()` ha de 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 /
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` ha creado un proceso en el cuál ha ejecutado 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 `?`⁴. Se puede comprobar el valor de una variable utilizando el comando `echo` y colocando el símbolo `«$»` delante del nombre de la variable, tal que:

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

⁴Sí, el signo de interrogación

Pero esto 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
```

echo

Escribe la cadena de texto que le se indique como argumento, expendiendo las variables que incluya (identificadores precedidos del símbolo '\$').

La documentación de cada programa debe indicar lo que significa cada uno de los posibles valores de retorno. Si consultamos la página de manual de `ls` (`man ls`) veremos que:

```
1      Exit status:
2      0          if OK,
3      1          if minor problems (e.g., cannot access subdirectory),
4      2          if serious trouble (e.g., cannot access command-line argument).
```

Las señales también se utilizan para indicar que un programa terminó como consecuencia de una situación anómala y el SO (o otro programa) lo terminó («lo mató»). Esta situación implica el uso de *señales*. Por ejemplo, si el usuario pulsa Control-C mientras un programa se ejecuta en una shell, esta le enviará la señal SIGINT (-2) y ese será el valor de retorno del programa.

1.3. 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. Cualquier proceso puede crear procesos hijos, que heredan algunas de sus propiedades tales como privilegios y descriptores de ficheros abiertos. Para listas procesos se utiliza `ps`, que es un abreviatura de *process status*. Por ejemplo, un usuario puede ver los procesos asociados al terminal al que está conectado con el comando `ps T`.

```
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 Process IDentifier (PID) de cada proceso. La segunda, (TTY), el terminal al que están asociados. La tercera, (TIME), el

tiempo de procesador otorgado al proceso hasta el momento. Y por último (CMD), el programa que se está ejecutando.

El programa `ps` dispone de una amplia variedad de opciones⁵ para filtrar qué procesos se desean listar, qué atributos o en qué formato. Por ejemplo, el siguiente comando muestra las relaciones de «genealogía»:

```
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
5300	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*).

Por ejemplo, si un proceso no responde a su interfaz gráfica (si la tiene) y el usuario no tiene control de otro modo, la operación más habitual es «matarlo» (con el comando `kill`). A pesar de su nombre, el comando `kill` sirve para enviar una señal a un proceso. La señal por defecto es SIGTERM, pero se puede enviar otra si se especifica como argumento. Puede ver la lista de todas las señales con `kill -l`. Por ejemplo, el siguiente comando envía la señal SIGKILL (9) al proceso con PID 5200:

```
david@amy:~$ kill -9 5200
[1]+ Terminado (killed)      gedit
```

Algunas señales pueden ser ignoradas o bien capturadas por el programa para realizar acciones especificadas por el programador⁶ como es el caso de SIGTERM, que podríamos considerarla una solicitud amistosa de terminación. Otras señales (como SIGKILL) no pueden ser ignoradas para garantizar que el SO tiene el control sobre cualquier proceso.

1.3.1. Trabajos

La shell crea un nuevo proceso hijo⁷ 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

⁵Es lo habitual en el mundoPOSIX.

⁶Vea man signal

⁷La propia shell también se ejecuta en un proceso, que puede haber sido creado a su vez por otra shell.

de comprobar con el programa `sleep`. Esto se llama ejecución en «primer plano» o «foreground».

Sin embargo, si se le pide la shell puede no esperar, 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 comandos:

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

Como se aprecia en el listado anterior, la shell imprime una línea con dos números y luego queda disponible en espera de introducir un nuevo 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                  gedit &
[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. Opcionalmente admite como argumento el identificador del trabajo que se desea llevar a primer plano.

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

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

```
david@amy:~$ fg %1
emacs
^Z
[1]+  Stopped                  gedit
david@amy:~$ jobs
[1]+  Stopped                  gedit
[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`.

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

Los indicadores de trabajo también se puede usar con el comando `kill` en lugar del PID.

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

1.3.2. Otros modos de identificar procesos

A veces buscar en la lista de procesos (`ps`) no es la forma más cómoda de localizar un proceso. Puede ser más sencillo localizarlo por su nombre (con `pidof` o `pgrep`):

```
david@amy:~$ pidof firefox
105894
```

O bien lo podemos identificar por un fichero o dispositivo que el proceso tenga abierto, o un puerto al que esté vinculado (con `fuser`):

```
david@amy:~$ fuser 17500/tcp
17500/tcp:      10589
```

También se puede enviar una señal (por defecto con la intención de matarlo) con ambos criterios (nombre y puerto):

```
david@amy:~$ pkill firefox
david@amy:~$ fuser --kill 17500/tcp
17500/tcp:      10589
```

Por supuesto, estos comandos tienen opciones que les permiten buscar procesos por otros criterios (como el propietario o el orden de creación) que además se pueden combinar.

1.4. Entrada, salida y salida de error

En los sistemas POSIX los programas interaccionan con el SO únicamente por medio de 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 hace sobre su salida estándar. Del mismo modo, cuando el programa intenta leer o genera un mensaje de error, esas operaciones se realizan 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 como «consola» o «terminal»). 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.

1.5. 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 creador de ese 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 doble mayor-que (`>>`):

```
david@amy:~$ date > /tmp/now
david@amy:~$ date >> /tmp/now
david@amy:~$ cat /tmp/now
Mon Feb  8 15:40:59 CET 2021
Mon Feb  8 15:41:02 CET 2021
```

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
```

/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 al apagar el computador.

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.

Se indica con doble mayor-que (`>>`):

date

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

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

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).

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`:

```
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
```

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.

```
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 quieren 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 comandos 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-sorted-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 eficiente⁸.

1.6. Paquetes

Muchas distribuciones de GNU/Linux utilizan paquetes para distribuir sus programas. Un paquete no es más que una colección de ficheros (normalmente comprimidos) estructurados para ser colocados en rutas concretas dentro del sistema de ficheros cuando se instalan.

En Debian GNU/Linux o Ubuntu, estos paquetes son ficheros con extensión `.deb`. Si descargas uno de estos ficheros directamente, puedes instalarlo con `dpkg`:

⁸La web <https://www.commandlinefu.com/> es una buena prueba de la gran versatilidad de la redirección y las capacidades de la shell.

```
david@amy:~$ ls
gedit_3.38.1-1_amd64.deb
david@amy:~$ sudo dpkg -i gedit_3.38.1-1_amd64.deb
(Leyendo la base de datos ... 497325 ficheros o directorios instalados actualmente.)
Preparando para desempaquetar gedit_3.38.1-1_amd64.deb ...
Desempaquetando gedit (3.38.1-1) sobre (3.38.1-1) ...
Configurando gedit (3.38.1-1) ...
```

Una característica muy importante de estos paquetes es que tienen *dependencias*, es decir, para que el programa que instalas pueda funcionar necesita que otros paquetes concretos deban instalarse primero. Puedes ver las dependencias (y otra mucha información) sobre un paquete con:

```
david@amy:~$ apt-cache show gedit
Package: gedit
Version: 3.38.1-1
Installed-Size: 1738
Maintainer: Debian GNOME Maintainers <pkg-gnome-maintainers@lists.aliases.debian.org>
Architecture: amd64
Depends: gedit-common (< 3.39), gedit-common (>= 3.38), gir1.2-glib-2.0, gir1.2-gtk-3.0
(>= 3.22), gir1.2-gtksource-4, gir1.2-pango-1.0, gir1.2-peas-1.0,
gsettings-desktop-schemas, iso-codes, python3-gi (>= 3.0), python3-gi-cairo (>= 3.0),
[...]
```

Como se puede ver, para muchos de ellos se indica que deben corresponder a cierta versión. En ese ejemplo, puedes ver que se requiere una versión del paquete `python3-gi` que sea mayor o igual a la 3.0.

Obviamente encontrar, descargar e instalar manualmente (con `dpkg` todos esos paquetes (y sus respectivas dependencias) es una tarea terriblemente pesada. Afortunadamente existe hay otro modo de hacerlo.

El programa `apt` (o `apt-get`) puede determinar automática y recursivamente las dependencias de un paquete, descargarlos e instalarlos en el orden correcto.

```
david@amy:~$ sudo apt install gedit
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  gedit-common gir1.2-gtksource-4 libatk-5-0 libatk-5-common libtepl-5-0
Paquetes sugeridos:
  gedit-plugins
Se instalarán los siguientes paquetes NUEVOS:
  gedit gedit-common gir1.2-gtksource-4 libatk-5-0 libatk-5-common libtepl-5-0
0 actualizados, 6 nuevos se instalarán, 0 para eliminar y 172 no actualizados.
Se necesita descargar 59,0 kB/2.146 kB de archivos.
Se utilizarán 15,5 MB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n]
```


1.6.1. Repositorios de paquetes

La pregunta obvia es «¿Cómo sabe `apt` de dónde descargar todos esos paquetes?». Las distribuciones proporcionan servidores (normalmente web o FTP) dónde ponen accesibles los ficheros `.deb` públicamente. Por ejemplo, en el caso de Debian en <https://ftp.debian.org/debian/>, del cual existen cientos de «copias espejo» (*mirrors*), normalmente una por país, como <https://ftp.es.debian.org/debian/>.

En estos repositorios, los paquetes se organizan en «releases» (o versiones)⁹. En el caso de Debian, estas «releases» tienen un número de versión y un nombre asociado. Por ejemplo Debian 10 lleva el nombre «buster».

De este modo, si queremos instalar (o actualizar) a una release, debemos indicárselo a `apt` mediante el fichero `/etc/apt/sources.list`:

```
deb http://deb.debian.org/debian/ buster main contrib non-free
```

Esta línea en concreto dice que queremos poder instalar paquetes oficiales mantenidos por Debian (*main*), contribuciones de terceros (*contrib*) y software con licencias no libres (*non-free*).

Este fichero puede contener muchos repositorios, y también se pueden crear otros ficheros con extensión `.list` dentro de `/etc/apt/sources.list.d` que tienen el mismo tipo de contenido.

Para saber qué paquetes (y versiones) están disponibles en los repositorios configurados `apt` debe descargar una especie de índices que se encuentran allí mismo. Para eso debemos ejecutar:

```
david@amy:~$ sudo apt update
```

Esto debemos hacerlo regularmente¹⁰ porque el contenido de los repositorios cambia y regularmente se añaden nuevos paquetes o versiones.

En concreto en Debian hay siempre tres versiones activas:

stable Es la última versión publicada y su contenido no debería cambiar. Esta corresponde con «buster» en el momento de escribir este documento.

testing Contiene los paquetes que se están preparando para una futura versión estable y por tanto, cambia continuamente. En este momento

⁹<https://www.debian.org/releases/>

¹⁰De hecho las distribuciones actuales tienen sistemas que lo hacen de forma automática

se denomina «bullseye» y ese será el nombre que tendrá la siguiente versión estable.

unstable Que tiene paquetes recién incorporados, experimentales o que tienen algunos problemas importantes para ser incluidos en una futura versión. Esta versión siempre se llama «sid».

Por cuestiones de seguridad es importante que se puedan actualizar paquetes en los que se han descubierto vulnerabilidades o problemas graves, incluso aunque correspondan a una versión estable. Por eso, es conveniente tener los siguientes repositorios en el fichero `/etc/apt/sources.list`:

```
deb http://security.debian.org/debian-security buster/updates main
deb http://deb.debian.org/debian/ buster-updates main
```

Todo esto significa que los repositorios configurados determinan qué paquetes (y qué versiones) podemos instalar. Si queremos tener un sistema completamente actualizado, es decir, con las últimas versiones de todos los paquetes disponibles en esos repositorios podemos ejecutar:

```
david@amy:~$ sudo apt upgrade
```

Si incorporamos repositorios de versiones nuevas y queremos actualizarnos a ellos, tendremos que ejecutar:

```
david@amy:~$ sudo apt dist-upgrade
```

Esto hará que la versión de nuestro sistema operativo corresponda con la versión del repositorio más actual. Eso lo podemos ver con:

```
david@amy:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:   Debian GNU/Linux bullseye/sid
Release:       10.8
Codename:      buster
```

También podemos saber qué versiones de un determinado paquete tenemos disponibles en los repositorios configurados y en cuál de ellos están:

```
david@amy:~$ apt-cache policy openconnect
openconnect:
  Instalados: (ninguno)
  Candidato:  8.02-1+deb10u1
  Tabla de versión:
    8.10-2+b1 650
    650 http://deb.debian.org/debian testing/main amd64 Packages
    600 http://deb.debian.org/debian sid/main amd64 Packages
```

```
8.02-1+deb10u1 700
```

```
700 http://deb.debian.org/debian buster/main amd64 Packages
```

```
700 http://security.debian.org/debian-security buster/updates/main amd64 Packages
```

Esto significa, que no podremos instalar versiones de los paquetes que no estén en los repositorios configurados. La solución puede ser añadir un repositorio más reciente que sí lo contenga, pero teniendo cuidado porque las nuevas versiones de las dependencias podrían afectar a otros paquetes.

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

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.

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
```

diff muestra las diferencias entre dos ficheros.

find encuentra ficheros a partir de su nombre.

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).

mkdir (*make dir*)

crea un directorio.

mkfifo

crea una tubería con nombre ligado al sistema de ficheros.

mv (*move*)

es equivalente a **cp** pero borra el fichero original.

nl (*number lines*)

lee líneas de un fichero y las escribe a su salida precedidas del número de línea.

pwd (*print working directory*)

indica el nombre del directorio actual.

rm (*remove*)

elimina el fichero indicado.

rmdir (*remove dir*)

borra un directorio vacío.

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 al mismo tiempo a su salida estándar y al fichero indicado.

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, crea uno vacío.

tr (*translate*)

lee líneas de su entrada y las escribe a su salida reemplazando *tokens*¹¹ 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.

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

dd copia bloques de bytes en dispositivos (ficheros o discos).

df (*disk free*) muestra información sobre el uso de los sistemas de ficheros montados en el computador.

du (*disk usage*) calcula el espacio de disco utilizado por ficheros y directorios.

fdisk

muestra y manipula tablas de particiones.

mount

monta dispositivos de almacenamiento de cualquier tipo sobre el sistema de ficheros.

uname

(*unix name*) muestra información del sistema:

```
david@amy:~$ uname -a
Linux amy 5.8.0-3-amd64 #1 SMP Debian 5.8.14-1 (2020-10-10) x86_64 GNU/Linux
```

¹¹Un *token* es cualquier combinación de caracteres que cumpla una expresión regular concreta.

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 (SIG-HUP). Permite que un proceso creado por una shell sobreviva a la propia shell.

sleep

realiza una pausa del número de segundos indicados.

true, false

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

top , muestra una lista actualizada de los procesos ordenada por el consumo de CPU.

1.7.4. Usuarios y permisos

chmod (*change mode*)

cambia los permisos de lectura, escritura y ejecución de un fichero o directorio.

chmod (*change owner*)

cambia el propietario (usuario y grupo) de un fichero.

chgrp (*change group*)

cambia el grupo propietario de un fichero.

groups

muestra los nombres de los grupos a los que pertenece el usuario.

id muestra los identificadores del usuario y los grupos a los que pertenece.

sudo (*superuser do*) permite ejecutar un comando con los privilegios de otro usuario, por defecto el administrador.

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.

1.7.5. Servicios

Los servicios son programas en segundo plano (que arrancan normalmente el iniciar el sistema) que se encuentran bajo el control del SO que no se ejecutan directamente por los usuarios y se encargan de tareas específicas: firewall, sonido, bluetooth, etc., servidores: web, ssh, etc. u otras tareas de gestión: registro de eventos del sistema, montaje de dispositivos de almacenamiento, etc.

Actualmente muchas distribuciones GNU/Linux usan el `systemd`, pero aquí vamos a ver los comandos *legacy* que son compatibles.

`sudo service ---status-all`

muestra el estado de todos los servicios instalados.

`sudo service <service> status`

muestra el estado del servicio indicado.

`sudo service <service> stop/start/restart`

para, arranca o reinicia el servicio indicado.

Capítulo 2

Redes y protocolos

David Villa

Internet tiene un nombre perfectamente autodefinido. Internet es una inter-red (**inter-network**), es decir, un sistema formado por la interconexión de miles de redes de computadores.

Una red de computadores es simplemente un conjunto de computadores autónomos¹ conectados entre sí de modo que puedan compartir recursos (datos o dispositivos.). Todos los computadores de una red concreta utilizan una misma tecnología de comunicaciones (p.ej: WiFi). Denominamos a esto una LAN y tiene utilidad por si misma aunque esté aislada, como puede ser la red de control de una planta de fabricación industrial.

En este capítulo

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

- Qué es una red de computadores.
- Qué es y cuál es la utilidad de los modelos de referencia OSI, TCP/IP e híbrido.

2.1. Tecnología Internet

Como en tantos otros campos, ocurre que no existe una sola tecnología «perfecta» para resolver todos los problemas. En nuestro caso, no hay una tecnología universal para implementar una LAN, hay cientos. Por tanto, el problema clave que resuelve Internet (en concreto IP²), es interconectar redes heterogéneas –con tecnología potencialmente incompatibles.

¹que pueden realizar tareas por si mismos

²Por supuesto no es el único modo, pero es el que abordamos en este texto.

La tecnología IP, que significa literalmente *Internet Protocol*, es decir, **protocolo de inter-red**, lo logra mediante tres elementos:

- Un protocolo y un formato de mensaje universal –el protocolo y el paquete IP– que se debe utilizar en todo aquel computador y dispositivo de interconexión (*router*) que se quiera integrar en la inter-red, independientemente de su tecnología LAN. No hablamos de substituir «su protocolo» por IP sino de que lo encapsulen (ver FIXME).
- Un esquema de direccionamiento global (las direcciones IP). Para que un computador tenga la capacidad de enviar paquetes IP a cualquier otro –en cualquier parte– debe disponer de una dirección IP adecuada.
- Un dispositivo capaz de reenviar paquetes IP entre redes dos o más redes y hacerlos llegar a su destino. Hablamos de los routers o pasarelas IP (*gateways*).

Esto resuelve un problema importante: llevar paquetes de datos desde un computador a cualquier otro, solo conociendo la dirección IP del destino, sin importar dónde esté o la tecnología que utilice para conectarse a **su** red.

Pero hay muchos otros problemas...

Son **las aplicaciones** las que realmente necesitan compartir información. Las aplicaciones que utilizan la red (la inmensa mayoría hoy día) requieren un modo de enviar datos de su dominio específico a otra aplicación, o dicho con más precisión, a otro componente de la misma aplicación que se está ejecutando en otro computador. Llamamos a esto «aplicaciones distribuidas».

La red, o más concretamente, el sistema operativo³, debe ofrecer una forma de que dos procesos ejecutándose en computadores cualesquiera puedan «direccionarse» el uno al otro, es decir, que puedan identificar unívocamente a otro proceso que se ejecuta en un computador remoto accesible a través de la red⁴. Esto se logra mediante la combinación de dos datos: la dirección IP, que identifica el computador remoto, y el puerto, que identifica un proceso dentro de ese computador.

Además muchas aplicaciones necesitan garantías de que estos mensajes lleguen correctamente a su destino, sin cambios (respetando la integridad), sin partes ausentes (omisiones), recibidos en el mismo orden en el que se

³Y más concretamente aún, el subsistema de red del SO.

⁴El direccionamiento de procesos dentro de un computador se denomina en realidad «multiplexación».

enviaron, asegurando la privacidad, sin saturar al receptor y evitando congestionar la red.

La mayoría de estos problemas los resuelven los **protocolos de transporte**. En el caso de Internet estos protocolos son:

TCP – Proporciona un flujo (*stream*) de datos entre dos procesos de forma fiable y ordenada.

UDP – Ofrece un servicio de entrega de datagramas (mensajes independientes) entre dos procesos, pero sin ninguna garantía.

TLS – A grandes rasgos, ofrece mecanismos para cifrado de mensajes y permite asegurar la legitimidad del proceso remoto.

Estos tres protocolos son genéricos puesto que se pueden aplicar sobre cualquier carga útil (*payload*), es decir, son independientes del contenido de los mensajes que las aplicaciones necesitan enviar.

El formato específico de los datos que utiliza cada aplicación también está definido por protocolos –llamados **protocolos de aplicación**– aunque debemos entender el concepto «aplicación» de un sentido amplio. Por ejemplo, todo el software relacionado con la web utilizan el protocolo de aplicación HTTP. Al contrario de lo que sucede con los protocolos de inter-red y transporte, hay miles de protocolos de aplicación, muchos de ellos privativos y secretos. Sin embargo, hay unos cuantos que son públicos, bien documentados y de uso común. Algunos de estos son DNS, SMTP/IMAP/POP, FTP, SSH o el ya mencionado HTTP.

2.2. Pila de protocolos

Como vemos, hay varios problemas bastante diferentes a resolver para conseguir nuestro objetivo: que dos procesos puedan intercambiar mensajes. Y también vemos que hay varios tipos de protocolos –red, transporte, aplicación, etc.– que abordan distintos problemas a distintos niveles.

Para manejar más cómodamente esos problemas y sus soluciones, el estudio, diseño, especificación e implementación de todo lo relacionado con las redes, se suele organizar todo ello en capas o niveles. Por eso, los distintos protocolos, correspondientes a cada una de las capas forman una **pila** de protocolos (*protocol stack*). Se llama así porque está formada por un conjunto de protocolos «apilados» siguiendo las directrices del modelo correspondiente. Es sencillo entender que todo esto en realidad no es más que una conveniencia de abstracción.

2.2.1. modelo OSI

El modelo de referencia OSI —definido por la ISO— está pensado para aplicarse a cualquier tecnología de comunicaciones. El modelo describe las interfaces, protocolos y servicios que proporciona cada una de las siete capas que lo componen: aplicación, presentación, sesión, transporte, red, enlace y física. Como se ha dicho, cada capa se centra en resolver un conjunto de problemas específicos. Cada capa 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. Ejemplo: especifica las dimensiones de los conectores, el voltaje que puede haber en cada pin y su significado, el tamaño y forma de las antenas, las frecuencias que utiliza un emisor, etc.
2. **Enlace** – Se ocupa del intercambio de mensajes entre nodos vecinos, (directamente conectados). Cuando se usa un medio de difusión, es decir, varios nodos comparten un medio físico, 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*. Una de sus funciones más importantes en la capacidad de interconectar redes de tecnología distinta
4. **Transporte** – Proporciona un canal de comunicación libre de errores entre formando inter-redes. 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 específicos de cada aplicación.

2.2.2. Modelo TCP/IP

El modelo de referencia 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 computador a sus vecinos, es decir, otros computadores conectados al mismo enlace. En realidad no aborda la problemática específica que ello implica, simplemente da por hecho que es un problema resuelto.
2. **Inter-red** – Proporciona los mecanismos de interconexión de redes y encaminamiento de paquetes. Define el protocolo Internet Protocol (IP), que proporciona un servicio de entrega sin conexión.
3. **Trasporte** – Define dos protocolos de transporte: TCP que proporciona un servicio confiable y orientado a conexión y UDP que únicamente proporciona multiplexación y detección muy básica de errores.
4. **Aplicación** – Incluye los protocolos para los servicios comunes, tales como: DNS, SMTP, HTTP, FTP, etc.

Una aclaración necesaria es que las siglas TCP/IP no se refieren exclusivamente al par de protocolos TCP e IP, sino a la pila completa que se muestra en la figura y todas las implicaciones funcionales que conlleva.

2.2.3. Modelo híbrido

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:

2.3. Protocolos

Un protocolo define las pautas y normas específicas que se deben aplicar para efectuar una comunicación correcta⁵. Todo protocolo debe definir tres

⁵Similar a la acepción que se aplica en las relaciones diplomáticas.

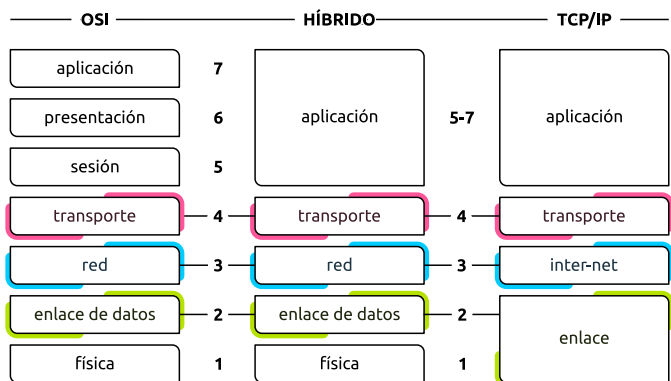


FIGURA 2.1: Correspondencia entre los diferentes modelos de referencia

aspectos clave: sintaxis, semántica y secuenciación, es decir, cuestiones como formato y tamaño de los mensajes, los rangos de valores admisibles, el tipo y significado de cada campo, condiciones de error, patrones válidos de intercambio de mensajes, etc.

En los siguientes capítulos se abordan algunos de los protocolos más importantes de Internet:

Capítulo 3

Conectividad

David Villa

La *conectividad* es 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 muy útiles para resolver una gran cantidad de fallos y problemas que pueden presentarse en redes de comunicaciones de cualquier tipo.

Aprender a manejar estas herramientas y conocer sus fundamentos teóricos es 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 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 para verificación de conectividad.

3.1. Configuración básica

Veamos en primer lugar algunas nociones sobre la configuración de su computador, especialmente lo referente a sus interfaces de red y servicios básicos de sistema operativo.

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 un terminal, muestra el nombre asignado al computador, sin incluir su dominio.

```
$ hostname
storm
```

Si dispone de privilegios de administrador (cuenta de root) puede utilizar `hostname` para cambiar el nombre del computador. Para que este cambio permanezca después de reiniciar debe editar el fichero `/etc/hostname` (lo cual también requiere permisos de administrador).

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

```
$ uname -a
Linux storm 5.8.0-3-amd64 #1 SMP Debian 5.8.14-1 (2020-10-10) x86_64 GNU/Linux
```

La información que aparece corresponde con:

- Nombre del núcleo: «Linux»
- Nombre del nodo: «storm»
- Versión del núcleo: «5.8.0-3»
- Fecha y hora en que fue compilado el núcleo
- Tipo de arquitectura: «amd64»
- Nombre del sistema operativo: «GNU/Linux»

3.2. Interfaces de red

La forma más sencilla de ver con qué interfaces de red cuenta el computador es el comando `ip addr`¹ (del paquete `iproute2`).

```
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 UP qlen 1000
8     link/ether 00:22:34:6b:c5:47 brd ff:ff:ff:ff:ff:ff
9     inet 161.67.101.12/24 brd 161.67.101.255 scope global eth0
10    inet6 fe80::225:32ee:cf53:c429/64 scope link
11    valid_lft forever preferred_lft forever
```

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

¹Se puede abreviar como `ip a`.

3.2.1. *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 indica en la **línea 4**: `scope host`.

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

La **línea 2** muestra otros datos interesantes:

- La interfaz de red está habilitada por parte del administrador: `UP`.
- Es de tipo bucle: `LOOPBACK`.
- Tiene una MTU de 16436 bytes.

3.2.2. `eth0`

La interfaz `eth0` identifica a una Network Interface Controller (NIC) de tipo Ethernet (**línea 7**). 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 7**).
- Tiene una MTU de 1500 bytes (inherente a Ethernet).

Una segunda NIC se llamaría `eth1`, y del mismo modo las NIC WiFi tienen nombres como `wlan0`, aunque dependiendo de la versión del SO y su configuración puede tener otros nombres como `eno1`, `enp0s32d5`, `wlp1s0`, etc.

3.3. Encaminamiento básico

Tanto los encaminadores (*routers*) como los computadores también tienen una **tabla de rutas**. Esa tabla indica qué debe hacer con los paquetes que *salen* del computador. Para ver la tabla de rutas puedes utilizar el comando `ip route`:

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

Esta tabla de rutas tiene únicamente dos filas. La primera indica que todo el tráfico dirigido a la red 161.67.101.0/24 debe enviarse directamente al destino (scope link). Esto se llama *entrega directa* y ocurre cuando el destino es un vecino, es decir, el computador destino y el nuestro comparten el mismo enlace, en este caso una LAN Ethernet.

La segunda fila dice que el tráfico dirigido a cualquier otra parte (*default*) debe enviarse al encaminador (via 161.67.101.1) a través de la interfaz `eth0`. Éste es el encaminador por defecto, que en el caso de ser una LAN se llama también encaminador local, pasarela (*gateway*) o puerta de enlace. Ese encaminador repetirá el proceso y así sucesivamente hasta que el paquete llegue a su destino. Esto se llama *entrega indirecta* porque nuestro computador no lo entrega al destino sino que delega la tarea a un tercero (el router).

Esta tabla de rutas es la habitual en cualquier computador, móvil, consola de videojuegos, etc. que no trabaje como router.

3.4. Conectividad, por capa

Tomando como guía el modelo de referencia OSI se puede considerar la conectividad a distintos niveles, 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. Por ejemplo, se puede utilizar un servidor web y un navegador, 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 El cambio era incorrecto, estaba bien la solución y la calificación original. Saludosconectar un cliente desde otro computador. Se trata de una comunicación extremo a extremo, es decir, dos procesos 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 un par de computadores para intercambiar paquetes IP a través de

una 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. En este caso las herramientas pueden ser muy variadas puesto que dependen del protocolo y tecnología concreta: Ethernet, X.25, frame relay, etc.

3.4.1. 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 3.1).

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.

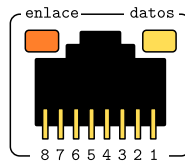


FIGURA 3.1: Conector RJ-45 hembra de un equipo Ethernet

Aparte de la comprobación visual anterior, es posible utilizar software que interroge 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)² de 100 Mbps (línea 12) y el enlace de datos está correctamente establecido (línea 22).

E 3.01 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?

E 3.02 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?.

3.4.2. 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.

3.4.2.1. 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 [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.

²Sólo se puede establecer un enlace *full duplex* si se trata de Ethernet conmutada, es decir, el PC está conectado a un conmutador (*switch*).

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 Round Trip Time (RTT) (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 3.03 Ejecuta ping hacia «localhost» y hacia la IP de tu propio equipo. ¿Hay alguna diferencia? ¿Por qué?

E 3.04 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 3.05 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 3.06 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 3.07 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?

3.4.2.2. 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 3.08 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?

3.4.2.3. 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.

3.4.2.4. **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

```

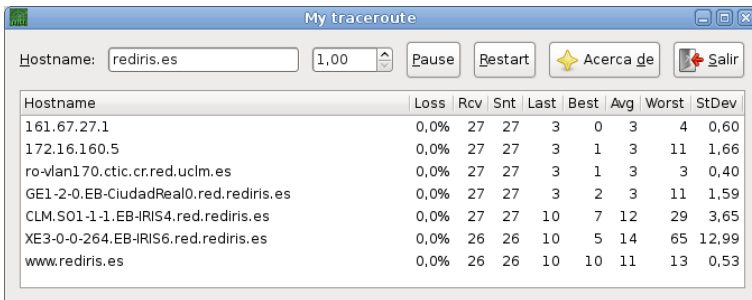
El primer resultado es para un TTL=1 (línea 3) y corresponde a la puerta de enlace (161.67.101.1). Al lado aparecen Los tiempos RTT para cada uno de los tres paquetes enviados. Para cada encaminador aparece primero su nombre simbólico (si tiene) y después su dirección IP. Como se puede apreciar

la ruta completa requiere 7 saltos, siendo el último el destino solicitado.

E 3.09 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?

3.4.2.5. mtr

`mtr` es muy similar a `traceroute` pero puede utilizarse con interfaz gráfica, tal como se muestra en la Figura 3.2.



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 3.2: Aspecto del programa `mtr`

3.4.3. 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).

3.4.3.1. 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.

E3.10 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.

E3.11 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.

3.4.4. 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, Post Office Protocol (POP3) o Internet Relay Chat (IRC).

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.

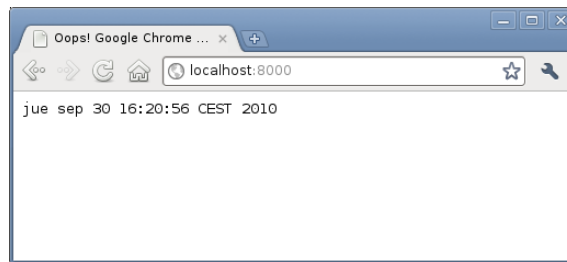


FIGURA 3.3: *Chromium* muestra la «página web» que devuelve `ncat`

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 3.3.

Pila de protocolos TCP/IP

David Villa

La figura 4.1 muestra la pila TCP/IP con sus protocolos más importantes. De la capa «host a red», sólo Address Resolution Protocol (ARP) se puede considerar TCP/IP, y por su función –como veremos– hay algunos autores que lo consideran un protocolo de la capa de inter-red.

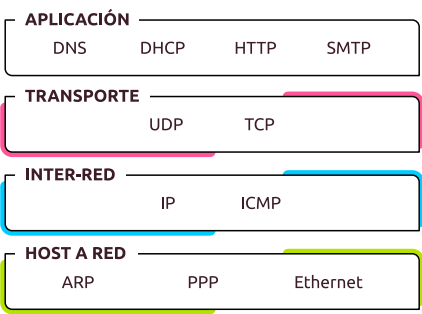


FIGURA 4.1: Pila de protocolos TCP/IP

En este capítulo...

Se introduce aquí 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 mientras que los de aplicación suelen estar disponibles en forma de librerías, o en las propias aplicaciones si son muy específicos. 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 Application-Specific Integrated Circuit (ASIC) para conseguir mayor productividad y menor consumo.

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.

4.1. Protocolos esenciales

Esta sección describe brevemente los protocolos más importantes de TCP/IP además de algunos otros de uso común que sirven para completar el contexto de su funcionamiento (principalmente los protocolos de enlace).

4.1.1. Ethernet

La capa de enlace de datos se encarga de llevar mensajes desde un computador o equipo de comunicaciones hasta sus vecinos¹.



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 dos tipos de medios:

punto a punto que permite enviar mensajes desde un vecino a otro. Un ejemplo sencillo de esto puede ser un enlace de infrarrojos.

de difusión (*broadcasting*) que permiten enviar el mismo mensaje a un conjunto determinado de vecinos. Esto es posible un medio físico compartido al que todos ellos tienen acceso. El aire en una comunicación WiFi sería un ejemplo de medio físico compartido.

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

Las tecnologías LAN más utilizadas con diferencia en la actualidad son Ethernet IEEE 802² y WiFi³ que, salvando las distancias, podríamos ver como una especie de «ethernet inalámbrica».

¹dispositivos cercanos que pueden ser alcanzados sin ayuda de intermediarios

²<http://wikipedia.org/wiki/Ethernet>

³<http://wikipedia.org/wiki/Wifi>

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 aparece en la figura 4.2.

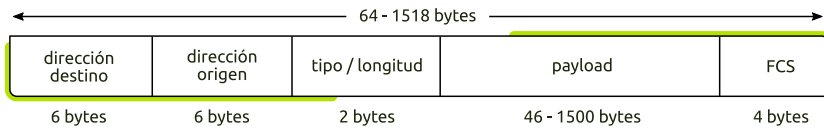


FIGURA 4.2: Formato de la trama Ethernet

Como muestra la figura 4.2, la trama Ethernet incluye:

- Las direcciones MAC de la tarjeta emisora y receptora,
- Puede indicar el tamaño de campo *payload* o bien el tipo del *payload* que suele ser un código de protocolo,
- La carga útil (*payload* que es de tamaño variable, y
- El Frame Check Sequence (FCS), una especie de suma de comprobación que permite al receptor comprobar si la trama ha sufrido alguna alteración.

Obviando todos los detalles del funcionamiento de Ethernet, el servicio que ofrece a la capa de red es bastante simple: Lleva la trama hasta la NIC con la dirección destino que aparece en la cabecera, y por tanto, hasta el computador asociado.

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 la misma dirección. Este tipo de identificación se denomina «identificador único global» o Universally Unique Identifier (UUID). En concreto la dirección MAC Ethernet 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 (3 bytes) Identifica al fabricante de la tarjeta. Este prefijo debe solicitarse a la Internet Assigned Numbers Authority (IANA). La lista de

todos los Organizationally Unique Identifier (OUI) asignados actualmente se puede consultar en su web⁴.

NIC (3 bytes) Es un número de serie que el fabricante garantiza que es único en su producción.

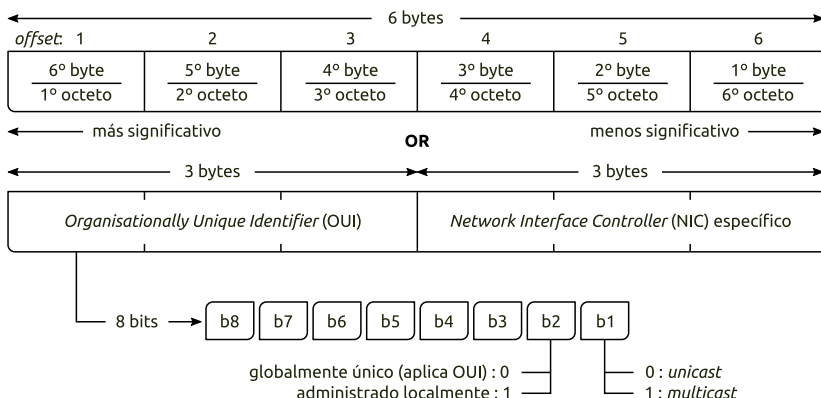


FIGURA 4.3: Formato de la dirección MAC Ethernet [Fuente:Wikimedia Commons]

Podemos ver la trama Ethernet como un contenedor en el que colocar 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, el receptor lo descartará 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.

Aparte de la entrega para un único destinatario (*unicast*), Ethernet proporciona 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.

⁴<http://www.iana.org/assignments/ethernet-numbers>

4.1.2. IP

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 los dispositivos conectados a una red comparten el mismo prefijo. Las redes grandes (con muchos computadores tienen prefijos cortos mientras que las redes pequeñas tienen prefijos largos.

Este direccionamiento jerárquico es lo que permite que los routers puedan hacer su trabajo de forma eficiente ya que las tablas de rutas no necesitan incluir las direcciones de todos los posibles destinos, sólo los prefijos comunes. la figura 4.4 muestra el formato de las direcciones IP.

FIGURA 4.4: Formato de las direcciones IP

IP es un protocolo de red, es decir, es capaz de llevar un paquete IP de cualquier punto de Internet⁵ a cualquier otro. A esto se llama entrega «extremo a extremo» (*end-to-end*). Pero IP es un protocolo *best-effort*, es decir, que hará lo posible por llevar el paquete a su destino, pero sin ninguna garantía.

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 Internet Group Management Protocol (IGMP) [Fen97].

El formato del paquete IP [Pos81b], que aparece en la figura 4.5 es también bastante simple.

A continuación se describe brevemente el significado de cada campo:

Version

Versión del protocolo. Siempre contiene el valor 4.

⁵O cualquier otra inter-red que use tecnología TCP/IP

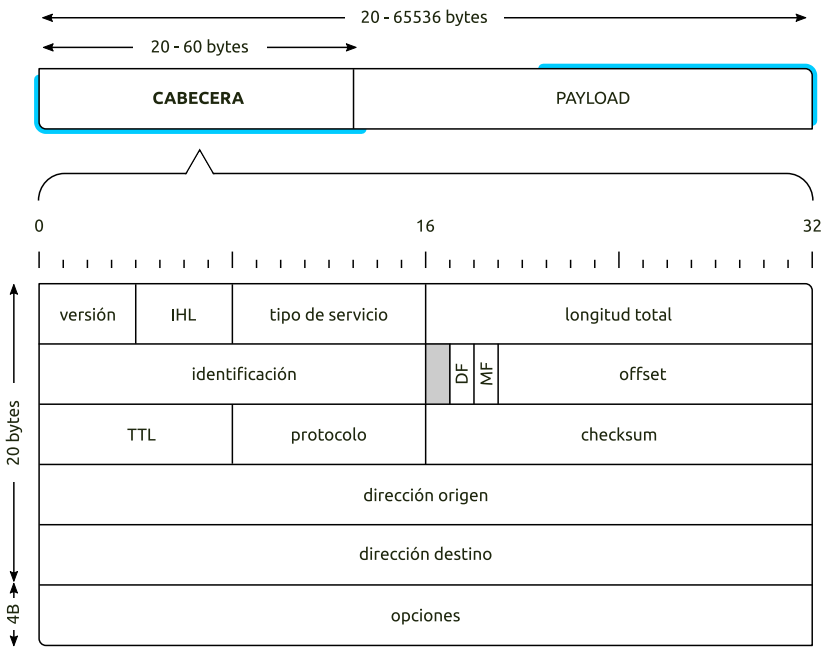


FIGURA 4.5: Formato del paquete IP

IHL Internet Header Length, es decir, el tamaño de la cabecera, pero expresado en palabras de 4 bytes. Eso significa que el 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.

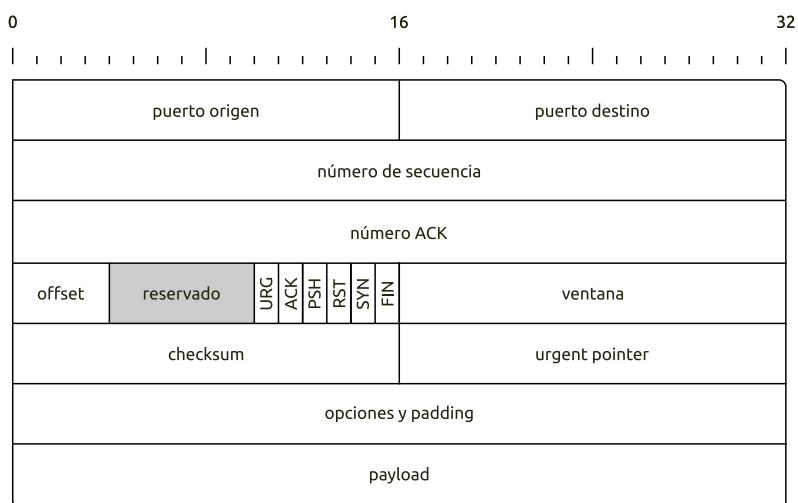


FIGURA 4.6: Formato del segmento TCP

4.1.3. UDP

4.1.4. TCP

4.2. Ejercicios

4.2.1. Enlace

- E 4.01** 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 4.02** Sobre el ejercicio anterior, implementar un algoritmo de detección de errores usando CRC32.
- E 4.03** Sobre el ejercicio **E 4.01** , implementar un algoritmo de corrección de errores Hamming.
- E 4.04** 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 4.05 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.

4.2.2. Red

E 4.06 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 4.07 Escriba un programa que a partir de una captura de paquetes IP, calcule el checksum y compruebe si corresponde con el capturado.

E 4.08 Escriba un programa que acepte por línea de comandos una IP y una máscara en notación CIDR, por ejemplo "161.67.27.18/26".^e 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 4.09 Escriba un programa que acepte una dirección IP por línea de comandos e informe si esa dirección es pública, privada o multicast.

E 4.10 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.

4.2.3. Transporte

E 4.11 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 4.12** 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.uc1m.es 20-2000.
```

- E 4.13** Servidor Echo concurrente (TCP o UDP). Ejemplo de invocación:

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

- E 4.14** Servidor Daytime concurrente (TCP o UDP). Ejemplo de invocación:

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

- E 4.15** Servidor Time concurrente (TCP o UDP). Ejemplo de invocación:

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

- E 4.16** Servidor Discard concurrente (TCP o UDP).

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

- E 4.17** Cliente que explote la vulnerabilidad SYN Flood.

- E 4.18** Programa que implemente ARP spoofing. El programa debe aceptar por línea de parámetros la IP del objetivo y del host que le suplanta.

Capítulo 5

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 tal como aparecen en la red, puede ser determinante para localizar el problema. 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.

5.1. tshark

Uno de esos analizadores de protocolos es tshark. Es la versión para línea de comando del programa wireshark (que también veremos en este manual) y su uso es similar a tcpdump. La forma más sencilla de entender lo que hace es verlo en funcionamiento. El listado 5.1 muestra tshark capturando el tráfico ICMP correspondiente al comando:

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

LISTADO 5.1: 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=64
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 se 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 (opción `-r`) y, por supuesto, también crear este tipo de ficheros de captura para un análisis posterior (opción `-w`).

5.1.1. Advertencia de seguridad

Quizá haya advertido en la línea 1 del listado 5.1 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 (por defecto `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 grupo específico. Consulte <http://wiki.wireshark.org/CaptureSetup/CapturePrivileges> para más detalles sobre esta última opción.

5.1.2. 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.

5.1.3. 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 5.1 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	tramas Ethernet no broadcast

CUADRO 5.1: tshark: ejemplos de filtros de captura

5.1.4. 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 5.2). En el listado 5.2 hay 4 secciones bien delimitadas:

- 1. *Frame 1 (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 § 4.1.1).
- 3. *Internet Protocol (línea 27)* muestra los detalles del paquete IP (vea § 4.1.2).
- 4. *Internet Control Message (línea 49)* contiene la información de la carga útil del paquete IP, que es 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

LISTADO 5.2: 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 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: 8c110d000000000101112131415161718191a1b1c1d1e1f...
65         [Length: 48]

```

LISTADO 5.3: 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
```

(`-T fields`) y la lista con los campos concretos, que pueden depender de las condiciones de filtrando. El ejemplo del listado 5.3 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 5.19 En el listado 5.3 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.

5.1.5. 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 5.2 muestra algunos ejemplos representativos similares a <http://wiki.wireshark.org/DisplayFilters>.

Filtro	Significado
<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 5.2: 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 elaboradas⁴.

chuleta

Filtros de visualización³

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

5.1.6. Generación de estadísticas

tshark genera una gran variedad de estadísticas útiles⁵. 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>

⁴http://www.wireshark.org/docs/wsug_html_chunked/ChWorkBuildDisplayFilterSection.html

⁵Puede ver todas las estadísticas disponibles con `tshark -z help`

5.1.6.1. *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
```

5.1.6.2. *icmp,srt[,filter]*

Calcula las estadísticas Service Response Time (SRT) 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
=====
```

5.1.6.3. *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
```

=====

5.1.6.4. *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 |
|-----|
| |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 |
|-----|
```

5.1.6.5. *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 |
|-----|
| |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	
=====										

5.1.6.6. 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
```

5.1.6.7. 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	192.168.2.49:38216 <-> 73.233.104.123:80
7	192.168.2.49:58949 <-> 92.184.220.111:80
8	192.168.2.49:58948 <-> 92.184.220.111:80
9	192.168.2.49:58947 <-> 92.184.220.111:80
10	192.168.2.49:58946 <-> 92.184.220.111:80
11	192.168.2.49:56569 <-> 92.100.127.144:80
12	192.168.2.49:38838 <-> 95.172.94.11:80
13	192.168.2.49:38837 <-> 95.172.94.11:80
14	=====

E 5.20 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?

5.2. Respuestas

E 5.19 FIXME

- E 5.20** 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.

6.1. En este capítulo...

Al finalizar este capítulo el lector será capaz de:

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

6.2. 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.

6.3. 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 qué red está conectado y qué tráfico puede monitorizar. En la Figura 6.1 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.

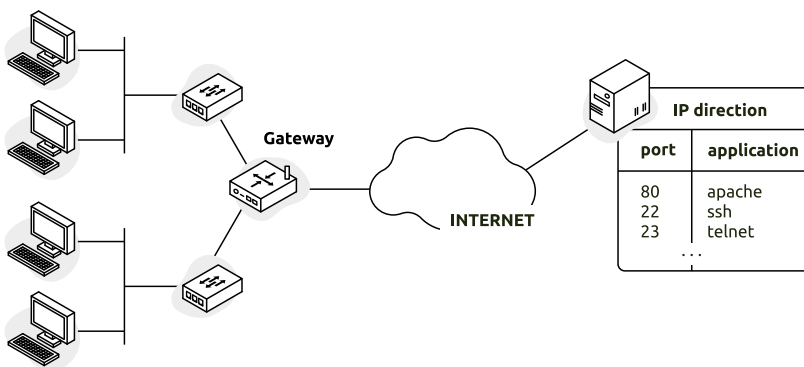


FIGURA 6.1: Situación inicial

E 6.21 Suponiendo que estamos en host 1 ¿Qué tráfico podríamos ser capaces de inspeccionar?

6.4. 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 6.2 (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 mas atajos para las opciones más comunes.

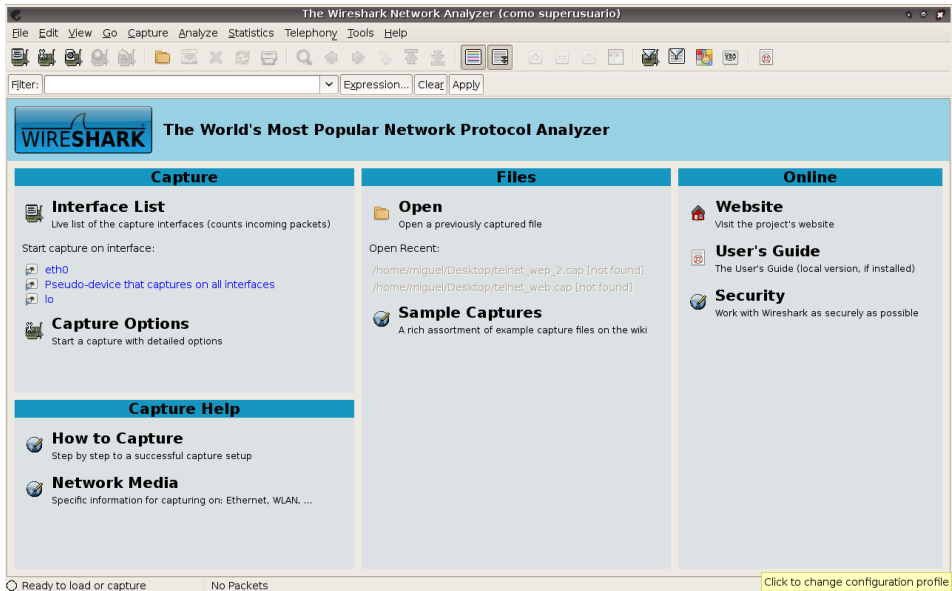


FIGURA 6.2: Ventana inicial

6.4.1. 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 6.3. 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).

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

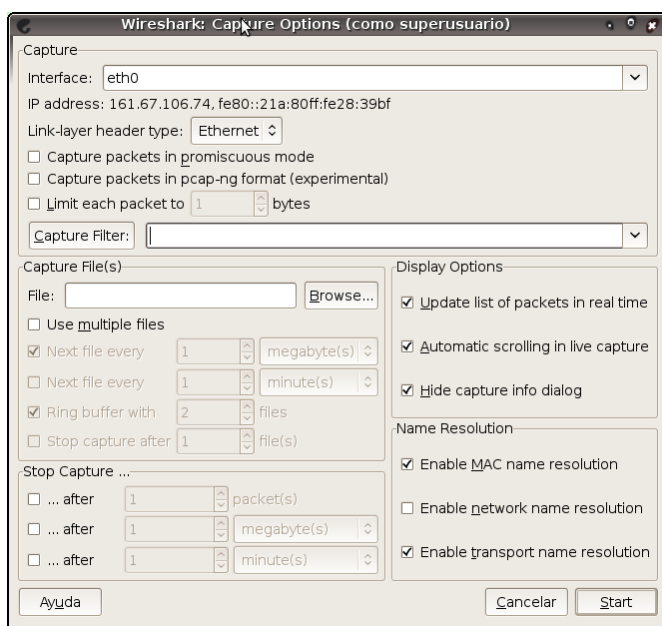


FIGURA 6.3: Opciones de captura

que te interesen para la captura.

E 6.22 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 qué aparece? ¿Qué protocolo utiliza ping?

6.4.2. 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 6.4. 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.

En la parte en la que se muestran los paquetes podemos apreciar distintas

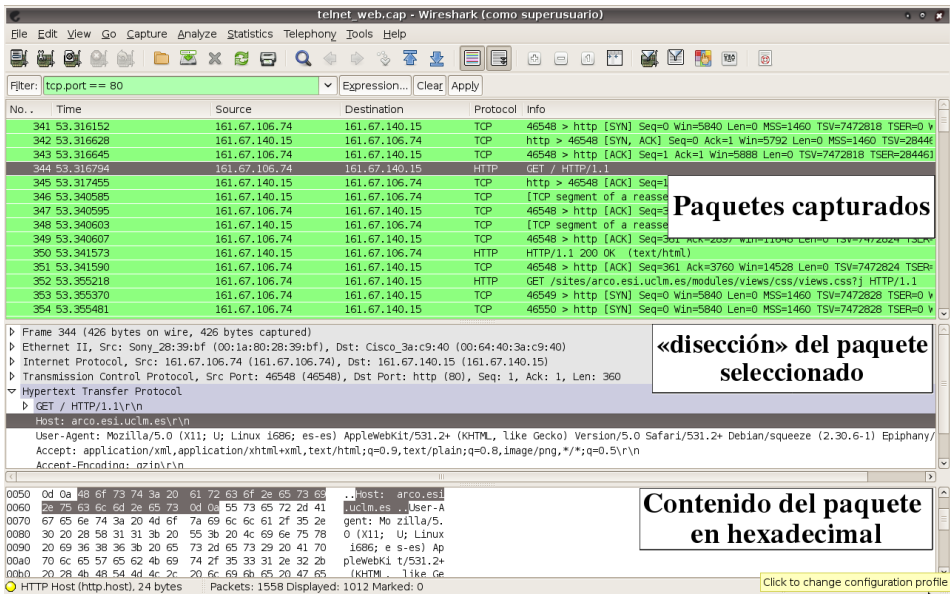


FIGURA 6.4: Ventana principal

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.

E6.23 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 6.24 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?

6.4.3. 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 direc-

ción 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 6.25** Escriba y pruebe un filtro que permita recuperar únicamente el tráfico de una comunicación TELNET[PR83].
- E 6.26** Escriba y pruebe un filtro que permita recuperar únicamente los paquetes que contienen HTML[DVG96] que interpreta el navegador web. (Pista: slice en filtros)

6.4.4. 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 6.5. 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.

- E 6.27** 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 6.28** Utilizando un archivo de capturas que contenga una conexión TELNET, o generando una tú mismo, realice un seguimiento del stream TELNET. ¿Qué conclusiones obtiene?

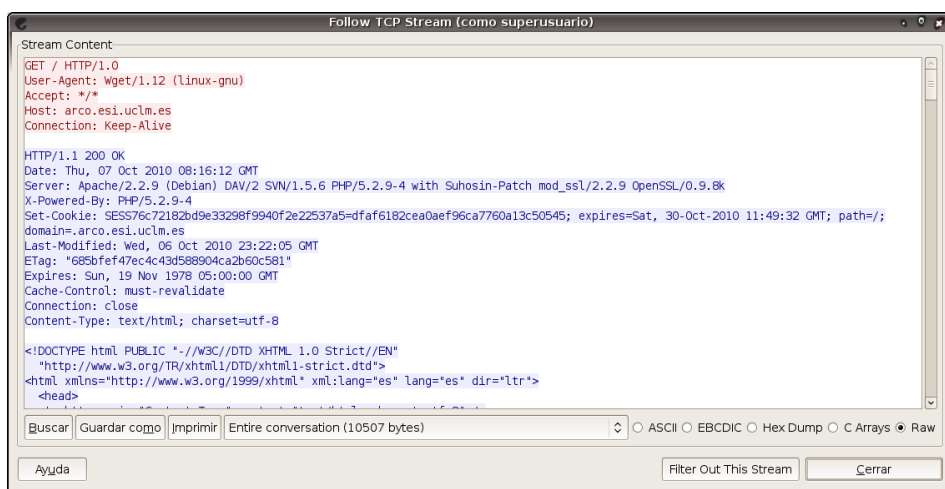


FIGURA 6.5: Wireshark: siguiendo un stream

Capítulo 7

Encaminamiento

Soledad Escolar
David Villa

El encaminamiento de paquetes es uno de los objetivos esenciales de la capa de **inter-red**. Y este nombre merece la pena ser comentado. Es muy común que la bibliografía denomine a esta capa simplemente *de red*¹, sin embargo, *inter-red* (así en minúscula) es un término mucho más preciso. Veamos el motivo, y cómo se concreta en el caso de la tecnología Internet.

La capa de inter-red define:

- Un sistema de direccionamiento lógico universal (el direccionamiento IP) que se aplica a todos los computadores y dispositivos de interconexión, independientemente de la tecnología de red que utilicen sus NIC.
- Un formato común de mensaje (el paquete IP) que deben emplear todos los participantes.
- Un medio para llevar los mensajes de un lugar a otro en función de esas direcciones.

El objetivo por tanto no es mover paquetes dentro de una red. De eso se encargan los protocolos de enlace (como el caso de una LAN Ethernet). El protocolo IP y la capa de inter-red en general se encargan de mover paquetes *entre redes*. Por ese motivo se suele llamar red de redes a Internet, aunque es igualmente cierto si nos referimos a cualquier conjunto de redes interconectadas.

Como resumen de lo anterior podemos citar la RFC 791, que dice:

“The purpose of Internet Protocol is to move datagrams through an interconnected set of networks”.

El dispositivo encargado de *mover paquetes entre redes* es el **router** (habitualmente llamado *encaminador* en la bibliografía en castellano). También

¹De hecho es el nombre que le da el modelo OSI.

se le llama **pasarela** (*gateway*), aunque este segundo término se suele utilizar a veces para referirse específicamente a routers que interconectan redes heterogéneas (es decir, con tecnologías de enlace a priori incompatibles).

7.1. En este capítulo...

Al terminar este capítulo, el lector será capaz de comprender:

- Cómo se mueven los paquetes IP a través de una inter-red.
- El significado de la información que contiene una tabla de rutas.
- Cómo un router colabora con otros para llevar los paquetes hasta su destino.
- La necesidad, características y funcionamiento de los algoritmos y protocolos de encaminamiento dinámico.
- Los mecanismos de encaminamiento multicast.

7.2. Almacenamiento y reenvío

Un router es todo dispositivo que dispone de conectividad con más de una red y tiene la capacidad de aceptar paquetes *dirigidos a terceros* y reenviarlos (del inglés *forwarding*) para colaborar en la tarea de hacerlos llegar a su destino. La diferencia clave con un nodo convencional (como puede ser un PC), es que éste descartará automáticamente cualquier paquete cuya dirección destino no corresponda con la de la interfaz de red (NIC) por la que llega.

En particular, un router IP recibe, almacena y procesa los paquetes que van llegando por cualquiera de sus interfaces, independientemente de la tecnología de enlace. El router procesa los paquetes como unidades independientes desprovistas de la encapsulación, normalmente una trama, que se empleó para hacerlos llegar hasta él.

Por esta forma de trabajar, se dice que el router es un elemento de *almacenamiento y reenvío* (*store and forward*). Veamos con algo más de detalle el proceso que realiza sobre cada paquete entrante:

- Una vez recibido por la interfaz el paquete se almacena en memoria (normalmente en una cola) y esperará allí hasta que el router disponga de capacidad de cómputo.
- Calcula su *checksum* y comprueba que coincida con el que aparece en la cabecera². Si esta comprobación falla, el paquete es descartado.
- Comprueba el campo TTL de la cabecera³ y si es 0, descarta el pa-

²Esto solo aplica a IPv4, dado que en IPv6 no existe *checksum*.

³O su equivalente *hop-count* en IPv6.

quete y envía a la dirección origen un mensaje ICMP de tipo *tiempo excedido* (*time exceeded*). Si el valor de TTL es mayor que 0, lo decremента en 1.



Las expresiones Time To Live y *time exceeded* son una reminiscencia de la Internet primigenia en la que los routers tardaban aproximadamente un segundo en procesar cada paquete. Hoy en día los routers pueden procesar millones de paquetes por segundo y el significado práctico del campo TTL es el número de saltos que aún puede dar el paquete.

- Determina la interfaz de salida en función exclusivamente de la dirección destino del paquete y del contenido de la tabla de rutas (*routing table*). Si la tabla de rutas no contiene información sobre cómo proceder, el paquete se descarta y el router envía a la dirección origen un mensaje ICMP de tipo *destino inalcanzable* (*destination unreachable*).
- El paquete se coloca en la cola correspondiente a la interfaz de salida elegida.
- Cuando el medio físico asociado a la interfaz está libre, la NIC encapsula el paquete IP con el formato de trama correspondiente y lo transforma en una señal hasta alcanzar el próximo dispositivo. Cuando esta interfaz de salida está conectada a un enlace de difusión (p.ej: una LAN Ethernet), el router debe determinar también (por medio de la tabla de rutas) la dirección física destino de la trama.

7.2.1. Entrega directa e indirecta

Cuando el router realiza el proceso de reenvío se dan principalmente dos posibilidades, que se conocen como dos tipos de entrega:

Entrega directa Ocurre cuando el router es un vecino (*neighbor*) del computador destino, es decir, cuando una de sus interfaces del router y el computador cuya dirección IP aparece en el campo *dirección destino* del paquete, están conectados a la misma red.

En este caso, el router entregará el paquete directamente al destino (como ocurre siempre entre vecinos).

Entrega indirecta Ocurre cuando el *siguiente dispositivo*, al que el router envía el paquete, no es el destino, sino otro router. Se entiende que se lo entrega *indirectamente*.

7.3. Tabla de rutas

Si asumimos que «ruta» es un «itinerario o camino para un viaje», lo primero que debemos entender es que, a pesar de su nombre más común, la *tabla de rutas* no contiene rutas. Probablemente una mejor traducción de *routing table* sería *tabla de encaminamiento*. Aunque puede aparecer diversa información, toda tabla de rutas contiene esencialmente 4 columnas:

Destino (*dst*)

Una dirección IP que identifica normalmente una red ⁴ a la que el router enviar paquetes.

Máscara (*mask*)

Es la máscara del valor indicado en la columna anterior. Es decir, técnicamente forma parte del campo *destino*, ya que no es posible determinar una red en *classless addressing* (lo normal desde hace muchos años) si no se especifica una máscara. En algunas ocasiones la columna *destino* aparece especificado en formato Classless Interdomain Routing (CIDR) (p.ej: 120.10.12.0/24) de modo que no es necesaria la columna *máscara*.

Siguiente salto (*next-hop*)

Una dirección IP de otro router.

Interfaz (*iface*)

El nombre de la interfaz de salida por la que se debe enviar el paquete.

Con esto podemos considerar que cada fila contiene esencialmente dos cosas: una condición (formada por *destino* y *máscara*) y una acción (formada por *siguiente salto* e *interfaz*). La condición se evalúa tomando como entrada la dirección destino del paquete. Si se cumple, la acción dice dónde enviar el paquete.

Para entender perfectamente cómo se procesa la tabla de rutas debemos tener presente dos cuestiones:

- La tabla se procesa siempre desde el principio hacia el final. Es decir, el orden de las filas *importa*.
- Si el router puede aplicar lo especificado en una fila, las siguientes ya no se procesarán.

Como se ha indicado anteriormente, si el router procesa la tabla completa y ninguna de las filas es aplicable, el paquete será descartado. Sin embargo, existe una excepción a este caso: la tabla de rutas puede incluir una única fila para especificar un *router por defecto* (*default router*). Esta fila especial (que por convenio se coloca la última) contiene la dirección IP del router al

⁴Excepcionalmente también puede ser un host

que se enviará el paquete si ninguna de las otras filas es aplicable.

Veamos un fragmento de pseudocódigo que ilustra la forma en el que router procesa la tabla de rutas cada vez que evalúa un nuevo paquete:

```
dst = datagram.destination
for row in routing_table:
    n = dst & row.mask
    if n == row.destination:
        forward(datagram, row.next_hop)
        return

if default_router:
    forward(datagram, default_router)
else:
    send_icmp(destination_unreachable)
```

Es decir, para cada fila, se hace una operación AND a nivel de bits entre la dirección destino del paquete y la *máscara*. Si el resultado es igual a la columna *destino* la condición se cumple. En ese caso se envía el paquete conforme al valor de las columnas *siguiente salto* e *interfaz*.

7.3.1. Un ejemplo

La tabla 7.1 muestra la tabla de rutas de un supuesto router R0 que podría ser un ejemplo perfectamente realista (la columna «n» se ha añadido simplemente para referencias las filas más fácilmente). Veamos con detalle la información que ofrece:

- La tabla corresponde a un router que dispone de 3 interfaces (e0, e1 y s0). Aunque cada fabricante y sistema operativo sigue su propia nomenclatura para nombrar la interfaces, las Ethernet suelen llamar «eX» o «ethX» y las interfaces serie «sX» o bien con el protocolo de enlace que utilizan, como «pppX», siendo «X» un número empezando en 0.
- Las filas 1 y 2 corresponden a entrega directa. Se reconocen porque el valor para el campo *next-hop* es la dirección IP nula: 0.0.0.0. Esto significa que el router es vecino de la red 30.0.0.0/24 por medio de su interfaz e0 y también lo es de la red 40.0.0.0/24 a través de su interfaz e1. En ambos casos, el propio router debe encargarse de entregar el paquete directamente al computador destino.
- La fila 3 especifica un entrega indirecta. Indica que todo paquete dirigido a la red 130.10.20.0/24 debe reenviarse al router 30.0.0.2.

- La fila 4 también indica una entrega indirecta. Todo paquete dirigido a la red 120.20.30/17 debe reenviarse al encamiandor 40.0.0.3.
- Por último, la fila 5 establece el router por defecto, que como hemos visto, significa que cualquier paquete que no haya encajado con la «condición» establecida en cada una de las otras filas debe enviarse al router 50.1.1.2.

n	dst	mask	next hop	iface
1	30.0.0.0	255.255.255.0	0.0.0.0	e0
2	40.0.0.0	255.255.255.0	0.0.0.0	e1
3	130.10.20.0	255.255.255.0	30.0.0.2	e0
4	210.20.30.0	255.255.64.0	40.0.0.3	e1
5	0.0.0.0	0.0.0.0	50.1.1.2	s0

CUADRO 7.1: Ejemplo: Tabla de rutas de router R0

Es importante señalar que la tabla de rutas no nos dice nada sobre dónde están las redes 130.10.20.0/24 y 210.20.30.0/17 o a qué distancia están (en número de saltos). Simplemente nos dice que para llegar a ellas, debemos delegar la entrega al router indicado. Con esta información podemos deducir parte de la topología de la inter-red en la que se encuentra nuestro router R0 (ver figura 7.1).

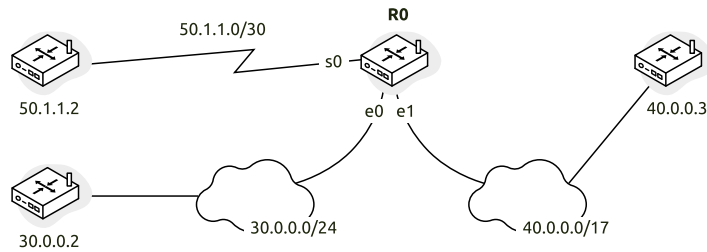


FIGURA 7.1: Topología parcial que se deduce de la tabla de rutas 7.1

7.3.2. Tabla de rutas básica

Cualquier computador o dispositivo conectado a Internet tiene una tabla de rutas (incluido un teléfono móvil) aunque no por tener una tabla de rutas es un router. Recuerda que para ser considerado un router, el dispositivo debe aceptar y reenviar tráfico que no va dirigido a él. Esa tabla de rutas «mínima» le indica al sistema cómo enviar tráfico a sus vecinos y qué hacer con el tráfico que debe ir fuera del enlace (la LAN). Una tabla de rutas típica en un computador doméstico con conexión WiFi (que suele estar detrás de

un router NAT) es similar a la siguiente:

dst	mask	next hop	iface
192.168.0.0	255.255.255.0	0.0.0.0	wlan0
0.0.0.0	0.0.0.0	192.168.0.1	wlan0

Esta tabla dice justamente lo que el sistema necesita: la fila 1 dice quienes son los vecinos (los de la red 192.168.0.0/24) y que a ellos lógicamente debe hacer entrega directa (*next-hop* = 0.0.0.0). La segunda fila indica el router por defecto, que corresponderá con el *router* ADSL o fibra de la red domestica; y a través de éste, el sistema enviará tráfico hacia Internet. Los sistemas operativos de escritorio suelen denominar a este router *pasarela de enlace*.

En un sistema GNU/Linux de escritorio, puedes ver la tabla de rutas con el comando `ip route`, con el que obtendrás algo similar:

```
$ ip route
default via 192.168.0.1 dev eth0
192.168.0.0/24 dev eth0 proto kernel scope link src 192.168.0.18 metric 100
```

Aunque el formato es bastante espartano, obviando los datos adicionales, podemos extraer la siguiente información:

dest/mask	next hop	iface
default	192.168.0.1	eth0
192.168.0.0/24	scope link	eth0

Que corresponde esencialmente a esa tabla «mínima» a la que nos referíamos anteriormente. Las diferencias más significativas son:

- Las columnas *destino* y *máscara* aparecen como un solo dato en notación CIDR.
- En lugar de 0.0.0.0 como destino, muestra *default*.
- En lugar de 0.0.0.0 como siguiente salto muestra *scope link*, que significa que esos dispositivos (192.168.0.0/24) son accesible en el «ámbito del enlace local» (son los vecinos).

E 7.01 Ejecuta el comando `ip route` en tu sistema GNU/Linux e interpreta el resultado, tal como hemos visto.

E 7.02 Si dispones de una aplicación de máquina virtual, arranca una máquina y comprueba si afecta, y cómo a la tabla de rutas de computador.

E 7.03 Si dispones de un cliente de VPN, establece una conexión y comprueba si afecta, y cómo a la tabla de rutas de computador.

7.4. Estático versus dinámico

Una tabla de rutas puede ser estática o dinámica:

Tabla estática

la información en la tabla se suministra manualmente por el administrador de la red.

Tabla dinámica

la información en la tabla se modifica automáticamente tan pronto como se detecta un cambio en algún lugar de Internet, por ejemplo, un router o enlace que deja de estar disponible o una mejor ruta para alcanzar un destino en una subred.

Si nuestra red es muy pequeña, las tablas de encaminamiento pueden ser creadas y mantenidas por el administrador de red. Sin embargo, en grandes redes, donde los cambios se producen continuamente, mantener tablas estáticas es completamente inmanejable. Se necesita, por tanto, una forma de generar las tablas de rutas de forma dinámica y para ello, **protocolos de encaminamiento** que obtengan de la red la información necesaria, calculen las nuevas rutas, propaguen los cambios necesarios y mantengan las tablas de rutas actualizadas. Véamos qué es un protocolo de encaminamiento y ejemplos de los más actualmente utilizados.

Multicast

David Villa

La multidifusión (*multicast* en inglés) se refiere a los mecanismos que permiten entregar un mismo mensaje a un conjunto de destinatarios. Una forma obvia de conseguirlo sería que el emisor se encargara de enviar una copia del mensaje a cada uno de los destinatarios¹. Eso implica un inconveniente grave: Multiplica por n el tiempo y el ancho de banda necesarios para enviar el mensaje (siendo n el número de destinatarios, y provoca una latencia considerable en la recepción del mensaje entre el primero y el último de los destinatarios.

Las alternativas para conseguir un envío a múltiples destinos, requieren la colaboración de los routers y son esencialmente dos:

Inundación El emisor envía el paquete al primer router, indica de algún modo que se trata de un mensaje multicast y los routers se encargan de llevar el paquete a todas las redes. Este método es obviamente inviable en una red de medio o gran tamaño y supone una carga posiblemente mayor que la propuesta inicial.

Encaminamiento multicast En este caso los routers determinan por qué interfaces de salida deben enviar una copia del mensaje para que llegue a todos los destinatarios.

Evidentemente este último enfoque implica un consumo de recursos sensiblemente menor, ya que al menos en el caso ideal, no circulan por la red más copias de las necesarias, y además esas copias se harán en el último punto posible. El trabajo del algoritmo de encaminamiento multicast es determinar las rutas que seguirán estos paquetes.

Un aspecto importante es el modo en que se gestionan los destinatarios del envío. Aquí también hay varias posibilidades:

- El emisor coloca en la cabecera del mensaje las direcciones de los

¹De hecho, la mayoría ni siquiera considera que esto pueda llamarse multicast

destinatarios. Sería una solución muy flexible, pero ineficiente incluso para muy pocos destinatarios: por ejemplo, 100 destinatarios con direcciones lógicas de 4 bytes implicarían 400 bytes en la cabecera de cada mensaje.

- Grupos estáticos, que quedan determinados por el valor concreto de una parte de la dirección lógica. Esta solución resulta muy eficiente en términos de espacio en los mensajes y no requiere ninguna administración de los grupos, pero es totalmente inflexible ya que un host pertenece o no a un grupo en función de su dirección.
- Grupos dinámicos, en los que los hosts podrían solicitar entrar o salir en cualquier grupo en cualquier momento. Es una solución muy flexible, pero requiere un sistema complejo para gestionar estos grupos y permitir que los routers cuenten con información actualizada de los miembros de cada.

En los enfoques basados en grupos existen direcciones especiales *de grupo*. El emisor coloca una dirección de grupo (o simplemente *dirección multicast*) como dirección destino de los paquetes. Curiosamente eso implica que el emisor no sabe quiénes son los destinatarios.

8.1. IP multicast

El soporte de envío multicast en TCP/IP se basa en grupos y ofrece tanto grupos dinámicos como estáticos. La IANA reserva para grupos multicast todas las direcciones de clase D, es decir, aquellas cuyos primeros 4 bits son '1110' con máscara de 8 bits, lo que implica los bloques 224.0.0.0/8 a 239.0.0.0/8 (ver [AAMS01]).

Un protocolo específico llamado IGMP permite a los hosts solicitar la entrada o salida de cualquier grupo dinámico, permitiendo pertenecer a varios al mismo tiempo. Los grupos dinámicos existen mientras haya algún miembro activo.

Por la naturaleza de la distribución de los paquetes, solo es posible utilizar UDP como protocolo de transporte, por lo que en principio el soporte multicast en TCP/IP ofrece un servicio *best effort*.

8.2. Encaminamiento multicast

En encaminamiento unicast, las tablas de rutas contienen la información para llegar a todos los caminos conocidos (por el router) utilizando e principio la ruta óptima. Lógicamente el router unicast enviará el paquete por la interfaz correspondiente al único destinatario.

Un router multicast sin embargo, deberá determinar si existe algún miembro del grupo destino del paquete en cada uno de las rutas posibles (las filas de la tabla) y tendrá que hacer una copia por cada una de ellas. Si todos los miembros del grupo son alcanzables según lo indicado en una de las filas de la tabla, el router no hará ninguna copia, solo reenviará el paquete.

En cualquier caso, podemos comprobar que cada tabla de rutas contiene parte de un árbol cuya raíz son los posibles emisores y las hojas los posibles destinos, un árbol formado por todas las rutas óptimas. A este árbol se le denomina *árbol sumidero* (*sink tree*²). Lógicamente si en la red existen g grupos y cada grupo involucra a r routers, en la red habrá $g \times r$ árboles sumidero. Aunque los routers no son en principio emisores ni destinatarios (miembros) del grupo, los podemos considerar como participantes necesarios en la distribución de paquetes para ese grupo.

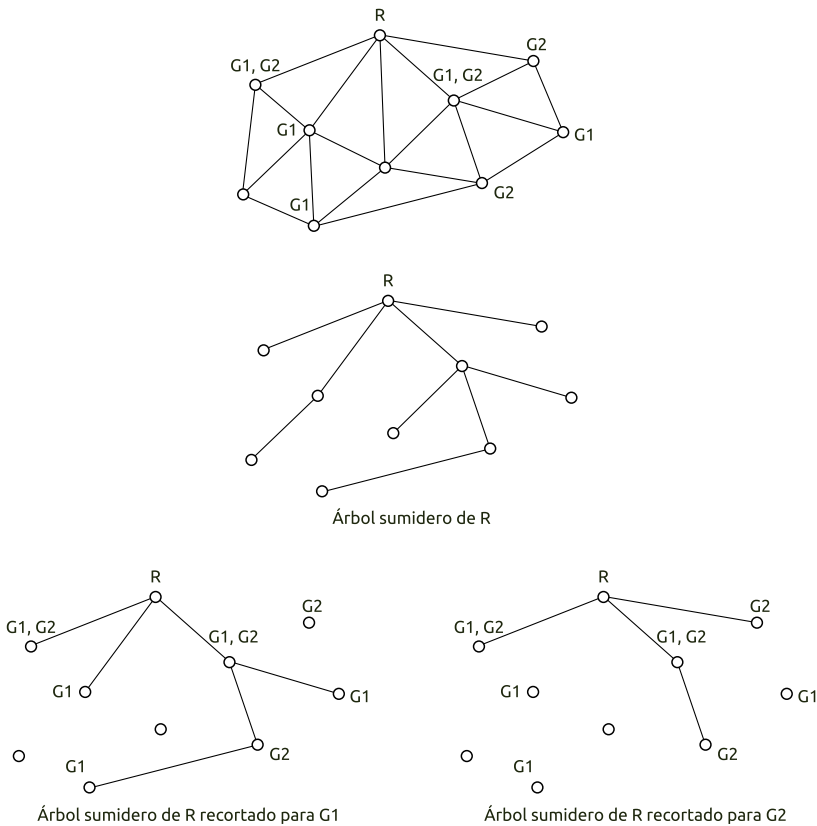


FIGURA 8.1: Árbol sumidero para R y árboles podados para los grupos G1 y G2

²También denominado *shortest path tree* por razones obvias

Obviamente no todos los routers participan en todos los grupos. Eso significa que a partir del mismo árbol sumidero (para un mismo origen) se pueden obtener los árboles correspondientes para cada grupo podando las rutas que no llevan a ningún miembro del grupo. En el diagrama superior de la figura 8.1 se muestra la topología de una subred. En el segundo diagrama se muestra el árbol sumidero para el router R y una métrica arbitraria. Los interiores serían por tanto los árboles podados o recortados (*pruned trees*) que R utilizaría para enviar paquetes a los miembros de los grupos G1 y G2.

8.2.1. Reenvío según ruta inversa

Para que la ruta que siguen los paquetes depende del árbol sumidero de cada emisor, los routers utilizan la dirección origen en lugar de la dirección destino (que identifica el grupo, pero no los destinos) para decidir por qué interfaces enviar las copias de los mensajes.

El reenvío según ruta inversa o RPF (Reverse Path Forwarding) es una técnica utilizada por los protocolos de encaminamiento multicast para evitar bucles de encaminamiento, es decir, para evitar que un mismo paquete (o una copia en este caso) vuelva al mismo router. La técnica consiste en reenviar únicamente los paquetes multicast que lleguen al router por la interfaz que coincide con el árbol sumidero para ese grupo, es decir, por la ruta que se usaría para enviar un paquete al origen (la ruta inversa). Si el paquete llega por otra interfaz, se descarta.

8.2.2. Árboles de núcleo

Para evitar calcular esos $g \times r$ árboles existe una aproximación subóptima en cuanto a las rutas, pero que permite simplificar los cálculos de rutas y los recursos de cómputo que dedican los routers. Se trata de calcular un único árbol por cada grupo (llamado *árbol de núcleo* (*core tree*) o *árbol compartido*). Se designa un único router (llamado *router núcleo* (*core router*) por cada grupo. Cuando un emisor desea enviar un paquete a un determinado grupo, realiza un envío unicast al router núcleo de dicho grupo y él se encargará de la distribución por medio del árbol de núcleo.

El nombre *router núcleo* se debe a que ese router debería estar en el centro (en términos de coste) de todos los miembros del grupo, de modo que el coste medio del envío de los mensajes sea aproximadamente el mismo sin importar quién sea el emisor.

Obviamente, este enfoque tiene el coste extra que implica determinar de forma descentralizada cuál es el router núcleo para cada grupo, algo que

puede ser especialmente complejo cuando se trata de grupos dinámicos.

Capítulo 9

Redes Privadas

David Villa
Soledad Escolar

Una red privada, tal como su nombre indica, es una red¹ de uso privado. Su objetivo principal suele ser compartir recursos dentro de una organización, por lo que todos los elementos físicos que conforman la red (computadores, dispositivos de interconexión, cableado, etc.) son propiedad de dicha organización, y por tanto, también es responsable de su diseño, implementación, gestión y explotación.

El concepto «red privada» no es equivalente a «red aislada». La red privada más común hoy día es la red WiFi doméstica que encontramos en la mayoría de los hogares y que, obviamente, está conectada a Internet a través de un mal-llamado router² y la red del ISP.

9.1. Líneas alquiladas

Una red privada puede ser también una inter-red privada, es decir, una colección de LAN interconectadas mediante routers, ya sea en una o varias localizaciones. Durante los años 70 a 90, cuando Internet no existía o más tarde su uso era limitado o demasiado caro, era habitual que muchas empresas tuvieran una red privada que conectaba las LAN de sus oficinas (por ejemplo, los concesionarios de una marca de automóviles). Para conectar estas oficinas entre sí, la organización podía instalar cableado propio o bien alquilar líneas a una compañía de telecomunicaciones. La primera opción solo era económicamente viable para distancias muy cortas (unos cientos de metros). Este planteamiento se muestra en la figura 9.1.

Cuando partiendo de una topología de este tipo, alguno de las oficinas obtiene acceso a Internet puede ofrecer conectividad a todas las demás oficinas.

¹Habitualmente una LAN

²Como veremos más adelante, el dispositivo que nos instala el Internet Service Provider (ISP) incluye muchas otras funciones.

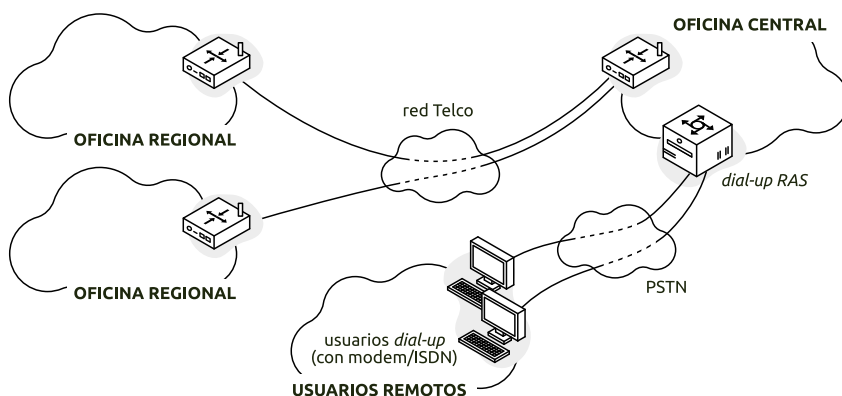


FIGURA 9.1: Red privada que utiliza líneas alquiladas

Esta imagen tiene una licencia Creative Commons Attribution-Share Alike 4.0. Disponible en <https://commons.wikimedia.org>

Este esquema (que se muestra en la figura 9.2) es lo que se denomina *red híbrida*.

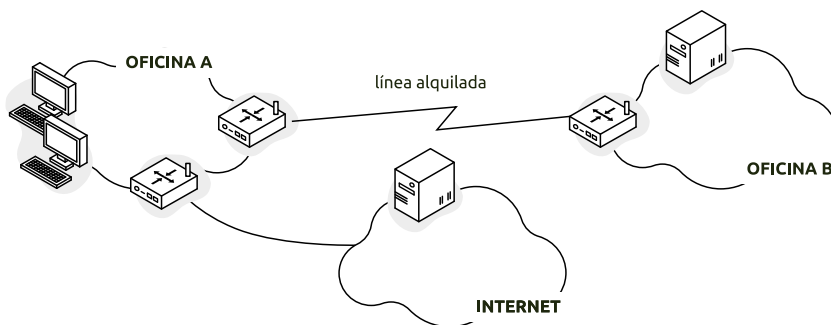


FIGURA 9.2: Red privada híbrida

9.2. Redes privadas TCP/IP

Precisamente por ser privada, la organización tiene plena libertad para elegir cualquier pila de protocolos disponible, o incluso implementar una tecnologías propia; algo común en redes de datos industriales.

En la práctica, utilizar la pila TCP/IP resulta muy conveniente por variedad y disponibilidad de software, soporte y sobre todo, como forma de simplificar la conexión de la red privada con otras redes.



Durante la primera mitad de la década de los 90, Novel NetWare Internetwork Packet Exchange (IPX) tuvo gran popularidad como protocolo para redes privadas, debido a varios factores: tarjetas NIC asequibles, los primeros videojuegos multijugador en red (p.ej: Quake), incorporación en Microsoft Windows, etc. Pocos años después, con la llegada de Internet, fue desbancado rápidamente por IP.

Una *intranet* es una red privada que utiliza tecnología TCP/IP, pero que únicamente es accesible para los dispositivos y usuarios de la organización. Sin embargo, el nombre *intranet* se utiliza hoy día, erróneamente, para identificar una o varias aplicaciones o servicios (normalmente web) destinados específicamente para el personal de una organización y que requieren autenticación específica.

Como caso particular de *intranet*, una *extranet* permite a ciertos usuarios y mediante acceso controlado, con un sistema de autenticación y autorización, acceso a los recursos de la red privada desde el exterior.

9.2.1. Direccionamiento privado

Como la red privada es responsabilidad exclusiva de la organización, ésta tiene la libertad de elegir de elegir cómo quiere plantear el direccionamiento de sus dispositivos. El diseñador de la red privada tiene por tanto tres alternativas.

- Solicitar un bloque de direcciones públicas globales. Esto implica realizar una petición, y su correspondiente pago, a las autoridades de Internet: IANA o las entidades regionales en las que haya delegado la tarea de asignación de direcciones. Si la red privada está aislada, o al menos sus computadores no van a proporcionar servicios hacia Internet, puede ser un gasto injustificado.
- Utilizar un bloque público arbitrario sin conocimiento de las autoridades. Si efectivamente la red privada va a estar esencialmente aislada, no supone ningún problema técnico, pero puede plantear graves problemas logísticos y administrativos si en el futuro esa red acaba formando parte de Internet.
- Utilizar uno de los bloques reservados específicamente para redes privadas.

Esta tercera alternativa es la recomendada por las autoridades por medio de la RFC 1918 [RMK⁺96] y consiste en la elección arbitraria de uno de

los bloques definidos para ello (ver Cuadro 9.1). A estas direcciones se las denomina simplemente «direcciones privadas».

inicio		fin	prefijo CIDR
10.0.0.0	-	10.255.255.255	10 /8
172.16.0.0	-	172.31.255.255	172.16 /12
192.168.0.0	-	192.168.255.255	192.168 /16

CUADRO 9.1: Bloques IP reservados para direccionamiento privado

La direcciones privadas deben ser consideradas *no rutable*s, es decir, los routers del ISP y de la WAN en general descartarán cualquier paquete IP que tenga como destino una dirección privada. Precisamente por esto, cualquier organización puede elegir uno de estos bloques sin necesidad de autorización. Aunque existan millones de redes alrededor del mundo que hayan elegido exactamente el mismo bloque no supone ningún problema, ya que su tráfico nunca podrá ser confundido con el de otro computador con la misma dirección. Es decir, las direcciones privadas deben ser localmente únicas, pero al contrario de las públicas, no es necesario que sean globalmente únicas (y difícilmente lo serán).

Aunque el núcleo de Internet y los ISP descarten este tráfico, la organización es libre de encaminarlos dentro de su red corporativa. Esto le otorga una gran flexibilidad a la organización, pudiendo crear varias subredes interconectadas para diferentes propósitos o comunidades de usuarios.

9.3. Conectividad en redes privadas

Las redes domésticas actuales, y la mayoría de las que se utilizan en empresas y organizaciones, técnicamente son *redes privadas híbridas con tecnología* TCP/IP. En una red doméstica el ISP nos proporciona un dispositivo (ver figura 9.3) que conocemos informalmente como *router ADSL o router de fibra* que permite conectar nuestros dispositivos a Internet. En realidad ese dispositivo (esa caja) es más que un router. Incorpora normalmente:

- Un router IP.
- Un servidor DHCP.
- Un conmutador Ethernet.
- Un punto de acceso WiFi.
- Un módem que puede utilizar distintas tecnologías: ADSL, Symmetric Digital Subscriber Line (SDSL) o fibra óptica con FTTH, HFC y otras. En las conexiones de fibra, el módem puede ser un dispositivo distinto.

En la figura 9.3 podemos ver las conexiones de uno de estos equipos domésticos. El primer conector de la izquierda (de tipo RJ-11) conecta el equipo

a una roseta telefónica e iría a la entrada del módem ADSL incorporado. Los conectores RJ-45 amarillos numerados de 1 a 4 son los puertos del conmutador Ethernet incluido. Y la antena, que incorporan muchos de estos dispositivos, corresponde al punto de acceso WiFi.



FIGURA 9.3: Conexiones de un router ADSL doméstico

Esta imagen tiene una licencia Creative Commons Attribution-Share Alike 3.0. Disponible en <https://commons.wikimedia.org>

Cuando conectamos nuestro computador, móvil, televisor, etc. (sea por Ethernet o por WiFi) el servidor DHCP incorporado asignará direcciones privadas a estos equipos, que suelen ser del bloque 192.168.0.0/24, pero podrían ser de cualquiera de los bloques que hemos visto en la sección 9.2.1.

Si como hemos visto, los routers del ISP y de Internet van a descartar estos paquetes ¿cómo pueden estos dispositivos utilizar servicios públicos de Internet? La solución consiste en *traducir* las direcciones privadas a direcciones públicas en el momento de salir de la red privada. Esta traducción la realiza un proceso llamado NAT (Network Address Translation).

9.3.1. Traducción de Direcciones de Red (NAT)

NAT (Network Address Translation) es un programa (es software) que opera en algunos router IP, los que llamamos *routers* NAT (ver RFC 3022 [SE01]). En esencia el router NAT, que se coloca entre la red privada y la red pública, traduce (reescribe) las direcciones de los paquetes IP que lo atraviesan. La topología sería similar a la figura 9.4. El router NAT tiene una interfaz WAN (la que conecta con la red del ISP) con una dirección IP pública (180.20.30.13) y una interfaz LAN con una dirección privada (192.168.0.1) que, como en la figura, suele ser la primera del bloque. Los tres computadores tienen direcciones privadas del mismo bloque 192.168.0.0/24 asignadas por el servidor DHCP.

El uso más habitual de NAT es el que permite a los computadores de la red privada establecer conexiones con servidores de la red pública. Es el

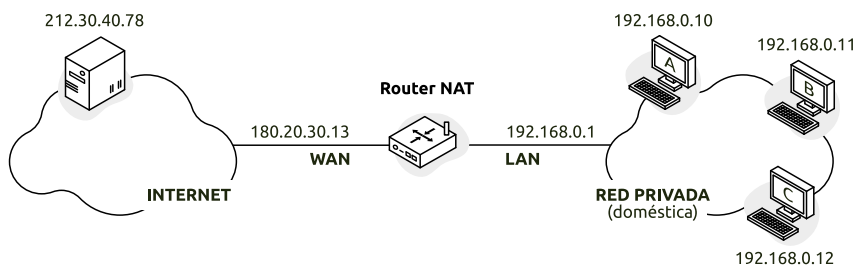


FIGURA 9.4: Ejemplo de configuración NAT en una red doméstica

llamado SNAT que se muestra en la figura 9.5. Si por ejemplo, el computador A quiere conectar con un servidor público, cuando los paquetes IP correspondientes a esa conexión salen de la red, el router substituye la dirección **origen**³ (privada) por la dirección WAN (pública) del router. Al volver la respuesta del servidor remoto, substituye la dirección destino (que es la pública del router) por la dirección privada del computador (ver figura 9.5). Hay que tener en cuenta que el servidor, ni ningún otro elemento en Internet, participan ni son conscientes de que esta traducción está ocurriendo.

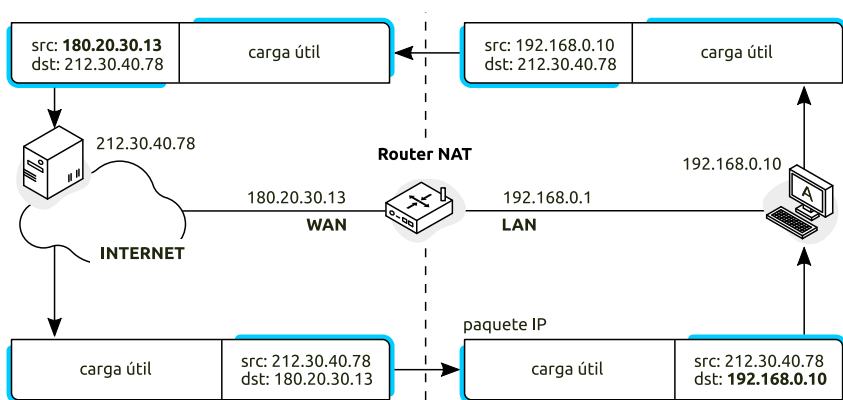


FIGURA 9.5: Ejemplo de SNAT

Para que la traducción de la respuesta funcione, el router debe saber cuál de los computadores de la red privada hizo la petición. Para ello, el software NAT apunta en la tabla NAT⁴ esas correspondencias en el momento de hacer la traducción de salida (ver cuadro 9.2).

³Por eso se llama *Source NAT*

⁴Formalmente *Address Translation Table*

Dirección local	Dirección remota
192.168.0.12	200.25.34.56
192.168.0.10	212.30.40.78

CUADRO 9.2: Tabla NAT para el envío de la figura 9.5

Sin embargo, esta solución tiene dos inconvenientes importantes:

- Dos o más computadores de la red privada no pueden mantener conexiones con el mismo servidor remoto al mismo tiempo. Al volver la respuesta, no se podría determinar quién fue el que originó la petición.
- Las conexiones deben ser establecidas siempre desde los computadores locales hacia el exterior.

Para resolver la primera limitación se propone una técnica mejorada denominada NAPT (Network Address Port Translation) (ver RFC 2663 [SH99]). Consiste en incorporar a la tabla NAT los números de puerto origen y destino del segmento TCP o el datagrama UDP. La probabilidad de que dos computadores de la red privada utilicen el mismo puerto origen en conexiones simultaneas al mismo servidor remoto es realmente muy baja. La figura 9.6 y el cuadro 9.3 ilustran esta técnica.

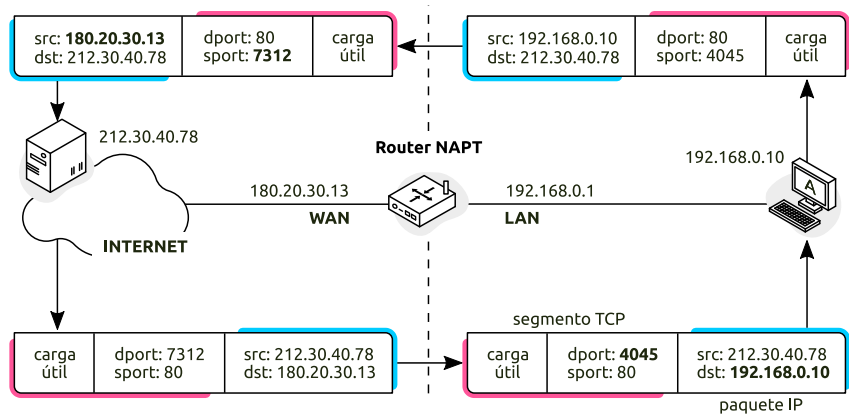


FIGURA 9.6: Ejemplo de NAPT

Dirección local	P. local	Pseudo	Dirección remota	P. remoto	Proto
192.168.0.10	4045	7312	212.30.40.78	80	TCP
192.168.0.12	32400	45012	130.0.23.12	22	TCP

CUADRO 9.3: Tabla NAPT para el envío de la figura 9.6

9.4. Red Privada Virtual (VPN)

Tanto las redes privadas como las redes híbridas tienen un importante inconveniente: las líneas dedicadas suponen un alto coste mensual. Una *red privada virtual*, del inglés VPN (Virtual Private Network), permite a las organizaciones usar Internet para comunicaciones tanto públicas como privadas. Una VPN es privada, dado que garantiza privacidad dentro de la organización, y es virtual, porque no usa recursos dedicados sino la red pública de Internet, es decir, la red es físicamente pública pero virtualmente privada. Para proporcionar seguridad, el acceso a la VPN requiere de autenticación y los datos se suelen transferir cifrados. Cuando se crea una VPN, se establece una conexión virtual punto-a-punto entre la red local del usuario y la red de la organización.

IPSec es una colección de protocolos diseñados para proporcionar seguridad, autenticación, integridad y privacidad, en el nivel de red IP. En VPNs, IPSec se utiliza en *modo túnel* cuando uno de los dos extremos del túnel (o los dos) no son el dispositivo final de la comunicación, sino un router, para proteger los datos de la intrusión de terceros. En este modo, IPSec encapsula tanto la cabecera IP como la carga útil de cada paquete IP para uso privado en un nuevo paquete IP, al que añade una nueva cabecera IP. La cabecera que los routers procesan es, por tanto, la nueva cabecera IP mientras que la original queda oculta y nunca será procesada por los dispositivos de la red pública. En este modo, se necesitan por tanto dos conjuntos de direcciones: un conjunto privado que usa el paquete original, y otro público, que se usa para alcanzar los routers de la organización. El router en el otro extremo del túnel, llevará a cabo la función inversa, desencapsular el paquete IP original y entrega (o encaminamiento) usando la dirección privada. Así, los paquetes privados que atraviesan la red global de Internet quedan «ocultos» y sólo serán procesados por los extremos del túnel.

9.4.1. Acceso a VPN

Típicamente, el acceso a la VPN por usuarios autorizados desde el exterior de la organización se realiza a través de un NAS (Network Access Server). Un NAS es el punto de entrada a la red para los usuarios de servicios de red. Lo que diferencia un NAS de un router típico es que proporciona servicios dinámicos para cada usuario, en base a su autenticación, el servicio solicitado y los recursos disponibles. Esta contabilidad, puede llevar a regular políticas de uso y control sobre los recursos demandados. Los servicios de un NAS incluyen (RFC 2637 [HPV+99]):

1. Interfaz de acceso al medio físico, incluyendo una gran variedad de

tecnologías de acceso (tanto analógicas como digitales): PSDN (Public Switched Data Network), ISDN (Integrated Services Digital Network), líneas de módem, adaptadores a distintos tipos de red, etc.

2. Terminación lógica de una sesión LCP (Link Control Protocol) del protocolo PPP (Point to Point Protocol).
3. Participación en protocolos de autenticación PPP.
4. Agregación de canales y gestión del ancho de banda en Protocolos PPP multi-enlace.
5. Terminación lógica de varios protocolos de control de red (NCPs).
6. Encaminamiento multi-protocolo y conexión entre múltiples interfaces NAS.

9.4.2. Autenticación y gestión remota

Debido a la necesidad de implementar funciones de tipo AAA (Authentication, Authorization and Accounting) y por razones prácticas, los servidores NAS típicamente dependen de servidores externos que almacenan bases de datos con las credenciales y otra información de autenticación de los usuarios. Esta separación de funciones plantea ciertas ventajas: los NAS y los servidores de autenticación podrían ser implementados sobre infraestructuras de hardware y software mejor adaptados a sus necesidades.

El protocolo RADIUS (Remote Authentication Dial In User Service), documentado en la RFC 2138 [RRSW97], es uno de los protocolos que se encargan del intercambio de la información entre el NAS y el servidor de autenticación. RADIUS sigue un modelo cliente/servidor: por un lado, el propio NAS actúa como cliente de RADIUS, transmitiendo información de usuario (por ejemplo, credenciales) al servidor RADIUS que, por otro lado, es responsable de aceptar/denegar intentos de conexión a VPN de los usuarios, y de los procesos de autenticación (usando diversos esquemas como PAP, CHAP o EAP), autorización y contabilidad del servicio consumido, en términos de tiempo, paquetes o datos transferidos u otra información específica del servicio y que pueda usarse para cuantificar el coste del servicio.

9.5. Protocolos punto-a-punto

La comunicación entre los routers local y remoto o entre el computador del usuario y el NAS de la organización suele realizarse mediante algún protocolo punto a punto como PPP. PPP (RFC 1661 [Sim94]) es un protocolo de la capa de enlace de datos (nivel 2) que permite conectar dos routers directamente sin ningún otro computador o dispositivo de red intermedio entre ambos, simulando un enlace punto-a-punto. El protocolo PPP pro-

porciona un método estándar para transportar datagramas de múltiples protocolos en el nivel de red (IPv4, IPv6, Novell, etc.) sobre esos enlaces punto-a-punto. Adicionalmente, puede proporcionar autenticación, cifrado y compresión de datos. PPP es un protocolo en capas que comprende tres componentes principales:

1. Un componente para encapsular paquetes de múltiples protocolos de red.
2. Un Protocolo de Control de Enlace, del inglés, LCP (Link Control Protocol), que permite establecer, configurar y evaluar el enlace, además de negociar opciones y parámetros del enlace.
3. Un conjunto de protocolos de Control de Red, del inglés NCP (Network Control Protocol), que permiten negociar configuraciones adicionales específicas del protocolo de red. Existe un NCP por cada protocolo de red soportado.

La RFC 1661 [Sim94] describe las distintas fases del protocolo: establecimiento del enlace, autenticación, protocolo de la capa de red y terminación del enlace. La transición entre fases se realiza enviando comandos de acciones encapsulados en mensajes entre los extremos del enlace.

9.6. Protocolos tunelizados

Un *protocolo tunelizado* (del inglés *tunneling protocol*) es una técnica que permite encapsular la Protocol Data Unit (PDU) de un protocolo en la PDU de otro protocolo (protocolo encapsulador), de forma que los paquetes originales quedan protegidos frente a terceros no deseados. El protocolo tunelizado crea un túnel, definido por los extremos (endpoints) y el protocolo de comunicación empleado:

```
fuelle ---> endpoint1 <-----túnel-----> endpoint2 ---> destino
```

El *endpoint1* realizará la función de encapsulado mientras que *endpoint2* realizará la función inversa de desencapsulado. El paquete original queda oculto mientras es transportado a lo largo del túnel dado que sólo los *endpoints* procesarán su cabecera. Con esta técnica un paquete puede atravesar una red que, en condiciones normales, no lo aceptaría. Se emplean para crear redes privadas virtuales, comunicar islas multicast, o redirección de tráfico en escenarios IP móvil.



A over B implica que la PDU del protocolo A es encapsulado en la PDU del protocolo B.

Los siguientes son ejemplos de protocolos tunelizados:

- IP over IP (RFC 2003 [Per96]): se utiliza cuando la fuente necesita incluir opciones de encaminamiento para alcanzar el destino, en escenarios donde los nodos son móviles, multicast o tarificación de usuarios.
- IPv6 over IPv4 (RFC 7059): usado durante la transición de IPv4 a IPv6, para garantizar que redes aisladas que usan IPv6 puedan ser transportadas a través de una red IPv4.
- IPSEC (*IP Security*, RFC 6071), es un conjunto de protocolos que proporcionan seguridad en el nivel de red IP (IPv4 e IPv6), empleado fundamentalmente para proporcionar seguridad a las VPNs. En IP-Sec, la protección del paquete IP se realiza incluyendo dos cabeceras especiales: ESP (Encapsulating Security Payload) Header, la cual proporciona confidencialidad, y AH (Authentication Header), que proporciona integridad.
- PPTP, L2TP, son protocolos tunelizados que transfieren tramas PPP sobre redes IP (PPP over IP), y que se emplean fundamentalmente para implementar VPNs. A continuación veremos un resumen de estos dos protocolos.

9.6.1. PPP over IP

Existen varios protocolos para transferir tramas PPP sobre redes IP: PPTP (Point-to-Point Tunneling Protocol) , definido en la RFC 2637 [HPV+99] y L2TP (Layer 2 Tunneling Protocol), RFC 2661 [TVR+99], ambos emplean otro protocolo tunelizado GRE (Generic Routing Encapsulation), RFC 2784 [FLH+00] para el encapsulado y definición de opciones de encaminamiento.

PPTP define un protocolo para transportar tramas PPP sobre una red IP entre ambos extremos de un túnel. El objetivo es establecer un túnel para implementar VPNs. Por este motivo, PPTP implementa las funciones tradicionales de un NAS (ver Sección 9.4.1) distribuyéndolas en un modelo cliente/servidor: El cliente se denomina PPTP Access Concentrator (PAC) e implementa las funciones 1, 2 (y a veces 3), y el servidor se denomina PNS (PPTP Network Server) y es responsable de las funciones 4,5 y 6. Dependiendo de la implementación, la función 3 puede ser implementada tanto en el PAC como en el PNS.

El protocolo funciona en dos fases:

- En una primera fase, se establece una conexión TCP al puerto 1723

entre el PAC y el PNS, a petición de cualquiera de los dos. Tras establecer conexión, el par PAC-PNS intercambia información sobre calidad del enlace a usar, capacidades básicas de PAC y PNS, etc. El intercambio de información se lleva a cabo a través de mensajes de protocolo definidos en la RFC, y tras este paso se inicia la solicitud de una o varias sesiones, por cualquiera de los extremos PAC o PNS. El control de conexión se mantiene vía mensajes *keep-alive*.

- En la segunda fase, tras solicitar el inicio de sesión, se establece un túnel para cada par PAC-PNS. Sobre el mismo túnel se puede transferir información de múltiples sesiones iniciadas en la primera fase del protocolo. Para distinguir las sesiones dentro del túnel se utiliza un campo de la cabecera especial del protocolo GRE, que se añade antes del encapsulado IP. La cabecera GRE contiene además información sobre confirmación y secuenciamiento de los mensajes para implementar control de flujo, control de congestión y detección de errores sobre el túnel. Incluye además información adicional de encaminamiento de los mensajes hacia su destino. La figura 9.7 proporciona una idea del encapsulado final en PPTP.

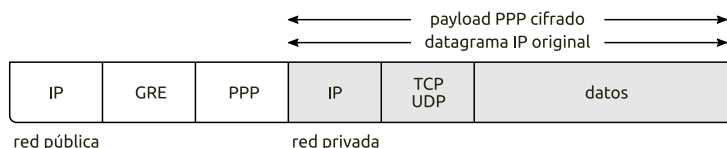


FIGURA 9.7: Encapsulado PPTP

L2TP tiene el mismo objetivo que PPTP y, sin embargo, corrige un problema asociado a la encapsulación de TCP denominado *TCP meltdown problem*, que surge cuando se encapsula TCP y se transfiere sobre una conexión TCP, y que lleva a una importantísima pérdida de rendimiento. En su lugar, L2TP usa UDP como protocolo encapsulador. Al igual que PPTP, se basa en una arquitectura cliente/servidor: el cliente L2TP se denomina LAC (L2TP Access Concentrator) y el servidor L2TP se denomina LNS (L2TP Network Server). L2TP usa el puerto 1701 de UDP para el intercambio de paquetes L2TP, tanto de control como de datos. Los paquetes L2TP se encapsulan como datagramas UDP, como se muestra en la figura:

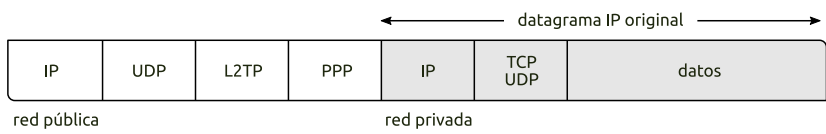


FIGURA 9.8: Encapsulado L2TP

Capítulo 10

Python

David Villa
Ana Rubio

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 que tenga un lenguaje de programación «convencional». 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 récord, lo cual lo hace ideal como primer lenguaje.

Python es perfecto como lenguaje «pegamento» y para prototipado rápido de aplicaciones, aunque finalmente fuera necesario rehacerlas en otros lenguajes. Aún así, eso no implica que no puedan ser definitivas. Dispone de librerías para desarrollar aplicaciones multihilo, distribuidas, bases de datos, con interfaz gráfico, juegos, gráficos 3D, cálculo científico, machine learning y una larga lista.

Este pequeño tutorial utiliza la versión actual de Python (3.9) y presupone que el lector tiene nociones básicas de programación con algún lenguaje como C, Java o similar.

10.1. 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 3.9.1+ (default, Feb  5 2021, 13:46:56)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

10.2. 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).

```
$ python3
>>> 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
```

En realidad no hemos cambiado el tipo de `a`, sino que hemos creado una nueva variable que reemplaza la anterior.

10.3. Tipos de datos

10.3.1. Valor nulo

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

10.3.2. Booleanos

Lo habitual:

```
>>> a = True
>>> a
True
```



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

10.3.3. 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 pueden operar entre sí 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 dato numérico, ya que permiten operaciones aritméticas. En Python, sin embargo, se utilizan cadenas de caracteres de tamaño 1 y no permiten operaciones aritméticas.

10.3.4. 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 '
```

El tipo **bytes** permite manejar secuencias de bytes. Es especialmente útil para serialización de datos. Por ejemplo, podemos convertir una cadena de caracteres UTF-8 a bytes y viceversa:

```
>>> word = 'ñandú'
>>> coded = word.encode()
>>> coded
b'\xc3\xba\xc3\xba'
>>> coded.decode()
'ñandú'
```

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. El tipo list sí es 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 números 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, incluso 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.

10.3.5. Orientado a objetos

Casi todo en Python está orientado a objetos, incluyendo las variables de tipos básicos. ¡Hasta los literales!

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

10.4. Módulos

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

```
>>> import math
>>> math.cos(math.pi)
-1.0
```

10.5. Estructuras de control

Están disponibles las más comunes: `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 en 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
```

10.6. 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 passed = 0
3 grades = {'antonio':6, 'maria':9}
4 for i in grades.values():
5     total += i
6     if i >= 5:
7         passed += 1
8
9 print('Average:', total / len(grades))
```

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 dos puntos al final. No se necesitan llaves ni ninguna otra cosa para delimitar los bloques.

10.7. 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))
```

10.8. 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 from operator import mul
3 factorial = lambda x: reduce(mul, range(1, x+1), 1)
4 print(factorial(10))
```

Muy aconsejable leer [\[Eby\]](#).

10.9. Hacer un fichero ejecutable

Lo normal es escribir un programa en un fichero de texto –con extensión .py utilizando algún buen editor (por ejemplo 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/python3
```

Y no olvides darle permisos de ejecución al fichero:

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

10.10. Type checking

Aunque Python es un lenguaje de tipado dinámico, desde la versión 3.0 se ofrece la opción de hacer «anotaciones» (*hints*) de tipo sobre variables y argumentos, y además existe un módulo donde se definen la semántica y convenciones para especificar de forma opcional esas anotaciones.

```
>>> pi: float = 3.14
```

El comportamiento del lenguaje es el mismo, y las variables pueden cambiar el tipo de valor que almacenan independientemente de su definición sin producir errores:

```
>>> pi: float = 3.14
>>> pi = "Tres coma catorce"
```

Sirve principalmente para facilitar la comprensión del código y poder llevar a cabo comprobaciones de tipo o *type checking* sobre el código escrito con herramientas como MyPy¹.

Si creamos un fichero ejecutable `factorial.py` con una versión de la función vista en la sección 10.7, modificada para hallar el factorial de un número que recibe como parámetro y con anotaciones en sus atributos y valor de retorno:

```
1  import sys
2
3  def factorial(n: int) -> int:
4      if n == 0:
5          return 1
6
7      return n * factorial(n-1)
8
9  factorial(sys.argv[1])
```

Y ejecutamos MyPy para que busque errores en nuestra implementación, obtenemos lo siguiente:

```
$ mypy factorial.py
factorial.py:13: error: Argument 1 to "factorial" has incompatible type "str"; expected "int"
"
Found 1 error in 1 file (checked 1 source file)
```

El programa nos avisa de que el número que estamos pasado a la función es en realidad una cadena, un tipo de dato que no se corresponde con el esperado. El error desaparece si transformamos el parámetro de entrada del programa en un entero con `int(sys.argv[1])`.

¹<http://mypy-lang.org/>

Serialización

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, decimal, cadena de caracteres, fecha, 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 las computadoras, todo dato es a fin de cuentas una secuencia de bits.

La serialización es el proceso de codificar los datos que manejan los programas (enteros, cadenas, imágenes, etc.) en secuencias de bytes susceptibles de ser almacenadas en un fichero o enviadas a través de la red. Por el contrario, la des-serialización es el proceso inverso. Más concretamente, estamos tratando lo «serialización binaria». También existen muchos formatos de serialización textual como XML, JavaScript Object Notation (JSON), YAML Ain't Markup Language (YAML), etc., que no vamos a tratar en este capítulo.

Es importante aclarar que no se debe confundir «codificación» con «cifrado» (o «encriptación»). Codificar es simplemente aplicar una transformación que modifica el modo en que se representan los datos, pero no implica el uso de ninguna clave ni ocultación u ofuscación del mensaje. Por ejemplo, un mensaje codificado en Morse es perfectamente legible por cualquiera que conozca el código.

11.1. Representación, sólo eso

Una de las excepciones a las que alude la sección anterior es la *programación de sistemas*, es decir, aquellos 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. Por eso es necesario manejar adecuadamente datos con estas representaciones.

Sin embargo, 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 11.1.

binario: `0b101010`

decimal: `42`

octal: `0o52`

hexadecimal: `0x2A`

LISTADO 11.1: Literales numéricos en Python

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

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 11.2 que indican claramente que se trata de cadenas.

LISTADO 11.2: Conversión a representación binaria, octal y hexadecimal

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

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 11.3.

LISTADO 11.3: 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
```

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.

11.2. 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.

11.3. 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

¹Estamos hablando de Python 3.0.

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 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 enteros) y se puede obtener fácilmente la lista de caracteres o secuencia de enteros 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]

```

11.4. 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 11.1 los muestra para un dato de 4 bytes.

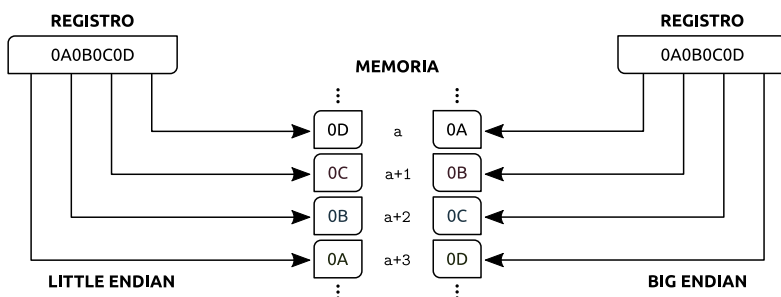


FIGURA 11.1: Ordenación de bytes

Para que el computador realice las operaciones (aritméticas, lógicas, etc.) que correspondan sobre el dato, es esencial que los programas manipulen la memoria de acuerdo al ordenamiento de la arquitectura correspondiente.

Puedes comprobar de una manera muy sencilla de qué tipo de ordenamiento tiene la computadora en la que estás utilizando el módulo `struct` (más adelante lo veremos con detalle).

LISTADO 11.4: Averiguar el ordenamiento de bytes con Python.

```
if struct.pack('H', 1) == b'\x00\x01':
    print("big endian")
else:
    print("little endian")
```

Aunque Python ofrece una forma específica de conocer el ordenamiento:

```
>>> sys.byteorder
'little'
```

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

Algo parecido a lo que ocurre en la memoria, ocurre con también 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 11.01 Escriba un programa Python que determine el tipo de ordenamiento que utiliza el computador que lo ejecuta.

E 11.02 Escriba la versión C del ejercicio anterior.

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

LISTADO 11.5: Funciones de conversión de ordenamiento del módulo `socket`.

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

³del inglés *on the wire*

⁴retornan el mismo valor que se les pasa como parámetro

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.

11.5. 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 en 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 11.6: 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 11.7: 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 bien 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'

```

11.6. Empaquetado

El módulo struct de la librería estándar de Python puede hacer transformaciones en la representación de los datos de un modo mucho más flexible. 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 número 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

LISTADO 11.9: 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@'
```

especificar formato. La tabla 11.1 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 11.8: 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 11.1: struct: especificación de ordenamiento

La tabla 11.2 corresponde a la especificación de formato, y el listado 11.9 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 § 4.1.1).

x	relleno (alineado al siguiente dato)
c	carácter (char)
b	byte con signo
B	byte sin signo
?	booleano/char
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	float
d	double
s	cadena de caracteres (un número previo indica tamaño)
P	entero que puede almacenar una dirección de memoria
q ó Q	entero equivalente al long long de C en la misma arquitectura.

CUADRO 11.2: struct: especificación de formato

El valor para los campos de la trama será:

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 11.10 se puede ver cómo construir dicha cabecera, la secuencia de bytes que se obtiene y su equivalente numérico. La cadena de formato ('!6s6sh') indica que debe codificarse con ordenamiento de red ('!') y que 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 11.10: 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\xd3\x07\x08\x06'
4 >>> list(header)
5 [255, 255, 255, 255, 255, 255, 255, 196, 133, 8, 237, 211, 7, 8, 6]
```

E 11.03 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 11.04 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*.

11.7. 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 11.11 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 11.11: struct: desempaquetando una cabecera Ethernet

```

1 >>> header
2 b'\xff\xff\xff\xff\xff\xff\x04\x85\x08\xed\x07\x08\x06'
3 >>> struct.unpack('!6s6sh', header)
4 (b'\xff\xff\xff\xff\xff\xff', b'\x04\x85\x08\xed\x07', 2054)
5 >>> hex(2054)
6 '0x806'
```

E 11.05 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)?`

Capítulo 12

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.

12.1. 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».

12.1.1. Historia

De las diferentes formas de IPC (Inter Process Communication), los sockets son con mucho la más popular. En una plataforma dada, probable-

mente 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).

12.2. 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.

12.2.1. IPC

Si necesita Inter-Process Communication (IPC) 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.

12.3. 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("conexion 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 descubre 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.

12.3.1. 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.

12.4. 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.

12.4.1. 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.

12.5. 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.

12.5.1. 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()`.

12.6. 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>

Capítulo 13

Modelo Cliente-Servidor

David Villa
Fernando Rincón

El modelo cliente-servidor (figura 13.1) es, probablemente, el enfoque más simple y 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 hoy, la mayoría de aplicaciones que utilizamos siguen aplicando esta misma arquitectura básica¹.

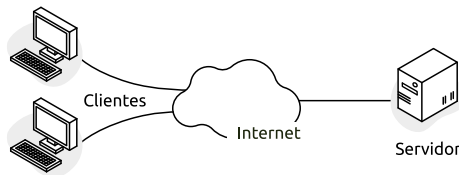


FIGURA 13.1: Modelo Cliente/Servidor [Fuente:Wikimedia Commons]

En este modelo se asumen dos roles bien diferenciados:

- El **servidor** es la parte pasiva. Permanece inactiva a la espera de una petición para realizar una tarea o proveer 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.

Decimos que son **roles** porque una misma aplicación puede tomar uno, otro o ambos en distintos momentos o en distintos componentes, es decir, no definen tipos de aplicaciones. Es cierto que normalmente se habla de «servidores» y «clientes» refiriéndose a programas concretos –el servidor y

¹Por ejemplo REpresentational State Transfer (REST)

el cliente web (navegador)–, pero eso se debe simplemente a que uno de esos roles destaca fuertemente sobre el otro.

E 13.01 Nombre los tres servidores y clientes HTTP más usados en el mundo (y cite la fuente).

También es habitual referirse a los computadores como «clientes» y sobre todo «servidores» para denotar que su utilidad principal es la de alojar programas con la finalidad 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 o, más probablemente, en máquinas virtuales. Incluso en un entorno doméstico es relativamente común 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 con un resultado o un código indicando 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 (por ej. una impresora). Este tipo de recurso se dice que es *de tiempo compartido*, pero es muy habitual que el recurso tenga una granularidad mucho menor —como los registros de una base de datos— que lo convierte en un problema de acceso concurrente, obligando a gestionar los accesos de modo que se pueda asegurar la integridad de los recursos (datos en este caso).

13.1. 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.

13.1.1. Paso 1: Mensaje unidireccional

En este primer paso el objetivo es el siguiente:

- Implementar un servidor UDP que ha de 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 contiene un mensaje de texto.
4. Imprimir el mensaje en consola.

Estas tareas corresponden línea a línea con el listado 13.1.

LISTADO 13.1: Servidor de chat UDP básico
udp-chat/server1.py

```
1 import socket
2
3 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4 sock.bind(('', 12345))
5 message, client = sock.recvfrom(1024)
6 print(message.decode(), client)
7 sock.close()
```

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 determina 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 simplemente ```), que le dice al SO que vincule este socket a 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 en este momento, y eso no es siempre una tarea sencilla...

E 13.02 ¿Cómo se puede averiguar si un puerto determinado está libre u ocupado (vinculado a un servidor activo) en el propio computador? Y si está ocupado, ¿cómo puedes averiguar qué programa es el responsable?

E 13.03 ¿Cómo se puede averiguar si hay un servidor vinculado a un puerto en un computador remoto?

E 13.04 Escriba un programa Python que cree un socket UDP e intente vincularlo a un puerto ocupado. ¿Qué error obtiene?

E 13.05 La dirección IP 0.0.0.0 es una dirección especial según indica la RFC3330 [[IAN02](#)]. ¿Cuál es su propósito?

E 13.06 ¿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 (en realidad 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:

```
$ python3 udp-chat/server1.py
```

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

Cliente

Antes de escribir un programa cliente en Python es buena idea 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 13.2.

LISTADO 13.2: Cliente de chat UDP básico
udp-chat/client1.py

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

La **línea 3** de este listado crea el socket UDP (idéntico al del servidor) y la **línea 4** 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 13.07 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 13.08 Utiliza `netcat` para probar el cliente, es decir, identifica el comando adecuado para sustituir el servidor UDP mediante un comando `netcat`. ¿Cuál es ese comando?

13.1.2. Paso 2: Lo educado es responder

El servidor del paso anterior sólo imprime el mensaje recibido. Un pequeño cambio le permitirá devolver el saludo al cliente.

Utilizando la dirección del cliente, que se obtiene 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 13.3).

LISTADO 13.3: Servidor de chat UDP con respuesta
udp-chat/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.decode(), peer)
7 sock.sendto("qué tal?".encode(), peer)
8 sock.close()

```

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 hace el servidor: usar el método `recvfrom()` (ver listado 13.4).

LISTADO 13.4: Cliente de chat UDP con respuesta
udp-chat/client2.py

```

1 import socket
2
3 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4 sock.sendto("hola".encode(), ('127.0.0.1', 12345))
5 message, peer = sock.recvfrom(1024)
6 print("{} from {}".format(message.decode(), peer))
7 sock.close()

```

E 13.09 Ejecuta servidor y cliente y comprueba que efectivamente el servidor responde y el cliente imprime el mensaje en su consola.

E 13.10 Prueba servidor y cliente por separado con netcat.

13.1.3. Paso 3: Libertad de expresión

Con 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 para 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 13.5 muestra el código completo del servidor:

LISTADO 13.5: Servidor de chat UDP por turnos
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      message_in, peer = sock.recvfrom(1024)
9      print(message_in.decode())
10
11     if message_in == QUIT:
12         break
13
14     message_out = input().encode()
15     sock.sendto(message_out, peer)
16
17     if message_out == QUIT:
18         break
19
20 sock.close()

```

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 13.6) es muy similar.

LISTADO 13.6: Cliente de chat UDP por turnos
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      message_out = input().encode()
9      sock.sendto(message_out, server)
10
11     if message_out == QUIT:
12         break
13
14     message_in, peer = sock.recvfrom(1024)
15     print(message_in.decode())
16
17     if message_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 13.11 Ejecuta servidor y cliente en computadores diferentes. ¿Por qué no funciona? Haz las modificaciones necesarias para lograr que funcione.

E 13.12 Reemplaza el servidor por netcat y explica las diferencias.

E 13.13 Reemplaza el cliente por netcat y explica las diferencias.

13.1.4. 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 ambos han de esperar a que su interlocutor envíe un mensaje antes de poder escribir de nuevo.

Es un problema clásico en software que maneja E/S o comunicaciones. 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*² y es típico también de las GUI.

Existen varias formas de abordar este 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 13.7.

LISTADO 13.7: Servidor de chat UDP simultaneo
udp-chat/server4.py

```

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

```

²http://es.wikipedia.org/wiki/Programación_dirigida_por_eventos

```

27         if message == QUIT:
28             self.sock.sendto(QUIT, self.peer)
29             break
30
31
32 if __name__ == '__main__':
33     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
34     sock.bind(server)
35     message, client = sock.recvfrom(0, socket.MSG_PEEK)
36     Chat(sock, client).run()

```

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()` (**línea 35**) se utiliza únicamente para obtener la dirección del cliente, pero no lee nada del buffer del socket gracias al flag `MSG_PEEK`. En la **línea 36** se crea una instancia de la clase `Chat` pasando el socket y la dirección del cliente como parámetros.

El cliente solo difiere en la creación del socket. Simplemente crea el socket (pero no lo vincula) y una instancia de la clase `Chat` (la misma del servidor), a la que le pasa dicho socket y la dirección del servidor. Puede verlo en el listado 13.8.

LISTADO 13.8: Cliente de chat UDP simultáneo
udp-chat/client4.py

```

1 import socket
2 from server4 import Chat, server
3
4 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5 Chat(sock, server).run()

```

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 la lectura de la salida. 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 alcance de

este ejemplo.

E 13.14 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?

E 13.15 Cuando se usa netcat como chat se pueden enviar y recibir mensajes en cualquier momento. ¿Cómo lo hace?

13.1.5. 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 comporte como servidor o cliente en función de un parámetro de línea de comandos. Puedes verlo en el listado 13.9.

LISTADO 13.9: Chat UDP multihilo (servidor y cliente)
udp-chat/chat-thread.py

```
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      def run(self):
12          _thread.start_new_thread(self.sending, ())
13          self.receiving()
14          self.sock.close()
15
16      def sending(self):
17          while 1:
18              message = input().encode()
19              self.sock.sendto(message, self.peer)
20
21              if message == QUIT:
22                  break
23
24      def receiving(self):
25          while 1:
26              message, peer = self.sock.recvfrom(1024)
27              print("other> {}".format(message.decode()))
28
29              if message == QUIT:
30                  self.sock.sendto(QUIT, self.peer)
31                  break
32
```

```

33 if __name__ == '__main__':
34     if len(sys.argv) != 2:
35         print(__doc__ % sys.argv[0])
36         sys.exit()
37
38     mode = sys.argv[1]
39     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
40
41     if mode == '--server':
42         sock.bind(SERVER)
43         message, client = sock.recvfrom(0, socket.MSG_PEEK)
44         Chat(sock, client).run()
45
46     else:
47         Chat(sock, SERVER).run()

```

13.2. Chat UDP con select()

LISTADO 13.10: Chat UDP (servidor y cliente) con select()
udp-chat/chat-select.py

```

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      def run(self):
12          _thread.start_new_thread(self.sending, ())
13          self.receiving()
14          self.sock.close()
15
16      def sending(self):
17          while 1:
18              message = input().encode()
19              self.sock.sendto(message, self.peer)
20
21              if message == QUIT:
22                  break
23
24      def receiving(self):
25          while 1:
26              message, peer = self.sock.recvfrom(1024)
27              print("other> {}".format(message.decode()))
28
29              if message == QUIT:
30                  self.sock.sendto(QUIT, self.peer)
31                  break
32
33  if __name__ == '__main__':
34      if len(sys.argv) != 2:

```

```
35     print(__doc__ % sys.argv[0])
36     sys.exit()
37
38     mode = sys.argv[1]
39     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
40
41     if mode == '--server':
42         sock.bind(SERVER)
43         message, client = sock.recvfrom(0, socket.MSG_PEEK)
44         Chat(sock, client).run()
45
46     else:
47         Chat(sock, SERVER).run()
```

13.3. Chat UDP con asyncio

ToDo!

13.4. Chat TCP

ToDo!

13.5. Puertos ocupados

Al ejecutar la operación `socket.bind()` puede ocurrir una excepción `OSE-RROR: Address already in use`. Esto puede deberse a que efectivamente ya hay otro servidor escuchando en el mismo puerto, algo que podemos comprobar con `ss -lpn`. Obviamente esto no es un problema exclusivo de Python. Ocurrirá con cualquier programa escrito en cualquier lenguaje que solicite este servicio al SO.

Sin embargo, hay veces en que un programa se pueden vincular sin problemas a un determinado puerto, pero al reiniciarlo aparece el fallo. Esto se debe a que aunque cerremos un socket servidor³ con `socket.close()`, éste queda en un estado llamado `TIME_WAIT` a cargo del SO. En este estado, el socket seguirá activo para garantizar que el último ACK es recibido por cliente, y también para gestionar posibles retransmisiones⁴ que lleguen de forma diferida. Puedes profundizar más en esta cuestión consultado la sección «The TCP Quiet Time Concept» en la RFC793 [Pos81c]. El socket quedará en este estado entre 20 segundos y 4 minutos, dependiendo de la implementación.

Si el programador prefiere desactivar esta salvaguarda, algo nada recomendable en servidores en producción, puede hacerlo con la siguiente senten-

³un socket sobre el que se ejecutó `bind()`

⁴llamados «duplicados errantes» (*wandering duplicates*)

cia.

```
1 sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

También puede ocurrir que necesitemos abrir un servidor, pero no nos importe en qué puerto a condición de que esté libre, por ejemplo cuando lo utilizamos como *callback*. En este caso podríamos generar números de puerto en un bucle hasta que al hacer el `bind()` no obtuvieramos dicha excepción. Sin embargo, hay un modo más simple y efectivo: vincular al puerto 0. En realidad no existe un puerto 0, lo que significa realmente es que queremos que el SO asigne un puerto libre. Para averiguar cuál se ha asignado se usa el método `socket.getsockname()`:

```
1 sock = socket.socket()
2 sock.bind(('', 0))
3 print("Vinculado a {}".format(sock.getsockname()))
```

13.6. Ejemplos rápidos de sockets

13.6.1. Un cliente HTTP básico

LISTADO 13.11: Cliente HTTP básico
examples/http_mini_client.py

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

13.6.2. Un servidor HTTP

LISTADO 13.12: Servidor HTTP
examples/http_server.py

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

o simplemente:

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

Capítulo 14

Puertos y servicios

David Villa

Junto con las herramientas que ya se han mostrado (`ping`, `tracert` 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 listar conexiones de red, tablas de encaminamiento, estadísticas de interfaces, grupos multicast, etc.

nmap

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

IPTraf

Es un monitor protocolos TCP/IP.

Antes de pasar a ver el uso 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 capítulo es comprender y aprender a manejar estas herramientas 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.

14.1. 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.

14.1.1. 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
---------	--------------	---------	----	-----	--------

14.1.2. 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

14.1.3. 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
[...]						

14.1.4. 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	

14.1.5. 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.

14.2. 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.

14.3. 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:

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.

14.4. Referencias

- [Página oficial de nmap.](#)
- [Página oficial de IPTraf.](#)

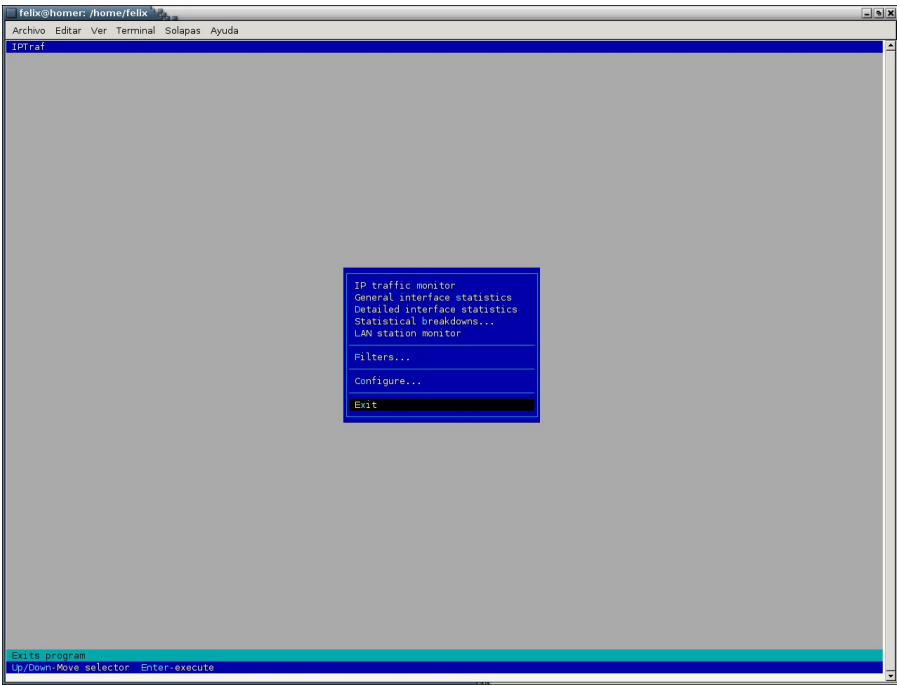


FIGURA 14.1

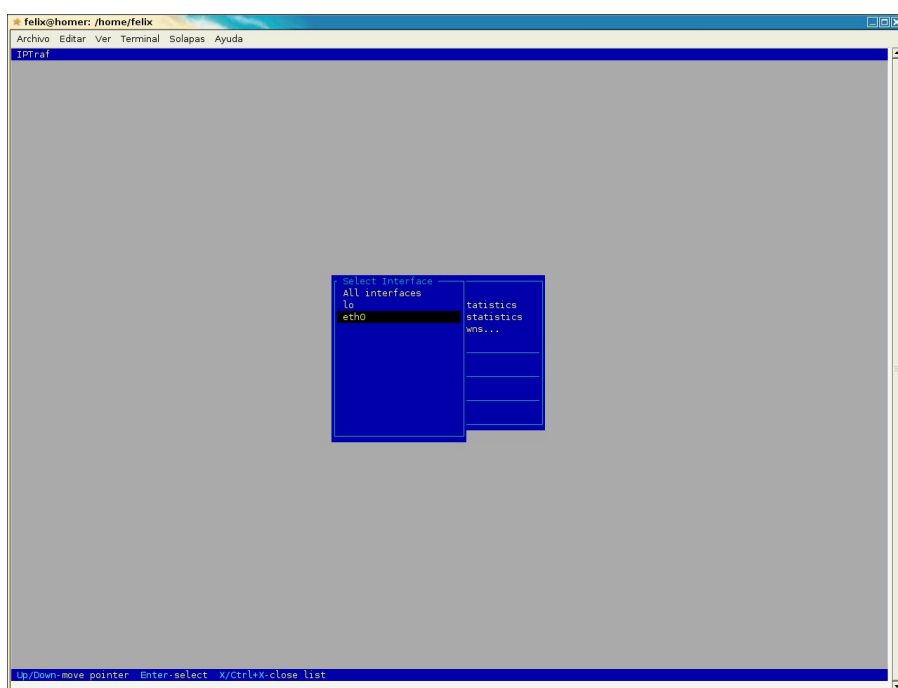


FIGURA 14.2

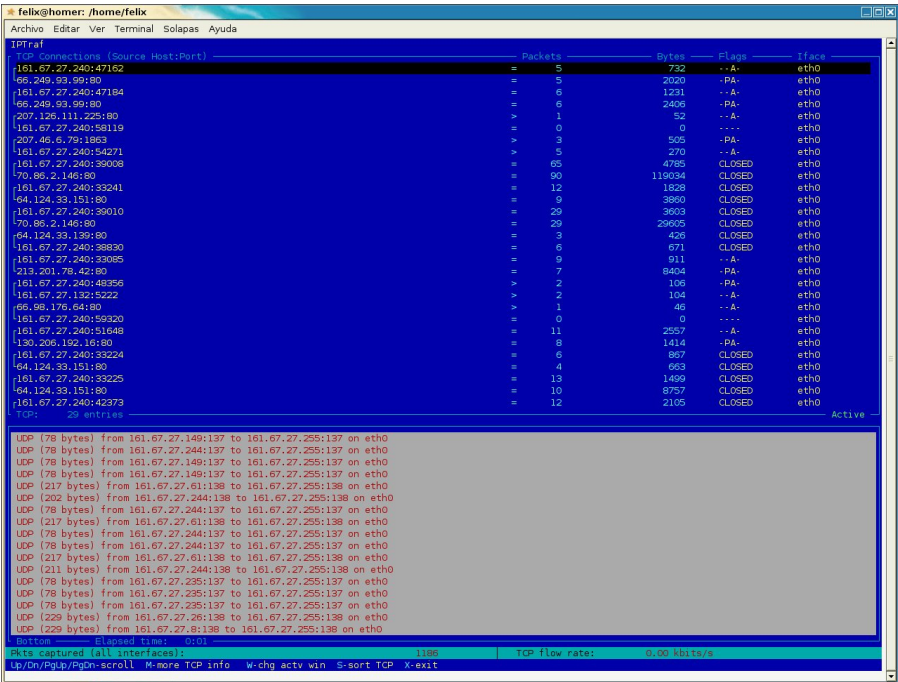
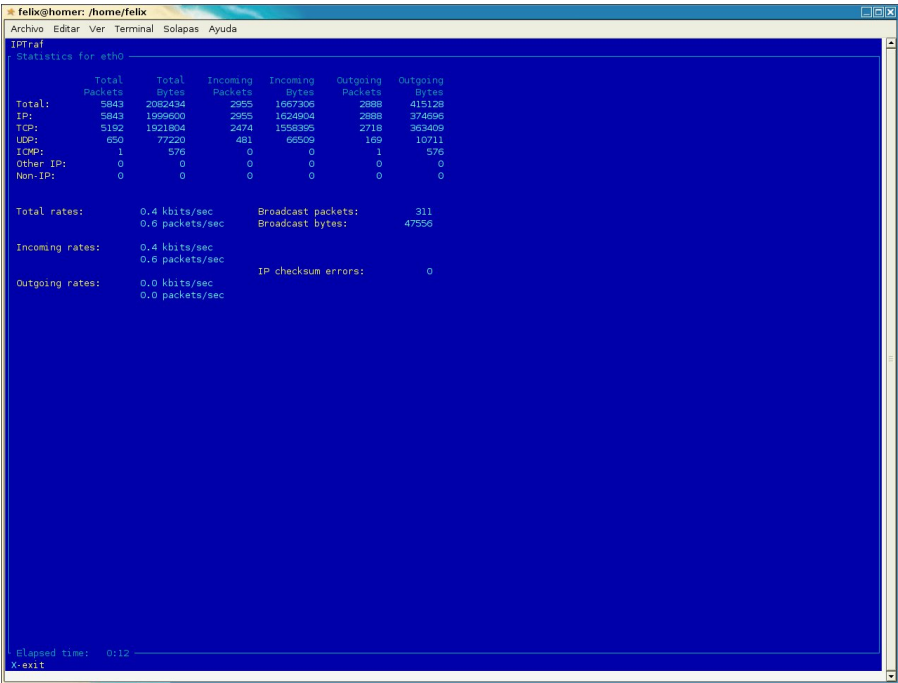


FIGURA 14.3



Capítulo 15

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»¹.

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)². 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 de la capa de transporte o inferior, ya sea IP, ICMP, ARP, TCP, etc. Esos sockets únicamente permiten indicar cuál será la carga útil de los segmentos TCP o UDP y solo indirectamente se puede

¹Concepto a veces traducido como «conector directo».

²El adjetivo «raw» (crudo) se utiliza como contraposición a «cooked» (cocinado).

influir en algunos de los campos de sus cabeceras: puerto origen y destino y poco más³.

En raras situaciones 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 hacen estos programas? La respuesta, como podrás adivinar, pasa por los sockets RAW.

15.1. Acceso privilegiado

Como se 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 privilegios⁴ 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 equivalente 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
```

³a menos que acudamos a llamadas al sistema como setsockopt().

⁴bit SUID

```

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

```

15.2. Tipos

Lo primero a 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.

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.

15.3. 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* extremadamente básico que imprime por consola las tramas Ethernet/WiFi completas recibidas por cualquier interfaz y portando cualquier protocolo.

LISTADO 15.1: Sniffer básico con AF_PACKET:SOCK_RAW
raw/sniff-all.py

```

1  import socket
2
3  ETH_P_ALL = 3
4
5  sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
6                      socket.htons(ETH_P_ALL))
7
8  while 1:
9      print("--\n{!r}".format(sock.recvfrom(1600)))

```

Y a continuación se muestra cómo ejecutar este programa, y su salida:

```
$ sudo ./sniff-all.py
--
(b'\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', ('eth0', 2054, 1, 1, '\
xa8\x92,
\xce\xcd\xb3'))
```

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

Tipo de trama

Es decir, el código que identifica el protocolo encapsulado como carga útil.⁵ 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» queda demostrado en el siguiente programa, llamado `sniff-arp.py`. Sólo muestra mensajes ARP recibidos por la interfaz que se indique como argumento:

LISTADO 15.2: Sniffer AF_PACKET:SOCK_RAW filtrando ARP
raw/sniff-arp.py

```
1 import sys
2 import socket
3
4 if len(sys.argv) != 2:
5     print("usage: {} <iface>".format(sys.argv[0]))
6     exit(1)
7
8 ETH_P_ARP = 0x0806
9
10 sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
11                      socket.htons(ETH_P_ARP))
12 sock.bind((sys.argv[1], ETH_P_ARP))
13
14 while 1:
15     print("--\n{!r}".format(sock.recv(1600)))
```

Y el programa en acción:

```
$ sudo ./sniff-arp.py wlan0
--
b'\xff\xff\xff\xff\xff\xffl>m\x84y\x1d\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01l>m\x84y
\x1d\xa1c\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'
```

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

Si te fijas, es fácil identificar la cabecera Ethernet en esa secuencia de bytes. Aparecen 6 bytes ‘0xff’, que corresponden a la dirección broadcast de Ethernet; y más adelante ‘0x0806’, que como hemos visto en el programa, corresponden al tipo de payload ARP.

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 veremos cómo 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.

15.3.1. 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`⁶.

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)

LISTADO 15.3: Enviando una trama Ethernet con AF_PACKET:SOCK_RAW
raw/send-wrong-eth.py

```

1  import sys
2  import socket
3  import struct
4
5  if len(sys.argv) != 2:
6      print("usage: {} <iface>".format(sys.argv[0]))
7      exit(1)
8
9  ETH_P_ARP = 0x0806
10
11 sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
12                      socket.htons(ETH_P_ARP))
13 sock.bind((sys.argv[1], ETH_P_ARP))
14
15 sock.send(struct.pack('!6s6sh', 6 * b'\xFF',
16                          b'\x00\x01\x02\x03\x04\x05', ETH_P_ARP))

```

⁶Consulte el capítulo 11 para más información sobre dicho módulo.

Si capturas esa trama con wireshark o tshark verás que aparece con un «malformed packet», y con razón: es sólo una cabecera ¡no tiene carga útil!

```
$ tshark -a duration:7 -n -f arp
Capturing on 'wlan0'
1 0.000000000 00:01:02:03:04:05 → ff:ff:ff:ff:ff:ff ARP 14 [Malformed Packet]
```

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.

15.3.2. 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.

15.3.2.1. Generando mensajes

El programa necesita enviar mensajes ARP Request, que irán encapsulados en tramas Ethernet. Una forma de implementar esta tarea (llamada a veces «sintetizar paquetes») y aprovechar la POO 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 serialize(self):
12        retval = struct.pack("!6s6sh", self.hwdst, self.hwsrc,
13                               self.payload.proto) + self.payload.serialize()
14
15        return retval + (60-len(retval)) * "\x00"
```

Lo único a destacar de la clase `Ether` es el método `serialize()` 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.


```

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 serialize(self):
10         return struct.pack("!HHbbH6s4s6s4s", 0x1, 0x0800, 6, 4, 1,
11                                self.frame.hwsrc, self.psrc, "\x00", self.pdst)

```

Esos datos se empaquetan en binario gracias a `struct.pack()`⁷ indicando que se trata de 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:

15.3.2.2. 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

⁷Ver <http://docs.python.org/library/struct.html#format-characters>

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, self.psrc,
10              self.hwdst, self.pdst) = struct.unpack("!6s4s6s4s", msg[8:28])
11         except struct.error:
12             raise DissectionError

```

El constructor de la clase acepta por parámetro una secuencia de bytes, 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`.

15.3.2.3. 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, iface):
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((iface, ETH_P_ARP))
7
8     frame = Ether(sock.getsockname()[-1], BROADCAST)
9     frame.set_payload(ArpRequest(ipsrc, ipdst))
10
11     sock.send(frame.serialize())
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, y la interfaz de red (línea 1). Primero crea y vincula el socket a la interfaz solicitada (líneas 4-6). A continuación crea una trama Ethernet con destino *broadcast* y origen la MAC de la interfaz (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 una versión ampliada en el fichero `raw/arping.py`.

15.4. 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 engorroso 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.

15.4.1. Capturando mensajes

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

LISTADO 15.4: Sniffer de mensajes UDP con `AF_INET:SOCK_RAW`
`raw/sniff-udp.py`

```

1 import socket
2
3 sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
4                      socket.getprotobyname('udp'))
5
6 while 1:
7     print("--\n{!r}".format(sock.recv(1600)))

```

La función `getprotobyname()` devuelve el número de protocolo⁸ a partir de

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

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).

15.4.2. Enviando

Para enviar datos sobre este tipo de socket debes utilizar el método `sendto` indicando la dirección destino. El listado 15.5 envía envía un segmento UDP «sintético», pero válido, que contiene el texto «hello Inet».

LISTADO 15.5: Sintetizando un mensaje UDP con `AF_INET:SOCK_RAW`
raw/send-udp.py

```

1  import socket
2  import struct
3
4  sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
5                      socket.getprotobyname('udp'))
6
7  payload = b'hello Inet'
8  udp_pkt = struct.pack('!4h', 0, 2000, 8+len(payload), 0) + payload
9  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:

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

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

15.4.2.1. 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 nosotros algunos de los campos más latosos si así queremos (poniendo ceros en ellos). Esos campos son:

- El checksum.
- La dirección IP origen.
- El identificador del mensaje.
- El campo de longitud total.

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

```
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 ha de crearse un socket de un *protocolo* especial identificado como `IPPROTO_RAW`:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, 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.

15.5. 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 15.01 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 15.02 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 15.03 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 15.04 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 15.05 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 Kpbs
```

- E 15.06** Escriba un programa que calcule el *tiempo medio de inactividad* (Average Idle Time) de un enlace durante el tiempo de ejecución del programa.
- E 15.07** 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 15.08** 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 15.09** 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

16.1. 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 Red Telefónica Conmutada (RTC), 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.

16.2. 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.

16.3. 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.

16.4. Comandos básicos

Este es un resumen de comandos habituales simplificados.

16.4.1. Ver la configuración

```
# iptables -L [-t tabla] [-v]
```

16.4.2. Borrar todas las reglas de una tabla

```
# iptables -F [-t tabla]
```

16.5. 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
```

16.5.1. 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
```

16.5.2. 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
```

16.5.3. 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
```


Parte I

Seguridad

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 vulnerabilidades explotables.

17.1. 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 el servicio 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 escaneado **pasivo** obviamente se puede realizar con un sniffer de propósito general:

```
$ tshark -a duration:7 -n -f arp -i wlan0
Capturing on 'wlan0'
% 1 0.000000000 00:64:40:3a:c9:40 -> ff::ff ARP 60 Who has 161.67.27.73? Tell
161.67.27.1
% 2 1.140641871 00:64:40:3a:c9:40 -> ff::ff ARP 60 Who has 161.67.27.228? Tell
161.67.27.1
% 3 1.140652343 00:64:40:3a:c9:40 -> ff::ff ARP 60 Who has 161.67.27.118? Tell
161.67.27.1
% 4 1.140653455 00:64:40:3a:c9:40 -> ff::ff ARP 60 Who has 161.67.27.63? Tell
161.67.27.1
% 5 1.495058591 70:54:d2:96:4f:d7 -> ff::ff ARP 60 Who has 161.67.27.20? Tell
161.67.27.97
% 6 1.495062336 70:54:d2:96:4f:d7 -> ff::ff ARP 60 Who has 161.67.27.21? Tell
161.67.27.97
% 7 1.495377239 70:54:d2:96:4f:d7 -> ff::ff ARP 60 Who has 161.67.27.60? Tell
161.67.27.97

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, suponiendo por supuesto que se trate de tráfico legítimo. En cualquier caso, este tipo de información siempre se debe considerar como indicios y no como evidencias.

E 17.01 Las técnicas descritas en este capítulo suelen aplicarse a un rango de direcciones (expresado en notación CIDR, por ejemplo, 161.67.0.0/24). Escribe un programa que genere todas las direcciones posibles en un rango expresado de este modo.

E 17.02 Escribe un programa que utilice sockets *raw* para listar las direcciones IP que aparecen en las peticiones ARP capturadas.

El escaneado **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 17.03 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 contenidas 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

Pero 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 activo.

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.

17.2. Otros protocolos

Algunas aplicaciones como Universal Plug and Play (UPnP), Common Internet File System (CIFS), Common Unix Printing System (CUPS), Dropbox,

etc.) utilizan protocolos específicos de descubrimiento de servicios y dispositivos que pueden aprovecharse para detectarlos. Estos protocolos por regla general utilizan mensajes broadcast o multicast en la red local, de modo que se pueden capturar de forma pasiva o incluso sintetizar mensajes de descubrimiento para forzar respuestas de los dispositivos. La herramienta `nmap` también tiene soporte para este tipo de descubrimiento.

```
# nmap -sP --script discovery,broadcast 192.168.0.0/24
```



Muchos de los scripts de `nmap` pueden tener consecuencias graves en el funcionamiento de los sistemas escaneados. Solo deberían usarse con sistemas propios.

E 17.04 El programa `nmap` proporciona soporte para todas las técnicas de descubrimiento anteriormente descritas. Determine qué parámetros de llamada corresponden a cada una de ellas y demuestre con una captura de tráfico que efectivamente es así.

17.3. Escaneo de puertos

El escaneo de puertos 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 un 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. Es decir, un puerto cerrado ofrece una respuesta (por parte del SO).

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 cortafuegos. 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 computadores 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 más comunes para escaneo de puertos:

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 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 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, PSH 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 hace uso de computador adicional (denominado *zombie*) para realizar el escaneo. Se basa en el hecho de que el número utilizado como 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 un paquete 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 se obtiene el identificador del *zombie*. Si ambos números difieren en más de uno significa que el puerto consultado está abierto.

Este mecanismo 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 17.05** Escribe un programa que implemente la técnica TCP SYN y que acepte una dirección IP y un rango de puertos a escanear.
- E 17.06** Escribe un programa que verifique el funcionamiento de la técnica de escaneo *Idle*.
- E 17.07** Determina qué opción de nmap permite utilizar cada una de las técnicas descritas y ejecuta un ejemplo funcional.

17.4. Identificación de servicios

Por defecto nmap indica el nombre del servicio/protocolo registrado o típico para cada puerto. Sin embargo, determinar qué servidor hay detrás de puerto abierto no es tan sencillo. Aunque IANA tiene puertos registrados para la mayoría de los servicios comunes, configurar un programa en un puerto arbitrario a decisión del administrador es una tarea trivial en la mayoría de los casos.

Por eso, en un análisis remoto habremos de considerar cuál es el contenido exacto de los mensajes que acepta y genera el servidor si queremos determinar qué servicio proporciona y con mayor precisión si se desea concretar el fabricante y la versión del programa responsable. Esta información es crítica a la hora de averiguar a qué vulnerabilidades puede estar expuesto cada nodo. Estas técnicas se denominan «service fingerprinting» o directamente «version detection».

Las herramientas de escaneado (como nmap) utilizan patrones conocidos para detectar el servicio, el nombre de la aplicación y su versión. El siguiente comando activa la detección de versiones con nmap:

```
$ sudo nmap -sV -F example.org
Starting Nmap 7.91 ( https://nmap.org ) at 2020-12-15 18:30 CET
Host is up (0.17s latency).
Not shown: 95 filtered ports
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 7.4 (protocol 2.0)
25/tcp    open  smtp      Postfix smtpd
80/tcp    open  http      Apache httpd 2.4.6
113/tcp   closed ident
443/tcp   open  ssl/ssl Apache httpd (SSL-only mode)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/
.
Nmap done: 1 IP address (1 host up) scanned in 20.32 seconds
```


17.5. Identificación del Sistema Operativo

Por las mismas razones que es interesante averiguar las versiones de los servidores, también lo es determinar el sistema operativo y versión de cada nodo. La técnica más utilizada (y probablemente la más eficaz) es identificar la implementación de la pila TCP/IP del nodo. El comportamiento de la implementación, incluso en detalles no especificados en los estándares puede revelar el SO y la versión. Algunos de estos detalles pueden ser:

- El valor de flags de TCP en determinados momentos de la conexión.
- Valores de Initial Sequence Number (ISN) o patrones en su elección para conexiones sucesivas.
- Tamaños iniciales de la ventana TCP.
- Valor del campo ACK al conectar a puertos cerrados o en desconexiones forzadas.
- Bits de fragmentación de IP, valor del offset cuando no es un datagrama fragmentado, etc.
- Número de secuencia en datagramas IP.
- Identificadores de mensajes ICMP.
- etc.

Un ejemplo sencillo de detección del sistema operativo puede ser:

```
$ sudo nmap -sV -O scanme.nmap.org
Starting Nmap 7.91 ( https://nmap.org ) at 2020-12-15 19:14 CET
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.17s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 996 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
9929/tcp   open  nping-echo
31337/tcp  open  Elite
Device type: general purpose|storage-misc|broadband router|WAP|media device
Running (JUST GUESSING): Linux 5.X|3.X|4.X|2.6.X (95%), HP embedded (93%), Ubiquiti embedded
[...]
OS CPE: cpe:/o:linux:linux_kernel:5.0 cpe:/h:hp:p2000_g3 cpe:/o:linux:linux_kernel:3 cpe:/o:
l[...]
Aggressive OS guesses: Linux 5.0 (95%), Linux 5.0 - 5.4 (95%), Linux 5.4 (94%), HP P2000 G3
N[...]
No exact OS matches for host (test conditions non-ideal).
Network Distance: 12 hops
```


Herramientas de criptografía

David Villa

En este capítulo se introducen algunas herramientas de uso común para cifrado de datos y mensajes. Si bien, no todas están directamente relacionadas con las redes de computadores, los conceptos involucrados son los mismos y, de hecho, pueden estar involucradas en un procesos de comunicación sobre redes.

18.1. Cálculo de resumen

Un «cálculo (o algoritmo) de resumen, reducción o *hash*» es una fórmula matemática que, a partir secuencia de bytes cualquiera (un documento o archivo), permite obtener otra secuencia mucho más pequeña (y normalmente de tamaño fijo). Esta secuencia se llama «resumen», «huella» (o más comúnmente por su término en inglés: *digest*) debería ser diferente y única para cada documento de entrada.

Este tipo de algoritmos tiene muchos usos, pero el más común es la comprobación de **integridad**. Un cambio en un solo byte en el documento de partida devolverá un *digest* diferente.

Uno de los algoritmos de *digest* más usado es Message-Digest Algorithm 5 (MD5). Veamos unos ejemplos con MD5. Produce una secuencia de 128 bits, representada como un número hexadecimal de 32 caracteres.

```
$ echo "this is my important message" | md5sum
28c768bd1d604eafa3e93e0aa56cb302 -
$ echo "thas is my important message" | md5sum
6a3133cadf5cbe1860f56535308a2f24 -
```

Los algoritmos de la familia Secure Hash Algorithm (SHA) son también muy usados. El más común es SHA-1 y produce una secuencia de 160 bits (40 caracteres expresado en hexadecimal):

```
$ echo "this is my important message" | sha1sum
```

```
a511385c478a71e4ac84184b4c059bdf3fc64ae0 -
$ echo "thas is my important message" | sha1sum
0910f46cd3cbdc67e77a9548d2a82583d16ae52a -
```

Estos algoritmos no son (ni deben ser) reversibles, es decir, no es posible obtener el documento de partida a partir del *digest*, incluso aunque fuesen unos pocos bytes. Esta cualidad los hace adecuados para la gestión de claves secretas. En lugar de almacenar las claves de las cuentas de los usuarios en claro, se almacenan los *digest* de éstas. Cuando un usuario introduce una clave, se aplica el mismo algoritmo y se compara con el valor almacenado.

18.2. Cifrado simétrico

El cifrado simétrico permite transformar un documento de partida (texto en claro) en una secuencia ininteligible (texto cifrado) mediante una clave secreta. El texto cifrado puede ser transmitido utilizando un medio no seguro (accesible para el enemigo). Al llegar a su destinatario, puede obtenerse el texto en claro original utilizando la misma clave. Veamos un ejemplo usando GnuPG¹.

```
$ echo "this is my important message" > msg
$ gpg --output cipher-msg.gpg --symmetric msg
Enter passphrase:

$ hexdump -C cipher-msg.gpg
00000000  8c 0d 04 03 03 02 80 44  44 8c 3d 89 62 bb 60 c9  |.....DD.=.b.`.|
00000010  34 69 69 48 47 8f 14 63  c3 d7 66 8d 83 6e 85 35  |4iIHG...c..f..n.5|
00000020  cd 96 d9 fc d6 bd d3 be  e6 70 d7 28 4e 46 82 81  |.....p.(NF..|
00000030  c0 29 0d 90 f0 ae 1f 27  9f 81 ce 8c 37 84 f1 0a  |.).....'....7...|
00000040  64 1b 4d 50 45                |d.MPE|

$ gpg --output rcv-msg --decrypt cipher-msg.gpg
Enter passphrase:
gpg: CAST5 encrypted data
gpg: encrypted with 1 passphrase
gpg: WARNING: message was not integrity protected
```

Por defecto, el fichero cifrado es binario, pero también es posible generar un fichero ASCII, para de ese modo evitar posibles problemas relacionados con la configuración regional (*encoding*) o incluso para poder preservarlo impreso en un papel:

```
$ gpg --output cipher-msg.gpg --armor --symmetric msg
$ $ cat cipher-msg.gpg
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1
```

¹gnupg.org

```
jA0EAwMC3Zphd0yQrJJgyTRfvcXhfUJgZA5C749dB2ccXgPKdiJzbi9US0mgFWYH
dw7YewQ9FAyvZa6liCuMuTiM76Y2
=fTgq
-----END PGP MESSAGE-----
```

18.3. Cifrado asimétrico

En el cifrado asimétrico (o «de clave pública») cada usuario tiene un par de claves privada/pública. Como su nombre indica la clave pública debería estar públicamente disponible mientras que la privada debe ser conocida únicamente por su propietario. En esta sección se utilizará el GnuPG.

18.3.1. Creación del par de claves

Lo primero es por tanto crear el par de claves:

```
$ gpg --gen-key
Please select what kind of key you want:
  (1) RSA and RSA (default)
  (2) DSA and Elgamal
  (3) DSA (sign only)
  (4) RSA (sign only)
Your selection?
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
    0 = key does not expire
<n>  = key expires in n days
<n>w = key expires in n weeks
<n>m = key expires in n months
<n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y

You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: David Villa Alises
Email address: David.Villa@uclm.es
Comment: key example
You selected this USER-ID:
    "David Villa Alises (key example) <David.Villa@uclm.es>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? 0
You need a Passphrase to protect your secret key.
```

Una vez recogidos los datos, GPG necesita generar una clave robusta, de modo que necesita recoger datos aleatorios de calidad.

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy! (Need 288 more bytes)

+++++

+++++

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

....+++++

.....+++++

gpg: /home/david/.gnupg/trustdb.gpg: trustdb created

gpg: key DD379A41 marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb

gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model

gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u

pub 2048R/DD379A41 2015-09-28

Key fingerprint = 3FA0 C819 DE98 CFC9 3957 89FA 9A5B 4248 DD37 9A41

uid David Villa Alises (key example) <David.Villa@uclm.es>

sub 2048R/9AA0A899 2015-09-28

E 18.01 Crea tu par de claves para tu cuenta de correo oficial de la Universidad. Identify la huella de tu clave principal.

18.3.2. Anillo de claves

La creación de la clave ha creado también el anillo de claves públicas, en el que por ahora solamente existen las del propio usuario. Veámoslas:

```
$ gpg --list-keys
/home/vagrant/.gnupg/pubring.gpg
-----
pub 2048R/DD379A41 2015-09-28
uid David Villa Alises (key example) <David.Villa@uclm.es>
sub 2048R/9AA0A899 2015-09-28
```

Veamos cómo importar una clave pública de otro usuario en nuestro anillo. La forma más sencilla es averiguar la huella de la clave pública de la persona que nos interesa (8BC6A1DC en este ejemplo) y asumiendo que está publicada en un servidor conocido, ejecutar:

```
$ gpg --keyserver pgp.rediris.es --recv-keys 8BC6A1DC
gpg: requesting key 8BC6A1DC from hkp server pgp.rediris.es
gpg: key 8BC6A1DC: public key "María José Santofimia Romero <MariaJose.Santofimia@uclm.es>"
imported
gpg: Total number processed: 1
```

```
gpg: imported: 1 (RSA: 1)
```

Listemos de nuevo el contenido del anillo de claves:

```
$ gpg --list-keys
/home/vagrant/.gnupg/pubring.gpg
-----
pub  2048R/DD379A41 2015-09-28
uid          David Villa Alises (key example) <David.Villa@uclm.es>
sub  2048R/9AA0A899 2015-09-28

pub  2048R/8BC6A1DC 2014-10-06
uid          María José Santofimia Romero <MariaJose.Santofimia@uclm.es>
sub  2048R/8888FA80 2014-10-06
```

Obviamente también podemos subir al servidor nuestra clave pública para que pueda estar disponible para el resto de usuarios:

```
$ gpg --keyserver pgp.rediris.es --send-keys DD379A41
```

E 18.02 Sube tu clave pública al servidor pgp.rediris.es. Pide a tus compañeros las huellas de sus claves públicas e impórtalas en tu anillo. Cada uno de tus compañeros debe asegurarte personalmente (no sirve correo electrónico o chat) que efectivamente se trata de su huella.

E 18.03 ¿Por qué crees que el procedimiento de firma de las claves de otras personas tiene que ser tan riguroso?

18.3.3. Cifrado

Para cifrar se debe indicar la clave pública (la huella de la clave) del receptor del mensaje (María José en este ejemplo). Por tanto, solo ella podrá descifrarlo:

```
david@host:~$ gpg --encrypt --armor --recipient 8BC6A1DC msg
gpg: 8888FA80: There is no assurance this key belongs to the named user

pub  2048R/8888FA80 2014-10-06 María José Santofimia Romero <MariaJose.Santofimia@uclm.es>
   Primary key fingerprint: 86C2 D5C2 CC70 FC19 1F43 FE87 970E 5176 8BC6 A1DC
   Subkey fingerprint: 8EE0 4372 F8E3 3C57 6593 7C97 06F3 7B6A 8888 FA80

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
```

Observe la advertencia dado que se trata de una clave sin firmar (no existen garantías de que la clave realmente pertenezca a María José). Esto ha creado el fichero cifrado `msg.asc` con el siguiente contenido:

```

1  -----BEGIN PGP MESSAGE-----
2  Version:  GnuPG v1
3
4  hQEMAwbze2qIiPqAAQgAzZs5qsQAGeI17dTIPq0++fDCSjKH+4cztDidZ5PL6qYy
5  miWXPgOJ1mg7a9fte2Sq+uWIxb94njJPD/1/pogrYiLBnqDmTXLPhX8WstpkpaEW
6  bJLzMRVisx6JU2VEDOUQCqWdNYyScjn0mjVT2v0C1GIqIwu3X13+0RTUmb+CQKrK
7  b/liLS2WLqZ9fX4fDiw3uM1JREvc4iWXF1kdnZu2ePx/XYjBwLKZojunxjF0Ax6
8  Ych6KHP4P0+FEjsLCltFHvHSFHRamUwsI7aXZ/uHsUuhc6wWHxNJ3/buL2xMH8MN
9  na07StT+rgvkGqrGcDSKEhDu0+avXvt8dxq2eX3N0NJZAXmgi/MVrnVzuqxZSY5j
10  09d4cxDZDHX/aLKeQIKQJ8ypbGM5+oXUhnR6ibSGTgqHSffvJdSYvxyCnCKFsADA
11  7TbeAz058TLJ5n2Vn7B/Oj7eC+a47Up0JQ=
12  =PBn6
13  -----END PGP MESSAGE-----

```

E 18.04 Envía un fichero de texto cifrado a un compañero. Descifra un fichero de texto recibido de un compañero.

18.3.4. Descifrado

El receptor simplemente tiene que indicar el fichero, puesto que `gpg` por defecto intentará descifrarlo por defecto con la clave privada del usuario:

```

mariajose@host:~$ gpg --output recv-msg --decrypt msg.asc

You need a passphrase to unlock the secret key for
user: "María José Santofimia Romero <MariaJose.Santofimia@uclm.es>"
2048-bit RSA key, ID 8888FA80, created 2014-10-06 (main key ID DD379A41)

Enter passphrase:

gpg: encrypted with 2048-bit RSA key, ID 8888FA80, created 2015-09-28
      "David Villa Alises (key example) <David.Villa@uclm.es>"

```

E 18.05 Envía un fichero binario (un PDF, por ejemplo) y envíalo a un compañero. Descifra un fichero binario recibido de un compañero.

18.4. Autenticación mediante firma digital

La autenticación de mensajes implica la adición de un bloque de bytes al mensaje original que demuestra que el mensaje efectivamente procede del emisor.

```

david@host:~$ gpg --output msg.sig --sign msg

You need a passphrase to unlock the secret key for

```



```
user: "David Villa Alises <David.Villa@uclm.es>"
4096-bit RSA key, ID 9AA0A899, created 2011-12-24 (main key ID DD379A41)
```

El mensaje se comprime, firma y cifra con la clave privada de David, de modo que CUALQUIER usuario podrá descifrarlo:

```
mariajose@host:~$ gpg --output msg.recv --decrypt msg.sig
gpg: Signature made Mon 28 Sep 2015 12:52:53 PM UTC using RSA key ID DD379A41
gpg: Good signature from "David Villa Alises (key example) <David.Villa@uclm.es>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 3FA0 C819 DE98 CFC9 3957  89FA 9A5B 4248 DD37 9A41
Subkey fingerprint: 2184 29E5 377E AF6C C2AE  EDED F394 5D2D C734 3A71
```

Puede haber situaciones en las que se desea que el mensaje autenticado viaje en claro. Para conseguirlo se puede utilizar la «firma en claro» (*clear-sign*):

```
$ gpg --clearsign msg
```

En este caso el fichero resultante (msg.asc) está formado por el mensaje original en claro y una firma ASCII. Es el siguiente:

```
1  -----BEGIN PGP SIGNED MESSAGE-----
2  Hash: SHA1
3
4  This is my important message
5  -----BEGIN PGP SIGNATURE-----
6  Version: GnuPG v1
7
8  iQEcBAEBAGAGBQJWCTmgAAoJEJpbQkjdn5pBnhIH/RRstB4l6q7D59wT/Z5gc8V5
9  gr90f51bkGbbJ000a7qBPbz1BcxK8sshqDxL5uMb0TMg14wNdI8MgBD+g4P4Mvnc
10 Ca7d0sMOIkC7Tic5rmUf00tzqTfTSviG5wcUctDvXpESYwgcRkxP8oxo910WjQXn
11 pReWRHoAbwFG6Shulb+FFQ8D0iK2+LGMU9oKJMQfw2iBCsk9SPuEk9WVzdS2Y8Bg
12 NKxsTIHP1CqqnhLvIZiHcvB93Dp9EqfXYAmr1ho50xvdadaTIpLOFsoy30h5Ifng
13 HgNGz1ljhHW8qrPbTLZWniDkBF7CQRCEnico9Zw3QVnUclYqa+HVtGBWElokbs=
14 =vAH2
15 -----END PGP SIGNATURE-----
```

La autenticidad de un fichero firmado de este modo puede ser comprobada con:

```
$ gpg --verify msg.asc
gpg: Signature made Mon 28 Sep 2015 01:07:20 PM UTC using RSA key ID DD379A41
gpg: Good signature from "David Villa Alises (key example) <David.Villa@uclm.es>"
```

Si el fichero original es un binario no es posible realizar la firma en claro de ese modo. En ese caso se utiliza una «firma adjunta» (opción `--detach-sig`) que genera la firma en un fichero adicional.

Los mecanismos de cifrado y firma son ortogonales, es decir, es posible utilizar uno sin el otro o ambos a la vez.

E 18.06 Envía un fichero de texto firmado a un compañero. Verifica que el fichero de texto recibido de un compañero fue efectivamente enviado por él.

E 18.07 Envía un fichero binario firmado a un compañero. Verifica que el fichero binario recibido de un compañero fue efectivamente enviado por él.

18.4.1. Relaciones de confianza

En los ejemplos anteriores con claves públicas importadas aparecen a menudo mensajes como el siguiente:

```
gpg: WARNING: This key is not certified with a trusted signature!  
gpg:          There is no indication that the signature belongs to the owner.
```

Esto significa que el propietario del anillo de claves no ha expresado su confianza en dicha clave. Para hacerlo, el usuario debe comprobar que efectivamente la clave corresponde con su supuesto dueño (incluso personalmente²) y entonces firmar su clave pública con su propia clave. Estas firmas a su vez pueden ser conocidas públicamente, de modo que por ejemplo, el usuario Paco puede saber que el usuario Juan ha firmado determinada clave de Pedro.

A partir de esa información, gpg permite definir «relaciones de confianza», es decir, con cuanto rigor crees que un determinado usuario (cuya clave has firmado) ha verificado las claves que ha firmado a su vez. Si confías plenamente en el usuario Paco, entonces todas las claves que él haya firmado merecen para ti de la misma confianza que las que has firmado tú personalmente. Los niveles de confianza son: *unknown*, *none*, *marginal* y *full*.

E 18.08 Añade las claves públicas de tus compañeros a tu anillo de claves (previa adecuada comprobación). Determina el grado de confianza en tus compañeros en base a tu percepción de su proceso de firma de claves.

²https://en.wikipedia.org/wiki/Key_signing_party

Capítulo 19

Autenticación

David Villa
María José Santofimia

La autenticación es una necesidad esencial en múltiples situaciones. Los sistemas de autenticación más comunes para el usuario medio son:

- Inicio de sesión en el sistema operativo, o en una aplicación.
- Autenticación remota, incluyendo web, shell o escritorio remoto.
- Desbloqueo de un smartphone: biométrico, gestual, etc.

Obviando los sistemas que almacenan las claves de usuario en claro (algo que ocurre con más frecuencia de lo imaginable) lo habitual es que el sistema aplique una función de resumen (hash) y almacene el resultado en un fichero o base de datos. Este almacén de hashes es especialmente sensible como vamos a ver, por lo que debe ser accesible únicamente con el nivel más alto de privilegios.

Un ejemplo de este almacén de hashes es el fichero `/etc/shadow` en sistemas POSIX, del que se muestra un fragmento a continuación:

```
root:$6$20y8Tf74019Avrs ... Z83pxswrwyurP4m8q7yux0AvZ41duALvGbx1DvBCLdnP
.:16694:0:99999:7:::
daemon*:16694:0:99999:7:::
bin*:16694:0:99999:7:::
sys*:16694:0:99999:7:::
sync*:16694:0:99999:7:::
man*:16694:0:99999:7:::
lp*:16694:0:99999:7:::
mail*:16694:0:99999:7:::
news*:16694:0:99999:7:::
uucp*:16694:0:99999:7:::
proxy*:16694:0:99999:7:::
www-data*:16694:0:99999:7:::
backup*:16694:0:99999:7:::
sshd*:16694:0:99999:7:::
david:$6$cYj9Tz/y$LLfZ ... DpCcZesW7nWYtMJhRKtpX0CaIBn19ytzElf1XqN0g3UzI/:17830:0:99999:7:::
```

Cuando un usuario introduce su clave, el sistema aplica el mismo algoritmo de resumen. Si el resultado corresponde con el registrado en el almacén se otorga acceso al usuario. Si el algoritmo de resumen es correcto, el resultado debería ser único para cada posible entrada. Es esencialmente el mismo tipo de algoritmos que se utilizan como comprobación de integridad que vimos en la sección 18.1.

La mayoría de los ataques de autenticación no buscan la clave legítima, si no una que produzca el mismo valor de hash almacenado para un usuario dado. Obviamente, si el algoritmo de hash es robusto, la probabilidad de que haya varias claves que producen el mismo hash es extremadamente baja.

Resumiendo, si el atacante tiene acceso al almacén de hashes tiene resuelta una parte importante del problema, y Por este motivo el almacén de hashes es crítico. Si un atacante lo consigue, puede realizar un ataque *offline* (es decir, sin utilizar el sistema objetivo ni estar bajo su control) y empleando una cantidad de tiempo y recursos desconocidos.

19.1. Espacio de claves

Por supuesto, la fortaleza del sistema depende directamente de la «calidad» de la clave. Esa calidad depende de la longitud de la clave y la variedad de caracteres utilizados (conjunto de símbolos). Esto da lugar al llamado «espacio de claves» (*keyspace*). Por simple combinatoria es sencillo determinar el espacio de claves para un tamaño de clave dado en función del grupo de símbolos utilizado.

Para ilustrar la importancia de estos dos factores veamos unos ejemplos de cómo afecta el grupo de símbolos usado para una clave de 4 símbolos:

- letras ASCII minúsculas: $26^4 = 456\,976$
- letras ASCII: $52^4 = 7\,311\,616$
- letras ASCII + números: $62^4 = 14\,776\,336$
- letras ASCII + números + símbolos de puntuación: $94^4 = 78\,074\,896$

Y cómo afecta la longitud de la clave para un conjunto de símbolos fijo (letras mayúsculas y minúsculas):

- Longitud de 4 símbolos: $52^4 = 7\,311\,616$
- Longitud de 8 símbolos: $52^8 = 53\,459\,728\,531\,456$
- Longitud de 10 símbolos: $52^{10} = 390\,877\,006\,486\,250\,192\,896$

19.2. Fuerza bruta

Un ataque por fuerza bruta genera claves utilizando combinaciones de símbolos, aplica el algoritmo de resumen y compara con el valor de hash conocido. El rendimiento de ese programa (llamado comúnmente *cracker*) determina las posibilidades de éxito del atacante, en función del tiempo necesario para probar todas las combinaciones. Los *cracker* más sofisticados nos ofrecen un cálculo de su rendimiento para el computador concreto con el que cuente el auditor.

Como ejemplo, se muestra la salida del test que podemos realizar con *john the ripper*, uno de los *cracker* más famosos y veteranos:

```
~$ /usr/sbin/john --test
Created directory: /home/david/.john
Benchmarking: descrypt, traditional crypt(3) [DES 128/128 SSE2-16]... DONE
Many salts:      5814K c/s real, 5814K c/s virtual
Only one salt:   5574K c/s real, 5585K c/s virtual

Benchmarking: bsdictcrypt, BSDI crypt(3) ("_J9..", 725 iterations) [DES 128/128 SSE2-16]...
DONE
Many salts:      194099 c/s real, 194488 c/s virtual
Only one salt:   189670 c/s real, 189670 c/s virtual

Benchmarking: md5crypt [MD5 32/64 X2]... DONE
Raw:      18559 c/s real, 18559 c/s virtual

Benchmarking: bcrypt ("2a$05", 32 iterations) [Blowfish 32/64 X2]... DONE
Raw:      1086 c/s real, 1086 c/s virtual

Benchmarking: LM [DES 128/128 SSE2-16]... DONE
Raw:      78679K c/s real, 78679K c/s virtual

Benchmarking: AFS, Kerberos AFS [DES 48/64 4K]... DONE
Short:     586444 c/s real, 586444 c/s virtual
Long:      1938K c/s real, 1938K c/s virtual

Benchmarking: tripcode [DES 128/128 SSE2-16]... DONE
Raw:      5029K c/s real, 5039K c/s virtual

Benchmarking: dummy [N/A]... DONE
Raw:      163990K c/s real, 163990K c/s virtual

Benchmarking: crypt, generic crypt(3) [?/64]... DONE
Many salts:   388742 c/s real, 388742 c/s virtual
Only one salt: 385843 c/s real, 385843 c/s virtual
```

Para cada uno de los algoritmos de resumen soportados por la herramienta, calcula la cantidad de claves que puede probar cada segundo para este computador concreto. Por ejemplo, para una clave de 8 letras minúsculas de la que disponemos de su hash DES, requiere:

$$26^8 / 5\,574\,000 = 37\,464s = 10,4 \text{ horas}$$

Pero hoy en día, gracias a las plataformas *cloud*, un atacante puede disponer de una capacidad de cómputo muy importante a un precio muy asequible, de modo que el rendimiento del *cracker* puede aumentar sensiblemente. John soporta OpenMP y MPI, lo cual le permite utilizar un grid, multiplicando su rendimiento por 100 fácilmente.

19.3. Ataques de diccionario

La experiencia demuestra que los usuarios no eligen combinaciones aleatorias dentro del espacio de claves. Lógicamente eligen claves que puedan recordar con facilidad, es decir, palabras que existen en su idioma, o quizá combinaciones de ellas.

Así pues, en lugar de generar todas las combinaciones posibles, un *cracker* puede resultar mucho más eficaz si genera hashes a partir de claves predefinidas y almacenadas en un fichero (denominado «diccionario» o «word-list»).

Existen muchos diccionarios disponibles en Internet¹, pero uno de los más famosos es `rockyou.txt`. Contiene más de 14 millones de claves reales filtradas desde el sitio rockyou.com en 2009.

Veamos un ejemplo de uso de `john` con el diccionario *rockyou*:

```
~$ sudo unshadow /etc/passwd /etc/shadow > /tmp/hashes
~$ /usr/sbin/john --wordlist=/tmp/rockyou.txt /tmp/hashes
Loaded 4 password hashes with 4 different salts (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
```

Los ataques de diccionario son más eficaces contra un conjunto de cuentas que contra una si lo que pretende el atacante es conseguir acceso a la máquina.

19.4. Rainbow tables

Cuando el espacio de claves es limitado por una mala decisión de diseño, una limitación técnica, el uso de un teclado numérico, etc., se pueden generar todas las posibles claves y sus hashes, y almacenarlas en un fichero. De ese modo, para recuperar una clave, basta con buscar la hash para conseguir la clave de forma instantánea sin realizar ningún cálculo específico. En Internet² es sencillo encontrar tablas para distintos algoritmos y tamaños de clave.

¹<https://wiki.skullsecurity.org/Passwords>

²<https://project-rainbowcrack.com/table.htm>

19.5. Caso de estudio: Microsoft LAN Manager

LAN Manager proporcionaba un servicio de autenticación para sistemas Microsoft Windows NT, aunque su soporte continuó en productos posteriores como Windows XP y Windows Vista. El sistema de registro de las claves de usuario (conocido como Microsoft LAN Manager (LM) hash) fue especialmente famoso debido a su debilidad del algoritmo de resumen utilizado.

Esta debilidad se debe principalmente a las limitaciones que el sistema impone a las claves que el usuario puede utilizar:

- La longitud máxima de la clave es de 14 caracteres, pero el algoritmo divide la clave en 2 grupos de 7 caracteres y aplica la función de resumen (DES) a cada parte. Después las concatena.
- Si la clave es menor de 7 caracteres, el hash para la segunda parte es un valor fijo y reconocible.
- La clave es convertida a mayúsculas antes de aplicar el algoritmo de resumen, de modo que el espacio de claves es mucho menor.
- No utiliza salt.

Con lo visto anteriormente, se puede determinar el espacio de claves para una clave de 7 símbolos mayúsculas y números como:

$$36^7 = 78\,364\,164\,096$$

Usando john en el mismo computador en el hicimos el test anterior, implicaría:

$$36^7 / 78679000 = 995,9s = 16,6 \text{ minutos}$$

Es posible extraer las hashes de las claves LM con programas como `pwdump7` desde Windows ³ o `samdump2` desde un GNU/Linux montando la partición Windows, simplemente con:

```
root@live:/mnt/sda4/Windows/System32/config# samdump2 SYTEM SAM > lm-hashes.txt
```

Y después utilizar john para hacer un ataque de fuerza bruta.

```
root@live:~# john lm-hashes.txt
```

Pero dado el pequeño espacio de claves que representa, LM es un claro candidato a la generación/uso de *rainbow tables*. En la figura 19.1 aparece

³http://www.tarasco.org/security/pwdump_7/pwdump7.zip

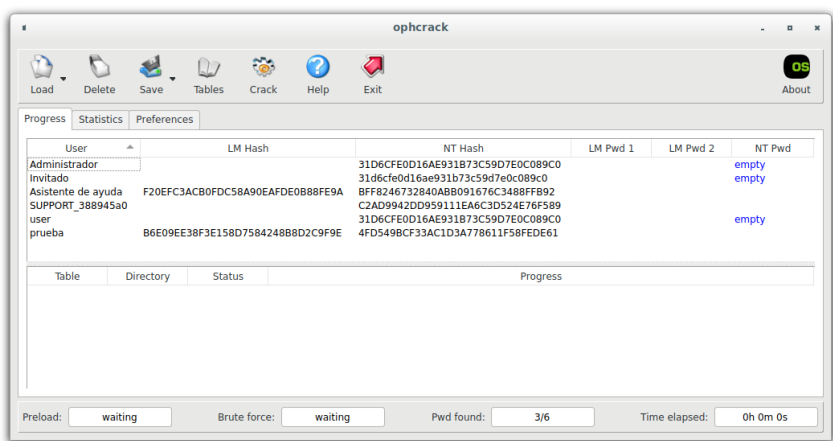


FIGURA 19.1: Recuperación de claves Microsoft LAN Manager con ophcrack

un ejemplo de extracción de claves LM con el programa ophcrack. De este modo, buscar entre todas las hashes posibles requiere menos de 2 minutos en un computador actual.

19.6. Autenticación remota

En el acceso remoto más simple y habitual, el usuario establece una sesión enviando sus credenciales. Por ejemplo, en el uso de una aplicación web, el usuario utiliza un formulario HTML para introducir un usuario y una contraseña. Salvo que el atacante pueda acceder a los datos internos de la aplicación en el servidor mediante otro tipo de ataque, lo único que puede hacer es probar credenciales arbitrarias. Esto significa que es un ataque *online*, es decir, la víctima podría detectar el ataque, y quizá, aplicar contramedidas. Lo mismo es extrapolable a cualquier servicio que utilice autenticación remota basada en credenciales, tales como SSH, Remote Desktop Protocol (RDP), VNC, FTP, SMB/CIFS y muchos otros.

Por el papel esencial que juega la víctima, un ataque de fuerza bruta remoto es muy diferente a un *cracker* de hash. Se basa en generar tráfico sintético emulando al cliente legítimo. Su efectividad es obviamente mucho menor puesto que debe utilizar el mismo mecanismo de petición/respuesta, de modo que puede probar con suerte, unas pocas claves por segundo, en lugar de las decenas de miles que puede probar un ataque contra una hash.

Aún así, no se deben subestimar este tipo de ataques. Como en el *cracking*, también es posible usar múltiples clientes coordinados para conseguir un

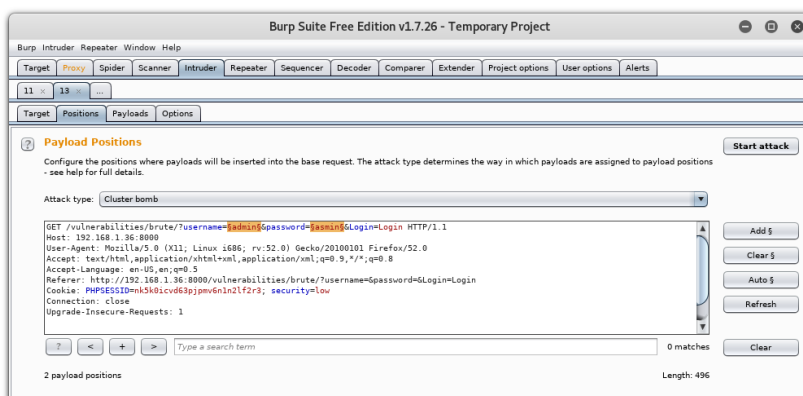


FIGURA 19.2: Edición de la plantilla para una ataque de fuerza bruta con Burp Suite

ratio (intentos por segundo) mucho más alto.

Las herramientas más comunes en este campo son Burp Suite, Ncrack y HTC Hydra. Por ejemplo, Burp Suite permite generar patrones a partir del tráfico entre un navegador web y el servidor. Identificando los valores que corresponden con usuario y contraseña, permite generar peticiones automáticas aplicando valores de un diccionario. En la figura 19.2 se muestra la obtención de dicho patrón para la aplicación DVWA ⁴.

El programa nmap dispone también de muchos *plugins* para realizar ataques remotos de autenticación por fuerza bruta. Como ejemplo, en el siguiente listado se muestra un ataque a un servidor SSH:

```
~$ nmap -p 22 --script ssh-brute --script-args ssh-brute.timeout=4s localhost
Starting Nmap 7.70 ( https://nmap.org ) at 2018-10-30 09:13 CET
NSE: [ssh-brute] Trying username/password pair: root:root
NSE: [ssh-brute] Trying username/password pair: admin:admin
NSE: [ssh-brute] Trying username/password pair: administrator:administrator
NSE: [ssh-brute] Trying username/password pair: webadmin:webadmin
NSE: [ssh-brute] Trying username/password pair: sysadmin:sysadmin
```

⁴<http://www.dvwa.co.uk/>

Parte II

Servicios básicos

Capítulo 20

SSH

David Villa
Ana Rubio

SSH (Secure SHell) es un protocolo de aplicación que permite establecer conexiones con máquinas remotas. Podríamos decir que es una versión mejorada del veterano¹ `telnet`, sin embargo, es mucho más que eso. SSH proporciona un canal seguro (cifrado) para ejecutar comandos, transferir ficheros (con `scp`) o crear túneles que pueden transportar datos de conexiones arbitrarias.

20.1. Shell segura

Como ya indica su nombre, el uso más común de la aplicación SSH es el de «shell remota», es decir, un programa que permite abrir una sesión en un computador remoto y ejecutar programas. En este documento vamos a usar la aplicación `OpenSSH`, que en el caso de sistemas GNU/Linux suele estar disponible como dos paquetes: `openssh-server` y `openssh-client`. Por supuesto, existen otras muchas implementaciones para casi cualquier SO.

Se trata, por supuesto, de una aplicación cliente-servidor: El servidor SSH, que debe estar instalado en cada computador que queramos que sea accesible, siempre está a la espera de conexiones, mientras que el cliente SSH es el que inicia la conexión.

En un sistema GNU/Linux utilizar SSH es muy sencillo. Basta con abrir un terminal e indicar a través del comando `ssh` el nombre de usuario, el computador remoto y el puerto (si es distinto del 22):

```
ana@wozniak:~$ ssh vagrant@localhost -p 2200
vagrant@localhost's password:
[...]
vagrant@sandbox:~$
```

¹...e inseguro

El comando `ssh` trata de establecer una conexión segura con el servidor SSH que hay en `localhost:2200` (en este ejemplo es una máquina virtual). Concretamente queremos conectar con el usuario `vagrant`. Este proceso por defecto pide la contraseña del sistema asociada a ese usuario, que no se muestra al introducirla.

20.2. Configuración

Para simplificar el comando de conexión se puede escribir un fichero de configuración llamado `~/.ssh/config` en el **computador cliente**. Para el caso anterior, el fichero podría ser algo como:

```
1 Host sandbox
2     Hostname localhost
3     Port 2200
4     User vagrant
```

Es importante señalar que este fichero debe tener permisos de lectura/escritura solo para el usuario (`-rw-r--r--`) o será ignorado. Una vez tengas esta configuración puedes conectar simplemente con:

```
ana@wozniak:~$ ssh sandbox
vagrant@localhost's password:
```

Este modo de autenticación con usuario/clave se llama de «clave privada» porque se asume que solamente el usuario debe conocer dicha clave.

20.3. Acceso con clave pública

Para evitar tener que escribir constantemente la clave cada vez que conectas a una misma máquina, SSH ofrece un sistema de autenticación de «clave pública». Este sistema utiliza un par de claves privada/pública. La clave pública se copia en el sistema al que quieres acceder y, de hecho, no hay ningún problema en que cualquiera la pueda leer o copiar. Mientras, la clave privada debe quedar custodiada por el usuario.

Para generar el par de claves ejecuta el siguiente comando aceptando todos los valores por defecto (pulsa ENTER):

```
ana@wozniak:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ana/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ana/.ssh/id_rsa
Your public key has been saved in /home/ana/.ssh/id_rsa.pub
The key fingerprint is:
```

```
SHA256:KXUrUqPxk/EPiXEBfgqK3HlSqvhphSeh60+qm9tz1DM ana@wozniak
```

Esto genera dos ficheros llamados `id_rsa` (clave privada) y `id_rsa.pub` (clave pública) en el directorio `/home/ana/.ssh/`.

Ahora se debe enviar la clave pública al servidor. Esto se puede hacer fácilmente con:

```
ana@wozniak:~$ ssh-copy-id sandbox
vagrant@localhost's password:

Number of key(s) added: 1

Now try logging into the machine, with:  "ssh 'sandbox'"
and check to make sure that only the key(s) you wanted were added.
```

Este programa almacena la clave pública del usuario —por defecto, `~/.ssh/id_rsa.pub`— en el fichero `~/.ssh/authorized_keys` del servidor (`sandbox` en este ejemplo).

Y tal como indica, ahora deberíamos poder entrar en la máquina `sandbox` simplemente con:

```
ana@wozniak:~$ ssh sandbox
Last login: Tue Mar  2 10:30:40 2021 from 10.0.2.2
vagrant@sandbox:~$
```

Esto no solo es interesante porque ahorra al usuario tener que escribir su contraseña constantemente. Además permite que las aplicaciones puedan automatizar tareas sin la intervención de un usuario. Los sistemas de control de versiones (como `git`) o sistemas de Integración o Despliegue Continuo (CI/CD) utilizan habitualmente autenticación SSH con clave pública.

También es posible ejecutar un único comando sin tener que conectar y ver la salida en el computador cliente. Simplemente hay que escribir el comando a continuación:

```
ana@wozniak:~$ ssh sandbox ls /home/
vagrant
```

20.4. Autenticación mediante certificado

Otra forma de autenticar un usuario es por medio de certificados, una opción algo más avanzada que la autenticación con clave pública que hemos visto en la sección anterior. Estos certificados son emitidos por una autoridad de certificación o CA (Certificate Authority), que firma (con su clave

privada) el par de claves (privada/pública) de los usuarios. El resultado son certificados que permiten autenticar usuarios en hosts y viceversa.

La ventaja de este sistema es que el administrador de un servidor puede expedir certificados para un número arbitrario de usuarios. Un mismo par de claves firmado por un administrador puede otorgar derecho de acceso a múltiples servidores sin tener que modificar su configuración.

Las claves de la CA se crean también con el comando `ssh-keygen`:

```
$ ssh-keygen -t rsa -f clave_ca -P "" -C clave_ca
Generating public/private rsa key pair.
Your identification has been saved in clave_ca
Your public key has been saved in clave_ca.pub
The key fingerprint is:
SHA256:nbN+pdYB3paUUnoPe60KJbkC3f1KiXCVbSIH51wQXbo clave_ca
```

Aunque `ssh-keygen` proporciona multitud de opciones, en este caso solamente utilizamos algunas de ellas:

- t permite indicar el algoritmo (en este caso cifrado Rivest, Shamir y Adleman (RSA)).
- f especifica el nombre del fichero destino.
- P define la contraseña.
- C añade un comentario que aparece al final de la clave pública.

La clave privada (`clave_ca`), por supuesto, debe almacenarse de forma segura, ya que es con la que se firma y emite los certificados. La clave pública (`clave_ca.pub`) es la que se utiliza el servidor SSH en el proceso de autenticación de los clientes.

20.4.1. Un certificado para el usuario

Los certificados SSH de usuario hacen posible autenticar usuarios en un servidor determinado. Para ilustrar cómo funcionan vamos a generar un certificado para el usuario `vagrant` de la máquina `sandbox` (que ejecuta el servidor SSH). Primero creamos un par de claves pública/privada para el usuario, con el nombre `clave_usuario_vagrant`.

```
$ ssh-keygen -f clave_usuario_vagrant
```

Después, se firma la clave pública del usuario (`clave_usuario_vagrant.pub`) con la clave privada de la CA (`clave_ca`):

```
$ ssh-keygen -s clave_ca -I sandbox -n vagrant clave_usuario_vagrant.pub
```


Con este comando se ha generado también el certificado `clave_usuario_vagrant-cert.pub`.

Las opciones que se han usado en la firma son:

- s para indicar la clave con la que se quiere firmar (*sign*).
- I para determinar la identidad del certificado, normalmente el nombre del *host* (puede utilizarse en el futuro para revocar un certificado, por ejemplo).
- n que permite definir una lista de usuarios y/o *hosts* donde el certificado es válido (en este caso únicamente el usuario *vagrant*).

En el servidor, es necesario almacenar la clave pública de la CA con los permisos adecuados (644). En la máquina *sandbox*:

```
root@sandbox:/etc/ssh# chmod 644 clave_ca.pub
root@sandbox:/etc/ssh# ls -la clave_ca.pub
-rw-r--r-- 1 vagrant vagrant 562 Mar 3 09:13 clave_ca.pub
```

Para indicar que todas las claves de usuario firmadas por la CA se pueden autenticar en el servidor, se debe añadir lo siguiente en el archivo de configuración `/etc/ssh/sshd_config`:

```
TrustedUserCAKeys /etc/ssh/clave_ca.pub
```

Para que los cambios tengan efecto reiniciamos el servidor SSH (*sshd*) con:

```
root@sandbox:/home/vagrant# systemctl restart sshd
```

Con esto termina la configuración el servidor (*sandbox* en este ejemplo).

Por otro lado, el cliente tiene que copiar su clave privada (*usuario_vagrant_clave*) y su certificado (*usuario_vagrant_clave-cert.pub*) en su directorio `~/.ssh/`:

```
ana@wozniak:~$ ls -l ~/.ssh
-rw----- 1 ana ana 2590 mar 3 09:35 clave_usuario_vagrant
-rw-rw-r-- 1 ana ana 2020 mar 3 09:40 clave_usuario_vagrant-cert.pub
-rw-rw-r-- 1 ana ana 137 mar 3 10:04 config
```

Además, habrá de indicar que quiere usar esa clave a la hora de acceder a la máquina *sandbox*:

```
ana@wozniak:~$ ssh -i ~/.ssh/clave_usuario_vagrant sandbox
```

Para mayor comodidad, puede añadirse la clave a la configuración de *sandbox* en `~/.ssh/config`:

```
1 Host sandbox
2     Hostname localhost
3     Port 2200
4     User vagrant
5     IdentityFile ~/.ssh/clave_usuario_vagrant
```

Y con esto se podrá acceder simplemente con:

```
ana@wozniak:~$ ssh sandbox
```

La diferencia respecto a lo explicado en la sección 20.3 es que el usuario nunca necesitó una clave textual para acceder al servidor. De hecho, el servidor se puede configurar para que la única forma de autenticación permitida sea mediante clave pública ², de modo que no hay posibilidad de usar `ssh-copy-id`. Esto se considera más seguro.

20.5. Copia de ficheros con SCP

SCP (Secure Copy Protocol) es un protocolo de copia segura de ficheros entre computadores cualesquiera conectados a Internet. Resulta especialmente cómodo ya que el propio servidor de `OpenSSH` lo incorpora y está activado por defecto.

Con la configuración previa es tan sencillo como:

```
$ scp un-fichero sandbox:
un-fichero                               100%   5    16.6KB/s   00:00
```

También es posible copiar directorios si se utiliza la opción `-r`.

² `PasswordAuthentication no`

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:guillermito@microchof.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 tráfico 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

- [AAMS01] Z. Albanna, K. Almeroth, D. Meyer, y M. Schipper. IANA Guidelines for IPv4 Multicast Address Assignments. RFC 3171 (Best Current Practice), Agosto 2001. url: <http://www.ietf.org/rfc/rfc3171.txt>.
- [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>.
- [Eby] P. J. Eby. Python Is Not Java. url: <https://dirtsimple.org/2004/12/python-is-not-java.html>.
- [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>.
- [FLH⁺00] D. Farinacci, T. Li, S. Hanks, D. Meyer, y P. Traina. Generic Routing Encapsulation (GRE). RFC 2784 (Proposed Standard), Marzo 2000. Updated by RFC 2890. url: <http://www.ietf.org/rfc/rfc2784.txt>.
- [HPV⁺99] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, y G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637 (Informational), Julio 1999. url: <http://www.ietf.org/rfc/rfc2637.txt>.
- [IAN02] IANA. Special-Use IPv4 Addresses. RFC 3330 (Informational), Septiembre 2002. url: <http://www.ietf.org/rfc/rfc3330.txt>.
- [Lib90] D. Libes. Choosing a name for your computer. RFC 1178 (Informational), Agosto 1990. url: <http://www.ietf.org/rfc/rfc1178.txt>.

- [Per96] C. Perkins. IP Encapsulation within IP. RFC 2003 (Proposed Standard), Octubre 1996. url: <http://www.ietf.org/rfc/rfc2003.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>.
- [Pos81c] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Septiembre 1981. Updated by RFCs 1122, 3168. url: <http://www.ietf.org/rfc/rfc793.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>.
- [RMK⁺96] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, y E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), Febrero 1996. url: <http://www.ietf.org/rfc/rfc1918.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>.
- [RRSW97] C. Rigney, A. Rubens, W. Simpson, y S. Willens. Remote Authentication Dial In User Service (RADIUS). RFC 2138 (Proposed Standard), Abril 1997. Obsoleted by RFC 2865. url: <http://www.ietf.org/rfc/rfc2138.txt>.
- [SE01] P. Srisuresh y K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), Enero 2001. url: <http://www.ietf.org/rfc/rfc3022.txt>.
- [SH99] P. Srisuresh y M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), Agosto 1999. url: <http://www.ietf.org/rfc/rfc2663.txt>.
- [Sim94] W. Simpson. The Point-to-Point Protocol (PPP). RFC 1661 (Standard), Julio 1994. Updated by RFC 2153. url: <http://www.ietf.org/rfc/rfc1661.txt>.

-
- [TVR⁺99] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, y B. Palter. Layer Two Tunneling Protocol “L2TP”. RFC 2661 (Proposed Standard), Agosto 1999. url: <http://www.ietf.org/rfc/rfc2661.txt>.